BROWN UNIVERSITY
Department of Computer Science
Master's Project
CS-89-M7

"Using Influence Diagrams in Recognizing Locally-Distinctive Places"

by
Robert Alan Chekaluk

Using Influence Diagrams in Recognizing Locally-Distinctive Places

by

Robert Alan Chekaluk

B.S., University of Michigan, 1982

Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science at Brown University.

May 1990

This thesis by Robert Alan Chekaluk

is accepted in its present form by the Department of Computer Science

as satisfying the thesis requirement for the degree of Master of Science

Date 10/20/85 Thomas L. Dean

Thomas L. Dean

Using Influence Diagrams in Recognizing Locally-Distinctive Places

Robert A. Chekaluk

October 20, 1989

Abstract

A method is described for using influence diagrams for the high-level control of a mobile robot in the vicinity of locally-distinctive places. The mobile robot has the mission of recognizing each locally-distinctive place it encounters. In order to recognize a locally-distinctive place, the diagrams are used to select sensor actions and to perform inference over sensor results.

1 Introduction

In robot exploration, there seems to be a consensus emerging for qualitative and topological models of space. Kuipers [1] and Levitt *et al.* [2] have proposed a topological model consisting of nodes and arcs, where nodes correspond to *locally-distinctive places* (LDP's), and arcs correspond to direct travel routes between LDP's. An LDP is a place that is geometrically distinguished from its immediate surroundings.

In robot exploration, the robot must be able to recognize each LDP it encounters. Recognition provides the robot with the local topology of the LDP, allowing it to continue exploration by continuing through the LDP. The robot can recognize LDP's by classifying them into classes of a common local topology.

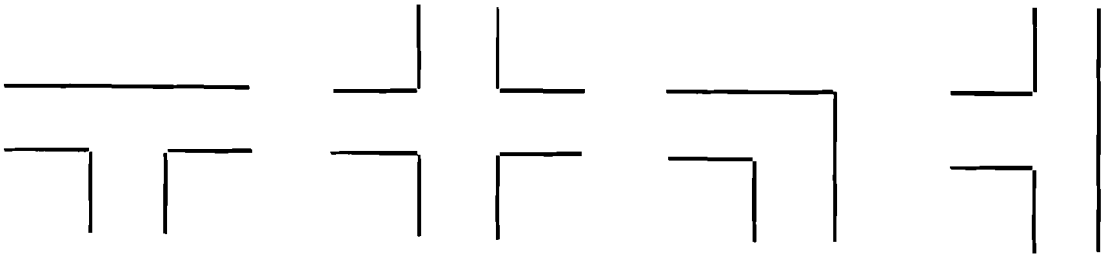

Figure 1: Example LDP equivalence classes

We report on a system for classifying LDP's encountered by a mobile robot wandering randomly in an indoor environment. This system is a closely related, but simplified, version of work by Levitt *et al.* [3, 4] and Cameron and Durrant-Whyte [5]. It also shares ideas with Hutchinson *et al.* [6] and Hager [7]. The robot's environment consists solely of corridors and doorways into the corridors; there are no large areas of open space. In this environment, corridors correspond to arcs, while corridor junctions or doorways into corridors correspond to LDP's. We assume that the corridors are nearly straight, and that the corridors intersect at nearly right angles.

In order to classify LDP's, we partition the set of all LDP's the robot will encounter into equivalence classes, with each equivalence class representing a similar topology. Examples of these classes include Forward-T-junction, Cross, Left-L-junction, and Left-T-junction (Figure 1). Given that the robot is in some LDP, the problem is to determine which equivalence class the LDP belongs to. Using the terminology of Bayse *et al.* [8], we assume all LDP's are distinguishable.

We approach this problem using influence diagrams for decision analysis and inference. The decision problem is to decide which sensing operation, from a set of candidate operations, to institute at each cycle. The inference problem is to combine the results of multiple sensing operations to maintain a set of beliefs about the LDP being classified.

We are not concerned with low-level interpretation of raw sensor data as in Walter [9] or Moravec and Elfes [10]. Whereas Kuipers' hill-climbing

strategy for finding the exact location of an LDP [11] assumes continuous error-free low-level sensing, we assume error always exists in low-level sensing. To compensate for this error, we employ complex, but atomic, sensing operations. The hill-climbing strategy is not applicable with these complex operations. We consider a node as encompassing the local area around an LDP, and we merely need to reach its vicinity via an arc. From the vicinity of an LDP, we are able to sense and classify it.

2 Describing Locally-Distinctive Places

We assume the environment contains a set of LDP's L. Each $l \in L$ represents an LDP found in the environment. $C = \{C_1, C_2, ..., C_n\}$ is a partition of L. Each C_i is an equivalence class of LDP's having a similar topology, and is characterized by a model LDP. This model LDP represents the ideal topology of LDP's in the class. There is a set of features F, which is the set of all possible identifiable subparts of LDP's. Examples of features are convex corners, concave corners, and walls. Every LDP class is decomposed into constituent features $f \in F$, that stand in some spatial relation to each other. Each LDP class is uniquely characterized by its features and their spatial relationships.

To specify the spatial relationships, we postulate a coordinate system in the vicinity of each model LDP. This local coordinate system is based on a local reference point and orientation, and is like the local coordinate system of [2], except that it cannot be reacquired by sensory action. The reference point is maintained by dead reckoning alone. We arbitrarily designate this reference point as that point where an ideal robot would instantaneously notice that it is no longer in a corridor (Figure 2). The reference orientation is arbitrarily designated as a vector originating from the reference point, parallel to the corridor the robot just emerged from.

The spatial relations among consitituent features in a model LDP may be described by specifying the location and orientation of each feature in the local coordinate system. However, to simplify this implementation, we assume that the orientation can be ignored. This is a reasonable assumption because there are no LDP's where orientation is required to distinguish them.

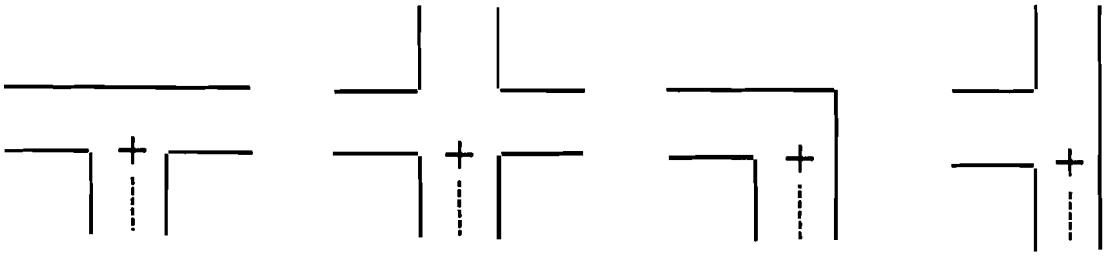

Figure 2: LDP reference point

The spatial relationships may be completely specified by the locations of the features.

We can specify a feature location by the polar coordinates of a reference point on the feature. However, it is not necessary to include distance in locating features. All LDP's in the environment may be uniquely characterized solely by the angular locations of features. This is because there are no LDP's where feature distance is required to distinguish them.

We partition angular space about the reference point into a set of equiangular wedges W. This partition is oriented with respect to the reference orientation. The *location* $w \in W$ of a feature is determined using a function $wfy : \theta \rightarrow W$, where θ is from its polar coordinates. w tells us which pie section a feature is in.

3 System Terminology

In order to determine which LDP the robot is in, we must determine which features are actually present in the environment and where they are located. A *feature detector* is a function $d_f : (W \times L) \rightarrow \{0, 1\}$ that performs sensing operations to determine whether a feature is present in the specified location and LDP. Each feature detector can be considered as a specialized "macrosensor" that senses only occurrences of its associated feature. Note that the LDP l is not passed explicitly to the function, but is implicit and represents

the current LDP under scrutiny. A set of these feature detectors is assumed by this system. Feature detectors may be aimed in various directions to look for features in different locations. An *aimed detector* or *detector* is a function $d_{f,w} : L \rightarrow \{0,1\}$, where $d_{f,w}(l) = d_f(w,l)$. An aimed detector is an aimed instance of a feature detector. We also assume the existence of a detector $d_{0,0}$ that performs a noop sensing operation.

There is a set of hypotheses $H \subseteq C$ on the identity of the current LDP. We have a belief function

$$Bel : C \rightarrow \Re, \text{ such that } 0 \leq Bel(C_i) \leq 1 \text{ and } \sum_{C_i \in C} Bel(C_i) = 1$$

that supplies the current probabilistic belief value for each hypothesis. These beliefs must reflect the results of all previous sensing operations. Once a feature detector has been invoked, the problem of *sensor fusion* is to incorporate its result and determine a new belief distribution [1].

Let D be the set of all possible aimed detectors $d_{f,w}$. A *descriptor* $s_i \in 2^D$ for class C_i is a set of aimed detectors that should return 1 if invoked in an LDP of class C_i. Each descriptor uniquely characterizes its LDP class by providing a list of the constituent features and their locations in the model LDP for that class.

There is a pool of available detectors

$$P_0 = \bigcup_{C_i \in H} \{d_{f,w} \mid d_{f,w} \in s_i \text{ or } d_{f,w} = d_{0,0}\}.$$

By selectively invoking these detectors, the robot attempts to gather enough information to classify the LDP. After detector $d_{f,w}$ is invoked, $P_{i+1} = P_i - \{d_{f,w}\}$. For a given LDP, let D_{inv} be the set of detectors that have been invoked.

The detectors may only be invoked sequentially, therefore the problem of *sensor selection* is to determine the invocation sequence of the detectors. We do not calculate this sequence exhaustively at the outset, but use a greedy algorithm to select the next detector. We may, at any time, choose

[1] This incarnation of multi-sensor fusion is simpler and of higher level than the usual treatments of multi-sensor fusion [12, 13, 14]. While our "macro-sensors" can be viewed as different sensors, their results are all of the same form and are easily fused.

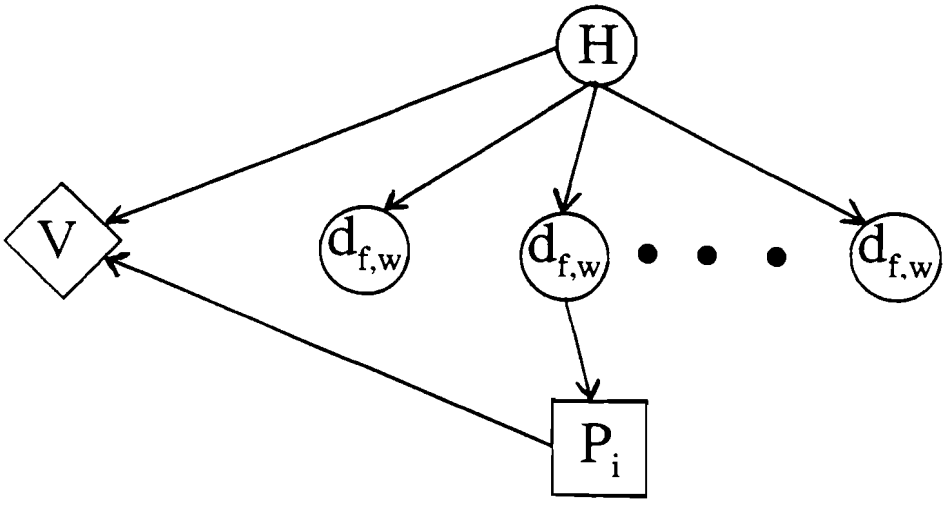

Figure 3: Influence diagram for LDP recognition

to invoke $d_{0,0}$. Invocation of $d_{0,0}$ implies ending the recognition process and committing to the hypothesis with the highest belief. We cast this task as a decision problem seeking to choose, within the current context, the detector from the pool (including $d_{0,0}$) that has the greatest utility. We use a decision influence diagram to evaluate the possible choices of detectors in light of a utility function.

4 Overview of System Operation

An influence diagram [15] (Figure 3) is used to select each detector from P_i and to assimilate the ensuing results into the hypothesis belief distribution. It has chance nodes representing the hypotheses in H, the detectors in P_0, and a value and decision node for the detectors. Different portions of this diagram are used for sensor fusion and sensor selection.

When, after traversing a corridor, the robot reaches a place that no longer qualifies as a corridor, it is, by definition, in an LDP. The decision of what constitutes a corridor is relegated to lower level processing. The classification task is to decide which class the LDP belongs to. The classification begins with a set of initial LDP hypotheses H, and assumes that the LDP belongs to one of these classes. How this initial set is arrived at is unimportant to the recognition processes. A methodology for arriving at an optimal initial

set could be derived, however for simplicity, we create hypotheses for each $C_i \in C$. This is a reasonable simplification because the cardinality of C is small and because the sensor selection algorithm converges on an answer quickly. Each hypothesis in this initial set has a descriptor. We form P_0 by pooling all detectors from the descriptors in H. All invoked detectors will be selected from this pool.

A cycle is begun: One aimed detector from the pool P_i is selected and invoked. Its result is used to update the belief distribution. Exactly one detector will be selected and invoked during each cycle. Under normal circumstances, a detector will only be invoked once. This cycle continues until either the LDP is recognized with certainty, or the cost of invoking further detectors outweighs the benefit. The best hypothesis is then returned as the result of the classification.

After invoking a detector, it is removed from P_i to produce P_{i+1}, unless it had aborted during execution. In this case, $P_{i+1} = P_i$ so that the detector is eligible to be reinvoked. Detectors may be reinvoked only a limited number of times before they are finally removed from P_i (but not added to D_{inv}).

5 Sensor Fusion

The influence diagram of figure 3 is used to incrementally assimilate the acquired detector evidence. Only the chance nodes in the diagram are necessary for this function; the decision and value nodes are extraneous. After each detector is invoked, we incorporate its results into the diagram, and evaluate it to obtain a new belief distribution.

Each aimed detector $d_{f,w}$ provides three items of information: the feature f that was looked for, the location w where it was looked for, and the result r representing whether it was found. Although in general there will be error in the result r, for simplicity, we assume it is correct. To use these items of information in obtaining a new belief distribution, the influence diagram must capture how all LDP's in L are decomposed into their constituent features and feature locations.

The diagram has a single chance node representing the hypotheses H. The range of this hypothesis node is C. The diagram also contains a single

Convex corner at 45°						
Existence	T_{forw}	T_{left}	T_{right}	L_{left}	L_{right}	Cross
no	1.0	1.0		1.0	1.0	
yes			1.0			1.0

Concave corner at 315°						
Existence	T_{forw}	T_{left}	T_{right}	L_{left}	L_{right}	Cross
no	1.0	1.0	1.0	1.0		1.0
yes					1.0	

Wall at 90°						
Existence	T_{forw}	T_{left}	T_{right}	L_{left}	L_{right}	Cross
no		1.0	1.0	1.0	1.0	1.0
yes	1.0					

Figure 4: Example $Pr\{f, w, X_{f,w} \mid C_i\}$ for "detector" nodes

chance node for every possible aimed detector $d_{f,w}$; there is a "detector" node for each element of $F \times W$. In this approach, each detector node represents $X_{f,w}$, the boolean random variable representing the existence or nonexistence of the specified feature at the specified location. Each $X_{f,w}$ has a range of $\{0, 1\}$

Each $X_{f,w}$ node is conditioned on the hypothesis node, giving each a prior distribution of $Pr\{f, w, X_{f,w} \mid C_i\}$. The set of prior distributions of all "detector" nodes captures how LDP's are built from features in certain locations. These distributions are like the "membership functions" of [5]. These distributions are obtained by inspection of the LDP classes. See figure 4 for examples of these distributions. The distribution $Pr\{C_i\}$ for the hypothesis node is initially set to the prior belief distribution, and is not changed.

To determine a new belief distribution, the diagram must be evaluated. The hypothesis node is set as the goal node. When a detector is invoked, the corresponding $X_{f,w}$ node is set as a conditioning node. The diagram is then evaluated using the algorithm of Rege and Agogino [16], producing

the distribution $Pr\{C_i \mid D_{inv}\}$. This distribution is exactly the needed new belief distribution. Each successive detector that is invoked causes another detector node to be marked as a conditioning node. Previous conditioning nodes remain as conditioning nodes.

6 Sensor Selection

The influence diagram of figure 3 is also used to select aimed detectors from P_i for invocation. The single decision node in the diagram represents the choices of detector. The value node in the diagram and its corresponding utility function attempt to measure the utility of each choice.

Precedessors of the value node represent parameters to the utility function. As will be shown, only the hypotheses and the the uninvoked detectors influence the utility function. Because the decision node covers the uninvoked detectors, the hypothesis and decision nodes are the only nodes connected to the value node.

Predecessors of the decision node represent the information known at decision time. The only information known at decision time are the results of the invoked detectors. Therefore, the detector nodes representing detectors in D_{inv} are connected to the decision node; the nodes representing uninvoked detectors provide no context and are extraneous. As each detector is invoked, an arc is added dynamically between the corresponding detector node and the decision node. At any time, only a subset of all detector nodes will be connected to the decision node.

To select a detector for invocation, the diagram must be evaluated. The value node is first marked as the goal node. The diagram is evaluated using the IDES "greedy" algorithm of Agogino and Ramamurthi [17] to produce a decision policy for choosing a detector. This decision policy is a function $Policy : D_{inv} \rightarrow P_i$, and dictates which detectors to choose. Also associated with the decision policy is an expected value function $EV : D_{inv} \rightarrow \Re$. The detector to invoke is found by indexing into *Policy* using the results of all invoked detectors in D_{inv}.

6.1 The Utility Function

Because the utility function is the prime determinant of which detectors are chosen, we will describe the utility function. We seek to invoke a detector from P_i based on the following factors:

- the ability of the detector to discriminate between hypotheses
- the cost of deploying the detector
- the probability that the currently best hypothesis is correct
- the cost of misidentifying the LDP

To simplify the implementation, we currently incorporate only the first two of the above factors into the utility function. The two remaining factors are used to decide when to choose $d_{0,0}$ and stop the recognition process. This decision is simplified in our implementation using a threshold approach where beliefs above a threshold are considered "proven" and beliefs below a separate threshold are considered "disproven." The process stops when the belief of exactly one hypothesis is "proven" and the beliefs of the remaining hypotheses are "disproven." If no hypothesis becomes "proven" once $P_i = \{d_{0,0}\}$, the hypothesis with the largest belief value is returned as the best guess.

We can capture these intuitive notions by a utility function $Util : (P_i \times H) \rightarrow \Re$, where

$$Util(d_{f,w}, h) = k_1 \times Discrim(d_{f,w}) - k_2 \times Cost(d_{f,w}, h)$$

for $d_{f,w} \in P_i$, and $h \in H$. The discriminant and cost measures are described below. The constants k_1 and k_2 are arbitrary constants used for scaling and weighting. Similar constants k_3 and k_4 will also be used.

The discrimination function $Discrim : P_i \rightarrow \Re$ is the function of Cameron and Durrant-Whyte [5] applied to our domain. To derive this discriminant, first consider the function

$$U(r, d_{f,w}) = \sum_{C_i \in C} |Pr(C_i \mid r, d_{f,w}) - Pr(C_i)|$$

where $r = d_{f,w}(l)$ in some LDP l. This summation of differences gives high values to detectors that strongly alter the beliefs about the current LDP, and gives low values to detectors that do not alter these beliefs (provide no new information). The discrimination function is obtained from this function by calculating the expected value of U over the distribution of detector results. That is,

$$Discrim(d_{f,w}) = \sum_{r \in X_{f,w}} U(r, d_{f,w}) \times Pr(r, d_{f,w})$$

or equivalently,

$$Discrim(d_{f,w}) = \sum_{C_i \in C} Pr(C_i) \left[\sum_{r \in X_{f,w}} |Pr(r, d_{f,w} \mid C_i) - Pr(r, d_{f,w})| \right]$$

The distributions needed for this formula are easily obtained. The distribution $Pr(C_i)$ is the hypothesis belief distribution. The distribution $Pr(r, d_{f,w} \mid C_i)$ is the distribution associated with the corresponding $X_{f,w}$ node. The probability of each detector result $Pr(r, d_{f,w})$ can be calculated using

$$Pr(r, d_{f,w}) = \sum_{C_i \in C} Pr(r, d_{f,w} \mid C_i) \times Pr(C_i).$$

An additional factor in the utility function is the cost of the detectors. The cost measure is a function $Cost : (P_i \times H) \rightarrow \Re$ and is intended to measure the cost of performing the detector. The major cost of each detector is the *time* each takes to execute as in Hager [7, p. 32] and Levitt *et al.* [3]. Because we seek to find the simplest detector for the closest feature, we decompose the cost measure into two components.

The *complexity* of the detector measures how complex each is to execute. Complexity is a function $Complexity : P_i \rightarrow \Re$ and can be characterized by the amount of time the detector requires, or the distance the robot traverses, while executing the detector. An additional cost factor is the *latency* required before the detector begins. Latency is a function $Latency : (P_i \times H) \rightarrow \Re$ and may also be characterized by the time or distance needed to reach the point where the detector may begin. These two factors combine to form the cost measure as follows:

$$Cost(d_{f,w}, h) = k_3 \times Complexity(d_{f,w}) + k_4 \times Latency(d_{f,w}, h)$$

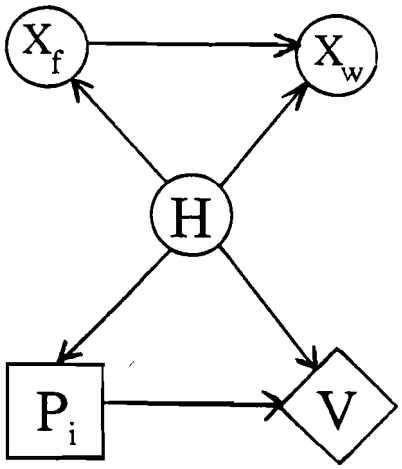

Figure 5: Compact influence diagram

7 Compact Influence Diagram

There are other ways of decomposing the problem to capture the relationship between LDP's and their constituent features and feature locations. We have experimented with and report on a different decomposition and the associated influence diagram.

This alternative influence diagram (Figure 5) is a more compact representation than the previous one. As before, there is a hypothesis node representing the possible LDP classes. However, there are no "detector" nodes. This diagram has a single chance node representing X_f, the random variable for feature type, and a single chance node representing X_w, the random variable for feature location.

Let f_{null} be the null feature. We say that f_{null} exists in those locations of an LDP where no feature in F is found. Let $F_{null} = F \cup f_{null}$. The range of the X_f node is F_{null}, and the range of the X_w node is W.

