Submission instruction: Students will submit the assignment using an online system. The student will receive the submission link via email. Student should submit the homework in a single document, in the same format as the assignment. There will be enough space for students to answer the questions in the assignment document. Make sure the page numbers are in proper order. Failure to follow instructions will result in delay and/or errors in grading.

Please do not use additional pages.

Due Date: February 10th

- **Late homework:** you lose 20% of the homework's grade per 24-hour period that you are late. Beware, the penalty grows very fast: grade = points * (1 – n * 0.2) where n is the number of days late (n=0 if submitted on time, n=1 is submitted between 1 second and 24h late, etc).

- **Grade review/adjustment:** Requests will be considered up to 2 weeks after the grade is released. After that, it will be too late and requests for grading review will be denied.

- Homework assignments are to be solved **individually.**

- You are welcome to discuss class material in review groups, but do not discuss how to solve the homework.

Question-1 [20 Points]:

For each of the following activities, give a PEAS description of the task environment and characterize it in terms of the properties listed in Section 2.3.2. State your assumptions clearly if any.

- [5 Points] Playing soccer
- [5 Points] Bidding on an item at an auction
- [5 Points] Performing a high jump
- [5 Points] Knitting a sweater

1. playing Scoccer: P win the game or not

E: field, players, ball, weather, judges

A: foot, leg, head, arm ... body parts of robot

S: eyes, ears (visual, sound detector)

properties: partially observable, stochastic, sequential, dynamic

continuous, unknown

2. Bidding on an item at an aution

P: bid result, win the item we want or not

E: Other bidders, item to bid on, host for the bid, action house

A: mouth (voice system), hands (or body behavior) that can make response to bid or not

S: ears, hear others' bid, eyes: to see the item, observe the environment

Properties: partially observable, stochastic, episodic, dynamic

discrete and multi-agent, known

3. Performing a high jump.

P: successful jump over the desired height or not

E: The ground for jumping, weather (if outdoors), the rod, the mattress for landing.

A: body parts like leg, feet, arms that can perform the jump

S: body parts like eyes, can see how far to run, how high to jump. maybe use ears to follow instructions

Properties: fully observable. stochastic, sequential, dynamic, continuous. single-agent, unknown

4. Knitting a sweater.

P: Actual sweater is done and can fit into a person or not

E: The tools for knitting, the place for knitting, sketch needed of the sweater

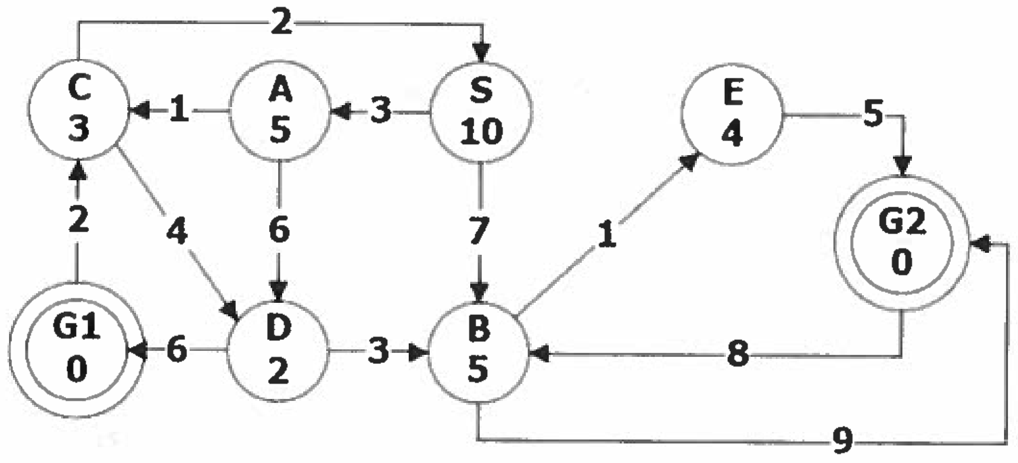
A: hands for knitting.

S: eyes, ears to see the current process and follow others' advice.

Properties: fully observable. stochastic, sequential, static, discrete single-agent, unknown

Question-2 [25 Points]:

Assume that you have the following search graph, where *S* is the start node and *G1* and *G2* are goal nodes. Edges are labeled with the cost of traversing them and the estimated cost to a goal is reported inside nodes.



