# Algorithm Development Report

My [GitHub repository](https://github.com/Kate732/Pathfinding-Algorithms) with code.

## A description of the problem

Given any two points from a list of them, the algorithm should find a path of points that goes from point A to point B, if such a path exists.

We were given sample data consisting of a set of 2D points that represent intersections on a map. These points are listed in pairs along with the distance between them. Each pair of connected points has a distance such that: 0 < distance < 1000.

## Storing data

There are multiple ways to store given data. Usually, tasks have two ways of solving: effective and right one, or random and straightforward. Here are the examples:

* **Dictionary**. Each letter is a key that holds value of array of tuples with the second point and its distance. Value can also be a dictionary, where each point (connected to the main key) is a key and value would be distance between them. It becomes even better when we use unordered dictionary in C++ (which takes more memory but is faster due to hashing the keys; since we don’t use a lot of memory I will stick to this choice). To access a distance between two points with this dictionary inside of the dictionary would take O(1) + O(1) -> so only two operations and constant complexity.
* **Map**. This is not another name for dictionary, but a “table” (2-dimensional array) that has two types of indexes (letters) that symbolize rows and columns, and values of the cross of two letters is the distance between them (row goes first and then the column, to save the order, if needed)
* **Array of tuples**. This is just an array that contains all the connections as tuples with points in order and third parameter (distance). It is not efficient, because searching through it would take O(n) and the length of this array will be bigger (includes all points in pair + number). So, it also takes the most storage.

## A detailed description of the Dijkstra's

First algorithm that I am going to talk about is Dijkstra's. This is one of the most efficient algorithms for searching the shortest path from one point to all the others.

It works step by step, analyzing each next possible point to find if that path is more efficient (the distance is smaller) than the saved one, updating it. It moves in order too: saving the next points from smallest to largest and, when repeating process, moves to the smallest one. It stopes when all the points were searched or when the shortest path to a second point was found.

## Artefacts. Dijkstra's

This is an example of path finding with Dijkstra's (on step 3). Going from point A to point F, choosing the smallest path possible and saving the other option to check next round.

A diagram of a connected line with Silverstone Circuit in the background

Description automatically generated

Here is a pseudocode for this algorithm:

Make an priority queue and total distances containing starting point.

* + If it reaches the destination point :
    - Save it to a path list.

We are finished.

* Else :

Continue searching.

Take the current point and check its neighbors.

* For each neighbor of the current point :
  + If the neighbor has a lower cost than the current point:
    - Replace neighbor with this new point as the neighbor’s parent
    - Push it to the priority queue

## Performance details of the algorithm. Dijkstra's

It is a good option for the task, because this algorithm can work with undirected data, our structure doesn’t require a lot of storage space (so it will be more efficient), and we don’t have the negative distance, so it means pros and cons of this algorithm fit well in this case.

Big O notation can be different, depending on the presence of extra help of the order (priority). So:

* **Without priority**. It is usually O(V^2).
* **With priority**. It is O((P + C) log P)

Where P - is the number of points and C – total number of connections (edges) in the given data.

Space complexity is O(P), plus additional space for priority list and other variables to store something temporarily.

## Potential applications of this algorithm. Dijkstra's

As I said before, it is a very popular and useful algorithm, so here are some examples of its applications:

* **Geographical purposes**. Dijkstra’s Algorithm plays a very important role in geographical applications, especially for finding optimal routes. It usually has locations of objects as nodes, connections (distance, time, etc.) as edges and cost or complexity of path as a weight. It takes a lot of space (all locations should be connected and available), it needs to have an access to geo data (include live events/schedules on the path), but efficient in time.
* **AI**. This is a very powerful pair of tools. For example: in computer games, Dijkstra’s algorithm can ensure efficient movement and pathfinding for NPC characters navigate by finding optimal paths. Data set for this will be map of possible moves, time of activity and conditions for this specific character. It can also help train AI itself: AI systems often require efficient allocation of lots of resources and Dijkstra’s algorithm can help with that.

## A detailed description of the A\*

Second algorithm that I am going to talk about is A\*. This is also one of the most efficient algorithms for searching the shortest path from one point to the other. It becomes especially powerful, when you add a function, that will search faster, depending on the specific task you’ve got.

I would like to compare it to hashing. There are different types of hashing, but the main idea is that you create a unique hashed string for each element (and sometimes, you can create an algorithm that will return you the original value) so that you will have a key to search for and a new level of security. So with A\* you have similar sort of function which results will tell you if this path is the best (in this case, the shortest one).

The best thing about A\* is, of course, heuristic function. I would like to give this example: maze solver. I love mazes and when I try to solve them, I always think of “lighting” all paths to see which one will reach the end. But tis is Dijkstra’s method. Imagine, if you take the coordinates of start point and finish destination… and prioritize “lighting up” only those paths that are closer to the coordinates of the maze exit. In lots of cases, it would solve it much quicker than previous algorithm. Or you can set the opposite and say: I want it to find paths that are as far from the beginning as possible. Or (I’ve heard it in orienting learning video) prioritize the paths that take left turns.

The most common formulas like that are:

* **Manhattan distance**. The standard heuristic for a square grid (where you can move in 4 directions)
* **Diagonal distance**. If your map allows diagonal movement this would be the best choice. In best case it allows you to shorten your path twice, so it’s worth a try.
* **Euclidean distance**. This easily gets more complicated, but if your units can move at any angle – this choice is effective with large set of data and obstacles.

So, this is what it’s all about: adopting to the task and your set of data.

