Navigation on Turkey Map

Alp Gokcek
gokcekal@mef.edu.tr
MEF University
January 3, 2021

***Abstract*—Search problem is a well-known issue in the field of Artificial Intelligence. There are various algorithms designed for this problem, such as Dijkstra, A-star, Breadth-First Search, and much more. This paper has demonstrated and applied one of the most efficient search algorithms for the problem, which is the A-star algorithm for finding the shortest path on a given map.**

***Keywords*—Shortest-path, A-star search, search algorithms.**

I. Introduction & Problem Statement

Searching is a comprehensive technique of problem-solving in AI. Before evaluating the problem, there are several terminological words and phrases that have to be known.

A. Terminology

There are several terminological things to know during the paper. These are:

- **Node(State):** All potential outcomes of the problem with a unique identification
- **Transition:** Traveling between states or nodes.
- **Starting Node:** Where to start searching
- **Goal Node:** The target to stop searching.
- **Cost:** Cost for the path from a node to another node. This value can be the distance between nodes, time, and much more..
- **g(n):** This function returns the exact cost of the path from the starting node to any node n.
- **h(n):** This function returns heuristic estimated cost from node n to the goal node.
- **f(n):** Lowest cost in the neighboring node n

[3]

B. Problem

These algorithms aim to travel from a starting node to a goal node by transiting through transitional nodes. It is not just used in shortest-path problems but also in single-player games like puzzles, Sudoku, and many more. For instance, in N-puzzle or also known as Sliding Puzzle, there is a square-shaped board, and it has N-tiles with one empty space that is available for sliding a block. In this case, the approach is to find all possible solutions through moving each block one by one until reaching the goal state.

In the shortest-path problem, as the name refers, the main goal is to find the shortest path between 2 nodes, in our case these nodes are cities, if any possible path exists. [2] This problem is also a search problem and a search operation is performed from starting node to the goal node. There are several algorithms designed for solving searching problem and they have different advantages over each other. The most known algorithms can be listed as:

- Breadth-First Search
- Depth-First Search
- A-star(A*) Search
- Greedy Best First Search
- Uniform Cost Search (Dijkstra's Algorithm)

The most used ones are A* Search Algorithm, Breadth-First, and the Dijkstra Algorithm. [1] Search pattern differences of Breadth-First, Dijkstra and A-star search algorithms is illustrated in Figure 1.

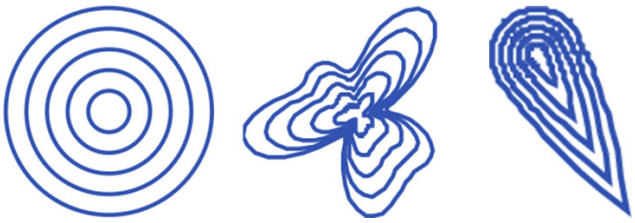

Fig. 1: Search patterns of Breadth-First, Dijkstra and A* [1]

C. A* (A-star) Search Algorithm

A algorithm* is a heuristics-based algorithm for the shortest path-finding. It is the optimized version of the Dijkstra Algorithm for a single target. A* algorithm aims to reach only the goal node in the shortest path possible, whereas, in the Dijkstra Algorithm, all possible paths from all nodes would be found. In other words, the Dijkstra algorithm is finding the path blind by calculating all the paths possible, whereas, in A-star, it senses the shortest path with the help of the heuristic function. So, if the problem is to reach from one node to another, the Dijkstra Algorithm is not optimal. Therefore, the need for the A* algorithm arose. Its working principle is based on heuristics, at each node that expanded, the heuristic function would be executed, and the cost of transition is calculated. [1] The pros and cons of the algorithm could be found below.

1) Advantages: The A-star algorithm is a better performing algorithm compared to other search algorithms. There are several pros that can be mentioned:

- On the worst case, its complexity is $\mathcal{O}(n \log n)$. Other popular search algorithms such as Dijkstra Algorithm has the complexity of $\mathcal{O}(n^2)$.
- A-star is optimal since there is a significant difference between the number of nodes expanded during the execution.
- If the branching factor is finite and each transition has a fixed cost, then the algorithm becomes complete. [4]

2) Disadvantages: As each algorithm has downsides, A-star has downsides too. Several cons that can be listed are:

- A-star algorithm uses a lot of memory.
- The execution speed is highly dependent on the heuristic function $h(n)$. The heuristic function must be admissible
- A-star algorithm is not a uniformed algorithm; in other words, it can only target one node in one execution. For instance, if there are n targets on the graph, A-star would be executed n times, which may or may not be the optimal solution.

