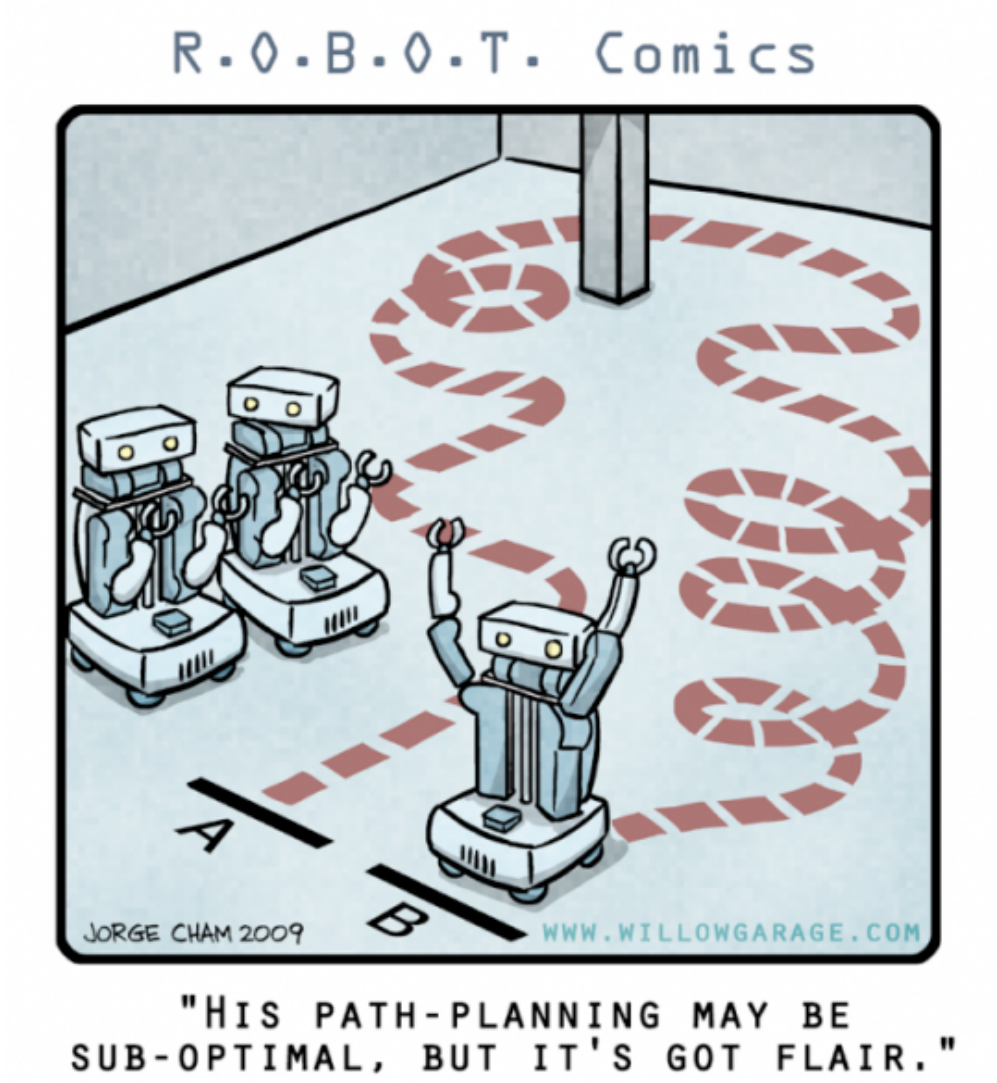


Path planning

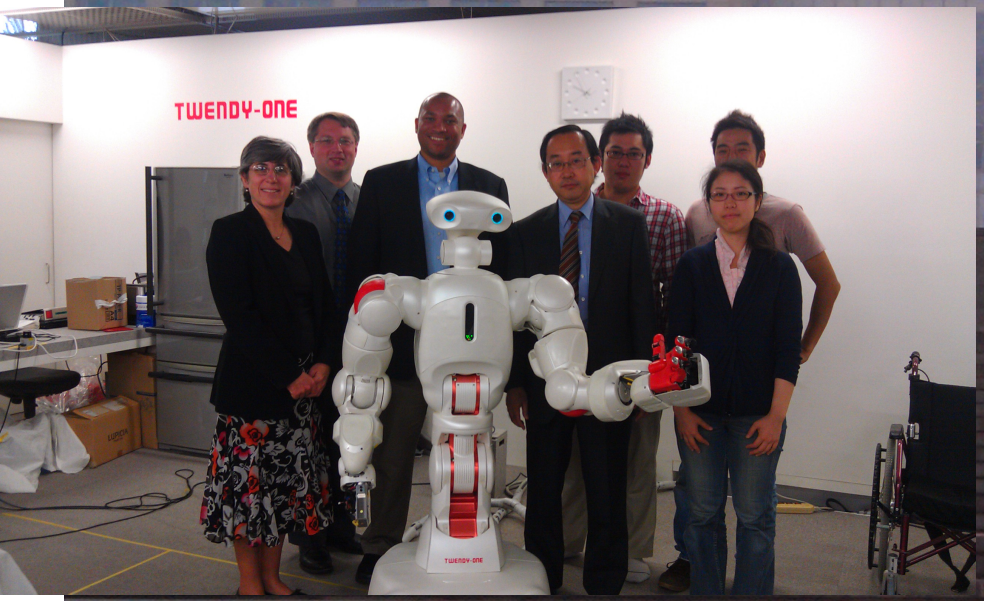
the best way to get from A to B

EECS 367
Intro. to Autonomous Robotics

ROB 511
Robot Operating Systems

Fall 2020

autorob.org


TWENDY-ONE


CORNELL

Manuela Veloso: RoboCup's Champion

This roboticist has transformed robot soccer into a global phenomenon

By **Prachi Patel**

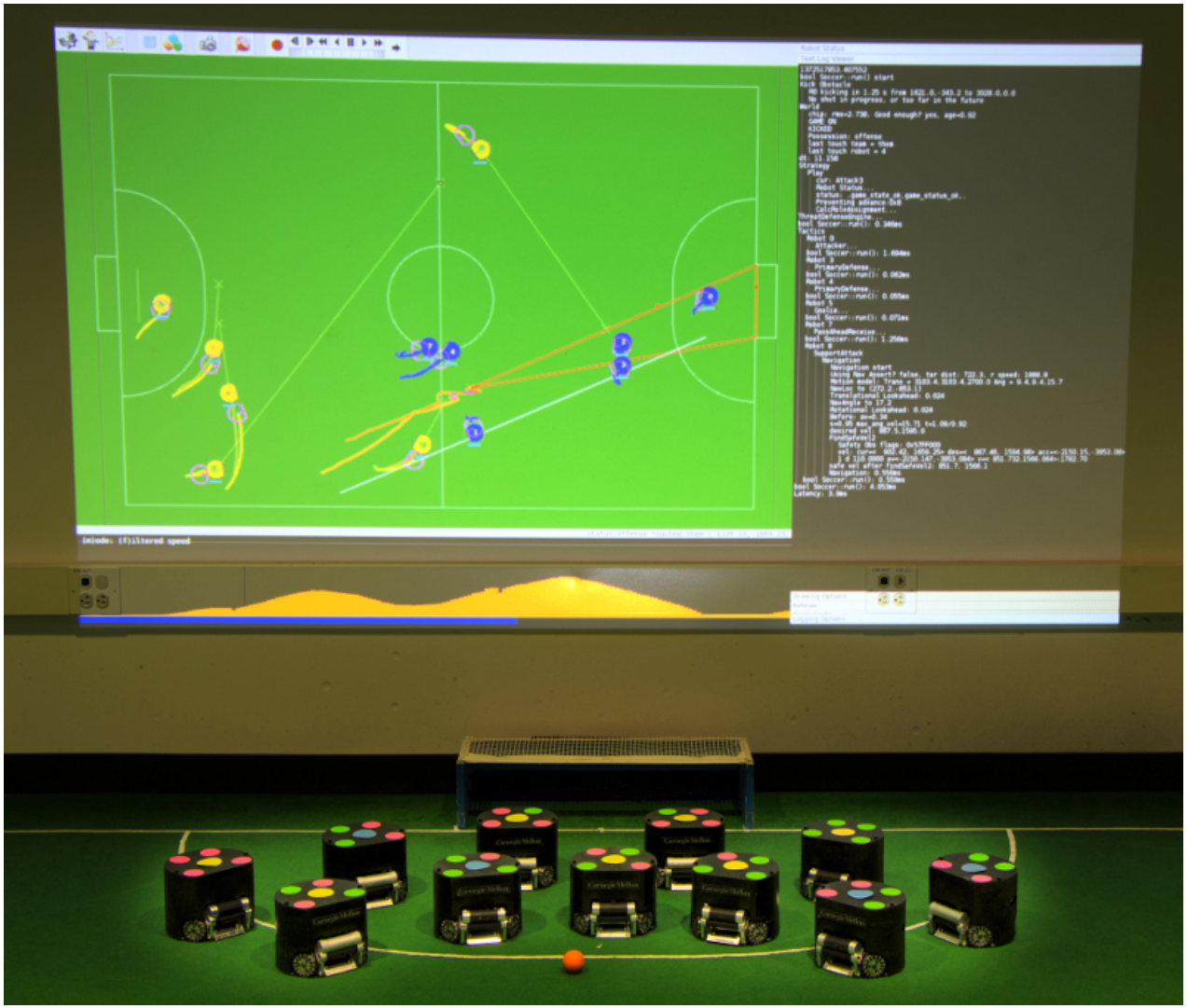
Stepping out of the elevator on the seventh floor of Carnegie Mellon University's Gates Center for Computer Science, I'm greeted by an ungainly yet courteous robot. It guides me to the office of Manuela Veloso, who beams at the bot like a proud parent. Veloso then punches a few buttons to send it off to her laboratory a few corridors away.

Veloso, a computer science professor at CMU, in Pittsburgh, has worked for over two decades to develop such autonomous mobile robots. She believes that humans and robots will one day coexist, and my robot escort, named CoBot (for Collaborative Robot), is one of her contributions to that future.

Photo: Ross Mantle

CMDragons
RoboCup Small
2006

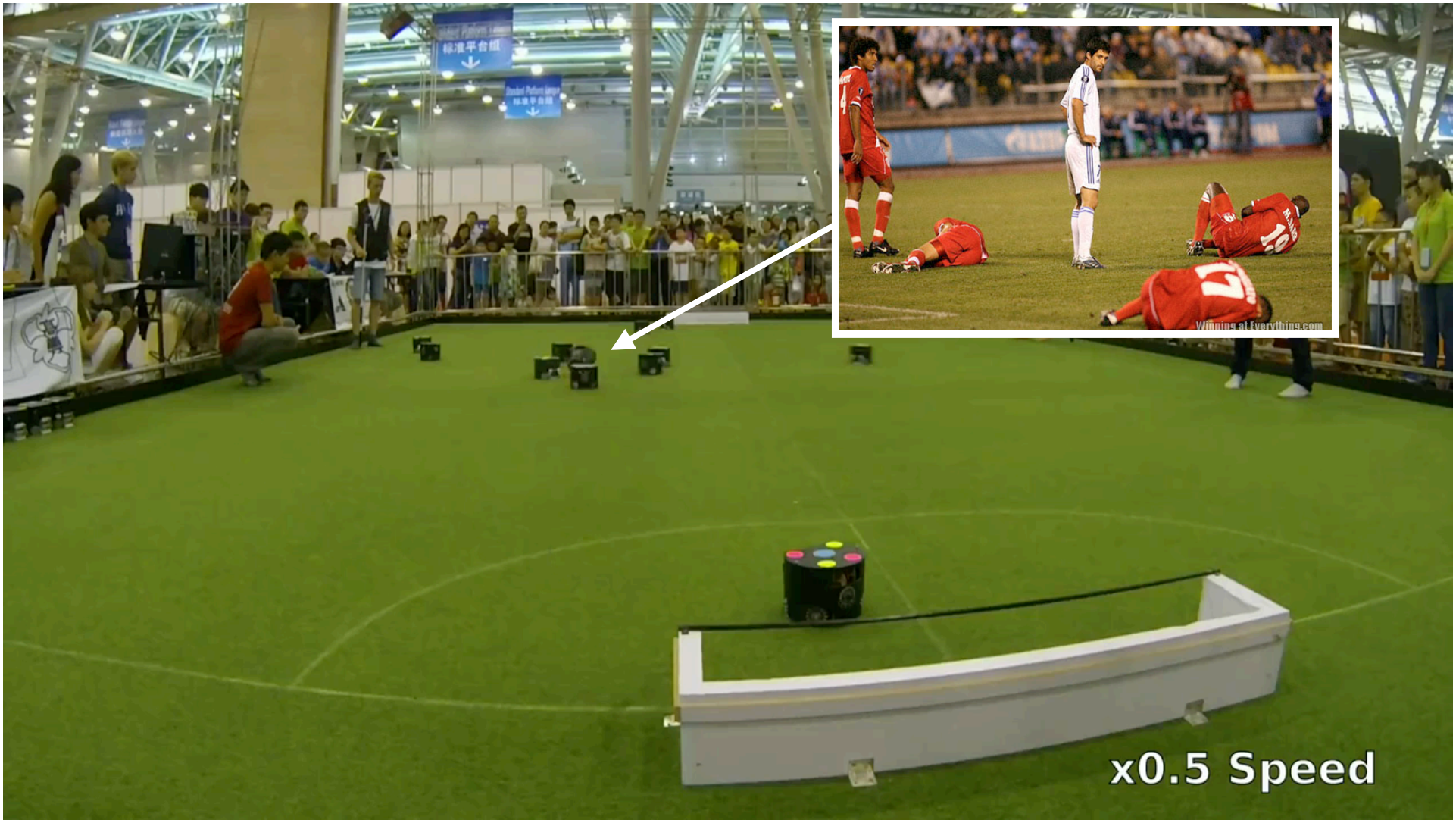
https://youtu.be/-Y4H3Sox_4I



CMDragons - http://www.cs.cmu.edu/~robosoccer/small/

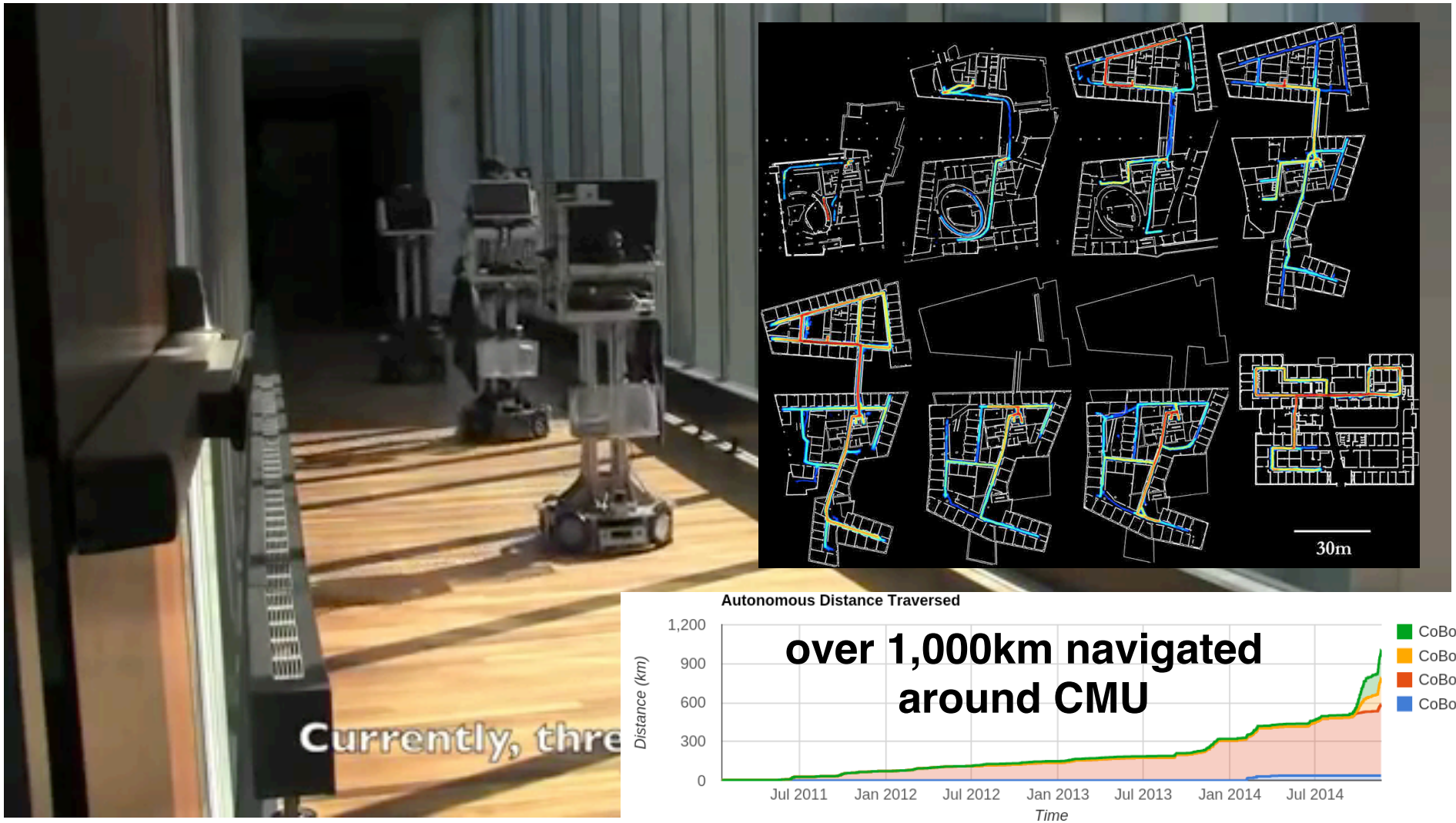
CMDragons 2015 Pass-ahead Goal

CMDragons 2015 slow-motion multi-pass goal



CMDragons 2015 slow-motion multi-pass goal

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



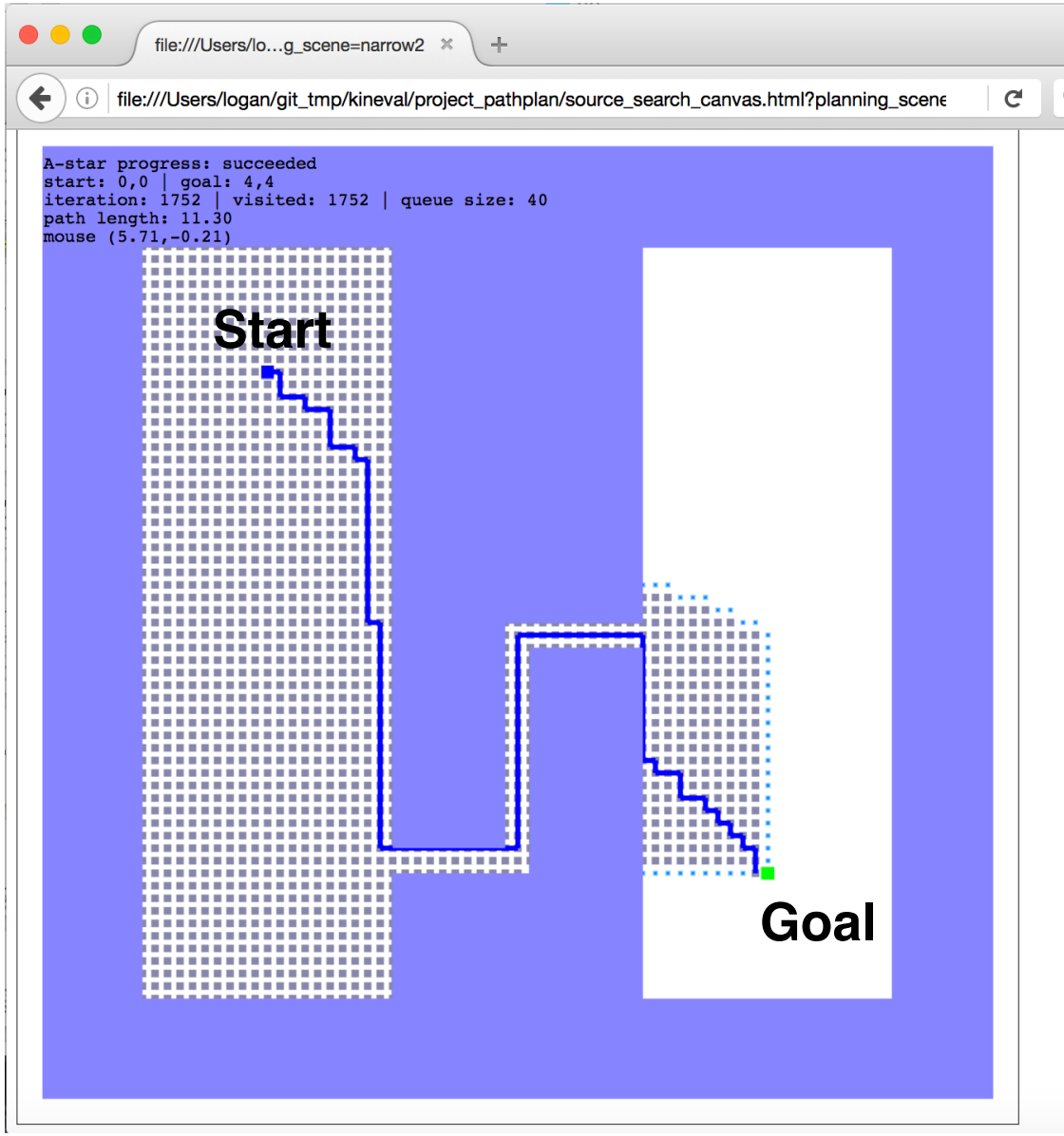
http://www.cs.cmu.edu/~coral/projects/cobot/

https://www.joydeepb.com/research.html

CoBot greets me in 2013

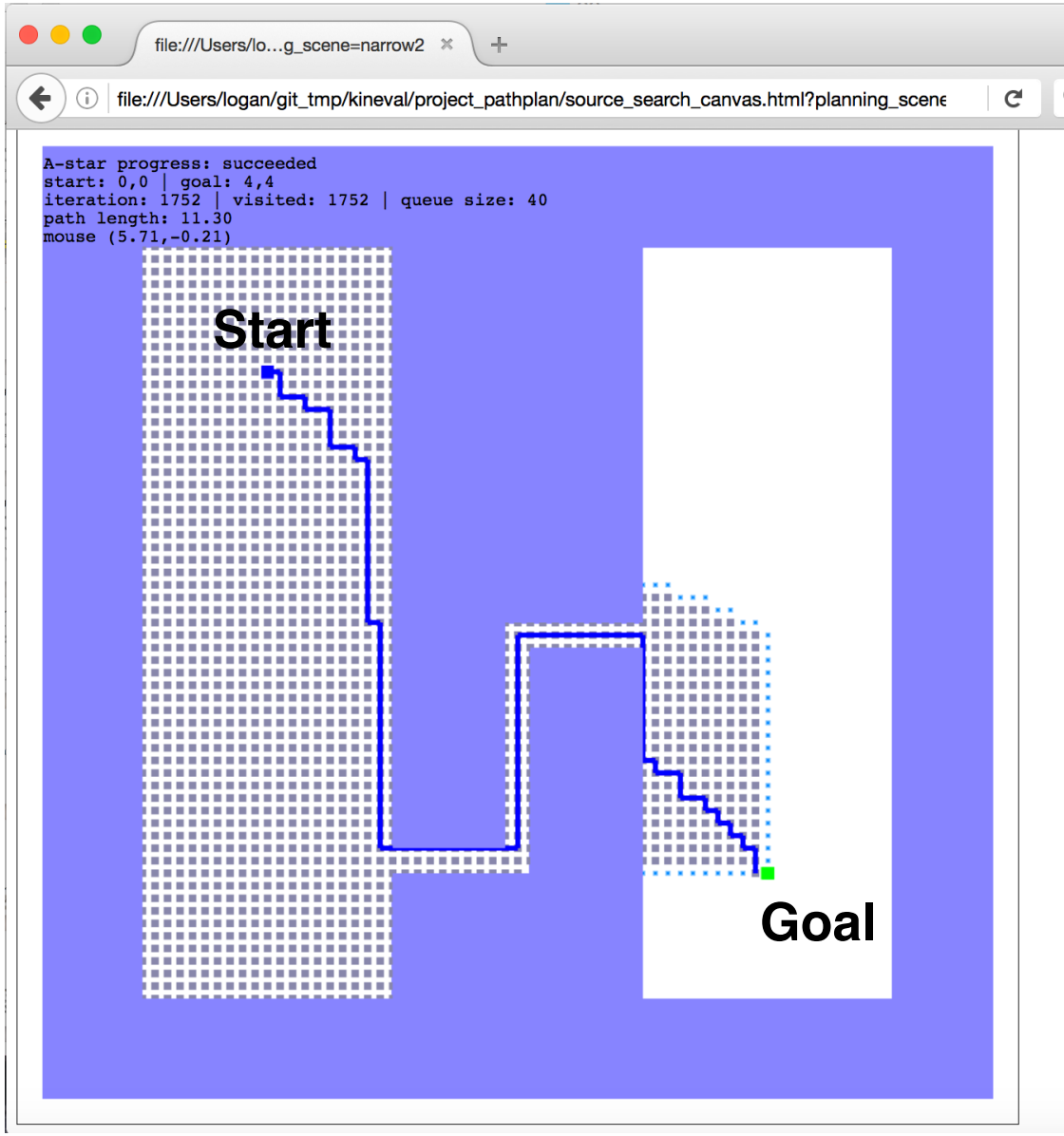
Project 1:
2D Path Planning

- 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



Path Planning

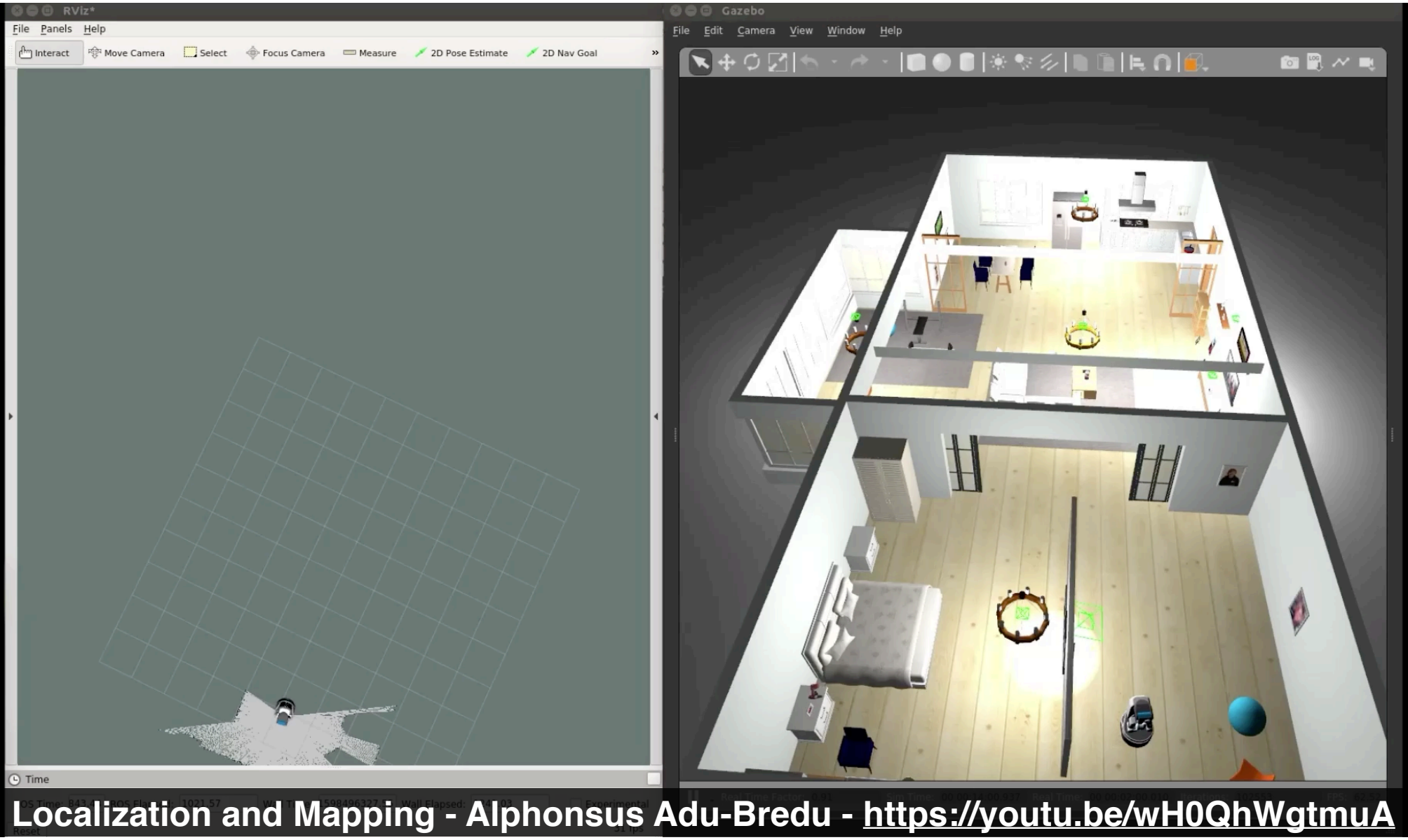
- The robot knows:
  - **Localization**: where it is now
  - **Goal**: where it needs to go
  - **Map**: where it will hit something
- Infer:
  - **Path**: Collision-free sequence of locations to follow to goal

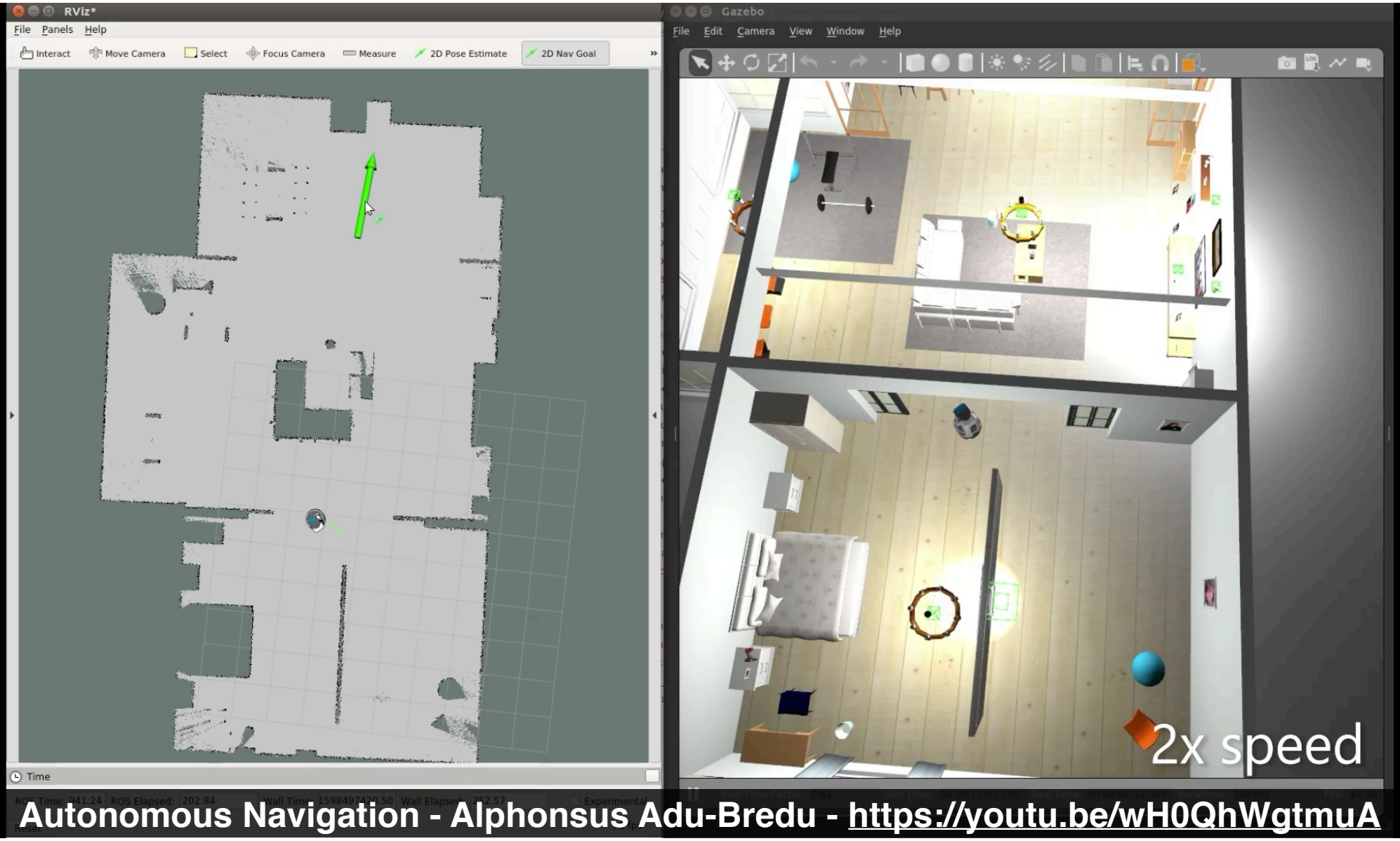


~~In-class navigation demo with Lab4Progress Fetch robot~~

CLOSED DUE TO COVID-19

quarantine := simulation

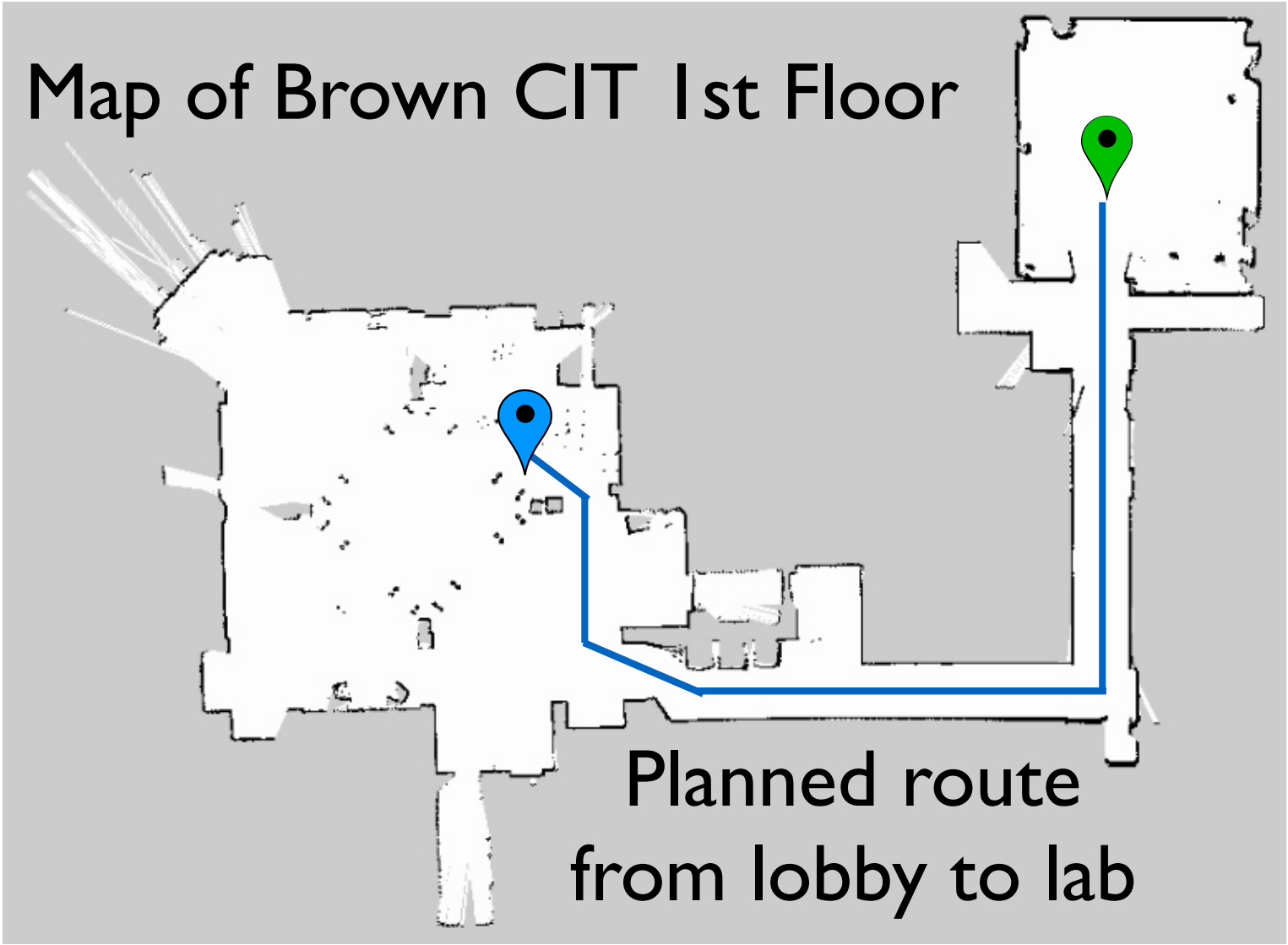

RViz*
File Panels Help
Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Nav Goal
Time
Gazebo
File Edit Camera View Window Help
Localization and Mapping - Alphonsus Adu-Bredu - https://youtu.be/wH0QhWgtmuA

