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.

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

Playing soccer:

P – Win/Lose
E – Soccer field
A – Legs, Head, Upper body
S – Eyes/Camera, Ears/Receiver

Bidding on an item at an auction:

P – Item acquired, Final price paid for the item
E – Auction house/online
A – Bidding
S – Eyes/Camera, Ears/Receiver

Performing a high jump:

P – Clearing the jump or not
E – Track
A – Legs, Body
S – Eyes/Camera

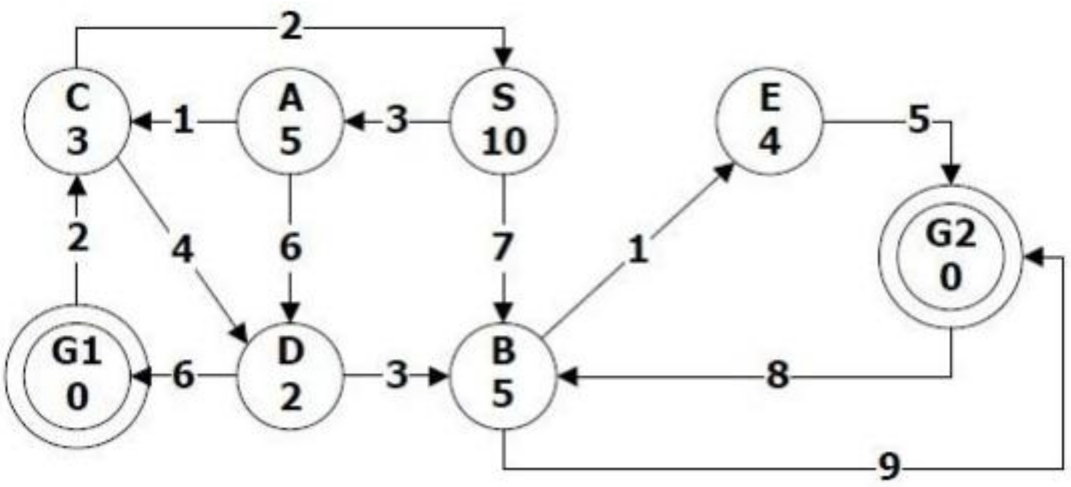
Knitting a sweater:

P – Quality of resulting sweater
E – Rocking chair
A – Hands, Needles
S – Eyes/Camera

	Playing soccer	**Bidding on an item**	**Performing a high jump**	**Knitting a sweater**
Observable	Partially observable	Partially observable	Fully observable	Fully observable
Agents	Multi-agent	Multi-agent	Single agent	Single agent
Deterministic	Stochastic	Stochastic	Stochastic	Stochastic
Episodic	Sequential	Episodic	Sequential	Sequential
Static	Dynamic	Dynamic	Dynamic	Dynamic
Discrete	Continuous	Continuous	Continuous	Continuous

