Homework 2

Name: CHITRADEVI MARUTHAVANAN
PSU ID: 950828319

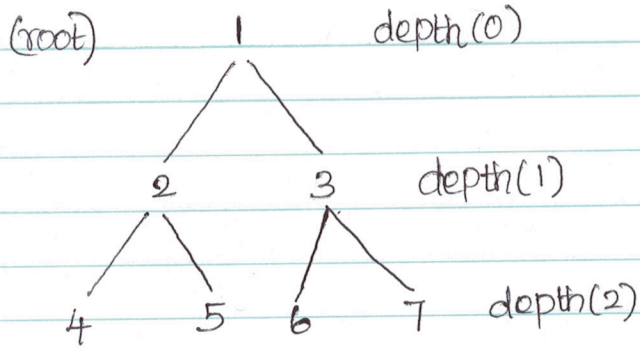
1(i) The proof can be demonstrated using mathematic induction

Base case:

* For a binary tree with depth 0 (ie., only a root node) the number of leaves is 1 ($2^0 = 1$).

* At depth d = 1, there are 2 ~~leaves~~ leaf nodes ($2^1 = 2$)
* At depth d = 2, there are 4 leaf nodes ($2^2 = 4$)

Therefore, as depth increases, the number of nodes doubles



Inductive Steps:

$N(i) = 2^i$ for $0 \leq i < d$ where d = depth, N(d) = number of nodes in depth 'd'

A perfect binary tree with depth d has two binary trees of depth d-1

$$N(d) = 2 * N^{d-1}$$
$$= 2 * 2^{d-1}$$
$$= 2^{d-1+1} = 2^d$$

Hence the maximum number of leaves in a binary tree of depth d is 2^d.

1 ii) Let the number of nodes in a binary tree of depth d be 2^d

For binary tree of depth 0, it has exactly one node - root node.

$$N(0) = 2^{0+1} - 1$$
$$= 2^1 - 1 = 1$$

where $2^d = 2^{d+1} - 1$

Assume by induction, N(d) = number of nodes in a perfect binary tree of depth 'd'

$N(i) = 2^{i+1} - 1$ for $0 \le i \le d$

we know that, a binary tree of depth d consists of two perfect binary trees of depth d-1 along with a root node

$$N(d) = 2 * N(d-1) + 1$$
$$= 2 * (2^{d+1-1} - 1) + 1$$
$$= 2 * 2^d - 2 + 1$$
$$N(d) = 2^{d+1} - 1$$

∴ The maximum number of nodes in a binary tree of depth d is $2^{d+1} - 1$

2. a) In this scenario, the robot begins at the center of the maze facing north. It can turn to face north, south, east or west and move forward a certain distance until it hits a wall.

To simplify the problem, we use a coordinate system where the ~~system~~ center is represented by (0,0). The maze is shaped like a square, with the boundaries ranging from (-1,-1) to (1,1).

The robot starts at the coordinate (0,0) facing north and the goal is to reach a state (x,y) where either $|x| > 1$ or $|y| \ge 1$. The robot can move in four different directions.

Hence, the state space is 4n, where n represents the size of location.

2b. When the robot is placed in the center, facing north, the state space for a location of size n is 4n. At intersections, the robot will turn, causing the state space to be represented by 4k. In the case of an intersection, the robot will change direction, but if there are no intersections, then k is included in n. For the remaining (n-k) sites, the robot moves straight, either moving forward or stopping if there is no turn. The overall state space can be represented as 4n, with 4k+2(n-k) states.

2c, No, we are no longer required to maintain track of the robots orientation. Predicting the outcome of our acts is pointless. However, the new actions involve the turning motion. The system must be aware of the robot's current orientation in order to determine when to rotate

2d) The simplifications are:

* Robots that face only 4 directions
* Robots can work as long as possible without any recharge
* Ignoring the height of the robot off the ground, whether it is tilted off the vertical
* Robots can't get stuck or hit the wall

3a) In this search problem, all the possible city pairs are shown as (i,j). All the pairs of (x,y) such that adjacent (x,i) and (y,j) can be the successors of (i,j). The cost to go from the point (i,j) to the point (x,y) is max(d(i,x), d(j,y)).