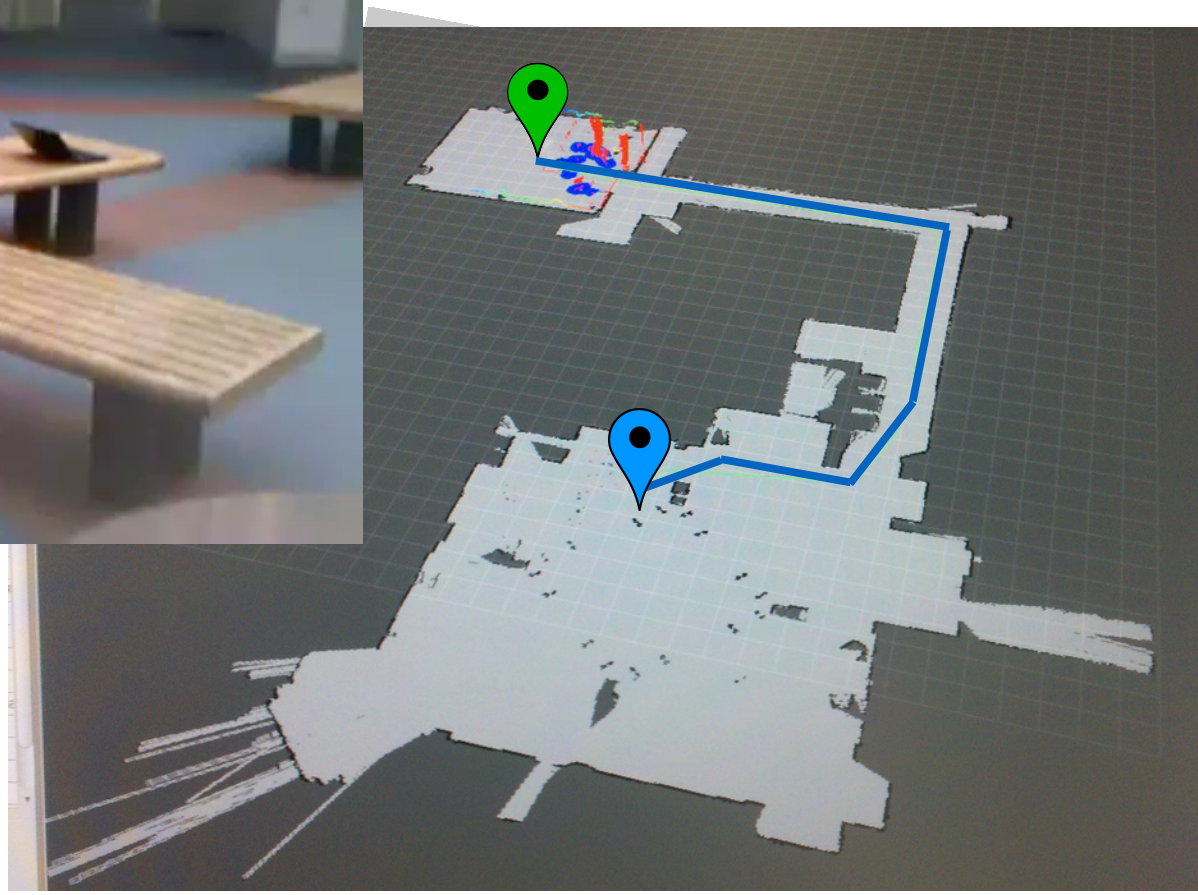


Autonomous Navigation - Alphonsus Adu-Bredu - https://youtu.be/wH0QhWgtmuA

How do we get from  to  ?

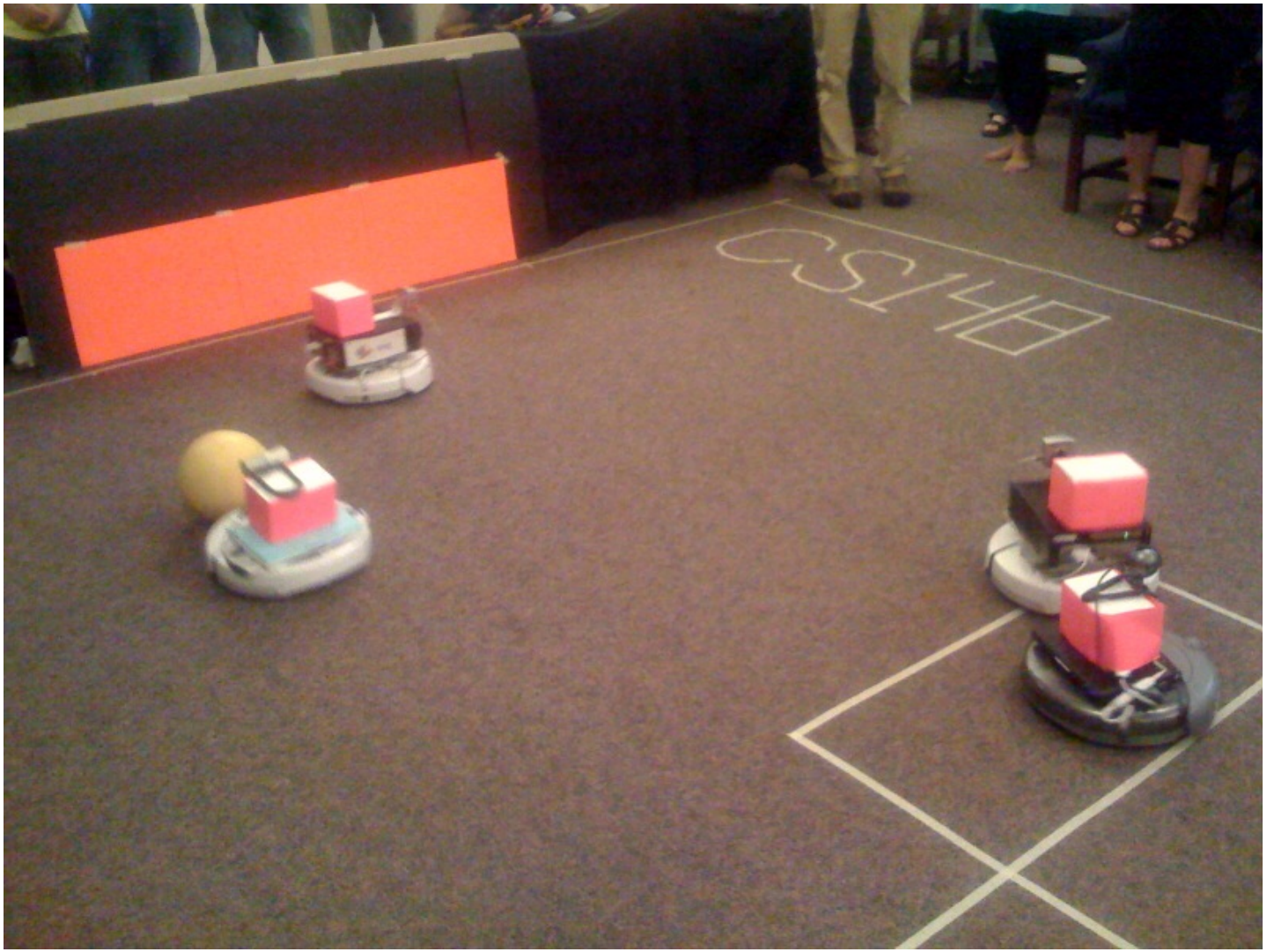
Map of Brown CIT 1st Floor





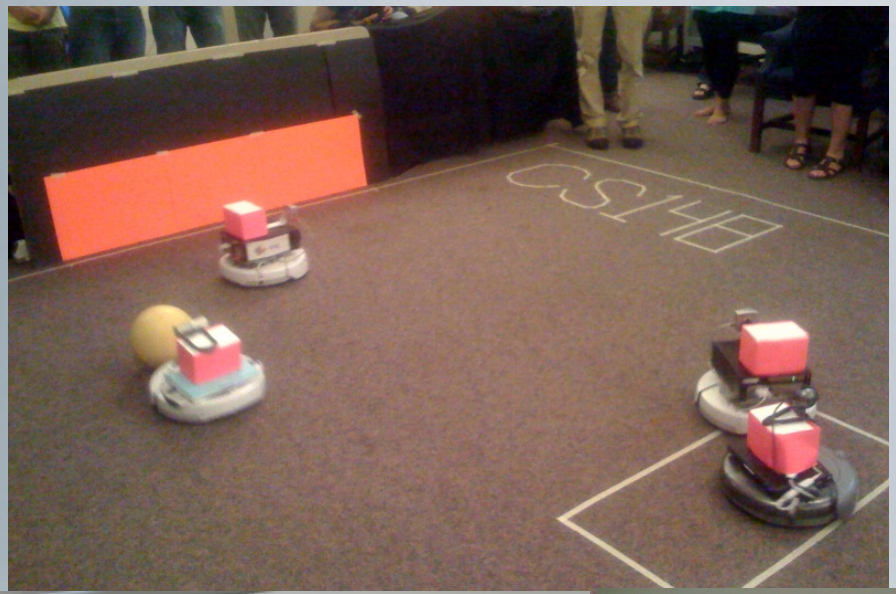
PR2 robot following
planned route
from lobby to lab

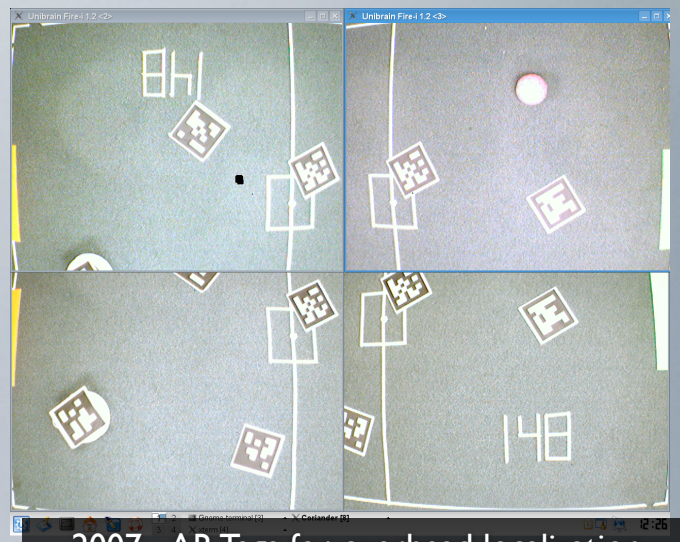
Going back to robot soccer…



Brown CS148 Promotional Video 2009 - https://youtu.be/bsvUQ5Kp2Q4

2007-10: SOCCER WITH IROBOT CREATE



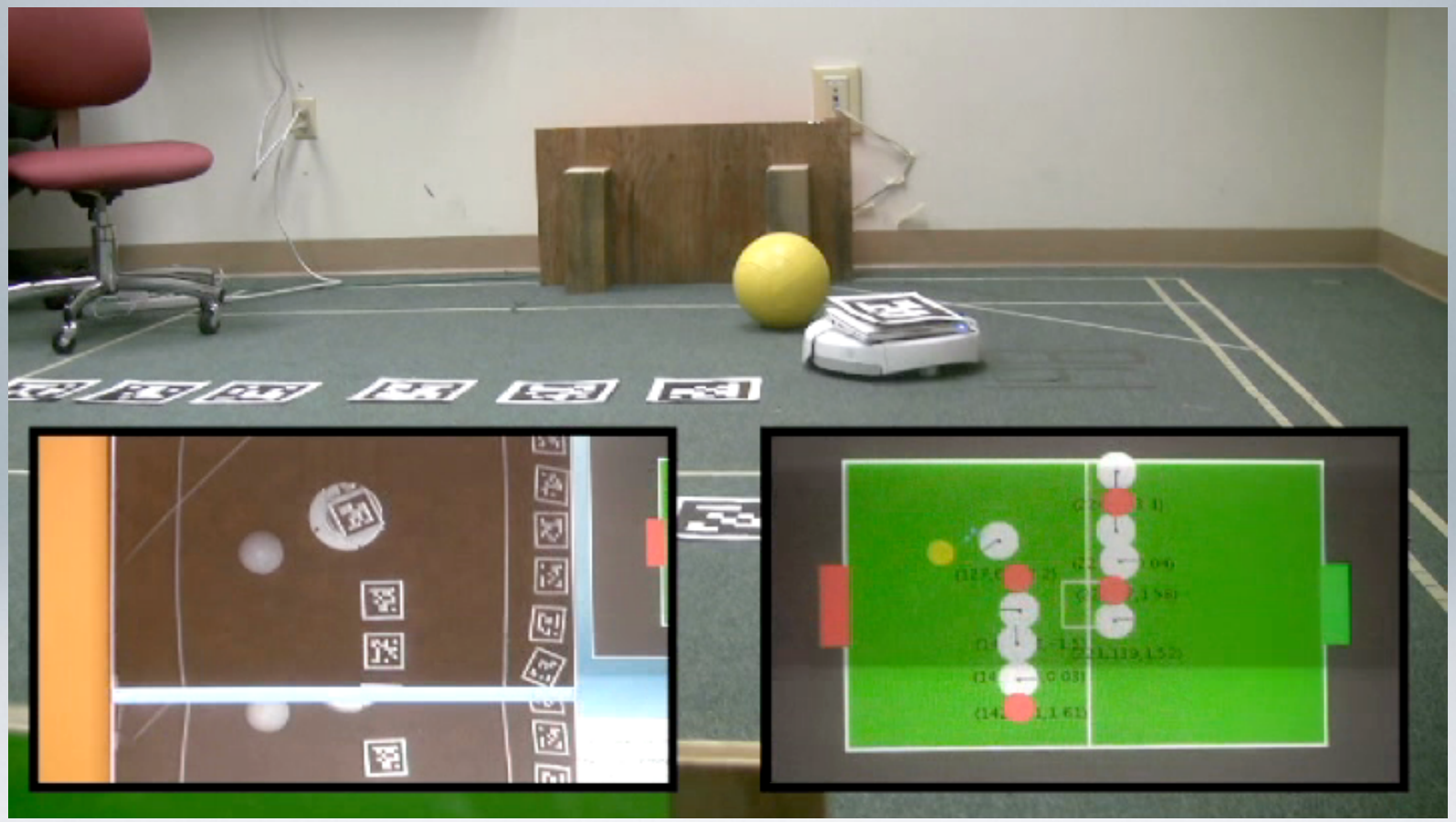


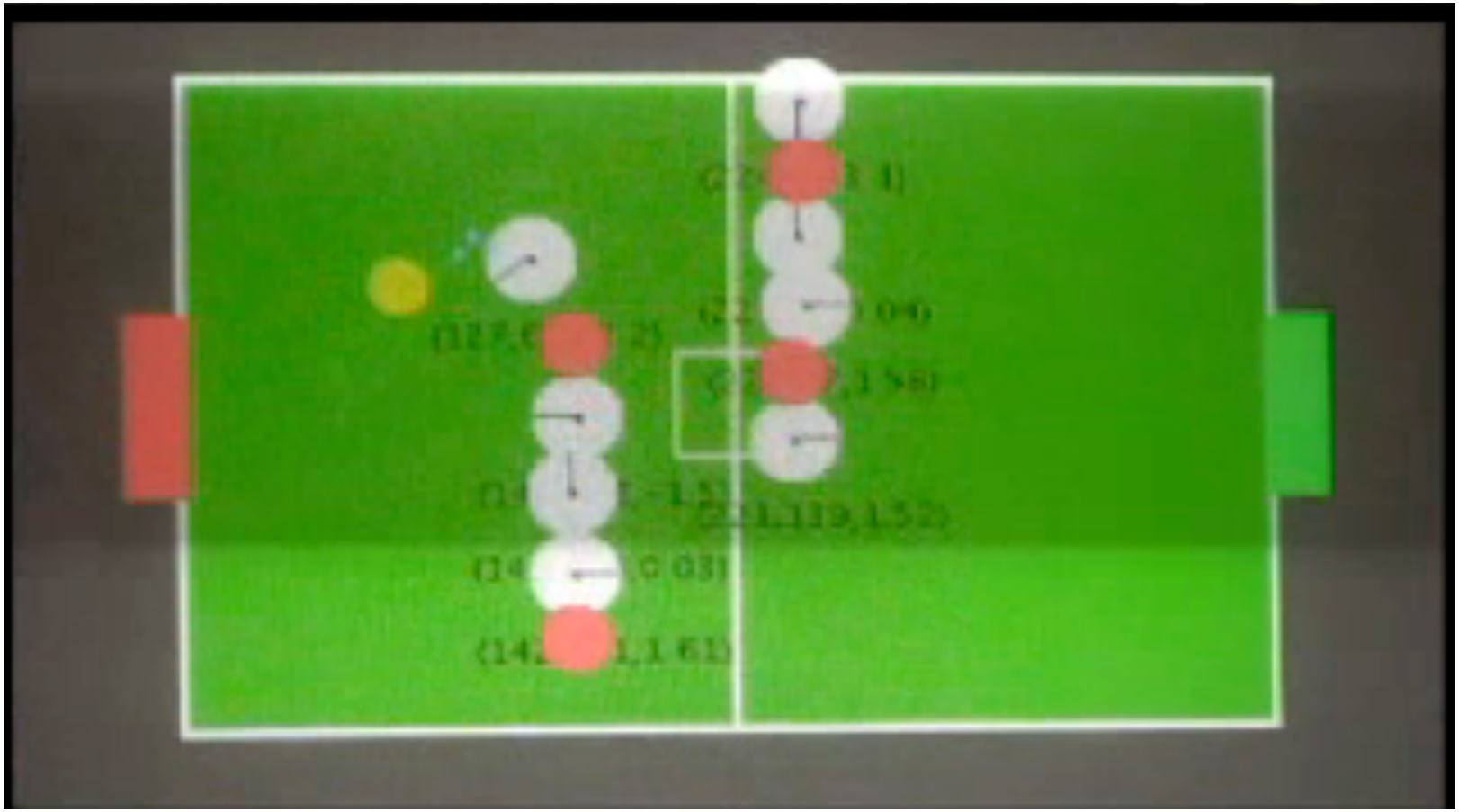
2007 - AR Tags for overhead localization

2007 - Mini ITX

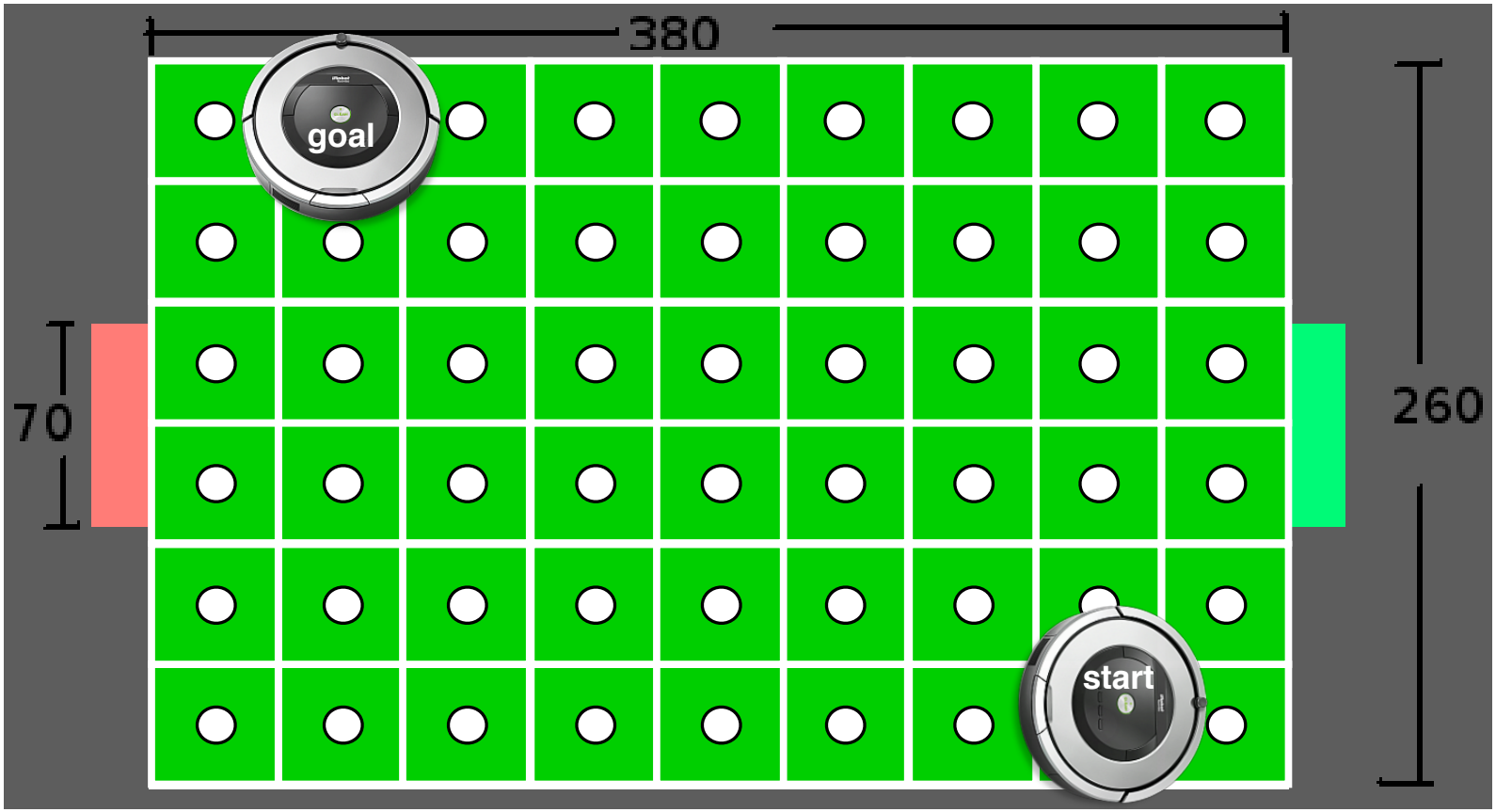
2009 - Asus EEE

youtube.com/watch?v=88zR6IC7S0g

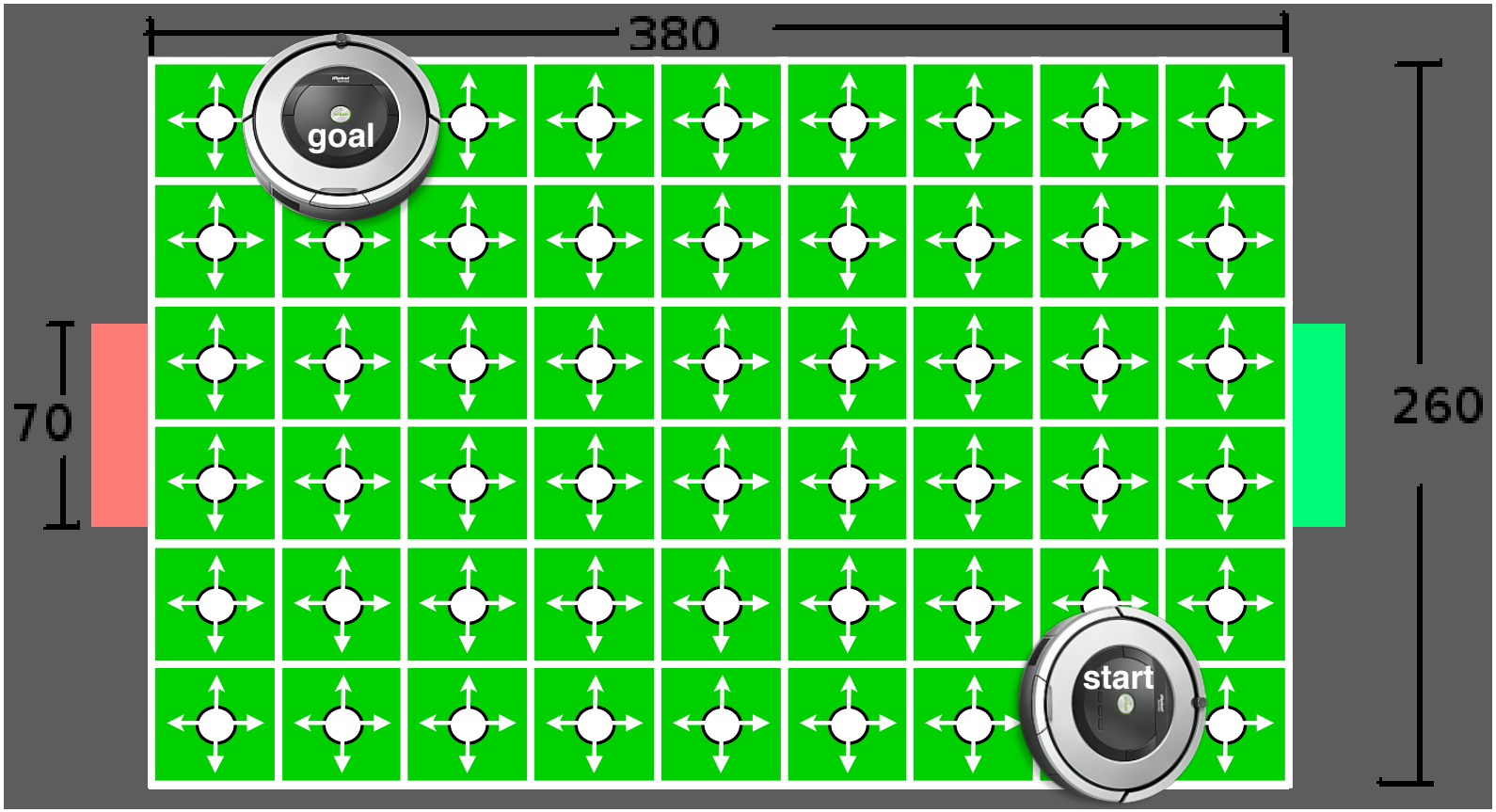




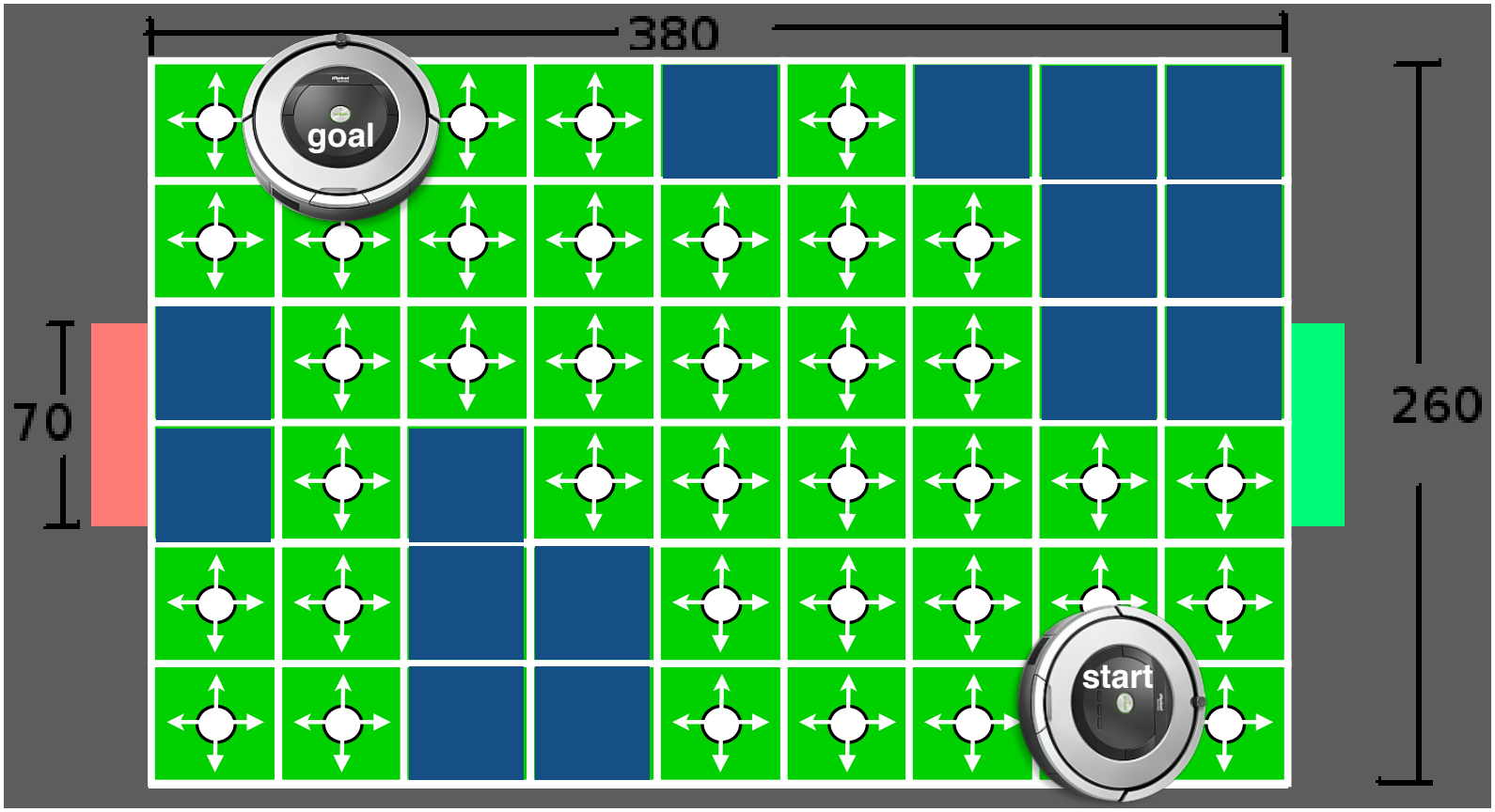
Consider all possible poses as uniformly distributed array of cells in a graph



Consider all possible poses as uniformly distributed array of cells in a graph
Edges connect adjacent cells, weighted by distance



Consider all possible poses as uniformly distributed array of cells in a graph
Edges connect adjacent cells, weighted by distance
Cells are invalid where its associated robot pose results in a collision

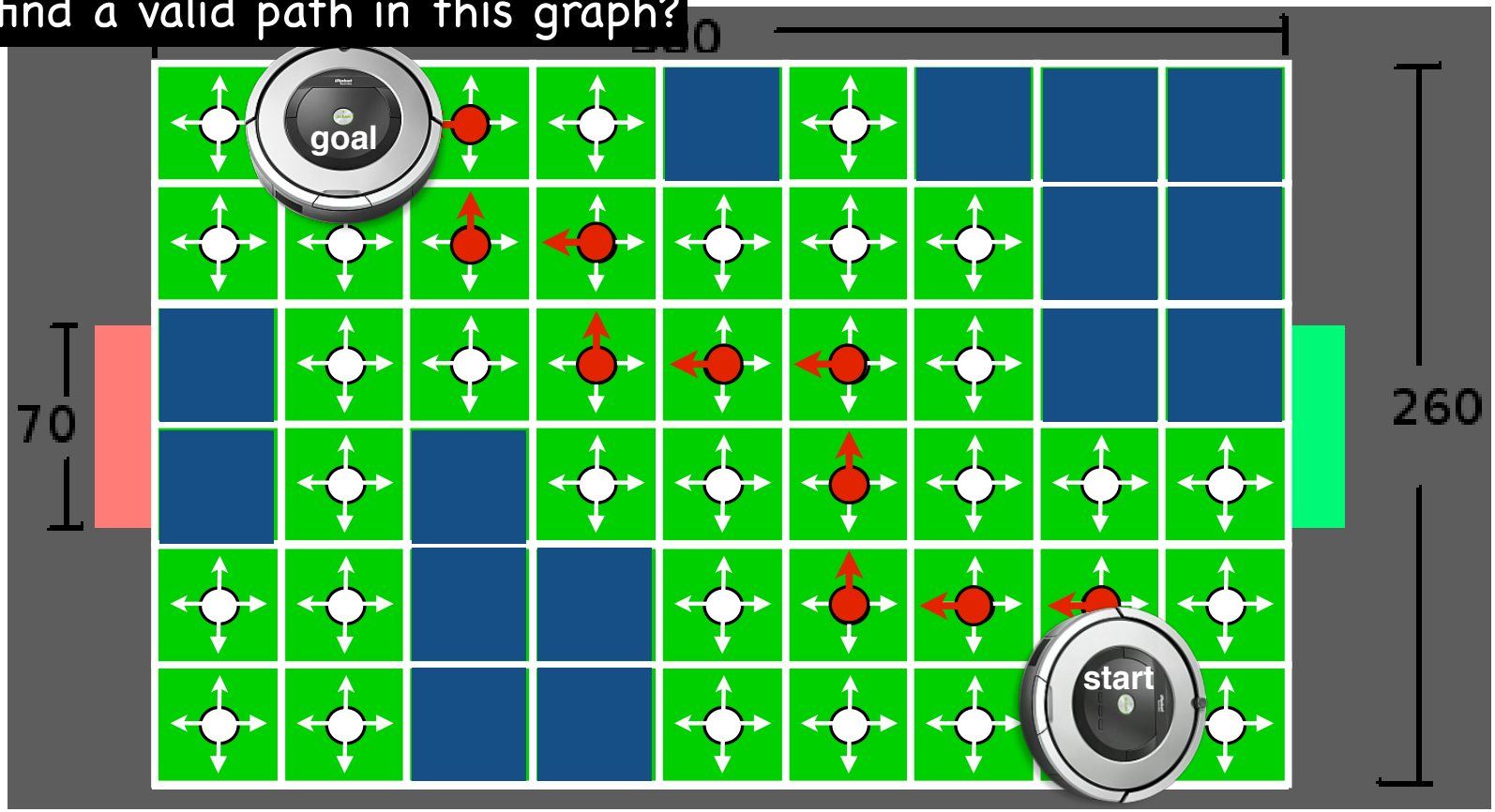


Consider all possible poses as uniformly distributed array of cells in a graph

Edges connect adjacent cells, weighted by distance

Cells are invalid where its associated robot pose results in a collision

How to find a valid path in this graph?



