Homework 2

Ruojun Li

1. Chapter 3, Ex 3.13 (14 points)

3.13 Prove that GRAPH-SEARCH satisfies the graph separation property illustrated in Figure 3.9. (*Hint*: Begin by showing that the property holds at the start, then show that if it holds before an iteration of the algorithm, it holds afterwards.) Describe a search algorithm that violates the property.

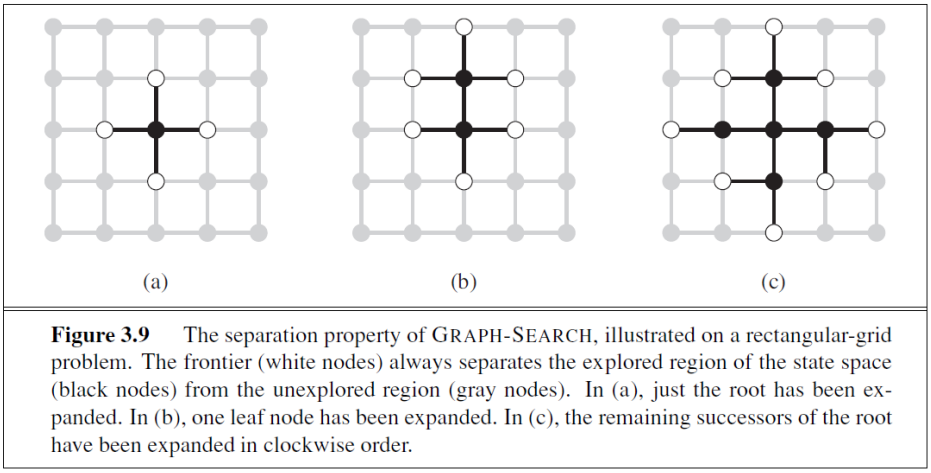

Figure 3.9 The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been expanded. In (c), the remaining successors of the root have been expanded in clockwise order.

The graph separation property is that each path from where the state starts to a state that the agent has not yet gone to has to go through a state in the frontier.
The separation property is illustrated in Figure 3.9. As every step moves a state from the frontier into the explored region while moving some states from the unexplored region into the frontier, we see that the algorithm is systematically examining the states in the state space, one by one, until it finds a solution.
All the successors of a node can be generated at once (method most commonly used) or they could be generated one at a time either in a systematic way or in a random way. The number of successors is called the branching factor. (In the example fig 3.9, we use the clockwise order to generate nodes.)

Prove:
At the start, in the figure 3.9, we have the initial node which is connected to itself apparently. Then we add its neighbors as its children nodes and they will be in the expanded list. After several iterations, since the parents of each node is must in the expanded list, it must has a path to the initial node.

Violation:
Besides these methods, it's not always necessary to generate all children nodes. In the algorithm, we move the not-totally-explored-node into the explored-list will violate the separation property.

2. Chapter 3, Ex 3.16 (16 points)
3.16 A basic wooden railway set contains the pieces shown in Figure 3.32. The task is to connect these pieces into a railway that has no overlapping tracks and no loose ends where a train could run off onto the floor.

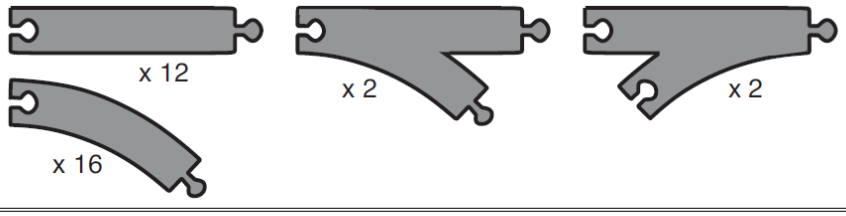

Figure 3.32 The track pieces in a wooden railway set; each is labeled with the number of copies in the set. Note that curved pieces and "fork" pieces ("switches" or "points") can be flipped over so they can curve in either direction. Each curve subtends 45 degrees.

a. Suppose that the pieces fit together *exactly* with no slack. Give a precise formulation of the task as a search problem.

