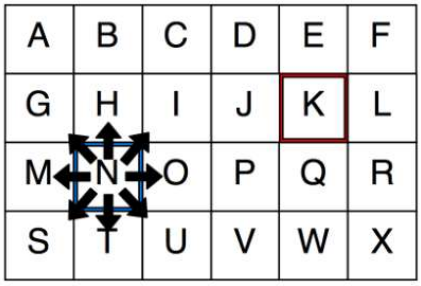
A* Search Algorithm

Friday, January 27, 2023 11:59 PM

The Dijkstra Algorithm is a good way to find the shortest path but its drawback is that it needs to expand from the starting node in all directions, regardless of where the goal node is located.
dijstrak_algorithm.mp4
In the previous video we could see how expensive is the Dijkstra Algorithm, because it needs to calculate all the possible paths and then find the shortest one. Here is where A* is introduced, it can optimize the process of searching.

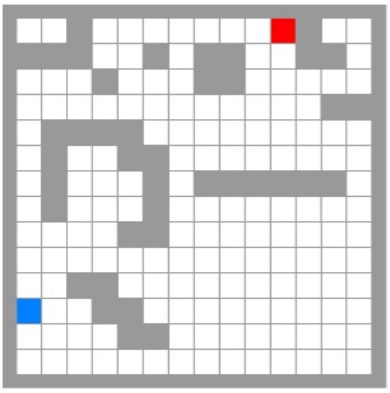
Informed Search Algorithms

Informed Search Algorithms utilize the information about the goal location to guide the search towards the target. We know if we want to drive to a location that is east of us, we would search a road that also goes to the east because road going to the west will most likely take us further away from our target.



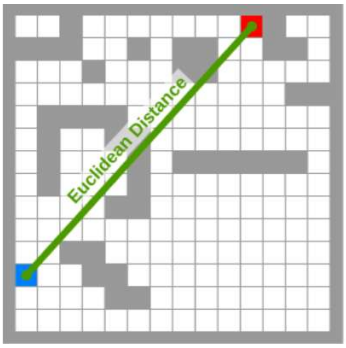
In this image we could say that node I and O located at the east are the best because they are more near to the goal. But M is the worst, these processes are named Informed Search Algorithms.

Heuristics
We can avoid unnecessary exploration, deciding which node to explore next. To provide an accurate answer to this question, we need to calculate all the possible paths and theirs travel distance, then we choose the one with the shortest path. But we were doing that before and it is not efficient. You can't say what is the shortest path in the following image. Because we don't know what is the shortest path.



A heuristic method help us to solve the problem but with approximations, but its very quickly and help us to solve the main problem.

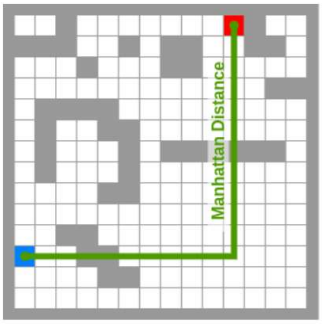
Euclidean Distance
The most common heuristic method for approximant the travel distance is the Euclidean Distance.



In the image below, we can see that the Euclidean Distance crosses the obstacles. All we need to guide

our search is a value that indicate us where to explore next.
$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$

Manhattan distance
It's the distance between two points measured along the axes at right angles. It only allows left/right and up/down movements.



The Manhattan distance is defined as:
$d = |x_1 - x_2| + |y_1 - y_2|$

Notes
Both are two of the most widely used heuristics used for approximating travel distances between nodes. There are however , many other well known heuristics methods. You could use a spherical method to estimate between two distant points on Earth's surface. An admissible heuristic is always lower or equal to the real distance.

