15-381 Spring 06 Assignment 1 Solutions

March 2, 2006

1 Formulating the Search Problem (15 points)

References (names of people I talked with regarding this problem or “none”):

The four-peg version of the Tower of Hanoi puzzle consists of four pegs mounted on a board and \( n \) disks of various sizes with holes in their centers (see Figure 1). If a disk is on a peg, only a disk of smaller diameter \( d \) can be placed on top of it. Given all the disks properly stacked on the leftmost peg as in Figure 1, the problem is to transfer the disks to the rightmost peg by moving one disk at a time. Formulate this problem as a search problem.

![Figure 1: Four-peg version of the Tower of Hanoi.](image)

1.1 (3 points)

Define a state representation.

*Multiple solutions are possible. One example:*

Each disk is represented by \( D_i \), where \( D_i \) is smaller than \( D_j \) for all \( i < j \). The state can be represented by four stacks, \( S_1 - S_4 \), representing the pegs numbered left to right. Each stack contains the disks that are on that peg, such that the top-most (front) element of the stack represents the top disk.

1.2 (3 points)

Give the initial and goal states in this representation.

*Initial State: \( S_1 : \{D_1, D_2, \ldots D_n\} \) \( S_2 : \{\} \) \( S_3 : \{\} \) \( S_4 : \{\} \)*

*Final State: \( S_1 : \{\} \) \( S_2 : \{\} \) \( S_3 : \{\} \) \( S_4 : \{D_1, D_2, \ldots D_n\} \)
1.3 (3 points)

Define the successor function, i.e. show how the successors of a state can be computed using your representation.

Disk \( D_i \) can be moved from \( S_i \) to \( S_j \) if:

- \( D_i \) is the topmost disk on \( S_i \)
- \( S_j \) is either empty, or the size of the topmost disk of \( S_j \), \( D_j \), is greater than \( i \)

1.4 (3 points)

What is the cost function for the above successor function?

Uniform cost

1.5 (3 points)

What is the total number of legal states?

\( 4^n \)

2 Uninformed Search (17 points)

References (names of people I talked with regarding this problem or "none"):

2.1 (9 points)

For each of the following statements, explain in terms of the cost function the circumstances under which it is true.

- Breadth First search is a special case of Uniform Cost search.
  
  **When all transitions have equal cost.**

- Breadth First search is a special case of Best First search.
  
  **When the evaluation function value is proportional to the depth of the state in the search tree (i.e. \( f(s) = \text{depth}(s) \)).**

- Uniform Cost search is a special case of A* search.
  
  **When \( h(s) = 0 \) for all states \( s \).**

2.2 (5 points)

Give a description of a search space in which Iterative Deepening performs much worse than Depth First search. Give the respective complexities for this domain in Big O notation.

**Multiple solutions are possible. One example:**

A tree with branching factor always equal to 1. For \( n \) nodes, DFS would visit \( O(n) \) states, while IDS would visit \( O\left(\frac{n(n+1)}{2}\right) = O(n^2) \).
2.3 (3 points)

Describe the criteria that determine the best direction (forward or backward) for search in a problem space. Assume a single start state and a single goal state.

*The direction with the smaller branching factor is preferred.*

3 Informed Search (18 points)

References (names of people I talked with regarding this problem or "none"):

3.1 (3 points)

Suppose that in addition to the admissible heuristic \( h(n) \), you are told that there is a solution whose true cost is \( K \). How would you change \( A^* \) to take advantage of this knowledge and reduce the number of nodes that need to be expanded?

Does this method maintain optimality?

*Since we know the true cost of the solution, we do not need to expand nodes with cost greater than \( K \). This method maintains optimality.*

3.2 (3 points)

Under what conditions does Simulated Annealing perform better than Hill Climbing? Would you ever prefer Hill Climbing? If so, when?

*SA performs better than HC in domains with many local optima. Hill climbing is easier to implement and is preferred for domains with no local optima.*
3.3 (12 points)
In an environment that is an $n \times n$ grid, we have $n$ cars that are located in squares \((1,1)\) through \((1,n)\) (i.e. in the bottom row). All of the cars have to be moved to the top row of the grid, but their order must be reversed. The car that started at \((1,1)\) must be moved to \((n,n-i+1)\), and so on. During each timestep, every one of the cars executes a legal move at the same time. There are five legal moves: North, East, South, West or Stay. Assume that multiple cars can be located in the same grid square.

(a) (3 points) The size of the state space is

(i) $O(n^2)$  
(ii) $O(n^3)$  
(iii) $O(n^{2n})$  
(iv) $O(n^{n^2})$

(b) (3 points) The branching factor is roughly

(i) 5  
(ii) $5n$  
(iii) $5^n$

(c) (3 points) Suppose that car $i$ is currently located at \((x_i,y_i)\). Write a nontrivial admissible heuristic $h_i$ for the number of moves it will take for that car to get to its goal location \((n,n-i+1)\), assuming that there are no other cars on the grid.

There are multiple possible heuristics. One simple example is the Manhattan distance between the car’s current location \((x_i, y_i)\) and the goal \((n,n-i+1)\): $h_i = |x_i - n| + |y_i - (n - i + 1)|$

(d) (3 points) Take the problem of moving all of the $n$ cars to their destination. Which of the following heuristics are admissible when considering all of the cars at the same time?

