Application of Greedy and A* Algorithm in Solving Flood-It Puzzle

Morgen Sudyanto 13518093
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
13518093@std.stei.itb.ac.id

***Abstract*—Flood-It is a puzzle where a player is given a grid separated into regions with various colors. The player need to flood the puzzle by coloring the top-left region with the minimum number of moves. The goal of this paper is to examine the performance of greedy and A* algorithm in finding the solution to solve the Flood-It puzzle.**

Keywords—Flood-It; greedy; A*; heuristic; NP-hard

I. INTRODUCTION

Flood-It is a puzzle made by LabPixies, an Israeli developer team, in March 2009. The game starts with a *n* x *n* grid that is separated into different regions with *c* colors. The player then needs to expand the top-left region by flooding it with different colors. The game ends when there is only one region left in the grid, which is the grid itself. This puzzle has already been proven to be NP-hard for *c* >= 3 [1].

In this paper, the author compares the difference in performance between two heuristic-based algorithms, Greedy and A*. The greedy algorithm tries to find the optimal solution in each step, whereas the A* algorithm uses both current and approximate knowledge to get to the optimal solution. Both of these algorithms can be used to find a solution to this puzzle, though the optimality of the solution can't be guaranteed.

II. THEORIES

A. *Flood-It Puzzle*

Flood-It is a puzzle made by LabPixies, an Israeli developer team, in March 2009. The game starts with a *n* x *n* grid that is separated into different regions with *c* colors. Let us call the top-left region the seed region. Flooding is an operation where the player chooses a color, change the seed region into the selected color, and floods the neighboring regions that has the same color as the chosen color. The player need to minimize the number of flooding operations done.

Below is an example of the puzzle with *n* = 5 and *c* = 3.

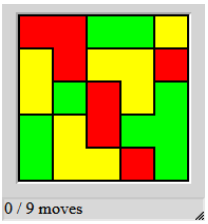

Figure 1: Starting a puzzle with *n* = 5 and *c* = 3. (Created using [2])

When the player chooses the green color, the grid turns into:

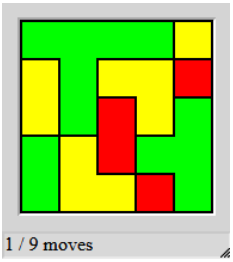

Figure 2: Grid after choosing green. (Created using [2])

A possible sequence of moves is: Green, Red, Yellow, Green, Red with a total of 5 moves. This solution is not optimal. This puzzle can also be solved with this sequence of moves: Green, Yellow, Red, Green. In fact, this sequence with 4 moves is optimal. There is no other way to solve this puzzle in less than 4 moves.

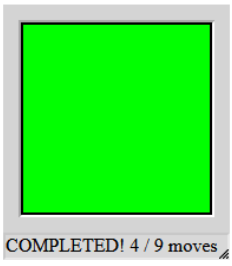

Figure 3: Completed puzzle. (Created using [2])

Finding the optimal moves has already been proven to be a NP-hard problem, as this game can be reduced from the shortest common supersequence problem [1].

B. *P and NP*

In computer science, a decision problem is a problem whose solution can only be "yes" or "no". An example of this is: Given a number. Determine if the number is prime. If the number is 6, then the answer is "yes". If the number is 7, then the answer is "no". There are no other possible answers to this problem.

Decision problems can be classified into two complexity classes: P (Polynomial) and NP (Non-deterministic Polynomial). A decision problem is classified as P if the problem can be solved with an algorithm whose complexity can be stated as a polynomial function, therefore requires a polynomial time to complete. A decision problem is classified as NP if the problem can be solved with a non-deterministic algorithm, whose verification can be done within a polynomial time. Here, verification means that if a candidate answer is given, the correctness of this solution can be determined within a polynomial time. For now, P is a subset of NP, as nobody has proven P = NP or P ≠ NP.

NP-Complete problems are decision problems that are classified as NP and all NP problems can be reduced into NP-Complete problems within a polynomial time. If an NP-Complete problem can be solved within a polynomial time, then all NP problems that can be reduced into this problem can be solved within a polynomial time. Examples of well known NP-Complete problems include the Satisfiability Problem (SAT), the Travelling Salesman Problem and the Sum of Subset Problem. NP-Hard problems are problems that are as hard as NP-Complete problems, but not limited to decision problems.

