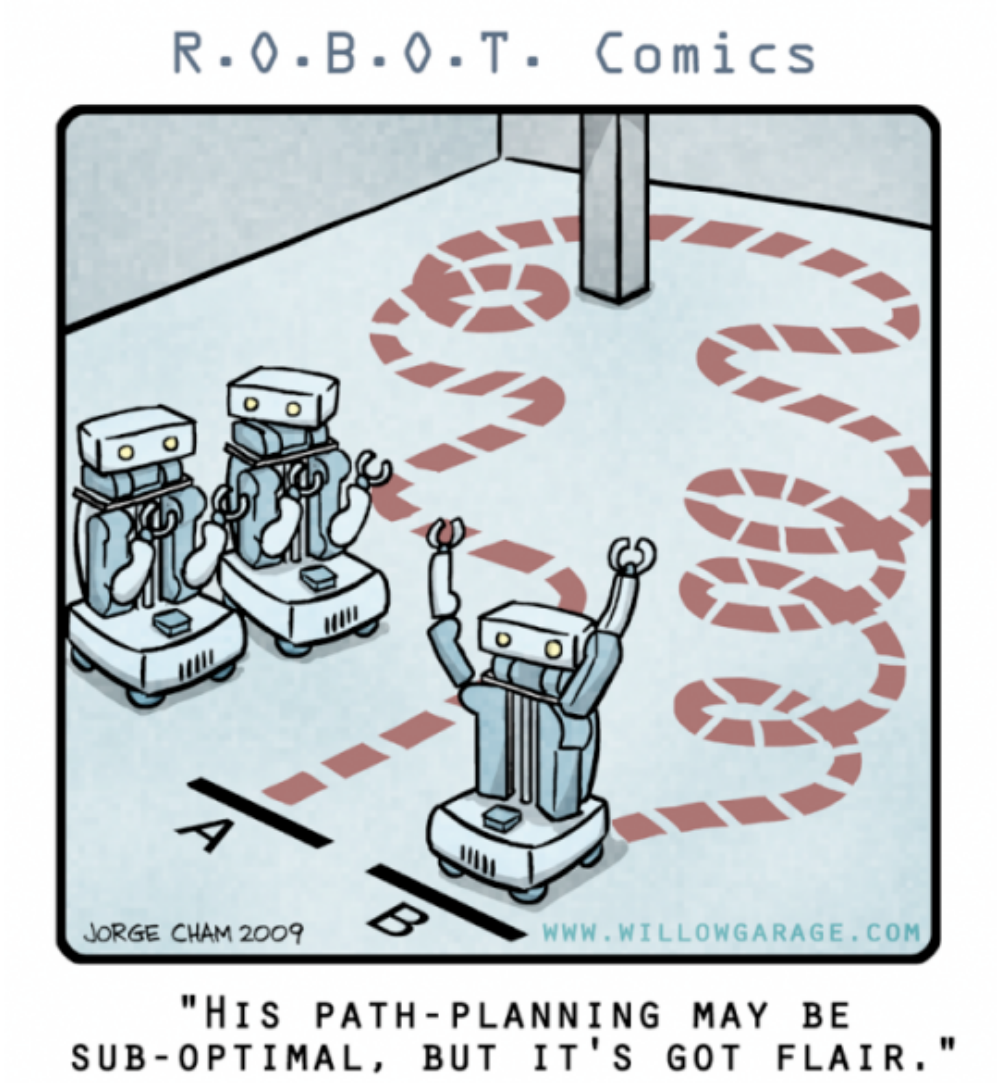


Path planning

the best way to get from A to B

EECS 398
Intro. to Autonomous Robotics

ME/EECS 567 ROB 510
Robot Modeling and Control

Fall 2018

autorob.org

Administrivia

- Assignment 1 (Path Planning) has been released
  - due 1:20pm, Monday, September 24, 2018
  - Send me your email, picture, and pointer to git repository
- Anyone still need permission to move from waitlist?
  - Please see me if you are not added by the end of Wednesday's lecture
- Did you see May Mobility on the TODAY show this morning!?

Feature | Geek Life | Profiles

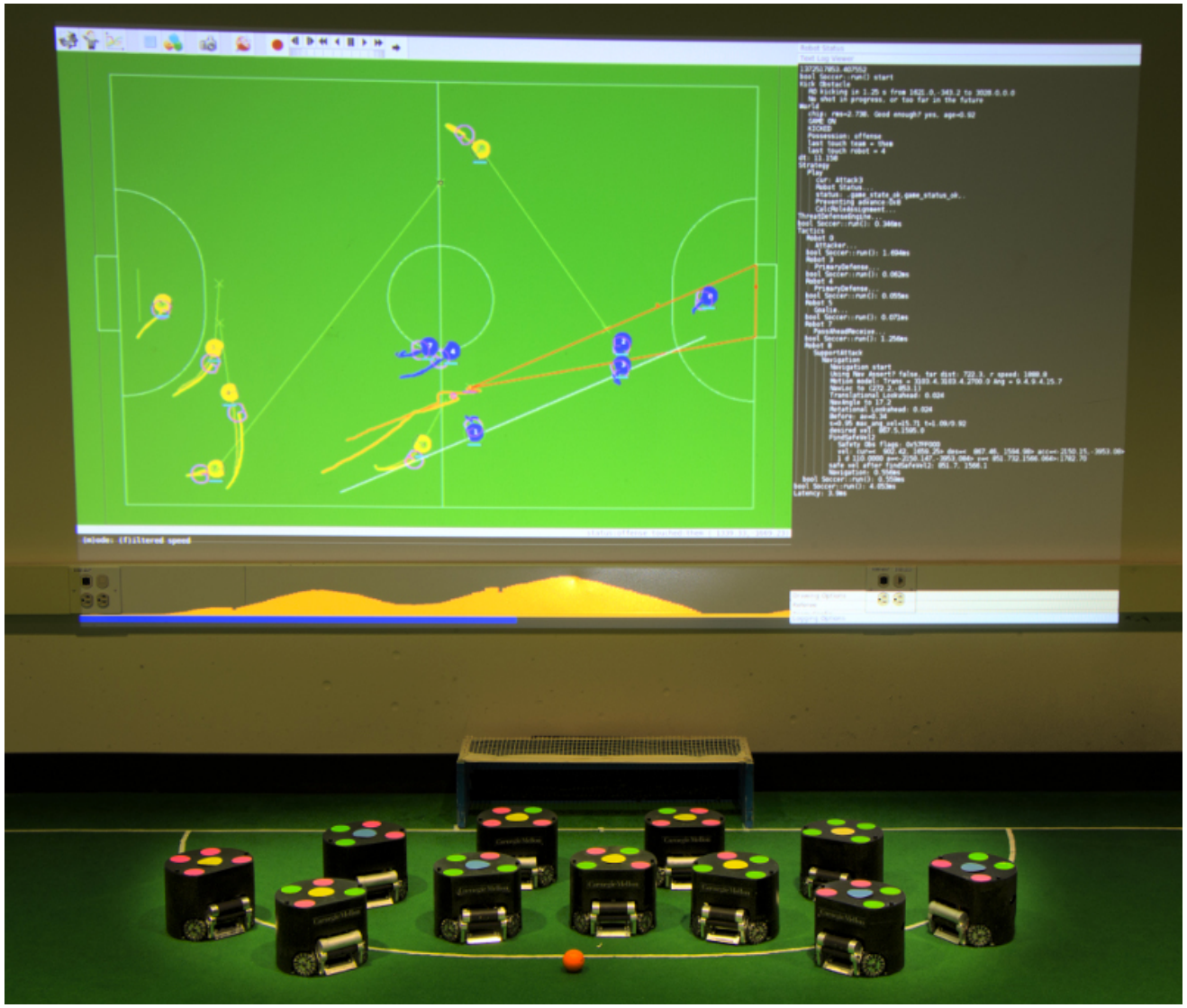
Manuela Veloso: RoboCup's Champion

This roboticist has transformed robot soccer into a global phenomenon

Posted 28 Feb 2015 | 17:00 GMT

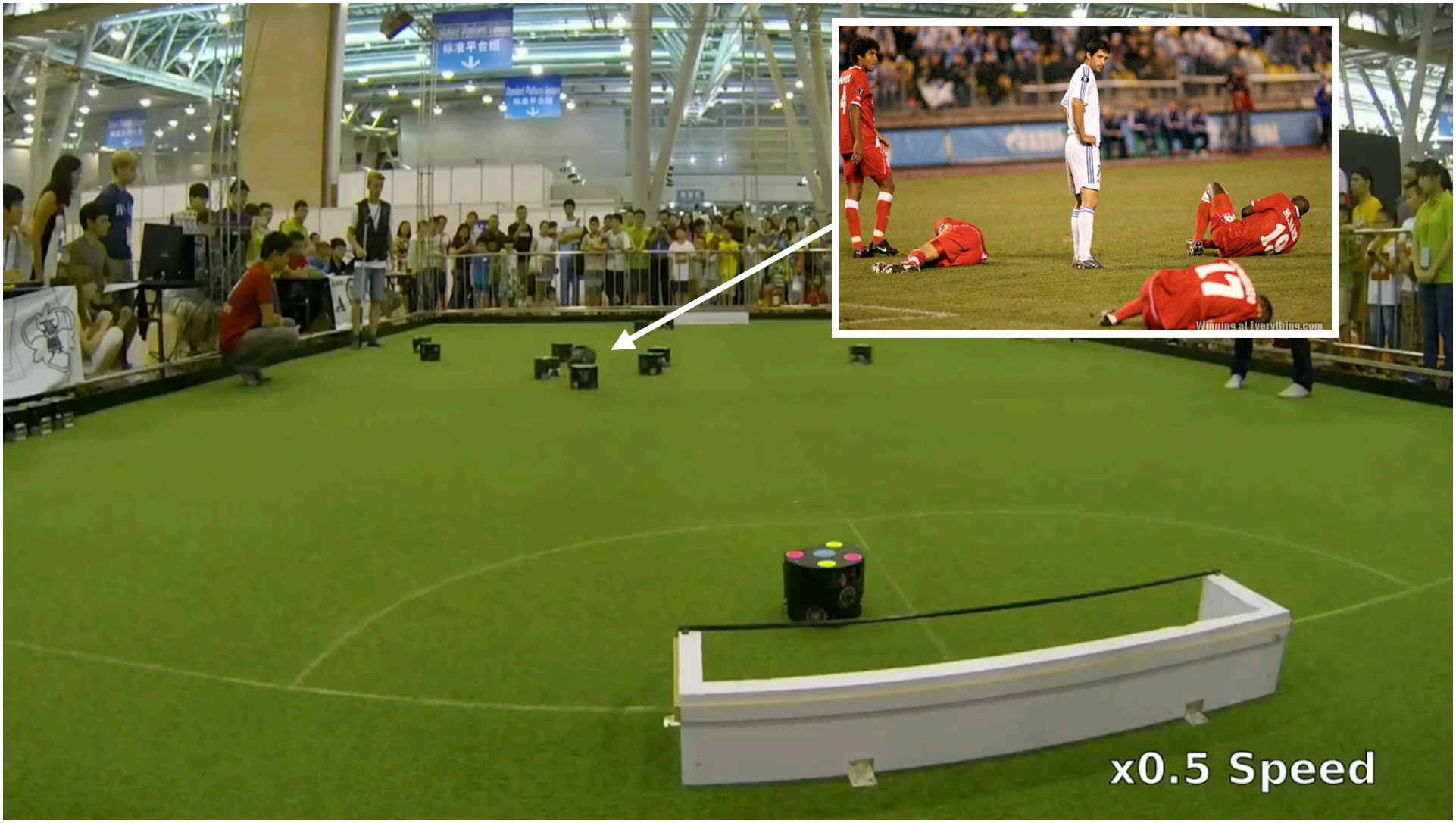
By **PRACHI PATEL**

CMDragons
RoboCup Small
2006



CMDragons 2016 Pass-ahead Goal

CMDragons 2016 slow-motion multi-pass goal



CMDragons 2016 slow-motion multi-pass goal

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

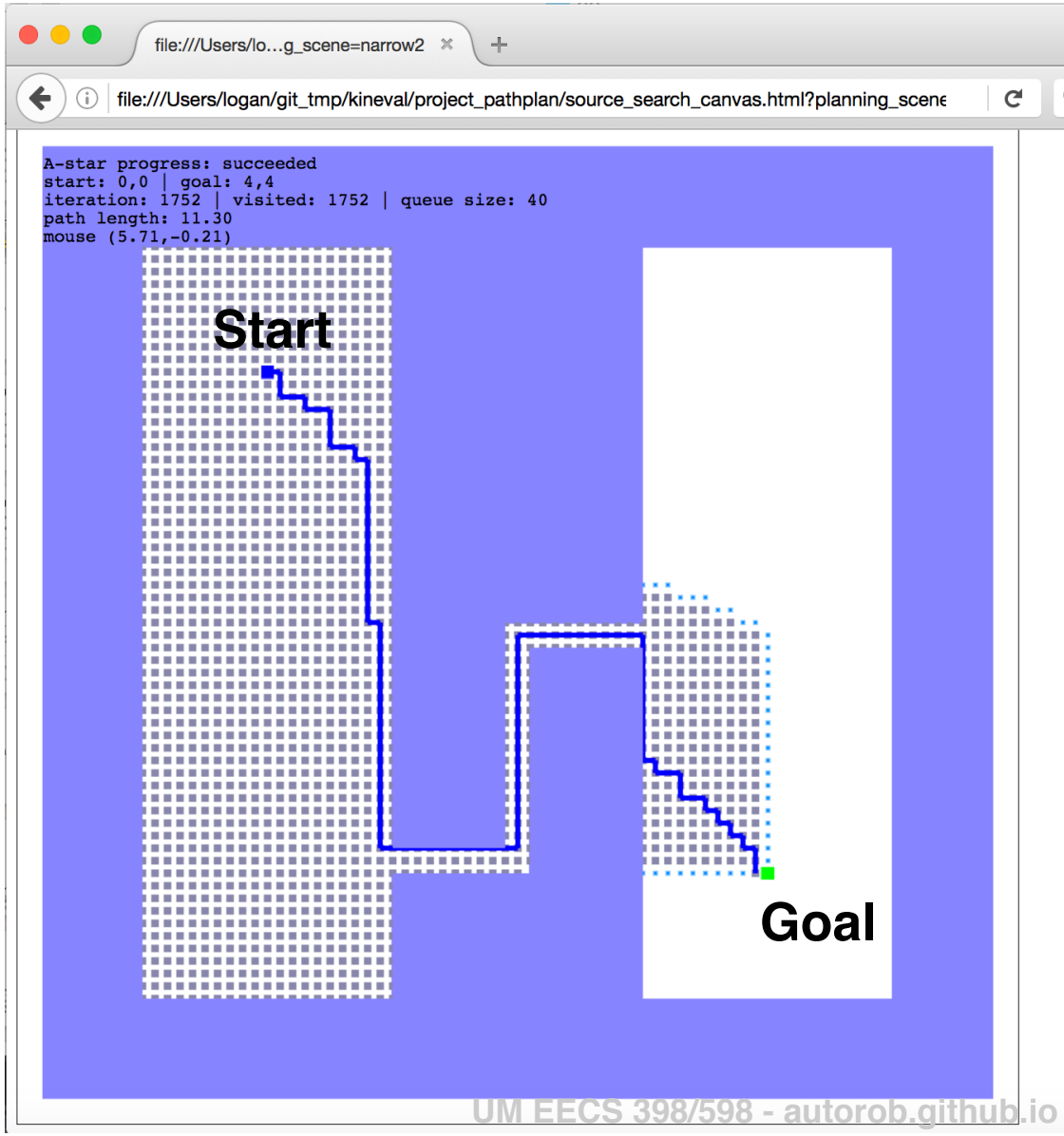
http://www.cs.cmu.edu/~coral/projects/cobot/

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

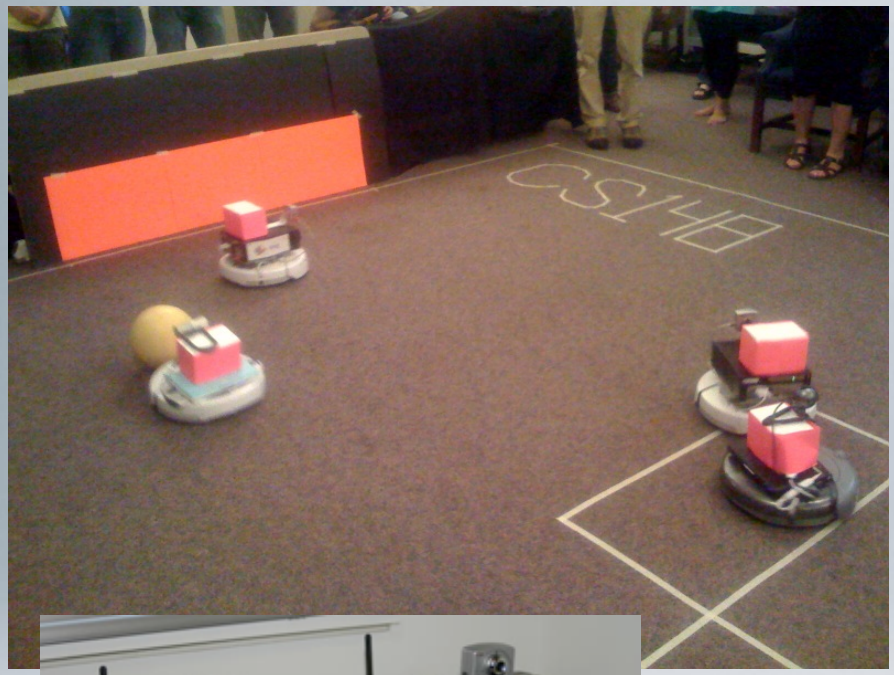
CoBot greets me in 2013

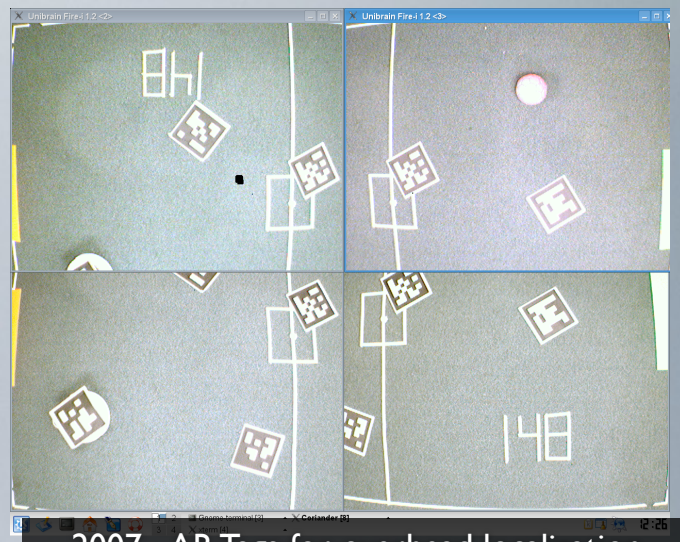
Project 1:
2D Path Planning

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

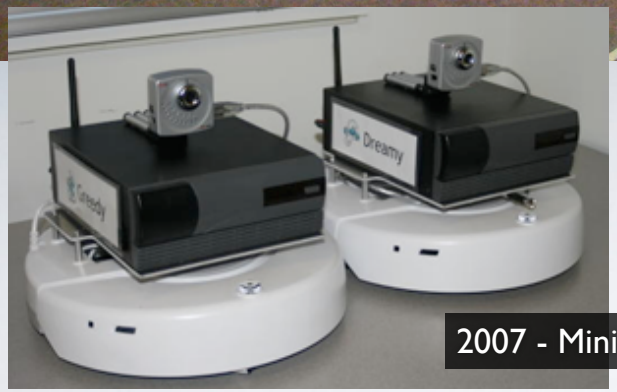


2007-10: SOCCER WITH IROBOT CREATE





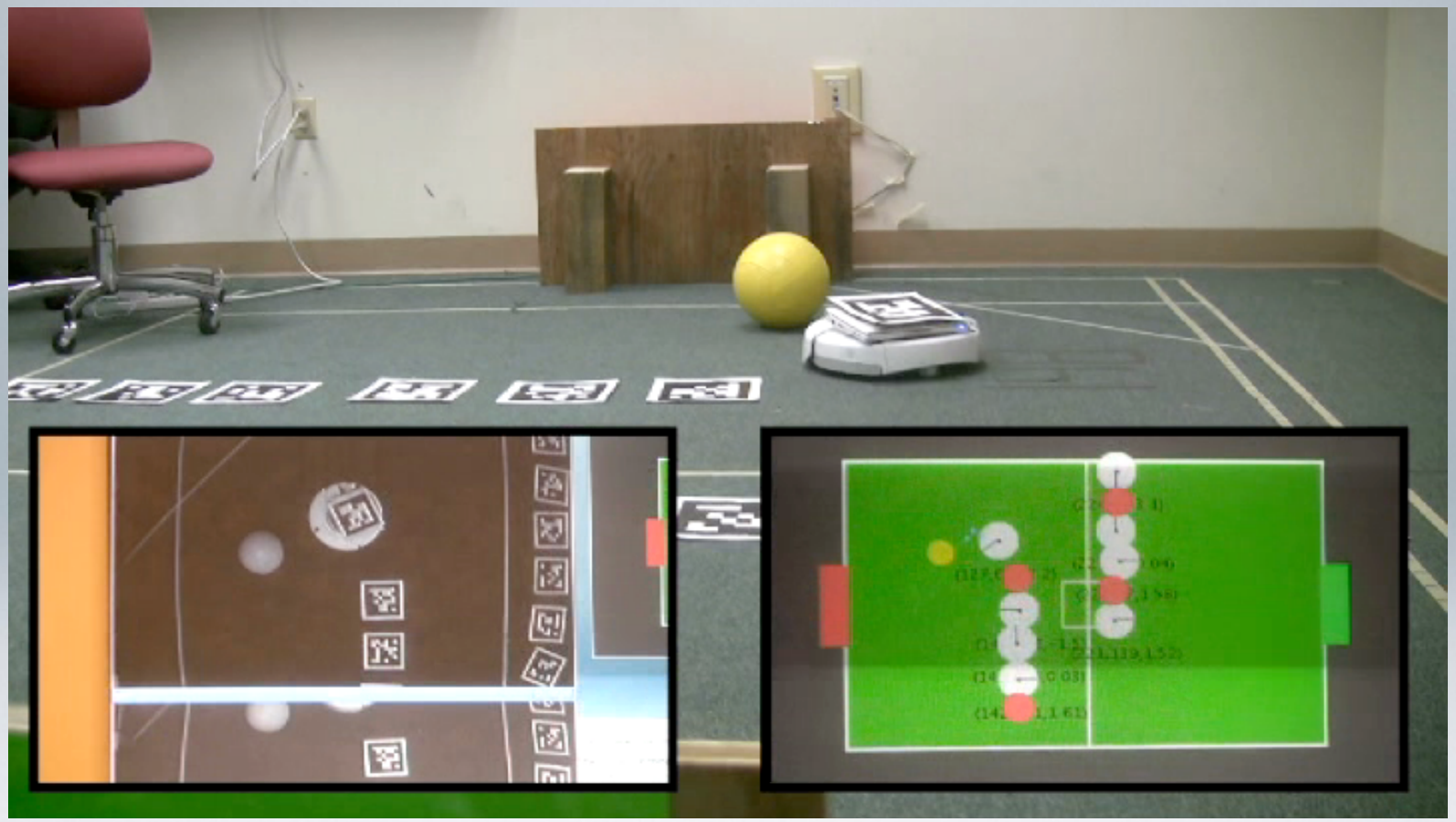
2007 - AR Tags for overhead localization

