

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

LOCALIZER A Modeling Language for Local Search

by

Laurent Michel

Candidatures en Sciences Économiques et de Gestion,
Facultés Universitaires Notre-Dame de la Paix, 1990

Licence et Maitrise en Informatique,
Facultés Universitaires Notre-Dame de la Paix, 1993
Sc. M., Brown University, 1996

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island

May 1999

UMI Number: 9932449

UMI Microform 9932449
Copyright 1999, by UMI Company. All rights reserved.

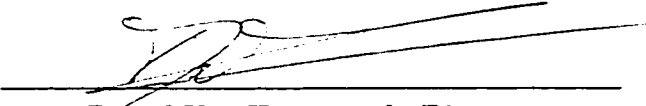
This microform edition is protected against unauthorized
copying under Title 17, United States Code.

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

© Copyright 1997,1998,1999 by Laurent Michel

This dissertation by Laurent Michel is accepted in its present form by
the Department of Computer Science as satisfying the dissertation requirement
for the degree of Doctor of Philosophy.

Date 10/27/98

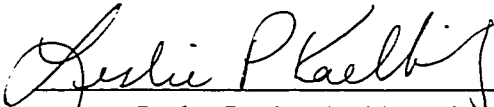

Pascal Van Hentenryck, Director

Recommended to the Graduate Council


Date 10/27/98


Steven P. Reiss, Reader

Date 10/27/98

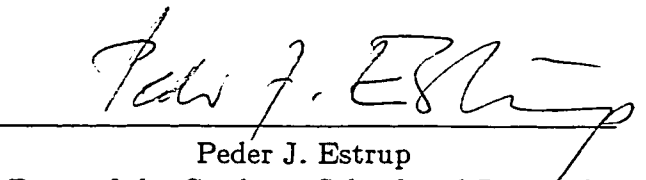

Leslie Pack. Kaelbling, Reader

Date 10/29/98


John Hooker, Reader
Carnegie Mellon University

Approved by the Graduate Council

Date 11/19/98


Peder J. Estrup
Dean of the Graduate School and Research

Vita

Name	Laurent D. Michel
Born	July 17, 1970 in Dinant, Belgium
Education	<i>Brown University</i> , Providence, RI Ph.D. in Computer Science, May 1999. <i>Brown University</i> , Providence, RI M.Sc. in Computer Science, May 1996. <i>Facultés Universitaires Notre-Dame de la Paix</i> , Namur, Belgium Licence et Maitrise en Informatique, 1994. <i>Facultés Universitaires Notre-Dame de la Paix</i> , Namur, Belgium Candidature en sciences Économiques et de Gestion, 1991.
Book	P. Van Hentenryck, L. Michel, Y. Deville <i>Numerica: A Modeling Language for Global Optimization</i> MIT Press, Cambridge, London

Acknowledgements

This thesis is the result of four years of effort, learning and the collaboration with my advisor, Pascal Van Hentenryck. I am extremely grateful for the opportunity Pascal gave me to come to Brown and work with him. Pascal made me discover and appreciate many aspects of computer science and research. His sense of humor, cheerful disposition, guidance, availability, and encouragement are invaluable to me. Our long conversations in the fifth floor kitchen are famed and will not be forgotten. I just wish all graduate students share similar experiences with their advisor.

I would like to specially thank Steve Reiss, Leslie Kaebbling and John Hooker for being on my thesis committee and reviewing this work. I was particularly lucky to get so many complementary comments and suggestions that helped to improve the thesis. Steve and Leslie made sure the thesis would reflect all aspects of the work and include chapters addressing semantic and modeling issues. John's early involvement, enthusiasm and comments on this project contributed to many improvements throughout all chapters. I also want to thank John for coming to Brown to attend the defense.

Two early influences on my grad school career were Paris Kanellakis and Baudouin Le Charlier. The atmosphere of the constraint lunches and Paris's teaching in general are vivid memories. I appreciate his influence since the thesis originates from comments he made on the usefulness of studying local search in the context of constraint programming. Baudouin is responsible for important parts of my college education, for triggering my interest in research, and introducing me to Pascal.

My stay at Brown would not have been so enjoyable without the friendship of many fellow graduate students. My officemates Sharon Carabello, Costas Busch and Gopal Pandurangan together with Vasiliki Chatzi, Jose Castaños, Manos Renieris, Luis Ortiz, Hagit Shatkay, Dimitrios Michailidis, Galina Shubina and Kostadis Roussos, to name a few, were quite often involved in work related or social activities that made the department such a lively, productive environment.

I am also thankful for the support of my family. I thank my parents for giving me the opportunity to go to college and to our relatives for their support and vote of confidence. Finally, many thanks to my wife Valérie for her endless patience, support, understanding and love throughout these years.

Contents

List of Tables	xi
List of Figures	xii
1 Introduction	1
1.1 Combinatorial Optimization	1
1.2 Languages for Combinatorial Optimization	2
1.3 LOCALIZER	3
1.4 Organization	4
2 A Brief Overview of Local Search	6
2.1 Informal Presentation	6
2.2 Formalization	8
2.3 Design Decisions	9
2.3.1 Design Choices for the Neighborhood	9
2.3.2 Design Choices for the Objective Function	10
2.3.3 Design Choices for the Acceptance Criterion	11
2.3.4 Design Choices for the Search Procedure	12
2.4 Overview of some Traditional Approaches.	12
2.4.1 Local Improvement	12
2.4.2 Threshold Algorithms	13
2.4.3 Tabu Search	14
2.5 Beyond Classic Techniques	16
3 A Tour of LOCALIZER	17
3.1 The Computation Model	17
3.2 The Structure of LOCALIZER Statements	17

3.3	The Running Example	19
3.4	Invariants	20
3.5	The Neighborhood	23
3.5.1	Neighborhood Specifications	23
3.5.2	The Acceptance Criteria	25
3.6	Incrementality Issues	28
4	The Language	34
4.1	Data Types	34
4.1.1	Primitive Data Types	34
4.1.2	Primitive Type Constructors	35
4.2	The Type Section	35
4.3	The Constant Section	36
4.3.1	Inline Initializations	37
4.3.2	Offline Initialization	37
4.3.3	Generic Data	38
4.3.4	Computed Data	38
4.4	The Variable Section	39
4.5	The Invariant Section	40
4.5.1	Arithmetic Invariants	40
4.5.2	Set Invariants	43
4.5.3	Set Dependent Invariants	44
4.5.4	Builtin Invariants	44
4.5.5	Invariant Declaration Syntax	45
4.5.6	Current Limitation	45
4.6	The Operator Section	45
4.6.1	Functions	46
4.6.2	Control Structures	46
4.7	Neighborhood	47
4.7.1	The Transformation Component	48
4.7.2	The Neighborhood Component	48
4.7.3	The Acceptance Criterion	51
4.7.4	The try Composition	52
4.8	The Objective Function Section	54
4.9	Termination Criteria	54

4.10	The Parameter Section	55
4.11	Advanced Support	56
4.11.1	Abstract Builtin Data Types	56
4.11.2	Graph Invariants	59
5	A Denotational Semantics of LOCALIZER	63
5.1	LITTLE LOCALIZER	63
5.2	Notations and Conventions	65
5.3	Semantic Algebras	66
5.4	The Semantics	71
5.4.1	Expressions	71
5.4.2	Statements	72
5.4.3	Declarations	73
5.4.4	Invariants maintenance	74
5.4.5	Neighborhood	75
5.4.6	Program	78
6	Implementation	80
6.1	Normalization	81
6.2	Static Invariants	81
6.2.1	The Planning Phase	82
6.2.2	The Execution Phase	83
6.2.3	Propagating the Invariants	83
6.2.4	Correctness	85
6.3	Dynamic Invariants	90
6.3.1	Motivation	91
6.3.2	Overview of the Approach	92
6.3.3	Formalization	92
6.3.4	The Execution Algorithm	94
6.4	Set Invariants	94
6.4.1	Extensional and Intentional Set Invariants	94
6.4.2	Sets as Abstract Data Types	95
6.4.3	Extensional Sets	96
6.4.4	Semi-Intentional Sets	99
6.4.5	Fully Intentional Sets	103

6.4.6	Open structures	105
6.5	Summary	106
7	Applications	107
7.1	Boolean Satisfiability	107
7.1.1	The Problem	107
7.1.2	The Local Search Algorithm	108
7.1.3	A Simple LOCALIZER Statement	108
7.1.4	Extensions	111
7.1.5	Experimental Results	112
7.2	Graph Coloring	113
7.2.1	The Problem	114
7.2.2	The Local Search Algorithm	114
7.2.3	A Simple Model for Graph Coloring	114
7.2.4	A More Incremental Statement	116
7.2.5	Experimental Results	117
7.3	Graph Partitioning	120
7.3.1	The Problem	120
7.3.2	The Local Search Algorithm	120
7.3.3	The LOCALIZER Statement	121
7.3.4	Experimental Results	121
7.4	Job-Shop Scheduling	125
7.4.1	The Problem	126
7.4.2	The Local Search Algorithm	127
7.4.3	A Simulation Statement	128
7.4.4	The LOCALIZER Statement Based on a Makespan Approximation . .	131
7.4.5	A Higher-Level Localizer Model	131
7.4.6	Experimental Results	135
7.5	The Vehicle Routing Problem	136
7.5.1	The Problem	137
7.5.2	The Local Search Algorithm	138
7.5.3	The LOCALIZER Statement	141
7.5.4	A More Advanced Algorithm: λ -interchange	146
7.5.5	The LOCALIZER Statement	146
7.5.6	Experimental Results	150

8	Modeling in LOCALIZER	155
8.1	Incrementality	155
8.1.1	Simulation Versus Differentiation	155
8.1.2	Exploiting Invariants for More Incrementality	158
8.1.3	Summary	160
8.2	Complexity Analysis of Invariants	162
8.2.1	Propagation Cost	162
8.2.2	Global Cost	163
8.2.3	Global Invariants	165
8.3	Constant Factors	166
8.4	Database Techniques	167
9	Related Work	168
9.1	Constraint Programming Languages	168
9.2	Modeling Languages	169
9.3	Graphical Constraint Systems	170
9.4	Finite Differencing	170
9.5	Programming with Invariants	171
9.6	INC: An Incremental Programming Language	172
9.7	Incremental Algorithms	173
10	Conclusion	175
A	LOCALIZER Syntax	178
	Bibliography	184

★ Parts of Chapter 3,5,6 and 7 were published in *Principles and Practice of Constraint Programming (CP'97)*, *INFORMS Journal on Computing* and *CONSTRAINTS An International Journal* in 1997, 1998 and 1999 by Laurent Michel and Professor Pascal Van Hentenryck.

List of Tables

4.1	Precedence of operators.	41
4.2	Programming Interface for Path	58
4.3	Programming Interface for Circuit	59
4.4	Graph	60
6.1	Space and time Complexity Bounds for Static Invariants	85
7.1	GSAT: Experimental Results.	113
7.2	Graph Coloring: Quality of the Solutions.	119
7.3	Graph Coloring: Efficiency of LOCALIZER.	120
7.4	Graph Partitioning: Experimental Results.	124
7.5	Graph Partitioning: Comparison Results.	125
7.6	Job-Shop Scheduling: Experimental Results.	136
7.7	Description of 2-interchange moves.	148
7.8	Insertion neighborhood for VRP.	152
7.9	Modern VRP Implementations	152
7.10	2-Interchange Neighborhood	152

List of Figures

2.1	The Algorithmic Template for Local Improvement.	7
2.2	The Impact of Limiting Tabu Status.	15
3.1	The Computation Model of Localizer for Decision Problems	18
3.2	The Computation Model of Localizer for Maximization Problems	18
3.3	The Structure of LOCALIZER Statements	19
3.4	A Local Improvement statement for Boolean Satisfiability	21
3.5	A GSAT-based Statement for Boolean Satisfiability	27
3.6	A Simulated Annealing Statement for SAT	29
3.7	A Tabu Search Statement for Boolean Satisfiability	30
3.8	An Incremental Statement for GSAT	33
4.1	Primitive Functions.	38
4.2	A try Combination for the Traveling Salesman Problem.	54
5.1	Syntactic categories and Syntax for LITTLE LOCALIZER.	64
5.2	Encoding for Various Notations	67
6.1	The Execution Phase for Static Invariants.	84
6.2	Propagation routine for the summation elementary invariant.	84
6.3	The Execution Algorithm for Dynamic Invariants	94
6.4	Propagation Routines for Extensional Sets.	98
6.5	Propagation Procedures for Semi-intentional Sets.	101
6.6	Propagation Procedure for Fully-intentional Sets	105
7.1	The GSAT Algorithm of Selman et al.	108
7.2	A Local Improvement statement for Boolean Satisfiability.	109
7.3	A More Incremental Statement of GSAT.	110

7.4	A LOCALIZER Statement for Graph Coloring.	115
7.5	A More Incremental Neighborhood for Graph Coloring.	118
7.6	A Graph Partitioning Statement.	122
7.7	A More Incremental Graph Partitioning Statement.	123
7.8	Swapping vertices v and w in a Job-shop schedule.	128
7.9	A First Job-shop Scheduling Statement.	129
7.10	An Approximation Based Job-shop Scheduling Statement.	132
7.11	A Higher-Level Job-shop Scheduling Model.	133
7.12	A Higher-Level Job-shop Scheduling Model (continued).	134
7.13	Detour to visit client i	139
7.14	Vehicle routing model. Insertion neighborhood, Part One.	144
7.15	Vehicle routing model. Insertion neighborhood, Part Two.	145
7.16	Moves allowed by a 2-interchange.	147
7.17	Arc-arc exchange validity.	149
7.18	Invariants for the 2-interchange VRP model.	153
7.19	Neighborhood for the 2-interchange VRP model.	154
8.1	Moving vertex i from class j to class k	157
8.2	A Statement for Graph Coloring Based on Differentiation.	159
8.3	Maintaining Connectivity Information Incrementally.	161
A.1	Grammar Fragment for Typed Declarations.	178
A.2	Grammar Fragment for Record Type Definition.	179
A.3	Grammar Fragment for the Syntax of Expressions.	180
A.4	Grammar Fragment for the Constant Section.	181
A.5	Grammar Fragment for the Variable Section.	181
A.6	Grammar Fragment for the Invariant Section.	181
A.7	Grammar Fragment for the Syntax of Statements.	182
A.8	Grammar Fragment for the Operator Section.	182
A.9	The Syntax of move Instructions.	183
A.10	Grammar Fragment for the Acceptance Criterion.	183

Chapter 1

Introduction

1.1 Combinatorial Optimization

Combinatorial optimization problems are ubiquitous in real life applications. Typical examples include scheduling and resource allocation problems, vehicle routing, network design and various type of planning applications. These problems often consist of finding a solution that optimizes an objective function in a finite, but extremely large, search space. Many of these problems are computationally challenging: They are NP-hard or worse and it is generally believed that there exists no polynomial algorithm to solve them. Traditional approaches to tackle these problems can be divided into three categories: global search, local search, and approximation algorithms¹.

These approaches implement different trade-offs between computation time and quality of the solutions.

Global search includes techniques such as branch and bound, constraint satisfaction, and dynamic programming. The basic idea in global search is to decompose a problem into sub-problems until the sub-problems are easy to solve. Global search algorithms are guaranteed to find optimal solutions and their main design issue is to find appropriate pruning techniques to explore only a fraction of the search space. Local search approaches often sacrifice optimality for computation speed. The basic idea in a local search is to move from configurations (e.g., a candidate solution) to configurations in the hope of improving the quality of the solutions. These approaches are not, in general, guaranteed to find the optimal solution but can often produce “good” solutions quickly. Approximation algorithms

¹Some approaches are in fact a combination of these.

also sacrifice optimality for computation speed but they do so in a way that provides guarantees on the quality of the solution. For instance, there exists an approximation algorithm for the symmetric traveling salesman problem that is guaranteed to be no worse than twice the optimal value.

Solving a combinatorial optimization problem is often a difficult task requiring creativity and, possibly, considerable development time and experimentation. After decades of research, this process remains an art, although significant insights have been obtained.

1.2 Languages for Combinatorial Optimization

This inherent difficulty was a primary motivation to develop better tools to solve combinatorial optimization problems. Almost all of the research in this area has focused on global search and much effort has been spent on providing tools that can simplify the design and implementation of global search algorithms. Global search is now supported by a variety of tools including constraint programming languages such as CHIP [15], Ilog Solver [49], CLP(\mathcal{R}) [27], PROLOG IV and Oz [26] and modeling languages such as AMPL [19], GAMS [5], LINDO [65] and NUMERICA [80].

Constraint languages support the global search paradigm in several ways. First, they provide a declarative component to describe the search space: constraints. These constraints specify what properties must be satisfied by the solutions and the underlying constraint solver uses them to prune the search space. Second, these languages provide suitable abstractions (e.g., non-determinism) to implement search procedures. As a consequence, solving a combinatorial optimization problem with these languages consists of

1. finding a set of constraints that define the solutions while pruning the search space effectively,
2. defining a search procedure to explore the search space in an “informed” way.

Constraint programming languages supporting the global search paradigm are now widely used in industry to solve combinatorial optimization problems. They are generally competitive with specialized implementations of similar algorithms, but they provide substantial reduction in development and maintenance effort.

Modeling languages generally focus on specifying the constraints in a high level algebraic notation. These languages are generally tailored for mathematical programming (e.g., linear and integer programming) and support very high-level data modeling constructs and

aggregate operators that considerably simplify the statements of these problems. Modeling languages generally do not support the specification of search procedures. A notable exception is the modeling language OPL [79].

In contrast, little attention has been devoted to the support of local search algorithms although practitioners frequently use these algorithms and they are more effective than global search approaches for some classes of problems. It should not be concluded however that local search algorithms are easy to design and implement. In fact, these algorithms often require much experimental work to determine a good trade-off between the quality of the solution and the running time. In addition, a careful examination of local search algorithms indicates that good implementations often require maintaining complex data structures incrementally to minimize the cost of moving from configuration to configuration. Designing and implementing these incremental algorithms is often tedious and error-prone.

1.3 LOCALIZER

This thesis originated as an attempt to determine whether a high-level constraint/modeling language could support local search. Its main contribution is to show that this is indeed possible: the thesis presents, a proof of concept, LOCALIZER [36],[35], [37] a modeling language for local search.

LOCALIZER supports the design and implementation of local search algorithms by providing abstractions of the most tedious and error-prone aspects of local search algorithms. Probably, the main abstraction is the concept of *invariant* that relieves users from the need of maintaining complex data structures incrementally. Invariants can be used to specify, in a declarative and high-level way, data structures that are maintained incrementally. These data structures can then be used to define neighborhood concisely and effectively. As a consequence, invariants, by focusing on what has to be maintained and not how to maintain it, relieves the programmer from the most tedious and error-prone aspects of local search algorithms. Invariants, in conjunction with other abstractions to support exploration strategies, may significantly reduce the development time of local search algorithms.

The second contribution of the thesis is to show that LOCALIZER can be implemented to compare well with special purpose implementations of local search algorithms. The implementation of LOCALIZER generalizes the planning/execution model found in constraint-based graphics systems and the finite differencing techniques used in the programming language community. The main novelty here is the ability to deal with dynamic invariants, whose inter-relationships cannot be determined statically. These dynamic invariants

are critical to solve many combinatorial optimization problems efficiently (e.g., job-shop scheduling and vehicle routing).

The last contribution of this thesis is to demonstrate LOCALIZER on a variety of combinatorial optimization problems. These problems include Boolean satisfiability, graph coloring, graph partitioning, scheduling and routing. LOCALIZER is shown to compare well with specific implementations.

It is important to stress at this point that LOCALIZER is a proof of concept. It shows that local search algorithms can be supported by a high-level modeling language, reducing the development time while exhibiting good performance. This thesis should not be viewed as the final specification of LOCALIZER. There are many extensions of the language that are being contemplated and many implementation techniques to be investigated. Nevertheless, this thesis opens, what we believe is, an interesting avenue for simplifying local search algorithms. It is also useful to mention that LOCALIZER differs from traditional modeling languages in a fundamental aspect. Traditional modeling languages (e.g., AMPL) focus on specifying the constraints of the problem. They are mostly independent from the underlying algorithms used to solve these constraints. In contrast, a LOCALIZER statement specifies a local search algorithm. It is thus not surprising that LOCALIZER contains both declarative and procedural components.

1.4 Organization

The rest of the thesis presents LOCALIZER in full detail after a brief overview of local search (Chapter 2) to fix the concepts and the notation. Chapter 3 is a gentle introduction to LOCALIZER and its computation model. It presents the various constructs of the language using Boolean satisfiability as a running example. Chapter 4 presents the language in detail and covers all the constructs (and their syntax) available in LOCALIZER at this point. The presentation is informal to ease readability. Chapter 5 presents a denotational semantics of a subset of LOCALIZER, called LITTLE LOCALIZER. The goal of the chapter is to complement the informal description of the previous chapters and to remove possible ambiguities. Only a subset of LOCALIZER is considered since it simplifies the presentation while preserving the most important concepts. Chapter 6 describes the implementation of the invariants that are the core of LOCALIZER. Chapter 7 applies LOCALIZER to a variety of combinatorial optimization problems to demonstrate that it reduces the development time significantly while preserving most of the efficiency of specific implementations. The problems include Boolean satisfiability, graph coloring, graph partitioning, job-shop scheduling and, finally,

vehicle routing. Chapter 8 discusses modeling issues in LOCALIZER. It addresses mostly the efficiency of LOCALIZER by discussing the complexity of invariants in space and time. Chapter 9 discusses related work and future directions. It covers constraint and modeling languages, finite differencing, invariants, and incremental algorithms. Chapter 10 concludes the thesis.

Chapter 2

A Brief Overview of Local Search

This chapter is a brief overview of local search. Its aim is mainly to review the major local search techniques supported in LOCALIZER. The chapter does not attempt to be comprehensive. Readers interested in a detailed presentation of local search should consult for instance [47], [30] and [1].

The chapter is organized as follows: Section 2.1 informally introduces local search and illustrates its use with a simple example. Section 2.2 formally defines local search and Section 2.3 discusses several design issues. Finally, Section 2.4 describes the instantiation of the framework to specific strategies.

2.1 Informal Presentation

Combinatorial optimization problems generally aim at finding a computation state (e.g., an assignment of values to variables) that satisfies a set of constraints and optimizes an objective function. Many combinatorial optimization problems are NP-complete, which makes them unlikely to be solvable in polynomial time.

Traditional approaches sacrifice either time (e.g., branch and bound algorithms) or the quality of the solution (e.g., local search). The main idea behind local search is to start from a state and to move to an adjacent state that slightly differs from the current one. Local search can thus be viewed as the exploration of the space of states through these neighborhood structures. Many issues arise in designing local search such as the choice of neighborhood structure and the choice of neighbor to move to.

Local improvement is probably the simplest local search technique. Its basic idea is to move to neighbors that improve the value of the objective function. A simple schema for local improvement is described in figure 2.1. In the schema, f is the objective function, and

$neighborhood(s)$ defines states adjacent to s .

```

procedure LOCAL IMPROVEMENT
begin
  1  $s := \text{startState}();$ 
  2 for  $\text{search} := 1$  to  $\text{MaxSearches}$  do
  3   for  $\text{trial} := 1$  to  $\text{MaxTrials}$  do
  4     if  $\text{satisfiable}(s)$  then
  5       if  $f(s) > \text{bestBound}$  then
  6          $\text{bestBound} := f(s);$ 
  7          $\text{best} := s;$ 
  8       select  $n$  in  $neighborhood(s);$ 
  9       if  $f(n) > f(s)$  then
 10          $s := n;$ 
 11    $s := \text{restartState}(s);$ 
 12 return  $\text{best};$ 
end

```

Figure 2.1: The Algorithmic Template for Local Improvement.

Most local search algorithms, including local improvement, enhance the basic skeleton by wrapping it in a multiple starting point strategy implemented with the outer loop of the model and the restarting sub routine.

Graph Partitioning To illustrate these ideas, consider the graph partitioning problem. The problem consists of finding a balanced partition of the node set of a graph that minimizes the number of edges with one endpoint in each partition. More formally, a balanced partition of a graph $G(V, E)$ is a pair $\langle V_1, V_2 \rangle$ such that $V_i \subseteq V$, $V_1 \cap V_2 = \emptyset$, $V_1 \cup V_2 = V$ and $|V_1| = |V_2|$. The cost of a partition is $|C_p|$ where $C_p = \{\langle a, b \rangle \in E \mid a \in V_1 \wedge b \in V_2\}$. The set S of solutions to the graph partitioning problem is the set of balanced partitions. The set of optimal solutions $S^* \subseteq S$ is defined as $\{s \in S : |C_s| = \min_{i \in S} |C_i|\}$.

To instantiate the template, it is necessary to define a neighborhood, an objective function, and a selection strategy. A simple local search algorithm for this problem starts from a balanced partition and moves to adjacent partitions of lesser cost. Given a balanced partition, its adjacent partitions can be obtained by swapping two nodes. More formally, the neighborhood $N(s)$ of a solution s is defined as

$$N(\langle V_1, V_2 \rangle) = \{s' = \langle V'_1, V'_2 \rangle : V'_1 = V_1 \setminus \{a\} \cup \{b\} \wedge V'_2 = V_2 \setminus \{b\} \cup \{a\}\} \quad \forall a \in V_1, b \in V_2$$

It is interesting to review this example to observe the numerous degrees of freedom in local search algorithms.

First, the above local search algorithm only considers states that are solutions. This is not a requirement and many local search algorithms consider states that are not solutions. They use a different objective function to guide the search towards a solution of optimal value. Section 7.3 presents such an algorithm for graph partitioning.

Second, there may be many neighbors that can be chosen at any time. A local search algorithm may choose the best neighbor or any neighbor that improves the objective function. It may even consider random moves and/or moves that decrease the value of the objective function. Of course, the neighborhood choice is crucial to achieve a good trade-off between computing time and quality of the resulting solutions.

2.2 Formalization

This section introduces a framework for local search and formalizes essential concepts that help in the design process of a specific algorithm.

We consider a combinatorial optimization problem \mathcal{P} of the form

$$\begin{aligned} \min f(\vec{x}) \quad & \text{subject to} \\ C_1(\vec{x}) \\ & \vdots \\ C_n(\vec{x}) \end{aligned}$$

where x is a vector ($|x| = n$) of discrete decision variables for the problem, f is a mapping $f : \mathcal{N}^n \rightarrow \mathcal{N}$ that associates a variable assignment with a performance measure and C_1 through C_n are Boolean functions defining the solution space. A solution to a combinatorial optimization problem is a variables assignment \hat{x} that satisfies $\{C_1(\hat{x}), \dots, C_n(\hat{x})\}$. Let $\mathcal{L}_{\mathcal{P}}$ be the set of all solutions for a given problem. The set of optimal solutions to a problem is then defined as $\mathcal{L}_{\mathcal{P}}^* \subseteq \mathcal{L}_{\mathcal{P}}$ such that

$$\mathcal{L}_{\mathcal{P}}^* = \{s \in \mathcal{L}_{\mathcal{P}} \mid f(s) = \min_{k \in \mathcal{L}_{\mathcal{P}}} f(k)\}$$

Definition 1 The *search space* for a combinatorial optimization problem \mathcal{P} is a set $\widetilde{\mathcal{L}}_{\mathcal{P}}$ such that $\mathcal{L}_{\mathcal{P}} \subseteq \widetilde{\mathcal{L}}_{\mathcal{P}} \subseteq \mathcal{N}^n$. Elements of the set $\widetilde{\mathcal{L}}_{\mathcal{P}}$ often satisfy a relaxation (i.e., a subset) of $\{C_1, \dots, C_n\}$.

Definition 2 A *Neighborhood* is a pair $(\widetilde{\mathcal{L}}_{\mathcal{P}}, N)$ where $\widetilde{\mathcal{L}}_{\mathcal{P}}$ is a search space and N is a mapping $N : \widetilde{\mathcal{L}}_{\mathcal{P}} \rightarrow 2^{\widetilde{\mathcal{L}}_{\mathcal{P}}}$ that defines, for each solution s , the set of adjacent solutions $N(s) \subseteq \widetilde{\mathcal{L}}_{\mathcal{P}}$. Whenever the relation $s \in N(j) \Leftrightarrow j \in N(s)$ holds, the neighborhood is said to be symmetric.

Definition 3 The *Transition graph* $G(\widetilde{\mathcal{L}_{\mathcal{D}}}, N)$ associated to a neighborhood $\langle \widetilde{\mathcal{L}_{\mathcal{D}}}, N \rangle$ is the graph where nodes, called computation state, are elements of $\widetilde{\mathcal{L}_{\mathcal{D}}}$ and arcs of the form $a \rightarrow b$ exists in G for all a and b such that $b \in N(a)$.

Definition 4 A solution s in $\mathcal{L}_{\mathcal{D}}$ is *locally optimal* with respect to N if $f(s) \leq \min_{i \in N(s)} f(i)$. The set of locally optimal solutions with respect to N is denoted $\mathcal{L}_{\mathcal{D}}^+$. Note that local optimality is defined with respect to a specific neighborhood function.

Definition 5 The *acceptance* is a function $A : 2^{\widetilde{\mathcal{L}_{\mathcal{D}}}} \rightarrow 2^{\widetilde{\mathcal{L}_{\mathcal{D}}}}$ that filters a set of elements of the search space, i.e., $A(\Omega) \subseteq \Omega$ where Ω is an element of $\widetilde{\mathcal{L}_{\mathcal{D}}}$.

Definition 6 A *selection rule* S is a function $S : 2^{\widetilde{\mathcal{L}_{\mathcal{D}}}} \rightarrow \widetilde{\mathcal{L}_{\mathcal{D}}}$ that picks an element s from its input set according to some strategy, i.e., $s = S(\Omega) \in \Omega$

Definition 7 A *local search algorithm* is an iterative process that produces a sequence of computation states s_1, s_2, \dots, s_k that satisfy the property

$$s_{i+1} = S(A(N(s_i))) \quad (1 \leq i \leq k)$$

The hope is to produce a final computation state s_k that belongs to $\mathcal{L}_{\mathcal{D}}^+$ (with respect to a given neighborhood function N), or even better, to $\mathcal{L}_{\mathcal{D}}^*$.

For convenience, the rest of the thesis abuses language and uses the terms transition graph and neighborhood interchangeably. The rest of this section further assumes, without loss of generality, that all problems are expressed as minimization.

2.3 Design Decisions

The discussion and the formalization of a local search algorithm clearly indicate that there are many degrees of freedom in the design of a local search algorithm. At the very least, it is necessary to choose the functions N, f, A and S in order to produce a specific algorithm. This section investigates some of the choices that influence the design of these four functions.

2.3.1 Design Choices for the Neighborhood

Designing a neighborhood is a big decision in any local search algorithm. This remains essentially an art, although substantial research is devoted to understanding what constitutes a good neighborhood. We now review some properties of interest.

Connectivity is an important property of neighborhoods, since it is a necessary condition to prove asymptotic convergence results. The neighborhood $\langle \widetilde{\mathcal{L}}_{\mathcal{P}}, N \rangle$ is said to be connected whenever there is at least one node s^* in $\mathcal{L}_{\mathcal{P}}^*$ that belongs to $\mathcal{L}_{\mathcal{P}}^+$ and, for any node s in $\widetilde{\mathcal{L}}_{\mathcal{P}}$ there is a path from s to s^* in $G(\widetilde{\mathcal{L}}_{\mathcal{P}}, N)$. Section 7.4 presents a connected neighborhood for job-shop scheduling.

Second, the size of $N(s)$ for any s in $\widetilde{\mathcal{L}}_{\mathcal{P}}$ is a key parameter in the design of N . Large neighborhoods are, in general, more expensive to explore but tend to produce better solutions. Smaller neighborhoods are cheaper to explore, but the small size damages the chances to have a connected neighborhood.

Third, we characterized $\widetilde{\mathcal{L}}_{\mathcal{P}}$ as satisfying $\mathcal{L}_{\mathcal{P}} \subseteq \widetilde{\mathcal{L}}_{\mathcal{P}} \subseteq \mathcal{N}^n$. When $\mathcal{L}_{\mathcal{P}} = \widetilde{\mathcal{L}}_{\mathcal{P}}$, the neighborhood always produces solutions to the instance \mathcal{P} and the algorithm remains simple. The other extreme option is to choose $\widetilde{\mathcal{L}}_{\mathcal{P}} = \mathcal{N}^n$. The difficulty is to choose a good objective function that reflects both solutions quality and the satisfaction of the constraint set $\{C_1, \dots, C_n\}$. The advantage of the approach is the increased flexibility. Indeed, it is always possible to design a connected neighborhood. The downside is that it is also necessary to superimpose, on the local search algorithm that explores $N(s)$, mechanisms to guide the algorithm towards computation state that represents solutions in $\mathcal{L}_{\mathcal{P}}$.

2.3.2 Design Choices for the Objective Function

The choice of objective function is intimately related to the choice of search space and neighborhood. The objective function can be used in the actual definition of the functions N, A and S . Its role is to guide the algorithm to the optimal solution. It can be designed so that local minima correspond to actual solutions, i.e., all local minima s of f belong to $\widetilde{\mathcal{L}}_{\mathcal{P}}$ and also to $\mathcal{L}_{\mathcal{P}}$. This property proves extremely useful to facilitate a local search design with a search space $\widetilde{\mathcal{L}}_{\mathcal{P}} \supset \mathcal{L}_{\mathcal{P}}$ where some constraints have been relaxed. Indeed, the search for a local minimum of f also indirectly searches for an actual solution to the problem \mathcal{P} . Section 7.2 illustrates this idea with a simulated annealing algorithm for graph coloring.

Ideally, the objective function f ought to have few local minima, hence few global minima. To effectively guide the algorithm, the ideal f should discriminate between solutions (“flat” functions offer little guidance), especially when the solutions are adjacent, hence only slightly different.

Guided Local Search is a subclass of local search algorithms that exploit f by dynamically adapting it to offer better guidance when the algorithm gets trapped in local minima. The

idea behind those algorithms is to parameterize f to alter the shape of the function and eliminate the local minima.

2.3.3 Design Choices for the Acceptance Criterion

The acceptance criterion embodied by the function A is a major component of a local search algorithm. Entire classes of algorithms can be captured with the specification of A .

Local Improvement Local improvement uses the definition

$$A : \widetilde{2^{\mathcal{L}_{\mathcal{P}}}} \rightarrow \widetilde{2^{\mathcal{L}_{\mathcal{P}}}} \quad \text{s.t.} \quad A(\Omega) = \{n \in \Omega \mid f(n) < f(s_i)\}$$

where s_i is the i^{th} ($i < k$) computation state in the sequence s_1, s_2, \dots, s_k produced by the algorithm $S \circ A \circ N$.

Threshold Algorithms Threshold algorithms use a different function A . Intuitively, they always accept improving moves and may accept worsening ones. Before defining A , it is useful to define a computation trace for a threshold algorithm as the pair of sequences

$$\begin{aligned} s_1, s_2, \dots, s_k \\ c_2, c_2, \dots, c_k \end{aligned}$$

where the sequence s_1, \dots, s_k is a local search sequence (i.e., a sequence of computation state that satisfies $s_{i+1} = S(A(N(s_i)))$ for all $1 \leq i \leq k$) and c_1, c_2, \dots, c_k is a sequence of real valued constants satisfying $0 \leq c_{k+1} \leq c_k$ and $\lim_{k \rightarrow \infty} c_k = 0$. The definition of A follows as

$$A : \widetilde{2^{\mathcal{L}_{\mathcal{P}}}} \rightarrow \widetilde{2^{\mathcal{L}_{\mathcal{P}}}} \quad \text{s.t.} \quad A(\Omega) = \{n \in \Omega \mid f(n) - f(s_i) \leq c_i\}$$

where s_i is once again the i^{th} computation state in the sequence s_1, \dots, s_k produced by the algorithm $S \circ A \circ N$.

Tabu The acceptance criterion for tabu search algorithms takes yet another form. Intuitively, tabu algorithms use special purpose data structures to identify computation states that have undesirable properties. These data structures are highly problem dependent. Let us assume that the structure takes the form of a Boolean function $T : \widetilde{\mathcal{L}_{\mathcal{P}}} \rightarrow \text{Boolean}$ that answers the question “Is this state tabu?”. The acceptance function is then defined as

$$A : \widetilde{2^{\mathcal{L}_{\mathcal{P}}}} \rightarrow \widetilde{2^{\mathcal{L}_{\mathcal{P}}}} \quad \text{s.t.} \quad A(\Omega) = \{n \in \Omega \mid \neg T(n)\}$$

2.3.4 Design Choices for the Search Procedure

The selection of a specific search procedure is captured by the choice of function for S . Three classic choices are available: best, first and random.

Best Strategy The idea is to select the new computation state from a set based on a local optimality condition. S is thus defined as

$$S : \widetilde{2^{\mathcal{L}_{\mathcal{P}}}} \rightarrow \widetilde{\mathcal{L}_{\mathcal{P}}} \quad \text{s.t.} \quad S(\Omega) \in \{n \in \Omega \mid f(n) = \min_{i \in \Omega} f(i)\}.$$

Ties are resolved with randomization in which case S returns any element of the optimal subset with equal probability.

First Strategy This strategy has an inherent operational semantics. The idea is to impose an ordering on the set Ω that is passed as input to S . S then proceeds by returning as its output the very first element of Ω .

Random Strategy This last option simply defines S as

$$S : \widetilde{2^{\mathcal{L}_{\mathcal{P}}}} \rightarrow \widetilde{\mathcal{L}_{\mathcal{P}}} \quad \text{s.t.} \quad S(\Omega) = \text{random}(\Omega)$$

where random is a function that picks an element from its input set at random with a uniform probability, i.e.,

$$\Pr[n = \text{random}(\Omega)] = \frac{1}{|\Omega|} \quad \forall s \in \Omega.$$

2.4 Overview of some Traditional Approaches.

A Local Search algorithm is captured with the composition $S \circ A \circ N$ of the search, acceptance, and neighborhood functions, all relying on the objective function f . Each function can be picked independently and the composition leads to a specific local search algorithm. This sections reviews some of the classic composition for S and A . We assume that N and f are fixed since those are problem dependent.

2.4.1 Local Improvement

Local improvement was first introduced with a 2-exchange neighborhood for TSP known as 2-opt. The original idea behind local improvement is to accept transition to states that are strictly better with respect to the function f . Actual algorithms are obtained by picking a specific function for S .

Random Walk Random walk picks an n in $A(N(s))$ at random. The stochastic component plays an important role in the overall performance of the algorithm. If it incorporates knowledge about which part of the neighborhood is likely to contain “good” neighbors, it can guide the algorithm in this direction. The approach has a modest computational cost, since a transition reduces to the generation of a single element from $A(N(s))$.

Random walk cannot, in general, determine local optimality. Instead, the termination condition relies on a probabilistic criterion. As the number of failed transitions grows larger than $|N(s)|$, the probability that s is locally optimal converges to 1. Random walk also generally limits the number of transitions.

Best Improvement In the best improvement variant, the selection function S picks the best acceptable element of $N(s)$. From a computational standpoint, an inconvenient of this greedy strategy is the necessity to scan the whole neighborhood to find the best transition. If all the elements in $N(s)$ have a performance measure worse than the current solution s , the solution s is locally optimal and the algorithm terminates. As for random walk, it is also possible to limit the number of transitions.

First Improvement First improvement is a compromise between random walk and best improvement. Here, the algorithm scans $N(s)$ until it finds a neighbor n that yields a better $f(n)$. This algorithm determines local optimality when there is no neighbor in $N(s)$ that leads to an improvement.

Note that all the strategies described above can be combined with the more permissive acceptance criterion

$$A : \widetilde{2^{\mathcal{L}_{\mathcal{D}}}} \rightarrow \widetilde{2^{\mathcal{L}_{\mathcal{D}}}} \quad \text{s.t.} \quad A(\Omega) = \{n \in \Omega \mid f(n) \leq f(s_i)\}$$

Instead of restricting valid transitions to transitions that improve the value of f , it is restricted to transitions that do not degrade the value of f . This rule allows the algorithm to transform the current solution in presence of a status quo with respect to the function f .

2.4.2 Threshold Algorithms

It is sometimes too restrictive to require an improvement. This is due to the fact that specific combinations of N and f can lead to situations where it is not possible to find a sequence of computation states s_1, s_2, \dots, s_k that have a monotonically decreasing sequence $f(s_1), f(s_2), \dots, f(s_k)$ long enough to get a good final state s_k .

An alternative is to further relax the acceptance rule and allow transitions that lead to worse neighbors. Threshold algorithms form a class of algorithmic solutions following this idea. Classic threshold algorithms often combine the acceptance criterion described in the previous section with a best search strategy.

Intuitively, if, at any iteration, the selected neighbor n improves on s , then $f(n) - f(s) \leq 0$ and the transition is accepted since $c_k \geq 0$ for all k . If n is worse than s , $f(n) - f(s) > 0$ and the acceptance of the transition is now conditional to the magnitude of the worsening since the criterion is $c_k \geq f(n) - f(s)$. Because the sequence of c_k is monotonically decreasing, the more time passes, the more the algorithm behaves like a best local improvement.

Simulated annealing [34], [31], is a special case of threshold algorithms where the constants c_k are replaced by random variables a_k and the search strategy is random. The requirement on the sequence of c_k is replaced by

$$0 \leq \mathcal{E}[a_{k+1}] \leq \mathcal{E}[a_k] \quad \forall k > 0$$

i.e., the expected value of a at iteration $k + 1$ should be less than the expected value of a at iteration k . The original definition of Kirkpatrick, Gelatt & Vecchi [31], relies on an exponential distribution for the random variable a_k . The distribution is parameterized by the distance $f(n) - f(s)$ and a real valued parameter t_k that monotonically decreases with the number of iterations, or, more precisely

$$Pr[\text{accept } n] = \begin{cases} 1 & \text{if } f(n) \leq f(s) \\ e^{-\frac{f(n)-f(s)}{t_k}} & \text{if } f(n) > f(s) \end{cases}$$

The mechanism responsible for determining t_k as a function of the iteration number k is called the cooling schedule. The quality of the schedule directly affects the quality of solutions produced and the time requirements for the algorithm.

An additional challenge when designing threshold algorithms, or even simulated annealing algorithms, is to come up with “good” thresholds or “good” cooling schedules.

2.4.3 Tabu Search

Classic tabu search algorithms [22], [23] are based on an acceptance criterion

$$A(\Omega) = \{n \in \Omega \mid \neg T(s)\}$$

and a best search strategy. The role of the Boolean function T , i.e., the tabu structures is to filter the neighborhood to eliminate undesirable computation states.

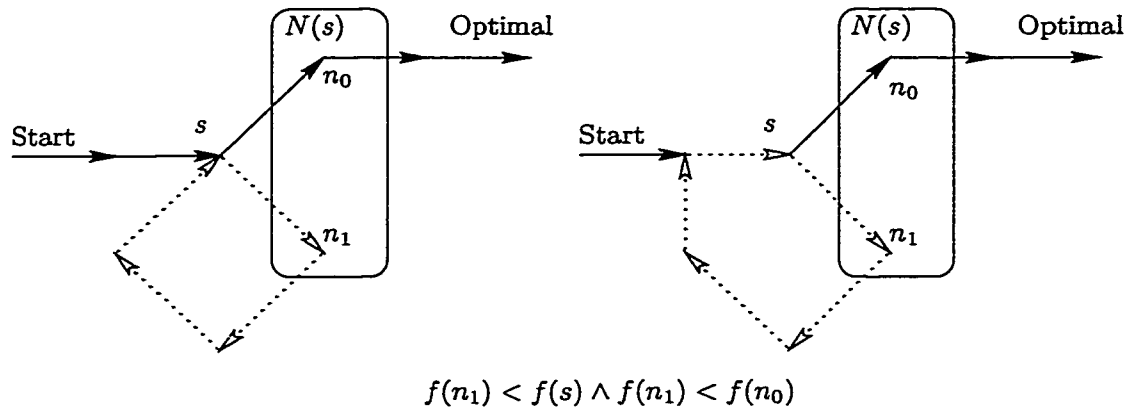


Figure 2.2: The Impact of Limiting Tabu Status.

The permissiveness of the search strategy has a negative impact. The algorithm can cycle, especially with symmetric neighborhoods. The Boolean function T answers this problem by marking some states as undesirable.

Tabu Structures Tabu algorithms use memory structures to store attributes of states visited in the recent past. A first attempt is to set aside the whole state and define $T(n)$ as $n \in M$ where M is the memory storing the previously seen states. However, this proves too limiting. Consider, for instance, the left transition graph of Figure 2.2. Permanently marking all states tabu prevents the algorithm from visiting node s twice and therefore from exploring a path leading to an optimal solution. If instead, the algorithm chooses to mark transition as tabu, the pitfall of the left schema can be avoided. However, the right schema reveals that it is still possible for the algorithm to miss the path leading to the global optima.

To overcome the difficulty, a tabu status of a transition is limited in time, so that other transitions from a given state get a chance to be followed.

Tabu Approximation Storing the whole state, or a specific transition has a prohibitive memory cost. A realistic tabu algorithm prefers to store generic transitions. Instead of marking the transition $s \rightsquigarrow n$ as tabu, it stores the characteristics of the transition itself, independently of the rest of the move. To make the idea more concrete, a tabu algorithm for graph partitioning could store the pair of vertices exchanged as tabu, not the pair of states. Consequently, the approximation of a specific transition $s \rightsquigarrow n$ with the parameters of the transition filters similar transitions.

Tabu Aspiration Suppose a neighbor n of the current state s has an outstanding cost $f(n)$, much smaller than $f(s)$, or even better than the current best. Further assume that the state n is currently tabu because it exhibits a characteristic shared by another state visited in the recent past. State n could possibly be mis-diagnosed as tabu because of the tabu approximations. It is thus desirable to overrule the tabu status and accept the transition towards state n when such exceptional conditions occur. This idea is referred to as the aspiration level and can be modeled as a refinement of the function T .

Diversification and Intensification A potential limitation for all local search algorithms is to keep on exploring the same part of the search space. If the algorithm gets a better chance at visiting different regions of the search space, it has better chances at finding a good solution. The opposite observation can also be made. Some regions of the search space may appear promising and forcing the algorithm to remain in that region can be profitable.

Both considerations can be seen as a form of Guided Local Search. In the tabu search community, they are referred to as diversification and intensification. Diversification and intensification can be obtained with dynamic alterations to the function f and appropriate restarting strategies.

2.5 Beyond Classic Techniques

Local search also encompasses other techniques. Genetic algorithms and neural networks are probably the two most important classes of local search algorithms with the techniques described above. Genetic algorithms are based on an evolutionary metaphor and work with sets of solutions called populations. These techniques mix the features of different individual solutions of the population and introduces perturbations called mutation. Good solutions are retained with a selection process reminiscent of natural selection. These last two classes of algorithms are not within the scope of this thesis, however the core ideas embedded in genetic algorithm are good candidates for future work.

Chapter 3

A Tour of LOCALIZER

This chapter gives a brief overview of the main concepts of LOCALIZER. It starts by reviewing the computational model of LOCALIZER and the general form of LOCALIZER statements. It then considers the two main building blocks of LOCALIZER: invariants and neighborhoods.

3.1 The Computation Model

To understand LOCALIZER, it is best to review its underlying computational model first. Figure 3.1 depicts the computational model of LOCALIZER for decision problems. This model captures the essence of most local search algorithms. The algorithm performs a number of local searches (up to **MaxSearches** and while a global condition is satisfied). Each local search consists of a number of iterations (up to **MaxTrials** and while a local condition is satisfied). For each iteration, the algorithm first tests if the state is satisfiable, in which case a solution has been found. Otherwise, it selects a candidate in the neighborhood and moves to this new state if this is acceptable. If no solution is found after **MaxTrials** or when the local condition is false, the algorithm restarts a new local iteration in the state *restartState(s)*. The computation model for optimization problems is depicted in Figure 3.2 for the case of a maximization. The main difference is the need in lines 6 to 10 to update the best solution so far when necessary.

3.2 The Structure of LOCALIZER Statements

LOCALIZER statements specify, for the problem at hand, the instance data, the state, and the generic parts of the computation model (e.g., the neighborhood and the acceptance criterion). They consist of a number of sections as depicted in Figure 3.3. The instance

```

procedure LOCALIZER
begin
  1  $s := \text{startState}();$ 
  2 for  $\text{search} := 1$  to  $\text{MaxSearches}$  while Global Condition do
  3   for  $\text{trial} := 1$  to  $\text{MaxTrials}$  while Local Condition do
  4     if  $\text{satisfiable}(s)$  then
  5        $\text{return } s;$ 
  6      $\text{select } n \text{ in } \text{neighborhood}(s);$ 
  7     if  $\text{acceptable}(n)$  then
  8        $s := n;$ 
  9    $s := \text{restartState}(s);$ 
end

```

Figure 3.1: The Computation Model of Localizer for Decision Problems

```

procedure LOCALIZER
begin
  1  $\text{bestBound} := -\infty;$ 
  2  $s := \text{startState}();$ 
  3 for  $\text{search} := 1$  to  $\text{MaxSearches}$  while Global Condition do
  4   for  $\text{trial} := 1$  to  $\text{MaxTrials}$  while Local Condition do
  5     if  $\text{satisfiable}(s)$  then
  6       if  $\text{value}(s) > \text{bestBound}$  then
  7          $\text{bestBound} := \text{value}(s);$ 
  8          $\text{best} := s;$ 
  9      $\text{select } n \text{ in } \text{neighborhood}(s);$ 
  10    if  $\text{acceptable}(n)$  then
  11       $s := n;$ 
  12     $s := \text{restartState}(s);$ 
  13 return  $\text{best};$ 
end

```

Figure 3.2: The Computation Model of Localizer for Maximization Problems

data is defined by the **Type**, **Constant**, and **Init** sections, using traditional data structures from programming languages. The state is defined as the values of the variables. The neighborhood is defined in the **Neighborhood** section, using objects from previous sections. The acceptance criterion is part of the definition of the neighborhood. The initial state is defined in section **Start**. The restarting states are defined in section **Restart**, the parameters (e.g. **MaxTrials**) are given in the **Parameter** section, and the global and local conditions are given in sections **Global Condition** and **Local Condition**. Note that all the identifiers in boldface in the description of computation model (e.g., **search** and **trial**) are in fact keywords of LOCALIZER.

