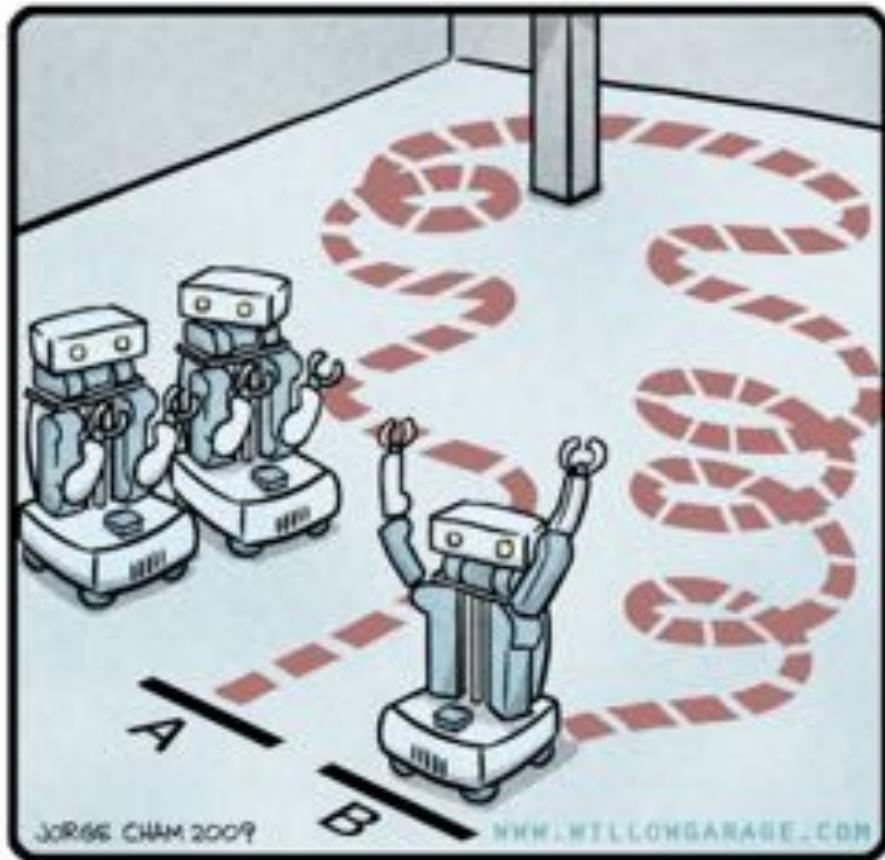


R.O.B.O.T. Comics



Path planning

the best way to get from A to B

EECS 367
Intro. to Autonomous Robotics

ME/EECS 567 ROB 510
Robot Modeling and Control

Fall 2019

autorob.org
Michigan Robotics 367/510/567 - autorob.org

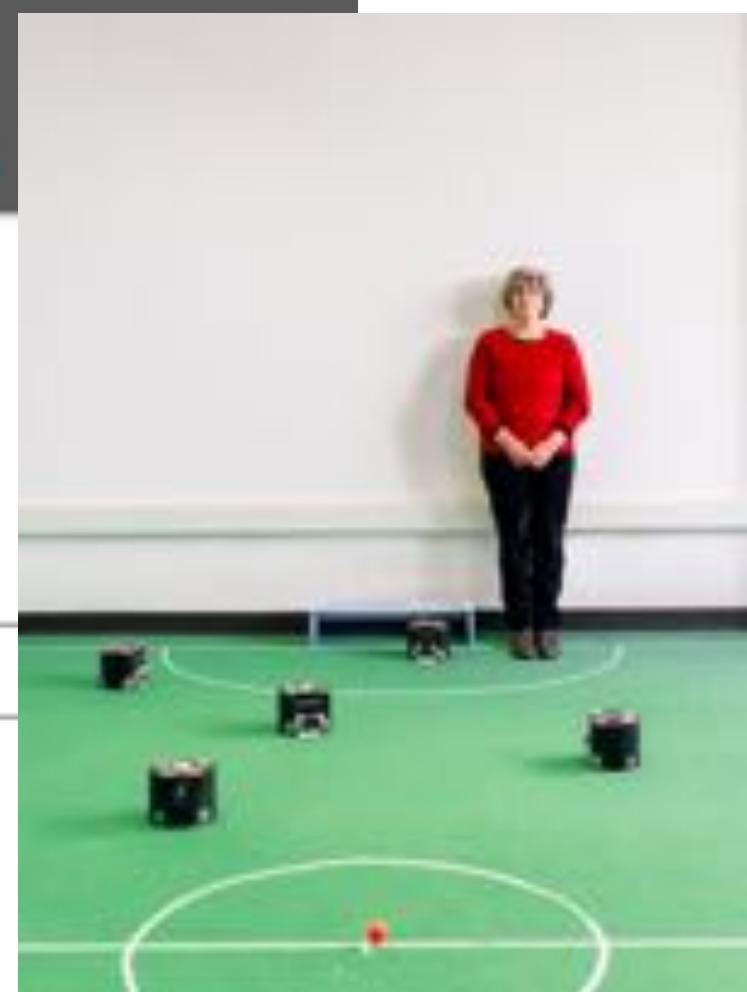
Feature | Geek Life | Profiles

Manuela Veloso: RoboCup's Champion

This roboticist has transformed robot soccer into a global phenomenon

Posted 28 Feb 2015 | 17:00 GMT

By **PRACHI PATEL**



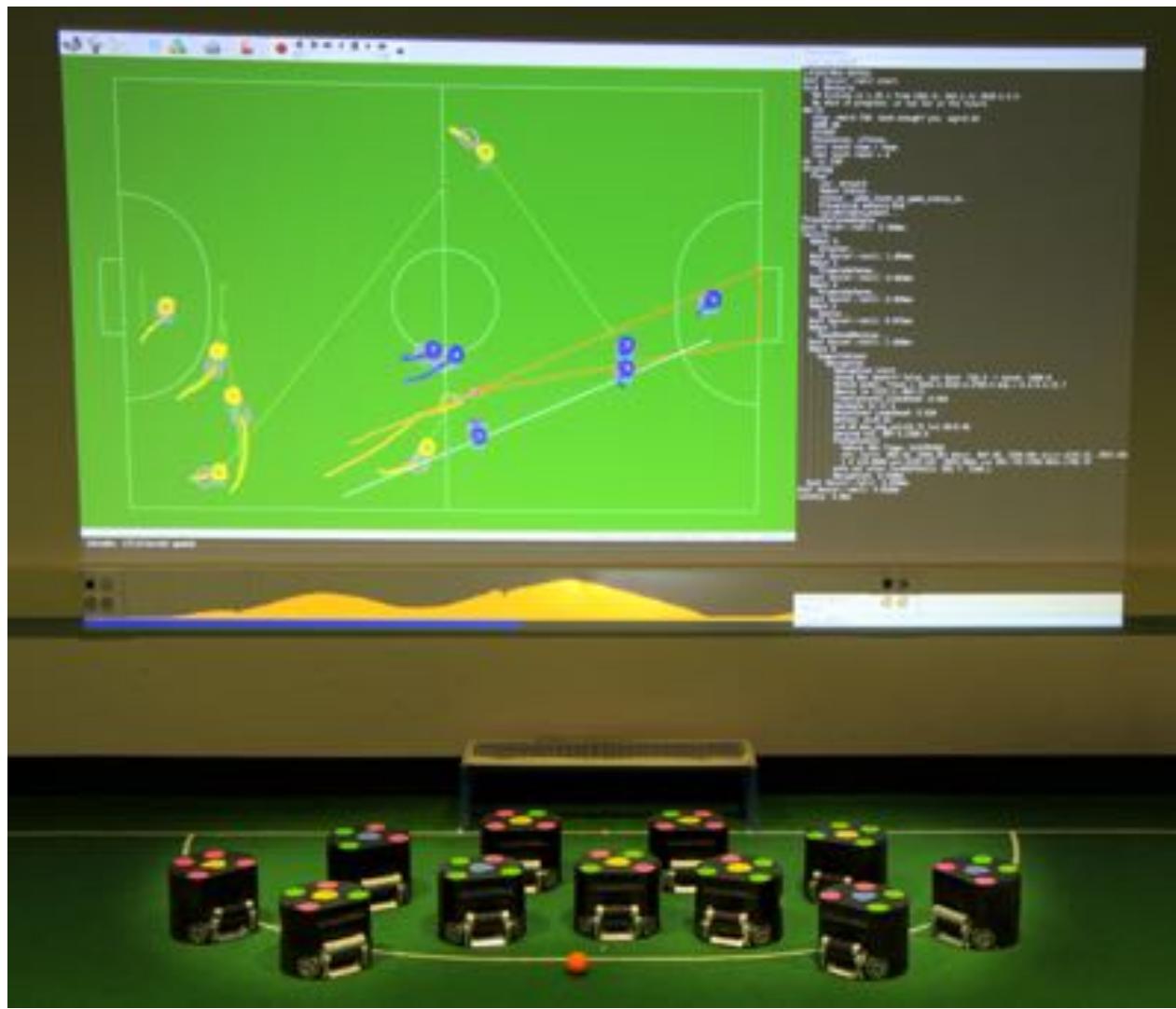
Michigan Robotics 367/510/567 - autorob.org



CM Dragons
RoboCup Small
2006

https://youtu.be/-Y4H3Sox_4I

367/510/567 - autorob.org



CMDragons - <http://www.cs.cmu.edu/~robosoccer/small/> Michigan Robotics 367/510/567 - autorob.org



CMDragons 2015 Pass-ahead Goal

Michigan Robotics 367/510/567 - autorob.org



CMDragons 2015 slow-motion multi-pass goal
Michigan Robotics 367/510/567 - autorob.org



x0.5 Speed

CMDragons 2015 slow-motion multi-pass goal
Michigan Robotics 367/510/567 - autorob.org



<http://www.cs.cmu.edu/~coral/projects/cobot/>

Michigan Robotics 367/510/567 - autorob.org



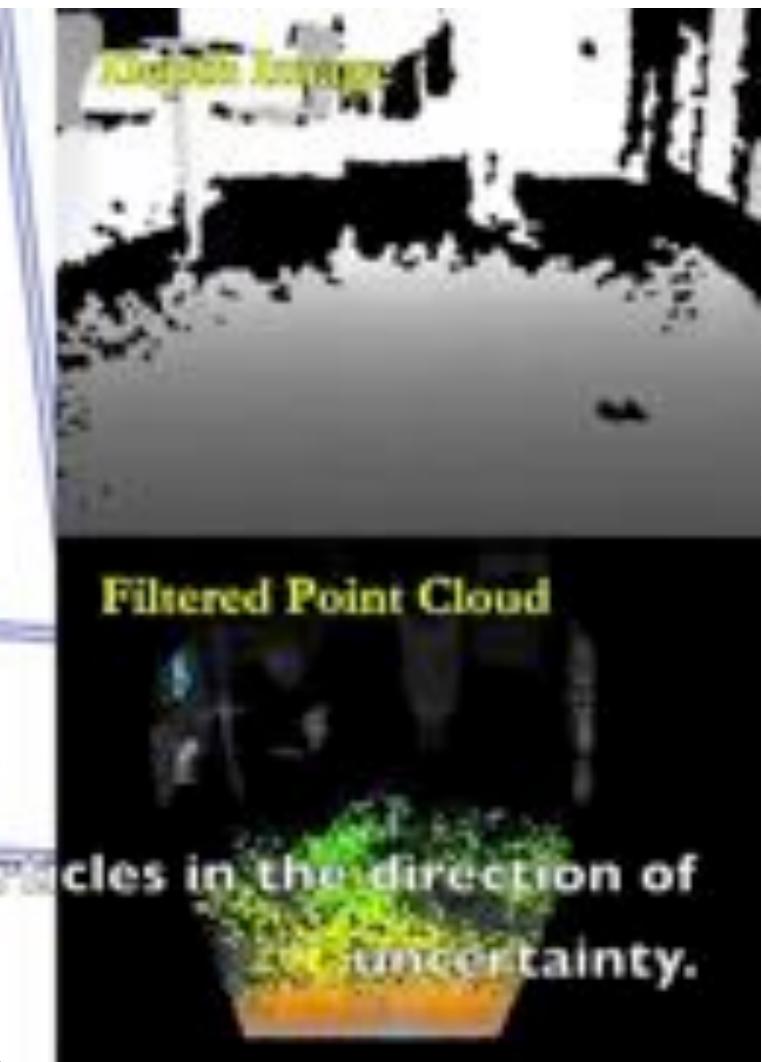
Cobots can navigate the building.

<http://www.cs.cmu.edu/~coral/projects/cobot/>

Michigan Robotics 367/510/567 - autorob.org



<http://www.cs.cmu.edu/~coral/projects/cobot/>



Michigan Robotics 367/510/567 - autorob.org

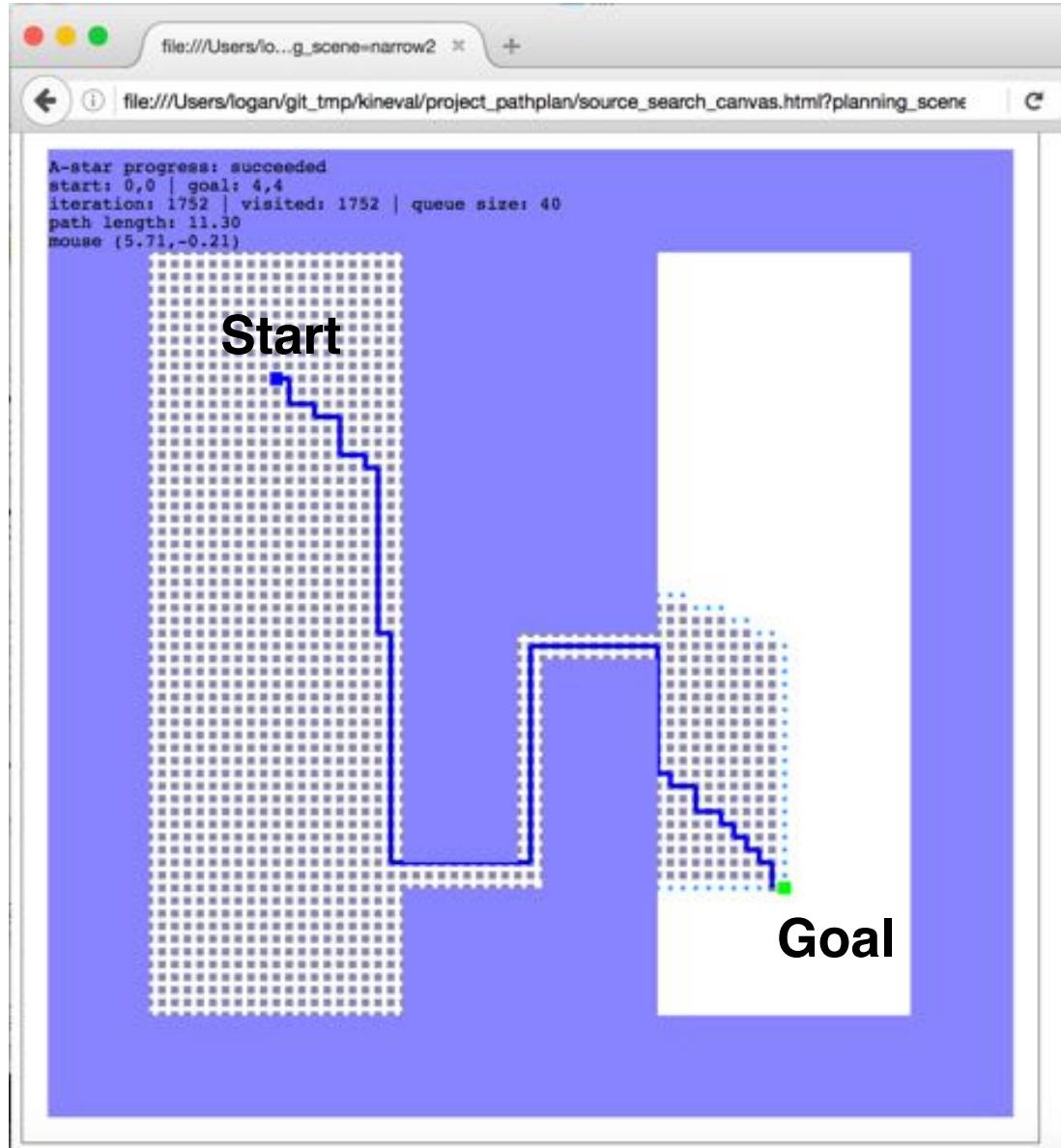


CoBot greets me in 2013

Michigan Robotics 367/510/567 - autorob.org

Project 1: 2D Path Planning

- A-star algorithm for search in a given 2D world
- Implement in JavaScript/HTML5
- Heap data structure for priority queue
- Grads: DFS, BFS, Greedy
- Submit through your git repository



2007-10: SOCCER WITH iROBOT CREATE



2007 - AR Tags for overhead localization



2007 - Mini ITX



2009 - Asus EEE

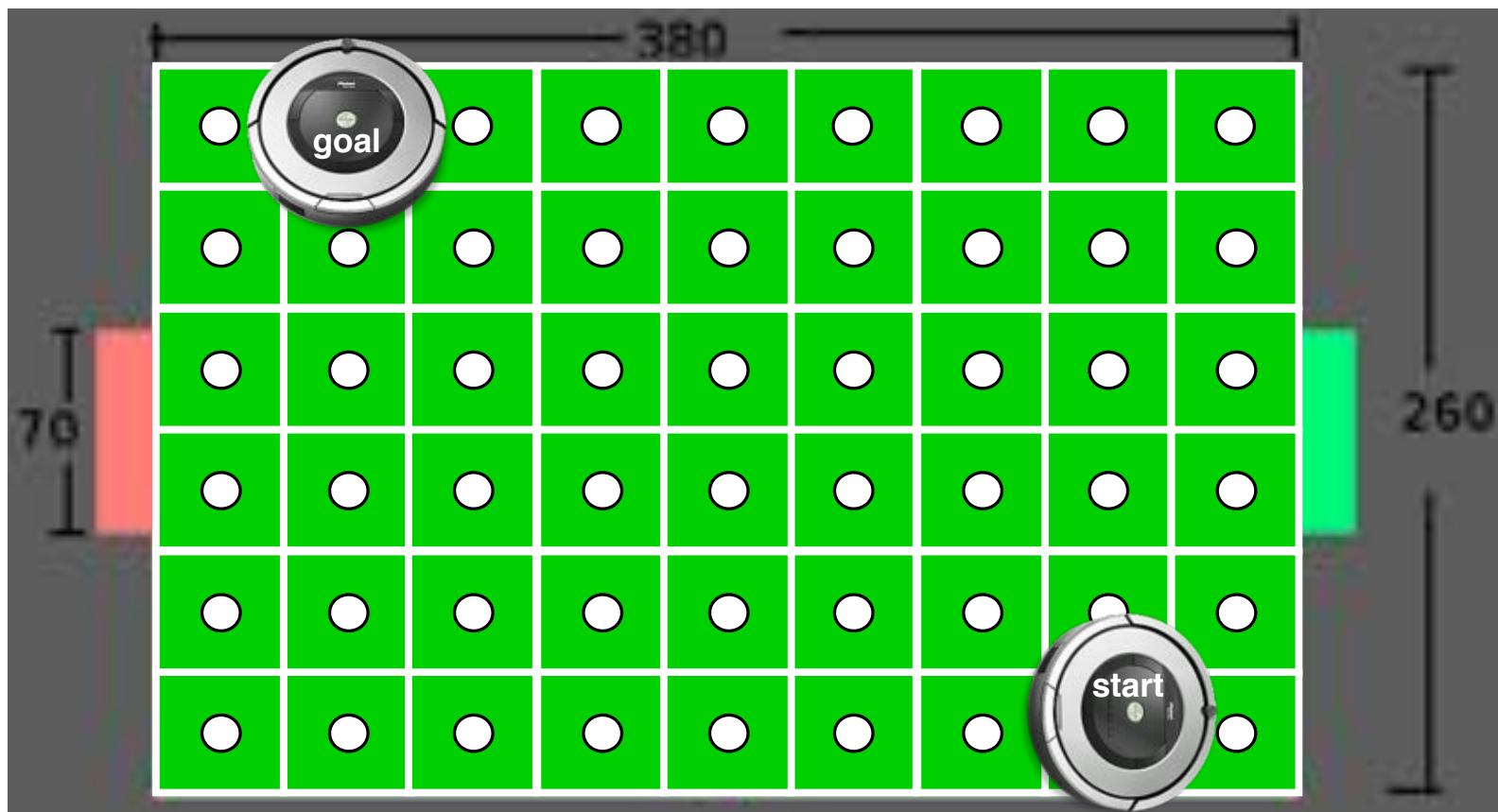
<http://www.youtube.com/watch?v=88zR6IC7S0g>
Michigan Robotics 367/510/567 - autorob.org



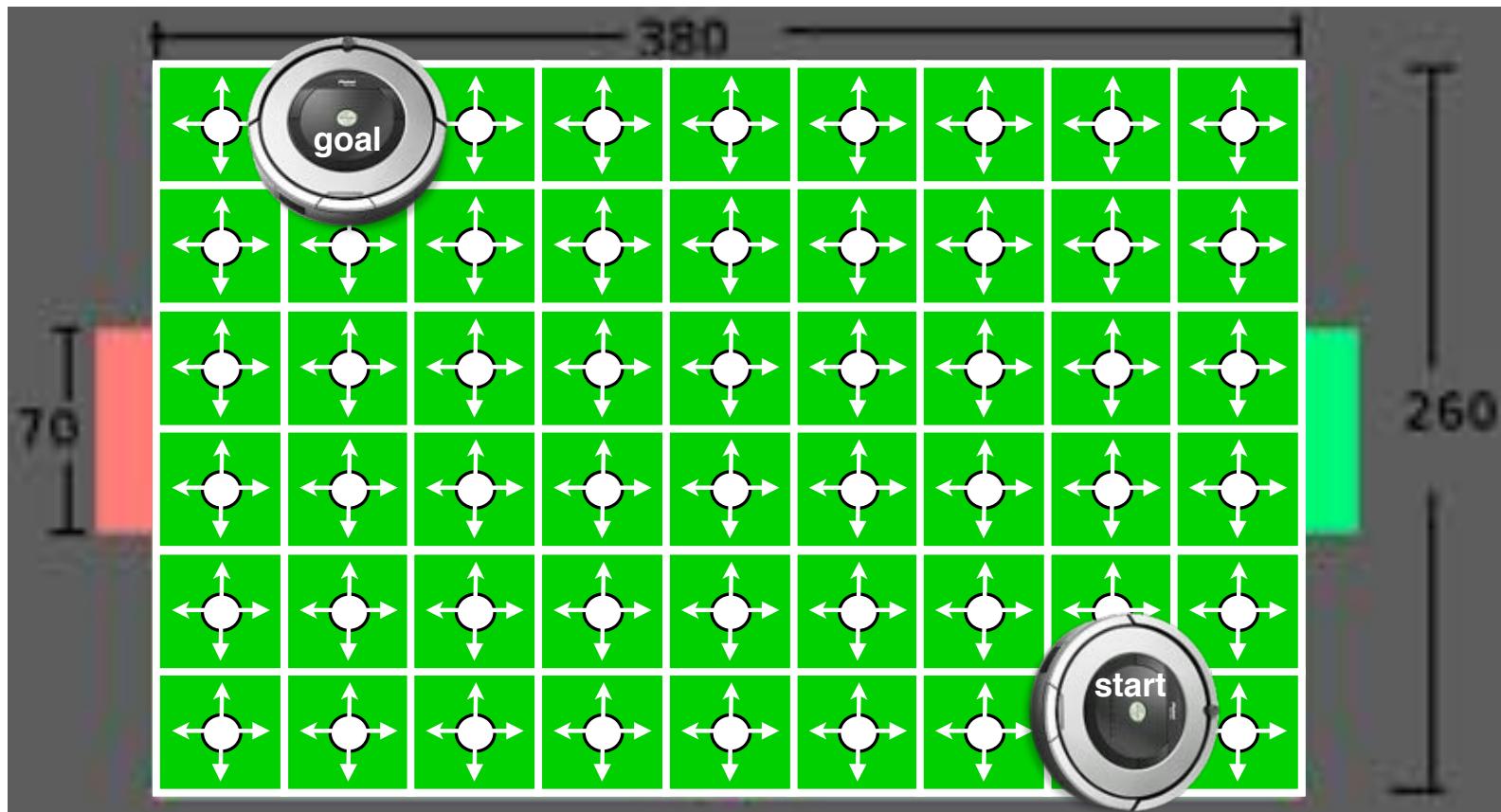


Michigan Robotics 367/510/567 - autorob.org

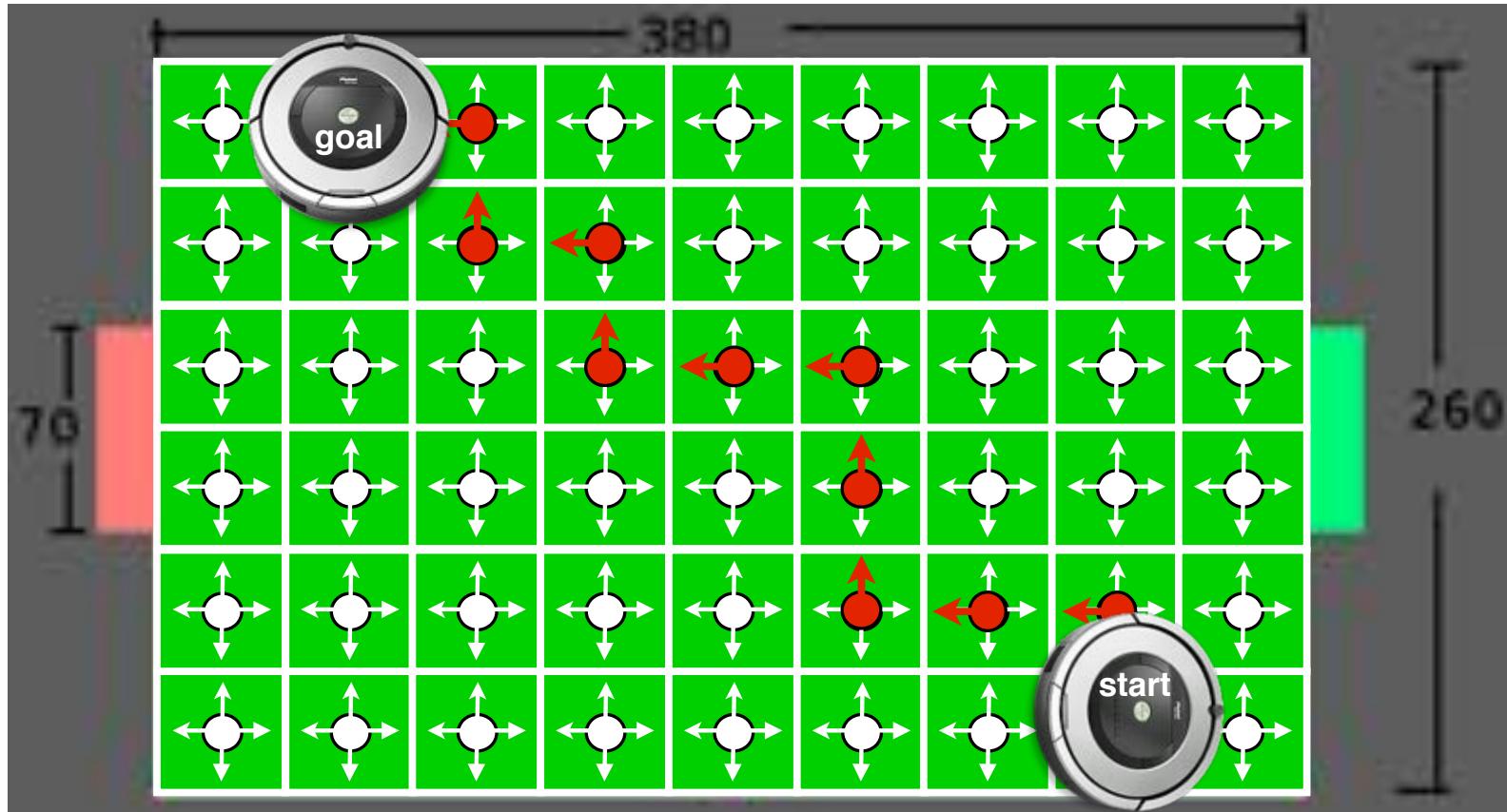
Consider all possible poses as uniformly distributed graph vertices



Consider all possible poses as uniformly distributed graph vertices
Edges connect adjacent (traversable) poses, weighted by distance



Consider all possible poses as uniformly distributed graph vertices
Edges connect adjacent (traversable) poses, weighted by distance
How to find a valid path in this graph?