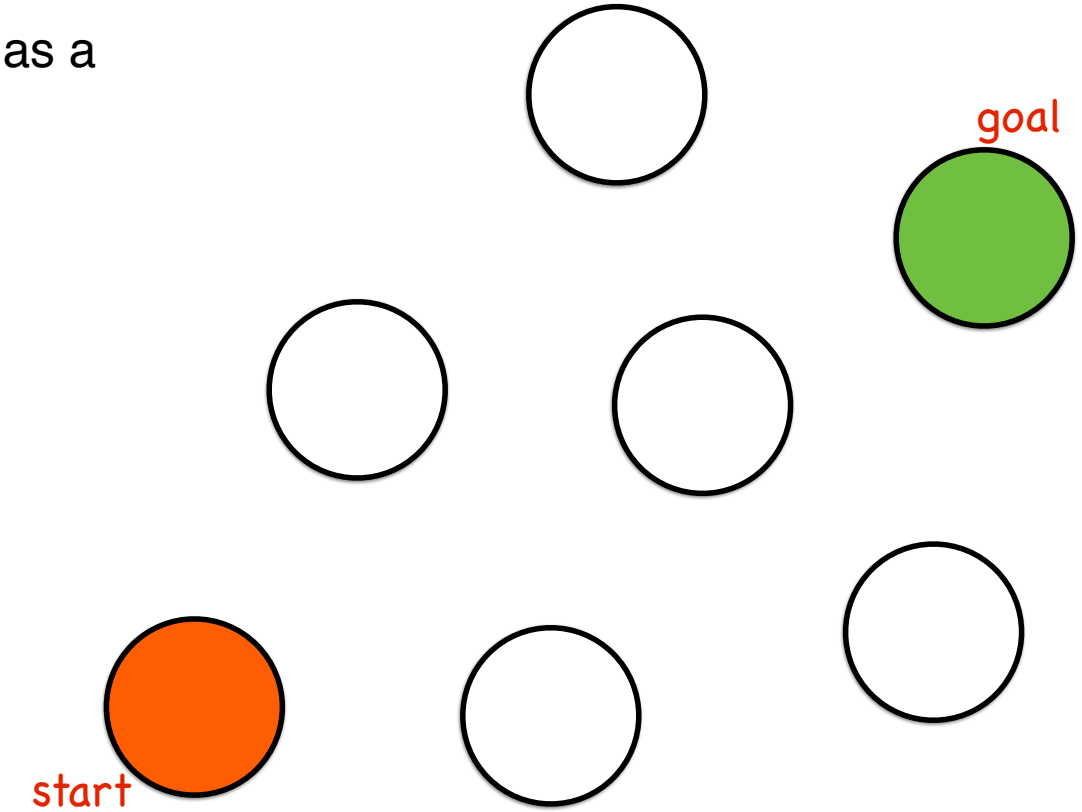
Approaches to motion planning

- 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 a simple search graph

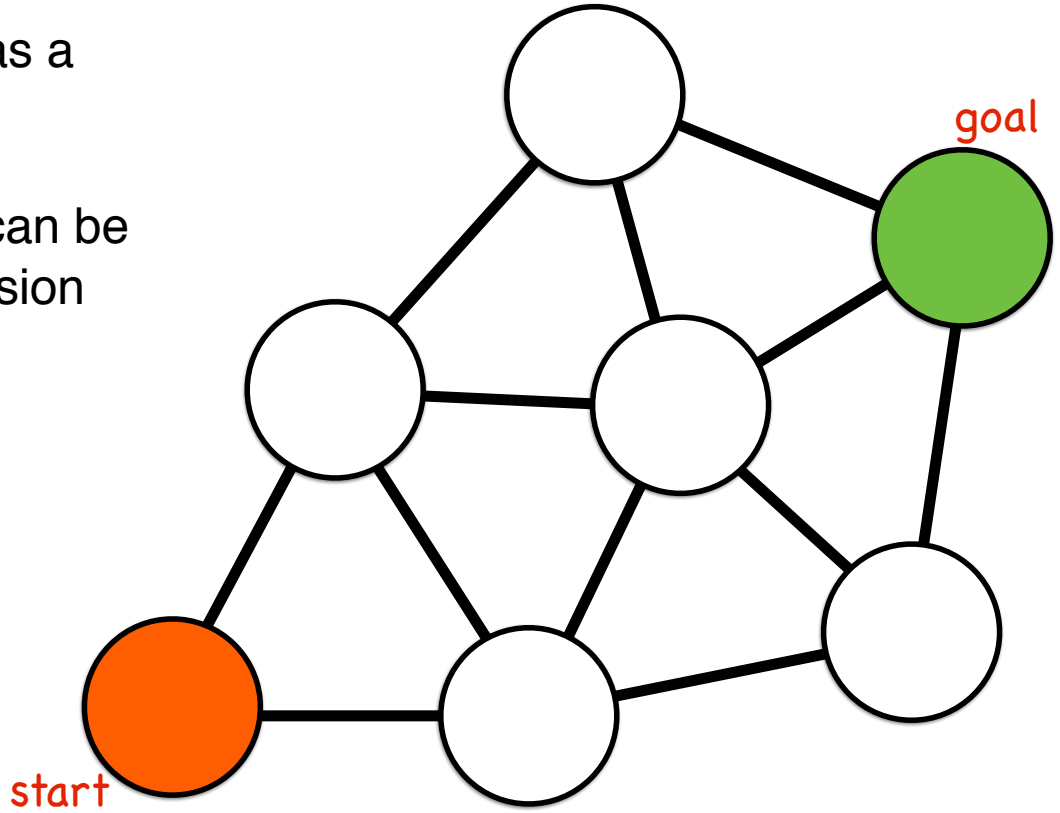
Consider each possible robot pose as a node V_i in a graph $G(V,E)$



Consider a simple search graph

Consider each possible robot pose as a node V_i in a graph $G(V,E)$

Graph edges E connect poses that can be reliably moved between without collision

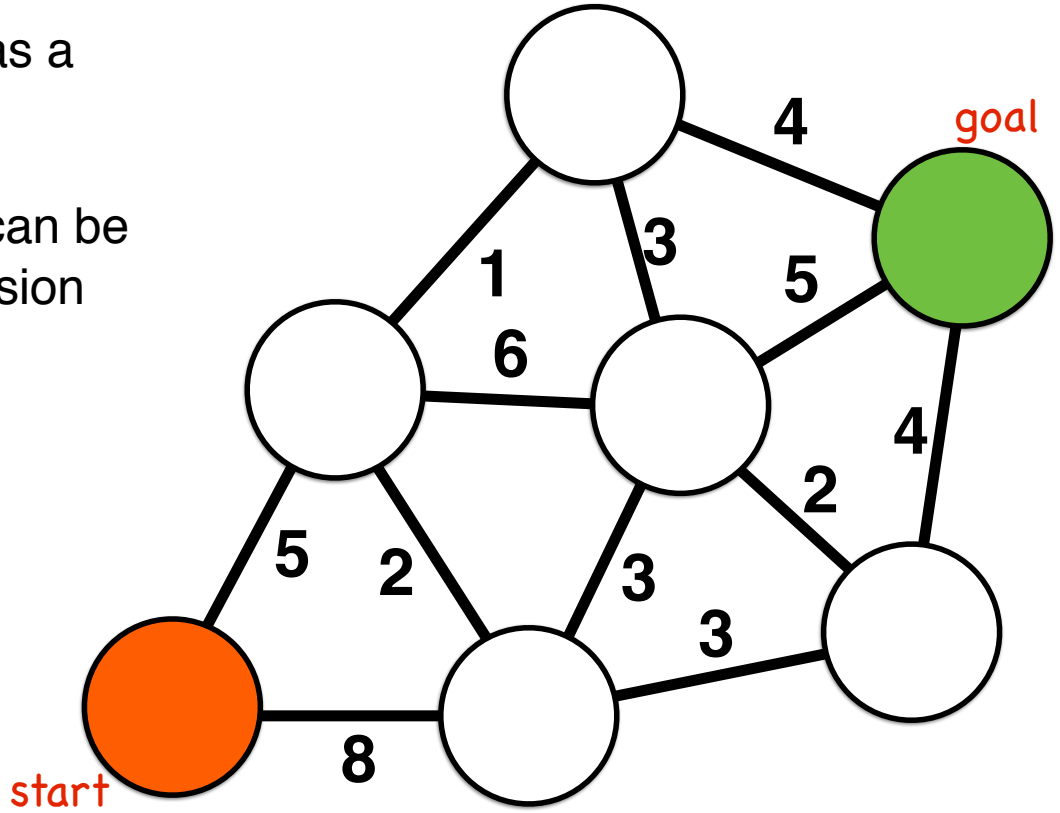


Consider a simple search graph

Consider each possible robot pose as a node V_i in a graph $G(V,E)$

Graph edges E connect poses that can be reliably moved between without collision

Edges have a cost for traversal



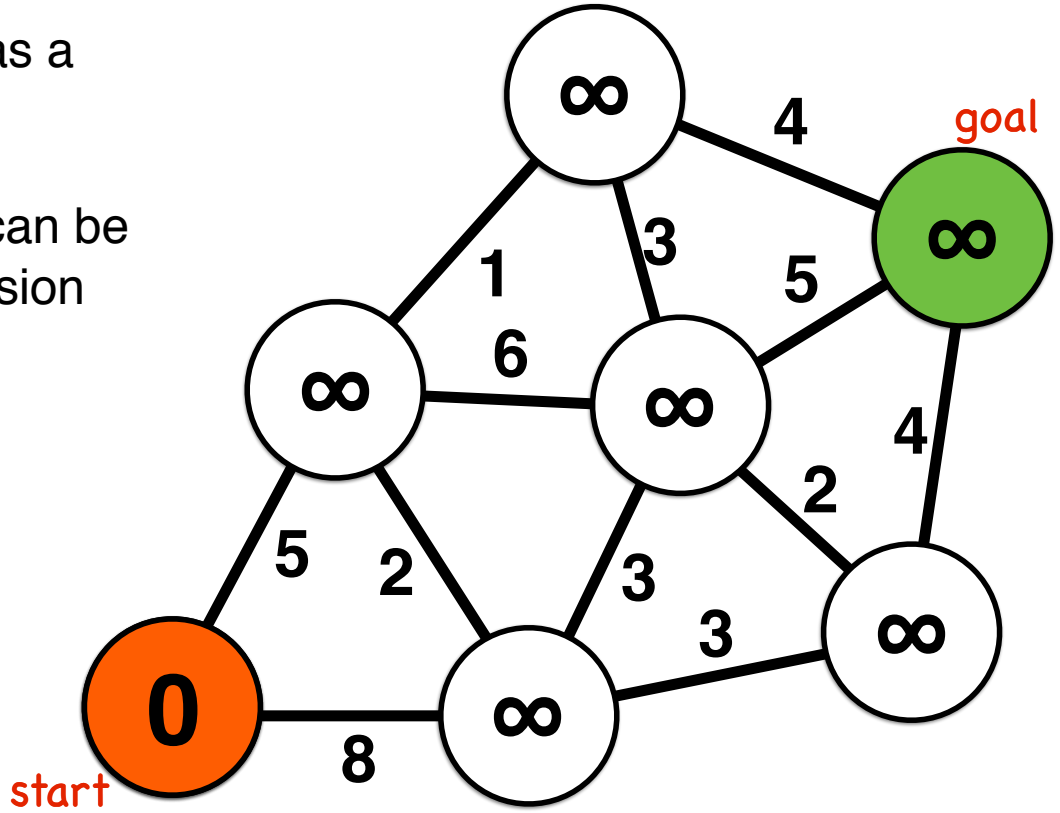
Consider a simple search graph

Consider each possible robot pose as a node V_i in a graph $G(V,E)$

Graph edges E connect poses that can be reliably moved between without collision

Edges have a cost for traversal

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



Consider a simple search graph

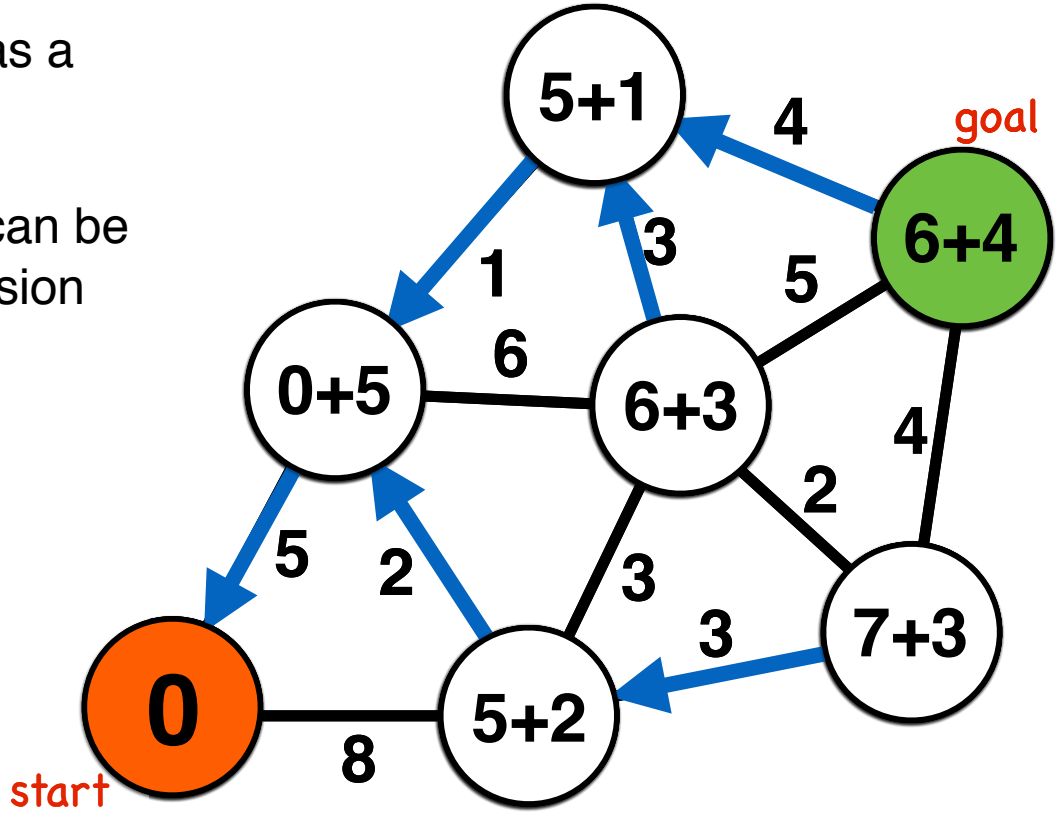
Consider each possible robot pose as a node V_i in a graph $G(V,E)$

Graph edges E connect poses that can be reliably moved between without collision

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

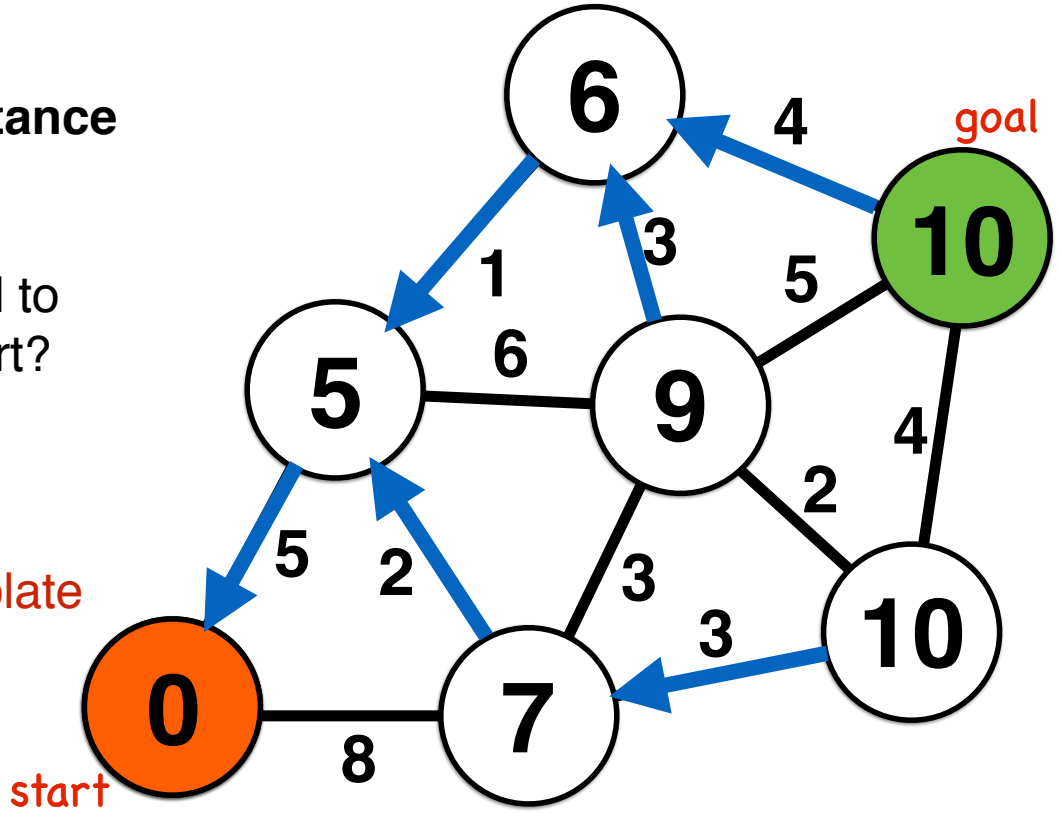


Path Planning as Graph Search

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

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

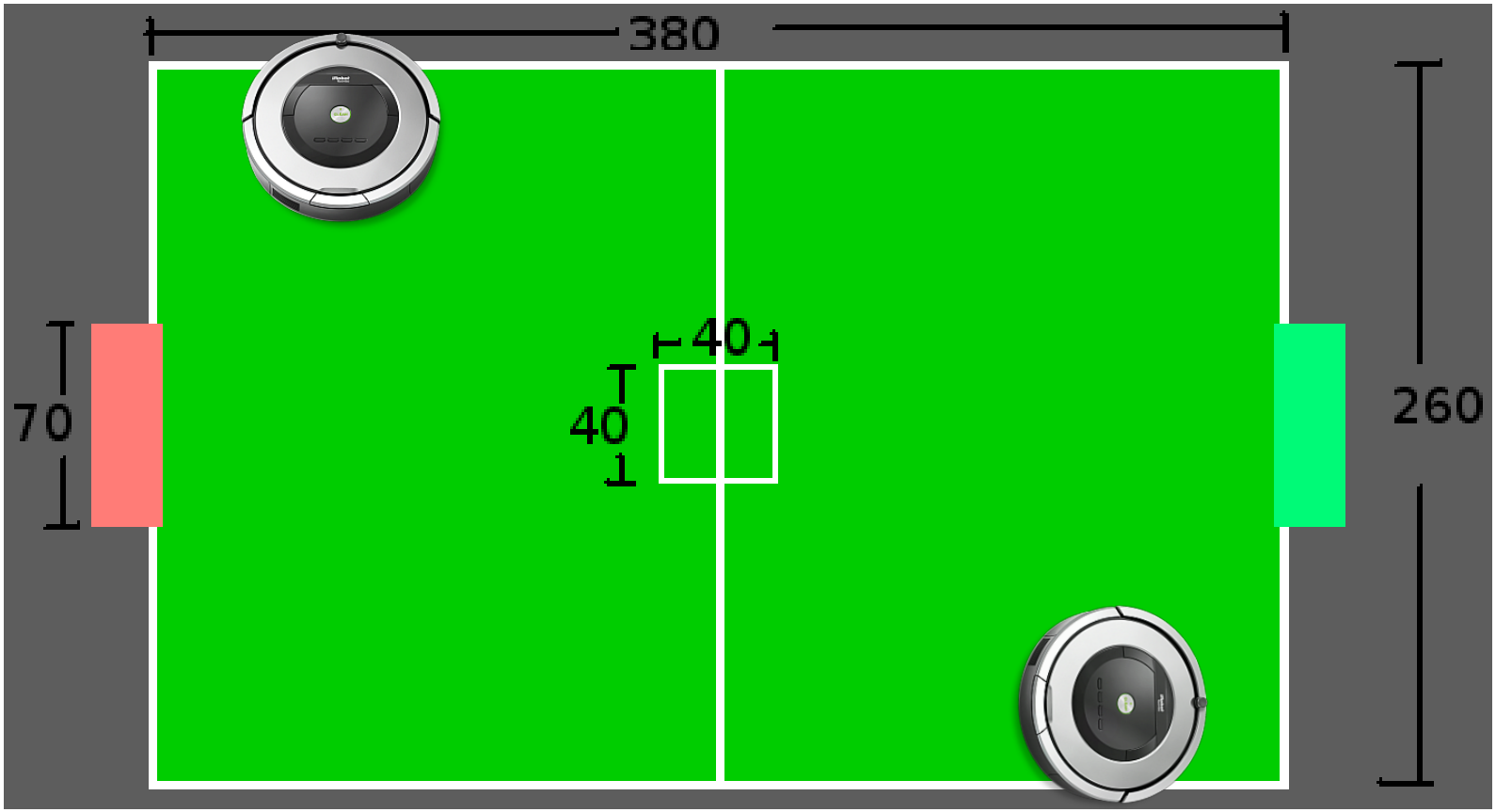
We will use a single algorithm template for our graph search computation



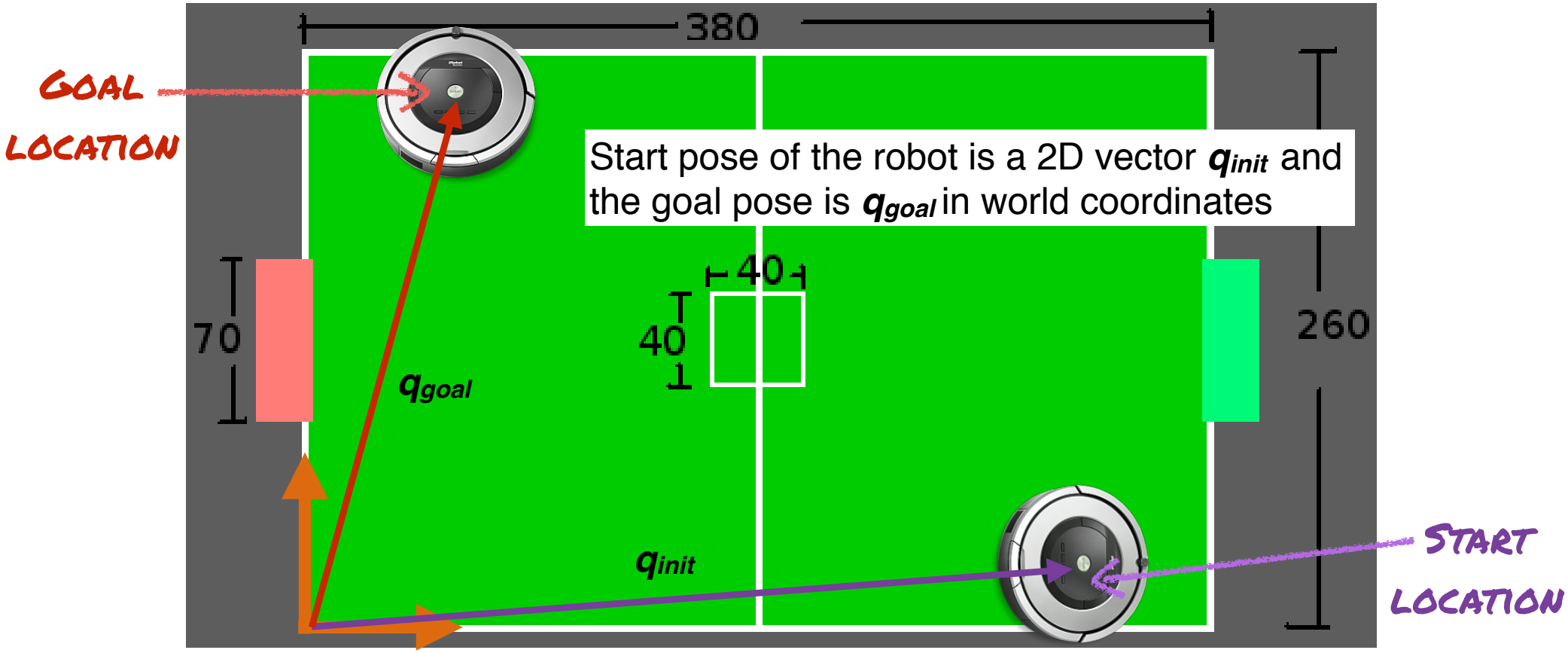
Depth-first search

intuition and walkthrough

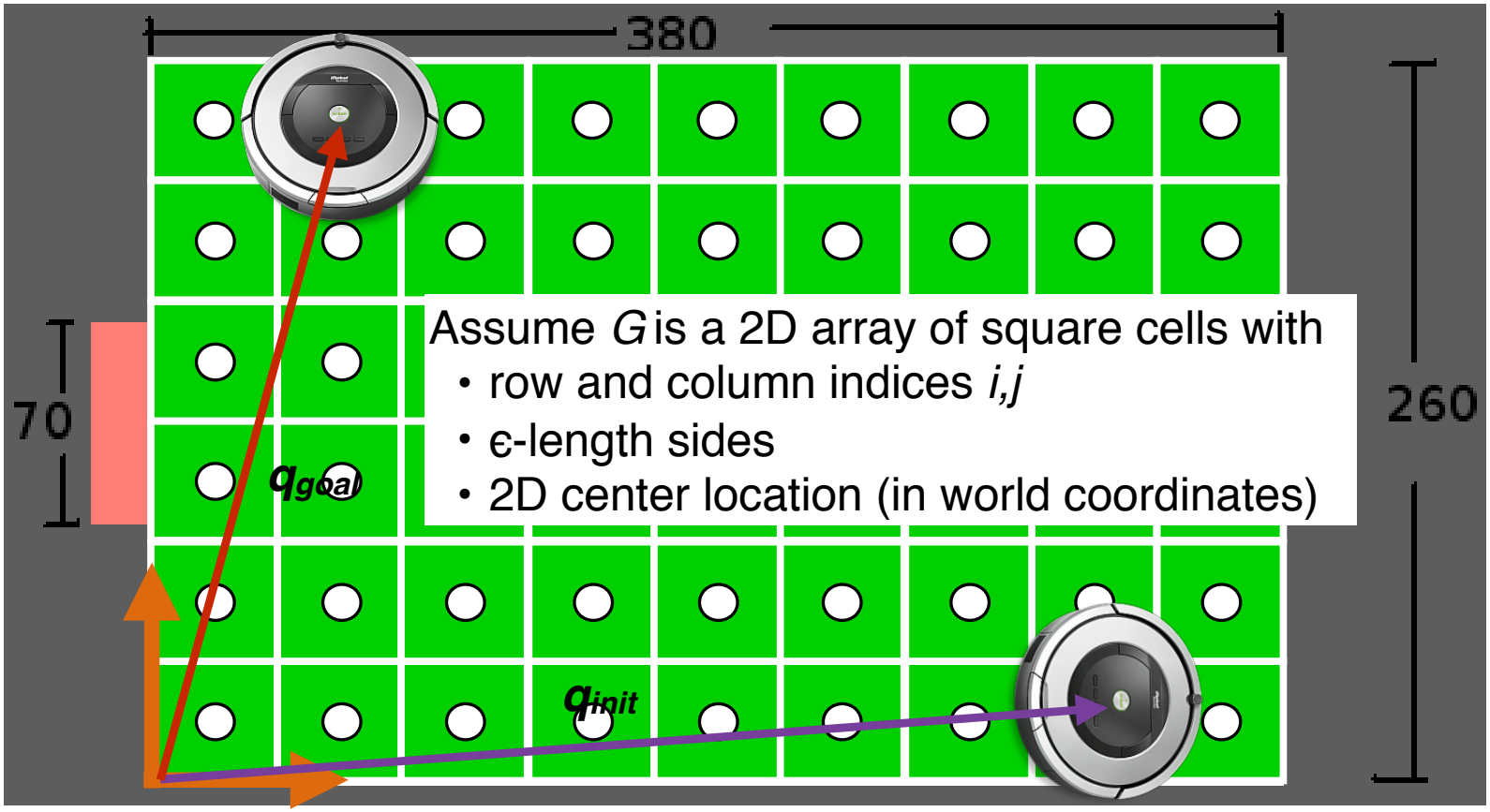
Depth-first search



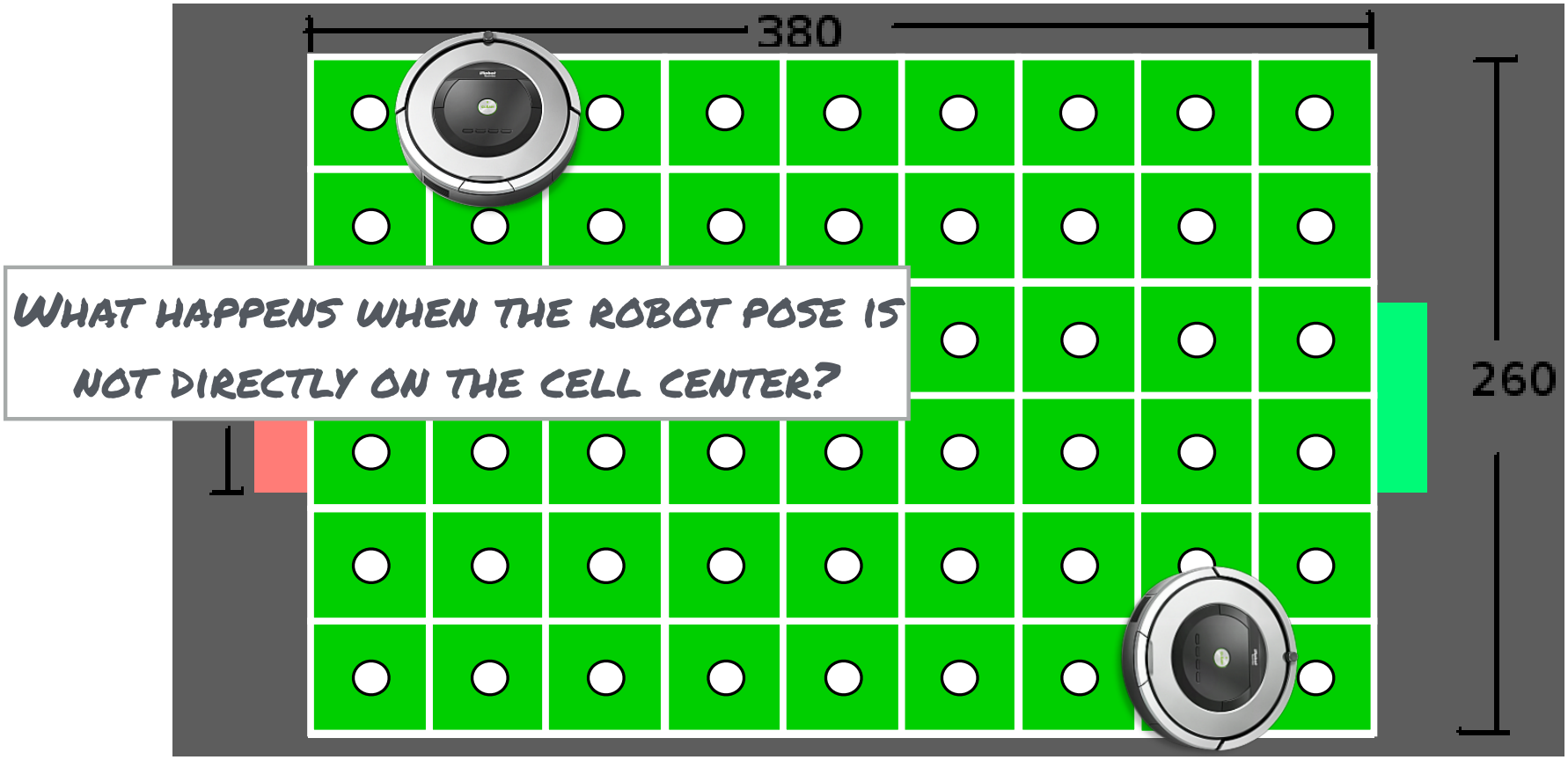
Depth-first search



Depth-first search



Depth-first search



Graph Accessibility

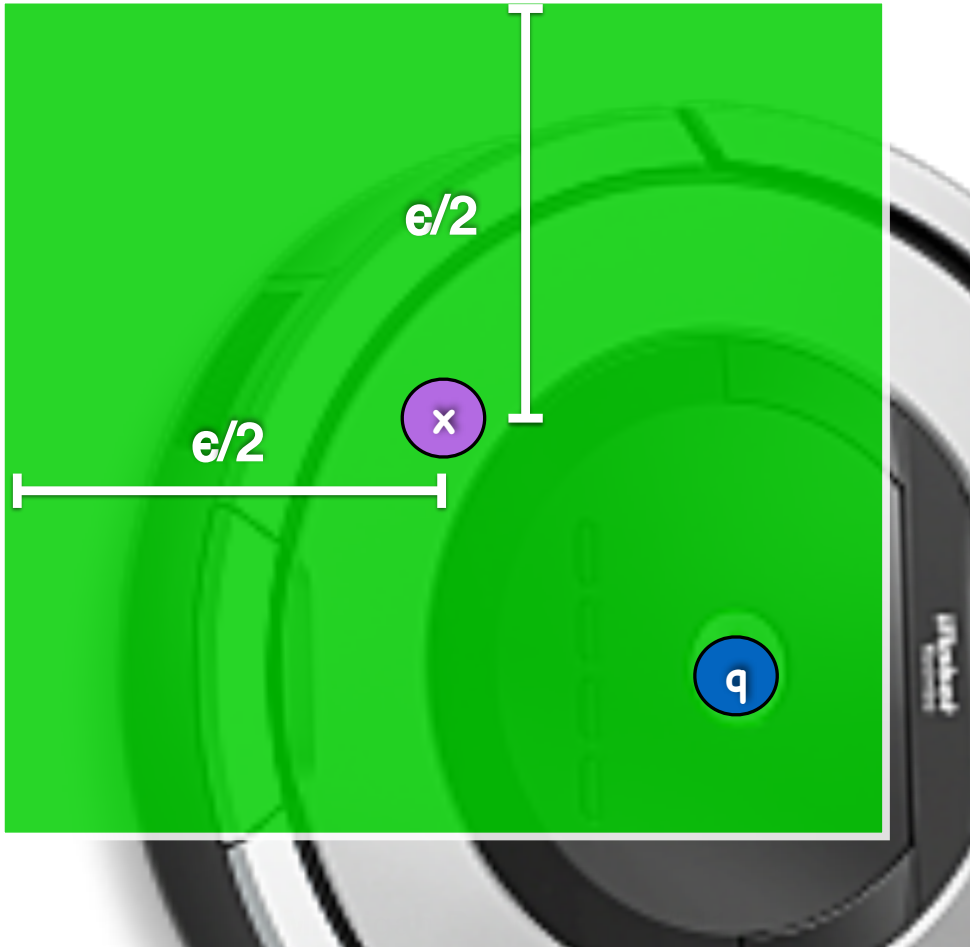
What happens when the robot pose is not directly on the cell center?