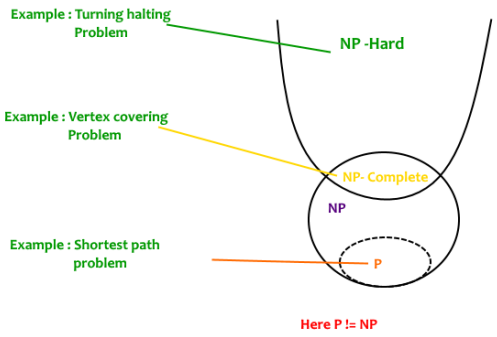

Figure 4: Euler diagram for P, NP, NP-Complete and NP-Hard set of problems. [3]

Although there are no polynomial time algorithms to find the solution of NP-Complete problems, we can use several techniques to find solutions to these problems, such as approximation, randomization, restriction, parameterization and heuristic.

Heuristic is an algorithm that can work reasonably well, but does not always produce optimal solutions. Greedy and A* are examples of algorithms that utilize heuristic approach.

C. *Greedy Algorithm*

A greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage with the intent of finding a global optimum [4].

The greedy algorithm consists of several elements:

- Candidate set (C), set of possible solutions.
- Solution set (S), set of selected solutions chosen from the candidate set.
- Selection function, function to select a candidate from the candidate set.
- Feasible function, function to determine if a candidate can be selected.
- Objective function, function that calculates a value based on a candidate set or solution set.

A common scheme for the greedy algorithm is as follows:

```
function greedy(input C: candidate_set) →
candidate_set
Declaration
  x : candidate
  S : candidate_set
Algorithm
  S ← {}
  while (not SOLUTION(S)) and (C ≠ {}) do
    x ← SELECTION(C)
    C ← C - {x}
    if (FEASIBLE(S ∪ {x}) then
      S ← S ∪ {x}

  if (SOLUTION(S)) then
    return S
  else
    write('No solution')
    return {}
```

The greedy algorithm does not always guarantee an optimal solution. Although it always choose a local optimal choice, that choice may lead to a suboptimal solution. Different selection functions may choose different choices. Finding a "good enough" selection function is the key in devising a greedy algorithm.

D. *Pathfinding Algorithms*

Pathfinding is the process of finding a route between two points that meets a defined criteria. This criteria can be anything from cheapest, shortest, most profitable, etc.

There are several pathfinding algorithms, such as:

- Breadth First Search (BFS)

  Starting from a node, the graph is traversed layer by layer. BFS explores all neighbor nodes at the current depth level before moving to the nodes at the next depth level. The agenda can be maintained using a queue. In an unweighted graph, BFS will always find the shortest path between any two nodes.

- Depth First Search (DFS)

  DFS explores as far as possible along a branch, and backtrack to the previous nodes if no solution is found in that branch. The agenda can be maintained using a stack. In an unweighted graph, DFS will not always find the shortest path between any two nodes.

- Dijkstra and Uniform Cost Search (UCS)

  Dijkstra works in a similar way to BFS. Other than traversing layer by layer, Dijkstra chooses the current shortest node to the source node in every step. The agenda can be maintained using a priority queue. Dijkstra will always find the shortest path in any graph (both weighted and unweighted). Dijkstra can be regarded as a variant of UCS, where the goal node is not defined and the processing continues until all nodes have been removed from the priority queue.

- Greedy Best First Search (GBFS)

  GBFS works by choosing the current "shortest" node to the goal node in every step. As the distance (cost) of each node to the goal node is unknown, we estimate the cost using a heuristic function. The problems with GBFS are that the algorithm may get stuck in a local optimum, it is undoable and the heuristic function may not be accurate enough. Therefore, GBFS will not always find the shortest path in a graph.

- A*

  A* algorithm works by combining both UCS and GBFS, where the previous cost (UCS) and an estimated future cost (GBFS) is combined (added together) to determine the value of a node. The optimality of A* algorithm is determined by the admissibility of the heuristic function. A heuristic function is admissible if for every node n, $h(n) \leq h^*(n)$, where $h(n)$ is the estimated cost and $h^*(n)$ is the true cost to reach the goal node from n. An example of an admissible heuristic function is the straight line distance between two points in a map, due to the triangle inequality. If the heuristic function is admissible, A* will always find the shortest path in any graph (both weighted and unweighted).