The X_f node is conditioned on the hypothesis node, giving it a prior distribution of $Pr\{f \mid C_i\}$, for $f \in F_{null}, C_i \in C$. The X_w node is conditioned on both the X_f node and on the hypothesis node, giving it a prior distribution of $Pr\{w \mid f, C_i\}$, for $w \in W$. These two prior distributions capture how LDP's are built from features in certain locations. As with the previous diagram, these distributions are obtained by inspection of the LDP classes. See figures 6 and 7 for examples of these distributions.

f	T_{forw}	T_{left}	T_{right}	L_{left}	L_{right}	Cross
Convex	0.25	0.25	0.25	0.125	0.125	0.5
Concave				0.125	0.125	
Wall	0.125	0.125	0.125			
f_{null}	0.625	0.625	0.625	0.75	0.75	0.5

Figure 6: Example $Pr\{f \mid C_i\}$ for compact influence diagram

Convex corner						
w	T_{forw}	T_{left}	T_{right}	L_{left}	L_{right}	Cross
0						
45			0.5			0.25
90	0.5		0.5		1.0	0.25
135						
180						
225						
270	0.5	0.5		1.0		0.25
315		0.5				0.25

Wall						
w	T_{forw}	T_{left}	T_{right}	L_{left}	L_{right}	Cross
0	1.0			0.125	0.125	0.125
45				0.125	0.125	0.125
90		1.0		0.125	0.125	0.125
135				0.125	0.125	0.125
180				0.125	0.125	0.125
225				0.125	0.125	0.125
270			1.0	0.125	0.125	0.125
315				0.125	0.125	0.125

Figure 7: Example $Pr\{w \mid f, C_i\}$ for compact influence diagram

Listing for LDP-Recognition Code

The following modules are used in the LDP-recognition application
for Huey:

