Path planning

- Dijkstra
- A*
- "car" planner

Dijkstra

front = start node

while front not empty:

- get node n with minimum cost
- mark n as visited

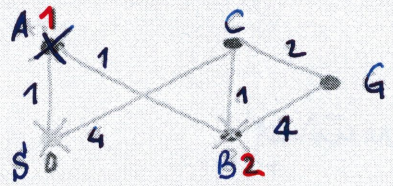
pick the node from front with the minimum cost, store it in visited and delete it from front

for any direct neighbor m of n, not visited:

- add m to front

-or-

- adjust cost if m is already in front and its new cost is lower



front

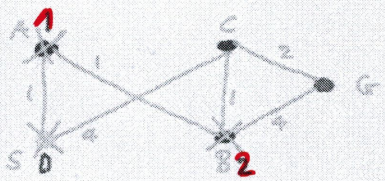
0,S | ~~1,A~~ | 4,C | ~~2,B~~ | 6,G

from B I can go to G with a cost of 6

from B I can go to C with a cost of 3.
G is already in front (not in visited) but with a cost of 4.

4 > 3

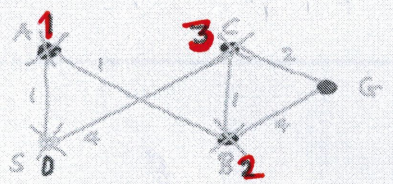
So, update the cost of G in front to 3



front

~~0,S~~ | ~~1,A~~ | ~~4,C~~ → 3,C | ~~2,B~~ | 6,G

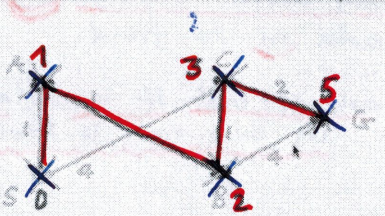
Pick the node in front with the minimum cost → It's C with a cost of 3.



front

~~0,S~~ | ~~1,A~~ | ~~4,C~~ 3,C | ~~2,B~~ | ~~6,G~~ 5,G

From C I can go to G with a cost of 5.
G is already in front (not visited) with a cost of 6. 6 < 5. Update the cost of G in front to 5.



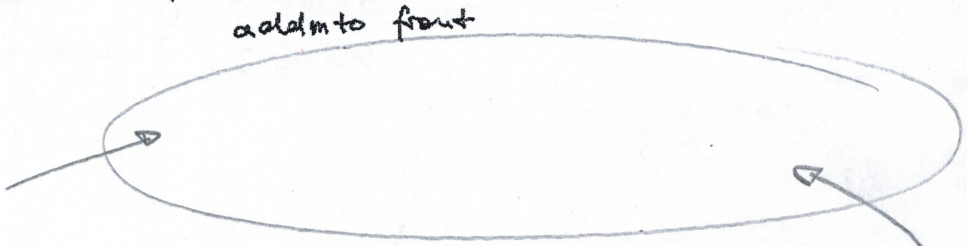
front

~~0,S~~ | ~~1,A~~ | ~~4,C~~ 3,C | ~~2,B~~ | ~~6,G~~ 5,G

A little modification with respect the Dijkstra algorithm shown in pp-

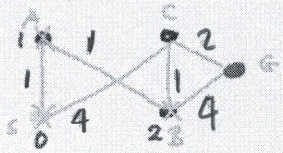
Dijkstra

```
front = start node
while front not empty:
    get node n with minimum cost
    skip if n has been visited already
    mark n as visited
    for any direct neighbour m of n, not visited:
        add m to front
```



We are going to get rid of the complicated part in here. So every time I reach a neighbour node, that is not in the visited group, I added it to the front, no matter if that node is already in front from a previous step. So there can be duplicated nodes in front with different costs. Then, when I scan later through front, looking for the node with the minimum cost I will pick the element with a lowest cost first. So, now I have to be aware that after I picked a node in front, there may be still other copies in front with a higher cost. Therefore, I have to make sure here that I handle each node only once. So, I have to skip the current iteration of the loop if the node n which I've just picked has already been visited.