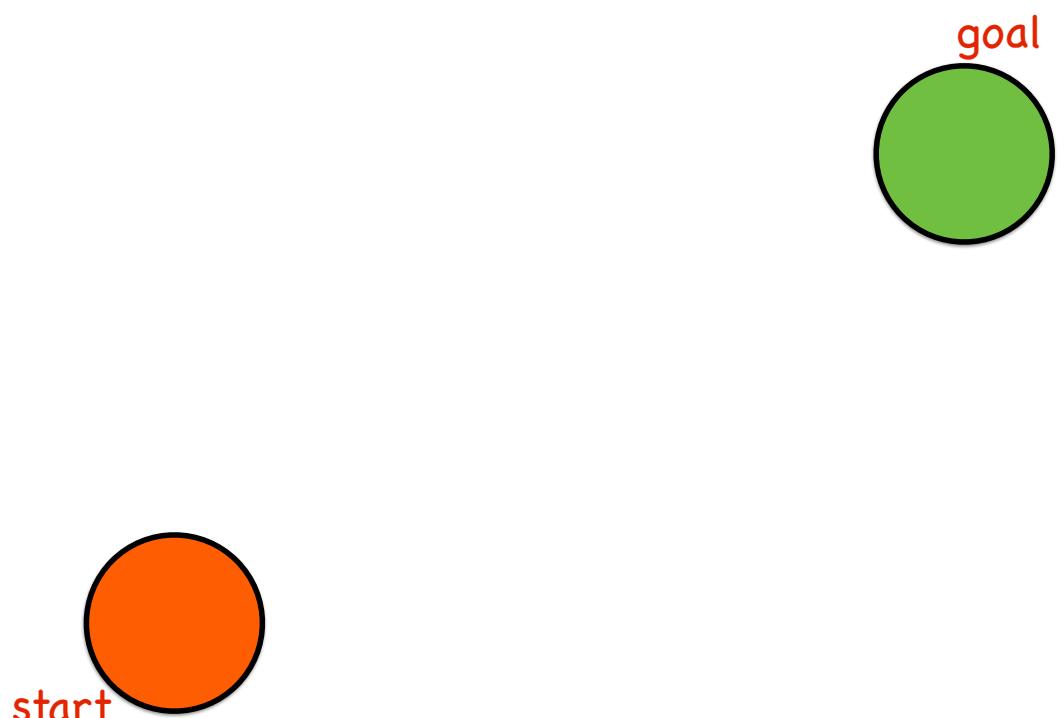


Approaches to motion planning

- Bug algorithms: Bug[0-2], Tangent Bug
- **Graph Search (fixed graph)**
 - **Depth-first, Breadth-first, Dijkstra, A-star, Greedy best-first**
- Sampling-based Search (build graph):
 - Probabilistic Road Maps, Rapidly-exploring Random Trees
- Optimization (local search):
 - Gradient descent, potential fields, Wavefront

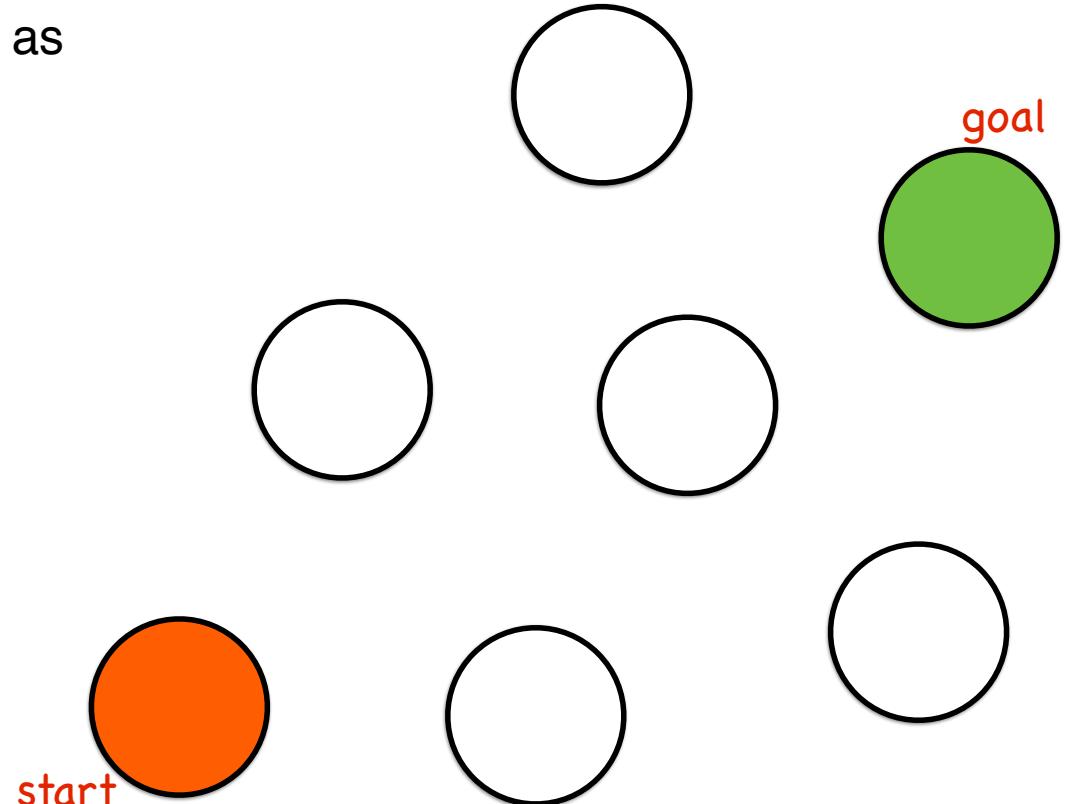
Consider a simple search graph

Consider each possible robot pose as node in a graph



Consider a simple search graph

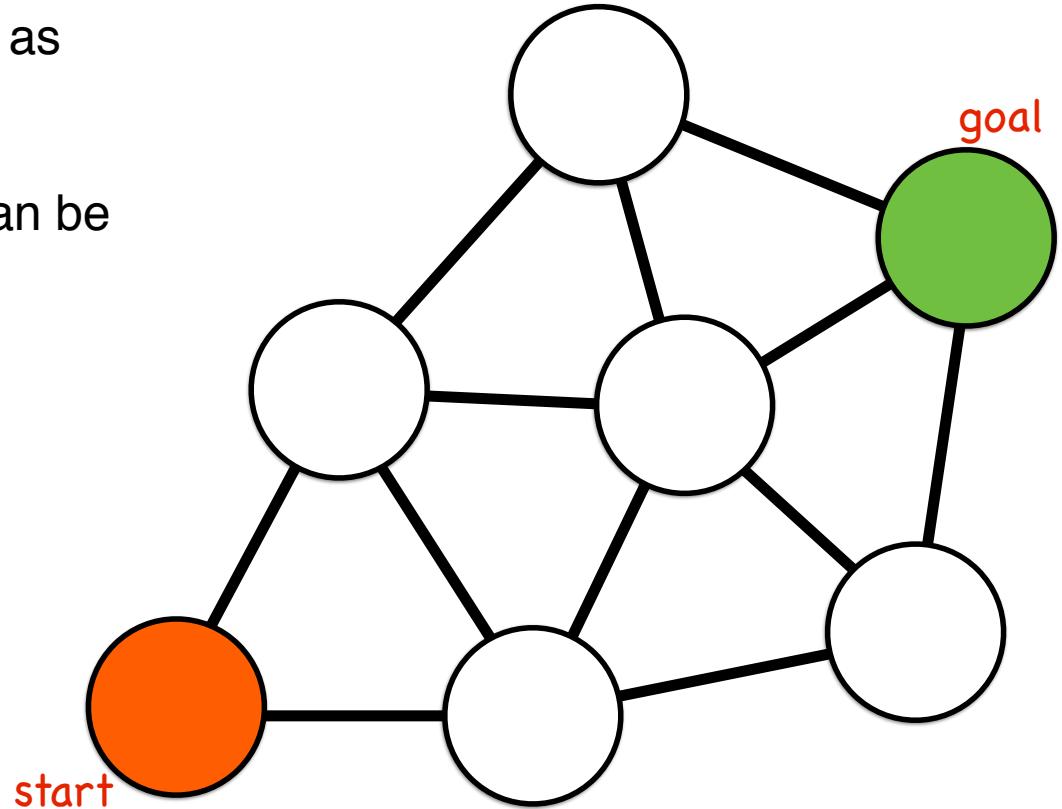
Consider each possible robot pose as node in a graph



Consider a simple search graph

Consider each possible robot pose as node in a graph

Graph edges connect poses that can be reliably moved between

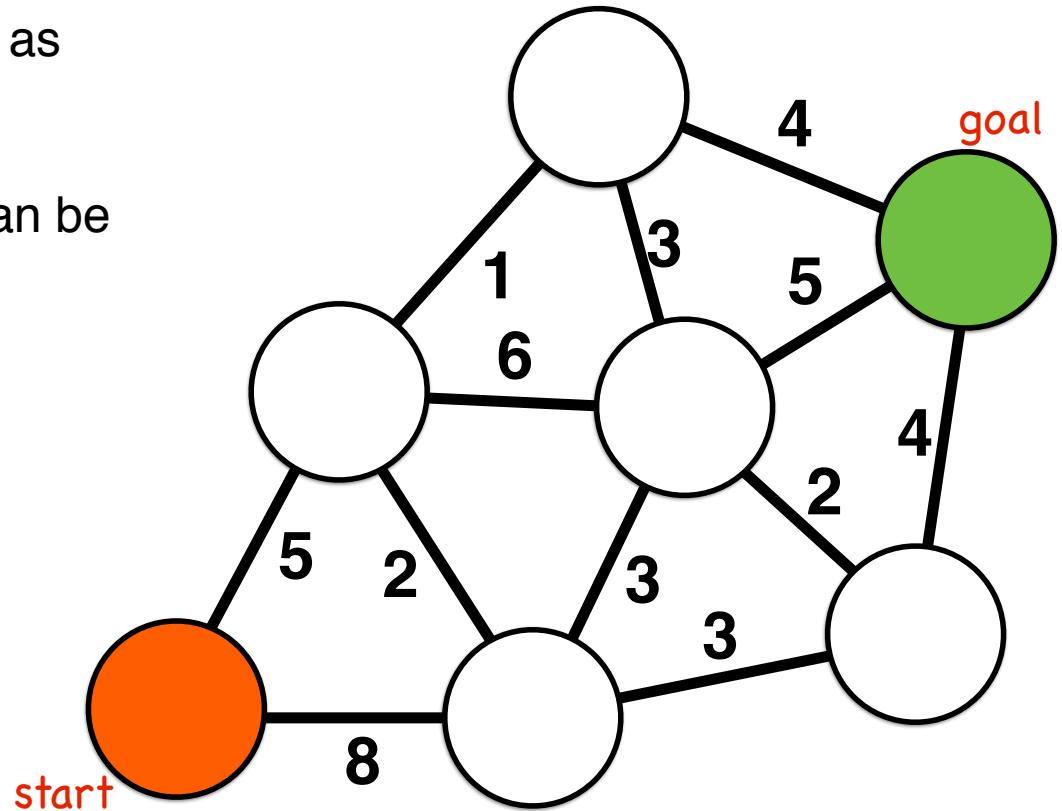


Consider a simple search graph

Consider each possible robot pose as node in a graph

Graph edges connect poses that can be reliably moved between

Edges have a cost for traversal



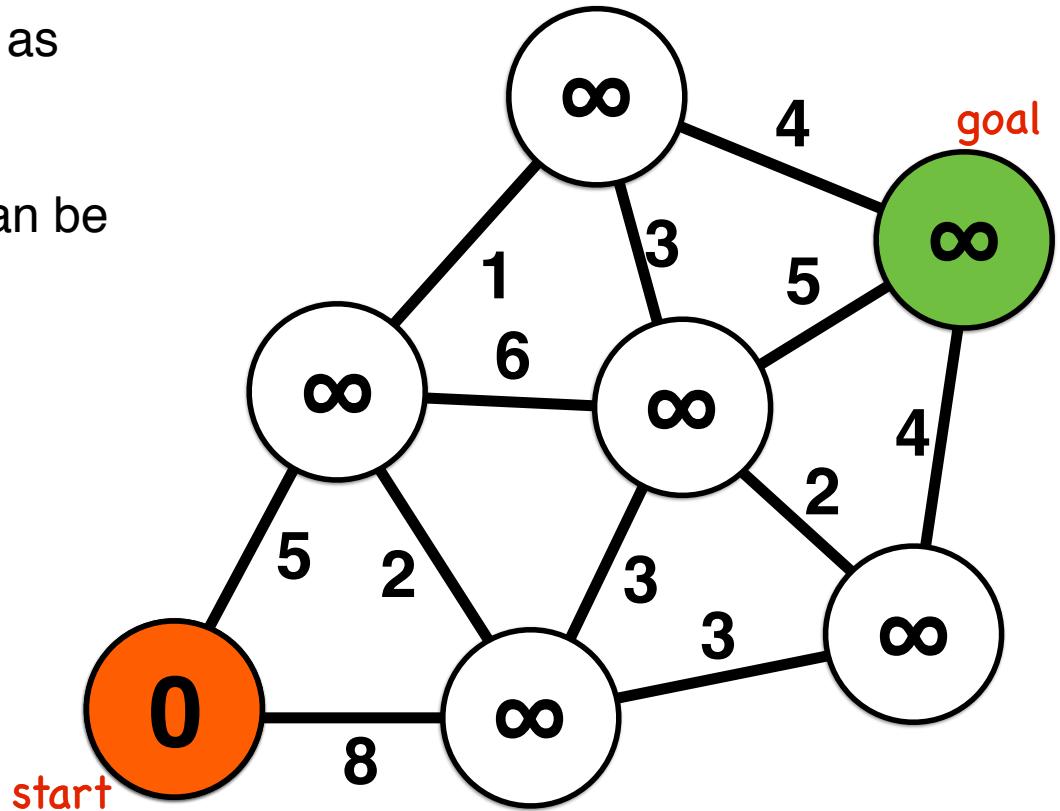
Consider a simple search graph

Consider each possible robot pose as node in a graph

Graph edges connect poses that can be reliably moved between

Edges have a cost for traversal

Each node maintains the **distance** traveled from start as a scalar cost



Consider a simple search graph

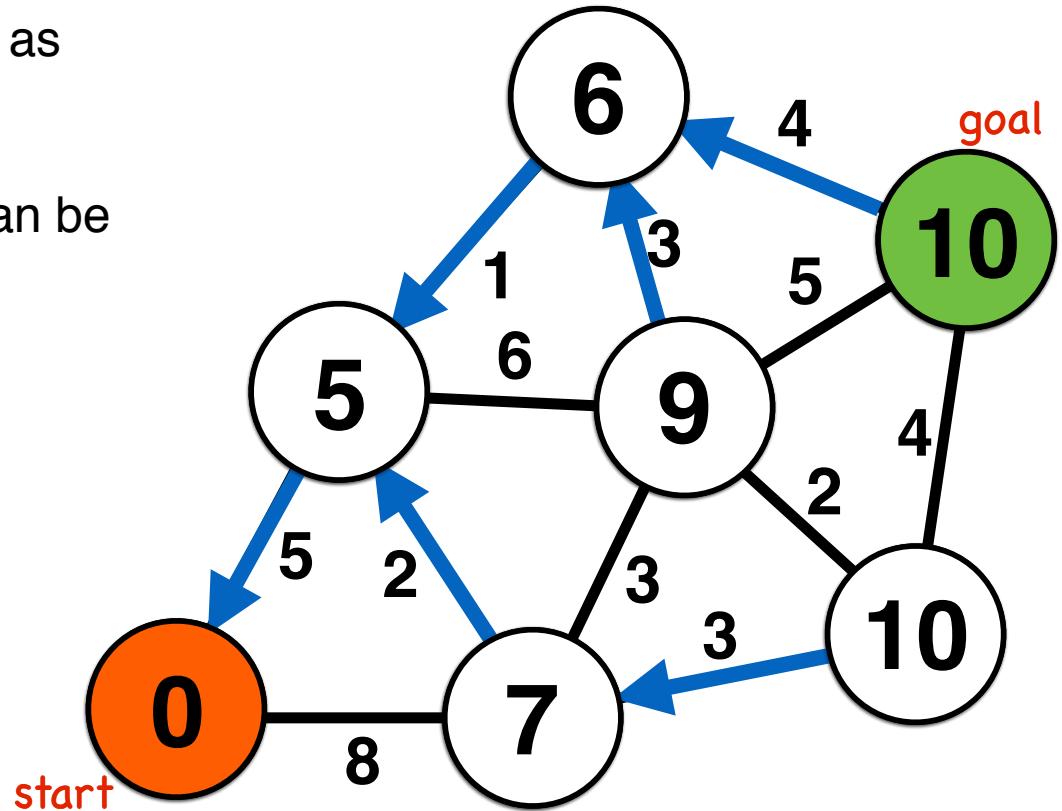
Consider each possible robot pose as node in a graph

Graph edges connect poses that can be reliably moved between

Edges have a cost for traversal

Each node maintains the **distance** traveled from start as a scalar cost

Each node has a **parent** node that specifies its route to the start node

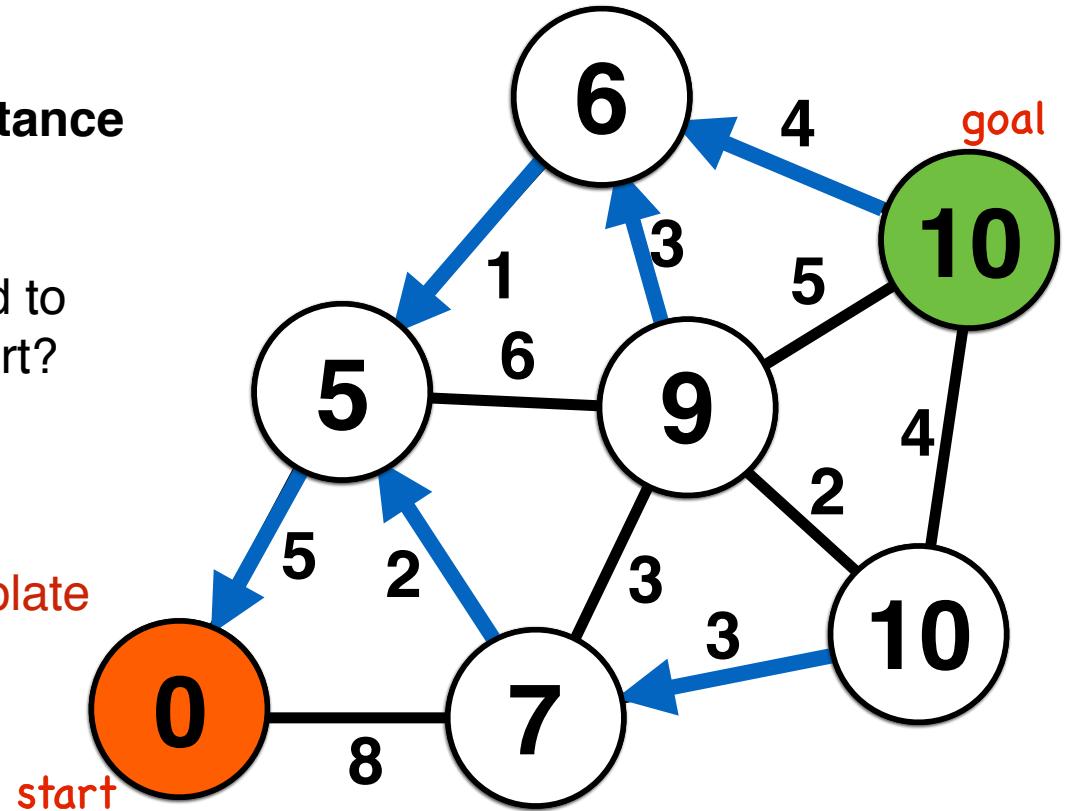


Path Planning as Graph Search

Which route is best to optimize **distance** traveled from start?

Which **parent** node should be used to specify route between goal and start?

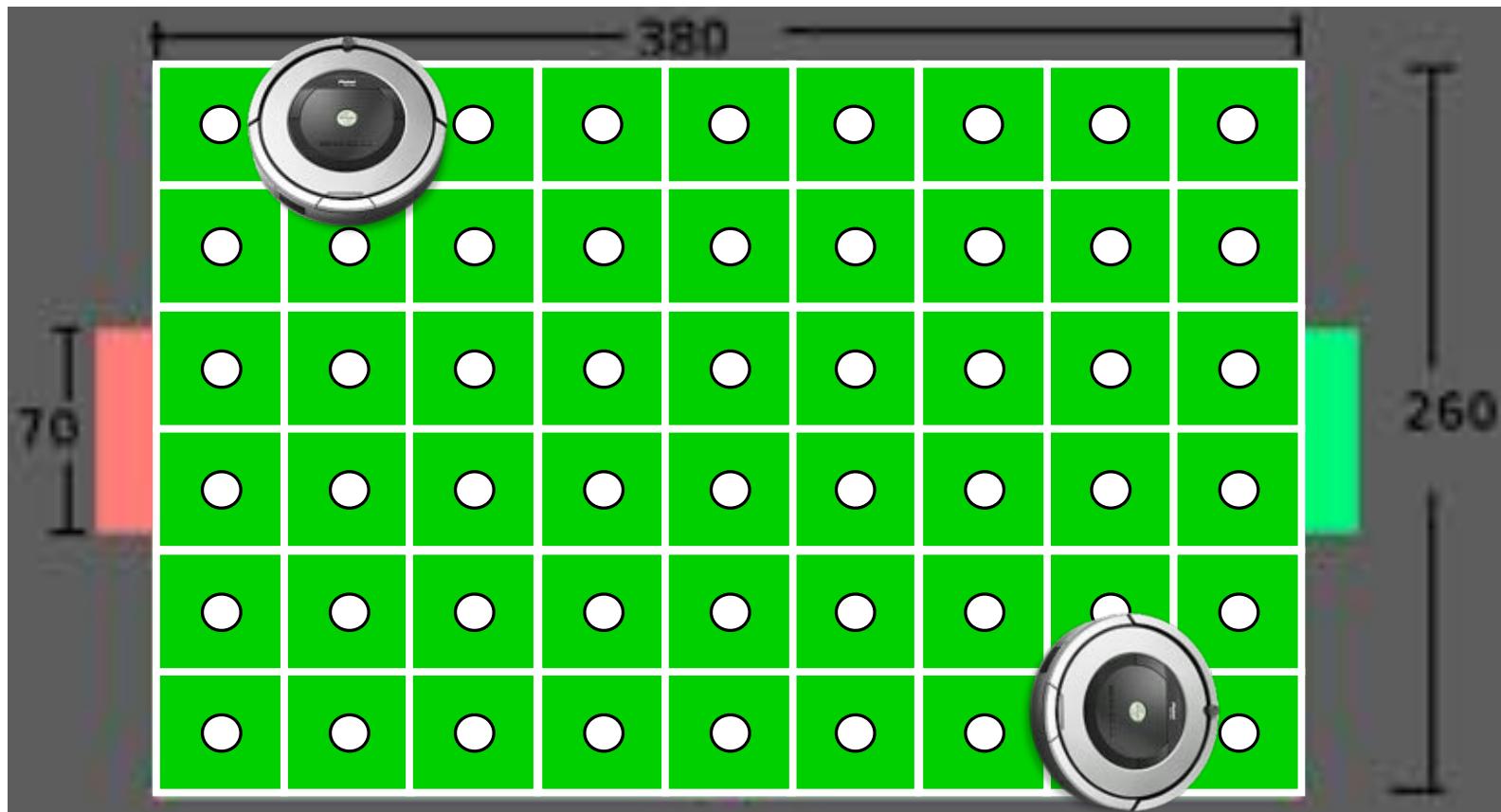
We will use a single algorithm template for our graph search computation