A common scheme for the A* algorithm is as follows:

```
function astar(input start: node, input end: node) → path
Declaration
  q : priority_queue of state, sorted according to increasing gval + hval
  cur, next : state {state consists of gval, hval, node}
Algorithm
  cur.gval ← 0 {starting node}
  cur.hval ← h(cur.node) {heuristic}
  cur.node ← start
  q.push(start)
  while (not EMPTY(q)) do
    cur ← q.top()
    q.pop()
    if (cur.node = end) then
      return RECONSTRUCT_PATH(cur)
    for (every possible transition) do
      next.gval ← cur.val + GET_DISTANCE(cur, transition)
      next.node ← GET_NODE(cur, transition)
      next.hval ← h(next.node) {heuristic}
      q.push(next)
  write('No solution')
  return {}
```

E. Flood Fill Algorithm

Flood fill is an algorithm that is used to find a bounded area that is connected to a given node. Flood fill can be implemented using different kinds of algorithms. We can implement flood fill using a DFS-like algorithm. In DFS, the algorithm stops when the goal node is found. In flood fill, there is no goal node. The algorithm stops only if the stack is empty.

Below is an example of how the flood fill algorithm works. The cells colored white are cells that are unexplored and available. The cells colored black are cells that are unavailable (can't be visited). The cells colored green are cells that have been visited by the flood fill algorithm. The starting cell is the bottom-left cell. The cells that are connected to the starting cell is filled step by step, until all the cells become green. Notice that the flood fill algorithm could not pass the black cells, and therefore leaving the top-right cell unvisited.

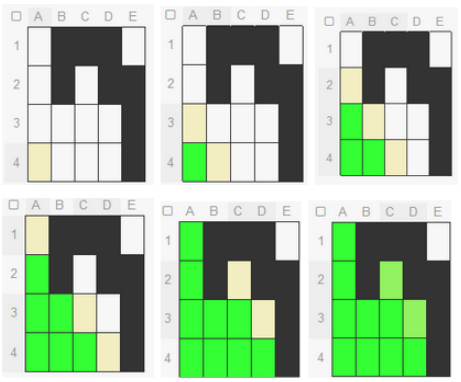

Figure 5: Illustration of the flood fill algorithm, starting from the bottom-left cell.

Below is the pseudocode for implementing flood fill using a DFS-like algorithm:

```
procedure floodfill(input start: point)
Declaration
  p, cur, next : point
  s : stack of point
Algorithm
  s.push(start)
  while (not EMPTY(s)) do
    cur ← s.top()
    s.pop()
    for (every possible direction) do
      next ← GET_POINT(cur, direction)
      if (not VISITED(next)) then
        s.push(next)
  return
```

III. Greedy Algorithm in Solving Flood-It

A. Identifying Elements

In order to solve the puzzle using the greedy algorithm, we need to first identify the elements.

1. Candidate set

The candidate set for this program are all the possible sequence of moves that can be selected by the player. This sequence can be infinitely long, as a cycle is possible when solving this puzzle.

For example, here is a puzzle with $n = 4$ and $c = 4$. The colors are represented with an integer ranging from 1 to c.

1133

3421

2341

2412

The player can make an alternating move: 212121...., making the sequence length infinite.

2. Solution set

The solution set consists of sequences of moves that lead to the puzzle solution. The puzzle is considered complete if the puzzle has only one color. An example of a complete puzzle is:

2222

2222

2222

2222

The move that can be made to the puzzle in part 1 to make this is: 324132.

3. Selection function

We can select any color from the possible colors, but in order to minimize the number of moves, we need to choose a color which (hopefully) leads to the optimal solution (shortest sequence of moves).

Observe that in order to reach the solution, we need to expand the top-left region as much as we can. In fact, if we don't expand it, we may make an infinite sequence. Intuitively, this can also be the basis of our selection function. In each move, we choose a color which expands the top-left region the most. In case of a tie, we can choose any color.

The author demonstrates this using the puzzle in section III. A. 1. We have this puzzle:

1133

3421

2341

2412

We skip color 1, as it is clearly useless to choose the same color. We also skip color 2, as it doesn't expand the region. Color 3 can be a chosen color, and it expands the region by 3 cells. Color 4 can also be chosen, but it only expands the region by 1 cell.

As color 3 is the best color in this step, we choose to color the region with color 3, turning the puzzle into:

3333

3421

2341

2412

4. Feasible function

It is always feasible to choose any color in this puzzle, but choosing some color may be worse than others, as discussed in section III. A. 3.

5. Objective function

The objective function of this puzzle is to minimize the number of moves to reach the solution.

B. Complexity Analysis

In each step we try to flood the puzzle using c colors and the flooding process takes n^2 steps. Notice that in each step, we eliminate at least one region. There are at most n x n regions in the puzzle, where each cell has a different color to all its neighbors. Therefore, the worst case complexity for the greedy algorithm is $O(cn^4)$.