Graph Accessibility

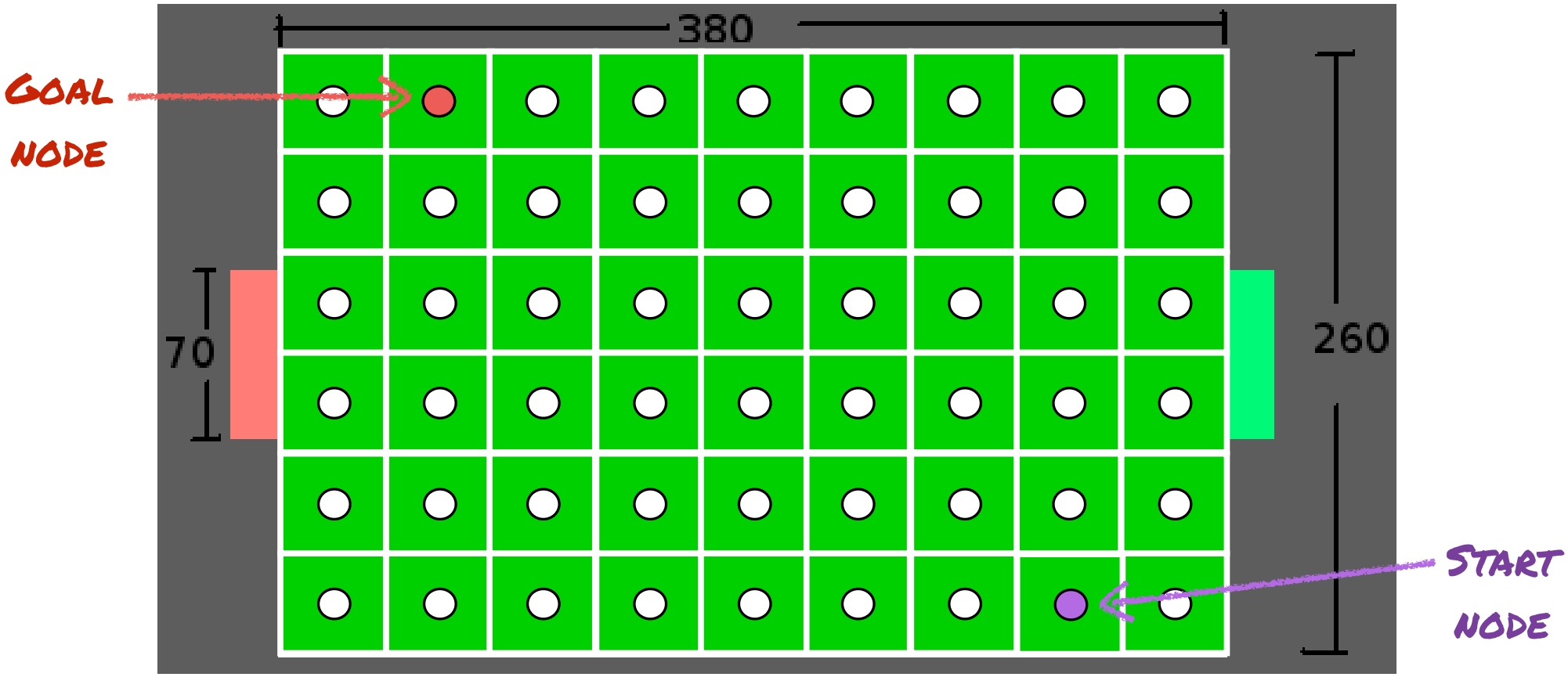
A graph node $G_{i,j}$ represents a region of space contained by its cell

Start node: the robot accesses graph G at the cell that contains location $\boldsymbol{q_{init}}$

Goal node: the robot departs graph G at the cell that contains location $\boldsymbol{q_{goal}}$



Depth-first search

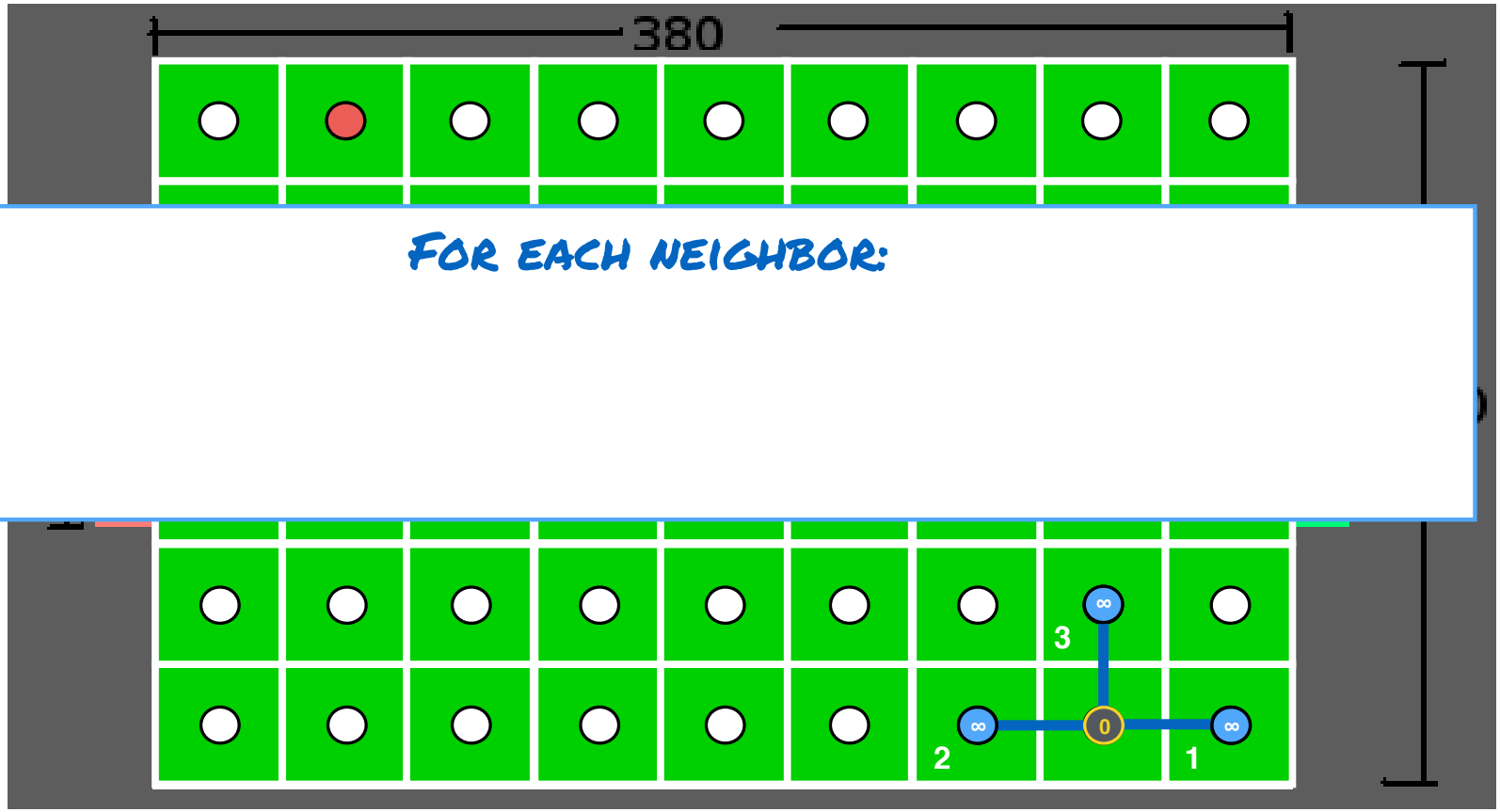


Depth-first search

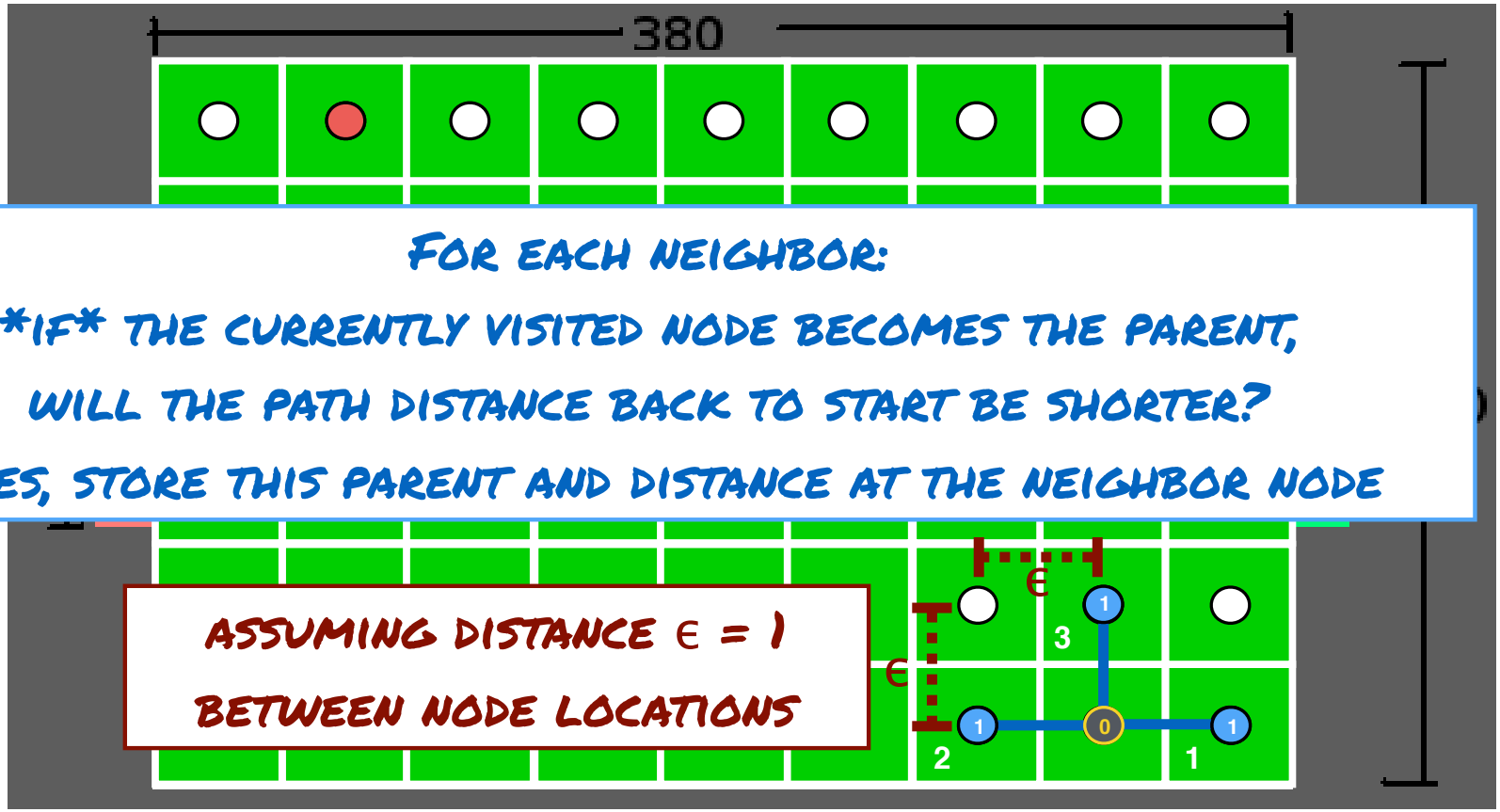
Depth-first search

Depth-first search

Depth-first search

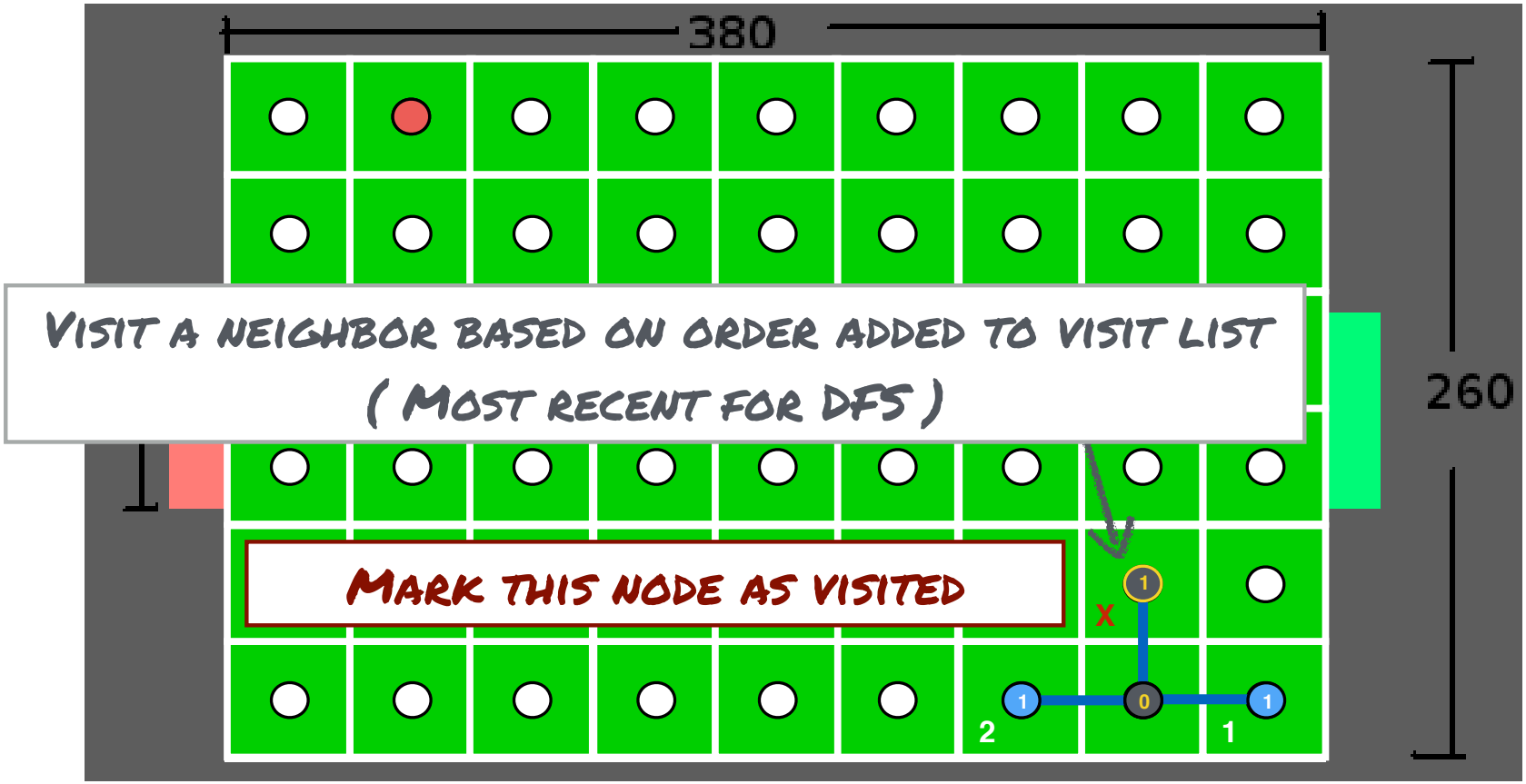


Depth-first search



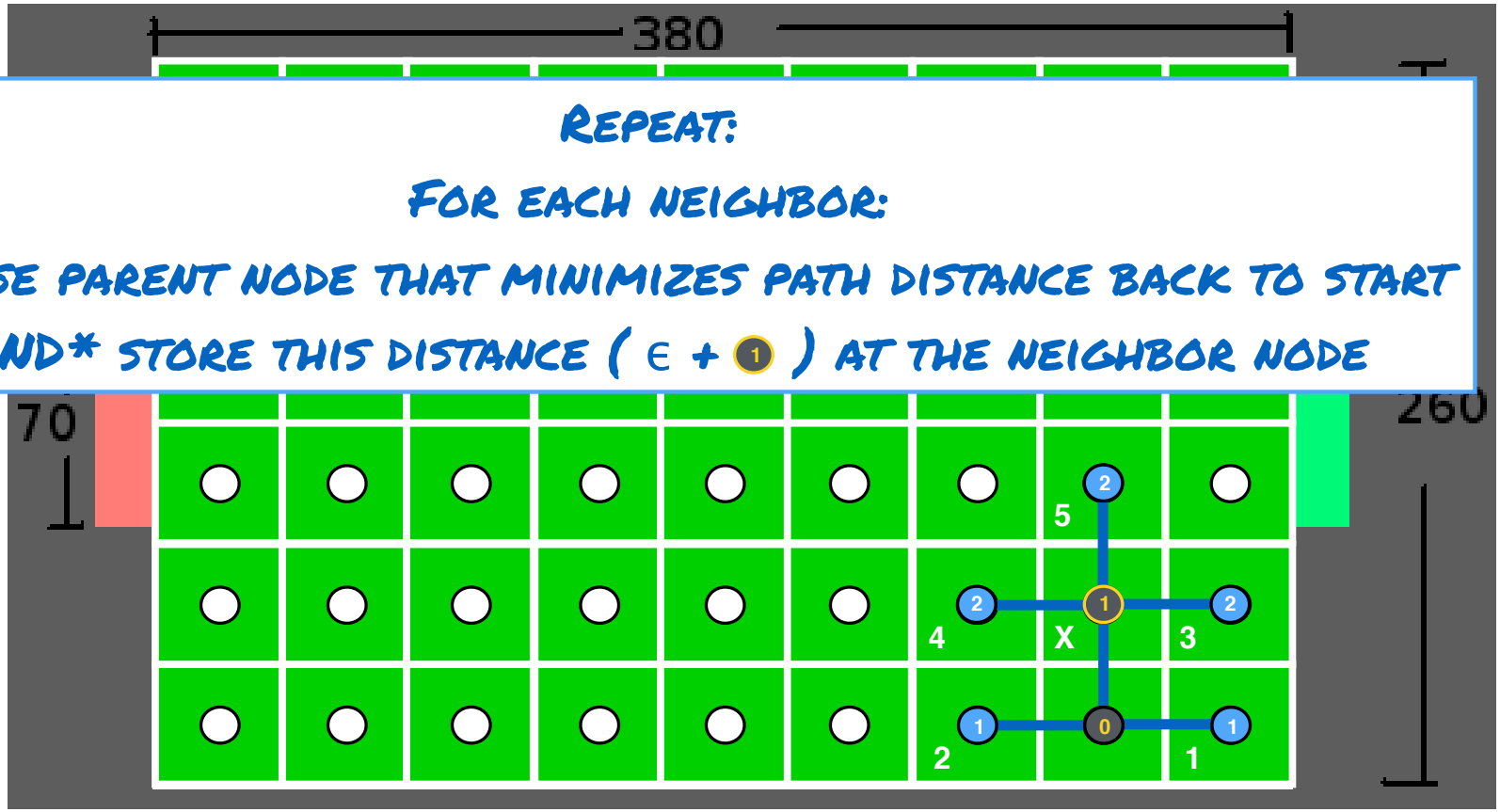
Depth-first search

Depth-first search



Depth-first search

Depth-first search



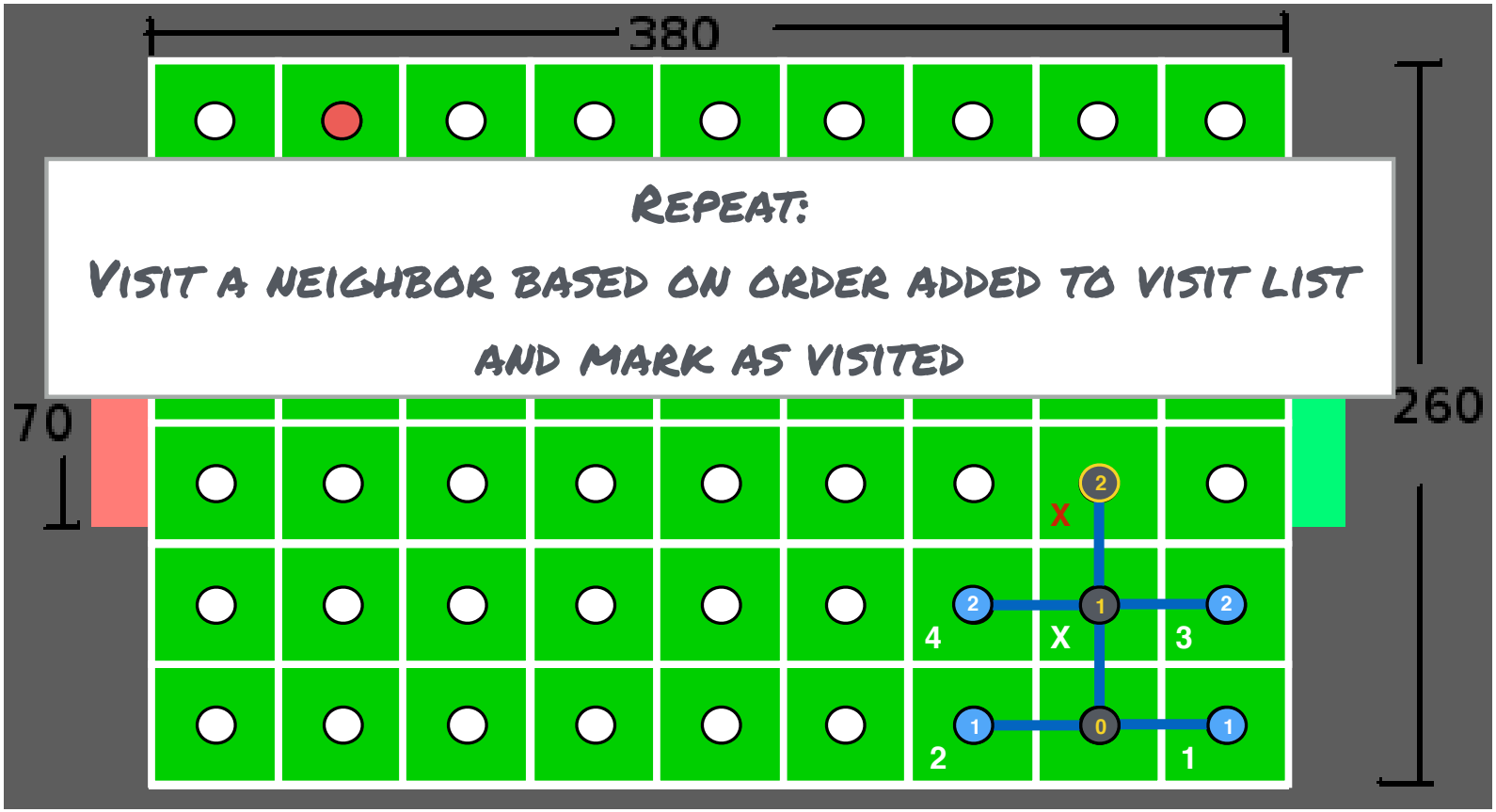
Repeat:

For each neighbor:

