CS 4365 Artificial Intelligence
Spring 2018
Assignment 1: Basic Search

Part I: *Due electronically by Friday, January 26, 11:59 p.m.*

Part II: *Due electronically by Tuesday, February 6, 11:59 p.m.*

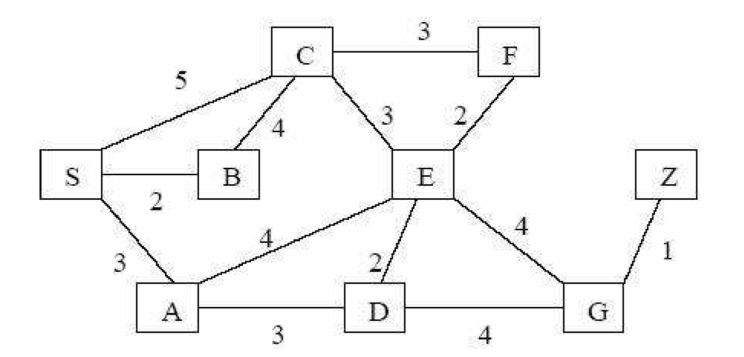
Note:

1. Your solution to this assignment must be submitted via eLearning. For the written problem, submit your solution as a single PDF file; do **not** write your solution in the Comments section of the submission page. For the programming part, upload a zip file containing your README and all your source files; do **not** upload multiple files separately to the submission directory.
2. You may work in a group of up to two people. If you work in a group, make sure the names of all group members appear in the Comments section of the submission directory and at the beginning of each file you submit. Each group should submit only *one* copy of the solution.

Part I: Written Problems (50 points)

1. **Heuristic Search (15 points)**

   Consider the graph below.



The start state is S and the goal state is Z. The numbers on the arcs indicate the cost of traversing that arc. In addition, when applying a search algorithm to this graph, note that:

- each node should be expanded at most once;
- if needed, break ties alphabetically;
- whenever the search algorithm requires a heuristic estimating function, use the function h defined below the graph.

n	S	A	B	C	D	E	F	G	Z
h(n)	8	7	6	5	4	3	2	1	0

(a) **(5 pts)** Using the graph above and **uniform-cost search**: (1) list the nodes in the order they would be expanded; (2) list the nodes that lie along the final path to the goal.

(b) **(5 pts)** Same as (a), but using **greedy best-first search**.

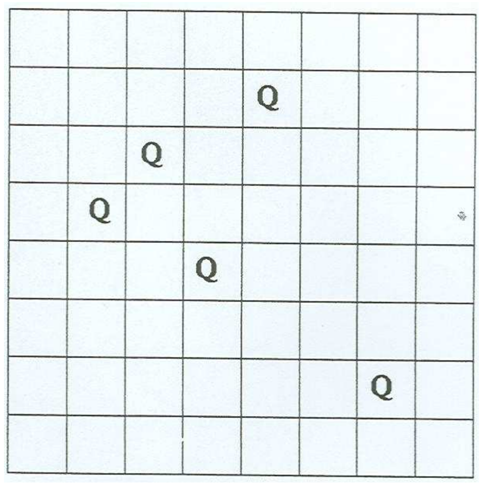
(c) **(5 pts)** Same as (a), but using **A* search**.

2. **Heuristic Search (10 points)**

We saw in class that the straight-line distance heuristic is misleading on the problem of going from Iasi to Fagaras. However, the heuristic is perfect on the opposite problem: going from Fagaras to Iasi. Are there problems for which the heuristic is misleading in both directions? If so, explain your solution via a specific example. If not, explain why not.

3. **State-Space Search (25 points)**

Consider the following puzzle: We say that a collection of queens on a chessboard *covers* the board if every square on the board is either a queen or is under attack by a queen. (A queen can move on rows, columns, or diagonals.) The problem is to find the minimum number of queens that can cover a $k \times k$ chessboard. For example, the diagram below shows that an 8×8 board can be covered with 5 queens. (Note: In parts c, d, and e below, a crude upper bound is sufficient. I am not looking for mathematically sophisticated solutions.)



(a) **(5 pts)** Show how this problem can be solved using a state-space search. What are the states? What are the operators?

(b) **(5 pts)** Show how the state space can be structured to be tree-structured (systematic); that is, for any state S, there is a unique path of operators that generates S.

(c) **(5 pts)** Let M_k be the minimum number of queens needed to cover a $k \times k$ board. Give a simple upper bound on M_k in terms of k.

(d) **(5 pts)** Give a simple upper bound on the branching factor in the state space.

(e) **(5 pts)** Give simple upper bounds on the time and space requirements for solving the problem, using (a) depth-first search; (b) breadth-first search (keep in mind that M_k is not known till after the search has been completed.) You may estimate time and space in terms of the number of nodes that must be generated and the number of nodes that must be kept simultaneously in memory; that is, you do not have to consider how much space is required to store a single node or how much time is required to generate and evaluate a single node.