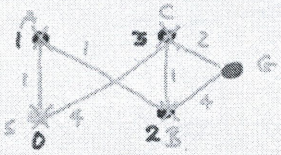
So this modification is a trade off between the lines of code I deleted and the lines of code I added.



front

0,S | ~~1,A~~ | 4,C | ~~2,B~~ | 6,G | 3,C

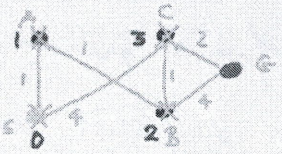
From B we can go to C with a cost of 3.
- C is not in visited.
- C is in front with a cost of 4.
- We add G again to front with a cost of 3. 2 Copies of C in front



front

~~0,S~~ | ~~1,A~~ | 4,C | ~~2,B~~ | 6,G | ~~3,C~~

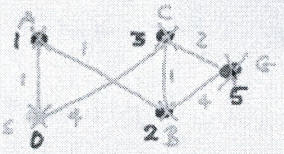
← We pick C with cost 3 from front
In front remains another C with cost of 4.



front

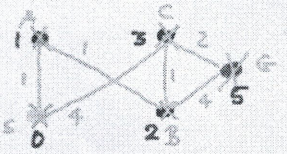
~~0,S~~ | ~~1,A~~ | ~~4,C~~ | ~~2,B~~ | 6,G | ~~3,C~~ | 5,G

We pick from front the element with minimum cost. In this case is the node C with cost 4.
But G has already been visited, So we skip the iteration of the loop and pick the next element from front with the minimum cost. In this case (5,G)



front

~~0,S~~ | ~~1,A~~ | ~~4,C~~ | ~~2,B~~ | ~~6,G~~ | ~~3,C~~ | ~~5,G~~



front

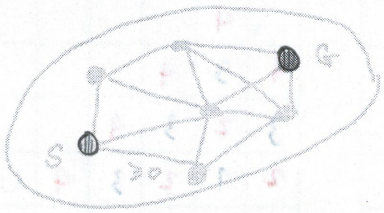
~~0,S~~ | ~~1,A~~ | ~~4,C~~ | ~~2,B~~ | ~~6,G~~ | ~~3,C~~ | ~~5,G~~

Finally we pick (6,G), but G has already been visited, So skip the loop iteration and we are done.

set of nodes and edges.

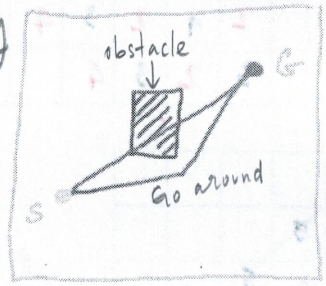
↳ Algorithm to find a path of minimum cost in a graph. Non negative costs allowed in the edges.

Dijkstra

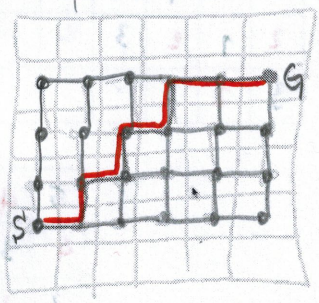


Route planning

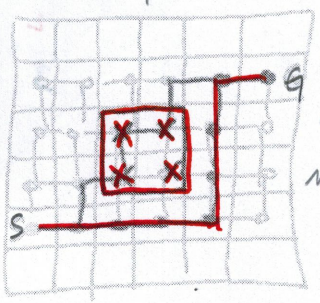
We are interesting

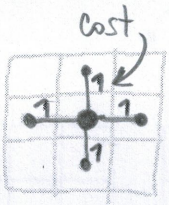


Space divided in cells that implicitly creates a graph, so Dijkstra can find a route between S and G.

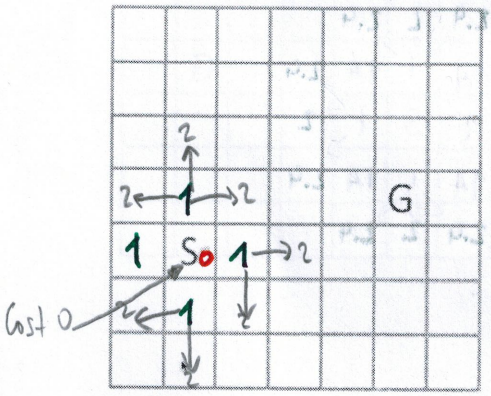


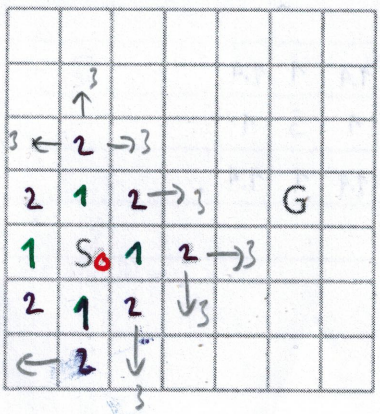
So, if there is an obstacle the algorithm can find another path (of minimum cost) between S and G

