Path planning

- Dijkstra
- A*
- "Car" planner

Dijkstra

front = start node

while front not empty:

  get node n with minimum cost

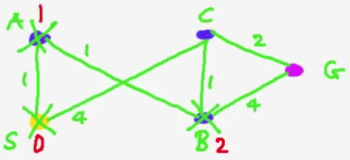
  mark n as visited

  for any direct neighbor m of n, not visited:

    add m to front

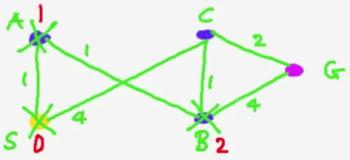
    -or-

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



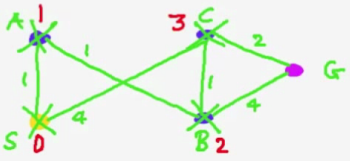
front

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



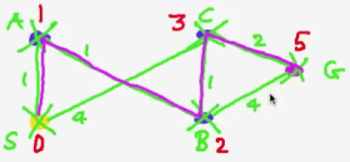
front

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



front

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

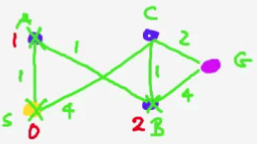


front

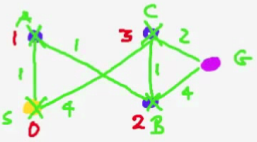
0, S | 1, A | 3, C | 2, B | 5, G

Dijkstra

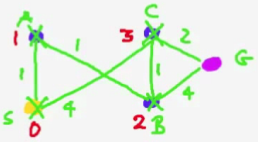
```
front = start node
while front not empty:
    get node n with minimum cost
    skip if n has been visited already
    mark n as visited
    for any direct neighbor m of n, not visited:
        add m to front
```



