CISC 481 Programming Assignment 1

September 9, 2019

[Goals: Basic blind/heuristic search techniques]

[Warning: For some students, these assignments can take longer than you anticipate, so START EARLY. Potential trouble areas include choice of representation for states and nodes; actually writing the search algorithms discussed in class (remember there is psuedocode in the textbook); issues with memory and processing time when working with truly large, NP-hard problems.]

The Switch Problem

Consider the problem of maneuvering railroad cars in a train yard. Many cars must be assembled into a train of cars in a given order, but the cars start out dispersed throughout the yard, and can only move in prescribed ways on existing track. Below is an example of a small train yard with sufficient connectivity to arrange railroad cars into any desired sequence. The different sections of track have been numbered from 1 to 6. The figure also shows the starting position for a small train, where each car is represented by a lowercase letter, and the engine is represented by an asterisk "*". Finally, we show the same cars in the goal position, with the cars lined up in order for a departure to the west.

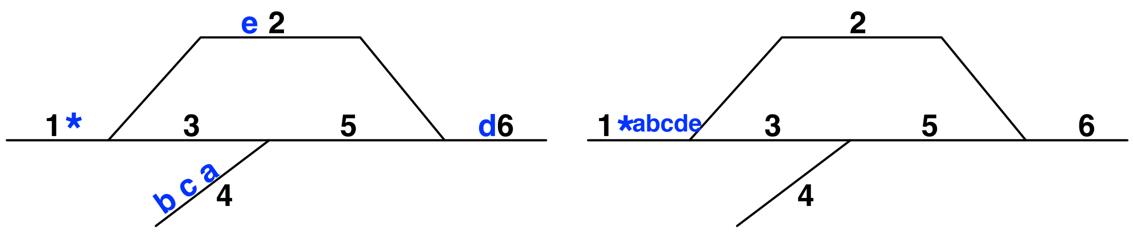

Figure 1: A picture of the initial state and final state of YARD-1.

Representing the Problem

We can represent a train *Yard* as a connectivity list. The simple nature of train yards used in this assignment allows us to describe the yard as a list of right and left edges connected by vertices (in the real world, such vertices are called ”switches” :-)). Thus, because the right edge of track 1 connects to the left edges of tracks 2 and 3, we find the sublists `(1 2)` and `(1 3)` included on our yard vertex list. *[My examples here are in Lisp, but feel free to use something similar that's easy to parse in your target language.]*

```
(defvar *YARD-1* '((1 2) (1 3) (3 5) (4 5) (2 6) (5 6)))
```

A *State* can be represented as a list of the cars on each section of the track in their respective orders. The first element is `(*)`, indicating only the engine is on track 1, and the sixth element is `(d)` because only car `d` is on track 6. Note that when there's more than one car on a track, they are listed left-to-right (`b` is on the left end of track 4).

```
(defvar *INIT-STATE-1* '((*) (e) nil (b c a) nil (d)))
(defvar *GOAL-STATE-1* '((* a b c d e) nil nil nil nil nil))
```

Rules of Movement (Actions)

The following rules describe an *Action* in the train yard `(DIRECTION FROM-TRACK TO-TRACK)`:

- `(LEFT y x)`: If the connectivity list contains a sublist `(x y)` and either track `x` or track `y` contains the engine, then the first car of track `y` can be removed from track `y` and placed at the end of track `x`. We call this a `LEFT` move; so if `(1 2)` is on the connectivity list, we can move from state `((*) (e))` to state `((* e) nil)`, or from state `((a * b) (c d))` to state `((a * b c) (d))`. We will notate this below as `(LEFT y x)`[1] meaning move one car from the left end of track `y` leftward to the (right) end of track x.

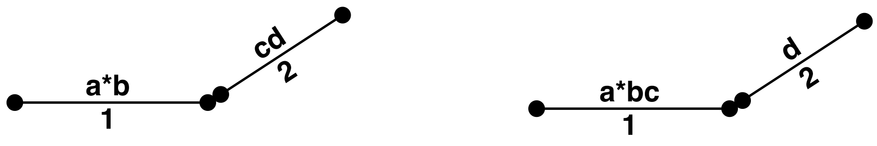

Figure 2: `(LEFT 2 1)`: Move leftmost car FROM track 2 LEFTward TO become the rightmost car on track 1; ”move one car LEFT from track 2 to track 1”

[1]**NB:** the transposition of `y` and `x`

- `(RIGHT x y)`: If the connectivity list contains the sublist `(x y)` and either track `x` or track `y` contains the engine, then the last car of track `x` an be removed from track x and placed at the front of track `y`. We call this a `RIGHT` move; so if `(1 2)` is on the connectivity list, then a legal move from `((*) (e))` is to `(nil (* e))`, and a legal move from state `((a * b) (c d))` is to `((a *)(b c d))`.[2] We will notate this as `(RIGHT x y)` below, meaning move one car rightward from (the right end of) track `x` to (the left end of) track `y`.