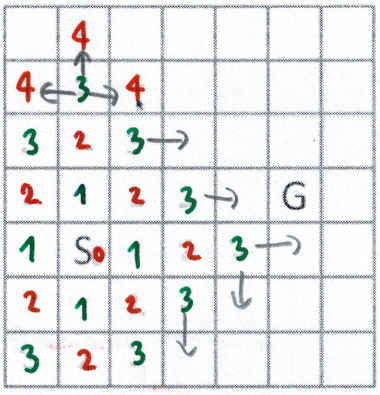


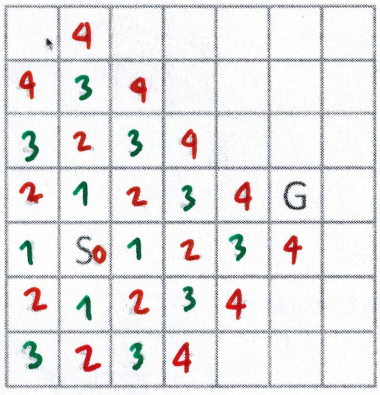


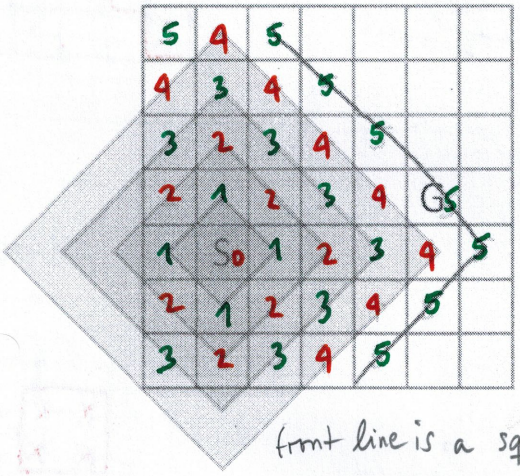
4N, Every node is connected by 4 edges to 4 neighbour nodes.







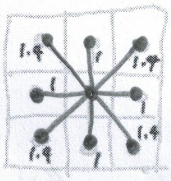


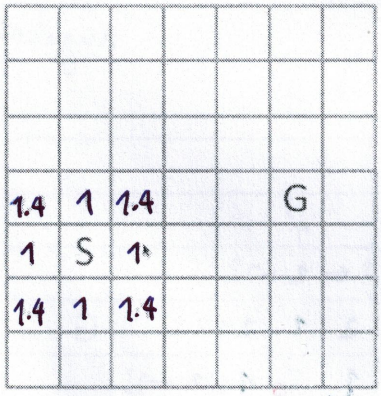


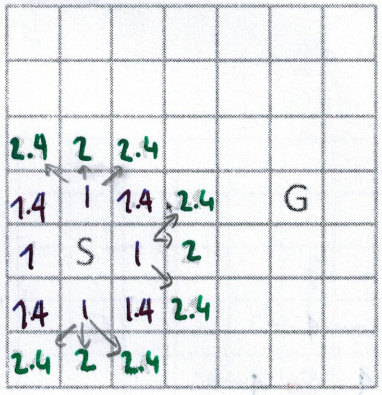
front line is a squared turned 45°

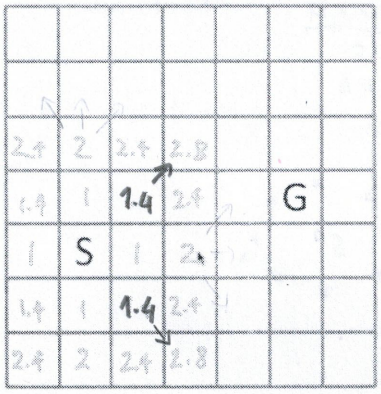
Each node is connected to 8 neighbour nodes.