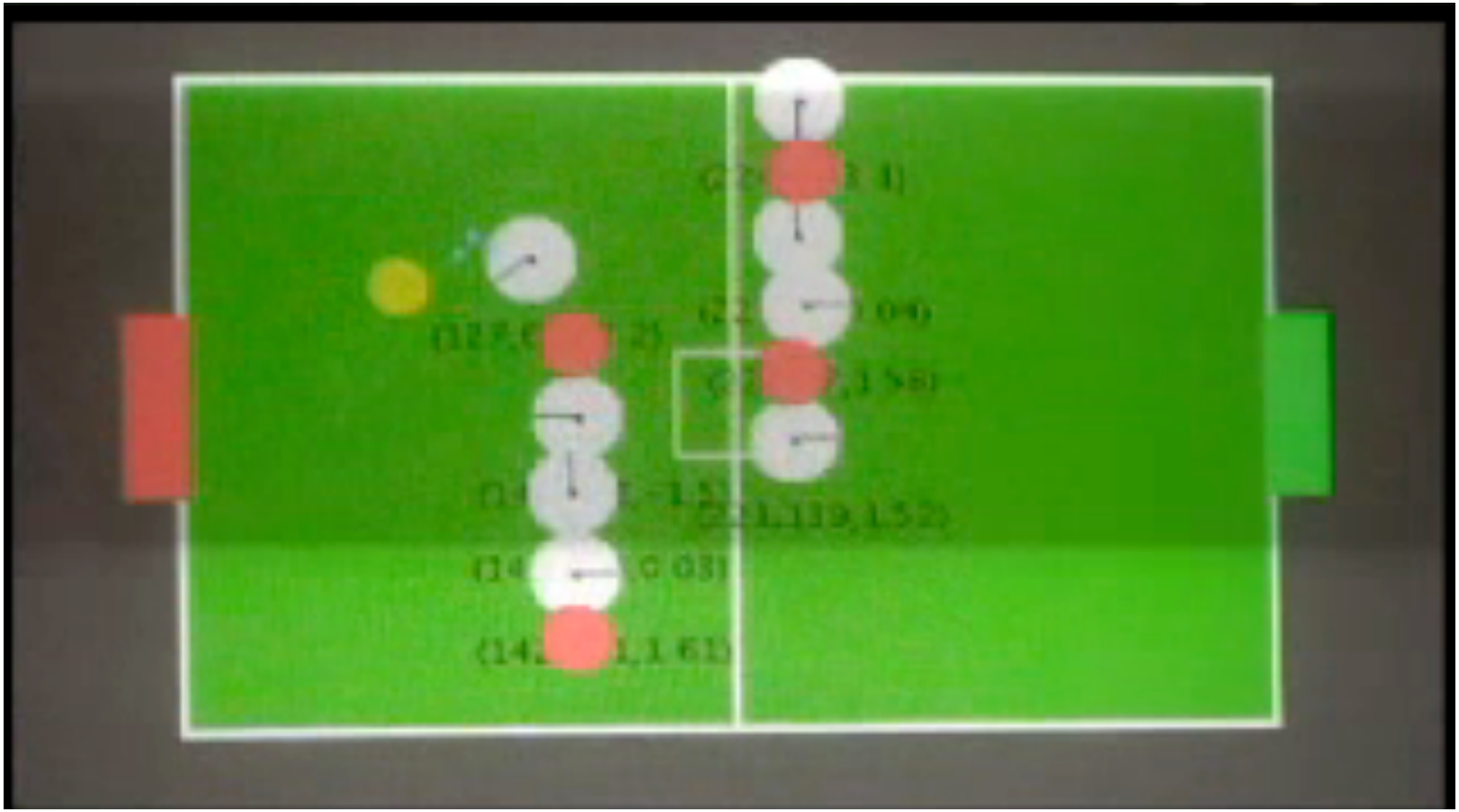


2007 - Mini ITX

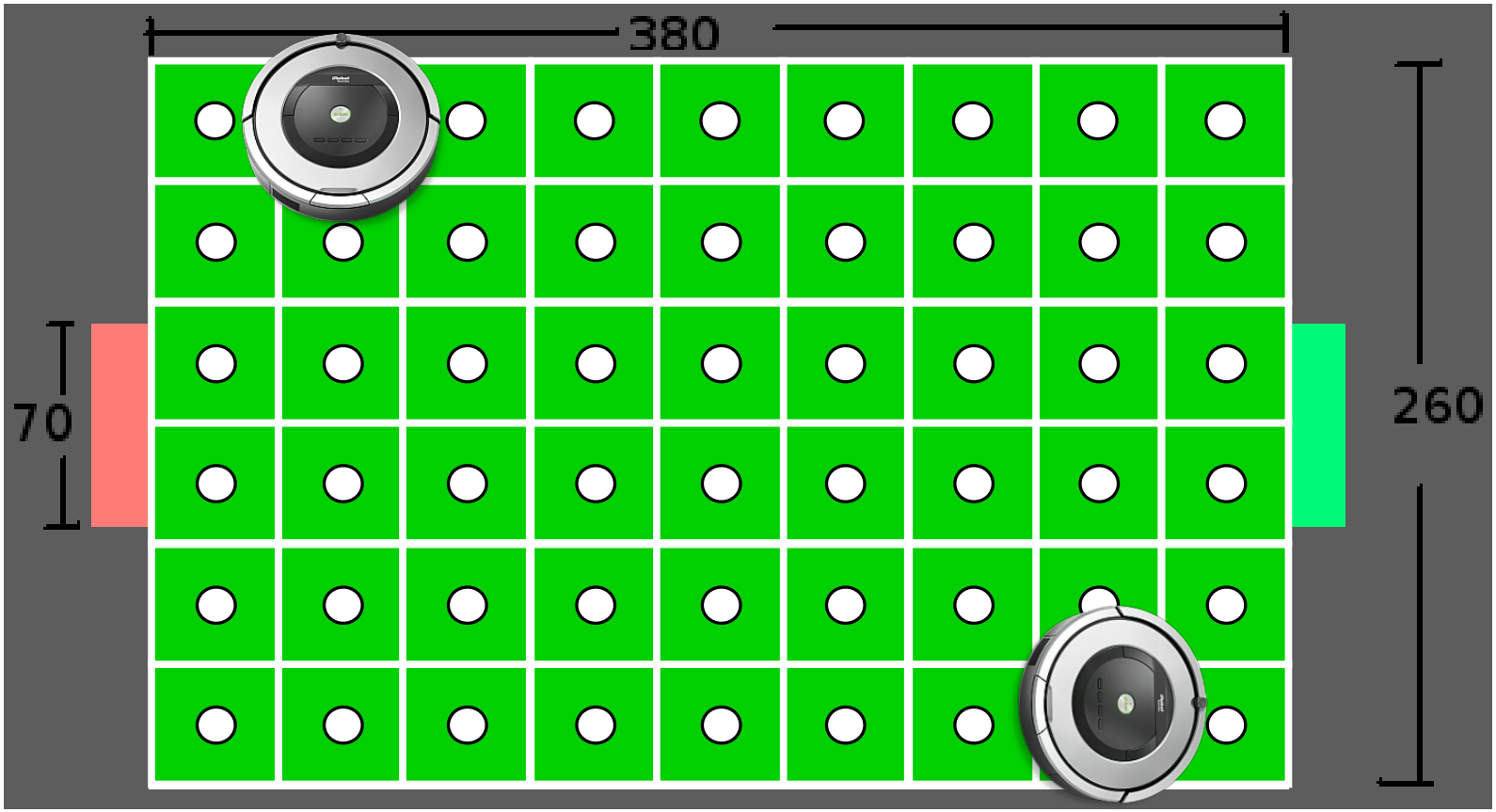
2009 - Asus EEE

http://www.youtube.com/watch?v=88zR6IC7S0g

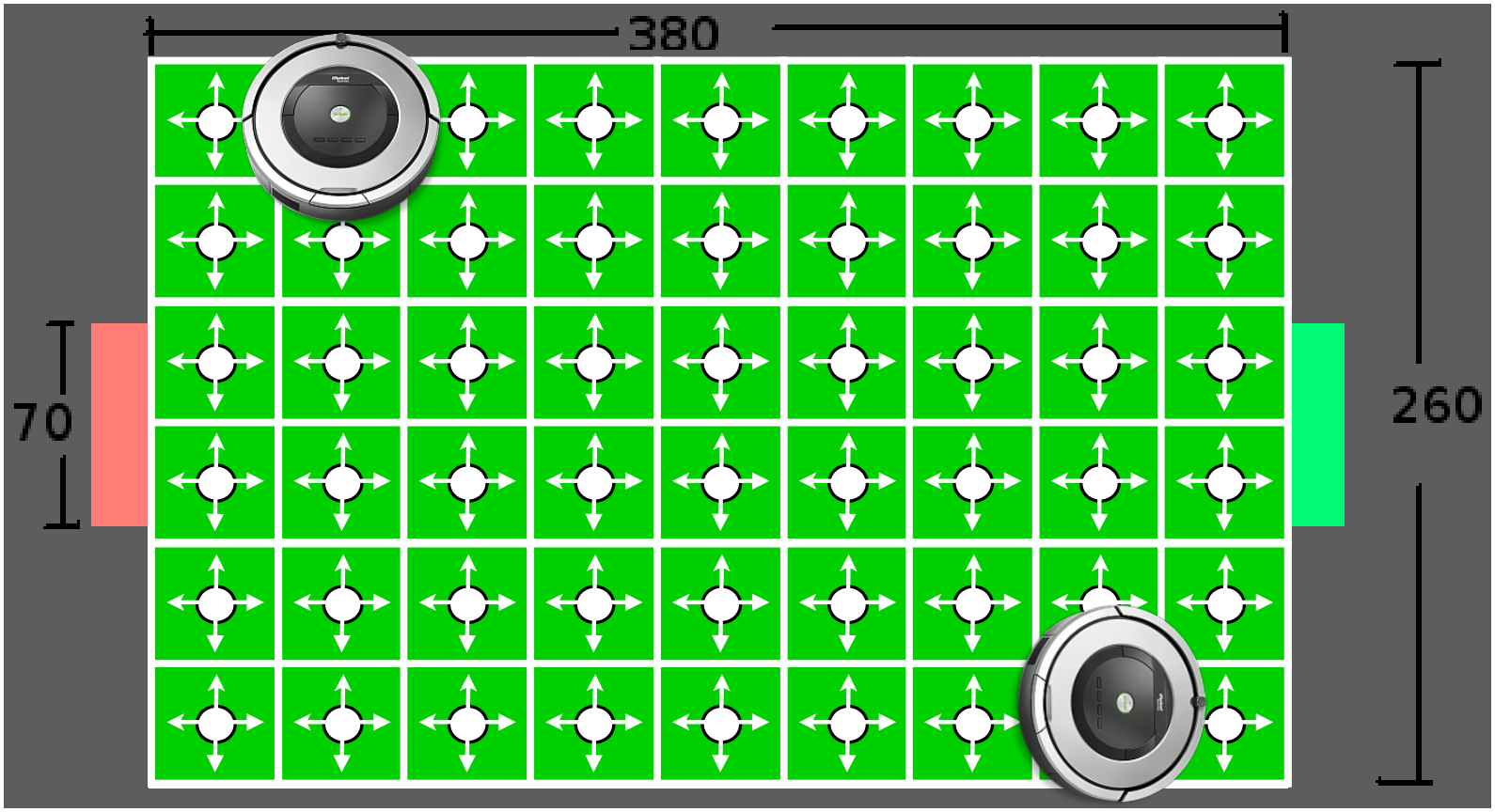




Consider all possible poses as uniformly distributed graph vertices



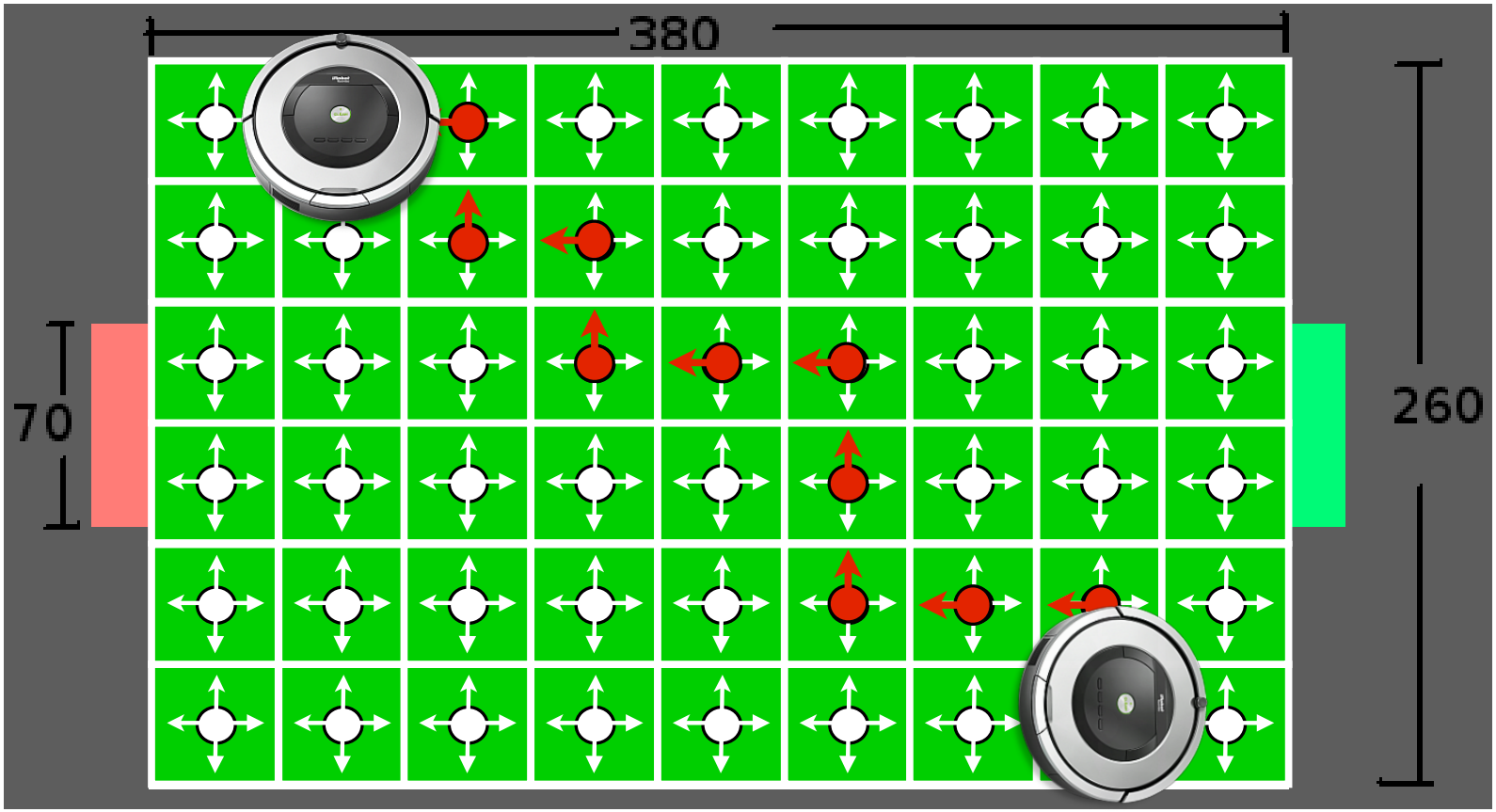
Consider all possible poses as uniformly distributed graph vertices
Edges connect adjacent (traversable) poses, weighted by distance



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?

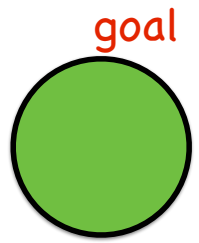


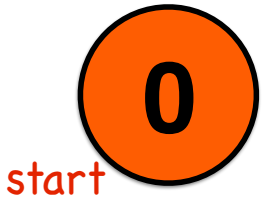
Approaches to motion planning

- 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

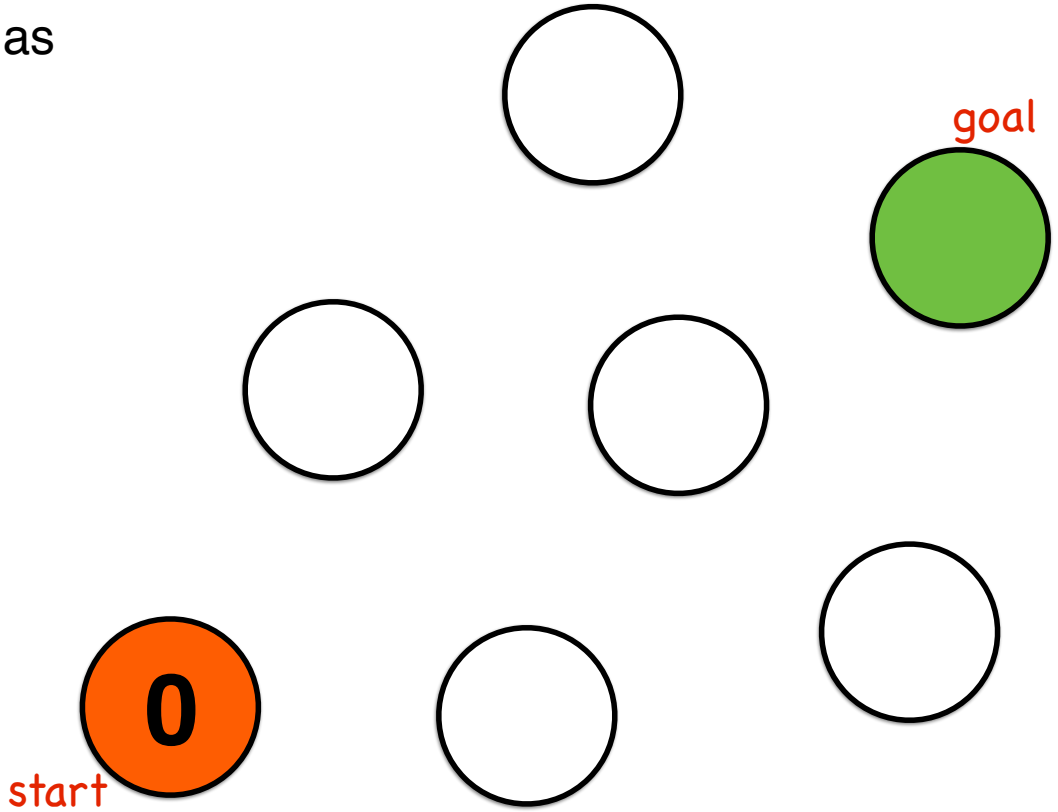
Consider each possible robot pose as
node in a graph





Consider a simple search graph

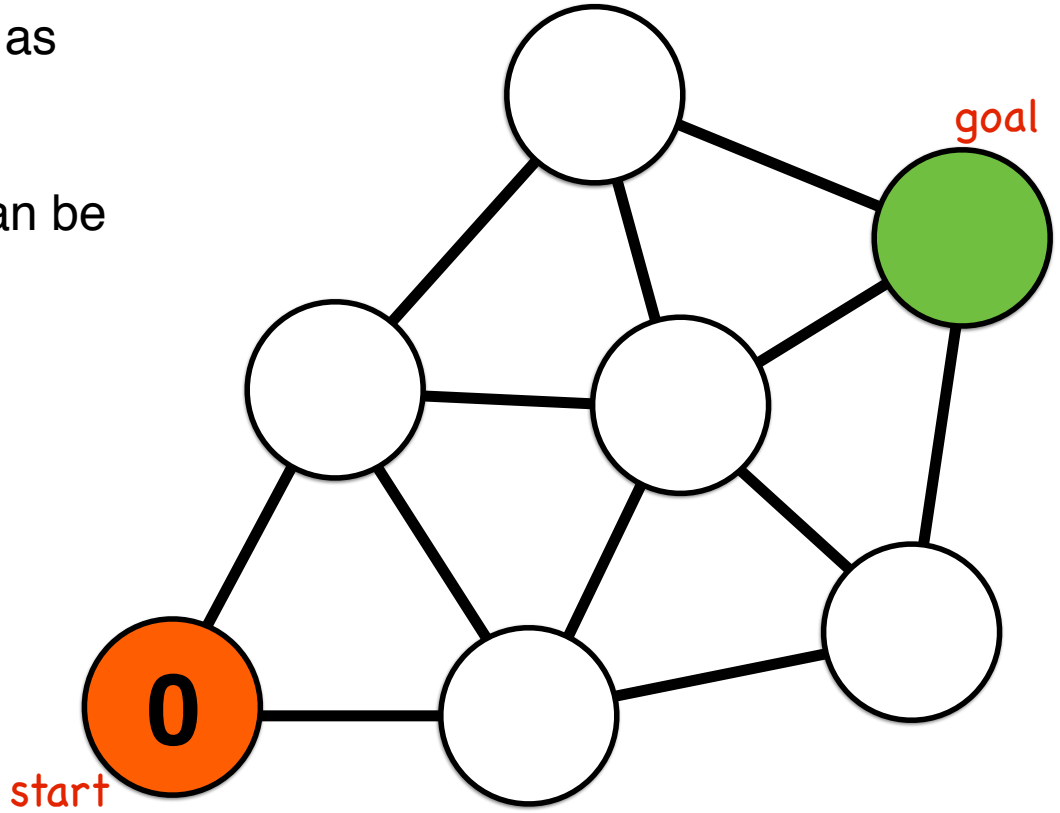
Consider each possible robot pose as node in a graph



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

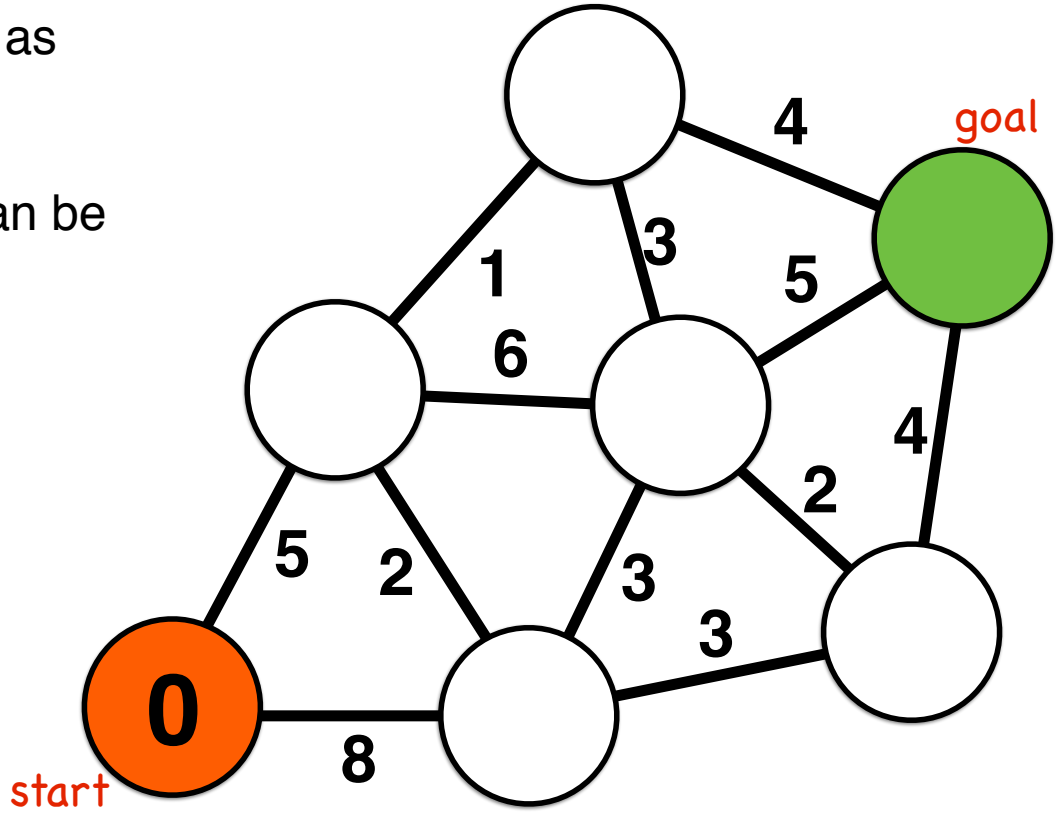


Consider a simple search graph

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

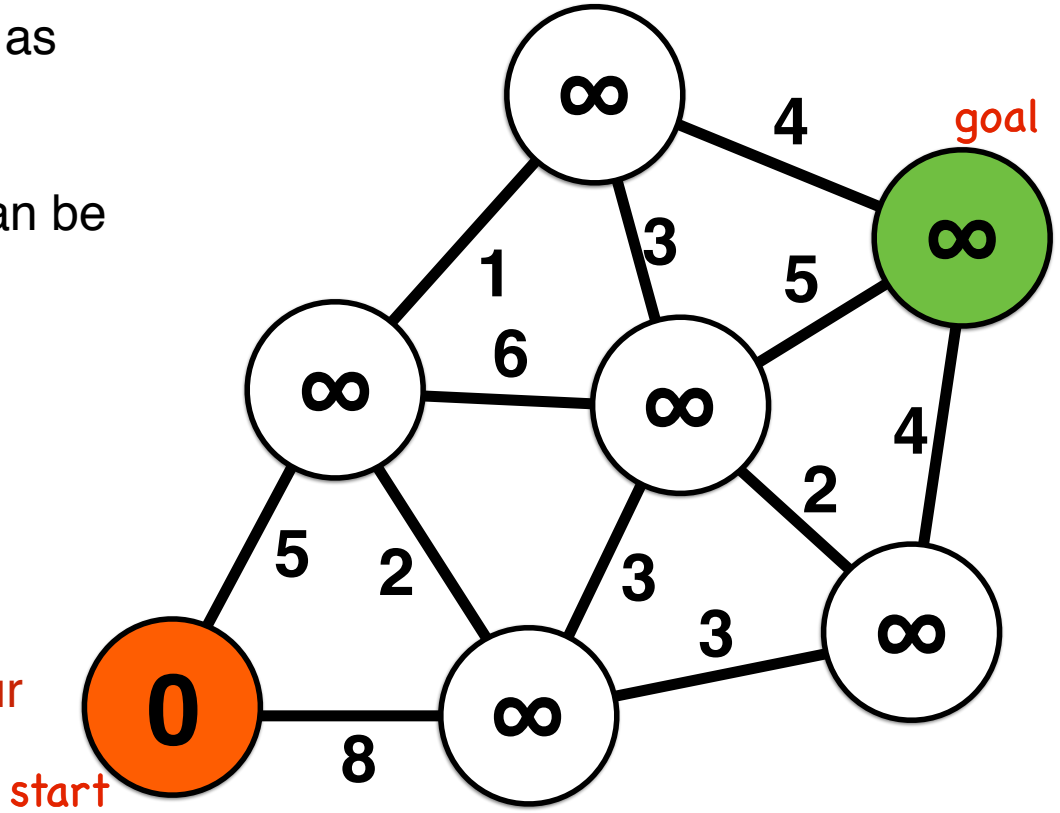
Consider each possible robot pose as node in a graph

Graph edges connect poses that can be reliably moved between

Edges have a cost for traversal

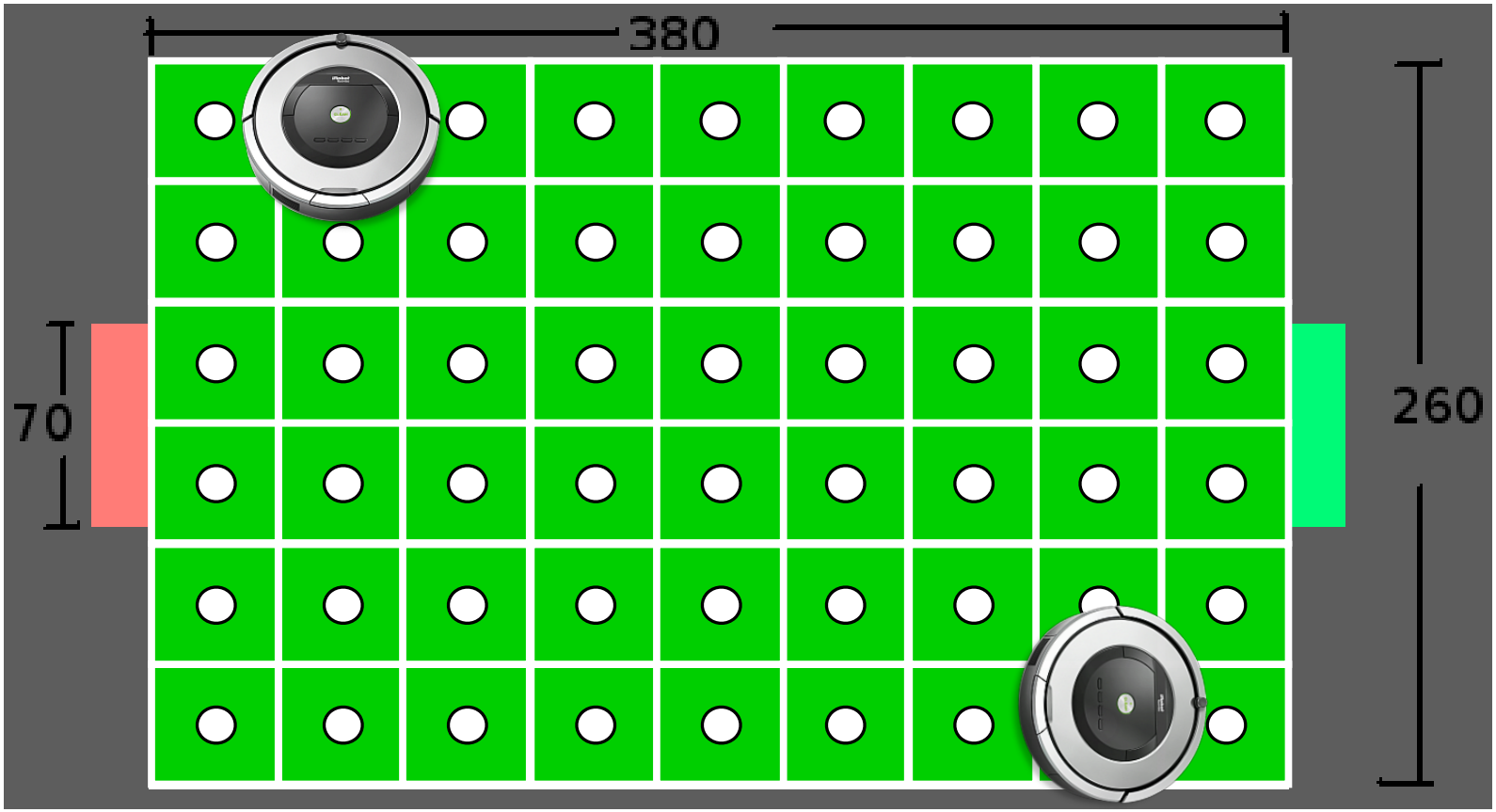
Each node maintains the distance traveled as a scalar cost