For each of the search strategies listed below,

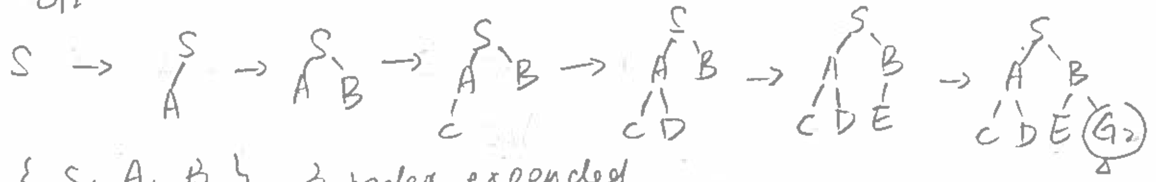
a) Indicate which goal state is reached if any
b) List, in order, the states expanded (Break the ties in alphabetical order)
c) State the number of visited (i.e. expanded) nodes (including goal node)

1. [5 Points] *Breath-first*
2. [5 Points] *Depth-first*
3. [5 Points] *Bidirectional (Start from S, and both G1 and G2)*
4. [5 Points] *Best-first (using f=h)*
5. [5 Points] *A* (using f=g+h)*

1 Breath-first Search

(a) G_2

(b)

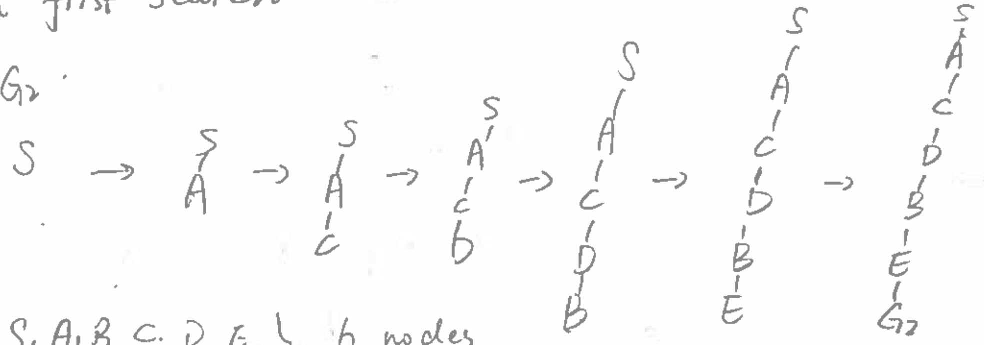


(c): { S, A, B } 3 nodes expended

2. Depth-first Search

(a) G_2

(b)

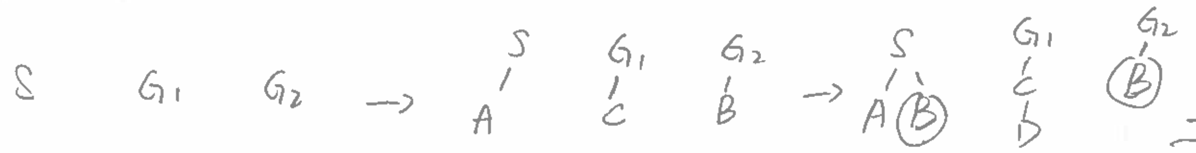


(c) { S, A, B, C, D, E } 6 nodes expended

3. Bidirectional

(a) G_2

(b)



common node B exist, found G_2

(c) { S, G_1, G_2, C } 4 nodes expended

4. Best Search

ⓐ G_1

ⓑ
$$\begin{tikzcd}[row sep=small]
S & {} \arrow[r] & S \arrow[d, no head] & {} \arrow[r] & S \arrow[d, no head] & {} \arrow[r] & S \arrow[d, no head] \\
 & & A & & A \arrow[d, no head] & & A \arrow[d, no head] \\
 & & & & D & & D \arrow[d, no head] \\
 & & & & & & G_1
\end{tikzcd}$$