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

a) G2
b) S, A, B (optimized) or S, A, B, C, D, E, G2
c) 3 (optimized) or 7

2. Depth-first

a) G1
b) S, A, C, D (optimized) or S, A, C, D, G1
c) 4 (optimized) or 5

3. Bidirectional

a) G2
b) S, G1, G2
c) 3

4. Best-first

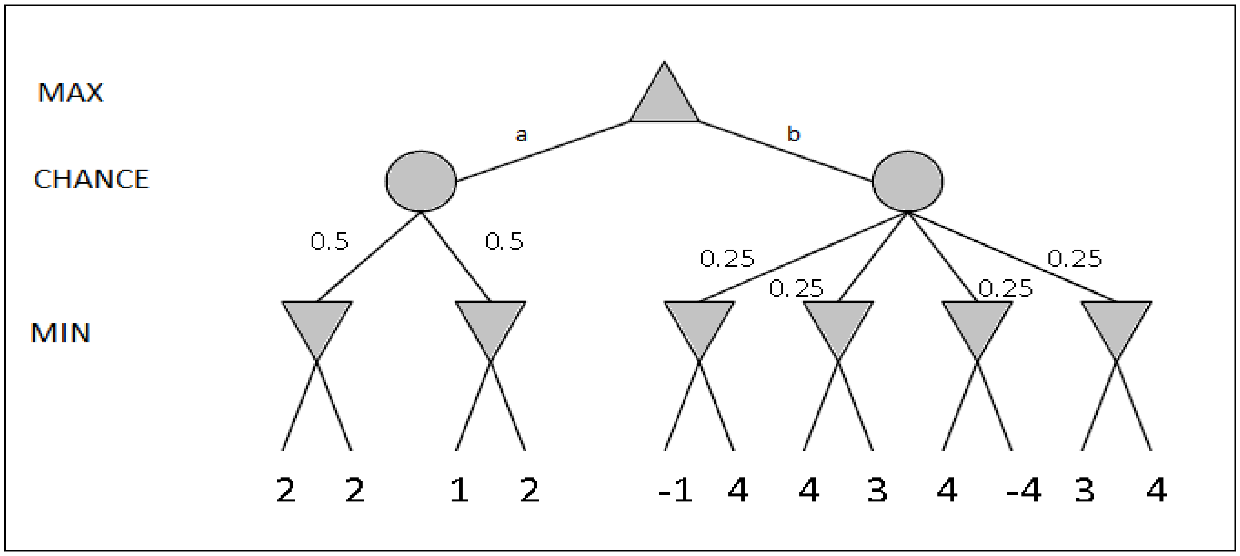
a) G1
b) S, A, D (optimized) or S, A, D, G1
c) 3 (optimized) or 4

5. A*

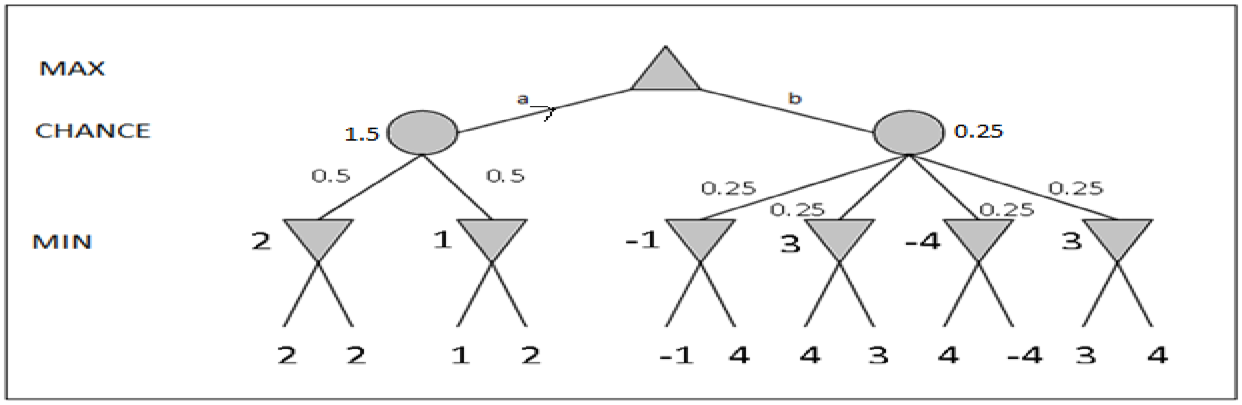
a) G2
b) S, A, C, D, B, E, G2
c) 7

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.