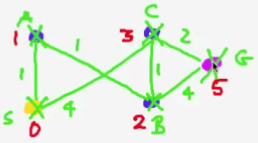
front



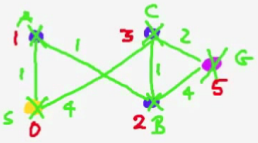
front



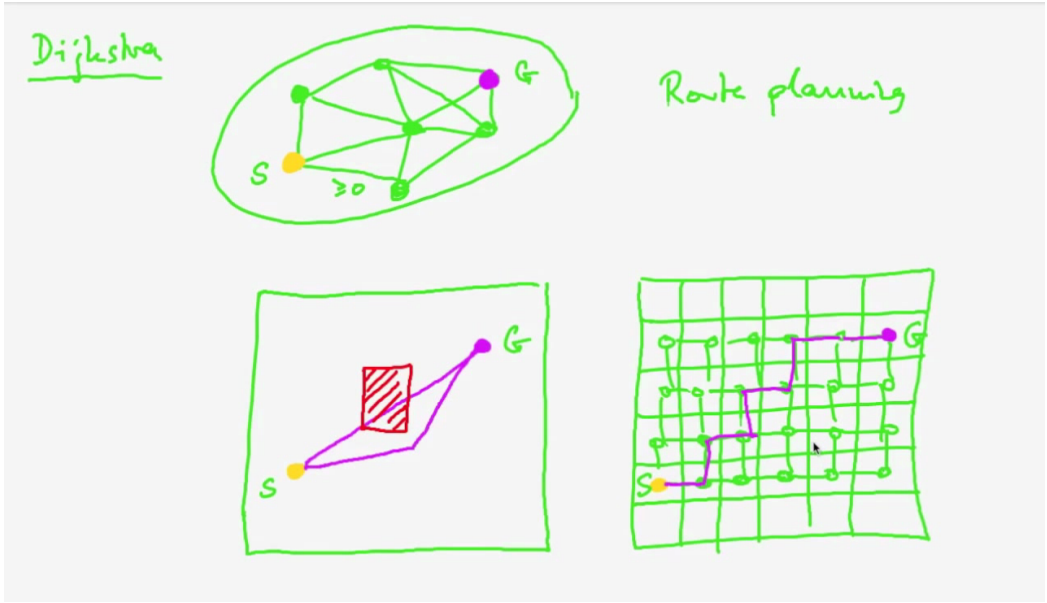
front

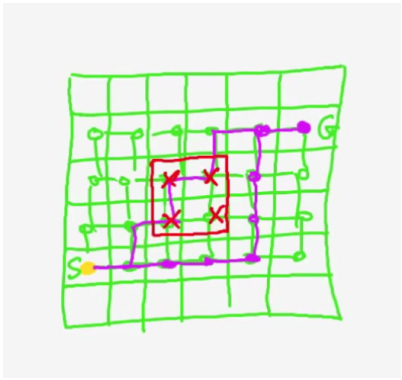


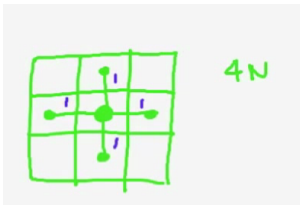
front

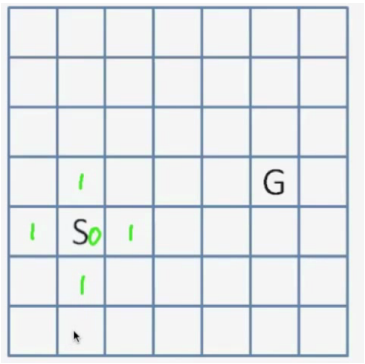


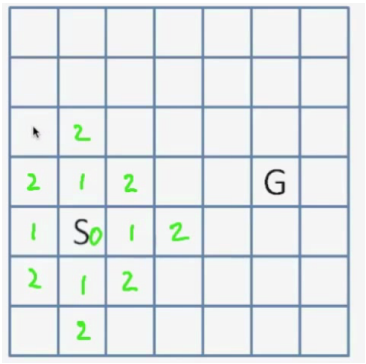
front


Dijkstra
G
S
≥0
Route planning
G
S
G
S


G
S

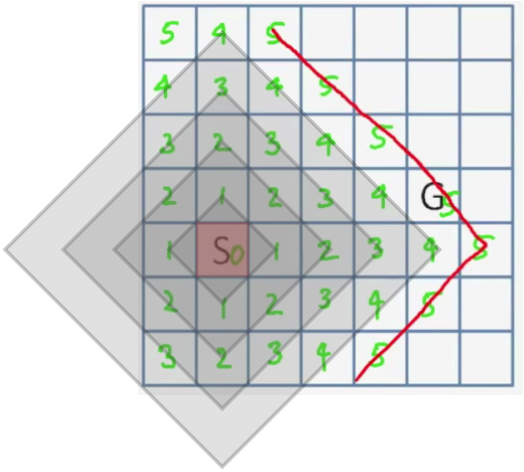

1
1
1
1
4N

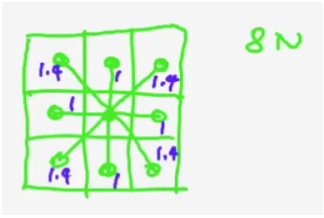

1
G
1
S0
1
1

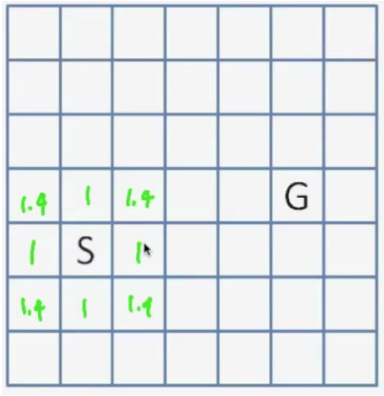

2
2
1
2
G
1
S0
1
2
2
1
2
2

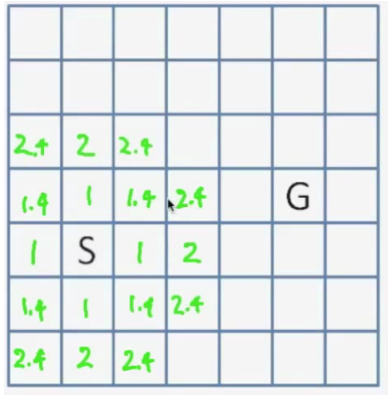

4
4 3 4
3 2 3
2 1 2 3 G