Greedy Best-First Search
This algorithm shares most of its code with Dijkstra's shortest path algorithm, except that it expands its way by selecting the node closest to the goal. Due to this change it runs faster than Dijkstra, it will tend to focus on nodes closest to the goal.

greedy_best_first_search.gif

This algorithm is faster than Dijkstra's but it's path is not the most optimal. To use this algorithm we need to replace the cost of Dijkstra by the heuristic cost the measures the distance from the goal towards the current node's neighbors.

gbfs_expected_result.gif

Greedy BFS Behavior

Sometimes the path found by this algorithm is not the best one, the problem with this algorithm is that it always expands to the nodes closest to the goal. This behavior can mislead the algorithm to expand that nodes that look promising.

gfbs_concave.gif

We can see in the above image that this algorithm doesn't found the most optimal path, instead it could find the worst path when the obstacles are concave or large.
As we can see, **every search method has its advantages and disadvantages**. Keeping track of the distance to the starting point, like Dijkstra does, is slower, but it produces the shortest path; using an heuristic function like Greedy BFS is faster, but we can produce long paths that are not optimal. Wouldn't it be nice to combine the best of both methods? How lucky that this is exactly what A* does!

A*'s Special Secret
This algorithm combine the distance from the start node to the current node, similar to Dijkstra, and the estimated distance from the current node to the goal node like GBFS does. These two cost are combined to create a new cost known as the total cost of the node, which will denote as f_{cost}
$f_{cost} = g_{cost} + h_{cost}$
Where:
g_{cost}: Dijkstra's cost that estimates the distance from the start node to the current node
h_{cost}: Represents the euristhic distance from a node to the goal location
Whenever A* expands its search to a new node, each candidate is evaluated according to its total cost
The node with the smallest f_{cost} is select as the next node to be explored.

astar_empty.gif

Concave obstacle example:

astar_concave.gif

Result:

astar_expected_result.gif

A is a complete algorithm , which is good because it means that it will always find that solution if a solution exists. But it has a cost. To find a complete and optimal solution, a so called deterministic algorithm is required A* is such deterministic path planning algorithm. Which means that it always will produce the same path and follows the same computation steps. Unfortunately deterministic algorithms don't scale well with the map size.

3.11 A* search limitations

A* is a **complete algorithm**, which is good because it means that it will always find that solution if a solution exists. A* is also an **optimal algorithm**, good again, because this means that it will always find the shortest path if one exists. But these features come at a cost. To find a **complete and optimal** solution, a so-called deterministic algorithm is required. **A* is such a deterministic path planning algorithm**, which means that it always produces (on a given start, goal and map input) the exact same path, following the exact same computation steps.

Unfortunately, deterministic algorithms do not scale well with the map size. The more nodes to process, the more difficult it becomes to keep up with the planning time requirements. Also, large maps require a lot of memory since each node discovered has to be be accounted for.

Below you will find situations that would normally result in a large map:

- A large area to cover
- A high resolution map
- A high-dimensional space

And all of the reasons above combined.

Want a concrete example: path planning for robotic arms. Normally, we want very precise movements (high resolution!) and because the configuration space has many dimensions (typically higher than 4), we require very large maps.

So, is it possible to create an algorithm that is faster and more memory-efficient than A*?

Yes, but we will have to give up on optimality and completeness to win in computational efficiency. In some cases, it can be convenient, in other cases we have no choice but to make that sacrifice. The next algorithm we look at is called **RRT**, which belongs to a famlily of algorithms called **probabilistic algorithms**. Continue with the next unit to check it out!

3.12 Summary

All right, let's walk through the core concepts of this unit one more time before we move on!

- Uninformed search methods do not take into account how close a node is to the target in order to select where to expand next
- Dijkstra is one example of uninformed search: it picks the node closest to the start as next node, without considering how close it is to the goal
- Informed Search algorithms take into consideration the distance of a node to the goal and use this information when expanding their search
- Informed Search algorithms cannot exist without a function that evaluates the distance to the goal: the Heuristic function
- Greedy BFS uses such an heuristic function; it picks the node with the lowest heuristic value at each iteration, and processes that node
- In general, Greedy BFS is not optimal, that is, the path it finds is sometimes not the shortest one
- A* combines the search strategy of Greedy BFS with Dijkstra's: it always expands to nodes with lowest total cost first
- The total cost value of A* is defined as the sum of the real travel distance and the heuristic value
- A* is optimal, that is, it always finds the optimal path between the starting node and the goal node (given an admissible heuristic function)
- A* is complete, that is, it will always find a solution if one exists (in finite maps)