Daniel McDonough (dmcodnough)
9/30
CS534

1 (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.

> Graph search satisfies graph separation property due to the fact that graph search only look at the connections of the state space. The state space does not have any connections to any unexplored region. A graph search algorithm starting at root node, A, assuming the search remembers previously visited nodes and will not repeat them, it will visit the nodes consistently expanding the frontier outward. The edges are searched systematically until the goal state is found. The algorithm should never select a node in the graph that is part of the unexplored region.
>
> A search algorithm that searches random nodes would break the graph separation property due to the probability of the random selection being part of the unexplored region. A search algorithm such as Random-restart hill climbing would fit this classification. Where if a search fails the program will select a random starting point, which may be part of the unexplored region.

2 (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.

> a. Suppose that the pieces fit together exactly with no slack. Give a precise formulation of the task as a search problem.
>
> Arranging the tracks as a tree where the children are possible connections through other available tracks and number of tracks left. Each height of the "tree" would have one less piece. The goal state is when all tracks are used and there are no open ends. The root node is an empty tack with all tracks left.
>
> b. Identify a suitable uninformed search algorithm for this task and explain your choice.
>
> The best uniformed search algorithm would be depth-first search, as it would go directly into a complete track and modify the track piece by piece. This gives the fastest way to solve the problem as it solves it in linear time because all solutions are at the same depth of the tree.
>
> c. Explain why removing any one of the "fork" pieces makes the problem unsolvable.
>
> Removing a fork piece would make it unsolvable because there would always be an edge where the train can run off the track. Because 3 edges * (4-1) = 9 edges which is not divisible by 2 thereby becoming not a closed circle.
>
> 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.)

Given 32 “unique pieces” there are 32! possible final configurations. Now to include the nodes leading to the solution would be the sum of N! From N = 0 to 31. This simplifies to ∑N! From N=0 to 32 which is approximately 2.72 * 10 ^ 35

3(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?

Assuming, the nodes are expanding in a clockwise fashion each pass:
4/N + 3*∑(4+4(N-2)+N-1)/N

b) How many distinct states are there at depth k (for k > 0)?

There would be 4K distinct states in the state space as at each depth is expanded by 4 directions

c) What is the maximum number of nodes expanded by breadth-first tree search?

∑2^N from 0 to N for N nodes

d) What is the maximum number of nodes expanded by breadth-first graph search?

$2N^2 + B + 2N – 1$
where N is the number of nodes on the graph and B is the branching factor
$2N^2$ is counting for every node in the graph plus cycles. 2N counts for the attempt expansion, of frontier nodes that are walls and minus 1 for the root node.

e) Is h = |u − x| + |v − y| an admissible heuristic for a state at (u, v)? Explain.

Yes, as it calculates the cost for each node in a cheapest fashion, never overestimating. The goal state will always be 0. h() is also linear, which can be shown as directional vectors because they will always point toward the goal state, even though that direction may not necessarily be the best path. It is because that is linear the algorithm is admissible.

f) How many nodes are expanded by A ∗ graph search using h?

(x*y) / N on average because it would expand all of the nodes/2

g) Does h remain admissible if some links are removed?

Yes, if nodes are removed the heuristic will still underestimate the cost of the path to the goal as it determines the path via a straight line.

h) Does h remain admissible if some links are added between nonadjacent states?

By adding links H not does remain admissible because it now may overestimate the cost to the goal even though the function is still linear in relation to node location.

4 (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. Compare the results with optimal solutions obtained from the A ∗ algorithm with the MST heuristic (Exercise 3.30).

b. Repeat part (a) using a genetic algorithm instead of hill climbing. You may want to consult Larrañaga et al. (1999) for some suggestions for representations.

5 (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.

For the search to keep track of not only repeated states in the current path but repeated states in general, one must store each visited node, give each node a label, and keep track of the connections those nodes have, and the requirement to get to that state (And or or). To check for duplicates the search will keep a list of routes to states. In each list of routes the algorithm will check for duplicate states. To check for a duplicate state, the algorithm would assign a unique label, determined by the children of the nodes. When the algorithm searches the children the info of he parent/ancestors is updated to keep track of unique nodes by the sub plan criteria. Then upon each each new sub plan, the algorithm is to unify the possible states to form a general map. This can be done by checking iteratively between nodes between paths to get a unified vision of the map. The algorithm then needs to check between paths for duplicates. If duplicates are found then sub plans can be compiled together by a list of path IDs. This can all be simplified by keeping track of the following data per state: a List of children/ connections with criteria, a list of route IDs taken passing the node, and a label of the node.