Pegboard Game Solver Report

Thomas Laverghetta,
Old Dominion University – Computational Modeling and Simulation Engineering

Introduction

In this report, I will be analyzing four different search algorithms when applied to a pegboard game. The pegboard game is single-player game where the objective is to move and remove pegs from a grid (play space) until a single peg is remaining. However, due to the complex nature of the game, there can be thousands or more possible moves states of which only $n-1$ (where n is number pegs) moves states produce a solution. Therefore, giving the perfect platform to analyze graph searching algorithms.

The graph searching algorithms I will be analyzing are breadth-first search, depth-first search, greedy-best search, and A*. These algorithms are used to search or traversing data graphs to find desired states/nodes. In this case, the data graph will be construct from the pegboard where each move corresponds to a new graph node, and the desired node is single peg state.

The analysis I will be conducting is given a pegboard organized into a NxN grid (N^2-1 pegs) determine how long each algorithm takes, whether it finds solution, and how much memory it consumes to find solution. The objective is to determine if informed searches (greedy-best and A* searches) outperform and uninformed searches (breadth-first and depth-first searches).

Pegboard Game Rules

Pegboard problems are single-player games played on a grid, in which moves are made by successively jumping and removing pegs from the pegboard. A peg can jump an adjacent peg if there is a slot adjacent to that peg in the opposite direction – horizontally or vertically. Diagonal jumps are not allowed. After a peg has been jumped, it is removed from the board (and possibly eaten). A typical objective of this problem is to begin with a full pegboard from which one peg has been removed and determine a sequence of jumps which will result in one peg remaining. The pegboard game has been a challenging problem for a human being. Figure 1 illustrates a solution of a 6x6 pegboard game. (Li, 2020)

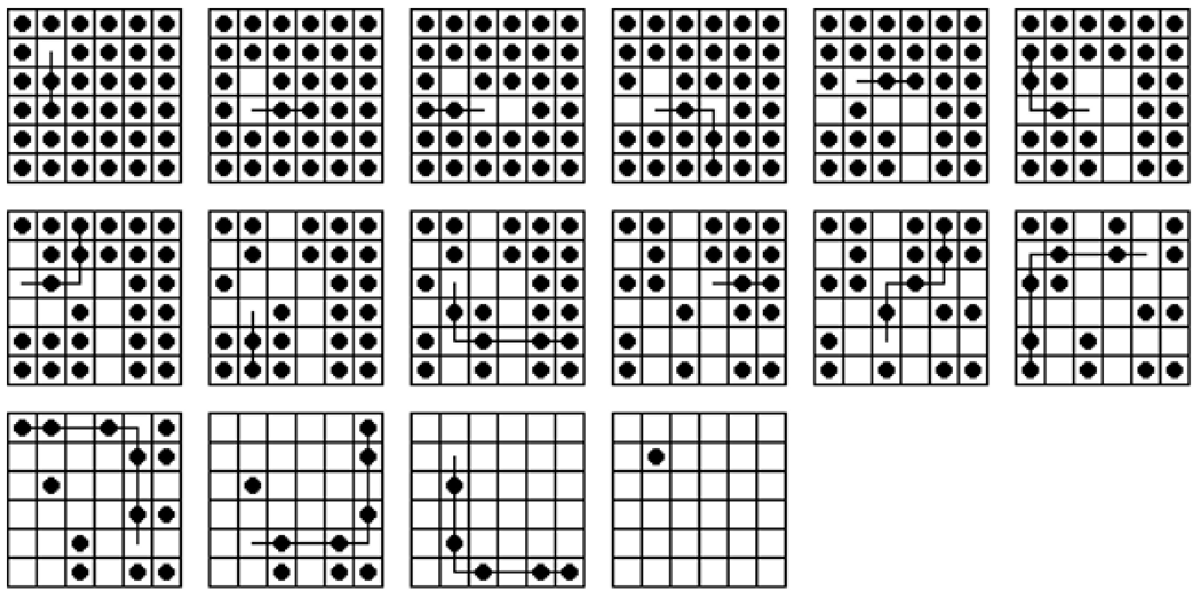

Figure 1. 6x6 Solution (Bell, 2016)

Algorithms

In this section, I will be discussing the algorithms used to search for solutions to pegboard game. The algorithms I will be discussing are breadth-first search (BFS), depth-first search (DFS), greedy-best search (GBS), and A* (A-star) search. Each algorithm is designed to search a given graph until a goal state/node has been found. For this paper, the graph will be constructed by expanding from an initial pegboard state to successor states (moves from initial pegboard state) until one peg is remaining (goal state).