Initial state: A Random Railway Piece
Successor function: For the pegs in the current railway, randomly choose a track piece from the left pieces in the Railway Set. We have three kinds holes,

1. Straight hole
2. Fork holes: Connect either of two with either orientation
3. Curve holes: Connect either orientation.

After connection one piece, make sure that the railway is not overlapping.
Goal test: All track pieces are used without overlapping way and loose end.
Step cost: One per piece

b. Identify a suitable uninformed search algorithm for this task and explain your choice.
Two commonly used uninformed search algorithms are:

1. Depth first: Proper. The goal is finding one solution but not find all solutions. The depth first search will search as deep as possible. When any branch success, we can finish the searching.
2. Breadth first: Not good. The Tree will grow widely. Also, this algorithm will be helpful in finding the shortest path. but in our problem, all solution has same cost.
3. Uniform cost: Not good. They have the same cost, can't get solution faster.
4. Depth-First Search Iterative-Deepening: Not good. All solutions have the same amount of track pieces. The depth and cost are the same. It's useless to set a limited depth in the algorithm.

c. Explain why removing any one of the "fork" pieces makes the problem unsolvable.

If we can solve this problem, that's mean each hole connect to a peg. In the simple word, we have the same number of pegs and holes. After remove one fork, the amount is not same and makes the problem unsolved.

d. Give an upper bound on the total size of the state space defined by your formulation. (*Hint*: think about the maximum branching factor for the construction process and the maximum depth, ignoring the problem of overlapping pieces and loose ends. Begin by pretending that every piece is unique.)

Maximum depth is 32. (The total pieces number in the trail set.)

We start from one peg, and the next piece has many possibilities:)

Straight: 12

Curve: 2 *16(2 orientations)

Switch: 2 *2(2 orientations)

Point: 2*2*2(2 orientations with 2 pegs)

In total: 3*56 = 168

$$\frac{168^{32}}{12!\,16!\,2!\,2!}$$

3. Chapter 3, Ex. 3.26 (16 points)
3.26 Consider the unbounded version of the regular 2D grid shown in Figure 3.9. The start state is at the origin, (0,0), and the goal state is at (x, y).

- **a.** What is the branching factor b in this state space?
  Four directions is the branching. Factor is four.
- **b.** How many distinct states are there at depth k (for $k > 0$)?
  2k*(k+1) + 1
- **c.** What is the maximum number of nodes expanded by breadth-first tree search?
  4^{X+Y}
- **d.** What is the maximum number of nodes expanded by breadth-first graph search?
  $2(x + y)(x + y + 1) - 1$.
- **e.** Is $h = |u - x| + |v - y|$ an admissible heuristic for a state at (u, v)? Explain.
  Admissible, the branch has only four directions on the grid.
- **f.** How many nodes are expanded by A* graph search using h?
  x + y, because the heuristic is perfect.
- **g.** Does h remain admissible if some links are removed?
  Yes; removing links will increase actual path lengths, and the heuristic will be an underestimate.
- **h.** Does h remain admissible if some links are added between nonadjacent states?

Not admissible; adding new links will decrease some actual path lengths and the heuristic will no longer be an underestimate.

4. Chapter 4, Ex. 4.3 (20 points)
Note: Please ignore the second part of 4.3.a, which asks for the following, "Compare the results with optimal solutions obtained from the A* algorithm with the MST heuristic (Exercise 3.30)."

4.3 In this exercise, we explore the use of local search methods to solve TSPs of the type defined in Exercise 3.30.
a. Implement and test a hill-climbing method to solve TSPs.
b. Repeat part (a) using a genetic algorithm instead of hill climbing. You may want to consult Larra˜naga *et al.* (1999) for some suggestions for representations.

3.30 The traveling salesperson problem (TSP) can be solved with the minimum-spanning-tree (MST) heuristic, which estimates the cost of completing a tour, given that a partial tour has already been constructed. The MST cost of a set of cities is the smallest sum of the link

costs of any tree that connects all the cities.