ldp.h	header file for this application
hueymain.c	main Huey driver routine
location.c	routines that do LDP recognition
communic.c	routines that send messages to other processes
network.c	defines the influence diagram and utility function

```
#define  ROBOT_RUN      1            /*  1 if robot, 0 if offline test  */

/*--------------------*/
/*  Define LDP types  */
/*--------------------*/
#define  TFORW          0
#define  TLEFT          1
#define  TRIGHT         2
#define  LLEFT          3
#define  LRIGHT         4
#define  CROSS          5
#define  RDOOR1         6
#define  RDOOR2         7
#define  LDOOR1         8
#define  LDOOR2         9
#define  DOORS1         10
#define  DOORS2         11

#define  NLDPS 12

/*-------------------------------------*/
/*  Define nodes for decision diagram  */
/*-------------------------------------*/
#define  HYPOTHESIS_NODE  0
#define  DECISION_NODE    1
#define  VALUE_NODE       2

/*-------------------------------------------*/
/*  Define detector nodes for belief network  */
/*-------------------------------------------*/
#define  CONCAVE45   3
#define  EDGE45      4
#define  CONVEX90    5
#define  WALL90      6
#define  EDGE90      7
#define  CONVEX270   8
#define  WALL270     9
#define  EDGE270     10
#define  CONVEX315   11
#define  CONCAVE315  12
#define  EDGE315     13
#define  WALL0       14
#define  CONVEX45    15

#define NDETECTORS 13
#define LASTNODE   15

/*-------------------------*/
/*  Define feature types  */
/*-------------------------*/
#define  CONVEX_CORNER   0
#define  CONCAVE_CORNER  1
#define  WALL            2
#define  EDGE            3
#define  PORT            4
#define  NOTHING         5

#define  NFEATURES 6

/*--------------------*/
/*  Define positions  */
/*--------------------*/
#define  ANGLE0          0
#define  ANGLE45         1
#define  ANGLE90         2
#define  ANGLE135        3
#define  ANGLE180        4
#define  ANGLE225        5
#define  ANGLE270        6
#define  ANGLE315        7

#define  NPOSNS  8

/*-------------------------*/
/*  Some useful constants  */
/*-------------------------*/
#define  BINARY          2
#define  TRUE            1
#define  FALSE           0
#define  YES             1
#define  NO              0
#define  NONZERO         4444

#define  FIRSTPORTNUM    1

/*-----------------------------------------------*/
```

```
/*  Establish possible distributions to infer for  */
/*-------------------------------------------------*/
#define MAINDIST    1
#define CONTROLPROB 2

/*-----------------------*/
/*  Feature found flags  */
/*-----------------------*/
#define FEATURE_NOT_FOUND  0
#define FEATURE_FOUND      1
#define DETECTOR_ABORTED   2

/*------------------------------*/
/*  Constants in LDP detection  */
/*------------------------------*/
#define DETECTOR_RETRIES   2


/*-------------------------------*/
/*  Define format of hypothesis  */
/*-------------------------------*/
struct hypothesis {
   int               LDP_type;                /*  Type of LDP the hypothesis is for  */
   int               index;                   /*  Sequential index of hypotheses     */
   float             certainty;               /*  Belief value of this hypothesis    */

   int               nports;                  /*  #paths (ports) leaving this LDP    */
   struct port       *portlist;               /*  List of these ports                */

   struct feature    *featurelist;            /*  List of features comprising this LDP  */
   struct hypothesis *next_hypothesis;        /*  Next hypothesis                    */
 };

#define HYPOSIZE  sizeof(struct hypothesis)


/*-----------------------------------*/
/*  Define format of port designator  */
/*-----------------------------------*/
struct port {
   FRM_POINT         pos;                     /*  Coordinates of this port           */
   int               theta;                   /*  Direction to face once at port     */
   int               move_strategy;           /*  Method to use in getting to port   */
   struct port       *nextport;               /*  Next port for this LDP             */
 };

#define PORTSIZE  sizeof(struct port)


/*-----------------------------------*/
/*  Define format of feature detector  */
/*-----------------------------------*/
struct feature {
   int       feature_type;                /*  Type of feature                          */
   int       detector_used;               /*  Flags if feature detector invoked yet    */
   int       retries_left;                /*  Number retries left for aborted detector */

   float     conceptual_angle;            /*  Compass direction to feature from ref point  */
   FRM_POINT pos;                         /*  Coords of point where detector begins    */
   FRM_POINT pos_error;                   /*  Allowed error in above coordinates       */
   int       theta;                       /*  Angle to face once at above coordinates  */
   int       side;                        /*  Some field needed by detector            */
   int       move_strategy;               /*  Method to use in getting to above coords */

   int       temp;                        /*  User field                               */

   float     complexity;                  /*  Complexity measure of feature detector   */
   float     latency;                     /*  Latency measure of feature detector      */

   struct hypothesis *assoc_hypothesis;   /*  Pts to parent hypothesis of this detector  */
   struct feature    *next_feature;       /*  Pts to next feature on "regular" list    */
   struct feature    *next_union;         /*  Pts to next feature on "union" list      */
 };

#define FEATSIZE  sizeof(struct feature)
```

```
#include "/pro/ai/robot/software/huey/utils/src/common.h"
#include "/u/rac/Masters/Ldp/src/ldp.h"
#include "/u/rac/Masters/Bride/src/bride.h"
#include <stdio.h>


/*-------------------------------------------------------------------------------------------*/
/*  Procedure MAIN                                                                           */
/*                                                                                           */
/*  This procedure is the main driver for the HUEY corridor-wandering robot.  This assumes   */
/*  upon startup that the robot is in some corridor section.  The control loop is as follows:*/
/*                                                                                           */
/*     1. Wander to the end of the corridor                                                  */
/*     2. Determine the type of LDP we have reached                                          */
/*     3. Select an outgoing port and traverse to that port                                  */
/*     4. Go to 1.                                                                           */
/*-------------------------------------------------------------------------------------------*/
main()
{
   struct hypothesis *masterhyps, *LDPhyps, *LDP;
   struct hypothesis *determine_location(), *Initialize_Hypotheses(), *Copy_Hypothesis_List();
   struct network *d, *LDPdiagram();
   int follow_corridor, Follow_Corridor();
   clientId myid, NSregisterSelf();

   /*------------------------*/
   /*  Set up for operation  */
   /*------------------------*/
   masterhyps = Initialize_Hypotheses();
   d = LDPdiagram();

   /*-------------------------------*/
   /*  Register to the nameserver   */
   /*-------------------------------*/
   if (ROBOT_RUN) myid = NSregisterSelf(LDP_NAME, LDP_IPC_PRIO);

   /*---------------------------------*/
   /*  Begin steady-state operation   */
   /*---------------------------------*/
   while (1) {

      /*------------------------*/
      /*  Follow the corridor   */
      /*------------------------*/
      follow_corridor = Follow_Corridor();

      if (follow_corridor == DETECTOR_ABORTED) {
         Lost_Robot();
         return; }

      /*------------------------*/
      /*  Prepare for new LDP   */
      /*------------------------*/
      Clear_WAI();
      LDPhyps = Copy_Hypothesis_List(masterhyps);

      /*----------------------*/
      /*  Determine the LDP   */
      /*----------------------*/
      LDP = determine_location(LDPhyps, d);

      if (LDP != NULL) printf("Robot is in location type %d with probability %g\n",
                              LDP->LDP_type, LDP->certainty);

      else {
         printf("Could not determine the robot's location\n");
         return; }

      /*-----------------------------------------*/
      /*  Go to a corridor leading out of the LDP  */
      /*-----------------------------------------*/
      Enter_New_Corridor(LDP);
      Free_Hypothesis_List(LDPhyps); }
 }


/*-------------------------------------------------------------------------------------------*/
/*  Procedure ENTER_NEW_CORRIDOR                                                             */
/*                                                                                           */
/*  This procedure moves the robot into one of the outgoing ports of the LDP.  This         */
/*  assumes that the robot is already at the reference position.                             */
/*-------------------------------------------------------------------------------------------*/
Enter_New_Corridor(hyp)
struct hypothesis *hyp;
{
   int i, random_index, Random_Port();
   struct port *portptr;
```

```
   /*------------------------------------------------------------------------*/
   /*  Get random port number and point to the corresponding port structure  */
   /*------------------------------------------------------------------------*/
   random_index = Random_Port(hyp->nports);
   printf("Port number %d selected\n", random_index);

   portptr = hyp->portlist;
   for (i = FIRSTPORTNUM; i < random_index; i++) portptr = portptr->nextport;
   printf("Port is at (%d %d) theta %d\n", portptr->pos.x, portptr->pos.y, portptr->theta);

   /*--------------------------*/
   /*  Go to the selected port  */
   /*--------------------------*/
   Move_To_Port(portptr);
 }


/*------------------------------------------------------------------------*/
/*  Function RANDOM_PORT                                                  */
/*                                                                        */
/*  This function chooses and returns a random port number in the range  */
/*  FIRSTPORTNUM <= port <= nports.  This port number is an integer.      */
/*------------------------------------------------------------------------*/
int Random_Port(nports)
int nports;
{
   float p;
   int retport, r, rand();

   r = rand();
   p = ((float) r) / 3E9;
   retport = FIRSTPORTNUM + (int) ((float) nports * p);

   /* printf("r = %d\tp = %g\tretport = %d\n", r, p, retport); */
   return(retport);
 }


/*------------------------------------------------------------------------------------*/
/*  Procedure LOST_ROBOT                                                              */
/*                                                                                    */
/*  This procedure is invoked when the robot becomes "lost" while following a corridor.  */
/*------------------------------------------------------------------------------------*/
Lost_Robot()
{
   printf("Help me!  Help me!  I am lost!\n");
 }
```

```
#include <stdio.h>
#include "/pro/ai/robot/software/huey/utils/src/common.h"
#include "/u/rac/Masters/Ldp/src/ldp.h"
#include "/u/rac/Masters/Bride/src/bride.h"


/*-------------------------------------------------------------------------------------------*/
/*  Function DETERMINE_LOCATION                                                              */
/*                                                                                           */
/*  This function takes a list of hypotheses about the current LDP and an influence diagram,  */
/*  and attempts to determine which LDP the robot is currently in.  The function returns that  */
/*  hypothesis corresponding to its best guess.                                              */
/*                                                                                           */
/*  The function has at its disposal a set of feature detectors, each associated with one of  */
/*  the given hypotheses.  At each cycle, this function selects one detector from the set to  */
/*  invoke by maximizing some decision utility function.  After the results of the detector   */
/*  are known, the function assimilates the results to determine a belief distribution over   */
/*  the hypotheses.                                                                          */
/*-------------------------------------------------------------------------------------------*/
struct hypothesis *determine_location(hypothesis_list, diagram)
struct hypothesis *hypothesis_list;
struct network *diagram;
{
   struct hypothesis   *hyptr,      *result,    *find_true_hypothesis(), *find_best_hypothesis();
   struct feature      *featptr,    *unionset,  *unionlast, *detector, *Remove_Duplicates();
   struct feature      *choose_next_detector_temp(), *Union_Detectors(), *choose_next_detector();
   struct selector     *sel,        *Select();
   struct node         *Hypothesis_Node();
   int                 H, true_hyps, false_hyps, detectors_left, reference;
   int                 count_detectors(),       Count_Hypotheses();
   float               prob;
   BOOLEAN             FeatureFound, detector_result, Invoke_Detector();
   struct node *detnode, *decnode, *valnode;
   struct node *Decision_Node(), *Value_Node(), *Detector_Node();

   /*-----------------------------------------------------*/
   /*  Initialize for starting into location recognition  */
   /*-----------------------------------------------------*/
   /*  Calculate initial probability for selected goal node hypotheses  */
   H = Count_Hypotheses(hypothesis_list, ANY);
   prob = 1.0 / H;

   /*  Initialize the goal node of the inference diagram with this probability  */
   Reset_Diagram(diagram);
   Init_Hypo_Distribution(hypothesis_list, diagram, prob);

   /*  Point to the decision and value nodes  */
   decnode = Decision_Node(diagram);
   valnode = Value_Node(diagram);

   /*  Create the union of all feature detectors in the hypothesis list  */
   unionset = Remove_Duplicates( Union_Detectors(hypothesis_list) );

   /*  Save the current point as a reference  */
   if (ROBOT_RUN) reference = Save_Point();

   /*-------------------------------------------------------------*/
   /*  Try various feature detectors until an LDP is confirmed  */
   /*-------------------------------------------------------------*/
   while (1)  {

      /*-------------------------------*/
      /*  Select a detector to invoke  */
      /*-------------------------------*/
      detector = choose_next_detector(unionset, diagram,
                                      Hypothesis_Node(diagram),
                                      hypothesis_list);

      /*--------------------------------------*/
      /*  Invoke the chosen feature detector  */
      /*--------------------------------------*/
      if (ROBOT_RUN) detector_result = Invoke_Detector_via_message(detector);
      else detector_result = Invoke_Detector(detector);

      /*-------------------------------------------*/
      /*  Return the robot to the initial position  */
      /*-------------------------------------------*/
      if (ROBOT_RUN) Move_To_Point(reference, detector->move_strategy);

      /*-----------------------------------------------------------------*/
      /*  Allow aborted detector to be invoked again (up to retry amount)  */
      /*-----------------------------------------------------------------*/
      if (detector_result == DETECTOR_ABORTED) Mark_Aborted_Detector(detector);

      else {
         /*-------------------------------------------------------*/
         /*  Process the return from the detector to get a probability  */
```

```
        /*  distribution of LDP hypotheses.                          */
        /*-----------------------------------------------------------*/
        Assimilate_Detector(detector, detector_result, diagram, hypothesis_list);

        /*-----------------------------------------------------------*/
        /*  Make dynamic arc between detector node and decision node  */
        /*  as preliminary step for next decision.                   */
        /*-----------------------------------------------------------*/
        detnode = Detector_Node(diagram, detector);
        Add_Dynamic_Arc(detnode, decnode); }

      /*---------------------------------------*/
      /*  Take stock of hypotheses and detectors  */
      /*---------------------------------------*/
      true_hyps  = Count_Hypotheses(hypothesis_list, TRUE);
      false_hyps = Count_Hypotheses(hypothesis_list, FALSE);
      detectors_left = count_detectors(unionset);

      /*-------------------------------------------------------------*/
      /*  When there is exactly 1 "true" hypothesis and the remaining  */
      /*  hypotheses are "false," stop and return the true one.      */
      /*-------------------------------------------------------------*/
      if ((true_hyps == 1  &&  false_hyps == H - 1)  ||
          (true_hyps == 1  &&  detectors_left == 0))  {
         result = find_true_hypothesis(hypothesis_list);

         return(result); }

      /*-------------------------------------------------------------------*/
      /*  When all detectors have been tried, and no hypothesis is "true,"  */
      /*  return the highest-certainty hypothesis as a best guess.         */
      /*-------------------------------------------------------------------*/
      if (true_hyps == 0  &&  detectors_left == 0)  {
         result = find_best_hypothesis(hypothesis_list);

         return(result); } }
}


/*-------------------------------------------------------------------------------------------*/
/*  Procedure ASSIMILATE_DETECTOR                                                            */
/*                                                                                           */
/*  This procedure takes the results of the given feature detector and infers new certainties  */
/*  for the list of hypotheses.  This is done using probabilistic inference on the given     */
/*  influence diagram.                                                                       */
/*                                                                                           */
/*  The newly-inferred hypotheses certainties are placed in the given hypothesis list.       */
/*-------------------------------------------------------------------------------------------*/
Assimilate_Detector(detector, FeatureFound, infer_diagram, hypothesis_list)
struct feature *detector;
BOOLEAN FeatureFound;
struct network *infer_diagram;
struct hypothesis *hypothesis_list;
{
   struct network *infer_copy, *feature_diagram, *LDP_diagram, *Copy_Diagram();
   struct distribution *Result_Distribution();
   struct node *corresp_node, *Detector_Node();

   /*-----------------------------------------------------------------*/
   /*  Set retain flag and knownvalue in the proper node of infer_diagram  */
   /*  corresponding to the given detector.                           */
   /*-----------------------------------------------------------------*/
   corresp_node = Detector_Node(infer_diagram, detector);
   corresp_node->retain = 1;

   if (FeatureFound) corresp_node->knownstate = TRUE;
               else corresp_node->knownstate = FALSE;

   /*-----------------------------------------------------------------*/
   /*  Determine new hypothesis certainties using probabilistic inference  */
   /*-----------------------------------------------------------------*/
   infer_copy = Copy_Diagram(infer_diagram);
   Set_Diagram(infer_copy, MAINDIST);

   infer(infer_copy);

   /*---------------------------------------*/
   /*  Assign new beliefs to the hypotheses  */
   /*---------------------------------------*/
   Result_Distribution(infer_copy->goalnode, hypothesis_list, detector);

   /*-----------------------------*/
   /*  Free up the original copy  */
   /*-----------------------------*/
   Free_Diagram(infer_copy);
}
```

```
/*-------------------------------------------------------------------------------------*/
/*  Function RESULT_DISTRIBUTION                                                       */
/*                                                                                     */
/*  This function calculates the final belief values for the various hypotheses from the  */
/*  inferred diagram.  The beliefs are obtained by fixing whether each detector found  */
/*  its feature, and reading out the array of values for the varying hypotheses from the  */
/*  goal node distribution.                                                            */
/*                                                                                     */
/*  The beliefs are set in the corresponding hypothesis structures as a side effect.   */
/*  A new distribution is also created of these beliefs for installation in the goalnode.  */
/*  However, this new distribution is currently unused by procedure Assimilate_Detector.  */
/*-------------------------------------------------------------------------------------*/
struct distribution *Result_Distribution(goalnode, hypos, detector)
struct node *goalnode;
struct hypothesis *hypos;
struct feature *detector;
{
   struct distribution *newdist, *New_Distribution();
   struct hypothesis *hyptr;
   struct selector *sel1, *newsel, *nodepreds, *nodesel;
   struct selector *Select_Predecessors(), *NewSel(), *Appended_Copy(), *Build_Access_Selector();
   float new_certainty, prob1, Get_Prob();

   /*------------------*/
   /*  Error checking  */
   /*------------------*/
   if (goalnode == NULL) {
      printf("ERROR: Result_Distribution passed null goal node\n"); return(NULL); }

   if (hypos == NULL) {
      printf("ERROR: Result_Distribution passed null hypothesis list\n"); return(NULL); }

   if (detector == NULL) {
      printf("ERROR: Result_Distribution passed null detector\n"); return(NULL); }


   /*-------------------------------------------------------------------------------*/
   /*  Build a selector to access the relevant values from the goalnode distribution  */
   /*-------------------------------------------------------------------------------*/
   nodepreds = Build_Access_Selector(goalnode);
   nodesel   = NewSel(goalnode, 0);
   sel1      = Appended_Copy(nodesel, nodepreds);

   Free_Selector(nodesel);
   Free_Selector(nodepreds);

   /*-------------------------------------------------------------*/
   /*  Loop thru all hypotheses, calculating a belief for each  */
   /*-------------------------------------------------------------*/
   newdist = New_Distribution();
   hyptr = hypos;

   while (hyptr != NULL) {

      /*-------------------------------------------------------*/
      /*  Get the probability from the goalnode distribution  */
      /*-------------------------------------------------------*/
      sel1->state = hyptr->LDP_type;
      prob1 = new_certainty = Get_Prob(goalnode->header, sel1);

      /*-------------------------------------------------------*/
      /*  Set the corresponding value in the new distribution  */
      /*  and in the hypothesis itself.                        */
      /*-------------------------------------------------------*/
      newsel = Select(hyptr->LDP_type, -1);
      Set_Prob(newdist, newsel, new_certainty);

      hyptr->certainty = new_certainty;
      printf("Hypothesis %d certainty is %g\n", hyptr->LDP_type, new_certainty);

      /*-----------------------*/
      /*  Get next hypothesis  */
      /*-----------------------*/
      hyptr = hyptr->next_hypothesis; }


   /*---------------------------------*/
   /*  Free the remaining selectors  */
   /*---------------------------------*/
   Free_Selector(sel1);

   return(newdist);
}



/*-------------------------------------------------------------------------------------*/
/*  Function INIT_HYPO_DISTRIBUTION                                                    */
```

```
/*                                                                                    */
/*  This procedure initialized the goal node distribution by setting one distribution */
/*  element for each hypothesis in the given list.  Each element is set to the given  */
/*  probability.                                                                      */
/*------------------------------------------------------------------------------------*/
Init_Hypo_Distribution(hypos, diag, prob)
struct hypothesis *hypos;
struct network *diag;
float prob;
{
   struct hypothesis *hyptr;
   struct node *hypnode, *Hypothesis_Node();
   struct selector *sel, *Select();

   /*------------------*/
   /*  Error checking  */
   /*------------------*/
   if (hypos == NULL) { printf("ERROR: Init_Hypo_Distribution passed null hypo list\n"); return; }
   if (diag == NULL ) { printf("ERROR: Init_Hypo_Distribution passed null diagram\n");   return; }

   hypnode = Hypothesis_Node(diag);

   /*-------------------------------------------------*/
   /*  Give the hypothesis node a blank distribution  */
   /*-------------------------------------------------*/
   Free_Distribution(hypnode);
   hypnode->header = New_Distribution();

   /*-----------------------------------------------------------------*/
   /*  Loop thru all hypotheses in the list, setting the corresponding  */
   /*  value in the hypothesis node distribution to prob.             */
   /*-----------------------------------------------------------------*/
   hyptr = hypos;

   while (hyptr != NULL) {
      printf("Hypothesis %d initial belief = %g\n", hyptr->LDP_type, prob);
      sel = Select(hyptr->LDP_type, -1);
      Set_Prob(hypnode->header, sel, prob);

      Free_Selector(sel);

      hyptr->certainty = prob;
      hyptr = hyptr->next_hypothesis; }
 }


/*------------------------------------------------------------------------------------*/
/*  Function UNION_DETECTORS                                                          */
/*                                                                                    */
/*  This function builds a list of all feature detectors in the given hypothesis list.  */
/*  It also saves a backpointer to the hypothesis in each feature detector.           */
/*                                                                                    */
/*  This IGNORES any overlap in detectors!                                            */
/*------------------------------------------------------------------------------------*/
struct feature *Union_Detectors(hypos)
struct hypothesis *hypos;
{
   struct hypothesis *hyptr;
   struct feature *unionset, *unionlast, *featptr;

   /*------------------*/
   /*  Error checking  */
   /*------------------*/
   if (hypos == NULL) {
      printf("ERROR: Union_Detectors passed null hypothesis list\n");  return(NULL); }

   /*-----------------------------*/
   /*  Loop thru all hypotheses  */
   /*-----------------------------*/
   hyptr = hypos;
   unionset = NULL;

   while (hyptr != NULL) {

      /*-------------------------------------------*/
      /*  Prepare for loop thru feature detectors  */
      /*-------------------------------------------*/
      featptr = hyptr->featurelist;
      if (unionset == NULL)  unionset = unionlast = featptr;

      /*-----------------------------------------------------*/
      /*  Loop thru all feature detectors in this hypothesis  */
      /*-----------------------------------------------------*/
      while (featptr != NULL)  {

         /*-----------------------------------*/
         /*  Save back-pointer to hypothesis  */
         /*-----------------------------------*/
```

```
        featptr->assoc_hypothesis = hyptr;

        /*-----------------------------------------*/
        /*  Add this feature detector to union list  */
        /*-----------------------------------------*/
        unionlast->next_union = featptr;
        featptr->next_union = NULL;
        unionlast = featptr;

        featptr = featptr->next_feature; }

      /*-----------------------*/
      /*  Get next hypothesis  */
      /*-----------------------*/
      hyptr = hyptr->next_hypothesis; }

  return(unionset);
}


/*-------------------------------------------------------------------------------------------*/
/*  Function COUNT_HYPOTHESES                                                                */
/*                                                                                           */
/*  This function counts the number of hypotheses in the given list are of the specified   */
/*  category.  Categories are: true hypotheses, false hypotheses, all hypotheses.           */
/*-------------------------------------------------------------------------------------------*/
int Count_Hypotheses(hypos, which)
struct hypothesis *hypos;
int which;
{
   struct hypothesis *hyptr;
   static float true_threshold  = .8;
   static float false_threshold = .5;
   int tally = 0;

   /*------------------*/
   /*  Error checking  */
   /*------------------*/
   if (which != TRUE  &&  which != FALSE  &&  which != ANY  &&  which != NONZERO) {
      printf("ERROR: Count_Hypotheses passed invalid second argument\n");  return(-1); }

   if (hypos == NULL) {printf("ERROR: Count_Hypotheses passed null hypothesis list\n"); return(-1);}

   /*-----------------------------------------------------------*/
   /*  Loop thru all hypotheses, tallying the appropriate ones  */
   /*-----------------------------------------------------------*/
   hyptr = hypos;
   while (hyptr != NULL)  {
      if (which == ANY    ||
         (which == NONZERO   &&  hyptr->certainty != 0.0)  ||
         (which == TRUE      &&  hyptr->certainty >= true_threshold)  ||
         (which == FALSE     &&  hyptr->certainty <= false_threshold))
          tally++;

      hyptr = hyptr->next_hypothesis;
      );

return(tally);
}


/*-------------------------------------------------------------------------------------------*/
/*  Function INVOKE_DETECTOR                                                                 */
/*                                                                                           */
/*  This is a dummy function used in testing the LDP-recognition algorithm without          */
/*  needing the actual robot. The value in the temp field of the detector is                 */
/*  returned as the result of the function.                                                  */
/*-------------------------------------------------------------------------------------------*/