Breadth First Search Algorithm

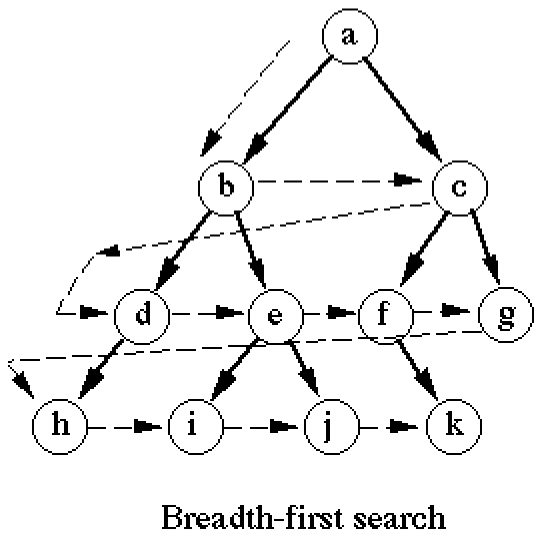

Figure 2. (Viva, 2020)

The breadth-first search algorithm (BFS) is a class of uninformed searches for traversing or searching a graph data structure. BFS starts from initial graph node (initial pegboard state) and explores all neighbor nodes at the present depth prior to moving on to the nodes at the next depth level. Figure 3 illustrates the strategy. The BFS algorithm is has follows:

```
Bread First Search (State S)
1. create FIFO Queue
2. push S into Queue
3. While Queue Is Not Empty
4.      T = Pop Queue
5.      if T is the goal
6.          return T
7.      else
8.          foreach successor of T
9.              if successor state has not been visited
10.                 mark successor state has visited
11.                 push successor into Queue
12.             end if
13.         end for
14.     end if
15. end while
16. return no solution found
```

Depth First Search

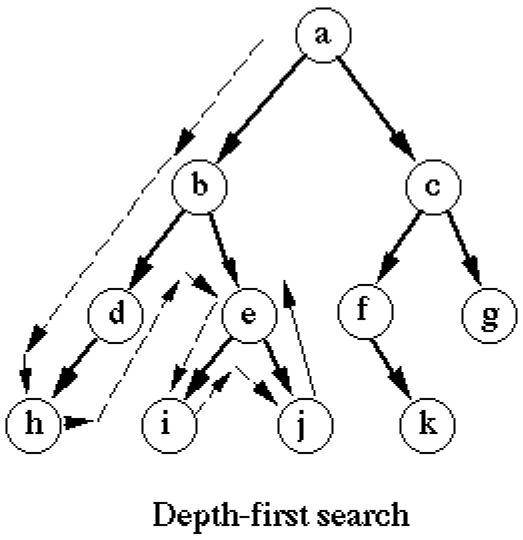

Figure 3. (Viva, 2020)

Depth-first search algorithm (DFS) is a class of uniformed searches for traversing or searching a data graph structure. DFS starts at the initial graph node (initial pegboard state) and explores as far as possible along each branch before backtracking to the next available branch search down. Figure 3 illustrates strategy. DFS algorithm is has follows using recursion:

```
Depth First Search (State S)
1. if S is the goal
2.     mark S has goal
3.     return S
4. else
5.     foreach successor of S
6.         if successor state has not been visited
7.             mark successor state has visited
8.             if Depth First Search (S) is the goal
9.                 return Depth First Search (S)
10.            end if
11.        end if
12.    end for
13. end if
```

Heuristic

The term heuristic is used for algorithms which find solutions among all possible ones, but they do not guarantee that the best will be found, therefore they may be considered as approximately and not accurate algorithms. These algorithms usually find a solution close to the best one and they find it fast and easily. Sometimes these algorithms can be accurate, that is they find the best solution, but the algorithm is still called heuristic until this best solution is proven to be the best. The method used from a heuristic algorithm is one of the known methods, such as greediness, but to be easy and fast the algorithm ignores or even suppresses some of the problem's demands. (Drossos Kikolaos, n.d.)

I will be using a heuristic function for greedy-best and A* search algorithms. There are many tactics for creating a heuristic function. For my heuristic, I want the pegs to group around the center. Therefore, I will be using is the Manhattan distance formula (Taxicab Geometry) as my heuristic function. The Manhattan distance is the distance between two points measured along axes at right angles. On a plane with p_1 at (x_1, y_1) and p_2 at (x_2, y_2) this can be formulated as $d_m = |x_1 - x_2| + |y_1 - y_2|$. For my heuristic, I will be summing distances from all pegboard pieces to the center. This can be summarized in [1] where there are n pegboard piece (s_i) whose row and column are $s_i[r], s_i[c]$, respectively, and the center of the pegboard is r_0, c_0.

