Lecture 31: Graph Search & Game
10:00 AM, Nov 16, 2018

Contents

1 Rooted-tree search 1
2 DAG search 2
3 Search in a graph with cycles 5
4 The GAME Signature 5
5 Functors 6

1 Rooted-tree search

Start with rooted tree search

Here is a graph defined by the procedure

\[
\text{let } g : \text{int } \rightarrow \text{int list } = \text{function } n \rightarrow \\
\quad \text{if } n < 30 \text{ then } [3*n-1; 3*n; 3*n+1] \text{ else } []
\]

Remember that a path is a sequence of vertices \( v_1, \ldots, v_k \) such that \( v_2 \) is a neighbor of \( v_1 \), \( v_3 \) is a neighbor of \( v_2 \), and so on.

Our goal is to write the procedure \( \text{pathP} \) with the following spec.

- **input:** graph, source node, destination node
- **output:** is there a path in the graph from source to destination?

Recall that last time we wrote the procedure \( \text{successP} \) with the following spec:
input: a procedure f, and a list
output: does f return true on any element of the list?

The procedure successP is supposed to stop when it finds an element on which f returns true.

In the example graph, there is a path from s to some node d if there is a path from one of s’s neighbors to n.

What case is not covered by the above rule? Is there a situation in which there is path from s to d but it does not involve a neighbor of s? Yes, when s = d. This is the base case.

let rec pathP : 'a graph * 'a * 'a -> bool = function (g, source, dest) ->
  if source = dest then true
  else successP ((function v -> pathP (g, v, dest)), (g source));;

We can use boolean logic to simplify this to the following code:

let rec pathP : 'a graph * 'a * 'a -> bool = function (g, source, dest) ->
  source = dest || successP ((function v -> pathP (g, v, dest)), (g source));;

Now let’s modify it to return a path. We can represent a path by a list of nodes. But wait, what if there isn’t a path that exists? To account for this case, we should use an option instead.

find_path:
input: graph, source, destination
output: if there is a path p in the graph from source to destination, return Some(p). Otherwise, return None.

Let’s say a recursive call finds a path (and therefore returns a path option). How do we derive the output for the original call? Let’s draw a recursion diagram:

original input: source = 1, destination = 17
recursive input: source = 2, destination = 17
recursive output: Some([2; 6; 17])
original output: Some ([1;2;6;17])

We can see from this recursion diagram that we derive the original path from the recursive path by consing. However, it is slightly more complicated because the recursive and original outputs are options. How do we get around this? You may remember from eliza that we can use pattern matching!

Additionally for the base case, where the source and the destination are the same, what’s the path? The answer to this is that it should be a sequence consisting of a single node, the source/destination.

From this, let us write find_path.

let rec find_path = function (g, origin, dest) ->
  if origin = dest then Some([dest])
  else match first_success ((function v -> find_path (g, v, dest)), g origin) with
      None -> None
    | Some path -> Some (origin::path)
2 DAG search

What if the graph is not a tree? Consider this graph on the vertices \(\{1, 2, 3, 5, 6, 7, 8\}\).

```ml
let dag = function
  1 -> [2; 3]
  | 2 -> [5; 6; 7]
  | 3 -> [7; 8]
  | 5 -> []
  | 6 -> []
  | 7 -> [9]
  | 8 -> [9]
```

Let’s say we search for a path from 1 to 9.

```ml
# find_path(dag, 1, 9);;
- : int list option = Some [1; 2; 7; 9]
```

Okay, first note that there are multiple paths from 1 to 9. Why did the procedure return that one? The reason is because it’s the first one that succeeded in the order explored. The neighbor 2 of 1 is explored before the neighbor 3 of 1.

What if we tried to find a 1-to-8 path? The algorithm will explore the paths \([1; 2; 5], [1; 2; 6], [1; 2; 7], [1; 3; 7; 9], [1; 3; 8]\). Note that it considers two paths to 7. Should this worry us?

Let’s consider a modified diamond graph:
Let’s say we search for a 1-to-11 path. The algorithm will first explore paths [1;2;4;5;7;8;10], [1;2;4;5;7;9;10], [1;2;4;6;7;8;10], [1;2;4;6;7;9;10], [1;3;4;5;7;8;10], [1;3;4;5;7;9;10], [1;3;4;6;7;8;10], [1;3;4;6;7;9;10].