Depth-first search

intuition and walkthrough

Depth-first search



Depth-first search



Depth-first search



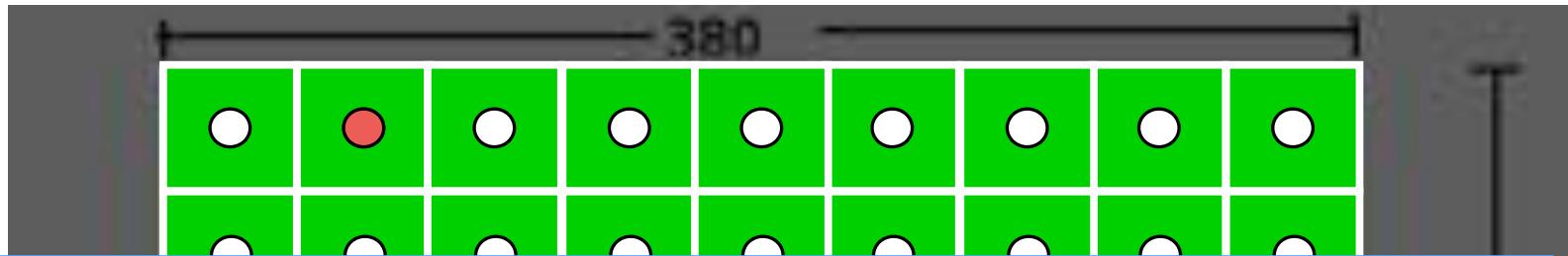
Depth-first search



Depth-first search



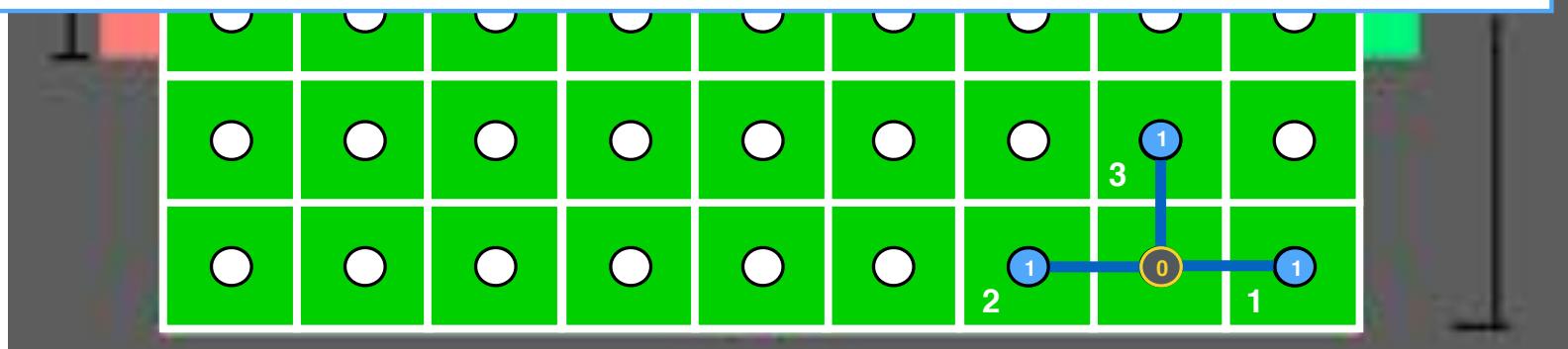
Depth-first search



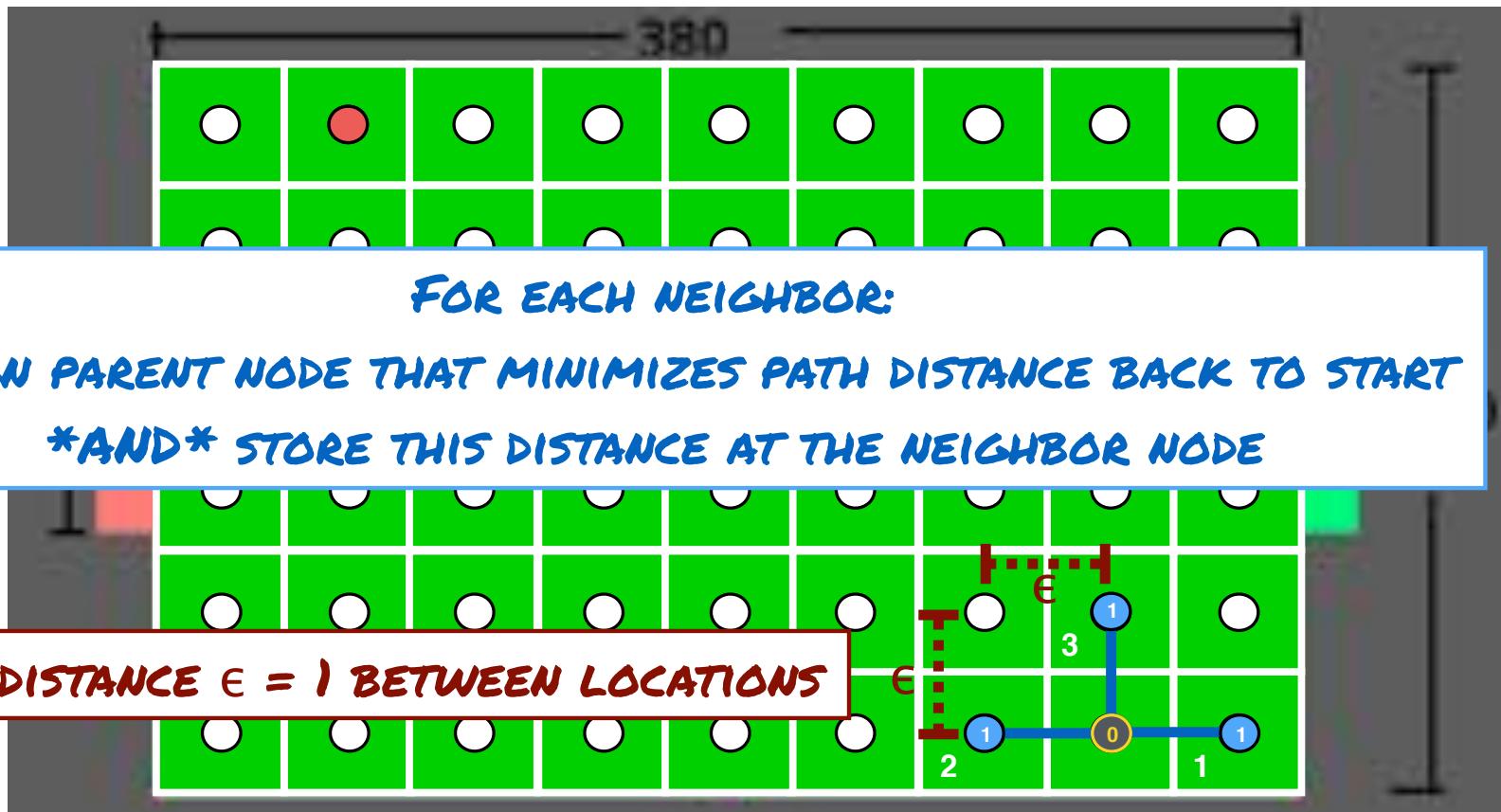
FOR EACH NEIGHBOR:

