

CS 540-2: Introduction to Artificial Intelligence**Homework Assignment #1****Assigned: Thursday, February 1, 2018****Due: Sunday, February 11, 2018****Hand-in Instructions:**

This homework assignment includes two written problems and a programming problem in Java. Hand in all parts electronically to your Canvas assignments page. For *each* written question, submit a single **pdf** file containing your solution. Handwritten submissions *must* be scanned. **No photos or other file types allowed.** Each file should have your name at the top of the first page. For the programming question, submit a **zip** file containing all the Java code necessary to run your program, whether you modified provided code or not.

You should submit the following three files (with exactly these names) for this homework:

problem1.pdf	problem2.pdf	problem3.zip
--------------	--------------	--------------

Late Policy:

All assignments are due **at 11:59 p.m.** on the due date. One (1) day late, defined as a 24-hour period from the deadline (weekday or weekend), will result in 10% of the total points for the assignment deducted. So, for example, if a 100-point assignment is due on a Wednesday and it is handed in between any time on Thursday, 10 points will be deducted. Two (2) days late, 25% off; three (3) days late, 50% off. No homework can be turned in more than three (3) days late. Written questions and program submission have the same deadline. A total of three (3) free late days may be used throughout the semester without penalty. Assignment grading questions must be discussed with a TA within one week after the assignment is returned.

Collaboration Policy:




You are to complete this assignment individually. However, you are encouraged to discuss the general algorithms and ideas with classmates, TAs, peer mentors and instructor in order to help you answer the questions. You are also welcome to give each other examples that are not on the assignment in order to demonstrate how to solve problems. But we require you to:

- not explicitly tell each other the answers
- not to copy answers or code fragments from anyone or anywhere
- not to allow your answers to be copied
- not to get any code from the Web

In those cases where you work with one or more other people on the general discussion of the assignment and surrounding topics, we suggest that you specifically record on the assignment the names of the people you were in discussion with.

Problem 1: [20] Search Algorithms

In the figure below is a modified chessboard on a 5×5 grid. The task is to capture the black king (*fixed* in square **H**) by moving the white knight (starting in square **W**) using only “knight moves” as defined in Chess. Assume the successor function *expands* legal moves in clockwise order: (1 right, 2 up); (2 right, 1 up); (2 right, 1 down); (1 right, 2 down); (1 left, 2 down); (2 left, 1 down); (2 left, 1 up); (1 left, 2 up). Note that not all these moves may be legal from a given square. It is *not* legal for the knight to move to a square within the attack range of the black rook in square **A** (i.e., the knight cannot be in the same row or column with the rook). But it is legal for the knight to move to the attack range of the black king (i.e., any of the eight squares around the king).

A 	B	C	D	E
F	G	H 	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	W 	X	Y

(a) [5] Depth-First Search

Perform depth-first search. List the squares in the order they are expanded, including the goal node if it is found. State **W** is expanded first. Using a stack for the *Frontier* means you should add states to the stack in the *reverse order* from that given above. So, state **N** will be popped and expanded second. Use the Graph-Search algorithm (Figure 3.7), which uses both a *Frontier* set and an *Explored* set so that each square will be expanded at most once. Write down the list of states you expanded in the order they are expanded, including the goal state **H**. Write down the solution path found (if any), or explain why no solution is found.

(b) [5] Iterative-Deepening Search

Perform iterative-deepening search until a solution is reached. Draw the sequence of trees built as you ran the algorithm (i.e. there should be a tree drawn at the end of each iterative-deepening iteration of the algorithm). Use the same order of expanding nodes as in (a), and also use the same method of repeated state checking as in (a).

(c) [4] Heuristic Function

Let each “move” of the knight have cost 1. Consider the heuristic function $h(n) = |u - p| + |v - q|$, where the grid square associated with node n is at coordinates (u, v) on the board, and the goal node **H** is at coordinates (p, q) . That is, $h(n)$ is the “Manhattan” distance between n and **H**. Is $h(n)$ admissible?

(d) [6] A* Search

Regardless of your answer to (c), perform A* Search using the heuristic function $h(n)$ defined in (c). In the case of ties, expand states in alphabetical order. Use repeated state checking by keeping track of both the *Frontier* and *Explored* sets. If a newly generated node, n , does not have the same state as any node already in *Frontier* or *Explored*, then add n to *Frontier*. If n 's state does already exist in either *Frontier* or *Explored*, then check the g values of both n and the existing node, m . If $g(n) \geq g(m)$, then just throw away the new node, n . If $g(n) < g(m)$, then remove m from either *Frontier* or *Explored*, and insert n into *Frontier*. List each cell in the order they are added to *Frontier* and mark it with $f(n) = g(n) + h(n)$ (show f , g and h separately). When expanded, label a state with a number indicating when it was expanded (state **W** should be marked "1", as shown below). Highlight the solution path found (if any) or explain why no solution is found.