BOOLEAN Invoke_Detector(d)
struct feature *d;
{
   /*-----------------------------------------*/
   /*  Invoke the appropriate detector routine  */
   /*-----------------------------------------*/
   switch (d->feature_type) {
   case CONVEX_CORNER:
      printf("In CONVEX CORNER detector\n");
      break;

   case CONCAVE_CORNER:
      printf("In CONCAVE CORNER detector\n");
      break;

   case WALL:
      printf("In WALL detector\n");
      break;

   case EDGE:
```

```
      printf("In EDGE detector\n");
      break; }

   /*------------------------------*/
   /*  Mark this detector as used  */
   /*------------------------------*/
   d->detector_used = 1;

   printf("Detector %d @ %g returned %d\n",
          d->feature_type, d->conceptual_angle, d->temp);

   return(d->temp);
   /* return(DETECTOR_ABORTED); */
 }


/*-------------------------------------------------------------------------------------------*/
/*  Function INVOKE_DETECTOR_VIA_MESSAGE                                                     */
/*                                                                                           */
/*  This function invokes an actual feature detector for execution by the robot.  It sends */
/*  a message to the Feature Recognition Module to do this.  The result that comes back    */
/*  from the FRM is returned as the result of this function.                                 */
/*-------------------------------------------------------------------------------------------*/
BOOLEAN Invoke_Detector_via_message(d)
struct feature *d;
{
   BOOLEAN result;
   int FRMtype, Map_Feature();
   FRM_REQUEST sendmsg;
   FRM_RESPONSE *response, *FRM_Operation();


   /*-------------------------------------------------------------*/
   /*  Initialize the message to the feature recognition module  */
   /*-------------------------------------------------------------*/
   printf("Asking FRM if there is a feature %d at angle %g and point (%d %d) strategy %d\n",
          d->feature_type, d->conceptual_angle, d->pos.x, d->pos.y, d->move_strategy);

   FRMtype             = Map_Feature(d->feature_type);
   sendmsg.type        = FRMtype;
   sendmsg.pos.x       = d->pos.x;
   sendmsg.pos.y       = d->pos.y;
   sendmsg.pos_error.x = d->pos_error.x;
   sendmsg.pos_error.y = d->pos_error.y;
   sendmsg.theta       = d->theta;
   sendmsg.strategy    = d->move_strategy;
   sendmsg.side        = d->side;

   /*---------------------------------*/
   /*  Get the response from the FRM  */
   /*---------------------------------*/
   response = FRM_Operation(sendmsg);

   /*-----------------------------------------*/
   /*  Map received results to internal states  */
   /*-----------------------------------------*/
   if      (response->result == FRM_FEATURE_FOUND)     result = FEATURE_FOUND;
   else if (response->result == FRM_FEATURE_NOT_FOUND) result = FEATURE_NOT_FOUND;
   else if (response->result == FRM_ABORT)             result = DETECTOR_ABORTED;

   else printf("ERROR: Invoke_Detector received invalid response from FRM: %d\n",
               response->result);

   /*------------------------------*/
   /*  Mark this detector as used  */
   /*------------------------------*/
   d->detector_used = 1;

   printf("Detector %d for (%d %d) returned %d\n",
          sendmsg.type, sendmsg.pos.x, sendmsg.pos.y, result);

   /*---------------------------------*/
   /*  Free the response and return  */
   /*---------------------------------*/
   free(response);
   return(result);
 }


/*-------------------------------------------------------------------------------------------*/
/*  Function CHOOSE_NEXT_DETECTOR_OLD                                                        */
/*                                                                                           */
/*  This function makes a very simple choice of the next detector to invoke; in particular */
/*  it chooses the next uninvoked detector.  This is primarily used in testing.              */
/*-------------------------------------------------------------------------------------------*/
struct feature *choose_next_detector_old(featureset, decision_diagram, hyp_node, hyplist)
struct feature *featureset;
struct network *decision_diagram;
```

```
struct node *hyp_node;
struct hypothesis *hyplist;
{
   struct feature *featptr;

   featptr = featureset;
   while (featptr != NULL)  {
      if (!featptr->detector_used) return(featptr);
      featptr = featptr->next_union;
      };

return(NULL);
}


/*-------------------------------------------------------------------------------------------*/
/*  Function CHOOSE_NEXT_DETECTOR                                                            */
/*                                                                                           */
/*  This function chooses the next feature detector to invoke from the given list of detectors.  */
/*  This is done using decision analysis on the given decision influence diagram.  The diagram   */
/*  is evaluated to get a decision policy for choosing detectors, then the policy is applied     */
/*  for the most likely hypothesis.                                                          */
/*-------------------------------------------------------------------------------------------*/
struct feature *choose_next_detector(detector_list, decision_diagram, hyp_node, hyplist)
struct feature *detector_list;
struct network *decision_diagram;
struct node *hyp_node;
struct hypothesis *hyplist;
{
   struct network *decision_copy, *Copy_Diagram();
   struct distribution *Copy_Distribution(), *Utility();
   int choice, best_hyp, count_detectors(), Count_Hypotheses();
   float Get_Prob();
   struct hypothesis *find_best_hypothesis();
   struct selector *hyp_selector, *Select(), *Find_Best_Utility();
   struct feature *best_detector, *Nth();
   struct node *dhyp_node, *Hypothesis_Node(), *Decision_Node();

   /*------------------*/
   /*  Error checking  */
   /*------------------*/
   if (detector_list == NULL)    printf("ERROR: choose_next_detector passed null detector list\n");
   if (decision_diagram == NULL) printf("ERROR: choose_next_detector passed null decision diagram\n");
   if (hyp_node == NULL)         printf("ERROR: choose_next_detector passed null hypothesis node\n");
   if (hyplist == NULL)          printf("ERROR: choose_next_detector passed null hypothesis list\n");

   /*----------------------------------------*/
   /*  Set up work copy of decision diagram  */
   /*----------------------------------------*/
   decision_copy = Copy_Diagram(decision_diagram);
   Set_Diagram(decision_copy, CONTROLPROB);

   /*-------------------------------------------------------------------------------*/
   /*  Set number of states in decision node = number of detectors in detector list  */
   /*  and number of states in hypothesis node = number of hypotheses in hyplist.    */
   /*-------------------------------------------------------------------------------*/
   Decision_Node(decision_copy)->nstates   = count_detectors(detector_list);
   Hypothesis_Node(decision_copy)->nstates = Count_Hypotheses(hyplist, ANY);

   /*----------------------------------------*/
   /*  Build utility function in value node  */
   /*----------------------------------------*/
   Free_Distribution(decision_copy->goalnode);
   decision_copy->goalnode->header = Utility(detector_list, hyplist, decision_diagram);

   /*-----------------------------*/
   /*  Perform decision analysis  */
   /*-----------------------------*/
   decide(decision_copy);

   /*-------------------------------------------------------------------------------*/
   /*  Get the optimal detector to invoke from first (only?) node in d* and return it  */
   /*-------------------------------------------------------------------------------*/
   hyp_selector = Find_Best_Utility(decision_copy->dstar);
   choice = (int) Get_Prob(decision_copy->dstar->header, hyp_selector);

   best_detector = Nth(detector_list, choice);
   printf("Decision policy dictates detector %d @ %g\n",
          best_detector->feature_type, best_detector->conceptual_angle);

   Free_Selector(hyp_selector);

   /*-------------------------*/
   /*  Free the work diagram  */
   /*-------------------------*/
   Free_Diagram(decision_copy);
   return(best_detector);
}
```

```
/*-------------------------------------------------------------------------------------*/
/*  Function MAP_FEATURE                                                               */
/*                                                                                     */
/*  This function maps the internal LDP types into the corresponding types for the FRM.*/
/*-------------------------------------------------------------------------------------*/
int Map_Feature(LOCtype)
int LOCtype;
{
   /*-----------------------------------------------------*/
   /*  Map LOC internal types into message types for FRM  */
   /*-----------------------------------------------------*/
   if (LOCtype == CONVEX_CORNER)   return(FRM_CONVEX_CORNER);
   if (LOCtype == CONCAVE_CORNER)  return(FRM_CONCAVE_CORNER);
   if (LOCtype == WALL)            return(FRM_WALL);
   if (LOCtype == EDGE)            return(FRM_EDGE);
   if (LOCtype == PORT)            return(FRM_PORT);

   printf("ERROR: Cannot map type %d into a message type for FRM\n", LOCtype);
   return(0);
 }


/*************************************************************************************/
/***  DETECTOR OPERATIONS                                                          ***/
/***                                                                               ***/
/***  This set of functions performs various operations on feature detectors       ***/
/***  and lists of them.                                                           ***/
/*************************************************************************************/

/*-------------------------------------------------------------------------------------*/
/*  Procedure MARK_ABORTED_DETECTOR                                                    */
/*                                                                                     */
/*  This procedure processes a feature detector that has aborted.  It first decrements*/
/*  the number of retries allowed for the detector, and if there are retries left, it  */
/*  resets the "used" flag so that the detector can be re-invoked.                     */
/*-------------------------------------------------------------------------------------*/
Mark_Aborted_Detector(detector)
struct feature *detector;
{
   /*---------------------------------------*/
   /*  First decrement the number of retries  */
   /*  remaining for this detector.           */
   /*---------------------------------------*/
   detector->retries_left--;

   /*---------------------------------------*/
   /*  If retries are left, reset the detector  */
   /*  so it may be re-invoked.                 */
   /*---------------------------------------*/
   if (detector->retries_left > 0) detector->detector_used = 0;

   printf("Detector %d @ %g aborted. Retries = %d. Detector_used = %d\n",
          detector->feature_type, detector->conceptual_angle,
          detector->retries_left, detector->detector_used);
 }


/*-------------------------------------------------------------------------------*/
/*  Function DETECTOR_NODE                                                       */
/*                                                                               */
/*  This function returns a pointer to the detector node in the given diagram  */
/*  corresponding to the given detector.                                         */
/*-------------------------------------------------------------------------------*/
struct node *Detector_Node(diagram, detector)
struct network *diagram;
struct feature *detector;
{
   int nodeindex, Map_Detector();

   if (diagram == NULL)  printf("ERROR: Detector_Node passed null diagram\n");
   if (detector == NULL) printf("ERROR: Detector_Node passed null detector\n");

   nodeindex = Map_Detector(detector->feature_type, detector->conceptual_angle);
   /* printf("Detector %d @ %g is index %d\n",
          detector->feature_type, detector->conceptual_angle, nodeindex); */

   return( &(diagram->nodelist[nodeindex]) );
 }


/*-------------------------------------------------------------------------------------*/
/*  Function COUNT_DETECTORS                                                           */
/*                                                                                     */
/*  This function counts and returns the number of feature detectors that are still unused  */
/*-------------------------------------------------------------------------------------*/
int count_detectors(featureset)
```

```
struct feature *featureset;
{
   struct feature *featptr;
   int tally = 0;

   /*------------------*/
   /*  Error checking  */
   /*------------------*/
   if (featureset == NULL) {
      printf("ERROR: count_detectors passed null featureset\n"); return(0); }

   /*------------------------------*/
   /*  Count the unused detectors  */
   /*------------------------------*/
   featptr = featureset;
   while (featptr != NULL)  {
      if (!featptr->detector_used) tally++;

      featptr = featptr->next_union; }

return(tally);
}


/*-------------------------------------------------------------------------------------------*/
/*  Function NTH                                                                             */
/*                                                                                           */
/*  This function searches the given detector list for the nth item and returns that item.   */
/*  Specify n = 0 for the first item in the list, n = 1 for the second, etc.                 */
/*-------------------------------------------------------------------------------------------*/
struct feature *Nth(detectlist, n)
struct feature *detectlist;
int n;
{
   struct feature *d;
   int i, end;

   /*------------------*/
   /*  Error checking  */
   /*------------------*/
   if (detectlist == NULL) printf("ERROR: Nth passed null detector list\n");
   if (n < 0) printf("ERROR: Nth passed negative integer\n");

   /*--------------------*/
   /*  Set up for loop   */
   /*--------------------*/
   i = 0;
   end = n + 1;
   d = detectlist;

   /*-----------------------------------------------------------*/
   /*  Bump thru detectlist until n unused detectors are found  */
   /*-----------------------------------------------------------*/
   while (1)  {
      if (!d->detector_used) {
         i++;
         if (i == end) return(d); }

      d = d->next_union;
      if (d == NULL) printf("ERROR: Nth ran off end of list searching for unused item %d\n", n); }
 }


/*-------------------------------------------------------------------------------------------*/
/*  Function REMOVE_DUPLICATES                                                               */
/*                                                                                           */
/*  This function takes a union list of feature detectors, and returns a union list with all */
/*  duplicate feature detectors marked as used.                                              */
/*-------------------------------------------------------------------------------------------*/
struct feature *Remove_Duplicates(detector_union)
struct feature *detector_union;
{
   struct feature *d;

   /*------------------*/
   /*  Error checking  */
   /*------------------*/
   if (detector_union == NULL) printf("ERROR: Remove_Duplicates passed null detector list\n");

   /*-----------------------------------------*/
   /*  Loop thru all detectors in detector list  */
   /*-----------------------------------------*/
   d = detector_union;
   while (d != NULL)  {

      /*-----------------------------------------------------------------*/
      /*  Mark as used any detector that duplicates the current detector  */
      /*-----------------------------------------------------------------*/
```

```
      if (!d->detector_used) Mark_Duplicates(d->next_union, d);
      d = d->next_union; }

   return(detector_union);
 }


/*-------------------------------------------------------------------------------------------*/
/*  Function MARK_DUPLICATES                                                                 */
/*                                                                                           */
/*  This procedure searches the given detector list for detectors that duplicate the given   */
/*  chosen detector.  Duplicate detectors are those that search for the same feature type at */
/*  the same angular region.  Any duplicate detectors are marked as used to keep them from   */
/*  being invoked later.                                                                     */
/*-------------------------------------------------------------------------------------------*/
Mark_Duplicates(detector_list, chosen_detector)
struct feature *detector_list, *chosen_detector;
{
   struct feature *d;
   int Map_Posn();

   /*------------------*/
   /*  Error checking  */
   /*------------------*/
   if (chosen_detector == NULL) printf("ERROR: Mark_Duplicates passed null detector\n");

   /*-----------------------------------------*/
   /*  Loop thru all detectors in detector list  */
   /*-----------------------------------------*/
   d = detector_list;
   while (d != NULL)  {

      /*-----------------------------------------------------------------*/
      /*  Mark as used any detector that duplicates the chosen detector  */
      /*-----------------------------------------------------------------*/
      if (!d->detector_used  &&
          d->feature_type == chosen_detector->feature_type  &&
          Map_Posn(d->conceptual_angle) == Map_Posn(chosen_detector->conceptual_angle)) {

          /*printf("Marking duplicate detector %d @ %g\n", d->feature_type, d->conceptual_angle);*/
          d->detector_used = 9; }

      d = d->next_union; }
 }


/*****************************************************************************/
/***  HYPOTHESIS OPERATIONS                                                ***/
/***                                                                       ***/
/***  This set of functions performs various operations on hypotheses and  ***/
/***  their various hangers-on (ports, features).                          ***/
/*****************************************************************************/

/*-------------------------------------------------------------------------------------------*/
/*  Function FIND_TRUE_HYPOTHESIS                                                            */
/*                                                                                           */
/*  This function searches the given hypothesis list for a hypothesis with a certainty      */
/*  greater than the "true threshold."  It returns this hypothesis.                         */
/*-------------------------------------------------------------------------------------------*/
struct hypothesis *find_true_hypothesis(hyplist)
struct hypothesis *hyplist;
{
   struct hypothesis *hyptr;
   static float true_threshold  = .8;
   static float false_threshold = .5;

   /*------------------*/
   /*  Error checking  */
   /*------------------*/
   if (hyplist == NULL) {
      printf("ERROR: find_true_hypothesis passed null hypothesis list\n"); return(NULL); }

   /*-----------------------------------------------------------------------------*/
   /*  Loop thru all hypotheses until we find the first one with a "true" certainty  */
   /*-----------------------------------------------------------------------------*/
   hyptr = hyplist;
   while (hyptr != NULL)  {
      if (hyptr->certainty >= true_threshold) return(hyptr);
      hyptr = hyptr->next_hypothesis;
      };

return(NULL);
}


/*-------------------------------------------------------------------------------------------*/
/*  Function FIND_BEST_HYPOTHESIS                                                            */
/*                                                                                           */
```

```
/*  This function searches the given hypothesis list for the one with the highest certainty.  */
/*  It returns this hypothesis.                                                                */
/*-------------------------------------------------------------------------------------------*/
struct hypothesis *find_best_hypothesis(hyplist)
struct hypothesis *hyplist;
{
   struct hypothesis *hyptr, *bestptr;
   float highest = 0.0;

   hyptr = hyplist;
   bestptr = NULL;

   /*------------------*/
   /*  Error checking  */
   /*------------------*/
   if (hyplist == NULL) {
      printf("ERROR: find_best_hypothesis passed null hypothesis list\n"); return(NULL); }

   /*-----------------------------------------------------------------*/
   /*  Loop thru all hypotheses, saving the one with highest certainty  */
   /*-----------------------------------------------------------------*/
   while (hyptr != NULL)  {
      if (hyptr->certainty >= highest) {
         highest = hyptr->certainty;
         bestptr = hyptr; }

      hyptr = hyptr->next_hypothesis;
      };

return(bestptr);
}


/*-------------------------------------------------------------------------------------------*/
/*  Function NEW_HYPOTHESIS                                                                  */
/*                                                                                           */
/*  This function creates and returns a new hypothesis structure.                           */
/*-------------------------------------------------------------------------------------------*/
struct hypothesis *New_Hypothesis(type, ind)
int type, ind;
{
   struct hypothesis *h;
   char *malloc();

   h = (struct hypothesis *) malloc(HYPOSIZE);
   h->LDP_type        = type;
   h->index           = ind;
   h->certainty       = 0.0;
   h->nports          = 0;
   h->portlist        = NULL;
   h->featurelist     = NULL;
   h->next_hypothesis = NULL;

   return(h);
 }


/*-------------------------------------------------------------------------------------------*/
/*  Function ATTACH_PORT                                                                     */
/*                                                                                           */
/*  This procedure gets a new port structure, and adds it to the head of the portlist of     */
/*  the given hypothesis. Also bumps the nports counter in the hypothesis.                   */
/*-------------------------------------------------------------------------------------------*/
Attach_Port(hypo, x, y, theta, strategy)
struct hypothesis *hypo;
int x, y, theta, strategy;
{
   struct port *p, *Newport();

   p = Newport(x, y, theta, strategy);

   p->nextport = hypo->portlist;
   hypo->portlist = p;

   hypo->nports++;
 }


/*-------------------------------------------------------------------------------------------*/
/*  Function NEWPORT                                                                         */
/*                                                                                           */
/*  This function creates and returns a new port structure.                                  */
/*-------------------------------------------------------------------------------------------*/
struct port *Newport(x, y, theta, strategy)
int x, y, theta, strategy;
{
   struct port *p;
   char *malloc();
```

```
   p = (struct port *) malloc(PORTSIZE);
   p->pos.x         = x;
   p->pos.y         = y;
   p->theta         = theta;
   p->move_strategy = strategy;
   p->nextport      = NULL;

   return(p);
 }


/*-------------------------------------------------------------------------------------------*/
/*  Function ATTACH_FEATURE                                                                  */
/*                                                                                           */
/*  This procedure gets a new feature structure, and adds it to the head of the featurelist  */
/*  of the given hypothesis.                                                                 */
/*-------------------------------------------------------------------------------------------*/
Attach_Feature(hypo, type, angle, x, y, xerror, yerror, theta, side, strategy)
struct hypothesis *hypo;
int type;
float angle;
int x, y, xerror, yerror, theta, side, strategy;
{
   struct feature *f, *New_Feature();

   f = New_Feature(hypo, type, angle, x, y, xerror, yerror, theta, side, strategy);

   f->next_feature = hypo->featurelist;
   hypo->featurelist = f;
 }


/*-------------------------------------------------------------------------------------------*/
/*  Function NEW_FEATURE                                                                     */
/*                                                                                           */
/*  This function creates and returns a new feature structure.                               */
/*-------------------------------------------------------------------------------------------*/
struct feature *New_Feature(hypo, type, angle, x, y, xerror, yerror, theta, side, strategy)
struct hypothesis *hypo;
int type;
float angle;
int x, y, xerror, yerror, theta, side, strategy;
{
   float Complexity(), Latency();
   struct feature *f;
   char *malloc();

   f = (struct feature *) malloc(FEATSIZE);
   f->feature_type     = type;
   f->detector_used    = 0;
   f->retries_left     = DETECTOR_RETRIES;
   f->conceptual_angle = angle;
   f->pos.x            = x;
   f->pos.y            = y;
   f->pos_error.x      = xerror;
   f->pos_error.y      = yerror;
   f->theta            = theta;
   f->side             = side;
   f->move_strategy    = strategy;

   f->temp             = 0;
   f->complexity       = Complexity(type);
   f->latency          = Latency(f);            /* This must follow asgn of pos.x & pos.y */

   f->assoc_hypothesis = hypo;
   f->next_feature     = NULL;
   f->next_union       = NULL;

   return(f);
 }


/*-------------------------------------------------------------------------------------------*/
/*  Function COPY_HYPOTHESIS_LIST                                                            */
/*                                                                                           */
/*  This function makes a duplicate copy of the given linked list of hypotheses.            */
/*-------------------------------------------------------------------------------------------*/
struct hypothesis *Copy_Hypothesis_List(oldhyplist)
struct hypothesis *oldhyplist;
{
   struct hypothesis *rest, *newhyp, *Copy_Hypothesis_List(), *Copy_Hypothesis();

   /*----------------*/
   /*  Trivial case  */
   /*----------------*/
   if (oldhyplist == NULL) return(NULL);
```

```
   /*--------------------------*/
   /*  Copy CDR of oldhyplist  */
   /*--------------------------*/
   rest = Copy_Hypothesis_List(oldhyplist->next_hypothesis);

   /*-----------------------------------------*/
   /*  Get and init a new hypothesis element  */
   /*-----------------------------------------*/
   newhyp = Copy_Hypothesis(oldhyplist);
   newhyp->next_hypothesis = rest;

   return(newhyp);
 }


/*-------------------------------------------------------------------------------*/
/*  Function COPY_HYPOTHESIS                                                     */
/*                                                                               */
/*  This function makes a duplicate copy of the given hypothesis structure.  */
/*-------------------------------------------------------------------------------*/
struct hypothesis *Copy_Hypothesis(oldhyp)
struct hypothesis *oldhyp;
{
   struct hypothesis *newhyp, *New_Hypothesis();
   struct port *Copy_Portlist();
   struct feature *Copy_Featurelist();

   /*----------------*/
   /*  Trivial case  */
   /*----------------*/
   if (oldhyp == NULL) return(NULL);

   /*-------------------------------------------*/
   /*  Get and init a new hypothesis structure  */
   /*-------------------------------------------*/
   /* printf("Copying hypothesis %d index %d\n", oldhyp->LDP_type, oldhyp->index); */
   newhyp = New_Hypothesis(oldhyp->LDP_type, oldhyp->index);

   newhyp->certainty   = oldhyp->certainty;
   newhyp->nports      = oldhyp->nports;

   newhyp->portlist    = Copy_Portlist(oldhyp->portlist);
   newhyp->featurelist = Copy_Featurelist(oldhyp->featurelist, newhyp);

   return(newhyp);
 }


/*-----------------------------------------------------------------------------------*/
/*  Function COPY_PORTLIST                                                           */
/*                                                                                   */
/*  This function makes and returns a duplicate copy of the given linked list of ports.  */
/*-----------------------------------------------------------------------------------*/
struct port *Copy_Portlist(portlist)
struct port *portlist;
{
   struct port *newport, *rest, *Newport(), *Copy_Portlist();

   /*----------------*/
   /*  Trivial case  */
   /*----------------*/
   if (portlist == NULL) return(NULL);

   /*-------------------------*/
   /*  Copy CDR of portlist  */
   /*-------------------------*/
   rest = Copy_Portlist(portlist->nextport);

   /*-----------------------------------*/
   /*  Get and init a new port element  */
   /*-----------------------------------*/
   /* printf("\tCopying port %d %d %d\n", portlist->pos.x, portlist->pos.y, portlist->theta); */
   newport = Newport(portlist->pos.x, portlist->pos.y, portlist->theta, portlist->move_strategy);
   newport->nextport = rest;

   return(newport);
}


/*-----------------------------------------------------------------------------------*/
/*  Function COPY_FEATURELIST                                                        */
/*                                                                                   */
/*  This function makes and returns a duplicate copy of the given linked list of features.  */
/*-----------------------------------------------------------------------------------*/
struct feature *Copy_Featurelist(f, hypo)
struct feature *f;
struct hypothesis *hypo;
{
```

```
   struct feature *rest, *new_feature, *Copy_Featurelist(), *New_Feature();

   /*----------------*/
   /*  Trivial case  */
   /*----------------*/
   if (f == NULL) return(NULL);

   /*---------------------------*/
   /*  Copy CDR of featurelist  */
   /*---------------------------*/
   rest = Copy_Featurelist(f->next_feature, hypo);

   /*-----------------------------------*/
   /*  Get and init a new feature element  */
   /*-----------------------------------*/