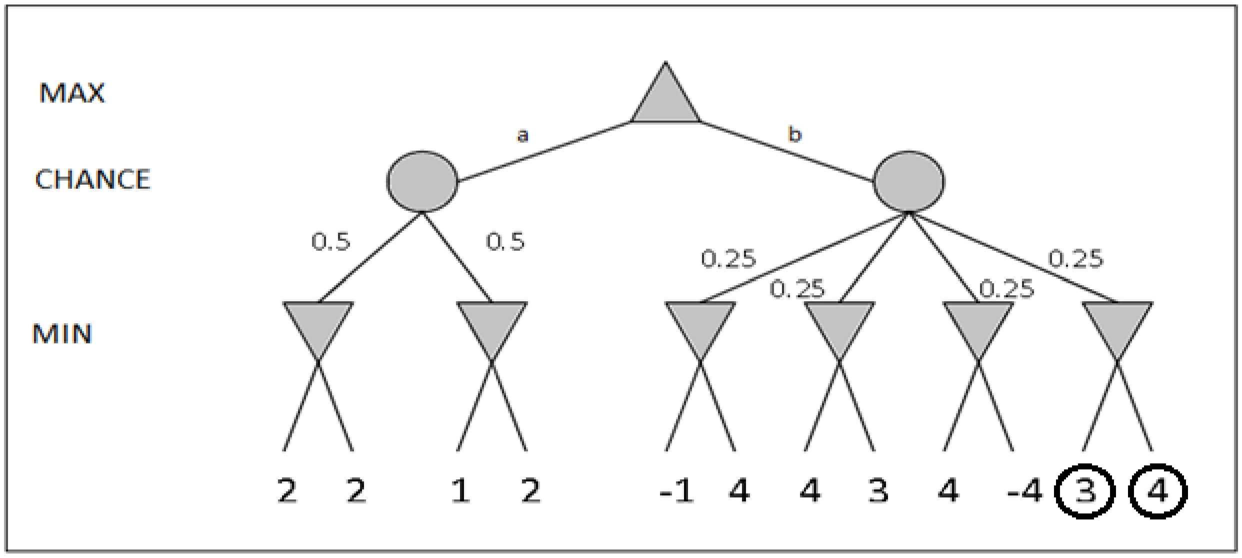
b) [4 Points] Given the values of the first eight leaves (counting from left to right), do we need to evaluate the ninth and tenth leaves? Explain why.

If we evaluate the first 8 leaves, then the weighted average for left chance node is 1.5. Now if we calculate the weighted average of the first 4 leaves of the right chance node, the value comes to be 0.5. So, if we get value 4 for all the remaining leaves i.e. 9th to 12th, the weighted average for right chance node would be 2.5 which would be greater than left chance node. So we need to evaluate 9th and 10th leaves.

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

The range would be [-1, 3].

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



After evaluating first 10 leaves, we get 1.5 for left chance node and -0.5 for right chance node. So, even if we get 4 (i.e. maximum) for 11th and 12th leaves, the weighted average for right chance node would go as high as 0.5 which would still be less than the left chance node. So once we evaluate first 10 leaves, we need not evaluate 11th and 12th leaves.

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?

This is a combination problem, so if we have n locations in the current state, we pick two locations out of n and swap them. So, we have C_2^n = n*(n-1)/2 neighbors.

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.

This is a permutation problem so the answer is P_n^n = n!. Assuming the start city is arbitrary and a tour that does not return to the start city. If the start city is given, then (n-1)! is okay. If you consider A-B-C and C-B-A as the same, n!/2 is also acceptable.

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.

For the initial state <L-O-S-K> which has a cost 0.9 + 0.3 + 0.2 = 1.4, we have 6 successors:

<O-L-S-K>: 0.9 + 0.6 + 0.2 = 1.7

<S-O-L-K>: 0.3 + 0.9 + 0.7 = 1.9

<K-O-S-L>: 0.4 + 0.3 + 0.6 = 1.3

<L-S-O-K>: 0.6 + 0.3 + 0.4 = 1.3

<L-K-S-O>: 0.7 + 0.2 + 0.3 = 1.2

<L-O-K-S>: 0.9 + 0.4 + 0.2 = 1.5

So, choose <L-K-S-O>

The algorithm stops if no neighbor has lower distance.

The solution is, in general, a local optimal one, not a global optimal solution.