choose parent node that minimizes path distance back to start

AND store this distance (∈ + 1) at the neighbor node

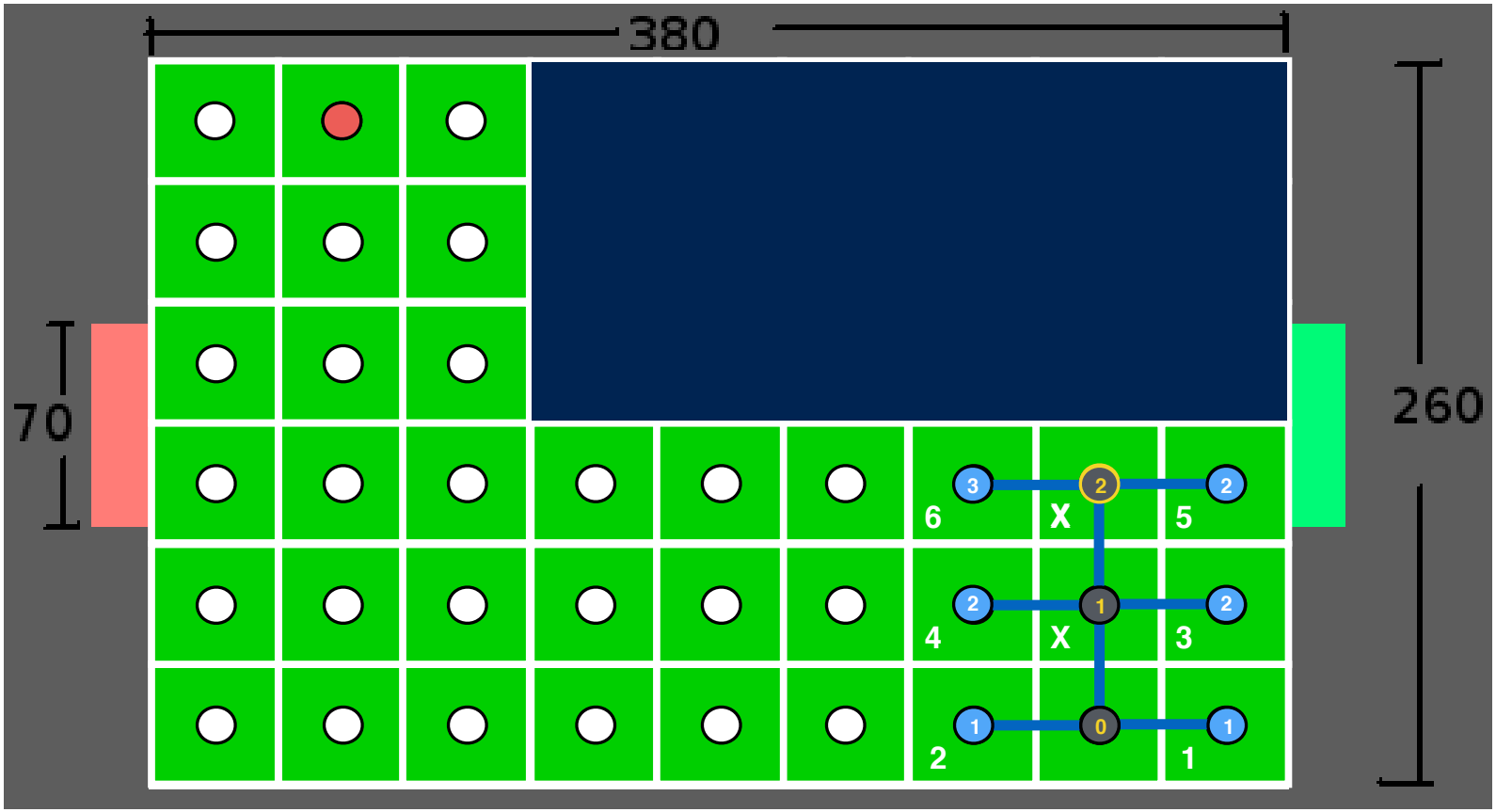
Depth-first search



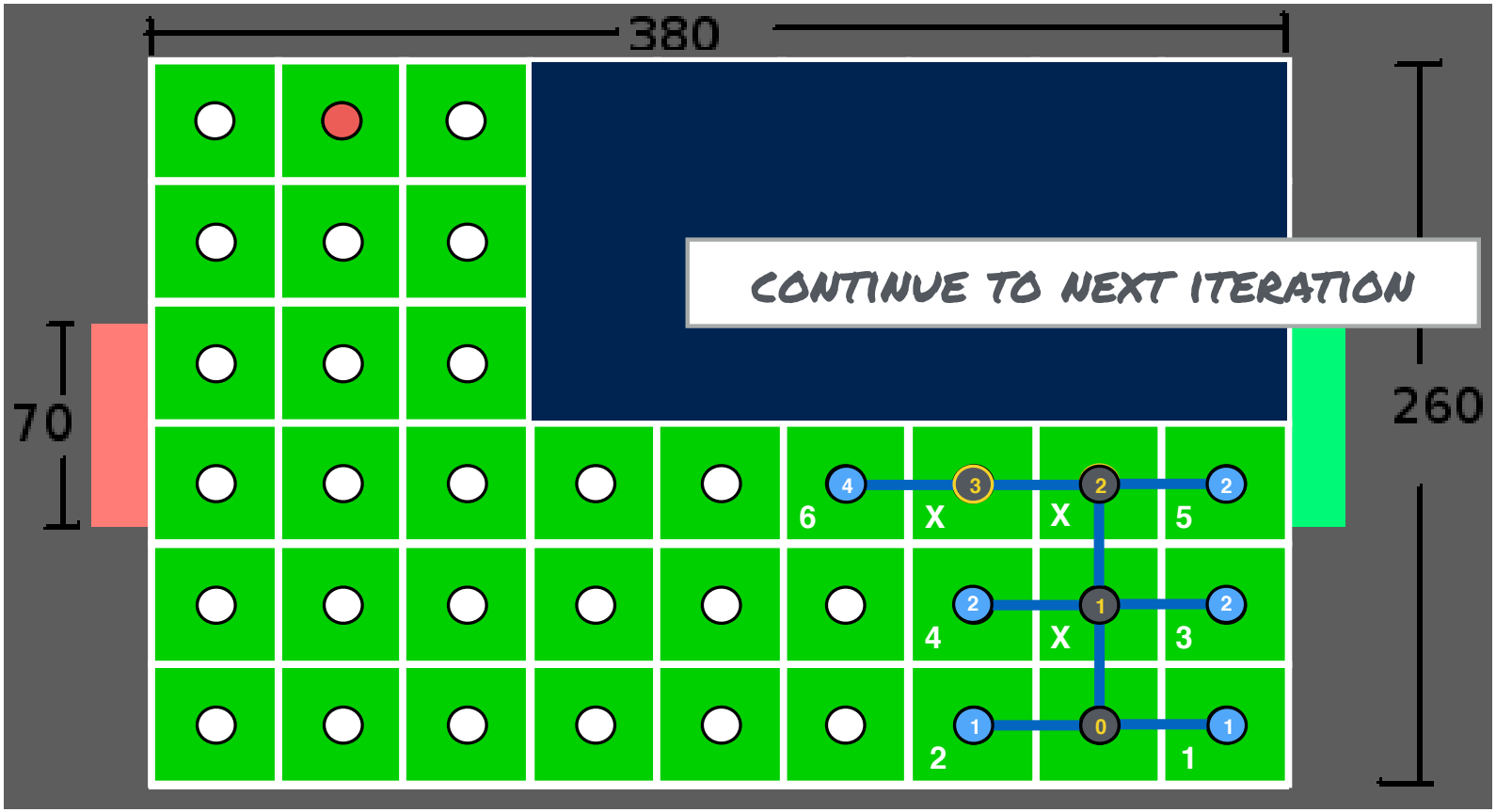
Depth-first search

Depth-first search

Depth-first search

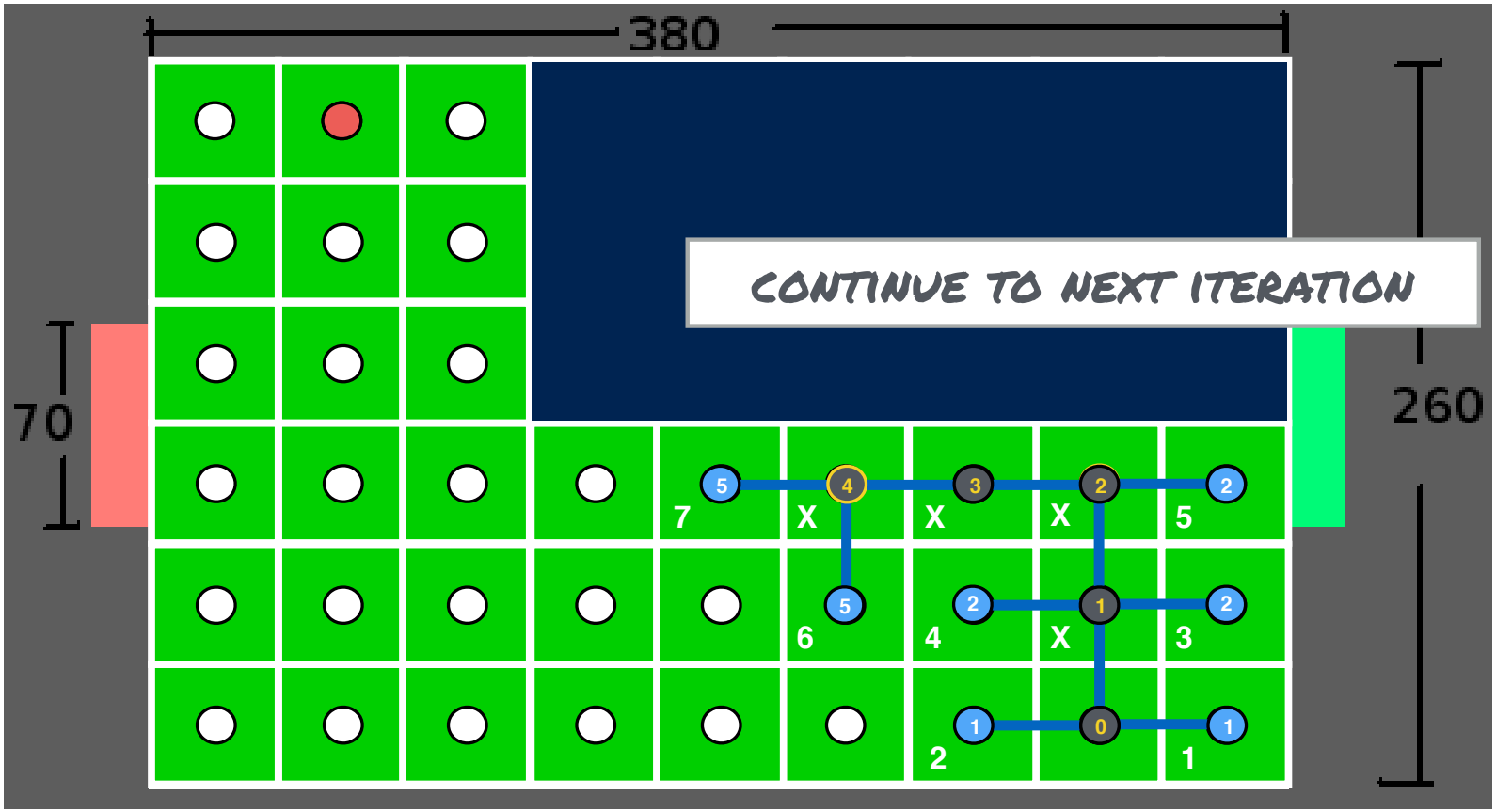


Depth-first search

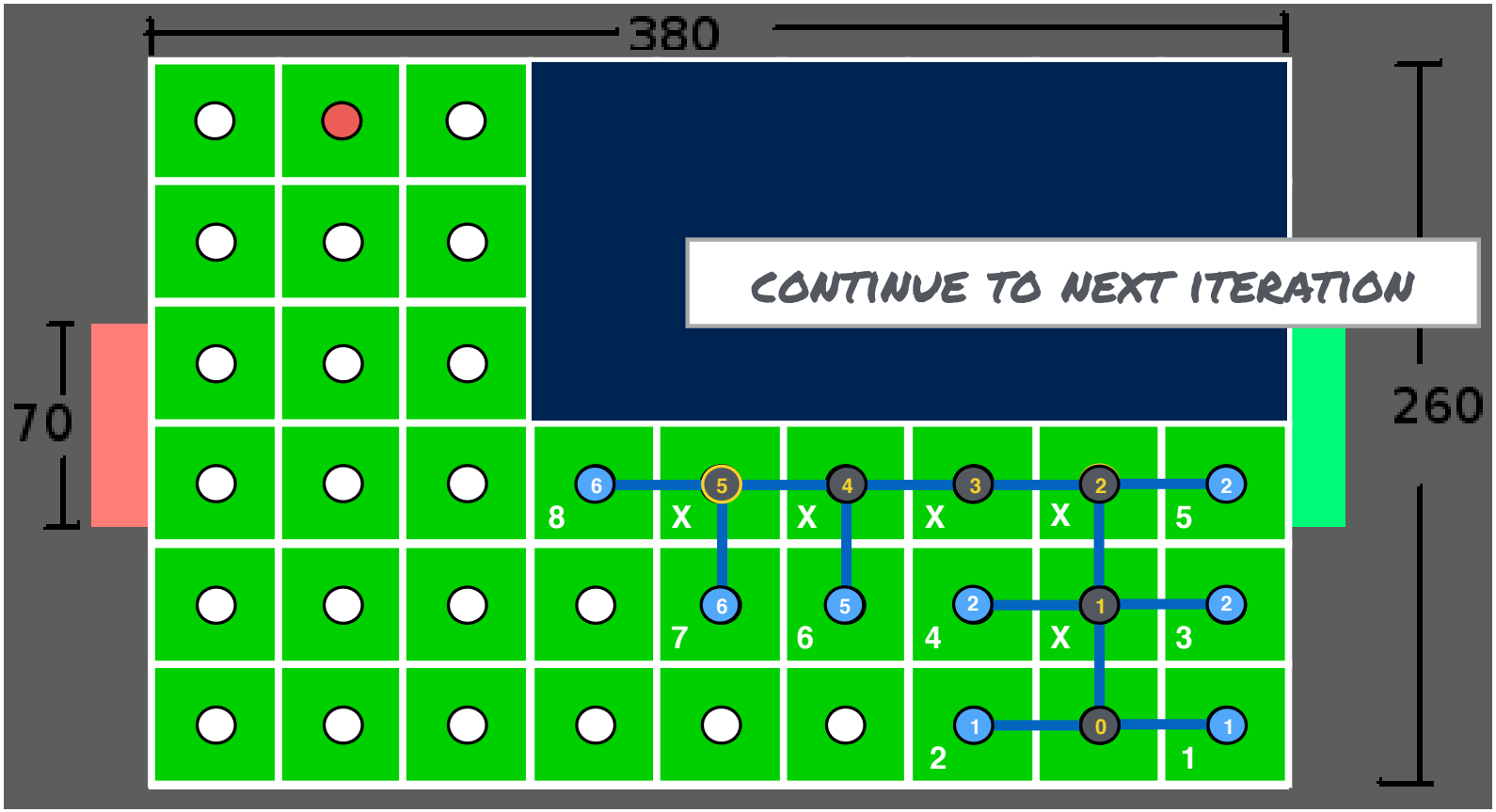


Depth-first search

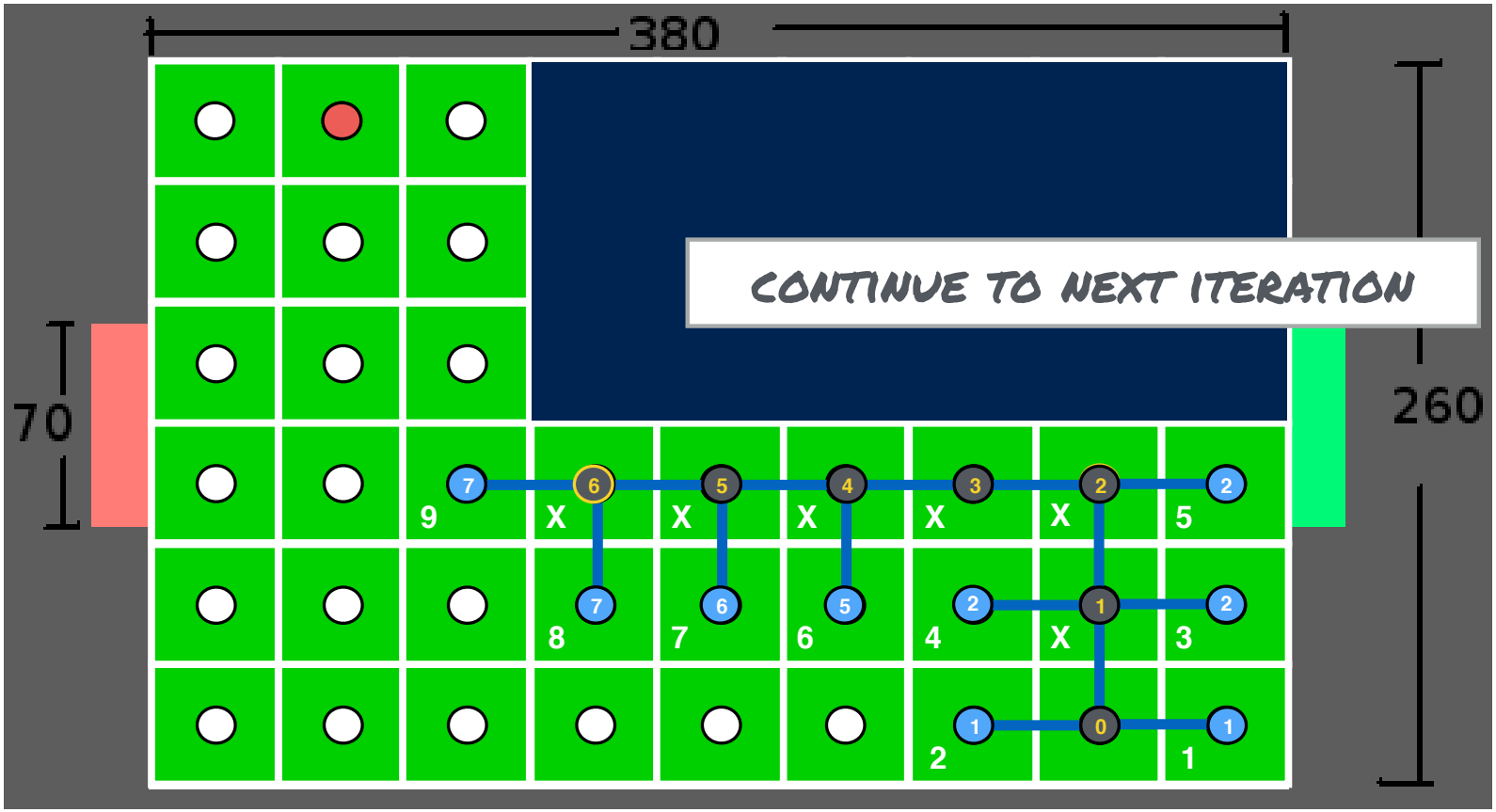
Depth-first search



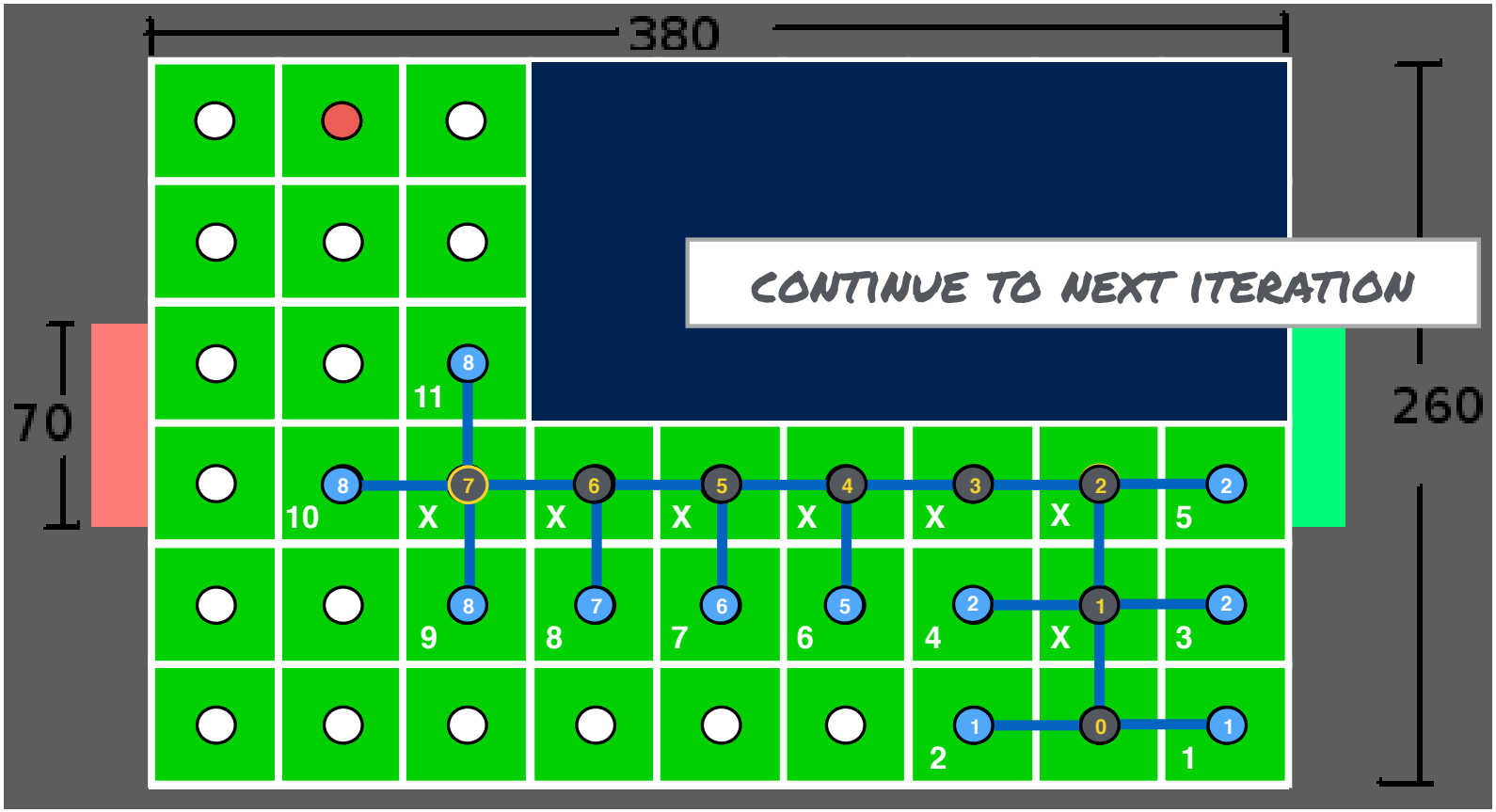
Depth-first search



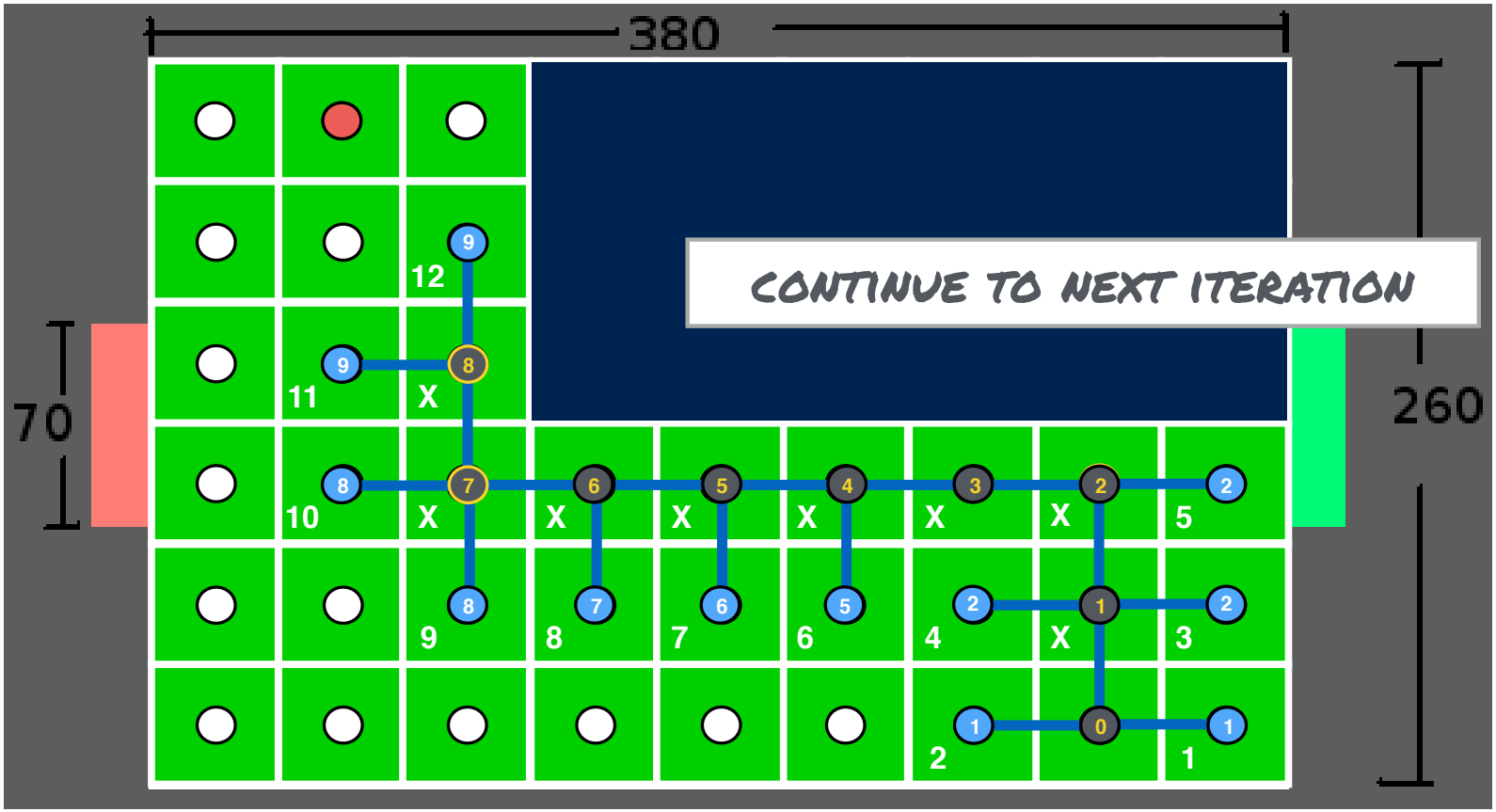
Depth-first search



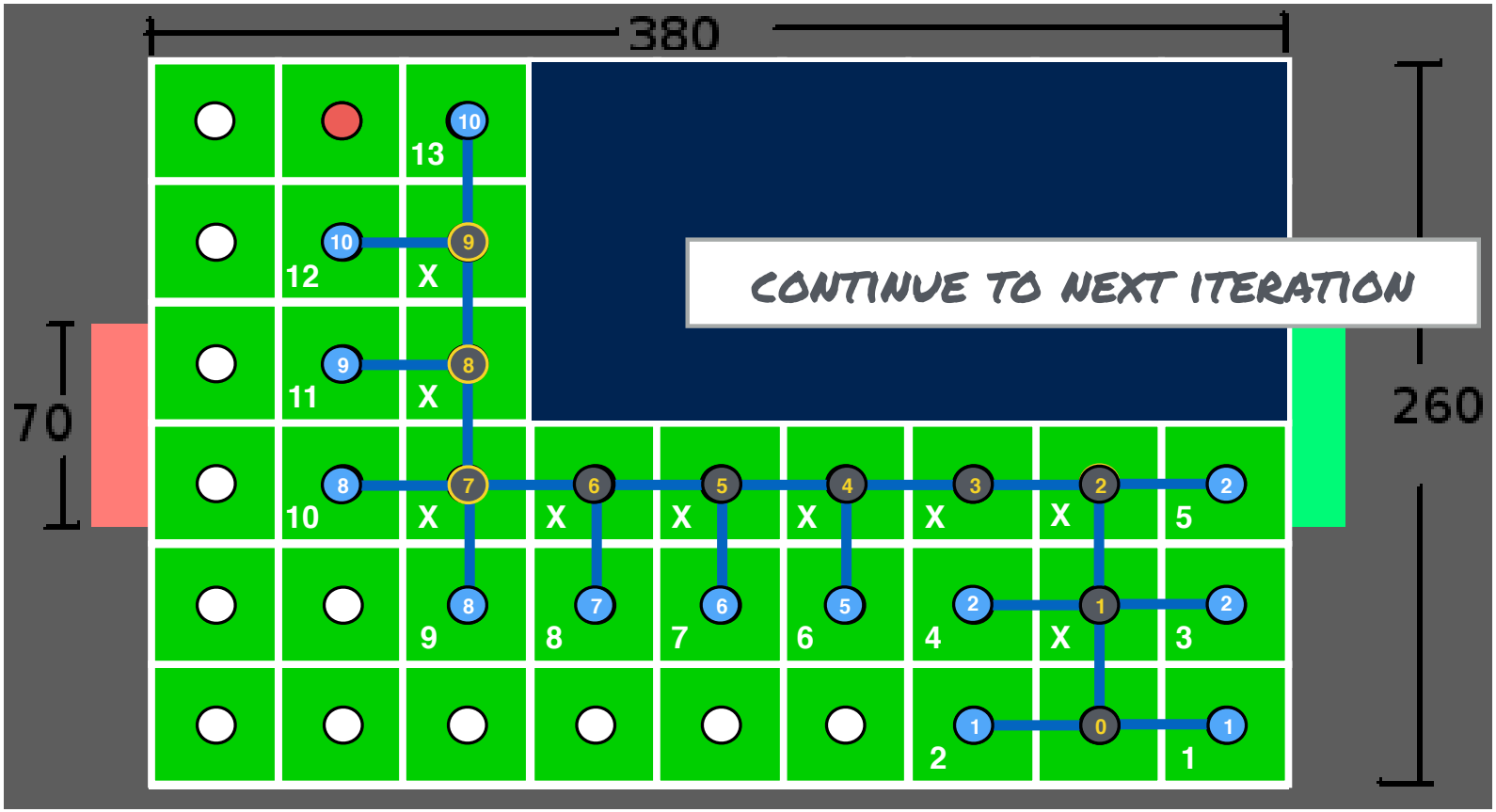
Depth-first search



Depth-first search

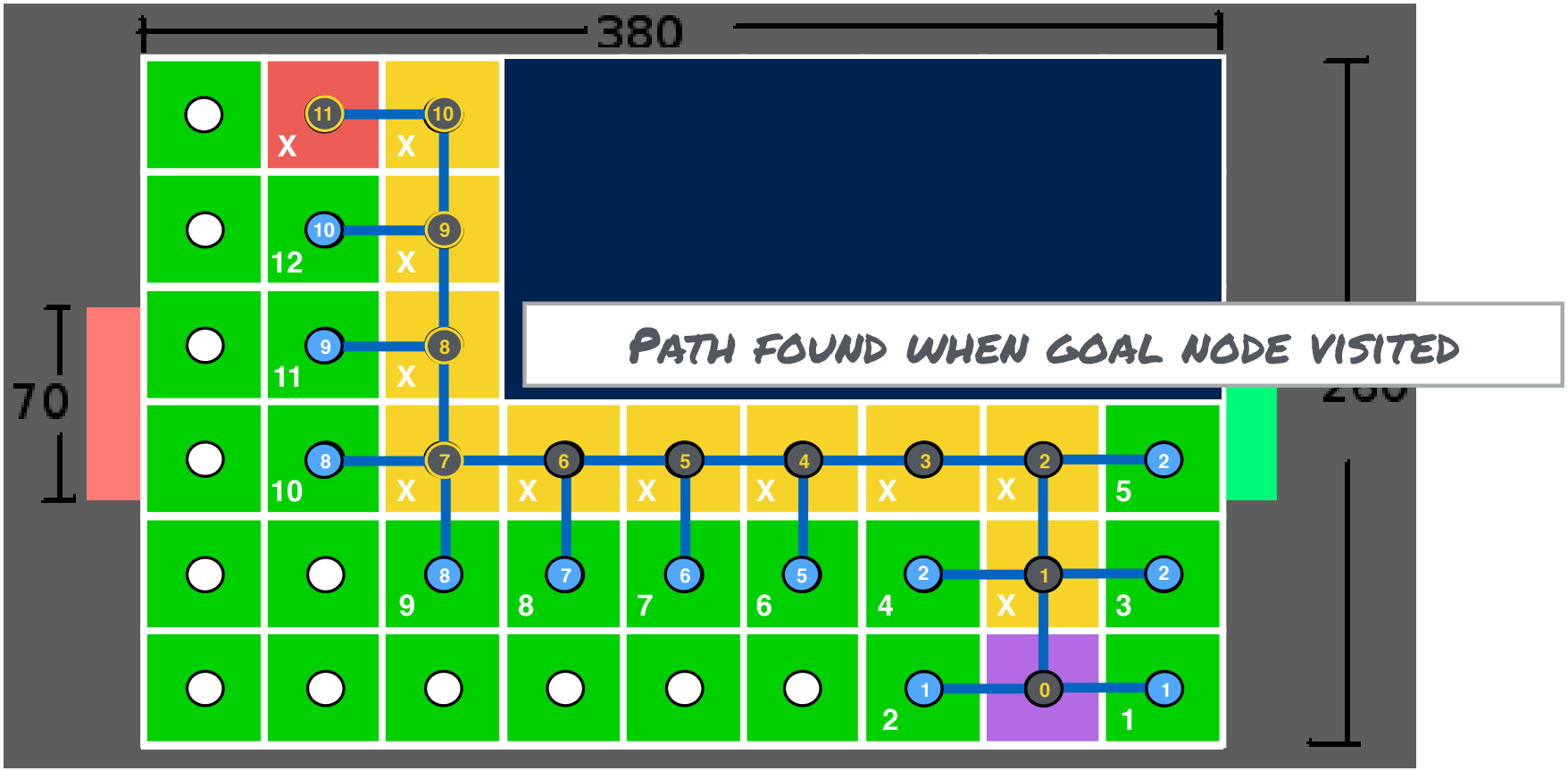


Depth-first search



Depth-first search

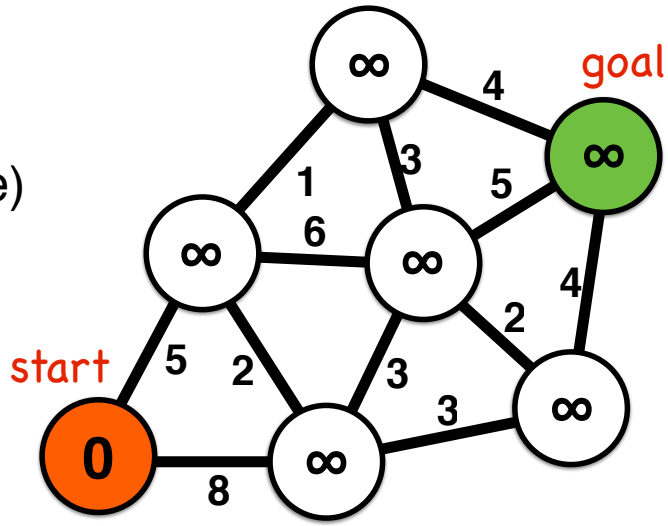
Depth-first search



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



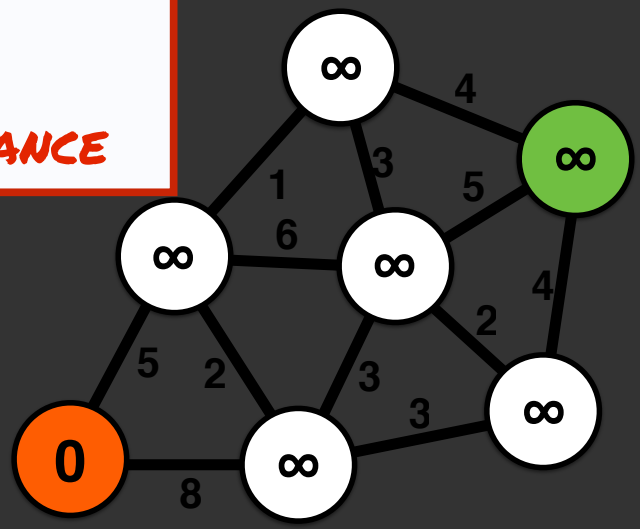
Search algorithm template

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

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



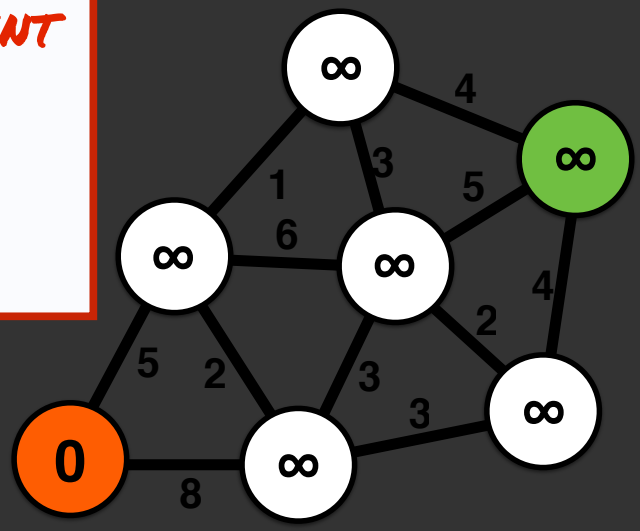
Search algorithm template

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

end for loop
end while loop
output ← parent, distance

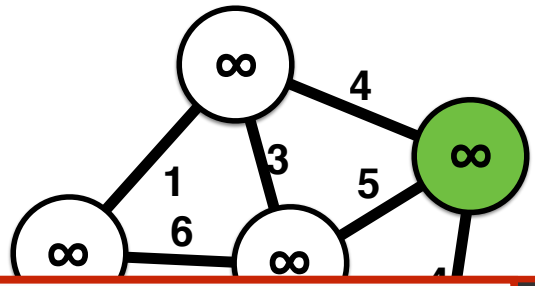
MAIN LOOP

- 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



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



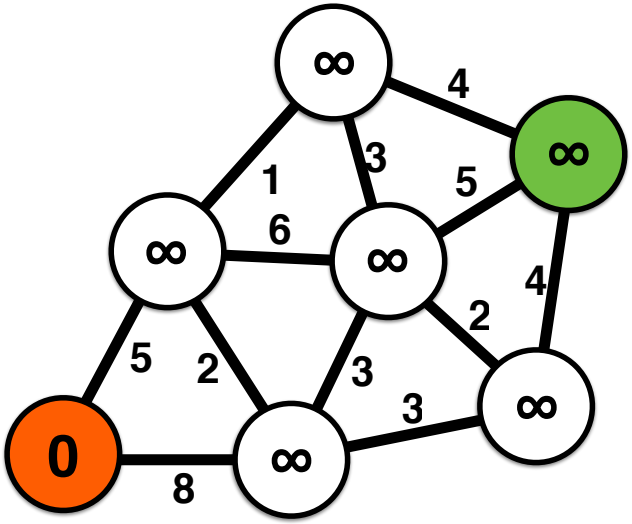
FOR EACH ITERATION ON A SINGLE NODE

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

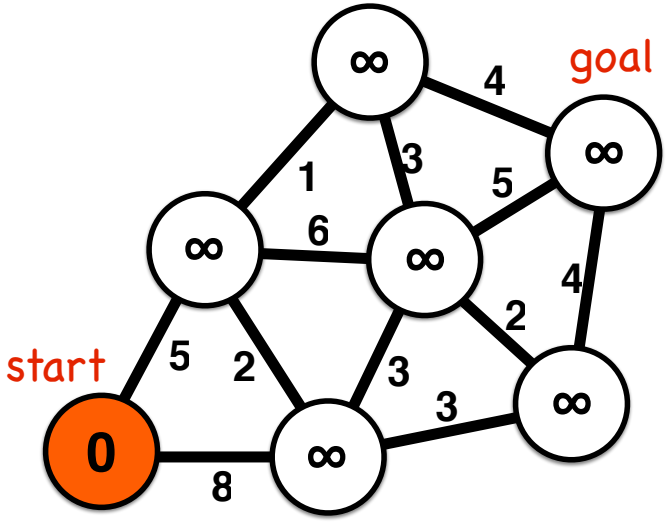
OUTPUT THE RESULTING ROUTING AND PATH DISTANCE AT EACH NODE



Depth-first search

Search algorithm template

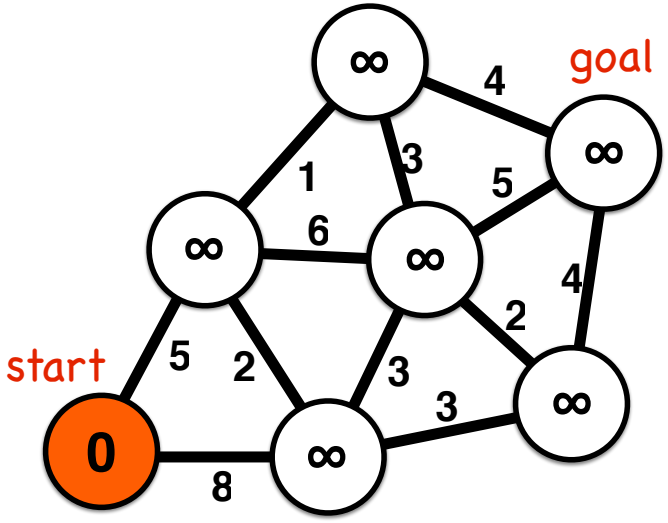
all nodes ← {$dist_{start}$ ← infinity, $parent_{start}$ ← none, $visited_{start}$ ← false}
start_node ← {$dist_{start}$ ← 0, $parent_{start}$ ← none, $visited_{start}$ ← true}
visit_list ← start_node
    while visit_list != empty && current_node != goal
        cur_node ← **highestPriority**(visit_list)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