3b. (i) and (ii) are both acceptable. The two friends would take equal-sized steps straight to each other, reducing their separation by twice the cost of each step.

3c. Yes, there are maps that are completely connected yet have no solutions. If the two friends start an odd number of steps apart, they will always swap places.

3d. Yes, there are maps in which all solutions require one friend to visit the same city twice. If the two friends start with an odd number of step apart, a move in which one of the friends take the self-loop changes the distance by 1, rendering the problem to be solvable.

4. The initial configuration of the 8 puzzle problem reveals that not all final configurations can be reached. The collection of all possible configurations in the 8 puzzle problem divides into two equal classes that are equivalent to each other. Any two configurations within the same class can be achieved from one another.

When we implement the search for an 8 puzzle solution, it is important prior to executing the search that we determine whether a final configuration is reachable. In order to determine when a configuration is reachable from an initial configuration, one standard approach is to show that the two states have the same parity with respect to the "total number of inversions". Given a board, an inversion is any pair of blocks i and j, where $i<j$ but i appears after j, when considering the board in row-major order

Eg:

For counting the total number of inversions for an instance of the 8 puzzle problem, the key result is that, a state S-1 is reachable from another state S-0, if they agree in the parity when counting the number of inversions. For each state even they are both even or both odd, for the example given, the right hand state is not reachable from the left hand state because the states have been even and odd parities with respect to the total number of inversions

1	2	3
–	4	6
8	5	7

No. of inversion = 3

–	1	3
4	2	5
7	8	6

No of inversion = 4

5 a) False

Depth first search may expand d nodes exactly to reach the goal, but it does not expand as many nodes in order to reach the goal state. A* search is better than Depth first search.

b) True

h(n) = 0 is an admissible heuristic for the 8 puzzle because the cost is non negative.

c) True

A* search is of no use in Robitics because the space can be discretized or skeletonized.

d) True

Breadth first search depends on the depth of the solution but not on the cost of the step.

e) False

The manhattan distance from the start to finish is 8, where a rook can move across the board in the move one.

6. Consider a single goal such that every state has a single successor and a single goal at depth n. The depth first search will find the goal in n steps, whereas iterative deepening search performs much worse. It will take

$$1 + 2 + 3 + \cdots + n = O(n^2) \text{ steps}$$

7. Given:

$$f(n) = (2 - w)\, g(n) + w h(n)$$

If $w = 0$,

$$f(n) = (2 - 0)\, g(n) + 0 \cdot h(n)$$
$$= 2g(n)$$

It behaves similarly to uniform cost search.

If $w = 1$,

$$f(n) = (2 - 1)\, g(n) + 1 \cdot h(n)$$
$$= g(n) + h(n)$$

It performs similar to A* search

If $w = 2$

$$f(n) = (2 - 2)\, g(n) + 2 \cdot h(n)$$
$$= 2h(n)$$

It performs similar to Greedy Best First Search.

It is said to be complete $0 \leq w < 2$

For $f(n) = (2-w)g(n) + wh(n)$ which behaves exactly like A* with heuristic value. Here $f(n)$ is always less than $h(n)$. Here this is admissible.

8 a) The branching factor b in this state space is 4
b) There are $4k$ distance states at depth 'k'
c) The maximum number of nodes expanded by Breadth First Search is $[((4^{x+y+1}-1)/3)-1]$
d) The maximum number of nodes expanded by Breadth first Search is $2(a+b)(a+b+1)-1$
e) Yes, $h = |u-x| + |v-y|$ is manhattan admissible
f) False, All the nodes might be expanded in the worst case since there are quadratically many of them
g) Yes, h remains admissible if some links are removed. Here, removing the links requires more steps
h) No, h does not remain admissible if some links are added between non-adjacent states. Because the non local links can reduce the actual path length below the manhattan distance

9 a) Hill Climbing search with $K=1$
b) Breadth First Search with one initial state and no limit on the number of states retained.
c) Hill Climbing with simulated annealing with $T=0$ at all times ignoring the termination test
d) Random walk search with $T=\infty$ at all times
e) Random walk search with population size $N=1$

10. *The AND-OR solution Graph, each state information will be recorded (information about finding a solution and the solution found).