   /* printf("\tCopying feature %d at %g\n", f->feature_type, f->conceptual_angle); */
   new_feature = New_Feature(hypo, f->feature_type, f->conceptual_angle,
                             f->pos.x, f->pos.y, f->pos_error.x, f->pos_error.y,
                             f->theta, f->side, f->move_strategy);
   new_feature->next_feature = rest;

   return(new_feature);
 }


/*-------------------------------------------------------------*/
/*  Procedure FREE_HYPOTHESIS_LIST                             */
/*                                                             */
/*  This procedure frees the given linked list of hypotheses.  */
/*-------------------------------------------------------------*/
Free_Hypothesis_List(hyp)
struct hypothesis *hyp;
{
   struct hypothesis *rest;

   /*----------------*/
   /*  Trivial case  */
   /*----------------*/
   if (hyp == NULL) return;

   /*---------------------------------------------------------*/
   /*  Free the first hypothesis, then the rest of the list  */
   /*---------------------------------------------------------*/
   rest = hyp->next_hypothesis;
   Free_Hypothesis(hyp);

   Free_Hypothesis_List(rest);
 }


/*-----------------------------------------------------------------*/
/*  Procedure FREE_HYPOTHESIS                                      */
/*                                                                 */
/*  This procedure frees the given hypothesis structure.           */
/*-----------------------------------------------------------------*/
Free_Hypothesis(hyp)
struct hypothesis *hyp;
{
   struct hypothesis *rest;

   /*----------------*/
   /*  Trivial case  */
   /*----------------*/
   if (hyp == NULL) return;

   /*---------------------------------------------------------*/
   /*  Free the hypothesis after freeing its attached lists  */
   /*---------------------------------------------------------*/
   /* printf("Freeing hypothesis %d index %d\n", hyp->LDP_type, hyp->index); */
   Free_Portlist(hyp->portlist);
   Free_Featurelist(hyp->featurelist);

   free(hyp);
 }


/*-----------------------------------------------------------------*/
/*  Procedure FREE_PORTLIST                                        */
/*                                                                 */
/*  This procedure frees the given linked list of port structures.  */
/*-----------------------------------------------------------------*/
Free_Portlist(port)
struct port *port;
{
   struct port *rest;
```

```
   /*----------------*/
   /*  Trivial case  */
   /*----------------*/
   if (port == NULL) return;

   /*---------------------------------------*/
   /*  Free port, then the rest of the list */
   /*---------------------------------------*/
   /* printf("\tFreeing port %d %d %d\n", port->pos.x, port->pos.y, port->theta); */
   rest = port->nextport;
   free(port);

   Free_Portlist(rest);
 }


/*-----------------------------------------------------------------------*/
/*  Function FREE_FEATURELIST                                            */
/*                                                                       */
/*  This procedure frees the given linked list of feature structures.  */
/*-----------------------------------------------------------------------*/
Free_Featurelist(feature)
struct feature *feature;
{
   struct feature *rest;

   /*----------------*/
   /*  Trivial case  */
   /*----------------*/
   if (feature == NULL) return;

   /*---------------------------------------*/
   /*  Free feature, then the rest of the list */
   /*---------------------------------------*/
   /* printf("\tFreeing feature %d at %g\n", feature->feature_type, feature->conceptual_angle); */
   rest = feature->next_feature;
   free(feature);

   Free_Featurelist(rest);
 }
```

```
#include "/pro/ai/robot/software/huey/utils/src/common.h"
#include "/pro/ai/robot/software/huey/utils/src/utils.h"
#include "/u/rac/Masters/Bride/src/bride.h"
#include "/u/rac/Masters/Ldp/src/ldp.h"

#include <stdio.h>
#include <math.h>

/* #define PRINT_UTILITY   1 */
/* #define PRINT_INDEPTH   1 */

/*-------------------------------------------------------------------------------*/
/*  Function LDPDIAGRAM                                                          */
/*                                                                               */
/*  This function builds and returns an influence diagram to be used for recognition   */
/*  of locally-distinctive places (LDP's) from feature detectors on a mobile robot.    */
/*  The diagram is used both for inferring a belief value over the set of possible     */
/*  LDP's (using probabilistic inference), and for choosing each successive feature    */
/*  detector to be invoked (using decision analysis).                            */
/*-------------------------------------------------------------------------------*/
struct network *LDPdiagram()
{
   struct network *diag;
   struct selector *s, *Select();
   int i, j;
   char *malloc();

   /*--------------------------------------*/
   /*  Get the influence diagram storage  */
   /*--------------------------------------*/
   diag = (struct network *) malloc(DIAGSIZE);

   diag->lastnode   = LASTNODE;
   diag->dstar      = NULL;
   diag->lastdstar  = NULL;
   diag->goalnode   = NULL;
   diag->RemoveList = NULL;

   /*--------------------*/
   /*  Initialize nodes  */
   /*--------------------*/
   New_Node(diag, WALL0,       STATE, BINARY);
   New_Node(diag, CONVEX45,    STATE, BINARY);
   New_Node(diag, CONCAVE45,   STATE, BINARY);
   New_Node(diag, EDGE45,      STATE, BINARY);
   New_Node(diag, CONVEX90,    STATE, BINARY);
   New_Node(diag, WALL90,      STATE, BINARY);
   New_Node(diag, EDGE90,      STATE, BINARY);
   New_Node(diag, CONVEX270,   STATE, BINARY);
   New_Node(diag, WALL270,     STATE, BINARY);
   New_Node(diag, EDGE270,     STATE, BINARY);
   New_Node(diag, CONVEX315,   STATE, BINARY);
   New_Node(diag, CONCAVE315,  STATE, BINARY);
   New_Node(diag, EDGE315,     STATE, BINARY);

   New_Node(diag, HYPOTHESIS_NODE, STATE,    NLDPS);
   New_Node(diag, DECISION_NODE,   DECISION, 0);
   New_Node(diag, VALUE_NODE,      VALUE,    VALUESTATES);

   /*-------------------------------------------------------*/
   /*  Zero out connectivity matrix and initialize connections  */
   /*-------------------------------------------------------*/
   Copy_Connectivity(diag->matrix, NULL);

   diag->matrix [HYPOTHESIS_NODE] [WALL0]      = 1;
   diag->matrix [HYPOTHESIS_NODE] [CONVEX45]   = 1;
   diag->matrix [HYPOTHESIS_NODE] [CONCAVE45]  = 1;
   diag->matrix [HYPOTHESIS_NODE] [EDGE45]     = 1;
   diag->matrix [HYPOTHESIS_NODE] [CONVEX90]   = 1;
   diag->matrix [HYPOTHESIS_NODE] [WALL90]     = 1;
   diag->matrix [HYPOTHESIS_NODE] [EDGE90]     = 1;
   diag->matrix [HYPOTHESIS_NODE] [CONVEX270]  = 1;
   diag->matrix [HYPOTHESIS_NODE] [WALL270]    = 1;
   diag->matrix [HYPOTHESIS_NODE] [EDGE270]    = 1;
   diag->matrix [HYPOTHESIS_NODE] [CONVEX315]  = 1;
   diag->matrix [HYPOTHESIS_NODE] [CONCAVE315] = 1;
   diag->matrix [HYPOTHESIS_NODE] [EDGE315]    = 1;

   diag->matrix [DECISION_NODE]   [VALUE_NODE]    = 1;
   diag->matrix [HYPOTHESIS_NODE] [VALUE_NODE]    = 1;


   /*-----------------------------------------*/
   /*  Initialize a priori node distributions  */
   /*-----------------------------------------*/
   Init_Known_Distributions(diag);
```

```
   return(diag);
 }


/*-------------------------------------------------------------------------------*/
/*  Function HYPOTHESIS_NODE                                                     */
/*                                                                               */
/*  this function returns a pointer to the node in the given diagram that contains  */
/*  the distribution of beliefs about ldp hypotheses.                            */
/*-------------------------------------------------------------------------------*/
struct node *Hypothesis_Node(diagram)
struct network *diagram;
{
   if (diagram == NULL)  printf("ERROR: Hypothesis_Node passed null diagram\n");
   return(&(diagram->nodelist[HYPOTHESIS_NODE]));
 }


/*-------------------------------------------------------------------------------*/
/*  Function DECISION_NODE                                                       */
/*                                                                               */
/*  This function returns a pointer to the node in the given diagram that contains  */
/*  the decision policy for choosing the next feature detector.                  */
/*-------------------------------------------------------------------------------*/
struct node *Decision_Node(diagram)
struct network *diagram;
{
   if (diagram == NULL)  printf("ERROR: Decision_Node passed null diagram\n");
   return(&(diagram->nodelist[DECISION_NODE]));
 }


/*-------------------------------------------------------------------------*/
/*  Function VALUE_NODE                                                    */
/*                                                                         */
/*  This function returns a pointer to the node in the given diagram that  */
/*  represents the utility function of the decision problem.               */
/*-------------------------------------------------------------------------*/
struct node *Value_Node(diagram)
struct network *diagram;
{
   if (diagram == NULL)  printf("ERROR: Value_Node passed null diagram\n");
   return(&(diagram->nodelist[VALUE_NODE]));
 }


/*-------------------------------------------------------------------------------*/
/*  Procedure INIT_KNOWN_DISTRIBUTIONS                                           */
/*                                                                               */
/*  This procedure initializes those nodes in the ldp inference diagram with known   */
/*  a priori distributions.                                                      */
/*-------------------------------------------------------------------------------*/
Init_Known_Distributions(diag)
struct network *diag;
{
   struct selector *s, *Select();
   float p;
   int n;

   /*-----------------------------*/
   /*  Set wall @ 0 distribution  */
   /*-----------------------------*/
   Set_Node_Prob(diag, WALL0, (s = Select(YES, TFORW,  -1)), 1.0);   Free_Selector(s);
   Set_Node_Prob(diag, WALL0, (s = Select(NO,  TLEFT,  -1)), 1.0);   Free_Selector(s);
   Set_Node_Prob(diag, WALL0, (s = Select(NO,  TRIGHT, -1)), 1.0);   Free_Selector(s);
   Set_Node_Prob(diag, WALL0, (s = Select(NO,  LLEFT,  -1)), 1.0);   Free_Selector(s);
   Set_Node_Prob(diag, WALL0, (s = Select(NO,  LRIGHT, -1)), 1.0);   Free_Selector(s);
   Set_Node_Prob(diag, WALL0, (s = Select(NO,  CROSS,  -1)), 1.0);   Free_Selector(s);
   Set_Node_Prob(diag, WALL0, (s = Select(NO,  RDOOR1, -1)), 1.0);   Free_Selector(s);
   Set_Node_Prob(diag, WALL0, (s = Select(NO,  RDOOR2, -1)), 1.0);   Free_Selector(s);
   Set_Node_Prob(diag, WALL0, (s = Select(NO,  LDOOR1, -1)), 1.0);   Free_Selector(s);
   Set_Node_Prob(diag, WALL0, (s = Select(NO,  LDOOR2, -1)), 1.0);   Free_Selector(s);
   Set_Node_Prob(diag, WALL0, (s = Select(NO,  DOORS1, -1)), 1.0);   Free_Selector(s);
   Set_Node_Prob(diag, WALL0, (s = Select(NO,  DOORS2, -1)), 1.0);   Free_Selector(s);

   /*--------------------------------*/
   /*  Set CONVEX @ 45 distribution  */
   /*--------------------------------*/
   Set_Node_Prob(diag, CONVEX45, (s = Select(NO,  TFORW,  -1)), 1.0);   Free_Selector(s);
   Set_Node_Prob(diag, CONVEX45, (s = Select(NO,  TLEFT,  -1)), 1.0);   Free_Selector(s);
   Set_Node_Prob(diag, CONVEX45, (s = Select(YES, TRIGHT, -1)), 1.0);   Free_Selector(s);
   Set_Node_Prob(diag, CONVEX45, (s = Select(NO,  LLEFT,  -1)), 1.0);   Free_Selector(s);
   Set_Node_Prob(diag, CONVEX45, (s = Select(NO,  LRIGHT, -1)), 1.0);   Free_Selector(s);
   Set_Node_Prob(diag, CONVEX45, (s = Select(YES, CROSS,  -1)), 1.0);   Free_Selector(s);
   Set_Node_Prob(diag, CONVEX45, (s = Select(YES, RDOOR1, -1)), 1.0);   Free_Selector(s);
   Set_Node_Prob(diag, CONVEX45, (s = Select(NO,  RDOOR2, -1)), 1.0);   Free_Selector(s);
   Set_Node_Prob(diag, CONVEX45, (s = Select(NO,  LDOOR1, -1)), 1.0);   Free_Selector(s);
   Set_Node_Prob(diag, CONVEX45, (s = Select(NO,  LDOOR2, -1)), 1.0);   Free_Selector(s);
```

```
Set_Node_Prob(diag, CONVEX45, (s = Select(NO,  DOORS1, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONVEX45, (s = Select(YES, DOORS2, -1)), 1.0);   Free_Selector(s);

/*---------------------------------*/
/*  Set CONCAVE @ 45 distribution  */
/*---------------------------------*/
Set_Node_Prob(diag, CONCAVE45, (s = Select(NO,  TFORW,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONCAVE45, (s = Select(NO,  TLEFT,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONCAVE45, (s = Select(NO,  TRIGHT, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONCAVE45, (s = Select(YES, LLEFT,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONCAVE45, (s = Select(NO,  LRIGHT, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONCAVE45, (s = Select(NO,  CROSS,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONCAVE45, (s = Select(NO,  RDOOR1, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONCAVE45, (s = Select(NO,  RDOOR2, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONCAVE45, (s = Select(NO,  LDOOR1, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONCAVE45, (s = Select(NO,  LDOOR2, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONCAVE45, (s = Select(NO,  DOORS1, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONCAVE45, (s = Select(NO,  DOORS2, -1)), 1.0);   Free_Selector(s);

/*-------------------------------*/
/*  Set edge @ 45 distribution  */
/*-------------------------------*/
Set_Node_Prob(diag, EDGE45, (s = Select(NO,  TFORW,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE45, (s = Select(NO,  TLEFT,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE45, (s = Select(NO,  TRIGHT, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE45, (s = Select(NO,  LLEFT,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE45, (s = Select(NO,  LRIGHT, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE45, (s = Select(NO,  CROSS,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE45, (s = Select(NO,  RDOOR1, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE45, (s = Select(YES, RDOOR2, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE45, (s = Select(NO,  LDOOR1, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE45, (s = Select(NO,  LDOOR2, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE45, (s = Select(YES, DOORS1, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE45, (s = Select(NO,  DOORS2, -1)), 1.0);   Free_Selector(s);

/*---------------------------------*/
/*  Set CONVEX @ 90 distribution  */
/*---------------------------------*/
Set_Node_Prob(diag, CONVEX90, (s = Select(YES, TFORW,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONVEX90, (s = Select(NO,  TLEFT,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONVEX90, (s = Select(YES, TRIGHT, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONVEX90, (s = Select(NO,  LLEFT,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONVEX90, (s = Select(YES, LRIGHT, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONVEX90, (s = Select(YES, CROSS,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONVEX90, (s = Select(NO,  RDOOR1, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONVEX90, (s = Select(YES, RDOOR2, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONVEX90, (s = Select(NO,  LDOOR1, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONVEX90, (s = Select(NO,  LDOOR2, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONVEX90, (s = Select(YES, DOORS1, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONVEX90, (s = Select(NO,  DOORS2, -1)), 1.0);   Free_Selector(s);

/*-------------------------------*/
/*  Set wall @ 90 distribution  */
/*-------------------------------*/
Set_Node_Prob(diag, WALL90, (s = Select(NO,  TFORW,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, WALL90, (s = Select(YES, TLEFT,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, WALL90, (s = Select(NO,  TRIGHT, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, WALL90, (s = Select(NO,  LLEFT,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, WALL90, (s = Select(NO,  LRIGHT, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, WALL90, (s = Select(NO,  CROSS,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, WALL90, (s = Select(NO,  RDOOR1, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, WALL90, (s = Select(NO,  RDOOR2, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, WALL90, (s = Select(YES, LDOOR1, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, WALL90, (s = Select(YES, LDOOR2, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, WALL90, (s = Select(NO,  DOORS1, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, WALL90, (s = Select(NO,  DOORS2, -1)), 1.0);   Free_Selector(s);

/*-------------------------------*/
/*  Set edge @ 90 distribution  */
/*-------------------------------*/
Set_Node_Prob(diag, EDGE90, (s = Select(NO,  TFORW,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE90, (s = Select(NO,  TLEFT,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE90, (s = Select(NO,  TRIGHT, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE90, (s = Select(NO,  LLEFT,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE90, (s = Select(NO,  LRIGHT, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE90, (s = Select(NO,  CROSS,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE90, (s = Select(YES, RDOOR1, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE90, (s = Select(NO,  RDOOR2, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE90, (s = Select(NO,  LDOOR1, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE90, (s = Select(NO,  LDOOR2, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE90, (s = Select(NO,  DOORS1, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE90, (s = Select(YES, DOORS2, -1)), 1.0);   Free_Selector(s);

/*---------------------------------*/
/*  Set CONVEX @ 270 distribution */
/*---------------------------------*/
Set_Node_Prob(diag, CONVEX270, (s = Select(YES, TFORW,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONVEX270, (s = Select(YES, TLEFT,  -1)), 1.0);   Free_Selector(s);
```

```
Set_Node_Prob(diag, CONVEX270, (s = Select(NO,  TRIGHT, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONVEX270, (s = Select(YES, LLEFT,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONVEX270, (s = Select(NO,  LRIGHT, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONVEX270, (s = Select(YES, CROSS,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONVEX270, (s = Select(NO,  RDOOR1, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONVEX270, (s = Select(NO,  RDOOR2, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONVEX270, (s = Select(NO,  LDOOR1, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONVEX270, (s = Select(YES, LDOOR2, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONVEX270, (s = Select(NO,  DOORS1, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONVEX270, (s = Select(YES, DOORS2, -1)), 1.0);   Free_Selector(s);

/*-------------------------------*/
/*  Set wall @ 270 distribution */
/*-------------------------------*/
Set_Node_Prob(diag, WALL270, (s = Select(NO,  TFORW,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, WALL270, (s = Select(NO,  TLEFT,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, WALL270, (s = Select(YES, TRIGHT, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, WALL270, (s = Select(NO,  LLEFT,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, WALL270, (s = Select(NO,  LRIGHT, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, WALL270, (s = Select(NO,  CROSS,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, WALL270, (s = Select(YES, RDOOR1, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, WALL270, (s = Select(YES, RDOOR2, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, WALL270, (s = Select(NO,  LDOOR1, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, WALL270, (s = Select(NO,  LDOOR2, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, WALL270, (s = Select(NO,  DOORS1, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, WALL270, (s = Select(NO,  DOORS2, -1)), 1.0);   Free_Selector(s);

/*-------------------------------*/
/*  Set EDGE @ 270 distribution */
/*-------------------------------*/
Set_Node_Prob(diag, EDGE270, (s = Select(NO,  TFORW,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE270, (s = Select(NO,  TLEFT,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE270, (s = Select(NO,  TRIGHT, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE270, (s = Select(NO,  LLEFT,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE270, (s = Select(NO,  LRIGHT, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE270, (s = Select(NO,  CROSS,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE270, (s = Select(NO,  RDOOR1, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE270, (s = Select(NO,  RDOOR2, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE270, (s = Select(YES, LDOOR1, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE270, (s = Select(NO,  LDOOR2, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE270, (s = Select(YES, DOORS1, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE270, (s = Select(NO,  DOORS2, -1)), 1.0);   Free_Selector(s);

/*---------------------------------*/
/*  Set CONVEX @ 315 distribution */
/*---------------------------------*/
Set_Node_Prob(diag, CONVEX315, (s = Select(NO,  TFORW,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONVEX315, (s = Select(YES, TLEFT,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONVEX315, (s = Select(NO,  TRIGHT, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONVEX315, (s = Select(NO,  LLEFT,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONVEX315, (s = Select(NO,  LRIGHT, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONVEX315, (s = Select(YES, CROSS,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONVEX315, (s = Select(NO,  RDOOR1, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONVEX315, (s = Select(NO,  RDOOR2, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONVEX315, (s = Select(YES, LDOOR1, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONVEX315, (s = Select(NO,  LDOOR2, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONVEX315, (s = Select(YES, DOORS1, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONVEX315, (s = Select(NO,  DOORS2, -1)), 1.0);   Free_Selector(s);


/*---------------------------------*/
/*  Set CONCAVE @ 315 distribution */
/*---------------------------------*/
Set_Node_Prob(diag, CONCAVE315, (s = Select(NO,  TFORW,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONCAVE315, (s = Select(NO,  TLEFT,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONCAVE315, (s = Select(NO,  TRIGHT, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONCAVE315, (s = Select(NO,  LLEFT,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONCAVE315, (s = Select(YES, LRIGHT, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONCAVE315, (s = Select(NO,  CROSS,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONCAVE315, (s = Select(NO,  RDOOR1, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONCAVE315, (s = Select(NO,  RDOOR2, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONCAVE315, (s = Select(NO,  LDOOR1, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONCAVE315, (s = Select(NO,  LDOOR2, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONCAVE315, (s = Select(NO,  DOORS1, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, CONCAVE315, (s = Select(NO,  DOORS2, -1)), 1.0);   Free_Selector(s);

/*-------------------------------*/
/*  Set EDGE @ 315 distribution */
/*-------------------------------*/
Set_Node_Prob(diag, EDGE315, (s = Select(NO,  TFORW,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE315, (s = Select(NO,  TLEFT,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE315, (s = Select(NO,  TRIGHT, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE315, (s = Select(NO,  LLEFT,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE315, (s = Select(NO,  LRIGHT, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE315, (s = Select(NO,  CROSS,  -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE315, (s = Select(NO,  RDOOR1, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE315, (s = Select(NO,  RDOOR2, -1)), 1.0);   Free_Selector(s);
Set_Node_Prob(diag, EDGE315, (s = Select(NO,  LDOOR1, -1)), 1.0);   Free_Selector(s);
```

```
   Set_Node_Prob(diag, EDGE315, (s = Select(YES, LDOOR2, -1)), 1.0);   Free_Selector(s);
   Set_Node_Prob(diag, EDGE315, (s = Select(NO,  DOORS1, -1)), 1.0);   Free_Selector(s);
   Set_Node_Prob(diag, EDGE315, (s = Select(YES, DOORS2, -1)), 1.0);   Free_Selector(s);

 }


/*-------------------------------------------------------------------------------------------*/
/*  Function SET_DIAGRAM                                                                     */
/*                                                                                           */
/*  This procedure sets the goalnode and retain flags in the given diagram for the appropriate  */
/*  nodes.  These flags dictate what distribution will be inferred by the inference machinery.  */
/*  The second argument indicates which of the supported distributions is to be set.         */
/*-------------------------------------------------------------------------------------------*/
Set_Diagram(diagram, disttype)
struct network *diagram;
int disttype;
{

   /*------------------*/
   /*  Error checking  */
   /*------------------*/
   if (diagram == NULL) { printf("ERROR: Set_Diagram passed NULL diagram\n");  return; }

   if (disttype != MAINDIST  &&  disttype != CONTROLPROB) {
      printf("ERROR: Set_Diagram passed invalid distribution selection flag\n"); return; }

   /*-----------------------------------------------------------------*/
   /*  Now set the proper nodes in preparation for inference based on  */
   /*  which distribution is desired.                                 */
   /*-----------------------------------------------------------------*/
   switch(disttype) {

      /*-------------------------------------------------*/
      /*  For distribution pr{ldp | <detector readings> }  */
      /*-------------------------------------------------*/
      case MAINDIST:
         diagram->type = INFERENCE;
         diagram->goalnode = &(diagram->nodelist[HYPOTHESIS_NODE]);
         diagram->nodelist[HYPOTHESIS_NODE].goalnode = 1;
         break;

      /*-----------------------------------------*/
      /*  For decision analysis on control diagram  */
      /*-----------------------------------------*/
      case CONTROLPROB:
         diagram->type = CONTROL;
         diagram->goalnode = &(diagram->nodelist[VALUE_NODE]);
         diagram->nodelist[VALUE_NODE].goalnode = 1;
         break; }
 }



/*-------------------------------------------------------------------------------------------*/
/*  Function BUILD_ACCESS_SELECTOR                                                           */
/*                                                                                           */
/*  This function returns a selector that selects all the predecessors of the goal node.     */
/*  The STATE of each selector element is initialized from the known STATE of the            */
/*  the corresponding predecessor.                                                           */
/*                                                                                           */
/*  NOTE: This function is very similar to select_predecessors.                             */
/*-------------------------------------------------------------------------------------------*/
struct selector *Build_Access_Selector(goalnode)
struct node *goalnode;
{
   struct node *pred, *Next_Predecessor();
   struct selector *front, *rear, *s, *NewSel();

   /*------------------*/
   /*  Error checking  */
   /*------------------*/
   if (goalnode == NULL) printf("error: build_access_selector passed NULL goal node\n");

   /*--------------------------------*/
   /*  Set up for predecessor loop  */
   /*--------------------------------*/
   pred = Next_Predecessor(goalnode, NULL, ANY);
   front = rear = NULL;

   /*-------------------------------------------------------*/
   /*  Loop thru predecessors of nodeptr, adding a selector  */
   /*  to the selector list for each.                        */
   /*-------------------------------------------------------*/
   while (pred != NULL) {
      s = NewSel(pred, pred->knownstate);

      if (front == NULL) front = s;
```

```
      else rear->Next = s;
      rear = s;

      pred = Next_Predecessor(goalnode, pred, ANY); }

   /*---------------------------------------*/
   /*  Return the constructed selector list  */
   /*---------------------------------------*/
   return(front);
 }


/*-------------------------------------------------------------------------------------*/
/*  Function MAP_XYPOSN                                                                */
/*                                                                                     */
/*-------------------------------------------------------------------------------------*/
int Map_xyposn(pos)
FRM_POINT pos;
{
   int Map_Posn();
   double anglerad, angledeg, anglecompass, atan2();
   static double pi = 3.14159;

   anglerad = atan2((double) pos.y, (double) pos.x);
   angledeg = anglerad * 180.0 / pi;

   if (pos.x < 0  &&  pos.y >= 0)  anglecompass = 450.0 - angledeg;
   else anglecompass = 90.0 - angledeg;

   printf("point (%d %d) maps to angle %g\n", pos.x, pos.y, (float) anglecompass);
   return(Map_Posn(anglecompass));
 }


/*-------------------------------------------------------------------------------------*/
/*  Function MAP_POSN                                                                  */
/*                                                                                     */
/*  This function maps a continuous position value (given in degrees theta from straight  */
/*  ahead), into the discrete representation used for the influence diagram.           */
/*-------------------------------------------------------------------------------------*/
int Map_Posn(theta)
float theta;
{
   if (theta >= 0.0    &&  theta < 23.0)   return(ANGLE0);
   if (theta >= 23.0   &&  theta < 67.0)   return(ANGLE45);
   if (theta >= 67.0   &&  theta < 112.0)  return(ANGLE90);
   if (theta >= 112.0  &&  theta < 157.0)  return(ANGLE135);
   if (theta >= 157.0  &&  theta < 202.0)  return(ANGLE180);
