CSC384 Fall 2019 Assignment 1 - Search Algorithms

Name: Anirudh Sampath
SN: 1002630218

Q1. Below is the pseudo-code of S and G functions for the flood fill algorithm:

```
def S(state):
    x,y = state
    successors = []
    neighbours = [state+up,state+down,state+left ,state+right]
    for neighbour in neighbours:
        if get_colour(neighbour) == T:
            successors.append(neighbour)
    set_colour(x,y,R)
    return successors

def G(state):
    return False
```

The S function here takes in just the state and will output a list of the successors of the inputted state that have the same colour T. Here, we are creating a list of states that can result from up,down,left and right movements from the initial position. We are then checking to ensure that the color of the new state matches the color T, of the initial state, and adding it to the solution set if it passes this condition. Since breadth-first search will call the S function every time it pops a path from the open list, we can just colour the state during the call to the S function.

The G function is the goal test of the flood fill algorithm and will take a state in as input, and produce a boolean output of true or false depending on whether a particular goal condition has been met. In essence, the goal test function is a condition for early termination of the algorithm. In this problem, we do not need an early termination clause, as we need to keep running the loop until all states that have been deemed appropriate and added to the open list are iterated through.

Q2.1. Below is the state-space representation for this problem:

- States:
  - These are the different positions that either Agent1 or Agent2 can occupy on the board. These provided locations will have x and y coordinates.
- Actions:
  - Moving Agent1 up, down, left, right or staying still
  - Moving Agent2 up, down, left, right or staying still
  - **NOTE:** Both agents are expected to move simultaneously, and both agents cannot stay still on the same turn (the overall board must change each iteration)
- Initial State:
  - Initial location (x_0, y_0) for both Agent1 and Agent2.
  - **NOTE:** Both agents are not allowed to occupy the same initial position.
- Desired Condition:
  - Agent1 and Agent2 can never occupy a blocked location, or exist on the same location at any given time.
  - Usually the goal test function represents an early termination condition for the game, but in this case there does not appear to be a goal. We just want the agents to move around into legal positions, as defined by the legality criteria below.

Below is the implemented pseudo-code of the S function for this problem:

NOTE: The solution below assumes that the state given as input to the S function is of the form (state1,state2) where state1 and state2 are tuples of the form $((x_1, y_1)$ and denote the current positions of the 2 agents.

```
def S(state):
    cs1, cs2 = state[0], state[1]
    succ1, succ2 = state[0][1], state[1][1]
    successors = []
    actions = [up, down, left, right, stay]
    for act1 in actions: #iterate through possible A1 actions
        for act2 in actions: #iterate through possible A2 actions
            if not blocked(cs1+act1) and not blocked(cs2+act2)
            and (cs1+act1)!=(cs2+act2) and ((act1!=stay) and (act2!=stay)):
                successors.append(cs1+act1, cs2+act2)
    return successors
```

As seen in the pseudo-code above, we are assuming that the state that is inputted into the S function is of a certain form, containing information regarding the current position of the 2 agents. Since agent1 and agent2 can each do 1 of 5 actions, there are 25 potential successor states that can be achieved. Prior to appending the successor states to the output list, we need to ensure that the new states for agent1 and agent2 are legal. We have defined a legal action as having to meet 3 criteria, shown below:

1. Blocked States:
   - We need to perform a quick check to ensure that neither the new state of agent1 or agent2 is in a location that is considered blocked.
2. Occupying the same location:
   - We must ensure that the new states occupied by agent1 and agent2 are not the same.
3. Stay action:
   - Since we need the overall state of the pacman game board to get updated with every iteration, we cannot have both agents perform stay operations.

If a combination of act1 and act2 passes the legality criteria, we then append the state of agent1 and agent2 that is achieved by actions act1 and act2 to the successors output list.

Q2.2 Below is the state-space representation for this problem:

- States:
  - These are the different positions that either Agent1 or Agent2 can occupy on the board. These provided locations will have x and y coordinates.
- Actions:
  - Moving Agent1 up, down, left, right or staying still
  - Moving Agent2 up, down, left, right or staying still
  - **NOTE:** Both agents are expected to move simultaneously, and both agents cannot stay still on the same turn (the overall board must change each iteration)
- Initial State:
  - Initial location (x_0, y_0) for both Agent1 and Agent2.
  - Initial Successor Lists for Agents 1 and 2. This list contains the order of successor states that the agents must visit.
  - **NOTE:** Both agents are not allowed to occupy the same initial position.
- Desired Condition:
  - Agent1 and Agent2 can never occupy a blocked location, or exist on the same location at any given time.
  - Agent1 and Agent2 cannot both execute stay actions at the same time.
  - Our overall goal condition is to ensure that both agents visit all successor states in their respective successor lists, in the orders determined by those lists.