Essential difference from Dijkstra’s algorithm is that calculation of total length of the path. In more complicated tasks there are additional information that will allow you to create formula (like the one with coordinates) and add that result to the total length to “bias” pathfinder.

## Artefacts. A\*

This is an example of Manhattan distance that has four options for directions. And it finds the first suitable path for it (without any loops, going backwards, not even one square wrong).

Heuristics:

function heuristic(node) =

dx = abs(node.x - destination.x)

dy = abs(node.y - destination.y)

return D \* (dx + dy)

For the perfect paths, D usually sets to the lowest cost between squares that are neighbors.

A graph with green and yellow squares

Description automatically generated

Diagonal distance gets a bit more interesting, allowing to move in 8 directions (4 of 90 degrees angle and four 45 degrees angles which means all the edges and vertexes). This is updated:

return D \* (dx + dy) + (D2 - 2 \* D) \* min(dx, dy)

So, the math here gets trickier) It also takes some analytical thinking to actually come up with an efficient formula for even more complex problems.

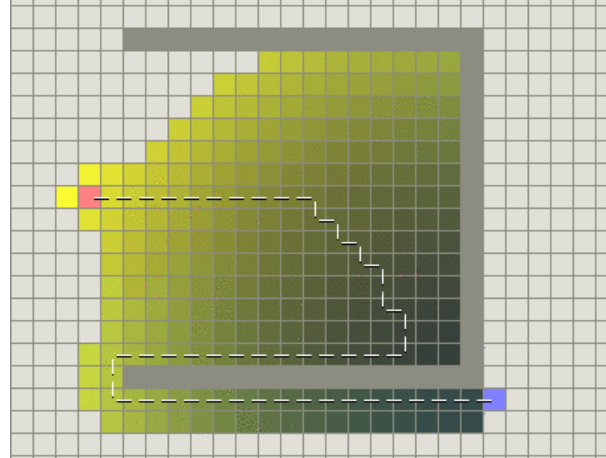
But, in my opinion, the most interesting is the Euclidean distance. You can move in any direction, and it allows you to adopt to much complicated problems and areas.

return D \* sqrt(dx \* dx + dy \* dy)

A graph with a line going up

Description automatically generated

Sometimes it gets funny to see algorithms that are trying to be as greedy as possible in cases where they just don’t fit. And after getting the result you immediately realize why:



Here is a simplified pseudocode for A\* algorithm:

Make an open list containing starting point.

* + If it reaches the destination point :
    - Save it to a closed list.

We are finished.

* Else :

Continue searching.

Put the current node in the list and check its neighbors.

* For each neighbor of the current node :
  + If the neighbor has a lower g value than the current node and is in the closed list:
    - Replace neighbor with this new node as the neighbor’s parent
    - Add it to the open list and set its g

## Performance details of the algorithm. A\*

In the worst case (especially if you have pointless heuristic), A\* goes through all available points, making it: O(2^N).

The efficiency depends a lot on:

* Heuristic function. If it won’t be informative then the whole benefit of A\* will disappear.
* Number of points and connections (it would also affect memory because of the data structure efficiency. If it is just a list of all repeating points, connections, and additional information, such as coordinates, it will slow down the whole algorithm)
* If priority queue is not sorted. Without reversing the queue, it will have the largest value on top, when the whole algorithm is about finding the smallest path. So, it would be much more efficient to change the direction and store the smallest value on top.
* Data structures that do not use efficient indexing can slow the algorithm down.

Memory depends on:

* Priority queue. The more points are in it, the slower it will run through queue.
* Number of points and connections

## Potential applications of this algorithm. A\*

* It calculates the shortest and fastest routes between two points, taking real-world factors like traffic and road conditions into consideration.
* As I mentioned before, it can be used in game development if you need a puzzle solver)
* It can also be a part of specific pathfinding in action games where decision needs to be made quickly.
* Logistics industries utilize A\* for vehicle routing and scheduling
* A\* can also be a part of AI systems, such as natural language processing and machine learning, to optimize decision-making processes

## Additional one

Third, kind of additional choice could be random pathfinder. The main idea of it is that you can always compare efficiency of your algorithms by comparing them with the worst case possible. This is it.

The implementation of it could be (with the same data storage) taking the starting point and take the first connection it has -> then dive in more, till it reaches either the end of connections (which means there is no such path) or the first random path. It could go through loops of all connections, until it reaches the destination, but sometimes it could get lucky (because I implemented unordered map, which means that order inside of it could change). And that is the best thing to compare “smart” algorithms to.

## Conclusions

Obviously, A\* is a better algorithm in terms of its capabilities. But for this task I would stick to simple Dijkstra. It is easy to get, straight forward and without unnecessary complications. A\* was trickier (so interesting) to do, but I wish we could open its full potential with task complexity (maybe a different data set, or even a simple maze implementation). I had a [similar task](https://github.com/Kate732/Doodle-Maps) before (when we were starting to study algorithms and data structures). We were given a maze printed out to console and we needed to find a best way out. This would be the perfect place to implement A\* (I wish, I knew it before)).

## Reference list

GeeksForGeeks (2024) What is Dijkstra’s Algorithm? Available at: <https://www.geeksforgeeks.org/introduction-to-dijkstras-shortest-path-algorithm/>

Adeel J. (2014) Understanding Dijkstra's Algorithm. Available at: <https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2340905>

Lester P. (2005) A\* Pathfinding for Beginners. Available at: <https://csis.pace.edu/~benjamin/teaching/cs627/webfiles/Astar.pdf>

Red blob games (2016) Heuristics. Available at: <https://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#S7>