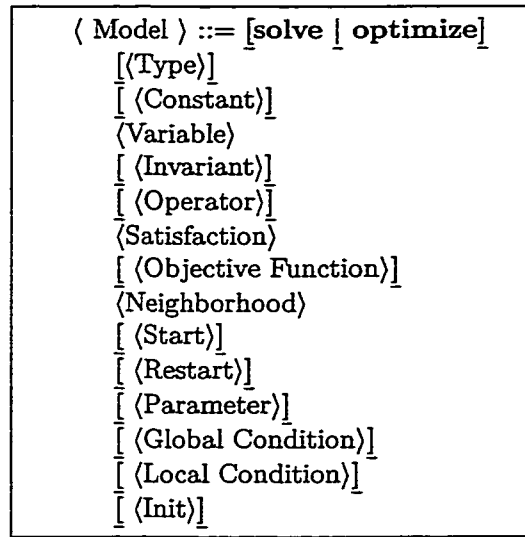


Figure 3.3: The Structure of LOCALIZER Statements

As mentioned previously, the most original aspects of LOCALIZER are in the specifications of the neighborhood and the acceptance criterion. Of course, some of the notations are reminiscent of languages such as AMPL and Claire at the syntactical level but the underlying concepts are fundamentally different. In the rest of this chapter, we describe the most original aspects of LOCALIZER without trying to be comprehensive.

3.3 The Running Example

This overview mostly uses Boolean satisfiability to illustrate the concepts of LOCALIZER. A Boolean satisfiability problem amounts to finding a truth assignment for a Boolean formula expressed in conjunctive normal form. The input is given as a set of clauses, each clause consisting of a number of literals. As is traditional, a literal is simply an atom (positive atom) or the negation of an atom (negative atom). A clause is satisfied as soon as at least one of its positive atoms is true or at least one of its negative atoms is false. The local search statement considered for Boolean satisfiability is based on the GSAT algorithm by Selman et al. in [68], where a local search move consist of flipping the truth value of an atom.

A local improvement statement for Boolean satisfiability is described in Figure 3.4. In the statement, atoms are represented by integers 1 to n and a clause is represented by two sets: the set of its positive atoms p and the set of its negative atoms n . This data representation is specified in the Type section. A problem instance is specified by an array

of m clauses over n variables. The instance data is declared in the **Constant** section and initialized in the **Init** section which is not shown. The state is specified by the truth values of the atoms and is captured in the array a of variables in the **Variable** section. Variable $a[i]$ represents the truth value of atom i . The **Invariant** section is the key component of all LOCALIZER statements: it describes, in a declarative way, the data structures which must be maintained incrementally. Invariants are reviewed in detail in Section 3.4. In the statement depicted in Figure 3.4, they maintain the number of true literals $nbtl[c]$ in each clause c and the number of satisfied clauses $nbClauseSat$. The **Satisfiable** section describes when the state is a solution (all clauses are satisfied), while the **Objective Function** section describes the objective function (maximize the number of satisfied clauses) used to drive the search. The **Neighborhood** section describes the actual neighborhood and the acceptance criterion. The neighborhood consists of all the states which can be obtained by flipping the truth value of an atom and a move is accepted if it improves the value of the objective function. The **Neighborhood** section is another important part of LOCALIZER and is reviewed in more detail in Section 3.5. The **Start** and **Restart** sections describe how to generate an initial state and a new state when restarting a search. They both use a simple random generation in the statement.

It is interesting at this point to stress the simplicity of the statement of Figure 3.4, since it is difficult to imagine a more concise formal statement of the algorithm.

3.4 Invariants

Invariants are probably the most important tool offered by LOCALIZER to support the design of local search algorithms. They make it possible to specify **what** needs to be maintained incrementally without considering **how** to do so. Informally speaking, an invariant is an expression of the form $v : t = exp$ and LOCALIZER guarantees that, at any time during the computation, the value of variable v of type t is the value of the expression exp (also of type t). For instance, the invariant

$$nbtl : \text{array}[i \text{ in } 1..m] \text{ of int} = \text{sum}(j \text{ in } cl[i].p) a[j] + \text{sum}(j \text{ in } cl[i].n) !a[j];$$

in the Boolean satisfiability statement specifies that $nbtl[c]$ is equal to the sum of all true positive atoms and all false negative atoms in clause c , for all clauses in $1..n$. LOCALIZER uses efficient incremental algorithms to maintain these invariants during the computation, automating one of the tedious and time-consuming tasks of local search algorithms. For instance, whenever a value $a[k]$ is changed, $nbtl[c]$ is updated in constant time.

```

Solve
Type:
  clause = record
    p : {int};
    n : {int};
  end;
Constant:
  m: int = ...;
  n: int = ...;
  cl: array[1..m] of clause = ...;
Variable:
  a: array[1..n] of boolean;
Invariant:
  nbtl : array[ i in 1..m] of int = sum(i in cl[i].p) a[j] + sum(j in cl[i].n) !a[j];
  nbClauseSat : int = sum(i in 1..m) (nbtl[i] >0);
Satisfiable:
  nbClauseSat = m;
Objective Function:
  maximize nbClauseSat;
Neighborhood:
  move a[i] := !a[i]
  where i from {1..n}
  accept when improvement;
Start:
  forall(i in 1..n) a[i] := random({true,false});
Restart:
  forall(i in 1..n) a[i] := random({true,false});

```

Figure 3.4: A Local Improvement statement for Boolean Satisfiability

LOCALIZER allows a wide variety of invariants over complex data structures. The invariant (also from the Boolean satisfiability statement)

```
nbClauseSat : int = sum(i in 1..m) (nbtl[i] >0);
```

illustrates the use of relations inside an invariant. A relation, when used inside an expression, is considered a 0-1 integer, i.e., the relation evaluated to 1 when true and 0 otherwise. The excerpt

```

C          : array[1..n] of int = distribute(x,1..n,1..n);
Empty      : {int} = { i : int | select i from 1..n where size(C[i]) = 0 };
NEmpty     : {int} = { i : int | select i from 1..n where size(C[i]) > 0 };
unused     : int = minof(Empty);
Candidates : {int} = NEmpty union unused;
B          : array[k in 1..n] of {edge} = { (<i,ji from C[k]
                                                    select j from C[k]
                                                    where A[i,j];
f          : int = sum(i in 1..n) (2×size(C[i])×size(B[i]) - size(C[i])2)
countB     : int = sum(i in 1..n) size(B[i]);

```

is taken from a graph-coloring statement implementing an algorithm in [29]. The graph-coloring problem amounts to finding the smallest number of colors to label a graph such that adjacent vertices have different colors. Let us review some of the main ideas behind the algorithm before discussing the invariants. For a graph with n vertices, the algorithm considers n colors that are the integers between 1 and n . Color class C_i is the set of all vertices colored with i and the bad edges of C_i , denoted by B_i , are the edges whose vertices are both colored with i . The main idea of the algorithm is to minimize the objective function $\sum_{i=1}^n 2|B_i||C_i| - |C_i|^2$ whose local minima are valid colorings. To minimize the function, the algorithm chooses a vertex and chooses a color whose color class is non-empty or one of the unused colors. It is important to consider only one of the unused colors to avoid a bias towards unused colors.

The invariant

```

B: array[k in 1..n] of {edge} = { (<i,ji from C[k]
                                                    select j from C[k]
                                                    where A[i,j];

```

expresses that $B[k]$ is the set of bad edges obtained by selecting two adjacent vertices in color class k . It illustrates that LOCALIZER can maintain queries over sets varying in time (since $C[k]$ evolves during the local search). The invariant

```

C: array[1..n] of int = distribute(x,1..n,1..n);

```

is equivalent to, but more efficient than,

```

C: array[i in 1..n] of int = { j : int | select i from 1..n where x[j] = i };

```

This function is provided as a primitive in LOCALIZER, since it is useful in a large variety of applications. The invariant

```

NEmpty : {int} = { i : int | select i from 1..n where size(C[i]) > 0 };

```

defines the non-empty classes. Note that here the set $1..n$ does not vary but the condition $\text{size}(C[i]) > 0$ evolves over time.

Once again, it is important to emphasize the significant support provided by LOCALIZER with invariants. These invariants maintain complex data structures incrementally, but users only have to specify them in a declarative way.

3.5 The Neighborhood

As discussed in Chapter 1, many strategies such as local improvement, simulated annealing, and tabu search have been proposed in the last decades for local search algorithms. This section reviews how they are modeled in the **Neighborhood** section, the other fundamental conceptual tool provided by LOCALIZER. We start by reviewing the structure of neighborhood specifications before presenting how traditional meta-heuristics can be expressed.

3.5.1 Neighborhood Specifications

Neighborhoods are specified in LOCALIZER by instructions based on the skeleton:

```

move  $op(x_1, \dots, x_n)$ 
where
   $x_1$  from  $S_1$ ;
  ...
   $x_n$  from  $S_n$ 
[ accept when  $\langle \text{AcceptanceCriterion} \rangle$  ]

```

The move instruction uses both declarative and procedural components. The first part of the statement specifies, with the imperative code $op(x_1, \dots, x_n)$, the transformation of the current state into one of its neighbors. op is an operator or a sequence of instructions using traditional programming language constructs. Assuming that τ is the current configuration and $Post(\tau, i)$ represents the configuration obtained by executing i in τ , the instruction defines the neighborhood

$$\{Post(\tau, op(x_1, \dots, x_n)) \mid x_1 \in S_1 \ \& \ \dots \ \& \ x_n \in S_n\},$$

selects one of its elements, and checks if it satisfies the acceptance criterion (if any). The modeling effort is primarily devoted to the definition of the sets S_i and the invariants they are based on. The sets in the **where** statement can be defined inline, can refer to constant

or invariants and are possibly augmented with filtering clauses. The details on the syntax are provided in Chapter 4.

Neighborhoods in LOCALIZER are also defined by instructions based on the skeleton:

```

best move  $op(x_1, \dots, x_n)$ 
where
     $x_1$  from  $S_1$ ;
    ...
     $x_n$  from  $S_n$ 
[ accept when  $\langle \text{AcceptanceCriterion} \rangle$  ]

```

The instruction defines the same neighborhood as before but selects the move m with the best value of the objective function in $Post(\tau, m)$. More formally, the objective function can be viewed as the definition of a mapping $f : \tau \rightarrow \mathbb{R}$ from computation states to real numbers and the addition of the keyword **best** specifies to select an element from the set $B(\tau)$ defined as

$$\begin{aligned}
 N(\tau) &= \{Post(\tau, op(x_1, \dots, x_n)) \mid x_1 \in S_1 \ \& \ \dots \ \& \ x_n \in S_n\} \\
 B(\tau) &= \{\tau' \in N(\tau) \mid f(\tau') = \max_{s \in N(\tau)} f(s)\}
 \end{aligned}$$

for a maximization problem and to submit it to the acceptance criterion. Note that, in this case too, the acceptance criterion is applied last. It is also important to mention the concept of difference between the values of the objective function in states τ and τ' . The difference, which is used in most of the strategies, is computed automatically by LOCALIZER and can be accessed directly (using the keyword **delta**) or indirectly (using acceptance criteria such as **improvement**).

The ability to specify complex set expressions in the **where** clause makes it possible to develop complex neighborhoods. For instance, a move instruction of the form

```

move
     $q[i] := v$ 
where
     $i$  from  $Conflicts$ ;
     $v$  from  $\{1..n\}$  minimizing
         $sizeof(\{ j : \text{integer} \mid \text{select } j \text{ from } 1..n \text{ where } q[j] = v \text{ or } q[j] = v + i - j \text{ or } q[j] = v + j - i \})$ 
accept when ...

```

illustrates how the min-conflict heuristics of [38] can be expressed in LOCALIZER.

Finally, LOCALIZER also allows neighborhoods to be composed. For instance, the neighborhood

```

    try
      Pr(0.1):
        move
           $a[i] := !a[i]$ 
        where
           $i$  from OccurInUnsatClause
        accept when always ...;
      default:
        best move
           $a[i] := !a[i]$ 
        where
           $i$  from  $\{1..n\}$ 
        accept when noDecrease;
    end

```

implements the random walk/noise strategy of GSAT. Here, LOCALIZER flips an arbitrary variable in an unsatisfied clause with a probability of 0.1 and applies the standard strategy with a probability of 0.9. Note that LOCALIZER simply goes to the next iteration if the selected neighborhood is empty, since other neighborhoods may be non-empty.

3.5.2 The Acceptance Criteria

This section provides some overview of the acceptance criteria and it shows how they can be used to specify traditional local search strategies.

3.5.2.1 Local Improvement

Local improvement, and its variations, can be implemented in various ways in LOCALIZER.

Stochastic Local Improvement The statement depicted in Figure 3.4 uses a stochastic local improvement approach. The neighborhood section

```

Neighborhood:
  move  $a[i] := !a[i]$ 
  where  $i$  from  $\{1..n\}$ 
  accept when improvement;

```

specifies the following strategy: randomly select a value i in $1..n$ (i.e., select an atom), flip $a[i]$, and take the move if the resulting state improves the value of the objective function. If the state does not improve the value of the objective function, the move is not taken and LOCALIZER proceeds to the next iteration of the innermost loop in the computational model. Note that the keyword **improvement** is an abbreviation of the condition $\text{delta} > 0$.

Greedy Local Improvement The stochastic local improvement strategy can be made greedy by adding the keyword **best** in front of the move instruction, as in

Neighborhood:
best move $a[i] := !a[i]$
where i **from** $\{1..n\}$
accept when improvement;

This excerpt specifies the following strategy: consider each value i in $1..n$, select the one with the best value of the objective function, and accept the move if it is an improvement. If the move is not an improvement, LOCALIZER terminates the innermost loop, since the next iteration will produce the same result. Note that this strategy explores the neighborhood $N(\tau)$ in a systematic way, while the previous strategy was selecting a random move and testing it for improvement.

Deterministic Local Improvement The neighborhood section

Neighborhood:
first move $a[i] := !a[i]$
where i **from** $\{1..n\}$
accept when improvement;

is another approach that explores the neighborhood systematically until a move improving the value of the objective function is found. More precisely, the exploration scans $N(\tau)$ until it finds an acceptable move. If no such move exists, LOCALIZER terminates the innermost loop of the computation model, since the next iteration will produce the same result.

Non degradation Sometimes it is important to allow more flexibility in the local search and to allow moves which may not improve the objective function. The neighborhood

Neighborhood:
best move $a[i] := !a[i]$
where i **from** $\{1..n\}$
accept when noDecrease;

accepts the best move which does not decrease the value of the objective function. The resulting model, depicted in Figure 3.5, captures the essence of GSAT. The keyword **noDecrease** is an abbreviation for the condition $\delta \geq 0$.

3.5.2.2 Simulated Annealing

A simulated annealing strategy for the Boolean satisfiability problem is easily expressed by the LOCALIZER neighborhood

```

Solve
Type:
  clause = record
    p : {int};
    n : {int};
  end;
Constant:
  m: int = ...;
  n: int = ...;
  cl: array[1..m] of clause = ...;
Variable:
  a: array[1..n] of boolean;
Invariant:
  nbtI : array[ i in 1..m] of int = sum(i in cl[i].p) a[j] + sum(j in cl[i].n) !a[j];
  nbClauseSat : int = sum(i in 1..m) (nbtI[i] > 0);
Satisfiable:
  nbClauseSat = m;
Objective Function:
  maximize nbClauseSat;
Neighborhood:
  best move a[i] := !a[i]
  where i from {1..n}
  accept when noDecrease;
Start:
  forall(i in 1..n) a[i] := random({true,false});
Restart:
  forall(i in 1..n) a[i] := random({true,false});

```

Figure 3.5: A GSAT-based Statement for Boolean Satisfiability

```

Neighborhood:
  move a[i] := !a[i]
  where i from {1..n}
  accept
    when improvement → ch++;
    cor noDecrease;
    cor Pr(exp(-delta/t)) : always → ch++;

```

The key novelty here is that the accept statement may have a number of acceptance conditions that are tried in sequence until one succeeds or all fail. In addition, each acceptance condition can be associated with an action. The simulated annealing neighborhood specifies that a move is accepted when it improves the objective function, when it does not decrease the objective function, or with the standard probability of simulated annealing, that depends on a temperature parameter and the variation **delta** of the objective function. Note that the variable *ch* is incremented when there is an improvement or a decrease in the objective function.

The complete statement for a simulated annealing approach to Boolean satisfiability is shown in Figure 3.6. The statement illustrates also several new features of LOCALIZER. The **Operator** section describes two procedures which are used subsequently in the **Start** and **Restart** sections. Operators in LOCALIZER uses traditional constructs from imperative programming languages (e.g., loops and conditions) as well as some new primitives for randomization. These features are once again described in more detail in Chapter 4. Note also the variables t (the temperature) and ch (the change counter) which are used in various places in the statement.

3.5.2.3 Tabu Search

We now show how a tabu search strategy can be implemented in LOCALIZER for the Boolean satisfiability problem. There are many ways to implement a tabu search and this sections simply presents one of the possibilities. See Chapter 7 for another discussion of this topic.

The neighborhood

```

Neighborhood:
  best move  $a[i] := !a[i]$ 
  where  $i$  from  $\{1..n\}$  such that  $!tabu(i)$ 
  accept when always  $\rightarrow t[v] := trial;$ 

```

is an excerpt from a simple tabu search for Boolean satisfiability. The tabu search keeps track of when an atom was last flipped by using the keyword **trial** that gives access to the current iteration number for the innermost loop of the computation model. An atom is tabu if it has been flipped recently, which can be expressed as

```

boolean tabu( $idx$  : int)
{
  return  $t[idx] > trial - tl;$ 
}

```

where tl is a parameter specifying the time an atom stays on the tabu list. Only atom that are not tabu are considered in the neighborhood. The complete statement is described in Figure 3.7. Of course, more complicated tabu search algorithms (e.g., using aspiration criteria to overwrite the tabu status or a tabu list whose size varies over time or diversification and intensification techniques) can be implemented easily.

3.6 Incrementality Issues

Incrementality is a fundamental issue in local search and it is thus appropriate to discuss it in some detail in this chapter. It will be a frequent theme of this thesis.

```

Solve
Type:
  clause = record
    p : {int};
    n : {int};
  end;
Constant:
  m: int = ...;
  n: int = ...;
  cl: array[1..m] of clause = ...;
Variable:
  a: array[1..n] of boolean;
  t: real;
  ch: int;
Invariant:
  nbtl : array[ i in 1..m] of int = sum(i in cl[i].p) a[j] + sum(j in cl[i].n) !a[j];
  nbClauseSat : int = sum(i in 1..m) (nbtl[i] > 0);
Operator:
  void genState() {
    forall(i in 1..n) x[i] := random({true,false});
    t := 2.0;
    ch := 0;
  }
  void lowTemp() {
    t := t * 0.95;
    ch := 0;
  }
Satisfiable:
  nbClauseSat = m;
Objective Function:
  maximize nbClauseSat;
Neighborhood:
  move a[i] := !a[i]
  where i from {1..n}
  accept
    when improvement → ch++;
    cor noDecrease
    cor Pr(exp(-delta/t)) : always → ch++;
Start: genState();
Restart: lowTemp();
  local condition: ch < n

```

Figure 3.6: A Simulated Annealing Statement for SAT

```

Solve
Type:
  clause = record
    p : {int};
    n : {int};
  end;
Constant:
  m: int = ...;
  n: int = ...;
  tl: int= 10;
  cl array[1..m] of clause = ...;
Variable:
  a: array[1..n] of boolean;
  t: array[1..n] of int;
Invariant:
  nbtl : array[ i in 1..m] of int = sum(i in cl[i].p) a[j] + sum(j in cl[i].n) !a[j];
  nbClauseSat : int = sum(i in 1..m) (nbtl[i] > 0);
Operator:
  void genState() {
    forall(i in 1..n) x[i] := random({true,false});
    forall(i in 1..n) t[i] := -tl;
  }
  boolean tabu(idx : int) {
    return t[idx] > trial - tl;
  }
Satisfiable:
  nbClauseSat = m;
Objective Function:
  maximize nbClauseSat;
Neighborhood:
  best move a[i] := !a[i]
  where i from {1..n} such that !tabu(i)
  accept when always → t[v] := trial;
Start:
  genState();
Restart:
  genState();

```

Figure 3.7: A Tabu Search Statement for Boolean Satisfiability

In the statements presented so far, LOCALIZER simulates the move to find out how the objective function evolves. This simulation can become very expensive when few moves are accepted. In practice, local search implementations often try to evaluate the impact of the move in the current state. LOCALIZER supports this practice by allowing to specify acceptance criteria that are evaluated in the current state. For instance, the neighborhood definition

Neighborhood:
 first move $a[i] := !a[i]$
 where i from $\{1..n\}$
 accept when in current state
 $gain[i] \geq 0$;

evaluates the condition $gain[i] \geq 0$ in the current computation state to determine whether to take the move. Of course, this requires to generalize the invariants to maintain $gain[i]$ incrementally. The invariants now become

Invariant:
 $nbtl$: array[i in $1..m$] of int = $\sum(i \text{ in } cl[i].p) a[j] + \sum(j \text{ in } cl[i].n) !a[j]$;
 $g01$: array[i in $1..n$] of int = $\sum(j \text{ in } po[i]) (nbtl[j] = 0) - \sum(j \text{ in } no[i]) (nbtl[j] = 1)$;
 $g10$: array[i in $1..n$] of int = $\sum(j \text{ in } no[i]) (nbtl[j] = 0) - \sum(j \text{ in } po[i]) (nbtl[j] = 1)$;
 $gain$: array[i in $1..n$] of int = if $a[i]$ then $g10[i]$ else $g01[i]$;
 $nbClauseSat$: int = $\sum(i \text{ in } 1..m) (nbtl[i] > 0)$;

It is worth spending some time understanding these invariants. The informal meaning is the following. $g01[i]$ represents the change in the number of satisfied clauses when changing the value of atom i from false to true, assuming that atom i is currently false. Obviously, the flip produces a gain for all unsatisfied clauses where atom i appears positively. It also produces a loss for all clauses where i appears negatively and is the only atom responsible for the satisfaction of the clause. $g10[i]$ represents the change in satisfied clauses when changing the value of atom i from true to false, assuming that atom i is currently true. It is computed in a way similar to $g01$. $gain[i]$ represents the change in satisfied clauses when changing the value of atom i . It is implemented using a conditional expression in terms of $g01[i]$, $g10[i]$, and the current value of atom i . No simulation is necessary in the resulting statement.

The GSAT statement can be made even more incremental. Since GSAT only selects the move with the best objective value, it is possible to maintain these candidate moves incrementally. The only change is to add the two invariants

$maxGain$: int = $\max(i \text{ in } 1..n) gain[i]$;
 $Candidates$: {int} =
 $\{i : \text{int} \mid \text{select } i \text{ from } 1..n \text{ where } gain[i] = maxGain \text{ and } gain[i] \geq 0\}$;

Here *maxGain* is simply the maximum of all gains and *Candidates* describes the set of flipping candidates as the set of atoms whose gain is positive and maximal. Once the invariants have been described, the neighborhood is defined by flipping one of the candidates. There is no need to use the keyword **best** or a **noDecrease** acceptance criteria, since they are already enforced by the invariants. The complete statement is depicted in Figure 3.8. Of course, the same transformation can be performed for the tabu search statement. As shown in the experimental section for GSAT in Chapter 7, the benefits of using the incremental statement are substantial.

```

    Solve
Data Type:
  clause = record
    p : {int};
    n : {int};
  end;
Constant:
  m: int = ...;
  n: int = ...;
  cl: array[1..m] of clause = ...;
  po: array[ i in 1..n] of {int} :=
    { c : int | select c from 1..m where i in cl[c].p };
  no: array[ i in 1..n] of {int} :=
    { c : int | select c from 1..m where i in cl[c].n };
Variable:
  a: array[1..n] of boolean;
Invariant:
  nbtl: array[i in 1..m] of int = sum(i in cl[i].p) a[j] + sum(j in cl[i].n) !a[j];
  g0l: array[i in 1..n] of int = sum(j in po[i]) (nbtl[j] = 0) - sum(j in no[i]) (nbtl[j] = 1);
  g10: array[i in 1..n] of int = sum(j in no[i]) (nbtl[j] = 0) - sum(j in po[i]) (nbtl[j] = 1);
  gain: array[i in 1..n] of int = if a[i] then g10[i] else g0l[i];
  maxGain : int = max(i in 1..n) gain[i];
  Candidates : {int} =
    { i : int | select i from 1..n where gain[i] = maxGain and gain[i] ≥ 0 };
  nbClauseSat : int = sum(i in 1..m) (nbtl[i] > 0);
Satisfiable:
  nbClauseSat = m;
Neighborhood:
  move a[i] := !a[i]
  where i from Candidates;
Start:
  forall(i in 1..n)
    random(a[i]);
Restart:
  forall(i in 1..n)
    random(a[i]);

```

Figure 3.8: An Incremental Statement for GSAT

Chapter 4

The Language

This chapter describes LOCALIZER in detail. The presentation is mostly organized along the textual order of the sections in a statement. The following grammatical conventions are used throughout the text. Keywords appear in boldface as in **symbol**, non terminals of the language are noted $\langle \text{symbol} \rangle$. Optional segments in rules are enclosed in brackets as in $[\langle \text{symbol} \rangle]$ and iterative fragments are enclosed in curly braces as in $\{ \langle \text{symbol} \rangle \}$. Grammatical symbols like pairs of brackets and curly braces are underlined to easily distinguish them from the same terminals in LOCALIZER. Valid identifiers in the language must start with a letter and are composed of any number of alphanumeric characters and the `'_'` symbol.

4.1 Data Types

We first review the primitive data types, type constructors, and then introduce user defined types.

4.1.1 Primitive Data Types

The basic data types are **int** (i.e., values in between $-2^{31} + 1$ and $2^{31} - 1$), **float** (i.e., double precision floating points) and **boolean** which contains the values **true** and **false**. In addition to the scalar types mentioned, LOCALIZER supports abstract data types for graph theoretic concepts. Those are discussed in Section 4.11.

4.1.2 Primitive Type Constructors

It is possible to use type constructors together with the scalar types of the system to build more complex data structures. The two elementary type constructors are the array and the set.

Arrays LOCALIZER supports multi-dimensional arrays of arbitrary types. The declarations

```
array[1..5] of int;
```

defines a one-dimensional array of integers. The declaration

```
array[1..5,1..5] of int;
```

declares a matrix of integers. The array dimensions are defined with sets of values. In the above example, the range 1..5 is interpreted as the set {1,2,3,4,5} and any value in this set denotes a distinct element in the array. Note that LOCALIZER does not support arrays of arrays but prefers multi-dimensional arrays instead. Also note that sets of records can be used to define dimensions.

Sets LOCALIZER supports sets of scalars and user defined types (records). For instance, the fragment

```
{int}
```

declares the type "set of integers". In general object declaration can use the grammar fragment of Figure A.1 to construct complex types.

4.2 The Type Section

The Type section of LOCALIZER is used to define record types. Records within LOCALIZER are identical to records found in conventional imperative languages like Pascal or C. They aggregate a number of named fields, possibly of different types.

The declaration

```
Edge = record
    o : int;
    d : int;
end;
```


declares a record type *Edge* consisting of two integers (the origin and destination nodes), while the excerpt

```

clause = record
    pl : {int};
    nl : {int};
end;

```

declares a record type *clause* with the two fields *pl* and *nl* of type “sets of integers”. User-defined types can be combined with the type constructor exactly like builtin types. For instance the fragment

```

Type:
    E = record
        o : int;
        d : int;
    end;
    T = record
        a : array[1..5] of int;
        b : array[1..5] of {int};
    end;

```

defines a record type *E* as a pair of integers and a record type *T* consisting of two fields *a* and *b* while the fragment

```

array[1..3] of E;
array[1..5] of T;
{T}

```

list declarations of an array of *E*, an array of *T* and a set of *T*. Figure A.2 depicts the syntax of the type section.

4.3 The Constant Section

The Constant section declares the input data and, possibly, some derived data which are useful in stating the problem. All constants are typed, *read only* (i.e., they cannot be modified by assignments), and initialized. There are various ways to initialize data in LOCALIZER. Initialization can be done inline (i.e., together with the declaration) or offline to separate the statement of the model from the instance data. The initialization can also be generic or computed. This section introduces the different form of constant initializations.

Each constant object is stated in LOCALIZER with a declaration statement that associates an identifier with a type, i.e., *Name : Type*. The actual nature of the initialization is dictated by the syntax that follows the declaration.

4.3.1 Inline Initializations

Constants can be initialized inline with a literal value. A literal can be an integer, a Boolean (true or false), a floating point, a vector of scalar, or a list of scalars. The excerpt

```

N: int      = 6;
S: {int}    = {1,2,3,4,5,6,7,8,9,10};
C: E        = <10,11>;
D: T        = <[11,12,13,14,15],[{1},{2},{3},{4},{5}]>;
F: {T}      = {<[11,12,13,14,15],[{1},{2},{3},{4},{5}]>,
               <[111,112,113,114,115],[{6},{7},{8},{9},{10}]>,
               <[116,117,118,119,120],[{1,2,8,7},{},{3,4,5},{},{}]>};
G: array[1..5] of E = [<1,2>,<2,3>,<3,4>,<4,5>,<5,6>];

```

gives several constant declarations that make use of the types described earlier.

4.3.2 Offline Initialization

Constants can also be initialized *offline* in the Init section to separate the model from the instance data, which is usually a good practice. The excerpt

```
f : float = ...;
```

declares a float f whose initialization is given in the Init section. The Init section consists of a set of pairs (identifier,value) and, of course, the type of the initialization must match the type of the declaration. Off-line initializations can be used for arbitrary types. For instance, the Boolean satisfiability statement may contain an initialization section of the form

```

Init:
  n = 6;
  m = 9;
  cl = [<{1,2,3},{}>,<{4,5,6},{}>,
        <{6},{1,2}>,<{1,3},{4}>,<{6},{2,3}>,
        <{2},{4,5}>,<{1},{4,6}>,<{4},{5,6}>];

```

Here, cl is initialized with a vector of 9 tuples. Each tuple is a pair of sets. The first set of each pair is associated with the first field of the record of type *clause* and denotes the set of positive literals in that clause. The second set is matched with the second field of the record and denotes the negative atoms of the clause. Note that a tuple is compatible with a record type if it has the same number of fields and the type of each entry in the tuple is compatible with the type of the corresponding field in the record. Additionally, this example illustrates the initialization of sets of integers (the set of positive and negative atoms) with a list of integers.

4.3.3 Generic Data

LOCALIZER also supports the concept of generic data that was introduced in NUMERICA [80]. The basic idea here is to initialize the data using an expression which may depend on parameters of the declaration. Genericity is especially attractive to define derived data which are then used to simplify the statement. In the case of the fully incremental version of GSAT (see Figure 3.8), it is important to know the clauses where an atom i appears positively (and negatively). This information is derived from the data cl using the generic declarations

Constant:

```
po: array[i in 1..n] of {int} := {c : int | select c from 1..m where i in cl[c].p};
no: array[i in 1..n] of {int} := {c : int | select c from 1..m where i in cl[c].n};
```

There are a couple of important points to stress here. First, the declarations use parameters that range over the index sets of the array. For instance, parameter i ranges over $1..n$, i.e., the set of atoms. Second, these parameters are used in the expression defined on the right-hand side of $:=$ to specify the value of the array at the given position. Third, an object can be used in the definition of some other constant only after it is defined. In a sense, LOCALIZER follows a convention similar to Pascal. In the GSAT example, $po[i]$ is defined as the set of clauses where atom i appears positively. The expressions allowed for the right-hand side are very general and their syntax is given in Figure A.3. Figure A.4 gives the syntax for the Constant section. Figure 4.1 completes the discussion with the signatures of the primitive functions, that have the obvious meanings.

Arithmetic	Set related	Output
<code>min(int,int)→int</code>	<code>size({T})→int</code>	<code>print(...)→void</code>
<code>max(int,int)→int</code>	<code>random({T})→T</code>	<code>println(...)→void</code>
<code>min(real,real)→real</code>	<code>minof({T})→T</code>	<code>time()→int</code>
<code>min(real,real)→real</code>		
<code>floor(float)→int</code>		
<code>ceil(float)→int</code>		
<code>round(float)→int</code>		
<code>exp(float)→float</code>		

Figure 4.1: Primitive Functions.

4.3.4 Computed Data

When the initialization of a constant cannot be expressed by a generic expression, LOCALIZER offers the option of initializing data using an `init` statement that contains imperative

code.

The excerpt

```

W : array[1..5] of int = [2,1,6,3,7];
P : array[1..5] of int = [4,2,3,5,6];
D : array[0..5, 0..20] of int init {
    forall(c in 0..20) D[0, c] := 0;
    forall(i in 1..5) D[i, 0] := 0;
    forall(i in 1..5) {
        forall(c in 0..20) {
            if c - W[i] >= 0
            then D[i, c] := max(D[i - 1, c - W[i]] + P[i], D[i - 1, c])
            else D[i, c] := D[i - 1, c]
            endif;
        }
    }
};

```

declares two arrays P (Profits) and W (Weights) and computes a constant tableau D storing the solution of the sub-problems related to the knapsack defined on P and W . Note that the computation of D is carried out in a dynamic programming fashion.

4.4 The Variable Section

Variables are fundamental in LOCALIZER since they define the state of the computation. It is important to fix the terminology. LOCALIZER supports two kinds of variables. The variables declared in the Variable section are the *state variables* of the statement, i.e., the variables that can be directly altered by a user. The Invariant section also declares variables. However, the *invariant variables* cannot appear on the left hand side of any assignment operator or any other destructive operation. The runtime system is responsible for updating them and preserving a consistent computation state. In the rest of this thesis, we feel free to use the term *variable* for both of them when the context does not require a distinction.

Variables are declared in the same way as constants, albeit without an initialization. They are generally given an initial value in the Start section and they are modified in the Neighborhood and Restart sections. As should be clear from the examples, LOCALIZER has an assignment operator, whose right-hand side is an expression. Variables can assume the following types `int`, `boolean`, `real`, `path`, `circuit`, `array`, `record`, and `graph`. Sets cannot appear directly, or indirectly, in a variable declaration. Figure A.5 gives the syntax for this section.

4.5 The Invariant Section

The Invariant section lists all the invariants defined in a statement. The invariants can be declared in any order and can reference each other. Note that circular references can only appear at the syntactic level. At the semantic level, LOCALIZER requires that, at all time any variable does not depend on itself, directly or indirectly. If this condition is violated, the runtime system produces an execution error. This restrictions makes it possible to maintain the invariants efficiently.

4.5.1 Arithmetic Invariants

Simple Expressions The simplest arithmetic invariant maintains a direct functional dependence between the variables that appear in the expression and the variable it defines. For instance, assume that x, y and z are integer variables, than the invariant

$$a : \text{int} := x * y + z / 2;$$

defines a as an integer variable and constraints a to have the value denoted by the expression on the left hand side of the assignment. Arithmetic expressions can use all the conventional arithmetic operators, with the addition of the aggregates $\max, \min, \text{argmax}, \text{argmin}, \text{sum}$ and prod . These aggregates have a natural semantics

$$\begin{aligned} m : N &:= \min(i \text{ in } S) e(i) &\Leftrightarrow & m = \min_{i \in S} e(i) \\ M : N &:= \max(i \text{ in } S) e(i) &\Leftrightarrow & M = \max_{i \in S} e(i) \\ a : T &:= \text{argmin}(i \text{ in } S) e(i) &\Leftrightarrow & a \in \{i \in S \mid e(i) = \min_{j \in S} e(j)\} \\ A : T &:= \text{argmax}(i \text{ in } S) e(i) &\Leftrightarrow & a \in \{i \in S \mid e(i) = \max_{j \in S} e(j)\} \\ s : N &:= \text{sum}(i \text{ in } S) e(i) &\Leftrightarrow & s = \sum_{i \in S} e(i) \\ p : N &:= \text{prod}(i \text{ in } S) e(i) &\Leftrightarrow & s = \prod_{i \in S} e(i) \end{aligned}$$

where N is a numeric type, i.e. either int or float and T is the type of the element in S , i.e., S is of type $\{T\}$. Notice that the definition of argmin and argmax only specifies that the return value should belong to the set of minimal (respectively maximal) elements. In case of ties, ($|\{i \in S \mid e(i) = \min_{j \in S} e(j)\}| > 1$) the implementation randomly picks an element from the set with a probability of $1/|\{i \in S \mid e(i) = \min_{j \in S} e(j)\}|$. Note that $e(i)$ is a parametric expression that is of type N .

LOCALIZER enforces the usual conventions for arithmetic expressions. All operators are associative to the left, except the exponentiation operator. The precedence of the operators are given in table 4.1. Note that the expressions

$$\begin{aligned} a : \text{int} &:= \text{sum}(i \text{ in } 1..n) \ x[i] + y[i]; \\ b : \text{int} &:= \text{sum}(i \text{ in } 1..n) \ x[i] \times y[i]; \end{aligned}$$

must be interpreted as

$$\begin{aligned} a : \text{int} &:= (\text{sum}(i \text{ in } 1..n) \ x[i]) + y[i]; \\ b : \text{int} &:= (\text{sum}(i \text{ in } 1..n) \ x[i] \times y[i]); \end{aligned}$$

The invariant a is incorrect since the i in $y[i]$ is out of scope, hence undefined. Since the precedence of the product operator is higher, the definition of b is correct.

Class	Operator	Precedence
Logical	or	0
Logical	and	1
Relational	$=, \neq, \leq, \geq, <, >, \text{in}$	2
Binary	$-, +, \text{union}$	3
Aggregate	sum , argmin, argmax , min , max, union	4
Binary	$\times, /, \%$	5
Aggregate	prod	6
Binary	$^$	7
Unary	not , $-$, $+$, $++$, $--$ (prefixed and suffixed)	8
Binary	$.., [], ()$	9

Table 4.1: Precedence of operators.

Array Expressions Often, many invariant expressions assume the same generic form and the difference can be captured in terms of an assignment to parameters. For instance, consider the example of the *nbtI* invariant from the GSAT statement. The GSAT algorithm maintains for each clause i in $\{1..m\}$ of a boolean formula the number of true literals. Given that the state is represented with an array x of booleans, the natural definition is

$$nbtI : \text{array}[i \text{ in } 1..m] \text{ of int} := \text{sum}(j \text{ in } F[i].p) \ x[j] + \text{sum}(j \text{ in } F[i].n) \ \text{not } x[j];$$

where F is the constant array of clauses. Each entry of the *nbtI* array is mapped to the left hand side expression that is partially evaluated with respect to the corresponding i . The expressions also use the Boolean operator not that returns the negation of a Boolean expression.

Referencing Arrays One of the interesting feature of LOCALIZER is the ability to index arrays with variables. Many models, including the job-shop scheduling model presented later on in the thesis, make use of this facility. For instance, the invariant

$$a : \text{int} := b[x + 1] * y;$$

maintains the value of a , when b or y changes but also when variable x changes. The mechanism actually generalizes to arbitrary expressions as shown in

$$a : \text{int} := b[c[2 * x] + 1] * y;$$

where both b and c are arrays of variables.

Relations Boolean relations are part of the expressions supported by LOCALIZER. The relational operators $\{<, >, \leq, \geq, \neq, =, \in\}$ can be used and their result is typed as a Boolean. Naturally, Boolean invariants can be defined with Boolean expressions. In addition, LOCALIZER allows meta expressions. A meta expression uses the result of a relation expression and interprets that value as the integer 1 in case the relation holds and 0 otherwise. This gives the opportunity to define invariants that count the number of Boolean relations that are satisfied. For instance, the invariant

$$nb : \text{int} : \text{sum}(i \text{ in } 1..n) (x[i]\%2 = 0);$$

where x is an array of integers, defines nb as the number of even integers in the array x .

Recurrence Relations It is interesting to realize that LOCALIZER supports the definition of recurrence relation with invariants. Several models in the application chapter make use of this feature. To illustrate what is possible, consider the excerpt

$$Fact : \text{array}[i \text{ in } 0..n] \text{ of int} := \text{if } i = 0 \text{ then } 1 \text{ else } i * Fact[i - 1];$$

It defines $Fact$ as an array of integers that stores the values of $i!$ for all i in $\{0..n\}$. Note that the expressions defining $Fact$ makes use of $Fact$ itself. In this specific case, $Fact[0]$ is the basic case but in fact the recurrence can be arbitrary as illustrated by the examples

$$\begin{aligned} F : \text{array}[i \text{ in } 0..6] \text{ of int} &:= \text{if } i = 6 \text{ then } 720 \text{ else } F[i + 1]/(i + 1); \\ G : \text{array}[i \text{ in } 0..N] \text{ of int} &:= \text{if } i = 3 \text{ then } 6 \\ &\quad \text{else if } i < 3 \text{ then } G[i + 1]/(i + 1) \\ &\quad \text{else } G[i - 1] * i; \end{aligned}$$

that both define factorial. It is important to keep in mind that the expressions above remain invariants. If, for instance, the base case 0 was replaced by an expression depending upon another variable x , any change to x would trigger a re-computation of the entries of the *Fact* array affected by this change.

4.5.2 Set Invariants

Maintaining complex sets is ubiquitous in many local search algorithms and LOCALIZER provides a rich collection of set invariants.

Extensional sets The first category is composed of extensional sets that can be defined in two different ways. The simplest form is as a list of expressions. The invariant

$$s : \{\text{int}\} := \{ a + b, a * b, 2, b \};$$

defines s as a set of integers that contains the values of the expressions appearing on the left hand side of the definition. LOCALIZER does not maintain multi-sets. Assuming that, in some computation state, $a = 2$ and $b = 1$, s is equal to $\{3, 2, 1\}$. If the variable b changes from 1 to 3, s becomes $\{5, 6, 2, 3\}$.

Extensional sets can also use generic mechanisms. When all the expressions appearing in the list are similar, i.e., only differ in the assignment of values to some parameters, LOCALIZER offers a compact definition of the form

$$\begin{aligned} x : \{T\} &:= \{ t : T && | \text{select } t \text{ from } S \text{ where } P(t) \} \\ x : \{T\} &:= \{ \langle v_1, \dots, v_n \rangle : T && | \text{select } v_1 \text{ from } S_1 \text{ where } P_1(v_1) \\ &&& \vdots \\ &&& \text{select } v_i \text{ from } S_i \text{ where } P_i(v_1, \dots, v_i) \\ &&& \vdots \\ &&& \text{select } v_n \text{ from } S_n \text{ where } P_n(v_1, \dots, v_n) \} \end{aligned}$$

where the sets S, S_1, \dots, S_n are all defined as constants or invariants and the Boolean expressions $P(t), P_1(v_1), \dots, P_n(v_1, \dots, v_n)$ can refer to variables.

The excerpt

$$s : \{\text{int}\} := \{ i : \text{int} \mid \text{select } i \text{ from } 1..n \text{ where } a[i]\%2 = 1 \};$$

defines s as a subset of $\{1..n\}$ that satisfies the condition $a[i]\%2 = 1$. Note that a could be a constant or a variable. Several select statements can be chained to join sets and each select can have its own where filter as in

$$s : \{\text{int}\} := \{ i : \text{int} \mid \text{select } i \text{ from } 1..N \text{ where } i\%2 = 1 \\ \text{select } j \text{ from } 1..N \text{ where } x[i] = j\};$$

where x is once again an array of variables. Nested select statements are also used to build sets of tuples as in

$$\begin{aligned} s1 : \{\text{int}\} &:= \{a, b, c\}; \\ s2 : \{\text{int}\} &:= \{d, e, c\}; \\ s3 : \{\text{Pair}\} &:= \{(a, b) : \text{Pair} \mid \text{select } a \text{ from } s1 \\ &\quad \text{select } b \text{ from } s2\}; \\ s4 : \{\text{Pair}\} &:= \{(a, b) : \text{Pair} \mid \text{select } a \text{ from } s1 \\ &\quad \text{select } b \text{ from } s2 \text{ where } (a + b)\%2 = 0\}; \end{aligned}$$

where a, b, c, d, e are all variables. $s3$ maintains the cross product of $s1$ and $s2$ while $s4$ maintains the subset of $s3$ satisfying the condition $(a + b)\%2 = 0$.

4.5.3 Set Dependent Invariants

Sets can serve many purposes in a statement. It is possible to query the size of a set, check whether it contains a specific element, or even to aggregate some other data. Consider the statement

$$\begin{aligned} s : \{\text{int}\} &:= \{ i : \text{int} \mid \text{select } i \text{ from } Z \text{ where } x[i]\%2 = 0\}; \\ t : \text{int} &:= \text{sum}(i \text{ in } s) \ y[i]; \end{aligned}$$

where x and y are two arrays of variables (or constants) and Z is a set (variable or constant too). Given that the definition of s relies on the array x , a value change for any element of x can modify s . However, it does not impede the definition of t as the sum of the elements of y referred to by s . Indeed, a modification to an entry of y that t depends on forces t to be updated. Similarly, when elements appear or disappear from s , t is updated to reflect the change.

4.5.4 Builtin Invariants

In addition to the standard expressions, invariants can also be defined in terms of **distribute** and **dcount**. They take as input a one-dimensional array of integer variables A and two sets I and O . I must be a subset of the index set of A . The result type of a **distribute** expression is a one-dimensional array B , of variables of type set of integer, whose index set is O . i.e., $B : \text{array}[O] \text{ of } \{\text{int}\}$. The result type of a **dcount** expression is one-dimensional array of integer variables whose index set is O . Their meanings are given by the following

equivalences:

$$\begin{aligned} B = \text{distribute}(A, I, O) &\Leftrightarrow B[i] = \{j \in I \mid A[j] = i\} \forall i \in O \\ B = \text{dcount}(A, I, O) &\Leftrightarrow B[i] = \text{size}(\{j \in I \mid A[j] = i\}) \forall i \in O \end{aligned}$$

For instance, the excerpt

Variable:
truck : array[1..*nbC*] of int;
Invariant:
trip : array[0..*nbC* + 1] of int := **distribute**(*truck*, {1..*nbC*}, {0..*nbC* + 1});

from the VRP statement defines *truck* as an array of variables of type integer that associate with each client identifier in 1..*nbC* the number of the truck in charge of its delivery. The invariant *trip* defines an array (from 0 to *nbC* + 1) of sets of integers. An entry *trip_k* stores the set of client identifiers assigned to truck *k*.

These expressions were introduced because of their ubiquity in practical applications.

4.5.5 Invariant Declaration Syntax

The left-hand side of any invariant obeys the syntax of expressions give in Figure A.3. The invariant section itself obeys the syntax shown in Figure A.6.

4.5.6 Current Limitation

The current implementation imposes one restriction on set definitions. The fragment

$$\begin{aligned} A &: \text{array}[i \text{ in } 1..n] \text{ of } \{\text{int}\} := \{ i : \text{int} \mid \text{select } \dots \}; \\ Z &: \{\text{int}\} := A[b]; \end{aligned}$$

where *A* is an array of set variables and *b* is a an integer variable is invalid. LOCALIZER currently does not allow an array of sets to be indexed by a variable. Note that the restriction holds even if the set is encapsulated within a record. Generally speaking an expression pattern *A*[*E*]*B* is invalid if the pattern *A*[*E*]*B* is supposed to denote a set and *E* refers to a variable.

4.6 The Operator Section

The Operator section is devoted to the definition of traditional functions that can be used in several other sections such as the Neighborhood, Start and Restart sections. The body of these operators are constructed from statements whose syntax is depicted in Figure A.7.

4.6.1 Functions

Functions in LOCALIZER are essentially similar to functions in C and use traditional assignment, conditional, and iterative statements, as well as recursion. Functions can take any number of arguments and can have a return type. Each argument is typed and the return type is restricted to scalars plus void to signal the absence of a return value. All scalar parameters are passed by value, others are passed by reference. Scalar variables are also passed by value. Complex object variables (e.g., an array of int defined in the invariant section) cannot be passed as argument. This last point is not a restriction in itself, since variables are globally accessible.

A simple recursive procedure like factorial can be implemented as

```
int factorial(i : int) {
    if i=0
        then return 1
        else return i * factorial(i - 1)
    endif;
}
```

A function can also declare local variables as in traditional languages. Their scope is the function definition. Once declared, a local variable can be assigned to, and passed as argument to other functions. It is possible to declare local variables of complex type like arrays or sets or any combination thereof. LOCALIZER also accepts an initialization together with the declaration. For instance the statements

```
a : array[i in 1..N] of int := i + 1;
b : array[1..N];
c : int := sum(i in 1..N) a[i];
d : {int} := { i : int | select i from 1..N where x[i]%2 = 0 };
```

are all valid local declarations. It is important however to remember that none of $\{a, b, c, d\}$ are invariants. If an $x[i]$ changes, the change will not affect the content of d . Local variables behave like conventional variables in imperative languages.

The entire state, including constants and variables, are accessible to functions. However, only state variables and locals can be modified by assignments. Figure A.8 gives the syntax for function declarations.

4.6.2 Control Structures

In addition to the standard control structure, LOCALIZER provides constructs which are useful for local search algorithms. We start by briefly reviewing the standard constructs and then move to the special purpose control structures.

Standard control All traditional control structures are available in the language. The iterative statement **while-do**, and the branching statement **if-then-else** are sufficient to construct any imperative procedure. The **return** instruction is used for operators that have a return value. The expression syntax available for imperative code is identical to the expression syntax in effect in the **Constant** section and the **Invariant** section.

Additionally, pre and post increment/decrement operators are available. The pre-increment and the pre-decrement operators first modify the operand and return the result while the post-operators first use the value of the operand and carry out the operation once the expression is evaluated.

The Forall Instruction The **forall** instruction provides a convenient way of iterating over the elements of a set. The instruction automatically declares an argument with a scope restricted to the **forall** instruction itself. The type of the argument is the type of the elements in the set. If the set is a simple range of integers, successive iteration produce the values in ascending order. If the set is more complex, the order in which values are produced is implementation dependent.

The Choose Instruction The **choose** instruction can be used to select an element from a set in a nondeterministic way. It is possible to filter elements of the given set or to select an element optimizing a function. The following examples illustrate the various forms of **choose** statements:

```
choose i from ⟨Set Expr⟩;
choose i from ⟨Set Expr⟩ minimizing ⟨arexr⟩;
choose i from ⟨Set Expr⟩ maximizing ⟨arexr⟩;
choose i from ⟨Set Expr⟩ such that ⟨arexr⟩;
```

The first form non-deterministically picks an element from the set. The second and third version filters the set and non-deterministically pick an element from the subset of minimal/maximal ones. The last form selects an element from the subset that satisfies the condition.