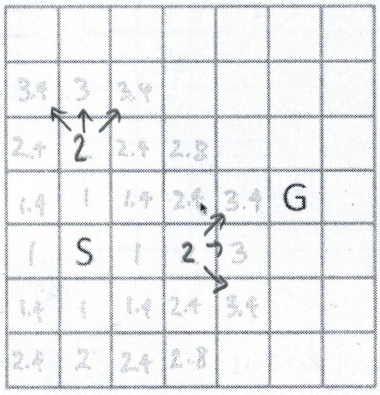
8N, 8 neighbours. $\sqrt{1^2+1^2}=\sqrt{2}=1.4142$

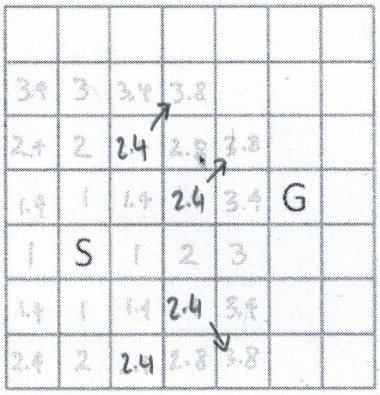


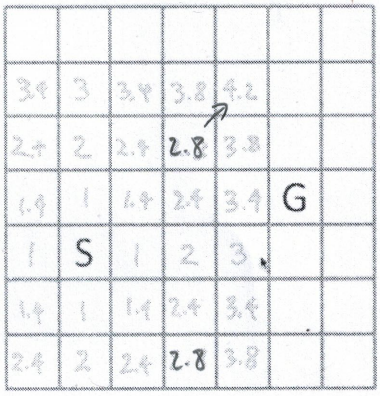


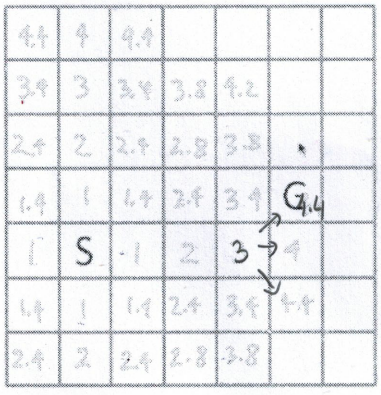


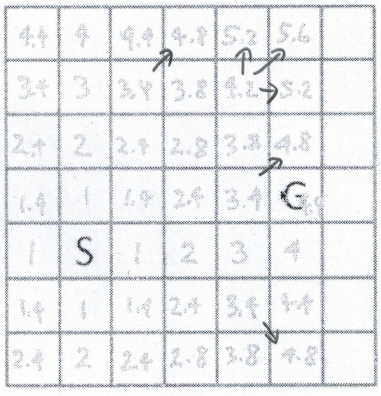

1.4
1.4
G
S

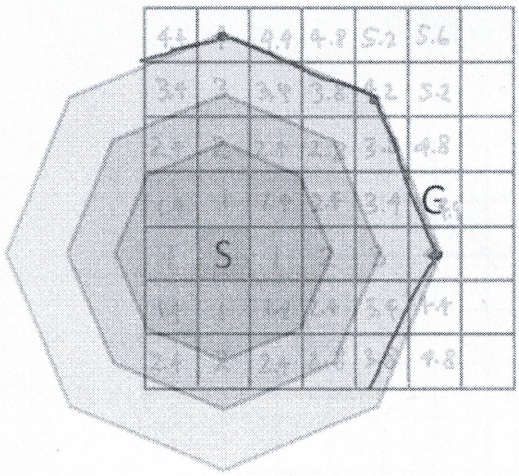

2
2
G
S


2.4
2.4
2.4
G
S


2.8
G
S

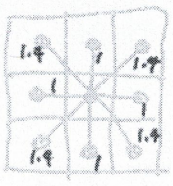

G
4.4
3
S


G
S



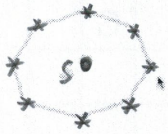
front line is an octagon

Type	Cost to reach G
4N	5
8N	4.4

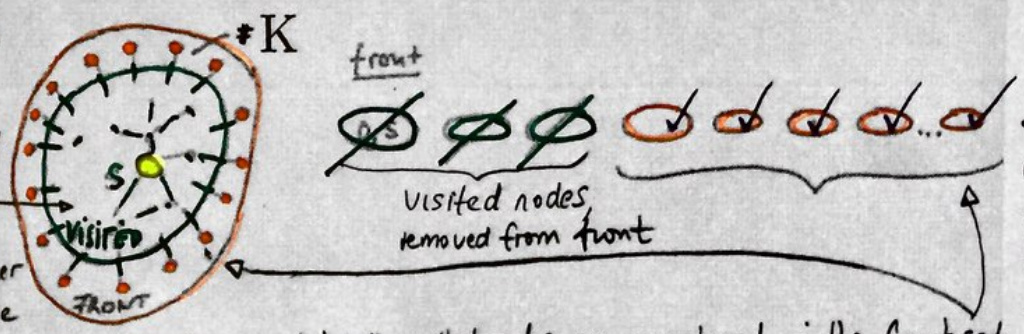


8N

Euclidean: $\sqrt{4^2+1^2} = \sqrt{17} \approx 4.12$ Real distance from S to G

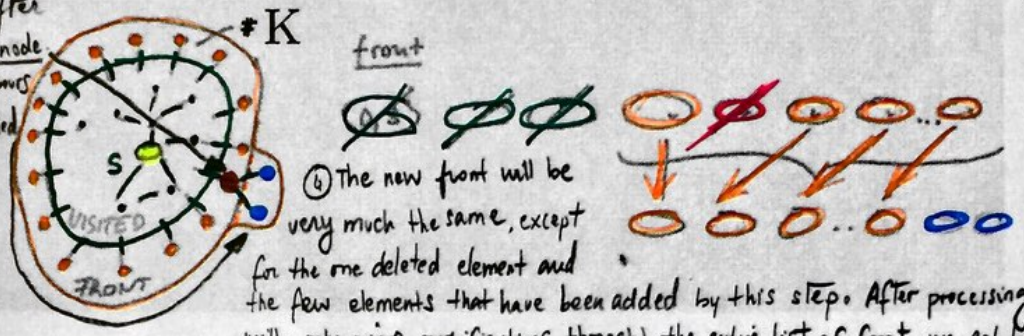


Until now the algorithm has been pretty slow.



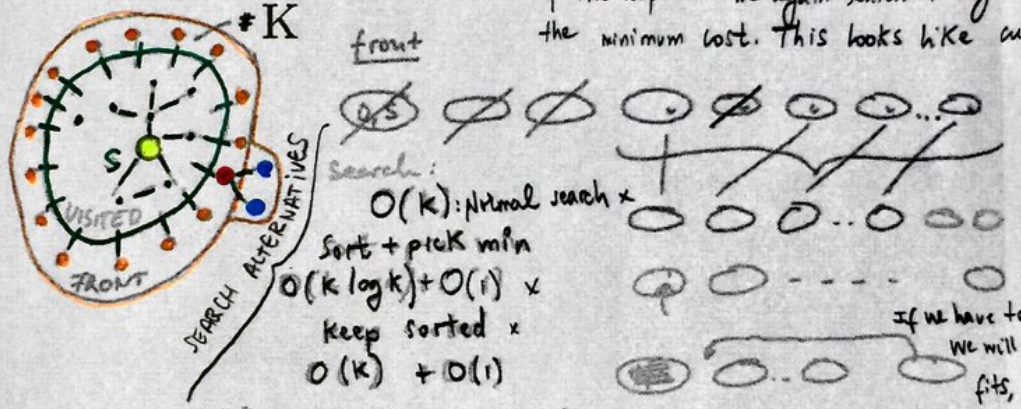
① At some stage of the algorithm we have a set of nodes already visited. These nodes are connected to other nodes by edges. These last nodes we have mentioned that are connected to the visited nodes are registered in the front set