ASSIGN PARENT NODE THAT MINIMIZES PATH DISTANCE BACK TO START

***AND* STORE THIS DISTANCE AT THE NEIGHBOR NODE**



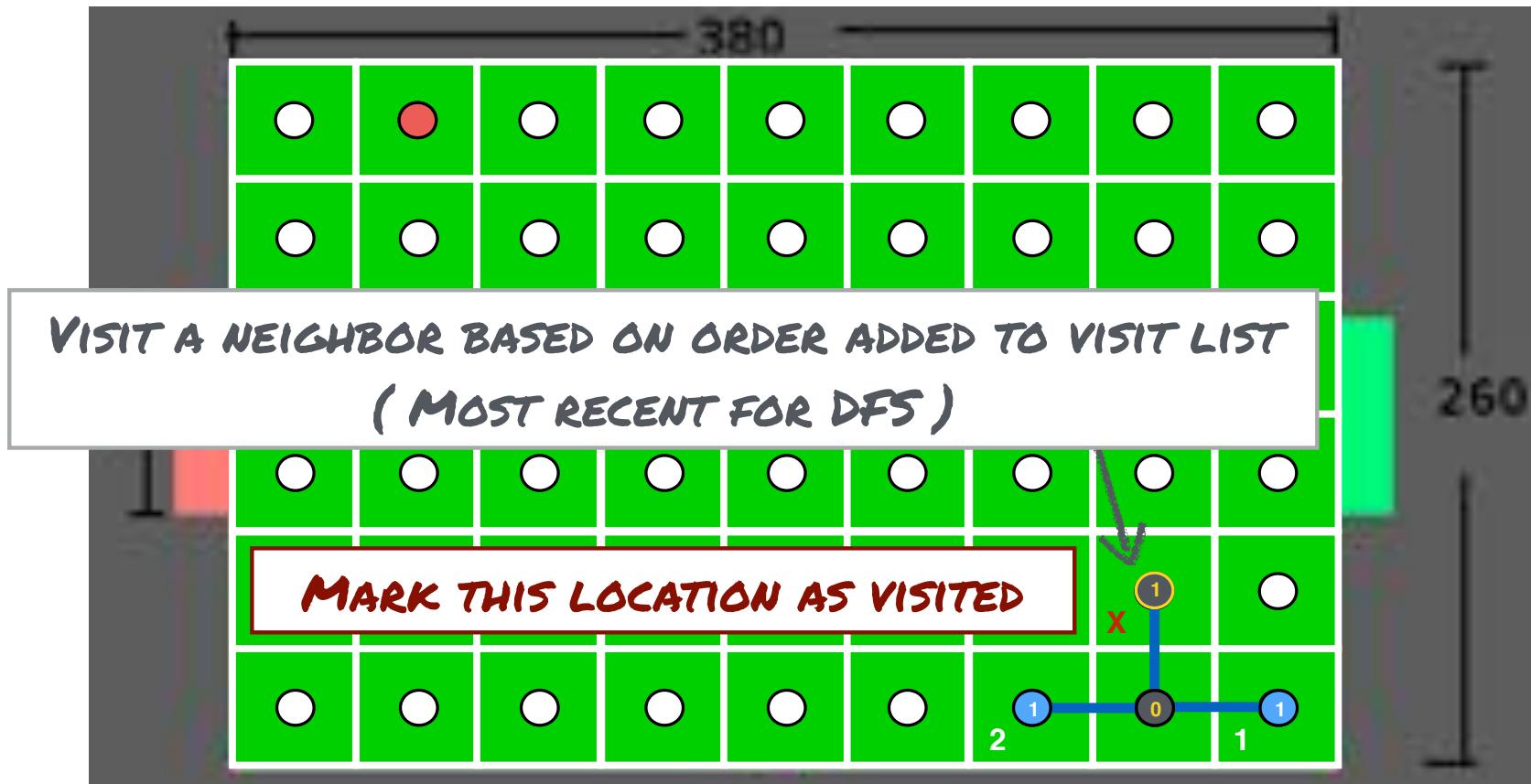
Depth-first search



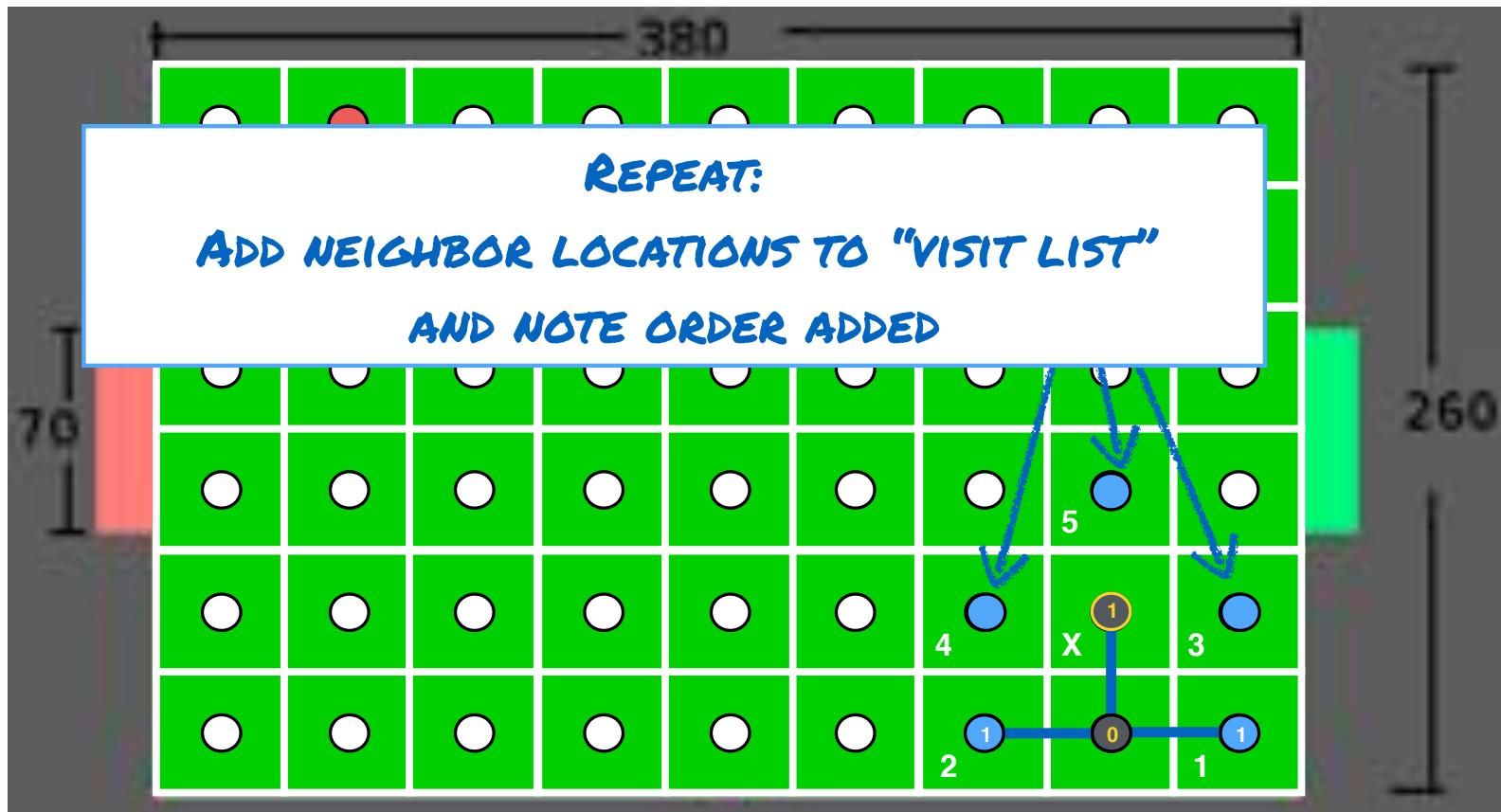
Depth-first search



Depth-first search



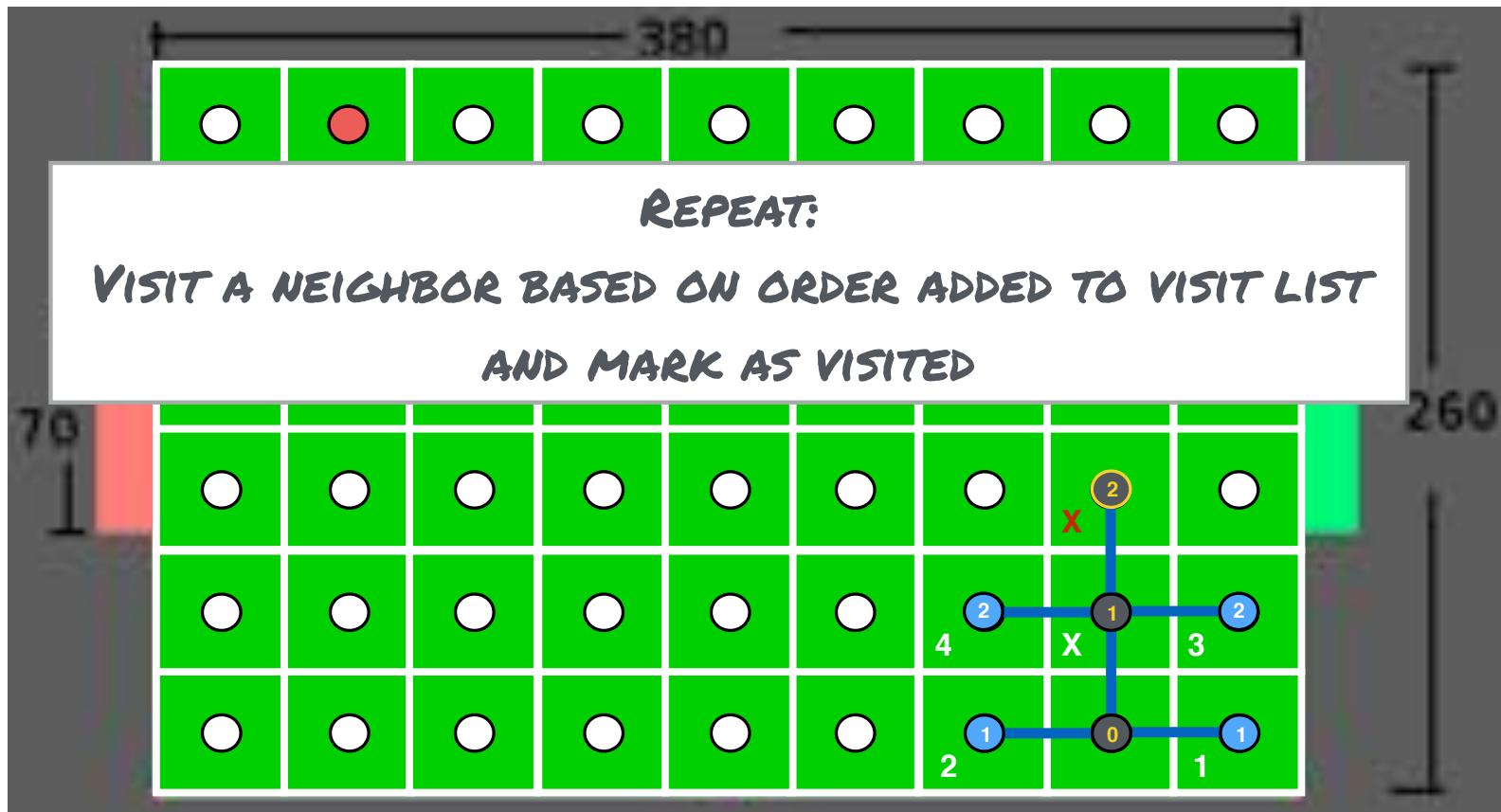
Depth-first search



Depth-first search



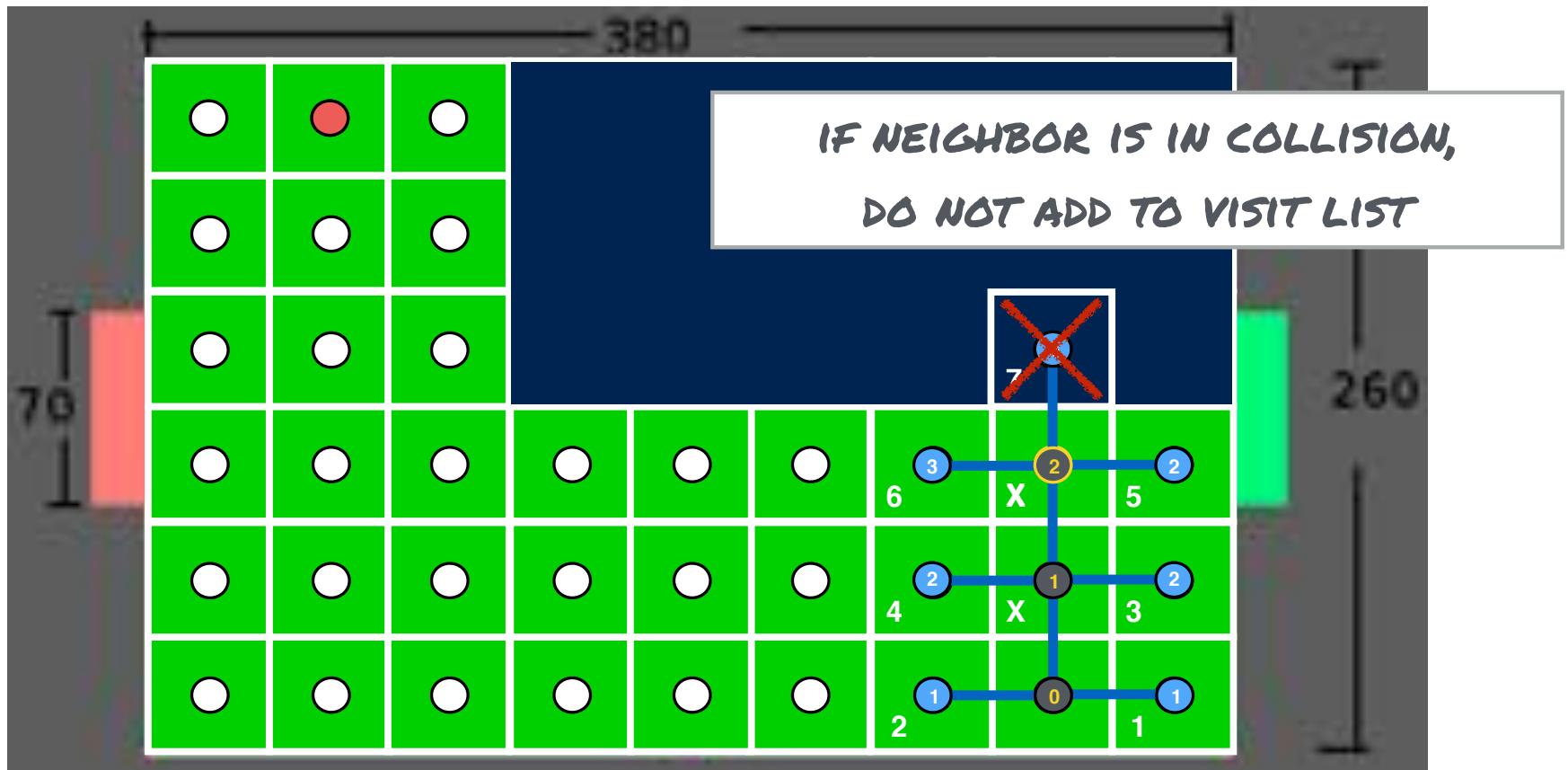
Depth-first search



Depth-first search



Depth-first search



Depth-first search



Depth-first search



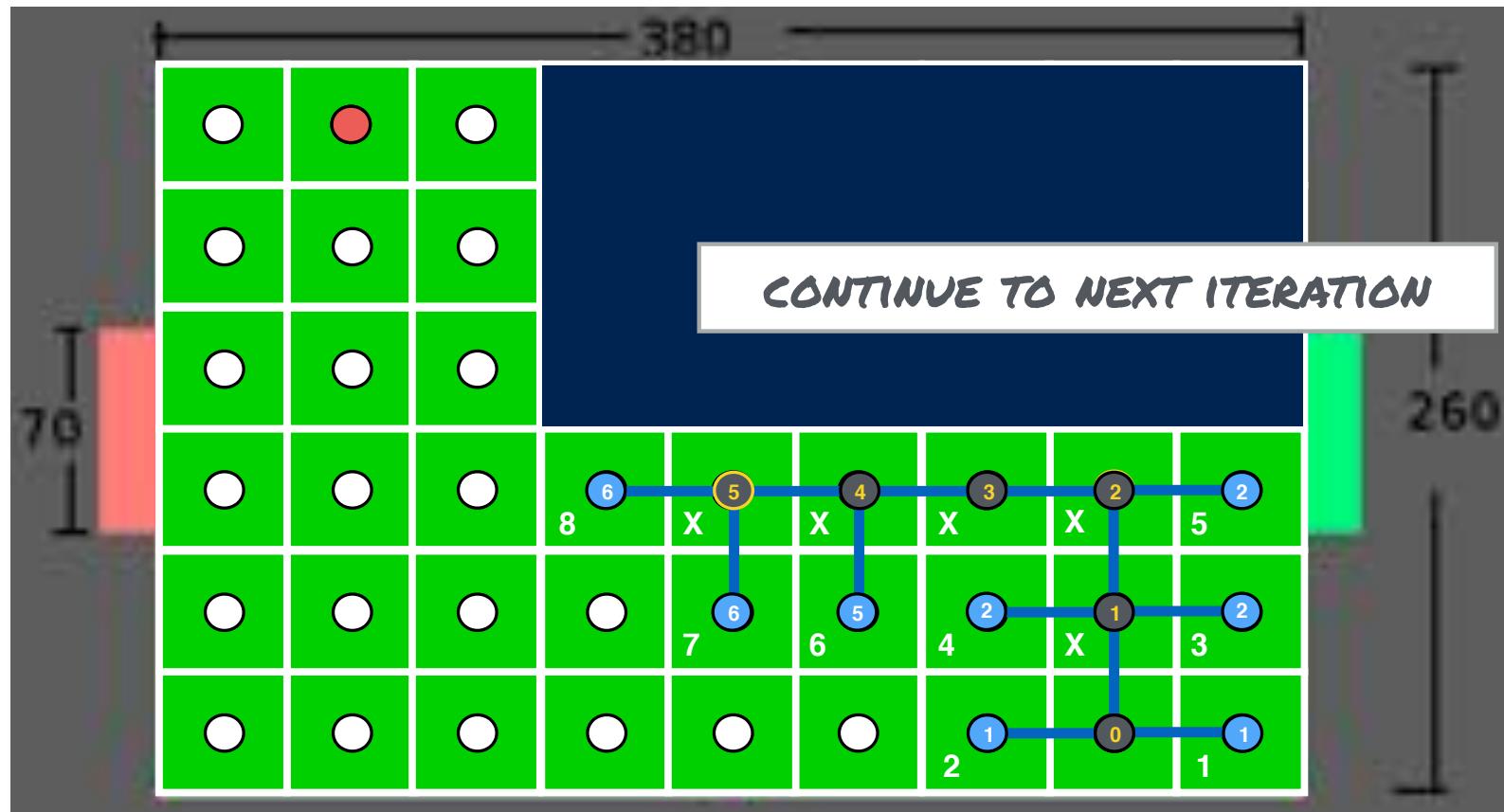
Depth-first search



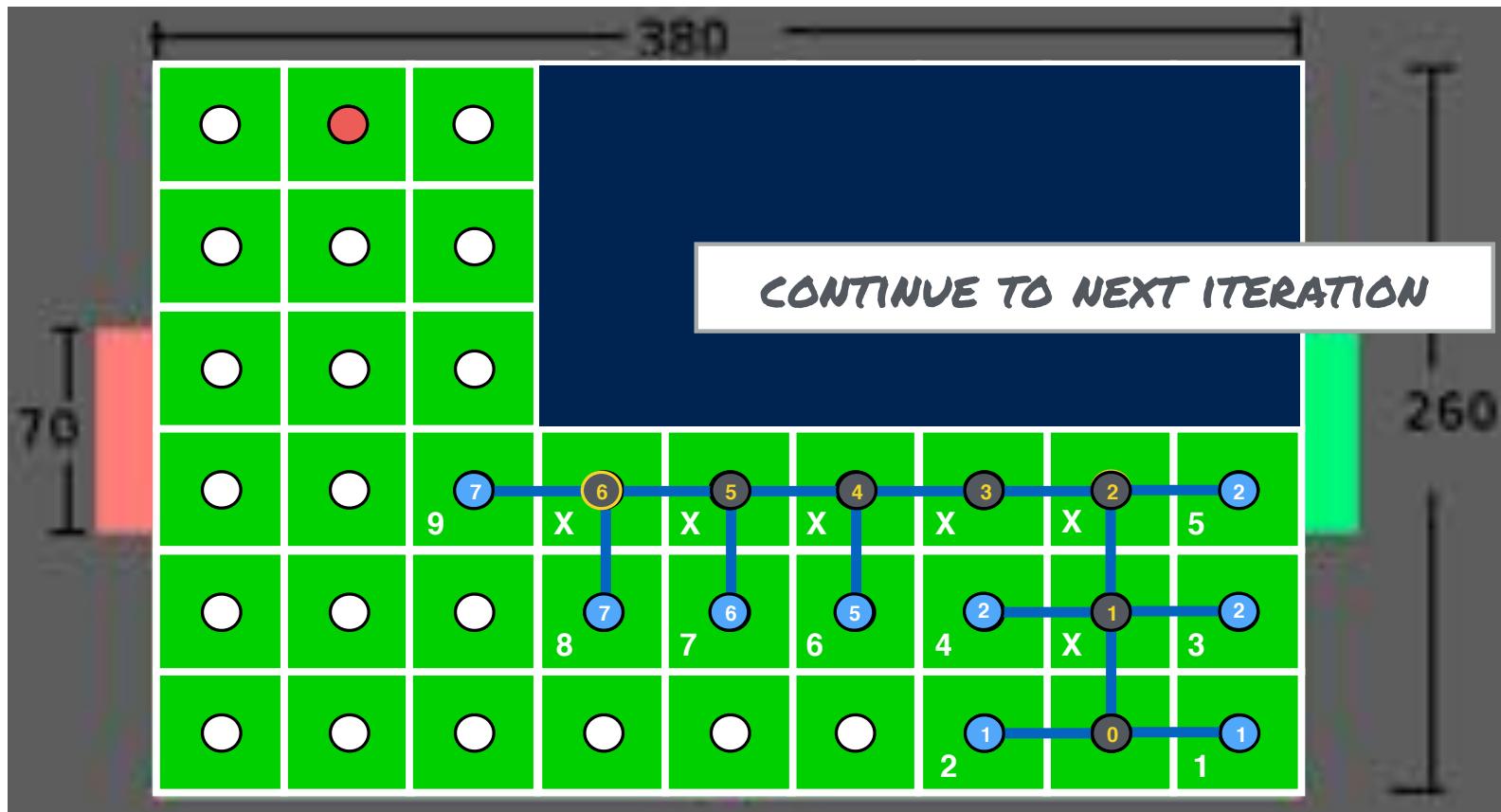
Depth-first search



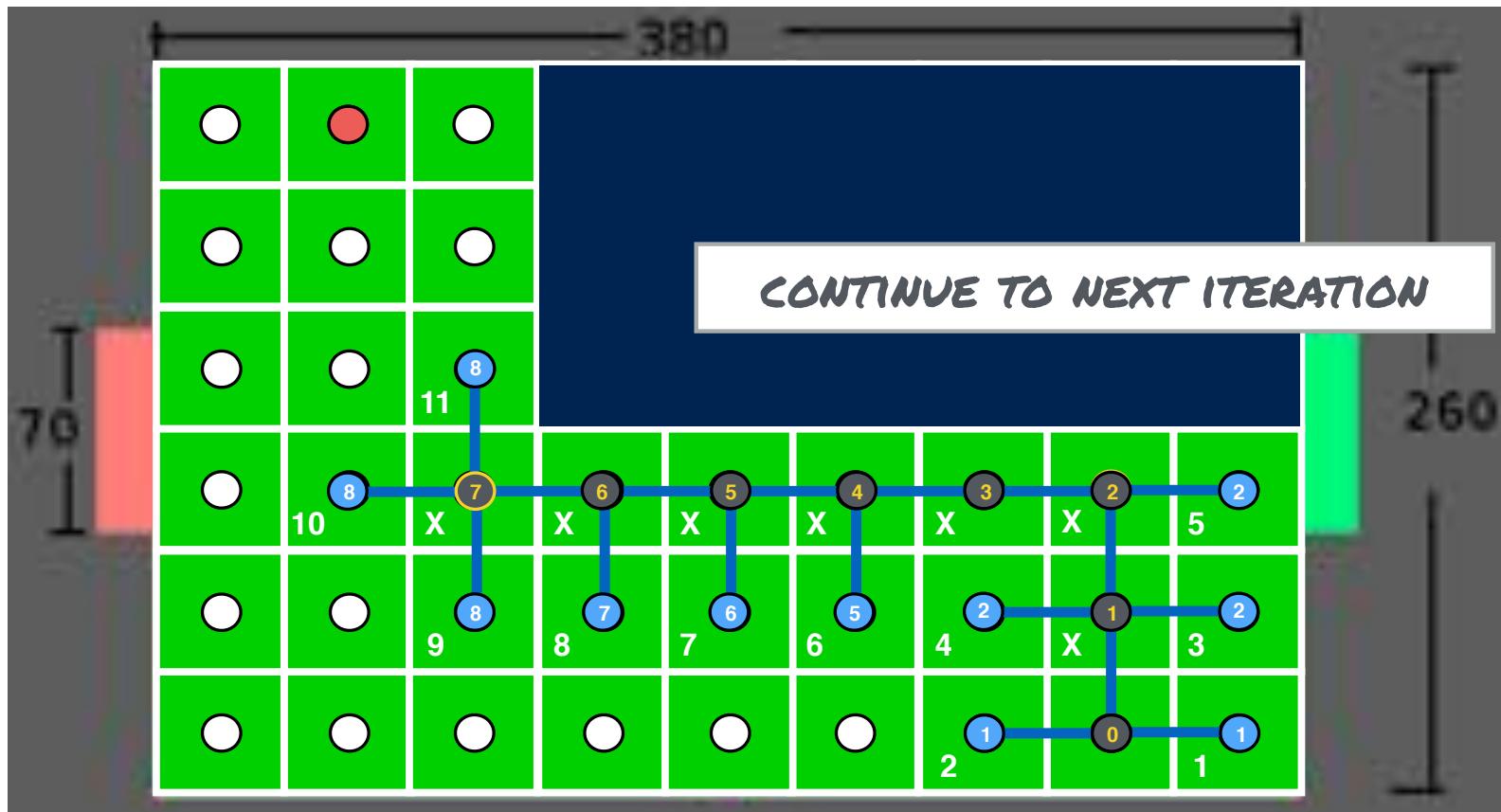
Depth-first search



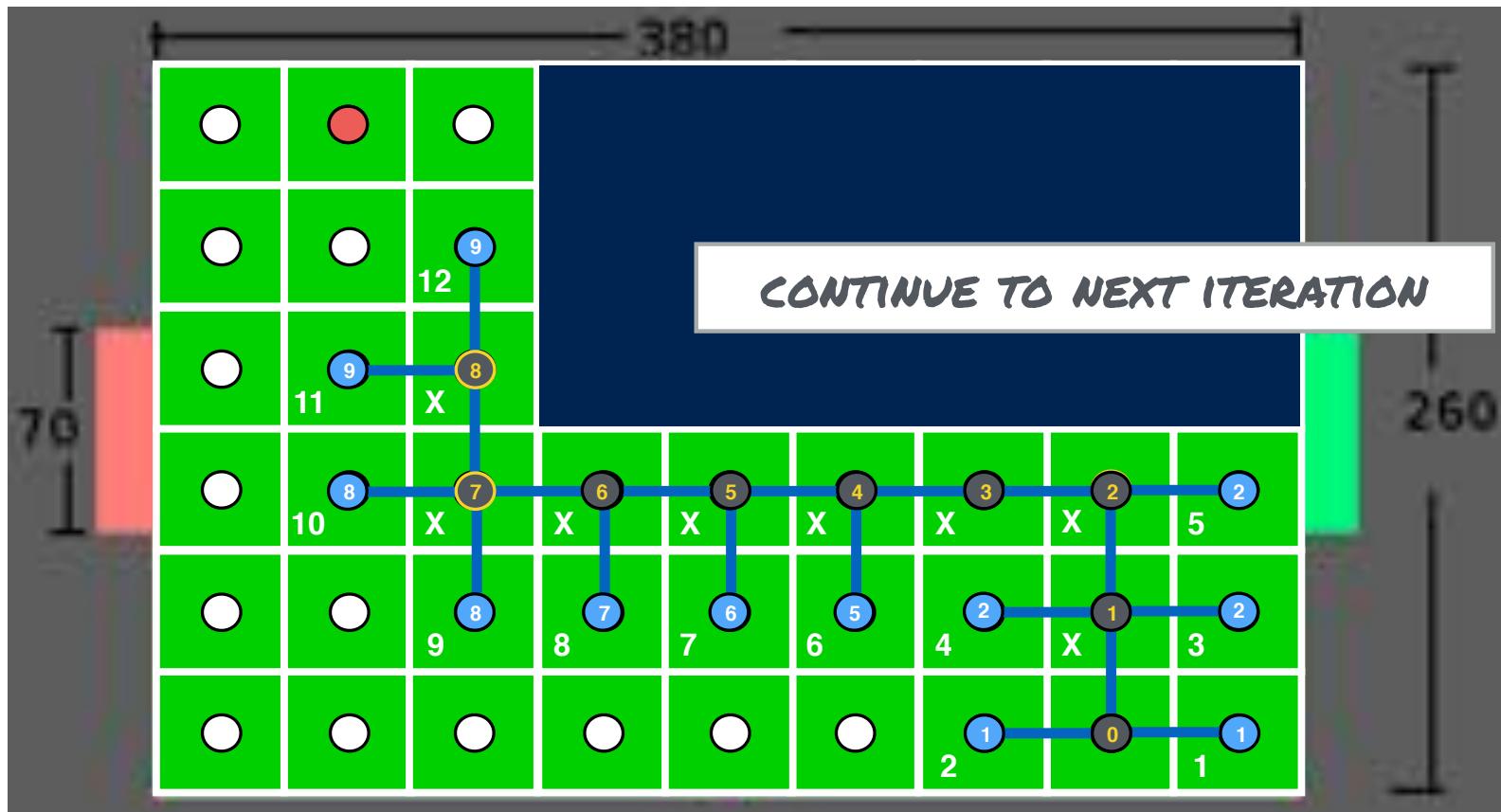
Depth-first search



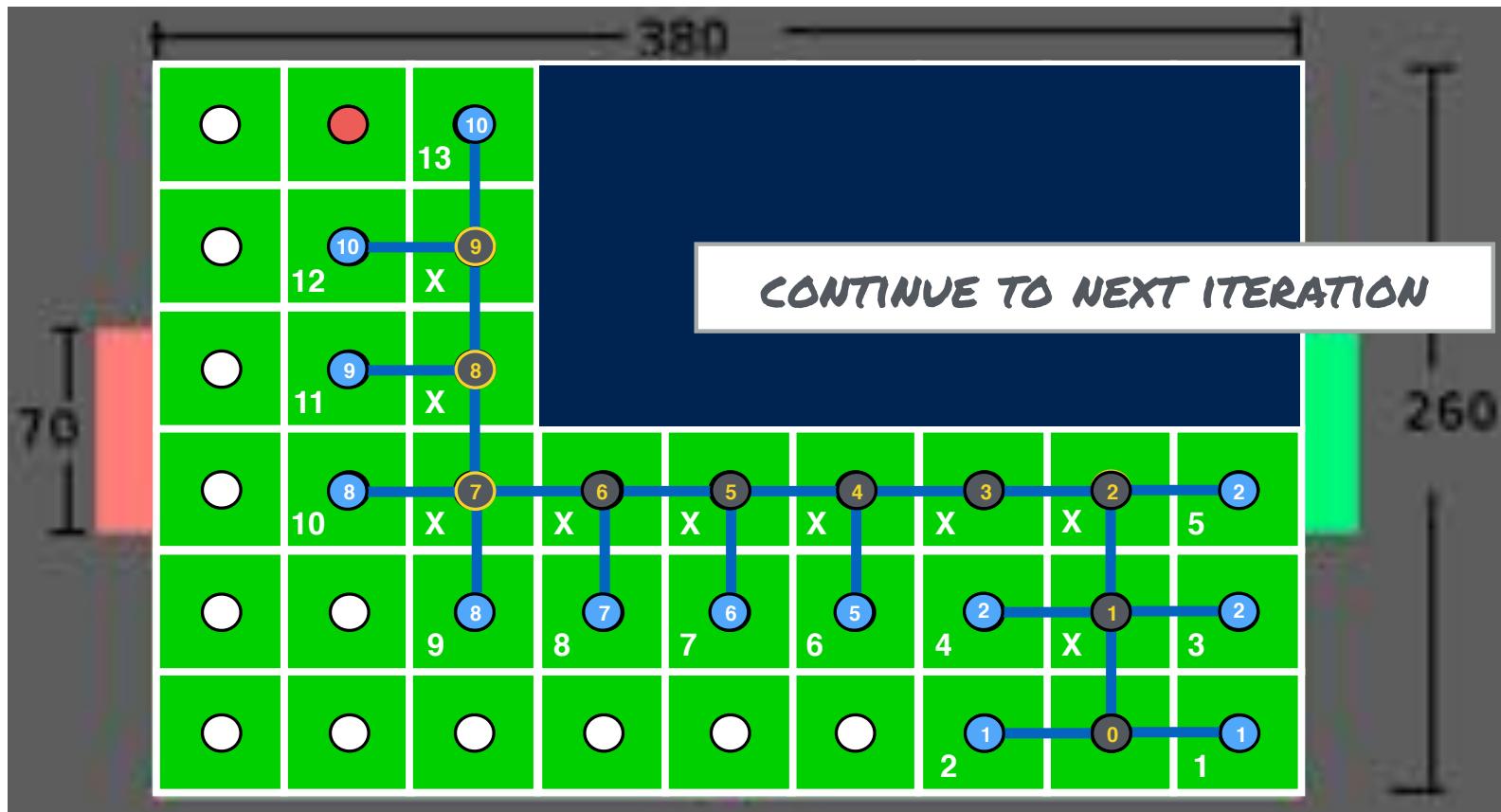
Depth-first search



Depth-first search



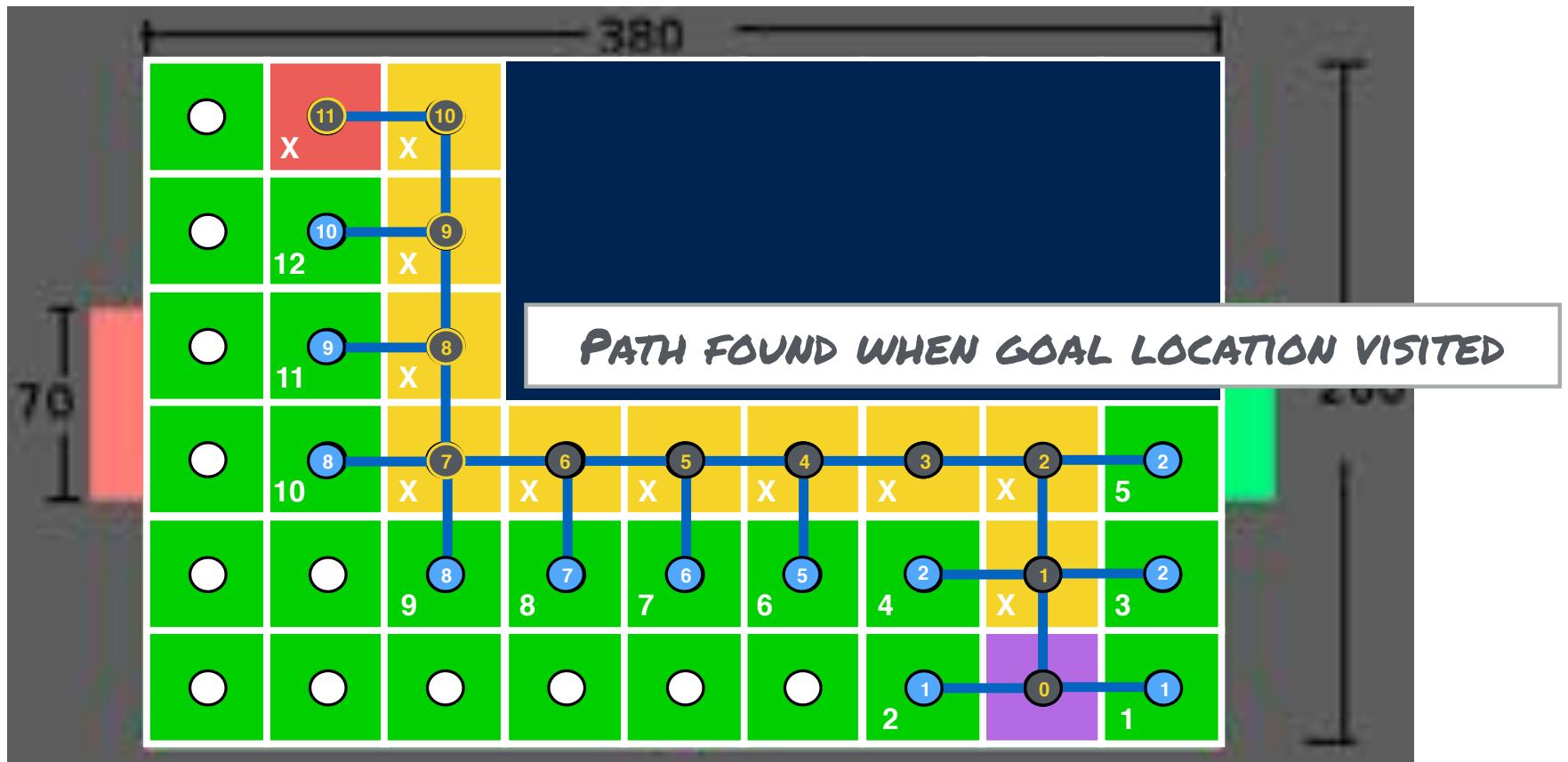
Depth-first search



Depth-first search



Depth-first search



Let's turn this idea into code

Search algorithm template

```
all nodes  $\leftarrow \{\text{dist}_{\text{start}} \leftarrow \text{infinity}, \text{parent}_{\text{start}} \leftarrow \text{none}, \text{visited}_{\text{start}} \leftarrow \text{false}\}$ 
start_node  $\leftarrow \{\text{dist}_{\text{start}} \leftarrow 0, \text{parent}_{\text{start}} \leftarrow \text{none}, \text{visited}_{\text{start}} \leftarrow \text{true}\}$ 
visit_list  $\leftarrow \text{start\_node}$ 
```

```
while visit_list != empty && current_node != goal
```

```
    cur_node  $\leftarrow \text{highestPriority}(\text{visit\_list})$ 
```

```
    visitedcur_node  $\leftarrow \text{true}$ 
```

```
    for each nbr in not_visited(adjacent(cur_node))
```

```
        add(nbr to visit_list)
```

```
        if distnbr > distcur_node + distStraightLine(nbr,cur_node)
```

```
            parentnbr  $\leftarrow \text{current\_node}$ 
```

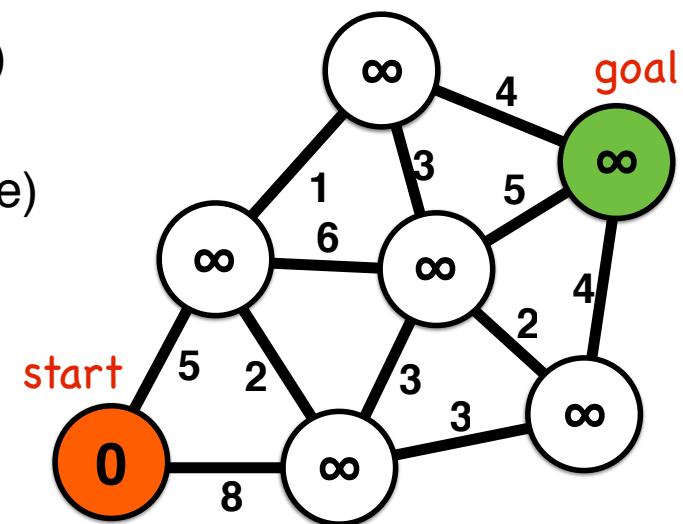
```
            distnbr  $\leftarrow \text{dist}_{\text{cur\_node}} + \text{distStraightLine}(\text{nbr}, \text{cur\_node})$ 
```

```
        end if
```

```
    end for loop
```

```
    end while loop
```

```
output  $\leftarrow \text{parent, distance}$ 
```



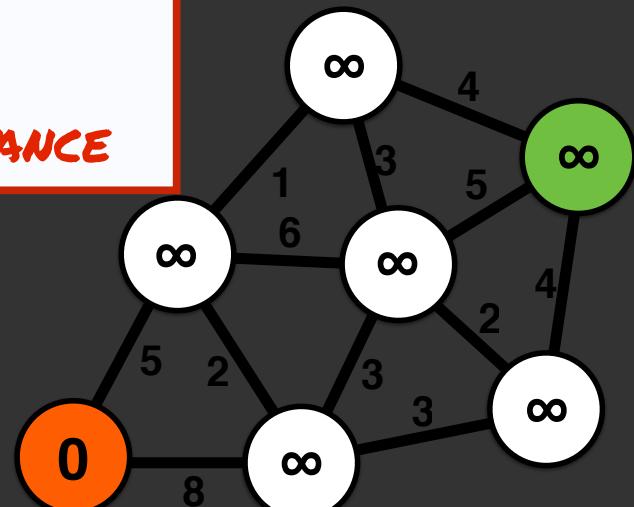
Search algorithm template

```
all nodes ← {diststart ← infinity, parentstart ← none, visitedstart ← false}  
start_node ← {diststart ← 0, parentstart ← none, visitedstart ← true}  
visit_list ← start_node
```

INITIALIZATION

- EACH NODE HAS A DISTANCE AND A PARENT
 - DISTANCE: DISTANCE ALONG ROUTE FROM START
 - PARENT: ROUTING FROM NODE TO START
- VISIT A CHOSEN START NODE FIRST
- ALL OTHER NODES ARE UNVISITED AND HAVE HIGH DISTANCE

```
    diststart ← infinity  
    distcur_node ← infinity  
    if distcur_node < infinity then  
        distcur_node ← diststart + diststraight(start, cur_node)  
        parentcur_node ← start  
        visit_list.append(cur_node)  
    end if  
    end for loop  
end while loop  
output ← parent, distance
```



Search algorithm template

```
all nodes ← {diststart ← infinity, parentstart ← none, visitedstart ← false}  
start_node ← {diststart ← 0, parentstart ← none, visitedstart ← true}  
visit_list ← start_node
```

```
while visit_list != empty && current_node != goal  
    cur_node ← highestPriority(visit_list)  
    visitedcur_node ← true
```

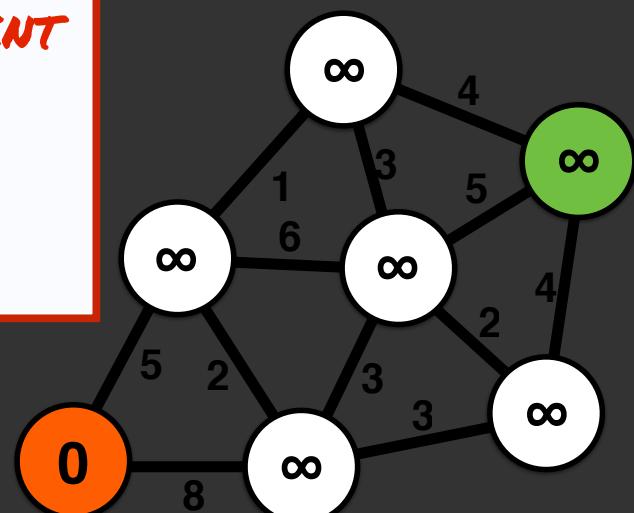
MAIN LOOP

- VISITS EVERY NODE TO COMPUTE ITS DISTANCE AND PARENT
- AT EACH ITERATION:
 - SELECT THE NODE TO VISIT BASED ON ITS PRIORITY
 - REMOVE CURRENT NODE FROM VISIT_LIST

end for loop

end while loop

output ← parent, distance



Search algorithm template

```
all nodes  $\leftarrow \{dist_{start} \leftarrow \text{infinity}, parent_{start} \leftarrow \text{none}, visited}_{start} \leftarrow \text{false}\}$   
start_node  $\leftarrow \{dist_{start} \leftarrow 0, parent_{start} \leftarrow \text{none}, visited}_{start} \leftarrow \text{true}\}$   
visit_list  $\leftarrow \text{start\_node}$ 
```

```
while visit_list != empty && current_node != goal
```

```
    cur_node  $\leftarrow \text{highestPriority(visit\_list)}$ 
```

```
    visitedcur_node  $\leftarrow \text{true}$ 
```

```
for each nbr in not_visited(adjacent(cur_node))
```

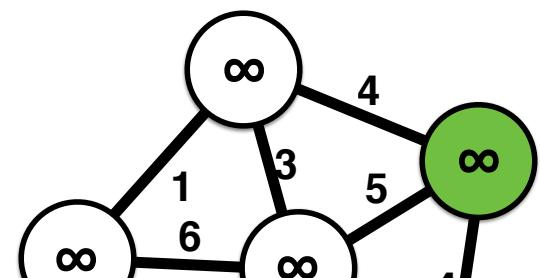
```
    add(nbr to visit_list)
```

```
    if distnbr > distcur_node + distStraightLine(nbr,cur_node)
```

```
        parentnbr  $\leftarrow \text{current\_node}$ 
```

```
        distnbr  $\leftarrow dist_{cur\_node} + distStraightLine(nbr,cur\_node)$ 
```

```
end if
```



FOR EACH ITERATION ON A SINGLE NODE

- ADD ALL UNVISITED NEIGHBORS OF THE NODE TO THE VISIT LIST

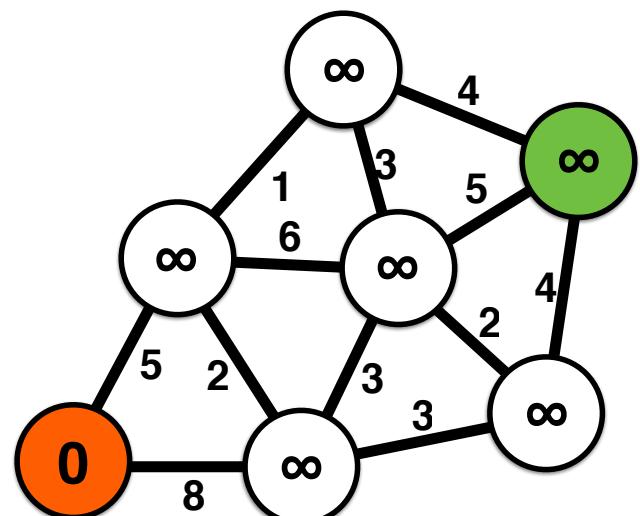
- ADD THE NODE AS A PARENT TO A NEIGHBOR, IF IT CREATES A SHORTER ROUTE

Search algorithm template

```
all nodes  $\leftarrow \{dist_{start} \leftarrow \text{infinity}, parent_{start} \leftarrow \text{none}, visited}_{start} \leftarrow \text{false}\}$ 
start_node  $\leftarrow \{dist_{start} \leftarrow 0, parent_{start} \leftarrow \text{none}, visited}_{start} \leftarrow \text{true}\}$ 
visit_list  $\leftarrow start\_node$ 

while visit_list != empty && current_node != goal
    cur_node  $\leftarrow \text{highestPriority(visit\_list)}$ 
    visitedcur_node  $\leftarrow \text{true}$ 
    for each nbr in not_visited(adjacent(cur_node))
        add(nbr to visit_list)
        if distnbr > distcur_node + distance(nbr,cur_node)
            parentnbr  $\leftarrow current\_node$ 
            distnbr  $\leftarrow dist_{cur\_node} + distance(nbr,cur\_node)$ 
        end if
    end for loop
end while loop
output  $\leftarrow parent, distance$ 
```

OUTPUT THE RESULTING ROUTES AND DISTANCES



Depth-first search

Search algorithm template

```
all nodes  $\leftarrow \{dist_{start} \leftarrow \text{infinity}, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{false}\}$ 
start_node  $\leftarrow \{dist_{start} \leftarrow 0, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{true}\}$ 
visit_list  $\leftarrow \text{start\_node}$ 
```

```
while visit_list != empty && current_node != goal
```

```
    cur_node  $\leftarrow \text{highestPriority}(\text{visit\_list})$ 
```

```
    visitedcur_node  $\leftarrow \text{true}$ 
```

```
    for each nbr in not_visited(adjacent(cur_node))
```

```
        add(nbr to visit_list)
```

```
        if distnbr > distcur_node + distance(nbr,cur_node)
```

```
            parentnbr  $\leftarrow \text{current\_node}$ 
```