ⓒ { S. A. D } 3 nodes expanded

5. A*

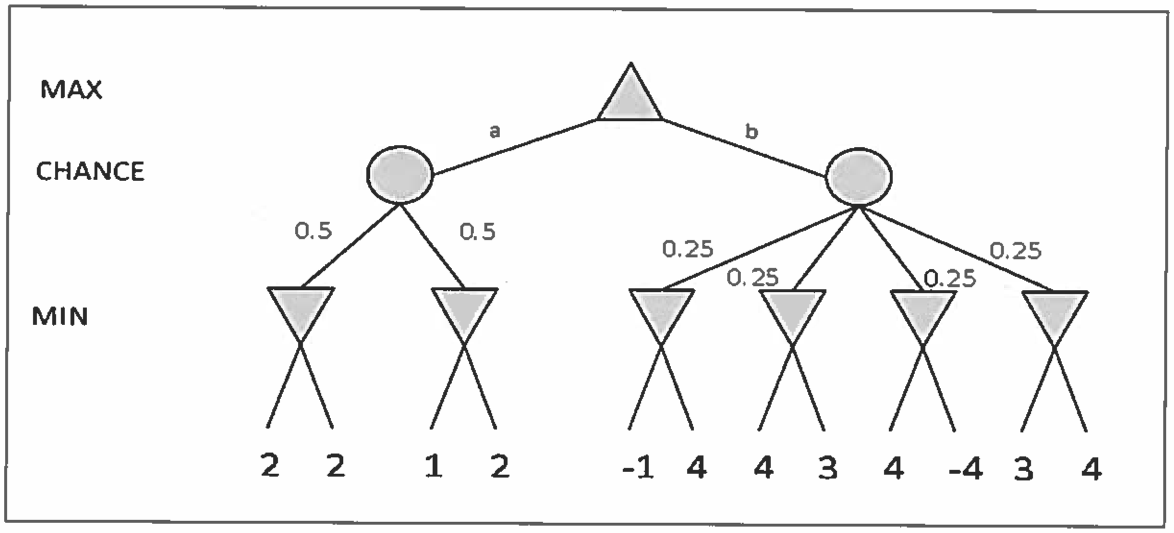
ⓐ G_1

ⓑ
$$\begin{tikzcd}[row sep=small]
S & {} \arrow[r] & S \arrow[d, no head] & {} \arrow[r] & S \arrow[d, no head] & {} \arrow[r] & S \arrow[d, no head] & {} \arrow[r] & S \arrow[d, no head] \\
 & & A & & A \arrow[d, no head] & & A \arrow[d, no head] & & A \arrow[d, no head] \\
 & & & & C & & C \arrow[d, no head] & & C \arrow[d, no head] \\
 & & & & & & D & & D \arrow[d, no head] \\
 & & & & & & & & G_1
\end{tikzcd}$$

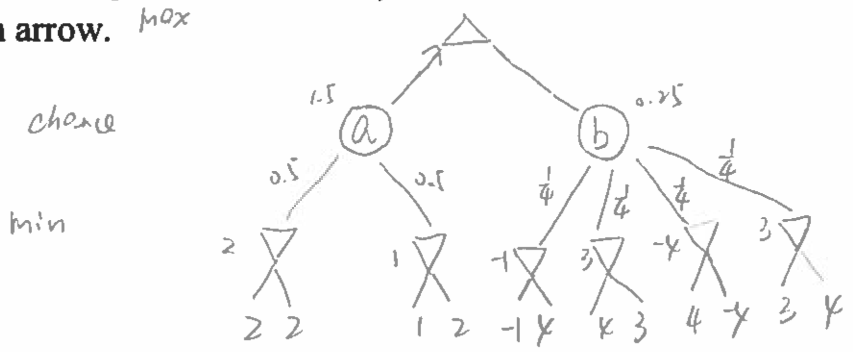
ⓒ { S. A. C. D } 4 nodes expended.

Question-3 [20 points]:

In this question we tackle the problem of stochastic games. We will be using the EXPECTIMINIMAX(s) function described in section 5.5 in the book to evaluate the value for the MAX node. The figure below shows the complete game tree. Assume that the evaluation order of the leaf nodes are from left-to-right order, and before the leaf node is evaluated, we do not know the value of that leaf node or any other node to right side of that leaf node. You may assume that the possible range of values is -4 to 4.



a) [4 Points] Copy the figure, mark the value of all the internal nodes (including the chance node), and indicate the best move to the root with an arrow.



a: 0.5 × 2 + 0.5 × 1 = 1.5

b: $\frac{1}{4}$ × (-1 + 3 - 4 + 3) = 0.25

The best move is from a to root

b) [4 Points] Given the values of the first eight leaves (counting from left to right), do we need to evaluate the ninth and tenth leaves? Explain why.

Given first 8 value, we have currently

$a = (2+1) \times 0.5 = 1.5$

$b = (-1+3) \times 0.25 = 0.5$

In order for b to beat a, the rest of b's value should be larger than $1.5 - 0.5 = 1$.