② Find the element in front with the minimum cost. We have to search in the whole list because this list is unsorted. So the algorithmic cost is O(K) (time complexity)



③ O(K) is bad, because after we find the minimum cost node we have to look at its neighbours. So the front will be extended

④ The new front will be very much the same, except for the one deleted element and the few elements that have been added by this step. After processing one single node with only minor modifications through the entire list of front we get back to the beginning of the loop and we again search through all elements for the minimum cost. This looks like quadratic cost.



Our situation is ideal for the heap insertion algorithm, which uses a heap to sort elements.

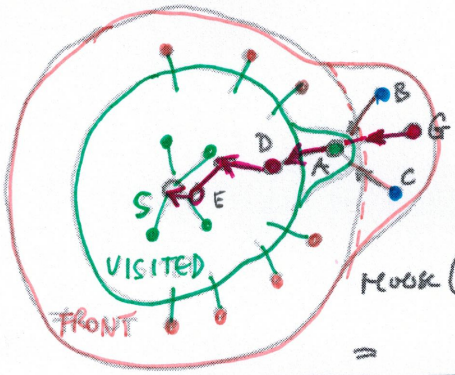
Python: heapq

```
front = start node

while front not empty:
  pop node n with lowest cost from heap (front)
  skip if n has been visited already
  record cost of n in 'visited'

  for any direct neighbor m of n, not visited:
    push m onto heap (front)
```

Path from start to goal



front

.... (G, A, D) ....