   if (theta >= 202.0  &&  theta < 247.0)  return(ANGLE225);
   if (theta >= 247.0  &&  theta < 292.0)  return(ANGLE270);
   if (theta >= 292.0  &&  theta < 337.0)  return(ANGLE315);
   if (theta >= 337.0  &&  theta <= 360.0) return(ANGLE0);

   printf("error: Map_Posn cannot categorize theta %g\n", theta);
   return(-1);
 }


/*-------------------------------------------------------------------------------------*/
/*  Function MAP_DETECTOR                                                              */
/*                                                                                     */
/*  This function maps the given feature type and angle into an integer corresponding to  */
/*  the feature detector that searches for that feature at that angle.                 */
/*-------------------------------------------------------------------------------------*/
int Map_Detector(feature, theta)
int feature;
float theta;
{
   int Map_Posn();

   if (feature == WALL            &&  Map_Posn(theta) == ANGLE0)    return(WALL0);
   if (feature == CONVEX_CORNER   &&  Map_Posn(theta) == ANGLE45)   return(CONVEX45);
   if (feature == CONCAVE_CORNER  &&  Map_Posn(theta) == ANGLE45)   return(CONCAVE45);
   if (feature == EDGE            &&  Map_Posn(theta) == ANGLE45)   return(EDGE45);
   if (feature == CONVEX_CORNER   &&  Map_Posn(theta) == ANGLE90)   return(CONVEX90);
   if (feature == WALL            &&  Map_Posn(theta) == ANGLE90)   return(WALL90);
   if (feature == EDGE            &&  Map_Posn(theta) == ANGLE90)   return(EDGE90);
   if (feature == CONVEX_CORNER   &&  Map_Posn(theta) == ANGLE270)  return(CONVEX270);
   if (feature == WALL            &&  Map_Posn(theta) == ANGLE270)  return(WALL270);
   if (feature == EDGE            &&  Map_Posn(theta) == ANGLE270)  return(EDGE270);
   if (feature == CONVEX_CORNER   &&  Map_Posn(theta) == ANGLE315)  return(CONVEX315);
   if (feature == CONCAVE_CORNER  &&  Map_Posn(theta) == ANGLE315)  return(CONCAVE315);
   if (feature == EDGE            &&  Map_Posn(theta) == ANGLE315)  return(EDGE315);

   printf("error: map_detector cannot categorize feature %d @ %g\n", feature, theta);
   return(-1);
 }
```

```
/*-------------------------------------------------------------------------------*/
/*  Function INITIALIZE_HYPOTHESES                                               */
/*                                                                               */
/*  This function sets up and returns the list of hypotheses, including the attached  */
/*  feature list.                                                                */
/*-------------------------------------------------------------------------------*/
struct hypothesis *Initialize_Hypotheses()
{
   struct hypothesis *top, *h1, *h2, *h3, *h4, *h5, *h6, *h7, *h8, *h9, *h10, *h11, *h12;
   struct hypothesis *New_Hypothesis();

   /*-----------------------------------------*/
   /*  Build each hypothesis with its features  */
   /*-----------------------------------------*/
   h1 = New_Hypothesis(TLEFT, 0);
   Attach_Feature(h1, CONVEX_CORNER,  270.0,  0,  -50, 20, 20, -90, FRM_LEFT, STRAIGHT);
   Attach_Feature(h1, WALL,            90.0,  0,   64,  0,  0,  90,        0, STRAIGHT);
   Attach_Feature(h1, CONVEX_CORNER,  315.0, -114, 64, 20, 20,   0, FRM_LEFT, Y_FIRST);
   Attach_Port    (h1, -64,  64, -90, Y_FIRST);
   Attach_Port    (h1,   0, 128,   0, STRAIGHT);

   h2 = New_Hypothesis(TFORW, 1);
   Attach_Feature(h2, CONVEX_CORNER,  270.0,   0,  -50, 20, 20, -90, FRM_LEFT,  STRAIGHT);
   Attach_Feature(h2, WALL,             0.0,   0,   64,  0,  0,   0,        0,  STRAIGHT);
   Attach_Feature(h2, CONVEX_CORNER,   90.0,   0,  -50, 20, 20,  90, FRM_RIGHT, STRAIGHT);
   Attach_Port    (h2, -64, 64, -90, Y_FIRST);
   Attach_Port    (h2,  64, 64,  90, Y_FIRST);

   h3 = New_Hypothesis(RDOOR1, 2);
   Attach_Feature(h3, EDGE,            90.0,  0,   -50, 20, 20,  90, FRM_RIGHT, STRAIGHT);
   Attach_Feature(h3, WALL,           270.0,  0,    64,  0,  0, -90,        0,  STRAIGHT);
   Attach_Feature(h3, CONVEX_CORNER,   45.0, 114,   64, 20, 20,   0, FRM_RIGHT, Y_FIRST);
   Attach_Port    (h3, 0, 128, 0, STRAIGHT);

   h4 = New_Hypothesis(LRIGHT, 3);
   Attach_Feature(h4, CONVEX_CORNER,   90.0,   0,  -50, 20, 20,  90, FRM_RIGHT, STRAIGHT);
   Attach_Feature(h4, CONCAVE_CORNER, 315.0,   0,   12, 20, 20, -90, FRM_LEFT,  STRAIGHT);
   Attach_Port    (h4, 64, 64, 90, Y_FIRST);

   h5 = New_Hypothesis(LLEFT, 4);
   Attach_Feature(h5, CONVEX_CORNER,  270.0,   0,  -50, 20, 20, -90, FRM_LEFT,  STRAIGHT);
   Attach_Feature(h5, CONCAVE_CORNER,  45.0,   0,   12, 20, 20,  90, FRM_RIGHT, STRAIGHT);
   Attach_Port    (h5, -64, 64, -90, Y_FIRST);

   h6 = New_Hypothesis(TRIGHT, 5);
   Attach_Feature(h6, CONVEX_CORNER,   90.0,  0,  -50, 20, 20,  90, FRM_RIGHT, STRAIGHT);
   Attach_Feature(h6, WALL,           270.0,  0,   64,  0,  0, -90,        0,  STRAIGHT);
   Attach_Feature(h6, CONVEX_CORNER,   45.0, 114,  64, 20, 20,   0, FRM_RIGHT, Y_FIRST);
   Attach_Port    (h6, 64,  64, 90, Y_FIRST);
   Attach_Port    (h6,  0, 128,  0, STRAIGHT);

   h7 = New_Hypothesis(CROSS, 6);
   Attach_Feature(h7, CONVEX_CORNER,  270.0,    0,  -50, 20, 20, -90, FRM_LEFT,  STRAIGHT);
   Attach_Feature(h7, CONVEX_CORNER,   90.0,    0,  -50, 20, 20,  90, FRM_RIGHT, STRAIGHT);
   Attach_Feature(h7, CONVEX_CORNER,   45.0,  114,   64, 20, 20,   0, FRM_RIGHT, Y_FIRST);
   Attach_Feature(h7, CONVEX_CORNER,  315.0, -114,   64, 20, 20,   0, FRM_LEFT,  Y_FIRST);
   Attach_Port    (h7, -64,  64, -90, Y_FIRST);
   Attach_Port    (h7,   0, 128,   0, STRAIGHT);
   Attach_Port    (h7,  64,  64,  90, Y_FIRST);

   h8 = New_Hypothesis(RDOOR2, 7);
   Attach_Feature(h8, CONVEX_CORNER,   90.0,   0,  -50, 20, 20,  90, FRM_RIGHT, STRAIGHT);
   Attach_Feature(h8, WALL,           270.0,   0,   64,  0,  0, -90,        0,  STRAIGHT);
   Attach_Feature(h8, EDGE,            45.0,   0,  178, 20, 20,  90, FRM_LEFT,  STRAIGHT);
   Attach_Port    (h8,  0, 128,  0, STRAIGHT);

   h9 = New_Hypothesis(LDOOR1, 8);
   Attach_Feature(h9, EDGE,           270.0,    0,  -50, 20, 20, -90, FRM_LEFT, STRAIGHT);
   Attach_Feature(h9, WALL,            90.0,    0,   64,  0,  0,  90,        0, STRAIGHT);
   Attach_Feature(h9, CONVEX_CORNER,  315.0, -114,   64, 20, 20,   0, FRM_LEFT, Y_FIRST);
   Attach_Port    (h9,  0, 128,  0, STRAIGHT);

   h10 = New_Hypothesis(LDOOR2, 9);
   Attach_Feature(h10, CONVEX_CORNER,  270.0,   0,  -50, 20, 20, -90, FRM_LEFT,  STRAIGHT);
   Attach_Feature(h10, WALL,            90.0,   0,   64,  0,  0,  90,        0,  STRAIGHT);
   Attach_Feature(h10, EDGE,           315.0,   0,  178, 20, 20, -90, FRM_RIGHT, STRAIGHT);
   Attach_Port    (h10,  0, 128,  0, STRAIGHT);

   h11 = New_Hypothesis(DOORS1, 10);
   Attach_Feature(h11, EDGE,           270.0,   0,  -50, 20, 20, -90, FRM_LEFT,  STRAIGHT);
   Attach_Feature(h11, CONVEX_CORNER,   90.0,   0,  -50, 20, 20,  90, FRM_RIGHT, STRAIGHT);
   Attach_Feature(h11, CONVEX_CORNER,  315.0, -114,  64, 20, 20,   0, FRM_LEFT,  Y_FIRST);
   Attach_Feature(h11, EDGE,            45.0,   0,  178, 20, 20,  90, FRM_LEFT,  STRAIGHT);
   Attach_Port    (h11,  0, 128,  0, STRAIGHT);

   h12 = New_Hypothesis(DOORS2, 11);
   Attach_Feature(h12, CONVEX_CORNER,  270.0,   0,  -50, 20, 20, -90, FRM_LEFT,  STRAIGHT);
```

```
   Attach_Feature(h12, EDGE,            90.0,   0,  -50, 20, 20,  90, FRM_RIGHT, STRAIGHT);
   Attach_Feature(h12, CONVEX_CORNER,   45.0,  114,  64, 20, 20,   0, FRM_RIGHT, Y_FIRST);
   Attach_Feature(h12, EDGE,           315.0,   0,  178, 20, 20, -90, FRM_RIGHT, STRAIGHT);
   Attach_Port    (h12,  0, 128,  0, STRAIGHT);

   /*-----------------------*/
   /*  Link the hypotheses  */
   /*-----------------------*/
   top = h1;
   h1->next_hypothesis  = h2;
   h2->next_hypothesis  = h3;
   h3->next_hypothesis  = h4;
   h4->next_hypothesis  = h5;
   h5->next_hypothesis  = h6;
   h6->next_hypothesis  = h7;
   h7->next_hypothesis  = h8;
   h8->next_hypothesis  = h9;
   h9->next_hypothesis  = h10;
   h10->next_hypothesis = h11;
   h11->next_hypothesis = h12;
   h12->next_hypothesis = NULL;

   return(top);
}


/*-------------------------------------------------------------------------------------------*/
/*  Function UTILITY                                                                         */
/*                                                                                           */
/*  This function calculates and returns a distribution containing the utility function     */
/*  for the LDPcontrol network.  The utility measures the utility of choosing to dispatch   */
/*  a particular feature detector, and is a function of the feature detector and the        */
/*  ldp hypothesis.  utility = k1 * value measure - k2 * cost measure.                       */
/*-------------------------------------------------------------------------------------------*/
struct distribution *Utility(detectlist, hyplist, detector_diagram)
struct feature *detectlist;
struct hypothesis *hyplist;
struct network *detector_diagram;
{
   struct distribution *utildist, *New_Distribution();
   int dindex, h;
   struct feature *d;
   float sum, utility, belief, measure, cost, Cost(), Measure();
   struct selector *sel, *decision_sel, *in, *Vpreds, *Inserted_Copy();
   struct selector *Select_Predecessors(), *Purge_Flagged(), *NewSel(), *Select();
   struct hypothesis *hyp;
   struct node *valnode, *decnode, *Value_Node(), *Decision_Node();
   BOOLEAN oflag, Odometer();

   /*--------------------------------*/
   /*  Weight value more than cost   */
   /*--------------------------------*/
   static float kvalue = 100.0;
   static float kcost  = 0.05;

   /*------------------*/
   /*  Error checking  */
   /*------------------*/
   if (detectlist == NULL)   printf("ERROR: Utility passed NULL detector list\n");
   if (hyplist == NULL)      printf("ERROR: Utility passed NULL hypothesis list\n");
   if (detector_diagram == NULL) printf("ERROR: Utility passed NULL diagram\n");

   /*-----------------------------------*/
   /*  Prepare for loop thru detectors  */
   /*-----------------------------------*/
   valnode = Value_Node(detector_diagram);
   decnode = Decision_Node(detector_diagram);

   decision_sel = NewSel(decnode, 0);
   utildist = New_Distribution();

   dindex = 0;
   d = detectlist;

   /*-----------------------------------*/
   /*  Loop thru all detectors in list  */
   /*-----------------------------------*/
   while (d != NULL)  {

      /*-------------------------------------------------*/
      /*  Only calculate utility if the detector is unused  */
      /*-------------------------------------------------*/
      if (!d->detector_used) {

#ifdef PRINT_UTILITY
         printf("Detect = %d (%d @ %g)", dindex, d->feature_type, d->conceptual_angle);
#endif
```

```
        h = 0;
        hyp = hyplist;
        sum = 0.0;

        /*-------------------------------------------------*/
        /*  Loop thru all hypotheses in the hypothesis node  */
        /*-------------------------------------------------*/
        while (hyp != NULL) {

           measure = Measure(d, hyp, hyplist, detector_diagram);
           belief = hyp->certainty;

           sum = sum + belief * measure;

           /*---------------------------*/
           /*  Bump to next hypothesis  */
           /*---------------------------*/
           h++;
           hyp = hyp->next_hypothesis; }


        /*-------------------------------------------------*/
        /*  Calc utility as a function of value and cost,  */
        /*  and set it in the distribution.                */
        /*-------------------------------------------------*/
        cost  = Cost(d);
        utility = d->retries_left * (kvalue * sum - kcost * cost);

#ifdef PRINT_UTILITY
        printf("\tSum = %g\t\tCost = %g\tUtility = %g\n", sum, cost, utility);
#endif PRINT_UTILITY

        /*-----------------------------------------------------------------*/
        /*  Save the utility in the distribution. This must ensure that    */
        /*  the distribution is of dimensionality equal to the number of   */
        /*  predecessors of the value node!                                */
        /*-----------------------------------------------------------------*/
        Vpreds = Purge_Flagged( Select_Predecessors(valnode, decnode));
        decision_sel->state = dindex;

        oflag = 0;
        Zero(Vpreds, REGULAR);

        while (Odometer(Vpreds, REGULAR, &oflag)) {
           in = Inserted_Copy(decision_sel, Vpreds);
           Set_Prob(utildist, in, utility);
           Free_Selector(in); }

        Free_Selector(Vpreds);

        /*-------------------------------------*/
        /*  Bump counter of unused detectors  */
        /*-------------------------------------*/
        dindex++; }

     /*-------------------------*/
     /*  Bump to next detector  */
     /*-------------------------*/
     d = d->next_union; }

  Free_Selector(decision_sel);
  return(utildist);
}


/*-------------------------------------------------------------------------------------------*/
/*  Function MEASURE                                                                         */
/*                                                                                           */
/*  This function calculates SUM [ Pr{detector, result | LDP} - Pr{detector, result} ],     */
/*  across all values of result.                                                             */
/*-------------------------------------------------------------------------------------------*/
float Measure(detector, hyp, hyplist, detector_diagram)
struct feature *detector;
struct hypothesis *hyp, *hyplist;
struct network *detector_diagram;
{
   float sum, pi, uniform, detector_prob, Get_Prob(), Detector_Prob(), fabs();
   int detector_result;
   struct node *detector_node, *Detector_Node();
   struct selector *sel, *Select();

   /*------------------*/
   /*  Error checking  */
   /*------------------*/
   if (detector == NULL)          printf("ERROR: Measure passed null detector\n");
   if (hyp == NULL)               printf("ERROR: Measure passed null hypothesis\n");
   if (hyplist == NULL)           printf("ERROR: Measure passed null hypothesis list\n");
   if (detector_diagram == NULL)  printf("ERROR: Measure passed null diagram\n");
```

```
   /*--------------------*/
   /*  Prepare for loop  */
   /*--------------------*/
   detector_node = Detector_Node(detector_diagram, detector);
   uniform = 1.0 / detector_node->nstates;
   sum = 0.0;

   /*-----------------------------------------------------*/
   /*  Loop thru states of detector node, summing values  */
   /*-----------------------------------------------------*/
   for (detector_result = 0; detector_result < detector_node->nstates; detector_result++) {

      /*-------------------------------------------------------------*/
      /*  Get Pr{detector, result | LDP} directly from detector node  */
      /*-------------------------------------------------------------*/
      sel = Select(detector_result, hyp->LDP_type, -1);
      pi  = Get_Prob(detector_node->header, sel);
      Free_Selector(sel);

      /*--------------------------------------*/
      /*  Get calculated Pr{detector, result}  */
      /*--------------------------------------*/
      detector_prob = Detector_Prob(detector, detector_result, hyplist, detector_diagram);

      sum = sum + fabs(pi - detector_prob); }

#ifdef PRINT_INDEPTH
    printf("\tMeasure = %g", sum);
#endif

   return(sum);
 }


/*-------------------------------------------------------------------------------------------*/
/*  Function DETECTOR_PROB                                                                   */
/*                                                                                           */
/*  This function calculates Pr{detector, detector_result} by conditioning on the set       */
/*  of hypotheses.  "Detector" is equivalent to "feature, angle".  "Detector_result" is      */
/*  currently out of the set {0, 1} since detectors return a binary result.                  */
/*-------------------------------------------------------------------------------------------*/
float Detector_Prob(detector, detector_result, hyplist, detector_diagram)
struct feature *detector;
int detector_result;
struct hypothesis *hyplist;
struct network *detector_diagram;
{
   float sum, pi, belief, Get_Prob();
   struct hypothesis *hyp;
   struct node *detector_node, *Detector_Node();
   struct selector *sel, *Select();

   /*------------------*/
   /*  Error checking  */
   /*------------------*/
   if (detector == NULL) printf("ERROR: Detector_Prob passed NULL detector\n");
   if (hyplist == NULL)  printf("ERROR: Detector_Prob passed NULL hypothesis list\n");
   if (detector_diagram == NULL)  printf("ERROR: Detector_Prob passed NULL diagram\n");

   /*--------------------*/
   /*  Prepare for loop  */
   /*--------------------*/
   detector_node = Detector_Node(detector_diagram, detector);
   hyp = hyplist;
   sum = 0.0;

   /*-------------------------------------------------------------------------*/
   /*  Loop thru all hypotheses in the hypothesis list to calculate           */
   /*  Pr{detector, result} = SUM [ Pr{detector, result | LDP} * Pr{LDP} ]    */
   /*-------------------------------------------------------------------------*/
   while (hyp != NULL) {

      /*-------------------------------------------------------------*/
      /*  Get Pr{detector, result | LDP} directly from detector node  */
      /*-------------------------------------------------------------*/
      sel = Select(detector_result, hyp->LDP_type, -1);
      pi  = Get_Prob(detector_node->header, sel);
      Free_Selector(sel);

      /*--------------------------------------*/
      /*  Get Pr{LDP} from belief distribution  */
      /*--------------------------------------*/
      belief = hyp->certainty;

      sum = sum + belief * pi;
      hyp = hyp->next_hypothesis; }
```

```
#ifdef PRINT_INDEPTH
      printf("\tPR{%d @ %g = %d} = %g",
             detector->feature_type, detector->conceptual_angle,
             detector_result, sum);
#endif

   return(sum);
 }



/*-------------------------------------------------------------------------------------------*/
/*  Function COST                                                                            */
/*                                                                                           */
/*  This function returns a cost function for the given feature detector, based on the       */
/*  complexity and latency ratings associated with the detector.                             */
/*-------------------------------------------------------------------------------------------*/
float Cost(detector)
struct feature *detector;
{
   /*-----------------------------------------------*/
   /*  Scale complexity and latency in same range.  */
   /*  Also weight complexity more than latency.    */
   /*-----------------------------------------------*/
   static float kcomplex = 1.0;
   static float klatency = 0.5;
   float cost;

   /*------------------*/
   /*  Error checking  */
   /*------------------*/
   if (detector == NULL) printf("ERROR: Cost passed null detector\n");

   /*-------------------------------------------------------------------------------*/
   /*  Calculate a cost function from the detector complexity and latency ratings   */
   /*-------------------------------------------------------------------------------*/
   cost = kcomplex * detector->complexity + klatency * detector->latency;

   return(cost);
 }


/*-------------------------------------------------------------------------------------------*/
/*  Function COMPLEXITY                                                                      */
/*                                                                                           */
/*  This function returns a complexity measure for the given feature detector.  This        */
/*  value is a measure of the time or distance complexity of the detector and is constant    */
/*  for each detector type.                                                                  */
/*                                                                                           */
/*  The current measures are the sum of the total number of centimeters translated and       */
/*  degrees rotated.                                                                         */
/*-------------------------------------------------------------------------------------------*/
float Complexity(detector_type)
int detector_type;
{
   if (detector_type == CONVEX_CORNER)  return(1213.0);
   if (detector_type == CONCAVE_CORNER) return(1126.0);
   if (detector_type == WALL)           return(831.0);
   if (detector_type == EDGE)           return(2046.0);
   return(100000.0);
 }


/*-------------------------------------------------------------------------------------------*/
/*  Function LATENCY                                                                         */
/*                                                                                           */
/*  This function calculates a latency measure for the given detector.  This value is       */
/*  currently a measure of the distance needed to reach the starting point for the           */
/*  detector.  This value will differ across detectors depending on which hypothesis         */
/*  each is for.                                                                             */
/*-------------------------------------------------------------------------------------------*/
float Latency(detector)
struct feature *detector;
{
   float x, y, d;
   double sqrt();

   x = detector->pos.x;
   y = detector->pos.y;

   /* d = (float) sqrt((double) (x * x + y * y)); */
   d = (float) abs((int) x) + abs((int) y);

   return(d);
 }
```

```
#include <stdio.h>
#include "/pro/ai/robot/software/huey/utils/src/common.h"
#include "/pro/ai/robot/software/huey/utils/src/utils.h"
#include "/u/rac/Masters/Ldp/src/ldp.h"
#include "/u/rac/Masters/Bride/src/bride.h"


#define DELAYSEC  5


/*-------------------------------------------------------------------------------*/
/*  Procedure MOVE_TO_PORT                                                       */
/*                                                                               */
/*  This procedure causes the robot to move to the specified port in the LDP.  It  */
/*  attempts to retry in case the movement aborts or otherwise fails.            */
/*-------------------------------------------------------------------------------*/
Move_To_Port(portptr)
struct port *portptr;
{
   MoveResult Attempt_Move_To_Port();

   while (1) {
      if (Attempt_Move_To_Port(portptr) == MOVE_OK) return;
      printf("Attempted Move_To_Port failed. Retrying...\n"); }

 }


/*-------------------------------------------------------------------------------------*/
/*  Function ATTEMPT_MOVE_TO_PORT                                                      */
/*                                                                                     */
/*  This function causes the robot to move to the specified port in the LDP by sending    */
/*  a message to the FRM to do so.  The results of the movement are returned as the       */
/*  result of the function.                                                            */
/*-------------------------------------------------------------------------------------*/
MoveResult Attempt_Move_To_Port(portptr)
struct port *portptr;
{
   FRM_REQUEST sendmsg;
   FRM_RESPONSE *response, *FRM_Operation();
   MoveResult retval;

   /*-----------------------------------------------------*/
   /*  Initialize the request to the FRM to go to the port  */
   /*-----------------------------------------------------*/
   sendmsg.type     = FRM_PORT;
   sendmsg.pos.x    = portptr->pos.x;
   sendmsg.pos.y    = portptr->pos.y;
   sendmsg.theta    = portptr->theta;
   sendmsg.strategy = portptr->move_strategy;

   /*---------------------------------*/
   /*  Get the response from the FRM  */
   /*---------------------------------*/
   response = FRM_Operation(sendmsg);

   if (response->result == FRM_FEATURE_FOUND) retval = MOVE_OK;
   else retval = MOVE_FAILED;

   /*---------------------------------*/
   /*  Free the response and return  */
   /*---------------------------------*/
   free(response);
   return(retval);
 }




/*-------------------------------------------------------------------------------*/
/*  Function FRM_OPERATION                                                       */
/*                                                                               */
/*  This function is a general purpose method for sending a request to the FRM.  It  */
/*  assumes that the given request buffer has been preformatted and is ready to be sent.  */
/*  The result of the operation is returned as the result of the function.       */
/*-------------------------------------------------------------------------------*/
FRM_RESPONSE *FRM_Operation(sendmsg)
FRM_REQUEST sendmsg;
{
   FRM_RESPONSE *recvmsg;
   clientId FRMid, NSgetClient(), IPCrecvMessage();
   int recvlen, result;
   static int timeout = 200;
   char *malloc();

   /*------------------------*/
```

```
   /*  Get client id of FRM  */
   /*------------------------*/
   while ((FRMid = NSgetClient(FRM_NAME)) == nullClient);

   /*--------------------------------*/
   /*  Send the FRM request message  */
   /*--------------------------------*/
   IPCsendMessage(&sendmsg, sizeof(FRM_REQUEST), FRMid);
   printf("Awaiting result from FRM...\n");

   /*-----------------------*/
   /*  Get result from FRM  */
   /*-----------------------*/
   recvlen = sizeof(FRM_RESPONSE);
   recvmsg = (FRM_RESPONSE *) malloc(recvlen);

   while (FRMid != IPCrecvMessage(recvmsg, &recvlen, timeout))
      recvlen = sizeof(FRM_RESPONSE);

   /*----------------------*/
   /*  Return the result  */
   /*----------------------*/
   printf("FRM returns message %d\n", recvmsg->result);
   return(recvmsg);
 }


/*-------------------------------------------------------------------------------------*/
/*  Function FOLLOW_CORRIDOR                                                           */
/*                                                                                     */
/*  This function causes the robot to follow a corridor until an LDP is detected.  It  */
/*  does this by sending a message to the CFM requesting this.  The result returned    */
/*  from the CFM is returned as the result of the function.                            */
/*-------------------------------------------------------------------------------------*/
int Follow_Corridor()
{
   CFMmsg sendmsg, recvmsg;
   clientId CFMid, NSgetClient(), IPCrecvMessage();
   int recvlen, result;
   static int timeout = 200;

   /*-----------------------------------------------*/
   /*  Get client id of corridor-following process  */
   /*-----------------------------------------------*/
   while ((CFMid = NSgetClient(CFM_NAME)) == nullClient);

   /*---------------------------------------*/
   /*  Tell CFM to start following corridor  */
   /*---------------------------------------*/
   sendmsg.type = typeCFMcorridor;
   IPCsendMessage(&sendmsg, sizeof(CFMmsg), CFMid);

   printf("Awaiting result from CFM...\n");

   /*-------------------------------------*/
   /*  Get result of corridor-following  */
   /*-------------------------------------*/
   recvlen = sizeof(CFMmsg);
   while (CFMid != IPCrecvMessage(&recvmsg, &recvlen, timeout))
      recvlen = sizeof(CFMmsg);

   /*-------------------------------------------*/
   /*  Map received results to internal states  */
   /*-------------------------------------------*/
   if      (recvmsg.returnMsg == CFM_OPEN)     result = FEATURE_FOUND;
   else if (recvmsg.returnMsg == CFM_DEADEND)  result = FEATURE_FOUND;
   else if (recvmsg.returnMsg == CFM_FAILED)   result = DETECTOR_ABORTED;
   else if (recvmsg.returnMsg == CFM_LOST)     result = DETECTOR_ABORTED;

   else printf("ERROR: Follow_Corridor received invalid response from CFM: %d\n",
               recvmsg.returnMsg);

   /*----------------------*/
   /*  Return the result  */
   /*----------------------*/
   printf("CFM returns message %d\n", recvmsg.returnMsg);
   return(result);
 }


/*-------------------------------------------------------------------------*/
/*  Procedure CLEAR_WAI                                                    */
/*                                                                         */
/*  This procedure causes the WAI module to clear its memory by sending it  */
/*  the appropriate request.                                               */
/*-------------------------------------------------------------------------*/
Clear_WAI()
{
```

```
   static int timeout = 200;
   int recvlen, sendreturn;
   clientId WAIid, NSgetClient(), IPCrecvMessage();
   WAIrequestMsg sendmsg;
   WAIreplyMsg recvmsg;

   /*-------------------------------------*/
   /*  Get client id of whereami process  */
   /*-------------------------------------*/
   while ((WAIid = NSgetClient(WAI_NAME)) == nullClient) printf("WAITING ON WAI\n");

   /*-------------------------------------*/
   /*  Loop until OK message is returned  */
   /*-------------------------------------*/
   while (1) {

      /*---------------------------------*/
      /*  Send request to save position  */
      /*---------------------------------*/
      sendmsg.type = typeFHMclearMemory;
      sendreturn = IPCsendMessage(&sendmsg, sizeof(WAIrequestMsg), WAIid);

      /*-------------------------------------*/
      /*  Get response back from whereami    */
      /*-------------------------------------*/
      recvlen = sizeof(WAIreplyMsg);
      while (WAIid != IPCrecvMessage(&recvmsg, &recvlen, timeout)) {
         printf("Waiting for response from WAI\n");
         recvlen = sizeof(WAIreplyMsg); }

      /*-----------------------------------------*/
      /*  Check response and return data if valid  */
      /*-----------------------------------------*/
      if (recvmsg.type == typeFHMclearMemory) {

         if (recvmsg.errorMsg == SORRY_RETRY_LATER) {
            printf("Clear_WAI cannot access WAI. Will retry...\n");
            sleep(DELAYSEC); }

         else {
            printf("Clear_WAI received response %d from WAI\n", recvmsg.errorMsg);
            return;} }

      /*-------------------------------------*/
      /*  Wrong message type received. Abort  */
      /*-------------------------------------*/
      else {
         printf("ERROR: Clear_WAI received message type %d from WAI; expecting type %d\n",
                recvmsg.type, typeFHMclearMemory);
         return; } }
}
```