Suppose all the leaf node left is 4 (which is the maximum value), we get $(4+4) \times \frac{1}{4} = 2$, which is larger than 1. That means we still have a chance to beat a.

Therefore we need to evaluate 9th and 10th leaf

c) [4 Points] After the first two leaves are evaluated, what is the value range for the left-hand chance node?

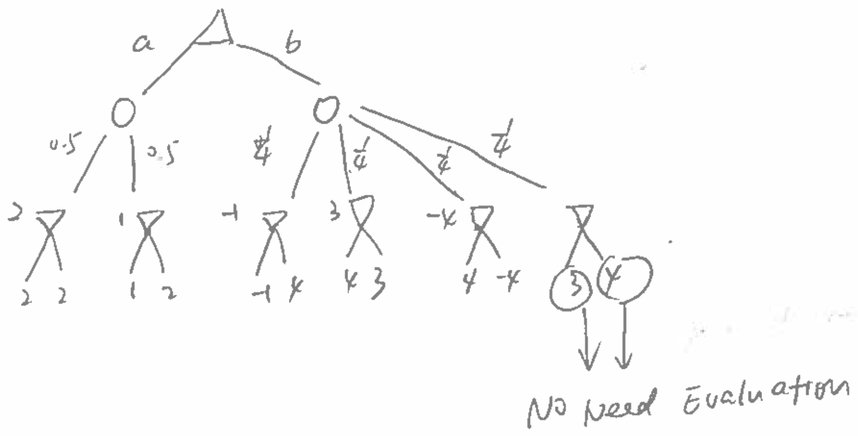
After evaluating first 2 leaves, a get $0.5 \times 2 = 1$

The lowest value for rest of a would be $-4 \times 0.5 = -2$

The highest value for rest of a would be $4 \times 0.5 = 2$

Put all together, range for a is -1 to 3.

d) [4 Points] Circle all the leaves that need not be evaluated assuming the evaluation order of left-to-right. Explain why.



The 11th and 12th node need no evaluation.

When evaluating the 10-th Node which is -4, we get

$a = 1.5$

$b = (-1+3-4) \times \frac{1}{4} = -\frac{1}{2}$

The largest value b can get is $-\frac{1}{2} + 4 \times \frac{1}{4} = \frac{1}{2} < 1.5$

Therefore there's no need to evaluate rest of leaves. a is the best choice

Question-4 [15 Points]:

Given a set of locations and distances between them, the goal of the Traveling Salesman Problem (TSP) is to find a shortest tour that visits each location exactly once. We would like to solve the TSP problem using a greedy hill-climbing algorithm. Each state corresponds to a permutation of all the locations (called a *tour*). The operator *neighbors(s)* generates all neighboring states of state s by swapping two locations. For example, if s = <A-B-C> is a tour, then <B-A-C>, <C-B-A> and <A-C-B> are the three neighbors generated by neighbors(s). We can set the evaluation function for a state to be the total distance of the tour where each pair wise distance is looked up from a distance matrix. Assume that ties in the evaluation function are broken randomly. (Note: We don't consider a tour as returning to the start city.)

a) [3 Points] If you have *n* locations, how many neighboring states does the *neighbors(s)* function produce?

for each state, could generate following number of neighbours.

$$(n-1) + (n-2) + (n-3) + \cdots + 2 + 1$$

$$= \frac{(1 + (n-1)) \times (n-1)}{2}$$

$$= \frac{n^2 - n}{2}$$

b) [3 Points] What is the total size of the search space, i.e. how many possible states are there in total? Assume again that there are *n* locations.

Total size of states is the number of permutations of n locations.