Below is the implemented pseudo-code of the S function for this problem:

NOTE: The solution below assumes that the state given as input to the S function is of the form (state1,state2) where state1 and state2 are tuples of the form $((x_1, y_1), succ_1)$ where $succ_1$ is a list of the specific successors states for agent1.

```
def S(state):
    cs1, cs2 = state[0][0], state[1][0]
    succ1, succ2 = state[0][1], state[1][1]
    successors = []
    if (cs1==succ1[0]): #first item in succ1
        succ1.pop[0] #remove that term as it has been visited
    if (cs2==succ2[0]): #first item in succ2
        succ2.pop[0] #remove that term as it has been visited
    actions = [up, down, left, right, stay]
    for act1 in actions: #iterate through possible A1 actions
        for act2 in actions: #iterate through possible A2 actions
            if not blocked(cs1+act1) and not blocked(cs2+act2)
            and (cs1+act1)!=(cs2+act2) and ((act1!=stay) and (act2!=stay)):
                if (cs1+act1) in succ1:
                    if(cs1+act1)!=succ1[0]:
                        continue
                if (cs2+act2) in succ2:
                    if(cs2+act2)!=succ2[0]:
                        continue
                successors.append(((cs+act1),succ1), ((cs2+act2),succ2))
    return successors
```

As seen in the pseudo-code above, we are assuming that the state that is inputted into the S function is of a certain form, containing information regarding the current position of the 2 agents as well as lists of the successor states that we are required to reach. Our first check is to make sure that the state of Agent1 or Agent2 is not the next state in the list of required successor states (should be at the front of the succ1 and succ2 lists respectively). If they have reached this next state, we are popping the first elements from those 2 lists. We are then using a nested for loop to test the different combinations of action sets that can be made during a given turn. Since agent1 and agent2 can each do 1 of 5 actions, there are 25 potential successor states that can be achieved. Our first check within this nested for loop is to ensure that the action is legal. This is done by checking to see whether the new state for agent1 or agent2 is blocked, while ensuring that both agents1 are not trying to occupy the same state, and they are both not trying execute the stay command at the same time. If a combination of act1 and act2 passes this condition, we perform a check to see whether either agent1 or agent2's new states are in the list of successor states that we must achieve. Since the order of the traversal of these states is important, we are discarding combinations of act1 and act2 that leave either agent1 or agent2 in a required state that is not the next one to be visited. Thus, the successor states that ultimately get appended to the output list are those that either place agent1 and agent2 in valid, unique positions or in the next successor state to be visited as per the input list.

Q2.3 We wish to interrupt the search algorithm after it has performed the last movement operation. Since the new successor positions need to be visited after the original set of successor positions, we do not need to interrupt the previous solution during its execution. This new search is effectively another call to the search algorithm where the input is the final state of the 2 agents as per the previous call to the search algorithm, and the respective lists of new successor functions for agents 1 and 2.

Q3.1. As per the problem statement, we assume that the A^* algorithm terminates.
According to Theorem 2, A^* with an admissible heuristic is guaranteed to find an optimal solution if it exists. Let n be the optimal solution path, where the cost is $f(n) = C^*$.

According to Proposition 1, if a solution path n exists where $f(n) = C^*$ then either:

1. n has been expanded by A^* or
2. a prefix of n is on OPEN

Since we know that if n is strictly bounded by C^*, then every prefix k of solution path n also has cost f(k) ¡ C^*; it is also strictly bounded. Thus, every prefix k of the solution path must be expanded prior to n being expanded.

Q3.2 Assume that h1 and h2 are 2 heuristics, where h1 is strictly dominated by h2.

$$\forall n : h_1(n) < h_2(n)$$

Recall from the definition of the total path cost:

$$f(n) = g(n) + h(n)$$

Where g(n) is the cost of the path till n, and h(n) is the heuristic estimate of the path from n to the goal state. Therefore:

$$f_1(n) = g(n) + h_1(n) < f_2(n) = g(n) + h_2(n)$$

It can be seen that $f_1(n)$ is strictly bounded by $f_2(n)$. From the result of 3.1, we have shown that if this is the case, then A^* will expand all strictly $f_2(n)$ bounded nodes. Therefore, A^* with heuristic h_1 will expand at least as many nodes as A^* with heuristic h_2.