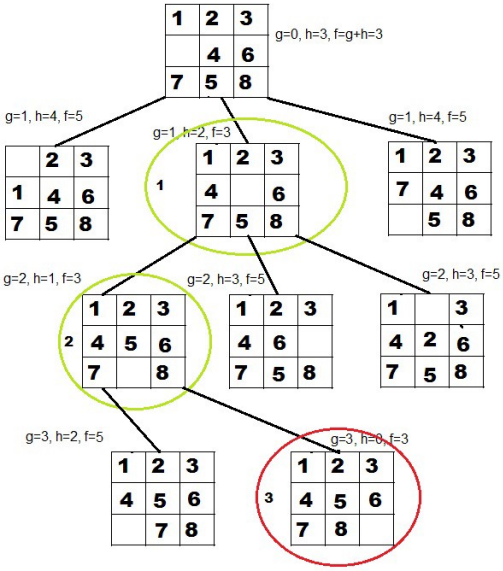

Fig. 2: Solution for N-puzzle with A* algorithm

In Figure 2, a sample solution could be found which is solved with A-star algorithm.

D. Breadth-First Search Algorithm

Breadth-First search is the intuitive solution for everyone. It is simple compared to other algorithms and it does not consider the costs, thus it explores equally in all directions. This method is significantly useful algorithm but it is not frequently used in shortest-path problems since it is not optimal for this problem at all.

E. Dijkstra Algorithm

Dijkstra's Algorithm calculates the shortest path from one desired node of the graph to every other node. It is widely used and adopted for many different applications, such as computer networks, road maps, social platforms and much more. There are various implementations of the algorithm; the main difference between them is how they store the vertices of the graph, denoted with Q. They can be stored in regular arrays, linked lists, adjacency lists, or tree structures. The algorithm's running time can be represented with the number of edges $|E|$ and the number of vertices $|V|$ using big-o notation. Running time mainly depends on these structures, storing Q in arrays it is $\mathcal{O}(|V|^2)$ but with the help of *min-priority queue* it drops down to $\mathcal{O}((|E| + |V|) \log |V|)$ time in the worst case.

II. Solution Design Process

For this problem, the initial approach was to implement the Dijkstra Shortest Path Algorithm and prompt the outcome to the user after the execution of the algorithm. As mentioned in the previous section, Dijkstra Algorithm is a uniformed algorithm; in other words, algorithm blindly finds the shortest paths from each node to every node. This approach is time consuming and not optimal since our aim is to find just a path between start node to goal node. It is redundant to find paths that will not be used. Thus, there was a need to improve the performance of the program. The second approach was to implement the A-star Search Algorithm. As mentioned in the previous section, the A-star algorithm was a better choice in case of this problem if the implementation uses a admissible heuristic function. After the research, it was decided that the best possible heuristic function for this problem is Manhattan distance formula.

A. Manhattan Distance

Mathattan distance or also known as Taxicab geometry is a type of geometry in which the standardized Euclidean geometry distance function is replaced by a new metric in which the distance between the two points corresponds to the sum of the absolute differences between their Cartesian coordinates. [5]

Formally, the Manhattan Distance d_1 between two vectors $\vec{p}$ and $\vec{q}$ where $\vec{p} = (p_1, p_2, ...)$ and $\vec{q} = (q_1, q_2, ...)$ in n-dimensional vector space can be defined as:

$$d_1(\vec{p}, \vec{q}) = \|\vec{p} - \vec{q}\|_1 \sum_{i-1}^{n} |p_i - q_i|.$$

For our problem, the map is a 2-D plane. Thus, the Manhattan distance between *City A* $\vec{city_a} = (city_{a_x}, city_{a_y})$ and *City B* $\vec{city_b} = (city_{b_x}, city_{b_y})$ is $|city_{a_x} - city_{b_x}| + |city_{a_y} - city_{b_y}|$.

III. Implementation

There were 3 main components for this problem and during the implementation phase, I have followed this order. These tasks were:

1) Creating and collecting data
   - Finding a map
   - Creating the data
2) Deciding and implementing algorithm
   - Heuristic Function
   - Shortest-path Algorithm
3) Creating the GUI

. With the help of this ordered list, I did not face any difficulties and completed these tasks easily.