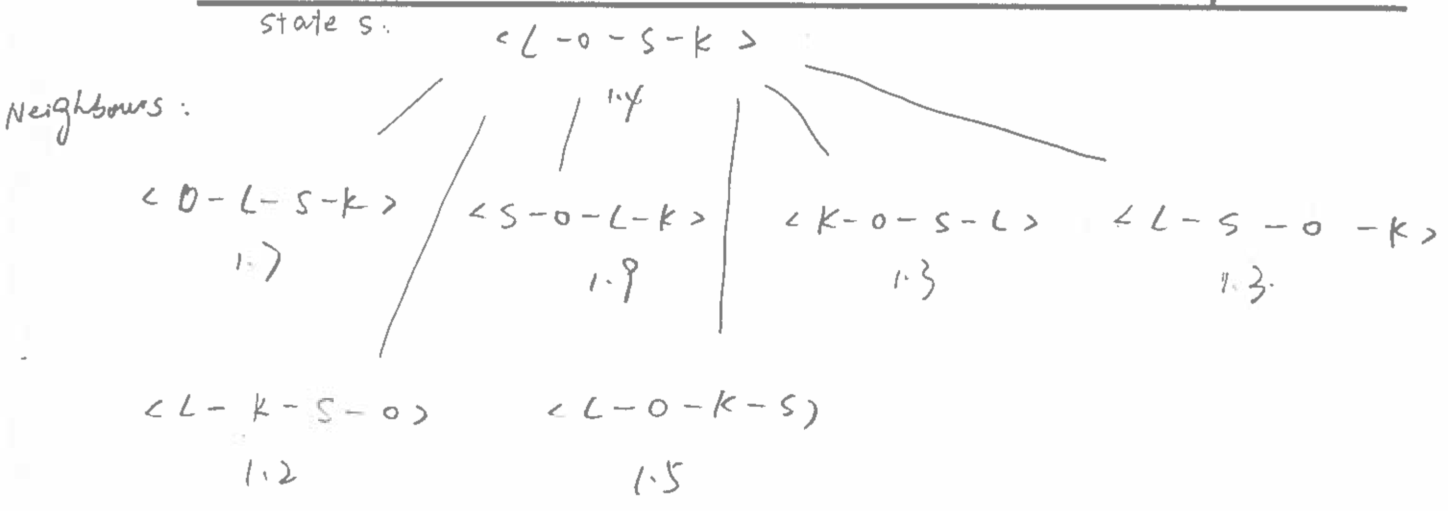
$n*(n-1)*(n-2)*(n-3)*\cdots*2*1$

$= n!$

c) [9 Points] Imagine that a student wants to hand out fliers about an upcoming programming contest. The student wants to visit the Leavey Library (L), Salvatori Computer Science Center (S), Olin Hall (O), and Kaprielian Hall (K) to deliver the fliers. The goal is to find a tour as short as possible. The distance matrix between these locations is given as follows:

	L	**S**	**O**	**K**
L	0	0.6	0.9	0.7
S	0.6	0	0.3	0.2
O	0.9	0.3	0	0.4
K	0.7	0.2	0.4	0

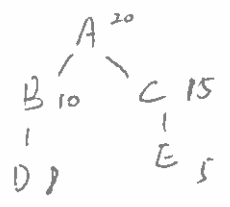
The student starts applying hill-climbing algorithm from the initial state: <L-O-S-K>. What is the next state reached by hill-climbing, or explain why there is no neighboring state. When will we know if we should stop or continue the search? Will we know if the state is a global optimal solution when we stop? Briefly explain your answers.



i) As we can see above, < L - K - S - O >, which is one of initial state's neighbour has lower cost of path 1.2 < 1.4. Therefore next step should choose the state < L - K - S - O >.

(ii) We stop until no neighbour of current state has a lower value cost. If there exist a neighbour that has lower cost path, we change the state into the neighbor one.

(iii) We cannot ensure our solution is global optimal. Given the example:



Suppose we are to find minimum value of the tree. When we at state A. we compare the value B and C. And choose to continue search through B as B < C. We ignore the C state, but actually its child E has lowest value of all. We can only get D = 8 as local minimal.

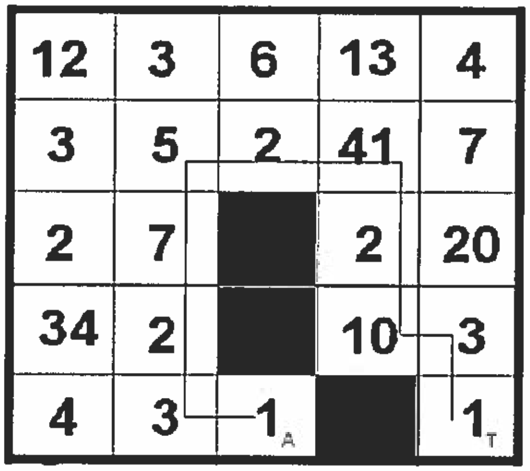
Question-5 [20 Points]:

In this programming problem, you are required to get familiar with *python* programming language and debug an algorithm implemented in python. We will be solving a navigation problem of a agent in a grid world.