* It checks for repeated states - if it has visited state more than once then it gives the solution immediately
* If the solution is not there then it will decide whether the state can solve then path taken. Then the value of the path is recorded.
* The search algo will retuen 'failure' if a path in any subset of its current value of a path that failed pseviously
* The sub-plans will be avoided by recording the labels and traveling the solutions
* If the state are being visited repeatedly it will retuen the labels which are stored

11 Initial belief state is {1, 2, 3, 4, 5, 6, 7, 8}
In sensor-less version of vaccuum world, Agent only knows geography of environment but it doesnot know its location or distribution of dirt
Action outcomes:

[Right] : possible Successor state {2, 4, 6, 8}
[Right, Suck] : {4, 8}
[Right, Suck, Left, Suck] : {7}

when the agent reaches dirty square the action cleans the dirty square only if the problem is fully observable
The problem is unsolvable because there is nopath that leads to belief state whose elements satisfies the goal.

12. This can be solved by deepening search; it can perform a back track from a previous state if we can observe the successor states. Inaddition to its current state the agent can observe all the

successor states and the actions that would lead to them
Online DFS starting at (0,0) will not reach (1,-1)
(0,0) (1,0) (-1,0) (0,0)
The given unit vectors can be expanded by this algo.
The states it visits in reaching (1,-1) are:
(0,0) (0,0) (0,-1) (0,0) (1,0) (2,0) (1,0) (0,0)
(1,0) (1,1) (1,0) (1,-1).

13 Given function
$f(x,y) = 5x^2 + 40x + y^2 - 12y + 127$

Apply gradient descent (GD) to find the global minimum of the function $f(x,y)$

```
# importing necessary libraries
import numpy as np
# define the gn function
def f(x,y):
    return 5*x**2 + 40*x + y**2 - 12*y + 127
def grad_f(x,y):
    return np.array([10*x + 40, 2*y - 12])
# x0 is random

def gradient_descent(x0, eta, iterations=500):
    x_prev = x0
    # calculating Xt
    for i in range(iterations):
        x_new = x_prev - eta * grad_f(x_prev[0], x_prev[1])
        # change negative to positive to find global_max
        x_prev = x_new
    return x_prev
def run_trials_and_report_best(eta, number_of_trials=10):
    best_x = None
```

```
for trial in range(number_of_trials):
    x0 = np.random.uniform(low=-10, high=10, size=(2,))
    x = gradient_descent(x0, eta)
    if (best_x is None) or (f(x[0], x[1]) < f(best_x[0],
                                              best_x[1])):
        best_x = x
return x
for exp in range(3):
    print('Experiment: {exp+1}')
    for eta in [0.1, 0.01, 0.001]:
        x = run_trials_and_report_best(eta)
        print('\t η: {eta}\n\t\tX: {(x[0], x[1])}\n\t\t
                                    f(x,y): {f(x[0], x[1])}')
```

The code here will yield us the global minimum using GD. The algo moves faster for larger η and finds more negative values. By changing the negative to positive GD fun gradientdescent(), we will find the local minimum as shown below.

```
experiment:1
        η: 0.1
                X: (13.540756325467783, 14.583788225053219)
                f(x,y): 480.6323051566221
        η: 0.01
                X: (5.540189756994667, 3.2173460503749576)
                f(x,y): 59.97734011315354
        η: 0.001
                X: (5.795784476550911, 0.7933316407770859)
                f(x,y): 27.379102339351043
experiment:2
        η: 0.1
                X: (-8.807788820049745, 3.042333801584582)
                f(x,y): 190.10282087044453
        η: 0.01
                X: (-4.572208141336156, 1.1025583276650375)
                f(x,y): 80.57955427827146
        η: 0.001
                X: (-0.8282848087201322, 4.4636429553868275)
                f(x,y): 90.21630256372006
experiment:3
        η: 0.1
                X: (8.168902380772288, -3.5321346016653186)
                f(x,y): 14.87218064958165
        η: 0.01
                X: (-4.590272563172256, 5.555954914749899)
                f(x,y): 155.04042174547806
        η: 0.001
                X: (-5.7021738937862825, -8.138507232499476)
                f(x,y): 71.57805812614704
```