A. Creating and collecting data

In this part there were two tasks which are finding a map and creating the data. In first task, it was pretty easy to find a map that can be used. I have picked a Turkey map that has spotted the centers of provinces with circles, so that I can create same circle over it and use pixel-coordinates in the algorithm for distance calculation as well. In the second task, I have made an comprehensive research whether there is a ready-to-use data published in web. I could not find any useful resources. Thus, I have started to create my own data. I have created files for cities and the roads between the cities. In Turkey, there are 81 provinces. In *cities* file, I have added the names of the provinces and their pixel-coordinates one-by-one. After that, I have researched the roads between neighbor cities. I have added them to the *roads* file one-by-one as well.

B. Deciding and implementing algorithm

In this part, similar to the previous part, there were two tasks which are deciding and implementing heuristic function and shortest-path algorithm. I have made an comprehensive research while deciding the search algorithm. During the research, I have found out that A* search algorithm with the Manhattan distance function as the heuristics was the best and appropriate solution. The reasons behind the chosen heuristic function and shortest-path algorithm is already mentioned in Section I and Section II.

Algorithm 1 Manhattan Distance as Heuristic Function

function HEURISTIC($city_a, city_b$)
    $x_1 \leftarrow$ x position of $city_a$
    $y_1 \leftarrow$ y position of $city_a$
    $x_2 \leftarrow$ x position of $city_b$
    $y_2 \leftarrow$ y position of $city_b$
return absolute of $(x_1 - x_2)$ + absolute of $(y_1 - y_2)$
end function

In *Algorithm 1*, pseudocode of the heuristic function could be found and in *Algorithm 2*, pseudocode of the shortest-path could be found. I have made the implementation of this pseudocode in Python programming language. I have created classes for *City*, *Road*, and *Map*.

C. Creating the GUI

Last part was to create a GUI for the user. As I mentioned previously, I have used Python programming language while implementing the shortest-path algorithm. Thus, I have chosen to use Tkinter library to create the GUI which I have already used in multiple projects.

IV. Testing and Bug-fixing

Before implementing the GUI, I have passed 2 *City* instances to the shortest-path method and check whether the result is true. After implementing the GUI, I have tested the program comprehensively. While checking the output is correct, I have checked and compared the path and final cost of the path which is an estimation of distance in terms of kilometers, with a real navigation app such as Google Maps and Yandex. Luckily, I have not faced with any issues during this process.

Algorithm 2 A* Search Path Finding

function A_STAR_SEARCH($start\_city, end\_city$)
    $frontier \leftarrow$ empty Priority Queue
    $came\_from \leftarrow$ empty dictionary
    $cost\_so\_far \leftarrow$ empty dictionary
    add $(0, start\_city)$ tuple to the $frontier$
    $came\_from[start\_city] \leftarrow None$
    $cost\_so\_far[start\_city] \leftarrow 0$
    while $frontier$ **is not** empty **do**
        $current \leftarrow$ popped first element in $frontier$
        if $current$ **is the** end_city **then**
            $current \leftarrow end\_city$
            while $current\_city$ **is not** None **do**
                add $current\_city$ to $path$
                $current\_city \leftarrow came\_from[current\_city]$
            end while
            $path \leftarrow$ reverse order the $path$ list
            return $path\, cost\_so\_far$
        end if
        for $next\_city$ **in the** current city's neighbors **do**
            $new\_cost \leftarrow cost\_so\_far[current] +$ cost of $next\_city$
            if $next\_city$ **not in** $cost\_so\_far$ **or** $new\_cost < cost\_so\_far[next\_city]$ **then**
                $cost\_so\_far \leftarrow new\_cost$
                $priority \leftarrow new\_cost + heuristic(next\_city, end\_city)$
                add $(priority, next\_city)$ tuple to the frontier
                $came\_from[next\_city] \leftarrow current$
            end if
        end for

    end while
    return $came\_from\, cost\_so\_far$
end function

References

[1] Redblobgames.com. (2020) Introduction to a*. [Online]. Available: https://www.redblobgames.com/pathfinding/a-star/introduction.html
[2] Wikipedia contributors. (2020) Shortest path problem - wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Shortest_path_problem
[3] B. Roy. (2019) A-star (a*) search algorithm. [Online]. Available: https://towardsdatascience.com/a-star-a-search-algorithm-eb495fb156bb
[4] Brainkart.com. (2019) A* search: Concept, algorithm, implementation, advantages, disadvantages. [Online]. Available: http://www.brainkart.com/article/A--Search--Concept,-Algorithm,-Implementation,-Advantages,-Disadvantages_8883/
[5] Wikipedia contributors. (2020) Taxicab geometry - wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Taxicab_geometry