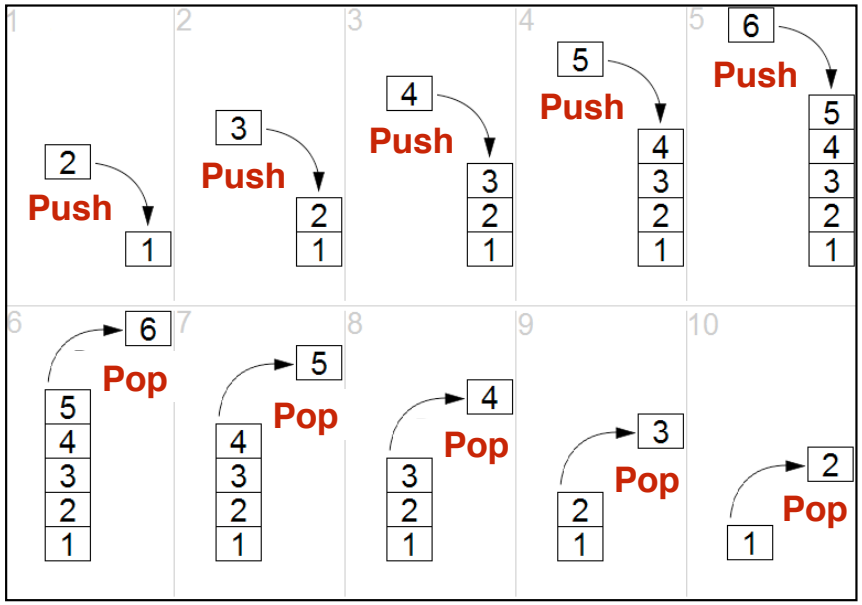
PRIORITY:
MOST RECENT

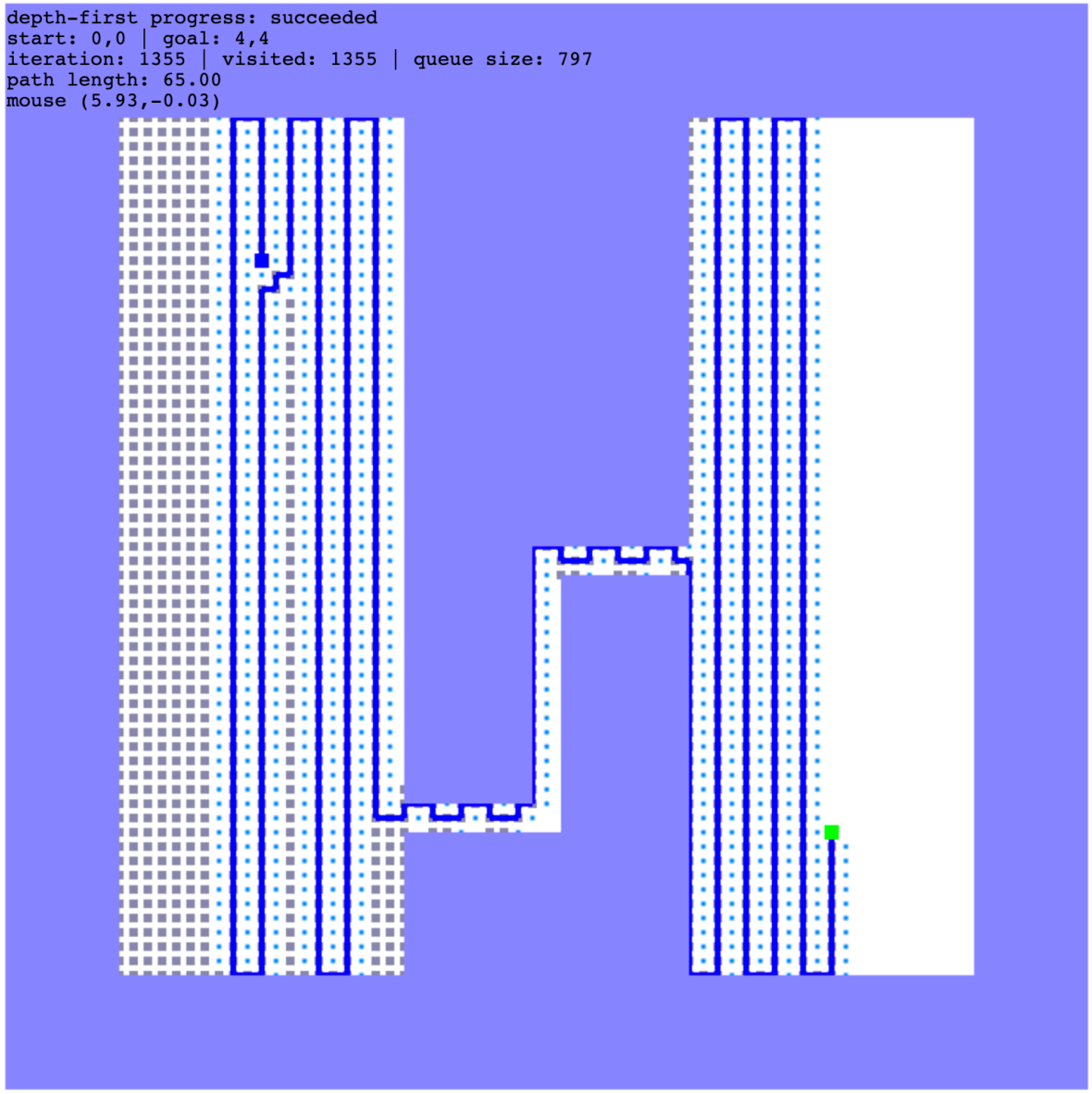


Stack data structure

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

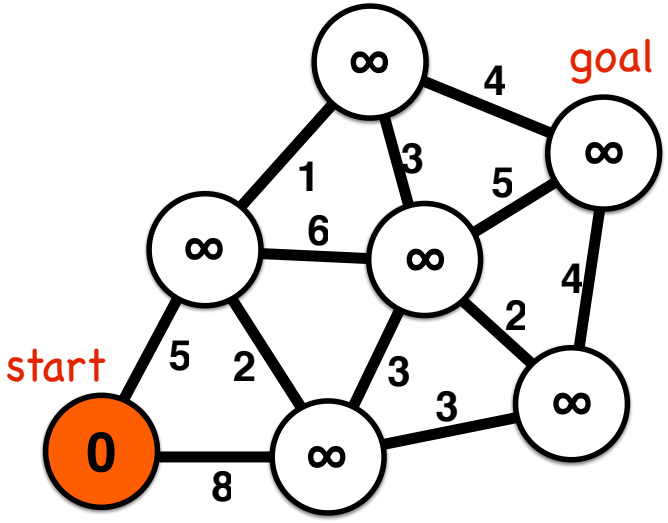



depth-first progress: succeeded
start: 0,0 | goal: 4,4
iteration: 1355 | visited: 1355 | queue size: 797
path length: 65.00
mouse (5.93,-0.03)

Breadth-first search

Search algorithm template

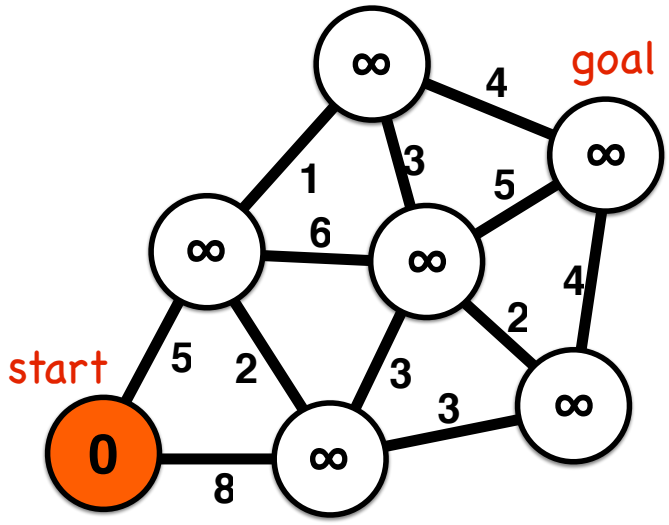
all nodes ← {$dist_{start}$ ← infinity, $parent_{start}$ ← none, $visited_{start}$ ← false}
start_node ← {$dist_{start}$ ← 0, $parent_{start}$ ← none, $visited_{start}$ ← true}
visit_list ← start_node
    while visit_list != empty && current_node != goal
        cur_node ← highestPriority(visit_list)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

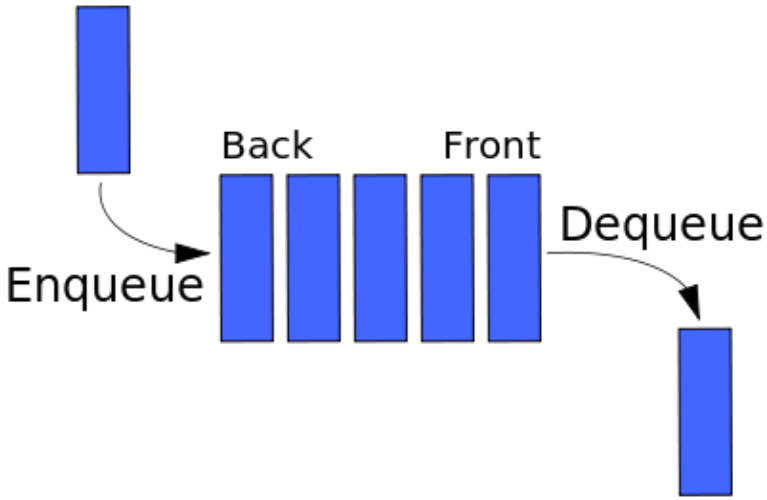
all nodes ← {$dist_{start}$ ← infinity, $parent_{start}$ ← none, $visited_{start}$ ← false}
start_node ← {$dist_{start}$ ← 0, $parent_{start}$ ← none, $visited_{start}$ ← true}
visit_queue ← start_node
    while **visit_queue** != empty && current_node != goal
        cur_node ← **dequeue**(**visit_queue**)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

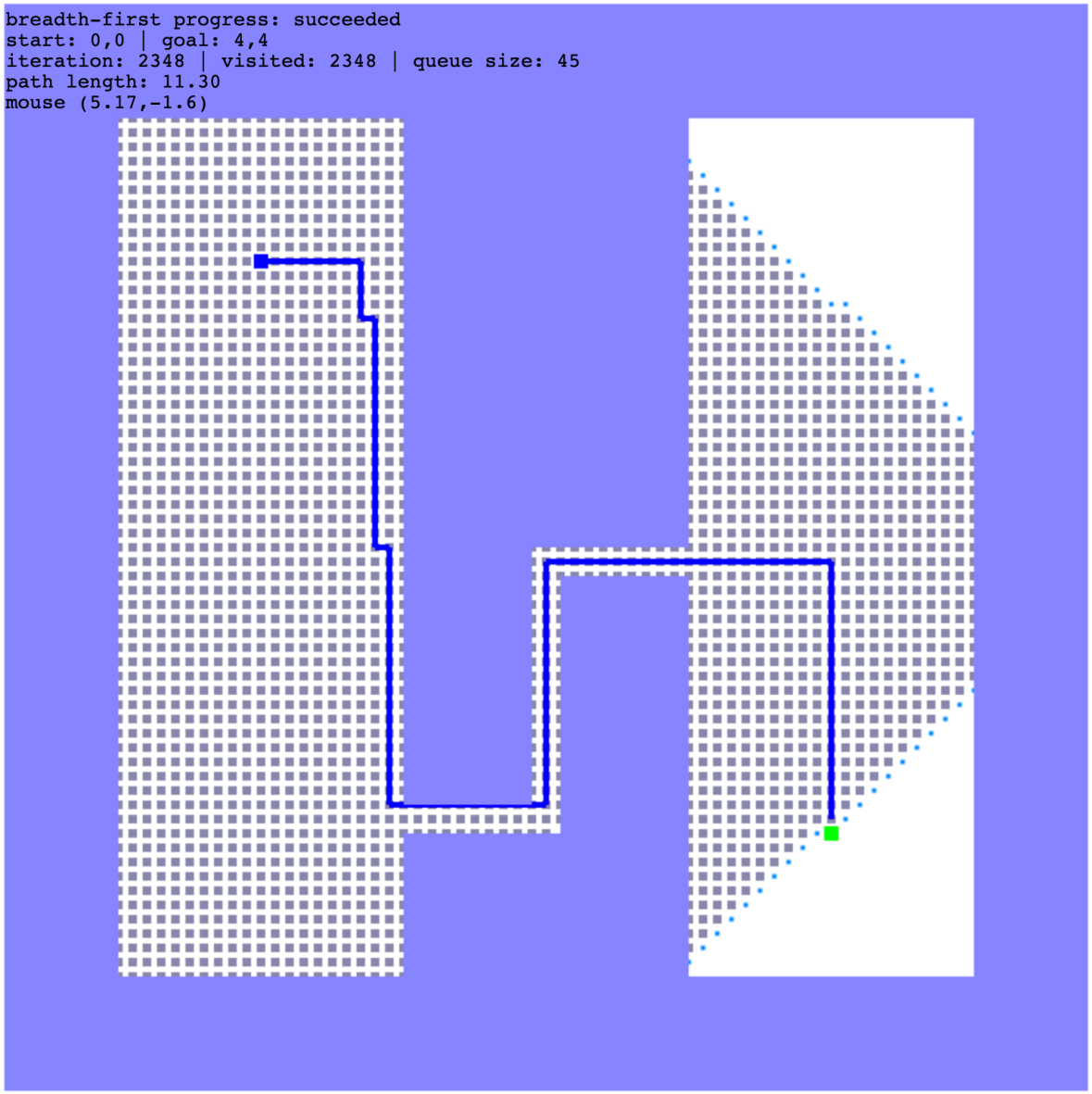
PRIORITY:
LEAST RECENT



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

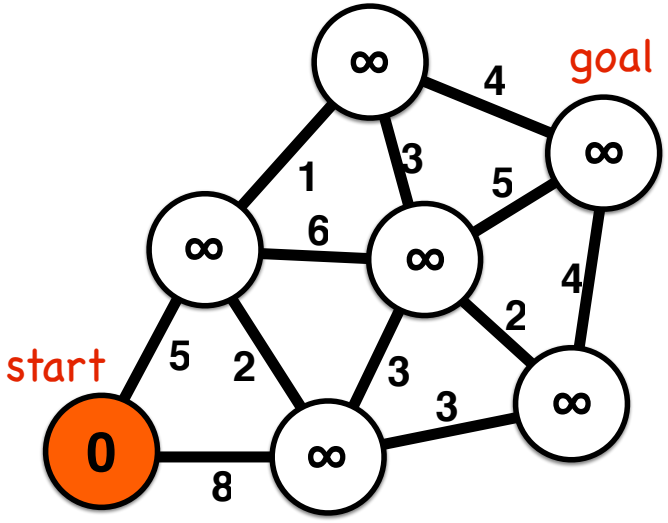



breadth-first progress: succeeded
start: 0,0 | goal: 4,4
iteration: 2348 | visited: 2348 | queue size: 45
path length: 11.30
mouse (5.17,-1.6)

Dijkstra shortest path

Search algorithm template

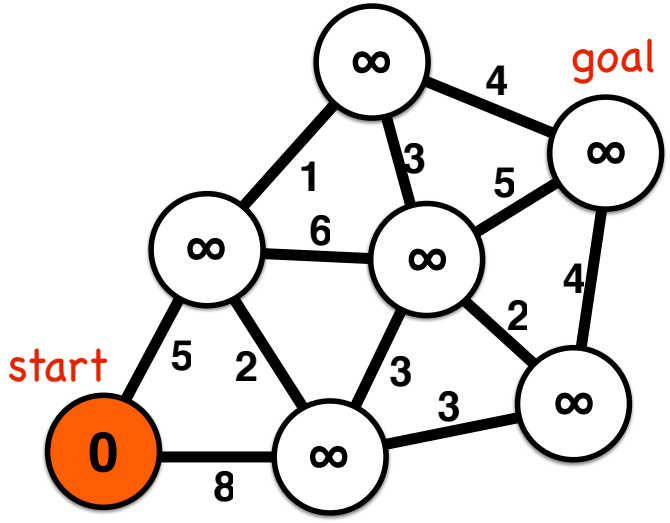
all nodes ← {$dist_{start}$ ← infinity, $parent_{start}$ ← none, $visited_{start}$ ← false}
start_node ← {$dist_{start}$ ← 0, $parent_{start}$ ← none, $visited_{start}$ ← true}
visit_list ← start_node
    while visit_list != empty && current_node != goal
        cur_node ← **highestPriority**(visit_list)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

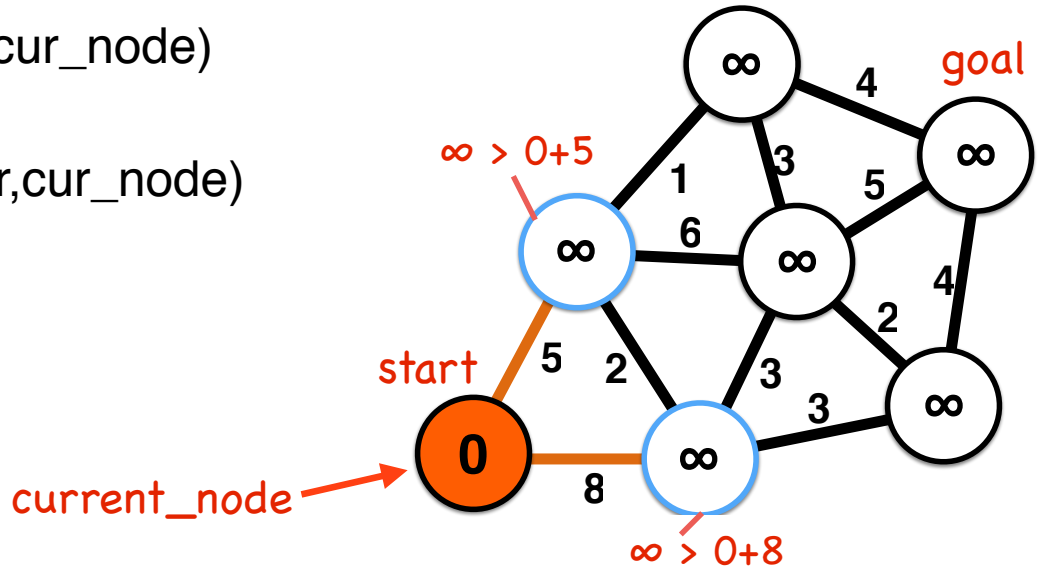
PRIORITY:
MINIMUM ROUTE DISTANCE
FROM START



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

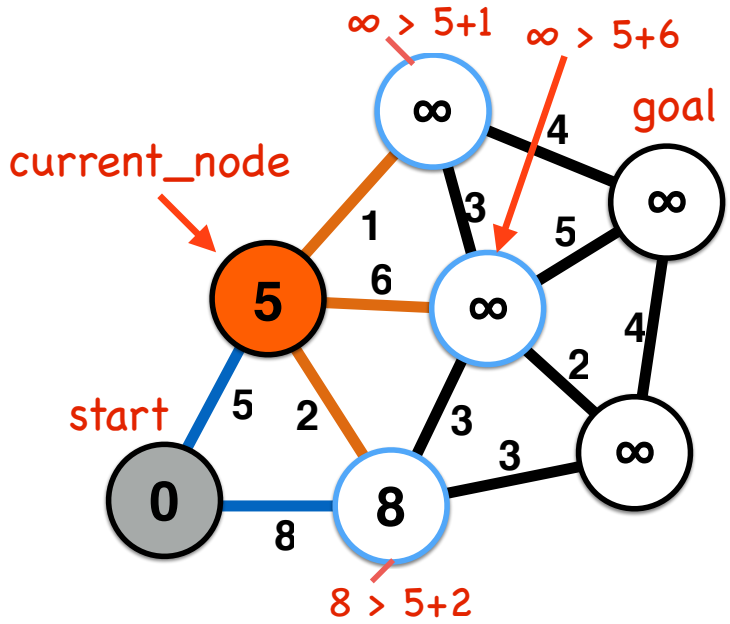
Dijkstra walkthrough



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

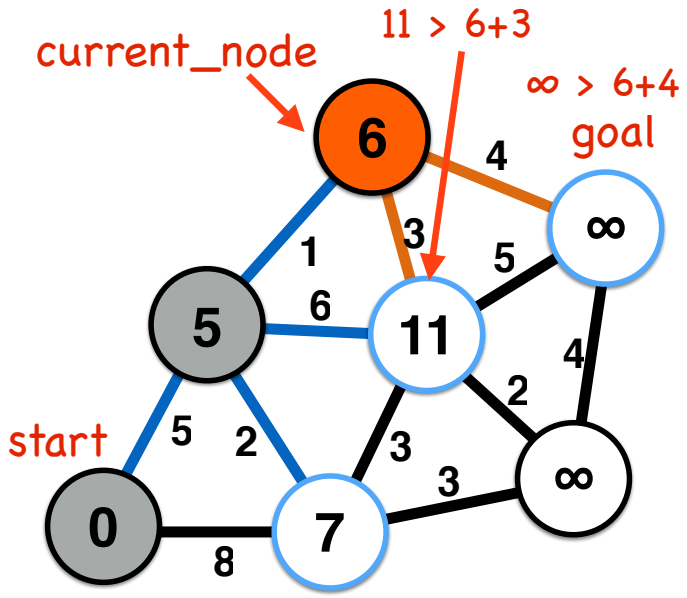
Dijkstra walkthrough



Dijkstra shortest path algorithm

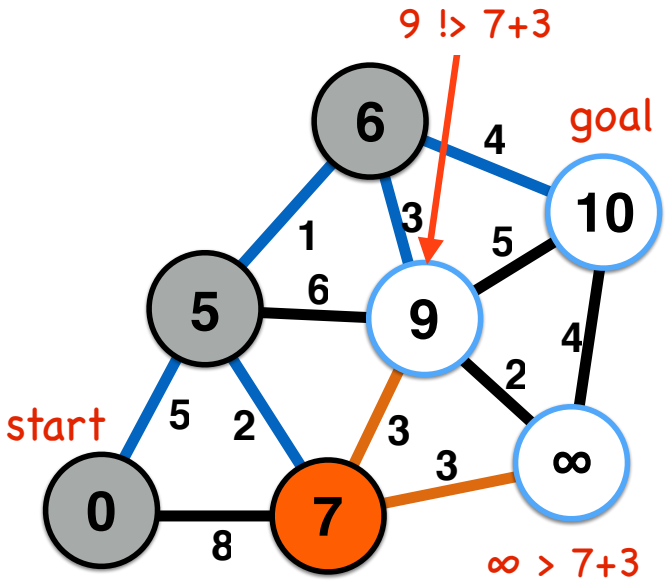
all nodes ← {$dist_{start}$ ← infinity, $parent_{start}$ ← none, $visited_{start}$ ← false}
start_node ← {$dist_{start}$ ← 0, $parent_{start}$ ← none, $visited_{start}$ ← true}
visit_queue ← start_node
    while visit_queue != empty ~~&& current_node != goal~~
        cur_node ← min_distance(visit_queue)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



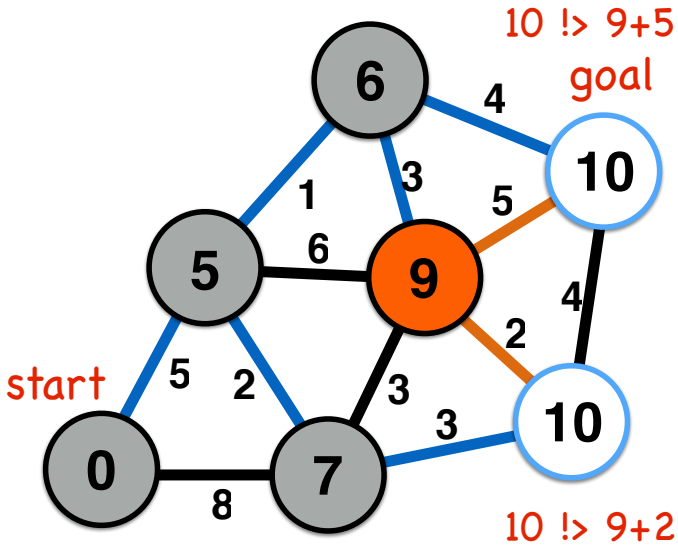
Dijkstra shortest path algorithm

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



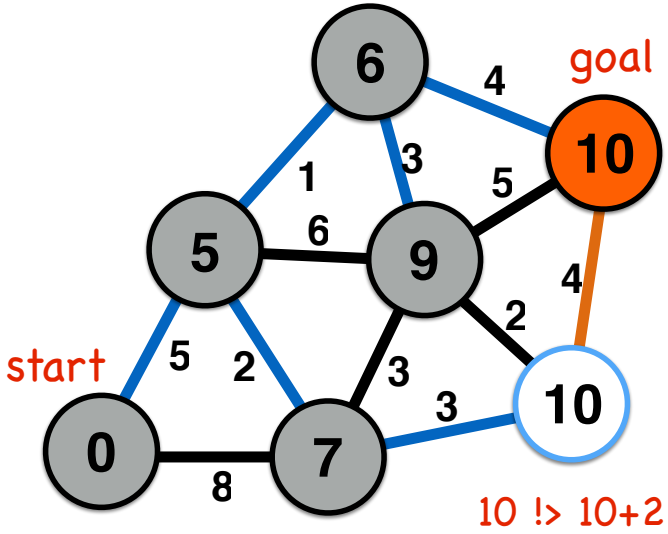
Dijkstra shortest path algorithm

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



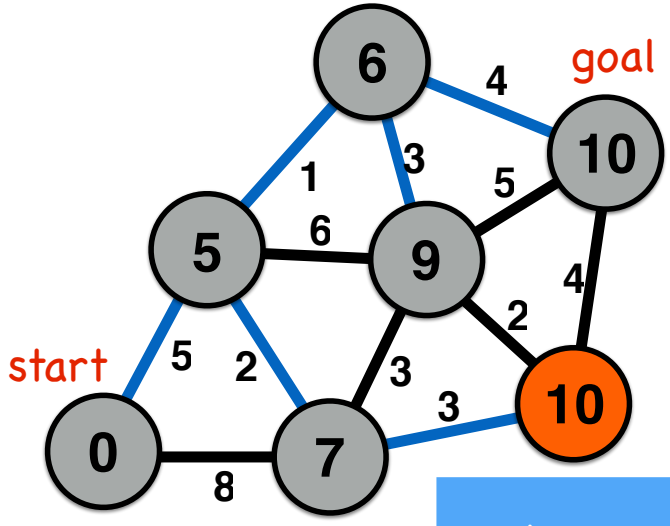
Dijkstra shortest path algorithm

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



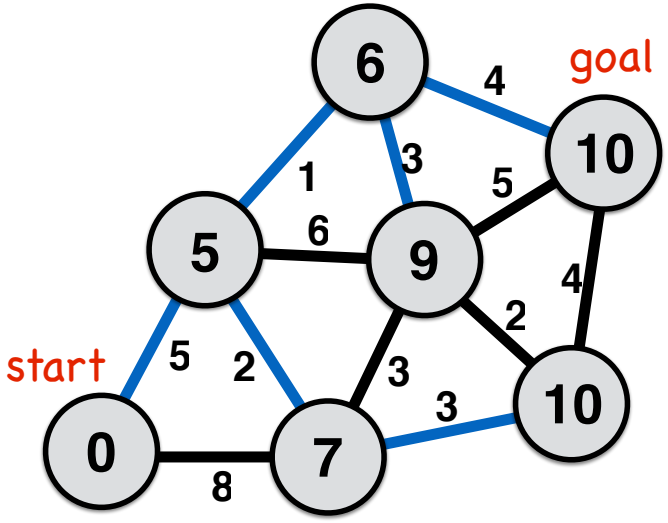
Dijkstra shortest path algorithm

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



Dijkstra shortest path algorithm

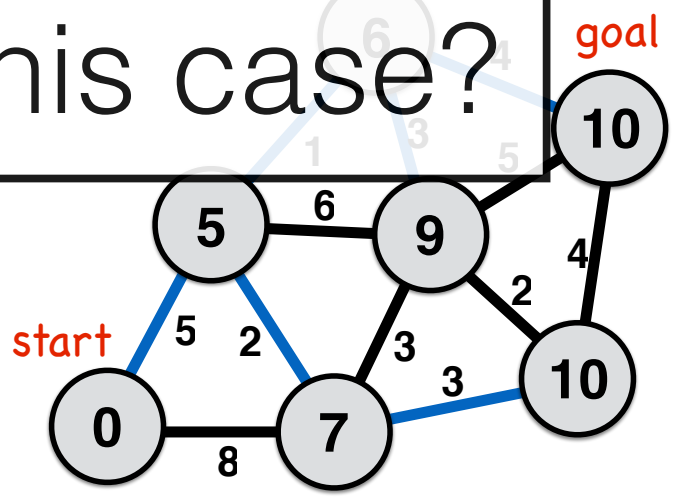
all nodes ← {$dist_{start}$ ← infinity, $parent_{start}$ ← none, $visited_{start}$ ← false}
start_node ← {$dist_{start}$ ← 0, $parent_{start}$ ← none, $visited_{start}$ ← true}
visit_queue ← start_node
    while visit_queue != empty ~~&& current_node != goal~~
        cur_node ← min_distance(visit_queue)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



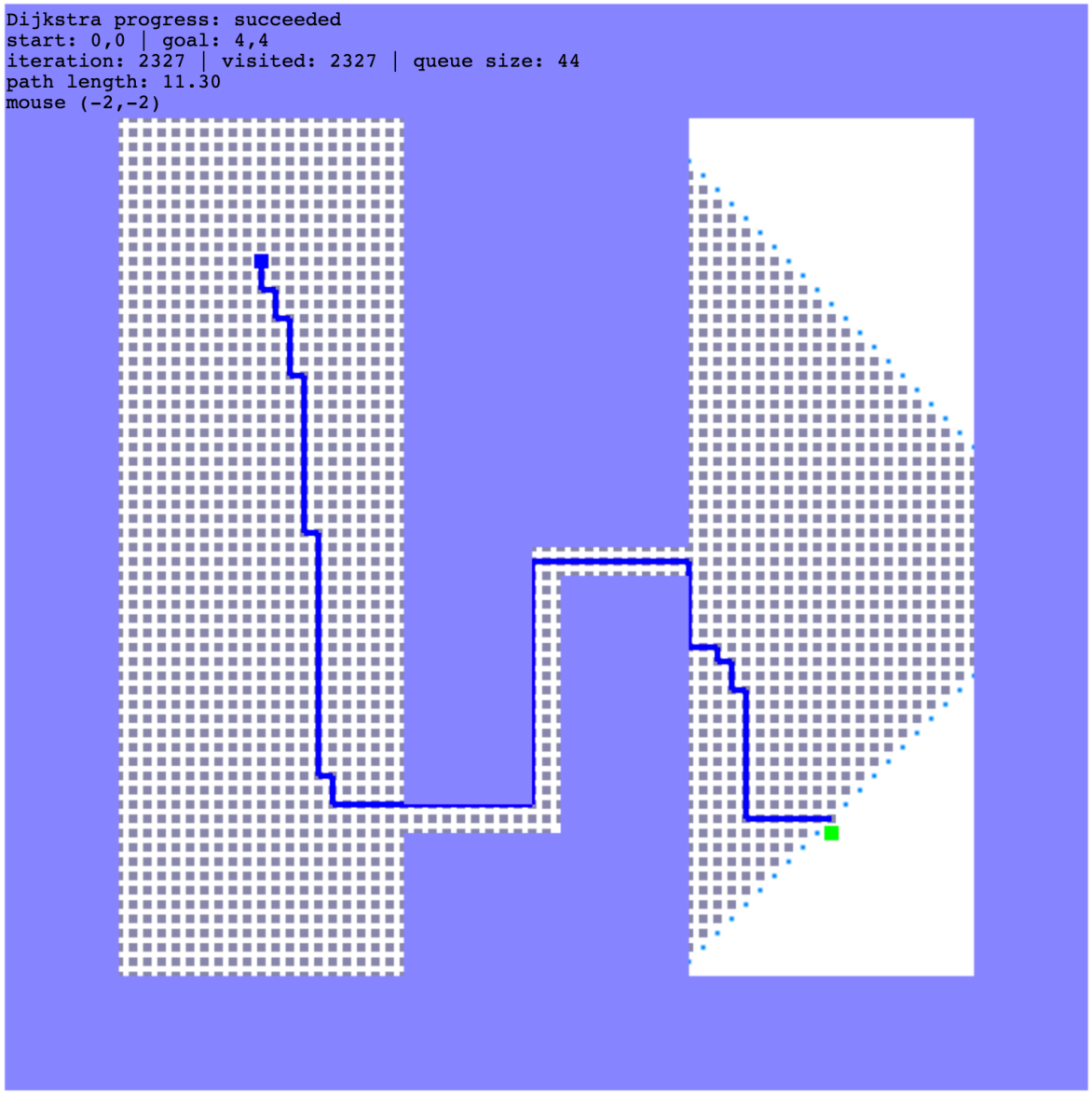
Dijkstra shortest path algorithm

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, 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

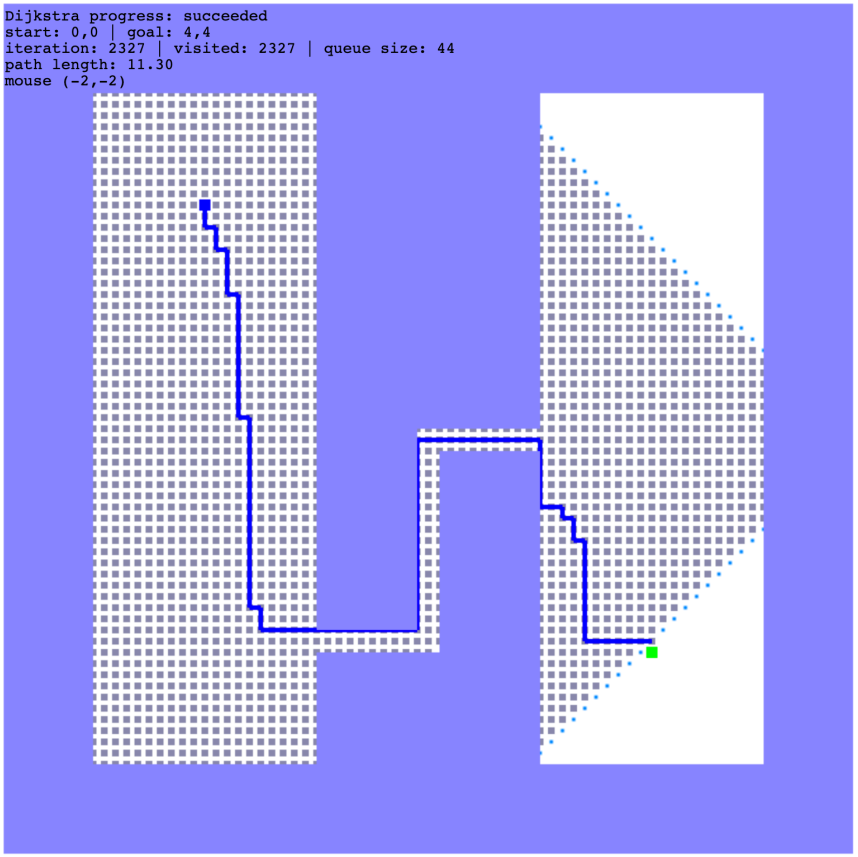
What will search with Dijkstra's algorithm look like in this case?



