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.

# 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. [1]

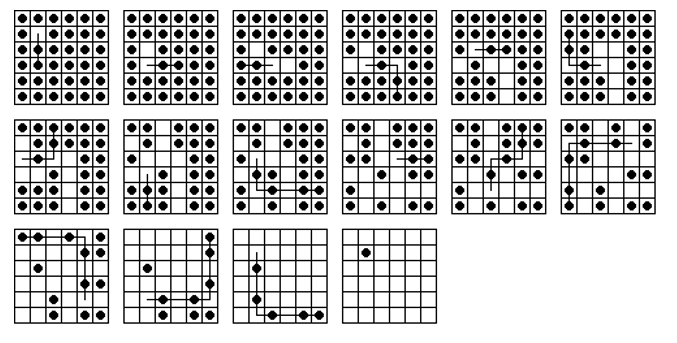


Figure . 6x6 Solution [2]

# 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), A\* (A-star) search, and heuristic to support greedy-best search and A\*.

## Breadth First Search Algorithm

*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

*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

*Manhattan (State S)*

*1. create Sum*

*2. foreach row in S*

*3. foreach column in S*

*4. if S (row, column) does not equal 0*

*5. Sum = Sum + |row – max row| + |column – max column|*

*6. end if*

*7. end for*

*8. end for*

*9. return Sum*

## Greedy Best Search

*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\* (A-Star)

*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*

# 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 (1) execution duration, (2) memory utilization, and (3) solution (whether a solution was found). 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.

Experiment inputs. The program will only have one input, the initial pegboard state. For my experiments, I am defining an initial pegboard state by (1) – 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:

Table 1 shows the data collected for each experiment. Solution has three possible values 1, 0, and -1 which are equivalent to solution, no solution, and termination, respectively.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Solution | | | | | Duration (s) | | | | Memory (MB) | | | | |
|  | DFS | BFS | GBS | A\* | DFS | BFS | GBS | A\* | DFS | BFS | GBS | A\* |
| 4x4 | 1 | 1 | 1 | 1 | 0.0781 | 0.6718 | 0.0781 | 0.0625 | 0.2023 | 1.4389 | 0.1318 | 0.132 |
| 5x5 | 0 | 0 | 0 | 0 | 651.91 | 668.64 | 651.27 | 663.98 | 406.545 | 621.95 | 406.316 | 406.317 |
| 6x6 | 1 | -1 | 1 | 1 | 1799.25 | 429.59 | 570.52 | 570.45 | 1027.74 | 3000 | 336.463 | 336.458 |
| 7x7 |  |  |  |  |  |  |  |  |  |  |  |  |
| 8x8 |  |  |  |  |  |  |  |  |  |  |  |  |
| 9x9 |  |  |  |  |  |  |  |  |  |  |  |  |
| 10x10 |  |  |  |  |  |  |  |  |  |  |  |  |

# Analysis