1 S0 1 2 3
2 1 2 3
3 2 3


4
4 3 4
3 2 3 4
2 1 2 3 4 G
1 S0 1 2 3 4
2 1 2 3 4
3 2 3 4


5 4 5
4 3 4 5
3 2 3 4 5
2 1 2 3 4 G
1 S0 1 2 3 4 5
2 1 2 3 4 5
3 2 3 4 5


1.4
1
1.4
1
1
1.4
1.4
1
8N


1.4 1 1.4 G
1 S 1
1.4 1 1.4


2.4 2 2.4
1.4 1 1.4 2.4 G
1 S 1 2
1.4 1 1.4 2.4
2.4 2 2.4


2.4 2 2.4 2.8
1.4 1 1.4 2.4 G
1 S 1 2
1.4 1 1.4 2.4
2.4 2 2.4 2.8

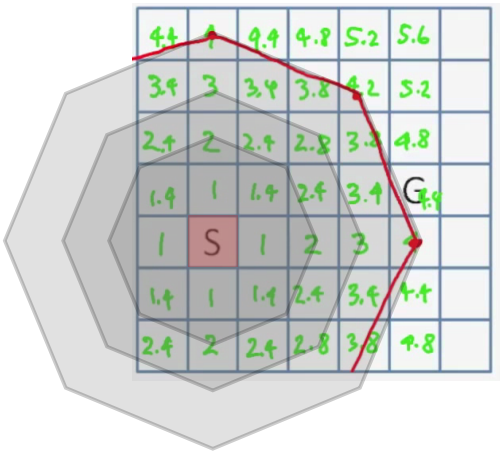

3.4 3 3.4
2.4 2 2.4 2.8
1.4 1 1.4 2.4 3.4 G
1 S 1 2 3
1.4 1 1.4 2.4 3.4
2.4 2 2.4 2.8

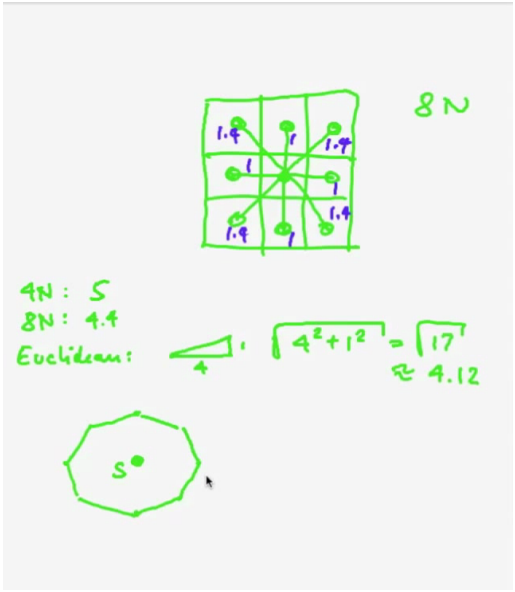

3.4 3 3.4 3.8
2.4 2 2.4 2.8 3.8
1.4 1 1.4 2.4 3.4 G
1 S 1 2 3
1.4 1 1.4 2.4 3.4
2.4 2 2.4 2.8 3.8