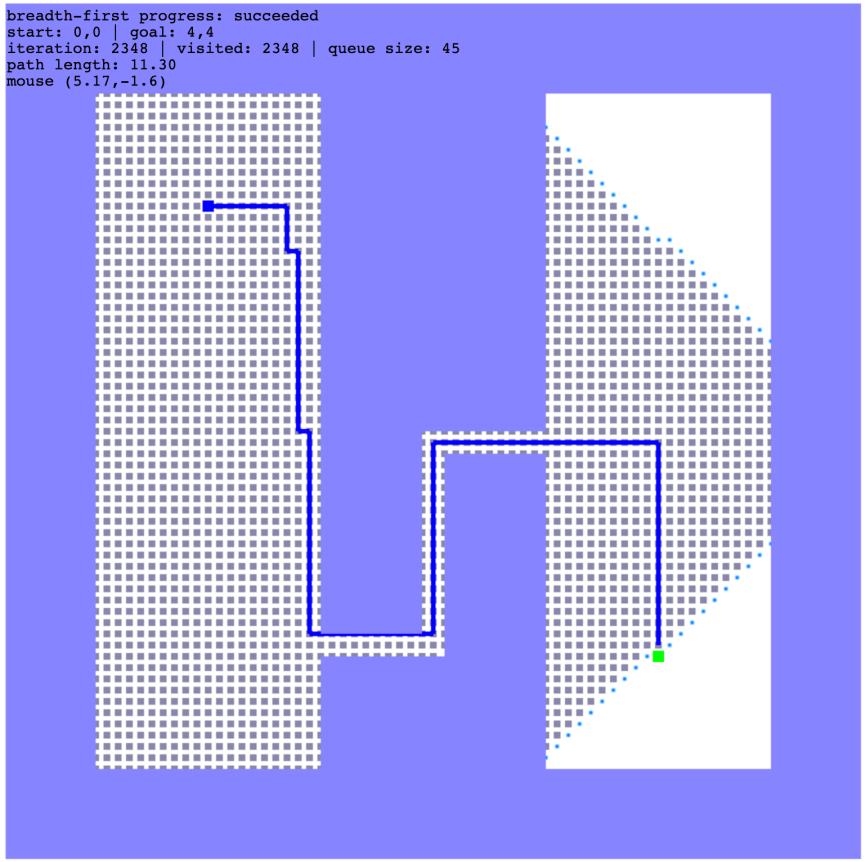
What will search with Dijkstra's algorithm look like in this case?


Dijkstra progress: succeeded
start: 0,0 | goal: 4,4
iteration: 2327 | visited: 2327 | queue size: 44
path length: 11.30
mouse (-2,-2)

Dijkstra



BFS

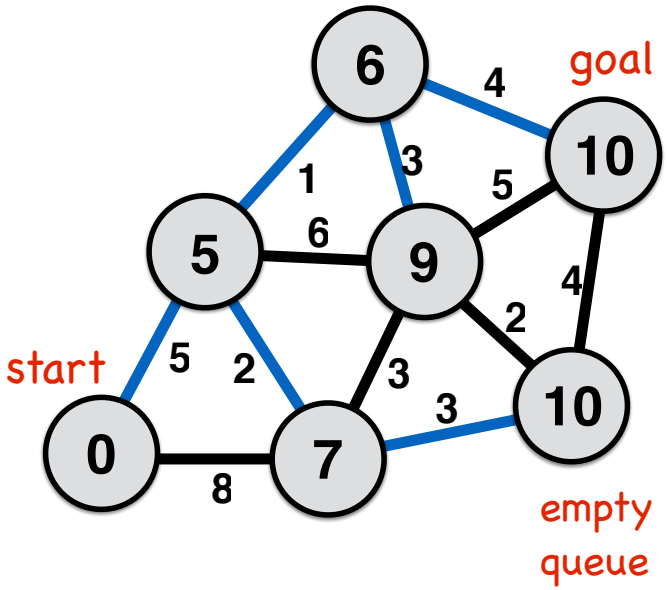


Why does their visit pattern look similar?