Add the previous node to each element in the list front

(G, B, A) (G, C, A)

Hook (G → A → D → ... → S)

= S → ... → D → A → G

VISITED [A] = cost
CAME-FROM [A] = D

CAME-FROM [GOAL] = A

new structure to remember what is the previous node for each node. Reversing this structure we get the optimal path.

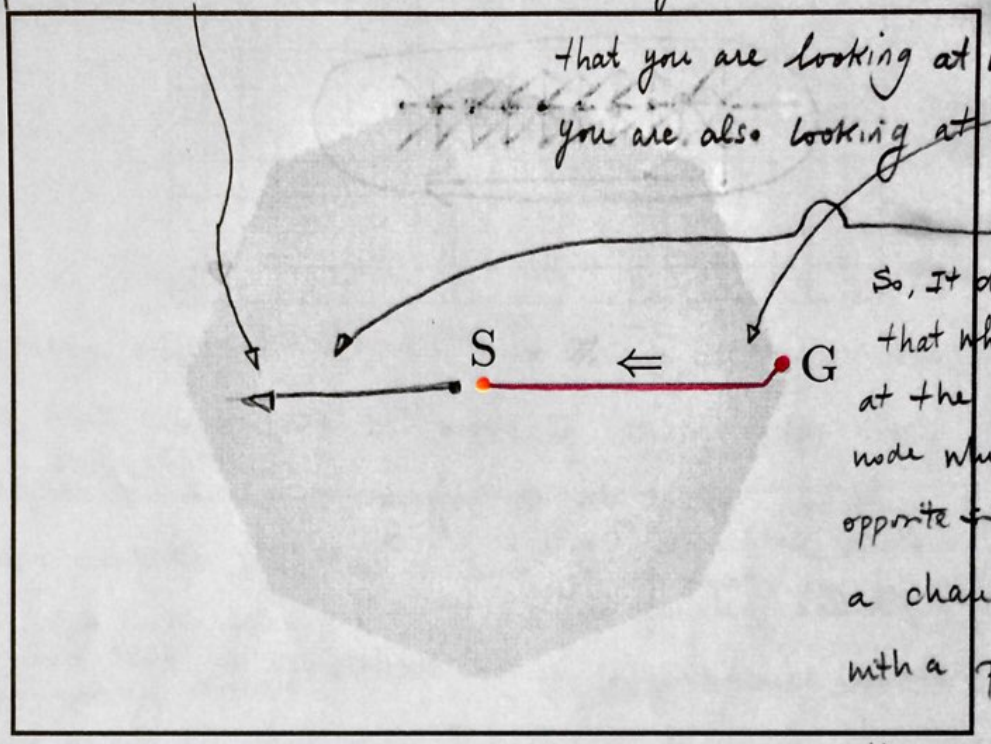
came-from (goal) = A

came-from (A) = D

came-from (D) = E

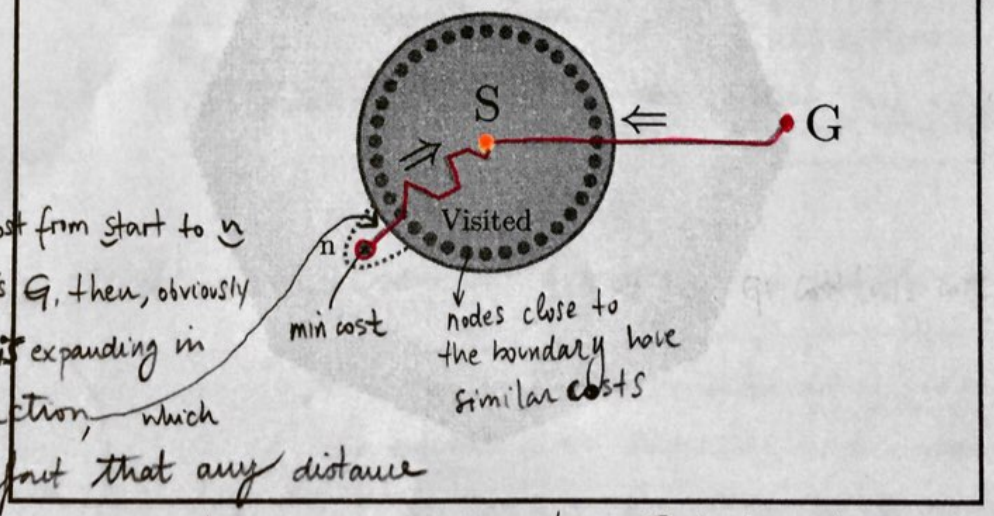
came-from (E) = start

The algorithm expands all the nodes, not only in the direction of the goal but also in the opposite direction. So, somehow this is strange because at the very moment that you are looking at nodes at here, you are also looking at nodes here.