4.7 Neighborhood

The neighborhood is specified in **LOCALIZER** with a move instruction whose syntax is given in Figure A.9. The move instruction uses both declarative and imperative statements. The move instruction assumes multiple roles. First, the instruction is responsible for specifying the procedure that transforms the computation state to produce a neighboring state.

Second, it is responsible for specifying the set of computation states that constitute the neighborhood. Finally, it is responsible for specifying which neighbors in $N(\tau)$ are deemed acceptable.

Consequently, the move instruction contains three parts. The first part specifies the state transformation procedure, the second specifies the neighborhood itself while the third part defines the acceptance criterion. Each component is now reviewed in detail.

4.7.1 The Transformation Component

The transformation is encoded with a fragment of procedural code. This fragment uses a number of parameters that specify which neighbor is to be produced. The parameters are discussed in the next section. The code fragment can use any imperative code, including calls to user defined operators. As usual, the whole computation state is directly accessible and the modification ought to alter the state variables only. The code fragment must appear directly after the move keyword.

4.7.2 The Neighborhood Component

Remember that the skeleton of a move instruction is:(the detailed syntax can be found in Figure A.9)

```

move  $op(x_1, \dots, x_n)$ 
where
     $x_1$  from  $S_1$ ;
    ...
     $x_n$  from  $S_n$ 
[ accept when  $\langle \text{AcceptanceCriterion} \rangle$  ]

```

Assuming that the transformation code is abstracted as the procedure op , and that $Post(\tau, i)$ represents the computation state obtained by executing i in τ , the neighborhood of the computation state τ is defined as

$$N(\tau) = \{Post(\tau, op(x_1, \dots, x_n)) \mid x_1 \in S_1 \ \& \ \dots \ \& \ x_n \in S_n\}.$$

4.7.2.1 Parameter Specification

The purpose of where clause is to construct the set of tuples that parameterize the operation op . Intuitively, the clause extracts tuples $\langle x_1, \dots, x_n \rangle$ from the cross product $S_1 \times \dots \times S_n$. In its simplest form, each statement in the clause picks a value in a set S_i and assigns it to

the corresponding x_i . Because one can easily conceive neighborhoods where some of the sets S_i reduces to singleton, LOCALIZER also authorizes statements of the form $x_i = e_i$ where e_i is an expression.

Each set S_i can denote a constant set or an invariant set. It is possible to define S_i inline as shown in the hypothetical neighborhood definition for graph coloring

```

move  $x[i] := j$ 
where  $i$  from  $\{1..N\}$  minimizing  $\text{size}(C[x[i]])$ 
       $j$  from  $NE$  such that  $x[i] \neq j$ 
accept ...

```

where a vertex is picked from the smallest color class while the target color class j , chosen from the set NE , is constrained to be different from the color class of i .

So far, the move instruction contributed to a static description of the neighborhood, or, more precisely, of the search space of the algorithm. Its role is also to specify the search procedure that explores this space.

4.7.2.2 Search Rule

Search algorithms can choose between different strategies to explore $N(\tau)$. Some algorithm explore $N(\tau)$ exhaustively looking for the best transition while others settle for the first improving transition encountered.

Any Mode With the addition of the keyword **any** in front of the move instruction, LOCALIZER is instructed to use a random walk exploration strategy. As a consequence, LOCALIZER select an element from $N(\tau)$ at random. More precisely, the state τ' is selected with a probability $1/|N(\tau)|$.

Best Mode With the addition of the keyword **best** in front of the move instruction, LOCALIZER takes advantage of the objective function to compute $B(\tau) \subseteq N(\tau)$ defined as follows

$$B(\tau) = \{\tau' \in N(\tau) \mid f(\tau') = \max_{s \in N(\tau)} f(s)\}.$$

LOCALIZER is then bound to select, at random, a state τ' from $B(\tau)$. The state τ' then constitutes the output of the move.

First Mode With the addition of the keyword **first** in front of the move instruction, LOCALIZER is instructed to explore $N(\tau)$ by producing its elements one by one until an element τ' satisfies the acceptance criterion. It is important to realize that all the strategies referred to above rely on the simulation mechanism of LOCALIZER since they must produce elements of $N(\tau)$. If the statement does not rely on simulation but is incremental instead, no modality keyword is required and LOCALIZER defaults to a randomized exploration strategy.

4.7.2.3 Differentiation and Best Heuristic

In certain circumstances, it might be appropriate to use a best search strategy even though it is difficult to come up with an invariant based specification of the optimal move. However, the cost of simulation can be prohibitive, especially when compared to the cost of a differential approach. In order to avoid simulation and still iterate through all of $N(\tau)$, LOCALIZER provides two additional statements minimizing and maximizing that can be used in the where component. Generally speaking, LOCALIZER allows move instructions of the form

```

move op( $x_1, \dots, x_n$ )
where
   $x_1$  from  $S_1$ ;
  ...
   $x_{i_1}$  from  $S_{i_1}$ ;
  optimizing  $f_1(x_1, \dots, x_{i_1})$ ;
   $x_{i_1+1}$  from  $S_{i_1+1}$ ;
  ...
   $x_{i_j}$  from  $S_{i_j}$ ;
  optimizing  $f_j(x_1, \dots, x_{i_j})$ 
[accept when ...]

```

where **optimizing** stands for **minimizing** or **maximizing** and the **optimizing** statement can appear more than once. In this case, LOCALIZER produces the sets

$$\begin{aligned}
 N_1(\tau) &= \{\langle x_1, \dots, x_{i_1} \rangle \mid f_1(x_1, \dots, x_{i_1}) = \max_{e \in S_1 \times \dots \times S_{i_1}} f_1(e)\} \\
 N_2(\tau) &= \{\langle x_1, \dots, x_{i_2} \rangle \mid f_2(x_1, \dots, x_{i_2}) = \max_{e \in N_1(\tau) \times S_{i_1+1} \times \dots \times S_{i_2}} f_2(e)\} \\
 &\vdots \\
 N_j(\tau) &= \{\langle x_1, \dots, x_{i_j} \rangle \mid f_j(x_1, \dots, x_{i_j}) = \max_{e \in N_{j-1}(\tau) \times S_{i_{j-1}+1} \times \dots \times S_{i_j}} f_j(e)\}
 \end{aligned}$$

and defines $N(\tau)$ as $N_j(\tau)$. $N(\tau)$ can thereafter be explored with the technique discussed earlier. This facility encodes the selection of the neighbors that are optimal with respect to the functions f_1 through f_j without the help of simulation or invariants. The example

```

move  $x[i] := !x[i]$ 
where  $i$  from  $\{1..N\}$ ;
      maximizing  $gain[i]$ 
accept in current state ...

```

illustrates the idea on the Boolean satisfiability neighborhood of GSAT. The move instruction does not use the keyword **best** and does not rely on simulation. Yet, the maximizing instruction is going to pick an i at random from the subset $N_1(\tau)$ of $\{1..N\}$ that maximizes the gain value $gain[i]$. The downside of this formulation is that this optimal subset is not maintained incrementally and must be recomputed from scratch at each transition.

4.7.3 The Acceptance Criterion

The acceptance criterion is instrumental in the definition of search strategies. Variations of local improvement and simulated annealing algorithms rely on this mechanism. The keywords **improvement**, **noDecrease** and **delta** offer the necessary support. The acceptance criterion is used as follows; Once an element of the neighborhood has been selected, LOCALIZER must determine if it is an appropriate move. The acceptance criterion lists Boolean conditions that must be tried in sequence. As soon as the state satisfies one of the conditions, it is accepted and the state transformation is performed. The criterion is build according to the syntax in Figure A.10.

Evaluation state Acceptance criteria are, by default, evaluated in the new state. This evaluation requires the generation of the state (or, at least, part of it). This may turn too expensive, since all the invariants may need to be propagated. Therefore, it is also possible to specify that the acceptance criterion should be evaluated in the current state by using the keywords **in current state**. Using the current state to evaluate moves may produce significant improvements in efficiency. The addition of the **in current state** keywords renders the keywords **improvement**, **noDecrease** and **delta** meaningless. The responsibility to evaluate the relative quality of the move is placed on the modeler. Nevertheless, the acceptance criterion remains useful to state the conditions for acceptance.

Multiple conditions It is possible to define several distinct conditions that lead to the acceptance of a move. Each condition can be associated with an acceptance statement. Statements based on simulated annealing for instance, must be able to react differently depending on the reason for accepting the move. A simulated annealing procedure often uses statistics about the type of moves recently accepted to implement the termination criterion.

A typical choice is to compute the frequencies of good moves, plateau moves or uphill moves. If the frequency of good moves falls below a fixed threshold, the algorithm predicts that it is not likely to make progress and stops the computation. Whenever distinctions are superfluous, a standard disjunction between all the acceptance conditions is appropriate. The **cor** construction can be used to discriminate between the conditions leading to acceptance. A statement can be associated with each **cor** disjunct. The excerpt

```

move     $x[i] := !x[i];$ 
where    $i$  from  $\{1..n\};$ 
           $myD = IT[i] - EX[i] + (\text{if } x[i] \text{ then } -4 * IMB + 4 \text{ else } 4 * IMB + 4) * \alpha;$ 
accept when in current state
           $myD < 0.0 \rightarrow \{$ 
             $\text{if } IMB = 0 \text{ and } OBJ < \text{bound}$ 
               $\text{then } fc := 0;$ 
             $\text{endif};$ 
             $ch := ch + 1;$ 
           $\}$ 
          cor  $myD = 0.0$ 
          cor  $\text{Pr}(\exp(-myD/t)) : \text{always} \rightarrow ch := ch + 1;$ 

```

from the simulated annealing statement for graph partitioning illustrates this point.

4.7.4 The try Composition

Probabilistic composition The try combinator is used to combine several neighborhoods. The simplest example is probably the random noise strategy introduced in [66] for GSAT. The idea is to take either the best possible improving move or any move, regardless of its impact on the objective function. The two neighborhoods are blended in a probabilistic strategy where each type of move is selected according to a stream of events generated by a random variable that obeys a specific probability distribution. This strategy can be expressed as

```

try
  Pr(0.1):
    move
      a[i] := !a[i]
    where
      i from OccurInUnsatClause
    accept when always ...;
  default:
    best move
      a[i] := !a[i]
    where
      i from {1..n}
    accept when noDecrease;
end

```

Conditional composition A second use for the try combinator arise from the necessity to enable and disable certain neighborhoods when circumstances dictates it. It is convenient to use invariants to keep track of the property at all time and enable the specific neighborhood only when the Boolean invariant holds. Imagine, for instance, a problem statement that defines two sets A and B as invariants. When the set A is not empty, it is possible to setup a neighborhood based on A . When the set is empty, it is preferable to use a second neighborhood based on B . If B is empty too, a default move might be specified as last resort. Such a neighborhood might look like

```

try
  when size(A) > 0 : move ...
  when size(B) > 0 : move ...
  default:          move ...
end

```

Sequential composition A third application permits the chaining of neighborhoods. It is easy to imagine a problem where several different neighborhoods of increasing complexity and cost can apply. It is possible to try them in sequence, until an improving move is found or all else fail. Consider for instance, the traveling salesman problem. The try combinator can be used to blend the 2-opt, Or-opt, and 3-opt neighborhoods. Naturally, 2-opt should be listed first. If 2-Opt cannot find an improvement, Or-opt or even 3-opt can then be used.

Figure 4.2 illustrates a try instruction that, for brevity, combines only two neighborhoods: a 2-opt and a 3-opt with reversal. Note that the LOCALIZER statement for this TSP model uses a circuit to represent the tour.

```

try
  best move {
    p.noarc(a,b);p.noarc(d,c);
    p.reverse(b,d);
    p.arc(b,c);p.arc(a,d);
  } where
    a from {1..n};
    b = p.succ(a);
    c from OV[b] such that (c <> a and p.pred(c) <> b) smaller than Dist[a,b];
    d = p.pred(c);
    perf = Dist[a,d] + Dist[b,c] - Dist[a,b] - Dist[c,d];
    minimizing perf;
  accept when in current state perf < 0
  best move {
    p.noarc(a,b);p.noarc(c,d);p.noarc(e,f);
    p.reverse(b,c);p.reverse(d,e);
    p.arc(b,e);p.arc(a,c);p.arc(d,f);
  } where
    a from {1..n};
    b = p.succ(a);
    c from OV[a] such that (c <> p.pred(p.pred(a)) and
                           c <> p.pred(a) and c <> b)
                           smaller than Dist[a,b];
    d = p.succ(c);
    ss2 = p.subSequence(d,p.pred(a));
    e from OV[b] such that (e in ss2);
    f = p.succ(e);
    perf = Dist[a,c] + Dist[b,e] + Dist[d,f] - Dist[a,b] - Dist[c,d] - Dist[e,f];
    minimizing perf;
  accept when in current state perf < 0
end

```

Figure 4.2: A try Combination for the Traveling Salesman Problem.

4.8 The Objective Function Section

The objective function is stated in a separate section and is used to assess the “quality” of a given state. Note that the objective function is optional and is only used when the acceptance criteria are evaluated in the resulting state.

4.9 Termination Criteria

Termination is handled via several sections and depends on the nature of the problem. Each model starts with a keyword that is either solve or optimize to specify either a decision or an optimization problem.

The Satisfiable Section The (optional) Satisfiable section is used to specify whether a state is a solution. In decision problems, LOCALIZER terminates whenever this is the case. In optimization problems, LOCALIZER updates the best solution whenever the state is a solution that improves the best bound found so far. When the section is omitted, LOCALIZER assumes that every state is a solution.

The Local and Global Conditions These optional sections, specify Boolean predicates that are used to control the innermost and outermost loops of the computation model. For instance, a simulated annealing model can use the local condition to implement a *cutOff* strategy that terminates the innermost loop and drops the temperature whenever the evaluation function has been updated sufficiently many times. The excerpt

local condition: $ch < NC$;
global condition: $fc < MFC$;

comes from graph partitioning and collaborates with the neighborhood to terminate the algorithm.

The variable *ch* keeps track of the number of improving or degrading changes. When its value exceeds the threshold *NC*, the innermost loop of the computation model terminates and the Restart instruction is executed (in this case, the restart lowers the temperature). The freeze count variable *fc* keeps track of the number of times the temperature has been lowered without resulting in an improvement. If it reaches the threshold *MFC*, the outer loop of the computation model terminates.

4.10 The Parameter Section

This section, also optional, is used to override the default values of some parameters of the system. In the current implementation, two parameters can be modified. The variables *maxSearches* and *maxTrials* that appear in the computation model can be assigned the result of any arithmetic expression. The excerpt

parameter: $maxTrials := sf * n$;

defines *maxTrials* as the product of two constants *sf* and *n* initialized in the Init section of the model.

4.11 Advanced Support

LOCALIZER also offers several additional abstract data types. The purpose of the new types is to offer support for ubiquitous objects in modeling. The support is graph-oriented and provides concepts such as arcs, paths, circuits and graphs. The new types are not strictly necessary, since they can be implemented in terms of the standard structures offered in LOCALIZER. However, their use significantly clarifies the statements, automates classic invariants on such structures, allow users to work at a higher level of abstraction and offer the opportunity to support special purpose incremental algorithms.

4.11.1 Abstract Builtin Data Types

Several new basic data types are introduced to represent and manipulate graph-related structures.

Arc The Arc type is used to encapsulate the notion of directed link between two nodes, represented as integers. An arc has a source and a destination. From the user perspective, it is similar to a record. Two accessors are provided to gain access to the source s and the destination t . The excerpt

```
constant:
  a : {Arc} = {<1,4>,<2,3>,<3,4>,<2,1>};
  b : {Arc} := { p : Arc | select p from a where a.s <= 2};
```

declares a as a constant set of arcs. Each arc is initialized with a tuple that must be compatible with the arc type, i.e., the tuple must be a pair of integers. The first element of the pair is the source of the arc while the second is the target. The constant b is a generic set of arcs defined as the subset of a where the source of the arc is less than or equal to 2.

Path The Path abstract data type is a directed sequence of nodes. It is defined on a set of integers that represents the nodes. At any time, the path uses a subset of the node set. Path cannot contain cycles and each node appearing in the path must have a degree one or two. The nodes with degree one are the endpoints of the path while the degree two nodes are internal to the path. More formally, a path is sequence of n nodes

$$(a_0, a_1, \dots, a_n)$$

drawn from a set N with $|N| \geq n$, that can also be interpreted as the set of arcs

$$\{\langle a_0, a_1 \rangle, \langle a_1, a_2 \rangle, \langle a_2, a_3 \rangle, \dots, \langle a_{n-1}, a_n \rangle\}.$$

Path is a data type that extends the type system of LOCALIZER. As a consequence, a path declaration uses the familiar syntax. The declaration of a Path requires an additional parameter of type `{int}` denoting N , the set of integers the path is defined upon. Note that the node set the path refers to is a constant range of integers.

Path can be used for many different purposes. A typical application is to encode an ordering, thereby indicating precedence relationships among the nodes. In the current implementation, path can be defined in the constant and in the variable section and can be the object of builtin invariants. The excerpt

<pre> constant: <i>SJ</i> : array[1..<i>nbJobs</i>] of Path(<i>N</i>) = ...; variable: <i>SM</i> : array[1..<i>nbMach</i>] of Path(<i>N</i>); </pre>

from a job-shop scheduling application illustrates the use of paths to encode the precedence relationship imposed on the various tasks. An array of constant path is used for the static job precedences, while an array of variables is used for the machines ordering. The initialization of the path *SJ* is postponed to the Init section. Since the ordering information is significant, a path is initialized with a sequence of values in between parenthesis as in

<pre> init: <i>SJ</i> = [(0,1,2,3,4,5,51), ... (0,46,47,48,49,50,51)]; </pre>

Table 4.2 provides the interface for the Path abstract data type. Both Accessors and modifiers are provided to operates on the arc set.

Circuit The Circuit abstract data type is a directed path that can be cyclic. More formally, a circuit is an infinite sequence of nodes

$$(a_0, a_1, \dots, a_n, a_0, a_1 \dots)$$

drawn from a set N with $|N| \geq n$, that can also be interpreted as the set of arcs

$$\{\langle a_0, a_1 \rangle, \langle a_1, a_2 \rangle, \langle a_2, a_3 \rangle, \dots, \langle a_{n-1}, a_n \rangle, \langle a_n, a_0 \rangle\}.$$

Operation	Type	Description
<code>arc(<i>i</i>, <i>j</i>)</code>	$int \times int$	Creates an arc from node <i>i</i> to node <i>j</i> .
<code>noarc(<i>i</i>, <i>j</i>)</code>	$int \times int$	Removes the arc $i \rightarrow j$ from the path.
<code>swapp(<i>i</i>)</code>	int	Swaps node <i>i</i> with its predecessor along the path. If <i>p(i)</i> and <i>s(i)</i> denote, respectively the predecessor and the successor of node <i>i</i> , this is equivalent to removing the arcs $\langle p(i), i \rangle$, $\langle i, s(i) \rangle$, $\langle p(p(i)), p(i) \rangle$ and adding $\langle p(p(i)), i \rangle$, $\langle i, p(i) \rangle$, $\langle p(i), s(i) \rangle$
<code>swapn(<i>i</i>)</code>	int	Swaps node <i>i</i> with its successor along the path. If <i>p(i)</i> and <i>s(i)</i> denote, respectively the predecessor and the successor of node <i>i</i> , this is equivalent to removing the arcs $\langle p(i), i \rangle$, $\langle i, s(i) \rangle$, $\langle s(i), s(s(i)) \rangle$ and adding $\langle p(i), s(i) \rangle$, $\langle s(i), i \rangle$, $\langle i, s(s(i)) \rangle$
<code>moveBefore(<i>i</i>, <i>j</i>)</code>	$int \times int$	Moves vertex <i>i</i> before vertex <i>j</i> along the path.
<code>toFront(<i>i</i>)</code>	int	Moves vertex <i>i</i> in the first position in the path
<code>toBack(<i>i</i>)</code>	int	Moves vertex <i>i</i> in last position in the path
<code>revall()</code>		Reverses the order of the nodes in the path. This corresponds to changing the path direction.
<code>reverse(<i>i</i>, <i>j</i>)</code>	$int \times int$	Reverses a section of the path starting at node <i>i</i> and ending at node <i>j</i> . The rest of the path is left untouched.
<code>ord(<i>i</i>)</code>	int	Returns the rank of node <i>i</i> in the path. The rank is the distance from the front node.
<code>at(<i>i</i>)</code>	int	Returns the vertex identifier of the node that sits <i>i</i> nodes away from the front node of the path. Naturally, the identities $at(ord(i)) = i$ and $ord(at(i)) = i$ always hold.
<code>pred(<i>i</i>)</code>	int	Returns a variable that denotes the predecessor of node <i>i</i> on the path.
<code>succ(<i>i</i>)</code>	int	Returns a variable that denotes the successor of node <i>i</i> on the path.
<code>head()</code>		Returns a variable that denotes the first node of the path.
<code>tail()</code>		Returns a variable that denotes the last node of the path.
<code>arcs()</code>		Returns a variable that denotes the set of arcs that are part of the path.

Table 4.2: Programming Interface for Path

All the nodes appearing in the circuit must have degree two. Aside from the different semantics, the Circuit abstract data type supports the same interface as Path. Note that some operations, e.g., `toFront`, `toBack`, `head`, `tail`, `ord`, `at` do not make sense in the context of circuits and were removed. Table 4.3 gives the revised interface.

Operation	Type	Description
<code>arc(<i>i</i>, <i>j</i>)</code>	$int \times int$	Creates an arc from node <i>i</i> to node <i>j</i> .
<code>noarc(<i>i</i>, <i>j</i>)</code>	$int \times int$	Removes the arc $i \rightarrow j$ from the circuit.
<code>swapp(<i>i</i>)</code>	int	Swaps node <i>i</i> with its predecessor along the circuit. If <i>p</i> (<i>i</i>) and <i>s</i> (<i>i</i>) denote, the predecessor and the successor of node <i>i</i> respectively, this is equivalent to removing the arcs $\langle p(i), i \rangle$, $\langle i, s(i) \rangle$, $\langle p(p(i)), p(i) \rangle$ and adding $\langle p(p(i)), i \rangle$, $\langle i, p(i) \rangle$, $\langle p(i), s(i) \rangle$
<code>swapn(<i>i</i>)</code>	int	Swaps node <i>i</i> with its successor along the circuit. If <i>p</i> (<i>i</i>) and <i>s</i> (<i>i</i>) denote, the predecessor and the successor of node <i>i</i> respectively, this is equivalent to removing the arcs $\langle p(i), i \rangle$, $\langle i, s(i) \rangle$, $\langle s(i), s(s(i)) \rangle$ and adding $\langle p(i), s(i) \rangle$, $\langle s(i), i \rangle$, $\langle i, s(s(i)) \rangle$
<code>moveBefore(<i>i</i>, <i>j</i>)</code>	$int \times int$	Moves vertex <i>i</i> before vertex <i>j</i> along the circuit.
<code>subSequence(<i>i</i>, <i>j</i>)</code>	$int \times int$	Returns the set of nodes that appear between node <i>i</i> and node <i>j</i> in the circuit.
<code>pred(<i>i</i>)</code>	int	Returns a variable that denotes the predecessor of node <i>i</i> on the circuit.
<code>succ(<i>i</i>)</code>	int	Returns a variable that denotes the successor of node <i>i</i> on the circuit.
<code>arcs()</code>		Returns a variable that denotes the set of arcs that are part of the circuit.

Table 4.3: Programming Interface for Circuit

Graph The Graph abstract data type is used to represent and manipulate general purpose directed graphs $G(V, A)$. There are no particular restriction on graphs, it is possible to describe cyclic and acyclic graphs, nodes can have a degree ranging from 0 to n where $n = |V|$. The only limitations is that for any given pair (i, j) with $i, j \in V$, there is at most one arc from *i* to *j*. Table 4.4 list the interface of the graph abstract data type.

4.11.2 Graph Invariants

The structure provided by the abstract data types is used, not only as a convenient shorthand, but more importantly, as a foundation to build invariants.

Operation	Type	Description
$\text{arc}(i, j)$	$\text{int} \times \text{int}$	Creates an arc from node i to node j .
$\text{noarc}(i, j)$	$\text{int} \times \text{int}$	Removes the arc $i \rightarrow j$ from the circuit.
$\text{pred}(i)$	int	Returns a variable that denotes the set of predecessors of node i in the graph.
$\text{succ}(i)$	int	Returns a variable that denotes the set of successors of node i in the graph.
$\text{arcs}()$		Returns a variable that denotes the set of arcs that are part of the graph.

Table 4.4: Graph

Direct Reference A first use is through the accessor methods that return variables such as `pred`, `succ` or `arcs`. The variables returned can be part of the definition of other invariants. The changes on the abstract data types are reflected on the internal representations and on the interface variables returned by those methods. The recurrence relation

$$R(i) = \begin{cases} \max_{j \in \text{Pred}(i)} R(j) + D_j & 1 \leq i \leq N + 1 \\ 0 & i = 0 \end{cases}$$

from a job-shop scheduling model illustrates the need for accessing the set of predecessors of a node in a graph in order to compute R , the longest path from node 0 to any node in the directed acyclic graph.

The recurrence can be expressed in many different ways. Since the precedence graph is composed of job precedences and machine precedences, it is possible to encode the recurrence with several arrays of integers. The job precedence are encoded in a constant array PJ and the machine precedence in an array of variable PM . These choice result in the statement

```

R : array[ i in 0..N + 1] of int := if i=0
                                then 0
                                else max(R[PJ[i]]+D[PJ[i]],R[PM[i]]+D[PM[i]]);

```

If, instead, the model uses two arrays of paths to encode the same concept, where a path encode the sequence to follow, the statement becomes

```

R : array[i in 0..N] of int := if i = 0
                                then 0
                                else max(R[SJ[JB[i]].pred(i)] + D[SJ[JB[i]].pred(i)] ,
                                             R[SM[MC[i]].pred(i)] + D[SM[MC[i]].pred(i)]);

```

The gain in clarity is mostly in the implementation of the procedure that alters the machine sequences. Note that the `pred` interface for paths returns an integer since any node in a

path can have a single predecessor. If a node does not have a predecessor, the system returns the conventional value NaN .

A direct encoding based on a graph structure allows the fragment

```

R : array[i in 0..N] of int := if i = 0
                                then 0
                                else max(j in g.pred(i)) (R[j] + D[j])

```

that is already remarkably easier to read and closer to the formal description. Note that, since a node can have multiple predecessor in a graph, the `pred` interface returns a (variable) set of integers.

Builtin Invariants on Graphs A number of standard invariants can be supported automatically in LOCALIZER by introducing builtins for them. The longest or shortest path in a graph is a typical example. In its current implementation LOCALIZER supports the following builtins:

$$\begin{aligned}
 B = \text{longest}(G, D, k, I) &\Leftrightarrow \begin{cases} P[i] = \{j : \langle j, i \rangle \in A\} \text{ where } G(V, A) \text{ is a directed graph} \\ B[i] = \max_{j \in P[i]} B[j] + D[j, i] \end{cases} \\
 B = \text{longest}(G, D, I, k) &\Leftrightarrow \begin{cases} S[i] = \{j : \langle i, j \rangle \in A\} \text{ where } G(V, A) \text{ is a directed graph} \\ B[i] = \max_{j \in S[i]} B[j] + D[i, j] \end{cases} \\
 B = \text{weight}(P, D) &\Leftrightarrow B = \sum_{(i,j) \in P} D[i, j] \text{ where } P \text{ is either a path, a circuit or a graph}
 \end{aligned}$$

The first invariant offers the possibility to maintain, in a graph G , the longest path with respect to a distance matrix D between a fixed vertex k and any vertex in I . The second takes care of the symmetric case where the source of the path can be any vertex and the sink is vertex k . The third invariant maintains the weight of a path P with respect to a weight matrix D . The weight is simply defined as the sum of the weight of the individuals arcs that compose the path.

It is important to realize that all the global invariants presented here could be expressed directly in terms of the basic blocks of LOCALIZER. However, the propagation cost associated to a global invariant is often substantially better than the cost for the corresponding encoding in terms of basic construction. For instance, a short path can be defined with a recurrence relation and therefore, directly supported within the language. However, the global propagation cost associated to the maintenance of this invariant (see Chapter 8 for a discussion of the global propagation cost) is $O(n\delta \log \delta)$. Ramalingam and Reps [53] proposed an incremental algorithm for shortest path computation that executes in $O(\delta \log \delta)$ where δ is the size of the variation in input and output. This better bound can only be

obtained if the structure and semantics of the problem are exploited. Global invariants thus offer the opportunity to make better incremental algorithm directly available to the system user.

Chapter 5

A Denotational Semantics of LOCALIZER

This chapter describes a denotational semantics for a subset of LOCALIZER called LITTLE LOCALIZER. The chapter is organized as follows. Section 5.1 introduces LITTLE LOCALIZER, Section 5.2 describes the notations used throughout the chapter, and Section 5.3 introduces the semantic algebras that support the semantic equations. Section 5.4 concludes the chapter with the presentation of the valuation functions for all the syntactic constructions.

This chapter can be skipped in a first reading without loss of continuity. Its main purpose is to clarify the concepts described informally in the previous chapters. Introductions to denotational semantics can be found in [64] and [70].

5.1 LITTLE LOCALIZER

LITTLE LOCALIZER is a subset of LOCALIZER that was chosen to simplify the presentation while preserving the main novel aspects of the language. The restrictions can be summarized as follows: All variables and invariants are integers, arrays are one-dimensional, their index set is the set of integers and their elements are assumed to be initialized to zero, expressions are restricted to the four basic arithmetic operators plus the summation aggregate, and the move instruction of the neighborhood is restricted to three most common variations. The abstract syntax and abstract categories of LITTLE LOCALIZER are given in Figure 5.1.

A LITTLE LOCALIZER statement is a sequence of sections as in LOCALIZER. The Constant section lists the literal and computed constants of the statement. The Variable

P	\in Program	V	\in Identifiers	N	\in Numerals
E	\in Expressions	I	\in Invariants Identifiers	B	\in Boolean expressions
S	\in Statements	A	\in Array Identifiers	M	\in Move instructions
D	\in Declarations	C	\in Constant Identifiers		

$P ::= \text{Constant: } D_1;$
 $\quad \text{Variable: } D_2;$
 $\quad \text{Invariant: } I;$
 $\quad \text{Local Condition: } B_1;$
 $\quad \text{Global Condition: } B_2;$
 $\quad \text{Satisfiable: } B_3;$
 $\quad \text{Objective Function: } E_4;$
 $\quad \text{Start: } S_1;$
 $\quad \text{Restart: } S_2;$
 $\quad \text{Neighborhood: } M$

$D ::= D_1 ; D_2 \mid A[v_1 \text{ in } E_1..E_2] := E_3 \mid C := E_1 \mid V$
 $I ::= I_1 ; I_2 \mid I[v_1 \text{ in } E_1..E_2] := E_3 \mid I := E$
 $B ::= B_1 \text{ and } B_2 \mid B_1 \text{ or } B_2 \mid \text{not } B \mid E_1 = E_2;$
 $E ::= E_1 + E_2 \mid E_1 - E_2 \mid E_1 * E_2 \mid E_1 / E_2 \mid V \mid C \mid I \mid N \mid \text{true} \mid \text{false} \mid a[E] \mid \text{sum}(v \text{ in } E_1..E_2) E_3;$
 $S ::= S_1; S_2 \mid V := E \mid A[E_1] := E_2 \mid \text{while } B \text{ do } S \mid \text{forall } V \text{ in } E_1..E_2 \text{ do } S \mid$
 $\quad \text{if } B \text{ then } S_1 \text{ else } S_2$
 $M ::= \text{move } S \text{ where } V \text{ in } E_1..E_2 \text{ accept when } B \mid$
 $\quad \text{best move } S \text{ where } V \text{ in } E_1..E_2 \text{ accept when } B \mid$
 $\quad \text{move } S \text{ where } V \text{ in } E_1..E_2 \text{ accept in current state when } B$

Figure 5.1: Syntactic categories and Syntax for LITTLE LOCALIZER.

section defines the state variables. The Invariant section defines the invariant variables and the expressions that give them their values. The Local Condition and Global Condition sections specify the conditions that drive the inner and outer loops of the computation model. The Satisfiable section is used to specify a boolean expression that test the satisfiability of a computation state. The Objective Function section defines an integer expression that is used to evaluate the quality of a computation state. The Start and Restart sections contain imperative code responsible for setting up the initial starting point and the restart point for multi-start strategies. Finally the Neighborhood section defines the neighborhood, the acceptance and the search functions.

There are three move instructions in LITTLE LOCALIZER. The instruction

move S where v in $E_1..E_2$ accept when B

corresponds to random walk. Note that v is a parameter and $E_1..E_2$ denotes an integer range. The instruction

best move S where v in $E_1..E_2$ accept when B

implements a greedy strategy. Both instructions are based on a simulation approach. The last move instruction

move S where v in $E_1..E_2$ accept in current state when B

is a random walk based on a differentiation approach. Finally, note that LITTLE LOCALIZER only considers decision problems.

5.2 Notations and Conventions

The basic idea of denotational semantics is to associate a function with each syntactic construct and data types of the language. It is a common practice to use the lambda calculus to define these functions. This chapter uses the lambda calculus as well, but we make use of a number of shorthands to improve clarity. For instance, when an integer value appears in a semantic object, it must be interpreted as the lambda expression that encodes the corresponding integer. Similar conventions apply to booleans, arithmetic and logical operators. As a result, a semantic expression

$$2 + 3$$

is understood as a convenient shorthand for

$$\lambda x. \lambda y. (xs)y \ 2 \ 3$$

where s stands for the natural successor function $(\lambda n. \lambda f. \lambda y. f(nfy))$. Conditionals can be easily expressed with the functions encoding truth values and we use $f_1 \rightarrow f_2 \square f_3$ to express a function that returns f_2 if f_1 is true and f_3 otherwise. It is often necessary to chain conditionals as in $f_1 \rightarrow f_2 \square f_3 \rightarrow f_4 \square f_5 \rightarrow \dots f_n$ with the obvious meaning. To highlight the case analysis, the expression is enclosed between the two keywords `case` and `end` as in

```
case
     $f_1 \rightarrow f_2$ 
   $\square$   $f_3 \rightarrow f_4$ 
   $\vdots$ 
   $\square$   $f_{n-1} \rightarrow f_n$ 
end
```

Quite often, it is necessary to alter the mapping of a function for a given value in its domain. The expression $[f_1 \rightarrow f_2]f_3$, which takes as argument f_1, f_2, f_3 , produces a function that behaves like f_3 in all its domain except for f_1 that is now mapped to f_2 . Whenever the equality symbol appears in a semantic expression, it must be interpreted as the equality between the mappings of the two functions¹.

Finally, the semantic equations make use of the fixpoint operator *fix* known in lambda calculus as the *Y* combinator. Table 5.2 shows a possible encoding for the various notational shorthands.

5.3 Semantic Algebras

Semantic algebras are the data structures for defining the meaning of programs. A semantic algebra is best described as an abstract data types that supports a variety of operations. Semantic algebras offer a convenient mechanism to abstract away details that would otherwise clutter the semantic equations. Note that some of the algebras make use of domain products. A domain $D = A \times B$ defined as the product of A and B is the set of pairs (a, b) with $a \in A$ and $b \in B$.

¹The two functions must have the same domain and, for all values in the domain, the associated values in the codomain must be equal.

0	\triangleq	$\lambda f.\lambda y.y$
1	\triangleq	$s\ 0 \triangleq \lambda f.\lambda y.(fy)$
2	\triangleq	$s\ 1 \triangleq \lambda f.\lambda y.f(fy)$
\vdots		
n	\triangleq	$\lambda f.\lambda y.\overbrace{f(f(f\ldots(fy)\ldots))}^{n\text{ times}}$
<i>true</i>	\triangleq	$\lambda x\lambda y.x$
<i>false</i>	\triangleq	$\lambda x.\lambda y.y$
<i>s</i>	\triangleq	$\lambda n.\lambda f.\lambda y.f(nfy)$
$\lambda x.\lambda y.x + y$	\triangleq	$\lambda x.\lambda y.(xs)y$
$a \rightarrow b \Box c$	\triangleq	$\lambda a.\lambda b.\lambda c.abc$
$[x \rightarrow y]f$	\triangleq	$\lambda x.\lambda y.\lambda f.\lambda z.(z = x) \rightarrow y \Box fz$

Figure 5.2: Encoding for Various Notations

The semantics of LITTLE LOCALIZER is defined in terms of the following algebras for Boolean values, natural numbers, arrays, stores, and environments.

The truth value domain contains two constant functions to encode truth and falsity. The conventional boolean operators are also provided. The Boolean domain is necessary to correctly define conditionals.

Truth values Domain : B

Operations :

true : B

false : B

\wedge : $B \rightarrow B \rightarrow B$

\vee : $B \rightarrow B \rightarrow B$

\neg : $B \rightarrow B$

The domain *Nat* is used to represent natural numbers. As is usual, *Nat* makes use of constant functions to encode all the naturals. In addition, it supports curried functions for the traditional arithmetic operators.

Naturals Domain : Nat

Operations :

0 : Nat
 1 : Nat
 \vdots
 $+$: $Nat \rightarrow Nat \rightarrow Nat$
 $-$: $Nat \rightarrow Nat \rightarrow Nat$
 $*$: $Nat \rightarrow Nat \rightarrow Nat$
 $/$: $Nat \rightarrow Nat \rightarrow Nat$

The *Array* domain is used to represent arrays that are viewed as functions mapping natural numbers to natural numbers. The first operation supported by the domain is the array constructor *anew* that creates a trivial mapping (all inputs are mapped to 0). The function *aset* can be used to alter the content of an array. It takes as input two naturals n, v , and an array a , and it produces a new array identical to a in all its entries but n that is mapped to v . Since a itself is encoded as a function $Nat \rightarrow Nat$ obtaining the value associated to a particular natural is a simple function application. For instance, the expression $(a\ 3)$ where a is an element of the *Array* domain is a function application and its result denotes the value associated with 3 in a .

Array Domain : $a \in Array : Nat \rightarrow Nat$

Operations :

$anew : Array = \lambda i.0$
 $aset : Nat \rightarrow Nat \rightarrow Array \rightarrow Nat \rightarrow Nat = \lambda n.\lambda v.\lambda a.[n \rightarrow v]a$

The *SStore* domain is similar to the *Array* domain. Its elements encode a mapping from identifiers to naturals. It is being used to associate identifiers of constants, variables and invariants to their value in a store. Since LITTLE LOCALIZER supports naturals and arrays of naturals, the *SStore* domain specializes in the mapping of identifiers to scalars. As a consequence, the *SStore* constructor simply creates an empty mapping (all identifiers are mapped to 0) and the *set* functions creates a new store where the specified identifier is mapped with the prescribed scalar. Again, accessing the store reduces to a function application with an element from the identifier domain.

SStore Domain : $\sigma \in SStore : Id \rightarrow Nat$

Operations :

new : $SStore$	$= \lambda i.0$
set : $Id \rightarrow Nat \rightarrow SStore \rightarrow SStore$	$= \lambda i.\lambda v.\lambda \sigma.[i \rightarrow v]\sigma$

The *AStore* domain is another mapping that associates identifiers with arrays. Its role is complementary to the *SStore* domain. Its behavior is identical to the previous domain, however, it must be duplicated in order to avoid typing conflicts in the semantics.

AStore Domain : $\alpha \in AStore : Id \rightarrow Array$

Operations :

new : $AStore$	$= \lambda i.\lambda j.0$
set : $Id \rightarrow Array \rightarrow AStore \rightarrow AStore$	$= \lambda i.\lambda v.\lambda \alpha.[i \rightarrow v]\alpha$

The *Store* domain reduces to pairs (*SStore*, *AStore*). The purpose of a *Store* is to keep track of a computation state. As such, it keeps all the scalar and array mappings declared in the applications. Because a store is defined with a product of domains, accessors are provided for elements of each sub domain.

Store Domain : $\tau \in Store : SStore \times AStore$

Operations :

new : $Store$	$= \lambda(\sigma, \alpha).(new, new)$
sset : $Id \rightarrow Nat \rightarrow Store \rightarrow Store$	$= \lambda i.\lambda v.\lambda(\sigma, \alpha).([i \rightarrow v]\sigma, \alpha)$
aset : $Id \rightarrow Array \rightarrow Store \rightarrow Store$	$= \lambda i.\lambda v.\lambda(\sigma, \alpha).(\sigma, [i \rightarrow v]\alpha)$
sget : $Id \rightarrow Store \rightarrow Nat$	$= \lambda i.\lambda(\sigma, \alpha).\sigma i$
aget : $Id \rightarrow Store \rightarrow Array$	$= \lambda i.\lambda(\sigma, \alpha).\alpha i$

The *Env* domain is an essential component of LITTLE LOCALIZER's semantics. Informally, an element from *Env* is a tuple $\langle l, g, s, o, r \rangle$ of five functions. The function *l* is responsible for evaluating the local condition of the computation model. The function *g* plays the same role for the global condition. The satisfiability takes the form of a Boolean function *s* and is a necessary and sufficient condition to assert that a state is indeed a solution. The Objective function *o* is devoted to the evaluation of a computation state quality. The function *r* is responsible for generating a new starting point for the algorithm.

In a sense, an environment is also a store that memorizes functions that can be invoked later on. An environment is the product of five domains. The modifier functions (SETL, SETG, SETS, SETO, SETR) can be used to “install” the local condition, the global condition, the satisfiability criteria, the objective function and the restart operator in the environment. If no function is given, the environment provides defaults with its constructor. The last five accessors provide high level functionalities. The function SAT is used to decide whether a computation store is satisfiable, LGO and GGO are used to determine the local and global conditions. The innermost loop can enter a new iteration if the local, global and the negation of the satisfiability condition simultaneously hold. The outermost loop can enter a new iteration if the local condition and the satisfiability do not hold while the global condition does. The RST function is in charge of the restart. Finally, EVAL provides a mechanism to evaluate the performance measure of a computation store.

Environment

Domain $= e \in \text{Env} : (Store \rightarrow B) \times (Store \rightarrow B) \times (Store \rightarrow B) \times$
 $(Store \rightarrow Nat) \times (Store \rightarrow Store)$

Operations =

ENEW: $\text{Env} = \lambda e. (\lambda \tau. \text{true}, \lambda \tau. \text{true}, \lambda \tau. \text{true}, \lambda n. 0, \lambda \tau. \tau)$

SETL : $(Store \rightarrow B) \rightarrow \text{Env} \rightarrow \text{Env} = \lambda f. \lambda (l, g, s, o, r). (f, g, s, o, r)$

SETG : $(Store \rightarrow B) \rightarrow \text{Env} \rightarrow \text{Env} = \lambda f. \lambda (l, g, s, o, r). (l, f, s, o, r)$

SETS : $(Store \rightarrow B) \rightarrow \text{Env} \rightarrow \text{Env} = \lambda f. \lambda (l, g, s, o, r). (l, g, f, o, r)$

SETO : $(Store \rightarrow Nat) \rightarrow \text{Env} \rightarrow \text{Env} = \lambda f. \lambda (l, g, s, o, r). (l, g, s, f, r)$

SETR : $(Store \rightarrow Store) \rightarrow \text{Env} \rightarrow \text{Env} = \lambda f. \lambda (l, g, s, o, r). (l, g, s, o, f)$

SAT : $\text{Env} \rightarrow Store \rightarrow B = \lambda (l, g, s, o, r). \lambda \tau. s\tau$

LGO : $\text{Env} \rightarrow Store \rightarrow B = \lambda (l, g, s, o, r). \lambda \tau. (l\tau) \wedge (g\tau) \wedge (\neg s\tau)$

GGO : $\text{Env} \rightarrow Store \rightarrow B = \lambda (l, g, s, o, r). \lambda \tau. (\neg l\tau) \wedge (g\tau) \wedge (\neg s\tau)$

RST : $\text{Env} \rightarrow Store \rightarrow Store = \lambda (l, g, s, o, r). \lambda \tau. r\tau$

EVAL : $\text{Env} \rightarrow Store \rightarrow Nat = \lambda (l, g, s, o, r). \lambda \tau. o\tau$

Note that several domains use the same function name to define similar operations. This choice is motivated by the desire to keep the valuation function readable. When one of the function name appears in a function definition, the exact function invoked should be clear from the context, i.e, the types of the surrounding expressions.

5.4 The Semantics

This section gives the semantic equations for each syntactic construction of LITTLE LOCALIZER. For clarity, the discussion starts with the description of the valuation functions for expressions. As we progress, more and more complex equations are introduced, with the valuation function for $P \in \mathcal{P}$ presented last.

5.4.1 Expressions

LITTLE LOCALIZER manipulates Boolean and Natural expressions. All expressions are functions that take as input a store and return either a boolean or a natural.

The denotational semantics for a Boolean expression B is given by a function \mathcal{B} that maps expressions to computation stores to truth functions. The first two functions simply deal with Boolean constants while the last four are responsible for the usual operators and the basic equality relation.

Valuation Function \mathcal{B} : $\text{Expression} \rightarrow \text{Store} \rightarrow B$

$$\begin{aligned}
 \mathcal{B}[\text{true}] &= \lambda\tau. \text{true} \\
 \mathcal{B}[\text{false}] &= \lambda\tau. \text{false} \\
 \mathcal{B}[B_1 \text{ and } B_2] &= \lambda\tau. \mathcal{B}[B_1]\tau \wedge \mathcal{B}[B_2]\tau \\
 \mathcal{B}[B_1 \text{ or } B_2] &= \lambda\tau. \mathcal{B}[B_1]\tau \vee \mathcal{B}[B_2]\tau \\
 \mathcal{B}[\text{not } B] &= \lambda\tau. \neg \mathcal{B}[B]\tau \\
 \mathcal{B}[E_1 = E_2] &= \lambda\tau. \mathcal{E}[E_1]\tau = \mathcal{E}[E_2]\tau
 \end{aligned}$$

The denotational semantics of integer expressions is given by a semantic function \mathcal{E} that takes as input a computation store and returns a natural number. The semantic equations for literals are straightforward. The equations for variables (v), invariant variables (i) and constants (c), lookup the object in the computation store τ . The next two functions deal with array dereferencing by first looking up the array in the store and then passing to the array the value of the index to get to the element. The equations for the binary operators are traditional in the sense that they simply compose the semantics of the operands with the appropriate function.

Valuation Function \mathcal{E} : $\text{Expression} \rightarrow \text{Store} \rightarrow \text{Nat}$

In the following equations, the lower case letters n, v, i, c, a are elements of the classes N, V, I, C, A . n stands for an integer while v, i, c , and a are specific identifiers.

$$\begin{aligned}
\mathcal{E}[[n]] &= \lambda\tau.n \\
\mathcal{E}[[v]] &= \lambda\tau.sget\ v\ \tau \\
\mathcal{E}[[i]] &= \lambda\tau.sget\ i\ \tau \\
\mathcal{E}[[c]] &= \lambda\tau.sget\ c\ \tau \\
\mathcal{E}[[a[E_1]]] &= \lambda\tau.(aget\ a\ \tau)(\mathcal{E}[[E_1]]\tau) \\
\mathcal{E}[[i[E_1]]] &= \lambda\tau.(aget\ i\ \tau)(\mathcal{E}[[E_1]]\tau) \\
\mathcal{E}[[E_1 + E_2]] &= \lambda\tau.(\mathcal{E}[[E_1]]\tau) + (\mathcal{E}[[E_2]]\tau) \\
\mathcal{E}[[E_1 - E_2]] &= \lambda\tau.(\mathcal{E}[[E_1]]\tau) - (\mathcal{E}[[E_2]]\tau) \\
\mathcal{E}[[E_1 * E_2]] &= \lambda\tau.(\mathcal{E}[[E_1]]\tau) * (\mathcal{E}[[E_2]]\tau) \\
\mathcal{E}[[E_1 / E_2]] &= \lambda\tau.(\mathcal{E}[[E_1]]\tau) / (\mathcal{E}[[E_2]]\tau) \\
\mathcal{E}[[\text{sum}(v\ \text{in}\ E_1..E_2)E_3]] &= \lambda\tau.\text{let } l = \mathcal{E}[[E_1]]\tau \text{ in } \sum_{k=l}^u \mathcal{E}[[E_3]](sset\ v\ k\ \tau) \\
&\quad u = \mathcal{E}[[E_2]]\tau
\end{aligned}$$

5.4.2 Statements

We now turn to the semantic function \mathcal{S} for statements. \mathcal{S} is a function that maps statements to semantic functions that transform computation stores. This function is of signature $\text{Statement} \rightarrow \text{Store} \rightarrow \text{Store}$ and is defined as follows:

Valuation Function \mathcal{S} : $\text{Statement} \rightarrow \text{Store} \rightarrow \text{Store}$

$$\begin{aligned}
\mathcal{S}[[S_1; S_2]] &= \lambda\tau.\mathcal{S}[[S_2]](\mathcal{S}[[S_1]]\tau) \\
\mathcal{S}[[v := E]] &= \lambda\tau.sset\ v\ (\mathcal{E}[[E]]\tau)\tau \\
\mathcal{S}[[a[E_1] := E_2]] &= \lambda\tau.\text{let } l = \mathcal{E}[[E_1]]\tau \\
&\quad v = \mathcal{E}[[E_2]]\tau \\
&\quad d = aset\ l\ v\ aget\ a\ \tau \\
&\quad \text{in } aseta\ d\ \tau \\
\mathcal{S}[[\text{if } B \text{ then } S_1 \text{ else } S_2]] &= (\mathcal{B}[[B]]\tau) \rightarrow (\mathcal{S}[[S_1]]\tau) \square (\mathcal{S}[[S_2]]\tau) \\
\mathcal{S}[[\text{while } B \text{ do } S]] &= fix\ (\lambda f.\lambda\tau.\mathcal{B}[[B]]\tau \rightarrow f(\mathcal{S}[[S]]\tau) \square \tau)
\end{aligned}$$

$$\begin{aligned}
\mathcal{S}[\text{forall } v \text{ in } E_1..E_2 \text{ do } S] &= \lambda\tau. \text{let } \tau_0 = sset\ v\ (\mathcal{E}[E_1]\tau)\tau \\
&\quad F = \lambda f. \lambda\tau. (sget\ v\ \tau) \leq (\mathcal{E}[E_2]\tau) \rightarrow \\
&\quad \quad f\ (sset\ v\ ((sget\ v\ \tau) + 1)(\mathcal{S}[S]\tau)) \square \tau \\
&\quad \text{in } fix\ F\ \tau_0
\end{aligned}$$

The first equation expresses the chaining rule. The next two equations define the meaning of the assignment operator to a scalar variable and to an element in an array of scalars. The last three equations are responsible for defining the meaning of branching and of two forms of iterative statements. The novelty in the equations is the use of the fix-point operator to define iterative statements as the least fix-point of an auxiliary function.