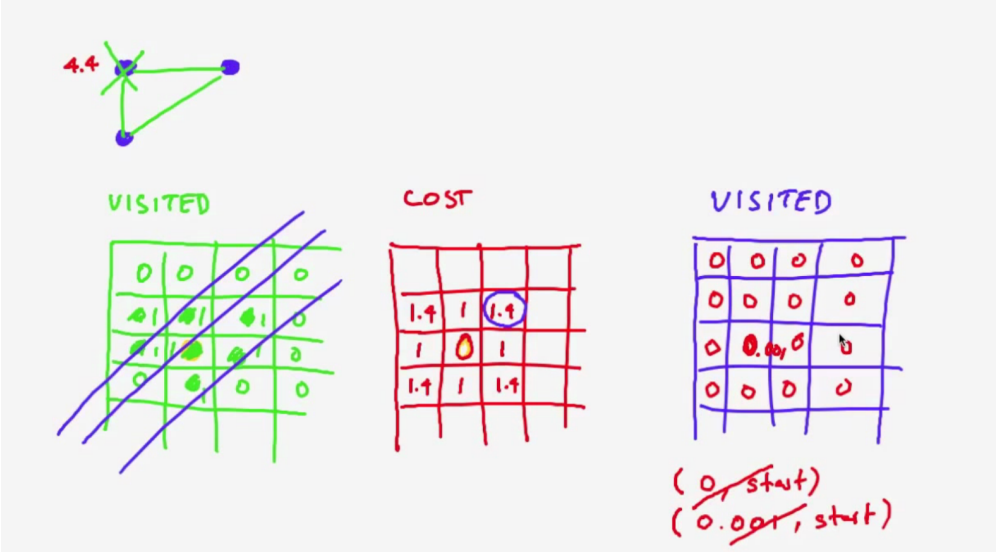

3.4 3 3.4 3.8 4.2
2.4 2 2.4 2.8 3.8
1.4 1 1.4 2.4 3.4 G
1 S 1 2 3
1.4 1 1.4 2.4 3.4
2.4 2 2.4 2.8 3.8


4.4 4 4.4
3.4 3 3.4 3.8 4.2
2.4 2 2.4 2.8 3.8
1.4 1 1.4 2.4 3.4 G
1 S 1 2 3 4
1.4 1 1.4 2.4 3.4 4.4
2.4 2 2.4 2.8 3.8

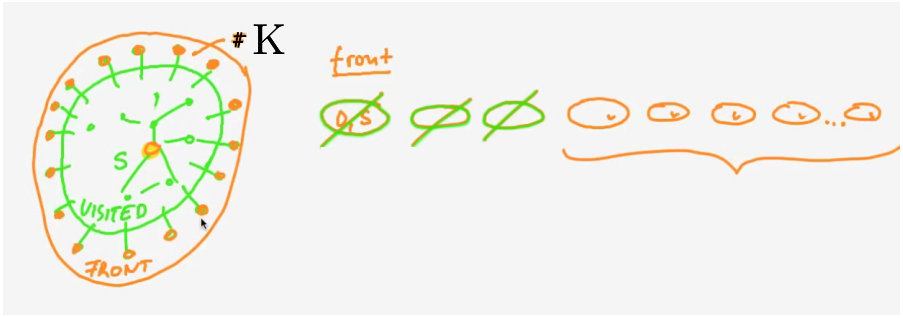

4.4 4 4.4 4.8 5.2 5.6
3.4 3 3.4 3.8 4.2 5.2
2.4 2 2.4 2.8 3.8 4.8
1.4 1 1.4 2.4 3.4 G
1 S 1 2 3 4
1.4 1 1.4 2.4 3.4 4.4
2.4 2 2.4 2.8 3.8 4.8

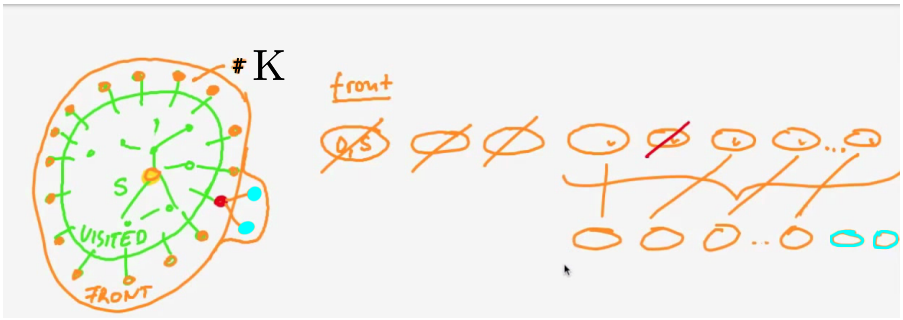

4.4 4 4.4 4.8 5.2 5.6
3.4 3 3.4 3.8 4.2 5.2
2.4 2 2.4 2.8 3.2 4.8
1.4 1 1.4 2.4 3.4 G
1 S 1 2 3 4
1.4 1 1.4 2.4 3.4 4.4
2.4 2 2.4 2.8 3.8 4.8