So, It doesn't seem possible that when you are almost at the Goal, then another node which is in the opposite side will still have a chance to come up with a path that reaches the goal, when this node is so far away, in fact in our example, about two times the distance from S to G

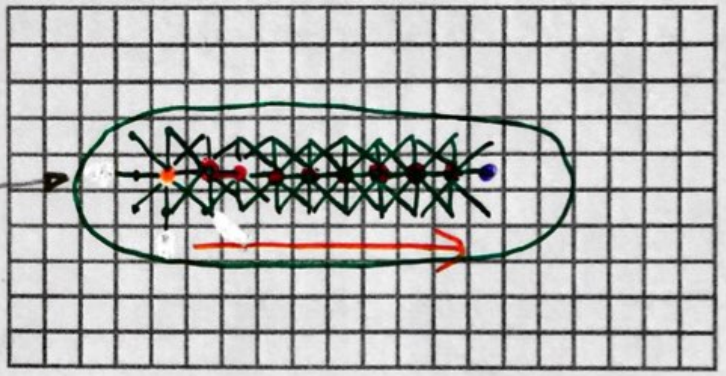
So, we are looking for an improvement in the algorithm (Search) so that nodes are expanded in the direction to the goal and not so much in the direction away from the goal.



$C_n \equiv \text{Cost}(n) \equiv$ Cost from start to n

If the goal is G, then, obviously the algorithm is expanding in the wrong direction, which is due to the fact that any distance to the goal is not part of our computation here, But it makes sense to use the cost we have been using so far as well.

So, let's explore a Greedy solution.



We look at the direct neighbours of S. I pick the one that gets me closer to G. And starting from that node I check its neighbours and again pick the one that gets me closer to G. Again I check its neighbours, pick the one that get me closer to G, and so on and so for.

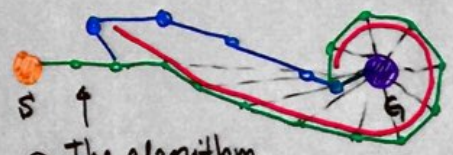
If you look at the boundary of visited nodes, you see that the algorithm expands far less nodes than the number of nods that we would have obtained if we had used the Dijkstra algorithm.

In the greedy case we used the following cost:

$$\text{cost} = \underbrace{\text{estimated distance to the goal}}_{\text{Direct line (Pythagoras)}}$$

We expand far less nodes, but is the result optimal yet?

Greedy



① The algorithm will proceed walking towards the Goal (Green line)

② Let's say we have an obstacle (in red)

③ The algorithm still tries to walk towards the Goal. Indeed the distance to the Goal is always decreasing, but the algorithm will walk a long way until it reaches the goal. (Green line)

④ The correct solution would have been to go back one step an then the algorithm could have walked directly to the Goal which had been considerably shorter. (Blue line)

DIJKSTRA	GREEDY
• Uses know distance from start to n. [Cost function called g]	• Uses estimated distance to the goal [Cost function called h]
• Expands many nodes	• Expand fewer nodes
• Finds the optimal solution	• Good results in general but it doesn't guarantee to find the optimal solution.

Idea: Let's define a new cost function, f.

$f = g + h$

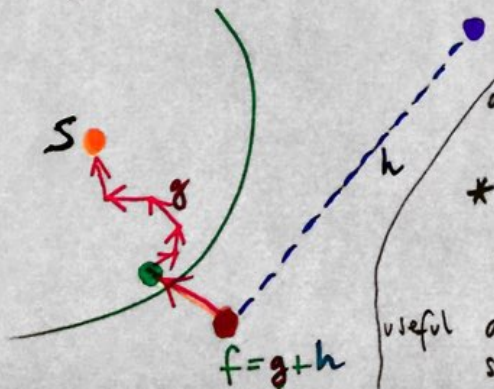
g: Dijkstra's Cost function

h: Greedy's Cost function

With this new cost function we get the properties

- The algorithm will expand less nodes than the Dijkstra algorithm.
- Finds the optimal solution!

The algorithm that uses this new cost function f is called A^*



The h function must be admissible to allow A* to give an optimal result. Admissible means that:

* actual cost from n to Goal $\geq h(n)$

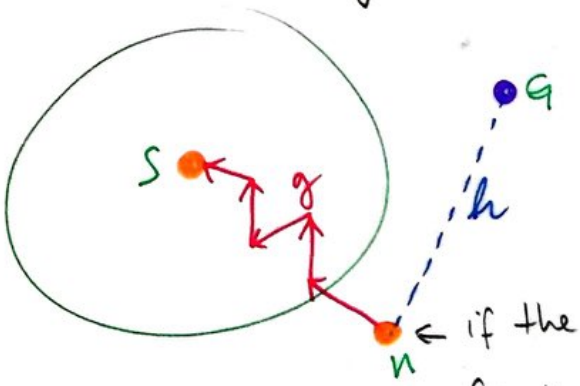
But, fortunately the straight line (direct line) is a useful admissible estimate because the direct line is always shorter or equal to the real distance between n and G.

$f = g + h$

g: know cost from S to n

h: estimated cost from n to G

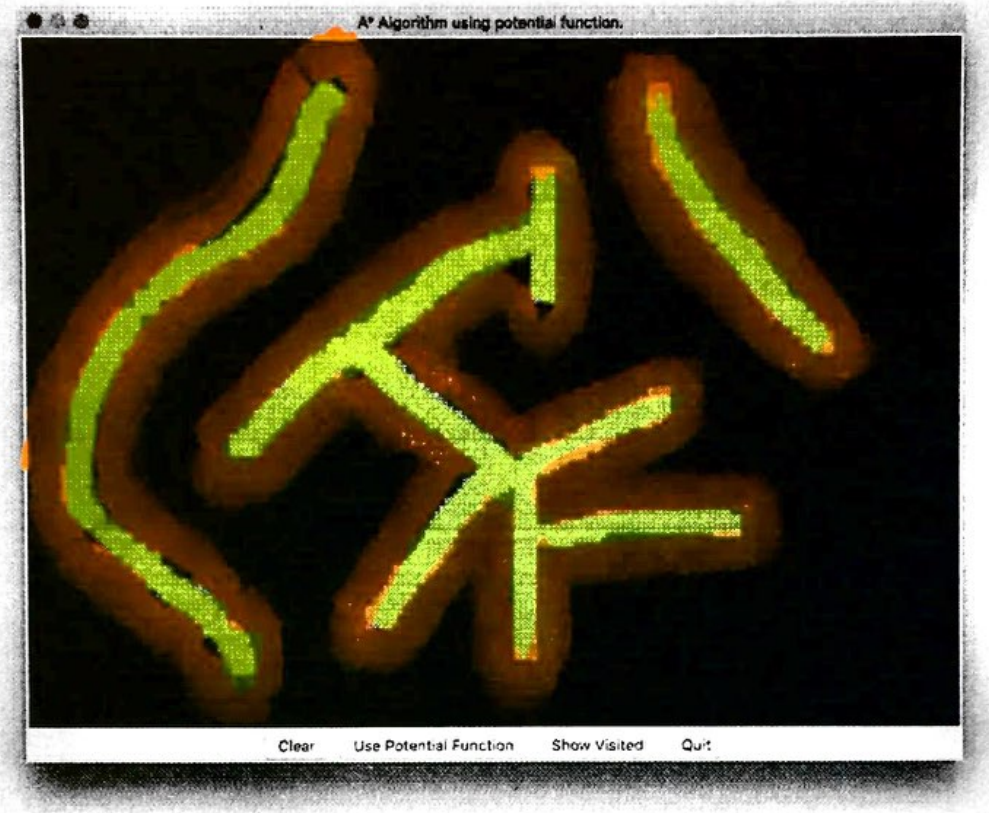
f: minimum cost of a path from S to G which goes through n ☺



← if the path goes through this node then I know for the red part of the path it has cost g and for the blue part it has at least cost of h, so the sum is the minimum of the cost I will have from S over n to G.

Potential function

→ obstacle
→ potential field around the obstacles.



A potential field around the obstacles is a surrounding area around the obstacles that has an associated cost higher than the cost in free space. So if we reach a node with cost C, the same node under the influence of a potential field would have a cost bit greater than C. The potential field technique is an useful way to create an optimal path that is not so close to the obstacles than the path that standard A* creates.

So, with the potential field technique we get an optimal path more likely to the path that we would follow when going from point S to point G using a vehicle.

The first step to get the new matrix that contains the obstacles, the free spaces and the potential field is to apply the distance transform (Euclidean type, Manhattan type or chessboard type) to the original matrix containing the world (Binary matrix. In our case 0 - Free space 255 - Obstacle)