5.4.3 Declarations

The valuation function presented in this section is responsible for building the store constructor. \mathcal{D} is responsible for the meaning of the constant and variables declarations.

Valuation Function \mathcal{D} : Declaration \rightarrow Store \rightarrow Store

$$\begin{aligned}
\mathcal{D}[D_1; D_2] &= \lambda\tau. \mathcal{D}[D_2](\mathcal{D}[D_1]\tau) \\
\mathcal{D}[c := E] &= \lambda\tau. sset\ (c)\ (\mathcal{E}[E]\tau)\tau \\
\mathcal{D}[a[v \text{ in } E_1..E_2] := E_3] &= \lambda\tau. \text{let } \tau_0 = sset\ v\ (\mathcal{E}[E_1]\tau)\tau \\
&\quad F = \lambda f. \lambda\tau. \text{let } u = \mathcal{E}[E_2]\tau \\
&\quad \quad a_1 = aget\ a\ \tau \\
&\quad \quad r = \mathcal{E}[E_2]\tau \\
&\quad \quad a_2 = aset\ v\ r\ a \\
&\quad \text{in } sget\ v\ \tau \leq u \rightarrow f\ (sset\ v\ ((sget\ v\ \tau) + 1) \\
&\quad \quad (aset\ aa_2\ \tau)) \square \tau \\
&\quad \text{in } fix\ F\ \tau_0 \\
\mathcal{D}[v] &= \lambda\tau. sset\ v\ 0\ \tau
\end{aligned}$$

The role of \mathcal{D} is to build a store transformer. Again, the first equation defines the chaining of two declarations. The second equation takes care of a scalar constant declaration. It produces a function that alters its input (a store) to associate the evaluation of E_2 with the identifier c . The next equation fulfills the same objective for an array, it must use a fixpoint operator to produce the mapping for the whole array a . The last equation declares a variable by associating the identifier v to the value 0 in the modified store.

The function \mathcal{J} is responsible for producing a function that alters the computation store by adding new mappings for all the invariants.

Valuation Function \mathcal{J} : $\text{Invariants} \rightarrow \text{Store} \rightarrow \text{Store}$

$$\begin{aligned}\mathcal{J}[I_1; I_2] &= \lambda\tau. \mathcal{J}[I_2](\mathcal{J}[I_1]\tau) \\ \mathcal{J}[i := E] &= \lambda\tau. sset\ i\ 0\ \tau \\ \mathcal{J}[i[v\ \text{in}\ E_1..E_2] := E_3] &= \lambda\tau. aset\ i\ anew\ \tau\end{aligned}$$

The maintenance of invariants is discussed in the next section.

5.4.4 Invariants maintenance

The denotational semantics presented in this section abstracts away the details of maintaining the invariants, which are discussed at length in the next chapter, in a function *propagate* whose signature is $\{\text{Invariant Declarations}\} \rightarrow \text{Store} \rightarrow \text{Store}$. The purpose of *propagate* is to compute a new store with the updated values of the invariants declared in the statement. Here we only provide a (non-executable) specification of the function.

The function *propagate* returns a new store τ' where the variables associated to element in \mathcal{V} are unchanged and variables associated to \mathcal{J} are modified in order to verify all the invariant identities in I . It uses a validity function \mathcal{C} to check that the evaluation of an invariant in a given store is satisfied. \mathcal{C} is thus a valuation function of signature $\text{Invariant Identifier} \rightarrow \text{Store} \rightarrow B$ that takes as input an invariant declaration and a store and returns a Boolean. More formally,

$$\text{propagate } I\ \tau = \tau'$$

where

$$\begin{aligned}\forall j \in V : \tau'j &= \tau j \\ \forall j \in I : \mathcal{C}[I]\tau' &= \text{true}\end{aligned}$$

with

$$\mathcal{C}[i := E] = \lambda\tau. ((sget\ i\ \tau) = \mathcal{C}[E]\tau) \rightarrow \text{true} \sqcap \text{false}$$

$$\begin{aligned}
\mathcal{C}[[i[v \text{ in } E_1..E_2] := E_3]] &= \lambda\tau. \text{ let } \tau_0 = sset \ v \ (\mathcal{C}[[E_1]]\tau)\tau \\
F &= \lambda f. \lambda\tau. \text{ let } n_1 = sget \ v \ \tau \\
&\quad n_2 = (sget \ v \ \tau) + 1 \\
&\quad v_1 = \mathcal{C}[[E_3]]\tau \\
&\quad v_2 = (aget \ i \ \tau)n_1 \\
&\text{ in } (v_1 = v_2 \rightarrow (n_1 \leq (\mathcal{C}[[E_2]]\tau)) \rightarrow \\
&\quad f \ (sset \ v \ n_2 \ \tau) \\
&\quad \Box true \\
&\quad \Box false \\
&\text{ in } fix \ F \ \tau_0
\end{aligned}$$

The first equation of \mathcal{C} is used for scalar invariants. It returns `true` if and only if the evaluation of the expression E in the store is equal to the value associated with the identifier i in the same store. The second equation follows the same principle with the additional complexity coming from the inspection of all the entries in the array.

5.4.5 Neighborhood

This section focuses on the valuation functions for the three types of move instructions supported by `LITTLE LOCALIZER`. The function \mathcal{M} is responsible for creating a function with signature $Env \rightarrow Store \rightarrow Store$ from a move statement. Recall that Env denotes the environment, i.e., a store of functions used to query and manipulate a computation store.

Valuation Function \mathcal{M} : $Move \rightarrow Env \rightarrow Store \rightarrow Store$

We first consider

$$\mathcal{M}[[\text{best move } S \text{ where } v \text{ in } E_1..E_2 \text{ accept when } B]] = fix \ F$$

where

$$\begin{aligned}
F = & \lambda f. \lambda e. \lambda \tau. \text{ case } SAT \ e \ \tau \rightarrow \tau \\
& GGO \ e \ \tau \rightarrow \text{let } s = \mathcal{E}[\![searches]\!]\tau + 1 \\
& \quad \tau_1 = sset \ searches \ s \ (RST \ e \ \tau) \\
& \quad \tau_2 = sset \ trials \ 0 \ \tau_1 \\
& \quad \text{in } f \ e \ \tau_2 \\
& LGO \ e \ \tau \rightarrow \text{let } l = \mathcal{E}[\![E_1]\!]\tau \\
& \quad u = \mathcal{E}[\![E_2]\!]\tau \\
& \quad \tau^1 = sset \ trials \ ((\mathcal{E}[\![trials]\!]\tau) + 1) \tau \\
& \quad \tau_k^2 = \mathcal{S}[\![S]\!](sset \ v \ k \ \tau^1) \ l \leq k \leq u \\
& \quad \tau_k^3 = propagate \ I \ \tau_k^2 \\
& \quad p = argmax_{l \leq k \leq u} (EVAL \ e \ \tau_k^3) \\
& \quad \tau^+ = sset \ delta \ (EVAL \ e \ \tau - EVAL \ e \ \tau_p) \tau_p \\
& \quad \text{in } \mathcal{B}[\![B]\!]\tau^+ \rightarrow f \ e \ \tau^+ \square \tau \\
& \text{end}
\end{aligned}$$

The meaning of a move instruction is defined as the fixpoint of F . F itself defines the core of LITTLE LOCALIZER. It is a template for a function that takes as input an environment and a store and returns a new store. F represents one step of the computation, i.e., one transition in the neighborhood. The whole computation is obtained by computing the least fixpoint of F . To understand F itself, it is necessary to look at the various cases. When the store passed to the function is satisfiable, F simply returns the store itself. A new store is produced as the result of the restart operation and the setting of *searches* and *trials* whenever the store satisfies the global condition². F is then called recursively with this new store. The core of LITTLE LOCALIZER appears in the local case. Intuitively, the function produces all the stores that result from assigning a value k in the range $l..u$ to v and propagating the invariants I . The function then selects the store τ_p that has the best performance. The acceptance condition is finally evaluated with respect to a store τ^+ that differs from τ_p in the assignment of the keyword *delta* to the actual gain. If the move is acceptable, the function is called recursively with the new store, otherwise it simply returns τ since the best move is not acceptable.

The two other variants of \mathcal{M} are organized around the same skeleton and the intuitive meaning should be clear. Their formal definitions follow.

$$\mathcal{M}[\![\text{move } S \text{ where } v \text{ in } E_1..E_2 \text{ accept when } B]\!] = fix \ F_1$$

²See the Environment algebra for a definition of the meaning for the global condition.

$\mathcal{M}[\text{move } S \text{ where } v \text{ in } E_1..E_2 \text{ accept in current state when } B] = \text{fix } F_2$

where

```

F1 = λf.λe.λτ. case SAT e τ → τ
      GGO e τ → let τ1 = RST e τ
                  s     =  $\mathcal{E}[\text{searches}]\tau_1 + 1$ 
                  τ2   = sset searches s τ1
                  τ3   = sset trials 0 τ2
                  in f e τ3
      LGO e τ → let k   = random ( $\mathcal{E}[E_1]\tau$ ) ( $\mathcal{E}[E_2]\tau$ )
                  τ1 = sset trials (( $\mathcal{E}[\text{trials}]\tau$ ) + 1)τ
                  τ2 =  $\mathcal{S}[S](\text{sset } v \text{ } k \text{ } \tau^1)$ 
                  τ3 = propagate I τ2
                  τ+ = sset delta (EVAL e τ - EVAL e τ3)τ3
                  in  $\mathcal{B}[B]\tau^+ \rightarrow f \text{ } e \tau^+ \square f \text{ } e \tau^1$ 
end

```

```

F2 = λf.λe.λτ. case SAT e τ → τ
      GGO e τ → let τ1 = RST e τ
                  s     =  $\mathcal{E}[\text{searches}]\tau_1 + 1$ 
                  τ2   = sset searches s τ1
                  τ3   = sset trials 0 τ2
                  in f e τ3
      LGO e τ → let k   = random ( $\mathcal{E}[E_1]\tau$ ) ( $\mathcal{E}[E_2]\tau$ )
                  τ1 = sset trials (( $\mathcal{E}[\text{trials}]\tau$ ) + 1)τ
                  τ2 =  $\mathcal{S}[S](\text{sset } v \text{ } k \text{ } \tau^1)$ 
                  τ3 = propagate I τ2
                  in  $\mathcal{B}[B]\tau \rightarrow f \text{ } e \tau^3 \square f \text{ } e \tau^1$ 
end

```

The last two valuation function make use of the expression *random*. This expression can be seen as a function that takes two natural numbers. Its output is another natural number that is drawn at random from the range defined by its arguments. Additionally, the sequence of values obtained by subsequent application of *random* to the same interval must be uniformly distributed over the interval.

5.4.6 Program

The definition of the valuation function for a program is now simple.

Valuation Function \mathcal{P} : Program \rightarrow Store

$$\begin{aligned}
 \mathcal{P}[[P]] = \text{let } \tau_1 &= \mathcal{D}[[D_2]](\mathcal{D}[[D_1]]_{new}) \\
 \tau_2 &= \mathcal{S}[[S_1]](\mathcal{I}[[I]]\tau_1) \\
 e_1 &= SETS \ \mathcal{B}[[B_3]](SETG \ \mathcal{B}[[B_2]](SETL \ \mathcal{B}[[B_1]]_{enew})) \\
 e_2 &= SETR \ \mathcal{S}[[S_2]](SETO \ \mathcal{E}[[E_4]]e_1) \\
 \tau_3 &= sset \ trials \ 0 \ \tau_2 \\
 \tau_4 &= sset \ searches \ 0 \ \tau_3 \\
 \tau_5 &= propagate \ I \ \tau_3 \\
 &\text{in } \mathcal{M}[[M_1]]e_2 \ \tau_5
 \end{aligned}$$

where

$P ::=$ Constant: D_1 ;
Variable: D_2 ;
Invariant: I ;
Local Condition: B_1 ;
Global Condition: B_2 ;
Satisfiable: B_3 ;
Objective Function: E_4 ;
Start: S_1 ;
Restart: S_2 ;
Neighborhood: M_1 ;

The last function \mathcal{P} gives the semantics for a LITTLE LOCALIZER statement. The equation constructs the store and the environment with the help of the auxiliary functions \mathcal{D} , \mathcal{S} , \mathcal{E} and then uses the function \mathcal{M} to produce the output state.

A denotational semantics for a complete version of LOCALIZER is possible. Extending LITTLE LOCALIZER semantics does not raise major difficulties. The extensions consist of introducing new semantic domains for the additional data types supported by LOCALIZER such as tuples, sets, paths and graphs. Some of the valuation functions, in particular those related to expressions, must be extended to cover these new cases. Finally, a complete semantics must also cover all possible elementary move instructions. New valuation functions

can be added for the first strategy and variants can be introduced for the optimization based model together with modification to the environment semantic domain to keep track of the best known solution.

Chapter 6

Implementation

This chapter reviews the implementation of invariants which are the cornerstone of LOCALIZER. Informally speaking, invariants are implemented using a planning/execution model. The planning phase generates a specific order for propagating the invariants, while the execution phase actually performs the propagation. This model makes it possible to propagate only differences between two states and mimics, to a certain extent, the way specific local search algorithms are implemented.

The planning/execution model imposes some restrictions on the invariants. Intuitively, these restrictions make sure that there is an order in which the invariants can be propagated so that a pair (variable,invariant) is considered at most once. Various such restrictions can be imposed. Static invariants can be ordered at compile time and are thus especially efficient. However, static invariants rule out some interesting models for scheduling and resource allocation problems. Dynamic invariants still make it possible to produce an ordering so that a pair (variable,invariant) is considered at most once. However, they require to interleave the planning and execution phases. Dynamic invariants seem to be a good compromise between efficiency and expressiveness.

The rest of this chapter is organized as follows. The algorithms use normalized invariants that correspond to a simpler internal representation of the high level invariants and Section 6.1 reviews the normalization process. Section 6.2 describes static invariants (i.e., invariants that can be completely planned at compile time) and their implementation. Section 6.3 describes dynamic invariants, (i.e., the planning and the execution phases must be interleaved) and their implementation. These two sections restrict their attention to arithmetic invariants and should give readers a preliminary understanding of the implementation. Sections 6.4.3, 6.4.4, 6.4.5 and Section 6.4.6 revisit static and dynamic invariants to support a

variety of set invariants.

6.1 Normalization

The invariants of LOCALIZER are rewritten into primitive invariants by flattening expressions and arrays. The primitive invariants are of the form:

$x := c$	
$x := y \oplus z$	$(y \neq z)$
$x := \prod(x_1, \dots, x_n)$	$(x_i \neq x_j)$
$x := \text{element}(e, x_1, \dots, x_n)$	$(x_i \neq x_j)$

where c is a constant, x, y, z, x_1, \dots, x_n are variables (e.g., state variables, invariant variables or even intermediate variables), \oplus is an arithmetic operator such as $+$ and $*$ or an arithmetic relation such as $\geq, >$ and $=$, and \prod is an aggregate operator such as **sum**, **prod**, **max**, **min**, **argmax**, and **argmin**. Relations return 1 when true and 0 otherwise. An invariant

$$x := \text{element}(e, x_1, \dots, x_n)$$

assigns to x the element in position e in the list $[x_1, \dots, x_n]$. This last invariant is useful for arrays which are indexed by expressions containing variables.

At any given time, LOCALIZER maintains a set of invariants \mathcal{I} over variables \mathcal{V} . Given an invariant $I \in \mathcal{I}$ of the form $x := e$ (where e is an expression), $\text{def}(I)$ denotes x while $\text{exp}(I)$ denotes e . Given a set of invariants \mathcal{I} over \mathcal{V} and $x \in \mathcal{V}$, $\text{invariants}(x, \mathcal{I})$ returns the subset of invariants $\{I_i\} \subseteq \mathcal{I}$ such that x occurs in $\text{exp}(I_i)$. The set of variables $\{\text{def}(I) | I \subseteq \mathcal{I}\}$ are the invariant variables and cannot be directly modified. The set of variables $\mathcal{V} \setminus \{\text{def}(I) | I \subseteq \mathcal{I}\}$ are the state variables appearing in the system of invariants. These variables only can be modified in the neighborhood definitions. Note also that an invariant variable x can be defined by at most one invariant, i.e., there exists at most one $I \subseteq \mathcal{I}$ such that $\text{def}(I) = x$.

6.2 Static Invariants

The basic assumption behind LOCALIZER implementation is that invariants only change marginally when moving from one state i to a neighboring state in $A(N(i))$. Consequently, the goal of the implementation is to run in time proportional to the amount of changes. More precisely, the implementation makes sure that a pair $\langle x, I \rangle$ where x occurs in the definition of

the invariants I is considered at most once, i.e., when the variable x is updated, the invariant I is propagated and the variable $def(I)$ is updated but it will never be reconsidered again because of variable x .

To achieve this goal, the implementation uses a planning/execution model where the planning phase determines an ordering for the updates and the execution phase actually performs them. The existence of a suitable ordering is guaranteed by the restrictions imposed on the invariants by the system. Note also that planning/execution models are often in graphical constraint systems (e.g., [6]) in techniques such as finite differencing (e.g., [46]) and in techniques based on attribute grammars (e.g., [56],[54],[55],[4]).

This section describes static invariants that impose a static restriction (i.e. a restriction at compile time). Although this restriction may seem strong, it accommodates many models for applications such as satisfiability and graph coloring to name a few. The main practical limitation is that elements of arrays cannot depend on other elements in the same array. This restriction is lifted by dynamic invariants. Note, however, that static invariants have the nice property that the planning phase can be entirely performed at compile time.

6.2.1 The Planning Phase

The basic idea behind static invariants is to require the existence of a topological ordering on the variables (and thus on the invariants). This topological ordering is obtained by associating a topological number $t(x)$ with each variable x . The topological number of an invariant I is simply $t(def(I))$. The topological numbers are obtained from constraints derived from the invariants.

Definition 8 The topological constraints I are defined inductively using the following rules:

$$\begin{aligned}
 tc(x := c) &= \{t(x) = 0\} \\
 tc(x := y \oplus z) &= \{t(x) = \max(t(y), t(z)) + 1\} \\
 tc(x := \prod(x_1, \dots, x_n)) &= \{t(x) = \max(t(x_1), \dots, t(x_n)) + 1\} \\
 tc(x := element(e, x_1, \dots, x_n)) &= \{t(x) = \max(t(e), t(x_1), \dots, t(x_n)) + 1\}
 \end{aligned}$$

Definition 9 The topological constraints of a set of invariants \mathcal{I} , denoted by $tc(\mathcal{I})$, is $\bigcup_{I \in \mathcal{I}} tc(I)$.

Definition 10 A set of invariants \mathcal{I} over t is static if there exists an assignment $t : \mathcal{V} \rightarrow \mathcal{N}$ such that t satisfies $tc(\mathcal{I})$.

The planning phase for static invariants consists of finding the topological assignment. The planning phase can be performed at compile time since the topological constraints do not depend on the values of the variables in a given state. Indeed, each topological constraints express relationships among variables and never use the computation state in their definition. The independence from the computation state is a direct consequence of the choice of topological constraint for the element invariant

$$tc(x := element(e, x_1, \dots, x_n)) = \{t(x) = \max(t(e), t(x_1), \dots, t(x_n)) + 1\}$$

which states that $t(x)$ depends on $t(e)$ and all the topological values $t(x_1), \dots, t(x_n)$. This formalization captures the union of all possible dependencies between x and the x_i irrespective of the value of e . Note that the existence of a topological assignment is conditional to the absence of cycles in the topological graph.

6.2.2 The Execution Phase

The execution phase is given a set of variables \mathcal{M} which have been updated and a topological assignment t . It then propagates the changes according to the topological ordering. The algorithm uses a queue which contains pairs of the form $\langle x, I \rangle$. Intuitively, such a pair means that invariant I must be reconsidered because variable x has been updated. The main step of the algorithm consists of popping the pair $\langle x, I \rangle$ with the smallest $t(I)$ and to propagate the change, possibly adding new elements to the queue. The algorithm is shown in Figure 6.1.

6.2.3 Propagating the Invariants

To complete the description of the implementation of static invariants, it remains to describe how to propagate the invariants themselves. The basic idea here is to associate two values x^o and x^c with each variable x . The value x^o represents the value of variable x at the beginning of the execution phase, while the value x^c represents the current value of x . At the beginning of the execution phase $x^o = x^c$. By keeping these two values, it is possible to compute how much a variable has changed and to update the invariants accordingly. For instance, the propagation of the invariant

$$x := sum(x_1, \dots, x_n)$$

is performed by the procedure shown in Figure 6.2.


```

procedure execute( $\mathcal{I}, \mathcal{M}, t$ )
begin
  1  $Q := \{ \langle x, I \rangle \mid x \in \mathcal{M} \wedge I \in \text{invariants}(x, \mathcal{I}) \}$ 
  2 while  $Q \neq \emptyset$  do
  3    $i := \min_{\langle y, I_2 \rangle \in Q} t(I_2)$ ;
  4    $W := \{ \langle x, I_1 \rangle \in Q \mid t(I_1) = i \}$ ;
  5    $Q := Q \setminus W$ ;
  6   while  $W \neq \emptyset$  do
  7      $\langle x, I \rangle := \text{POP}(W, t)$ ;
  8     propagate( $x, I, \mathcal{I}, Q'$ );
  9      $W := W \setminus \{ \langle x, I \rangle \}$ ;
  10     $Q := Q \cup Q'$ ;
  11  endwhile
  12 endwhile
end;

function POP( $Q, t$ )
  Pre:  $Q$  is not empty
  Post:  $\langle x, I \rangle \in Q$  such that  $\forall \langle x', I' \rangle \in Q : t(I') \geq t(I)$ 

```

Figure 6.1: The Execution Phase for Static Invariants.

```

procedure propagate( $x_i, x := \text{sum}(x_1, \dots, x_n), \mathcal{I}, Q$ )
begin
   $x^c := x^c + (x_i^c - x_i^o)$ ;
  if  $x^c \neq x^o$  then
     $Q := \text{invariants}(\mathcal{I}, x)$ ;
  end

```

Figure 6.2: Propagation routine for the summation elementary invariant.

The procedure updates x^c according to the change of x_i . Note that, because of the topological ordering, x_i^c has reached its final value. Note also that x^c is not necessarily final after this update, because other pairs $\langle x_j, x := \text{sum}(x_1, \dots, x_n) \rangle$ may need to be propagated. Finally, note that the static elementary invariants described here obey the complexity bounds reported in Table 6.1.

Invariant	Time	Space
$x := c$	$\Theta(1)$	$\Theta(1)$
$x := y \oplus z$	$\Theta(1)$	$\Theta(1)$
$x := \min(i \text{ in } \{1..n\})x_i$	$O(\log n)$	$\Theta(n)$
$x := \max(i \text{ in } \{1..n\})x_i$	$O(\log n)$	$\Theta(n)$
$x := \text{argmin}(i \text{ in } \{1..n\})x_i$	$O(\log n)$	$\Theta(n)$
$x := \text{argmax}(i \text{ in } \{1..n\})x_i$	$O(\log n)$	$\Theta(n)$
$x := \text{sum}(i \text{ in } \{1..n\})x_i$	$\Theta(1)$	$\Theta(n)$
$x := \text{prod}(i \text{ in } \{1..n\})x_i$	$\Theta(1)$	$\Theta(n)$

Table 6.1: Space and time Complexity Bounds for Static Invariants

6.2.4 Correctness

Intuitively, the correctness of the propagation scheme is proved by induction on the topological order of the constraint graph. The idea is to show that the invariants hold before the propagation with respect to the old computation state and, by the end of the propagation, the invariants hold with respect to the new computation state. The propagation routine of Figure 6.1 considers the invariants in steps. Step i is responsible for the propagation of all the invariants with topological number i .

First, it is useful to introduce several notations. A *computation state* τ is an assignment of values to the variables of the program. Let \mathcal{S} denote the set of state variables, i.e., $\mathcal{S} = \mathcal{V} \setminus \{\text{def}(I) | I \in \mathcal{I}\}$. A *restricted invariant set* \mathcal{I}_i is the subset of \mathcal{I} whose elements have topological values no greater than i , i.e., $\mathcal{I}_i = \{I \in \mathcal{I} | t(\text{def}(I)) \leq i\}$. The complement of a restricted set of invariant \mathcal{I}_i^c is defined as $\mathcal{I} \setminus \mathcal{I}_i$. The *valuation* of an expression e with respect to a store τ is denoted by $\tau(e)$ and represents the evaluation of e when all the variables are substituted by their values in τ . Let $\tau(\mathcal{I})$ denote the relation

$$\bigwedge_{I \in \mathcal{I}} \tau(\text{def}(I)) = \tau(\text{exp}(I)).$$

Intuitively, $\tau(\mathcal{I})$ holds if and only if the valuation of all invariants $I \in \mathcal{I}$ with respect to τ is equal to the value associated to $\text{def}(I)$ in τ . In the rest of this section, we will assume

that τ'_i denotes the new computation state at the beginning of step i and that n denotes the total number of steps. Therefore, τ'_1 is the initial computation state and τ'_{n+1} is the final computation state after the propagation of step n .

Formally, proving the correctness of the global propagation algorithm shown in Figure 6.1 amounts to showing that the properties

$$\begin{aligned} (i) \quad & \tau'_i(\mathcal{I}_{i-1}) \quad \text{holds} \\ (ii) \quad & Q = \{ \langle x, I \rangle \mid I \in \mathcal{I}_{i-1}^c \wedge x \in \text{exp}(I) \wedge \tau'_{i-1}(x) \neq \tau(x) \} \\ (iii) \quad & \tau'_i(y) = \tau(y) \quad \forall y \in \{ z \in \mathcal{V} \mid z \in \text{def}(I) \wedge I \in \mathcal{I}_{i-1}^c \} \end{aligned}$$

hold each time the algorithm reaches the program point (4). The first property indicates that the computation state τ'_i is consistent with respect to all the invariants that appear in steps strictly earlier than i . The second condition indicates that the queue contains pairs for all the invariants appearing in future steps for which there is a change in one of the parameter variables. The third one indicates that variables appearing in future steps are left unchanged by the propagation of all the steps up to i . The proof establishes that, if the conditions hold at the beginning of step i , they will also hold at the end of step i , hence at the beginning of step $i + 1$. Once the last step is finished, condition (i) alone establishes the correctness of the overall algorithm.

It is necessary to specify the procedure *propagate* to prove these properties. Informally speaking, the specification says that propagating the changes on the variables of the invariants restore the invariant relation while leaving everything else unchanged. It also says that the order of propagation has no importance. The following notations are used throughout the rest of this section. Let the function G (which can be viewed as a store) be defined as

$$G(\tau_1, \tau_2, W, I) = \lambda y. \begin{cases} \tau_1(y) & \Leftarrow \langle y, I \rangle \in W \\ \tau_2(y) & \Leftarrow \langle y, I \rangle \notin W \end{cases}$$

G is a mapping from a pair of stores, a queue, and an invariant to a store. Intuitively, G behaves like τ_1 when the queue W is “full” and like τ_2 when W is empty. In between, the values of the variables depend on whether or not they are in W . In the specification, τ_1 will be the initial store and τ_2 will be the current store. Let x^g be an abbreviation for $G(\tau, \tau'_k, W, I)(x)$. Let the property $P(\tau, \tau', \tau'', W, I)$ where I is $x := E(y_1, \dots, y_n)$, be defined as

$$\tau''(x) = E(G(\tau, \tau', W, I)(y_1), \dots, G(\tau, \tau', W, I)(y_n)).$$

Intuitively, P holds when the evaluation of $E(y_1, \dots, y_n)$ in $G(\tau, \tau', W, I)$ is equal to the value associated to x in τ'' .

Definition 11 Let τ and τ'_i be computation states, and $W \cup \{\langle y_k, I \rangle\}$ (I is $x := E(y_1, \dots, y_n)$) be a queue such that $P(\tau, \tau'_i, \tau'_i, W \cup \{\langle y_k, I \rangle\}, I)$ holds. The propagation routine

$$\text{propagate}(y_k, x := E(y_1, \dots, y_n), \mathcal{I}, Q)$$

is *locally consistent* if it produces a store τ'_{i+1} that satisfies the postconditions

$$\begin{aligned} (p_1) \quad & P(\tau, \tau'_i, \tau'_{i+1}, W, I) \\ (p_2) \quad & \bigwedge_{z \in \mathcal{V} \setminus \{x\}} \tau'_{i+1}(z) = \tau'_i(z) \\ (p_3) \quad & Q = \{\langle x, J \rangle \mid J \in \mathcal{I} \wedge x \in \text{exp}(J) \wedge \tau'_{i+1}(x) \neq \tau'_i(x)\} \end{aligned}$$

Intuitively, the precondition requires that the value of x in τ'_i be consistent with the evaluation of I in $G(\tau, \tau'_i, W \cup \{\langle y_k, I \rangle\}, I)$: a store where y_k is bound to its old value. The post condition p_1 states that, after the propagation the value of x in τ'_{i+1} is consistent with the evaluation of I in a computation store that takes into account the new value of y_k . Post condition p_2 indicates that τ'_{i+1} and τ'_i are identical for all entries but x . Post condition p_3 states that the output argument Q contains the set of pairs $\langle x, J \rangle$ where J is an invariant that depends on the modified variable x .

Lemma 1 The loop in Figure 6.1 ranging from line (6) to (11) has the following properties

1. it terminates;
2. the precondition of local consistency is verified at each iteration provided that it starts with a pair of stores τ and τ'_{k_0} satisfying $P(\tau, \tau'_{k_0}, \tau'_{k_0}, W, I)$ for all I in $\{I \mid \langle z, I \rangle \in W\}$;
3. If $\tau'_{k_j}(\mathcal{I}_{k-1})$ holds at the entry of iteration j , then $\tau'_{k_{j+1}}(\mathcal{I}_{k-1})$ holds at the exit of iteration j ($0 \leq j \leq |W|$).

Proof Each iteration of the loop removes a pair from W . No pairs are added to W during this round hence the loop terminates. Let us now show, by induction on W , that the precondition of local consistency remains true at each iteration of the loop.

1. Base: On entry of the loop we know by hypothesis that $P(\tau, \tau'_{k_0}, \tau'_{k_0}, W \cup \{\langle z, I \rangle\}, I)$ holds for all combinations of $\langle z, I \rangle$ in W and I in $\{I \mid \langle z, I \rangle \in W\}$.
2. Induction: Let us assume that $P(\tau, \tau'_{k_j}, \tau'_{k_j}, W \cup \{\langle z, I \rangle\}, I)$ holds at iteration j . Let us further assume that $\tau'_{k_j}(\mathcal{I}_{j-1})$ holds at the beginning of the iteration. Given that propagate is locally consistent, the propagation of $\langle z, I \rangle$ yields a store $\tau'_{k_{j+1}}$ satisfying the three properties of local consistency. Instruction (9) removes the propagated pair

$\langle z, I \rangle$ from W , hence $P(\tau, \tau'_{k_j}, \tau'_{k_{j+1}}, W, I)$ is identical to $P(\tau, \tau'_{k_{j+1}}, \tau'_{k_{j+1}}, W, I)$ because of property p_2 . Furthermore, $t(\text{def}(I)) > k$, and this change cannot affect P for any J in $\{I | \langle z, I \rangle \in W\}$. Given that the two stores are identical for all variables but $\text{def}(I)$, we can safely conclude that $\tau'_{k_{j+1}}(\mathcal{I}_{k-1})$ also holds since all the invariants in \mathcal{I}_{k-1} use variables with topological value strictly smaller than k . Therefore, point (2) and (3) are proven. At each iteration the queue produced by propagate contains all the pair $\langle z, I \rangle$ for which z has changed and appears in $\text{exp}(I)$ and is merged back into the main queue, thus enforcing the correctness of property (ii).

□

Theorem 1 The algorithm in Figure 6.1 is correct, i.e., given a subset of modified variables $\mathcal{M} \subseteq \mathcal{S}$ and a pair of computation states τ and τ' satisfying

$$\begin{aligned} \tau'(x) &\neq \tau(x) & \forall x \in \mathcal{M} \\ \tau'(x) &= \tau(x) & \forall x \in \mathcal{V} \setminus \mathcal{M} \\ \tau(\mathcal{I}) &\text{ holds} \end{aligned}$$

it produces a new computation state τ'' satisfying

$$\begin{aligned} \tau''(x) &= \tau'(x) & \forall x \in \mathcal{M} \\ \tau''(\mathcal{I}) &\text{ holds} \end{aligned}$$

Proof The induction proof must show that the execution of lines 5 to 11 establishes properties (i), (ii) and (iii) when the execution reaches program point (4). First, let us establish that the properties hold true the first time the algorithm reaches (4).

1. Base: Since, for all the variables $x \in \mathcal{S}$, $t(x) = 0$, the initialization pushes onto Q the set $\{\langle x, I \rangle \mid x \in \mathcal{M} \wedge I \in \text{invariants}(x, \mathcal{I})\}$. Without loss of generality, we can assume that $i = 1$ at program point (4) since a non-empty queue implies that an invariant variable depends on a state variable of topological rank 0. Clearly, \mathcal{I}_0 is an empty set and property (i) trivially holds. Since $\mathcal{I}_0 = \emptyset$, we have that $\mathcal{I}_0^c = \mathcal{I}$ and the queue initialization enforces property (ii) since \mathcal{M} is the set of modified variables. Finally property (iii) also holds, since τ'_1 is τ' .
2. Induction: Assume that all three properties hold for step $k - 1$. Let us show that, once the program reaches the program point (4) again, the properties hold for step k . Step k consists of the propagation of all the pairs $\langle x, y := e \rangle$ in Q with $t(y) = k$. The

instruction at line (4) builds W as the subset of Q with $t(I) = k$. The instructions (6) to (11) are responsible for computing step k . Since conditions (ii) and (iii) hold, we know that the hypothesis of Lemma 1 are satisfied. Indeed, W contains all the pairs $\langle x, I \rangle$ with $t(I) = k$, and hence the function G within P selects $\tau(x)$ (the old value) for x and the invariants are automatically consistent (we evaluate everything in the old state).

Therefore, by Lemma 1, the loop terminates, maintains the precondition of local consistency throughout and makes sure that property (i) holds on exit of the loop since it holds on entry. Once W is empty, $P(\tau, \tau'_{k_n}, \tau'_{k_{n+1}}, \emptyset, I)$ holds for all I in step k . Given that

$$\begin{aligned} \mathcal{I}_k &= \mathcal{I}_{k-1} \cup \{I \in I \mid t(\text{def}(I)) = k\}, \\ P(\tau, \tau'_{k_n}, \tau'_{k_{n+1}}, \emptyset, I) &\text{ holds, and} \\ \tau'_k(\mathcal{I}_{k-1}) &\text{ holds,} \end{aligned}$$

we have established that $\tau'_{k_{n+1}}(\mathcal{I}_k)$ holds. By choosing $\tau'_{k+1} = \tau'_{k_{n+1}}$ at the end of step k , the property (i) holds again. Note that condition (ii) is trivially satisfied by definition of Q' and property p_2 of Lemma 1. Property (iii) is satisfied as a direct consequence of p_2 .

Once Q is empty, the last computation state produced is simply τ'' and, since the induction hypothesis states that $\tau_{k+1}(\mathcal{I}_k)$ holds, the theorem follows. \square

For completeness, it is necessary to prove that the implementation of each elementary invariant satisfies the local consistency properties defined earlier. The next lemma establishes this fact for the summation aggregate. Other invariants can be proven correct similarly.

Lemma 2 The routine in Figure 6.2 is locally consistent for the elementary invariant

$$x := \text{sum}(y_1, \dots, y_n)$$

Given a queue $W \cup \langle z, I \rangle$ for some z in $\{y_1, \dots, y_n\}$ and a pair of stores τ, τ'_k , it produces a store τ'_{k+1} satisfying

$$\begin{aligned} \tau'_k(x) &= \text{sum}(G(\tau, \tau'_k, W \cup \langle z, I \rangle, I)(y_1), \dots, G(\tau, \tau'_k, W \cup \langle z, I \rangle, I)(y_n)) \\ \tau'_{k+1}(x) &= \text{sum}(G(\tau, \tau'_k, W, I)(y_1), \dots, G(\tau, \tau'_k, W, I)(y_n)) \end{aligned}$$

Proof Let z^c denote $\tau'_k(z)$ and z^o denote $\tau(z)$ in the propagation routine. The input to a call is a pair $\langle z, I \rangle$ and the routine computes $z^c - z^o$ to add it to x^c . By definition of G we

have

$$\begin{aligned} G(\tau, \tau'_k, W \cup \langle z, I \rangle, I)(z) &= \tau(z) \\ G(\tau, \tau'_k, W, I)(z) &= \tau'_k(z) \end{aligned}$$

Before the assignment x^c is $\tau'_k(x)$ (by hypothesis), hence after the assignment x^c becomes

$$\begin{aligned} &G(\tau, \tau'_k, W \cup \langle z, I \rangle, I)(y_1) + \dots + \\ &G(\tau, \tau'_k, W \cup \langle z, I \rangle, I)(z) + \dots + \\ &G(\tau, \tau'_k, W \cup \langle z, I \rangle, I)(y_n) + \tau'_k(z) - \tau(z). \end{aligned}$$

By the commutativity and associativity laws of addition this expression simplifies to

$$G(\tau, \tau'_k, W \cup \langle z, I \rangle, I)(y_1) + \dots + \tau'_k(z) + \dots + G(\tau, \tau'_k, W \cup \langle z, I \rangle, I)(y_n)$$

which is identical to

$$G(\tau, \tau'_k, W \cup \langle z, I \rangle, I)(y_1) + \dots + G(\tau, \tau'_k, W, I)(z) + \dots + G(\tau, \tau'_k, W \cup \langle z, I \rangle, I)(y_n).$$

Since for all terms of the form $G(\tau, \tau'_k, W \cup \langle z, I \rangle, I)(y_k)$ we have $y_k \neq z$ (by definition of *sum*), it follows that $G(\tau, \tau'_k, W \cup \langle z, I \rangle, I)(y_k) = G(\tau, \tau'_k, W, I)(y_k)$. Thus, x^c can be rewritten as

$$G(\tau, \tau'_k, W, I)(y_1) + \dots + G(\tau, \tau'_k, W, I)(z) + \dots + G(\tau, \tau'_k, W, I)(y_n).$$

It is now easy to set $\tau'_{k+1}(x)$ to x^c and $\tau'_{k+1}(y) = \tau'_k(y)$ for all $y \neq x$. The lemma follows.

□

6.3 Dynamic Invariants

Static invariants are attractive since the planning phase can be performed entirely at compile time. However, there are interesting applications in the areas of scheduling and resource allocation where sets of invariants are not static. This section introduces dynamic invariants to broaden the class of invariants accepted by LOCALIZER. Dynamic invariants are updated by a series of planning/execution phases where the planning phase takes place at execution time. As a consequence, they generalize the technique of finite differencing proposed in [45].

6.3.1 Motivation

The main restriction of static invariants comes from the invariant

$$x := \text{element}(e, y_1, \dots, y_n).$$

The static topological constraint for this invariant is

$$t(x) = \max(t(e), t(y_1), \dots, t(y_n)) + 1$$

and it prevents LOCALIZER from accepting expressions where some elements of an array may depend on some other elements of the same array. This constraint is strong, because the value of e is not known at compile time. In fact, it may not even be known before the start of the execution phase since some invariants may update it.

However, there are many applications in scheduling or resource allocation where such invariants occur naturally. For instance, a scheduling application may be modeled in terms of an invariant

$$\text{start}[3] := \max(\text{end}[\text{prec}[3]], \text{end}[\text{disj}[3]]);$$

where $\text{start}[i]$ represents the starting date of task i , $\text{prec}[i]$ represents the predecessor of the task in the job and $\text{disj}[i]$ represents the predecessor of task i in the disjunction. Variable $\text{disj}[i]$ is typically updated during the local search and the above invariant is normalized into a set of the form:

$$\begin{aligned} \text{start}_3 &:= \max(p, d) \\ p &:= \text{element}(\text{prec}_3, \text{end}_1, \dots, \text{end}_n) \\ d &:= \text{element}(\text{disj}_3, \text{end}_1, \dots, \text{end}_n) \end{aligned}$$

Of course, such an application has also invariants of the form

$$\begin{aligned} \text{end}_1 &:= \text{start}_1 + \text{duration}_1 \\ &\vdots \\ \text{end}_3 &:= \text{start}_3 + \text{duration}_3 \\ &\vdots \\ \text{end}_n &:= \text{start}_n + \text{duration}_n \end{aligned}$$

implying that the resulting set of invariants is not static.

6.3.2 Overview of the Approach

The basic idea behind dynamic invariants is to evaluate the invariants by levels. Each invariant is associated with one level and, inside one level, the invariants are static. Once a level is completed, planning of the next level can take place using the values of the previous level since lower levels are never reconsidered. With this computation model in mind, the topological constraint associated with an invariant

$$x := \text{element}(e, y_1, \dots, y_n)$$

can be reconsidered. The basic idea is to require that e be evaluated before x (i.e., the level of x is the level of $e + 1$). Once e is updated, then it is easy to find a weaker topological constraint since the value of e is known. The invariant can be simplified to

$$x := y_{e^c}$$

and x only depends on one element in $\{y_1, \dots, y_n\}$. The planning phase is thus divided in two steps. A first step, which can be carried out at compile time, partitions the invariants in levels. The second step, which is executed at runtime, topologically sorts the invariants within each level whenever the invariants at the lower level have been propagated.

6.3.3 Formalization

The basic intuition is formalized in terms of two assignments $l : \mathcal{V} \rightarrow \mathcal{N}$ and $t : \mathcal{V} \rightarrow \mathcal{N}$ and their sets of topological constraints.

Definition 12 The level constraints associated with an invariant I , and denoted by $lc(I)$, are defined as follows:

$$\begin{aligned} lc(x := c) &= \{l(x) = 0\} \\ lc(x := y \oplus z) &= \{l(x) = \max(l(y), l(z))\} \\ lc(x := \prod(x_1, \dots, x_n)) &= \{l(x) = \max(l(x_1), \dots, l(x_n))\} \\ lc(x := \text{element}(e, x_1, \dots, x_n)) &= \{l(x) = \max(l(e) + 1, l(x_1), \dots, l(x_n))\} \end{aligned}$$

The level constraints are not strong except for the invariant `element` where the level of x is strictly greater than the level of e . Informally, it means that e must be evaluated in an earlier phase than x .

Definition 13 The level constraints associated with a set of invariants \mathcal{I} and denoted by $l(\mathcal{I})$ is simply $\bigcup_{I \in \mathcal{I}} lc(I)$.

Definition 14 A set of invariants \mathcal{I} is *serializable* if there exists an assignment $l : \mathcal{V} \rightarrow \mathcal{N}$ satisfying $lc(\mathcal{I})$.

A serializable set of invariants can be partitioned into a sequence $\langle \mathcal{I}_0, \dots, \mathcal{I}_p \rangle$ such the invariants in \mathcal{I}_i have level i . This serialization can be performed at compile-time. Note that the serialization $l : \mathcal{V} \rightarrow \mathcal{N}$ of a set of invariants \mathcal{I} is unique (if it exists). Indeed, by definition a variable x cannot appear in $def(I_i)$ and $def(I_j)$ for $i \neq j$, hence its level assignment is functionally dependent on the right hand side of the topological constraint. Note that a state variables x is assigned a level $l(x) = 0$ since it does not depend on anything. The second step consists of ordering the invariants inside each partition. This ordering can only take place at runtime, since it is necessary to know the values of some invariants to simplify the *element* invariants.

Definition 15 Let τ be a computation state and let $\tau(x)$ denote the value of x in τ . The topological constraints associated with an invariant I with respect to τ , denoted $tc(I, \tau)$, is defined as follows:

$$\begin{aligned} tc(x := c, \tau) &= \{t(x) = 0\} \\ tc(x := y \oplus z, \tau) &= \{t(x) = \max(t(y), t(z)) + 1\} \\ tc(x := \prod(x_1, \dots, x_n), \tau) &= \{t(x) = \max(t(x_1), \dots, t(x_n)) + 1\} \\ tc(x := element(e, x_1, \dots, x_n), \tau) &= \{t(x) = t(x_{\tau(e)}) + 1\} \end{aligned}$$

Definition 16 The topological constraints associated with a set of invariants \mathcal{I} with respect to a state τ , denoted by $tc(\mathcal{I}, \tau)$, is simply $\bigcup_{I \in \mathcal{I}} tc(I, \tau)$.

Definition 17 A set of invariants \mathcal{I} is static with respect to a state τ if there exists an assignment $t : \mathcal{V} \rightarrow \mathcal{N}$ satisfying $tc(\mathcal{I}, \tau)$.

The main novelty of course is in the invariant *element* where the topological constraint can ignore e since its value is known. In addition, since the final value of e is known, the topological constraints can be made precise since the element y_{e^c} that x depends upon is known.

Definition 18 Let τ_0 be a computation state. A set of invariants \mathcal{I} is dynamic with respect to τ_0 if

1. \mathcal{I} is serializable and can be partitioned into a sequence $\langle \mathcal{I}_0, \dots, \mathcal{I}_p \rangle$;
2. \mathcal{I}_i is static with respect to τ_i where τ_i ($i > 0$) is the state obtained by propagating the invariants \mathcal{I}_{i-1} in τ_{i-1} .

```

procedure execute( $\mathcal{I}, \mathcal{M}$ )
begin
   $\langle \mathcal{I}_0, \dots, \mathcal{I}_p \rangle := \text{serialize}(\mathcal{I});$ 
  for ( $i := 0; i \leq p; i++$ ) do
     $t := \text{plan}(\mathcal{I}_i);$ 
    execute( $\mathcal{I}_i, \mathcal{M}, t$ );
  endfor
end

```

Figure 6.3: The Execution Algorithm for Dynamic Invariants

Of course, dynamic invariants cannot be recognized at compile-time and may produce an execution error at runtime when LOCALIZER is planning a level. Note that the propagation of a dynamic invariant like `element` requires $\Theta(1)$ in time and space.

6.3.4 The Execution Algorithm

The new execution algorithm is a simple generalization of the static algorithm and is shown in Figure 6.3. Note the planning step that is called for each level. Note that the correctness of this algorithm is a direct consequence of the correctness of the static algorithm since each level is considered in sequence, independently of the subsequent levels.

6.4 Set Invariants

As should be clear from previous chapters, LOCALIZER offers rich set invariants and this section reviews their implementation in detail. Set invariants are implemented differently according to their complexity and our implementation classifies sets in three categories: extensional, semi-intentional and fully-intentional. The rest of this section is organized along this classification. Section 6.4.1 reviews preliminaries. Section 6.4.2 presents the basic set operations. Sections 6.4.3, 6.4.4 and 6.4.5 describe the implementation of set invariants for the three categories.

6.4.1 Extensional and Intentional Set Invariants

In this section, we mainly consider set invariants of the form

$$x : \{T\} := \{e_1, \dots, e_n\};$$

and

$$x : \{ T \} := \{ v : T \mid \text{select } v \text{ from } S \text{ where } E \};$$

Invariants based on nested selection can be handled with chains of cross products and are discussed after the basic case. As mentioned, set invariants are classified as: extensional, semi-intentional and fully intentional. Invariants of the form

$$x : \{ T \} := \{ e_1, \dots, e_n \};$$

are always extensional. Extensional set invariants select their elements from a constant set; semi-intentional set invariants select their elements from a set invariant and filter them with a constant expression; and fully-intentional sets select their elements from a set invariant and filter them with an expression defined in terms of at least one variable.

Definition 19 A set invariant $x : \{ T \} := \{ v : T \mid \text{select } v \text{ from } S \text{ where } E \}$ is extensional if $\text{vars}(S) = \emptyset$.

Definition 20 A set invariant $x : \{ T \} := \{ v : T \mid \text{select } v \text{ from } S \text{ where } E \}$ is semi-intentional if $\text{vars}(S) \neq \emptyset \wedge \text{vars}(E) = \emptyset$.

Definition 21 A set invariant $x : \{ T \} := \{ v : T \mid \text{select } v \text{ from } S \text{ where } E \}$ is fully intentional if $\text{vars}(S) \neq \emptyset \wedge \text{vars}(E) \neq \emptyset$.

6.4.2 Sets as Abstract Data Types

In the LOCALIZER implementation, sets can be viewed as abstract data types supporting the following operations

INSERT(S, e) : Destructively assigns S to $S \cup \{e\}$. The operation executes in $O(1)$ average time.

REMOVE(S, e) : Destructively assigns S to $S \setminus \{e\}$. The operation executes in $O(1)$ average time.

UNION(S_1, S_2) : Destructively assigns S_1 to $S_1 \cup S_2$. The operation executes in $O(|S_2|)$ average time.

UNION(S_1, \dots, S_n) : Destructively assigns S_1 to $\cup_{i \in \{1..n\}} S_i$. The operation executes in $O(\sum_{i \in \{1..n\}} |S_i|)$ average time.

In addition, it is convenient to assume the existence of a non-primitive operation

APPLY(S, Δ) : Destructively update S by applying all the operations that appear in the list Δ . Each operation is represented as a tuple $\langle \alpha, \beta \rangle$ where α denotes an operation and is drawn from $\{\text{INSERT}, \text{REMOVE}, \text{UNION}\}$ and β is the second operand of the operation. More formally

$$\begin{aligned} S &= \text{APPLY}(S, (\langle \delta_1, e_1 \rangle, \dots, \langle \delta_n, e_n \rangle)) \text{ where} \\ T_0 &= S \\ T_1 &= \delta_1(T_0, e_1) \\ &\vdots \\ S &= \delta_n(T_{n-1}, e_n) \end{aligned}$$

APPLY executes on average, in a time proportional to $O(\sum_{\delta_i \in \Delta} t(\delta_i))$ where Δ is the difference list and $t(\delta_i)$ is the time it takes to execute operation δ_i , i.e., $O(1)$ for insertions or removals and $O(|S_i|)$ for unions.