Problem Description:

Consider grid-worlds with square cells. Cells are either blocked or unblocked. The start cell of the agent is unblocked. The agent can only move from its current cell to one of the four adjacent cells within the proximity of the grid at each step (no diagonal moves allowed), the cost of move to blocked cells is infinity (which means the agent cannot move across the blocked cell), for unblocked cell, the cost is as given in the input grid. Its objective is to move from its start cell to a given goal cell minimizing the cost. The path cost is calculated as the sum of the cost of traversing all the cells in the path. The path cost for the path shown (red line) in the figure below is 76.

The following figure shows an example of 5*5 grid terrain with black cells referring to the blocked cells, white cells referring to the unblocked cell. The number inside each cell indicates the cost to reach the cell. Here, Start cell = Agent (A). Goal cell = Target (T).



17 + 43 + 12 + 1 + 3
60 + 16 = 76

For the problem above, you are given a python program that implements *Breath First Search (BFS)* and *Uniform Cost Search (UCS)* search strategies. You are required to analyze BFS in order to get familiar with programming in python. Then you need to debug the UCS function implemented in the program in order for it to function correctly.

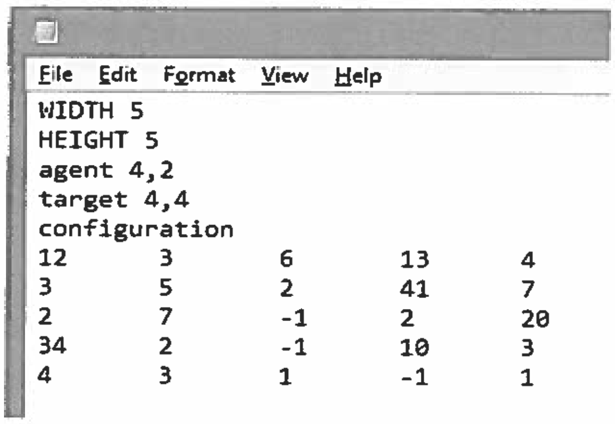
Files:

searchPath.py: python program to solve the navigation problem

inputfilex.txt: input files to the python program describing the grid environment. '*x*' indicates the input file number

Input:

6 input files are provided, which are named from "inputfile0.txt" to "inputfile5.txt". The format of a sample input file is given below.



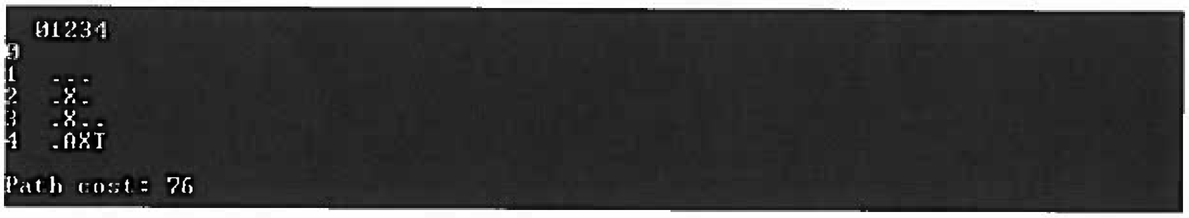
```
WIDTH 5
HEIGHT 5
agent 4,2
target 4,4
configuration
12	3	6	13	4
3	5	2	41	7
2	7	-1	2	20
34	2	-1	10	3
4	3	1	-1	1
```

The size of the input grid is defined by "WIDTH" and "HEIGHT" which corresponds to the number of cells in each row and the number of cells in each column respectively. Cell(0,0) is top-left cell of the input grid. The (x,y) coordinates of the agent and target are given in the third and fourth line respectively, with x presents the vertical position, y presents the horizontal position. The blocked cells are shown as "-1". The unblocked cells are shown as positive numbers which also indicate the costs to traverse them. The numbers in the same line are separated by tab.