And that’s only three levels of diamonds. The time required ends up being exponential in the number of levels of diamonds. Bad!

But it gets worse. This graph is directed—we don’t allow travel in both directions—and it has no directed cycle. (A cycle is essentially a path that starts and ends at the same node.) It is called a directed acyclic graph, abbreviated DAG.
3 Search in a graph with cycles

Consider this graph

On this graph, the algorithm will never finish.
We need to augment the algorithm so that it does not re-visit vertices.
To get around this, we can use a set! The signature for sets is as follows:

```ocaml
module type SET =
  sig
    type 'a set
    (* outputs an empty set *)
    val empty: 'a set
    (* outputs a set with the new element added to the old set *)
    val insert: 'a set * 'a -> 'a set
    (* outputs true if set contains given element, false otherwise *)
    val containsP: 'a set * 'a -> bool
  end
```

The algorithm will maintain a set, visited, and insert into it as it goes. Before visiting the neighbor of the current node, it will verify that the neighbor is not in the set visited.

```ocaml
let rec dfs_helper : 'a Set.set * 'a -> 'a list option
```

Is this worth it? Each invocation is much more expensive but this saves the algorithm from being exponential or infinite. It is definitely worth it especially once we use a set representation for which the per-operation time is $O(\log n)$ or $O(1)$.

4 The GAME Signature

Here is a simplified version of the GAME signature
module type GAME =
sig
  type which_player = P1 | P2
  type status =
    | Win of which_player |
    | Draw |
    | Ongoing of which_player
  type state
  type move

  val initial_state : state
  val legal_moves : state -> move list
  val game_status : state -> status
  val next_state : state -> move -> state
end

The types state and move are abstract. This means we can write programs that use states and
moves without the programs knowing the details of these types—which means those programs can
be reused for different realizations of GAME.

5 Functors

Just as OCaml has procedures that take values and output values, OCaml has functors, which “take”
modules and “output” modules. These are not intended to be used during the running of a program.
Instead, these are used to create programs.

As an example, consider the zero-sum game of perfect information, NIM. Initially, there is a pile of
matches. To move, a player removes any number of matches from the pile. The losing player is the
one who removes the final match.

Let’s say we want to create a module for Nim. However, we want someone to be able to take in a
value for the initial number of matches in each pile. To do this, we can use a functor that looks like
the following:

module Nim = functor (I: sig val initial: int end) ->
struct
  type which_player = P1 | P2
  type status =
    Win of which_player |
    Draw |
    Ongoing of which_player
  type state = ....
  type move = int
  let initial_state = .... I.initial ...
  let legal_moves = ...
  let other_player = ...
  let game_status = ...
  let next_state = ...
end

This way, we can create a game of Nim with any number of matches in the pile. For instance, if we
wanted to create a game with 22 matches, we could say
module NimGame = Nim(struct let initial = 22 end)

In addition to the initial game module, there is a HumanPlayer module and a Referee module, both of which are defined using functors. The signatures for both of these can be found of the lecture slides. To put each of these pieces together, the Referee must take in a GAME as well as two PLAYERS. Thus, we can initiate a game using the Referee signature between two HumanPlayers as follows:

```ocaml
module Referee = functor
    (Game : GAME)
    (Player1 : PLAYER)
    (Player2 : PLAYER)
  struct
    module CurrentGame = Game
    let play_game () = function ....
  end
module Ref = Referee
    (NimGame)
    (HumanPlayer(NimGame))
    (HumanPlayer(NimGame))
Ref.play_game()
```

However, with the above code, because each player takes in a game as well, it would be possible for you to input two players which were meant for separate games, which would result in errors. To fix this, we can change the module for Referee to the following:

```ocaml
module Referee = functor
    (Game : GAME)
    (Player1 : PLAYER with module PlayerGame = Game)
    (Player2 : PLAYER with module PlayerGame = Game)
  struct
    module CurrentGame = Game
    let play_game () = function ....
  end
```

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/csci0170/feedback](http://cs.brown.edu/courses/csci0170/feedback)