As was the case for traditional invariants, the variable x associated with a set invariant maintains two values x^c and x^o that represent its current value and its value at the end of last propagation stage. Once again, $x^c = x^o$ at the beginning of a propagation and x^c incorporates the changes that occur in the propagation algorithm. It is useful to assume the existence of an operation $\text{DIFF}(x)$ that returns the set of operations Δ such that

$$x^c = \text{APPLY}(x^o, \Delta).$$

6.4.3 Extensional Sets

We now consider the implementation of extensional set invariants. As before, the implementation is presented in terms of the normalization, the planning phase and the execution phase.

6.4.3.1 Normalization

An invariant of the form

$$x := \{e_1, \dots, e_n\};$$

is normalized into the set of elementary invariants

$$\left\{ \begin{array}{l} x := \text{UNION}(s_1, \dots, s_n) \\ s_1 := \text{SINGLETON}(x_1) \\ \vdots \\ s_n := \text{SINGLETON}(x_n) \end{array} \right\}$$

assuming that the x_1, \dots, x_n are variables, i.e., the innermost expressions have been normalized already. The obvious strategy to normalize a set in extension is thus to normalize the expressions e_i ($1 \leq i \leq n$) first to obtain auxiliary variables x_1, \dots, x_n and to apply the above transformation.

The normalization of an invariant

$$x = \{v \mid \text{select } v \text{ from } \{e_1, \dots, e_n\} \text{ where } E\}$$

consists of normalizing $E[v/e_1], \dots, E[v/e_n]$ to obtain b_1, \dots, b_n and of generating the set of invariants

$$\left\{ \begin{array}{l} x := \text{UNION}(s_1, \dots, s_n), \\ b_1 := E[v/e_1]; s_1 := \text{CSINGLETON}(b_1, e_1) \\ \vdots \\ b_n := E[v/e_n]; s_n := \text{CSINGLETON}(b_n, e_n) \end{array} \right\}$$

where $\text{CSINGLETON}(b, v)$ returns $\{v\}$ when b is true and \emptyset when b is false. This set can be turned into a set of elementary invariants by normalizing $E[v/e_i]$ ($1 \leq i \leq n$).

6.4.3.2 Planning

The planning phase can be extended to take into account the new elementary invariants. The UNION invariant does not need any special treatment and obeys the same set of static and level constraints as the other operations covered by Π .

Definition 22 The level constraints associated with an invariant I and denoted by $lc(I)$ are defined as follows:

$$\begin{aligned} lc(x := \text{SINGLETON}(y)) &= \{l(x) = l(y)\} \\ lc(x := \text{CSINGLETON}(b, y)) &= \{l(x) = \max(l(b), l(y))\} \end{aligned}$$

Definition 23 Let τ be a computation state and let $\tau(x)$ denote the value of x in τ . The topological constraint associated with an invariant I with respect to τ , denoted $tc(I, \tau)$, is defined as follows:

$$\begin{aligned} tc(x := \text{SINGLETON}(y), \tau) &= \{t(x) = t(y) + 1\} \\ tc(x := \text{CSINGLETON}(b, y), \tau) &= \{t(x) = \max(t(b), t(y)) + 1\} \end{aligned}$$

```

procedure propagate( $y, x := \text{SINGLETON}(y), \mathcal{I}, Q$ )
begin
    REMOVE( $x^c, y^o$ );
    INSERT( $x^c, y^c$ );
     $Q := \text{invariants}(\mathcal{I}, x)$ ;
end

procedure propagate( $b, x := \text{CSINGLETON}(b, e), \mathcal{I}, Q$ )
begin
    if  $b^c$  then INSERT( $x^c, e$ );
    else REMOVE( $x^c, e$ );
     $Q := \text{invariants}(\mathcal{I}, x)$ ;
end

procedure propagate( $y_i, x := \text{UNION}(y_1, \dots, y_n), \mathcal{I}, Q$ )
begin
    APPLY( $x^c, \text{DIFF}(y_i)$ );
    if  $|\text{DIFF}(x)| > 0$  then
         $Q := \text{invariants}(\mathcal{I}, x)$ ;
end

```

Figure 6.4: Propagation Routines for Extensional Sets.

6.4.3.3 Execution

The propagation of the new elementary invariants is similar to what has been described in the context of arithmetic invariants. The propagation procedure is scheduled in accordance to the topological and the level assignments that satisfy lc and tc . This guarantees that a given pair $\langle \text{variable}, \text{invariant} \rangle$ is propagated at most once. Figure 6.4 shows the definition of the propagation rules for these invariants. Note that the propagation `SINGLETON` and `CSINGLETON` require $O(1)$ in time given that both the insertion and the removal of an element is done in $O(1)$ and at most two operations (insertion or removal) is executed in the propagation. They also both occupy $\Theta(1)$ in space given that the invariant does not need extra memory. Each propagation of the `UNION` invariant costs $O(|\text{DIFF}(y_i)|)$ in time given that the incremental complexity of union is the complexity of `APPLY` and the space requirement is $\Theta(n)$ since the invariants must keep references to the n sets y_1, \dots, y_n .

6.4.3.4 Dealing with Sets of Tuples

Sets of tuples do not raise any particular difficulty. The only requirement is to unfold the sequence of primitive select expressions and to recombine the outputs of each elementary invariant with cross products. More details on this process are given in the context of

semi-intentional sets.

6.4.4 Semi-Intentional Sets

Semi-intentional sets are defined by invariants of the form

$$x = \{v \mid \text{select } v \text{ from } S \text{ where } E\}$$

where S contains variables and E is a variable free expression. Once again, the objective is to devise a planning/execution model that leads to an effective implementation. Ideally, the algorithm should be linear in the size of the minimal changes on the output or, more precisely, in the minimal number of tuples added to, and removed from, the destination set.

6.4.4.1 Normalization

Sets of scalars: Consider first an invariant

$$x = \{v \mid \text{select } v \text{ from } S \text{ where } E\}$$

The normalization process described for static sets does not apply, since S is not known when the invariants are normalized. As a consequence, it is necessary to introduce another elementary invariant

$$x := \text{FILTER}(y, F).$$

The elementary invariant `FILTER` expects a set as first argument and a function $F : T \rightarrow \text{bool}$ as second argument, where T is the type of the elements of y . It returns the set of elements v in y such that $F(v)$ holds.

The normalization of a semi-intentional set invariant consists of normalizing S to obtain a variable y , to transform the expression E into a partial function F , and to generate the set of elementary invariants

$$\{y := S; x := \text{FILTER}(y, F)\}.$$

Sets of Tuples: The normalization of an invariant of the form

$$x = \{ \langle v_1, \dots, v_n \rangle \mid \begin{array}{l} \text{select } v_1 \text{ from } S_1 \text{ where } E_1 \\ \vdots \\ \text{select } v_n \text{ from } S_n \text{ where } E_n \end{array} \}$$

is based on a sequence of simple semi intentional invariants interleaved with a sequence of cross products. This transformation leads to the set

$$\left\{ \begin{array}{lll} y_1 = S_1; & \delta_1 := y_1; & \sigma_1 := \text{FILTER}(\delta_1, F_1); \\ y_2 = S_2; & \delta_2 := \text{CROSS}(\sigma_1, y_2); & \sigma_2 := \text{FILTER}(\delta_2, F_2); \\ \vdots & & \\ y_n := S_n; & \delta_n := \text{CROSS}(\sigma_{n-1}, y_n); & x := \text{FILTER}(\delta_n, F_n) \end{array} \right\}$$

where each function F_i is derived from the expression $P_i(v_1, \dots, v_i)$ and the y_i are normalized from the corresponding S_i . This normalization introduces a new elementary invariant **CROSS** that takes two sets as input, and produces the Cartesian product of its inputs.

6.4.4.2 Planning

The definition for the lc set is extended as follows

Definition 24 The level constraints associated with an invariant I , and denoted by $lc(I)$, are defined as follows:

$$\begin{aligned} lc(x := \text{CROSS}(y, z)) &= \{l(x) = \max(l(y), l(z))\} \\ lc(x := \text{FILTER}(y, F)) &= \{l(x) = l(y)\} \end{aligned}$$

The topological constraints are also extended in the obvious way.

Definition 25 Let τ be a computation state and let $\tau(x)$ denote the value of x in τ . The topological constraints associated with an invariant I with respect to τ , denoted $tc(I, \tau)$, is defined as follows:

$$\begin{aligned} tc(x := \text{CROSS}(y, z), \tau) &= \{t(x) = \max(t(y), t(z)) + 1\} \\ tc(x := \text{FILTER}(y, F), \tau) &= \{t(x) = t(y) + 1\} \end{aligned}$$

6.4.4.3 Execution

The propagation of the new elementary invariants is based upon the differential structure and the pair of values x^c and x^o . The propagation procedure for $x := \text{FILTER}(y, F)$ loops through all the differences for the set variable y . In case of an element removal, the operation is applied to x without further ado. In case of an element insertion, the insertion in x is conditional to the result of $F(e)$ where e is the operand of the insertion.

The implementation for $x := \text{CROSS}(y, z)$ obtains the list of differences for the set variable responsible for the propagation. It then performs, for each difference, the corresponding

operation on all the partial tuples resulting from the cross product between the value and the second operand of CROSS. Note that, depending on the nature of the operation, the operand of the cross product is either the current value of z or the old one. In the event z changes, the propagation procedure has a symmetric counterpart. Finally, note that if the two operands y and z of CROSS change, the two pairs $\langle y, x := \text{CROSS}(y, z) \rangle$, $\langle z, x := \text{CROSS}(y, z) \rangle$ are propagated in any order. The complexity of this procedure for the propagation of the pair $\langle y, x := \text{CROSS}(y, z) \rangle$ is $O(|\text{DIFF}(y)| \cdot |z|)$. Figure 6.5 depicts the implementation of the two procedures.

```

procedure propagate( $y, x := \text{FILTER}(y, F), \mathcal{I}, Q$ )
begin
  forall  $\langle \delta, e \rangle \in \text{DIFF}(y)$  do
    switch( $\delta$ ) do
      case REMOVE:  $x^c := \text{REMOVE}(x^c, e)$ ;
      case INSERT: if  $F(e)$  then  $x^c := \text{INSERT}(x^c, e)$ ;
    if  $|\text{DIFF}(x)| > 0$  then
       $Q := \text{invariants}(\mathcal{I}, x)$ ;
    end

  procedure propagate( $y, x := \text{CROSS}(y, z), \mathcal{I}, Q$ )
  begin
    forall  $\langle \delta, e \rangle \in \text{DIFF}(y)$  do
      forall  $\alpha \in z^g$  do
         $x^c := \delta(x^c, \langle e, \alpha \rangle)$ ;
      if  $|\text{DIFF}(x)| > 0$  then
         $Q := \text{invariants}(\mathcal{I}, x)$ ;
      end
  end

```

Figure 6.5: Propagation Procedures for Semi-intentional Sets.

6.4.4.4 Correctness

To show that the propagation routines for FILTER and CROSS are correct, it is necessary to demonstrate that the implementation verifies the local consistency conditions described earlier. Remember that τ is the computation state before the step starts, τ' is identical to τ but for a subset \mathcal{M} of the input variables to the invariant that are modified and τ'' is the output state.

Lemma 3 The routine for FILTER shown in Figure 6.5 is locally consistent for the elementary invariant $x := \text{FILTER}(y, F)$.

Proof Before the propagation start, we know that $\tau(x) = \{e \in \tau(y) | F(e)\}$. The routine can be called at most once since y is the only variable involved. The procedure iterates over the difference list of y . Since the difference list can be expressed as $A \cup D$ where A is the set of elements added to $\tau(y)$ and D is the set of elements removed from $\tau(y)$ ($A \cap D = \emptyset$), the final computation state $\tau'_1(x)$ stores $\tau(x) \setminus D \cup \{e \in A | F(e)\}$. Since the procedure does not modify any other element in the store, the other properties of local consistency are automatically enforced. \square

Lemma 4 The routine for CROSS shown in Figure 6.5 is locally consistent for the elementary invariant $x := \text{CROSS}(y, z)$.

Proof Without loss of generality, let us assume that $t(x) = k$. Before the propagation, $\tau'_k(x) = \{\langle e, f \rangle \in \tau(y) \times G(\tau, \tau'_k, W \cup \{\langle y, I \rangle\}, I)(z)\}$. Let us prove local consistency for the propagation of the pair $\langle y, I \rangle$ where I is $x := \text{CROSS}(y, z)$. Let us denote by τ'_{k_1} the computation state after the propagation. When y changes, the computation state $\tau'_k(y)$ contain the new value for y . It is easy to relate the two states as $\tau(y) = \tau'_k(y) \setminus A_y \cup D_y$ where A_y is the set of additions to y and D_y is the set of deletion from y . By the distributivity of cross product over set union and difference, we can rewrite $\tau(x)$ as

$$\begin{aligned} \tau(x) = & \{\langle e, f \rangle \in \tau'_k(y) \times G(\tau, \tau'_k, W \cup \{\langle y, I \rangle\}, I)(z)\} \cup \\ & \{\langle e, f \rangle \in D_y \times G(\tau, \tau'_k, W \cup \{\langle y, I \rangle\}, I)(z)\} \setminus \\ & \{\langle e, f \rangle \in A_y \times G(\tau, \tau'_k, W \cup \{\langle y, I \rangle\}, I)(z)\} \end{aligned}$$

and hence

$$\begin{aligned} \tau'_{k_1}(x) = & \tau(x) \setminus \\ & \{\langle e, f \rangle \in D_y \times G(\tau, \tau'_k, W \cup \{\langle y, I \rangle\}, I)(z)\} \cup \\ & \{\langle e, f \rangle \in A_y \times G(\tau, \tau'_k, W \cup \{\langle y, I \rangle\}, I)(z)\}. \end{aligned}$$

This last expression is the result of the computation carried out by the routine. Indeed, the routine echoes the operation (addition or deletion) and uses as a store z^g defined as $G(\tau, \tau'_k, W \cup \{\langle y, I \rangle\}, I)(z)$. Therefore, if the pair $\langle z, I \rangle$ was already propagated, it does not belong to W , hence $G(\tau, \tau'_k, W \cup \{\langle y, I \rangle\}, I)(z)$ is $\tau'_k(z)$. If the pair is still in W , $G(\tau, \tau'_k, W \cup \{\langle y, I \rangle\}, I)(z)$ is $\tau(z)$ and the behavior is correct. Local consistency follows since the procedure does not modify anything besides $\tau'(x)$. \square

6.4.5 Fully Intentional Sets

The third and last category of sets deals with fully intentional set expressions. Consider a set invariant

$$x = \{v \mid \text{select } v \text{ from } S \text{ where } E\}.$$

Variables can appear both in S and in the filter expression E . The implementation of the semi-intentional sets cannot be adapted easily to the context. The difficulty arises from the presence of variables in the filter that makes the insertion or the deletion of an element contingent to the values of these variables.

6.4.5.1 Normalization

To formalize the normalization, it is important to recognize that the function F to be used as an argument of the invariant is no longer a function of the parameter v , but also of all variables appearing in E . As a consequence, assuming that $\text{vars}(E) = \{y_1, \dots, y_n\}$ and that the types of these variables are T_1, \dots, T_n respectively, the function has the signature

$$F : (T_1 \times \dots \times T_n) \rightarrow T \rightarrow \text{bool}$$

In the rest of this section we use the following conventions and notations. If $\text{vars}(E) = \{y_1, \dots, y_n\}$ then $F^o(v) = F(y_1^o, \dots, y_n^o)(v)$ and $F^c(v) = F(y_1^c, \dots, y_n^c)(v)$. Additionally, y denotes the set variable resulting from the normalization of the expression S .

Set of Scalars Fully-intentional set are implemented through a new elementary invariant

$$x := \text{DFILTER}(y, F)$$

where y is the result of the normalization of S and F is the partial function described above. The normalization of a set

$$x = \{v \mid \text{select } v \text{ from } S \text{ where } E\}$$

then reduces to the normalization of S into a variable y , the transformation of E into a function $F : (T_1 \times \dots \times T_n) \rightarrow T \rightarrow \text{bool}$ and the creation of a set of elementary invariants

$$\{y := S, x := \text{DFILTER}(y, F)\}.$$

Sets of Tuples The normalization of an invariant of the form

$$x = \{ \langle v_1, \dots, v_n \rangle \mid \begin{array}{l} \text{select } v_1 \text{ from } S_1 \text{ where } E_1 \\ \vdots \\ \text{select } v_n \text{ from } S_n \text{ where } E_n \end{array} \}$$

is based on a sequence of simple intentional invariants interleaved with a sequence of cross products. This transformation leads to the set

$$\left\{ \begin{array}{lll} y_1 = S_1, & \delta_1 := y_1, & \sigma_1 := \text{DFILTER}(\delta_1, F_1), \\ y_2 = S_2, & \delta_2 := \text{CROSS}(\sigma_1, y_2), & \sigma_2 := \text{DFILTER}(\delta_2, F_2), \\ \vdots & & \\ y_n := S_n, & \delta_n := \text{CROSS}(\sigma_{n-1}, y_n), & x := \text{DFILTER}(\delta_n, F_n) \end{array} \right\}$$

6.4.5.2 Planning

The definition for the *lc* set is extended as follows

Definition 26 The level constraints associated with an invariant *I* and denoted by *lc(I)*, are defined as follows:

$$lc(x := \text{DFILTER}(y, F)) = \{l(x) = \max(l(y), \max_{v \in \text{vars}(F)} l(v))\}$$

The topological constraints are also extended in the obvious way.

Definition 27 Let τ be a computation state and let $\tau(x)$ denote the value of *x* in τ . The topological constraint associated with an invariant *I* wrt to τ , denoted *tc(I, τ)*, is defined as follows:

$$tc(x := \text{DFILTER}(y, F)) = \{t(x) = \max(t(y), \max_{v \in \text{vars}(F)} t(v)) + 1\}$$

6.4.5.3 Execution

This last section gives the definition of the propagation procedure for new elementary invariant $x := \text{DFILTER}(y, F)$. The skeleton of *DFILTER* proceeds by scanning the three following sets

1. The stable elements of *y* whose boolean conditions have changed are noted *DIFF(F)* and defined as $\{v \in y^c \cap y^o \mid F^c(v) \neq F^o(v)\}$.

2. The new elements of y whose boolean condition are true $\{v \in y^c \setminus y^o \mid F^c(v)\}$.
3. The old elements of y whose boolean condition were true $\{v \in y^o \setminus y^c \mid F^o(v)\}$.

```

procedure propagate( $y, x := \text{DFILTER}(y, F), \mathcal{I}, Q$ )
begin
  forall  $e \in \{s \in y^c \cap y^o \mid F^o(s) \neq F^c(s)\}$  do
    if  $F^c(e)$  then
       $x^c := \text{INSERT}(x^c, e);$ 
    else  $x^c := \text{REMOVE}(x^c, e);$ 
  forall  $e \in \{s \in y^c \setminus y^o \mid F^c(s)\}$  do
     $x^c := \text{INSERT}(x^c, e);$ 
  forall  $e \in \{s \in y^o \setminus y^c \mid F^o(s)\}$  do
     $x^c := \text{REMOVE}(x^c, e);$ 
  if  $|\text{DIFF}(x)| > 0$  then
     $Q := \text{invariants}(\mathcal{I}, x);$ 
end

```

Figure 6.6: Propagation Procedure for Fully-intentional Sets

In our implementation, the procedure described in Figure 6.6 is linear in

$$|\text{DIFF}(y)| + |\{s \in y^c \cap y^o \mid F^o(s) \neq F^c(s)\}|.$$

More precisely, the work accomplished by the procedure is proportional to the amount of change in the input, either directly through y , or induced through τ . This time complexity follows from the fact that `DFILTER` maintains the set $\{v \in y^c \cap y^o \mid F^c(v) \neq F^o(v)\}$ incrementally. The complexity bound directly follows from the procedure since it scans this set in the first loop and the set $\text{DIFF}(y)$ in the two other loops while performing a constant time operation at each iteration.

6.4.6 Open structures

The `DFILTER` invariant introduced in the previous section is a member of a more general class of elementary invariants. For instance, consider the excerpt of a `LOCALIZER` statement for vehicle routing

```

 $aas$ :  $\text{Arc} := \{ \langle q, w \rangle : \text{Arc} \mid \text{select } \langle q, w \rangle \text{ from } AP \text{ where } aaxv[q, w] \text{ and } notTabu[q] \};$ 
 $aa$ :  $\text{Arc} := \text{argmin}(a \text{ in } aas) \ aax[a.s, a.t];$ 
 $aag$ :  $\text{real} := - (\min(a \text{ in } aas) \ aax[a.s, a.t]);$ 

```

This fragment has the peculiarity to use the builtins `argmin` and `min` with an input set aas itself defined as an invariant. The $x := \prod (x_1, \dots, x_n)$ elementary invariant cannot deal with this statement since the list x_1, \dots, x_n is not known at compile time.

However, the infrastructure offered by DFILTER can be reused and an elementary invariant $x := \text{DMIN}(y, F)$ can easily accomodate such a statement. Indeed, it is easy to define a partial function $F : \text{vars}(E) \rightarrow T \rightarrow \mathfrak{R}$ that maps any element of type $\text{vars}(E)$ T and a value y to a performance measure. With the assistance of that partial function, a propagate procedure similar to DFILTER can be designed to maintain the minimum in a time proportional to

$$O((|\text{DIFF}(s)| + |\{s \in S^c \cap S^o \mid F^o(s) \neq F^c(s)\}|) \log(|y|))$$

All static elementary invariants covered by \square do have a dynamic counterpart.

6.5 Summary

This Chapter introduced a complete set of elementary invariants that correspond to the internal representation of high level invariants in LOCALIZER. The Chapter focused on the properties of each elementary invariant and on the incremental algorithms based on finite differencing that are used to update the invariant variables. The Chapter also presented the overall propagation routine that enforces a necessary topological ordering of the pairs $\langle x, I \rangle$ to propagate.

Chapter 7

Applications

This chapter illustrates the use of LOCALIZER on a number of applications. The applications considered are Boolean satisfiability, graph coloring, graph partitioning, Job-shop scheduling and vehicle routing. Boolean satisfiability is the simplest application (from a modeling standpoint) while vehicle routing is the most involved. For each application, one, or several, LOCALIZER statements are presented and compared with existing implementations. As a consequence, after this chapter, readers should have some reasonable idea of the expressiveness and efficiency of LOCALIZER.

7.1 Boolean Satisfiability

The first application is Boolean satisfiability (the first NP-complete problem) that has attracted much attention in the artificial intelligence community in recent years. We present very concise models of a greedy stochastic procedure called GSAT and proposed in [68]. Note that these models only use static invariants.

7.1.1 The Problem

Boolean satisfiability problems are described in terms of a number of clauses, each clause consisting of a number of literals. As is traditional, a literal is simply an atom (positive atom) or the negation of an atom (negative atom). The goal is to find an assignment of Boolean values to the atoms such that all clauses are satisfied, a clause being satisfied if one of its positive atoms is true or one of its negative atoms is false.

7.1.2 The Local Search Algorithm

The local search algorithm considered for Boolean satisfiability is GSAT, a greedy local search procedure. The basic idea is to start from a random assignment of Boolean values and to select the atom which, when its value is inverted, produces a state with the largest number of satisfied clauses. A transition is acceptable if it improves the number of satisfied clauses, or at the very least does not degrade that number. The algorithm described in the original paper can be found in Figure 7.1.

```

procedure GSAT
input:  a set of clauses  $F$ ,  $maxFlips$ ,  $maxTries$ 
output: a satisfying truth assignment of  $F$ , if found
begin
  for  $i := 1$  to  $maxTries$  do
     $T :=$  a randomly generated truth assignment
    for  $j := 1$  to  $maxFlips$  do
      if  $T$  satisfies  $F$  then
        return  $T$ ;
      Let  $k$  be the variable which, when flipped, leads to the
      largest increase in the number of satisfied clauses. If there
      is a tie, break randomly
       $T := T$  with the truth assignment of  $k$  reversed
    end
  end
  return "No satisfying assignment found";
end

```

Figure 7.1: The GSAT Algorithm of Selman et al.

7.1.3 A Simple LOCALIZER Statement

Instance Representation In the statement, atoms are represented by integers from 1 to n and a clause is represented by two sets: the set of its positive atoms p and the set of its negative atoms n . A SAT problem is simply an array of m clauses.

State Definition The state is specified by the truth values of the atoms and is captured in the array a , where $a[i]$ represents the truth value of atom i .

Neighborhood The neighborhood is defined as the set of boolean assignments differing from the current assignment in the value of a single atom.

```

Solve
Type:
  clause = record
    p : {int};
    n : {int};
  end;
Constant:
  m: int = ...;
  n: int = ...;
  cl: array[1..m] of clause = ...;
Variable:
  a: array[1..n] of boolean;
Invariant:
  nbt: array[ i in 1..m] of int = sum(i in cl[i].p) a[j] + sum(j in cl[i].n) !a[j];
  nbClauseSat : int = sum(i in 1..m) (nbt[i] > 0);
Satisfiable:
  nbClauseSat = m;
Objective Function:
  maximize nbClauseSat;
Neighborhood:
  move a[i] := !a[i]
  where i from {1..n}
  accept when improvement;
Start:
  forall(i in 1..n) a[i] := random({true,false});
Restart:
  forall(i in 1..n) a[i] := random({true,false});

```

Figure 7.2: A Local Improvement statement for Boolean Satisfiability.

LOCALIZER Statements The simple statement shown in Figure 7.2 is incremental in the computation of the invariants but it does not maintain the set of candidates for flipping incrementally: the candidates are obtained by evaluating the number of clauses satisfied in the new state obtained by flipping each variable. The statement shown in Figure 7.3 is completely incremental and maintains the set of candidates for flipping at any computation step. This new statement is significantly faster than the simple one, while remaining easy to design. Note that LOCALIZER cannot in general deduce such optimizations automatically, given the fact that operators may be arbitrarily complex and that some problems are not easily amenable to such optimizations. The interest of the statement for this chapter lies in the illustration of several interesting features, which we now describe. Note that the actual instance data is described in the `Init` section which is not shown.

Several sections of the statement differ from the simple version of GSAT.

Constant section The Constant section contains two additional sets: $po[i]$ represents the set of clauses in which atom i appears positively, while $no[i]$ represents the set of clauses in which atom i appears negatively.

```

    Solve
Data Type:
    clause = record
        p : {int};
        n : {int};
    end;
Constant:
    m: int = ...;
    n: int = ...;
    cl: array[1..m] of clause = ...;
    po: array[ i in 1..n] of {int} := {c : int | select c from 1..m where i in cl[c].p};
    no: array[ i in 1..n] of {int} := {c : int | select c from 1..m where i in cl[c].n};
Variable:
    a: array[1..n] of boolean;
Invariant:
    nbtl[i in 1..m] : int = sum(i in cl[i].p) a[j] + sum(j in cl[i].n) !a[j];
    g01[i in 1..n] : int = sum(j in po[i]) (nbtl[j] = 0) - sum(j in no[i]) (nbtl[j] = 1);
    g10[i in 1..n] : int = sum(j in no[i]) (nbtl[j] = 0) - sum(j in po[i]) (nbtl[j] = 1);
    gain[ i in 1..n ] : int = if a[i] then g10[i] else g01[i];
    maxGain : int = max(i in 1..n) gain[i];
    Candidates : {int} =
        {i : int | select i from 1..n where gain[i] = maxGain and gain[i] ≥ 0};
    nbClauseSat : int = sum(i in 1..m) (nbtl[i] > 0);
Satisfiable:
    nbClauseSat = m;
Neighborhood:
    move a[i] := !a[i]
    where i from Candidates;
Start:
    forall(i in 1..n)
        random(a[i]);
Restart:
    forall(i in 1..n)
        random(a[i]);

```

Figure 7.3: A More Incremental Statement of GSAT.

The Invariant section The invariants are more involved in this model and they maintain incrementally the set of candidates which can be selected for a flip. The traditional invariants $nbtl$ and $nbClauseSat$ maintain the number of true literal in a clause and the number of satisfied clauses. $nbtl[i]$ is computed by counting the number of positive atoms

and the negation of the negative atoms.

The informal meanings of the new invariants are the following. $g01[i]$ represents the change in satisfied clauses when changing the value of atom i from false to true, assuming that atom i is currently false. Obviously, the flip produces a gain for all unsatisfied clauses where atom i appears positively. It also produces a loss for all clauses where i appears negatively and is the only atom responsible for the satisfaction of the clause. $g10[i]$ represents the change in satisfied clauses when changing the value of atom i from true to false, assuming that atom i is currently true. It is computed in a way similar to $g01$. $gain[i]$ represents the change in satisfied clauses when changing the value of atom i . It is implemented using a conditional expression in terms of $g01[i]$, $g10[i]$, and the current value of atom i . $maxGain$ is simply the maximum of all gains. Finally, *Candidates* describes the set of candidates for flipping. It is defined as the set of atoms whose gain is positive and maximal.

The Neighborhood section Once the invariants have been described, the neighborhood is defined by flipping one of the candidates. There is no need to specify an optimization qualifier, since this information is already expressed in the invariants. Note that some invariants in this model involve sets, conditional expressions, and aggregation operators which are maintained incrementally. They clearly illustrate the significant support provided by LOCALIZER. Users can focus on describing the data needed for their application, while LOCALIZER takes care of maintaining these data efficiently.

7.1.4 Extensions

Adding Weights References [66],[67] propose to handle the special structure of some SAT problems by associating weights to the clauses and updating these weights each time a new local search is initiated. We now show how easy it is to integrate this feature. The changes consist in introducing weight variables $w[i]$ in the state, in modifying the computations of the invariants for $g01$ and for $g10$ by multiplying the appropriate terms by the weights, i.e.,

$$\begin{aligned} g01[i \text{ in } 1..n] : \text{int} &:= \text{sum}(j \text{ in } po[i]) \ w[j] \times (a[j] = 0) - \\ &\quad \text{sum}(j \text{ in } no[i]) \ w[j] \times (a[j] = 1); \\ g10[i \text{ in } 1..n] : \text{int} &:= \text{sum}(j \text{ in } no[i]) \ w[j] \times (a[j] = 0) - \\ &\quad \text{sum}(j \text{ in } po[i]) \ w[j] \times (a[j] = 1); \end{aligned}$$

and in updating the weights after each local search by changing the restarting section to

```

Restart:
  forall(i in 1..m)
     $w[i] := w[i] + (nbtl[i] = 0);$ 
  forall(i in 1..n)
    random(a[i]);

```

The rest of the statement remains exactly the same, showing the ease of modification of LOCALIZER statements.

Random walk/noise The same reference [66] also proposes a more extensive usage of randomization to escape from local minima. The proposal is to allow from time to time a variable flip regardless of its impact on the objective function. The frequency of this event is itself driven by a simple probability distribution such as: in 90% of the cases, proceed normally, in the other 10%, flip a variable at random. The next excerpt demonstrates how to express the strategy.

```

try
  Pr(0.1):
    move
       $a[i] := la[i]$ 
    where
      i from OccurInUnsatClause
    accept when always ...;
  default:
    best move
       $a[i] := la[i]$ 
    where
      i from {1..n}
    accept when noDecrease;
end

```

7.1.5 Experimental Results

GSAT is generally recognized as a fast and very well implemented system. The experimental results were carried out as specified in [68]. Table 7.1 gives the number of variables (*V*), the number of clauses (*C*), and MaxTrials (*I*) for each size of benchmarks as well as the CPU times in seconds of LOCALIZER (*L*), the CPU times in seconds of GSAT (*G*) as reported in [68], and the ratio *L/G*. The times of GSAT are given on a SGI Challenge with a 70 MHz MIPS R4400 processor. The times of LOCALIZER were obtained on a SUN SPARC-10 40MHz and scaled up by a factor 1.5 to account for the speed difference between the two machines. LOCALIZER times are for the incremental model. Note that this comparison is not perfect (e.g., the randomization may be different) but it is sufficient for showing that LOCALIZER can be implemented efficiently.

The class of formula used for the experiment correspond to hard instances where the ratio of $\frac{nbClause}{nbVars}$ is about 4.3. It is well known that whenever the actual ratio becomes much smaller or much larger, the instance becomes extremely easy to solve. For formulas with less than 200 variables, 100 benchmarks were generated. For larger formulas, 10 benchmarks were generated. The results reported in the table aggregates over 10 runs for each formula for a total of 1000 runs (100 for large formulas).

	<i>V</i>	<i>C</i>	<i>I</i>	<i>L</i>	<i>G</i>	<i>L/G</i>
1	100	430	500	19.54	6.00	3.26
2	120	516	600	40.73	14.00	2.91
3	140	602	700	54.64	14.00	3.90
4	150	645	1500	154.68	45.00	3.44
5	200	860	2000	873.11	168.00	5.20
6	250	1062	2500	823.06	246.00	3.35
7	300	1275	6000	1173.78	720.00	1.63

Table 7.1: GSAT: Experimental Results.

As can be seen, the distribution of ratios L/G is uniform over the range of benchmarks. This indicates that the LOCALIZER statement scales in the same way than GSAT. The gap between the two systems is about one machine generation (i.e., on modern workstations, LOCALIZER runs as efficiently as GSAT on machines of three years ago), which is really acceptable given the preliminary nature of our (unoptimized) implementation.

7.2 Graph Coloring

This section considers the graph-coloring problem, i.e., the problem of finding the smallest number of colors to label a graph such that two adjacent vertices have a different color. It shows how a simulated annealing algorithm proposed in [29] can be expressed in LOCALIZER. Of particular interest is once again the close similarity between the problem description and the statement. In addition, graph coloring makes it possible to discuss some interesting issues about the tradeoff between expressiveness and efficiency. Note finally that graph coloring could be expressed as an instance of SAT as could any NP-Complete problem. However, it is often desirable to specialize the local search to the problem at hand and LOCALIZER makes it possible to exploit the special structure of each problem.

7.2.1 The Problem

The problem consists of finding the chromatic number of a graph, i.e., the smallest number of colors needed to color each vertex so that no two adjacent vertices are given the same color. For a graph with n vertices, the algorithm considers n colors which are the integers between 1 and n . The objective is to find a coloring (assignment of colors to vertices) that is valid (no two adjacent vertices in the same color class) and minimal (uses as few colors as possible).

7.2.2 The Local Search Algorithm

This section considers a local search algorithm proposed in [29]. The first idea is to consider both valid and invalid colorings (a coloring is valid if no two adjacent vertices are given the same colors). The algorithm classifies the vertices in color classes and keeps track of the bad edges in these classes. Color class C_i is the set of all vertices colored with i and the bad edges of C_i , denoted by B_i , are the edges whose vertices are both colored with i .

The second idea of the algorithm is to minimize the objective function $\sum_{i=1}^n 2|B_i||C_i| - |C_i|^2$. This function is interesting since its local minima are valid colorings. To minimize the function, the algorithm chooses a vertex and chooses a color that comes either from a non-empty color class or one of the unused colors. It is important to consider only one of the unused colors to avoid a bias towards unused colors. A move is accepted if it improves or does not degrade the value of the objective function or, if not, with a standard probability of simulated annealing algorithms¹.

7.2.3 A Simple Model for Graph Coloring

Figure 7.4 depicts the simulated annealing statement for graph coloring. The statement closely follows the above description.

Instance data The instance data is described by the number of vertices n (each vertex being a number between 1 and n), the set of edges E between vertices, and the annealing parameters *cutOff*, *chPerc*, *maxFreeze* that are described subsequently. The adjacency matrix A is computed automatically from the edges.

¹As a consequence the Markov chain associated with the process has a unique stationary distribution, and, therefore, there exist a finite sequence of transitions that transforms i into j for any pair of state i, j . This guarantees that the algorithm converges asymptotically to the optimal solution.

```

Optimize
Data Type:
    edge = record s : int; t : int; end;
Constant:
    n      : int = ...;
    E      : { edge } = ...;
    cutOff : real = ...;
    chPerc : real = ...;
    maxFreeze: real = ...;
    A      : array[i in 1..n, j in 1..n] of boolean := ⟨i, j⟩ in E;
Variable:
    x : array[1..n] of int;
    t : int;
    fc : int;
    ch : int;
Invariant:
    C : array[i in 1..n] of {int} := distribute(x, {1..n}, {1..n});
    Empty : {int} = { i : int | select i from 1..n where size(C[i]) = 0 };
    NEmpty : {int} = { i : int | select i from 1..n where size(C[i]) > 0 };
    unused : int = minof(Empty);
    Candidates : {int} = NEmpty union unused;
    B : array[k in 1..n] : { edge } = { ⟨i, j⟩ : edge | select i from C[k]
                                                select j from C[k] where A[i, j] };
    f : int = sum(i in 1..n) (2 × size(C[i]) × size(B[i]) - size(C[i])2);
    countB : int = sum(i in 1..n) size(B[i]);
Satisfiable:
    countB = 0;
Objective Function:
    minimize f;
Neighborhood:
    move x[i] := c
    where
        i from {1..n};
        c from Candidates
    accept when
        improvement → { if countB = 0 then fc = 0 endif; ch := ch + 1; }
        cor noDecrease
        cor Pr( $e^{-\text{delta}/t}$ ) : always → ch := ch + 1;
Start:
    T := initTemp; fc := 0; ch := 0; forall(i in 1..n) random(x[i]);
Restart:
    T := factor × T; if ch/trial < chPerc then fc := fc + 1 endif;
Parameter: MaxTrials := 90 * sf * n;
Local Condition: ch < round(cutOff * n);
Global Condition: fc < maxFreeze;

```

Figure 7.4: A LOCALIZER Statement for Graph Coloring.

State representation The state is represented by the variables $x[i]$ that represent the colors of vertices, by the temperature t , and by two other annealing parameters fc and ch that are used to control the local and global conditions.

The Invariant section The invariants describe the sets C_i , B_i , and the objective function. The invariant `distribute($x, \{1..n\}, \{1..n\}$)` computes the sets C_i . The “unused” color is obtained by taking the smallest unused color. The set of candidate colors are thus all the “used” colors together with the selected “unused” color. The bad edges are maintained for each color class by considering adjacent vertices in the color class. The total number of bad edges (`countB`) is also maintained to decide satisfiability.

The Neighborhood section The neighborhood is described by choosing a vertex and a candidate color. Acceptance obeys the standard simulated annealing criterion.

Termination The stopping criteria are directly derived from the algorithm description in [29]. The inner local search can only perform `round($cutOff * n$)` variations of the objective function and the variable ch maintains the number of variations by incrementing ch in the acceptance actions. The motivation behind this choice is to control the search when the temperature is high. The global iteration is stopped when $fc \geq maxFreeze$. The intuition here is that fc represents the number of local searches without significant progress. fc is reset to 0 whenever there is an improvement. It is incremented each time the local search is restarted and there was no significant change in the previous local search.

7.2.4 A More Incremental Statement

The above statement uses the resulting state in the acceptance criterion. This means that, in order to evaluate if a move is acceptable, it is necessary to simulate the move, e.g., to update all the invariants and to undo the changes if the move is not acceptable. As mentioned previously, although the algorithms in LOCALIZER are incremental, such a simulation may become the most consuming part of the algorithm at low temperatures when many candidate moves are discarded.

Figure 7.5 depicts a new, more incremental, statement. The key insight here is that it is possible to evaluate the impact of the move in the current state by simply looking at the bad edges in color classes $C[x[i]]$ and $C[c]$ and deducing the variation d of the objective function. In particular, the old class color of vertex i , i.e., $x[i]$, decreases by one in size, while its number of bad edges decreases by

$$\text{sum}(j \text{ in } C[x[i]]) A[i, j].$$

Similarly, the new class color c of vertex i increases by one in size and its number of bad edges increases by

$$\text{sum}(j \text{ in } C[c]) A[i, j].$$

The variation d of the objective function is computed in the neighborhood definition and, in the acceptance criterion, keywords **improvement**, **noDecrease**, and **delta** are replaced by $d < 0$, $d = 0$, and d respectively. The performance gain on this problem is significant (about a factor of 5 on large instances (500 vertices)).

7.2.5 Experimental Results

Graph coloring was the object of an extensive experimental evaluation in [29] and this section reports on experimental results along the same lines. The experiments were conducted on graphs of densities 10, 50, and 90 and of sizes 125, 250, and 500. They were also conducted on so-called “cooked” graphs. Cooked graphs have a well-known optimal coloring. A cooked graph with n vertices and with a chromatic number κ is constructed as follows:

1. Randomly assign the vertices with equal probability to κ color classes.
2. For each pair (u, v) of vertices in different color classes, place an edge to connect them with a probability $\frac{\kappa}{2(\kappa-1)}$.
3. Pick κ vertices, one from each class, and make sure they form a clique in G .

Because of the nature of the experimental results reported in [29], it is not easy to compare the efficiency of LOCALIZER to the efficiency of their algorithm. As a consequence, we decided to build a very efficient C implementation of their algorithm from scratch and to compare it with LOCALIZER. This implementation was performed by a graduate student not connected to the LOCALIZER project. This student was closely supervised to obtain a very efficient incremental algorithm. As far as we can judge, the timings and the quality of this algorithm seem consistent with those in [29]. In the following, we discuss the development time of the two implementations, the quality of the solutions obtained (to make sure that the algorithms are comparable in quality), and the efficiency.

Development Time The C implementation of the algorithm is about 1500 lines long and required a full week. This should be compared with the concise model presented earlier in this chapter.

```

Optimize
Data Type:
  edge = record s : int; t : int; end;
Constant:
  n      : int = ...;
  E      : {edge} = ...;
  cutOff : real = ...;
  chPerc : real = ...;
  maxFreeze: real = ...;
  A      : array[i in 1..n, j in 1..n] of boolean := <i, j> in E;
Variable:
  x : array[1..n] of int;
  t : int;
  fc : int;
  ch : int;
Invariant:
  C : array[i in 1..n] of {int} := distribute(x, {1..n}, {1..n});
  Empty : {int} = { i : int | select i from 1..n where size(C[i]) = 0 };
  NEmpty : {int} = { i : int | select i from 1..n where size(C[i]) > 0 };
  unused : int = minof(Empty);
  Candidates : {int} = NEmpty union unused;
  B : array[k in 1..n] : {edge} = { <i, j> : edge | select i from C[k]
                                     select j from C[k] where A[i, j] };
  f : int = sum(i in 1..n) (2 × size(C[i]) × size(B[i]) - size(C[i])2)
  countB : int = sum(i in 1..n) size(B[i]);
Operator:
  int f(C : int, S : int) { return 2 * C * S - C2; }
  int diff(i : int, dc : int, ds : int) {
    return f(size(C[i]) + dc, size(B[i]) + ds) - f(size(C[i]), size(B[i])); }
Satisfiable:
  countB = 0;
Objective Function:
  minimize f;
Neighborhood:
  move x[i] := c
  where
    i from {1..n};
    c from Candidates;
    nb = sum(j in C[c]) A[i, j];
    ob = sum(j in C[x[i]]) A[i, j];
    d = diff(x[i], -1, -ob) + diff(c, 1, nb)
  accept when in current state
    d < 0 → { if countB = 0 then fc = 0 endif; ch := ch + 1; }
    cor d = 0
    cor Pr(e-d/t): always → ch := ch + 1;

```

Figure 7.5: A More Incremental Neighborhood for Graph Coloring.

<i>Data Set</i>				<i>Color Ranges</i>				<i>Frequencies</i>							
	<i>V</i>	<i>D</i>	<i>SF</i>					LOCALIZER				<i>C</i>			
random	125	50	3	19	20			92	8			87	13		
random	250	50	4	-32	33	34	35-	9	43	44	4	6	41	50	3
random	500	50	4	55	56	57	58-	4	54	41	1	8	48	36	8
random	125	10	1	6	7	8	-	48	52	-	-	43	54	3	-
random	250	10	1	9	10	11	-	27	70	3	-	29	70	1	-
random	500	10	2	15	16	17	-	1	79	20	-	3	86	11	-
random	125	90	1	-44	45	46	47-	7	30	30	33	15	32	26	27
random	250	90	1	-78	79	80	81-	4	15	35	46	4	19	25	52
random	500	90	1	-143	144	145	146-	30	22	16	32	15	11	38	36
cooked	125		4	9	-	-	-	100	-	-	-	100	-	-	-
cooked	250		1	15	-	-	-	100	-	-	-	100	-	-	-
cooked	500		2	25	26-			71	29			65	35		

Table 7.2: Graph Coloring: Quality of the Solutions.

Quality of the Solutions Table 7.2 describes the quality of the coloring found by LOCALIZER. These results agree with those of the C implementation and with those reported in [29]. Each set of rows corresponds to a class of graphs and to 100 executions of LOCALIZER on 10 graphs from this class (a total of 1000 runs). Each row describe a benchmarks in terms of its class, the number of vertices (V), the edge density (D) and the size factor parameter (SF). A row also reports on the various values found by LOCALIZER on these graphs and their frequencies. For instance, the first row reports that, on graphs of 125 vertices and density of 50%, 92% of the executions led to a coloring with 19 colors and 8% of the executions led to a coloring with 20 colors. The results are given both for random and cooked graphs and the frequencies are similar for both LOCALIZER and the C implementation. The last column C reports the frequencies for the C implementation. Note that LOCALIZER and the C implementation were compared on the same problem instances and only the random seed did change from one run to the next.

Efficiency Table 7.3 compares the efficiency of LOCALIZER with the C implementation on the same problems. Each row reports the average time of the two implementations for the 100 graphs in each class and computes the slowdown of LOCALIZER. The experiments were performed on a SUN Sparc Ultra-1 running Solaris 5.5.1 and the standard C++ compiler. The minimum and maximum slowdowns are respectively 3.56 and 5.54. On these problems, the slowdown is in general slightly higher than a machine generation but it remains reasonable given the preliminary nature of the implementation. This slowdown should also be contrasted with the substantial reduction in development time.

	<i>V</i>	<i>D</i>	<i>SF</i>	LOCALIZER (L)	<i>C Impl.</i> (C)	<i>L/C</i>
random	125	50	3	78.3	18.9	4.50
random	250	50	4	82.8	18.4	4.50
random	500	50	4	633.7	123.4	5.10
random	125	10	1	22.5	4.8	4.60
random	250	10	1	109.2	22.02	4.96
random	500	10	2	159.8	28.8	5.50
random	125	90	1	16.18	4.53	3.56
random	250	90	1	49.32	8.89	5.54
random	500	90	1	162.7	29.6	4.88
cooked	125		4	22.09	4.18	5.28
cooked	250		1	37.22	7.79	4.77
cooked	500		2	240.3	49.9	4.80

Table 7.3: Graph Coloring: Efficiency of LOCALIZER.

7.3 Graph Partitioning

This section considers graph-partitioning, the typical application used to illustrate local search in textbooks.

7.3.1 The Problem

The graph-partitioning problem consists of finding a partition of the vertices of a graph into two sets of equal size which minimizes the number of edges connecting the two sets.

7.3.2 The Local Search Algorithm

A Traditional local search algorithm for graph partitioning [47] maintains two sets of equal size and swaps vertices between these two sets. The local search algorithm considered here is based on a simulated annealing approach presented in [28]. This algorithm relaxes the idea of maintaining a feasible solution and a move consists of selecting a vertex and moving it to the other set. The objective function combines the objective of minimizing the connections between the two sets with the desire to favor balanced solutions. It is given as

$$SB + \alpha * IMB^2$$

where SB is the number of connections, IMB is the imbalance between the two sets, and α is a parameter of the algorithm.

7.3.3 The LOCALIZER Statement

The Instance Data The LOCALIZER statement is depicted in Figure 7.6. The instance data is described by the number of vertices n (each vertex being a number between 1 and n), the set of edges E between vertices, and the annealing parameters (which we will not describe for this statement since they are again taken directly from [28]). The adjacency matrix A is computed automatically from the edges.

The State Representation The state is represented by the variables $x[i]$ and some annealing variables. Variable $x[i]$ is true if i belongs to set S_0 and false otherwise. The invariants POS and PIS maintain the size of the sets S_0 and S_1 respectively. Invariants $EX[i]$ and $IT[i]$ represent the traditional internal and external costs of a vertex. The external cost of a vertex is the number of edges connecting the vertex to vertices in the other set, while the internal cost of a vertex is the number of edges connecting the vertex to vertices in the same set. IMB is the imbalance between the two sets and OBJ is the objective function mentioned earlier.

The Neighborhood section The neighborhood is essentially similar to the neighborhood of the graph-coloring problem, except that here a candidate move consists of flipping the value of a variable.

Once again, there is a small distance between the algorithm and its statement in LOCALIZER. As was true for graph coloring, it is easy to find a more incremental version of the model by evaluating the variation of the objective function in the current state using the variations on the internal and external costs. The reader can find the changes in the statement in Figure 7.7.