IV. A* Algorithm in Solving Flood-It

A. Heuristic Function

To get an optimal solution, the heuristic function that is used to estimate the cost from current state to the goal state must always be less than the true cost. Some of you may think that the number of regions in the puzzle is a good estimate. It is not a good estimate as the number of eliminated regions in every move can be greater than 1. In fact, it is pretty rare to only eliminate one region. Thus, this heuristic function is not admissible .

A better estimate is to divide the number of regions with c, the number of possible colors. This is due to the nature of the puzzle, where clearing a color can expose the top-left region to more regions. In a 14x14 puzzle with 6 color, there is a maximum of 196 regions. Dividing this with 6 gives us 32.67. This is a pretty good estimate, as usually the game can end with less than 30 moves. A randomly generated puzzle only has around 120 - 140 regions, making the expected average move length around 20 - 24. The classic game allows the player to make a maximum of 25 moves.

The optimal solution to the puzzle at section III. A. 1. can actually be found using the A* algorithm. A possible sequence of optimal moves is: 43412.

B. *Complexity Analysis*

We can calculate the value of the heuristic function by using the flood fill algorithm, keeping track of the number of regions filled. This process takes n x n operations. The search space of this algorithm is c^d, where c is the number of colors and d is the depth of the state space search tree. The agenda is implemented in a priority queue, with an insertion complexity of O(log n). Therefore, the complexity of this algorithm is O($n^2c^dlogc^d$).

V. Performance Analysis of Greedy and A* Algorithm in Solving Flood-It

A. *Implementation*

In order to find out the difference in performance of the Greedy and A* algorithm, the author implements both approaches using C++. No external libraries are used. My program also allows the user to play the game themselves and measure how well do the user perform compared to both of the algorithms.

Here are some screenshots from the program:

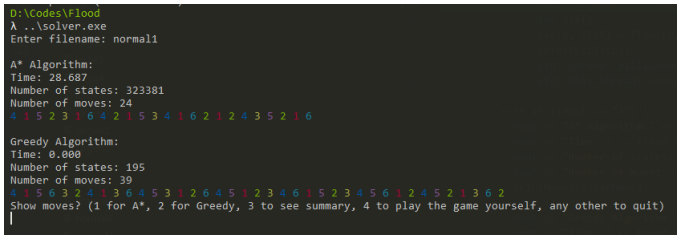

Figure 6: Solving the normal1 puzzle.

Figure 7: Playing the game yourself.

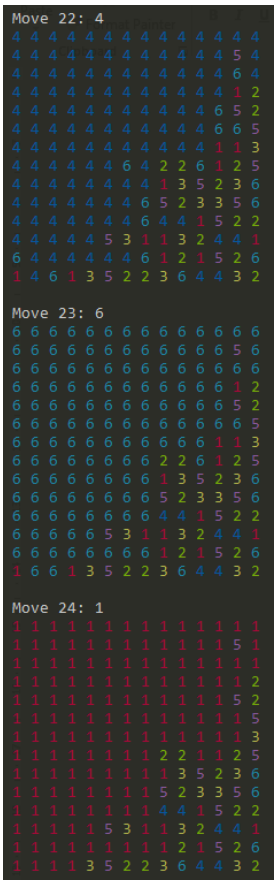

Figure 8: Showing moves taken by the solver step-by-step.

B. *Testing*

The author makes 5 batches of tests: small, smalln, normal, large and largen, each consisting of 5 puzzles. A generator has also been written to be used if a user wants to make a random puzzle with his/her own parameters. Small, normal and large puzzles are all 14x14 puzzles, but with c values of 5, 6 and 7 respectively. Smalln, normal and largen puzzles all have 6 colors, but with grid sizes of 12x12, 14x14 and 16x16 respectively. The normal batch is reused. The author also try to solve the 1st puzzle of every batch, serving as a benchmark to both of these algorithms.

1. c as free variable (small, normal, large)

n is always 14. c is 5, 6, and 7 for small, normal and large respectively. Time is measured in seconds.

Greedy						
No	Small		Normal		Large	
	Time	Moves	Time	Moves	Time	Moves
1	0.000	20	0.015	39	0.000	31
2	0.000	28	0.000	31	0.015	36
3	0.000	25	0.016	29	0.015	31
4	0.000	28	0.000	32	0.000	38
5	0.000	26	0.000	33	0.000	36
Avg	0.000	25.4	0.006	32.8	0.006	34.4

A*						
No	Small		Normal		Large	
	Time	Moves	Time	Moves	Time	Moves
1	0.015	17	29.063	24	24.484	26
2	0.046	23	0.328	23	401.813	25
3	0.031	21	0.172	22	151.657	22
4	0.109	23	4.828	24	16.999	25
5	0.015	18	0.328	24	42.078	24
Avg	0.432	20.4	6.944	23.4	127.406	24.4

Table 1: Performance of Greedy and A* in solving 14x14 puzzle with different *c*.