We will use a single template for our graph search algorithms

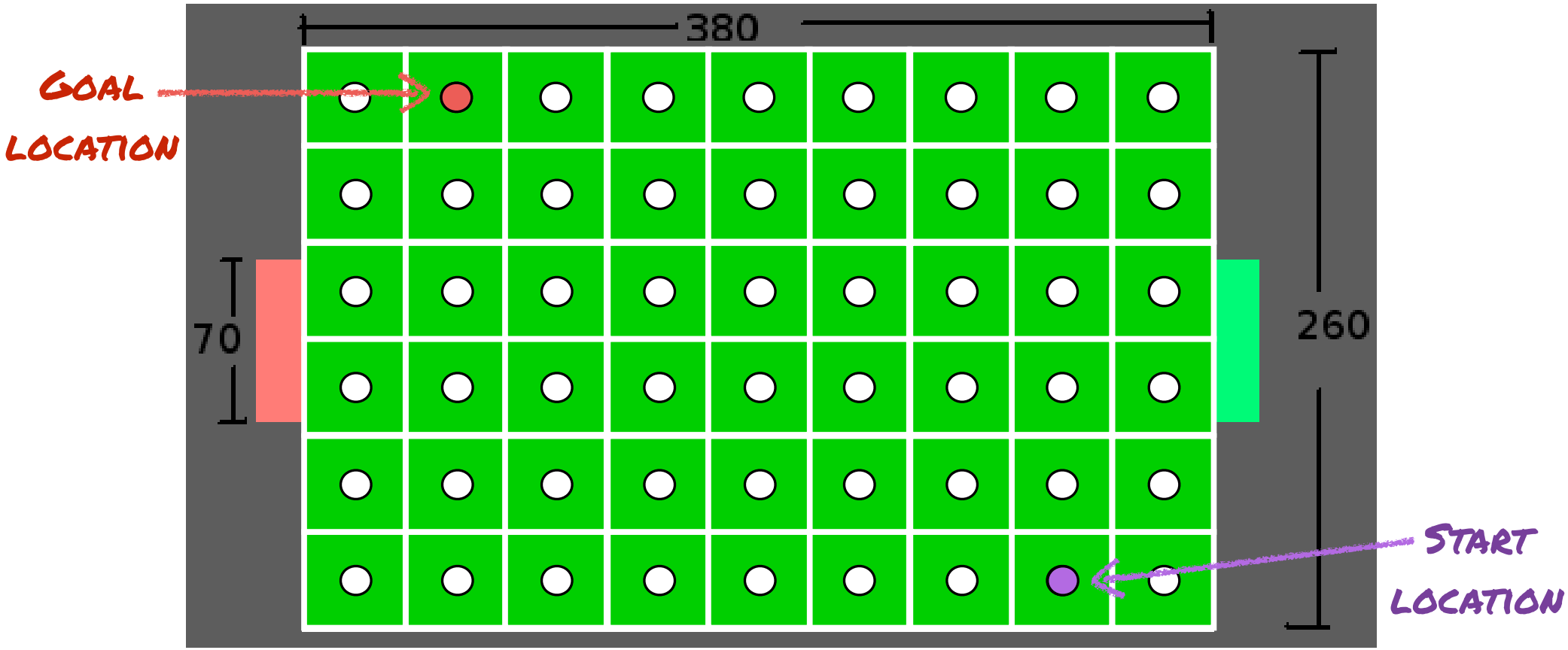


Depth-first search intuition

Depth-first search



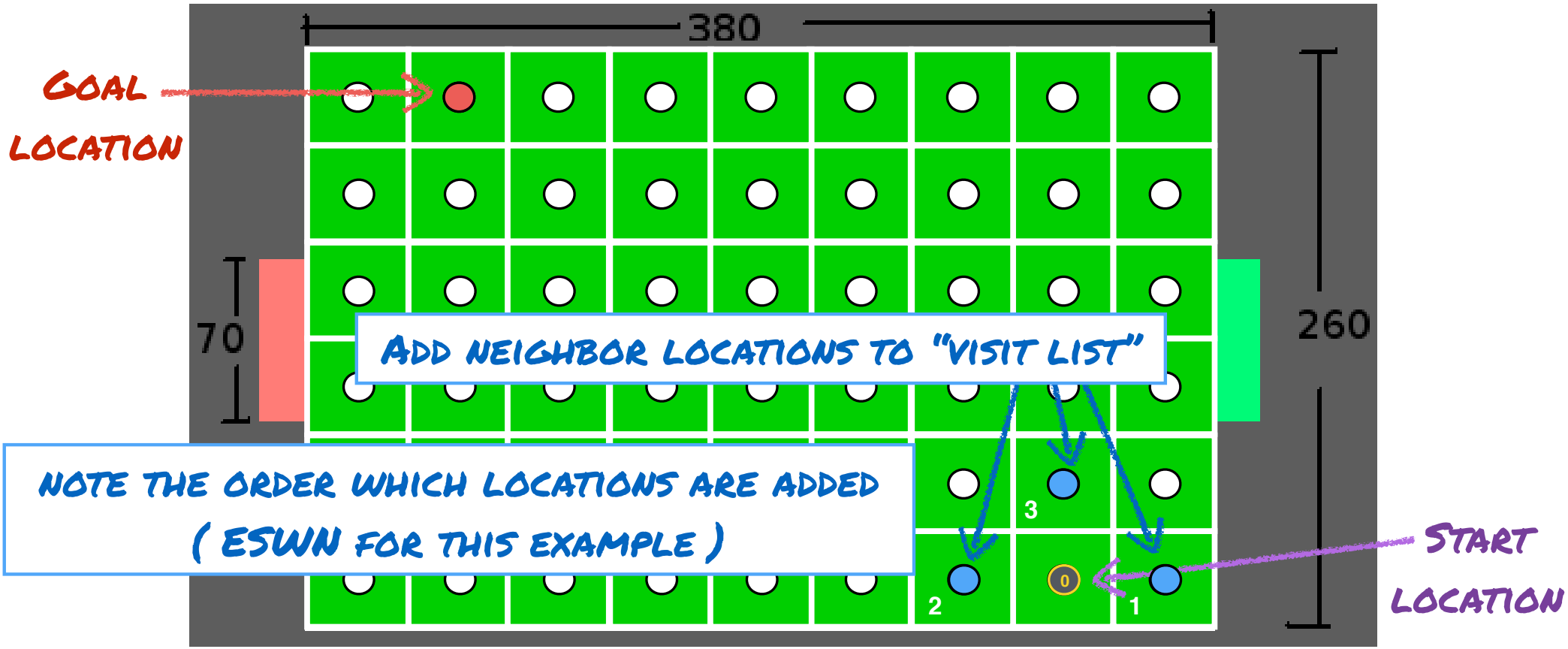
Depth-first search



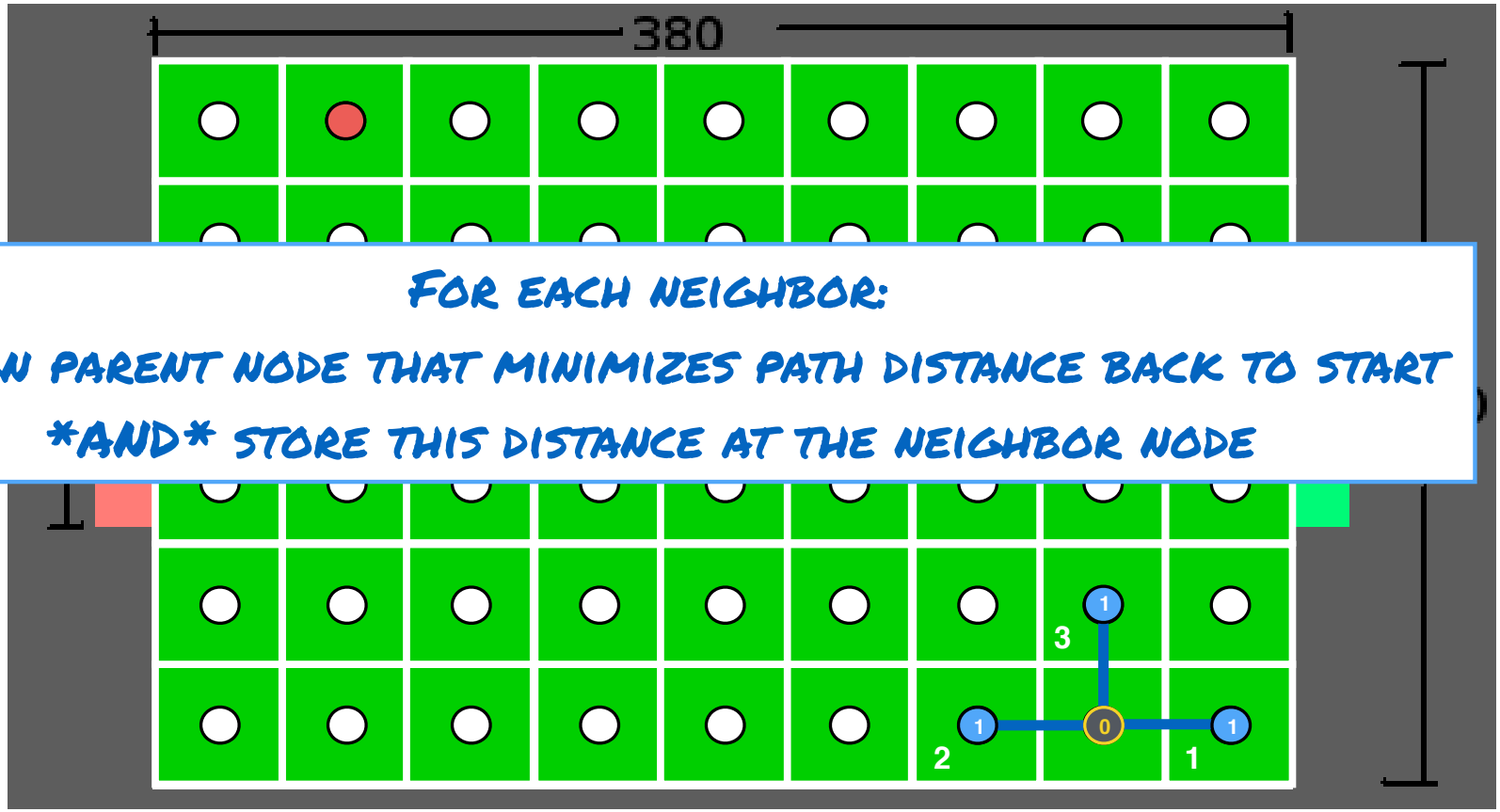
Depth-first search

Depth-first search

Depth-first search



Depth-first search

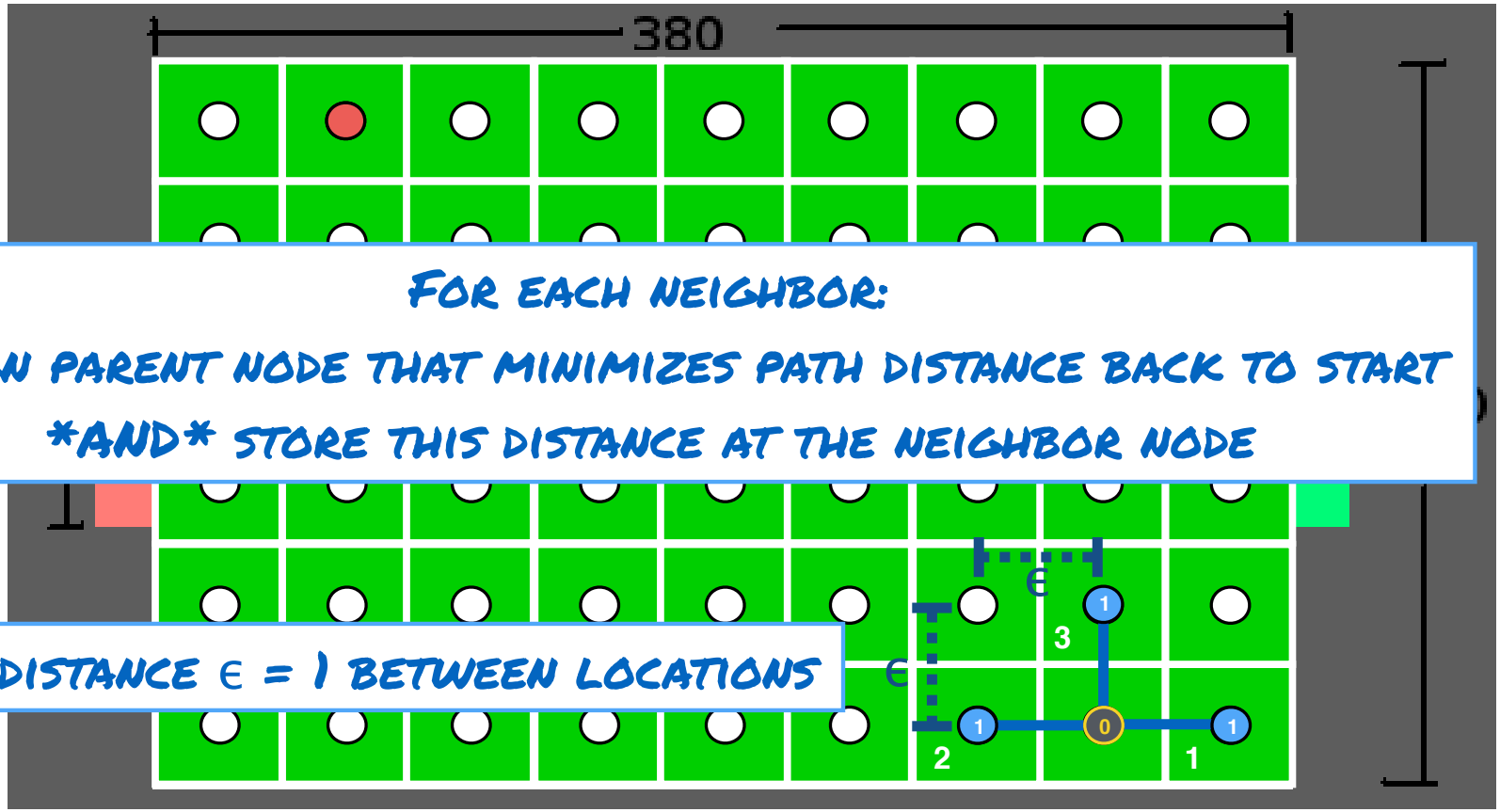


For each neighbor:

assign parent node that minimizes path distance back to start

and store this distance at the neighbor node

Depth-first search



For each neighbor:

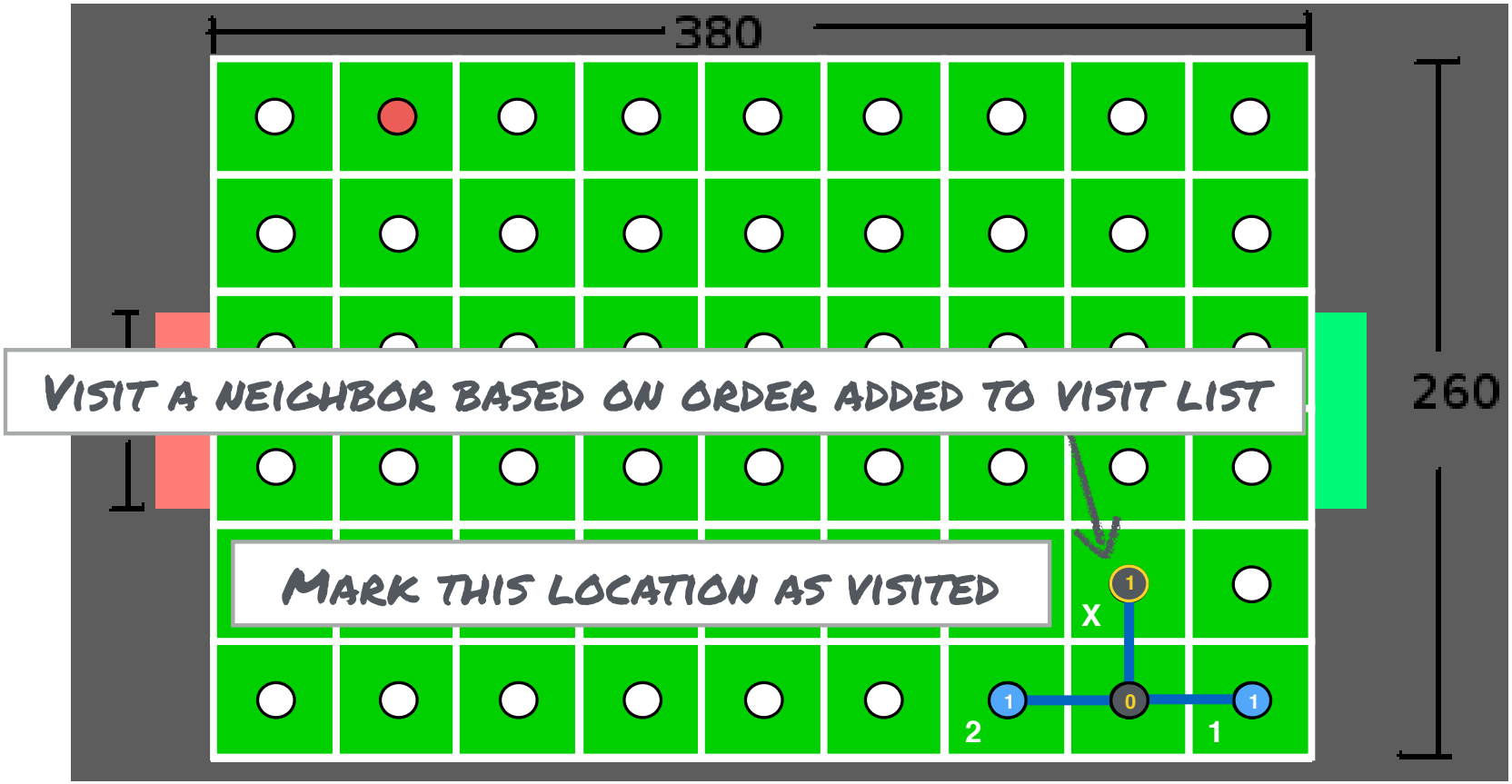
assign parent node that minimizes path distance back to start