Test result:

 # code is in the attachment

Example 1- a simple map with 5 cities

```
map_graph_data, map = map_example()
all_cities, distances = hill_climbing_cities(map)
tsp_problem = TSP_problem(all_cities)
routin = hill_climbing(tsp_problem,iterations=1000)
```

```
Hill Climbing Algorithms is finding the max path value in the neighbors.
This hill climbing will iterate the operation 1000 times to achieve the best
current rountin is:
 ['C_City', 'D_City', 'A_City', 'B_City', 'E_City']
Current path value is:  -682.842712474619
```

A test example with Romania Map

```
# Example to test the code
map = romania_map
all_cities, distances = hill_climbing_cities(map)
tsp_problem = TSP_problem(all_cities)
routin = hill_climbing(tsp_problem)
```

 # code is in the attachment

5. Chapter 4, Ex. 4.5 (14 points)

Note: Provide the updated algorithm first and then provide the required reasoning.

4.5 The AND-OR-GRAPH-SEARCH algorithm in Figure 4.11 checks for repeated states only on the path from the root to the current state. Suppose that, in addition, the algorithm were **to store *every* visited state and check against that list**. (See BREADTH-FIRST-SEARCH in Figure 3.11 for an example.) Determine the information that should be stored and how the algorithm should use that information when a repeated state is found. (*Hint*: You will need to distinguish at least between states for which a successful subplan was constructed previously and states for which no subplan could be found.) Explain how to use labels, as defined in Section 4.3.3, to avoid having multiple copies of subplans.

The previous pseudocode:

```
function AND-OR-GRAPH-SEARCH(problem) returns a conditional plan, or failure
  OR-SEARCH(problem.INITIAL-STATE, problem, [])

function OR-SEARCH(state, problem, path) returns a conditional plan, or failure
  if problem.GOAL-TEST(state) then return the empty plan
  if state is on path then return failure
  for each action in problem.ACTIONS(state) do
      plan ← AND-SEARCH(RESULTS(state, action), problem, [state | path])
      if plan ≠ failure then return [action | plan]
  return failure

function AND-SEARCH(states, problem, path) returns a conditional plan, or failure
  for each s_i in states do
      plan_i ← OR-SEARCH(s_i, problem, path)
      if plan_i = failure then return failure
  return [if s_1 then plan_1 else if s_2 then plan_2 else ... if s_{n-1} then plan_{n-1} else plan_n]
```

Figure 4.11 An algorithm for searching AND–OR graphs generated by nondeterministic environments. It returns a conditional plan that reaches a goal state in all circumstances. (The notation $[x \mid l]$ refers to the list formed by adding object x to the front of list l.)

In the above code, only the successful path is stored. The failure path return “failure” directly.

If the state is failed previously, still record the value of the path, with label “failure”

After recording all states:

```
function AND-OR-GRAPH-SEARCH(problem) returns a conditional plan, or failure
  OR-SEARCH(problem.INITIAL-STATE, problem, [])

function OR-SEARCH(state, problem, path) returns a conditional plan, or failure
  if problem.GOAL-TEST(state) then return the empty plan
  if state has previously been solved then return RECALL-SUCCESS(state)
  if state has previously failed for a subset of path then return failure
  if state is on path then
      RECORD-FAILURE(state, path)
      return failure
  for each action in problem.ACTIONS(state) do
      plan ← AND-SEARCH(RESULTS(state, action), problem, [state | path])
      if plan ≠ failure then
          RECORD-SUCCESS(state, [action | plan])
          return [action | plan]
  return failure

function AND-SEARCH(states, problem, path) returns a conditional plan, or failure
  for each s_i in states do
      plan_i ← OR-SEARCH(s_i, problem, path)
      if plan_i = failure then return failure
  return [if s_1 then plan_1 else if s_2 then plan_2 else ... if s_{n-1} then plan_{n-1} else plan_n]
```