For the first puzzle in each batch, the author took 22 moves for small, 31 moves for normal, and 37 moves for large.

2. *n* as free variable (smalln, normal, largen)

c is always 6. *n* is 12, 14, and 16 for small, normal and large respectively. Time is measured in seconds.

Greedy						
No	Small		Normal		Large	
	Time	Moves	Time	Moves	Time	Moves
1	0.000	23	0.015	39	0.015	34
2	0.000	27	0.000	31	0.000	36
3	0.000	25	0.016	29	0.000	31
4	0.000	23	0.000	32	0.000	38
5	0.016	30	0.000	33	0.015	36
Avg	0.003	25.6	0.006	32.8	0.006	35
A*						
No	Small		Normal		Large	
	Time	Moves	Time	Moves	Time	Moves
1	1.219	19	29.063	24	0.235	26
2	1.265	20	0.328	23	406.890	25
3	0.343	18	0.172	22	153.437	22
4	0.422	20	4.828	24	17.125	25
5	1.235	20	0.328	24	42.093	24
Avg	0.897	19.4	6.944	23.4	123.956	24.4

Table 2: Performance of Greedy and A* in solving a 6-color puzzle with different *n*.

For the first puzzle in each batch, the author took 22 moves for small, 31 moves for normal, and 28 moves for large.

C. *Analysis*

As we can see from the data above, the greedy algorithm produces sequences with more moves than A*, but performs the computation very fast (virtually instant). A* also uses a lot of memory, there are some large cases where it uses around 6GB RAM. This is due to the search state space tree that grows larger and larger as nodes with larger depths are explored.

The runtime of A* increases exponentially, both on varying *c* and *n*. A possible reason for this is that more moves are required to solve the puzzle, and thus increasing the complexity exponentially. We can also see that the runtime is not evenly distributed, implying that the positions of regions in the test cases greatly impacts the runtime. As the heuristic function used is admissible, the solution produced by A* algorithm is guaranteed to be optimal, i.e. there are no shorter sequence of moves that lead to the solution.

Although greedy produces a suboptimal result, this result is still better than solving the puzzle manually. Although the number of the author's moves are pretty close to the greedy algorithm, players can learn tricks and strategies to improve their score.

All the source code and test cases can be found in my GitHub repo: https://github.com/moondemon68/Flood-It-Solver.

VI. Conclusion

Both greedy algorithm and A* algorithm can be used to solve the Flood-It puzzle. Greedy algorithm uses a short time and takes low memory space to produce a suboptimal solution. A* algorithm uses a long time and takes a lot of memory space, but produces an optimal solution. There may be better heuristic functions or algorithms that can lead the solution of this puzzle, and finding them can be a topic for future researches in computer science.

Video Link at Youtube

The author makes a video explaining how Greedy and A* algorithm explained here works in a nutshell. The video is made in Bahasa Indonesia and can be found here: https://youtu.be/pqPSelI-jZs.

Acknowledgment

The author would like to thank God for His blessings and guidance in allowing the author to be able to finish the paper. The author would also like to thank Dr. Ir. Rinaldi, M. T. as the lecturer of the IF2211 – Algorithm Strategies course for the guidance and knowledge he shared. The author would also like to thank all of the author's colleagues for all the support and inspiration they had given.

References

[1] Clifford R., Jalsenius M., Montanaro A. and Sach B, "The Complexity of Flood Filling Games," Arxiv (https://arxiv.org/abs/1001.4420), accessed on May 1st, 2020.

[2] Tatham S., Flood-It game generator, (https://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/flood.html), accessed on May 1st, 2020.

[3] Euler diagram for P, NP, NP-Complete and NP-Hard set of problems. Retrived from https://www.geeksforgeeks.org/np-completeness-set-1/, accessed on May 1st, 2020.

[4] Black, Paul E. (2 February 2005). "greedy algorithm". Dictionary of Algorithms and Data Structures. U.S. National Institute of Standards and Technology (NIST). Retrieved 17 August 2012.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Jakarta, 3 Mei 2020

Morgen Sudyanto - 13518093