Listing for BRIDE: The Brown Influence Diagram Evaluator

Bride is tentatively the Brown Influence Diagram Evaluator. This is a general-purpose evaluator of influence diagrams, written in C. It is similar in mission to IDEAL, but only uses the influence diagram evaluation algorithms.

The evaluator itself resides in the src directory, and consists of the following modules:

bride.h	header file for bride
influence.c	functions for influence diagram evaluation
distribs.c	functions to update probability distributions during evaluation
matrix.c	functions supporting a sparse representation for probability distributions

```
/*------------------------*/
/*  Define diagram types  */
/*------------------------*/
#define  CONTROL   201
#define  INFERENCE 202

/*---------------------*/
/*  Define node types  */
/*---------------------*/
#define  STATE    301
#define  DECISION 302
#define  VALUE    303
#define  ANY      304

/*-------------------*/
/*  Some constants  */
/*-------------------*/
#define  VALUESTATES 1
#define  MAXNODES    20
#define  UNION       0
#define  REGULAR     1

/*------------*/
/*  Typedefs  */
/*------------*/
typedef  int       BOOLEAN;
typedef  int      NODETYPE;
typedef  int       NETTYPE;


/*----------------------------------------*/
/*  Define node probability distribution  */
/*----------------------------------------*/
struct distribution {
   float probability;                          /* Value of this distribution element  */
   int   index;                                /* Index of element in the current dimension */
   int   dimension;                            /* Current dimension */

   struct distribution *Next_Vector_Element;   /* Points to next element this dimension */
   struct distribution *Next_Dimension;        /* Points to first elt in next dimension */
 };                                            /* ... off of this element               */

#define DISTRIBSIZE sizeof(struct distribution)


/*----------------------------*/
/*  Define format of a node   */
/*----------------------------*/
struct node {
   NODETYPE type;                       /* STATE, DECISION, or VALUE */
   int      index;                      /* Unique network index */

   struct distribution *header;         /* Probability distribution header */

   BOOLEAN retain;                      /* Flags unremovable node */
   BOOLEAN removed;                     /* Flags removed node */
   BOOLEAN goalnode;                    /* Flags "goal" node */

   int successors;                      /* Number successors */
   int predecessors;                    /* Number predecessors */

   struct node *lastnode;               /* Marks return path for search backtrack */
   struct node *lastsucc;               /* Marks last arc followed in search */

   int knownstate;                      /* Instantiated state of node (if known) */
   int nstates;                         /* Number possible states */
   char *statenames;                    /* Names for those states */

   struct node *Next_dstar;             /* DECISION: Links decision nodes in d* */
   struct distribution *EV;             /* DECISION: Expected value for removed decision node */
   int decision_preds [MAXNODES] [MAXNODES];    /* DECISION: Saved connectivity matrix  */

   struct node *NextRemove;             /* Links nodes to be removed */
   struct network *Diagram;             /* Backpointer to source network */
 };


/*---------------------------------------*/
/*  Define format of an influence diagram  */
/*---------------------------------------*/
struct network {
   NETTYPE type;                          /* INFERENCE or CONTROL */
   int     lastnode;                      /* Index of highest node in the diagram */

   struct  node nodelist[MAXNODES];       /* Array of nodes in the diagram */
   int     matrix [MAXNODES] [MAXNODES];  /* Connectivity matrix of nodes  */
```

```
   struct node *dstar;                    /* Points to list of decision nodes after decision */
   struct node *lastdstar;                /* Points to last decision node in dstar list */

   struct node *goalnode;                 /* Pts to INFERENCE goal node or CONTROL value node */
   struct node *RemoveList;               /* Points to list of nodes to be removed ... */
 };                                       /*   during inference or decision           */

#define DIAGSIZE sizeof(struct network)


/*----------------------------------------*/
/*  Define distribution element selector  */
/*----------------------------------------*/
struct selector {
   int    state;                          /*  Selects value of state for the current dimension */
   int    omitflag;                       /*  Flags a selector elt to be omitted from list     */

   struct node *ofnode;                   /*  Points back to node defining this dimension  */

   struct selector *Next;                 /*  Next element in "regular" list  */
   struct selector *Next_Union;           /*  Next element in "union" list    */
   struct selector *duplicate;            /*  Ptr to selector elt that should mirror this elt  */
 };

#define SELSIZE sizeof(struct selector)


/*--------------------------------------*/
/*  Define format of a node set element */
/*--------------------------------------*/
struct setelt {
   struct node *elt;                      /*  Ptr to one node in the set  */
   struct setelt *Next;                   /*  Next element in the set     */
 };

#define ELTSIZE sizeof(struct setelt)


/*--------------------------------------*/
/*  Define an aggregate distribution    */
/*--------------------------------------*/
struct aggregate {
   struct distribution *vutil;            /*  Ptr to expected utility distribution  */
   struct distribution *dpolicy;          /*  Ptr to decision policy  */
 };

#define AGGSIZE sizeof(struct aggregate)
```

```
   Free_Selector(predY);
   Free_Selector(xsel);
   Free_Selector(ysel);

   return (newx);
}


/*-------------------------------------------------------------------------------------*/
/*  Function DISTRIB_AFTER_REMOVE                                                      */
/*                                                                                     */
/*  This function calculates the new probability distribution of the given node y      */
/*  assuming the arc from x to y will be removed.  This distribution is calculated as  */
/*  follows:                                                                           */
/*                                                                                     */
/*  Pr{y|Pred(x),Pred(y)} = SUM [ Pr{y|x,Pred(y)} * Pr{x|Pred(x)} ]   if y a STATE node */
/*                           x                                                         */
/*                                                                                     */
/*  Util{Pred(x),Pred(y)} = SUM [ Util{x,Pred(y)} * Pr{x|Pred(x)} ]   if y a VALUE node */
/*                           x                                                         */
/*                                                                                     */
/*  where Pred(x) is the set of predecessors of node x                                 */
/*    and Pred(y) is the set of predecessors of node y, except x                       */
/*-------------------------------------------------------------------------------------*/
struct distribution *Distrib_After_Remove(x, y)
struct node *x, *y;
{
   struct distribution *newy, *New_Distribution();
   struct selector *predX, *predY, *predXY, *s1, *s2, *s3, *in, *xsel, *ysel, *Union;
   struct selector *Select_Predecessors(), *Appended_Copy(), *NewSel(), *Select();
   struct selector *Purge_Flagged(), *Inserted_Copy(), *Union_Merge(), *Next_Copy();
   BOOLEAN oflag, Odometer();
   int statey, statex;
   float sum, p2, p3, Get_Prob();

   /*------------------*/
   /*  Error checking  */
   /*------------------*/
   if (x == NULL) { printf("ERROR: Distrib_After_Remove first node null\n");  return(NULL); }
   if (y == NULL) { printf("ERROR: Distrib_After_Remove second node null\n"); return(NULL); }

   if (x->type != STATE) { printf("ERROR: Distrib_After_Remove first node is not a state node\n");
                           return(NULL); }

   if (y->type == DECISION) { printf("ERROR: Distrib_After_Remove second node is a decision node\n");
                              return(NULL); }

   /*-----------------------------------------------------------------*/
   /*  Get a new distribution and build initial selector structures  */
   /*-----------------------------------------------------------------*/
   newy = New_Distribution();

   predX = Select_Predecessors(x, NULL);
   predY = Purge_Flagged( Select_Predecessors(y, x));

   xsel = NewSel(x, 0);
   ysel = NewSel(y, 0);

   oflag = 0;

   /*-------------------------------*/
   /*  Union the predecessor sets  */
   /*-------------------------------*/
   Union = Union_Merge(predX, predY);


#ifdef PRINT_REMOVE
   printf("\n\nREMOVAL CALCULATION\n");
#endif

   /*-------------------------------------------------------------*/
   /*  Loop thru all possible values of x's & y's predecessors  */
   /*-------------------------------------------------------------*/
   Zero(Union, UNION);
   while (Odometer(Union, UNION, &oflag)) {

#ifdef PRINT_REMOVE
   show(0, 'U', Union, UNION, 0.0);
#endif
      /*--------------------------------------*/
      /*  Loop thru all possible states of y  */
      /*--------------------------------------*/
      for (statey = 0; statey < y->nstates; statey++) {

#ifdef PRINT_REMOVE
         printf("\tState of y is: %d\n", statey);
#endif
         ysel->state = statey;
```

```
#include <stdio.h>
#include <varargs.h>
#include "/u/rac/Masters/Bride/src/bride.h"

/*****************************************************************************************/
/***                                                                                   ***/
/***  PROBABILITY DISTRIBUTION REPRESENTATION                                          ***/
/***                                                                                   ***/
/***  This set of functions provides support for multidimensional probability distributions  ***/
/***  associated with influence diagram nodes.  The distributions are represented as sparse  ***/
/***  matrices.                                                                        ***/
/*****************************************************************************************/

/* #define PRINT_DISTRIB 1 */

/*---------------------------------------------------------------------------------------*/
/*  Function SET_NODE_PROB                                                               */
/*                                                                                       */
/*  This procedure is an interface to procedure Set_Prob that requires a pointer to an   */
/*  influence diagram and a node index, rather than a pointer to a distribution.  An     */
/*  element in the specified distribution is set to the given value.                     */
/*---------------------------------------------------------------------------------------*/
Set_Node_Prob(diagram, index, selector, value)
struct network *diagram;
int index;
struct selector *selector;
float  value;
{
   Set_Prob(diagram->nodelist[index].header, selector, value); }



/*---------------------------------------------------------------------------------------*/
/*  Function SET_PROB                                                                    */
/*                                                                                       */
/*  This function sets an element of the given distribution to the given value. This     */
/*  function is similar in structure to GET_PROB. Which element to get is defined by     */
/*  the given selector structure as follows:                                             */
/*                                                                                       */
/*  Each element in the selector list represents one dimension of the distribution.      */
/*  The index for that particular dimension is contained in the selector element.        */
/*  In this way, distributions may be of arbitrary dimensionality.                       */
/*---------------------------------------------------------------------------------------*/
Set_Prob(distrib, selector, value)
struct distribution *distrib;
struct selector *selector;
float  value;
{
    struct distribution *vector_elt, *dimtop, *Index_Sparse_Vector(), *Add_Dimtop();
    struct selector *s;
    int i;

    /*------------------*/
    /*  Error checking  */
    /*------------------*/
    if (selector == NULL)
       {printf("ERROR: Set_Prob passed null selector\n"); return; }

#ifdef PRINT_DISTRIB
    printf ("\t\t\t\tSetting prob ");
#endif

    /*------------------------*/
    /*  Initialize for loop  */
    /*------------------------*/
    vector_elt = distrib;               /* Point to header node */
    dimtop = vector_elt->Next_Dimension;

    s = selector;
    i = s->state;

    /*-----------------------------------------------------------------*/
    /*  Loop thru selector elements (each represents one dimension).   */
    /*  At each dimension, index into the vector based on the state field  */
    /*  in the current selector element.                               */
    /*-----------------------------------------------------------------*/
    while (s != NULL) {

#ifdef PRINT_DISTRIB
       printf("%d, ",i);
#endif

       if (dimtop == NULL) dimtop = Add_Dimtop(i, vector_elt->dimension + 1, vector_elt);

       vector_elt = Index_Sparse_Vector(dimtop, i, vector_elt);
       dimtop = vector_elt->Next_Dimension;
```

```
        s = s->Next;
        if (s != NULL) i = s->state;
        }

    /*--------------------------------------*/
    /*  Set the probability in this element  */
    /*--------------------------------------*/

#ifdef PRINT_DISTRIB
    printf(" to value %g\n", value);
#endif

    vector_elt->probability = value;
    return;
  }


/*-------------------------------------------------------------------------------------*/
/*  Function GET_PROB                                                                  */
/*                                                                                     */
/*  This function gets the probability value of an element in the given distribution.  */
/*  This function is similar in structure to SET_PROB. Which element to set is         */
/*  defined by the given selector structure as follows:                                */
/*                                                                                     */
/*  Each element in the selector list represents one dimension of the distribution.    */
/*  The index for that particular dimension is contained in the selector element.      */
/*  In this way, distributions may be of arbitrary dimensionality.                     */
/*-------------------------------------------------------------------------------------*/
float Get_Prob(distrib, selector)
struct distribution *distrib;
struct selector *selector;
{
    struct distribution *vector_elt, *dimtop, *Index_Sparse_Vector(), *Add_Dimtop();
    struct selector *s;
    int i;

    /*------------------*/
    /*  Error checking  */
    /*------------------*/
    if (selector == NULL) printf("ERROR: Get_Prob passed null selector\n");

    /*------------------------*/
    /*  Initialize for loop  */
    /*------------------------*/
    vector_elt = distrib;                /* Point to header node */
    dimtop = vector_elt->Next_Dimension;

    s = selector;
    i = s->state;

    /*-------------------------------------------------------------------------*/
    /*  Loop thru selector elements (each represents one dimension).           */
    /*  At each dimension, index into the vector based on the state field      */
    /*  in the current selector element.                                       */
    /*-------------------------------------------------------------------------*/
    while (s != NULL) {
       if (dimtop == NULL) return(0.0);

       vector_elt = Index_Sparse_Vector(dimtop, i, vector_elt);
       dimtop = vector_elt->Next_Dimension;

       s = s->Next;
       if (s != NULL) i = s->state;
       }

    /*-----------------------------------------*/
    /*  Return the probability in this element  */
    /*-----------------------------------------*/
    return(vector_elt->probability);
  }


/*-------------------------------------------------------------------------------*/
/*  Function SELECT                                                              */
/*                                                                               */
/*  This function is a utility to build a selector structure for accessing       */
/*  a distribution. The function takes a variable number of arguments in the     */
/*  the following calling sequence: <index1> ... <indexn> <negative #>           */
/*                                                                               */
/*  Each <index> selects in the distribution along its own dimension. The indices */
/*  are ended by a negative number.                                              */
/*-------------------------------------------------------------------------------*/
struct selector *Select(va_alist)
va_dcl
{
   va_list pvar;
   char * malloc();
   int arg;
```

```
   struct selector *front, *rear, *s;

   va_start(pvar);

   /*-------------------*/
   /*  Get first index  */
   /*-------------------*/
   arg = va_arg(pvar, int);
   if (arg < 0) { printf("ERROR: No selection indices for Select\n"); return(NULL); }

   /*-----------------------------------------------------------*/
   /*  Loop thru all arguments, building a selector structure  */
   /*-----------------------------------------------------------*/
   front = rear = NULL;

   while (arg >= 0) {
      s = (struct selector *) malloc(SELSIZE);
      s->state      = arg;
      s->omitflag   = 0;
      s->ofnode     = NULL;
      s->Next       = NULL;
      s->Next_Union = NULL;
      s->duplicate  = NULL;

      /*---------------------------------------*/
      /*  Attach this selector to end of list  */
      /*---------------------------------------*/
      if (front == NULL) front = s;
      else rear->Next = s;
      rear = s;

      /*-----------------*/
      /*  Get next arg   */
      /*-----------------*/
      arg = va_arg(pvar, int);
      }

   /*-------------*/
   /*  Finish up  */
   /*-------------*/
   va_end(pvar);
   return(front);
 }


/*-------------------------------------------------------------------------------*/
/*  Function ADD_DIMTOP                                                          */
/*                                                                               */
/*  This function creates a new distribution element representing the first      */
/*  entry in a new dimension.  The element is attached to its source element     */
/*  by means of the Next_Dimension pointer.                                      */
/*-------------------------------------------------------------------------------*/
struct distribution *Add_Dimtop(i, dimension, origin)
int i, dimension;
struct distribution *origin;
{
   struct distribution *new;
   char *malloc();

   /*-------------------*/
   /*  Error checking   */
   /*-------------------*/
   if (origin == NULL) { printf("ERROR: Add_Dimtop passed null origin\n"); return(NULL); }
   if (dimension < 0)  { printf("ERROR: Add_Dimtop passed negative dimension\n"); return(NULL); }
   if (i < 0)          { printf("ERROR: Add_Dimtop passed negative index\n"); return(NULL); }

   /*-----------------------------------------------------------*/
   /*  Allocate the new distribution node and initialize it     */
   /*-----------------------------------------------------------*/
   new = (struct distribution *) malloc(DISTRIBSIZE);

   new->probability         = 0.0;
   new->index               = i;
   new->dimension           = dimension;
   new->Next_Vector_Element = NULL;
   new->Next_Dimension      = NULL;

   /*-----------------------------------------------*/
   /*  Attach new distribution node to its origin   */
   /*-----------------------------------------------*/
   origin->Next_Dimension   = new;

   return(new);
 }


/*-------------------------------------------------------------------------------*/
/*  Function INDEX_SPARSE_VECTOR                                                 */
```

```
/*                                                                          */
/*  This function returns the ith vector element in the given vector v.     */
/*  The vector is represented as a sparse vector, with elements allocated only  */
/*  for relevant elements (usually those that have nonzero probabilities).  */
/*  If an element already exists representing index i, that element is returned.  */
/*  If no element exists representing i, a new one is allocated and returned  */
/*  (however, it will have a zero probability).                             */
/*--------------------------------------------------------------------------*/
struct distribution *Index_Sparse_Vector(v, i, origin)
struct distribution *v, *origin;
int i;
{
   struct distribution *elt, *last, *Add_Vector_Element();

   /*------------------*/
   /*  Error checking  */
   /*------------------*/
   if (origin == NULL) {printf("ERROR: Index_Sparse_Vector passed null origin\n");    return(NULL);}
   if (v == NULL)      {printf("ERROR: Index_Sparse_Vector passed null vector\n");    return(NULL);}
   if (i < 0)          {printf("ERROR: Index_Sparse_Vector passed negative index\n"); return(NULL);}

   last = NULL;
   elt = v;

   /*----------------------------------------------------------*/
   /*  Search for ith element in this dimension.  If run off  */
   /*  the end of the vector, add a new element at the end.   */
   /*----------------------------------------------------------*/
   while (elt->index < i)  {
      last = elt;
      elt = elt->Next_Vector_Element;

      if (elt == NULL) return(Add_Vector_Element(i, last, NULL, v->dimension, origin));
      }

   /*--------------------------------------------------------------------*/
   /*  Return the element if found. If not found, create a new element  */
   /*  and place it just before element elt.                             */
   /*--------------------------------------------------------------------*/
   if (elt->index == i)  return(elt);
   else return(Add_Vector_Element(i, last, elt, v->dimension, origin));
 }



/*--------------------------------------------------------------------------*/
/*  Function ADD_VECTOR_ELEMENT                                             */
/*                                                                          */
/*  This function creates a new distribution element with index i,          */
/*  and enters it into the current dimension vector in the proper location.  */
/*  This location is between elements last and next if in the middle of the  */
/*  vector, and after element origin if at the beginning of the vector.     */
/*--------------------------------------------------------------------------*/
struct distribution *Add_Vector_Element(i, last, next, dimension, origin)
int i, dimension;
struct distribution *last, *next, *origin;
{
   struct distribution *new;
   char *malloc();

   /*------------------*/
   /*  Error checking  */
   /*------------------*/
   if (origin == NULL) {printf("ERROR: Add_Vector_Element passed null origin\n");        return(NULL);}
   if (dimension < 0)  {printf("ERROR: Add_Vector_Element passed negative dimension\n"); return(NULL);}
   if (i < 0)          {printf("ERROR: Add_Vector_Element passed negative index\n");     return(NULL);}

   /*----------------------------------------------------------*/
   /*  Allocate the new distribution node and initialize it  */
   /*----------------------------------------------------------*/
   new = (struct distribution *) malloc(DISTRIBSIZE);

   new->probability         = 0.0;
   new->index               = i;
   new->dimension           = dimension;
   new->Next_Vector_Element = next;
   new->Next_Dimension      = NULL;

   /*------------------------------------------------*/
   /*  Attach new node to the proper place in the chain  */
   /*------------------------------------------------*/
   if (last != NULL)  last->Next_Vector_Element = new;
   else origin->Next_Dimension = new;

   return(new);
 }
```

```
/*****************************************************************************/
/***  DISTRIBUTION OPERATIONS                                              ***/
/***                                                                       ***/
/***  This set of functions provide various operations on node distributions  ***/
/*****************************************************************************/

/*-------------------------------------------------------------------------*/
/*  Function NEW_DISTRIBUTION                                              */
/*                                                                         */
/*  This function returns a distribution structure representing the header */
/*  of an entire distribution for a node.                                  */
/*-------------------------------------------------------------------------*/
struct distribution *New_Distribution()
{
   struct distribution *d;
   char *malloc();

   d = (struct distribution *) malloc(DISTRIBSIZE);
   d->probability         = 0.0;
   d->index               = -1;
   d->dimension           = -1;
   d->Next_Vector_Element = NULL;
   d->Next_Dimension      = NULL;

   return(d);
 }


/*-----------------------------------------------------------------------------*/
/*  Function FREE_DISTRIBUTION                                                 */
/*                                                                             */
/*  This procedure frees an entire node's distribution by first freeing the    */
/*  vector originating from the header node, and then freeing the header itself.  */
/*-----------------------------------------------------------------------------*/
Free_Distribution(n)
struct node *n;
{
   Free_Vector(n->header);
   free(n->header);
   n->header = NULL;
 }


/*-------------------------------------------------------------------------*/
/*  Function FREE_VECTOR                                                   */
/*                                                                         */
/*  This procedure frees the vector that originates from the given origin  */
/*  distribution element (field NEXT_DIMENSION).  This procedure recursively  */
/*  frees any vectors that originate from this vector.                     */
/*-------------------------------------------------------------------------*/
Free_Vector(origin)
struct distribution *origin;
{
   struct distribution *vector, *temp;

   /*----------------------------*/
   /*  Prepare for freeing loop  */
   /*----------------------------*/
   if (origin == NULL) return;
   vector = origin->Next_Dimension;

   /*-------------------------------------------*/
   /*  Free each element in the vector in turn  */
   /*-------------------------------------------*/
   while (vector != NULL) {
      Free_Vector(vector);
      temp = vector->Next_Vector_Element;
      free(vector);

      vector = temp; }
 }



/*-------------------------------------------------------------------------*/
/*  Function COPY_DISTRIBUTION                                             */
/*                                                                         */
/*  This function makes a duplicate copy of the given distribution.        */
/*-------------------------------------------------------------------------*/
struct distribution *Copy_Distribution(distrib)
struct distribution *distrib;
{
   struct distribution *newdistrib, *New_Distribution(), *Copy_Vector();

   /*------------------*/
   /*  Error checking  */
   /*------------------*/
   if (distrib == NULL) {printf("ERROR: Copy_Distribution passed null distribution\n");
```

```
                        return(NULL); }

   /*--------------------------------------*/
   /*  Get and init a distribution header  */
   /*--------------------------------------*/
   newdistrib = New_Distribution();
   newdistrib->Next_Dimension = Copy_Vector(distrib);

   return(newdistrib);
}



/*-------------------------------------------------------------------------------------*/
/*  Function COPY_VECTOR                                                               */
/*                                                                                     */
/*  This function makes a duplicate copy of the probability distribution vector that   */
/*  is pointed to by the given originating distribution element.  Vectors originating  */
/*  from this vector represent new dimensions and are recursively copied.              */
/*-------------------------------------------------------------------------------------*/
struct distribution *Copy_Vector(origin)
struct distribution *origin;
{
   struct distribution *vectorelt, *first, *last, *newelement, *Copy_Vector();
   char *malloc();

   /*----------------*/
   /*  Trivial case  */
   /*----------------*/
   if (origin == NULL) return(NULL);

   /*--------------------------------------*/
   /*  Prepare for loop thru this vector  */
   /*--------------------------------------*/
   vectorelt = origin->Next_Dimension;
   first = last = NULL;

   /*-----------------------------------------*/
   /*  Loop thru vector, copying each element  */
   /*-----------------------------------------*/
   while (vectorelt != NULL) {
      newelement = (struct distribution *) malloc(DISTRIBSIZE);
      /* printf("Copying index %d dimension %d value %g\n",
            vectorelt->index, vectorelt->dimension, vectorelt->probability); */

      newelement->probability = vectorelt->probability;
      newelement->index       = vectorelt->index;
      newelement->dimension   = vectorelt->dimension;

      newelement->Next_Dimension = Copy_Vector(vectorelt);

      /*-----------------------------------*/
      /*  Link element into the new vector  */
      /*-----------------------------------*/
      newelement->Next_Vector_Element = NULL;

      if (first == NULL) first = last = newelement;
      else {
         last->Next_Vector_Element = newelement;
         last = newelement; }

      /*---------------------------------*/
      /*  Bump to next element to copy  */
      /*---------------------------------*/
      vectorelt = vectorelt->Next_Vector_Element; }

   return(first);
}
```

```
        /*-------------------------------------*/
        /*  Loop thru all possible states of x  */
        /*-------------------------------------*/
        sum = 0;

        for (statex = 0; statex < x->nstates; statex++) {

#ifdef PRINT_REMOVE
           printf("\t\tState of x is: %d\n", statex);
#endif
           xsel->state = statex;

           /*-----------------------------------------------*/
           /*  If y a state node, get Prob{y | x, Pred(y)}.  */
           /*  If y a value node, get U(x, Pred(y))          */
           /*-----------------------------------------------*/
           s2 = in = Inserted_Copy(xsel, predY);
           if (y->type == STATE) s2 = Appended_Copy(ysel, in);

           p2 = Get_Prob(y->header, s2);

#ifdef PRINT_REMOVE
           show(2, '2', s2, REGULAR, p2);
#endif
           /*-------------------------*/
           /*  Get Prob(x | Pred(x))  */
           /*-------------------------*/
           s3 = Appended_Copy(xsel, predX);
           p3 = Get_Prob(x->header, s3);

#ifdef PRINT_REMOVE
           show(2, '3', s3, REGULAR, p3);
#endif
           /*-------------------------------------------------------------------*/
           /*  Sum Prob{y | x, Pred(y)} * Prob{x | Pred(x)}, if y a state node  */
           /*  or     U(x, Pred(y))     * Prob{x | Pred(x)}, if y a value node  */
           /*-------------------------------------------------------------------*/
           sum += p2 * p3;

           Free_Selector(in);
           Free_Selector(s3);
           if (y->type == STATE) Free_Selector(s2); }

        /*-------------------------------------------------------------------*/
        /*  If y a state node, get selector for Prob{y | Pred(x), Pred(y)}  */
        /*  If y a value node, get selector for U(Pred(x), Pred(y))         */
        /*-------------------------------------------------------------------*/
        s1 = predXY = Next_Copy(Union);
        if (y->type == STATE) s1 = Appended_Copy(ysel, predXY);

        /*-------------------------------------------------------------------------*/
        /*  Special case: When removing last node before value node, and it is a state  */
        /*  node, must dummy in a selector since predX, predY, Union are all NULL.      */
        /*-------------------------------------------------------------------------*/
        if (y->type == VALUE  &&  s1 == NULL) s1 = Select(0, -1);

        /*-------------------------------------------------------------------------------/*
        /*  If y a state node, save the sum as new probability Prob{y | Pred(x), Pred(y)}  */
        /*  If y a value node, save the sum as new utility U(Pred(x), Pred(y))             */