AND store this distance at the neighbor node

assume distance ϵ = 1 between locations

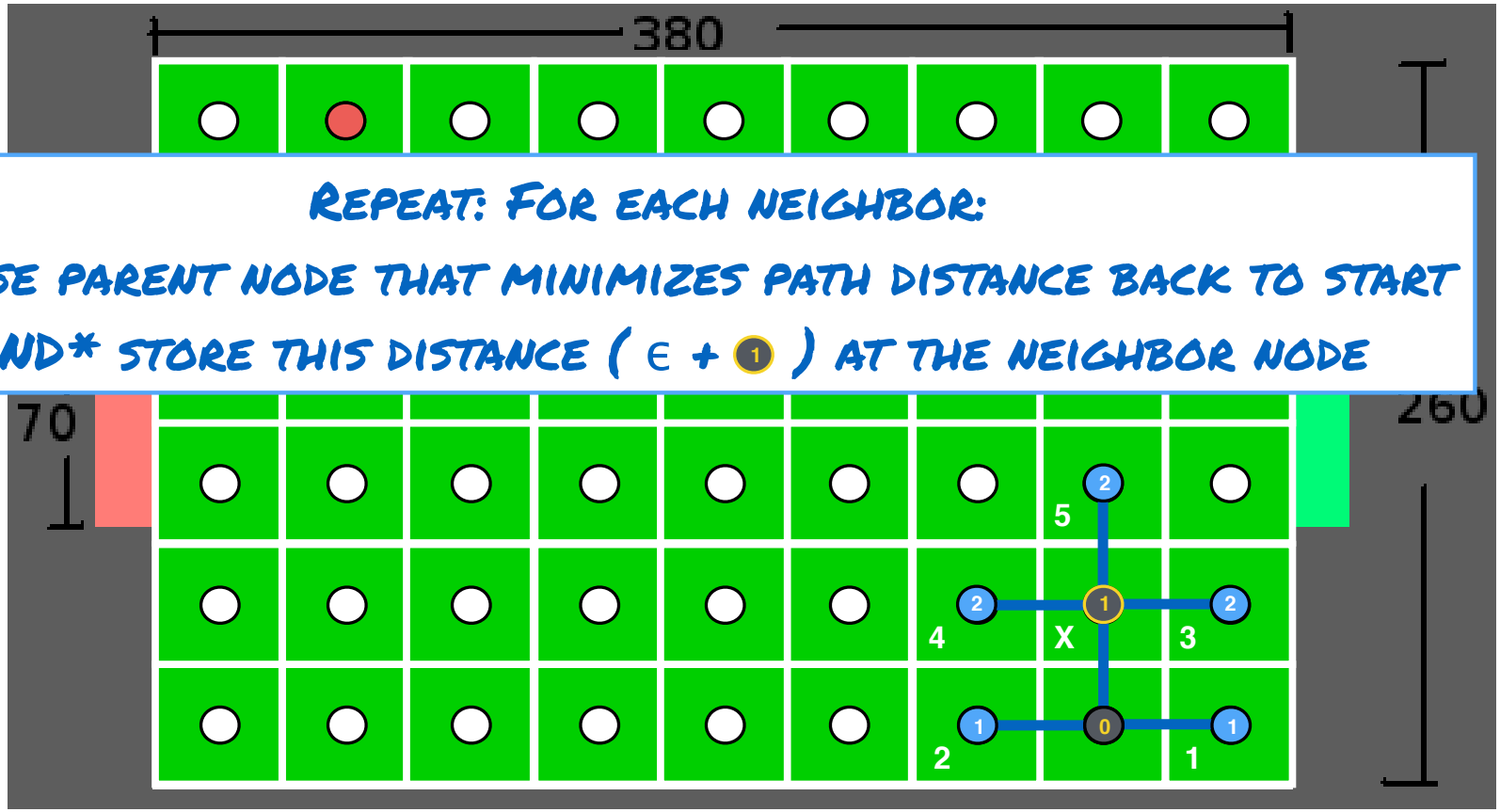
Depth-first search

Depth-first search



Depth-first search

Depth-first search

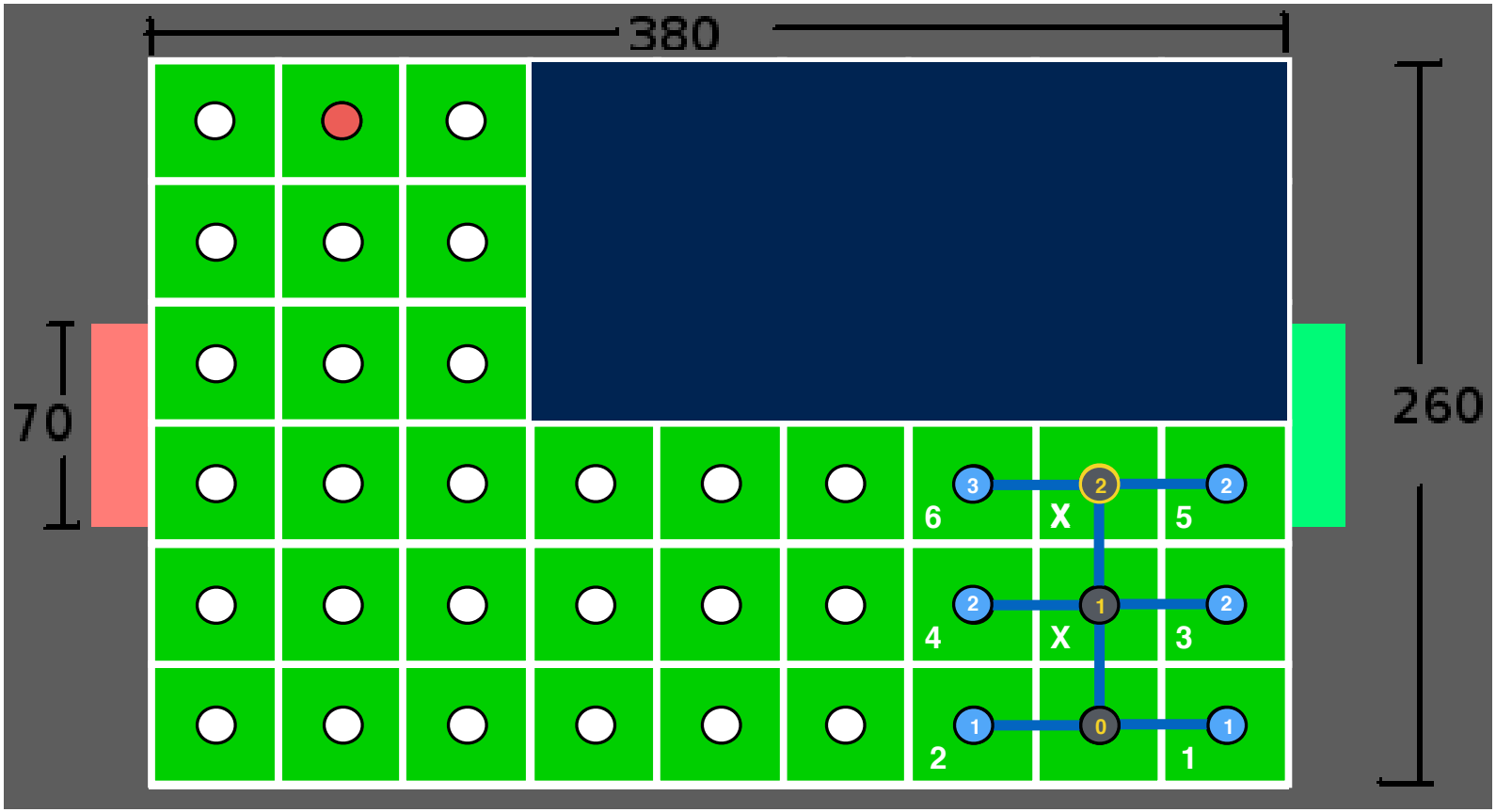


Depth-first search

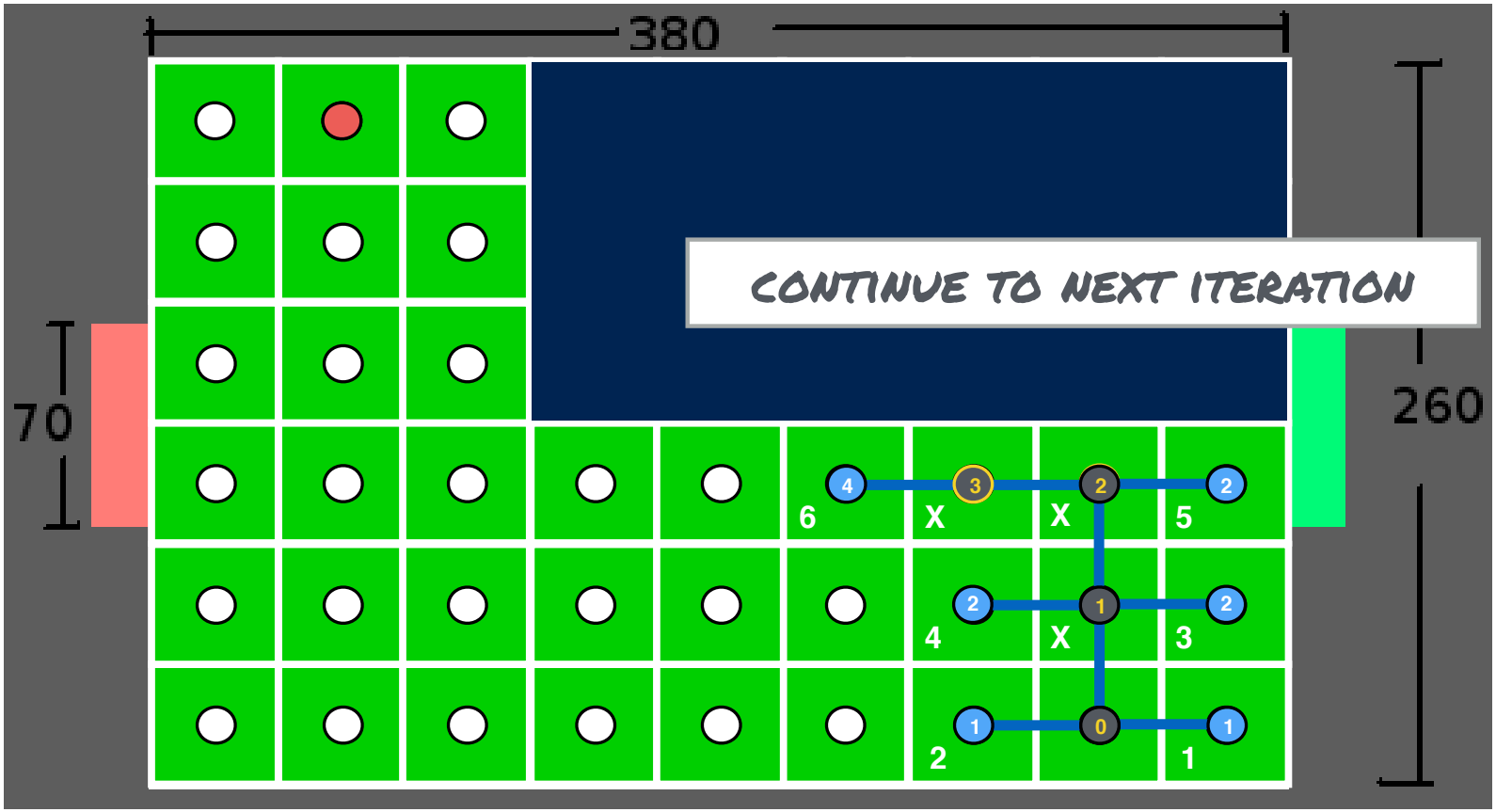
Depth-first search

Depth-first search

Depth-first search

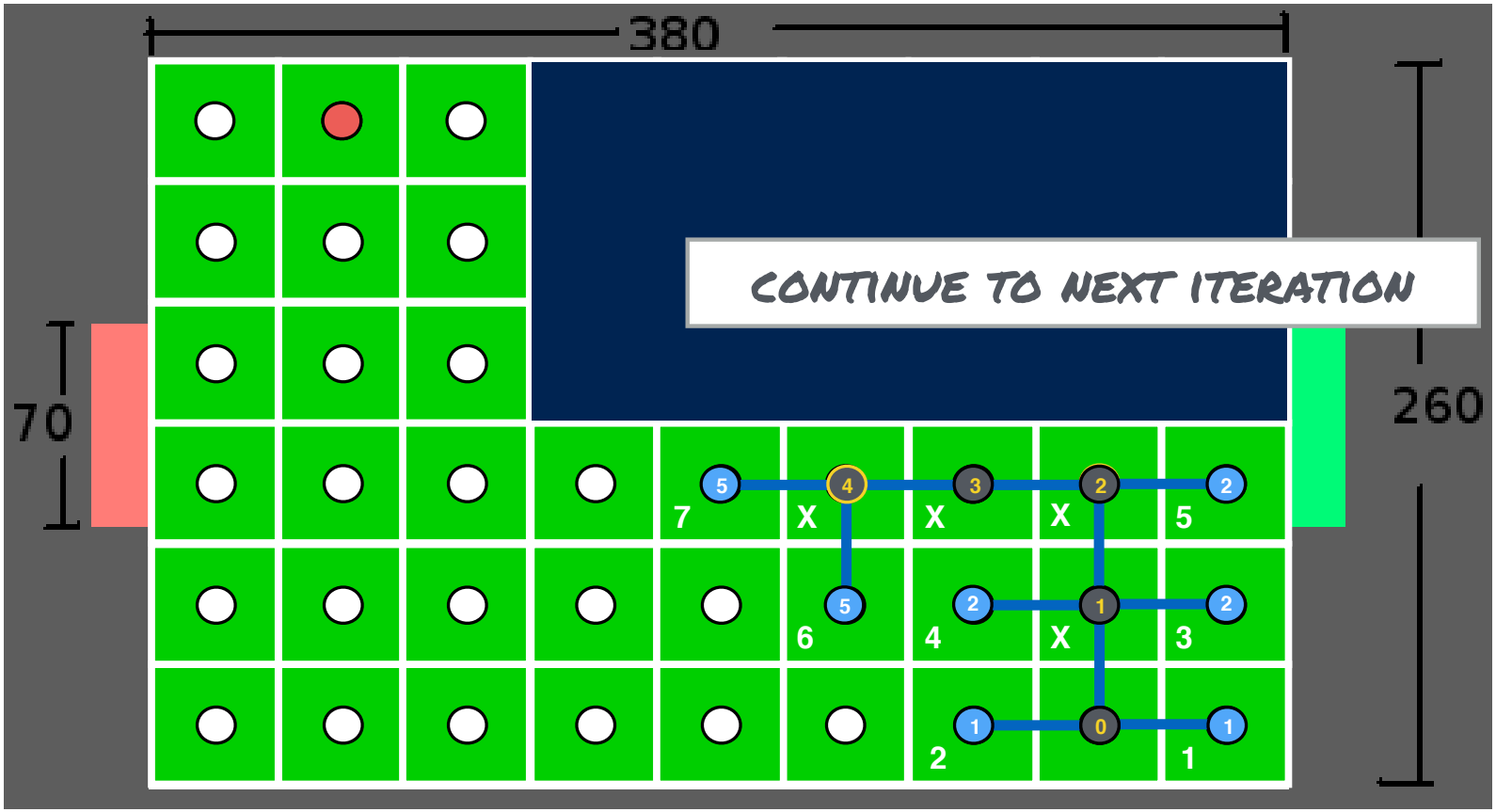


Depth-first search

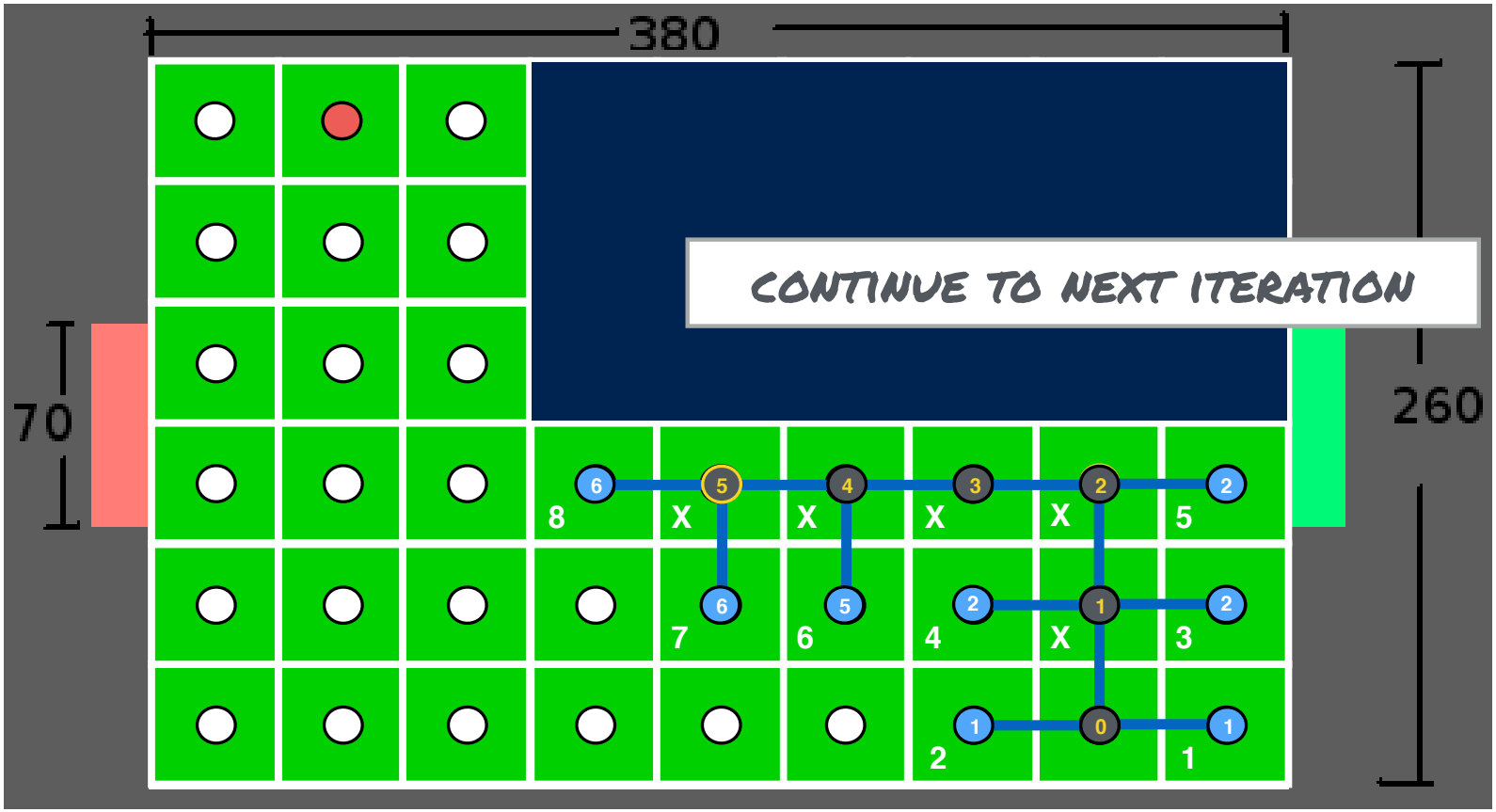


Depth-first search

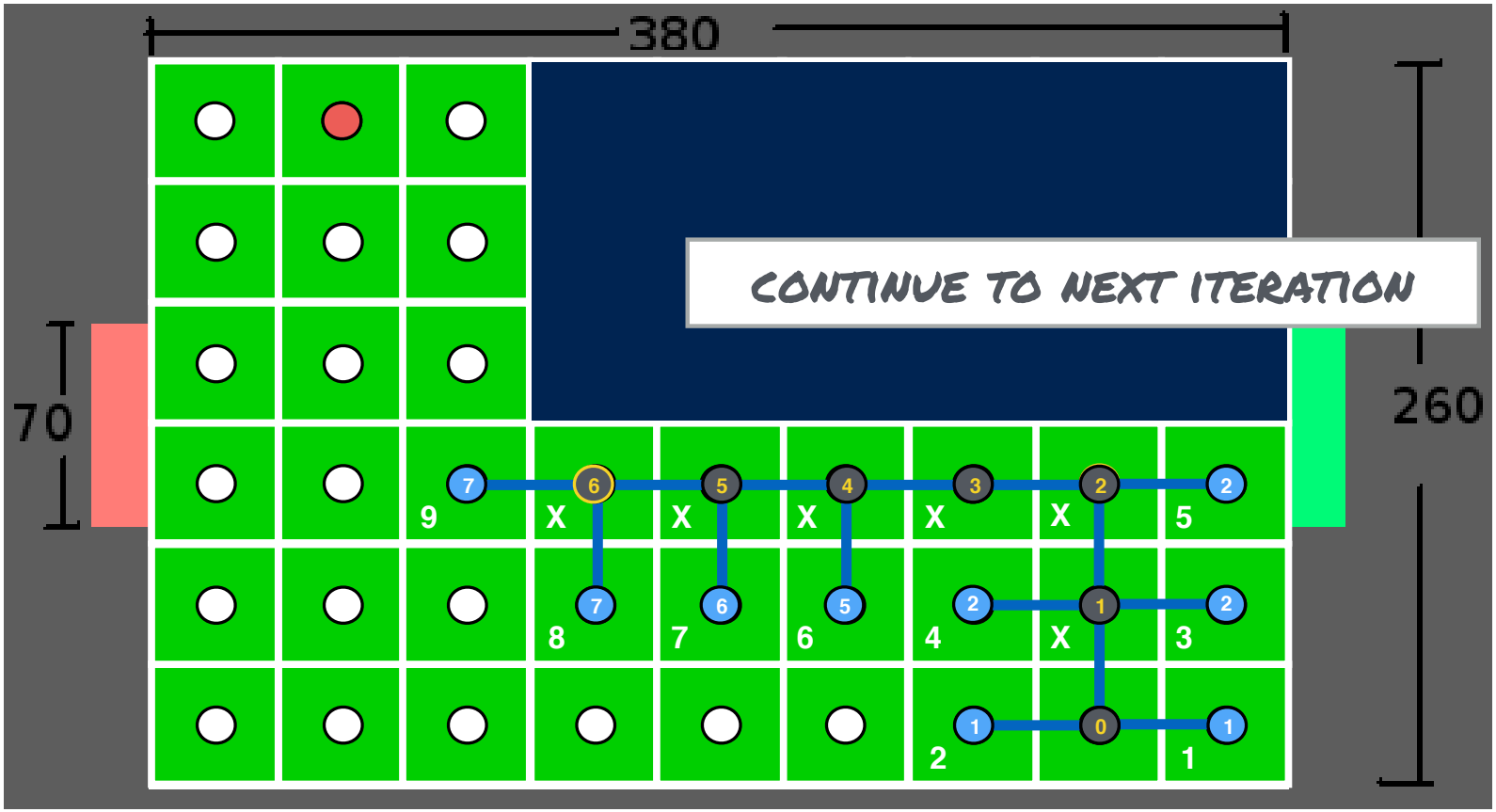
Depth-first search



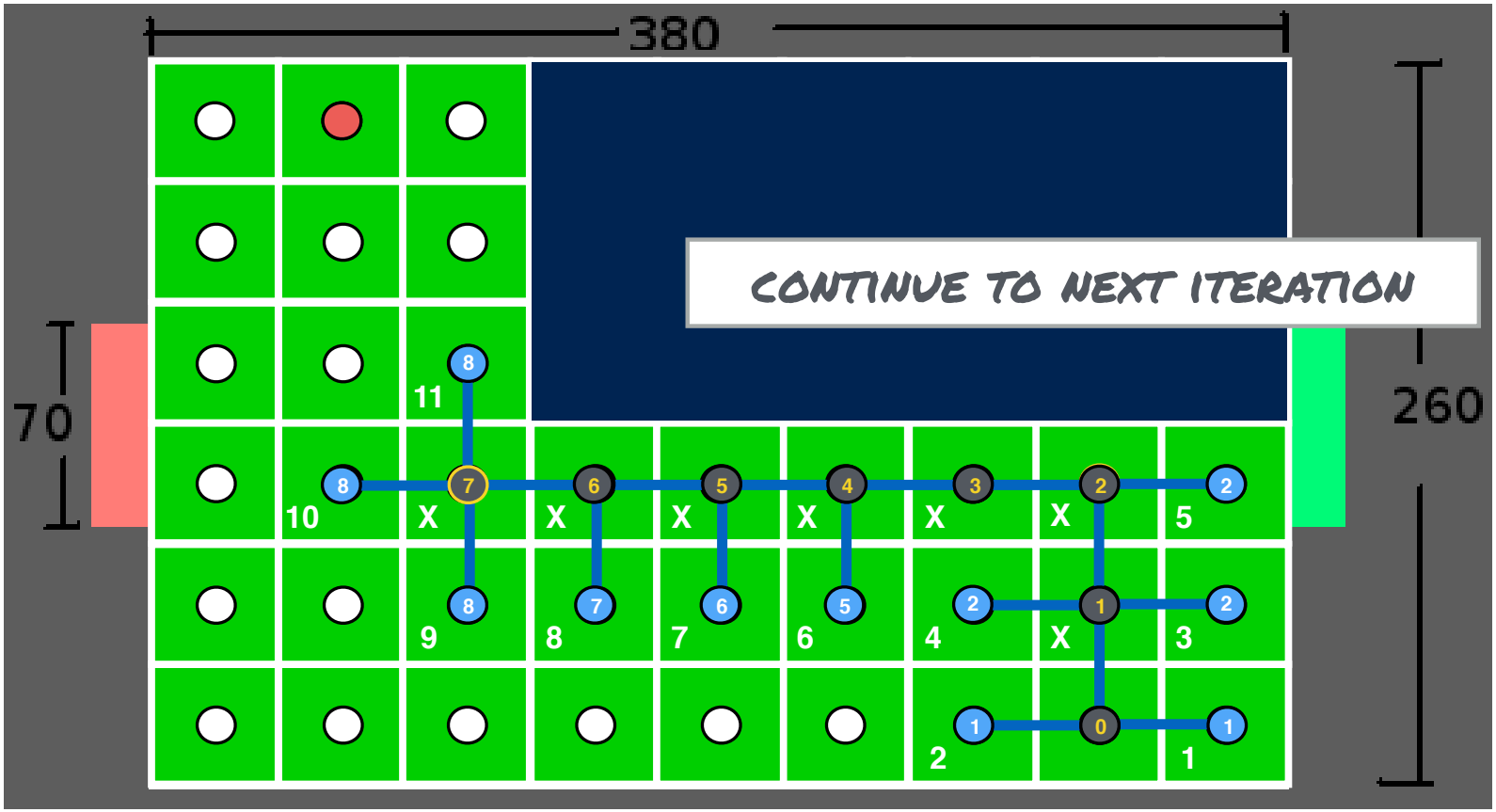
Depth-first search



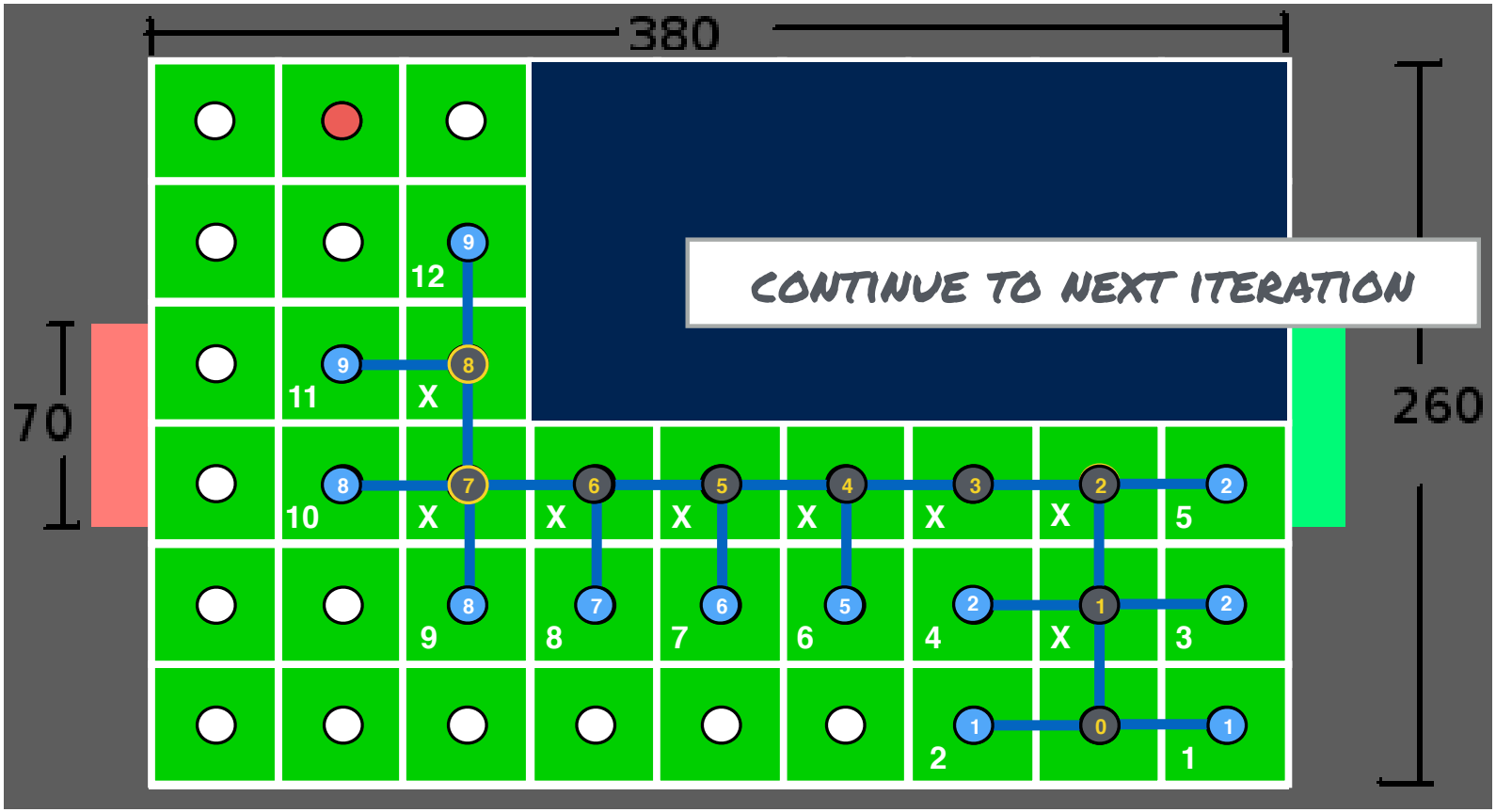
Depth-first search



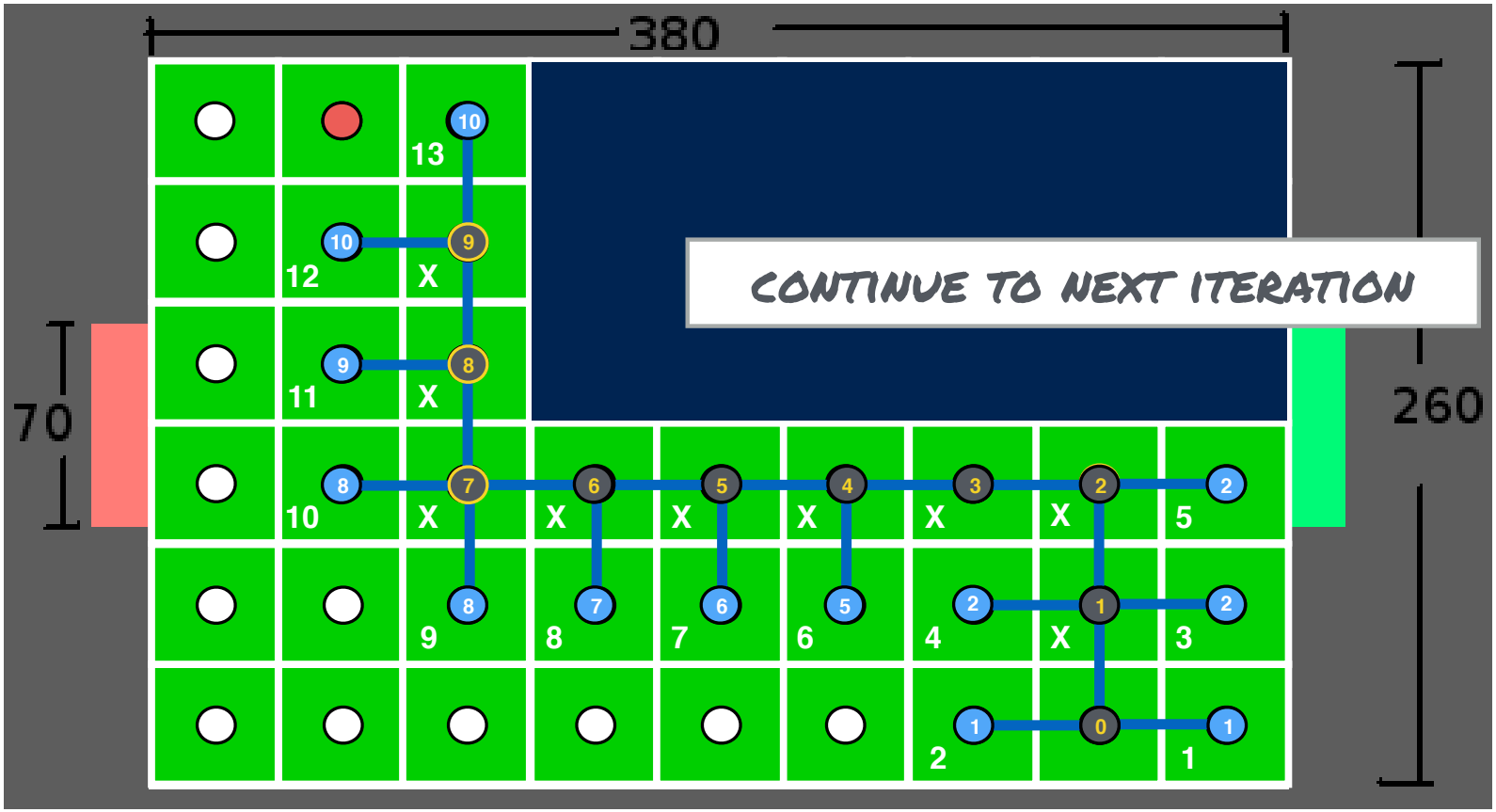
Depth-first search



Depth-first search

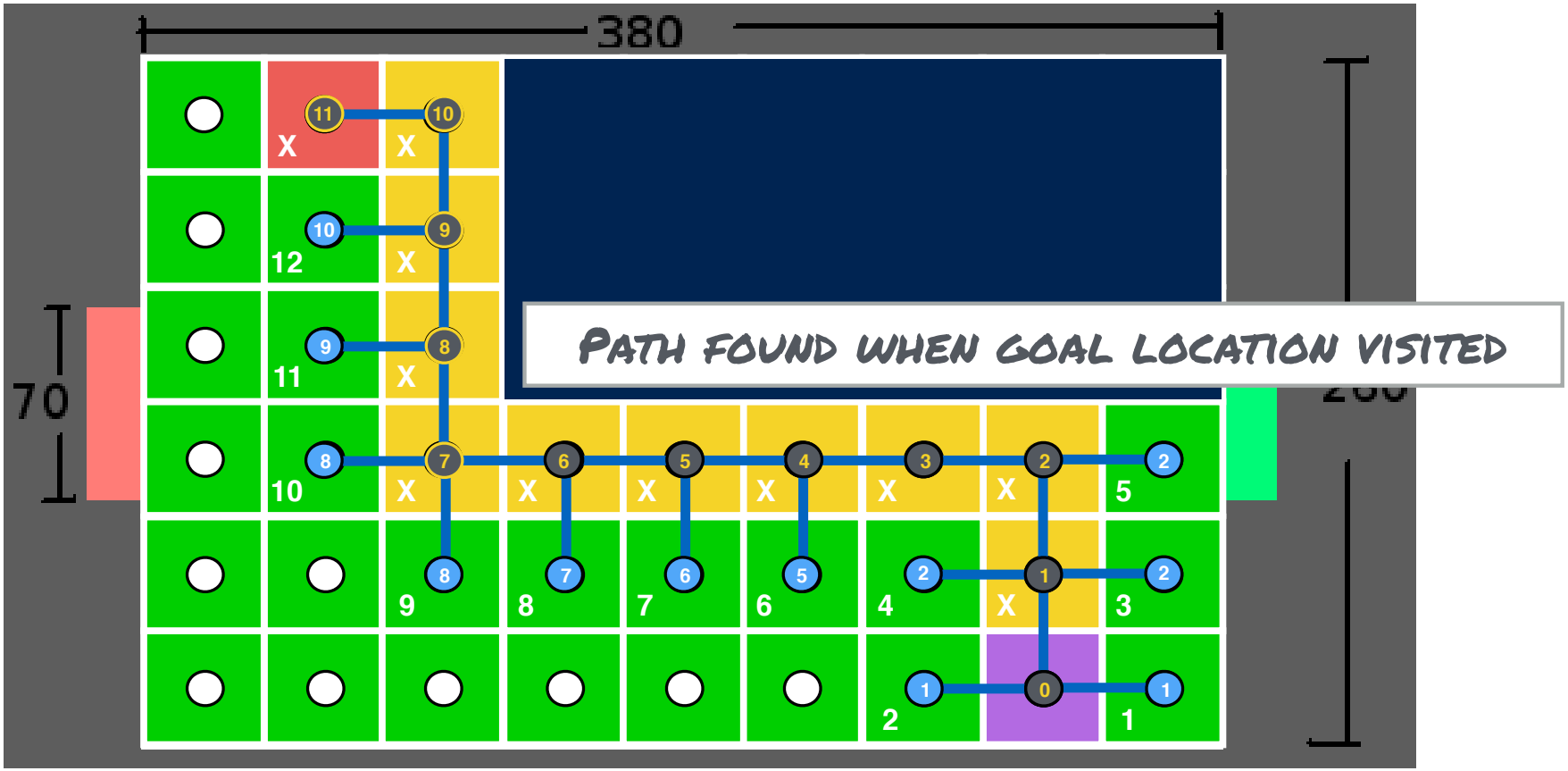


Depth-first search



Depth-first search

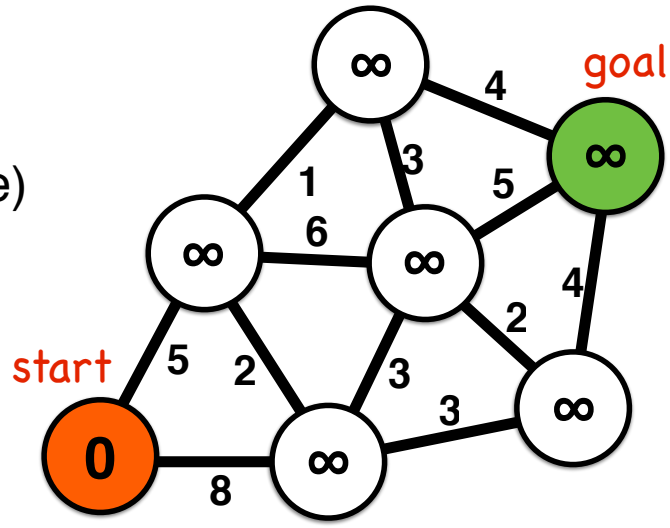
Depth-first search



Let’s turn this idea into code

Search algorithm template

all nodes ← {$dist_{start}$ ← infinity, $parent_{start}$ ← none, $visited_{start}$ ← false}
start_node ← {$dist_{start}$ ← 0, $parent_{start}$ ← none, $visited_{start}$ ← true}
visit_list ← start_node
    while visit_list != empty && current_node != goal
        cur_node ← highestPriority(visit_list)
        $visited_{cur\_node}$ ← true
        for each nbr in not_visited(adjacent(cur_node))
            add(nbr to visit_list)
            if $dist_{nbr}$ > $dist_{cur\_node}$ + distStraightLine(nbr,cur_node)
                $parent_{nbr}$ ← current_node
                $dist_{nbr}$ ← $dist_{cur\_node}$ + distStraightLine(nbr,cur_node)
            end if
        end for loop
    end while loop
output ← parent, distance



Search algorithm template

all nodes ← {$dist_{start}$ ← infinity, $parent_{start}$ ← none, $visited_{start}$ ← false}

start_node ← {$dist_{start}$ ← 0, $parent_{start}$ ← none, $visited_{start}$ ← 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

$parent_{nbr}$ ← current_node

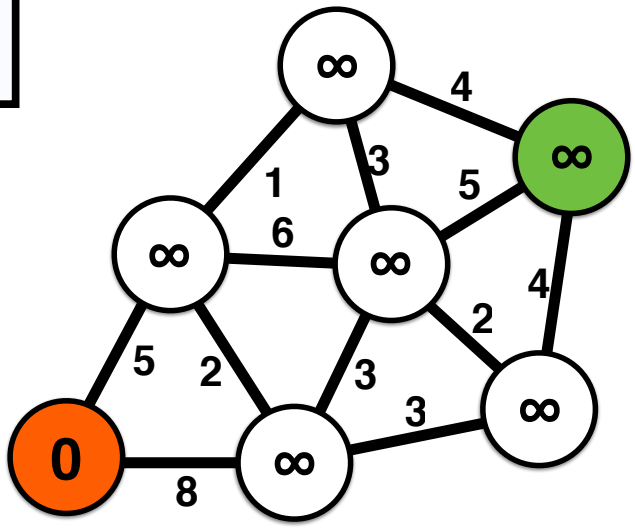
$dist_{nbr}$ ← $dist_{cur\_node}$ + distStraightLine(nbr,cur_node)

end if

end for loop

end while loop

output ← parent, distance

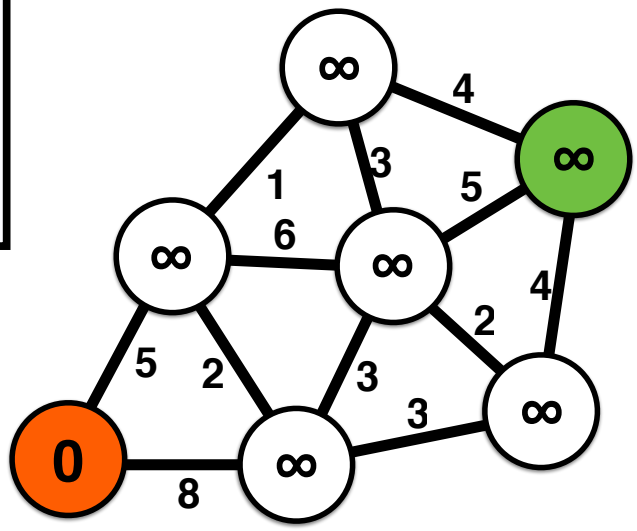


Search algorithm template

all nodes ← {$dist_{start}$ ← infinity, $parent_{start}$ ← none, $visited_{start}$ ← false}
start_node ← {$dist_{start}$ ← 0, $parent_{start}$ ← none, $visited_{start}$ ← true}
visit_list ← start_node
    while visit_list != empty && current_node != goal
        cur_node ← highestPriority(visit_list)
        $visited_{cur\_node}$ ← true