(i) $\sum_{i=1}^{n} h_i$  
(ii) $\max \{h_1, \ldots, h_n\}$  
(iii) $\min \{h_1, \ldots, h_n\}$  
(iv) None of these
4 Peg Solitaire (50 points)

References (names of people I talked with regarding this problem or “none”):

Peg Solitaire is a game played on a board consisting of a number of holes and pegs. The object of the game is to make a series of jumps such that there is only 1 peg left, this peg must end in the center hole. A jump is when a peg is moved over an adjacent one and lands on an empty cell next to it. The peg that was jumped over is then removed from the board. The jumping sequence is similar to that of checkers except that instead of diagonal jumps, the jumps are made either horizontally or vertically.

Different variations of the game exist, with varying board sizes and shapes. In this assignment we will use one standard board shape with several starting peg configurations. The figure below shows the board, an example peg configuration and the goal state. To try out the game and see more examples go to http://www.puzzle-factory.com/java/32peg2/java-solitaire2.html.

![Board configurations]

Figure 2: (a) The empty board, (b) the ”Latin cross” peg configuration, (c) final board. Squares with empty circles represent empty holes, full circles represent pegs.

Implement an algorithm that outputs a series of jumps for solving the puzzle given the initial board configuration as the input. Note that the goal is not only to have only one peg remaining, but also that it must be located in the center hole. Several example input and output files will be provided for testing.

Files describing the initial board configuration will visually represent the board using a string of characters. Each line of the file represents one row of the board. The following is a representation of example board (b) in the above figure:

```plaintext
... | ... | ...
-----|-----|-----
  o   |  o   | ...
-----|-----|-----
  o o o | o o o | ...
-----|-----|-----
  o o o o o o | o o o o o o | ...
-----|-----|-----
  o o o o o o | o o o o o o | ...
-----|-----|-----
  o o o o o o | o o o o o o | ...
-----|-----|-----
  o   |  o   | ...
-----|-----|-----
...
```
Given the initial board configuration as input, your program should output the sequence of jumps needed to solve the puzzle. Number each of the holes according to the diagram below. Each jump should be described by the pairing \([\text{initState}, \text{endState}]\), where a peg starting at \(\text{initState}\) jumps to location \(\text{endState}\) and the peg in the middle is removed. Here is the example output for the "Latin cross" puzzle:

\[
[9, 7] \\
[23, 9] \\
[10, 8] \\
[7, 9] \\
[4, 16]
\]

Note that many peg configurations have multiple solutions, including the one above.

\[
\begin{array}{cccccccc}
0 & 1 & 2 &   &   &   &   & \\
   &   &   & 3 & 4 & 5 &   &   \\
6 & 7 & 8 & 9 & 10 & 11 & 12 &   \\
13 & 14 & 15 & 16 & 17 & 18 & 19 &   \\
20 & 21 & 22 & 23 & 24 & 25 & 26 &   \\
   &   &   & 27 & 28 & 29 &   &   \\
   &   &   &   &   &   & 30 & 31 & 32 \\
\end{array}
\]

Figure 3: Hole numbering.

4.1 (2 points)
How many different possible game states are there in this puzzle? (Define game state as at least one peg is on the board)

\[2^{30} - 2 \text{ (subtract the states where the entire board is completely empty or full.)}\]

4.2 (2 points)
Define a state representation.

*There are multiple possible solutions. One example:*
A two dimensional matrix where each cell \((i, j)\) represents a location on the board. Each location is assigned one of the following values:

- not playable (-1)
- empty (0)
- full/has a peg (+1)

### 4.3 (6 points)

Give two admissible heuristics for this problem:

- Number of pegs on the board - 1
- (Manhattan distance of the furthest peg from the center) / 2

### 4.4 (10 points)

Implement Depth First Search to solve the Peg puzzle. In the table below, record the number of states visited by your algorithm and the effective branching factor for each of the puzzles. The effective branching factor is defined as follows: if during the search the algorithm expands \(N\) nodes, and the solution is found at depth \(d\), then the effective branching factor \(b^*\) is the branching factor that a uniform tree of depth \(d\) would have to have in order to contain \(N\) nodes. More specifically the variables are related by the equation: \(N = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d\).

<table>
<thead>
<tr>
<th>Puzzle Type</th>
<th># Pegs</th>
<th># States Visited</th>
<th>Branching Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small House</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fireplace</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pyramid</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Empty Diamond</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Diamond</td>
<td>24</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 4.5 (10 points)

Implement A* to solve the Peg puzzle. In the table below, record the number of states visited by your algorithm and the effective branching factor for each of the puzzles.

<table>
<thead>
<tr>
<th>Puzzle Type</th>
<th># Pegs</th>
<th># States Visited</th>
<th>Branching Factor</th>
<th>Max Queue Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small House</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fireplace</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pyramid</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Empty Diamond</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Diamond</td>
<td>24</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Describe the heuristic you are using for A*.

4.6 (3 points)
How do you expect the performance of IDS to compare to DFS in this domain?

Since the depth of all solutions in this domain is the same, IDS would find the same solution as DFS but would take significantly longer to do so. So IDS would not be a good choice for this domain.

4.7 (3 points)
Would Bi-Directional search be useful in this domain?

No, Bi-Directional search would not be useful in this domain because the branching factor of the backwards search is larger than the forward branching factor. Searching backwards, we would have to consider all possible peg configurations that could have led to the terminal state, instead of being limited by the current board setup. As a result, a large number of states that may not even be reachable from the start would be expanded.