<i>Frontier</i>	W												
$g(n)$	0												
$h(n)$	3												
$f(n)$	3												
Expanded	1												

Problem 2: [20] Stimulus Plan

The government wants to implement a stimulus program and has allocated N million dollars for this. They placed the money into an account called Stimulus. Due to budgetary and legal constraints, it must spend this money in a rather specific way:

- The money must be spent in chunks of 1 million dollars.
- In the first and last year, it must spend exactly 1 million dollars.
- In any given year, the amount of money spent must be within 1 million dollars of the amount spent in the previous year.
- The spending for any given year may be non-positive (i.e., it can be 0, or even negative, for example, by levying a tax to increase the amount of money).
- The amount of money in the account (starting at N million dollars) can be non-positive in a given year (i.e., run a deficit).

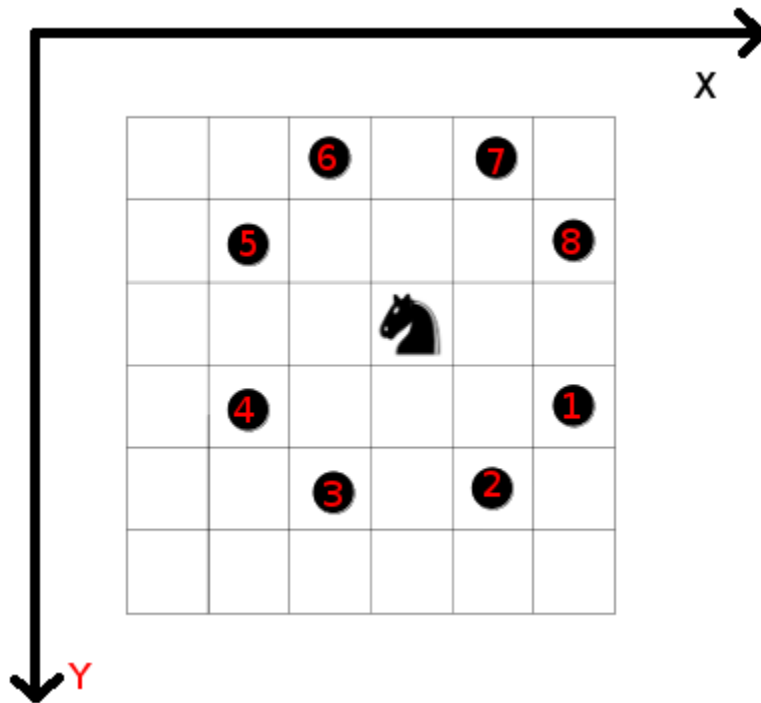
The goal is to **minimize the number of years to make the balance in the account equal 0**.

- (a) [4] Let's define each state as a tuple of three numbers, (1) the amount n in the account at the beginning of the current year, (2) the amount s that will be spent in the current year, and (3) the amount s' spent in the previous year: (n, s, s') . Define the initial state, the goal state, the step cost function, and the successor function. An example formulation is shown for the Water Jugs problem in the lecture slides.
- (b) [5] Is “the total amount of money in the account at the beginning of the current year” an admissible heuristic? Is it consistent? If not, provide a counterexample.
- (c) [5] Assume the amount of money spent in the current year is s million dollars. Is the heuristic function $h = \lceil s \rceil - 1$ admissible? Is it consistent? If not, provide a counterexample.
- (d) [6] Assume the amount of money remaining at the beginning of the current year is n million dollars. One possible admissible heuristic function can be defined as the smallest value of k such that $T_k \geq |n|$. T_k is the k -th triangular number, which can be calculated as $\sum_{i=1}^k i$. For example, when $n = 5$, our heuristic value is 3, since $T_2 < 5 < T_3$. Using A* search with this heuristic function, find an optimal solution when $N = 12$. List every state on the solution path and calculate its heuristic value.

Problem 3: [60] King's Nightmare

This problem is a modified chess problem. We have a modified chessboard that is size $n \times m$. On the chessboard is a white king, a black knight, and several obstacles. Once the game starts, the knight can make a possibly infinite number of moves, and the king does *not* move. Your goal is to write a program in Java that finds a path for the knight to capture the king, while avoiding all obstacles.