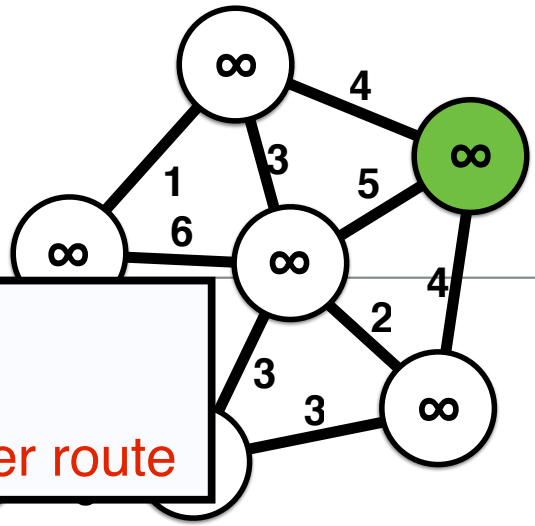
Main Loop
- visits every node to compute its distance and parent
- at each iteration:
  - select the node to visit based on its priority
  - remove current node from visit_list

end if
    end for loop
end while loop
output ← parent, distance



Search algorithm template

all nodes ← {$dist_{start}$ ← infinity, $parent_{start}$ ← none, $visited_{start}$ ← false}
start_node ← {$dist_{start}$ ← 0, $parent_{start}$ ← none, $visited_{start}$ ← true}
visit_list ← start_node
    while visit_list != empty && current_node != goal
        cur_node ← highestPriority(visit_list)
        $visited_{cur\_node}$ ← true
        for each nbr in not_visited(adjacent(cur_node))
            add(nbr to visit_list)
            if $dist_{nbr}$ > $dist_{cur\_node}$ + distStraightLine(nbr,cur_node)
                $parent_{nbr}$ ← current_node
                $dist_{nbr}$ ← $dist_{cur\_node}$ + distStraightLine(nbr,cur_node)
            end if



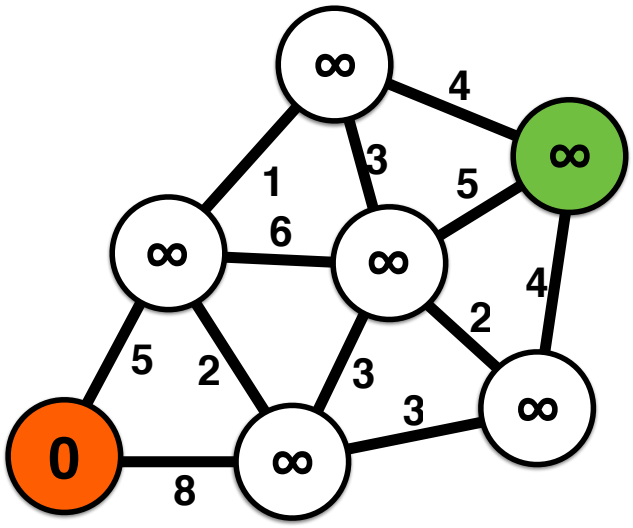
For each iteration on a single node

- 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 ← {$dist_{start}$ ← infinity, $parent_{start}$ ← none, $visited_{start}$ ← false}
start_node ← {$dist_{start}$ ← 0, $parent_{start}$ ← none, $visited_{start}$ ← true}
visit_list ← start_node
    while visit_list != empty && current_node != goal
        cur_node ← highestPriority(visit_list)
        $visited_{cur\_node}$ ← true
        for each nbr in not_visited(adjacent(cur_node))
            add(nbr to visit_list)
            if $dist_{nbr}$ > $dist_{cur\_node}$ + distance(nbr,cur_node)
                $parent_{nbr}$ ← current_node
                $dist_{nbr}$ ← $dist_{cur\_node}$ + distance(nbr,cur_node)
            end if
        end for loop
    end while loop
output ← parent, distance

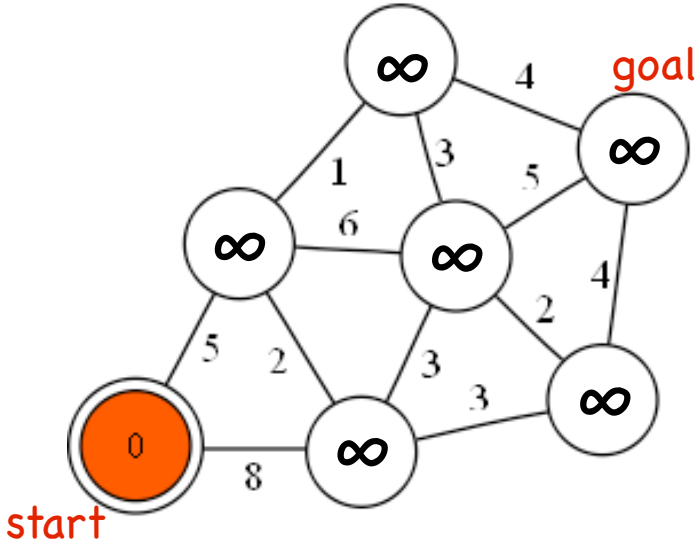
Output the resulting routes and distances



Depth-first search

Search algorithm template

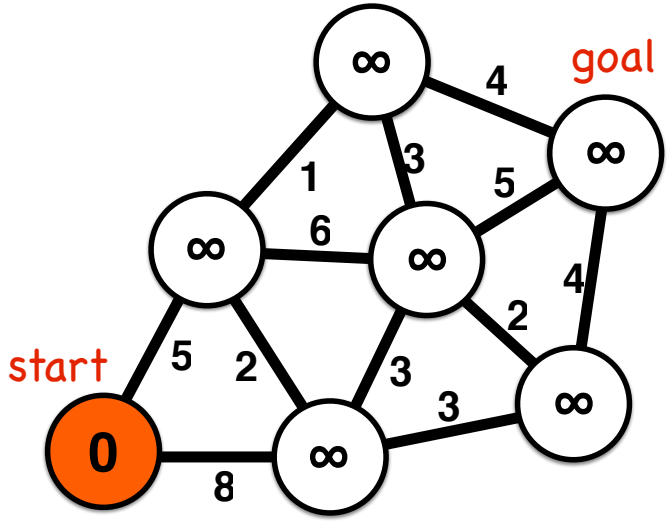
all nodes ← {$dist_{start}$ ← infinity, $parent_{start}$ ← none, $visited_{start}$ ← false}
start_node ← {$dist_{start}$ ← 0, $parent_{start}$ ← none, $visited_{start}$ ← true}
visit_list ← start_node
    while visit_list != empty && current_node != goal
        cur_node ← highestPriority(visit_list)
        $visited_{cur\_node}$ ← true
        for each nbr in not_visited(adjacent(cur_node))
            add(nbr to visit_list)
            if $dist_{nbr}$ > $dist_{cur\_node}$ + distance(nbr,cur_node)
                $parent_{nbr}$ ← current_node
                $dist_{nbr}$ ← $dist_{cur\_node}$ + distance(nbr,cur_node)
            end if
        end for loop
    end while loop
output ← parent, distance



Depth-first search

```
all nodes ← {dist_start ← infinity, parent_start ← none, visited_start ← false}
start_node ← {dist_start ← 0, parent_start ← none, visited_start ← true}
visit_stack ← start_node
    while visit_stack != empty && current_node != goal
        cur_node ← pop(visit_stack)
        visited_cur_node ← true
        for each nbr in not_visited(adjacent(cur_node))
            push(nbr to visit_stack)
            if dist_nbr > dist_cur_node + distance(nbr,cur_node)
                parent_nbr ← current_node
                dist_nbr ← dist_cur_node + distance(nbr,cur_node)
            end if
        end for loop
    end while loop
output ← parent, distance
```

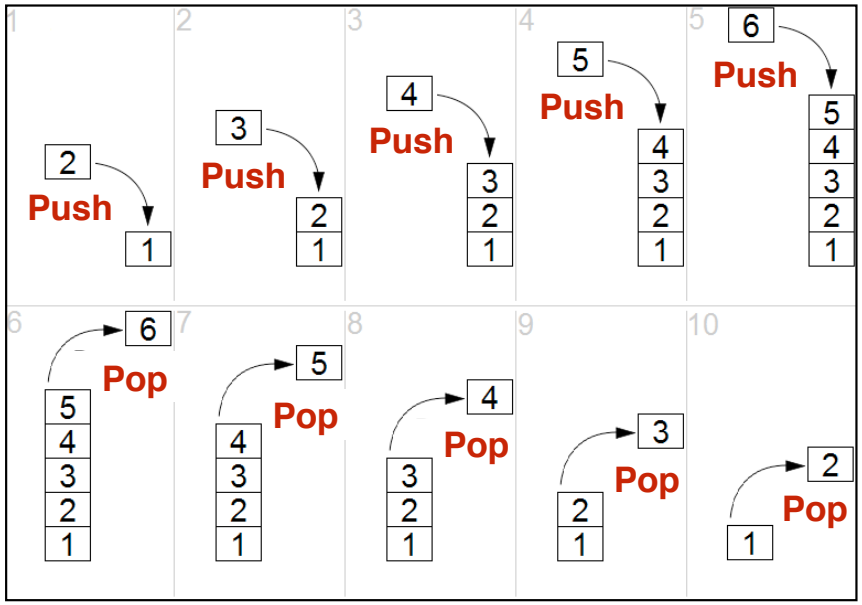
Priority: Most recent



Stack data structure

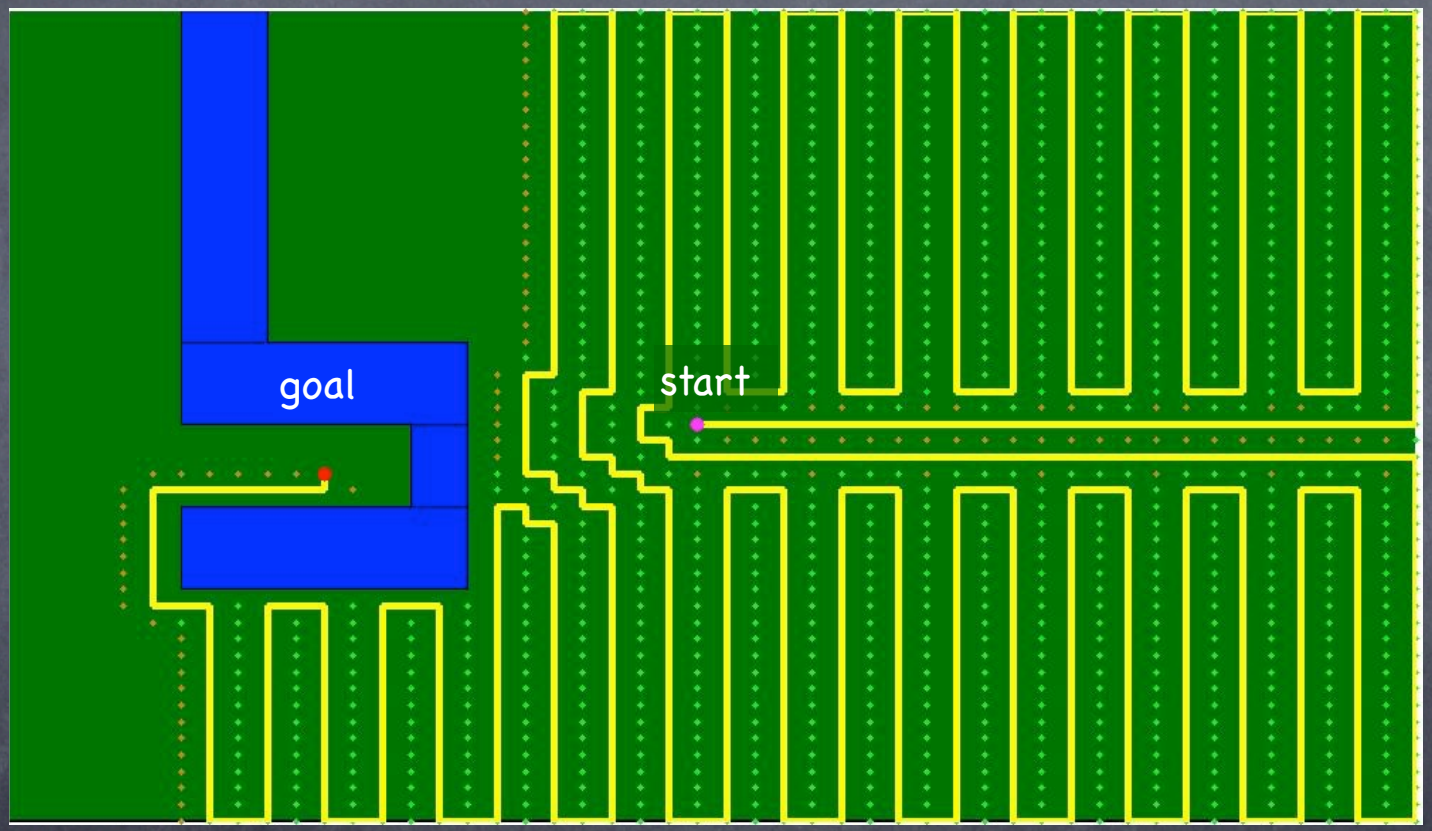
A stack is a “last in, first out” (or LIFO) structure, with two operations:
push: to add an element to the top of the stack
pop: to remove and element from the top of the stack

Stack example for reversing the order of six elements



matlab example: DFS

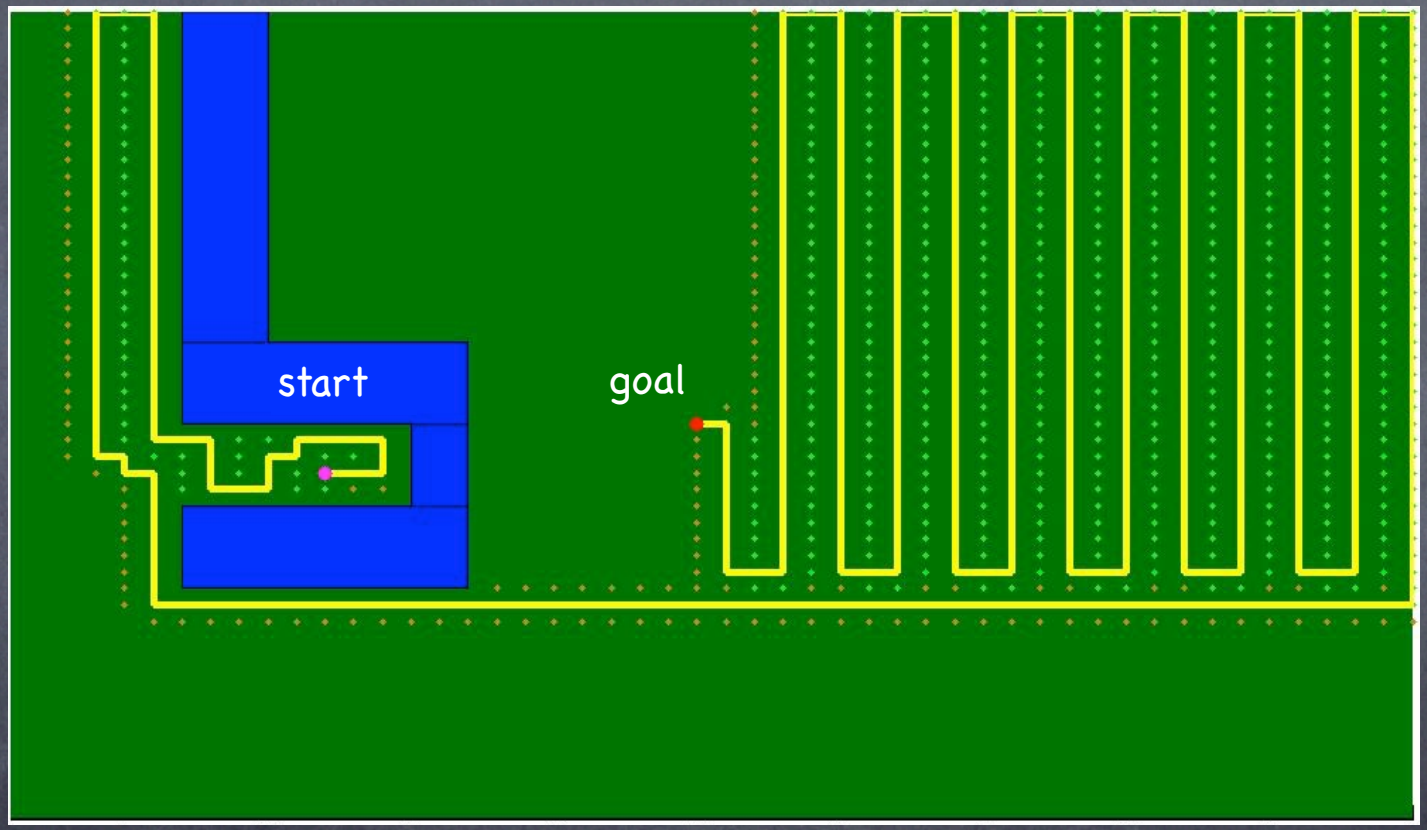
pathplan.m



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

matlab example: DFS

pathplan.m

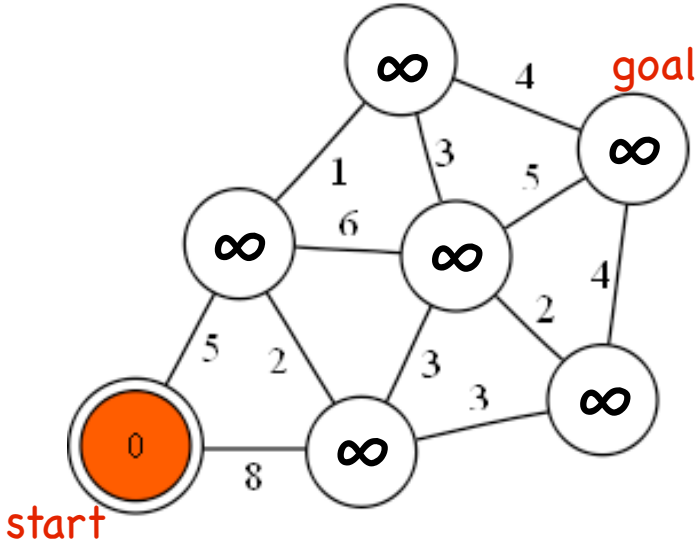


visited locations in light green
4-connected grid, right visited first

Breadth-first search

Search algorithm template

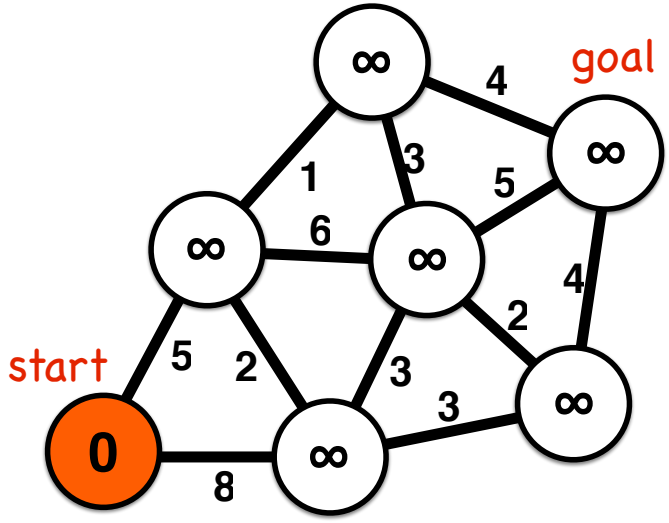
all nodes ← {$dist_{start}$ ← infinity, $parent_{start}$ ← none, $visited_{start}$ ← false}
start_node ← {$dist_{start}$ ← 0, $parent_{start}$ ← none, $visited_{start}$ ← true}
visit_list ← start_node
    while visit_list != empty && current_node != goal
        cur_node ← highestPriority(visit_list)
        $visited_{cur\_node}$ ← true
        for each nbr in not_visited(adjacent(cur_node))
            add(nbr to visit_list)
            if $dist_{nbr}$ > $dist_{cur\_node}$ + distance(nbr,cur_node)
                $parent_{nbr}$ ← current_node
                $dist_{nbr}$ ← $dist_{cur\_node}$ + distance(nbr,cur_node)
            end if
        end for loop
    end while loop
output ← parent, distance



Breadth-first search

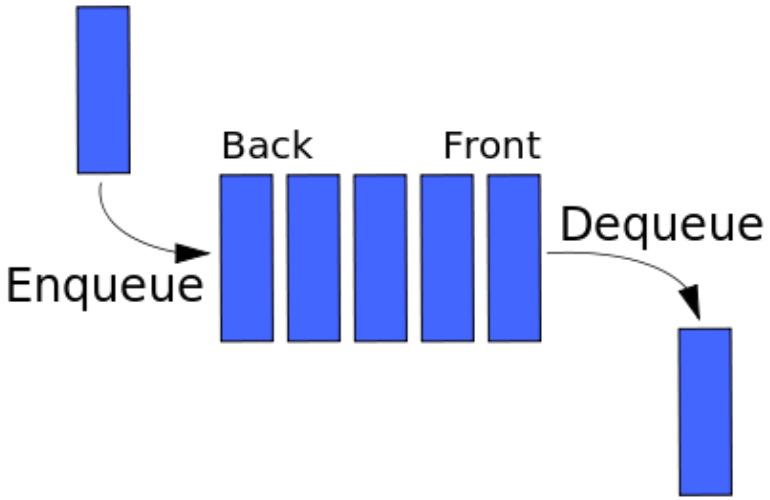
```
all nodes ← {dist_start ← infinity, parent_start ← none, visited_start ← false}
start_node ← {dist_start ← 0, parent_start ← none, visited_start ← true}
visit_queue ← start_node
    while visit_queue != empty && current_node != goal
        cur_node ← dequeue(visit_queue)
        visited_cur_node ← true
        for each nbr in not_visited(adjacent(cur_node))
            enqueue(nbr to visit_queue)
            if dist_nbr > dist_cur_node + distance(nbr,cur_node)
                parent_nbr ← current_node
                dist_nbr ← dist_cur_node + distance(nbr,cur_node)
            end if
        end for loop
    end while loop
output ← parent, distance
```

Priority: Least recent



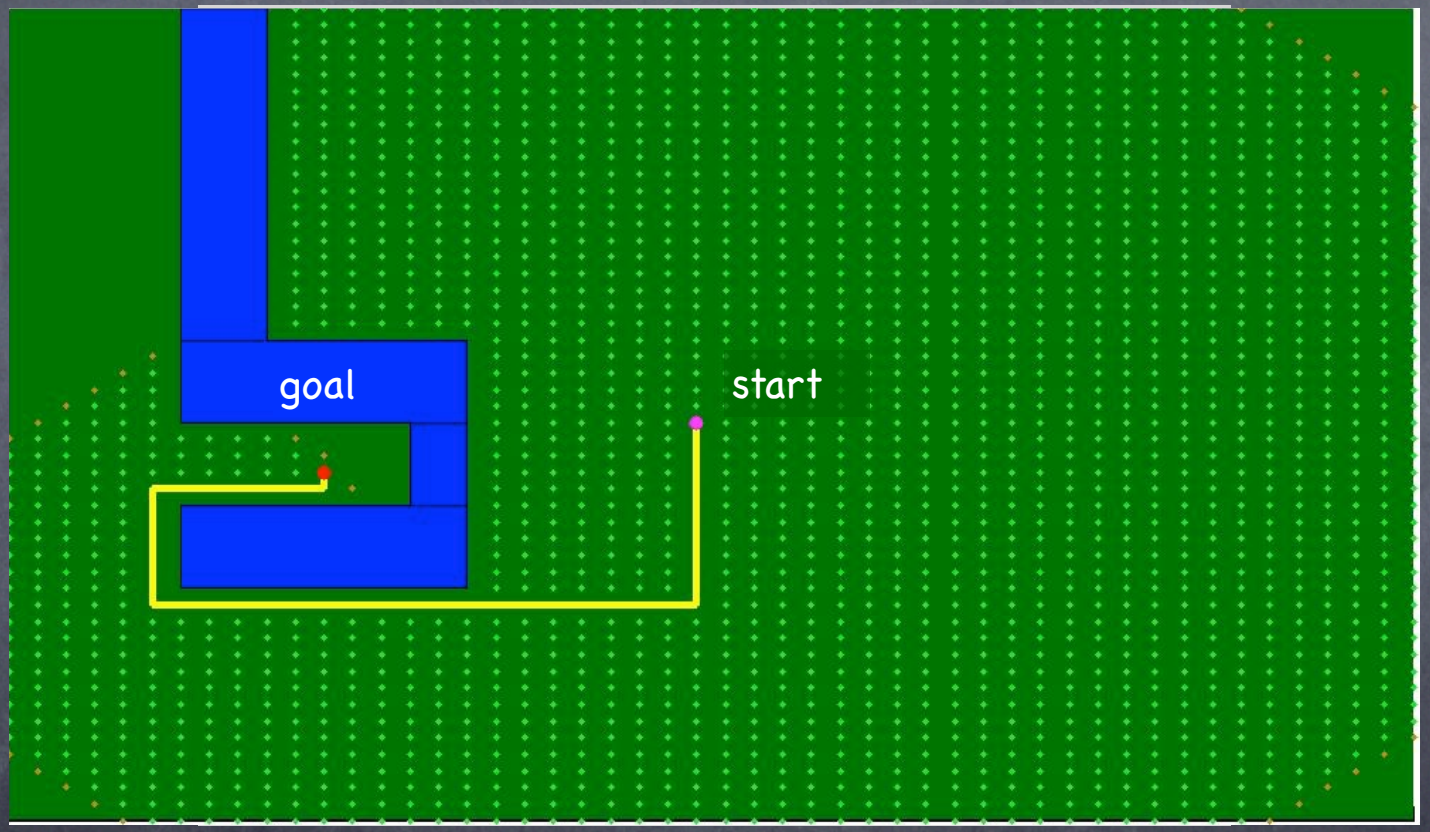
Queue data structure

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

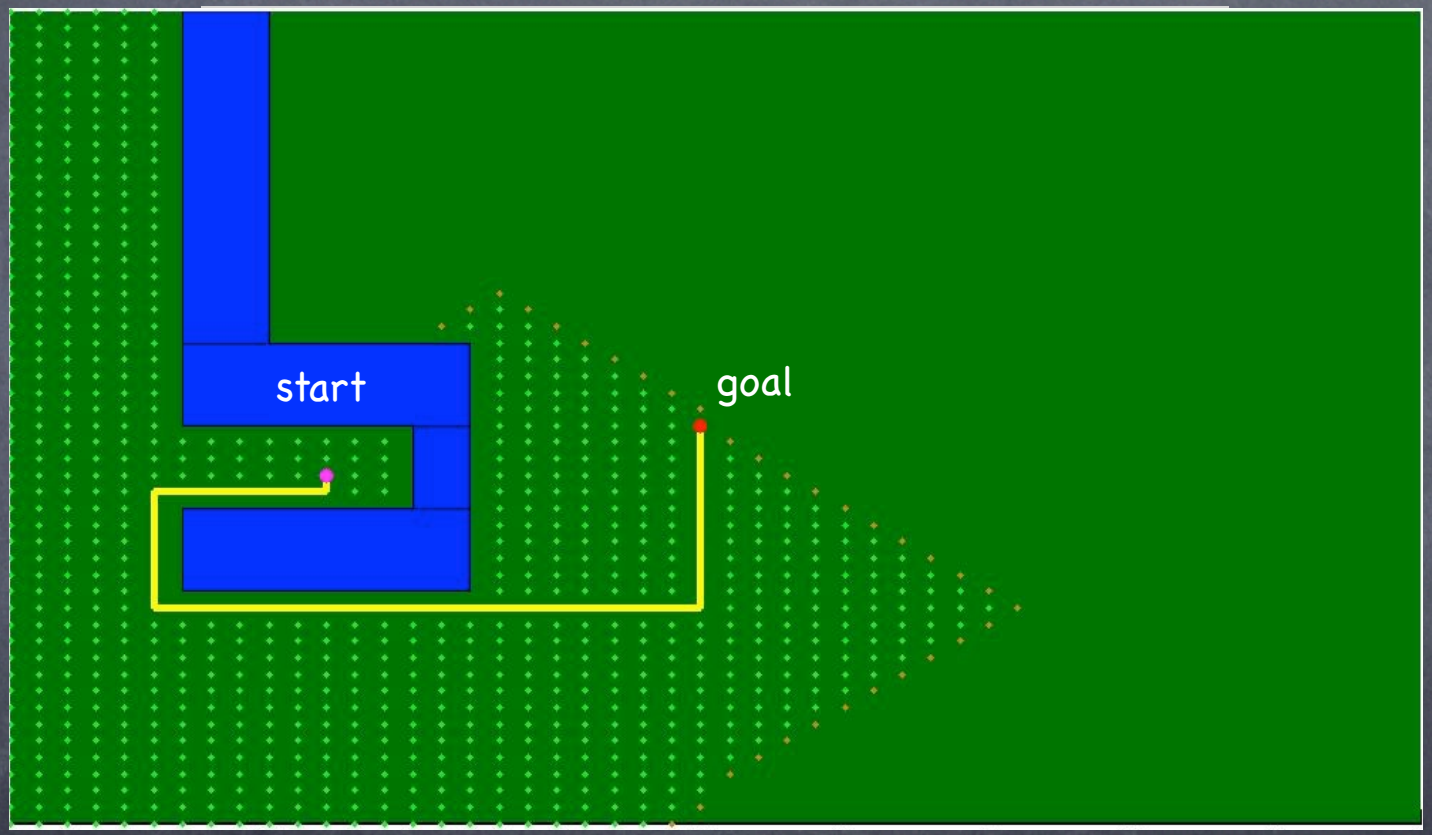
pathplan.m



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

matlab example: BFS

pathplan.m

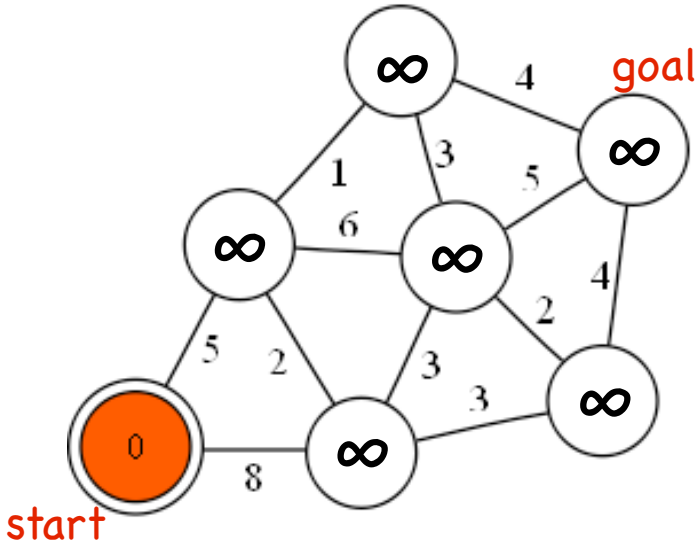


visited locations in light green
4-connected grid, right visited first

Dijkstra shortest path

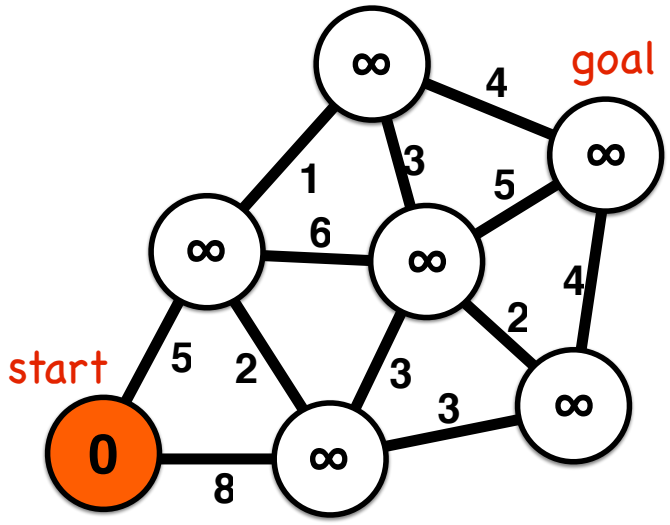
Search algorithm template

all nodes ← {$dist_{start}$ ← infinity, $parent_{start}$ ← none, $visited_{start}$ ← false}
start_node ← {$dist_{start}$ ← 0, $parent_{start}$ ← none, $visited_{start}$ ← true}
visit_list ← start_node
    while visit_list != empty && current_node != goal
        cur_node ← highestPriority(visit_list)
        $visited_{cur\_node}$ ← true
        for each nbr in not_visited(adjacent(cur_node))
            add(nbr to visit_list)
            if $dist_{nbr}$ > $dist_{cur\_node}$ + distance(nbr,cur_node)
                $parent_{nbr}$ ← current_node
                $dist_{nbr}$ ← $dist_{cur\_node}$ + distance(nbr,cur_node)
            end if
        end for loop
    end while loop
output ← parent, distance