7.3.4 Experimental Results

The problem has been studied experimentally in [28] and, once again, the experiments reported here are based on a similar setting. Two classes of graphs are considered: randomly generated graphs of various density and so-called *geometric* graphs. Geometric graphs are constructed as follows. Pick $2n$ numbers between 0 and 1. Interpret these numbers as the coordinates of vertices lying inside a unit square. Define an edge $\langle i, j \rangle$ between vertex i and j if the Euclidean distance between vertex i and j is less than or equal to a constant d . The average degree of vertices not too close to the boundary of the unit square have an average degree equal to $n\pi d^2$. The instance used in the experiment are generated by choosing a

```

Optimize

Type:
    edge = record s : int; t : int; end;

Constant:
    n : int = ...;
    E : {edge} = ...;
    alpha : real = ...;
    cutOff : real = ...;
    sf : int = ...;
    chPerc : int = ...;
    A : array[i in 1..n] of {int} := {j : int | select j from 1..n
                                         where <i,j> in E or <j,i> in E};

Variable:
    x : array[1..n] of boolean;
    t : real;
    fc : int;
    ch : int;

Invariant:
    POS : int = sum(i in 1..n) x[i];
    PIS : int = n - POS;
    EX : array[i in 1..n] of int = sum(k in A[i]) (x[k] <> x[i]);
    IT : array[i in 1..n] : of int = sum(k in A[i]) (x[k] = x[i]);
    SB : int = (sum(i in 1..n) EX[i])/2;
    IMB : int = POS - PIS;
    OBJ : real = SB + alpha * IMB2;

Satisfiable: IMB=0;
Objective Function: minimize OBJ;
Neighborhood:
    move x[i] := !x[i]
    where i from {1..n};
    accept when improvement → ch:=ch+1
           cor noDecrease
           cor Pr(e-delta/t) : always → ch:=ch+1;

Start:
    t:=10;fc:=0;ch:=0;
    forall(i in 1..n) x[i] := random({true,false});

Restart:
    t := t * 0.95;
    if ch * 100/trial < chPerc then fc := fc + 1 endif;
    ch := 0;

Parameter: MaxTrials := sf * n;
Local Condition: ch < cutOff * n;
Global Condition: fc < 5;

```

Figure 7.6: A Graph Partitioning Statement.

```

Optimize

Type:
    edge = record s : int; t : int; end;

Constant:
    n : int = ...;
    E : {edge} = ...;
    alpha : real = ...;
    cutOff : real = ...;
    sf : int = ...;
    chPerc : int = ...;
    A : array[i in 1..n] of {int} := {j : int | select j from 1..n
                                         where <i, j> in E or <j, i> in E};

Variable:
    x : array[1..n] of boolean;
    t : real;
    fc : int;
    ch : int;

Invariant:
    POS : int = sum(i in 1..n) x[i];
    P1S : int = n - POS;
    EX : array[i in 1..n] of int = sum(k in A[i]) (x[k] <> x[i]);
    IT : array[i in 1..n] of int = sum(k in A[i]) (x[k] = x[i]);
    SB : int = (sum(i in 1..n) EX[i])/2;
    IMB : int = POS - P1S;
    OBJ : real = SB + alpha * IMB2;

Satisfiable: IMB=0;
Objective Function: minimize OBJ;
Neighborhood:
    move x[i] := !x[i]
    where i from {1..n};
    D = IT[i] - EX[i] + (if x[i] then -4*IMB+4 else 4*IMB+4) * alpha
    accept when D < 0 → ch:=ch+1
    cor D = 0
    cor Pr(e-D/t) : always → ch:=ch+1;

Start:
    t:=10;fc:=0;ch:=0;
    forall(i in 1..n) x[i] := random({true,false});

Restart:
    t := t * 0.95;
    if ch * 100/trial < chPerc then fc := fc + 1 endif;
    ch := 0;

Parameter: MaxTrials := sf * n;
Local Condition: ch < cutOff * n;
Global Condition: fc < 5;

```

Figure 7.7: A More Incremental Graph Partitioning Statement.

value n , and a value for $n\pi d^2$.

Graph			Results (T=10,TF=0.95,SF=16,chPerc=2%,Cutoff=10%)						
<i>Class</i>	<i>V</i>	<i>D</i>	<i>Ranges</i>			<i>Frequencies</i>			<i>LOCAL.</i>
random	124	2	11-13	14-16	17-19	33	38	29	2.22
random		4	55-59	60-65	66-77	30	34	36	2.40
random		8	159-174	175-190	191-	23	53	24	3.24
random		16	481-560	561-640	641-	36	45	19	4.05
random	250	1	20-24	25-29	30-35	19	22	59	4.68
random		2	92-106	107-121	122-131	12	42	36	5.10
random		4	324-343	344-363	364-380	38	43	19	6.50
random		8	828-877	878-927	928-	37	40	23	9.98
random	500	0.5	48-54	55-59	60-66	15	43	42	10.08
random		1	219-231	232-244	245-256	17	52	31	10.76
random		2	637-661	662-686	686-718	10	15	75	14.44
random		4	1661-1701	1702-1741	1742-1824	22	58	20	20.67
random	1000	0.25	90-103	104-118	119-126	41	52	7	19.99
random		0.5	439-455	456-475	476-503	36	43	21	22.62
random		1	1326-1357	1358-1397	1398-1427	33	51	16	29.97
random		2	3253-3319	3320-3394	3394-3466	11	37	52	40.10
<i>Class</i>	<i>V</i>	$n\pi d^2$	<i>Ranges</i>			<i>Frequencies</i>			<i>LOCAL.</i>
geom.	500	5	4-13	14-23	24-37	7	58	35	8.26
geom.		10	35-59	60-84	85-123	19	42	39	9.50
geom.		20	148-246	247-346	347-450	41	44	15	11.40
geom.		40	441-840	841-1240	1241-3400	47	20	33	14.50
geom.	1000	5	24-43	44-63	64-78	37	57	6	18.70
geom.		10	65-114	115-164	165-205	16	54	30	21.20
geom.		20	196-399	400-599	600-816	32	60	8	23.84
geom.		40	537-1099	1100-1599	1600-5829	28	49	23	28.56

Table 7.4: Graph Partitioning: Experimental Results.

Quality of the Solutions Table 7.4 depicts the experimental results of LOCALIZER. The first row gives the setting of our parameters: T is the starting temperature, TF is the percentage of reduction of the temperature, SF is the size factor, and the remaining two parameters were described previously. The table reports both the quality of the results and their distribution, as well as the performance of LOCALIZER on these problems. Once again, each row of the table reports the result for 100 executions of LOCALIZER. The columns under *Ranges* give ranges for the solutions produced by LOCALIZER and the column *LOCAL* gives the running time for LOCALIZER in seconds. For a given benchmarks, all ranges have the same width, the lower bound of the first range is the best solution found. The upper bound of the last range, when present, gives the worst solution ever returned. If absent, it means that LOCALIZER produced at least one solution that would not fit in the range.

Graph			Results (T=10,TF=0.95,SF=16,chPerc=2%,Cutoff=10%)				
<i>Class</i>	<i>V</i>	<i>D</i>	<i>L.Best</i>	<i>L.Time</i>	<i>J.Best</i>	<i>J.Time</i>	<i>Ratio</i>
random	124	2	11	2.22	13	85.4	38.5
		4	55	2.4	63	82.2	34.3
		8	159	3.24	178	78.1	24.1
		16	481	4.05	449	104.8	25.9
random	250	1	20	4.68	29	190.6	40.7
		2	92	5.1	114	163.7	32.1
		4	324	6.5	357	186.8	28.7
		8	828	9.98	828	223.3	22.4
random	500	0.5	47	10.08	52	379.8	37.7
		1	219	10.76	219	308.9	28.7
		2	635	14.44	628	341.5	23.6
		4	1661	20.67	1744	432.9	20.9
random	1000	0.25	90	19.99	102	729.9	36.5
		0.5	439	22.62	451	661.2	29.2
		1	1326	29.97	1367	734.5	24.5
		2	3253	40.01	3389	853.7	21.3

Table 7.5: Graph Partitioning: Comparison Results.

Efficiency Table 7.5 compares LOCALIZER with the results reported in [28]. It is important to mention that [28] explicitly writes that their results are very hard to reproduce, since they chose the temperature of the annealing algorithm according to some preliminary observation of its behavior on each class of graphs. LOCALIZER, in contrast, is always executed with the given parameters. In addition, the relevant results are only given for the random graphs in [28]. The times for LOCALIZER are given on a SUN Sparc Ultra-1 running Solaris 5.5.1 and the standard C++ compiler, while the results in [28] are given for a slow VAX-750. In general, the quality of the results produced by LOCALIZER is slightly better than the quality in [28]. Once again, the performance results indicate that LOCALIZER behaves well on these problems.

7.4 Job-Shop Scheduling

Scheduling applications are ubiquitous in industry and have been the topic of extensive research in recent years [20], [84], [3], [33], [13], [76] and [77]. This section illustrates how LOCALIZER can be used to implement a tabu search procedure proposed in [33] for job-shop scheduling.

7.4.1 The Problem

A job-shop scheduling problem consists of a set of jobs. Each job is a sequence of tasks (e.g., the first task in a job must precede the second task and so on) and each task executes on a given machine. Tasks executing on the same machine cannot overlap in time. The job-shop scheduling problem amounts to finding an assignment of starting dates to the tasks satisfying the precedence and non-overlapping constraints and minimizing the makespan, i.e., the maximum duration of all jobs.

It is well-known that solving a job-shop problem mostly consists of determining an optimal ordering for the tasks on the various machines. Once an optimal ordering has been found, the problem can be solved by a PERT algorithm on the graph induced by the precedence constraints and the ordering.

We also assume for convenience that the graph contains a source vertex that precedes the first task of every job and a sink vertex that follows the last task of every job. Such a graph is called a *solution graph* and its arcs are of two types: the *precedence arcs* that are static and express the precedence constraints within a job and the *machine arcs* that express the precedence constraints induced by the chosen ordering. The weight of an arc is simply the duration of the task corresponding to the source of the arc (with the convention that the sink and the source have durations zero). Typically, a PERT algorithm computes, among other things, the earliest starting dates for each task. These earliest starting dates provide a solution to the job-shop scheduling problem.

The local search algorithm described later in this section applies local transformations to solution graphs. It is thus useful to review a number of notations and concepts on solution graphs. These notations assume that the source and the sink execute (first and last) on all machines and are the first and the last tasks of all jobs, which is not restrictive, since they have duration zero. The duration of task t is denoted by $d(t)$ and we assume that the problem has N tasks ($N + 2$ when the source and the sink are included). Tasks are identified by integers and the source and the sink are numbered 0 and $N + 1$ respectively.

Every task t in a solution graph (except the sink and the source) has two predecessors: a predecessor for the job, denoted by $pj(t)$, and a predecessor for the machine, denoted by $pm(t)$. Similarly, every task t (except the source and the sink) has two successors: a successor for the job, denoted by $sj(t)$, and a successor for the task, denoted by $sm(j)$. The release date of a task t , denoted by $r(t)$, is the longest path from the source to t . The tail of a task t , denoted by $q(t)$, is the longest path from t to the sink. Intuitively, the tail of a task represents what remains to be done, once t is released. The release dates and the tails

can be computed by simple recurrences:

$$r(t) = \begin{cases} 0 & \text{if } t = 0 \\ \max(r(pj(t)) + d(pj(t)), r(pm(t)) + d(pm(t))) & \text{otherwise} \end{cases}$$

$$q(t) = \begin{cases} 0 & \text{if } t = N + 1 \\ \max(q(sj(t)) + d(t), q(sm(t)) + d(t)) & \text{otherwise} \end{cases}$$

Of course, the release date of the sink is the makespan of the solution. Critical arcs play a fundamental role in the local search algorithm. An arc is critical if it belongs to a critical path. More precisely, an arc (t, u) is critical if $r(t) + d(t) = r(u)$ and if $r(t) + q(t)$ is equal to the makespan.

7.4.2 The Local Search Algorithm

This section describes a local search algorithm for the job-shop scheduling based on a neighborhood known as *N1* and a tabu-search strategy. This local search algorithm is the essence of the procedure proposed in [33]. Other neighborhoods and strategies have been proposed and it is not difficult to generalize the results presented here to these other algorithms.

The Neighborhood As mentioned, a solution to the job-shop scheduling problem mostly consists of ordering the tasks of the various machines. The idea underlying neighborhood *N1* is to consider all critical arcs (u, v) and to swap the two tasks u and v in the ordering of their machine. In the following, we often abuse language and talk about swapping an arc. Note however that such a swap in fact consists of removing and adding three arcs (see Figure 7.8. If p (resp. s) is the predecessor (resp. successor) of u (resp. v) on the machine, then the arcs (p, u) , (u, v) , and (v, s) are removed from, and the arcs (p, v) , (v, u) , and (u, s) are added to, the solution graph. Neighborhood *N1* has a number of interesting properties: it preserves feasibility, it is connected (i.e., there exists a sequence of moves from any given state to the optimal solution), and it is minimal in the sense that swapping non-critical arcs cannot improve the makespan.

The Exploration Strategy The exploration strategy is based on tabu search, which appears to be successful for job-shop scheduling. It consists of selecting the swap that results in the state with the best makespan (which, in fact, may be worse than the makespan of the current state). A tabu-list also keeps the inverse of the swaps performed recently (e.g.,

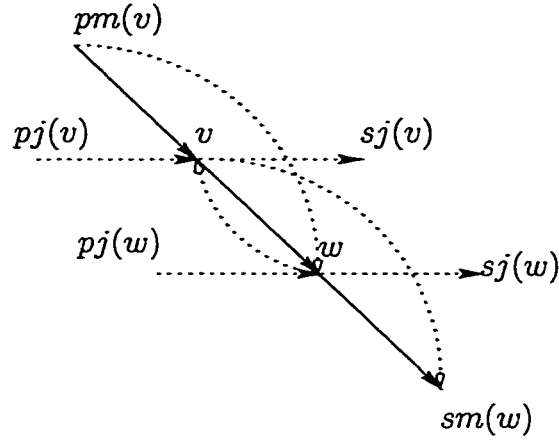


Figure 7.8: Swapping vertices v and w in a Job-shop schedule.

if (u, v) is swapped, then (v, u) is marked as tabu). The length of the tabu-list varies over time. It decreases (resp. increases) when the makespan decreases (resp. increases).

Incrementality Because running a PERT algorithm to evaluate the makespan of every possible move is relatively expensive, it is generally proposed to approximate its value by considering the paths going through the swapped vertices only. This approximation is of course a lower bound on the actual makespan but it can be computed in constant time. Indeed, for a swap (v, w) , it is sufficient to compute

$$\max(r'(v) + q'(v), r'(w) + q'(w))$$

where

$$r'(v) = \max(r(pj(v)) + d(pj(v)), r'(w) + d(w))$$

$$q'(v) = \max(d(v) + q(sj(v)), d(v) + q(sm(w)))$$

$$r'(w) = \max(r(pj(w)) + d(pj(w)), r(pm(v)) + d(pm(v)))$$

$$q'(w) = \max(d(w) + q(sj(w)), d(w) + q'(v))$$

7.4.3 A Simulation Statement

Figure 7.9 describes a LOCALIZER statement for job-shop scheduling based on a simulation approach.

The Constant Section The data in this model specifies the number of jobs nbJ , the number of machines nbM , the total number of tasks N (excluding the source and the sink),

```

    Optimize
Constant:
    nbJ   : int = ...;
    nbM   : int = ...;
    N     : int := nbJ × nbM;
    d     : array[0..N + 1] of int = ...;
    m     : array[1..N] of int = ...;
    sj    : array[i in 1..N] of int = ...;
    pj    : array[i in 1..N] of int = ...;
    F     : {int} = ...;
    L     : {int} = ...;
    minLen: int = ...;
    maxLen: int = ...;
Variable:
    pm    : array[1..N] of int;
    sm    : array[1..N] of int;
    tabu  : array[1..N, 1..N] of int;
    tabuLen: int = ...;
Invariant:
    r : array[i in 0..N] of int :=
        if i=0 then 0 else max(r[pj[i]] + d[pj[i]], r[pm[i]] + d[pm[i]]);
    q : array[i in 1..N+1] of int:=
        if i=N+1 then 0 else max(d[i] + q[sj[i]], d[i] + q[sm[i]]);
    p : array[i in 1..N] of int:= r[i] + q[i];
    makespan : int := max(i in L) p[i];
    Ca : {int} := { i : int | select i in 1..N where p[i] = makespan and
        r[pm[i]] + d[pm[i]] = r[i] and
        pm[i] ≠ 0};
Operator:
    void swap(i : int, j : int) {
        smj : int := sm[j];
        sm[i] := sm[j]; sm[j] := i; sm[pm[i]] := j;
        pm[j] := pm[i]; pm[i] := j; pm[smj] := i;
    }
Objective Function:
    minimize makespan
Neighborhood:
    best move swap(pm[u], u)
    where u from Ca
    such that tabu[PM[u], u] + tabuLen ≤ trials
    accept when
        improvement → {tabuLen := max(tabuLen-1, minLen); tabu[u, pm[u]] := trials;};
        always → {tabuLen := min(tabuLen+1, maxLen); tabu[u, pm[u]] := trials;};

```

Figure 7.9: A First Job-shop Scheduling Statement.

the duration d of the tasks, the machines m assigned to each task, the job successors sj and the job predecessors pj of the tasks, the set of tasks F starting the jobs, and the set of tasks L finishing the jobs. The integers $minLen$ and $maxLen$ also specify bounds on the length of the tabu list.

The State The state is described by specifying the machine predecessor and the machine successor of each task, as well as the tabu list and its length. The tabu list is represented by a matrix that associates with each move (i.e., a pair of tasks) the last “time” its inverse was performed. Note that the “time” is simply the value of the parameter `trials` in the computation model.

Invariants The invariants in this model essentially express the concepts introduced in Section 7.4.2: the release dates, the tails, the makespan, and the critical arcs. The release dates and the tails are maintained by the invariants

```

r : array[i in 0..N] of int :=
    if i=0 then 0 else max(r[pj[i]] + d[pj[i]], r[pm[i]] + d[pm[i]]);
q : array[i in 1..N+1] of int:=
    if i=N+1 then 0 else max(d[i] + q[sj[i]], d[i] + q[sm[i]]);

```

that directly encode the recurrence relations presented earlier. These invariants are much more difficult to maintain than the invariants occurring in applications such as GSAT and graph-coloring. On the one hand, they are expressed recursively. On the other hand, they use variables as indices of invariants (e.g., `r[pj[i]]`), which complicates incremental algorithms since the data dependencies vary over time. The invariant

```

p : array[i in 1..N] of int:= r[i] + q[i];

```

maintains the length of the longest path between the source and the sink going through task i , while the invariant

```

makespan : int := max(i in L) p[i];

```

maintains the makespan as the longest path going through all final tasks. Finally, the critical arcs are represented as the sets of their targets

```

Ca : {int} := { i : int | select i in 1..N where p[i] = makespan and
                                r[pm[i]] + d[pm[i]] = r[i] and
                                pm[i] ≠ 0};

```

i.e., Ca is the sets of tasks i such that $(pm[i], i)$ is critical. Once again, the invariant mostly follows the definition given earlier.

The Neighborhood The neighborhood is defined in terms of the critical arcs and it expresses that an arc $\langle pm[u], u \rangle$ must be considered for a swap unless it is tabu. The best such move is selected as the next state. Procedure `swap` performs the move and the move is tabu if its inverse was executed in the last *tabuLen* iterations. Note also that the tabu list and its length are updated in the acceptance clause.

7.4.4 The LOCALIZER Statement Based on a Makespan Approximation

The model can be modified to include the approximation of the makespan discussed in Section 7.4.2 and a “reasonable” starting point. Figure 7.10 depicts the new statement, with the constant section omitted for brevity.

The key idea is to compute the approximation using a simple function and to choose the move minimizing the approximation. The acceptance criteria is also evaluated in the current state, using the current makespan and the approximation.

Also, it is important in job-shop scheduling to start from a reasonable solution. Our model uses a greedy procedure based on procedural constructs over sets and arrays to find a starting point. It maintains a frontier which consists of the first unscheduled task of each job and it selects the task with the shortest duration first. More advanced procedures such as Bidir proposed by Dell’Amico et al. in [33] can be implemented without difficulty, since LOCALIZER includes a complete programming language.

7.4.5 A Higher-Level Localizer Model

The previous section presented a model for job-shop scheduling using only simple data structures such as sets and arrays. This section presents a LOCALIZER model based on the graph-theoretic concepts available in LOCALIZER and is depicted in Figures 7.11 and 7.12.

The Constant Section The constant section remains mostly unchanged. However, the model introduces a node for each task (these nodes are used later to specify the graph) and represents jobs as paths of tasks.

The Variable Section The state in this model is described as an array of paths, one for each machine. These paths represent how tasks using the same machine are ordered.

The Invariant Section The invariants are of course most interesting. They define a directed graph G whose nodes are the tasks and whose arcs are the precedence arcs (which


```

Optimize
Constant: // omitted
Variable:
    pm    : array[1..N] of int;
    sm    : array[1..N] of int;
    tabu  : array[1..N,1..N] of int;
    tabuLen: int = ...;
Invariant:
    r : array[i in 0..N] of int :=
        if i=0 then 0 else max(r[pj[i]] + d[pj[i]], r[pm[i]] + d[pm[i]]);
    q : array[i in 1..N+1] of int:=
        if i=N+1 then 0 else max(d[i] + q[sj[i]], d[i] + q[sm[i]]);
    p : array[i in 1..N] of int:= r[i] + q[i];
    makespan : int := max(i in L) p[i];
    Ca : {int} := { i : int | select i in 1..N where p[i] = makespan and
        r[pm[i]] + d[pm[i]] = r[i] and
        pm[i] ≠ 0};
Operator:
    void swap(i : int, j : int) {
        smj : int := sm[j];
        sm[i] := sm[j]; sm[j] := i; sm[pm[i]] := j;
        pm[j] := pm[i]; pm[i] := j; pm[smj] := i;
    }
    int appMakeSpan(v : int, w : int) {
        rpw : int := max(r[pj[w]] + d[pj[w]], r[pm[v]] + d[pm[v]]);
        rpv : int := max(r[pj[v]] + d[pj[v]], rpw + d[w]);
        qpj : int := max(d[v] + q[sj[v]], d[v] + q[sm[w]]);
        qpj : int := max(d[w] + q[sj[w]], d[w] + qpj);
        res : int := max(rpw + tpw, rpj + tpj);
        return res;
    }
Objective Function: minimize makespan;
move swap(pm[u], u)
where u from Ca;
    a = appMakespan(pm[u], u)
such that tabu[pm[u], u] + tabuLen ≤ trials
minimizing a
accept in current state when
    makespan - a > 0 → {tabuLen := max(tabuLen-1, minLen); tabu[u, pm[u]] := trials;};
    always → {tabuLen := min(tabuLen+1, maxLen); tabu[u, pm[u]] := trials;};

```

Figure 7.10: An Approximation Based Job-shop Scheduling Statement.

```

Optimize
Constant:
  nbJ : int = ...;
  nbM : int = ...;
  N : int := nbJ × nbM;
  d : array[0..N + 1] of int = ...;
  m : array[1..N] of int = ...;
  F : {int} = ...;
  L : {int} = ...;
  minLen: int = ...;
  maxLen: int = ...;
  JB : array[i in 0..N + 1] = ...;
  seqj : array[i in 1..J] of Path(Nodes) := ...
  Arcsj : {Arc} := union(i in 1..J) seqj[i].arcs();
Variable:
  seqm : array[1..nbM] of Path(Nodes);
  tabu : array[1..N, 1..N] of int;
  tabuLen: int = ...;
Operator:
  void greedy() {
    F : {int};
    LM : array[i in 1..M] of int := 0;
    RR : int := N;
    forall(t in FO) insert(F, t);
    while RR -- > 0 do {
      choose OP from F minimizing D[OP];
      remove(F, OP);
      seqm[m[OP]].arc(LM[m[OP]], OP);
      LM[m[OP]] := OP;
      if (seqj[JB[OP]].succ(OP) <> N + 1)
        then insert(F, seqj[JB[OP]].succ(OP))
      endif;
    };
    forall(t in 1..M) seqm[t].arc(LM[t], N + 1);
    tlen := IT;
    forall(i in 0..N)
      forall(j in 0..N) {
        tabu[i, j] := -tlen; freq[i, j] := 0;
      };
  }

```

Figure 7.11: A Higher-Level Job-shop Scheduling Model.

```

Invariant:
  Arcsm : {Arc} := union(i in 1..M) seqm[i].arcs();
  Arcs : {Arc} := Arcsm union Arcsj;
  G: Graph := Graph(Nodes, Arcs);
  r : array[i in 0..N + 1] of int := longest(G, d, 0, i);
  q : array[i in 0..N + 1] of int := longest(G, d, i, N + 1);
  makespan : int := r[N + 1];
  Ca : {Arc} := {<s, t> : Arc | select <s, t> in Arcsm
                                where r[t] + q[t] = makespan and
                                      r[s] + d[s] = r[t] and
                                      s ≠ 0};

Objective Function: minimize makespan;
Neighborhood:
  move seqm[m[s]].swap(<s, t>)
  where <s, t> from Ca;
        a = appMakespan(s, t)
  such that tabu[s, t] + tabuLen ≤ trials
  minimizing a
  accept in current state when
        makespan - a > 0 → {tabuLen := max(tabuLen-1, minLen);
                             tabu[t, s] := trials;};
  always → {tabuLen := min(tabuLen+1, maxLen); tabu[t, s] := trials;};

```

Figure 7.12: A Higher-Level Job-shop Scheduling Model (continued).

are static) and the machine arcs (which are dynamic). The release dates and the tails can now be specified directly on the graphs as longest paths between nodes, i.e.,

```

r : array[i in 0..N + 1] of int := longest(G, d, 0, i);
q : array[i in 0..N + 1] of int := longest(G, d, i, N + 1);

```

Note also that *r* and *q* are now defined for the source and the sink, making the model more uniform. The makespan is simply the release date of the sink. The critical arcs can be specified directly by manipulating arcs, i.e.,

```

Ca : Arc({int}) := { <s, t> : Arc(int) | select <s, t> in Arcsm
                    where r[t] + q[t] = makespan and
                          r[s] + d[s] = r[t] and
                          s ≠ 0};

```

The Neighborhood Section The neighborhood section remains mostly unchanged but it now manipulates arcs directly.

Comparing the Models It is useful to step back and study the benefits of this new model. First, the model is closer to the informal description of the algorithm and it manipulates the underlying concepts directly. Second, the new description is much more flexible

and extensible. It is easy to generalize to non-pure job-shop scheduling problems, e.g., problems with additional precedence or distance constraints. No change to the invariants is necessary, since they are expressed directly in terms of the graph. The first model in contrast, would require changing the precedence relation, making the model less compact and less natural. Note also that these extensions do not induce any overhead. LOCALIZER essentially compiles these high-level invariants in terms of traditional invariants over sets and arrays.

7.4.6 Experimental Results

This section reports some preliminary experimental results of LOCALIZER on a set of classic benchmarks. The goal of this section is to provide evidence that LOCALIZER is a viable tool to implement scheduling algorithms too. We do not aim at comparing local and global search algorithms (which have different functionalities). Similarly, we did not try to produce the fastest implementation possible but rather to demonstrate the potential of LOCALIZER. The experiments were based on the parameters reported in [33]: the maximal number of searches (`maxSearches`) is 1, the maximal number of iterations for the inner loop (`maxTrials`) is 12000, the tabu list has a varying length constrained in between 5 and 30 and its length is updated according to the rule of [33]. Contrary to [33], our model does not use a restarting strategy.

Table 7.6 reports the preliminary results on a Sun Sparc Ultra 1 for the graph model (the first model is about 15% slower in the average). The table reports a coarse histogram that summarizes the frequencies of the solutions (i.e., the value of their makespan). `Loc` is the CPU time in seconds to run until completion, while `L0` is the time to run to completion or to a known optimal solution (this is the time measure used in [33]). Column `Avg.` gives the average makespan, while column `%O` gives the average distance to the optimum in percent. Results in italics indicates that the optimal solution is not known (as of 1993). The results indicate that the LOCALIZER statement obtains near-optimal solutions quickly. For problems where the optimum is known, the solutions are always within 6% of the optimum. In fact, the model finds optimal solutions for 14 benchmarks. The table indicates that these results are also obtained quickly: from 20 seconds to 90 seconds. These results cannot be really compared with the results of [33], since his neighborhood is an extension of the neighborhood presented here and the results are not directly comparable. But a rough comparison suggests that the difference between LOCALIZER and a specialized implementation is once again of the order of a machine generation.

Benchmark			Results (MD=12000,MS=1,Neighborhood=N1)												
Name :	J/M	Opt	Ranges					Freq.				LOC	LO	Avg.	%O
LA06	15/5	926	926	926	926	926	100	0	0	0	22.1	0.8	926.0	0	
LA07	15/5	890	890	890	890	890	100	0	0	0	24.5	3.1	890.0	0	
LA08	15/5	863	863	863	863	863	100	0	0	0	23.7	1.5	863.0	0	
LA09	15/5	951	951	951	951	951	100	0	0	0	22.7	1.2	951.0	0	
LA10	15/5	958	958	958	958	958	100	0	0	0	21.9	1.1	958.0	0	
LA11	20/5	1222	1222	1222	1222	1222	100	0	0	0	25.2	3.6	1222.0	0	
LA12	20/5	1039	1039	1039	1039	1039	100	0	0	0	24.0	3.1	1039.0	0	
LA13	20/5	1150	1150	1150	1150	1150	100	0	0	0	24.0	6.8	1150.0	0	
LA14	20/5	1292	1292	1292	1292	1292	100	0	0	0	22.8	2.4	1292.0	0	
LA15	20/5	1207	1207	1207	1207	1207	100	0	0	0	27.4	5.5	1207.0	0	
LA16	10/10	945	947	961	975	988	1	11	21	67	35.4	35.2	975.1	3	
LA17	10/10	784	784	790	796	801	17	74	8	1	36.1	34.2	786.4	0	
LA18	10/10	848	848	857	866	873	1	31	56	12	36.4	35.9	860.1	1	
LA19	10/10	842	843	850	857	864	1	25	49	25	37.0	36.5	853.9	1	
LA20	10/10	902	902	908	914	918	13	24	59	4	36.7	33.1	909.5	1	
LA21	15/10	1048	1060	1078	1096	1114	1	27	57	15	48.4	49.0	1084.9	3	
LA22	15/10	927	935	948	961	974	1	28	56	15	47.2	47.9	952.9	3	
LA23	15/10	1032	1032	1033	1034	1034	99	0	1	0	49.2	22.0	1032.0	0	
LA24	15/10	935	945	959	973	985	2	22	67	9	47.2	47.8	964.3	3	
LA25	15/10	977	989	1010	1031	1051	1	39	51	9	46.0	46.7	1015.1	4	
ABZ5	10/10	1234	1236	1246	1256	1264	3	34	54	9	36.8	38.4	1248.8	1	
ABZ6	10/10	943	943	948	953	958	13	69	16	2	37.4	37.4	946.9	0	
ABZ7	20/15	667	686	723	760	797	1	58	39	2	79.3	84.7	721.5	8	
ABZ8	20/15	678	688	724	760	796	1	6	70	23	83.3	80.7	747.7	10	
ABZ9	20/15	692	715	735	755	774	2	50	43	5	93.9	88.9	735.2	6	
MT6	6/6	55	55	55	55	55	100	0	0	0	13.2	1.0	55.0	0	
MT10	10/10	930	941	959	977	941	1	35	43	0	36.6	23.6	966.1	4	
MT20	20/5	1165	1173	1194	1215	1173	8	67	23	0	29.8	18.7	1186.1	2	
ORB1	10/10	1059	1073	1095	1117	1160	1	2	25	72	36.2	36.4	1124.6	6	
ORB2	10/10	888	889	896	903	917	3	33	42	22	36.1	36.3	899.6	1	
ORB3	10/10	1005	1021	1081	1141	1261	2	89	8	1	37.4	37.6	1060.5	6	
ORB4	10/10	1005	1019	1031	1043	1064	1	30	44	25	33.8	34.1	1037.3	3	
ORB5	10/10	887	899	911	923	947	1	29	40	30	37.5	37.7	918.7	4	
ORB6	10/10	1010	1022	1034	1046	1069	2	26	42	30	35.8	36.1	1041.8	3	
ORB7	10/10	397	397	403	409	420	1	5	47	47	38.5	38.3	409.5	3	
ORB8	10/10	899	914	932	950	986	1	9	54	36	37.5	37.8	947.8	5	
ORB9	10/10	934	934	945	956	976	1	3	35	61	32.4	32.7	959.8	3	
ORB10	10/10	944	944	956	968	992	2	24	48	26	37.0	36.6	963.5	2	

Table 7.6: Job-Shop Scheduling: Experimental Results.

7.5 The Vehicle Routing Problem

The vehicle routing problem (VRP) is central to transportation applications and is extensively studied in [73], [57], [32], [74], [24]. The traveling salesman problem is the subject of intensive research and has contributed heuristics to the VRP. The effort to express modern local search algorithms for VRP within LOCALIZER is important to demonstrate its potential on increasingly complicated problems. The algorithms implemented here as LOCALIZER statement are based on a generalization of the insertion heuristics of Clark-Wright [11], [69] and on a λ -interchange neighborhood identical to the work of Osman [43].

7.5.1 The Problem

The VRP considered in this section can be described as follows. A company must deliver goods to a number of clients using an unlimited number of trucks of a given capacity. Each truck should start from, and return to, the company's depot. Each client must be visited exactly once. The problem is to find tours for the trucks minimizing the total travel cost, while satisfying the capacity constraints.

More formally, a VRP instance is defined as a tuple $\langle G, D, R, vC \rangle$. $G(V, A)$ is the connection graph where $V = \{v_0, v_1, \dots, v_n\}$ with v_0 representing the depot shared by all the tours and $\{v_1, \dots, v_n\}$ representing the clients. A is the set of arcs $\{(v_i, v_j) : v_i, v_j \in V, i \neq j\}$ that denotes the existing inter-client links. D is a non-negative distance matrix that indicates the cost for going from client i to client j . Whenever $D^T = D$, the problem is said to be symmetric. R is a vector of integers denoting the amount of goods requested by each client. vC is an integer denoting the maximal truck capacity. A tour T_i is an ordered sequence $(v_0, v_{i_1}, \dots, v_{i_m}, v_0)$ where $v_{i_j} \neq v_0$ for all j in $1..m$ and each client appears at most once. A solution is a set of tours

$$\{T_1, T_2, \dots, T_k\}$$

that has the following properties

Unique visit Each client is visited at most once, i.e., $\forall i, j$ in $1..k$: $T_i \cap T_j = \{v_0\}$

Covering All clients are visited at least once, i.e., $\cup_{i=1}^k T_i = V$

Capacity Each truck carries less than its capacity, i.e., $\forall i$ in $1..k$: $\sum_{v \in T_i, v \neq v_0} R_v \leq vC$

The objective is to find a set of tours that minimizes the total distance traveled by the trucks. This VRP has been extensively studied in the context of local search.

The VRP inherits the experience drawn from the traveling salesman problem. Most, if not all, algorithms rely on neighborhoods that transform tours. The simplest type of heuristics for TSP e.g. vertex insertion or vertex displacement, are also used in VRP neighborhoods. Similarly, VRP neighborhoods often preserve satisfiability, i.e., the solution is always a set of tours that share the depot only.

The rest of the section is organized as follows: Section 7.5.2 introduces a standard algorithm used to solve the VRP with local search. Section 7.5.3 presents the LOCALIZER statement based on this neighborhood. Section 7.5.4 introduces an advanced neighborhood

for VRP that is used in “state of the art” implementations. Section 7.5.5 gives the statement for this second algorithm. Finally, Section 7.5.6 reports on the experimental results.

7.5.2 The Local Search Algorithm

Classic algorithms for VRP are often based on a neighborhood structure that preserves satisfiability. Consequently, these algorithms try to achieve two different goals. On one hand, the transformation itself must be rich enough to avoid poor local minima. On the other hand, the algorithm must assess within reasonable resource bounds the quality and the satisfiability of a potential neighbor. This section discusses a simple transformation, the criterion used to assess the cost of a solution, and techniques to focus on satisfiable neighbors.

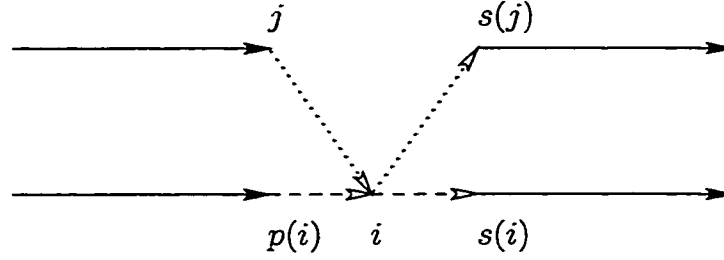
Notations Let *argmin* and *argmax* denote extension to the min and max function be defined as

$$\begin{aligned} m &= \operatorname{argmin}_{i \in S} F(i) \Leftrightarrow \text{any } m \in \{i \in S \mid F(i) = \min_{j \in S} F(j)\} \\ M &= \operatorname{argmax}_{i \in S} F(i) \Leftrightarrow \text{any } m \in \{i \in S \mid F(i) = \max_{j \in S} F(j)\} \end{aligned}$$

The Insertion Neighborhood A solution to the VRP is a set of tours that satisfy the three constraints described in the previous section. The insertion neighborhood is defined as the set of solutions that differ from the current solution in the location of a client. A solution S_2 is a neighbor of S_1 if S_2 was obtained from S_1 by applying an insertion transformation.

The insertion transformation selects a client i and relocates it on the same tour, or on a different one. The transformation thus simply removes i from its present location and inserts it between two adjacent clients at some other location if the new solution does not violate any constraint, i.e., the solution remains feasible. Figure 7.13 depicts this idea where client i is relocated between client j and $s(j)$. The dashed arcs are deleted, while the dotted arcs are inserted in the new solution.

The Quality Measurement The objective of the algorithm is to minimize the distance traveled. The quality of a solution is thus defined as the sum of the weights of each tour in the solution. The weight of a tour is the traditional weight used in the TSP, i.e., the sum of the distances connecting the vertices. More formally,

Figure 7.13: Detour to visit client i .

$$Cost = \sum_{i=1}^k \sum_{j \in T_i} D_{j,s(j)}$$

where $s(j)$ denotes the successor of vertex j in a tour.

Satisfiability The transformation described above guarantees that the first two sets of constraints, i.e., unique visit and covering are automatically satisfied. Indeed, the only constraint that could be violated as a result of moving client i is the capacity constraint. If client i is moved to a tour handled by a different truck, the addition of i 's request to this truck might cause an overload (removing i from its present site can only reduce the total quantity of goods carried by i 's truck).

Incremental Evaluation of the Cost The quality of an insertion can be computed incrementally. The saving associated to a transition is the sum of the saving for bypassing client i on the current tour and the cost for the detour initiated at client j to visit i . More formally, the saving $S(i)$ obtained when client i is bypassed (i.e., when the truck goes directly from $p(i)$ to $s(i)$). is simply

$$S(i) = D[p(i), i] + D[i, s(i)] - D[p(i), s(i)].$$

The cost $C(i, j)$ incurred when client i is inserted after client j is simply

$$C(i, j) = D[j, i] + D[i, s(j)] - D[j, s(j)].$$

For a given i , it is easy to identify $B(i)$, a minimal cost insertion point, as

$$B(i) = \operatorname{argmin}_{p \in V} C(i, p)$$

where V is the vertex set of the graph, i.e., the set of clients. Once an ideal insertion point is known, all that is left to define is the ideal i to relocate. The ideal value for i noted b is defined as

$$b = \operatorname{argmax}_{i \in V} S(i) - C(i, B(i))$$

The overall gain g associated to b simply is $S(b) - C(b, B(b))$

Preserving The Satisfiability As we have seen before, only the capacity constraints need to be verified. Each truck has a fixed maximal capacity, and a solution is invalid when the actual truck load exceeds the maximal capacity. The constraint can be easily verified when the set of clients that are part of the tour is available. Remember that a solution is

$$\{T_1, T_2, \dots, T_k\}$$

where the T_i are tours, i.e., sequences of clients. Clearly, each tour in the solution is assigned to a single truck. Let Cl_i denote the set of clients that appear in the tour T_i and let F_k be the truck identifier assigned to client k . Then Cl_i is defined as

$$Cl_i = \{k \in V | F_k = i\}.$$

It is now possible to define the load of a truck by summing up the requests of all the clients assigned to this truck. Let L_i , the load of truck i , be defined as

$$L_i = \sum_{k \in Cl_i} R_k$$

If $Fleet$ denotes the set of trucks used in the solution, i.e., the set of trucks delivering goods to at least one client, checking the capacity constraint for all the operational trucks reduces to verifying that the Boolean relation

$$Capacity \triangleq \sum_{i \in Fleet} (L_i > vC) = 0$$

holds. Consequently, the insertion neighborhood for the VRP can be defined as the set of feasible solutions that differ from the current solution in the placement of a single client. More formally,

$$N(s) = \{s' \in \mathcal{S} : s' = \operatorname{move}(s, i, j) \quad \forall i, j \in V, i \neq j \text{ s.t. } Capacity(s')\}$$

where \mathcal{S} is the set of solutions to the problem and $\operatorname{move}(s, i, j)$ denotes the relocation of i after client j in solution s .

7.5.3 The LOCALIZER Statement

We now illustrate how the local search algorithm described in the previous section can be expressed in LOCALIZER.

The Constant section The set of clients is given by an array C . Each element of C is a tuple that stores the client number, its Cartesian coordinates in the plane, and its product request. The record type is defined as

```

type: Customer = record
    id : int;
    x : real;
    y : real;
    d : int;
end;

```

The section is also responsible for pre-computing data structures used throughout the statement. The matrix D and the vector R are computed from the client array C . We simply assume that distances are Euclidean distances in the plane in this model but this is not restrictive of course. In addition to D , the constant section also computes two sets of arcs AP and AS . AP stores the set of all arcs (i, j) verifying $j < i$ where i and j are clients. AS stores the set of all arcs.

$$\begin{aligned}
 AP &= \{ \langle i, j \rangle \mid i, j \in [1, nbC] \wedge j < i \} \\
 AS &= \{ \langle i, j \rangle \mid i, j \in [1, nbC] \wedge i \neq j \}
 \end{aligned}$$

Finally, the constant array Cli associates with any arc $\langle i, j \rangle$, a valid client number, i.e., a number between 1 and nbC . The need for Cli arises from the presence of arcs of the form $\langle 0, i \rangle$ that encode the first arc of a tour. Cli associates with such arcs the endpoint i that denotes either a valid client number of $N + 1$. For the other arcs the value of $Cli[\langle i, j \rangle]$ is, by convention, the endpoint of the arc (j) .

The State Representation The natural choice for the state variables is an array of Path instances where each path represents a tour. An auxiliary array of variables is useful to keep track of which truck is assigned to each client. This array plays an essential role in the definition of the satisfiability. The variables of the statement are

```

tours : array[1..nbC] of Path(Nodes);
truck : array[1..nbC + 1] of int;

```

where *Nodes* denotes the set of all clients. For convenience, nodes 0 and $nbC + 1$ respectively correspond to the starting and ending point of any tour. This convention proves useful to record the fact that a truck does not leave the depot (the corresponding path is $(0, nbC + 1)$). Note that $truck[nbC + 1]$ contains the identifier of an empty truck and is used when a new tour is created by inserting a client between node 0 and $nbC + 1$.

The Invariant section In order to describe the neighborhood, it is important to restrict the search to the pairs of clients yielding a solution. A solution is said to be satisfiable if and only if no vehicle is overloaded and the set of tours is node and arc disjoint with respect to the set of clients (they obviously share the depot nodes, i.e., 0 and $nbC + 1$).

The first step is to define the set of clients visited by each truck, the load of each truck and the set of empty trucks. This can be expressed by the invariants

```

trip : array[0..nbC + 1] := distribute(truck, {1..nbC}, {0..nbC + 1});
load : array[i in 0..nbC + 1] of int := sum(j in trip[i]) C[j].d;
et : {int} := { i : int | select i from 0..nbC + 1 where size(trip[i]) = 0 };
fe : int := minof(et);

```

The invariants *fe* that selects an empty truck is useful since it may be necessary to create a new tour by inserting a client on the arc $(0, nbC + 1)$. The truck load definition is complemented with the definition of the total cost for all tours. This is required by the objective function of the statement. It is defined in terms of the set of arcs that are part of the tours.

```

ta : {Arc} := union(i in 1..nbC) tours[i].arcs();
totalCost : real := sum(a in ta) D[a.s, a.t];

```

From the *tours* array it is also convenient to define the predecessor and the successor for each client. Note that the arrays *s* and *p* could be avoided at the expense of a less readable statement.

```

p : array[i in 1..nbC] of int := tours[truck[i]].pred(i);
s : array[i in 0..nbC + 1] of int := if i in {1..nbC} then tours[truck[i]].succ(i) else 0;

```

Once these invariants are defined, it is possible to spell out the properties that guarantee the satisfiability of a move. Intuitively, moving client *i* after client *j* is valid if $i \neq j$ and $i \neq s(j)$ and the capacity of the truck visiting *j* can take the additional demand of client *i*. Given that, for each client, the truck variable denotes the truck assigned to that client, it is possible to check whether the truck assigned to client *j* can accommodate the new request. This is achieved in LOCALIZER with the invariant

```

val : array[i in 1..nbC] of {Arc} := {a : Arc | select a from ra
      where load[truck[Cli[a]]] + R[i] ≤ vC and a.t ≠ i and a.s ≠ i};

```

Finally, the equations defining $B(i)$ and b can be adapted to obtain the invariants

```

bi : array[i in 1..nbC] of Arc := argmin(k in val[i]) D[k.s, i] + D[i, k.t] - D[k.s, k.t];
cbi array[i in 1..nbC] of real := min(k in val[i]) D[k.s, i] + D[i, k.t] - D[k.s, k.t];
canMove : {int} := {i : int | select i from 1..nbC where size(val[i]) > 0 };
gi : int      := argmax(i in canMove) D[p[i], i] + D[i, s[i]] - D[p[i], s[i]] - cjStar[i];
g : real     := max(i in canMove) D[p[i], i] + D[i, s[i]] - D[p[i], s[i]] - cjStar[i];

```

A Tabu Search Heuristic for the Insertion Neighborhood The statement is almost complete since only the transformation procedure and the strategy itself are missing. The search heuristic is easy to define since most of the burden is now in the invariant section. Indeed, the strategy of choice is tabu search and this is a greedy procedure. Tabu prescribes the selection of the best non tabu element of the neighborhood. The invariants are already defined to maintain incrementally this best element. All that is required is the addition in the definition of *canMove* of a conjunct that filters tabu moves. The following excerpt shows the new and affected invariants

```

notTabu : array[i in 1..nbC] of boolean := trials > tabu[i] and
      freq[i] * 100 / (trials + 1) < 10;
canMove : {int} := {i : int | select i from 1..nbC
      where size(val[i]) > 0 and notTabu[i]};

```

A client is not considered tabu in this context whenever it has not moved recently and when it has not moved more than 10% of the time. Whenever a client actually moves from one location to another, its tabu number is set to *trials* + *tabuLen* and thus prevents the client from moving for the next *tabuLen* iterations. The accept statement of the neighborhood is also used to vary the length of the tabu list. Whenever the algorithm performs improving moves, it shortens the tabu list. On the contrary, when the algorithm is in a local minima, it increases the length of the tabu list. *tabuLen* is bounded from above and below. Figures 7.14 and 7.15 give the complete LOCALIZER statement.

Starting Point The last component of the model is the initialization routine. The objective of the starting point is twofold. On one hand, the routine must support diversification for the search procedure. Several restarts should be initiated from different points. On the other hand, the routine should attempt to find a satisfiable, and reasonably good, starting point. At one extreme, one can initialize the search with a set of $|V|$ tours, each of them

```

constant:
  nbC : int = ...;
  opt : real = ...;
  vC : int = ...;
  C : array[0..nbC] of Customer = ...;
  Nodes : Node := {0..nbC + 1};
  D : array[i in 0..nbC+1, j in 0..nbC+1] of real := sqrt((C[i].x - C[j].x)2 +
                                                         (C[i].y - C[j].y)2);

  AP : {Arc} := {{z1, z2} : Arc | select z1 from 1..nbC select z2 from 1..z1 - 1};
  AS : {Arc} := {{z1, z2} : Arc | select z1 from 1..nbC select z2 from 1..nbC};
  Cli : array[a in AS] of int := if a.s = 0 then a.t else a.s;
  tmax : int = 10;
  tmin : int = 1;

variable:
  tours : array[1..nbC] of Path(Nodes);
  truck : array[1..nbC + 1] of int;
  tabu : array[1..nbC] of int;
  freq : array[1..nbC] of int;
  tabuLen : int;

operator:
  void initSol() {
    tabuLen := tmin;
    forall(i in 1..nbC) { tabu[i] := -1; freq[i] := 0; };
    F : {int} = {1..nbC}; MW: int := 0; l : int := 0; R : int := 1;
    while size(F) > 0 do {
      choose V1 from F;
      MW := C[V1].d;
      tours[R].arc(0, V1);
      truck[V1] := R; l := V1;
      remove(F, V1);
      while size(F) > 0 and MW < vC do {
        choose V from F minimizing Dist[l, V];
        MW := MW + C[V1].d;
        if MW < vC then
          tours[R].arc(l, V);
          truck[V] := R; l := V;
          remove(F, V);
        } endif;
      };
      tours[R].arc(l, nbC + 1);
      R ++;
    };
    truck[nbC + 1] := R;
    while R < nbC + 1 do tour[R ++].arc(0, nbC + 1);
  }

```

Figure 7.14: Vehicle routing model. Insertion neighborhood, Part One.

```

void insertion(i : int, a : Arc){
    r : int := Cli[a];
    tours[truck[i]].noarc(p[i], i);
    tours[truck[i]].noarc(i, s[i]);
    tours[truck[r]].noarc(a.s, a.t);
    tours[truck[i]].arc(p[i], s[i]);
    tours[truck[r]].arc(a.s, i);
    tours[truck[r]].arc(i, a.t);
    truck[i] := truck[r];
    truck[nbC + 1] := fe;
}

Invariant:
ta : {Arc} := tours.arcs();
totalCost : real := sum(a in ra) D[a.s, a.t];
trip : array[0..nbC + 1] := distribute(truck, {1..nbC}, {0..nbC + 1});
load : array[i in 0..nbC + 1] of int := sum(j in trip[i]) C[j].d;
val : array[i in 1..nbC] of {Arc} := {a : Arc | select a from ra
    where load[truck[Cli[a]]] + R[i] ≤ vC and a.t ≠ i and a.s ≠ i};
p : array[i in 1..nbC] of int := tours[truck[i]].pred(i);
s : array[i in 0..nbC + 1] of int := if i in {1..nbC} then tours[truck[i]].succ(i) else 0;
cbi : array[i in 1..nbC] of real := min(k in val[i]) D[k.s, i] + D[i, k.t] - D[k.s, k.t];
bi : array[i in 1..nbC] of Arc := argmin(k in val[i]) D[k.s, i] + D[i, k.t] - D[k.s, k.t];
notTabu : array[i in 1..nbC] of boolean := trials > tabu[i] and
    freq[i] * 100 / (trials + 1) < 10;
canMove : {int} := {i : int | select i from 1..nbC
    where size(val[i]) > 0 and notTabu[i]};
gi : int := argmax(i in canMove) D[p[i], i] + D[i, s[i]] - D[p[i], s[i]] - cbi[i];
g : real := max(i in canMove) D[p[i], i] + D[i, s[i]] - D[p[i], s[i]] - cbi[i];
neighborhood:
    try when size(canMove) > 0:
        move insertion(gi, bi[gi])
        accept when in current state always → {
            if g < 0 and tabuLen > tmin then tabuLen -- endif;
            if g ≥ 0 and tabuLen < tmax then tabuLen ++ endif;
            tabu[gi] := trials + tabuLen;
            freq[gi] ++;
        }
    end

```

Figure 7.15: Vehicle routing model. Insertion neighborhood, Part Two.

with a single client. At the other extreme, a greedy approach can attempt to keep the inter-client distances to a minimum while not overloading any truck. A trade-off between starting point quality and randomization needs to be found.