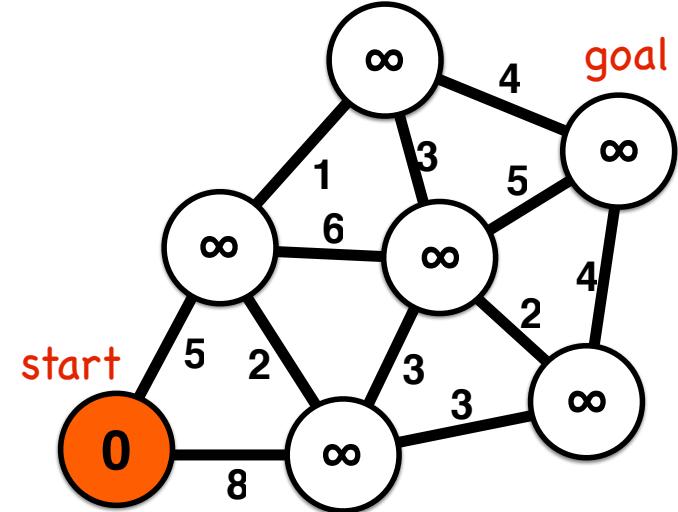
```
            distnbr  $\leftarrow dist_{cur\_node} + distance(nbr,cur\_node)$ 
```

```
        end if
```

```
    end for loop
```

```
    end while loop
```

```
output  $\leftarrow \text{parent, distance}$ 
```



Depth-first search

```
all nodes  $\leftarrow \{dist_{start} \leftarrow \text{infinity}, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{false}\}$ 
```

```
start_node  $\leftarrow \{dist_{start} \leftarrow 0, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{true}\}$ 
```

```
visit_stack  $\leftarrow \text{start\_node}$ 
```

```
while visit_stack != empty && current_node != goal
```

```
    cur_node  $\leftarrow \text{pop(visit\_stack)}$  
```

```
    visitedcur_node  $\leftarrow \text{true}$ 
```

```
    for each nbr in not_visited(adjacent(cur_node))
```

```
        push(nbr to visit_stack)
```

```
        if distnbr > distcur_node + distance(nbr,cur_node)
```

```
            parentnbr  $\leftarrow \text{current\_node}$ 
```

```
            distnbr  $\leftarrow dist_{cur\_node} + distance(nbr,cur\_node)$ 
```

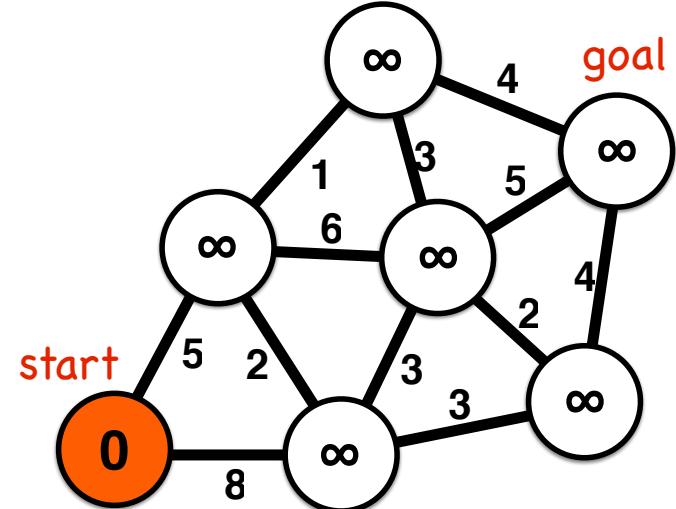
```
        end if
```

```
    end for loop
```

```
end while loop
```

```
output  $\leftarrow \text{parent, distance}$ 
```

PRIORITY:
MOST RECENT



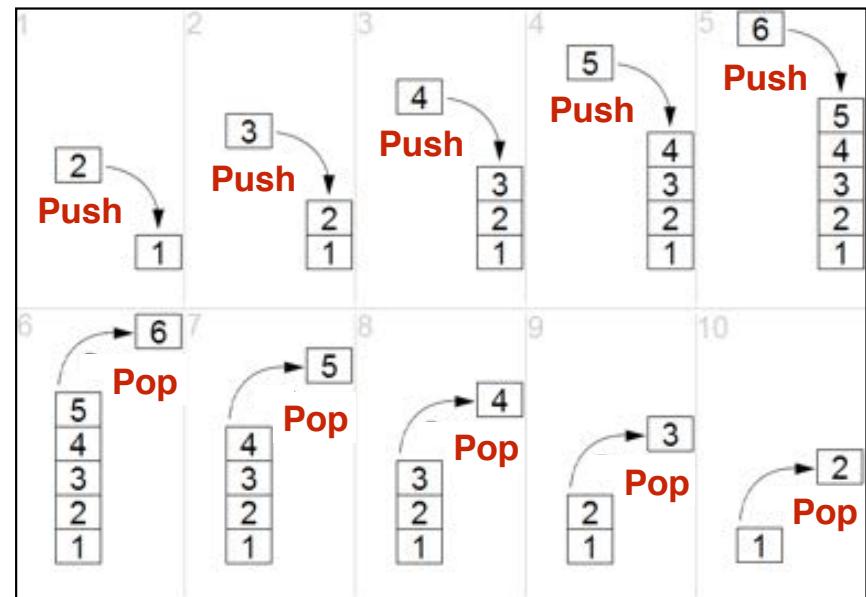
Stack data structure

A stack is a “last in, first out” (or LIFO) structure, with two operations:

push: to add an element to the top of the stack

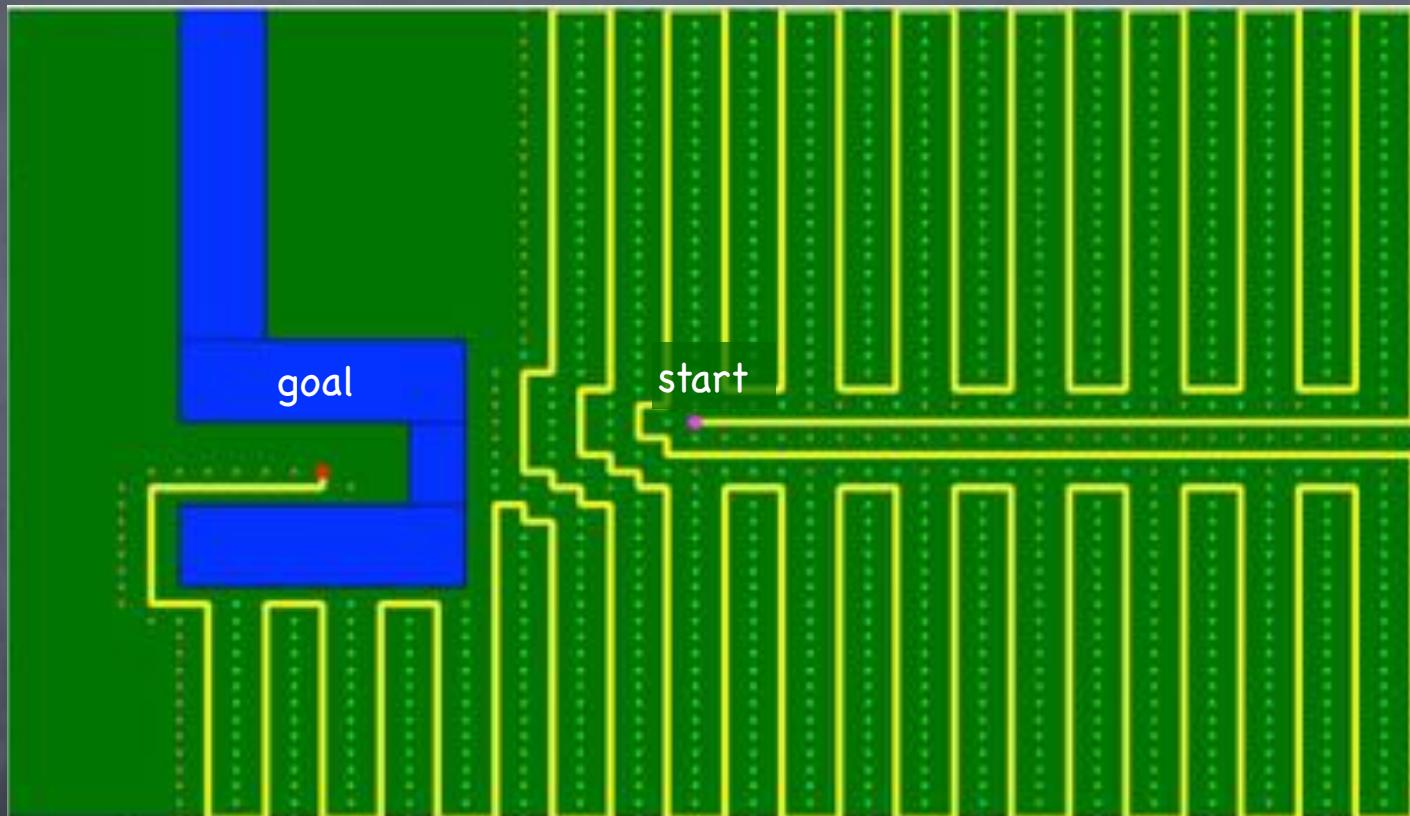
pop: to remove an element from the top of the stack

Stack example for reversing
the order of six elements



matlab example: DFS

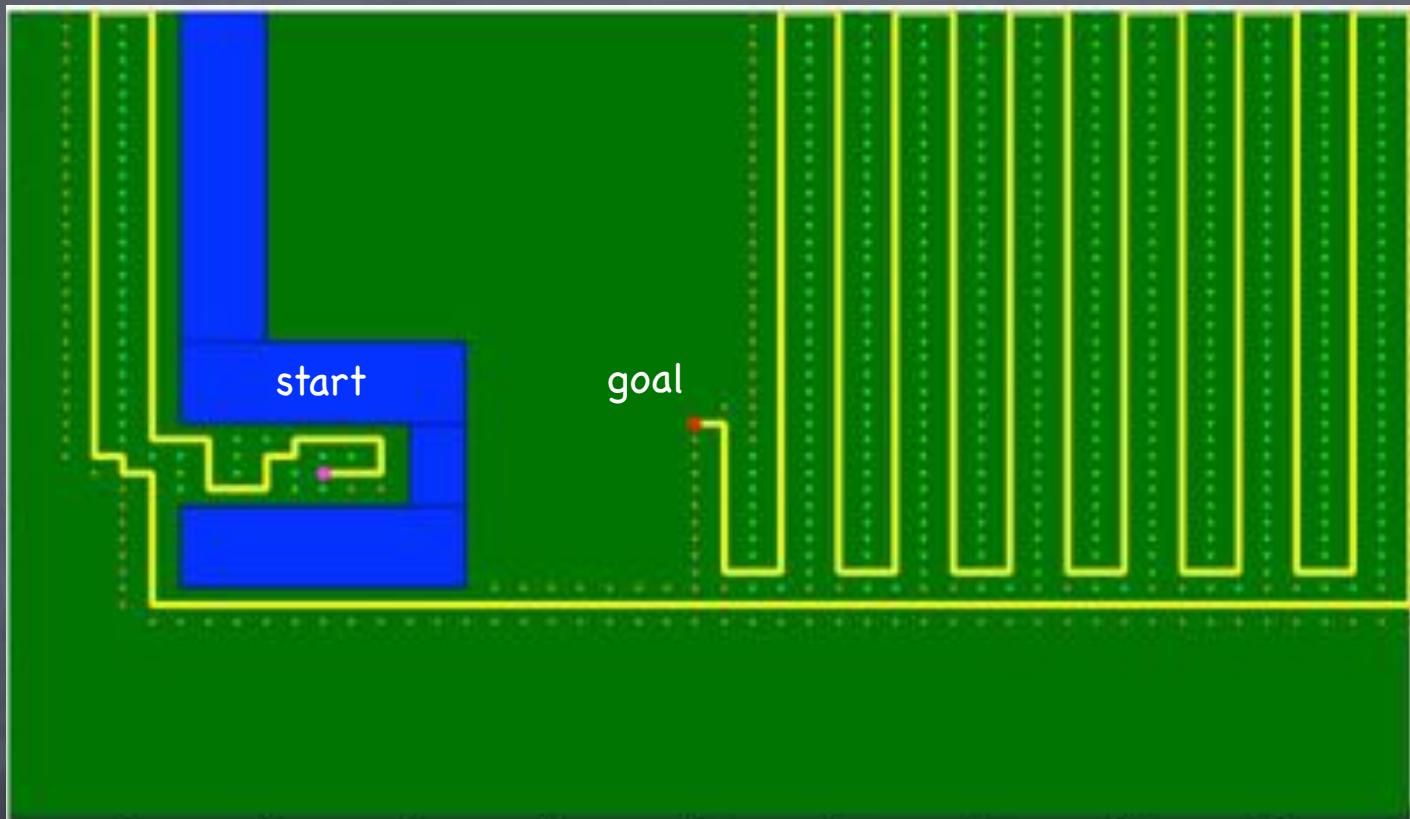
pathplan.m



visited locations in light green
4-connected grid, right visited first

matlab example: DFS

pathplan.m

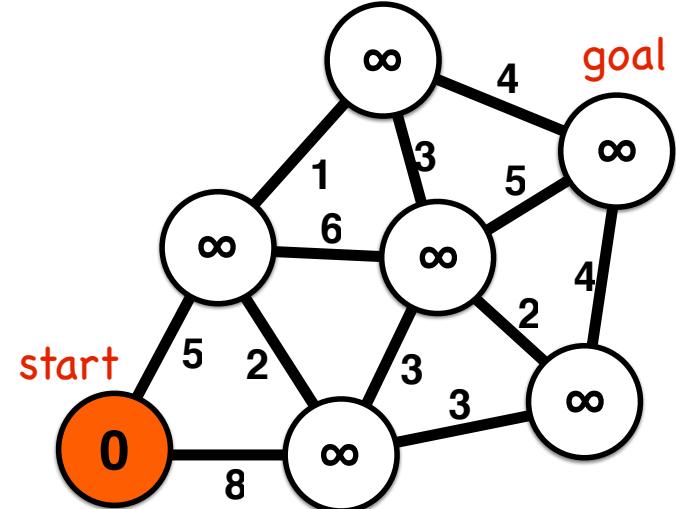


visited locations in light green
4-connected grid, right visited first

Breadth-first search

Search algorithm template

```
all nodes ← {diststart← infinity, parentstart ← none, visitedstart ← false}  
start_node ← {diststart← 0, parentstart ← none, visitedstart ← true}  
visit_list ← start_node  
while visit_list != empty && current_node != goal  
    cur_node ← highestPriority(visit_list)  
    visitedcur_node ← true  
    for each nbr in not_visited(adjacent(cur_node))  
        add(nbr to visit_list)  
        if distnbr > distcur_node + distance(nbr,cur_node)  
            parentnbr ← current_node  
            distnbr ← distcur_node + distance(nbr,cur_node)  
        end if  
    end for loop  
  end while loop  
output ← parent, distance
```



Breadth-first search

```
all nodes  $\leftarrow \{dist_{start} \leftarrow \text{infinity}, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{false}\}$ 
```

```
start_node  $\leftarrow \{dist_{start} \leftarrow 0, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{true}\}$ 
```

```
visit_queue  $\leftarrow$  start_node
```

```
while visit_queue != empty && current_node != goal
```

```
    cur_node  $\leftarrow$  dequeue(visit_queue) 
```

PRIORITY:

```
    visitedcur_node  $\leftarrow$  true
```

LEAST RECENT

```
    for each nbr in not_visited(adjacent(cur_node))
```

```
        enqueue(nbr to visit_queue)
```

```
        if distnbr > distcur_node + distance(nbr, cur_node)
```

```
            parentnbr  $\leftarrow$  current_node
```

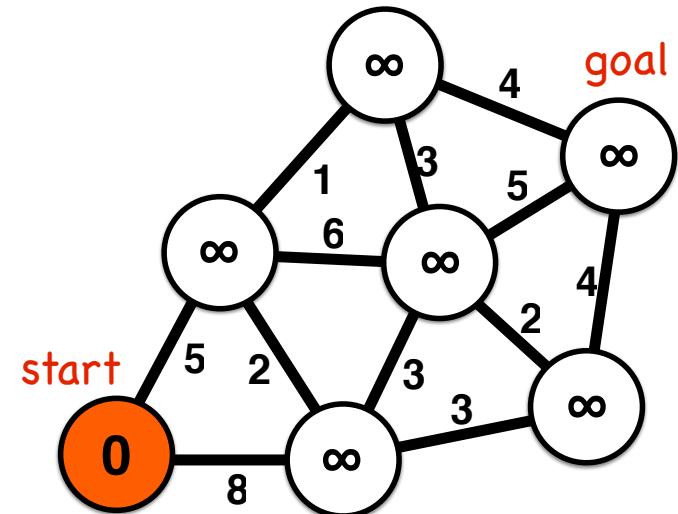
```
            distnbr  $\leftarrow$  distcur_node + distance(nbr, cur_node)
```

```
        end if
```

```
    end for loop
```

```
end while loop
```

```
output  $\leftarrow$  parent, distance
```

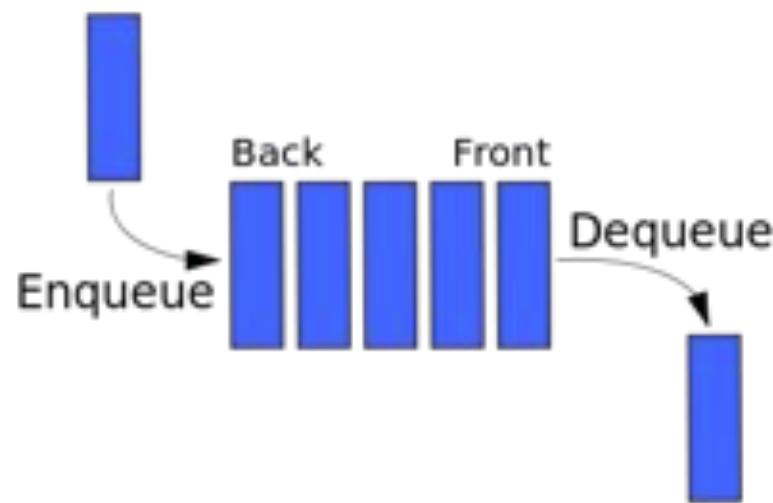


Queue data structure

A queue is a “first in, first out” (or FIFO) structure, with two operations

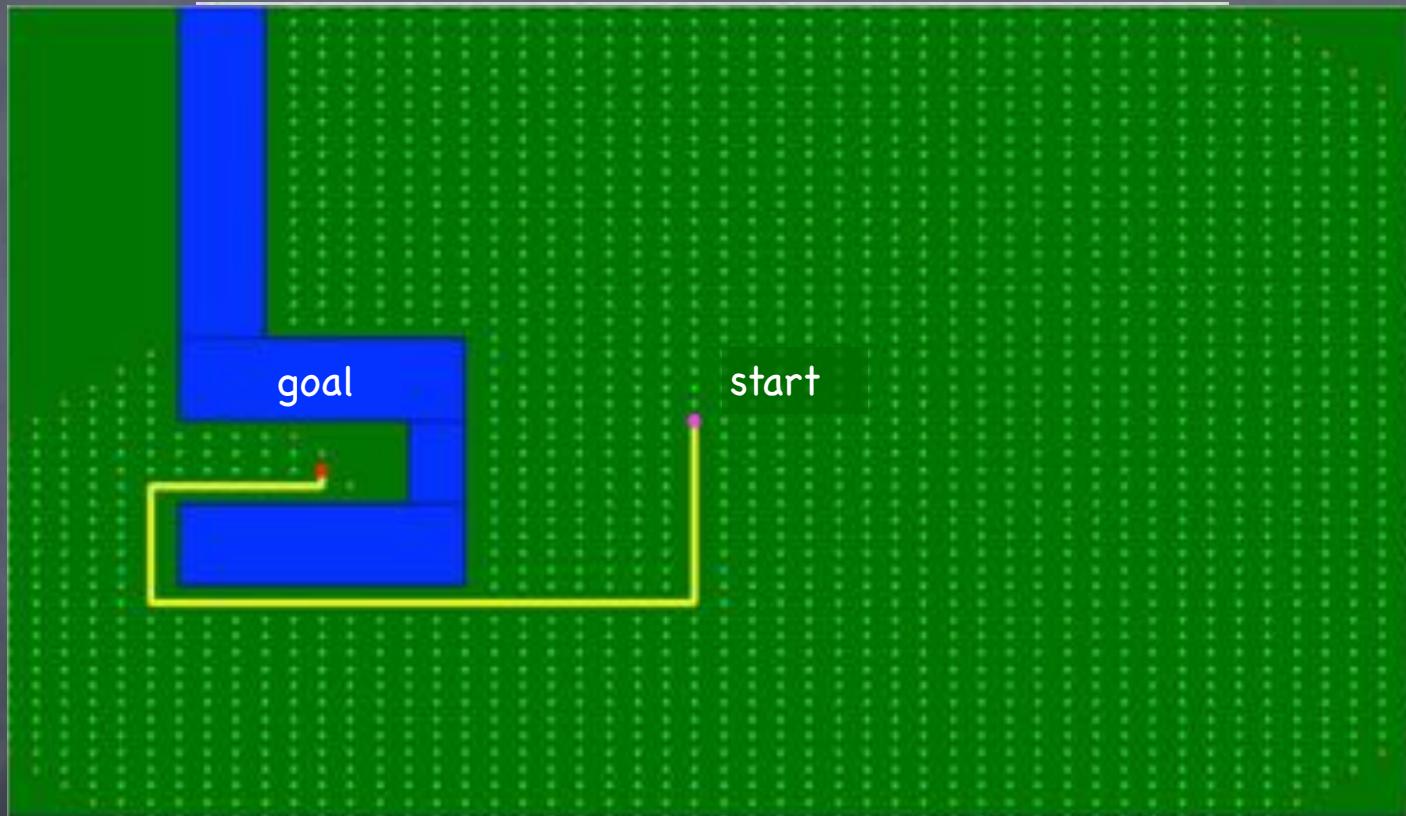
enqueue: to add an element to the back of the stack

dequeue: to remove an element from the front of the stack



matlab example: BFS

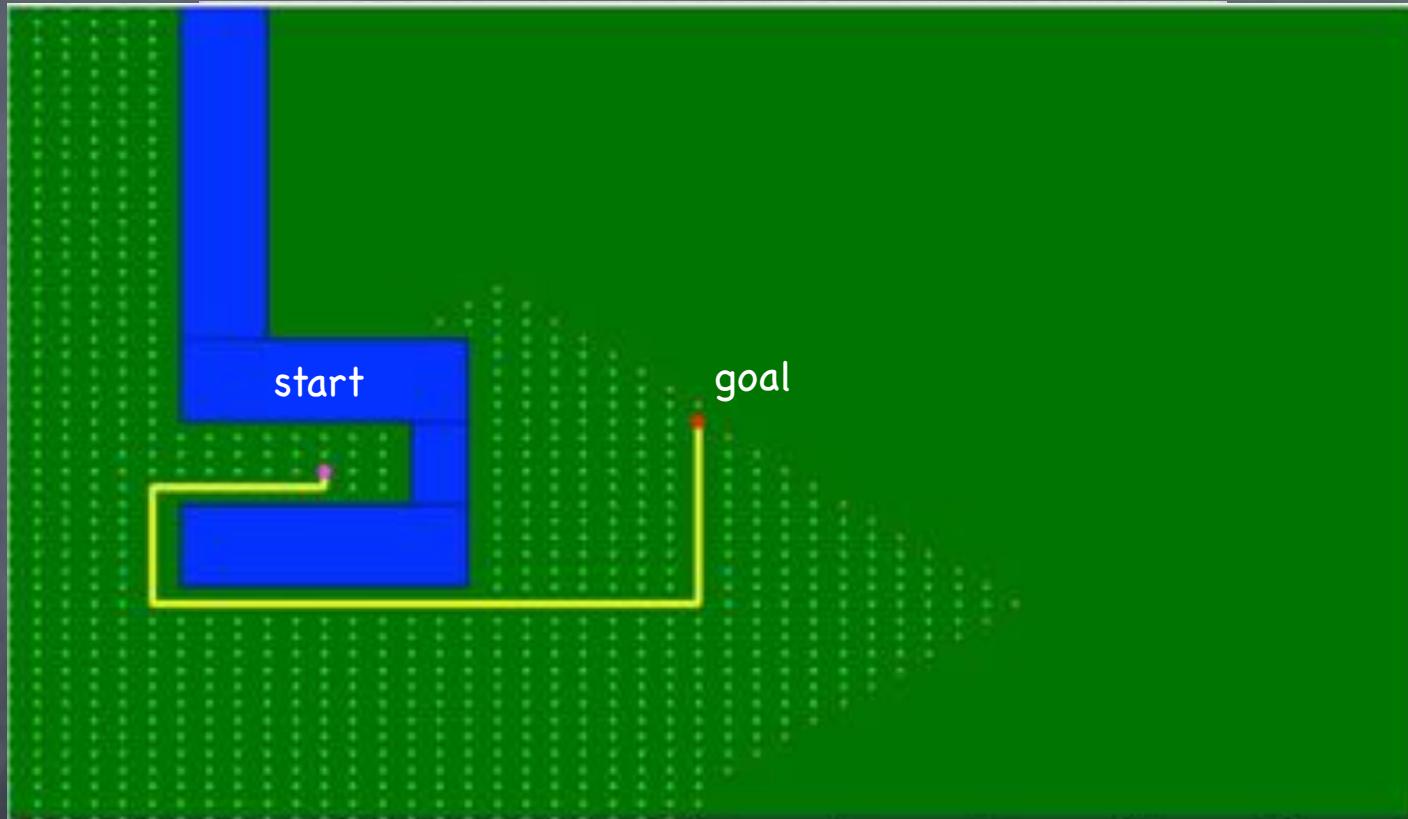
pathplan.m



visited locations in light green
4-connected grid, right visited first

matlab example: BFS

pathplan.m



visited locations in light green
4-connected grid, right visited first

Dijkstra shortest path

Search algorithm template

```
all nodes  $\leftarrow \{dist_{start} \leftarrow \text{infinity}, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{false}\}$ 
start_node  $\leftarrow \{dist_{start} \leftarrow 0, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{true}\}$ 
visit_list  $\leftarrow \text{start\_node}$ 
```

```
while visit_list != empty && current_node != goal
```

```
    cur_node  $\leftarrow \text{highestPriority}(\text{visit\_list})$ 
```

```
    visitedcur_node  $\leftarrow \text{true}$ 
```

```
    for each nbr in not_visited(adjacent(cur_node))
```

```
        add(nbr to visit_list)
```

```
        if distnbr > distcur_node + distance(nbr,cur_node)
```

```
            parentnbr  $\leftarrow \text{current\_node}$ 
```