        /*-------------------------------------------------------------------------------*/

#ifdef PRINT_REMOVE
        show(1, '1', s1, REGULAR, sum);
#endif
        if (sum != 0.0)
           Set_Prob(newy, s1, sum);

        Free_Selector(predXY);
        if (y->type == STATE) Free_Selector(s1); }}

   /*-------------------------------*/
   /*  Free the original selectors  */
   /*-------------------------------*/
   Free_Selector(predX);
   Free_Selector(predY);
   Free_Selector(xsel);
   Free_Selector(ysel);

   return (newy);
 }


/*-------------------------------------------------------------------------------*/
/*  Function MAXIMIZE_EXPECT_UTIL                                                */
/*                                                                               */
/*  This function calculates the new utility of the value node v assuming the arc  */
```

```
/*  from decision node d to v will be removed. This distribution is calculated as     */
/*  follows:                                                                          */
/*                                                                                    */
/*  Util(Pred(v)) = MAX [ Util(d,Pred(v)) ]                                           */
/*                   d                                                                */
/*                                                                                    */
/*  the optimal decision policy d* for node d is also determined by retaining that    */
/*  value of d producing each maximum expected utility:                               */
/*                                                                                    */
/*  d*(Pred(v)) = ARG MAX [ Util(d,Pred(v)) ]                                         */
/*                    d                                                               */
/*                                                                                    */
/*  where Pred(v) is the set of predecessors of value node v, except d                */
/*------------------------------------------------------------------------------------*/
struct aggregate *Maximize_Expect_Util(d, v)
struct node *d, *v;
{
   struct distribution *vdist, *dstar, *New_Distribution();
   struct aggregate *double_dist;
   struct selector *predV, *copypredV, *in, *dsel;
   struct selector *Select_Predecessors(), *NewSel();
   struct selector *Purge_Flagged(), *Inserted_Copy(), *Copy_Selector();
   BOOLEAN oflag, Odometer();
   int stated, best_decision;
   float maxutil, util, Get_Prob();
   char *malloc();

   /*------------------*/
   /*  Error checking  */
   /*------------------*/
   if (d == NULL) { printf("ERROR: Maximize_Expect_Util first node null\n");  return(NULL); }
   if (v == NULL) { printf("ERROR: Maximize_Expect_Util second node null\n"); return(NULL); }

   if (d->type != DECISION) { printf("ERROR: Maximize_Expect_Util first node is not a decision node\n");
                              return(NULL); }

   if (v->type != VALUE) { printf("ERROR: Maximize_Expect_Util second node is not a value node\n");
                           return(NULL); }

   /*-------------------------------------------------------------*/
   /*  Get a new distribution and build initial selector structures  */
   /*-------------------------------------------------------------*/
   vdist = New_Distribution();
   dstar = New_Distribution();

   predV = Purge_Flagged( Select_Predecessors(v, d));
   dsel = NewSel(d, 0);

   oflag = 0;

#ifdef PRINT_MAXIMIZE
   printf("\n\nMAXIMIZATION\n");
#endif

   /*-----------------------------------------------------*/
   /*  Loop thru all possible values of v's predecessors  */
   /*-----------------------------------------------------*/
   Zero(predV, REGULAR);
   while (Odometer(predV, REGULAR, &oflag)) {

#ifdef PRINT_MAXIMIZE
   show(0, 'V', predV, REGULAR, 0.0);
#endif

      /*-------------------------------------*/
      /*  Loop thru all possible states of d  */
      /*-------------------------------------*/
      maxutil = -1E20;
      best_decision = -1;

      for (stated = 0; stated < d->nstates; stated++) {

#ifdef PRINT_MAXIMIZE
         printf("\tState of d is: %d\n", stated);
#endif
         dsel->state = stated;

         /*-------------------------------------*/
         /*  Get the old utility U(d, pred(v))  */
         /*-------------------------------------*/
         in = Inserted_Copy(dsel, predV);
         util = Get_Prob(v->header, in);

#ifdef PRINT_MAXIMIZE
         show(2, 'U', in, REGULAR, util);
#endif

         /*------------------------------*/
```

```
        /*  Find maximum U(d, pred(v))  */
        /*-----------------------------*/
        if (util > maxutil) {
           maxutil = util;
           best_decision = stated; }

        Free_Selector(in); }

      /*-------------------------------------------------------*/
      /*  Special case: when predV is null, use selector of (0).  */
      /*  This is to match a similar case in Find_Best_Utility.   */
      /*-------------------------------------------------------*/
      if (predV == NULL) copypredV = Select(0, -1);
      else copypredV = Copy_Selector(predV);

      /*-------------------------------------------------------------------*/
      /*  Save the maximum utility in the value node new utility, and the  */
      /*  corresponding decision in the decision node table.               */
      /*-------------------------------------------------------------------*/
#ifdef PRINT_MAXIMIZE
      printf("%d ",best_decision); show(0, 'M', copypredV, REGULAR, maxutil);
#endif

      if (maxutil != 0.0)     Set_Prob(vdist, copypredV, maxutil);
      if (best_decision != 0) Set_Prob(dstar, copypredV, (float) best_decision);

      Free_Selector(copypredV); }

   /*-------------------------------*/
   /*  Free the original selectors  */
   /*-------------------------------*/
   Free_Selector(predV);
   Free_Selector(dsel);

   /*-----------------------------------------------------------------------*/
   /*  Create the aggregate structure containing the two distribution pointers  */
   /*-----------------------------------------------------------------------*/
   double_dist = (struct aggregate *) malloc(AGGSIZE);
   double_dist->vutil = vdist;
   double_dist->dpolicy = dstar;

   return (double_dist);
 }


/*************************************************************************/
/***  SELECTOR OPERATIONS                                              ***/
/***                                                                   ***/
/***  This set of functions provides various operations on selectors   ***/
/*************************************************************************/

/*-----------------------------------------------------------------------*/
/*  Function NEWSEL                                                      */
/*                                                                       */
/*  This function returns a new selector structure initialized using the  */
/*  input arguments.                                                     */
/*-----------------------------------------------------------------------*/
struct selector *NewSel(ofnodeptr, state)
struct node *ofnodeptr;
int state;
{
   struct selector *s;
   char *malloc();

   /*------------------*/
   /*  Error checking  */
   /*------------------*/
   if (ofnodeptr == NULL)
      printf("WARNING: NewSel passed null ofnode pointer\n");

   /*---------------------------------------*/
   /*  Create and init the selector element  */
   /*---------------------------------------*/
   s = (struct selector *) malloc(SELSIZE);
   s->state       = state;
   s->omitflag    = 0;
   s->ofnode      = ofnodeptr;
   s->Next        = NULL;
   s->Next_Union  = NULL;
   s->duplicate   = NULL;

   return(s);
 }


/*-----------------------------------------------------------------------*/
/*  Function ODOMETER                                                    */
/*                                                                       */
```

```
/*  This function is passed a union list whose elements are selecting      */
/*  various states.  Odometer alters those states such that the list as a  */
/*  whole is selecting a different distribution element.  It does this by  */
/*  simulating the action of an odometer. It increments the first element  */
/*  in the list, if that is past is maximum, it is reset, and the next     */
/*  selector element is incremented...                                     */
/*                                                                         */
/*  This function allows selecting in turn, each element in a multi-       */
/*  dimensional probability distribution.                                  */
/*-------------------------------------------------------------------------*/
BOOLEAN Odometer(ulist, which, oncethru)
struct selector *ulist;
BOOLEAN which, *oncethru;
{
   struct selector *ptr;

   /*---------------------------------------------------------------*/
   /*  Special case: When ulist is null, allow exactly 1 iteration.  */
   /*  Argument ONCETHRU used to keep track of this.                 */
   /*---------------------------------------------------------------*/
   if (ulist == NULL) {
      if (*oncethru) {
         *oncethru = 0;
         return(0); }

      else {
         *oncethru = 1;
         return(1); } }

   /*-----------------------------------------------------*/
   /*  Bump the state of the first selector in the list  */
   /*-----------------------------------------------------*/
   ptr = ulist;

   ptr->state++;
   if (ptr->duplicate != NULL) ptr->duplicate->state++;

   /*-------------------------------------------------------------------------*/
   /*  Percolate down list, bumping the state of selectors whose previous     */
   /*  previous one has reached its max.  Reset the max's to zero.  When      */
   /*  we percolate off the end of the list, the sequence is done, and we     */
   /*  should return this fact.  Mirror changes in any duplicated selectors.  */
   /*-------------------------------------------------------------------------*/
   while (ptr->state >= ptr->ofnode->nstates) {
      ptr->state = 0;
      if (ptr->duplicate != NULL) ptr->duplicate->state = 0;

      if (which == REGULAR) ptr = ptr->Next;
      else ptr = ptr->Next_Union;
      if (ptr == NULL) return(0);

      ptr->state++;
      if (ptr->duplicate != NULL) ptr->duplicate->state++; }

   /*---------------------------------------------------------------*/
   /*  We have not reached the end of the sequence. Return accordingly  */
   /*---------------------------------------------------------------*/
   return(1);
 }



/*-------------------------------------------------------------------------------*/
/*  Procedure SHOW                                                               */
/*                                                                               */
/*  This procedure prints all elements in the given selector or union list.      */
/*  The link to traverse (NEXT or NEXT_UNION) is determined by parameter WHICH.  */
/*-------------------------------------------------------------------------------*/
show(i, c, slist, which, val)
struct selector *slist;
char c;
int i;
BOOLEAN which;
float val;
{
   struct selector *s, *temp;
   int j;

   /*---------------------------*/
   /*  Prepare for freeing loop  */
   /*---------------------------*/
   if (slist == NULL) return;
   s = slist;

   /*-------------------------*/
   /*  Print header of tabs  */
   /*-------------------------*/
   for (j = 0; j < i; j++) printf("\t");
```

```
   printf("Selector list for %c: ", c);

   /*-----------------------------------------*/
   /*  Print each element in the vector in turn  */
   /*-----------------------------------------*/
   while (s != NULL) {
      printf("%d, ", s->state);

      if (which == REGULAR) s = s->Next;
      else s = s->Next_Union; }

   if (val != 0.0) printf(" = %g", val);
   printf("\n");
 }


/*-------------------------------------------------------------------------*/
/*  Procedure FREE_SELECTOR                                                */
/*                                                                         */
/*  This procedure frees all elements in the given selector list.          */
/*-------------------------------------------------------------------------*/
Free_Selector(slist)
struct selector *slist;
{
   struct selector *s, *temp;

   /*---------------------------*/
   /*  Prepare for freeing loop  */
   /*---------------------------*/
   if (slist == NULL) return;
   s = slist;

   /*-----------------------------------------*/
   /*  Free each element in the vector in turn  */
   /*-----------------------------------------*/
   while (s != NULL) {
      temp = s->Next;
      free(s);

      s = temp; }
 }


/*-------------------------------------------------------------------------*/
/*  Function SELECT_PREDECESSORS                                           */
/*                                                                         */
/*  This function builds a selector list for the given node.  This list    */
/*  contains one element for each predecessor of the node.  Additionally,  */
/*  the given flagnode is flagged for later omission (this is used when    */
/*  calculating new distributions after removing a node).                  */
/*-------------------------------------------------------------------------*/
struct selector *Select_Predecessors(nodeptr, flagnode)
struct node *nodeptr, *flagnode;
{
   struct node *pred, *Next_Predecessor();
   struct selector *front, *rear, *s, *NewSel();

   /*------------------*/
   /*  Error checking  */
   /*------------------*/
   if (nodeptr == NULL) { printf("ERROR: Select_Predecessors passed null node\n"); return(NULL); }

   /*--------------------------------*/
   /*  Set up for predecessor loop  */
   /*--------------------------------*/
   pred = Next_Predecessor(nodeptr, NULL, ANY);
   front = rear = NULL;

   /*-------------------------------------------------------*/
   /*  Loop thru predecessors of nodeptr, adding a selector  */
   /*  to the selector list for each.                        */
   /*-------------------------------------------------------*/
   while (pred != NULL) {
      s = NewSel(pred, 0);
      if (pred == flagnode) s->omitflag = 1;

      if (front == NULL) front = s;
      else rear->Next = s;
      rear = s;

      pred = Next_Predecessor(nodeptr, pred, ANY); }

   /*---------------------------------------*/
   /*  Return the constructed selector list  */
   /*---------------------------------------*/
   return(front);
 }
```

```
/*-------------------------------------------------------------------------*/
/*  Procedure ZERO                                                         */
/*                                                                         */
/*  This procedure sets the state field to zero for each selector element  */
/*  in the given union list.  The first element in the list is set to -1   */
/*  as a prelude to the first call of Odometer, which will set it to 0.    */
/*  Any duplicates of elements in the list are also zeroed.                */
/*-------------------------------------------------------------------------*/
Zero(ulist, which)
struct selector *ulist;
BOOLEAN which;
{
   struct selector *s;

   /*------------------*/
   /*  Error checking  */
   /*------------------*/
   if (ulist == NULL) return;

   /*-----------------------------------------------*/
   /*  Zero out all entries in the given union list  */
   /*-----------------------------------------------*/
   s = ulist;
   while (s != NULL) {
      s->state = 0;
      if (s->duplicate != NULL) s->duplicate->state = 0;

      if (which == REGULAR) s = s->Next;
      else s = s->Next_Union; }

   /*---------------------------------------------------------------*/
   /*  Set first entry to -1 in preparation for first Odometer call  */
   /*---------------------------------------------------------------*/
   ulist->state = -1;
   if (ulist->duplicate != NULL) ulist->duplicate->state = -1;
 }


/*-------------------------------------------------------------------------------*/
/*  Function UNION_MERGE                                                         */
/*                                                                               */
/*  This function takes two sorted selector lists (sorted by the OFNODE field    */
/*  and linked by the NEXT field), and merges them into a combined sorted union  */
/*  list (linked by the NEXT_UNION field).  This union list is made within the   */
/*  the existing selector lists; no copy is made.                                */
/*-------------------------------------------------------------------------------*/
struct selector *Union_Merge(slist1, slist2)
struct selector *slist1, *slist2;
{
   struct selector *front, *rear, *i, *j, *Union_Link();


   /*-------------------------*/
   /*  Handle trivial cases  */
   /*-------------------------*/
   if (slist1 == NULL) return( Union_Link(slist2) );
   if (slist2 == NULL) return( Union_Link(slist1) );

   /*---------------------------------------------------------------*/
   /*  Loop thru both selector lists, merging into a single list  */
   /*---------------------------------------------------------------*/
   front = rear = NULL;
   i = slist1;
   j = slist2;

   while (i != NULL  &&  j != NULL) {

      /*-------------------------------------------------------------------*/
      /*  Add selector i to the merged list, since it has the lowest index  */
      /*-------------------------------------------------------------------*/
      if (i->ofnode->index < j->ofnode->index) {
         if (front == NULL) front = rear = i;
         else {
            rear->Next_Union = i;
            rear = i; }

         i = i->Next; }

      else {
         /*---------------------------------------------------------------*/
         /*  If the indices are equal, do not include selector i in union  */
         /*---------------------------------------------------------------*/
         if (i->ofnode->index == j->ofnode->index) {
            j->duplicate = i;
            i = i->Next; }

         /*-------------------------------------*/
```

```
         /*  Add selector j to the merged list  */
         /*-------------------------------------*/
         if (front == NULL) front = rear = j;
         else {
            rear->Next_Union = j;
            rear = j; }

         j = j->Next; } }

   /*---------------------------------------------*/
   /*  Handle when one list shorter than the other  */
   /*---------------------------------------------*/
   if (i == NULL) rear->Next_Union = Union_Link(j);
   else rear->Next_Union = Union_Link(i);

   return(front);
   }


/*-------------------------------------------------------------------------------*/
/*  Function PURGE_FLAGGED                                                       */
/*                                                                               */
/*  This function removes elements flagged by omitflag in the given selector list  */
/*  This destructively removes the elements from the given list; no copy is made.  */
/*-------------------------------------------------------------------------------*/
struct selector *Purge_Flagged(slist)
struct selector *slist;
{
   struct selector *last, *front, *i;

   /*------------------*/
   /*  Error checking  */
   /*------------------*/
   if (slist == NULL) return(NULL);

   /*-----------------------------------------------*/
   /*  Make a pass thru list to remove flagged elements  */
   /*-----------------------------------------------*/
   last = NULL;
   i = front = slist;

   while (i != NULL) {

      /*---------------------------------------*/
      /*  Remove flagged node from head of list  */
      /*---------------------------------------*/
      if (i->omitflag) {
         if (last == NULL) {
            front = i->Next;
            free(i);
            i = front; }

      /*-----------------------------------------------*/
      /*  Or remove flagged node from middle of list  */
      /*-----------------------------------------------*/
         else {
            last->Next = i->Next;
            free(i);
            i = last->Next; } }

      /*-------------------------*/
      /*  Or bump to next node  */
      /*-------------------------*/
      else {
         last = i;
         i = i->Next; } }

   /*---------------------------*/
   /*  Return the altered list  */
   /*---------------------------*/
   return(front);
 }


/*-------------------------------------------------------------------------*/
/*  Function COPY_SELECTOR                                                 */
/*                                                                         */
/*  This function makes a duplicate copy of the given selector list.       */
/*-------------------------------------------------------------------------*/
struct selector *Copy_Selector(slist)
struct selector *slist;
{
   struct selector *front, *rear, *s, *n, *NewSel();

   /*------------------------*/
   /*  Set up for copy loop  */
   /*------------------------*/
   front = rear = NULL;
```

```
   s = slist;

   /*----------------------------------------*/
   /*  Copy each element in the list in turn  */
   /*----------------------------------------*/
   while (s != NULL) {
      n = NewSel(s->ofnode, s->state);
      n->omitflag = s->omitflag;

      if (front == NULL) front = rear = n;
      else {
         rear->Next = n;
         rear = n; }

      s = s->Next; }

   /*-----------------------*/
   /*  Return the new list  */
   /*-----------------------*/
   return(front);
 }


/*-------------------------------------------------------------------------------*/
/*  Function UNION_LINK                                                          */
/*                                                                               */
/*  This function copies the contents of the NEXT field to the NEXT_UNION field.  */
/*  This has the effect of making the union list the same as the selector list.   */
/*-------------------------------------------------------------------------------*/
struct selector *Union_Link(slist)
struct selector *slist;
{
   struct selector *s;

   /*----------------------------*/
   /*  Prepare for freeing loop  */
   /*----------------------------*/
   if (slist == NULL) return;
   s = slist;

   /*-------------------------------------------*/
   /*  Free each element in the vector in turn  */
   /*-------------------------------------------*/
   while (s != NULL) {
      s->Next_Union = s->Next;
      s = s->Next; }

   return(slist);
 }


/*-------------------------------------------------------------------------*/
/*  Function NEXT_COPY                                                     */
/*                                                                         */
/*  This function makes a duplicate copy, linked by the NEXT field, of     */
/*  the given union list.                                                  */
/*-------------------------------------------------------------------------*/
struct selector *Next_Copy(ulist)
struct selector *ulist;
{
   struct selector *front, *rear, *s, *n, *NewSel();

   /*-------------------------*/
   /*  Set up for copy loop  */
   /*-------------------------*/
   front = rear = NULL;
   s = ulist;

   /*----------------------------------------*/
   /*  Copy each element in the list in turn.  */
   /*  Field DUPLICATE unneeded by this copy.  */
   /*----------------------------------------*/
   while (s != NULL) {
      n = NewSel(s->ofnode, s->state);
      n->omitflag = s->omitflag;

      if (front == NULL) front = rear = n;
      else {
         rear->Next = n;
         rear = n; }

      s = s->Next_Union; }

   /*-----------------------*/
   /*  Return the new list  */
   /*-----------------------*/
   return(front);
 }
```

```
/*-------------------------------------------------------------------------*/
/*  Function INSERTED_COPY                                                 */
/*                                                                         */
/*  This function takes a sorted selector list (sorted by the OFNODE field)  */
/*  and a single selector element, and inserts the element into the proper   */
/*  location in the list.  The returned list is a copy of the originals.     */
/*-------------------------------------------------------------------------*/
struct selector *Inserted_Copy(item, slist)
struct selector *item, *slist;
{
   struct selector *s, *i, *first, *last, *Copy_Selector();

   /*------------------*/
   /*  Error Checking  */
   /*------------------*/
   if (slist == NULL)  return(Copy_Selector(item));

   if (item == NULL)  {printf("ERROR: Insert passed null item\n");
                       return(Copy_Selector(slist)); }

   if (item->Next != NULL)  {printf("ERROR: Insert passed item longer than 1 element\n");
                             return(NULL); }

   /*-----------------------------------------*/
   /*  Look for the proper place to insert item  */
   /*-----------------------------------------*/
   s = first = Copy_Selector(slist);

   while (s != NULL  &&  item->ofnode->index > s->ofnode->index) {
      last = s;
      s = s->Next; }

   /*-------------------*/
   /*  Insert the item  */
   /*-------------------*/
   i = Copy_Selector(item);

   if (s == first) {
      i->Next = first;
      return(i); }

   else {
      last->Next = i;
      i->Next = s;
      return(first); }
 }


/*-------------------------------------------------------------------------*/
/*  Function APPENDED_COPY                                                 */
/*                                                                         */
/*  This function appends the two given selector lists into a single list  */
/*-------------------------------------------------------------------------*/
struct selector *Appended_Copy(slist1, slist2)
struct selector *slist1, *slist2;
{
   struct selector *end, *s1, *Copy_Selector();

   /*------------------------*/
   /*  Handle trivial cases  */
   /*------------------------*/
   if (slist1 == NULL) return( Copy_Selector(slist2));
   if (slist2 == NULL) return( Copy_Selector(slist1));

   /*-------------------------*/
   /*  Find end of first list  */
   /*-------------------------*/
   end = s1 = Copy_Selector(slist1);
   while (end->Next != NULL) end = end->Next;

   /*--------------------*/
   /*  Append the lists  */
   /*--------------------*/
   end->Next = Copy_Selector(slist2);

   return(s1);
 }

/*************************************************************************/
/***                                                                   ***/
/***  DISTRIBUTION OPERATIONS                                          ***/
/***                                                                   ***/
/***  This set of functions provides some utilities on node distributions  ***/
/*************************************************************************/
```

```
/*-------------------------------------------------------------------------------*/
/*  Procedure INIT_SENSOR_DISTRIBUTION                                           */
/*                                                                               */
/*  This procedure initializes the entire distribution of a sensor node.  Each   */
/*  element in the distribution is given a uniform value of 1 / #states of the node.  */
/*-------------------------------------------------------------------------------*/
Init_Sensor_Distribution(diagram, index)
struct network *diagram;
int index;
{
   struct node *sensornode;
   float p;
   struct selector *preds, *nodesel, *sel;
   struct selector *Select_Predecessors(), *NewSel(), *Appended_Copy();
   BOOLEAN oflag, Odometer();

   /*------------------*/
   /*  Error checking  */
   /*------------------*/
   if (diagram == NULL)
      { printf("ERROR: Init_Sensor_Distribs passed null diagram\n"); return;}

   if (index < 0  ||  index > diagram->lastnode)
      { printf("ERROR: Init_Sensor_Distribs passed invalid node index\n");  return; }

   /*-------------------------------------------*/
   /*  Calculate the uniform probability to use  */
   /*-------------------------------------------*/
   sensornode = &(diagram->nodelist [index]);
   p = 1.0 / sensornode->nstates;

   /*-------------------------------------------*/
   /*  Set the entire distribution to this value  */
   /*-------------------------------------------*/
   Set_Entire_Distribution(sensornode, p);
 }



/*-------------------------------------------------------------------------------*/
/*  Procedure SET_ENTIRE_DISTRIBUTION                                            */
/*                                                                               */
/*  This procedure sets all values of the probability distribution of the given  */
/*  node to the given value.  The distribution dimensions (# of dimensions, size  */
/*  of each) are defined by the predecessors of the node and the node itself.    */
/*-------------------------------------------------------------------------------*/
Set_Entire_Distribution(setnode, value)
struct node *setnode;
float value;
{
   struct selector *preds, *nodesel, *sel;
   struct selector *Select_Predecessors(), *NewSel(), *Appended_Copy();
   BOOLEAN oflag, Odometer();

   /*------------------*/
   /*  Error checking  */
   /*------------------*/
   if (setnode == NULL) {printf("ERROR: Set_Entire_Distrib passed null node\n"); return;}

   /*-----------------------------------------------------------------*/
   /*  Build a selector to index the entire sensor node distribution  */
   /*-----------------------------------------------------------------*/
   preds = Select_Predecessors(setnode);
   nodesel = NewSel(setnode, 0);
   sel = Appended_Copy(nodesel, preds);

   /*-----------------------------------------------------------------*/
   /*  Loop thru all elements in the distribution, setting each to value  */
   /*-----------------------------------------------------------------*/
   oflag = 0;
   Zero(sel, REGULAR);

   while (Odometer(sel, REGULAR, &oflag)) Set_Prob(setnode->header, sel, value);

   /*----------------------*/
   /*  Free all selectors  */
   /*----------------------*/
   Free_Selector(preds);
   Free_Selector(nodesel);
   Free_Selector(sel);
 }


/*-------------------------------------------------------------------------------------*/
/*  Function FIND_BEST_UTILITY                                                         */
/*                                                                                     */
/*  This function finds the highest utility value in the given decision node's distribution,  */
/*  and returns a selector that will index that value.                                 */
```

```
/*-------------------------------------------------------------------------------------------*/
struct selector *Find_Best_Utility(decision_node)
struct node *decision_node;
{
   float EV;
   float highEV = -1E20;
   BOOLEAN oflag, Odometer();
   struct selector *highsel, *decpreds, *copypreds;
   struct selector *Select_Predecessors(), *Copy_Selector();

   /*------------------*/
   /*  Error checking  */
   /*------------------*/
   if (decision_node == NULL) printf("ERROR: Find_Best_Utility passed null decision node\n");
   if (decision_node->type != DECISION) printf("ERROR: Find_Best_Utility node not a decision node\n");

   /*-----------------------------------------------------------*/
   /*  Get selector list for all predecessors of decision_node  */
   /*-----------------------------------------------------------*/
   decpreds = Select_Predecessors(decision_node, NULL);
   Zero(decpreds, REGULAR);
   oflag = 0;

   /*-------------------------------------------------------------------*/
   /*  Loop thru all possible values of the decision node's predecessors  */
   /*-------------------------------------------------------------------*/
   while (Odometer(decpreds, REGULAR, &oflag)) {

      /*-------------------------------------------------------------*/
      /*  Special case: when decpreds is null, use selector of (0).  */
      /*  This is to match a similar case in Maximize_Expect_Util.   */
      /*-------------------------------------------------------------*/
      if (decpreds == NULL) copypreds = Select(0, -1);
      else copypreds = Copy_Selector(decpreds);

      /*-----------------------------------------------------*/
      /*  Get the expected value for this set of conditions  */
      /*-----------------------------------------------------*/
      EV = Get_Prob(decision_node->EV, copypreds);

#ifdef PRINT_UTILITY
      show(0, 'E', copypreds, REGULAR, EV);
#endif

      /*-----------------------------------------------------------------*/
      /*  Save the selector associated with the highest expected value  */
      /*-----------------------------------------------------------------*/
      if (EV > highEV) {
         highEV = EV;
         highsel = Copy_Selector(copypreds); }

      Free_Selector(copypreds); }

   /*-------------*/
   /*  Finish up  */
   /*-------------*/
   printf("Highest EV is %g\n", highEV);
   Free_Selector(decpreds);
   return(highsel);
}
```