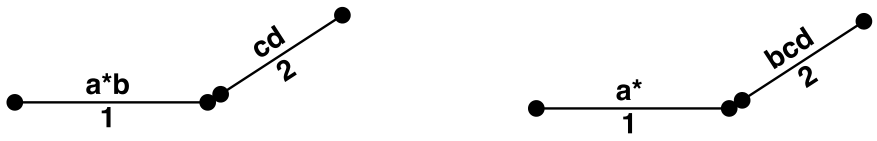

Figure 3: `(RIGHT 1 2)`: Move rightmost car FROM track 1 RIGHTward TO become leftmost car on track 2; ”move one car RIGHT from track 1 to track 2”

- **No other actions are allowed**; cars cannot move without using the engine, jump over other cars, or teleport from one track to another.

PROBLEM 1 [10 pts]

Write a function `possible-actions` that consumes a Yard (connectivity list) and a State, and produces a list of all actions possible in the given train yard from the given state. Run your function on at least three different yards and two different states for each yard, including the two large yards and initial states described pictorially in this handout. *[This is the $ACTIONS$ function we discussed in class and in the book.]* [3]

PROBLEM 2 [10 pts]

Write a function `result` that consumes an Action and a State and produces the new State that will result after actually carrying out the input move in the input state. Be certain that you do NOT accidentally modify the input state variable!!! Also, I’ve never seen a ”clever” solution to this, you just have to do it by cases. It’s ugly! Sorry! *[This is the $RESULT$ function we discussed in class; the state transition model]*

```
(prove:is (result '(left 2 1) *INIT-STATE-1*)
          '((* e) nil nil (b c a) nil (d)))

(prove:is (result '(right 1 2) *INIT-STATE-1*)
          '(nil (* e) nil (b c a) nil (d)))
```

1 *PROBLEM 3 [10 pts]

Write a function `expand` that consumes a State and a Yard, and produces a list of all states that can be reached in one Action from the given state. *[This is a trivial extension of Problems* 1 *and* 2.*]*

[2]**NB:** that only one symbol moves each time (the engine must have pushed car `b` right and then returned to track 1).
[3]**Hint:** Recall that all possible actions must be from or to a track with the engine.

```
(prove:is (expand *INIT-STATE-1* YARD-1)
          (list '(nil (* e) nil (b c a) nil (d))
                '(nil (e) (*) (b c a) nil (d))
                '((* e) nil nil (b c a) nil (d))))
```

PROBLEM 4 [30 pts]

Write a program that consumes a connectivity list (Yard), an initial State, and a goal State as inputs, and produces a list of Actions that will take the cars in the initial state into the goal state.

Use a blind (uninformed) search method. Briefly justify your choice (including a discussion of optimality).

Test your program on the following problem, `YARD-2`. *NB: You should of course try the little tiny yards, yards 3–5, listed at the bottom of this assignment, before trying* `YARD-2`*. Do NOT try to do* `YARD-1` *with blind search!!!*

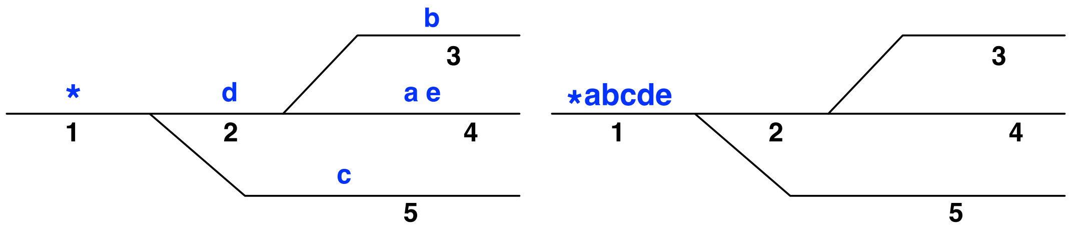

Figure 4: A picture of the initial state and final state of YARD-2.

PROBLEM 5 [10 pts]

How big is the search space for c cars on t tracks [i.e. how many possible states]??

PROBLEM 6 [30 pts]

Now describe at least one heuristic. Write a new search program to do a heuristic search using an algorithm of your choice (choose carefully; you should aim for an optimal solution). Run your program again on the same yards. You should expect a speedup of at least 2 or 3 times. You might not be able to solve `YARD-1` at all unless your heuristic is very good or you are very efficient.

NOTES

These problems are "hard" computationally. Make sure your code works on simple train yards first. Use extensive Unit Testing on your `possible-actions` and `result` functions.

Here're three simpler problems to test code on before trying a hard problem. Don't EVEN think of trying `YARD-1` or `YARD-2` unless you can solve these trivial yards!!!

```
(defvar *YARD-3 '((1 2) (1 3)))
(defvar *INIT-STATE-3* '((*) (a) (b)))
(defvar *GOAL-STATE-3* '((* a b) nil nil))

(defvar *YARD-4* '((1 2) (1 3) (1 4)))
(defvar *INIT-STATE-4* '((*) (a) (b c) (d)))
(defvar *GOAL-STATE-4* '((* a b c d) nil nil nil))

(defvar *YARD-5* '((1 2) (1 3) (1 4)))
(defvar *INIT-STATE-5* '((*) (a) (c b) (d))) ;Note c and b out of order
(defvar *GOAL-STATE-5* '((* a b c d) nil nil nil))
```

Document your code appropriately. That's part of your grade here.

Your code should run on similar inputs other than the 5 *shown here!!! Make sure you indicate how to run your code on other yards for the TA.*