CS 543: Homework Assignment 2

Aditya Mehrotra
amehrotra@wpi.edu

Chp 3, Ex 3.13

First, let's define the graph separation property. Each path from where the state begins to a state that the agent has not yet visited must pass via a state in the frontier, according to the graph separation condition. This is accurate in the sense that the border contains the starting state at the start of a search, therefore in order for the agent to travel to a new state, they must first pass through the frontier. The condition holds true for graph searches because the frontier node will always exist before moving on to an undiscovered state or node when you complete iterations and traverse through nodes seeking for the target state. Each path from the starting state to the undiscovered state has a node in the frontier when you do the graph search, as defined by the graph separation condition. There is a method to have the search algorithm work against you. This is what happens when you use an algorithm to shift nodes without taking into account all of the possible successors. It is feasible to write an algorithm that only generates specific nodes to travel from path a to path b, however this would break the graph separation property.

Chp 3, Ex 3.16

a Any part of the track will represent the beginning state. Depending on the type of open hole, the successor function will add a piece from what is left. The aim of the exam will be to see if you used all of the parts and made a linked track with no overlaps. The cost of each stage will be equal to the number of components.

b To address this, we should use a depth first search strategy. Because of the large state space, this would be the best option. To get the same result as a depth first search, a breadth first or iterative search would require far too many steps.

c One won't be able to solve the problem if the fork parts are removed. This is because when one uses a fork to divide two tracks, it creates a fork. A fork would be required to reunite these two tracks. Forks are required in order for the track to have a solution.

d The maximum possible number of open pegs is 3 (starts at 1, adding a two-peg fork increases it by one). Pretending each piece is unique, any piece can be added to a peg, giving at most 12+(2·16)+(2·2)+(2·2·2) = 56 choices per peg. The total depth is 32 (there are 32 pieces), so an upper bound is $168^{32}/(12!16!2!2!)$ where the factorials deal with permutations of identical pieces.

Chp 3, Ex 3.26

a 4

b The states at depth k form a square rotated at 45 degrees to the grid. There are a linear number of states along the boundary of the square, so the answer is 4k.

c Without repeated state checking, BFS expends exponentially many nodes: counting precisely, we get $((4^{x+y+1}-1)/3)-1$.

d 2(x + y)(x + y + 1) – 1 because there are quadratically many states within the square for depth x + y.

e Yes. The robot can only move in the grid directions (Manhattan Distance).

f False; all nodes in the rectangle defined by (0,0) and (x,y) are candidates for the optimal path, and there are quadratically many of them, all of which may be expended in the worst case.

g Yes; removing links will increase actual path lengths, and the heuristic will be an underestimate.

h No; adding new links will decrease some actual path lengths and the heuristic will no longer be an underestimate.

Chp 4, Ex 4.3

See attached .zip file for code to parts **a** **b**.

Chp 4, Ex 4.5

OR-SEARCH records the solution discovered in states when it finds one. If it returns to that situation in the future, it will return that as a solution. When OR-SEARCH is unable to discover a solution, it must exercise caution. Because we don't allow cycles, whether a state can be solved relies on the path taken to get there. As a result, if OR-SEARCH fails, the value of path is saved. OR-SEARCH returns failure if a state has previously failed when the path contained any subset of its current value.

Sub-solutions should not be repeated. We may label all new solutions discovered, keep track of these labels, and then return the label if the states are visited again.

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** SUCCESS(*state*)
  if *state* has previously been failed for subset of *path* **then return** *failure*
  if *state* is on *path*
    FAILURE(*state, path*)
  return *failure*
  for each *action* **in** *problem*.ACTIONS(*state*) do
    plan $\leftarrow AND-SEARCH(RESULTS(state, action), problem, [state|path])$
    if plan $\neq$ failure **then**
      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**
    $plan_i \leftarrow OR-SEARCH(s_i, problem, path)$
    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** $plan_n$]