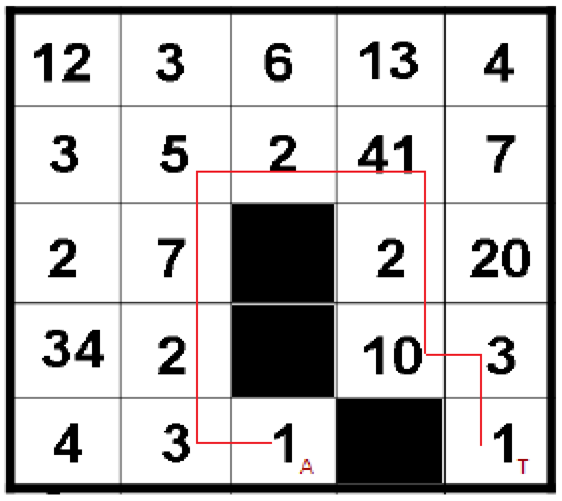
Question-5 [20 Points]:

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).



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.

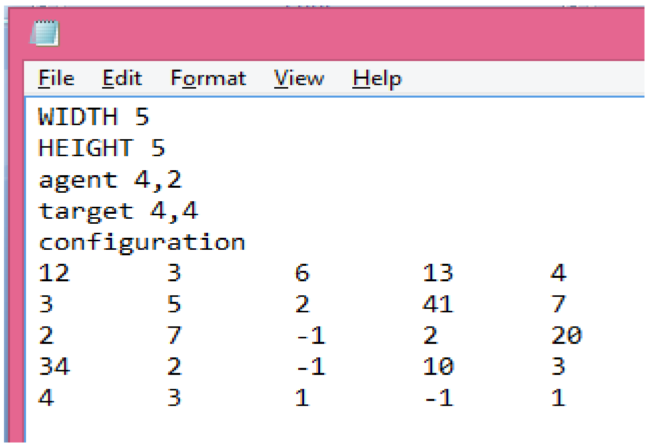
Files:

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.

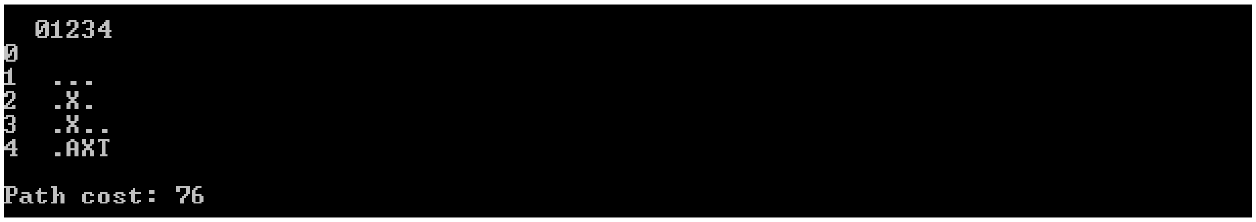


```
WIDTH 5
HEIGHT 5
agent 4,2
target 4,4
configuration
12      3       6       13      4
3       5       2       41      7
2       7       -1      2       20
34      2       -1      10      3
4       3       1       -1      1
```

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:



```
  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 128:
=> node = frontier.pop(0)
Remove this statement and add the following statements instead at the same place:
=> index = frontierCost.index(min(frontierCost))
=> node = frontier[index]
Reason: For UCS, we remove the node with lowest path cost.

2) Line 135: (Do this for all 4 actions)

```
=> if child==target:
=>          # Target found
=>          found = 1
=>          parent[childHash] = node
=>          break
```

Remove these statements and add the following statements instead after line 128 i.e. after removing the node from the queue