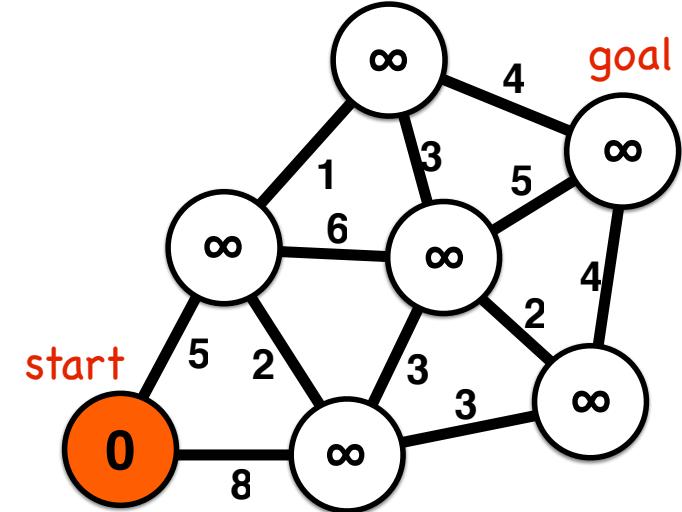
```
            distnbr  $\leftarrow dist_{cur\_node} + distance(nbr,cur\_node)$ 
```

```
        end if
```

```
    end for loop
```

```
    end while loop
```

```
output  $\leftarrow \text{parent, distance}$ 
```



Dijkstra shortest path algorithm

```
all nodes  $\leftarrow \{dist_{start} \leftarrow \text{infinity}, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{false}\}$ 
```

```
start_node  $\leftarrow \{dist_{start} \leftarrow 0, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{true}\}$ 
```

```
visit_queue  $\leftarrow$  start_node
```

```
while visit_queue != empty && current_node != goal
```

PRIORITY:

```
    cur_node  $\leftarrow \text{min\_distance(visit\_queue)}$  MINIMUM ROUTE DISTANCE  
FROM START
```

```
    visitedcur_node  $\leftarrow \text{true}$ 
```

```
    for each nbr in not_visited(adjacent(cur_node))
```

```
        enqueue(nbr to visit_queue)
```

```
        if distnbr > distcur_node + distance(nbr,cur_node)
```

```
            parentnbr  $\leftarrow$  current_node
```

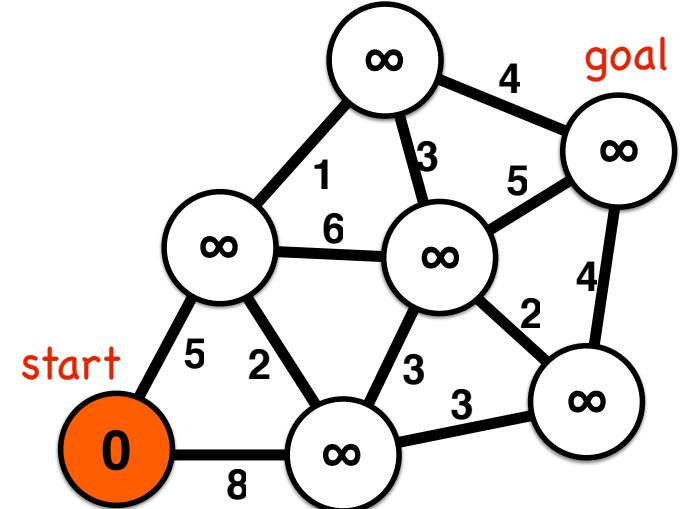
```
            distnbr  $\leftarrow$  distcur_node + distance(nbr,cur_node)
```

```
        end if
```

```
    end for loop
```

```
end while loop
```

```
output  $\leftarrow$  parent, distance
```

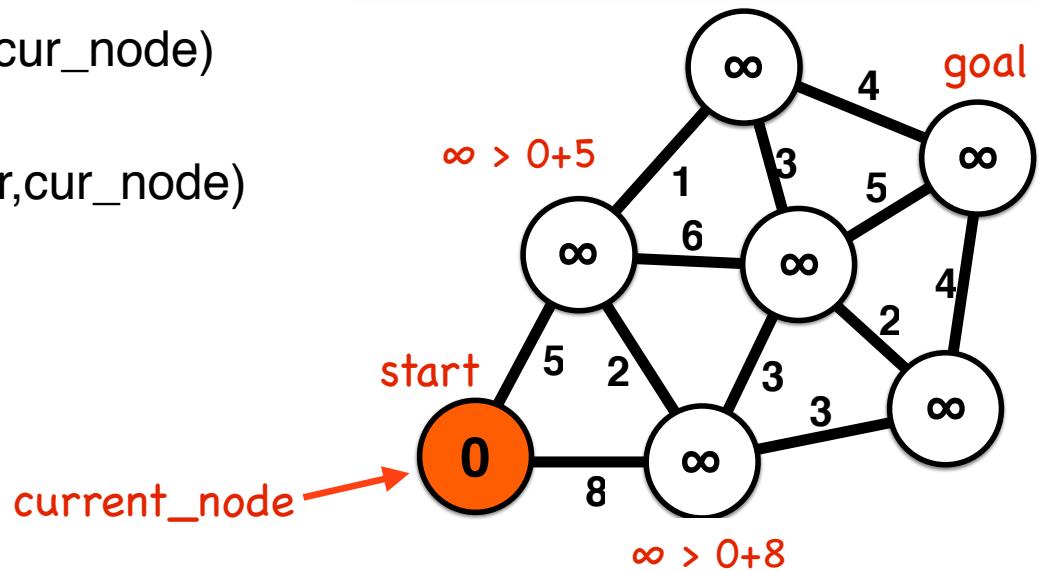


Dijkstra shortest path algorithm

```
all nodes  $\leftarrow \{dist_{start} \leftarrow \text{infinity}, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{false}\}$ 
start_node  $\leftarrow \{dist_{start} \leftarrow 0, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{true}\}$ 
visit_queue  $\leftarrow \text{start\_node}$ 

while visit_queue != empty && current_node != goal
    cur_node  $\leftarrow \text{min\_distance(visit\_queue)}$ 
    visitedcur_node  $\leftarrow \text{true}$ 
    for each nbr in not_visited(adjacent(cur_node))
        enqueue(nbr to visit_queue)
        if distnbr > distcur_node + distance(nbr,cur_node)
            parentnbr  $\leftarrow \text{current\_node}$ 
            distnbr  $\leftarrow dist_{cur\_node} + distance(nbr,cur\_node)$ 
        end if
    end for loop
end while loop
output  $\leftarrow \text{parent, distance}$ 
```

Dijkstra walkthrough

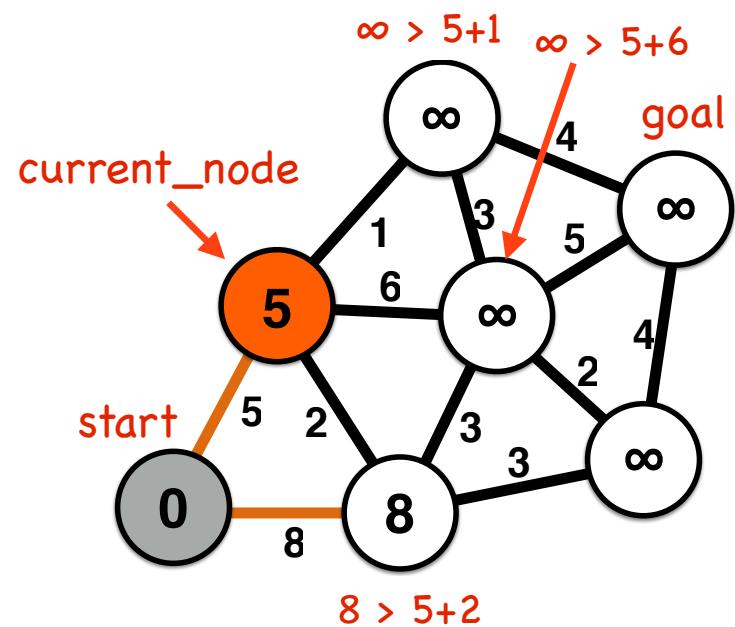


Dijkstra shortest path algorithm

```
all nodes  $\leftarrow \{dist_{start} \leftarrow \text{infinity}, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{false}\}$ 
start_node  $\leftarrow \{dist_{start} \leftarrow 0, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{true}\}$ 
visit_queue  $\leftarrow \text{start\_node}$ 

while visit_queue != empty && current_node != goal
    cur_node  $\leftarrow \text{min\_distance(visit\_queue)}$ 
    visitedcur_node  $\leftarrow \text{true}$ 
    for each nbr in not_visited(adjacent(cur_node))
        enqueue(nbr to visit_queue)
        if distnbr > distcur_node + distance(nbr,cur_node)
            parentnbr  $\leftarrow \text{current\_node}$ 
            distnbr  $\leftarrow dist_{cur\_node} + distance(nbr,cur\_node)$ 
        end if
    end for loop
end while loop
output  $\leftarrow \text{parent, distance}$ 
```

Dijkstra walkthrough

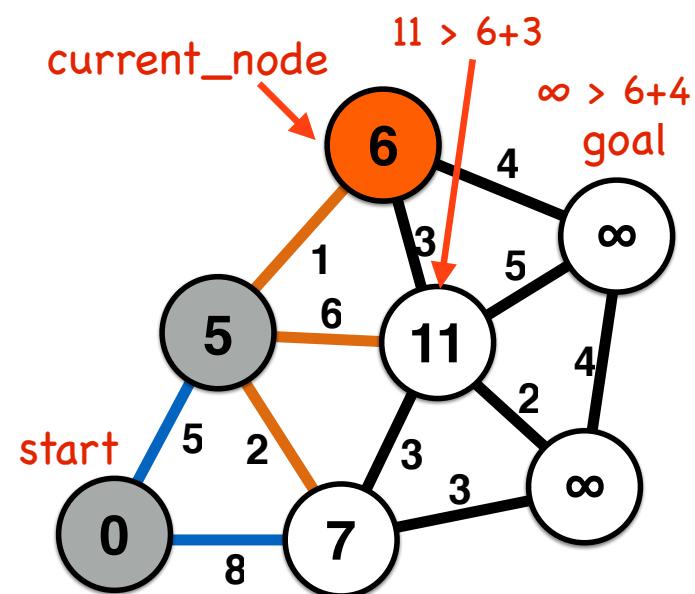


Dijkstra shortest path algorithm

```
all nodes  $\leftarrow \{dist_{start} \leftarrow \text{infinity}, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{false}\}$ 
start_node  $\leftarrow \{dist_{start} \leftarrow 0, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{true}\}$ 
visit_queue  $\leftarrow \text{start\_node}$ 

while visit_queue != empty && current_node != goal
    cur_node  $\leftarrow \text{min\_distance(visit\_queue)}$ 
    visitedcur_node  $\leftarrow \text{true}$ 
    for each nbr in not_visited(adjacent(cur_node))
        enqueue(nbr to visit_queue)
        if distnbr > distcur_node + distance(nbr,cur_node)
            parentnbr  $\leftarrow \text{current\_node}$ 
            distnbr  $\leftarrow dist_{cur\_node} + distance(nbr,cur\_node)$ 
        end if
    end for loop
end while loop
output  $\leftarrow \text{parent, distance}$ 
```

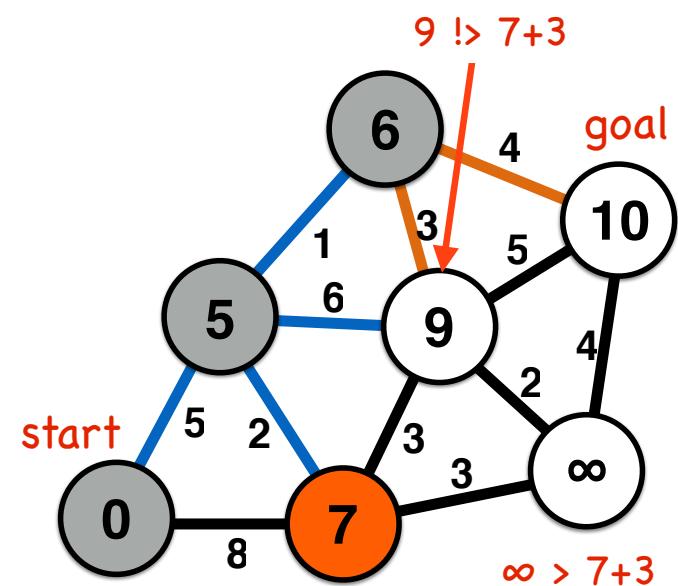
Dijkstra walkthrough



Dijkstra shortest path algorithm

```
all nodes  $\leftarrow \{dist_{start} \leftarrow \text{infinity}, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{false}\}$ 
start_node  $\leftarrow \{dist_{start} \leftarrow 0, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{true}\}$ 
visit_queue  $\leftarrow \text{start\_node}$ 

while visit_queue != empty && current_node != goal
    cur_node  $\leftarrow \text{min\_distance(visit\_queue)}$ 
    visitedcur_node  $\leftarrow \text{true}$ 
    for each nbr in not_visited(adjacent(cur_node))
        enqueue(nbr to visit_queue)
        if distnbr > distcur_node + distance(nbr,cur_node)
            parentnbr  $\leftarrow \text{current\_node}$ 
            distnbr  $\leftarrow dist_{cur\_node} + distance(nbr,cur\_node)$ 
        end if
    end for loop
end while loop
output  $\leftarrow \text{parent, distance}$ 
```



Dijkstra shortest path algorithm

```
all nodes  $\leftarrow \{dist_{start} \leftarrow \text{infinity}, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{false}\}$ 
start_node  $\leftarrow \{dist_{start} \leftarrow 0, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{true}\}$ 
visit_queue  $\leftarrow \text{start\_node}$ 
```

```
while visit_queue != empty && current_node != goal
```

```
    cur_node  $\leftarrow \text{min\_distance(visit\_queue)}$ 
```

```
    visitedcur_node  $\leftarrow \text{true}$ 
```

```
    for each nbr in not_visited(adjacent(cur_node))
```

```
        enqueue(nbr to visit_queue)
```

```
        if distnbr > distcur_node + distance(nbr,cur_node)
```

```
            parentnbr  $\leftarrow \text{current\_node}$ 
```

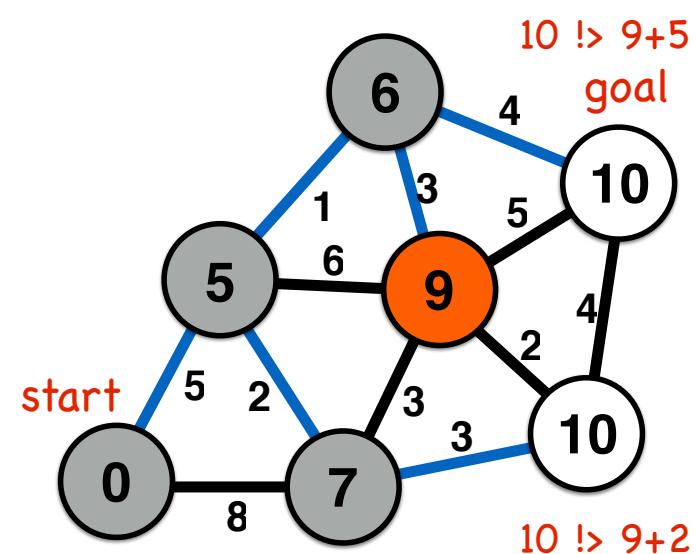
```
            distnbr  $\leftarrow dist_{cur\_node} + distance(nbr,cur\_node)$ 
```

```
        end if
```

```
    end for loop
```

```
    end while loop
```

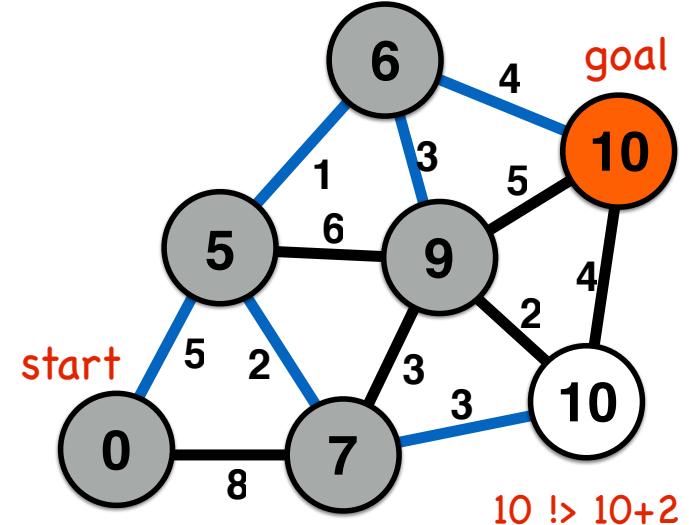
```
output  $\leftarrow \text{parent, distance}$ 
```



Dijkstra shortest path algorithm

```
all nodes  $\leftarrow \{dist_{start} \leftarrow \text{infinity}, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{false}\}$ 
start_node  $\leftarrow \{dist_{start} \leftarrow 0, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{true}\}$ 
visit_queue  $\leftarrow \text{start\_node}$ 

while visit_queue != empty && current_node != goal
    cur_node  $\leftarrow \text{min\_distance(visit\_queue)}$ 
    visitedcur_node  $\leftarrow \text{true}$ 
    for each nbr in not_visited(adjacent(cur_node))
        enqueue(nbr to visit_queue)
        if distnbr > distcur_node + distance(nbr,cur_node)
            parentnbr  $\leftarrow \text{current\_node}$ 
            distnbr  $\leftarrow dist_{cur\_node} + distance(nbr,cur\_node)$ 
        end if
    end for loop
end while loop
output  $\leftarrow \text{parent, distance}$ 
```



Dijkstra shortest path algorithm

```
all nodes  $\leftarrow \{dist_{start} \leftarrow \text{infinity}, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{false}\}$   
start_node  $\leftarrow \{dist_{start} \leftarrow 0, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{true}\}$   
visit_queue  $\leftarrow \text{start\_node}$ 
```

```
while visit_queue != empty && current_node != goal
```

```
    cur_node  $\leftarrow \text{min\_distance(visit\_queue)}$ 
```

```
    visitedcur_node  $\leftarrow \text{true}$ 
```

```
    for each nbr in not_visited(adjacent(cur_node))
```

```
        enqueue(nbr to visit_queue)
```

```
        if distnbr > distcur_node + distance(nbr,cur_node)
```

```
            parentnbr  $\leftarrow \text{current\_node}$ 
```

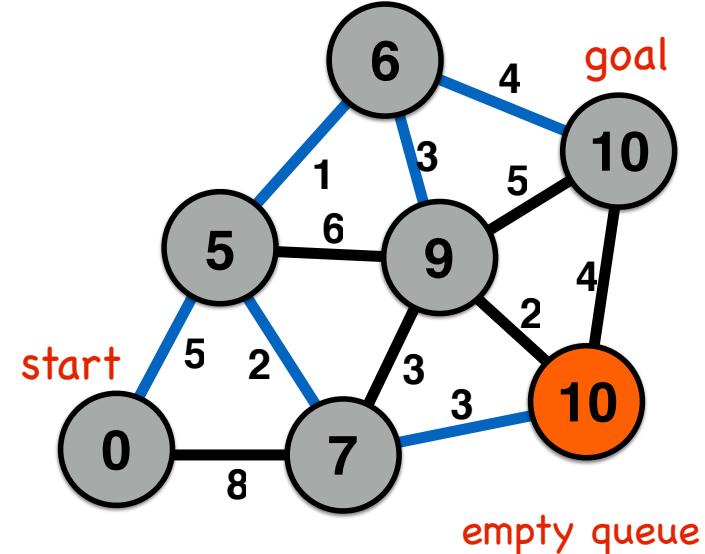
```
            distnbr  $\leftarrow dist_{cur\_node} + distance(nbr,cur\_node)$ 
```

```
        end if
```

```
    end for loop
```

```
    end while loop
```

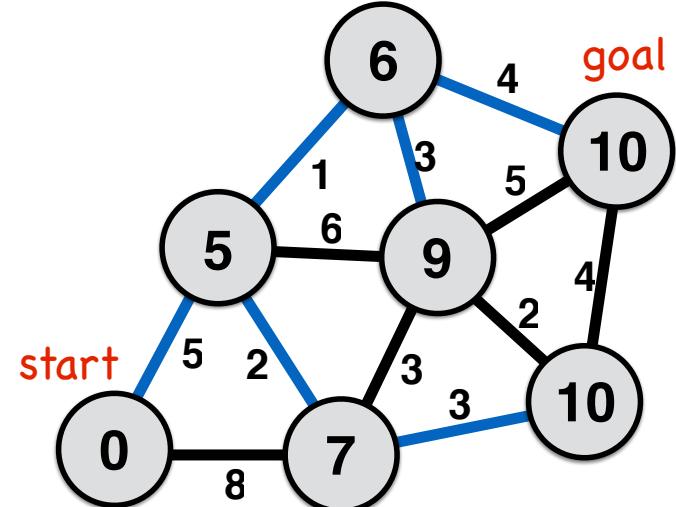
```
output  $\leftarrow \text{parent, distance}$ 
```



Dijkstra shortest path algorithm

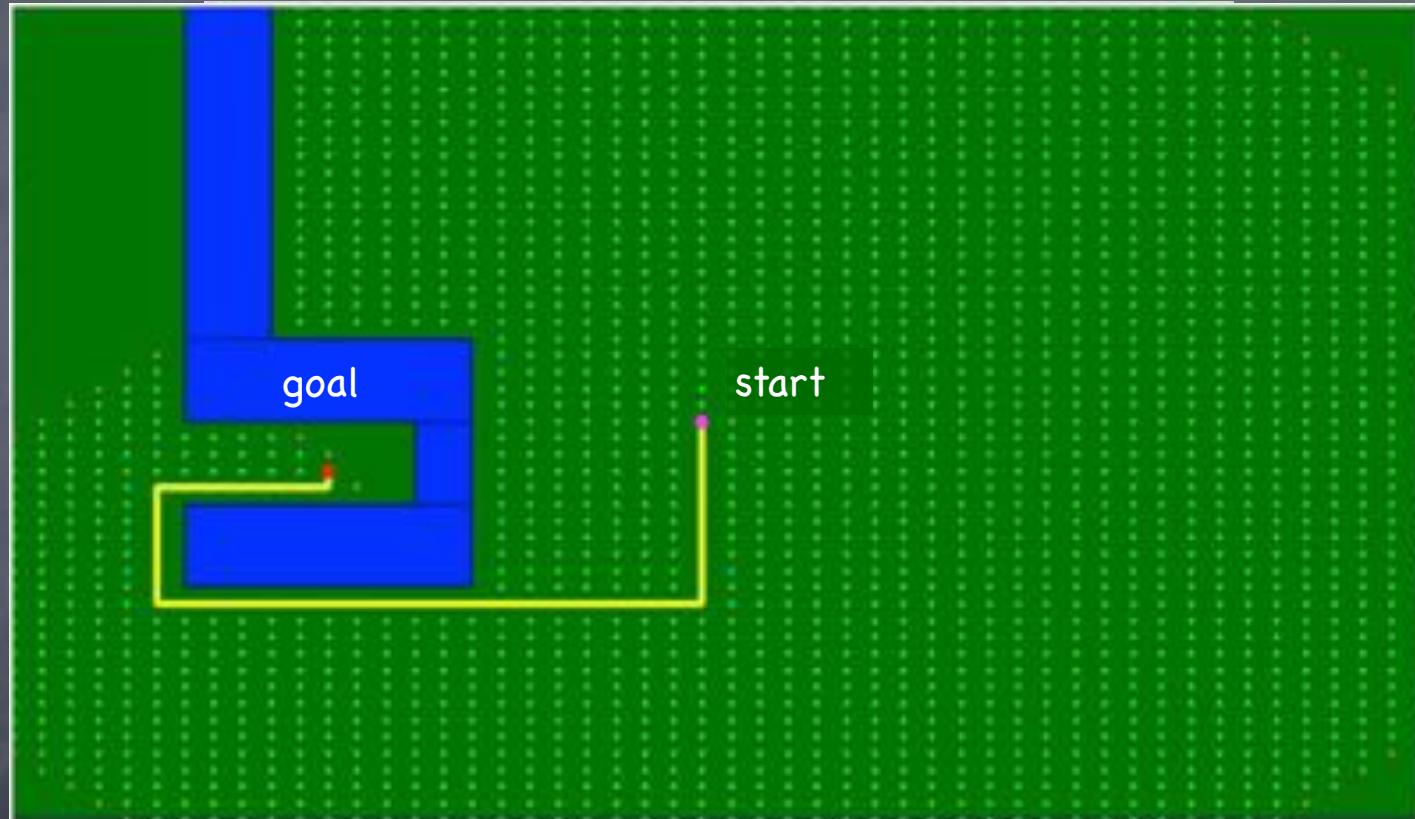
```
all nodes  $\leftarrow \{dist_{start} \leftarrow \text{infinity}, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{false}\}$ 
start_node  $\leftarrow \{dist_{start} \leftarrow 0, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{true}\}$ 
visit_queue  $\leftarrow \text{start\_node}$ 

while visit_queue != empty && current_node != goal
    cur_node  $\leftarrow \text{min\_distance(visit\_queue)}$ 
    visitedcur_node  $\leftarrow \text{true}$ 
    for each nbr in not_visited(adjacent(cur_node))
        enqueue(nbr to visit_queue)
        if distnbr > distcur_node + distance(nbr,cur_node)
            parentnbr  $\leftarrow \text{current\_node}$ 
            distnbr  $\leftarrow dist_{cur\_node} + distance(nbr,cur\_node)$ 
        end if
    end for loop
end while loop
output  $\leftarrow \text{parent, distance}$ 
```