Output:

The program shows the path between the agent cell and the target cell. Blocked cells are displayed as “X”, the Agent cell as “A”, the target cell as “T” and the cells on the path are displayed by “.”. Other cells are displayed as single white space. In the end the program displays the cost of traversing from agent cell to target cell through the path determined by the search strategy selected. The following figure shows an example of the output:

X
A
T
.



```
 01234
0
1  ...
2  .X.
3  .X..
4  .AXT

Path cost: 76
```

Running the program:

To run the program, use the following command:
python searchPath.py –i *inputfile_name* –s *search_strategy*
where,
inputfile_name is the name of the input file.
search_strategy is the selected search strategy (1=BFS and 2=UCS)

The following command shows an example where it selects “inputfile0.txt” as the input file and BFS as the search strategy:
python searchPath.py –i inputfile0.txt –s 1

NOTES:
1) You can download the python program and input files from the course website (https://courses.uscden.net/d2l/home/7593). Under the “Content” tab find “Homework”. Under that you will find all the files in “HW1” folder.
2) We use Python 2.7 for this implementation. (https://www.python.org/download/releases/2.7/)
3) The program assumes that the input files are in the same folder as the program.

Answer the following questions:

a) [10 Points] Debug the UCS function in the program to make it function as described in Figure 3.14 in the book. Indicate the modification/addition to the program in terms of python code. Use the line number in the program as the reference to where you would modify/add code. Explain why each modification/addition was required.

1. Line 122: change frontierCost definition from an array to a dictionary. The key is the hashed node, the value is the shortest distance from initial state to this node.
   All the operation about frontierCost is therefore changed.

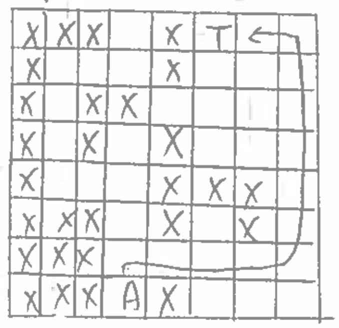
2. Line 139: Delete "break" after find the target. For UCS, find the target does not guarentee the optimal solution, therefore we need to keep searching

3. Line 132-142, 145-155, 158-168, 171-181 changed into a function called Expandable, content in the function also changed. My idea is if child is not in frontier and expandable, append the child, calculate the hash value for frontierCost. Add current node as father. Else if the child has explored or is in the frontier, and the new path cost to it is smaller than previous one change the frontierCost(childHash) to new value and new parent. Append the child in the frontier again in case any other path cost would been affected by this update. (Expandable is from line 161 to line 172 in the new file,

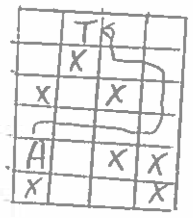
4. Line 128: Add function Reorder, it aims to always find the smallest cost value for frontier to pop. (Reorder is from 176 to 184 in the new file)

b) [6 Points] For each input file, show the path from Agent cell to target cell and report the path cost for the debugged UCS function.

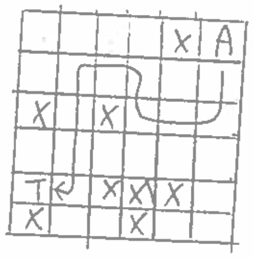
Input 1 path Cost 13.



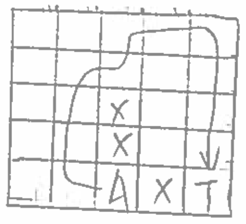
Input 4 path cost 60



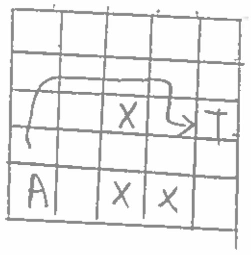
Input 5 path cost 60



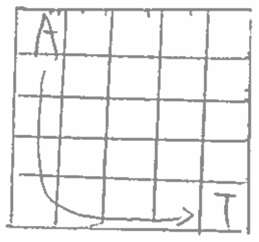
Input 2 path Cost 73.



Input 3 path Cost 38



Input 0 path cost 8



A* can be thought as an enhanced version of UCS that makes use of additional knowledge for estimating the cost of a path heuristics. If we were to convert the UCS program to do A* search, we would need to find heuristic for estimating the cost in this grid environment. With that in mind:

c) [2 Points] Give a simple argument that shows that the heuristic function directly using Manhattan Distance is consistent.

Manhattan distance is the shortest path between two nodes on a grid. Therefore whenever passing a node that is not optimal to the goal, the distance would increase

d) [2 Points] Design one admissible heuristic function other than using Manhattan Distance.

One possible heuristic is vertical grid distance to target, which is also the shortest vertical grid number to cross in order to reach goal.

As shortest vertical distance is always equal or smaller than actual distance, even if all costs are 1, actual distance may include horizontal distance. Therefore it's admissible