```
=> if node==target:
=>          found = 1
=>          break
```

Reason: Check for goal node while expanding the node and not while adding the child node in the frontier.

3) Line 141: (Do this for all 4 actions)
=> frontierCost.append(config[child[0]][child[1]])
Replace this statement with the following statement
=> frontierCost.append(frontierCost[index]+config[child[0]][child[1]])
Reason: We add path cost in the frontierCost queue rather than the cost to reach that particular node from its parent node as the priority queue (frontierCost) is based on path cost and not node cost.

4) Add the following statements after line 142: (Do this for all 4 actions)

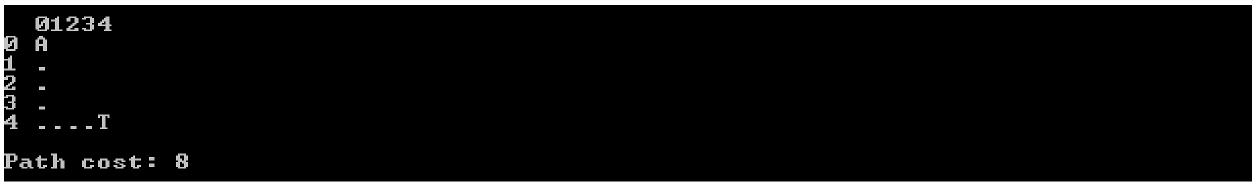
```
=> elif child in frontier:
=>          newCost = frontierCost[index]+config[child[0]][child[1]]
=>          if newCost<frontierCost[frontier.index(child)]:
=>                    frontierCost[frontier.index(child)] = newCost
=>                    parent[childHash] = node
```

Reason: If the child node is already in the frontier, we need to check whether the cost of the new path is less than the cost of the old one. And if it is, we need to update the path cost and the parent node for the child node for computing path later.

5) Add the following statements before line 182: (Not for the action but at the same indentation level as line 182)
=> frontier.remove(node)
=> frontierCost.remove(min(frontierCost))
Reason: Remove the explored node from the frontier and the frontierCost queues.

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.

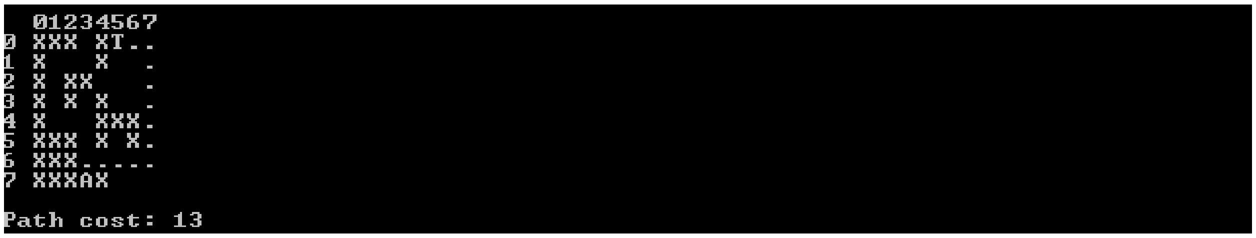
Inputfile0.txt:



```
  01234
0 A
1 .
2 .
3 .
4 ....T

Path cost: 8
```

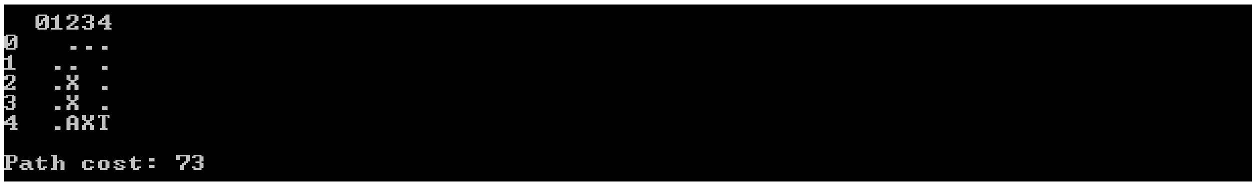
Inputfile1.txt:



