1 FifteenPuzzle Problem

The eight puzzle problem consists of 8 tiles with numbers 1...8 marked on them, and one empty space. The tiles need to be shifted around until the goal state is reached (where the first space is empty, and the tiles are in sequential order).

The assignment was to extend the eight-piece puzzle to a fifteen-piece puzzle.

2 Case 0

Here's the easy initial state used:

1	2		4
5	7	3	8
9	6	11	12
13	10	14	15

The easy state worked for DFS, BFS and IDS, but not for DLS—it cut off at 15 states and didn't find a solution.

3 Cases 1 and 2

Case 1: At least 59 steps needed:

11	5	12	14
15	2		9
13	7	6	1
3	10	4	8

Case 2: At least 38 steps needed:

13	5	8	3
7	1	9	4
14	10	6	15
2	12	11	

None of the algorithms worked for the harder states (ones with at least 38 and 59 steps to the goal). All four eventually resulted in some sort of computer error or never finished.

Depth-Limited Search did not find a solution with a limit of 15. The goal state is way beyond the limit.

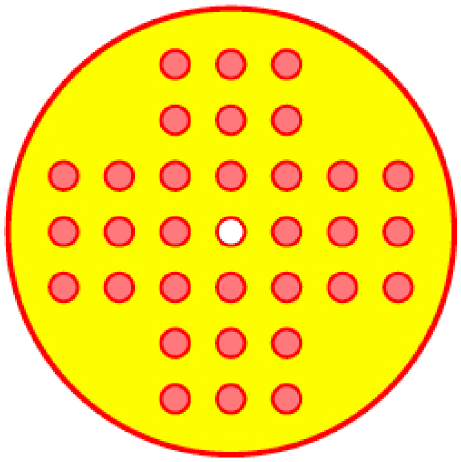
Depth-First Search didn't finish executing for the same reason that DLS didn't. Both DLS and DFS have a time complexity of $O(b^d)$ (or $O(b^l)$ in DLS's case) for b nodes and d depth.

Iterative-Deepening did not complete in a reasonable amount of time for the same reason as DFS.

Breadth-First Search ran out of memory. BFS keeps every node in memory and therefore has a space complexity of $O(b^{d+1})$.

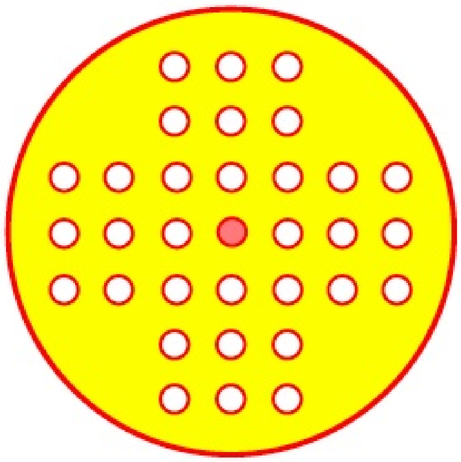
4 Peg Solitaire Problem

Peg Solitaire is a game consisting of a playing board with 33 holes together with 32 pegs. In the picture above, the hole in the center is empty and the remaining holes contain pegs. The goal is to remove all the pieces except one, which should be in the center. A piece can be removed by jumping an adjacent piece over it into an empty hole. Jumps are permitted horizontally or vertically, but not diagonally.



the start state

States change only when a peg moves. Every time a peg moves, another peg goes away. At most, there are $p-1$ moves that can be made for one peg to be left over, where p is the number of pegs. This means that for the 32-peg solitaire game, there exist exactly 31 states between the start and the goal.



the goal state

5 Breadth-First and Iterative Deepening

These are not good solutions for peg solitaire since the goal is at the leaves of the tree. BFS will search all nodes per level of the tree and will not get to the goal until the very end of its search, meeting its upper bound of$O(b^d)$ in both time and space.

IDS will not meet the goal unless $l \geq 31$ at which point it becomes a DFS.

6 Depth-First Search

This assignment has two implementations of DFS a "pure" DFS which implements tree-search and one that uses the book code using graph-search. Unsurprisingly, the tree-search one doesn't work; it eventually runs into a stack problem, since it doesn't check for cycles. The graph-search one takes a long time to find the solution. The tree-search does not use the book code.

7 A* Search

A* uses a knowledge-plus-heuristic cost function of node n (usually denoted $f(n)$) to determine the order in which the search visits nodes in the tree, defined by the following: (Wikipedia)

$f(n) = g(n) + h(n)$

In this implementation:

- G is the average distance between pegs from the distance the pegs have actually traveled so far.
- H is the average distance between pegs, relative to the center hole. The average distance includes the possibility of the pegs traveling diagonally, but G cannot since that isn't allowed during gameplay.

The idea of this heuristic is based on the facts that: 1. the more spread out the pegs are, the less likely the algorithm is approaching the goal, and 2. the more pegs there are remaining, the further it is from a solution.

It was found that using the remaining pegs as the heuristic didn't work well, since the cost to goal was uniform. The Manhattan distance (distance of pegs to the center) also didn't work well for two reasons: 1. the cost to goal was often uniform, and 2. the heuristic allowed isolated pegs to exist, so many states had pegs that couldn't move and led the search down the wrong path.

The best solution was to use the average distance of pegs from each other using the Manhattan distance as the heuristic. This covered the issue of dead-end paths (for the most part) and states fewer isolated pegs occurred.

Here's an outline of the algorithm:
Explore neighboring nodes to find the least $f(n)$.

1. Each iteration is a peg whose directions are searched (right, down, left, up) which generates a new state. If a peg can jump, do the following:

   a. The peg that was jumped goes away. Remove it and subtract its distance to the goal from $h(n)$ and calculate the new average.

   b. Move the peg and update the average estimate to $h(n)$ and calculate the new average distance taken. Update the average distance between pegs traveled to $g(n)$.

2. Push the new state (if one exists) into the priority queue, ordered by F.

3. Poll the first in the priority queue (one with the lowest F score) and search it. Each node keeps the parent state in memory.

8 Greedy Best-First Search

It was accidentally discovered, when attempting to create an A* heuristic, that using only the pegs remaining on the board as the heuristic for both G and H returned a solution faster. I concluded that this was not really A*, since G (or H) could have been set to zero, and the solution would be effectively the same. This was really a greedy best-first search, as one heuristic F (the desirability of the next node) was used. Greedy-BFS's procedure is almost identical to the A* one. This algorithm is also attached to this assignment (just for fun).

Both A* and Greedy-BFS find a solution in a very reasonable amount of time.