15-381 Problem Set 1: Search Algorithms

Out: Thursday, August 31, 2000
Due: Thursday, September 14, 2000

This is the first problem set for 15-381, and contains both written exercises and a programming assignment. Every person registered for the course (as of 8/29) has been randomly assigned a homework partner. (This is only for the duration of this homework; new partners will be assigned for each problem set.) To find out who your partner is, check the class web page. Note that you are responsible for contacting your partner: we strongly recommend that you do so as soon as possible. If you need to be assigned to a group (or reassigned because your partner decided not to take the class) please send email to Rune (runej@cs).

If you obtain concrete help from anyone other than your partner (e.g. TA's, fellow students, friends, etc.) you *must* document the nature of the help you receive, which we'll take into account in the grading. It is much better to seek help and turn in an assignment complete and on time than to not seek help and fail to complete it, either on time or at all. In general, we will not deduct much (if anything) for seeking help, as long as that help is indirect (e.g. help with debugging your program, rather than someone telling you exactly how to solve one of the problems).

Written Exercises

All exercises refer to problems in *Artificial Intelligence: A Modern Approach.*

1. **Exercise 3.9, pg. 89** - Be sure to illustrate this with a concrete example (i.e. define the initial and goal states, operators, cost function, and describe the search path of each algorithm).
2. **Exercise 3.12, pg. 89**
3. **Exercise 4.4, pg. 118**
4. **Exercise 4.6, pg. 118**

Universal Forward-Search Engine Implementation

You will be implementing a forward-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` (Fig. 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.)

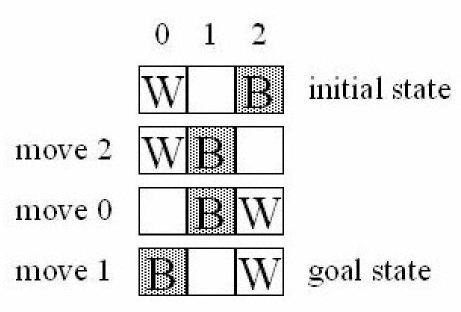

Figure 1: 3-position tile problem example

Part I: The basic program

Your implementation should read in an arbitrary starting state for tile problems containing up to 21 positions (10 black tiles, 10 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). For example, given the problem of Figure 1, your program should print the following lines:

```
WxB
move 2 WBx
move 0 xBW
move 1 BxW
```

Your program must support three different search strategies, selectable at run-time using a command-line keyword: breadth-first search (`BFS`), depth-first search (`DFS`), and A* (`A-star`). Note you should be implementing this using a single function, not three different functions disguised as one. (Hint: 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, and A* implies sorting the queue according to the heuristic function.) Also note that you will need a means of avoiding loops by checking whether or not a state has already been visited.

In the A* case, estimate the total path cost as $f(n) = g(n) + h(n)$ where:
$g(n)$ = number of moves executed so far
$h(n)$ = number of tiles out of place (e.g. in the 3-position example we started with 2 out-of-place tiles)

Part II: A more interesting cost function

Instead of considering all moves equally as having unit cost, augment your program to accept a command-line flag (`-cost`) which weights 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 5.

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

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

Note, you may also want to consider modifying the heuristic function for the A* search to reflect the varying weights. Be sure that any new $h(n)$ you use is admissable.

Extra Credit

The 1-dimensional tile problem is also suitable for bi-directional breadth-first search. For 10 points extra credit, implement a bi-directional search option (add a fourth keyword `BiBFS` for the search-type argument). Make sure you perform interwoven foward and backward fringe expansion and check for intersections.

Requirements

You and your partner should hand in a single set of solutions to the written exercises and a single copy of your universal search program, implemented in C or C++. You must turn in documented source code, as well as an executable compiled under Linux. Your executable 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|A-star> <inputfile>
```

Testing Your Program

Sample files on which you can test your program have been provided in:
`/afs/cs/academic/class/15381-f00/public/hw1-testfiles/`

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

Submission Instructions

After completing the assignment, place one copy of your written answers, source code, and executable in:
`/afs/andrew/scs/cs/15-381/handin/`*userid1_userid2*

Submission directories for each group should be available by next Tuesday (9/5), and will be closed at 11:59pm on September 14th.

Please contact either Laurie (hyaku@cs) or Rune (runej@cs) if you have any questions or encounter difficulties.