```
  01234567
0 XXX XT..
1 X   X  .
2 X XX   .
3 X X X  .
4 X   XXX.
5 XXX X X.
6 XXX.....
7 XXXAX

Path cost: 13
```

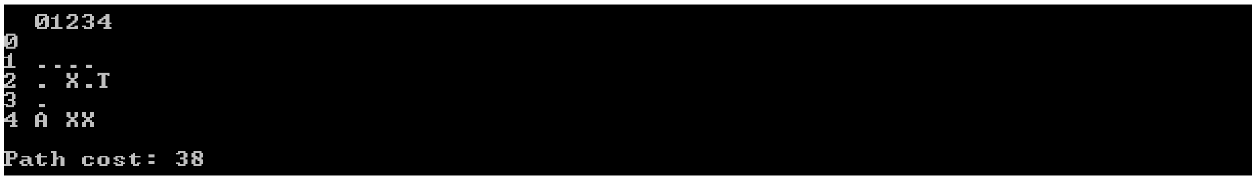
Inputfile2.txt:



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

Path cost: 73
```

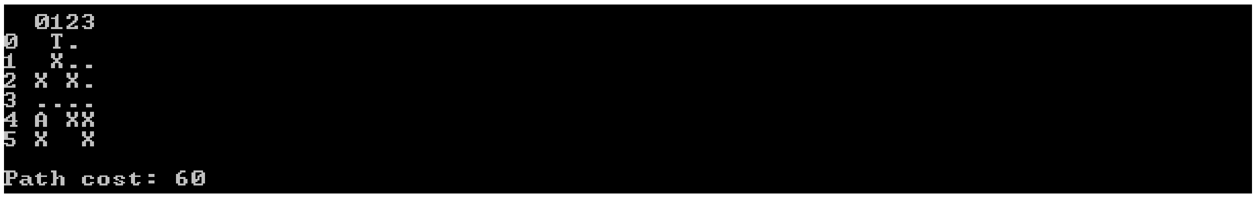
Inputfile3.txt:



```
  01234
0
1 ....
2 . X.T
3 .
4 A XX

Path cost: 38
```

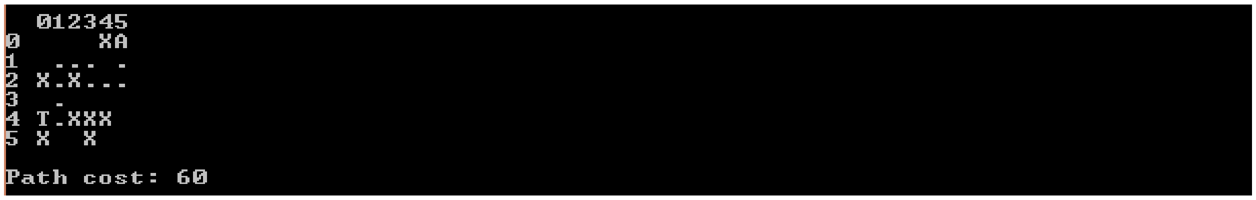
Inputfile4.txt:



```
  0123
0  T.
1  X..
2 X X.
3 ....
4 A XX
5 X  X

Path cost: 60
```

Inputfile5.txt:



```
  012345
0     XA
1  ... .
2 X.X...
3  .
4 T.XXX
5 X  X

Path cost: 60
```

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.

The Manhattan distance is given by the sum of the horizontal and vertical distance of a cell to the goal cell. Now, as the Agent can move only horizontally or vertically, this is the minimum distance the Agent has to go to reach the goal cell even when it doesn't have any data of the surrounding environment. And when it discovers the environment, the exact path cost can't be less than the Manhattan distance. Thus, Manhattan distance for each cell along a path to goal cell is non-decreasing and hence Manhattan distance heuristic is consistent.

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

Euclidian distance is an admissible heuristic function for the given problem. It is defined as the straight line distance between the agent cell and the goal cell. Similar to the argument given for the Manhattan distance, the exact path cost can't be less than the Euclidian distance and hence it is admissible.