$$H(S) = \sum_{i=0}^{n} |s_i[r] - r_0| + |s_i[c] - c_0| \qquad [1]$$

Greedy Best Search

Greedy-best search algorithm (GBS) is pure heuristic search. GBS starts at the initial graph node (initial pegboard state) and explores successors with the slowest heuristic value until the goal node has been reached (directly related to heuristic value). GBS algorithm is as follows:

```
Greedy Best Search (State S)
1. create FIFO Queue
2. push S into Queue
3. While Queue Is Not Empty
```

```
4.      T = Pop Queue
5.      if T is the goal
6.          return T
7.      end if
8.      foreach successor of T
9.          if successor state not visited
10.             mark successor state has visited
11.             calculate and save heuristic for successor
12.             push successor into Queue
13.         end if
14.     end for
15.     sort Queue in ascending order based on heuristic scores
16. end while
17. return no solution found
```

A* Search

A* algorithm (Astar) uses a heuristic and cost functions to find a solution. Unlike GBS, A* uses a cost function to determine if the route is the shortest route to take – i.e. the number of moves to get to state from initial state. Similarly, to GBS, A* starts at the initial graph node (initial pegboard state) and explores successors with slowest heuristic and cost values until goal node has been reached. The A* algorithm is as follows:

```
A* (State S)
1. if S is the goal
2.      return S
3. end if
4. create FIFO Queue
5. push S into Queue
6. While Queue Is Not Empty
7.      T = Pop Queue
8.      foreach successor of T
9.          if successor state has not been visited
10.             if successor is the goal
11.                 return successor
12.             end if
13.             calculate and save heuristic and cost function for successor
14.             if successor state in Queue
15.                 if successor's cost is less than equivalent state in Queue's cost
16.                     replace equivalent state with successor in Queue
17.                 end if
18.             else
19.                 push successor into Queue
20.             end if
21.         end if
```

```
22.        end for
23.        sort Queue in ascending order based on heuristic plus cost scores
24. end while
25. return no solution found
```

Although, lines 14-18 are not implemented for pegboard game since cost is the number of moves and to get the same state the cost could be the same then there is no need to check and replace. Therefore, for my implementation, I removed if statement at 14-18 and went straight to "*push successor into Queue*" on line 19.

Experiments

In this section, I will be running each algorithm described in previous section to generate solutions for given 4x4, 5x5, 6x6, 7x7, 8x8, 9x9, and 10x10 pegboard puzzles. The data I will be collecting for each experiment will be execution duration, memory utilization, solution (whether a solution was found), and number of nodes searched. The data collected with be analyzed in next section.

Experiment constrictions. For each experiment, there will be a limit on resources allowed duration execution: time and memory. I will only allow for the programs to run for 2-hours before a termination is initiated. I will only allow for the programs to use 3Gb of random-access memory before a termination is initiated. These cases are to allow for (1) I do not spend all my time testing and (2) does not crash my computer. If either case occurs during execution, then a failure will be saved in the data collection.

$$S_n = \left\{ \begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{bmatrix} \middle| a_{i,j} \in \{1,0\}, \exists!\, a_{i,j} = 0 \right\} \qquad [2]$$

Experiment inputs. The program will only have one input, the initial pegboard state. For my experiments, I am defining an initial pegboard state by [2] – i.e., NxN matrix (S) with elements in the set of {1,0} will have only one zero element. Using this definition, I created initial states for 4x4, 5x5, 6x6, …, 10x10 by choosing where the zero element is placed. Therefore, the input set is as follows:

$$Input\ Set{:}\ \left\{\{S_4 | a_{2,1} = 0\}, \{S_5 | a_{2,1} = 0\}, \{S_6 | a_{2,2} = 0\}, \{S_7 | a_{2,2} = 0\}, \{S_8 | a_{5,3} = 0\}, \{S_9 | a_{5,5} = 0\}, \{S_{10} | a_{6,5} = 0\}\right\}$$

Input set elements $S_6, S_8, \& \ S_9$ have known solutions (Bell, 2016). S_7 & S_{10} are unknown; they or may not have solutions.

Table 1 shows the data collected for each experiment. Input size is which input was run from declared above. Output is result from running input – i.e., was there a solution; where solution has three possible values S, NS, and F which are equivalent to solution, no solution, and failure (due to memory usage or time), respectively. #Searched is the number of nodes searched. Duration is process time in seconds. Memory is the max amount of memory allocated during execution.