A state will be represented by the current coordinates of the knight, (x, y) . The possible moves of a knight are shown in the figure below by the 8 positions containing dark circles. Use the upper-left corner of the map as the origin, with coordinates $(0, 0)$. When expanding a node to create its children nodes, each child's position is computed from the current state's position by: $(+2,+1)$, $(+1,+2)$, $(-1,+2)$, $(-2,+1)$, $(-2,-1)$, $(-1,-2)$, $(+1,-2)$, $(+2,-1)$; these positions are numbered, respectively, 1-8 in figure below. You *must* use this order for the children! Note that this is a *different* order than used in Problem 1. Some of these moves may *not* be legal, i.e., when there is an obstacle at the destination position, or when the destination position is off the map. Note: it does *not* matter if the positions in-between the current position and destination position in a single move are obstacles.



Write a Java program that finds a sequence of moves for the knight to capture the king using Depth-First Search (DFS), Breadth-First Search (BFS) and A* Search (AStar). The programming assumptions for each search method are as follows:

- (i) DFS: Use a stack for implementing *Frontier*, pushing child nodes on to the stack in the order 1-8 shown above, and popping them from *Frontier* in the reverse order of 8-1. You can use java's built-in `Stack` class for your implementation. Perform the goal test for

a node when it is created (i.e., *before* putting it on *Frontier*), not when it is removed from *Frontier*.

- (ii) BFS: Insert expanded nodes in *Frontier* using a queue in the order 1-8 shown above, and hence removing them from *Frontier* in the same order 1-8. You can use the built-in Java interface called `Queue` for your implementation of *Frontier*. Perform the goal test for a node when it is created (i.e., *before* putting it on *Frontier*), not when it is removed from *Frontier*.
- (iii) AStar: Follow the algorithm given in Figure 3.14 in your textbook. Assume each move of the knight costs 3. Thus, all the moves 1-8 in the figure above will have equal cost of 3. The heuristic function you will use is the Manhattan distance between a node's current position (x, y) and the goal position (x_goal, y_goal) . That is, $|x - x_goal| + |y - y_goal|$. If two child nodes have the same priority score, the tie must be broken based on the neighbor number shown above. For example, node 1 must be added to *Frontier* *before* node 8 in the priority queue if nodes 1 and 8 have same priority score. You *must* use the provided implementation of a priority queue given in the skeleton code, called `PriorityQ`. This implementation sorts and maintains the queue in ascending order of score. In case of ties, this priority queue places the node that is inserted *first* ahead of another node (sibling or non-sibling) with the same score. So, it is important to insert children nodes in the order 1-8, to enable the priority queue to take care of tie-breaking cases automatically. Perform the goal test for a node when it is *removed* from *Frontier*.

For efficiency, you need to maintain a set of explored positions to avoid repeated states in all three search methods. The *Explored* set can be implemented simply as a Boolean array indicating if each board position has been previously expanded or not. Each board position needs to be expanded at most once because, for this problem, when a node is expanded, it is guaranteed we have found the shortest or lowest cost path to that node's position.

Each test case will be in a separate text file. The first line contains the search method to use, either "bfs", "dfs" or "astar" (all lower case). The second line will be two integers, n (corresponding to the y coordinate, specifying the number of rows on the board) and m (corresponding to the x coordinate, specifying the number of columns on the board), indicating the size of the chessboard where $5 < n, m < 201$. Next will be n lines, each line containing m characters, each being either "S", "G", "0", or "1". "S" is the initial knight position. "G" is the king's position (i.e., the goal position). "0" is an empty space. "1" is an obstacle.

Sample Input:

```
bfs
6 6
000000
001000
000000
00G000
0000S0
001000
```

The output should be (1) the *path* you found using the search, and (2) the number of nodes you expanded during the search. This count should *not* include the goal node for any of the search methods. In each line, print the x and y coordinates, separated by one space, of the positions of the knight after each move in the solution path. Include the initial state and the goal state. After printing the path, in a new line, print out “Expanded Nodes: ” followed by the number of the nodes expanded by algorithm. If the goal state is *not* reachable, print one line “NOT REACHABLE” and then a second line with “Expanded Nodes: ” followed by the number of the nodes expanded by algorithm. Output to the console (i.e., to standard output).

Sample output:

```
4 4
2 3
Expanded Nodes: 1
```

Hand in all .java files in a zip file called `problem3.zip` with the main class named exactly `KingsKnightmare`. This name is CASE-SENSITIVE. Test your homework on a CS Linux machine as follows:

```
> unzip problem3.zip
> javac *.java
> java -cp ./KingsKnightmare testfile.txt
```

NOTES:

- We have provided skeleton code for the assignment, which you may choose to use.
- You *must* use our provided implementation of a priority queue, called `PriorityQ`.
- We have provided 2 sample test cases for each search method for debugging.
- Do *not* use any package statements in your Java source code.
- We will allow a maximum runtime limit of 3 minutes for each test case.
- The provided sample test cases will be worth 20% of the total score (12 points).
- The hidden test cases will be worth 80% of the total score (48 points).