Dijkstra shortest path algorithm

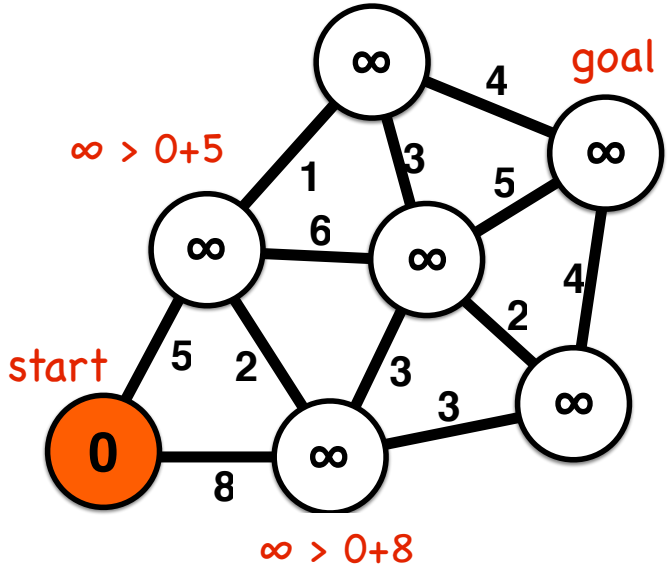
all nodes ← {$dist_{start}$ ← infinity, $parent_{start}$ ← none, $visited_{start}$ ← false}
start_node ← {$dist_{start}$ ← 0, $parent_{start}$ ← none, $visited_{start}$ ← true}
visit_queue ← start_node
    while visit_queue != empty ~~&& current_node != goal~~
        cur_node ← min_distance(visit_queue) ← Priority: Minimum distance
        $visited_{cur\_node}$ ← true
        for each nbr in not_visited(adjacent(cur_node))
            enqueue(nbr to visit_queue)
            if $dist_{nbr}$ > $dist_{cur\_node}$ + distance(nbr,cur_node)
                $parent_{nbr}$ ← current_node
                $dist_{nbr}$ ← $dist_{cur\_node}$ + distance(nbr,cur_node)
            end if
        end for loop
    end while loop
output ← parent, distance



Dijkstra shortest path algorithm

all nodes ← {$dist_{start}$ ← infinity, $parent_{start}$ ← none, $visited_{start}$ ← false}
start_node ← {$dist_{start}$ ← 0, $parent_{start}$ ← none, $visited_{start}$ ← true}
visit_queue ← start_node
    while visit_queue != empty ~~&& current_node != goal~~
        cur_node ← min_distance(visit_queue)
        $visited_{cur\_node}$ ← true
        for each nbr in not_visited(adjacent(cur_node))
            enqueue(nbr to visit_queue)
            if $dist_{nbr}$ > $dist_{cur\_node}$ + distance(nbr,cur_node)
                $parent_{nbr}$ ← current_node
                $dist_{nbr}$ ← $dist_{cur\_node}$ + distance(nbr,cur_node)
            end if
        end for loop
    end while loop
output ← parent, distance

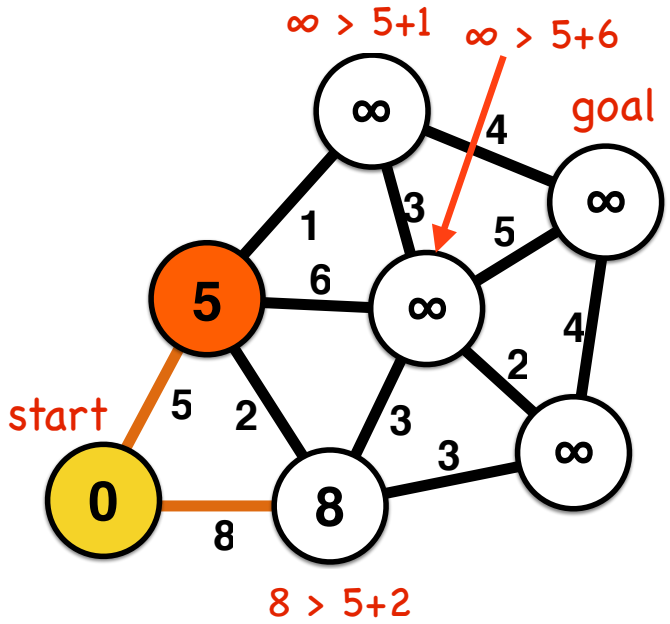
Dijkstra walkthrough



Dijkstra shortest path algorithm

all nodes ← {$dist_{start}$ ← infinity, $parent_{start}$ ← none, $visited_{start}$ ← false}
start_node ← {$dist_{start}$ ← 0, $parent_{start}$ ← none, $visited_{start}$ ← true}
visit_queue ← start_node
    while visit_queue != empty ~~&& current_node != goal~~
        cur_node ← min_distance(visit_queue)
        $visited_{cur\_node}$ ← true
        for each nbr in not_visited(adjacent(cur_node))
            enqueue(nbr to visit_queue)
            if $dist_{nbr}$ > $dist_{cur\_node}$ + distance(nbr,cur_node)
                $parent_{nbr}$ ← current_node
                $dist_{nbr}$ ← $dist_{cur\_node}$ + distance(nbr,cur_node)
            end if
        end for loop
    end while loop
output ← parent, distance

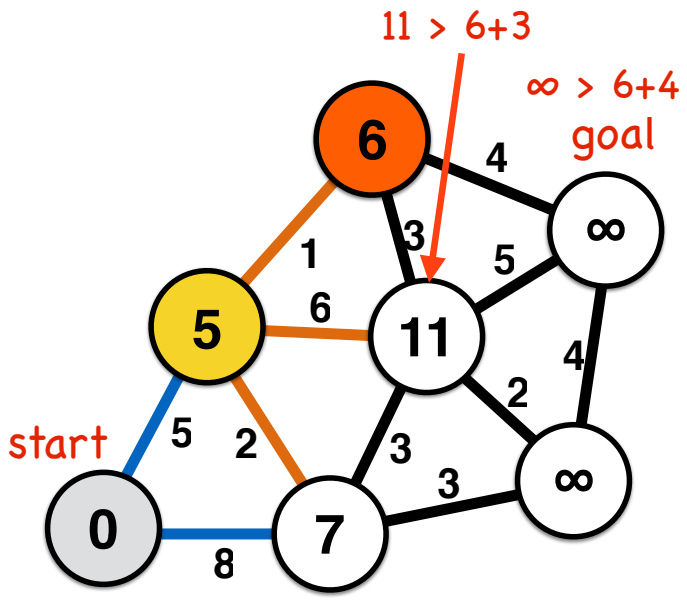
Dijkstra walkthrough



Dijkstra shortest path algorithm

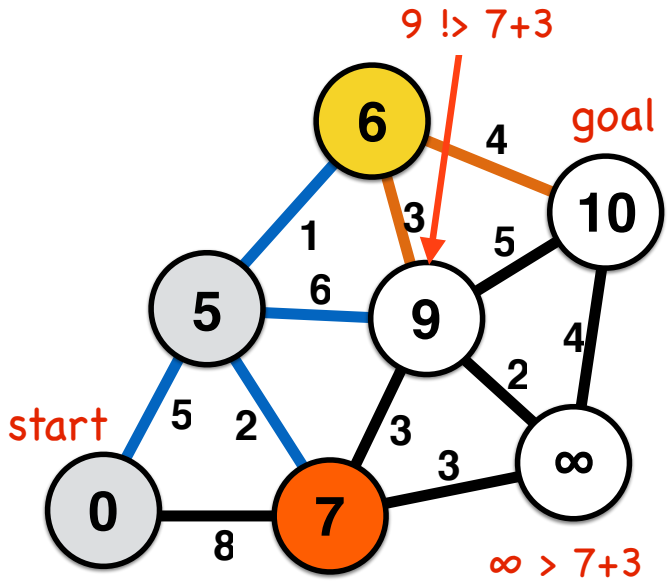
all nodes ← {$dist_{start}$ ← infinity, $parent_{start}$ ← none, $visited_{start}$ ← false}
start_node ← {$dist_{start}$ ← 0, $parent_{start}$ ← none, $visited_{start}$ ← true}
visit_queue ← start_node
    while visit_queue != empty ~~&& current_node != goal~~
        cur_node ← min_distance(visit_queue)
        $visited_{cur\_node}$ ← true
        for each nbr in not_visited(adjacent(cur_node))
            enqueue(nbr to visit_queue)
            if $dist_{nbr}$ > $dist_{cur\_node}$ + distance(nbr,cur_node)
                $parent_{nbr}$ ← current_node
                $dist_{nbr}$ ← $dist_{cur\_node}$ + distance(nbr,cur_node)
            end if
        end for loop
    end while loop
output ← parent, distance

Dijkstra walkthrough



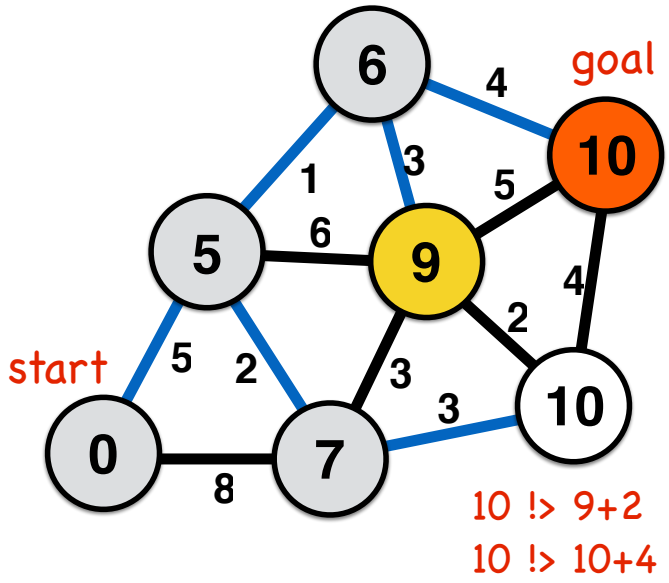
Dijkstra shortest path algorithm

all nodes ← {$dist_{start}$ ← infinity, $parent_{start}$ ← none, $visited_{start}$ ← false}
start_node ← {$dist_{start}$ ← 0, $parent_{start}$ ← none, $visited_{start}$ ← true}
visit_queue ← start_node
    while visit_queue != empty ~~&& current_node != goal~~
        cur_node ← min_distance(visit_queue)
        $visited_{cur\_node}$ ← true
        for each nbr in not_visited(adjacent(cur_node))
            enqueue(nbr to visit_queue)
            if $dist_{nbr}$ > $dist_{cur\_node}$ + distance(nbr,cur_node)
                $parent_{nbr}$ ← current_node
                $dist_{nbr}$ ← $dist_{cur\_node}$ + distance(nbr,cur_node)
            end if
        end for loop
    end while loop
output ← parent, distance



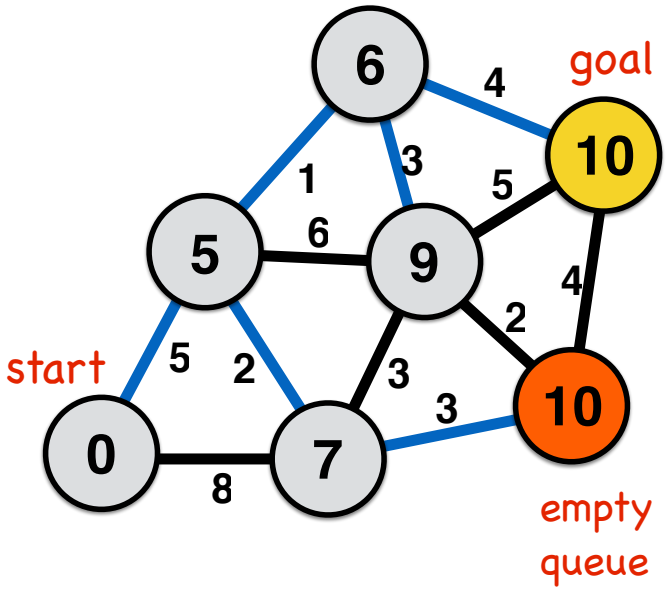
Dijkstra shortest path algorithm

all nodes ← {$dist_{start}$ ← infinity, $parent_{start}$ ← none, $visited_{start}$ ← false}
start_node ← {$dist_{start}$ ← 0, $parent_{start}$ ← none, $visited_{start}$ ← true}
visit_queue ← start_node
    while visit_queue != empty ~~&& current_node != goal~~
        cur_node ← min_distance(visit_queue)
        $visited_{cur\_node}$ ← true
        for each nbr in not_visited(adjacent(cur_node))
            enqueue(nbr to visit_queue)
            if $dist_{nbr}$ > $dist_{cur\_node}$ + distance(nbr,cur_node)
                $parent_{nbr}$ ← current_node
                $dist_{nbr}$ ← $dist_{cur\_node}$ + distance(nbr,cur_node)
            end if
        end for loop
    end while loop
output ← parent, distance



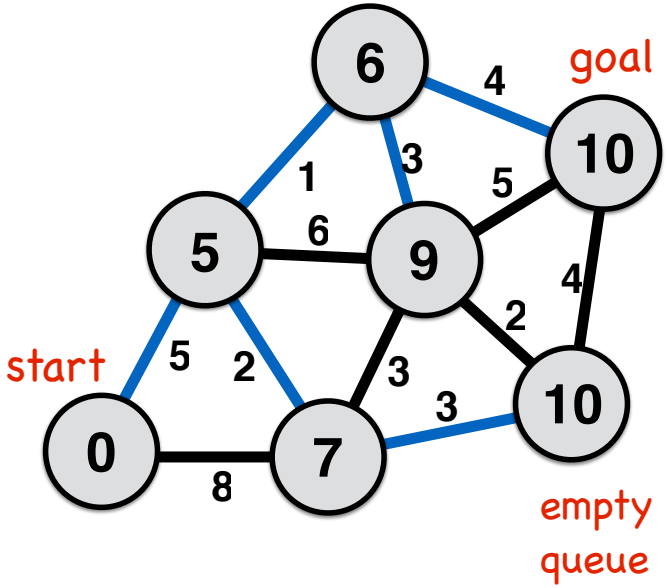
Dijkstra shortest path algorithm

all nodes ← {$dist_{start}$ ← infinity, $parent_{start}$ ← none, $visited_{start}$ ← false}
start_node ← {$dist_{start}$ ← 0, $parent_{start}$ ← none, $visited_{start}$ ← true}
visit_queue ← start_node
    while visit_queue != empty ~~&& current_node != goal~~
        cur_node ← min_distance(visit_queue)
        $visited_{cur\_node}$ ← true
        for each nbr in not_visited(adjacent(cur_node))
            enqueue(nbr to visit_queue)
            if $dist_{nbr}$ > $dist_{cur\_node}$ + distance(nbr,cur_node)
                $parent_{nbr}$ ← current_node
                $dist_{nbr}$ ← $dist_{cur\_node}$ + distance(nbr,cur_node)
            end if
        end for loop
    end while loop
output ← parent, distance



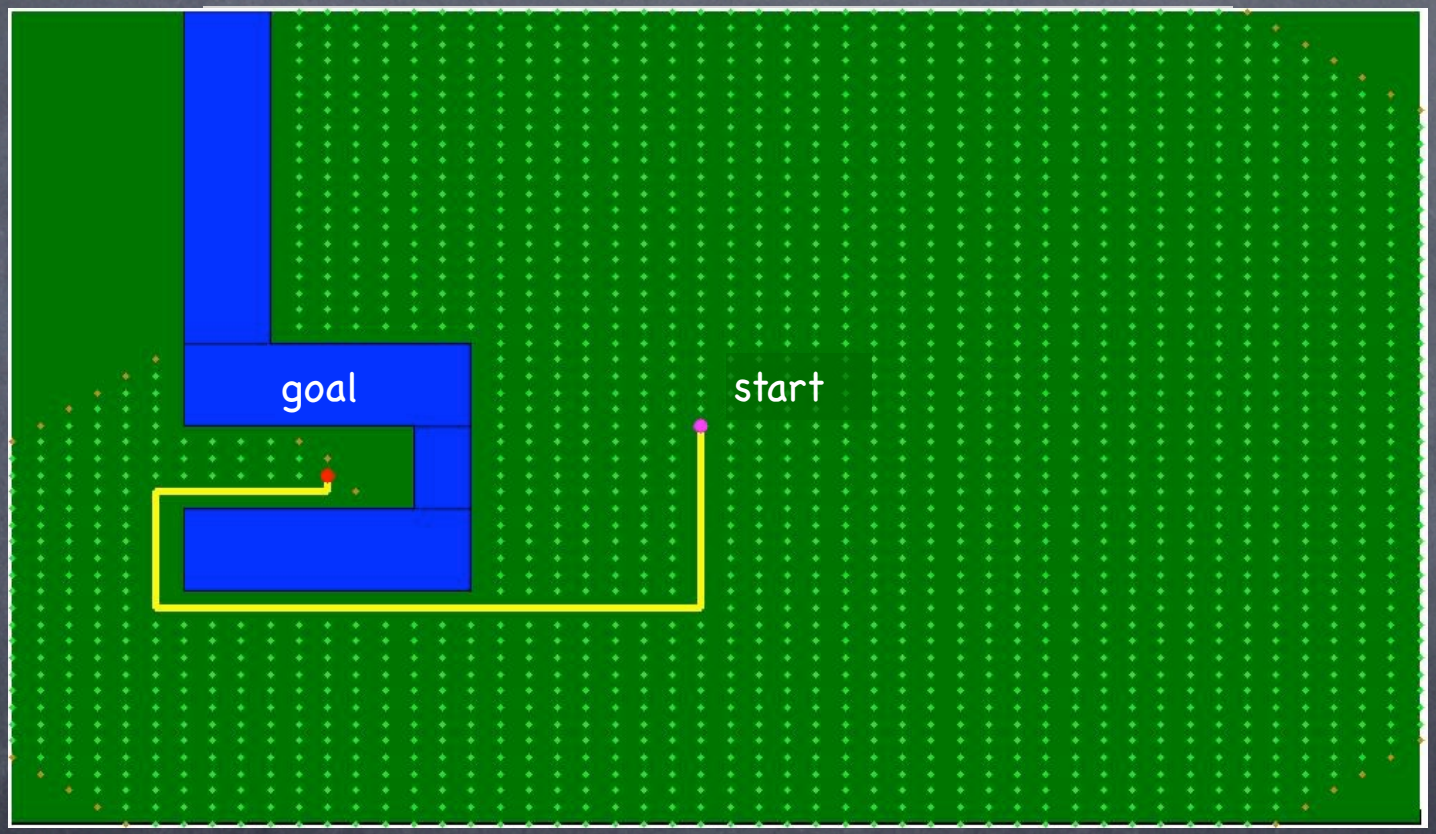
Dijkstra shortest path algorithm

all nodes ← {$dist_{start}$ ← infinity, $parent_{start}$ ← none, $visited_{start}$ ← false}
start_node ← {$dist_{start}$ ← 0, $parent_{start}$ ← none, $visited_{start}$ ← true}
visit_queue ← start_node
    while visit_queue != empty ~~&& current_node != goal~~
        cur_node ← min_distance(visit_queue)
        $visited_{cur\_node}$ ← true
        for each nbr in not_visited(adjacent(cur_node))
            enqueue(nbr to visit_queue)
            if $dist_{nbr}$ > $dist_{cur\_node}$ + distance(nbr,cur_node)
                $parent_{nbr}$ ← current_node
                $dist_{nbr}$ ← $dist_{cur\_node}$ + distance(nbr,cur_node)
            end if
        end for loop
    end while loop
output ← parent, distance



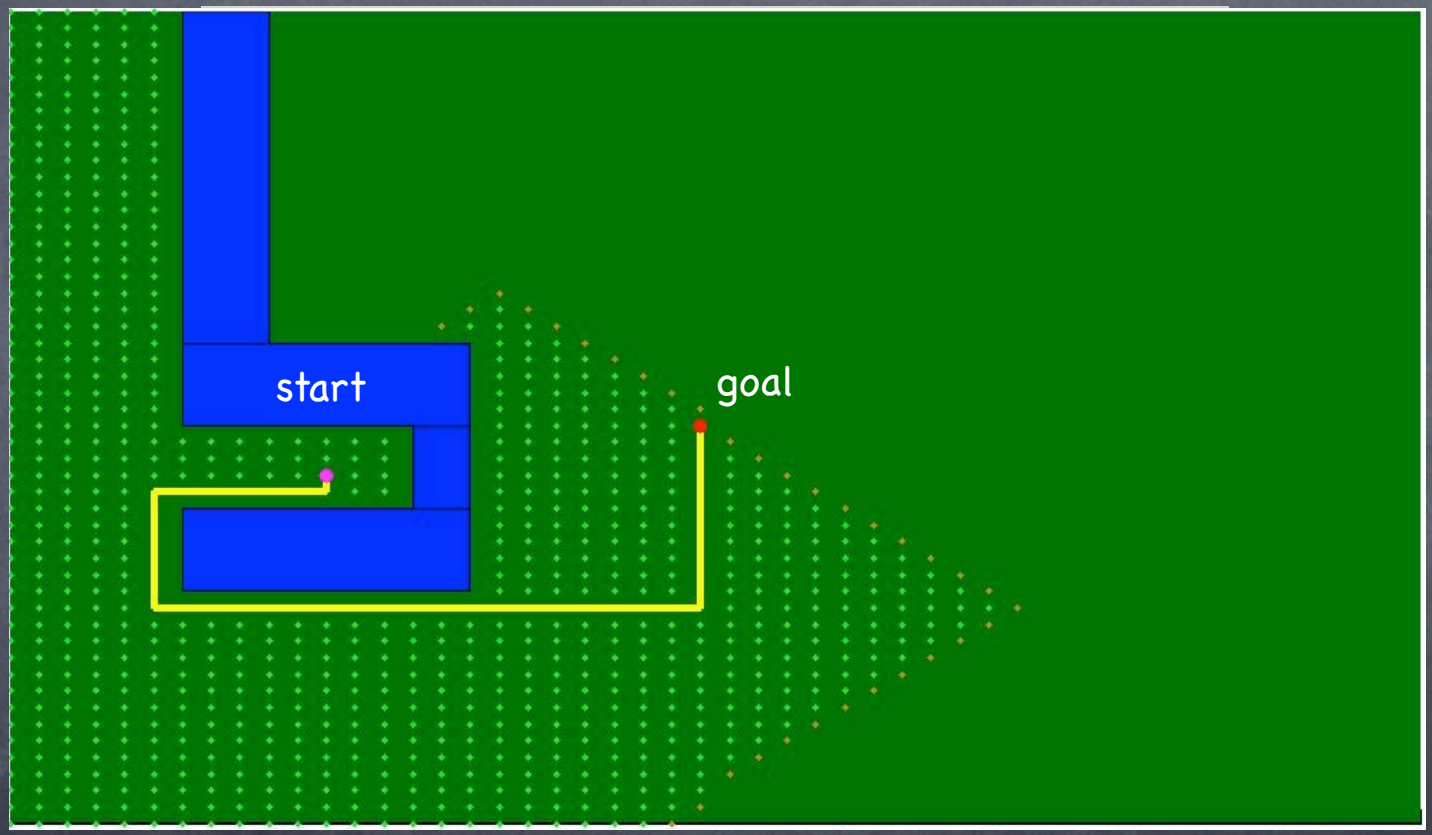
matlab example: Dijkstra

pathplan.m



matlab example: Dijkstra

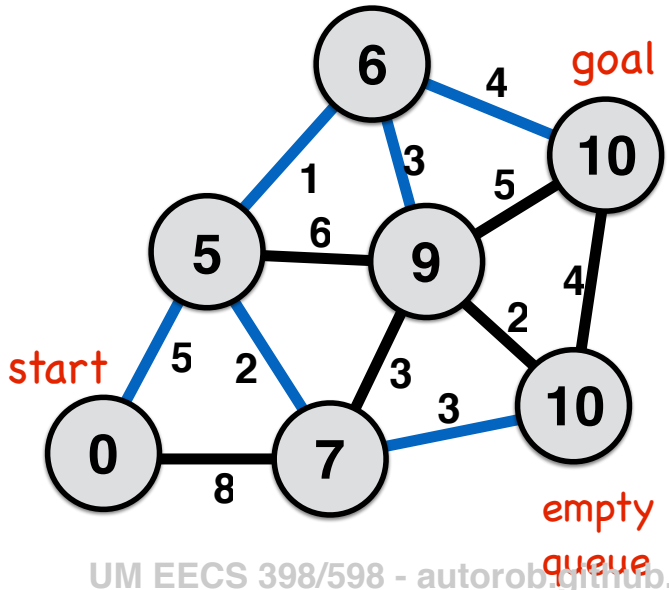
pathplan.m



A-star shortest path

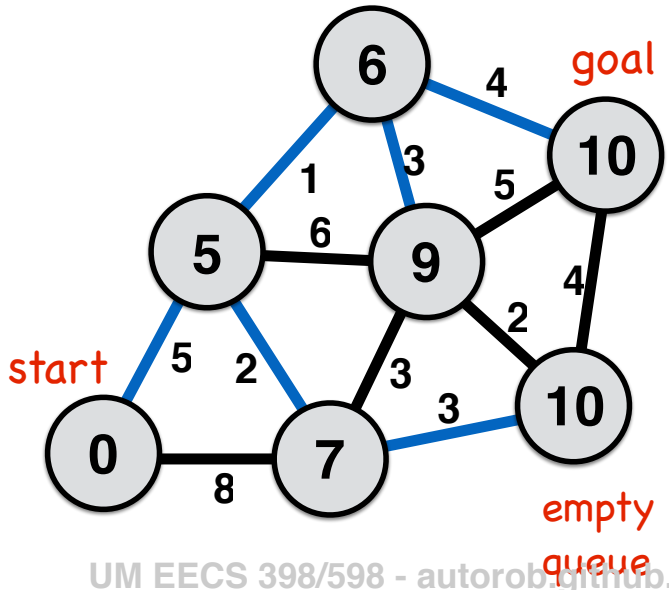
Dijkstra shortest path algorithm

all nodes ← {$dist_{start}$ ← infinity, $parent_{start}$ ← none, $visited_{start}$ ← false}
start_node ← {$dist_{start}$ ← 0, $parent_{start}$ ← none, $visited_{start}$ ← true}
visit_queue ← start_node
    while visit_queue != empty ~~&& current_node != goal~~
        cur_node ← min_distance(visit_queue)
        $visited_{cur\_node}$ ← true
        for each nbr in not_visited(adjacent(cur_node))
            enqueue(nbr to visit_queue)
            if $dist_{nbr}$ > $dist_{cur\_node}$ + distance(nbr,cur_node)
                $parent_{nbr}$ ← current_node
                $dist_{nbr}$ ← $dist_{cur\_node}$ + distance(nbr,cur_node)
            end if
        end for loop
    end while loop
output ← parent, distance