Table 1. Algorithm Data

	Input Size	Output	#Searched	Duration (sec)	Memory (MB)
Astar	4x4	S	15	0.0625	0.149416
Astar	5x5	NS	706391	683.828125	406.334576
Astar	6x6	S	2689	1.890625	3.207472
Astar	7x7	F	2928132	4913.375	2800.480776
Astar	8x8	F	2392079	5266.15625	2800.131576
Astar	9x9	F	1895795	5476.34375	2800.559416
Astar	10x10	F	1567451	5620.53125	2801.401596
GBS	4x4	S	18	0.046875	0.146436
GBS	5x5	NS	706391	675.296875	406.332556
GBS	6x6	S	3432	2.375	3.554396
GBS	7x7	F	2928895	4904.671875	2800.428732
GBS	8x8	F	2411359	5272.90625	2800.349076
GBS	9x9	F	1930829	5710.40625	2800.285012
GBS	10x10	F	1577632	5699.515625	2800.406088
DFS	4x4	S	56	0.078125	0.2033
DFS	5x5	F	706391	671.625	406.54498
DFS	6x6	S	1349689	1828.234375	1027.7365
DFS	7x7	T	2928497	5370.65625	2800.78906
DFS	8x8	T	2411738	5889.84375	2800.401828
DFS	9x9	T	1933020	5310.515625	2800.918084
DFS	10x10	T	1580794	5197.390625	2800.647912
BFS	4x4	S	1870	0.703125	1.439138
BFS	5x5	F	706391	669.390625	621.952696
BFS	6x6	T	221340	424.203125	2808.841704
BFS	7x7	T	137786	339.921875	2807.888504
BFS	8x8	T	109780	343.15625	2801.219944
BFS	9x9	T	71343	244.859375	2805.881672
BFS	10x10	T	60562	247.078125	2802.235436

Analysis

In this section, performance analysis of each algorithm will be conducted using data collected in Table 1.

Before I get into the analysis, I need to define theorical performance criteria for algorithms. A standard algorithm evaluation usually consists of:

- Completeness: Is the algorithm guaranteed to find a solution when there is one?
- Optimality: Does the strategy find the optimal solution? Optimal solution is defined as, “lowest path cost among all solutions” (Stuart Russell, 2019, p. 68). Where path cost is distance from initial state to goal state.

- Time complexity: How long does it take to find a solution?
- Space complexity: How much memory is needed to perform the search? (Stuart Russell, 2019, p. 80)

However, for this report, I will be only concerned with latter two (time and space complexity) because BFS, DFS, GBS, and A* are complete (Stuart Russell, 2019) and all solution found for pegboard game application are optimal solutions.

In artificial intelligence (AI), time and space complexity are calculated from a graph which often represented by implicitly by the initial state, actions, and transition model and is frequently infinite. For these reasons, complexity is expressed in terms of three quantities: *b*, the branching factor (i.e., maximum number of successors of any node); *d*, depth of the shallowest goal node (i.e., the number of steps along the path from the root); and *m*, the maximum length of any path in the state space (Stuart Russell, 2019, p. 80). For pegboard application, *m* is equal to *d* since goal state is the end state. Time complexity is number of nodes generated during search, and space complexity is maximum number of nodes stored during search. In analysis section, I will be evaluating performance using combination of time and space complexity.

Time and space complexities for each algorithm is the same, $O\left(b^d\right) = O(b^m)$. Table 2 shows the time and space complexities for each algorithm outside of pegboard application. The time and space complexity formulations for time complexities, along with, BFS space complexity of can be found in (Stuart Russell, 2019). DFS, GBS, and A* space complexity was calculated by determining the number of nodes generated (worst case) has shown in equation [3].

$$1 + b + b^2 + \cdots + b^m = O(b^m) \qquad [3]$$

Table 2. Time and Space Complexity

Algorithm	Time Complexity	Space Complexity
Breadth-First Search	$O(b^d)$	$O(b^d)$
Depth-First Search	$O(b^m)$	$O(b^m)$
Greed-Best Search	$O(b^m)$	$O(b^m)$
A* Search	$O(b^m)$	$O(b^m)$

Now that complexities are established, observations can be made between the complexities and data collected (Table 1). First clear observation is that for the solvable inputs (4x4 and 6x6), GBS and A* outperformed DFS and BFS (BFS did not even solve). GBS and A* outperformed in both time and space resources allocation by factors of magnitude. Second observation that can be made is that after 6x6, space required was over whelming for all algorithms. This is because all the algorithms have same complexities, therefore, if a solution is hard to find (more need to be searched), then the memory and time will increase dramatically. Hence failures across the board.

References

Bell, G. I. (2016, Jan 23rd). *Peg Solitaire*. (recmath) Retrieved Sep 26th, 2020, from http://recmath.org/pegsolitaire/

Drossos Kikolaos, P. A. (n.d.). *5.5 Heuristic Algorithms* . Retrieved from Algorithms Tutoring Web Page: http://students.ceid.upatras.gr/~papagel/project/contents.htm

Li, Y. (2020). CS480/580 Introduction to Artificial Intelligence Assignment 1. Norfolk: Old Dominion University.

Stuart Russell, P. N. (2019). *Artificial Intelligence: A Modern Approach.* Bergen County: Prentice Hall Press.

Viva. (2020, NA NA). *8 Difference Between DFS and BDFS in Artificial Intelligence*. Retrieved from VivaDifferences: https://vivadifferences.com/difference-between-dfs-and-bfs-in-artificial-intelligence/