Part II: Programming (100 points)

You will be implementing a simple search engine that supports multiple search strategies to solve 1-dimensional tile ordering problems. Given a row containing an equal number of black tiles, white tiles, and a single empty space in an arbitrary initial ordering, your goal is to rearrange the tiles such that all black tiles are to the left of the empty position, and all white tiles are to the right of the empty position. To achieve this goal, you are only allowed to move a tile from its current position to the empty position; you cannot swap two tiles directly.

To illustrate the tile domain, consider the trivial 3-position case consisting of a white tile, a black tile, and an empty space in the initial configuration: `white`, `space`, `black` (Figure 1). One possible solution to this problem is given by the sequence `move 2`, `move 0`, `move 1`, where the numeric argument indicates the position of the tile to be moved. (The destination of the move is evident — the empty position — so it need not be specified in the operator.)

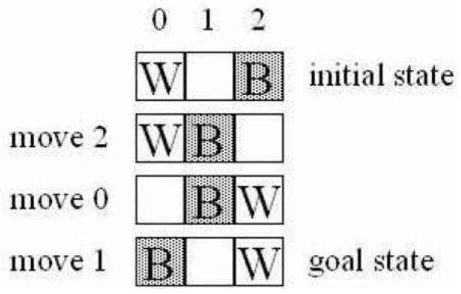


Figure 1: 3-position tile problem example

Part A: The basic program

Your implementation should read in an arbitrary starting state for tile problems containing up to 13 positions (6 black tiles, 6 white, and 1 space). Each input file contains a single line specifying the initial state using `B` to represent a black tile, `W` to represent white, and `x` to represent the space. For example, one possible starting state for a 7-position problem is:

```
BWBBWxW
```

You should use one operator, `move` x, to specify which tile to move into the current empty position. In the absence of any other heuristic function which determines which tile to move, all positions should be considered in increasing order (i.e., from 0 to n; `move 1` occurs before `move 3`).

Your program should print out the initial state, each move operation selected and its successor state (i.e., both the operator path and the state path). In addition, each line starts with a step number, which allows us to keep track of how many steps we need to take to reach the goal state. For example, given the problem of Figure 1, your program should print the following lines:

```
Step 0:  WxB
Step 1:  move 2 WBx
Step 2:  move 0 xBW
Step 3:  move 1 BxW
```

Your program must support five different search strategies, selectable at run-time using a command-line keyword: breadth-first search (`BFS`), depth-first search (`DFS`), uniform-cost search (`UCS`), greedy search (`GS`) and A* (`A-star`). Note that you should be implementing this using the generic search algorithm we discussed in class (i.e., use a next-state queue containing states which have been generated but not-yet visited as your primary data structure. BFS implies a FIFO discipline, DFS implies a LIFO or "stack" discipline, UCS implies sorting the queue according to the g function, GS implies sorting the queue according to the h function, and A* implies sorting the queue according to the evaluation function f, where $f = g + h$.) If two or more nodes in your data structure have the same f cost, break ties using a FIFO strategy (i.e., prefer nodes that have been in the data structure for the longest time). Also, note that you will need a means of avoiding loops by checking whether or not a state has already been visited. You should not expand a state that has previously been expanded.

For UCS, assume that $g(n)$ = number of moves executed so far

For GS, assume that $h(n)$ = number of tiles out of place (e.g., in the 3-position example we started with 2 out-of-place tiles)

For A*, estimate the total path cost as $f(n) = g(n) + h(n)$ where g and h are defined as above.

Part B: A more interesting cost function

Since we have considered all moves equally as having unit cost so far, UCS and BFS always return the same solution. To make things more interesting, augment your program to accept a command-line flag (`-cost`) which weighs the cost of each move by the number of positions the tile moves. For example, the new costs of performing the operations in Figure 1 are 1,2,1 (respectively), resulting in a total cost of 4.

In addition to the operator and state path, this version of your program should print out these operation costs:

```
Step 0:  WxB
Step 1:  move 2 WBx (c=1)
Step 2:  move 0 xBW (c=2)
Step 3:  move 1 BxW (c=1)
```

In essence, the `-cost` flag alters the g function that your program uses. This will in turn change the solution returned by UCS and A*.

Requirements

Each group should hand in a single copy of your tile ordering program, implemented in C/C++, Java, or Python (2.7, 3.3, or 3.4). If you want to use something else, please ask your dear TA, Stuart Taylor (`Stuart.Taylor1@utdallas.edu`), but it needs to be run/compile on the Linux boxes without installing extra software. You must turn in a zip file containing documented source code as well as a README file describing

- the names of all the group members;
- a list of all your source files;
- the platform on which your code is developed (e.g., Windows/Linux/Solaris);
- documentation of any problems you had; and
- how to run your program.

Your program should accept command-line arguments to specify the search type, whether to use constant or variable costs for moves, and the input file:

```
search [-cost] <BFS|DFS|UCS|GS|A-star> <inputfile>
```

There should be **no** graphical user interface (GUI) of any kind. Any program that does not conform to the above specification will receive no credit.

If the code is in your partner's submission directory, you should clearly indicate in the "Comments" section of your own submission directory who you worked with.

Grading Criteria

Grades will be based primarily on:

- Correctness of your search implementation
- Conformance to the execution and output formats specified above, including the general search function criteria
- Whether or not your program finds a valid (and when required, optimal) path