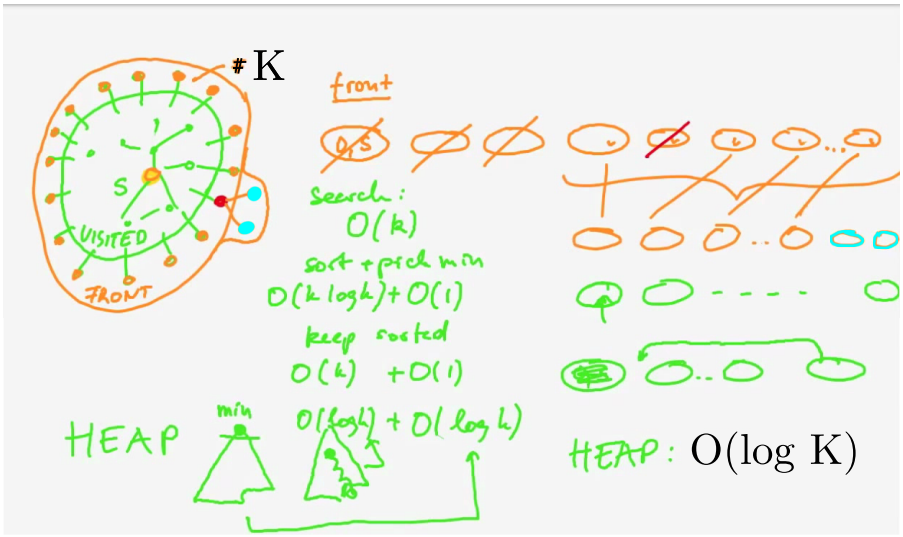

8N
1.4
1
1.4
1
1
1.4
1.4
1
4N: 5
8N: 4.4
Euclidean:
4
$\sqrt{4^2+1^2}=\sqrt{17}$
≈ 4.12
S



We want to paint the nodes for where the Dijkstra algorithm passes in different tonalities of the same color according to its cost. The first idea is to use two matrices, one for visited nodes and one for the cost of each node. But, we can achieve the same behavior using only one matrix, the visited node matrix. Instead of storing 1 or 0 in the visited node matrix, we store the cost of the node. So, a cost > 0 means that a node has already been visited. We have to do one trick, then. The starting node has to have another cost different from 0, because a cost of 0 means not visited yet. So, for the starting node we use the cost 0.001. This value is small enough to show that the starting point has been visited and also it will be the smallest cost in the graph.