The model described here relies on the operator `initSol` which computes a random, and reasonably good starting point. The procedure first initializes the arrays *tabu* and *freq* used for the tabu search. The second portion builds tours as long as un-routed customers exist. To build a tour, the procedure randomly selects a client i from the set F of un-routed clients and creates a tour that starts with client i . The subsequent clients are then added in a greedy fashion until the truck is filled at which point the tour is closed and the next tour is initiated.

7.5.4 A More Advanced Algorithm: λ -interchange

This section presents a more sophisticated neighborhood based on λ -interchange.

The λ -interchange introduced by Osman in [42] is described as the exchange of two segments extracted from the set of tours. Each segment consists of a sequence of zero to λ clients. The two segments must be node-disjoint. It is possible to extract the two segments from the same tour or from different tours. The neighborhood N_λ induced by λ -interchange is endowed with a wealthier collection of moves than the previous neighborhood. The insertion neighborhood described earlier is just a special case when an empty segment starting at a given client is exchanged against a segment of length 1.

The size of N_λ is quickly prohibitive since it is $O(n^2(\sum_{i=0}^{\lambda+1} i(i-1)))$, where n is the number of clients. For computational reasons, most algorithms for VRP use $\lambda = 2$. Still, a 2-interchange neighborhood contains 5 types of moves depicted in Figure 7.16 and described in Table 7.7.

7.5.5 The LOCALIZER Statement

The constant section and the variable section are identical to those used in the insertion based statement. The Invariant section is naturally more involved for λ -interchange. This section presents the statement for λ -interchanges.

The Neighborhood section To implement such an algorithm in LOCALIZER, it is necessary to use the `try` instruction. Each type of move is carried out by a specific neighborhood. The neighborhood supposed to execute a given transition is selected with a `when` clause.

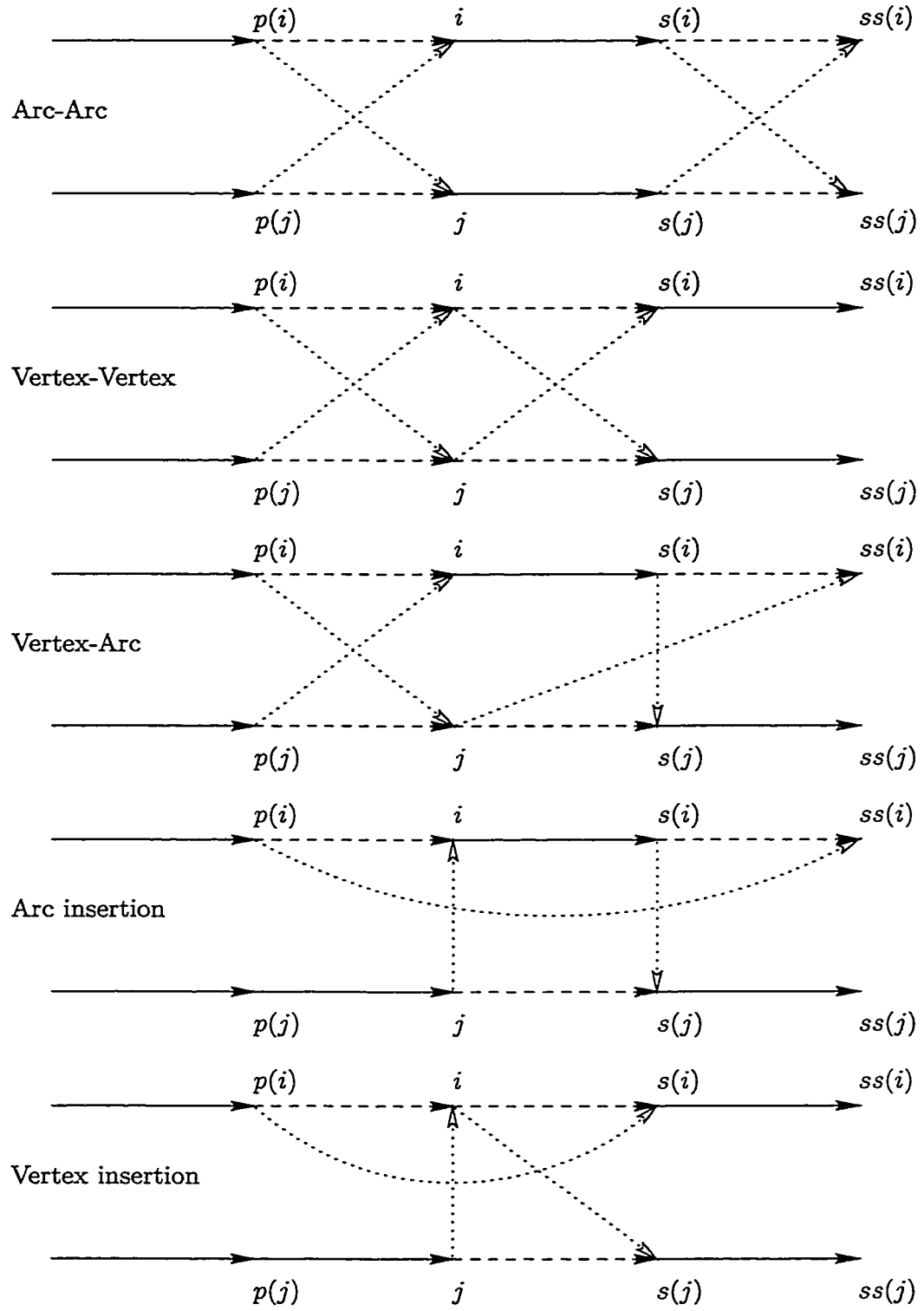


Figure 7.16: Moves allowed by a 2-interchange.

Operation	Description
Arc-Arc	A pair of consecutive clients is exchanged against another pair coming from the same tour or from a different one. Thus, N_2 subsumes the familiar 2-opt moves of TSP.
Vertex-Vertex	A single client site is exchanged against another client.
Arc-Vertex	A pair of consecutive clients is exchanged against a single client site. The length of the two tours involved is thus modified. It is also possible to perform this type of relocation with a single tour.
Arc Insertion	A pair of consecutive clients is extracted from its present location and inserted on some other tour.
Vertex Insertion	This move is the neighborhood transformation introduced at the beginning of the section.

Table 7.7: Description of 2-interchange moves.

New invariants must also be defined to assess the feasibility of each transition for the various type of moves together with the related gains. Finally, a last invariant defines the largest possible gain among all five types of moves. The condition that appears in the when clause reduces to a test of the best gain for this type of move against the best overall gain. If they match, the corresponding neighborhood is selected and the transition is performed.

The Revised Invariant section The invariants related to the insertion based neighborhood can be directly recycled in this new model. The new moves are all supported with invariants based on the same skeleton. Consider, for instance, the arc-arc exchange invariants.

A validity matrix first establishes for all (i, j) the validity of the move given that i and j are the first clients of the two segments of the lambda exchange.

```

aaxv : array[i in 1..nbC, j in 1..nbC] of boolean := if j >= i then false else
    s[i] <> nbC + 1 and s[j] <> nbC + 1 and
    j <> s[i] and p[j] <> s[i] and i <> s[j] and p[i] <> s[j] and
    load[truck[i]] - C[i].d - C[s[i]].d + C[j].d + C[s[j]].d <= vC and
    load[truck[j]] - C[j].d - C[s[j]].d + C[i].d + C[s[i]].d <= vC;

```

Since a move is symmetric, the values of the validity matrix are restricted to the lower triangle of *aaxv*. Given that the relocated clients are i , $s(i)$, j and $s(j)$, neither $s(i)$ nor $s(j)$ can denote the depot. When the clients involved in a move belong to the same tour, it is illegal to have

$$\begin{aligned}
 j &= s(i) \\
 p(j) &= s(i)
 \end{aligned}$$

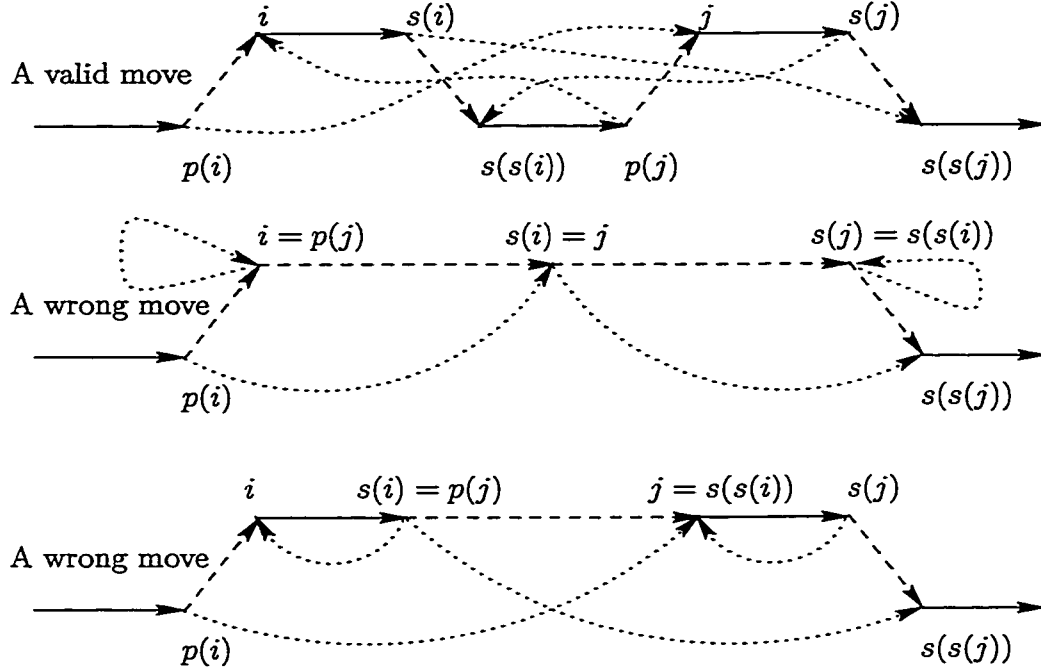


Figure 7.17: Arc-arc exchange validity.

since it would make the exchange ill-fated as demonstrated in Figure 7.17. Naturally, the symmetric situation holds true. The last two components of the predicate make sure that the exchange does not violate the vehicle capacity constraint and their definitions immediately follow from the truck load and the requests of the clients.

A second matrix *aax* defines for all (i, j) the gain for the move. Its definition follows from the arcs removed and the arcs inserted back in the tours.

```

aax  : array[i in 1..nbC, j in 1..nbC] of real := if j = i then 10000000.0 else
      D[p[j], i] + D[s[i], ss[j]] + D[p[i], j] + D[s[j], ss[i]] -
      (D[p[i], i] + D[s[i], ss[i]] + D[p[j], j] + D[s[j], ss[j]]);

```

The last two invariants define the maximal gain over all the valid moves and identify a responsible pair.

```

aas  : Arc := {⟨q, w⟩ : Arc | select ⟨q, w⟩ from AP
               where aaxv[q, w] and notTabu[q]};
aa   : Arc := argmin(a in aas) aax[a.s, a.t];
aag  : real := - (min(a in aas) aax[a.s, a.t]);

```

The *aas* invariant maintains the set of arc-arc exchanges that are valid and non-tabu, i.e. result in a satisfiable solution. Finally, *aa* and *aag* identify the best arc-arc exchange and its

saving. The definitions are straightforward since all that is left to do is pickup the maximal element in the *aax* matrix among the valid entries, i.e. those that belong to *aas*.

Each type of move, results in a set of invariant similar to those described for the arc-exchange. Figure 7.18 gives all the new invariants while Figure 7.19 illustrates the neighborhood section for this statement. The complete description of the statement is augmented with the operators that appear in Figure 7.19 and are responsible for performing each type of move. Each operator is similar to the *insertion* routine presented in the previous models and copes with the arcs insertions, arcs removal and truck assignment. Figure 7.16 gives a concise specification of each operator where the dotted arrows are the new arcs while the dashed arrows are the deleted arcs.

7.5.6 Experimental Results

This section reports some experimental results obtained for the capacitated VRP on several standard instances in the literatures. The objective is to demonstrate that even on complex problems such as the VRP, LOCALIZER can be competitive with special purpose implementations. The values reported here are for 15 benchmarks from Christofides, Mingozzi & Toth [10] (C series), Fisher [17] (f series) and Taillard [73] (tai series). Table 7.8 reports the result for the insertion neighborhood discussed above, while 7.10 reports the results for the more advanced 2-interchange neighborhood. Both tables use the same format. The first three columns list the average, worst and best solution found in 100 consecutive identical executions. The next three column list the average, maximum and minimum running time on an Ultra Sparc 1 running Solaris 2.6. The *B.K.* column gives the best solution reported in the literature and the last column gives the distance (as a percentage) of the best solution produced by LOCALIZER to the best-known solution. Both series of runs were conducted with the same parameter settings for Tabu search. The number of iteration is capped at 2000 and the tabu list can vary in length from 1 to 10.

Table 7.8 gives surprisingly good results especially when the running time is taken into account and compared to the running time for the 2-interchange neighborhood. It is important to notice that the tabu search with insertion neighborhood is a quasi-deterministic algorithm. The only sources of randomization are the tie-breaking mechanisms when several moves are equally appealing and the randomized starting point. We noticed that ties happen very infrequently, therefore the diversification introduced with starting points is essential to get good results.

Finally, note that the statement for the insertion neighborhood is slightly more restrictive than necessary. Indeed, the description of the neighborhood indicates that the relocation of a client i after some client j is valid even when j belongs to the same route. The definition of the invariant val

$$val : \text{array}[i \text{ in } 1..nbC] \text{ of } \{\text{Arc}\} := \{a : \text{Arc} \mid \text{select } a \text{ from } ra \\ \text{where } load[truck[Cli[a]]] + R[i] \leq vC \text{ and } a.t \neq i \text{ and } a.s \neq i\};$$

eliminates some move of this type since the relocation of i after client j is always conditional to $load[truck[Cli[a]]] + R[i] \leq vC$ even when i and j are on the same tour. This apparent oddity was the result of a modeling error but was preserved for a simple reason: the algorithm does a lot worse, quality wise, when the definition

$$val : \text{array}[i \text{ in } 1..nbC] \text{ of } \{\text{Arc}\} := \{a : \text{Arc} \mid \text{select } a \text{ from } ra \\ \text{where } (truck[i] = truck[Cli[a]] \text{ or} \\ load[truck[Cli[a]]] + R[i] \leq vC) \\ \text{and } a.t \neq i \text{ and } a.s \neq i\};$$

is used, the results significantly degrades. This phenomenon is explained by the presence of many moves that slightly improve the objective function. The presence of these moves overrides other transitions that appear worse but have the advantage to create major changes in solutions thereby introducing a good diversification mechanism. Indeed, a modification of val , to explicitly eliminate all client relocation within a tour, with the definition

$$val : \text{array}[i \text{ in } 1..nbC] \text{ of } \{\text{Arc}\} := \{a : \text{Arc} \mid \text{select } a \text{ from } ra \\ \text{where } (truck[i] <> truck[Cli[a]] \text{ and} \\ load[truck[Cli[a]]] + R[i] \leq vC \\ \text{and } a.t \neq i \text{ and } a.s \neq i\};$$

gives results similar to what is reported in this thesis.

We are not aware of any other experimental results for the insertion neighborhood. The Clark-Wright procedure that was generalized to obtain the insertion neighborhood is often used to compute starting points but was not considered on its own. Competitive implementations for the capacitated VRP were realized by Osman [43] and Gendreau, Hertz and Laporte [21]. These implementations are based on the more intricate 2-interchange neighborhood. The BA variant of Osman's algorithm looks for the best possible move in the 2-interchange neighborhood, while the FBA variant attempts to cut on exploration costs by settling for the first improving move. The results reported in Table 7.9 are the *best* solutions ever found by those algorithms over a range of parameters and executions.

Ins.	Avg	Max	Min	Avg	Max	Min	B.K.	%
C50	541.63	594.24	525.92	66.93	71.80	61.30	524.61	0.002
C75	870.49	912.57	844.97	97.11	196.00	61.80	835.26	0.012
F71	269.07	301.37	242.22	73.36	106.20	56.50	241.97	0.001
TAI75c	1332.61	1544.53	1179.19	94.89	111.60	70.10	1291.01	-0.087
TAI75d	1374.03	1498.95	1289.50	72.18	88.10	57.40	1365.42	-0.056
C100	878.07	965.62	835.51	173.96	213.40	111.70	826.14	0.011
C100b	881.03	1043.07	827.96	127.75	176.20	92.10	819.56	0.010
C120	1129.86	1328.56	1075.43	190.47	289.20	61.50	1042.11	0.032
C150	1137.50	1214.85	1058.30	266.78	329.70	179.40	1028.42	0.029
C199	1450.72	1571.56	1370.50	377.59	488.50	262.10	1291.45	0.061
F134	13213.53	14896.40	11999.70	237.82	350.10	134.90	11629.90	0.032
TAI100a	2241.12	2554.67	2115.71	136.10	166.10	112.80	2047.90	0.033
TAI100b	2071.74	2224.78	1972.61	133.14	163.90	91.80	1940.61	0.016
TAI100c	1540.02	1745.35	1456.48	155.56	174.40	130.50	1407.44	0.035
TAI100d	1731.95	1908.70	1632.48	152.48	183.80	115.10	1581.25	0.032

Table 7.8: Insertion neighborhood for VRP.

	Osman BA ²		Osman FBA ³		Taburoute ⁴		Taillard ⁵	
	Q	T	Q	T	Q	T	Q	T
C50	524.61	67.2	524.61	114	524.61	360	524.61	-
C75	844	70.8	844	50.3	835.77	3228	835.26	-
C100	835	675	838	1543	829.45	1104	826.14	-
C100b	819.59	407.5	819.59	892.2	819.56	960	819.56	-
C120	1042.11	1398.4	1043	1445.4	1073.47	1332	1042.11	-
C150	1052	3075	1044.35	3560.0	1036.16	3528	1028.42	-
C199	1354	1972.7	1334.55	3246	1322.65	5454	1298.79	-

Table 7.9: Modern VRP Implementations

Ins.	Avg	Max	Min	Avg	Max	Min	B.K.	%
C50	549.86	1422.30	526.83	379.05	498.10	244.50	524.61	0.004
C75	999.86	14336.10	837.66	859.94	1223.80	141.90	835.26	0.003
F71	269.26	311.54	244.92	850.00	1302.20	451.40	241.97	0.012
TAI75c	1428.55	1558.04	1353.19	811.68	1395.70	525.50	1291.01	0.048
TAI75d	1439.56	1520.07	1384.55	729.63	1080.20	508.60	1365.42	0.014
C100	878.43	950.06	839.50	2474.74	3766.80	1511.60	826.14	0.016
C100b	864.73	1033.08	828.51	2592.59	3654.70	1368.20	819.56	0.011
C120	1119.00	1219.89	1082.71	2724.79	4506.10	1568.80	1042.11	0.039

Table 7.10: 2-Interchange Neighborhood

```

ss: array[i in 1..nbC] of int := s[s[i]];

aarv: array[i in 1..nbC, j in 1..nbC] of boolean := if j >= i then false else
    s[i] <> nbC + 1 and s[j] <> nbC + 1 and
    load[truck[i]] - C[i].d - C[s[i]].d + C[j].d + C[s[j]].d <= vC and
    load[truck[j]] - C[j].d - C[s[j]].d + C[i].d + C[s[i]].d <= vC and
    j <> s[i] and p[j] <> s[i] and i <> s[j] and p[i] <> s[j];
aax : array[i in 1..nbC, j in 1..nbC] of real := if j = i then 10000000.0 else
    D[p[j], i] + D[s[i], ss[j]] + D[p[i], j] + D[s[j], ss[i]] -
    (D[p[i], i] + D[s[i], ss[i]] + D[p[j], j] + D[s[j], ss[j]]);
aas : Arc := {(q, w) : Arc | select (q, w) from AP where aarv[q, w] and notTabu[q]};
aa : Arc := argmin(a in aas) aax[a.s, a.t];
aag : real := - (min(a in aas) aax[a.s, a.t]);

vvrv: array[i in 1..nbC, j in 1..nbC] of boolean := if j >= i then false else
    load[truck[i]] - C[i].d + C[j].d <= vC and
    load[truck[j]] - C[j].d + C[i].d <= vC and
    j <> s[i] and p[i] <> j;
vvx : array[i in 1..nbC, j in 1..nbC] of real := if j >= i then 100000000.0 else
    D[p[j], i] + D[i, s[j]] + D[p[i], j] + D[j, s[i]] -
    (D[p[i], i] + D[i, s[i]] + D[p[j], j] + D[j, s[j]]);
vvs : Arc := {(q, w) : Arc | select (q, w) from AP where vvrv[q, w] and notTabu[q]};
vv : Arc := argmin(a in vvs) vvz[a.s, a.t];
vvg : real := - (min(a in vvs) vvz[a.s, a.t]);

avrv: array[i in 1..nbC, j in 1..nbC] of boolean := if j = i then false else
    s[j] <> nbC + 1 and j <> s[i] and p[i] <> j and p[i] <> s[j] and
    load[truck[i]] - C[i].d + C[j].d + C[s[j]].d <= vC and
    load[truck[j]] - C[j].d - C[s[j]].d + C[i].d <= vC;
avx : array[i in 1..nbC, j in 1..nbC] of real := if j = i then 10000000.0 else
    D[p[j], i] + D[i, ss[j]] + D[s[j], s[i]] + D[p[i], j] -
    (D[p[i], i] + D[i, s[i]] + D[p[j], j] + D[s[j], ss[j]]);
avs : Arc := {(e, r) : Arc | select (e, r) from AS where avrv[e, r] and notTabu[e]};
av : Arc := argmin(a in avs) avx[a.s, a.t];
avg : real := - (min(a in avs) avx[a.s, a.t]);

aiv : array[i in 1..nbC, j in 1..nbC] of boolean := if j = i then false else
    load[truck[j]] + C[i].d + C[s[i]].d <= vC and
    j <> s[i] and p[i] <> j and s[i] <> nbC + 1;
ai : array[i in 1..nbC, j in 1..nbC] of real := if j = i then 100000000.0 else
    D[j, i] + D[s[i], s[j]] + D[p[i], ss[i]] - (D[p[i], i] + D[s[i], ss[i]] + D[j, s[j]]);
ais : Arc := {(q, w) : Arc | select (q, w) from AS where aiv[q, w] and notTabu[q]};
ai : Arc := argmin(a in ais) ai[a.s, a.t];
aig : real := - (min(a in ais) ai[a.s, a.t]);
besti: real := maxd(aag, maxd(vvg, maxd(avg, maxd(aig, g))));

```

Figure 7.18: Invariants for the 2-interchange VRP model.

```

operator:
  void markTabu(a1 : int,a2 : int,oBest : real) {
    if tt < oBest and tabuLen > tmin then tabuLen -- endif;
    if tt >= oBest and tabuLen < tmax then tabuLen ++ endif;
    tabu[a1] := trials + tabuLen;
    tabu[a2] := trials + tabuLen;
    freq[a1] ++;
    freq[a2] ++;
  }
neighborhood:
  try
    when size(aas)> 0 and besti = aag:
      move lambda2(a1,a2)
      where a1 = aa.s;
            a2 = aa.t;
            oBest = tt;
      accept when in current state always → markTabu(a1,a2,oBest);
    when size(vvs)> 0 and besti = vvg:
      move lambda1(a1,a2)
      where a1 = vv.s;
            a2 = vv.t;
            oBest = tt;
      accept when in current state always → markTabu(a1,a2,oBest);
    when size(avs)> 0 and besti = avg:
      move lambda12(a1,a2)
      where a1 = av.s;
            a2 = av.t;
            oBest = tt;
      accept when in current state always → markTabu(a1,a2,oBest);
    when size(ais)> 0 and besti = aig:
      move insertion2(a1,a2)
      where a1 = ai.s;
            a2 = ai.t;
            oBest = tt;
      accept when in current state always → markTabu(a1,a2,oBest);
    when size(canMove)> 0:
      move insertion(i,jStar[i])
      where i = ig;
            oBest = tt;
      accept when in current state always → {
        if tt < oBest and tabuLen > tmin then tabuLen -- endif;
        if tt >= oBest and tabuLen < tmax then tabuLen ++ endif;
        tabu[i] := trials + tabuLen;freq[i] ++;
      }
  end

```

Figure 7.19: Neighborhood for the 2-interchange VRP model.

Chapter 8

Modeling in LOCALIZER

The purpose of this chapter is to investigate a number of modeling issues that arise in implementing local search algorithms in LOCALIZER. It discusses various ways of exploiting incrementality in the neighborhood and in the invariants and illustrates how simple complexity analysis can be performed to choose between different LOCALIZER statements.

This chapter only discusses implementation issues relevant to LOCALIZER. It does not discuss the design of local search algorithms. This topic deserves a full treatment on its own, since it involves numerous tradeoffs between the quality of the solutions and the efficiency of the algorithms. Interested readers should consult [1] for more information on this topic.

The chapter is organized as follow: Section 8.1 discusses incrementality issues that are central to the development of efficient LOCALIZER models. Section 8.2 shows how space and time analysis can be used to choose between different LOCALIZER statements. Section 8.3 discusses the impact of a formulation on the constant factors hidden in the big O notation. Section 8.4 reviews how standard database techniques apply in LOCALIZER.

8.1 Incrementality

Incrementality is a critical issue to obtain efficient LOCALIZER statements. This section reviews a number of techniques available in LOCALIZER and analyzes them in this respect.

8.1.1 Simulation Versus Differentiation

The basic step of a local search consists of moving to an element in the neighborhood. The choice of which neighbor to select is, in general, guided by the objective function of LOCALIZER that can be viewed as a function $f : \tau \rightarrow \mathbb{R}$, mapping a computation state into

a real value.

Typical exploration strategies (e.g. local improvement, tabu search, or simulated annealing) need to consider various neighbors and evaluate the objective function for each of them. Consider, for instance, a local improvement strategy. If τ is the current state and τ' is the state associated with a neighbor, the difference $\Delta = f(\tau) - f(\tau')$ is used to determine whether to take the move. This decision requires either an explicit computation of τ' from which Δ can be easily computed or a way to evaluate Δ in the current state. The former technique is called simulation, while the latter is called differentiation. We review both of them in some detail.

8.1.1.1 Simulation

Simulation evaluates the objective function in a neighboring state. It suffices to use the Objective Function section to define the function f and the type of optimization problem considered. The expression defining f can refer to any constant, variable, or invariant defined in the statement. Whenever a neighbor is considered by the `move` instruction, the computation state τ' is produced and the function $f(\tau')$ is evaluated. The value $\Delta = f(\tau) - f(\tau')$ is available through a variety of keywords (e.g. `delta`, `improvement`, `noDecrease`) that can be used in the acceptance criterion and many strategies such as local improvement and simulated annealing can be implemented easily. Greedy strategies such as tabu search use the value $f(\tau')$ and simply choose the best one. They are implemented using the keyword `best` in the neighborhood definition.

Simulation is the default technique in LOCALIZER and is especially useful early in the design of local search algorithms where experimentation plays a major role. When a given neighborhood is chosen, it may be useful to consider differentiation to improve efficiency. Indeed, the cost of evaluating a new state may be significant, since it amounts to updating all the invariants.

8.1.1.2 Differentiation

Differentiation consists of deriving an expression or a piece of code that computes the same quantity Δ but in the current state τ (not through simulation). For instance, consider the graph coloring application. The variation in the objective function is induced by a single assignment of a value k to an integer variable x_i . In this application, the objective function is defined as

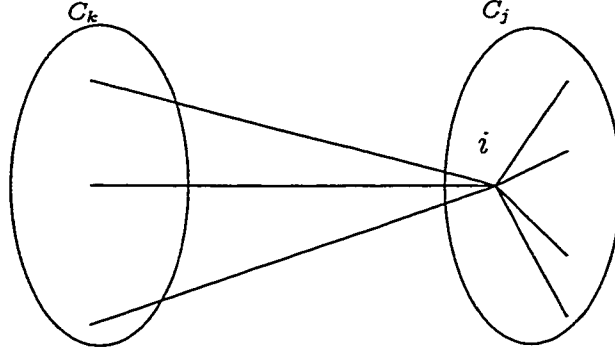


Figure 8.1: Moving vertex i from class j to class k .

$$F(\tau) = \sum_{i=1}^n (2|B_i||C_i| - |C_i|^2)$$

where C_i is the set of vertices colored with color i and B_i is the set of edges connecting two vertices in the class i .

The color change for vertex i from j to k implies changing both C and B . More specifically, the color class C_j loses one vertex while C_k gains one (as long as $j \neq k$). Additionally, the sets B_j and B_k are also affected. When vertex i moves, the edges that connect i to other vertices in the class j lose their bad-edge status. On the other hand, all the edges that connect vertex i to vertices in class k become bad edges as depicted in Figure 8.1. More formally, the cardinality nb_i of the set of edges connecting vertex i to vertices in class k and the cardinality ob_i of the set of edges connecting vertex i to vertices in class j may be defined as follows:

$$\begin{aligned} ob_i &= |\{\langle a, b \rangle | a = i \wedge b \in C_j\}| \\ nb_i &= |\{\langle a, b \rangle | a = i \wedge b \in C_k\}| \end{aligned}$$

The assignment of k to x_i implies the following identities

$$\begin{aligned} \tau'(|C_j|) &= \tau(|C_j|) - 1 \\ \tau'(|C_k|) &= \tau(|C_k|) + 1 \\ \tau'(|B_j|) &= \tau(|B_j|) - ob_i \\ \tau'(|B_k|) &= \tau(|B_k|) + nb_i \end{aligned}$$

where τ' is the computation state resulting from the assignment. From the definition of the objective function, it is now easy to see that only two terms of the summation get affected, i.e., the terms j and k . The variation for δ_j and δ_k are

$$\begin{aligned}\delta_j &= 2(\tau(|B_j|) - ob_i)(\tau(|C_j|) - 1) - (\tau(|C_j|) - 1)^2 - 2\tau(|B_j|)\tau(|C_j|) + \tau(|C_j|)^2 \\ \delta_k &= 2(\tau(|B_k|) - nb_i)(\tau(|C_k|) + 1) - (\tau(|C_k|) + 1)^2 - 2\tau(|B_k|)\tau(|C_k|) + \tau(|C_k|)^2\end{aligned}$$

and Δ simply becomes $\delta_j + \delta_k$. With this manual differentiation approach, the graph coloring statement shown in Figure 8.2 turns almost completely incremental.

The savings of this approach are significant since the cost of propagating all the invariants of the model is replaced by an expression that computes Δ in a time proportional to $O(|C_j| + |C_k|)$. The savings also come indirectly, since LOCALIZER no longer needs to save and restore the old state τ . The extra effort in modeling is more than balanced by the effectiveness of the new formulation. In practice, the new statement is about an order of magnitude faster than its simulated counterpart.

It would be nice to automatically derive this statement from the simulation approach. However, it is hard to do so in general, since an effective derivation may need semantic information (e.g. a matrix is symmetric). It is certainly a promising research topic.

8.1.2 Exploiting Invariants for More Incrementality

In a differentiation approach, it is useful to consider how to compute the difference Δ in the current state. It can be computed either by an expression in the neighborhood definition or as part of the invariants definition. This may seem insignificant but, in practice, it can make a substantial difference for at least two reasons. First, in typical local search algorithms, neighbors are close and the updating of invariants is, in general, proportional to how close they are. Second, even in strategies that do not require to scan the whole neighborhood (e.g., simulated annealing), a significant number of neighbors may still need to be explored when the quality of the solution improves. As a consequence, it is often useful to exploit invariants to compute this difference. We illustrate how this can be performed on the graph coloring application.

In graph coloring, the impact on the objective function was expressed in terms of the two numbers nb_i and ob_i defined in the previous section. There are only two alternatives to compute these numbers directly. The first possibility is to walk down the adjacency list of vertex i and count the number of vertices that lie in class j or class k . The second option is to iterate on classes j and k themselves and check whether an edge $\langle i, b \rangle$ belongs to the edge set of the graph for any vertex b in these classes. The worst case performance of the first approach is $O(n)$ where n is the number of vertices in the graph, while the second approach costs $O(|C_j| + |C_k|)$ which can be significantly smaller. This approach was used

```

Optimize
Data Type: edge = record s : int; t : int; end;
Constant:
  n      : int = ...;
  E      : {edge} = ...;
  cutOff : real = ...;
  chPerc : real = ...;
  maxFreeze: real = ...;
  A      : array[i in 1..n, j in 1..n] of boolean := ⟨i, j⟩ in E;
Variable:
  x : array[1..n] of int;
  t : int;
  fc: int;
  ch: int;
Invariant:
  C : array[1..n] of {int} := distribute(x, {1..n}, {1..n});
  Empty : {int} = { i : int | select i from 1..n where size(C[i]) = 0 };
  NEmpty : {int} = { i : int | select i from 1..n where size(C[i]) > 0 };
  unused : int = minof(Empty);
  Candidates : {int} = NEmpty union unused;
  B : array[k in 1..n] of {int} = { ⟨i, j⟩ : edge | select i from C[k]
                                     select j from C[k] where A[i, j] };
  Obj      : int = sum(i in 1..n) (2×size(C[i])×size(B[i]) - size(C[i])2);
  countB   : int = sum(i in 1..n) size(B[i]);
Operator:
  int f(C : int, S : int) { return 2 * C * S - C2; }
  int diff(i : int, dc : int, ds : int) {
    return f(size(C[i])+dc, size(B[i])+ds) - f(size(C[i]), size(B[i])); }
  void lowerTemp() { t := t * 0.95; ch := 0; fc := fc + (ch/trials < mp); }
Satisfiable: cb = 0;
Objective Function: minimize Obj;
Neighborhood: move x[i] := c
               where i from {1..n};
               c from Candidates;
               nb = sum(j in C[c]) A[i, j];
               ob = sum(j in C[x[i]]) A[i, j];
               d = if x[i] <> k then diff(x[i], -1, -ob) + diff(c, 1, nb) else 0;
               accept
               d < 0 → { if countB = 0 then fc = 0 endif; ch:=ch+1; }
               cor d=0
               cor Pr(e-d/t): always → ch:=ch+1;
Start: genState();
Restart: lowerTemp();
Parameter: maxTrials := round(sf * chest * n);
Local Condition: ch < NC;
Global Condition: fc < 5;

```

Figure 8.2: A Statement for Graph Coloring Based on Differentiation.

in the statement shown in the previous section. We now discuss how to implement the former.

The key observation underlying the first approach is that the move of vertex i from class j to class k only affects the class connectivity of few other vertices. The values nb_v and ob_v for any vertex v not directly connected to vertex i are left unaffected by the assignment of x_i . Hence it is possible and desirable to maintain incrementally the set of edges connecting a vertex i to any class k . If

$$CC_{ik} = |\{(a, b) | a = i \wedge b \in C_k\}|$$

denotes the cardinality of the set of edges connecting vertex i to class k , then $nb_i = CC_{ik}$ and $ob_i = CC_{ij}$. The matrix CC is easily defined as an array of arrays with the `dcount` global invariant

$$CC : \text{array}[i \text{ in } 1..n] := \text{dcount}(x, AL[i], \{1..n\});$$

where $AL[i]$ is a constant denoting the set of vertices adjacent to vertex i .

The potential gain in efficiency in this case comes from the fact that CC_{ik} can be updated in time proportional to the size of $AL[i]$ (in the worst case) and that the evaluation of nb_i and ob_i reduces to a lookup in the CC matrix. The assessment of the quality of a neighbor becomes a $O(1)$ time operation. Even though this seems more expensive than the direct model of the previous section, the technique brings about 20% in savings. To understand why, it is useful to remember that the search strategy is simulated annealing. A move is accepted unconditionally when it leads to an improvement. It is also accepted if it degrades the current solution, with a probability that decreases as time passes. As a result, simulated annealing tends to accept many moves early on and rejects most of the moves later on. The slight loss in performance for updating CC is thus amortized over the number of moves considered, but not accepted, since a rejected move now only costs $\Theta(1)$ to be evaluated. The final statement is shown in Figure 8.3.

8.1.3 Summary

It is useful to summarize the issues related to incrementality. In general it is recommended to use a simulation approach to study the neighborhood to understand the trade-offs between their sizes and the quality of solution produced. When a given neighborhood and strategy is chosen, it is useful to study whether it can use a differentiation approach and whether the invariants can be used to increase efficiency further. In general, we found out that using

```

Optimize
Data Type: edge = record s : int; t : int; end;
Constant:
  n      : int = ...;
  E      : { edge } = ...;
  cutOff : real = ...;
  chPerc : real = ...;
  maxFreeze: real = ...;
  A      : array[i in 1..n, j in 1..n] of boolean := ⟨i, j⟩ in E;
  AL: array[i in 1..n] of {int} := {j : int | select j
                                     from 1..n where A[i, j] or A[j, i]};

Variable:
  x : array[1..n] of int;
  t : int;
  fc: int;
  ch: int;

invariant:
  C : array[1..n] of {int} := distribute(x, {1..n}, {1..n});
  CC : array[i in 1..n] := dcount(x, AL[i], {1..n});
  Empty : {int} = { i : int | select i from 1..n where size(C[i]) = 0 };
  NEmpty : {int} = { i : int | select i from 1..n where size(C[i]) > 0 };
  unused : int = minof(Empty);
  Candidates : {int} = NEmpty union unused;
  B : array[k in 1..n] of {int} := { ⟨i, j⟩ : edge | select i from C[k]
                                     select j from C[k] where A[i, j] };
  Obj : int = sum(i in 1..n) (2 × size(C[i]) × size(B[i]) - size(C[i])2);
  countB : int = sum(i in 1..n) size(B[i]);

Operator:
  int f(C : int, S : int) { return 2 * C * S - C2; }
  int diff(i : int, dc : int, ds : int) {
    return f(size(C[i]) + dc, size(B[i]) + ds) - f(size(C[i]), size(B[i])); }
  void lowerTemp() { t := t * 0.95; ch := 0; fc := fc + (ch / trials < mp); }

Satisfiable: cb = 0;
Objective Function: minimize Obj;
Neighborhood: move x[i] := c
               where
                 i from {1..n};
                 c from Candidates;
                 nb = CC[i, c];
                 ob = CC[i, x[i]];
                 d = if x[i] <> k then diff(x[i], -1, -ob) + diff(c, 1, nb) else 0;
               accept
                 d < 0 → { if countB = 0 then fc = 0 endif; ch := ch + 1; }
                 cor d = 0
                 cor Pr(e-d/t): always → ch := ch + 1;

```

Figure 8.3: Maintaining Connectivity Information Incrementally.

invariants as much as possible is a good strategy but more experimental work needs to be done before reaching a conclusion. Note also that sometimes it may not be possible to move from a simulation to a differentiation approach. In these cases, it may be helpful to consider approximations of the evaluation function as shown in the scheduling application.

8.2 Complexity Analysis of Invariants

To write more effective statements in LOCALIZER, it is important to understand their cost structure in terms of time and space. The computation of this cost depends on both the complexity of propagation for elementary invariants and on the number and the nature of the elementary invariants. This section reviews both aspects.

8.2.1 Propagation Cost

The propagation cost is associated with a pair $\langle x, y := I \rangle$, where x is the variable triggering the propagation and $y := I$ is the invariant to be updated. Remember that an invariant $y := I$ is reconsidered at most once for any particular x but might have to be reconsidered several times due to different variables. We now quickly review the propagation cost of various invariants described in Chapter 6.

Arithmetic Invariants Invariants of the form $x := y \oplus z$ are updated in constant time.

Aggregate Invariants Summations and products are all updated in constant time, including the dynamic versions DSUM and DPROD. The MAX, MIN, ARGMAX, and ARGMIN invariants, together with their dynamic equivalents, are updated in $O(\log n)$ in the worst case, where n is the size of the set they are defined upon. This bound can be obtained easily by using a heap to maintain the set of pair $\langle i, e(i) \rangle$ for all elements i in the set.

Set Invariants The FILTER, SINGLETON, and CSINGLETON invariants are all updated in constant time. The CROSS invariant is a bit more expensive and is updated in $O(|\Delta||y|)$, where Δ is the list of changes for the operand triggering the propagation and y is the other operand. Finally, $x := \text{DFILTER}(y, F)$ has an update cost proportional to the size of the difference list Δ associated with y and the size of the difference list for the boolean function F , giving $O(|\Delta| + |\text{DIFF}(F)|)$ overall worst case performance.

8.2.2 Global Cost

Invariants depend on each other, since a variable x can occur in the definition of another invariant $y := I$. Global Cost analyzes the complexity of a set of invariants. It is necessary to consider the set of elementary invariants that result from the normalization phase. These elementary invariants expose the memory and computing costs more clearly.

For instance, aggregates are rewritten as $y := \Pi(x_1, \dots, x_n)$ showing the space required by the invariants is $\Theta(n)$ and that the number of propagation is $O(n)$ in the worst case. Note that the expected number of propagations is often much smaller than $O(n)$ due to locality. Similarly, a static set

$$x = \{v \mid \text{select } v \text{ from } \{e_1, \dots, e_n\} \text{ where } E\}$$

is replaced by the set of elementary invariants

$$\left\{ \begin{array}{l} x := \text{UNION}(s_1, \dots, s_n) \\ b_1 := E[v/e_1], s_1 := \text{CSINGLETON}(b_1, e_1) \\ \vdots \\ b_n := E[v/e_n], s_n := \text{CSINGLETON}(b_n, e_n) \end{array} \right\}$$

Once again, the normalization makes the memory and time requirements more explicit. The UNION invariant may need to be propagated up to n times and uses $\Theta(n)$ space. Moreover, there are $2n$ auxiliary invariants introduced to define the intermediate variables b_i and s_i . The number of propagations for UNION is $O(n)$ and $\Theta(1)$ for CSINGLETON in the worst case. The number of propagation for an arithmetic invariant such as

$$b_i := E[v/e_i]$$

is also related to the normalization. Elementary invariants are binary relations of the form

$$x := y \oplus z$$

hence, the normalization transforms an arbitrary expression

$$x := a + b + c * d + e$$

in a set of binary invariants

$$\left\{ \begin{array}{ll} \sigma_1 & := a + b \\ \sigma_2 & := c * d \\ \sigma_3 & := \sigma_1 + \sigma_2 \\ x & := \sigma_3 + e \end{array} \right\}$$

The cardinality of the set obtained is equal to n , the number of variable occurrences in the normalized expression. The space requirement follows as $\Theta(n)$ and the number of propagation is $O(n)$ in the worst case.

Let us now illustrate how the propagation and the global cost can help us to choose between several statements on graph coloring problems. More precisely, let us examine two ways of defining the bad edges and study the space requirements and the time necessary to update these invariants when a variable x is given a new color.

The first approach defines each B_i as a static set subject to the rewriting rules mentioned above.

$$B : \text{array}[k \text{ in } 1..n] \text{ of } Edge := \{ \langle i, j \rangle : Edge \mid \text{select } \langle i, j \rangle \text{ from } E \\ \text{where } x[i] = k \wedge x[j] = k \};$$

It implies that each B_k creates $\Theta(n^2)$ (exactly $4n^2$) elementary invariants since the edge set E is constant and of cardinality $n^2 = |E|$ where n is the number of vertices in the graph. For a given B_k and each tuple $\langle i, j \rangle$ in E , the normalization produces the set of elementary invariants

$$SE_{k, \langle i, j \rangle} = \left\{ \begin{array}{ll} \sigma_{k1} & := x[i] = k \\ \sigma_{k2} & := x[j] = k \\ \sigma_{k3} & := \sigma_{k1} \wedge \sigma_{k2} \\ \sigma_{k4} & := \text{CSINGLETON}(\sigma_{k3}, \langle i, j \rangle) \end{array} \right\}.$$

Let SE_k be the set of elementary invariants produced by a given B_k , i.e.,

$$SE_k = \bigcup_{\langle i, j \rangle \in E} SE_{k, \langle i, j \rangle}.$$

The analysis of SE_k reveals that, out of the $4n^2$ elementary invariants, only $2n$ of them contain a reference to variable $x[i]$. In turn, this implies that assigning $x[i]$ to a new value triggers the propagation of $\Theta(n)$ invariants in each set SE_k . Since there are exactly n entries in the array, B has a space complexity of $\Theta(n^3)$ and a time complexity of $\Theta(n^2)$.

Consider now the invariant

$$B : \text{array}[k \text{ in } 1..n] \text{ of } Edge := \{ \langle i, j \rangle : Edge \mid \text{select } i \text{ from } C[k] \\ \text{select } j \text{ from } C[k] \\ \text{where } A[i, j] \};$$

where C is an array of sets denoting the color classes of the graph. Assume that all $C[k]$ are updated in constant time when a given variable $x[i]$ is updated thanks to the global invariant distribute. The next section shows how to obtain this result.

This second formulation is also subject to rewrite rules. Clearly, the definition of an invariant B_k falls in the semi-dynamic set category because C is an array of variable and A is a constant matrix. The space complexity is $\Theta(n)$ since the result of the rewrite phase for a single B_k is spelled out as

$$SE_k = \left\{ \begin{array}{ll} \sigma_{k1} & := \{()\} \\ \sigma_{k2} & := \text{CROSS}(\sigma_{k1}, C[k]) \\ \sigma_{k3} & := \text{CROSS}(\sigma_{k2}, C[k]) \\ B_k & := \text{FILTER}(\sigma_{k3}, F) \end{array} \right\}$$

Consider now the time complexity. The definition of SE_k clearly indicates that only 2 elementary invariants link $C[k]$ and $B[k]$. This implies that, whenever a set $C[k]$ changes, a constant number of invariants are propagated to update the associated $B[k]$. Moreover, changing the color of a vertex i moves that vertex from class C_1 to class C_2 . This change implies that the classes C_1 and C_2 are both modified. The conclusion is that only a constant number of invariants (6 to be precise) are propagated to update the array B when $x[i]$ is assigned a new value. As a consequence, the overall time complexity is $\Theta(n)$.

Consequently, the latter formulation produces two orders of improvement in space and one order of magnitude in time complexity over the former and is clearly preferable.

8.2.3 Global Invariants

Note that the availability of `distribute` is critical to achieve the time complexity. A direct definition

$$C : \text{array}[k \text{ in } 1..n] \text{ of } \text{int} := \{i : \text{int} \mid \text{select } i \text{ from } 1..n \text{ where } x[i] = k\};$$

based on the semantics of `distribute`

$$\text{distribute}(x, I, O) \Leftrightarrow B_j = \{i \in I \mid x[i] = j\} \forall j \in O$$

gives much poorer bounds. In this definition, each C_j would be normalized into the set of invariants SO

$$SO = \bigcup_{j \in O} SO_{ji}.$$

where

$$SO_{ji} = \left\{ \begin{array}{ll} \sigma_{k1} & := x[i] = j \\ \sigma_{k2} & := \text{CSINGLETON}(\sigma_{k1}, i) \end{array} \right\}.$$

The analysis of the set SO reveals that each $x[i]$ appears in a total of $\Theta(n)$ invariants which would lead to a $\Theta(n^2)$ space complexity and a $\Theta(n)$ time complexity. The constant time implementation for distribute is based on the following observation. With the definition

$$\text{distribute}(x, I, O) \Leftrightarrow B_j = \{i \in I \mid x[i] = j\} \forall j \in O$$

when a variable $x[i]$ changes value from $x[i]^o$ to $x[i]^c$, it is sufficient to remove the value i from the set $B_{x[i]^o}$ and insert the value i in the set $B_{x[i]^c}$. Both operations can be carried out in constant time. Note that distribute introduces a topological constraint

$$tc(B := \text{distribute}(x, I, O)) = \{t(B_k) = \max_{i \in I} (t(x_i)) + 1 \mid k \in O\}$$

that relates B and all the variables in the array x , resulting in a space requirement of $O(|I|)$.

8.3 Constant Factors

The design of a model at a high level of abstraction often encourages conceptual structures that prove too powerful for the task at hand. Consider, for instance, the invariants

```

P : {int} := {i : int | select i from 1..n where x[i]};
N : {int} := {i : int | select i from 1..n where not x[i]};
I : int := size(P) - size(N);
C : {Edge} := {⟨a, b⟩ : Edge | select ⟨a, b⟩ from E where x[a] <> x[b]};
O : int := size(C) + 0.2 * I * I;

```

from a graph partitioning statement. P and N denote the set of vertices in each partition, I denotes the imbalance between the partitions and C is the cut set, i.e. the set of edges with one endpoint in each partition. Finally O is the objective function.

The first point to notice is the use of sets for the definition of P and N . The sets are maintained for one purpose only: obtain their sizes. In this case, it is far better to adopt a counting strategy instead. It does not change the asymptotic time and space complexities, but it does greatly reduce the constant factor and avoid the bookkeeping for the sets. The definitions

```

P : int := sum(i from 1..n) x[i];
N : int := sum(i from 1..n) not x[i];

```

are thus preferable. The same holds true for C and a better choice is

```

C : int := sum(i from E) (x[i.a] <> x[i.b]);

```

Finally, it is wise to exploit the symmetry of the problem to eliminate N altogether and define I as

$$I : \text{int} := 2 * P - n;$$

since $P + N = n$.

8.4 Database Techniques

All the techniques used in relational algebra for optimizing queries also apply in the context of sets. For instance, it is desirable to push the filtering expressions as early as possible in the set definition since they reduce the sizes of the intermediate sets. For instance, the definition

Constant:

$AP : \{\text{Arc}\} := \{ a : \text{Arc} \mid \text{select } a.s \text{ from } 1..n \text{ select } a.t \text{ from } 1..a \};$

Invariant:

$ok : \text{array}[i \text{ in } 1..n] \text{ of } \{\text{Arc}\} := \{ \langle a, b \rangle : \text{Arc} \mid \text{select } \langle a, b \rangle \text{ from } AP \\ \text{where } \text{tabu}[a] < \text{trials} \};$

with a space complexity of $O(n^2)$ is worse than

Invariant:

$ok : \text{array}[i \text{ in } 1..n] \text{ of } \{\text{Arc}\} := \{ \langle a, b \rangle : \text{Arc} \mid \text{select } a \text{ from } 1..n \\ \text{where } \text{tabu}[a] < \text{trials} \\ \text{select } b \text{ from } 1..a \};$

since the latter uses only $O(Mn)$ in space where M is the number of non-tabu elements in a .