A-star shortest path algorithm

all nodes ← {$dist_{start}$ ← infinity, $parent_{start}$ ← none, $visited_{start}$ ← false}
start_node ← {$dist_{start}$ ← 0, $parent_{start}$ ← none, $visited_{start}$ ← true}
visit_queue ← start_node
    while (visit_queue != empty) && current_node != goal
        cur_node ← dequeue(visit_queue, f_score)
        $visited_{cur\_node}$ ← true
        for each nbr in not_visited(adjacent(cur_node))
            enqueue(nbr to visit_queue)
            if $dist_{nbr}$ > $dist_{cur\_node}$ + distance(nbr,cur_node)
                $parent_{nbr}$ ← current_node
                $dist_{nbr}$ ← $dist_{cur\_node}$ + distance(nbr,cur_node)
                f_score ← $distance_{nbr}$ + $line\_distance_{nbr,goal}$
            end if
        end for loop
    end while loop
output ← parent, distance



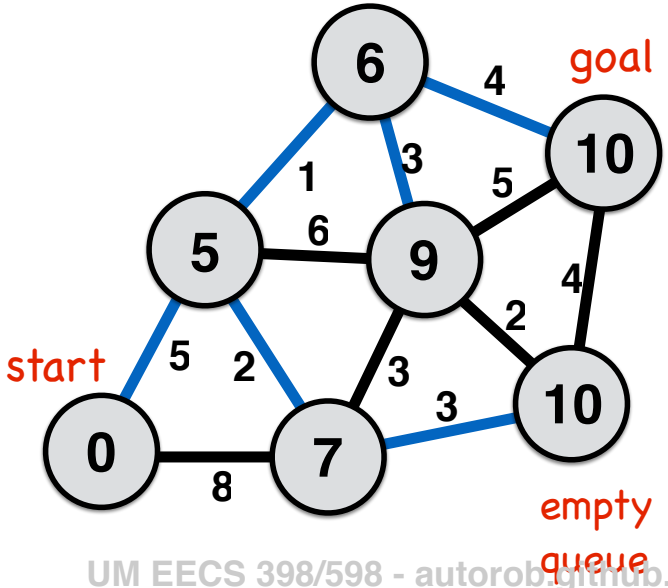
A-star shortest path algorithm

all nodes ← {$dist_{start}$ ← infinity, $parent_{start}$ ← none, $visited_{start}$ ← false}
start_node ← {$dist_{start}$ ← 0, $parent_{start}$ ← none, $visited_{start}$ ← true}
visit_queue ← start_node
    while (visit_queue != empty) && current_node != goal
        cur_node ← dequeue(visit_queue, f_score)
        $visited_{cur\_node}$ ← true
        for each nbr in not_visited(adjacent(cur_node))
            enqueue(nbr to visit_queue)
            if $dist_{nbr}$ > $dist_{cur\_node}$ + distance(nbr,cur_node)
                $parent_{nbr}$ ← current_node
                $dist_{nbr}$ ← $dist_{cur\_node}$ + distance(nbr,cur_node)
                f_score ← $distance_{nbr}$ + $line\_distance_{nbr,goal}$
            end if
        end for loop
    end while loop
output ← 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 ← {$dist_{start}$ ← infinity, $parent_{start}$ ← none, $visited_{start}$ ← false}
start_node ← {$dist_{start}$ ← 0, $parent_{start}$ ← none, $visited_{start}$ ← true}
visit_queue ← start_node
while (visit_queue != empty) && current_node != goal
cur_node ← dequeue(visit_queue, f_score)

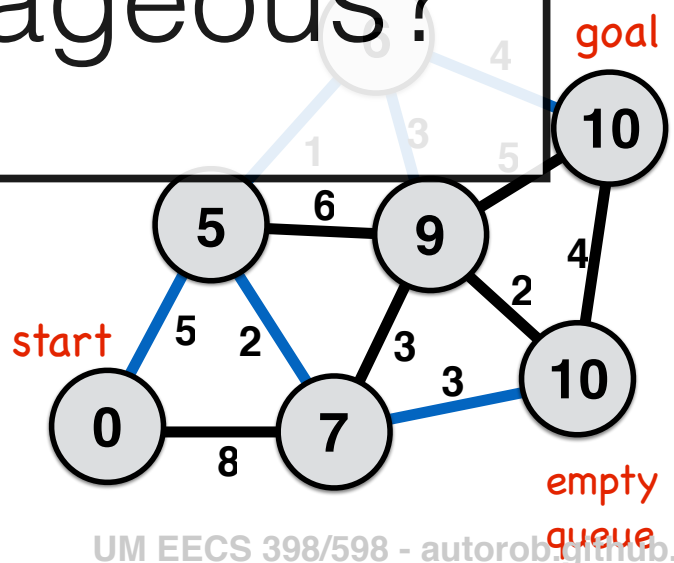
priority queue wrt. f_score (implement min binary heap)

$visited_{cur\_node}$ ← true
for each nbr in not_visited(adjacent(cur_node))
enqueue(nbr to visit_queue)
if $dist_{nbr} > dist_{cur\_node}$ + distance(nbr,cur_node)
$parent_{nbr}$ ← current_node
$dist_{nbr}$ ← $dist_{cur\_node}$ + distance(nbr,cur_node)
f_score ← $distance_{nbr}$ + $line\_distance_{nbr,goal}$
end if
end for loop
end while loop
output ← parent, distance

Why is A-star advantageous?

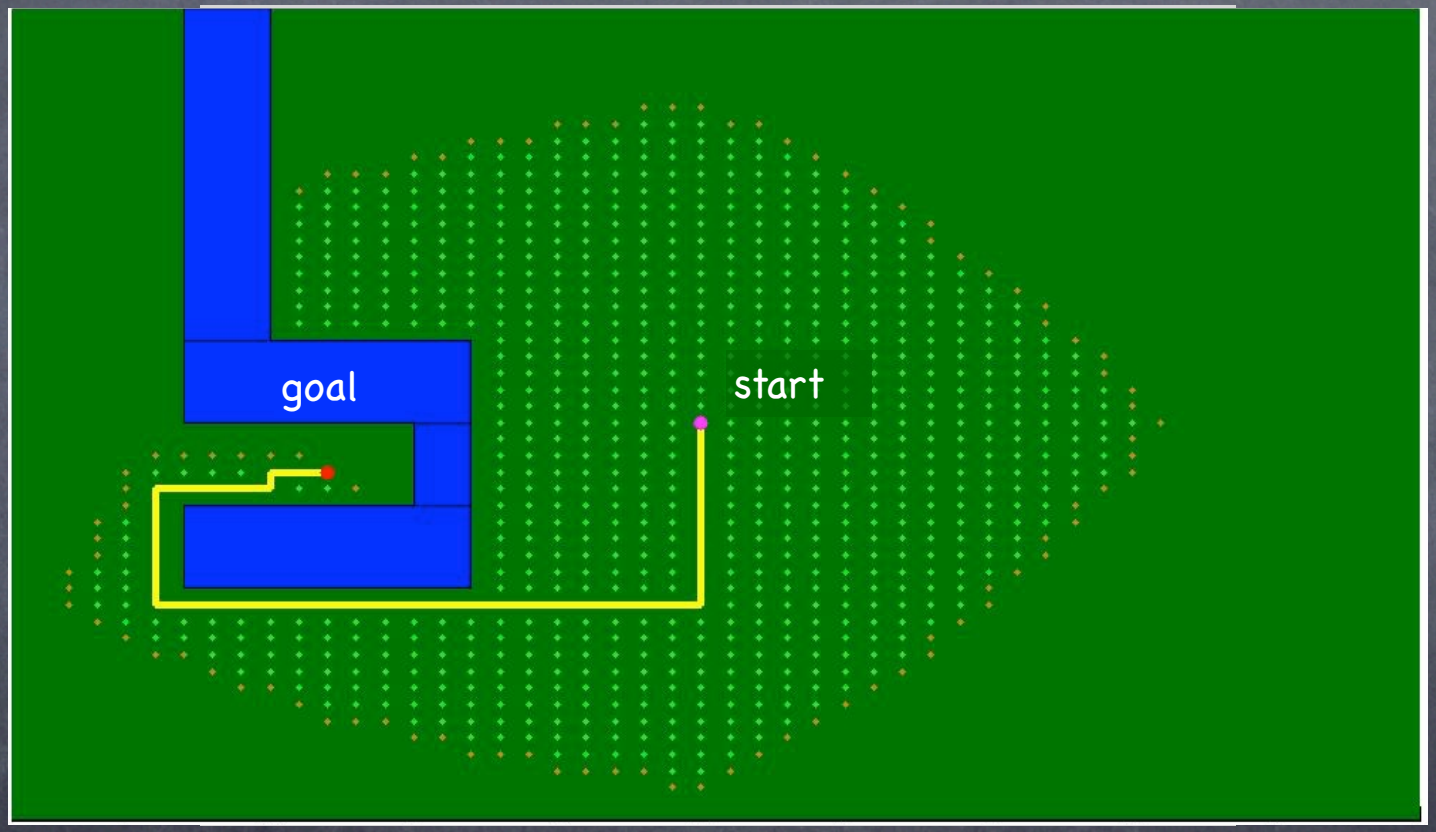
g_score: distance along current path back to start

h_score: best possible distance to goal

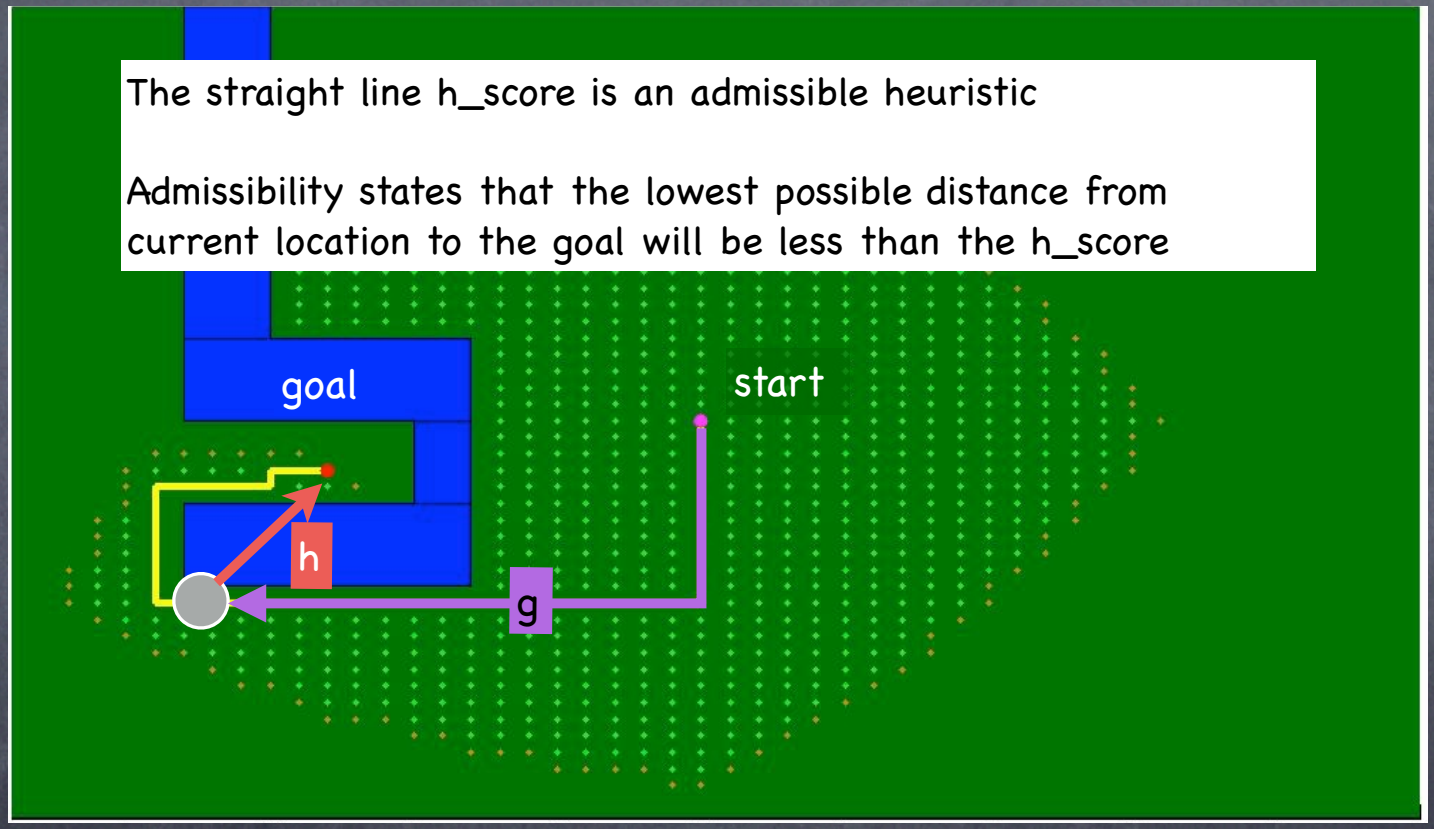


matlab example: A*

pathplan.m

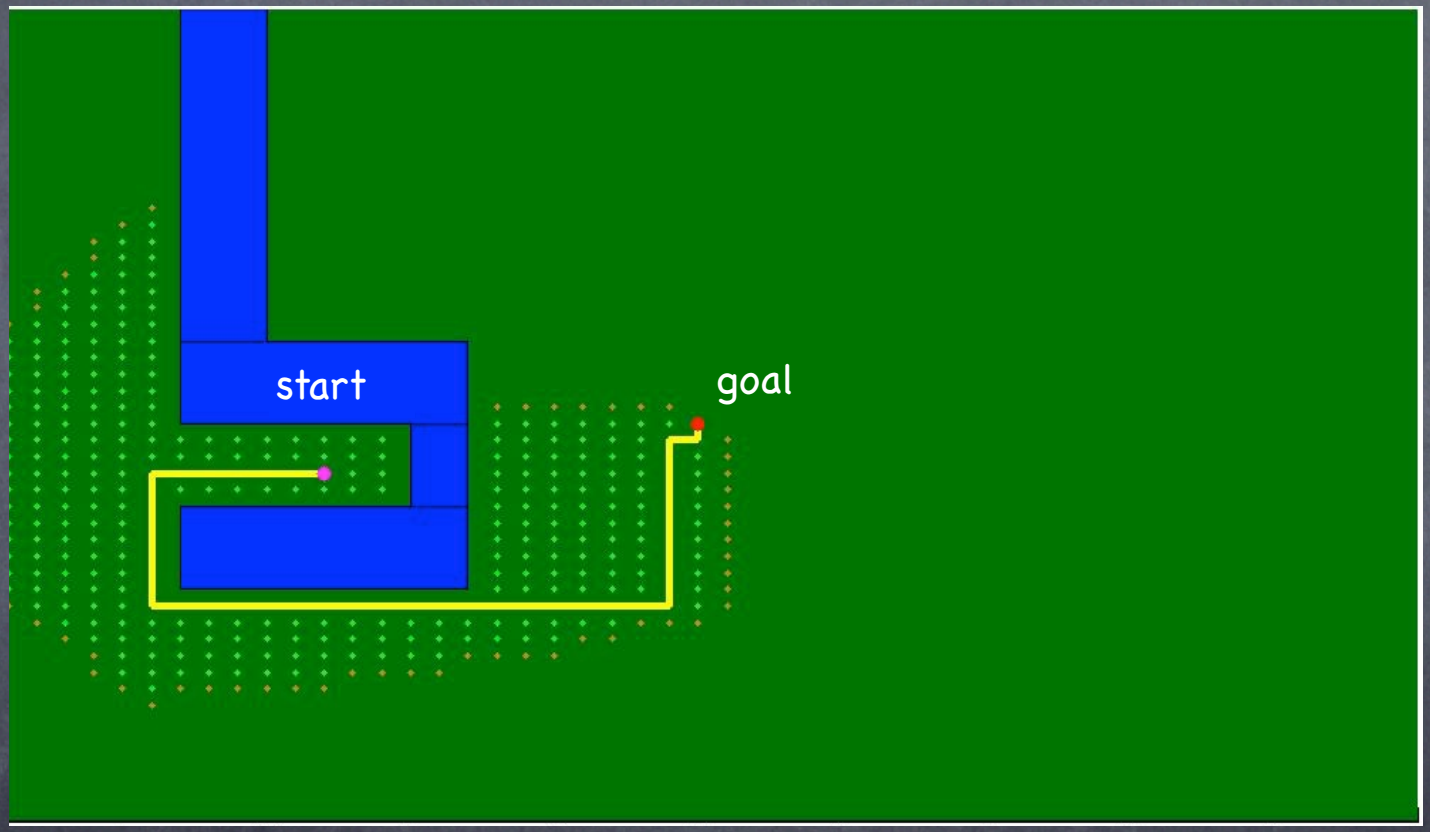


Admissibility: Straight line heuristic

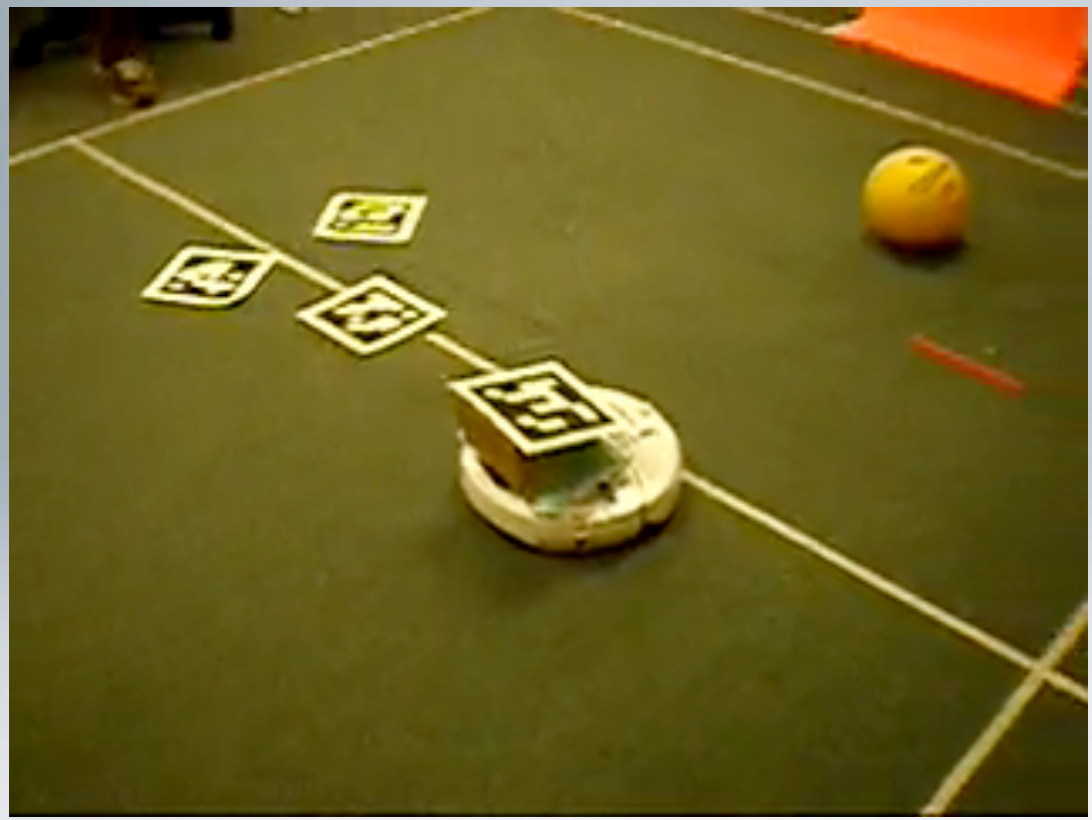


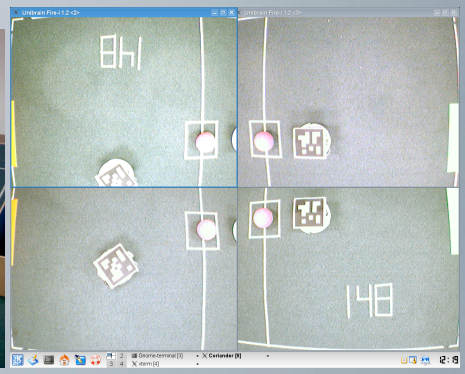
matlab example: A*

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



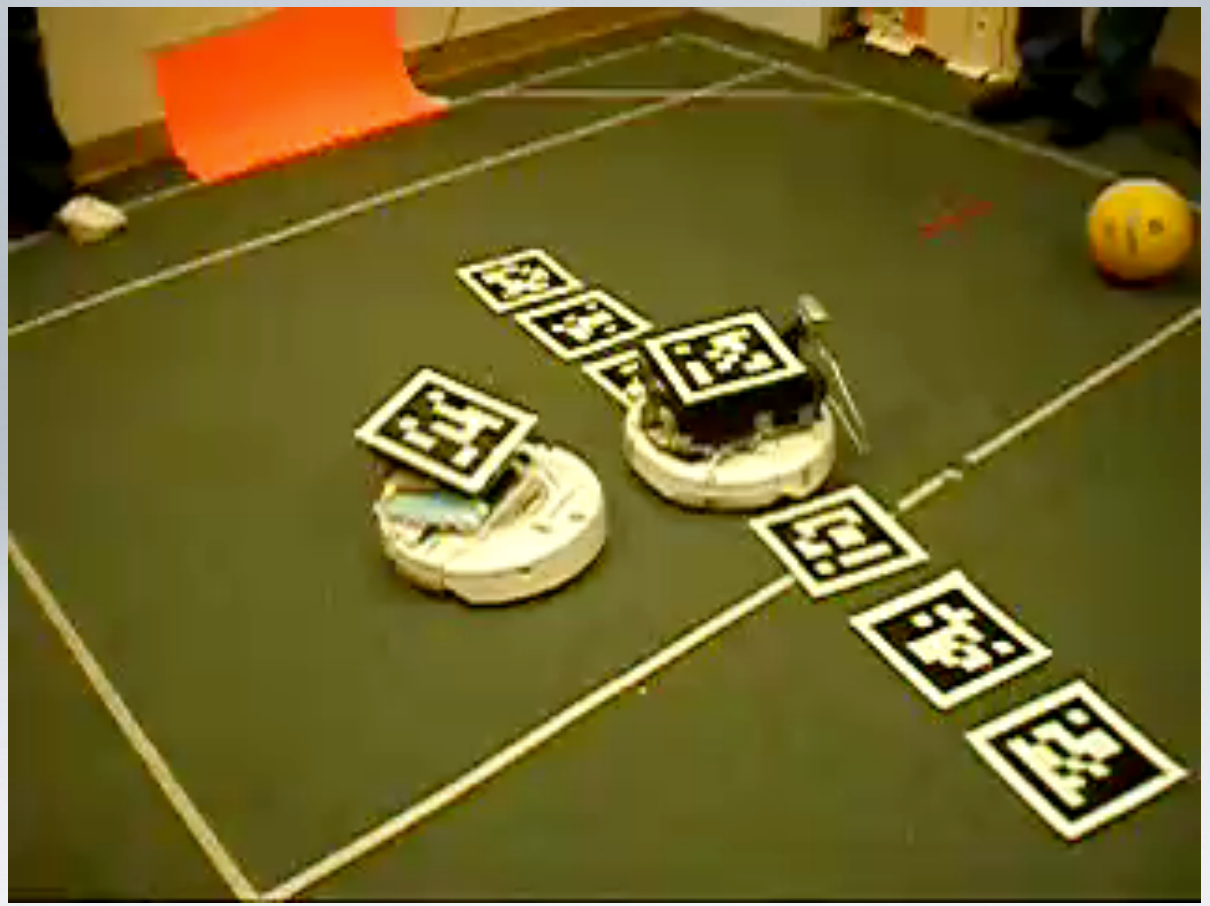
A* PATH PLANNING





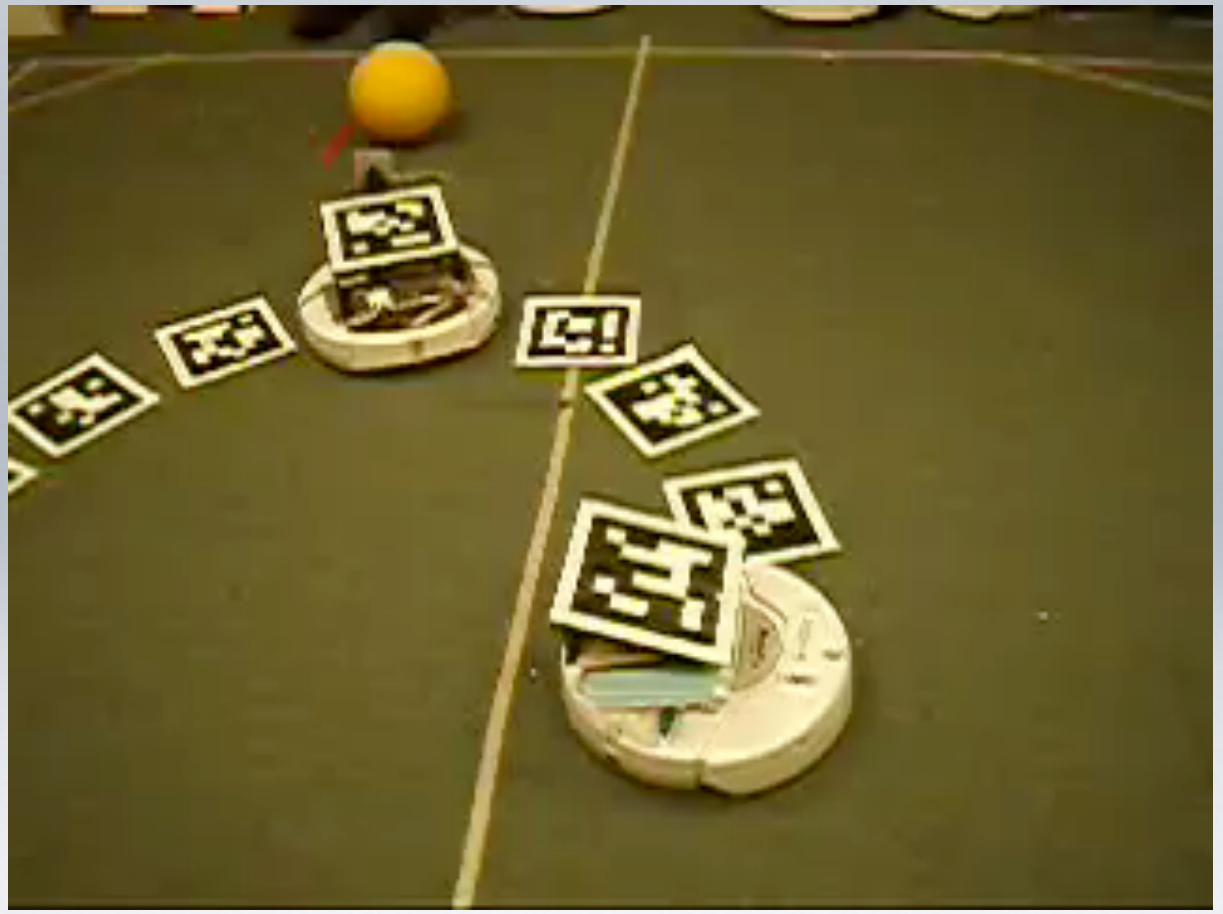
Lisa Miller, http://www.youtube.com/watch?v=2Z2RyeofsZg