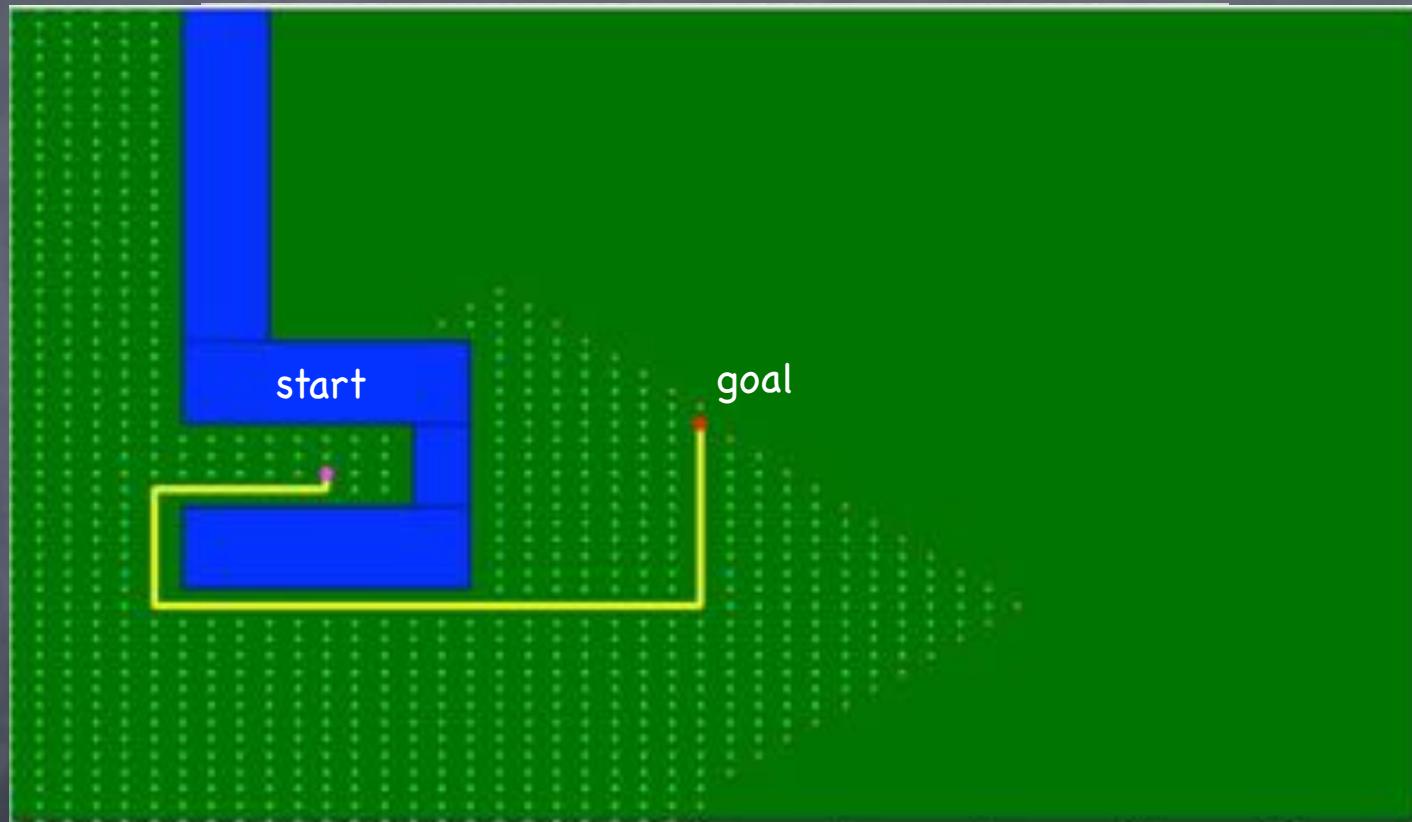
matlab example: Dijkstra

pathplan.m



matlab example: Dijkstra

pathplan.m

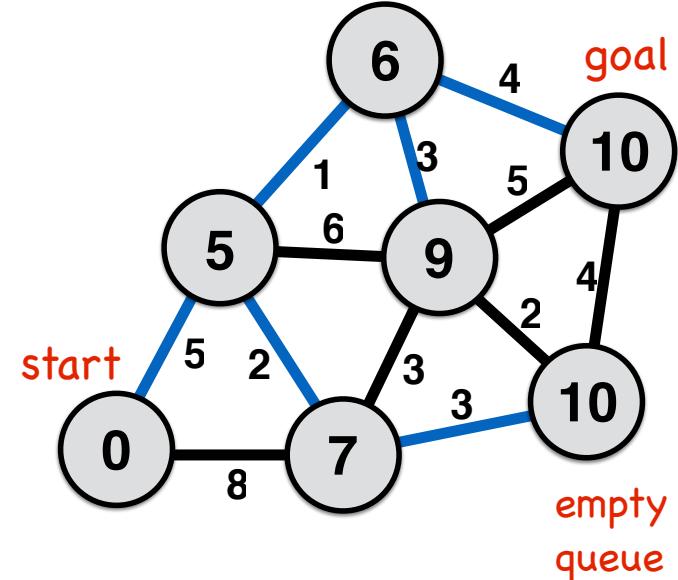


A-star shortest path

Dijkstra shortest path algorithm

```
all nodes  $\leftarrow \{dist_{start} \leftarrow \text{infinity}, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{false}\}$ 
start_node  $\leftarrow \{dist_{start} \leftarrow 0, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{true}\}$ 
visit_queue  $\leftarrow \text{start\_node}$ 

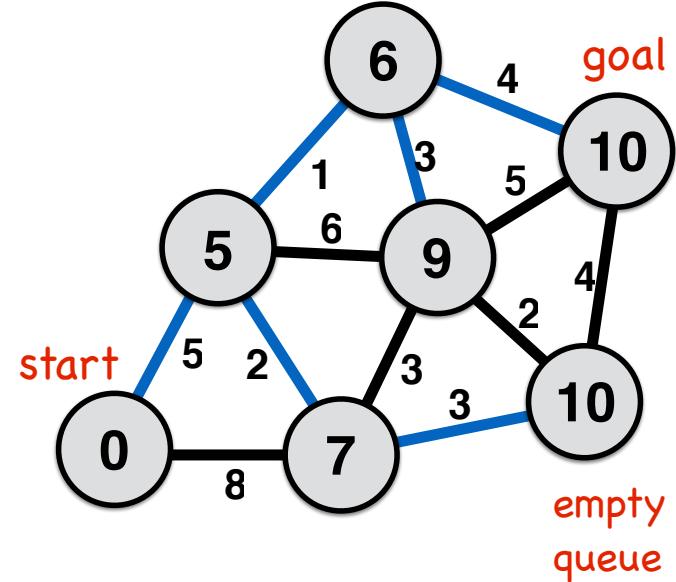
while visit_queue != empty && current_node != goal
    cur_node  $\leftarrow \text{min\_distance(visit\_queue)}$ 
    visitedcur_node  $\leftarrow \text{true}$ 
    for each nbr in not_visited(adjacent(cur_node))
        enqueue(nbr to visit_queue)
        if distnbr > distcur_node + distance(nbr,cur_node)
            parentnbr  $\leftarrow \text{current\_node}$ 
            distnbr  $\leftarrow dist_{cur\_node} + distance(nbr,cur\_node)$ 
        end if
    end for loop
end while loop
output  $\leftarrow \text{parent, distance}$ 
```



A-star shortest path algorithm

```
all nodes  $\leftarrow \{dist_{start} \leftarrow \text{infinity}, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{false}\}$ 
start_node  $\leftarrow \{dist_{start} \leftarrow 0, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{true}\}$ 
visit_queue  $\leftarrow \text{start\_node}$ 

while (visit_queue != empty)  $\&\&$  current_node != goal
    cur_node  $\leftarrow \text{dequeue(visit\_queue, f\_score)}$ 
    visitedcur_node  $\leftarrow \text{true}$ 
    for each nbr in not_visited(adjacent(cur_node))
        enqueue(nbr to visit_queue)
        if distnbr > distcur_node + distance(nbr,cur_node)
            parentnbr  $\leftarrow \text{current\_node}$ 
            distnbr  $\leftarrow dist_{cur\_node} + distance(nbr,cur\_node)$ 
            f_score  $\leftarrow distance_{nbr} + line\_distance_{nbr,goal}$ 
        end if
    end for loop
end while loop
output  $\leftarrow \text{parent, distance}$ 
```



A-star shortest path algorithm

```
all nodes  $\leftarrow \{dist_{start} \leftarrow \text{infinity}, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{false}\}$   
start_node  $\leftarrow \{dist_{start} \leftarrow 0, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{true}\}$   
visit_queue  $\leftarrow \text{start\_node}$ 
```

```
while (visit_queue != empty) && current_node != goal
```

```
    cur_node  $\leftarrow \text{dequeue(visit\_queue, f\_score)}$ 
```

```
    visitedcur_node  $\leftarrow \text{true}$ 
```

```
    for each nbr in not_visited(adjacent(cur_node))
```

```
        enqueue(nbr to visit_queue)
```

```
        if distnbr > distcur_node + distance(nbr,cur_node)
```

```
            parentnbr  $\leftarrow \text{current\_node}$ 
```

```
            distnbr  $\leftarrow dist_{cur\_node} + distance(nbr,cur\_node)$ 
```

```
            f_score  $\leftarrow distance_{nbr} + line\_distance_{nbr,goal}$ 
```

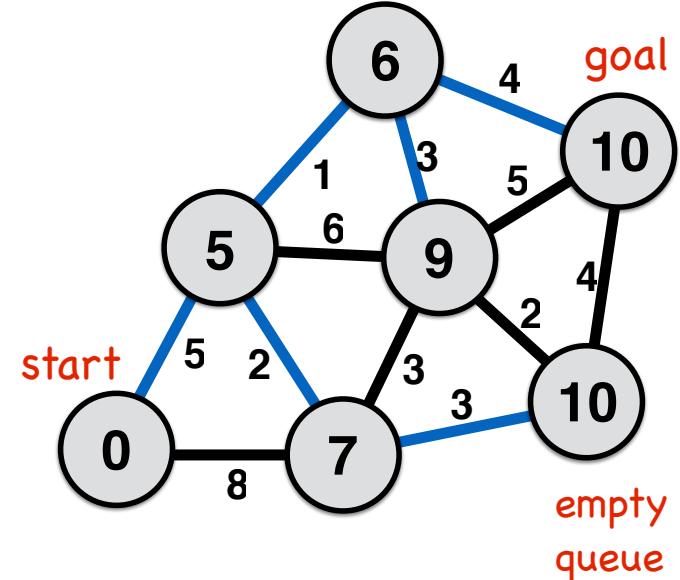
```
        end if
```

```
    end for loop
```

```
end while loop
```

```
output  $\leftarrow \text{parent, distance}$ 
```

priority queue wrt. f_score
(implement min binary heap)



g_score: distance along current path back to start

h_score: best possible distance to goal

A-star shortest path algorithm

```
all nodes  $\leftarrow \{dist_{start} \leftarrow \text{infinity}, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{false}\}$ 
start_node  $\leftarrow \{dist_{start} \leftarrow 0, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{true}\}$ 
visit_queue  $\leftarrow \text{start\_node}$ 
```

```
while (visit_queue != empty) && current_node != goal
    cur_node  $\leftarrow \text{dequeue(visit\_queue, f\_score)}$ 
```

priority queue wrt. f_score
(implement min binary heap)

```
visitedcur_node  $\leftarrow \text{true}$ 
```

```
for each nbr in not_visited(adjacent(cur_node))
```

```
    enqueue(nbr to visit_queue)
```

```
    if  $fScore_{nbr} > dist_{cur\_node} + dist_{cur\_node, nbr}$  then
```

```
        parentnbr  $\leftarrow \text{current\_node}$ 
```

```
        distnbr  $\leftarrow dist_{cur\_node} + \text{distance}(nbr, cur\_node)$ 
```

```
        f_score  $\leftarrow \text{distance}_{nbr} + \text{line\_distance}_{nbr, goal}$ 
```

```
    end if
```

```
end for loop
```

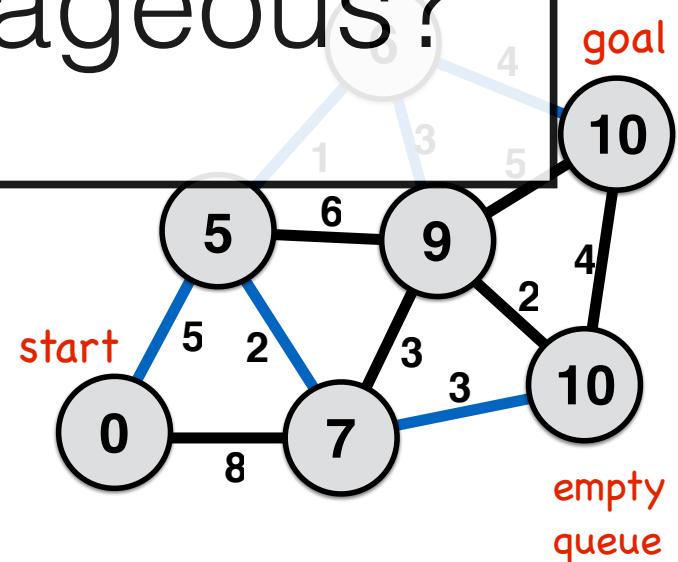
```
end while loop
```

```
output  $\leftarrow \text{parent, distance}$ 
```

g_score: distance along current path back to start

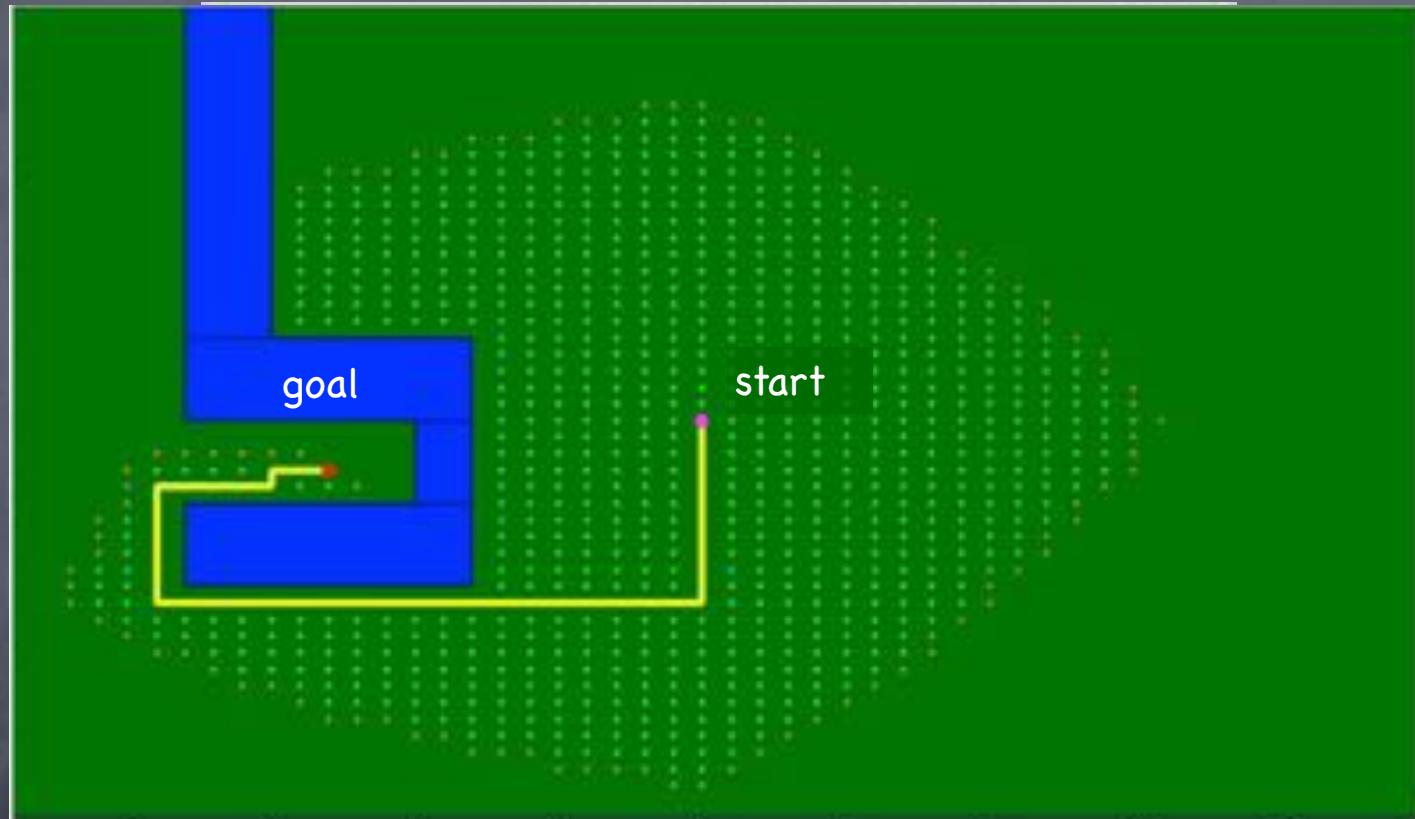
h_score: best possible distance to goal

Why is A-star advantageous?



matlab example: A*

pathplan.m



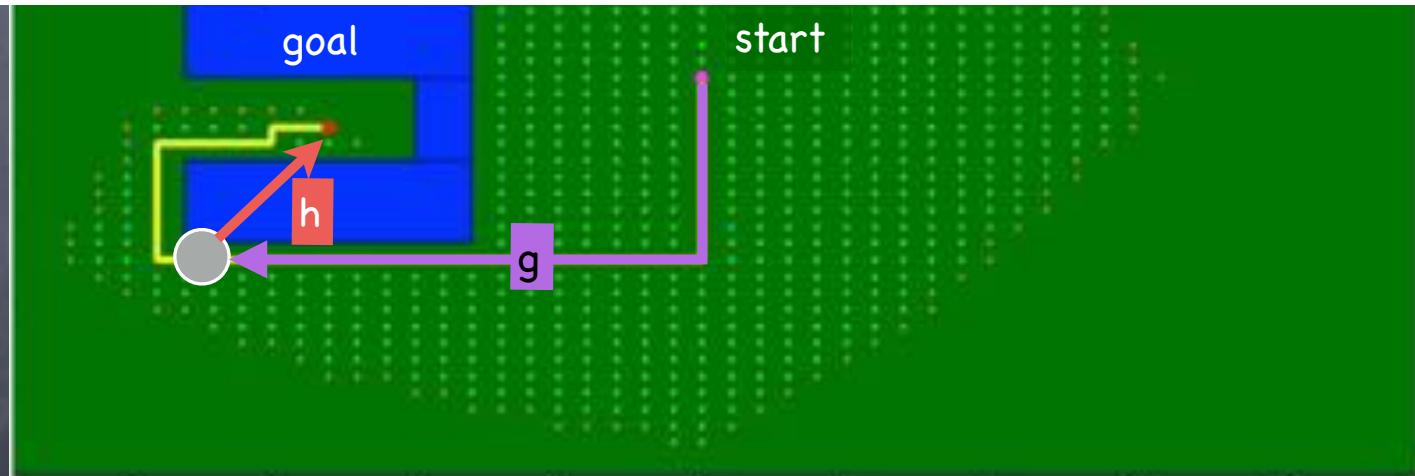
The straight line h_score is an admissible and consistent heuristic.

A heuristic function is **admissible** -> never overestimates the cost of reaching the goal.

Thus, $h_score(x) \leq$ the lowest possible cost from current location to the goal.

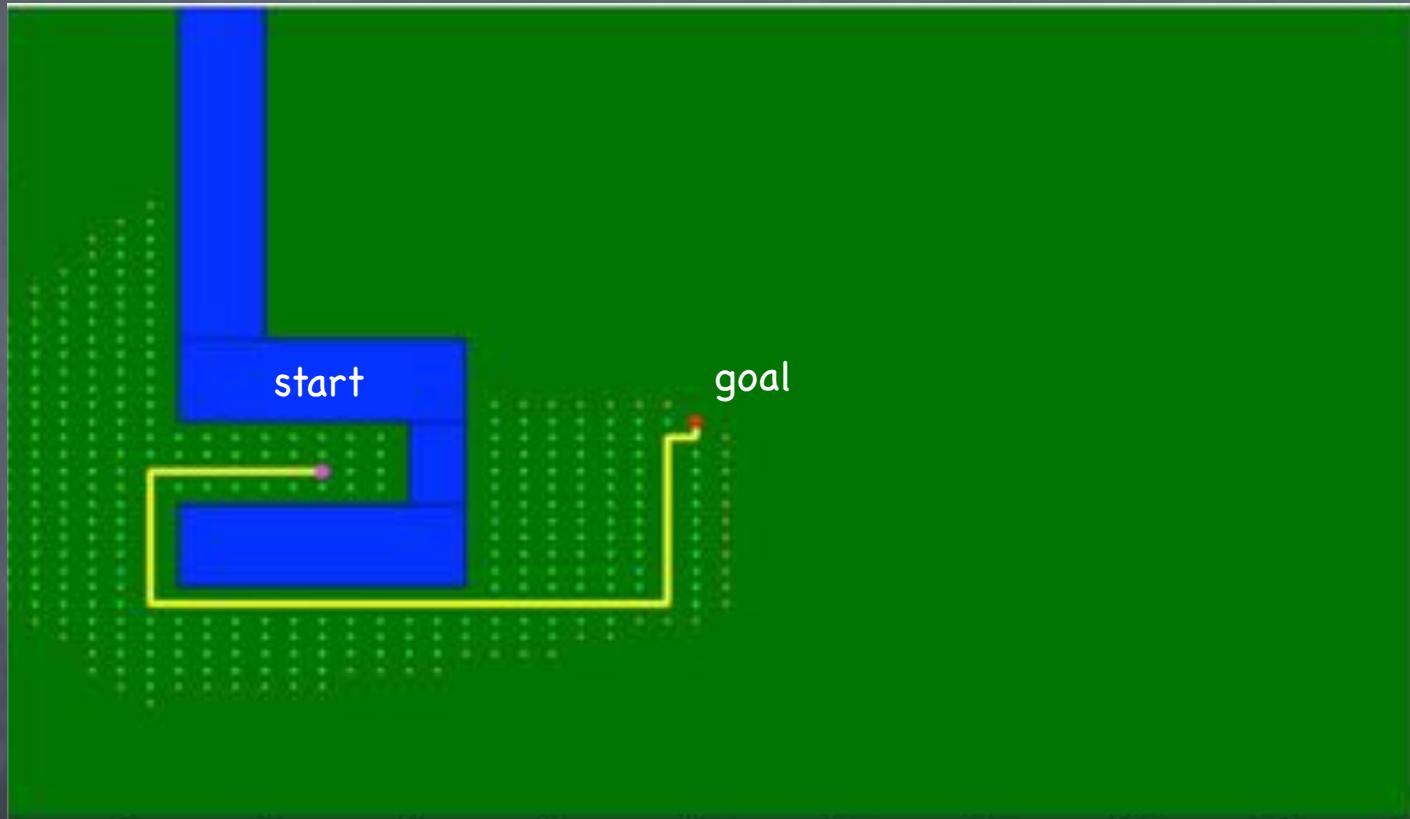
A heuristic function is **consistent** -> obeys the triangle inequality

Thus, the value of the $h_score(x) \leq \text{cost}(x, \text{action}, x') + h_score(x')$

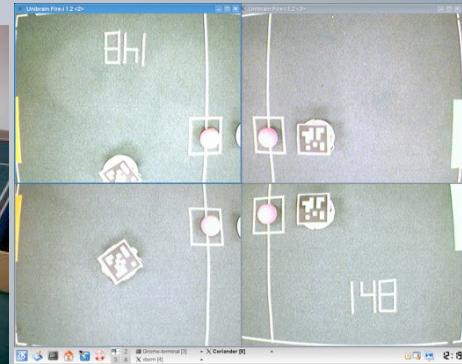


matlab example: A*

<http://www.cs.brown.edu/courses/cs148/pub/pathplan.m>



A* PATH PLANNING



Lisa Miller, <http://www.youtube.com/watch?v=2Z2RyeofsZg>

NAVIGATING AROUND A WALL



Lisa Miller

AVOIDING DEAD-ENDS



Lisa Miller

<http://www.youtube.com/watch?v=k6Kj4VjTKc8>

Suppose $f_score = g_score$

Greedy best-first search

A-star shortest path algorithm

```

all nodes ← {diststart ← infinity, parentstart ← none, visitedstart ← false}
start_node ← {diststart ← 0, parentstart ← none, visitedstart ← true}
visit_queue ← start_node

```

while (`visit_queue` != empty) && `current_node` != `goal`

`cur_node ← dequeue(visit_queue, f_score)`

`visitedcur node ← true`

for each nbr in not_visited(adjacent(cur_node))

enqueue(nbr to visit_queue)

if $\text{dist}_{\text{nbr}} > \text{dist}_{\text{cur_node}} + \text{distance}(\text{nbr}, \text{cur_node})$

`parentnbr ← current_node`

$\text{dist}_{\text{nbr}} \leftarrow \text{dist}_{\text{cur_node}} + \text{distance}(\text{nbr}, \text{cur_node})$

$f_score \leftarrow distance_{nbr} + line_distance_{nbr,goal}$

end if

end for loop

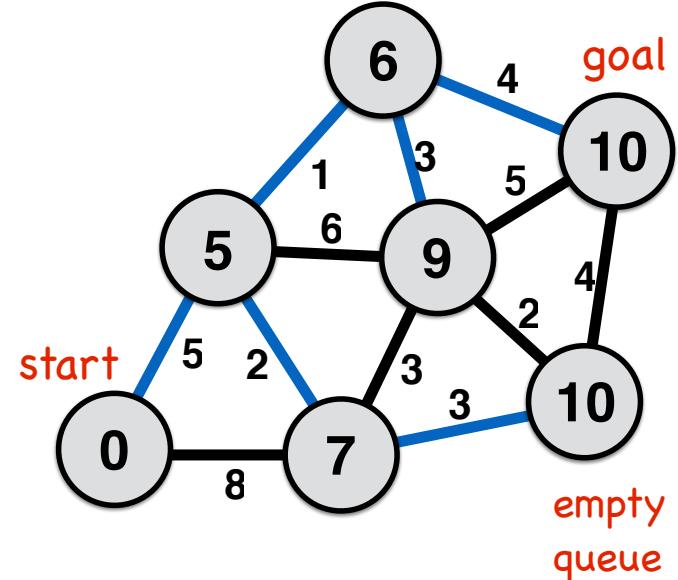
end while loop

output \leftarrow parent, distance

g_score: distance along current path back to start

h_score:
best possible
distance to goal

priority queue wrt. f_score
(implement min binary heap)



Greedy best-first search

```
all nodes  $\leftarrow \{dist_{start} \leftarrow \text{infinity}, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{false}\}$ 
```

```
start_node  $\leftarrow \{dist_{start} \leftarrow 0, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{true}\}$ 
```

```
visit_queue  $\leftarrow \text{start\_node}$ 
```

```
while (visit_queue != empty)  $\&\&$  current_node != goal
```

```
    cur_node  $\leftarrow \text{dequeue(visit\_queue, f\_score)}$ 
```

```
    visitedcur_node  $\leftarrow \text{true}$ 
```

```
    for each nbr in not_visited(adjacent(cur_node))
```

```
        enqueue(nbr to visit_queue)
```

```
        if distnbr > distcur_node + distance(nbr,cur_node)
```

```
            parentnbr  $\leftarrow \text{current\_node}$ 
```

```
            distnbr  $\leftarrow \text{dist}_{cur\_node} + \text{distance}(nbr,cur\_node)$ 
```

```
            f_score  $\leftarrow \text{distance}_{nbr} + \text{line\_distance}_{nbr,goal}$ 
```

```
        end if
```

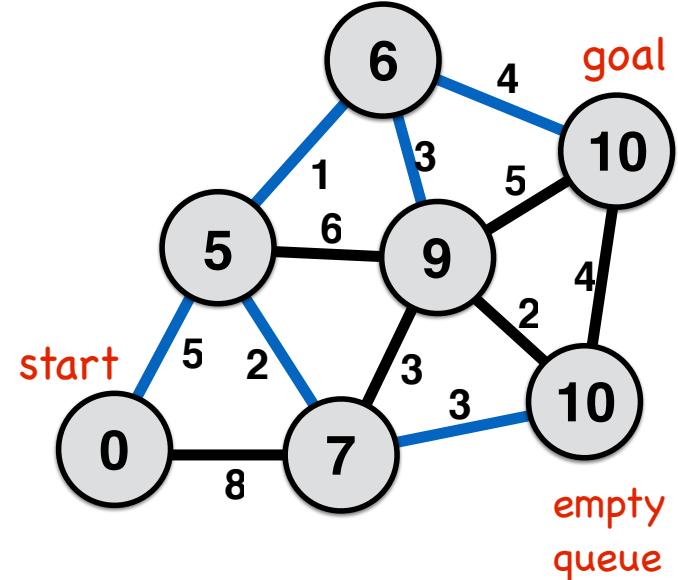
```
    end for loop
```

```
    end while loop
```

```
output  $\leftarrow \text{parent, distance}$ 
```