Chapter 9

Related Work

This chapter reviews related research in constraint programming, modeling languages, programming languages and incremental algorithms. It also discusses directions for future work when appropriate.

9.1 Constraint Programming Languages

Many constraints programming languages have been defined in the last 15 years to support the solving of combinatorial optimization problems. Well-known representatives include CHIP, Ilog Solver [49], CLP(\mathcal{R}) [27], PROLOG III [12] and Oz [26] and they are based on various programming paradigms (e.g. logic programming, functional programming, object-oriented programming). Almost all these languages support the global search paradigm (e.g. branch and bound or constraint satisfaction). More precisely, a program in these languages can be seen as a high-level specification of a global search algorithm. These specifications generally consist of two parts: a declarative component that specifies the set of constraints to be satisfied and an algorithmic component that searches for solutions.¹ These constraint programming languages generally use “logical variables”, i.e., variables that can be assigned once, and the purpose of a constraint program is generally to find values for the variables. Finally, at the core of these languages lie constraint-solving algorithms that may be rather sophisticated and include linear and non-linear programming algorithms as well as constraint satisfaction algorithms.

LOCALIZER differs fundamentally from these languages in that it supports the local search paradigm. LOCALIZER is the first language to support the local search paradigm.

¹Note that the algorithmic component may itself be declarative as in constraint logic programming.

There have been attempts (e.g., [71]) to embed local search algorithms into constraint languages but LOCALIZER is the first language that supports users in defining their own local search algorithms. A statement in LOCALIZER is a high-level specification of a local search algorithm and LOCALIZER also has declarative and algorithmic components. The declarative component specifies the invariants that must be maintained, while the algorithmic component modifies the value of some of the variables to move from neighbors to neighbors. Note that variables in LOCALIZER are closer to variables in imperative languages and that the core of a LOCALIZER statement is a collection of incremental algorithms to maintain invariants. These differences with traditional constraint programming languages are in no way artificial but reflect the fundamentally different way of organizing global and local search algorithms.

Note finally that CLAIRE [8] is a language that provides tools for implementing global search algorithms (e.g. deduction rules and non-determinism). However, since CLAIRE is based on traditional variables, it can also be used to implement local search algorithms by enhancing its functionality with invariants.

9.2 Modeling Languages

Modeling languages such as AMPL [19], GAMS [5], LINDO [65] and NUMERICA [80] have been proposed to simplify the design of mathematical programming problems. These modeling languages focus on stating the problem constraints and support high-level algebraic and set notations to express these constraints from the data. They are mostly independent from the underlying solving algorithms (e.g., linear programming solvers), although they may give access to some of the solver's parameters and options. These modeling languages do not support the search component of traditional constraint programming languages but they have rich data modeling features that make them accessible to a wide audience. An exception is the optimization programming language OPL [79], a recent addition in the field of modeling languages. OPL in fact allows both the specification of constraints as in traditional modeling languages and the definition of search procedures.

LOCALIZER shares with these languages the idea of supporting high-level algebraic and set notations to simplify problem statements. However, there is a fundamental difference between LOCALIZER and these languages: a LOCALIZER statement describes a local search algorithm, not a set of constraints or a global search algorithm.

It is also useful to point out that the scripting languages provided by some modeling languages such as AMPL aims at specifying algorithms in terms of models. Their contributions

take place at another level and are orthogonal to LOCALIZER.

9.3 Graphical Constraint Systems

Historically, constraints were first used in graphical systems such as Sketchpad [72] and Thinglab [6]. Many of the available constraint-based graphical packages containing constraints in fact support one-way constraints [39], [40] that can be viewed as invariants. LOCALIZER was in fact inspired by these systems and originated in our desire to determine if this approach could provide adequate support for local search algorithms. LOCALIZER, of course, goes far beyond typical one-way constraints found in graphical systems, both in terms of the expressiveness of its invariants and in terms of its implementation technology. LOCALIZER invariants may contain arrays (indexed by variables), sets, graphs and other higher-level data structures. The basic planning/execution model also had to be generalized to interleave the planning and the execution phases to support dynamic invariants.

Much recent work in constraint-based graphics has been concerned with multi-way constraints [58], [59], [62], [61], [81], [41] and constraint hierarchies [7], [82], [60]. In graphical systems, one is often interested in specifying relationships between graphical objects and one-way constraints are not always the best vehicle to support these relations. Multiway constraints specify how to maintain a relationship and constraint hierarchies are used to decide how to choose between different solutions. These functionalities are not needed in LOCALIZER, since invariants are used to build data structures incrementally, not to maintain relationships incrementally. Interested readers should consult [78] for a survey of constraint programming.

9.4 Finite Differencing

Finite differencing [46] is a technique used in optimizing compilers to improve efficiency. To review finite differencing, it is useful to start with a simple example. Consider the program

```

for (i:=0; i < n; i++) {
    f := f + i * c;
}

```

The time-consuming part in this program is the multiplication $i * c$. The basic idea behind finite differencing is to abstract this expensive component into a new variable, say i' , and to update the variable whenever the subcomponents i and c are modified. An optimizing

component can then generate the appropriate code for each of these modifications. For the above example, the compiler may be able to generate the program

```

i' := 0;
for (i:=0; i < n; i++) {
    :
    f := f + i';
    i' := i' + c;
}

```

where multiplications are replaced by additions.

More generally, the fundamental idea behind finite differencing is to abstract expensive computations by abstract data types that maintain a state and whose operations (called derivatives) specify how to update this state when one of the subcomponents of the abstraction is modified. This idea applies of course to numerical expressions and Paige [44] showed how it could be generalized to complex data structures. It is also useful to realize that techniques like Rete networks [18] fall in this category and can be helpful to implement incremental set operations.

The implementation of invariants can be viewed as a generalization of the finite differencing approach. The finite differencing approaches that we are aware of correspond to our static invariants. This comes from the fact that these techniques were used in the context of optimizing compilers and thus it was necessary to perform the transformations at compile time. LOCALIZER generalizes this approach to dynamic invariants that are critical for many local search algorithms. As mentioned, dynamic invariants require to interleave the planning (or compilation in this context) and execution phases.

9.5 Programming with Invariants

Finite differencing was also used by Paige [45] to propose a new syntactic construct called invariant.² The motivation behind the paper was the recognition that a number of efficient, but involved, programs are in fact incremental versions of much simpler algorithms. For instance, heap sort can be viewed as a version of selection sort where the minimum element is maintained incrementally. The paper then suggests using invariants to maintain these data structures incrementally. Here, an invariant is a syntactic construct that defines a state and whose operations are derivatives that specify how to update the state when some other variables are modified. In other words, Paige's invariants are abstractions to construct

²We became aware of the work of Paige after completion of the first implementation but it is interesting to observe that the same name was used for two different, but related, concepts.

what we called invariants in LOCALIZER. A pre-processor then rewrites these invariants into traditional code through successive transformation.

It is interesting to observe that LOCALIZER shares the same motivations and address the same issues with related concepts. There are however some important differences. First, at the conceptual level, LOCALIZER supports a rich set of invariants without requiring user intervention. Our basic motivation here is that the definition of these invariants is difficult and should be automated as much as possible. Second, at the implementation level, Paige's invariants are static (they are rewritten into conventional code at compile time) and dynamic invariants are not supported. As mentioned, dynamic invariants are critical for achieving good performance on a variety of local search algorithms.

The idea of letting users define their own invariants is however very appealing, since it may make the language more extensible. It is particularly attractive if the techniques of LOCALIZER are integrated in an object-oriented library. It is an interesting open issue to determine how this can be achieved for dynamic invariants.

9.6 INC: An Incremental Programming Language

Another related research, once again from the programming language field, is INC [83], a language for incremental computations. INC is a language that allows writing non-recursive functions in the style of functional programming. These functions are transformed by the INC compiler into an incremental algorithm which, given a variation of the input, produces a new output. As mentioned, INC uses functional notation and its primitives (e.g., FILTER, TUPLIFY, EQUIJOIN) are closely related to our normalized invariants. This is of course not surprising since bags and tuples are used as the main data structures. The implementation of these operations is based on finite differencing [44, 46].

Once again, there are fundamental differences between INC and LOCALIZER. At the conceptual level, there are three main differences. First, invariants in LOCALIZER are at a much higher level of abstraction than INC functions. Invariants are generally close the data used in informal descriptions of local search algorithms. INC functions, on the other hand, are roughly at the level of our elementary invariants. Since our goal is to provide high-level abstractions for local search, the transformation from invariants to normalized invariants is best left to the compiler, especially since this translation can be performed effectively. Second, and equally important, INC has no counterpart for our elementary invariants DFILTER and element. This is an important limitation since these invariants are fundamental in

practical applications. Third, INC has no support to define recursively defined data structures as in the job-shop and vehicle routing applications. It is of course possible to extend INC to remedy these limitations. High-order functions could be introduced to support dynamic invariants and a fixpoint operator can be added to support recursively defined data structures. However, it is not clear that the INC compiler could recover enough of the structure of the application to obtain the same efficiency as LOCALIZER. Also, implementing these extensions in full generality may be challenging and not needed in practice. At the implementation level, of course, INC supports only static invariants. As far as we know, no experimental results were reported on significant applications.

In summary, invariants seem to provide a much better expressiveness/efficiency trade-off³ than INC functions to support local search algorithms and extensions of INC are likely to be too general and thus probably inefficient or ad-hoc.

9.7 Incremental Algorithms

Finally, it is useful to link our research to the large body of research on incremental algorithms. It is not our goal to review this research in full detail but rather to explain what we mean by efficient incremental algorithms in this thesis.

The traditional online analysis of an algorithm uses, as a complexity parameter, the size of the input. The worst case analysis of an algorithm, either batch or incremental, gives an upper bound to the running time $O(f(|input|))$. The analysis of the problem complexity itself leads to a lower bound $\Omega(g(|input|))$. Combining the lower and the upper bound is the best way to determine whether it is actually possible to improve upon the given algorithm or if it is optimal. This approach can be inadequate. Indeed, the analysis can report that a batch algorithm is asymptotically optimal even for an incremental problem. Several techniques have been proposed to overcome this inadequacy. Among them are micro-analysis [9], average case or expected case analysis [16],[50], amortized analysis [14], [75] and analysis based on input and output sizes as a complexity parameter [2], [53],[51], [52].

The analysis of algorithms in this thesis is in the spirit of the work of Ramalingam and Reps [53] and is expressed in terms of the size of the variations in input and output. Our basic motivation in analyzing algorithms in this way comes from the fact that this analysis is much more informative in this context than a traditional worst-case online analysis. Indeed, the objective of incremental algorithms in LOCALIZER is to compute a new output for a,

³Note that INC has as transitive closure operator that is an extension that is being implemented in LOCALIZER

generally small, variation in the input. However, sometimes a small variation in the input may induce a dramatic variation in the output so that an analysis taking into account only the input will not be informative, even in an amortized sense. Note that, with this in mind, an incremental algorithm that turns out to be linear in the input/output variations is optimal since all the new bits in the input must be seen and at least all the new bits in the output must be written. The elementary invariants `SUM`, `SUMD`, `PROD`, `PRODD`, `FILTER`, `CROSS` and `FILTERD` all have optimal implementations in this sense in the current implementation of `LOCALIZER`. Similarly, high-level invariants such as `distribute` and `dcount` are optimal.

It is also interesting to discuss more global analysis as we did in Chapter 8. A global analysis of the model for the simple job-shop scheduling statement shows that `LOCALIZER` is running in time $O(\delta)$ where δ is the size of the change in input and output. This is the same bound as the incremental algorithm for single source shortest path proposed in [53]. The same result would not hold when the vertices have an unbounded degree. This is one of the primary motivations to consider global invariants. Global invariants not only increase the expressiveness of the language but also make it possible to obtain better incremental behaviour. We expect that research on global invariants will be as important as research on global constraints in constraint programming.

Chapter 10

Conclusion

This thesis was motivated by our desire to support local search in a modeling language in order to reduce development time significantly while preserving most of the efficiency of specialized algorithms. Local search is one of the main optimization techniques for combinatorial optimization and designing local search algorithms remains an art that requires considerable expertise and implementation skills.

This thesis is a proof of concept. Its main contribution is to present LOCALIZER, a modeling language for local search. LOCALIZER enables concise descriptions of local search algorithms that, when executed by the LOCALIZER interpreter, compare well with specialized programs. As far as we know, LOCALIZER is the first modeling or constraint language to support local search. LOCALIZER obviously needs to be applied to even more complex problems but we believe that this thesis provides enough evidence that this approach is viable.

It is however important to step back and analyze LOCALIZER's technical contributions in more detail to determine what has been learned and what is likely to be present in future such languages. The main conceptual contribution of LOCALIZER is the concept of invariants. Invariants provide a declarative specification of incremental data structures, automating one of the most tedious and error-prone aspects of local search algorithms. We believe that invariants will play a fundamental role in any language supporting local search. As this thesis shows, they allow for very compact statements, while making sure that they can be implemented by efficient incremental algorithms. Further evidence of their usefulness comes from the fact that they generalize the finite differencing approach. The LOCALIZER invariants are likely to evolve and to be generalized and they provide a suitable platform from which to start building. The main technical contribution of LOCALIZER is the concept

of dynamic invariants that generalizes the finite differencing techniques. As discussed several times in this thesis, dynamic invariants are critical to obtain good performance on many combinatorial optimization problems. Once again, dynamic invariants are likely to evolve and to be generalized but they will also be part, we believe, of any implementation of invariants. Their implementation techniques make it possible to implement sophisticated incremental algorithms.

In the rest of this conclusion, we mention some of the research directions that we find particularly promising for the future.

Invariants Global invariants, i.e., invariants over more complex data structure such as graphs, are likely to be an important research topic in the future. Global invariants are able to exploit additional structure and will lead to better incremental algorithms. We strongly believe that global invariants will be as important for local search as global constraints for global search. To illustrate this idea, recall that a problem like “single source shortest path” has an incremental algorithm with a complexity bound of $O(\delta \log \delta)$ while a direct statement of the recurrence that defines it lead to a weaker result of $O(\delta^2 \log n)$ where n is the size of the input and δ is the size of the variation in input and output. Clearly, many incremental algorithms could become part of LOCALIZER through global invariants such as longest.

It is also interesting to extend LOCALIZER to support a transitive closure operator, as mentioned earlier. Concepts like connected components in graphs or Kempe chains in graph coloring can be expressed naturally as transitive closures. The problem of maintaining transitive closures incrementally has been studied in the past [83].

Finally, the invariant construct of [45] would increase the extensibility of the language and its generalization to dynamic invariants should be studied.

Neighborhood and Parallelism Parallel explorations of neighborhood [63] provide an interesting alternative to differentiation approach. A parallel exploration of a neighborhood based on simulation may approach the efficiency of a sequential exploration based on differentiation while keeping the model simpler. Results with the *Mob* heuristic in graph-partitioning lead to excellent results both in term of time and quality. Similarly, Hamadi et al. in [25] implemented parallel hardware to explore the simple flip neighborhood of GSAT and obtained results that were similar to the fully incremental statement discussed here. We have seen how easy it is to express the corresponding simple GSAT algorithm. Consequently, it would be advantageous to provide both alternatives, i.e. a simple statement with

a parallel exploration of the neighborhood, and a fully incremental statement leading to a sequential exploration. Additionally, it might be interesting to investigate the potential of a parallel propagation algorithm for the invariants themselves.

Other Classes of Local Search Algorithms Genetic algorithms form a different class of local searches based on populations of solutions. Genetic algorithms generate new solutions via *reproductions* and *mutations*. Clearly, each individual solution could also be characterized in terms of state variables and invariant variables. In that respect, the infrastructure provided by LOCALIZER could be reused to support this class of algorithms. Additionally, genetic algorithms strongly encourage the use of parallel techniques to produce the successive *generations* of the population (iterations of the algorithm).

Specification of the Neighborhood In most of the statements presented in this thesis, neighborhoods are presented globally by the properties that they must satisfy. There are algorithms where the neighborhood is described locally, i.e., the algorithm specifies how to move from a neighbor to another without describing the properties of the states explicitly. An example of local specification is the 2-opt local search of the traveling salesman. An initial circuit is constructed and the search moves from one circuit to another without representing explicitly that a circuit is needed.

Specifying neighborhoods locally can easily be done in LOCALIZER using procedural constructs. An interesting research avenue is to determine whether there exist more declarative ways of specifying these local neighborhoods and a restricted form of first-order logic may be appropriate. Constraints, that are discussed in the next paragraph, are another possibility.

A Unified Framework Finally, the integration of global search with local search is a fundamental topic that needs to be addressed since many state-of-the-art algorithms (e.g., in scheduling applications) combines local and global search. Early proposals include [71] and [48] but much remains to be done. Traditional constraint languages may prove useful to specify the neighborhood declaratively and traditional constraint algorithms may help exploring the potential candidates effectively (as in [48]). This is likely to be a fundamental topic in coming years.

Appendix A

LOCALIZER Syntax

$\langle \text{scalarType} \rangle$	$::=$ int $::=$ boolean $::=$ float
$\langle \text{basicType} \rangle$	$::=$ $\langle \text{scalarType} \rangle$ $::=$ void
$\langle \text{setElemType} \rangle$	$::=$ $\langle \text{type Identifier} \rangle$ $::=$ $\langle \text{scalarType} \rangle$
$\langle \text{arrayElemType} \rangle$	$::=$ $\langle \text{type Identifier} \rangle$ $::=$ $\langle \text{scalarType} \rangle$ $::=$ $\langle \text{setType} \rangle$ $::=$ $\langle \text{abstractType} \rangle$
$\langle \text{arrayType} \rangle$	$::=$ array [$\langle \text{dimList} \rangle$] of $\langle \text{arrayElemType} \rangle$
$\langle \text{setType} \rangle$	$::=$ { $\langle \text{setElemType} \rangle$ }
$\langle \text{abstractType} \rangle$	$::=$ Arc $::=$ Path ($\langle \text{identifier} \rangle$) $::=$ Sequence ($\langle \text{identifier} \rangle$) $::=$ Circuit ($\langle \text{identifier} \rangle$) $::=$ Graph ($\langle \text{identifier} \rangle$)
$\langle \text{dimList} \rangle$	$::=$ $\langle \text{dimension} \rangle$ { $\langle \text{dimension} \rangle$ }
$\langle \text{dimension} \rangle$	$::=$ [$\langle \text{identifier} \rangle$ in] $\langle \text{expr} \rangle$ $::=$ [$\langle \text{identifier} \rangle$ in] $\langle \text{expr} \rangle$.. $\langle \text{expr} \rangle$
$\langle \text{declType} \rangle$	$::=$ $\langle \text{type Identifier} \rangle$ $::=$ $\langle \text{scalarType} \rangle$ $::=$ $\langle \text{arrayType} \rangle$ $::=$ $\langle \text{setType} \rangle$ $::=$ $\langle \text{abstractType} \rangle$
$\langle \text{declaration} \rangle$	$::=$ $\langle \text{identifier} \rangle$: $\langle \text{declType} \rangle$

Figure A.1: Grammar Fragment for Typed Declarations.

$\langle \text{typeSection} \rangle$	$::= \text{type: } \{ \langle \text{typeDecl} \rangle \}$
$\langle \text{typeDecl} \rangle$	$::= \langle \text{type Identifier} \rangle = \langle \text{recordType} \rangle ;$
$\langle \text{recFieldType} \rangle$	$::= \langle \text{type Identifier} \rangle$
	$::= \langle \text{scalarType} \rangle$
	$::= \langle \text{setType} \rangle$
$\langle \text{recordType} \rangle$	$::= \text{record } \langle \text{field} \rangle ; \{ \langle \text{field} \rangle ; \} \text{end}$
$\langle \text{field} \rangle$	$::= \langle \text{identifier} \rangle : \langle \text{recFieldType} \rangle$

Figure A.2: Grammar Fragment for Record Type Definition.

$\langle \text{bool Expr} \rangle$	$::= \langle \text{bool Expr} \rangle \text{ and } \langle \text{bool Expr} \rangle$ $::= \langle \text{bool Expr} \rangle \text{ or } \langle \text{bool Expr} \rangle$ $::= \langle \text{expr} \rangle$
$\langle \text{expr} \rangle$	$::= \langle \text{arexpr} \rangle \langle \text{relOperator} \rangle \langle \text{arexpr} \rangle$ $::= \text{if } \langle \text{expr} \rangle \text{ then } \langle \text{expr} \rangle \text{ else } \langle \text{expr} \rangle$
$\langle \text{relOperator} \rangle$	$::= [< > = <= >= <> \text{in}]$
$\langle \text{arexpr} \rangle$	$::= \langle \text{operand} \rangle \langle \text{binOperator} \rangle \langle \text{operand} \rangle$ $::= + \langle \text{arexpr} \rangle$ $::= - \langle \text{arexpr} \rangle$ $::= \langle \text{operand} \rangle$ $::= \text{sum} (\langle \text{identifier} \rangle \text{ in } \langle \text{Set Body} \rangle) \langle \text{arexpr} \rangle$ $::= \text{prod} (\langle \text{identifier} \rangle \text{ in } \langle \text{Set Body} \rangle) \langle \text{arexpr} \rangle$ $::= \text{min} (\langle \text{identifier} \rangle \text{ in } \langle \text{Set Body} \rangle) \langle \text{arexpr} \rangle$ $::= \text{max} (\langle \text{identifier} \rangle \text{ in } \langle \text{Set Body} \rangle) \langle \text{arexpr} \rangle$ $::= \text{argmin} (\langle \text{identifier} \rangle \text{ in } \langle \text{Set Body} \rangle) \langle \text{arexpr} \rangle$ $::= \text{argmax} (\langle \text{identifier} \rangle \text{ in } \langle \text{Set Body} \rangle) \langle \text{arexpr} \rangle$ $::= (\langle \text{bool Expr} \rangle)$
$\langle \text{binOperator} \rangle$	$::= [+ - * / \uparrow \text{union}]$
$\langle \text{operand} \rangle$	$::= \langle \text{integer literal} \rangle$ $::= \langle \text{boolean literal} \rangle$ $::= \langle \text{float literal} \rangle$ $::= \langle \text{identifier} \rangle$ $::= \langle \text{array identifier} \rangle [\langle \text{actuals} \rangle]$ $::= \langle \text{function identifier} \rangle (\langle \text{actuals} \rangle)$ $::= \langle \text{Set Expr} \rangle$
$\langle \text{Set Expr} \rangle$	$::= \{ \langle \text{Set Body} \rangle \}$
$\langle \text{Set Body} \rangle$	$::= \langle \text{expr} \rangle .. \langle \text{expr} \rangle$ $::= \langle \text{bool Expr} \rangle \{ , \langle \text{bool Expr} \rangle \}$ $::= \langle \text{identifier} \rangle : \langle \text{type identifier} \rangle \langle \text{query} \rangle \langle \text{index} \rangle$ $::= \langle \text{tuple} \rangle : \langle \text{type identifier} \rangle \langle \text{query} \rangle \langle \text{index} \rangle$ $::= \langle \text{operand} \rangle$
$\langle \text{index} \rangle$	$::= [\text{index by } \langle \text{arexpr} \rangle]$
$\langle \text{tuple} \rangle$	$::= < \langle \text{identifier} \rangle \{ , \langle \text{identifier} \rangle \} >$
$\langle \text{query} \rangle$	$::= \{ \langle \text{select} \rangle \}$
$\langle \text{select} \rangle$	$::= \text{select } \langle \text{select Expr} \rangle \text{ from } \langle \text{range} \rangle [\text{where } \langle \text{bool Expr} \rangle]$ $::= \text{select } \langle \text{select Expr} \rangle \text{ from } \langle \text{operand} \rangle [\text{where } \langle \text{bool Expr} \rangle]$
$\langle \text{select Expr} \rangle$	$::= \langle \text{identifier} \rangle$ $::= \langle \text{tuple} \rangle$
$\langle \text{actuals} \rangle$	$::= \langle \text{bool Expr} \rangle \{ , \langle \text{bool Expr} \rangle \}$
$\langle \text{range} \rangle$	$::= \langle \text{expr} \rangle .. \langle \text{expr} \rangle$

Figure A.3: Grammar Fragment for the Syntax of Expressions.

$\langle \text{constantSect} \rangle$	$::= \text{constant} : \{ \langle \text{constant} \rangle ; \}$
$\langle \text{constant} \rangle$	$::= \langle \text{identifier} \rangle : \langle \text{declType} \rangle = \langle \text{literal} \rangle$
	$::= \langle \text{identifier} \rangle : \langle \text{declType} \rangle := \langle \text{bool Expr} \rangle$
	$::= \langle \text{identifier} \rangle : \langle \text{declType} \rangle := \langle \text{arexpr} \rangle$
	$::= \langle \text{identifier} \rangle : \langle \text{declType} \rangle := \langle \text{Set Expr} \rangle$
	$::= \langle \text{identifier} \rangle : \langle \text{declType} \rangle \text{ init } \langle \text{statement} \rangle$
	$::= \langle \text{identifier} \rangle : \langle \text{declType} \rangle = \dots$
$\langle \text{literal} \rangle$	$::= \langle \text{int literal} \rangle$
	$::= \langle \text{float literal} \rangle$
	$::= \langle \text{boolean literal} \rangle$
	$::= [\text{literal } \{ , \langle \text{literal} \rangle \}]$
	$::= \{ \text{literal } \{ , \langle \text{literal} \rangle \} \}$
	$::= (\text{literal } \{ , \langle \text{literal} \rangle \})$
	$::= \langle \text{literal } \{ , \langle \text{literal} \rangle \} \rangle$

Figure A.4: Grammar Fragment for the Constant Section.

$\langle \text{variableSect} \rangle$	$::= \text{variable} : \{ \langle \text{varDecl} \rangle ; \}$
$\langle \text{varDecl} \rangle$	$::= \langle \text{identifier} \rangle : \langle \text{scalarType} \rangle$
	$::= \langle \text{identifier} \rangle : \langle \text{arrayType} \rangle$
	$::= \langle \text{identifier} \rangle : \langle \text{abstractType} \rangle$
	$::= \langle \text{identifier} \rangle : \langle \text{type Identifier} \rangle$

Figure A.5: Grammar Fragment for the Variable Section.

$\langle \text{invSect} \rangle$	$::= \text{invariant} : \{ \langle \text{invariant} \rangle ; \}$
$\langle \text{invariant} \rangle$	$::= \langle \text{identifier} \rangle : \langle \text{declType} \rangle := \langle \text{bool Expr} \rangle$
	$::= \langle \text{identifier} \rangle : \langle \text{declType} \rangle := \langle \text{arexpr} \rangle$
	$::= \langle \text{identifier} \rangle : \langle \text{declType} \rangle := \langle \text{Set Expr} \rangle$
	$::= \langle \text{identifier} \rangle : \text{distribute}(\langle \text{arexpr} \rangle, \langle \text{Set Expr} \rangle, \langle \text{SetExpr} \rangle)$
	$::= \langle \text{identifier} \rangle : \text{dcount}(\langle \text{arexpr} \rangle, \langle \text{Set Expr} \rangle, \langle \text{SetExpr} \rangle)$

Figure A.6: Grammar Fragment for the Invariant Section.

$\langle \text{statement} \rangle$	$::= \langle \text{statement} \rangle ; \langle \text{statement} \rangle$ $::= \langle \text{identifier} \rangle : \langle \text{type} \rangle$ $::= \langle \text{identifier} \rangle : \langle \text{type} \rangle := \langle \text{expr} \rangle$ $::= \langle \text{lvalue} \rangle := \langle \text{expr} \rangle$ $::= \text{if } \langle \text{expr} \rangle \text{ then } \langle \text{statement} \rangle [\text{else } \langle \text{statement} \rangle] \text{ endif}$ $::= \text{forall } (\langle \text{identifier} \rangle \text{ in } \langle \text{Set Body} \rangle) \langle \text{statement} \rangle$ $::= \text{while } \langle \text{expr} \rangle \text{ do } \langle \text{statement} \rangle$ $::= \text{return } \langle \text{expr} \rangle$ $::= \text{choose } \langle \text{identifier} \rangle \text{ from } \langle \text{expr} \rangle [\langle \text{optClause} \rangle]$
$\langle \text{lvalue} \rangle$	$::= ++ \langle \text{reference} \rangle$ $::= -- \langle \text{reference} \rangle$ $::= \langle \text{reference} \rangle [++]$ $::= \langle \text{reference} \rangle [--]$
$\langle \text{reference} \rangle$	$::= \langle \text{reference} \rangle [\langle \text{actuals} \rangle]$ $::= \langle \text{reference} \rangle (\langle \text{actuals} \rangle)$ $::= \langle \text{reference} \rangle . \langle \text{identifier} \rangle$ $::= \langle \text{array identifier} \rangle$ $::= \langle \text{function identifier} \rangle$ $::= \langle \text{identifier} \rangle$
$\langle \text{optClause} \rangle$	$::= \text{minimizing } \langle \text{expr} \rangle$ $::= \text{maximizing } \langle \text{expr} \rangle$ $::= \text{smaller than } \langle \text{expr} \rangle$ $::= \text{larger than } \langle \text{expr} \rangle$ $::= \text{such that } \langle \text{expr} \rangle [\text{minimizing } \langle \text{expr} \rangle [\text{with } \langle \text{letBlock} \rangle]]$ $::= \text{such that } \langle \text{expr} \rangle [\text{maximizing } \langle \text{expr} \rangle [\text{with } \langle \text{letBlock} \rangle]]$
$\langle \text{letBlock} \rangle$	$::= \{ \langle \text{identifier}_1 \rangle = \langle \text{expr}_1 \rangle ; \dots ; \langle \text{identifier}_n \rangle = \langle \text{expr}_n \rangle \}$

Figure A.7: Grammar Fragment for the Syntax of Statements.

$\langle \text{function} \rangle$	$::= \langle \text{retType} \rangle \langle \text{identifier} \rangle () \langle \text{statement} \rangle$ $::= \langle \text{retType} \rangle \langle \text{identifier} \rangle (\langle \text{formals} \rangle) \langle \text{statement} \rangle$
$\langle \text{formals} \rangle$	$::= \langle \text{formal} \rangle \{ , \langle \text{formal} \rangle \}$
$\langle \text{formal} \rangle$	$::= \langle \text{identifier} \rangle : \langle \text{declType} \rangle$
$\langle \text{retType} \rangle$	$::= \langle \text{basicType} \rangle$

Figure A.8: Grammar Fragment for the Operator Section.

```

⟨neighborhood⟩ ::= try { ⟨transformation⟩ } end
                ::= ⟨transformation⟩
⟨transformation⟩ ::= Pr( ⟨expr⟩ ) : ⟨moveInstr⟩
                ::= default      : ⟨moveInstr⟩
                ::= when ⟨bool Expr⟩ : ⟨moveInstr⟩
                ::= ⟨moveInstr⟩
⟨moveInstr⟩      ::= ⟨method⟩ move op( $x_1, \dots, x_n$ )
                where ⟨letExpr1⟩
                    ...
                    ⟨letExprn⟩
                    [accept when ⟨AcceptanceCriterion⟩]
⟨method⟩ ::= first
          ::= best
          ::= any
⟨letExpr⟩ ::= ⟨identifier⟩ from {⟨Set Body⟩} [⟨optClause⟩]
          ::= ⟨identifier⟩ from {⟨expr1⟩, ..., ⟨exprn⟩} [⟨optClause⟩]
          ::= ⟨identifier⟩ from {⟨identifier⟩ : ⟨type⟩ | ⟨select1⟩...⟨selectn⟩} [⟨optClause⟩]
          ::= ⟨identifier⟩ from ⟨lvalue⟩ [⟨optClause⟩]
          ::= ⟨identifier⟩ = ⟨expr⟩
          ::= maximizing ⟨expr⟩
          ::= minimizing ⟨expr⟩

```

Figure A.9: The Syntax of move Instructions.

```

⟨AcceptCriterion⟩ ::= ⟨AcceptStatement⟩ {cor ⟨AcceptStatement⟩}
                  ::= in resulting state ⟨AcceptStatement⟩ {cor ⟨AcceptStatement⟩}
                  ::= in current state ⟨AcceptStatement⟩ {cor ⟨AcceptStatement⟩}
⟨AcceptStatement⟩ ::= ⟨AcceptCondition⟩
                  ::= ⟨AcceptCondition⟩ → ⟨Statement⟩
                  ::= Pr( ⟨expr⟩ ) : ⟨AcceptCondition⟩ → ⟨Statement⟩
                  ::= default      : ⟨AcceptCondition⟩ → ⟨Statement⟩
⟨AcceptCondition⟩ ::= always
                  ::= improvement
                  ::= noDecrease
                  ::= ⟨expr⟩
                  ::= ⟨AcceptCondition⟩ and ⟨AcceptCondition⟩
                  ::= ⟨AcceptCondition⟩ or ⟨AcceptCondition⟩
                  ::= not ⟨AcceptCondition⟩

```

Figure A.10: Grammar Fragment for the Acceptance Criterion.

Bibliography

- [1] E. Aarts and J.K. Lenstra. *Local Search in Combinatorial Optimization*. Wiley-Interscience Series in Discrete Mathematics and Optimization, John Wiley & Sons Ltd, England, 1997.
- [2] Bowen Alpern, Roger Hoover, Barry K. Rosen, Peter F. Sweeney, and F. Kenneth Zadeck. Incremental evaluation of computational circuits. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 32–42, San Francisco, California, 22–24 January 1990.
- [3] D. Applegate and W. Cook. A Computational Study of the Job Shop Scheduling Problem. *ORSA J. of Comp.*, 3(2):149–156, 1991.
- [4] L. Barford and B. Vander Zanden. Attribute grammars in constraint-based graphics system. *Software Practice and Experience*, 19(4):309–328, April 1989.
- [5] J. Bisschop and A. Meeraus. On the Development of a General Algebraic Modeling System in a Strategic Planning Environment. *Mathematical Programming Study*, 20:1–29, 1982.
- [6] A. Borning. The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Transaction on Programming Languages and Systems*, 3(4):353–387, 1981.
- [7] A. Borning, B. Freeman-Benson, and M. Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, September 1992.
- [8] Y. Caseau and F. Laburthe. Claire: a brief overview. Technical report, LIENS, École normale supérieure, 1995.
- [9] G.A. Cheston. *Incremental algorithms in graph theory*. PhD thesis, Department of Computer Science, University of Toronto, Toronto, Canada, March 1976.

- [10] N. Christofides, A. Mingozzi, and P. Toth. *Combinatorial Optimization*, chapter The vehicle routing problem. Wiley, Chichester, 1979.
- [11] G. Clarke and J.W. Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research*, 12:568–581, 1964.
- [12] A. Colmerauer. An Introduction to Prolog III. *Commun. ACM*, 28(4):412–418, 1990.
- [13] Y. Colombani. Constraint Programming: An Efficient and Practical Approach to Solving the Job-Shop Problem. In *Second International Conference on Principles and Practice of Constraint Programming (CP'96)*, Cambridge, MA, August 1996. Springer Verlag.
- [14] T.H. Cormen, C.E. Leiserson, and R.L Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, 1990.
- [15] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, December 1988.
- [16] S. Even and H. Gazit. Updating distances in dynamic graphs. In P. Brucker and R. Pauly, editors, *IX Symposium on Operations Research*, volume 49 of *Methods of Operations Research*. Verlag Anton Hain, 1985.
- [17] Fisher. Optimal solutions of vehicle routing problems using minimum K-trees. *Operations Research*, 42:626–642, 1994.
- [18] C.L. Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, 19:17–37, 1982.
- [19] R. Fourer, D. Gay, and B.W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. The Scientific Press, San Francisco, CA, 1993.
- [20] M.S. Fox. Constraint-Directed Search: A Case Study of Job-Shop Scheduling. Technical Report CMU-CS-83-161, Carnegie-Mellon University, December 1983.
- [21] M. Gendreau, A. Hertz, and G. Laporte. A Tabu Search Heuristic for the Vehicle Routing Problem. *Management Science*, 40:1276–1290, 1994.
- [22] F. Glover. Tabu Search—Part I. *ORSA Journal on Computing*, 1(3):190–206, 1989.

- [23] F. Glover. Tabu Search—Part II. *ORSA Journal on Computing*, 2(1):4–32, Winter 1990.
- [24] Bruce L. Golden, G. Laporte, and Éric D. Taillard. An adaptative memory heuristic for a class of vehicle routing problems with minmax objective. Technical report, Centre de recherche sur les transports, November 1995.
- [25] Y. Hamadi and D. Merceron. Reconfigurable architectures: A new vision for optimization problems. In Gert Smolka, editor, *Principle and Practice of Constraint Programming - CP97*, Lecture Notes in Computer Science, pages 209–221. Springer, October 1997.
- [26] M. Henz, G. Smolka, and J. Würtz. Oz—a programming language for multi-agent systems. In Ruzena Bajcsy, editor, *13th International Joint Conference on Artificial Intelligence*, volume 1, pages 404–409, Chambéry, France, 30 August–3 September 1993. Morgan Kaufmann Publishers.
- [27] J. Jaffar, S. Michaylov, P.J. Stuckey, and R. Yap. The CLP(\mathcal{R}) language and system. *ACM Trans. on Programming Languages and Systems*, 14(3):339–395, 1992.
- [28] D. Johnson, C. Aragon, L. McGeoch, and C. Schevon. Optimization by Simulated Annealing: An Experimental Evaluation; Part I, Graph Partitioning. *Operations Research*, 37(6):865–893, 1989.
- [29] D. Johnson, C. Aragon, L. McGeoch, and C. Schevon. Optimization by Simulated Annealing: An Experimental Evaluation; Part II, Graph Coloring and Number Partitioning. *Operations Research*, 39(3):378–406, 1991.
- [30] D.S. Johnson, C.H. Papadimitriou, and M. Yannakakis. How easy is local search ? *Journal of Computer and System Sciences*, 37:79–100, 1988.
- [31] S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by Simulated Annealing. *Science*, 220:671–680, 1983.
- [32] G. Laporte. The Vehicle Routing Problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59:345–358, 1992.
- [33] Dell’Amico M. and Trubian M. Applying Tabu Search to the Job-Shop Scheduling Problem. *Annals of Operations Research*, 41:231–252, 1993.

- [34] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *J. Chem. Phys.*, 21(6):1087–1092, 1953.
- [35] L. Michel and Van Hentenryck P. Localizer: A Modeling Language for Local Search and its Implementation. *Constraints*, 1998.
- [36] L. Michel and P. Van Hentenryck. Localizer: A modeling language for local search. In *Second International Conference on Principles and Practice of Constraint Programming (CP'97)*, Linz, Austria, October 1997. (Extended Version invited to the special of *Constraints* on CP'97).
- [37] L. Michel and P. Van Hentenryck. Localizer: A Modeling Language for Local Search. *INFORMS*, 1998.
- [38] S. Minton, M.D. Johnston, and A.B. Philips. Solving Large-Scale Constraint Satisfaction and Scheduling Problems using a Heuristic Repair Method. In *AAAI-90*, August 1990.
- [39] B. Myers. The garnet user interface development environment; a proposal. Technical Report CMU-CS-88-153, Carnegie Mellon University, Carnegie Mellon University, Pittsburgh, PA 15213, September 1988.
- [40] B. Myers. Comprehensive support for graphical, highly-interactive user interface: The garnet user interface development environment. *IEEE Computer*, 1990.
- [41] B. Myers, D. Giuse, A. Mickish, and D. Kosbie. Making structured graphics and constraints practical for large-scale applications. Technical Report CMU-CS-94-150, Carnegie Mellon University, Carnegie Mellon University, Pittsburgh, PA 15213, May 1994.
- [42] I.H. Osman. *Metastrategy Simulated Annealing and Tabu Search for Combinatorial Optimization Problems*. PhD thesis, The Management School, Imperial College of Science and Medicine, University of London, London, 1991.
- [43] I.H. Osman. Metastrategy Simulated Annealing and Tabu Search Algorithms for the Vehicle Routing Problem. *Annals of Operations Research*, 41:421–451, 1993.
- [44] R. Paige. *Formal Differentiation*. PhD thesis, Dept. Of Computer Science, New York University, New York, N.Y., June 79 1981.

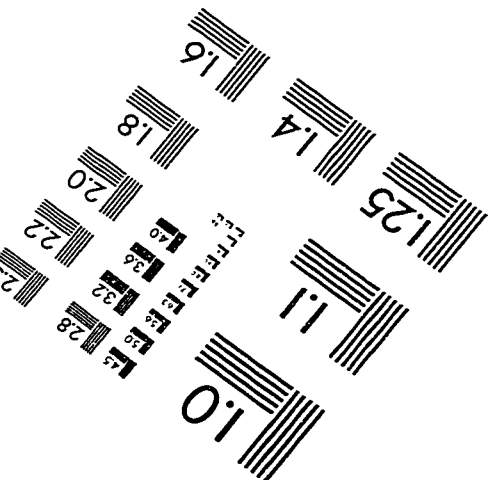
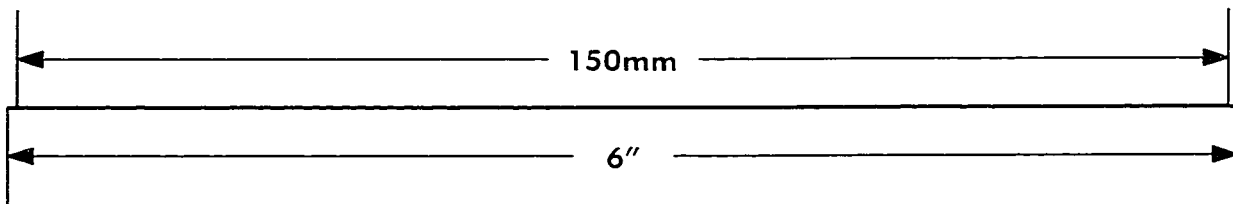
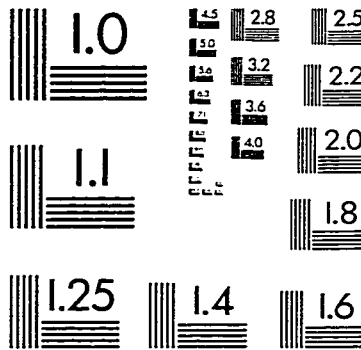
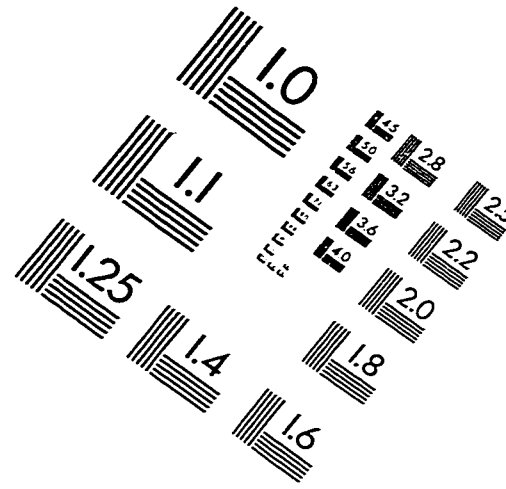
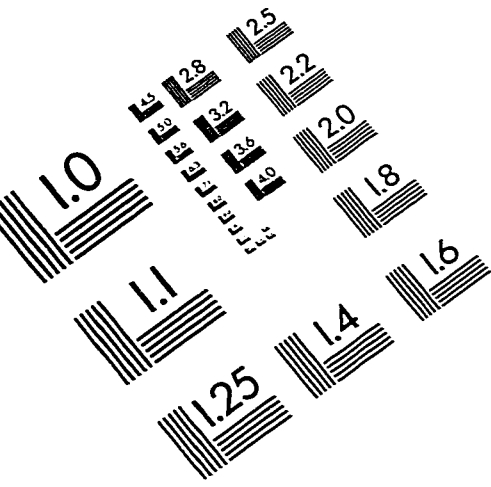
- [45] R. Paige. Programming with invariants. *IEEE Software*, pages 56–69, January 1986.
- [46] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, July 1982.
- [47] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, NJ, 1982.
- [48] G. Pesant, M. Gendreau, and J.M. Rousseau. Genius-cp: A generic single-vehicle routing algorithm. In Gert Smolka, editor, *Principle and Practice of Constraint Programming - CP97*, Lecture Notes in Computer Science, pages 420–434. Springer, October 1997.
- [49] J.F. Puget. A C++ Implementation of CLP. In *Proceedings of SPICIS*, November 1994.
- [50] W.W. Pugh. *Incremental computation and the incremental evaluation of functional programs*. PhD thesis, Department of computer science, Cornell University, Ithaca, NY, August 1988.
- [51] G. Ramalingam. *Bounded Incremental Computation*. PhD thesis, University of Wisconsin-Madison, 1993.
- [52] G. Ramalingam and T. Reps. On competitive on-line algorithms for the dynamic priority-ordering problem. *Information Processing Letters*, 51:155–161, 1994.
- [53] G. Ramalingam and Repts T. On the computational complexity of incremental algorithms. Technical report, University of Wisconsin-Madison, August 1991.
- [54] T. Reps. *Generating Language-based Environments*. MIT Press, Cambridge, Ma, 1984.
- [55] T. Reps and A. Demers. Sublinear-space evaluation algorithms for attribute grammars. *ACM Transactions on Programming Languages and Systems*, 9(3), July 87.
- [56] T. Reps, T. Teitelbaum, and A. Demers. Incremental Context-Dependent Analysis for Language-based Editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, July 83.
- [57] Y. Rochat and D. Éric. Taillard. Probabilistic Diversification and Intensification in Local Search for Vehicle Routing. *Journal of heuristics*, 1:147–167, 1995.

- [58] M. Sannella and A. Borning. Multi-garnet: Integrating multi-way constraints with garnet. Technical Report 92-07-01, University of Washington, 92.
- [59] M. Sannella, J. Maloney, B. Freeman-Benson, and A. Borning. Multi-way versus one-way constraints in user interfaces: Experience with the deltablue algorithm. *Software Practice and Experience*, 23(5):529–566, May 1992.
- [60] M. Sannella. Analysing and Debugging Hierarchies of Multi-Way Local Propagation Constraints. In A. Borning, editor, *Principles and Practice of Constraint Programming*. Lecture Notes in Computer Science 874, Springer Verlag, 1994.
- [61] M. Sannella. The SkyBlue Constraint Solver and its Applications. In V. Saraswat and P. Van Hentenryck, editors, *Principles and Practice of Constraint Programming*. The MIT Press, Cambridge, Massachusetts, 1995.
- [62] M. Sannella, J. Maloney, B. Freeman-Benson, and A. Borning. Multi-way versus One-way Constraints in User Interfaces: Experience with the DeltaBlue Algorithm. *Software Practice and Experience*, 23(5), May 1993.
- [63] J.E. Savage and Markus G. Wloka. Parallelism in Graph-Partitioning. *Journal of Parallel and Distributed Computing*, 13:257–272, 1991.
- [64] D. Schmidt. *Denotational Semantics*. Wm. C. Brown Publishers, Dubuque, Iowa, 1988.
- [65] Linus E. Lindo Schrage. *Optimization Modeling with LINDO*. Duxbury, 5th edition, February 1997.
- [66] B. Selman and H. Kautz. An Empirical Study of Greedy Local Search for Satisfiability Testing. In *AAAI-93*, pages 46–51, 1993.
- [67] B. Selman and H. Kautz. Domain-Independent Extensions to GSAT: Solving Large Structured Satisfiability Problems. In *Proceedings of IJCAI-93*, 1993.
- [68] B. Selman, H. Levesque, and D. Mitchell. A New Method for Solving Hard Satisfiability Problems. In *AAAI-92*, pages 440–446, 1992.
- [69] Marius M. Solomon. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations Research*, 35(2):254–265, April 1987.
- [70] J. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge Mass., 1977.

- [71] P. Stuckey and V. Tam. Models for Using Stochastic Constraint Solvers in Constraint Logic Programming. In *PLILP-96*, Aachen, August 1996.
- [72] I.E. Sutherland. *SKETCHPAD: a Man-Machine Graphical Communication System*. Cambridge, MA, MIT Lincoln Labs, 1963.
- [73] Éric D. Taillard. Parallel, Iterative Search Methods for Vehicle Routing Problem. *Networks*, 23:661–676, 1993.
- [74] Éric D. Taillard, G. Laporte, and M. Gendreau. Vehicle routing with multiple use of vehicles. Technical Report 964, Centre de recherche sur les transports, March 1995.
- [75] R.E. Tarjan. Amortized Computational Complexity. *SIAM Journal of Algebraic Discrete Methods*, 6:306–318, 1985.
- [76] R.J.M. Vaessens. *Generalized Job Shop Scheduling: Complexity and Local Search*. PhD thesis, Eindhoven University of Technology, September 1995.
- [77] R.J.M. Vaessens, E.H.L. Aarts, and J.K. Lenstra. Job shop scheduling by local search. *INFORMS Journal on Computing*, 8(3):302–317, 1996.
- [78] P. Van Hentenryck. Constraint Programming. *Encyclopedia of Computer Science and Technology*, 36(21):35–71, 1997.
- [79] P. Van Hentenryck. *OPL: The Optimization Programming Language*. The MIT Press, Cambridge, Mass., 1998.
- [80] P. Van Hentenryck, L. Michel, and Y. Deville. *Numerica: a Modeling Language for Global Optimization*. The MIT Press, Cambridge, Mass., 1997.
- [81] B. Vander Zanden. An incremental algorithm for satisfying hierarchies of multi-way, dataflow constraints. *ACM Transactions on Programming Languages and Systems*, 18(1):30–72, Jan 1996.
- [82] Molly Ann. Wilson. *Hierarchical Constraint Logic Programming*. PhD thesis, Department of Computer Science and Engineering, University of Washington, April 1993.
- [83] D.M. Yellin and Strom R.E. INC: A Language for Incremental Computations. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI)*, pages 115–124, Atlanta, Georgia, 22–24 June 1988. *SIGPLAN Notices* 23(7), July 1988.

- [84] J. Zhou. A Constraint Program for Solving the Job-Shop Problem. In *Second International Conference on Principles and Practice of Constraint Programming (CP'96)*, Cambridge, MA, August 1996. Springer Verlag.

IMAGE EVALUATION TEST TARGET (QA-3)



APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved