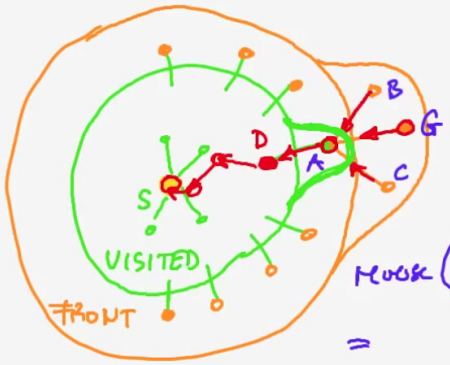




Python: heapq

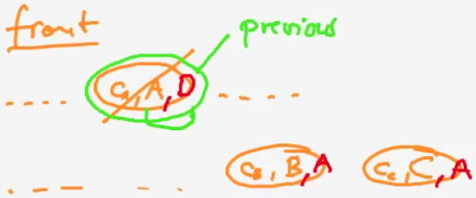
```
front = start node

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

previous

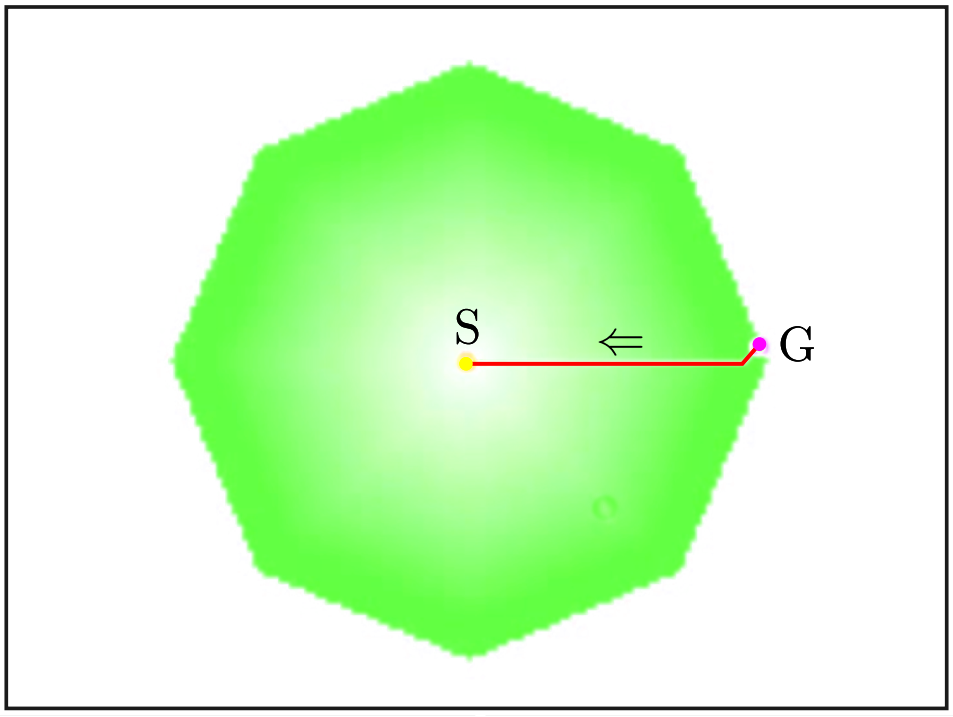


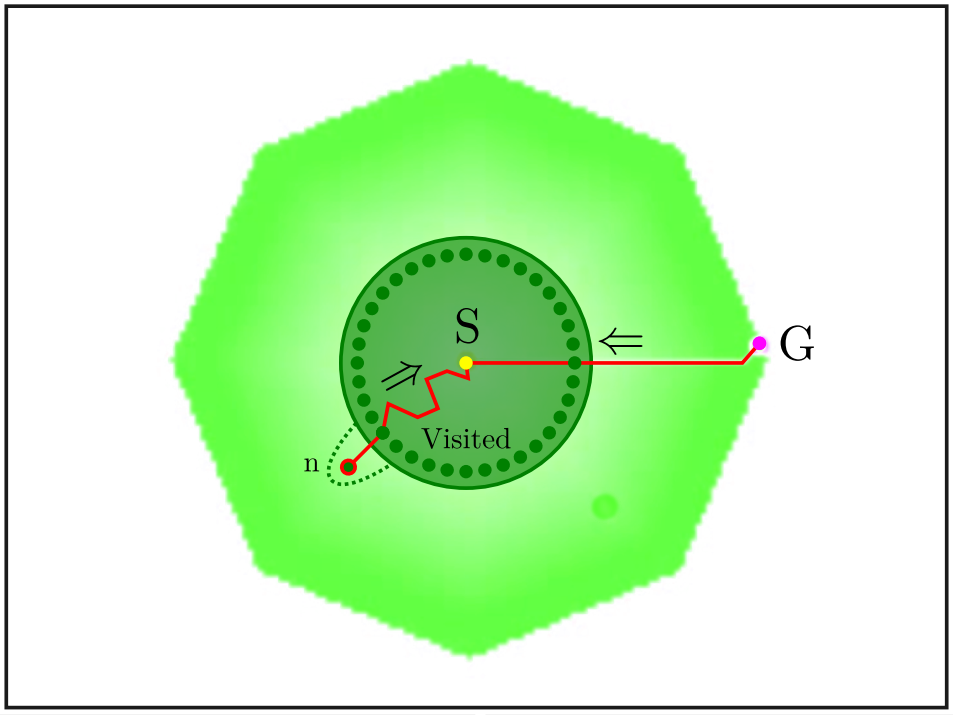
MOVE (G → A → D → ... → S)

= S → ... → D → A → G

VISITED [A] = cost

CAME-FROM [A] = D ←


S
G


S
G
Visited
n