NAVIGATING AROUND A WALL



Lisa Miller

AVOIDING DEAD-ENDS



Lisa Miller

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

Greedy best-first search

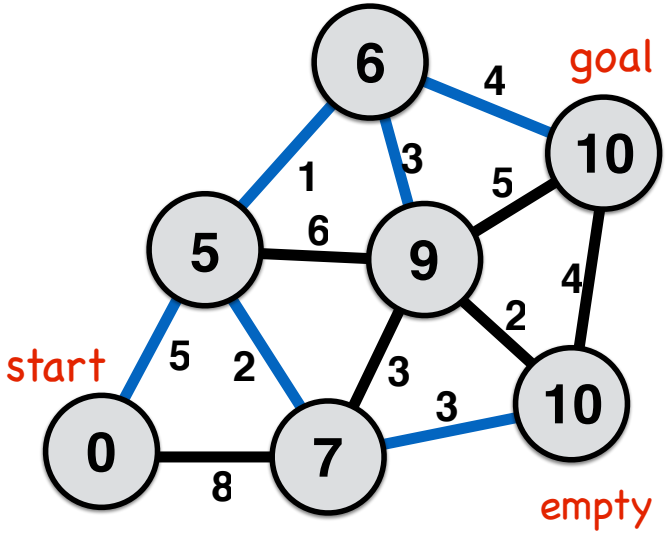
A-star shortest path algorithm

all nodes ← {$dist_{start}$ ← infinity, $parent_{start}$ ← none, $visited_{start}$ ← false}
start_node ← {$dist_{start}$ ← 0, $parent_{start}$ ← none, $visited_{start}$ ← true}
visit_queue ← start_node
    while (visit_queue != empty) && current_node != goal
        cur_node ← dequeue(visit_queue, f_score)
        $visited_{cur\_node}$ ← true
        for each nbr in not_visited(adjacent(cur_node))
            enqueue(nbr to visit_queue)
            if $dist_{nbr}$ > $dist_{cur\_node}$ + distance(nbr,cur_node)
                $parent_{nbr}$ ← current_node
                $dist_{nbr}$ ← $dist_{cur\_node}$ + distance(nbr,cur_node)
                f_score ← $distance_{nbr}$ + $line\_distance_{nbr,goal}$
            end if
        end for loop
    end while loop
output ← 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

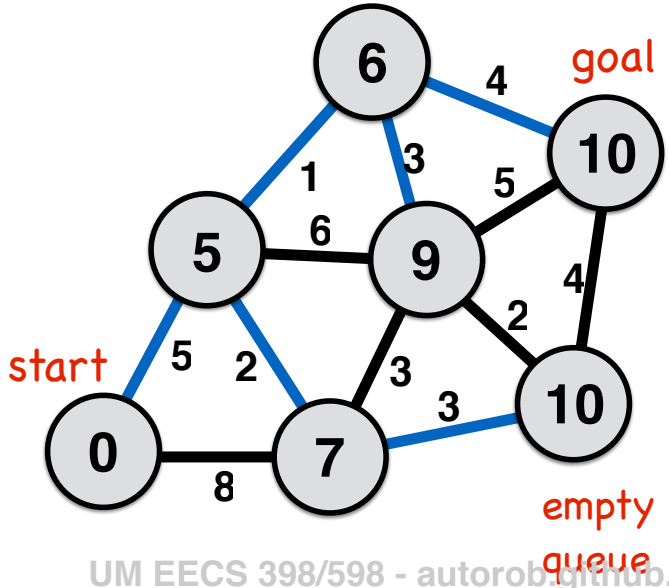


Greedy best-first search

all nodes ← {$dist_{start}$ ← infinity, $parent_{start}$ ← none, $visited_{start}$ ← false}
start_node ← {$dist_{start}$ ← 0, $parent_{start}$ ← none, $visited_{start}$ ← true}
visit_queue ← start_node
    while (visit_queue != empty) && current_node != goal
        cur_node ← dequeue(visit_queue, f_score)
        $visited_{cur\_node}$ ← true
        for each nbr in not_visited(adjacent(cur_node))
            enqueue(nbr to visit_queue)
            if $dist_{nbr}$ > $dist_{cur\_node}$ + distance(nbr,cur_node)
                $parent_{nbr}$ ← current_node
                $dist_{nbr}$ ← $dist_{cur\_node}$ + distance(nbr,cur_node)
                f_score ← ~~$distance_{nbr}$~~ + $line\_distance_{nbr,goal}$
            end if
        end for loop
    end while loop
output ← parent, distance

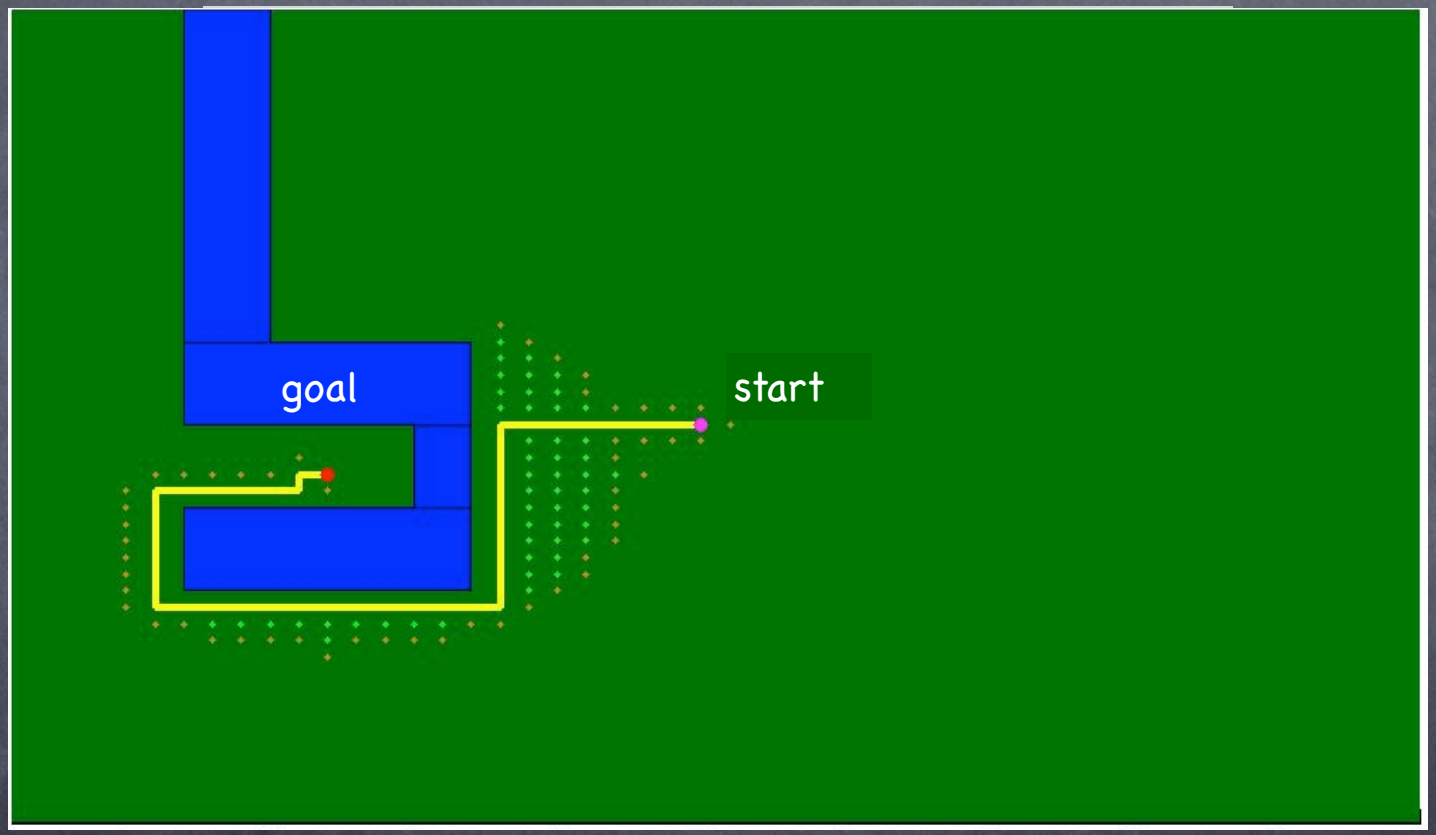
priority queue wrt. f_score
(implement min binary heap)

h_score:
best possible
distance to goal



matlab example: gr.best-first

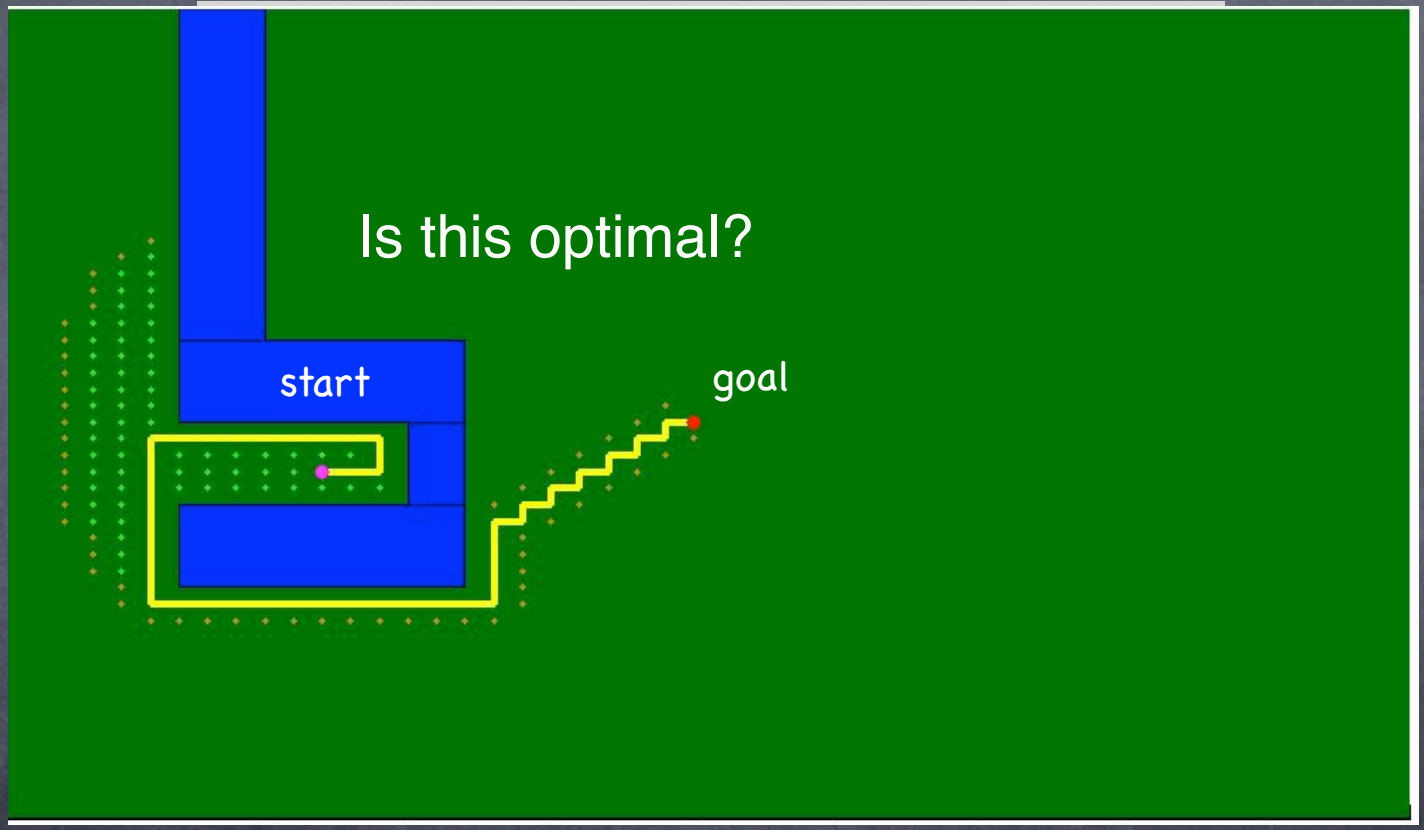
pathplan.m



visited locations in light green
4-connected grid

matlab example: gr.best-first

pathplan.m



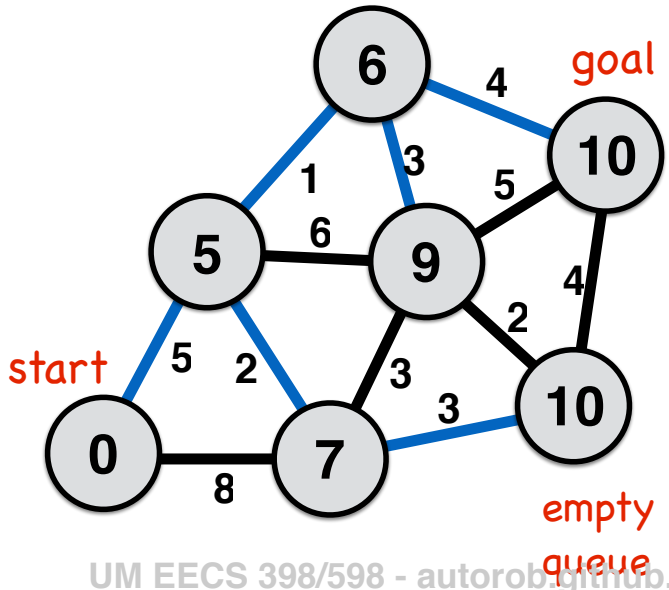
visited locations in light green
4-connected grid

Heaps and Priority Queues

A-star shortest path algorithm

all nodes ← {$dist_{start}$ ← infinity, $parent_{start}$ ← none, $visited_{start}$ ← false}
start_node ← {$dist_{start}$ ← 0, $parent_{start}$ ← none, $visited_{start}$ ← true}
visit_queue ← start_node
    while (visit_queue != empty) && current_node != goal
        dequeue: cur_node ← f_score(visit_queue)
        $visited_{cur\_node}$ ← true
        for each nbr in not_visited(adjacent(cur_node))
            enqueue: nbr to visit_queue
            if $dist_{nbr}$ > $dist_{cur\_node}$ + distance(nbr,cur_node)
                $parent_{nbr}$ ← current_node
                $dist_{nbr}$ ← $dist_{cur\_node}$ + distance(nbr,cur_node)
                f_score ← $distance_{nbr}$ + $line\_distance_{nbr,goal}$
            end if
        end for loop
    end while loop
output ← parent, distance

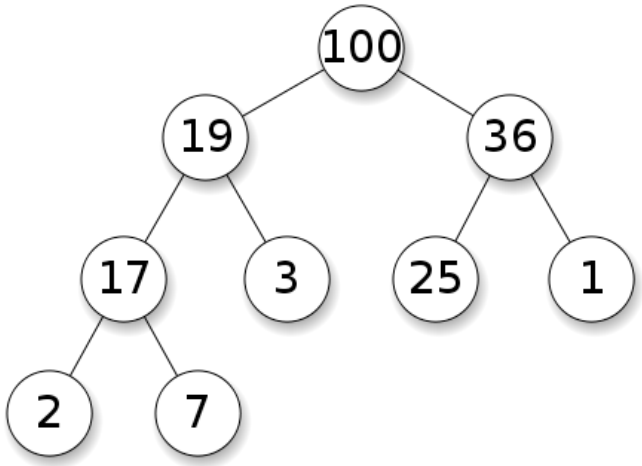
min binary heap for priority queue



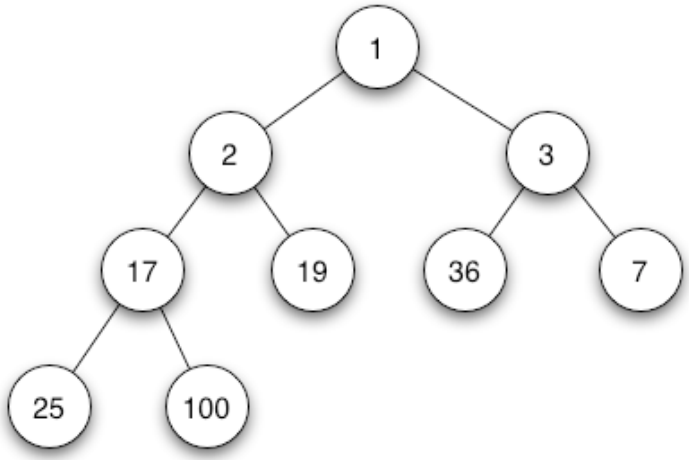
Binary Heaps

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

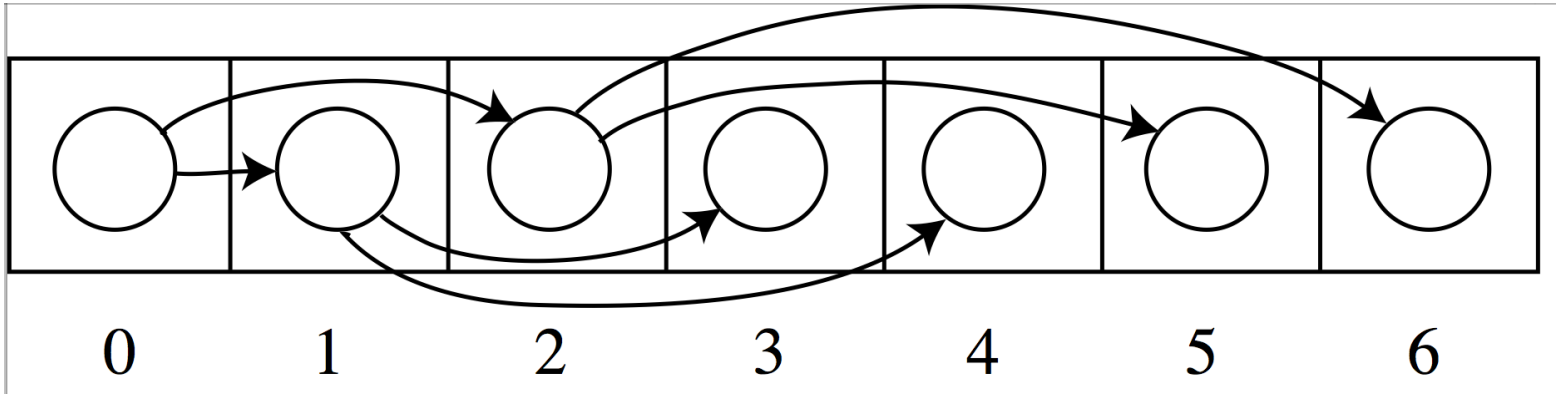


max heap



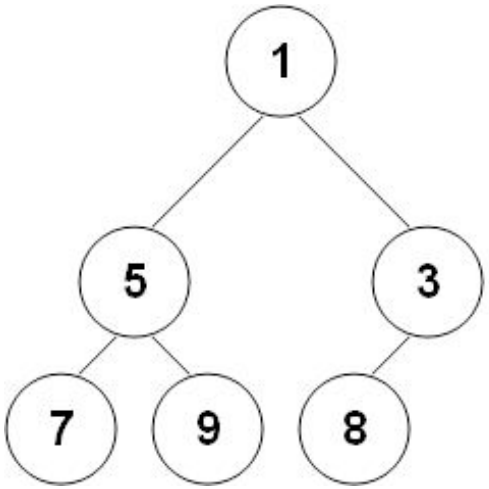
min heap

Heaps as arrays



- Heap element at array location *i* has
  - children at array locations *2i+1* and *2i+2*
  - parent at *floor((i-1)/2)*

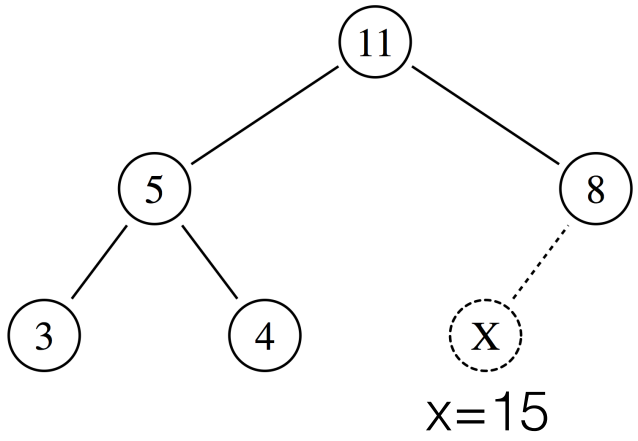
Heap array example



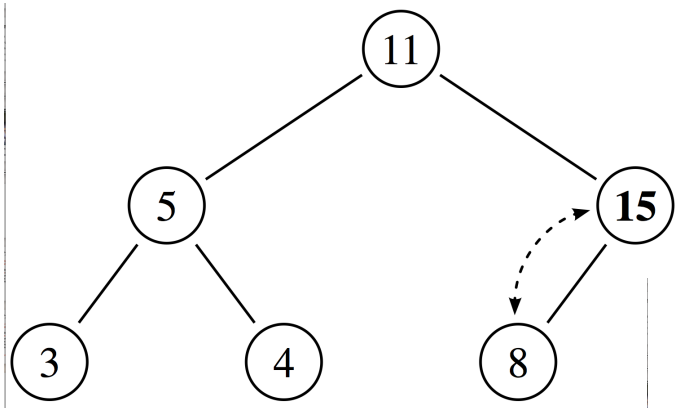
Node	1	5	3	7	9	8
Index	0	1	2	3	4	5

Heap operations: Insert

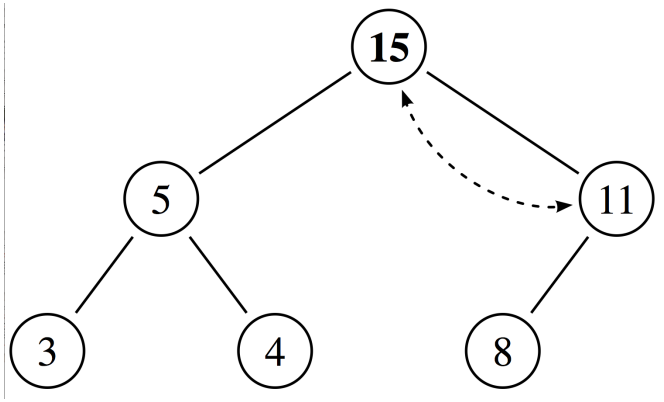
1) add new element to end of tree



2) swap with parent



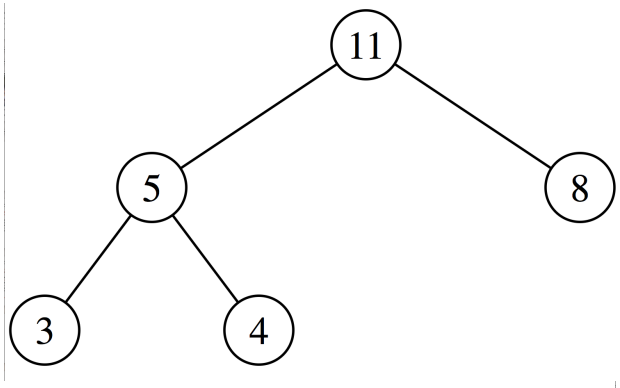
3) until heaped, do (2)



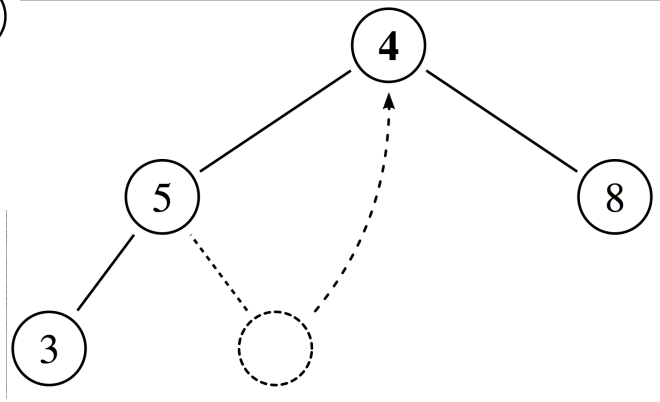
For priority queue, previously non-queued locations will be inserted with fscore priority

Heap operations: Extract

1) extract root element

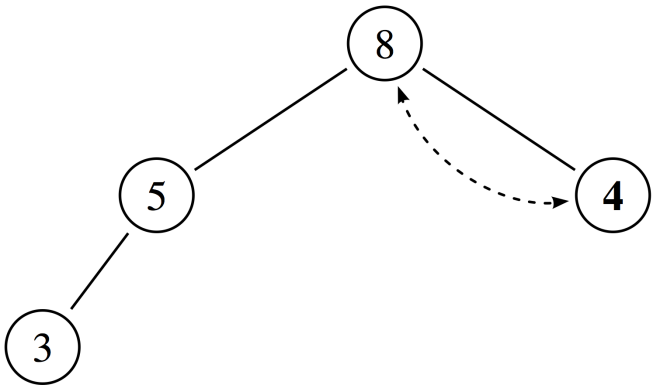


2) put last element at root



3) swap with higher priority child

4) until heaped, do (3)

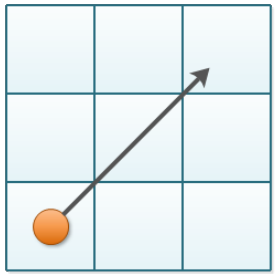


For priority queue, the root of the heap will be the next node to visit

Questions to consider

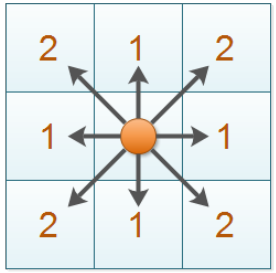
- How many operations are needed for heap insertion and extraction?
- How much better is a min heap than an array wrt. # of operations?
- How should we measure distance on a uniform grid?
- Is a choice of distance measure both metric and admissible?

Euclidean Distance



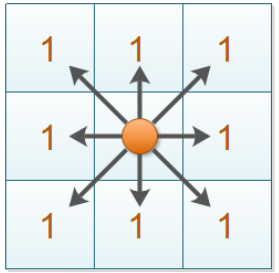
$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Manhattan Distance



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

Chebyshev Distance

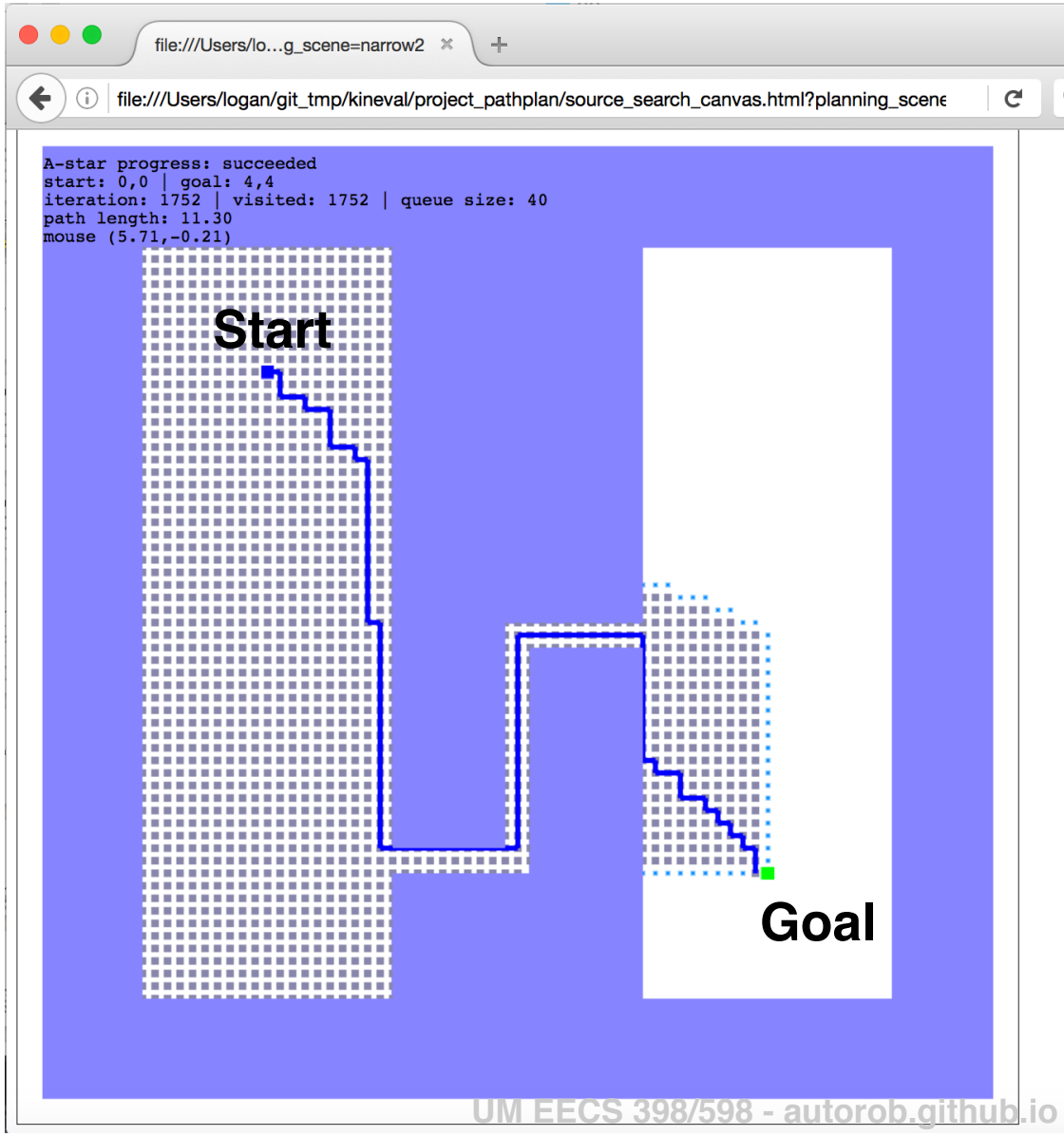


$$\max(|x_1 - x_2|, |y_1 - y_2|)$$

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

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
- 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 398 Introduction to Autonomous Robotics <br>
ME/EECS 567 Robot Kinematics and Dynamics
</a>

</body>
</html>
```