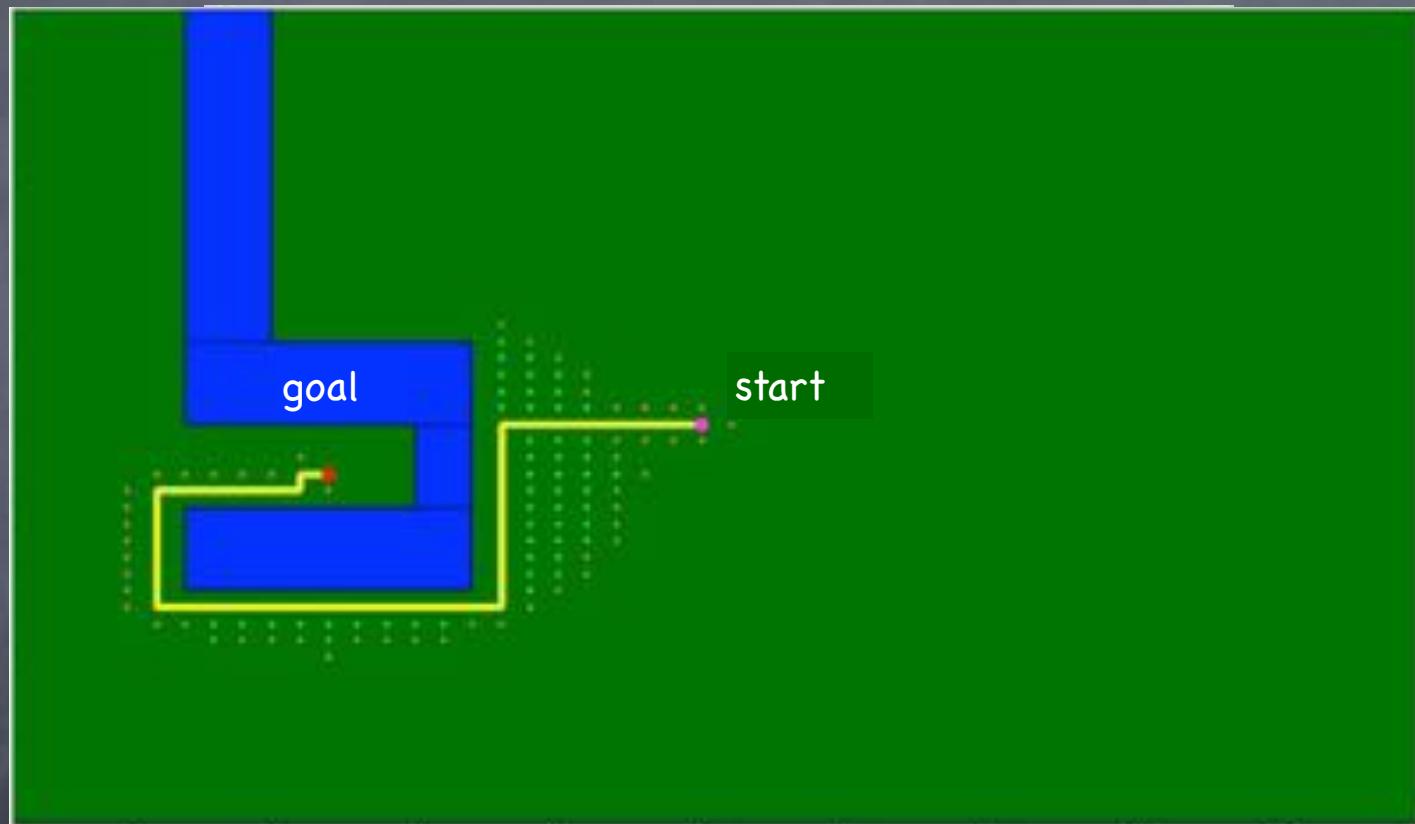
priority queue wrt. f_score
(implement min binary heap)

h_score:
best possible
distance to goal



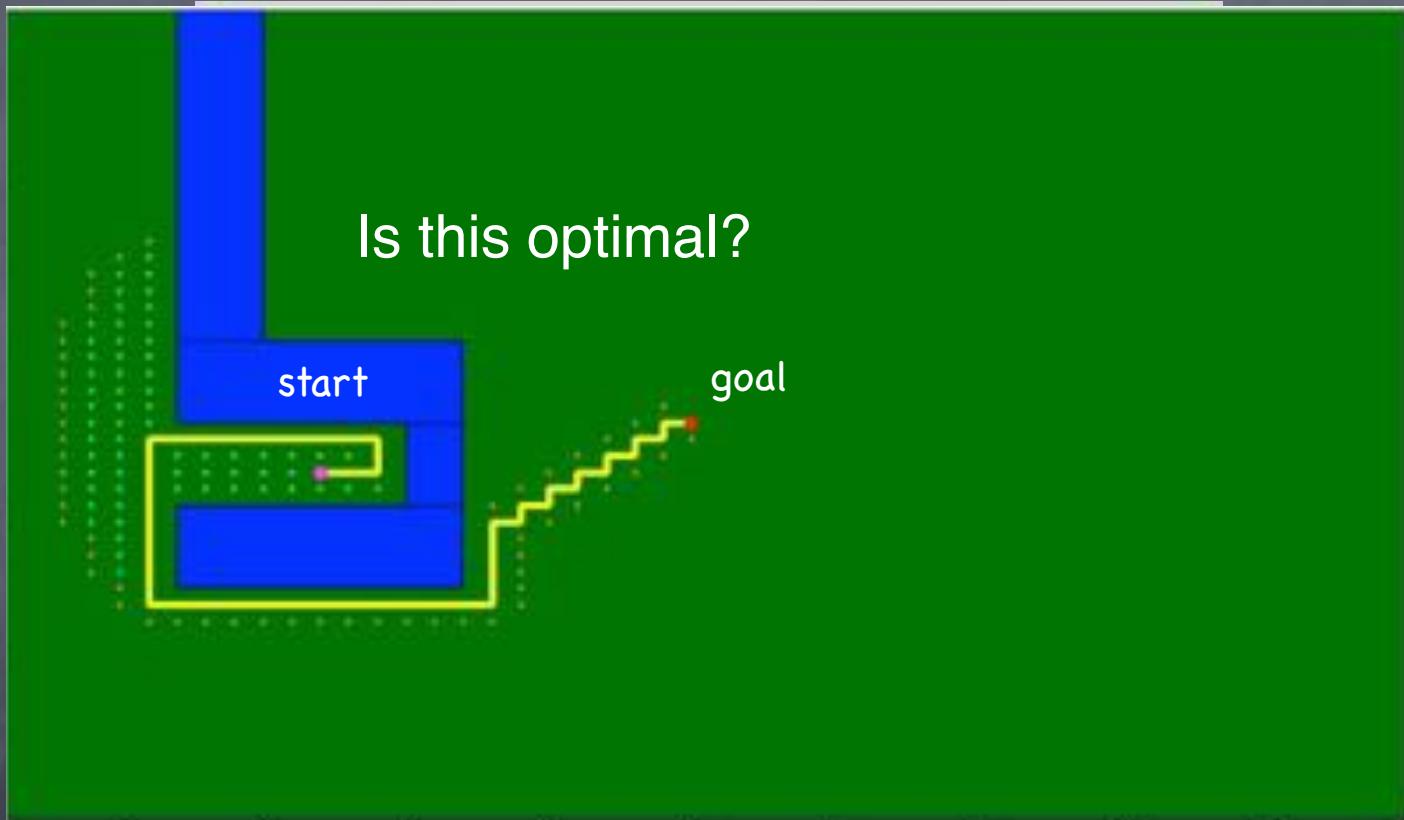
matlab example: gr.best-first

pathplan.m



matlab example: gr.best-first

pathplan.m

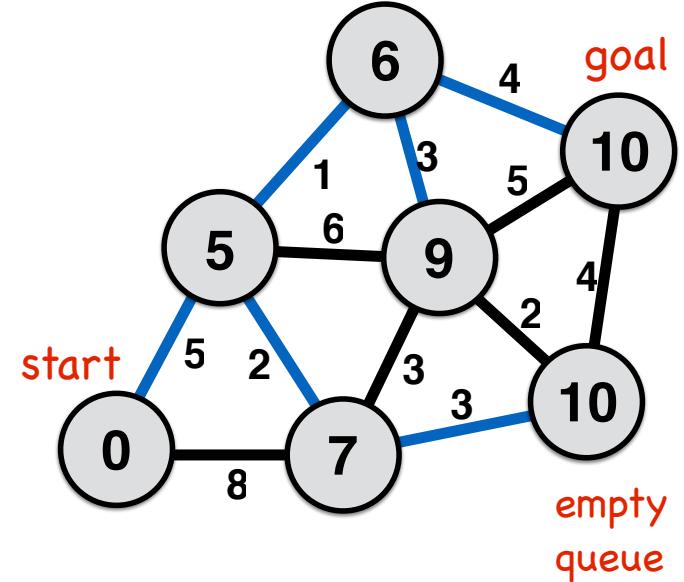


Heaps and Priority Queues

A-star shortest path algorithm

```
all nodes  $\leftarrow \{dist_{start} \leftarrow \text{infinity}, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{false}\}$ 
start_node  $\leftarrow \{dist_{start} \leftarrow 0, parent_{start} \leftarrow \text{none}, visited_{start} \leftarrow \text{true}\}$ 
visit_queue  $\leftarrow \text{start\_node}$ 

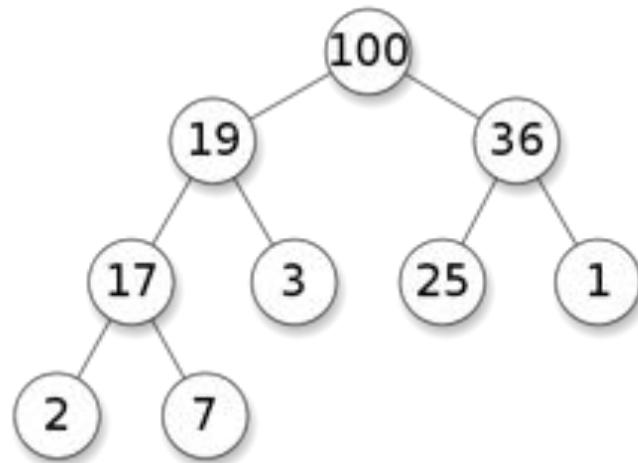
while (visit_queue != empty) && current_node != goal
    dequeue: cur_node  $\leftarrow f\text{-score}(visit\_queue)$  MIN BINARY HEAP FOR PRIORITY QUEUE
    visitedcur_node  $\leftarrow \text{true}$ 
    for each nbr in not_visited(adjacent(cur_node))
        enqueue: nbr to visit_queue
        if distnbr > distcur_node + distance(nbr,cur_node)
            parentnbr  $\leftarrow \text{current\_node}$ 
            distnbr  $\leftarrow dist_{cur\_node} + distance(nbr,cur\_node)$ 
            f_score  $\leftarrow distance_{nbr} + line\_distance_{nbr,goal}$ 
        end if
    end for loop
end while loop
output  $\leftarrow \text{parent, distance}$ 
```



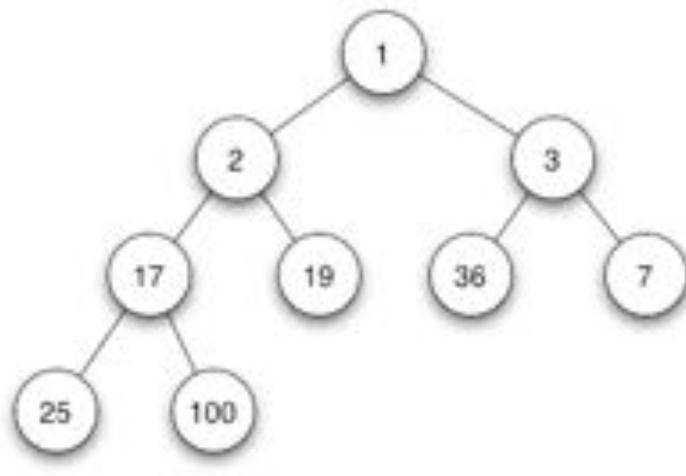
Binary Heaps

A heap is a tree-based data structure satisfying the heap property:
every element is greater (or less) than its children

Binary heaps allow nodes to have up to 2 children

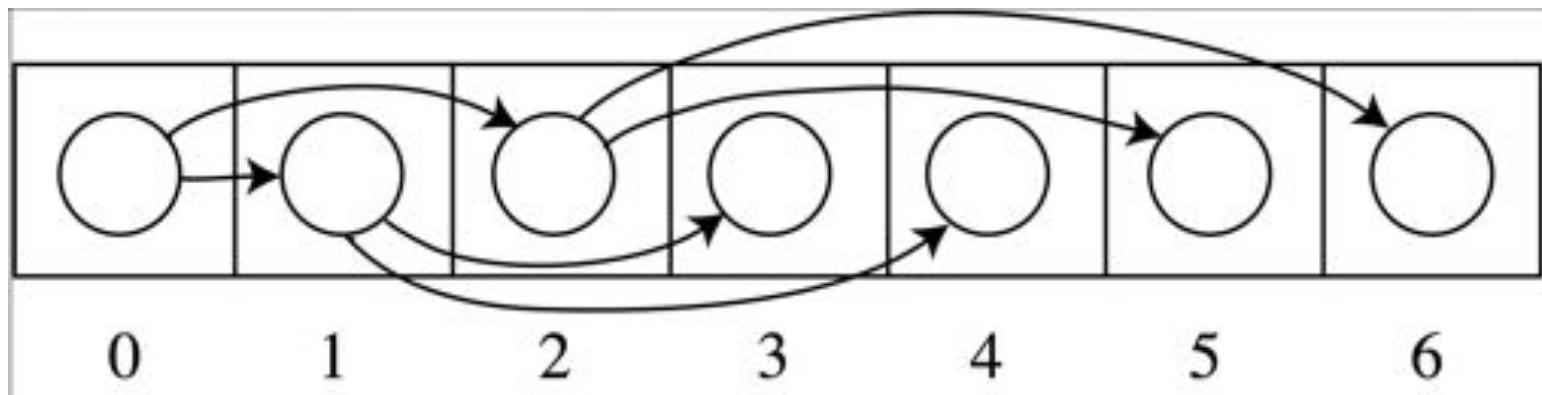


max heap



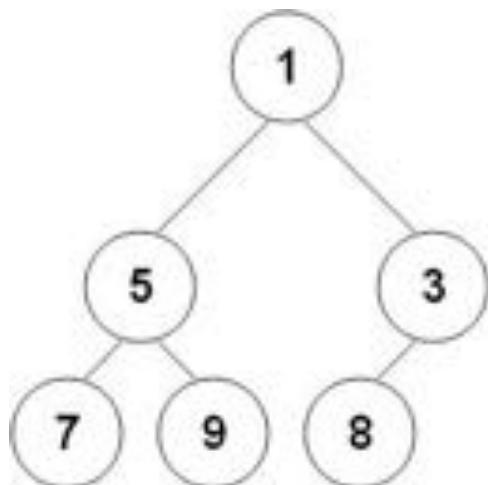
min heap

Heaps as arrays



- Heap element at array location i has
 - children at array locations $2i+1$ and $2i+2$
 - parent at $\text{floor}((i-1)/2)$

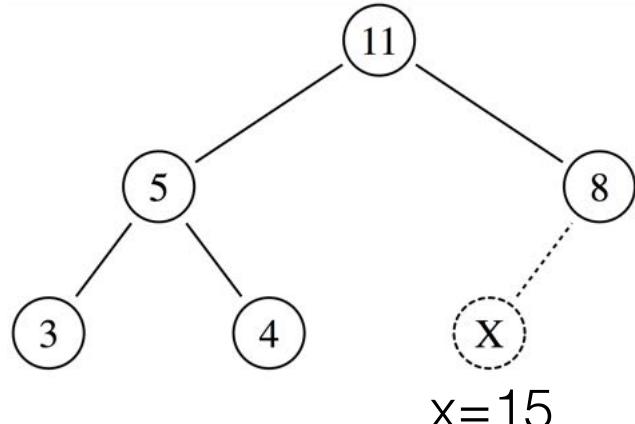
Heap array example



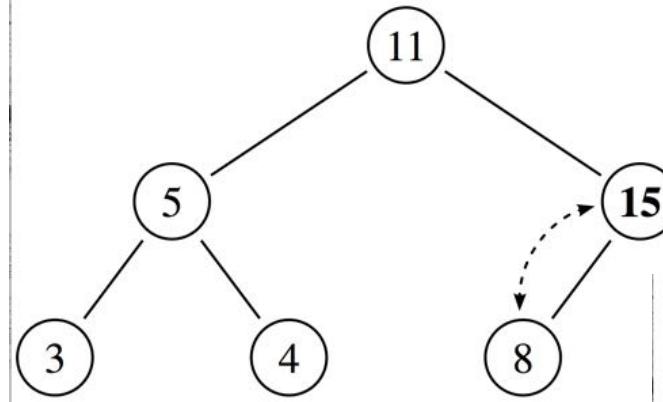
Node	1	5	3	7	9	8
Index	0	1	2	3	4	5

Heap operations: Insert

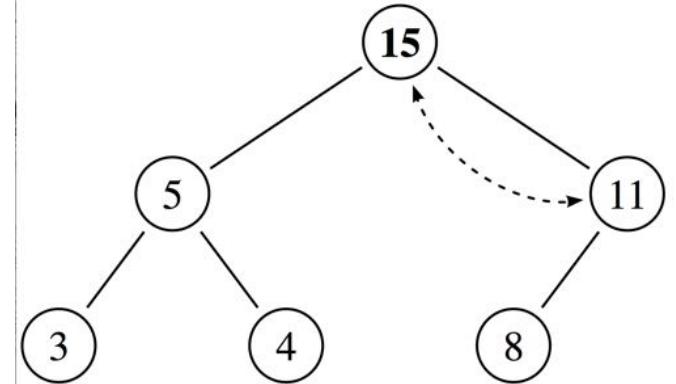
1) add new element to end of tree



2) swap with parent



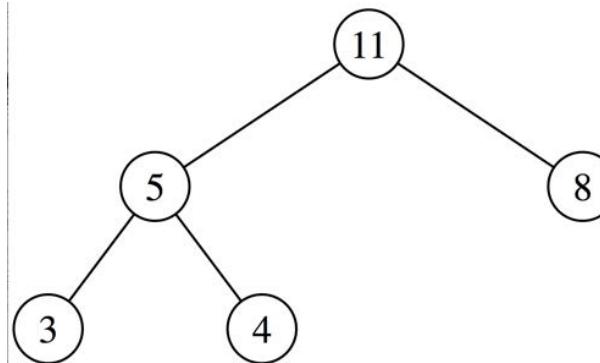
3) until heaped, do (2)



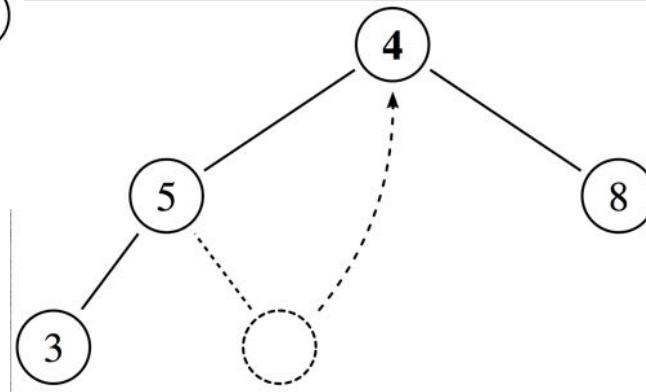
For priority queue, previously non-queued locations will be inserted with f_score priority

Heap operations: Extract

1) extract root element

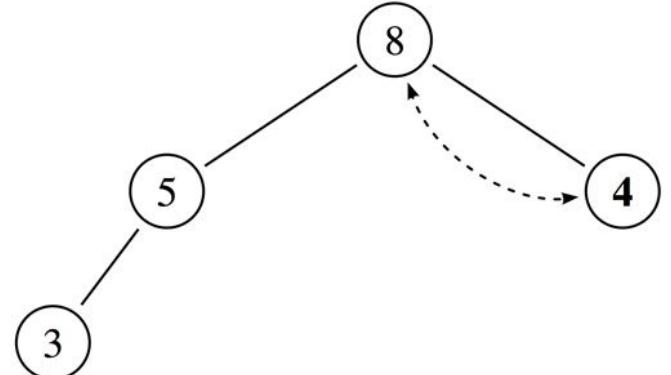


2) put last element at root



3) swap with higher priority child

4) until heaped, do (3)

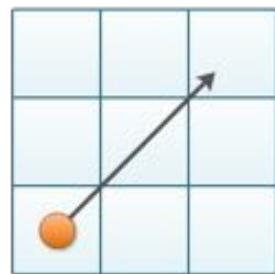


For priority queue, the root of the heap
will be the next node to visit

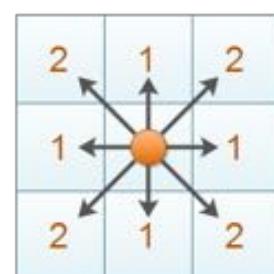
Considerations

- How many operations are needed for heap insertion and extraction?
- How much better is a min heap than an array wrt. # of operations?
- Can there be duplicate heap elements for the same robot pose?
- How should we measure distance on a uniform grid?
- Is a choice of distance measure both metric and admissible?

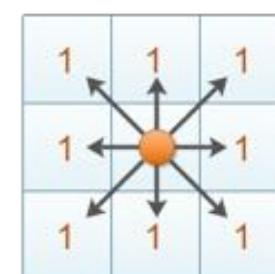
Euclidean Distance



Manhattan Distance



Chebyshev Distance



$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

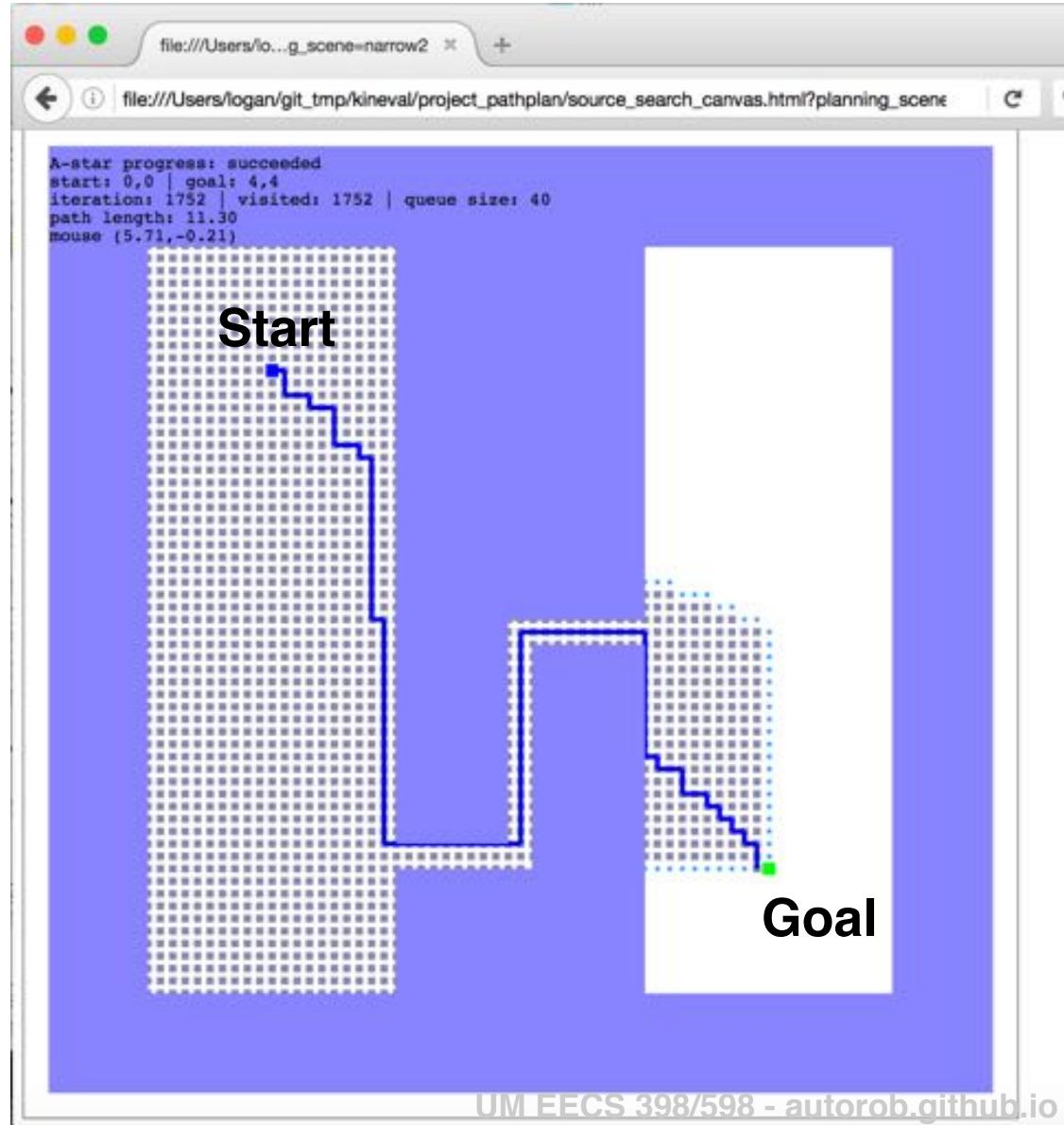
$$|x_1 - x_2| + |y_1 - y_2|$$

$$\max(|x_1 - x_2|, |y_1 - y_2|)$$

<https://lyfat.wordpress.com/2012/05/22/euclidean-vs-chebyshev-vs-manhattan-distance/.org>

Project 1: 2D Path Planning

- A-star algorithm for search in a given 2D world
- Heap data structure for priority queue
- Implement in JavaScript/HTML5 (next lecture)
- Grad: DFS, BFS, Greedy
- Submit through your git repository



```
<html>
<title>How do we implement this planner?</title>

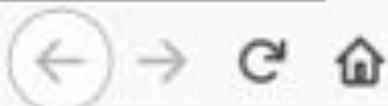
<body>
<h1>Next lecture:</h1>
<p>JavaScript/HTML5 and git Tutorial</p>

<a href="http://autorob.github.io">
EECS 367 Introduction to Autonomous Robotics <br>
ROB 510 ME/EECS 567 Robot Kinematics and Dynamics
</a>

</body>
</html>
```



How do we implement this planner? X



next_lecture.html

Next lecture:

JavaScript/HTML5 and git Tutorial

[EECS 367 Introduction to Autonomous Robotics](#)

[ROB 510 ME/EECS 567 Robot Kinematics and Dynamics](#)