A-star shortest path

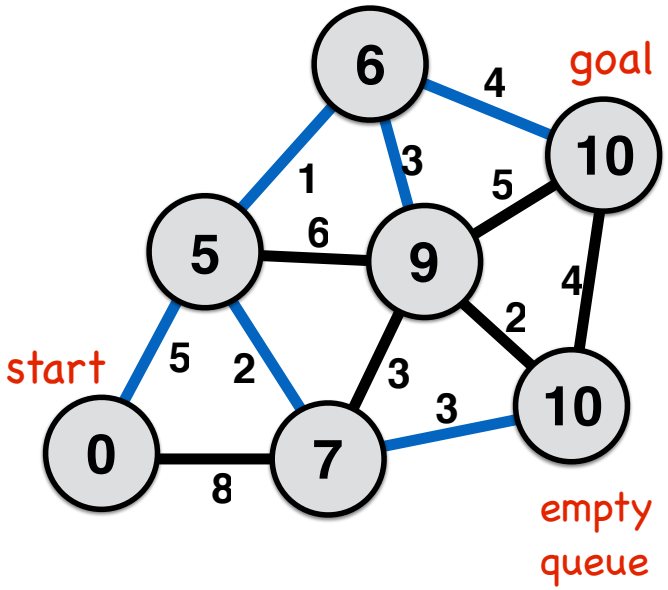
Dijkstra shortest path algorithm

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



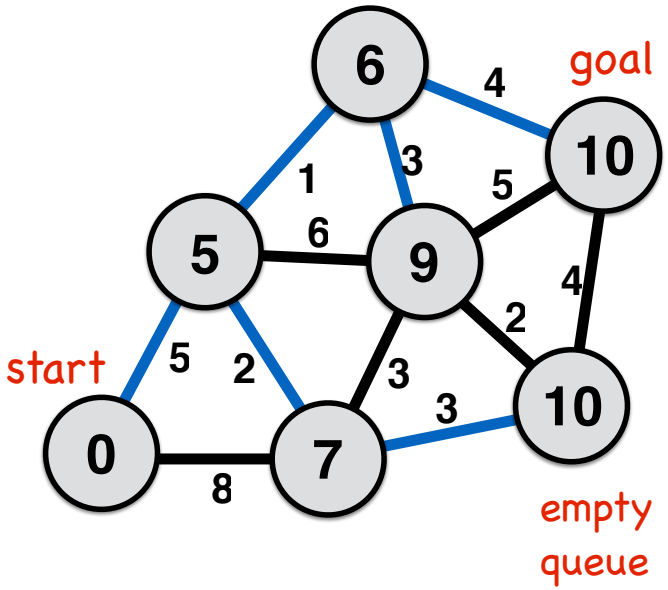
A-star shortest path algorithm

```
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)

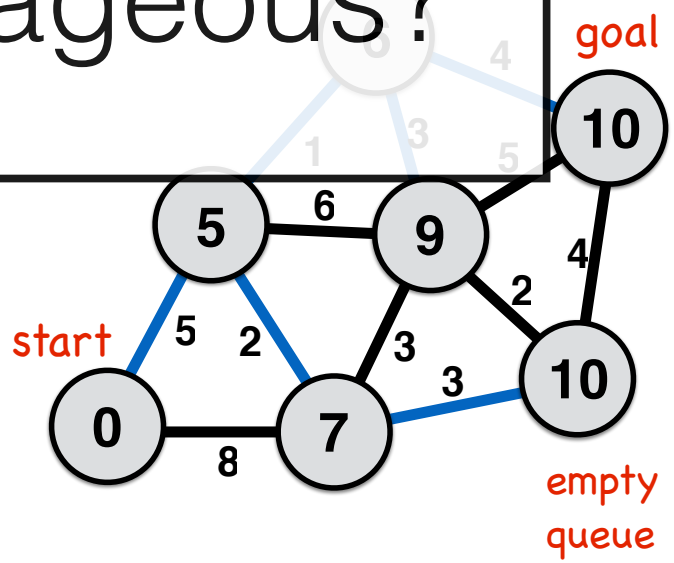
f_score ← $distance_{nbr}$ + $line\_distance_{nbr,goal}$

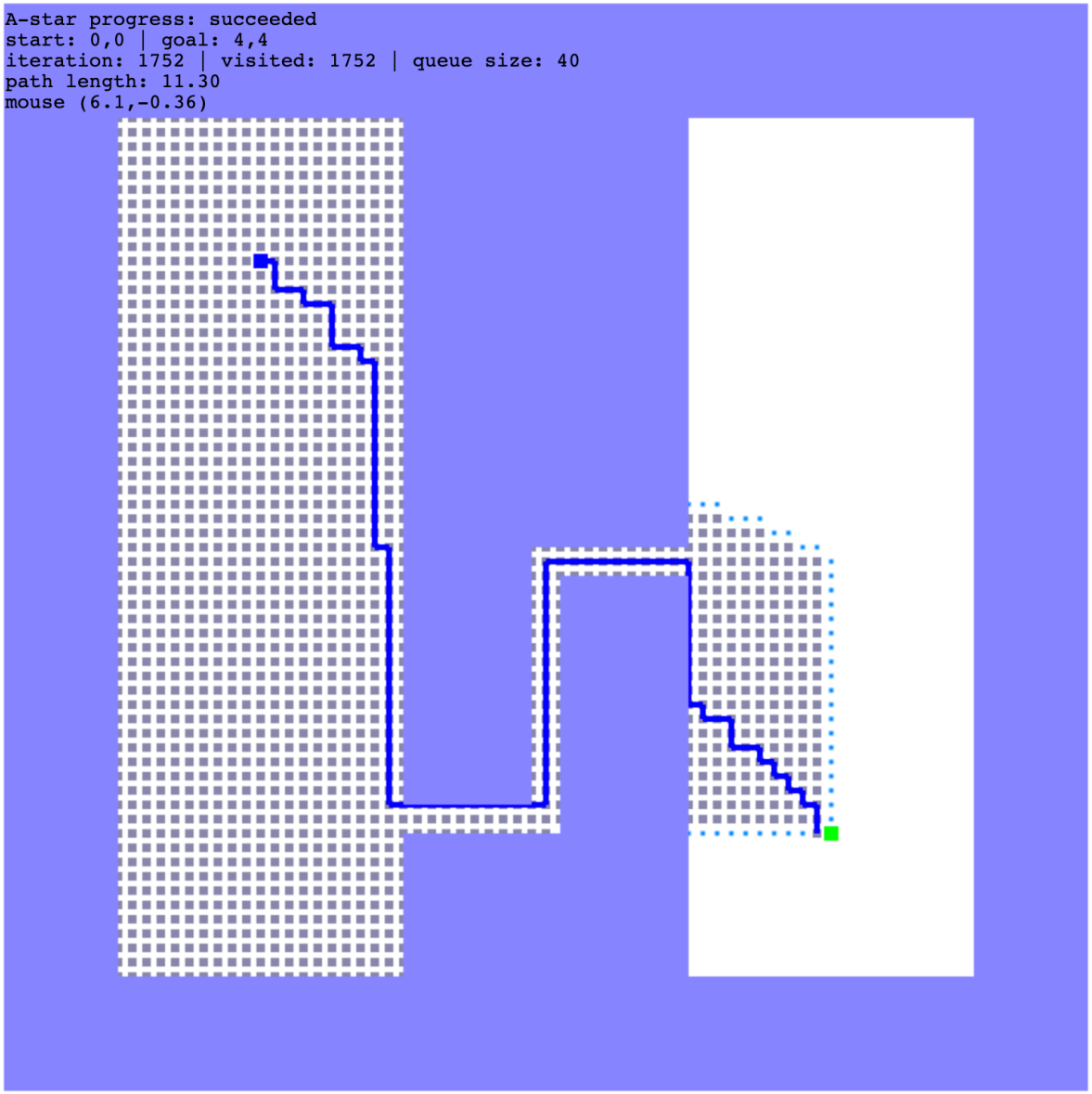
end if
end for loop
end while loop
output ← parent, distance

g_score: distance along current path back to start

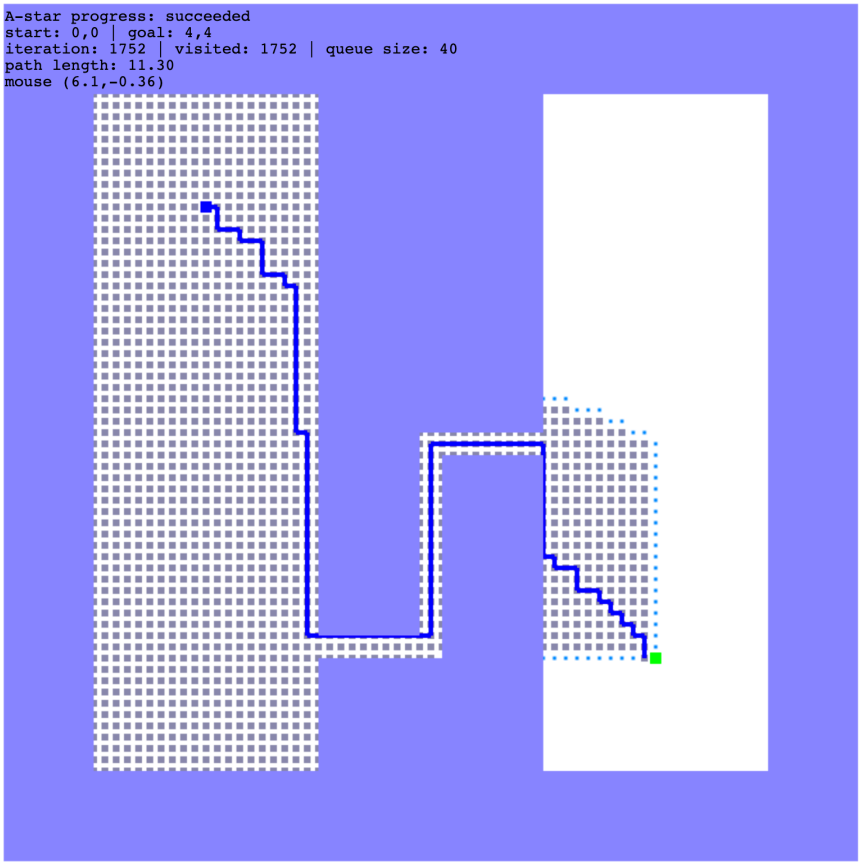
h_score: best possible distance to goal

Why is A-star advantageous?

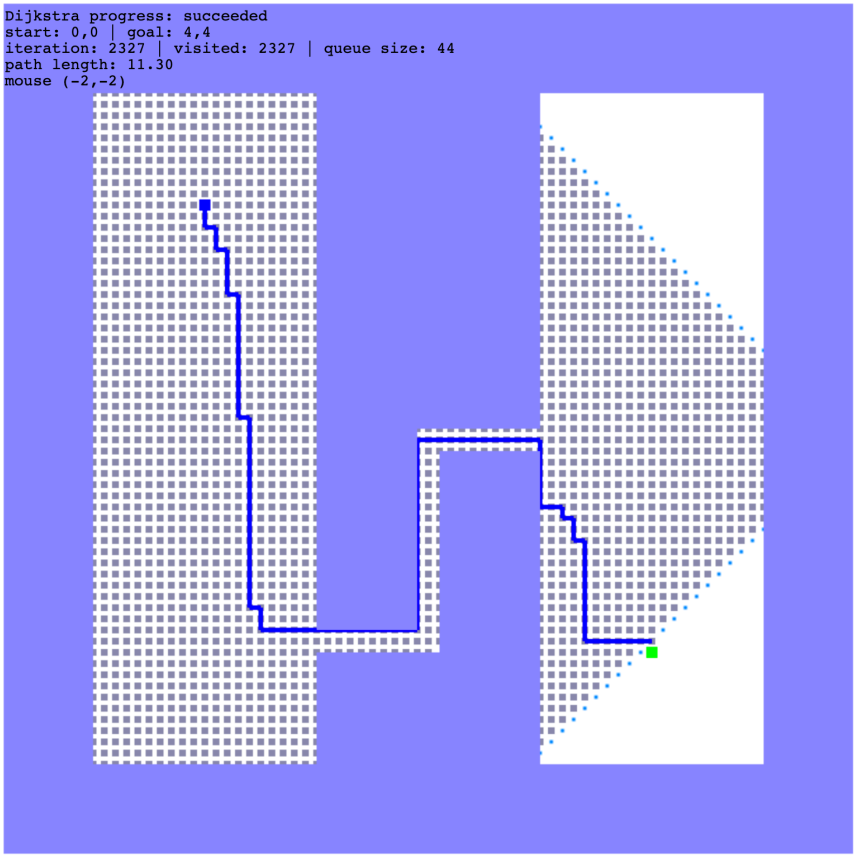



A-star progress: succeeded
start: 0,0 | goal: 4,4
iteration: 1752 | visited: 1752 | queue size: 40
path length: 11.30
mouse (6.1,-0.36)

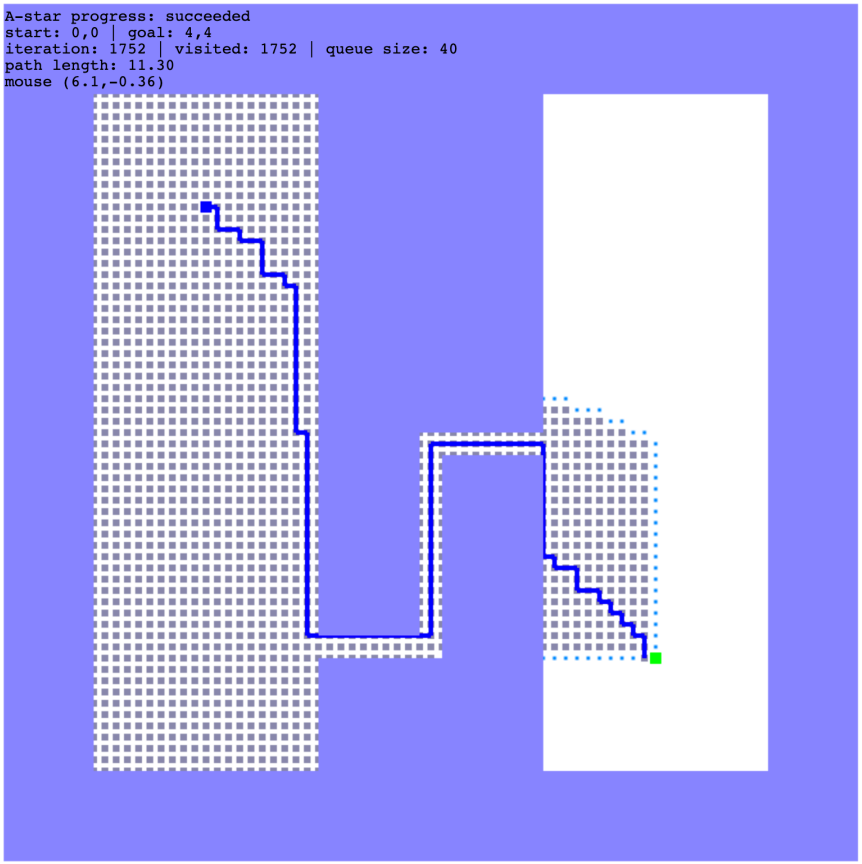
A-Star



Dijkstra



How can A-star visit few nodes?



How can A-star visit few nodes?

A-Star uses an admissible heuristic to estimate the cost to goal from a node

The straight line h_score is an admissible and consistent heuristic function.

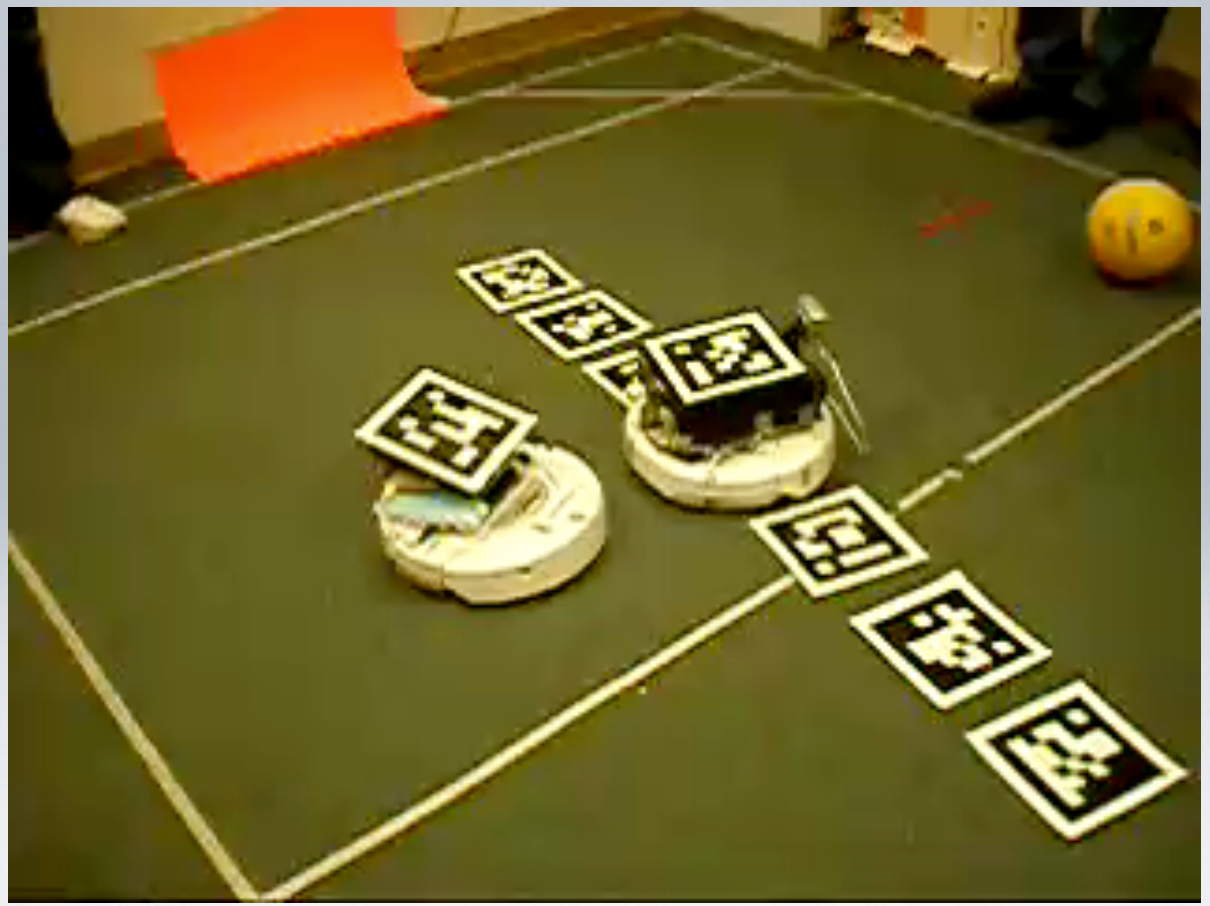
A heuristic function is **admissible** if it never overestimates the cost of reaching the goal.

Thus, h_score(x) is less than or equal to the lowest possible cost from current location to the goal.

A heuristic function is **consistent** if obeys the triangle inequality

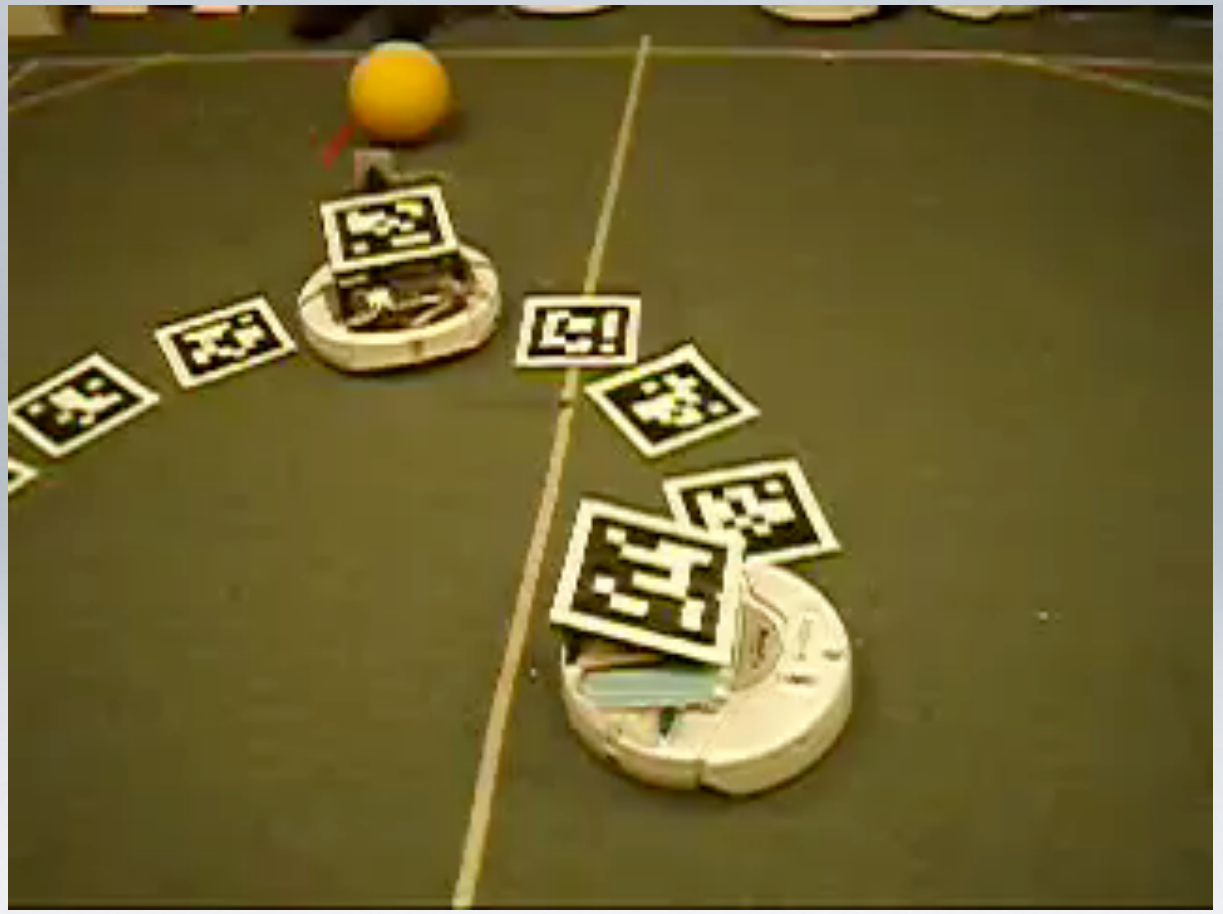
Thus, h_score(x) is less than or equal to cost(x,action,x') + h_score(x')

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

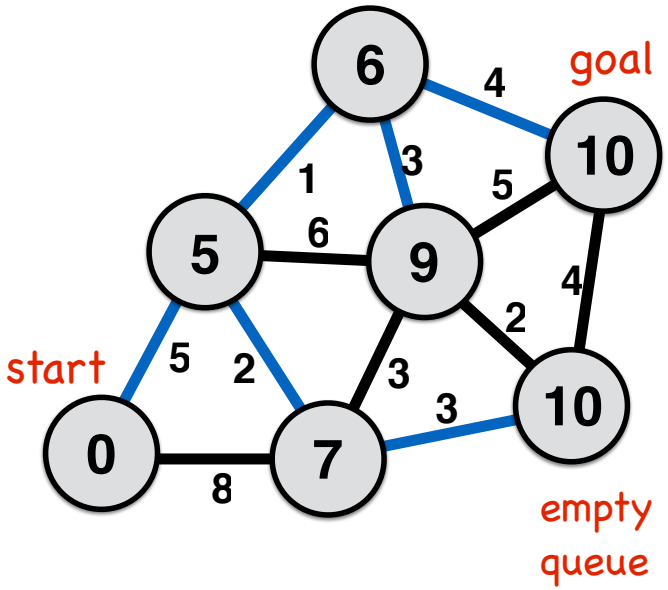
A-star shortest path algorithm

all nodes ← {$dist_{start}$ ← 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

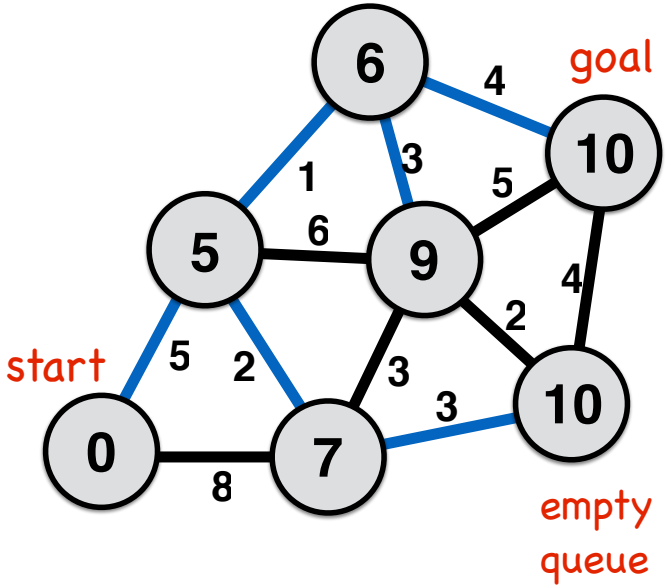


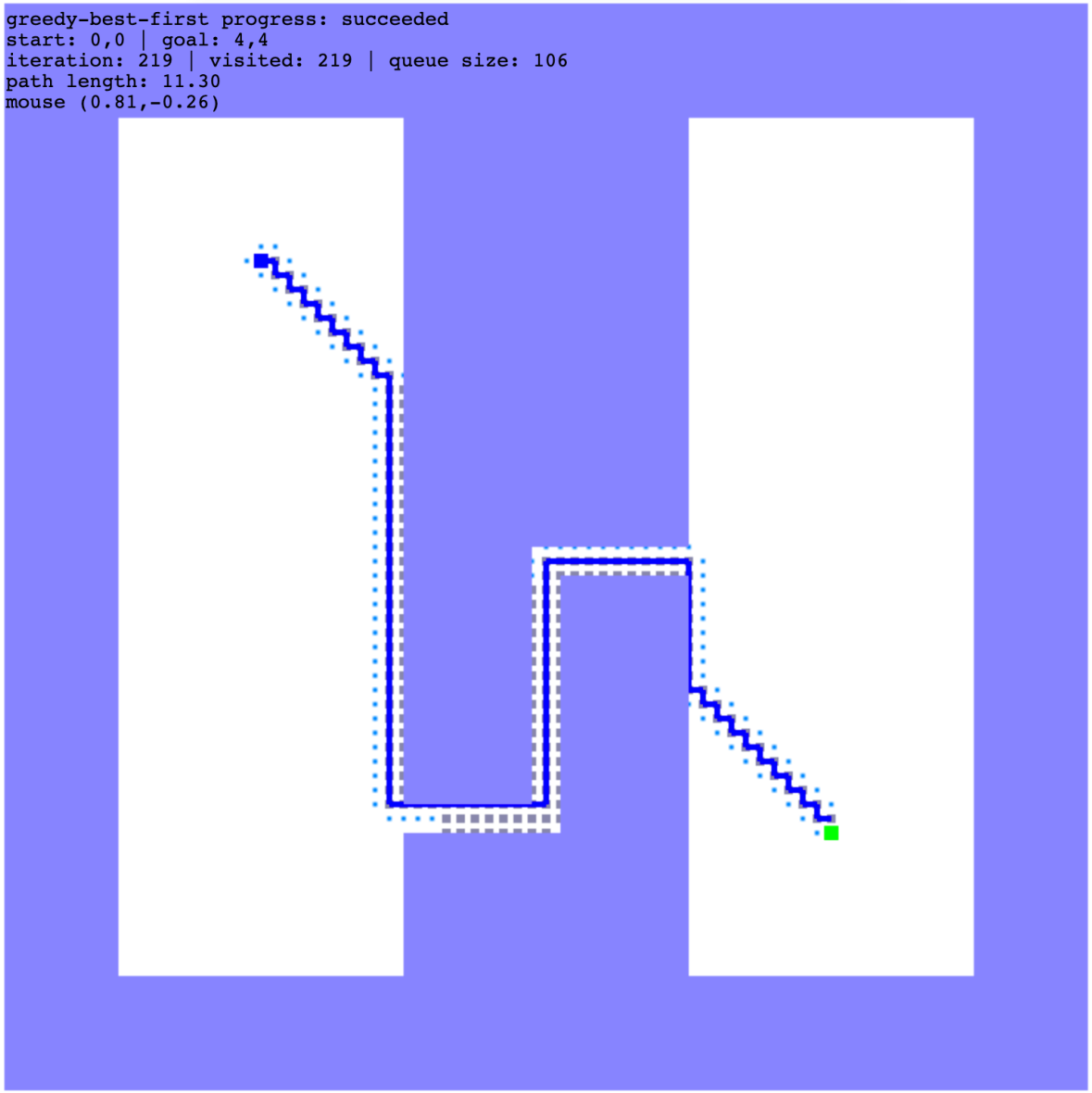
Greedy best-first search

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

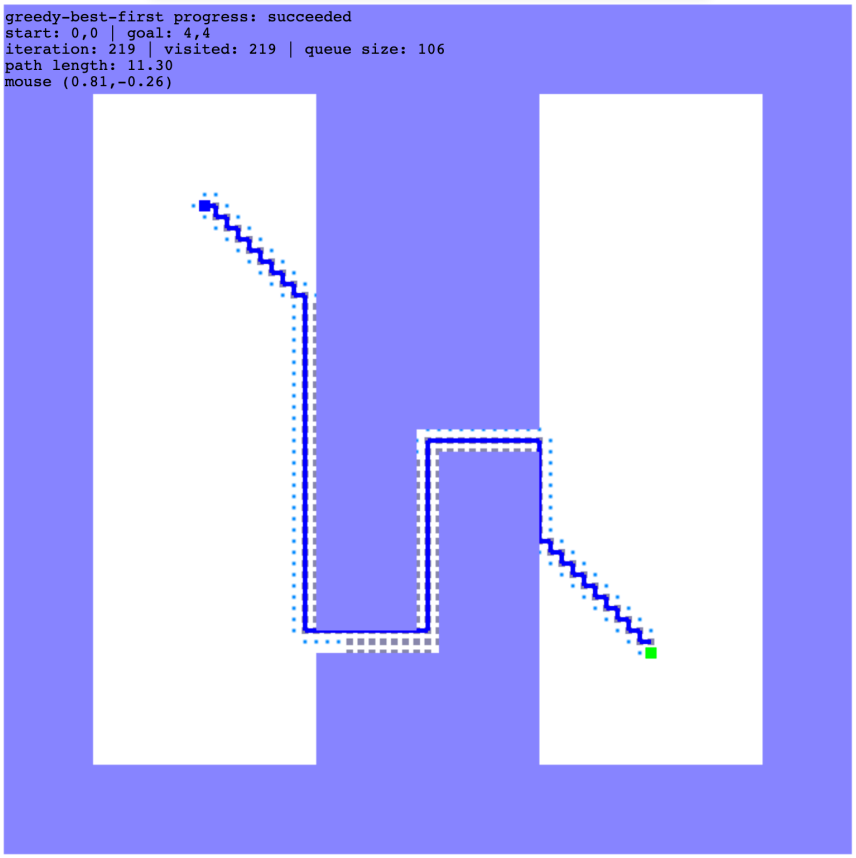
priority queue wrt. f_score
(implement min binary heap)

h_score:
best possible
distance to goal

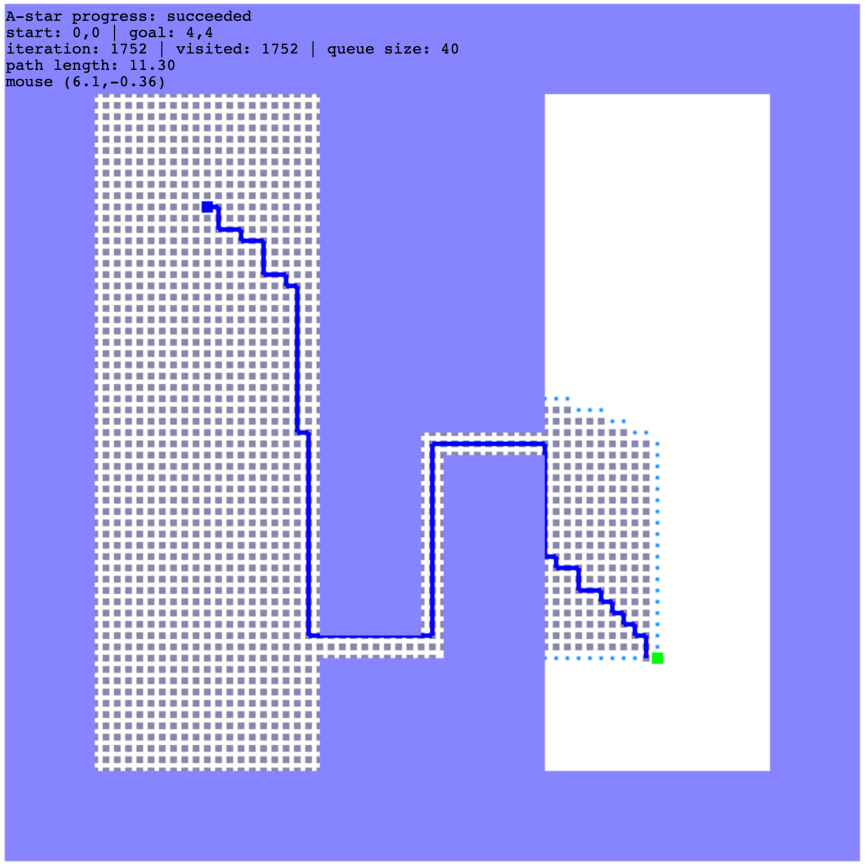



greedy-best-first progress: succeeded
start: 0,0 | goal: 4,4
iteration: 219 | visited: 219 | queue size: 106
path length: 11.30
mouse (0.81,-0.26)

Greedy Best First

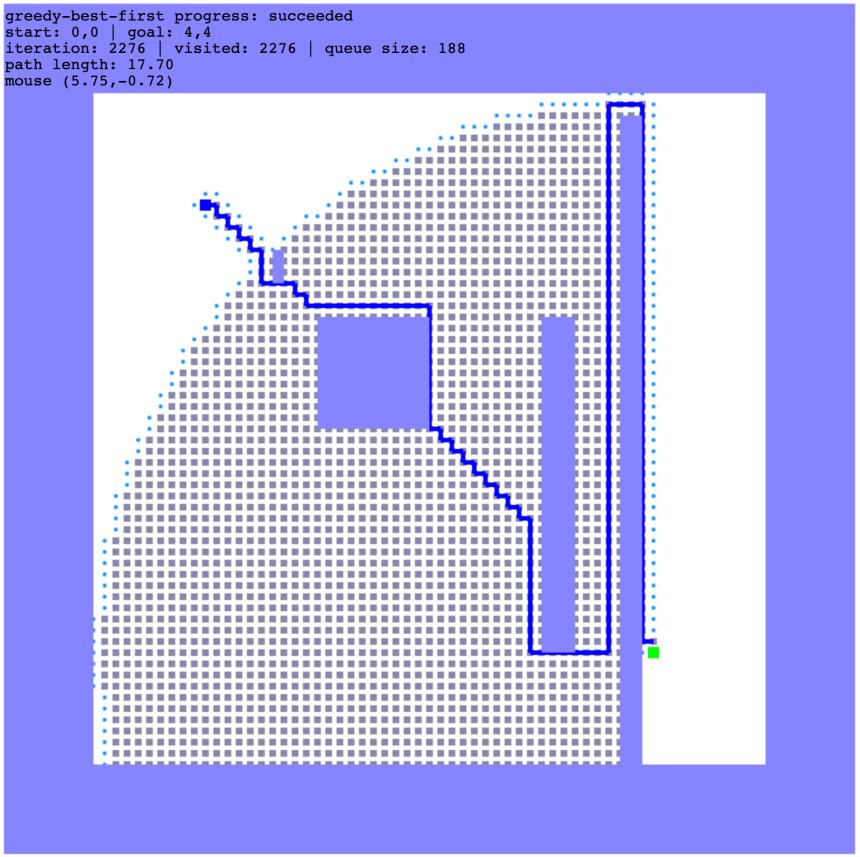


A-Star

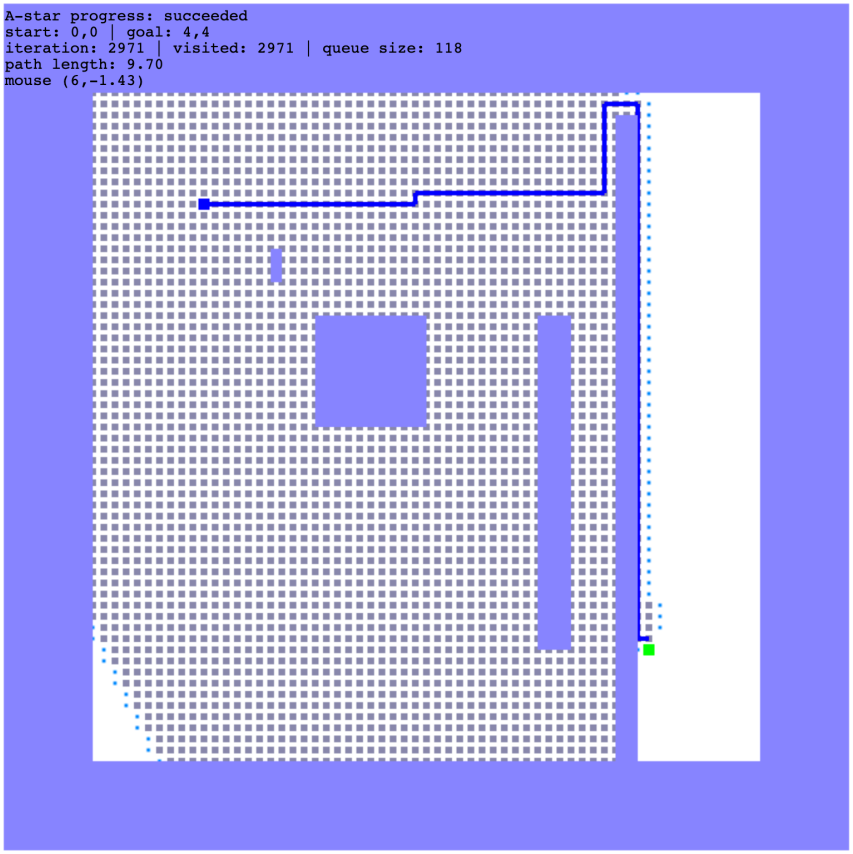


Wow! Best first did great…

Greedy Best First



A-Star



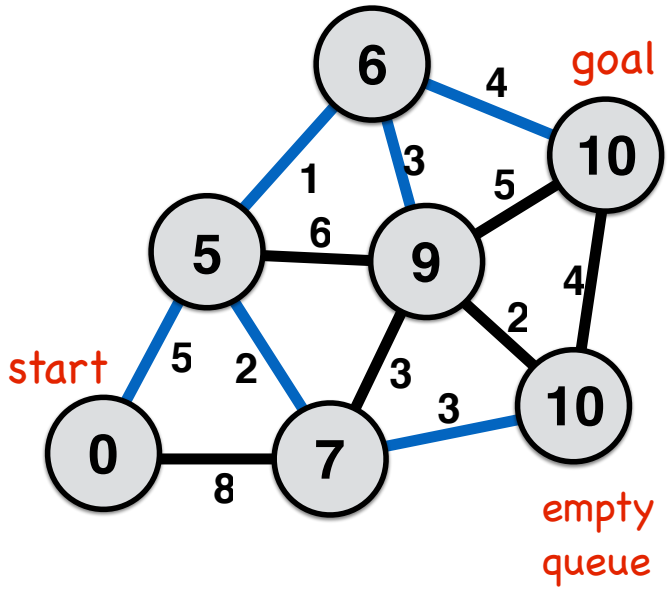
But, what happens if we try a different scene? Maybe not so great.

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

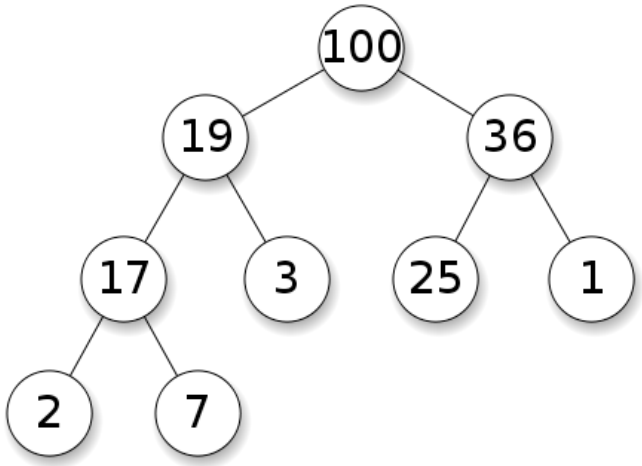
MIN BINARY HEAP FOR PRIORITY QUEUE



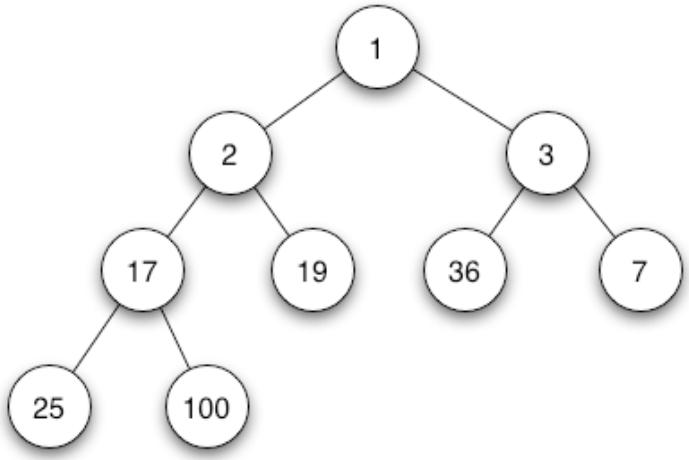
Binary Heaps

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

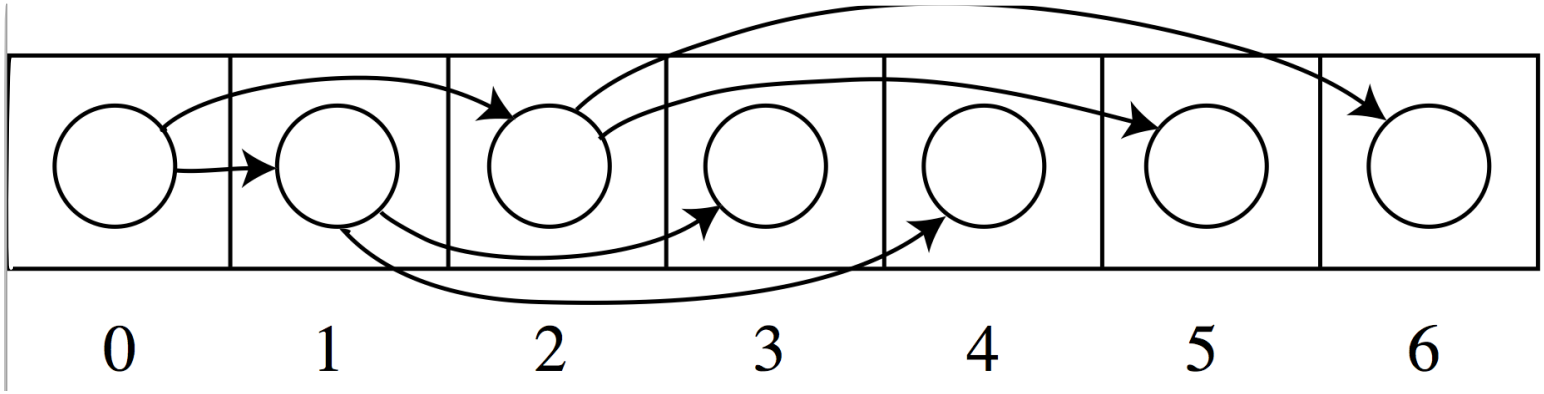


max heap



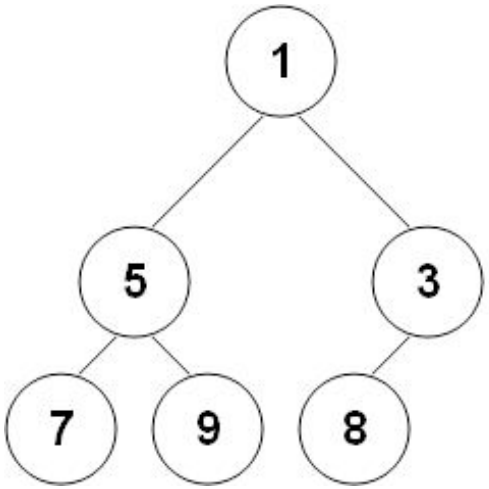
min heap

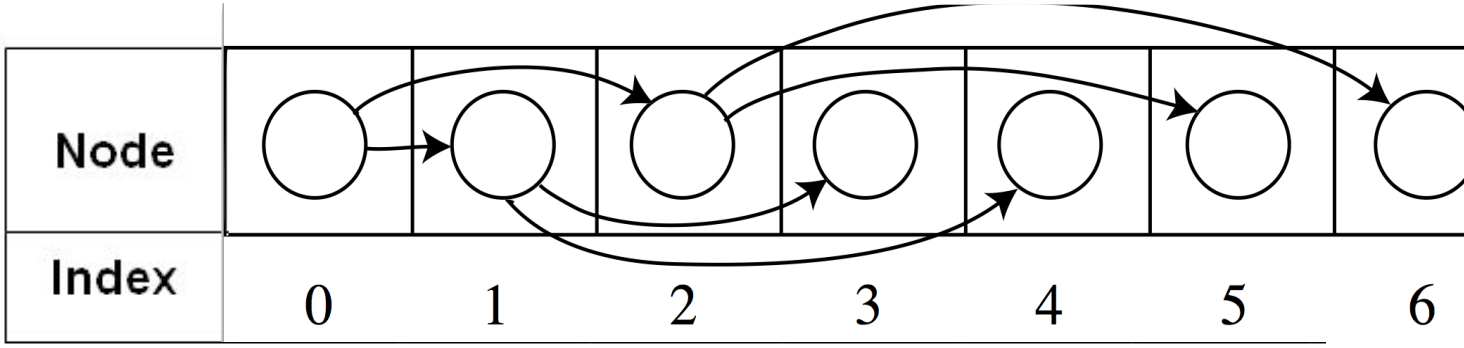
Heaps as arrays



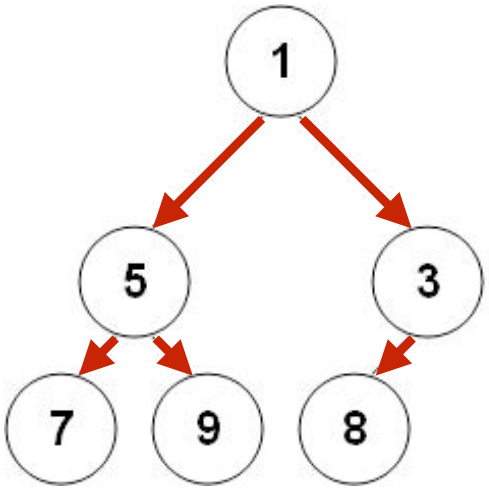
- Heap element at array location *i* has
  - children at array locations *2i+1* and *2i+2*
  - parent at *floor((i-1)/2)*

Heap array example





Heap array example



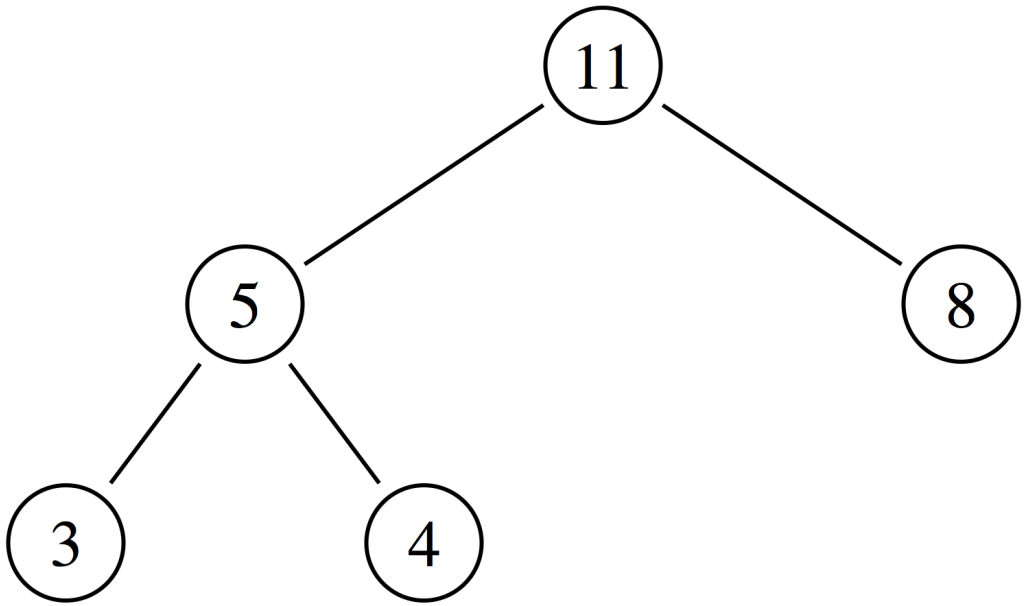
Node	1	5	3	7	9	8
Index	0	1	2	3	4	5

How do we insert a new heap element?

Heap operations: Insert

New element

Current heap

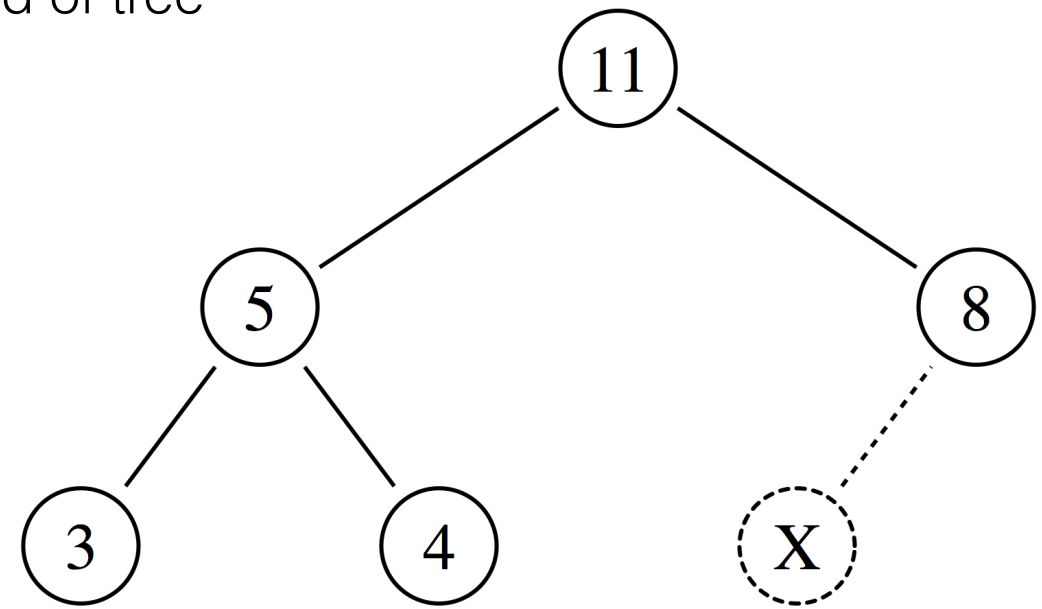


Heap operations: Insert

Step 1) add new element to end of tree

New element

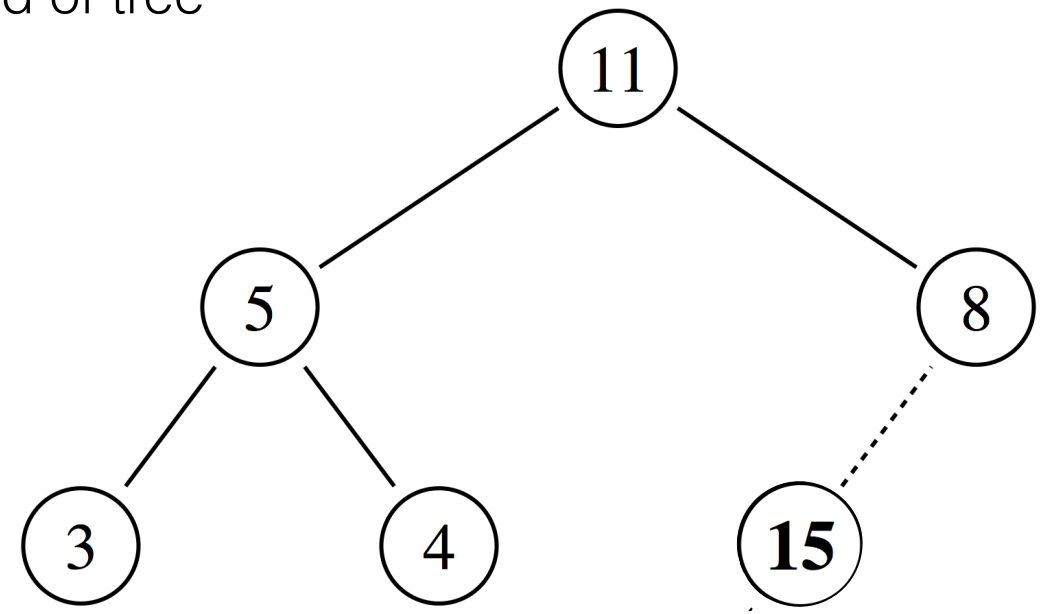
Current heap



Heap operations: Insert

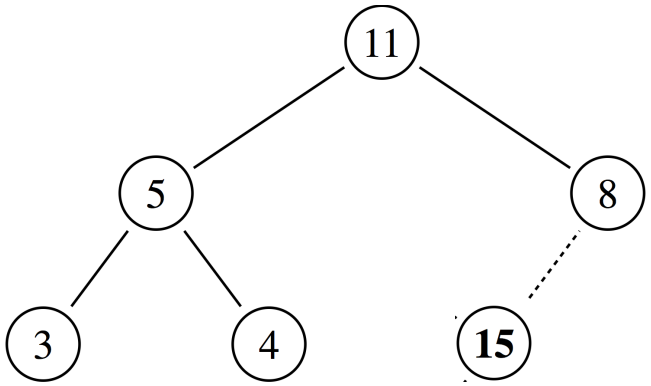
Step 1) add new element to end of tree

Current heap

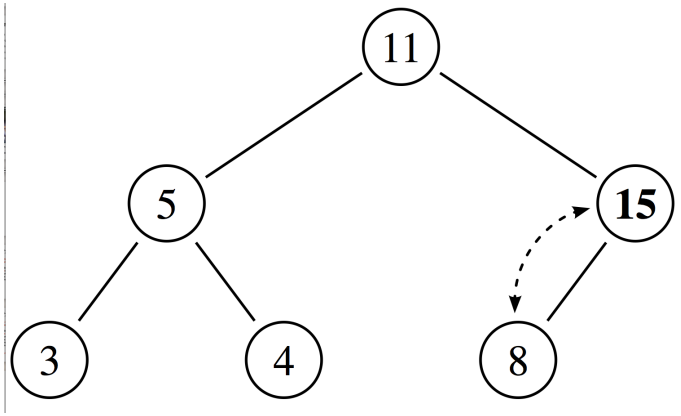


Heap operations: Insert

1) add new element to end of tree

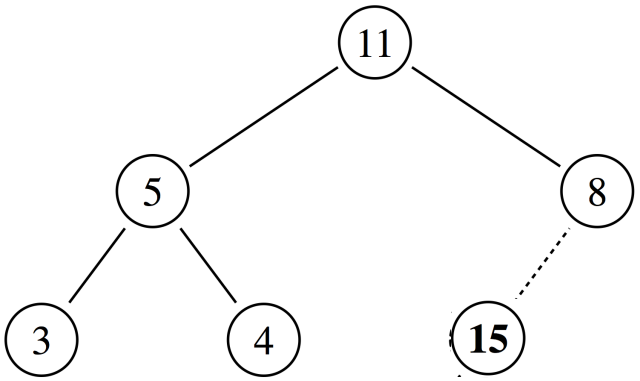


2) if heap condition not satisfied, swap inserted node with parent

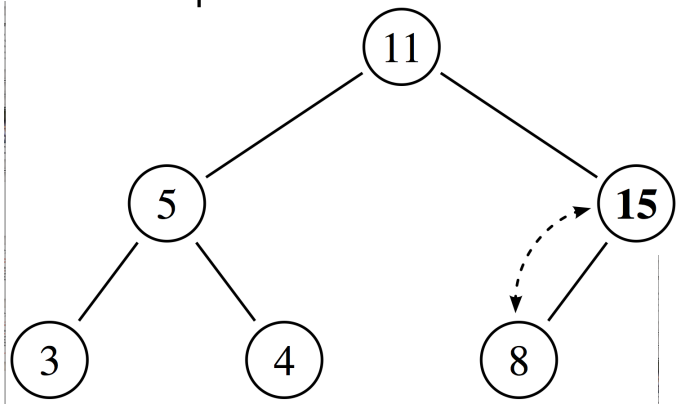


Heap operations: Insert

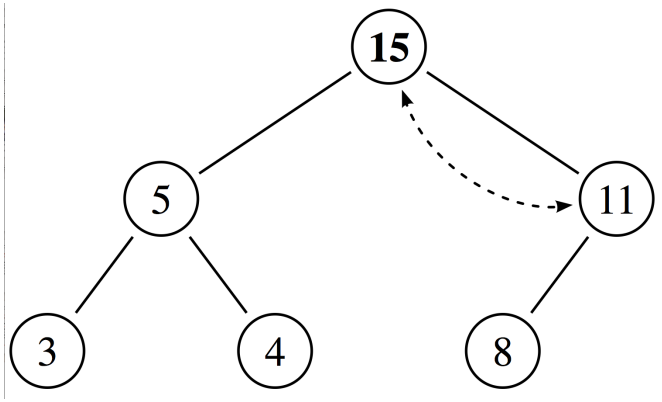
1) add new element to end of tree



2) if heap condition not satisfied, swap inserted node with parent



3) until heap condition satisfied, repeat (2)

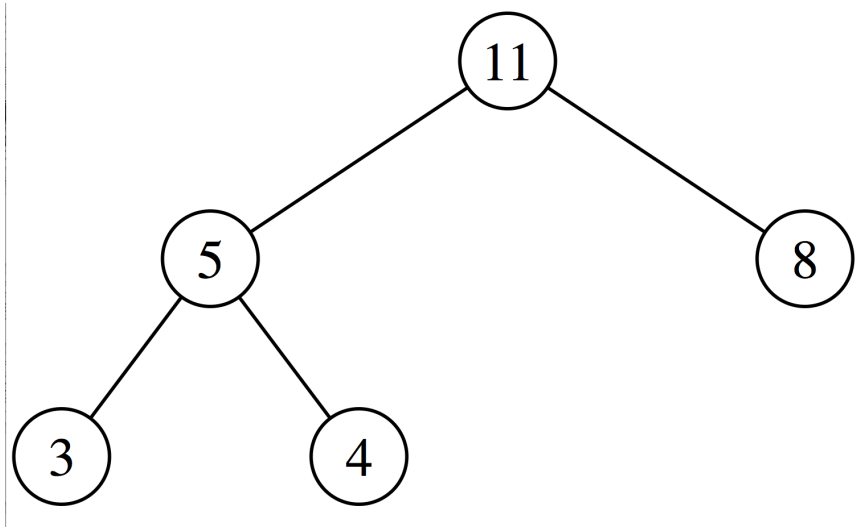


For priority queue, previously non-queued locations will be inserted with f_score priority

What happens when we extract a heap element?

Heap operations: Extract

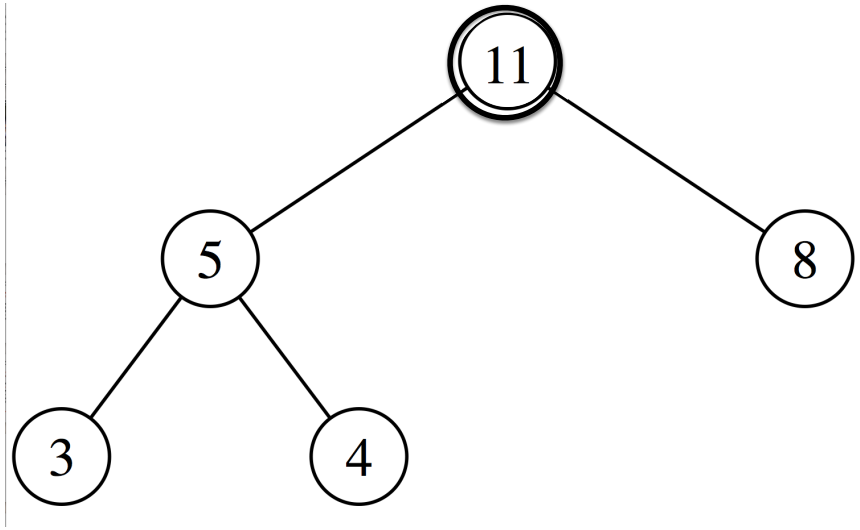
1) extract root element



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

Heap operations: Extract

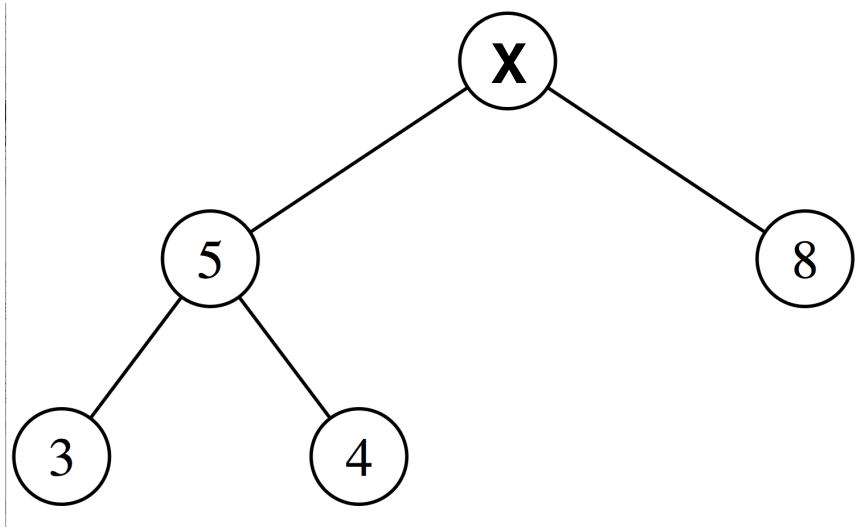
1) extract root element



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

Heap operations: Extract

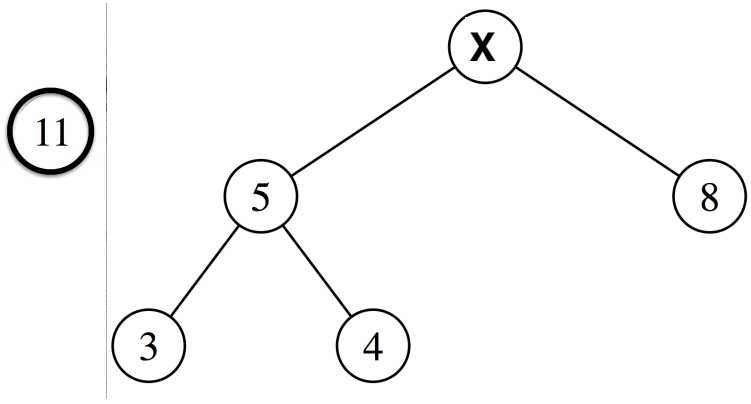
1) extract root element



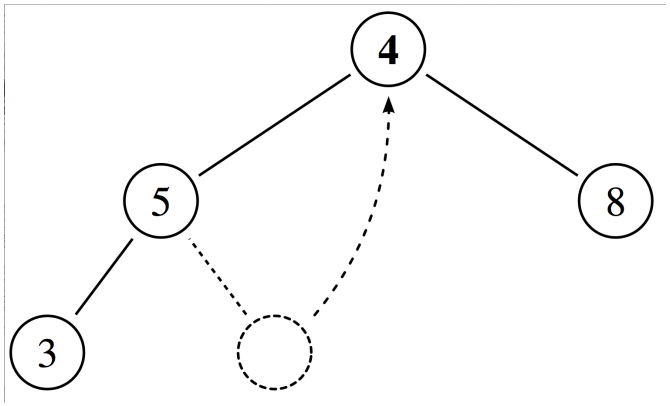
For priority queue, the root of the heap will be the next node to visit

Heap operations: Extract

1) extract root element

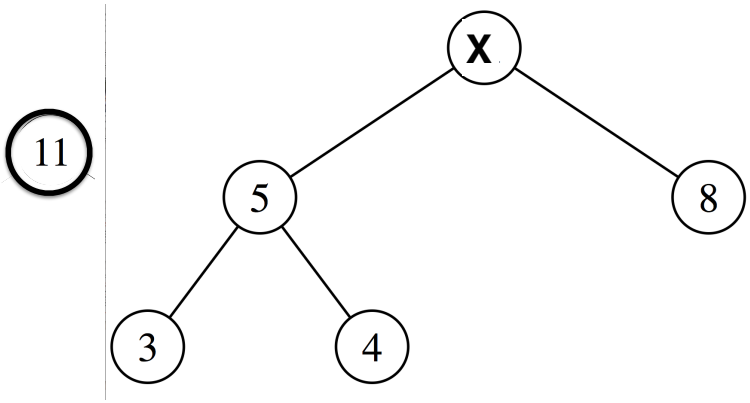


2) put last element at root

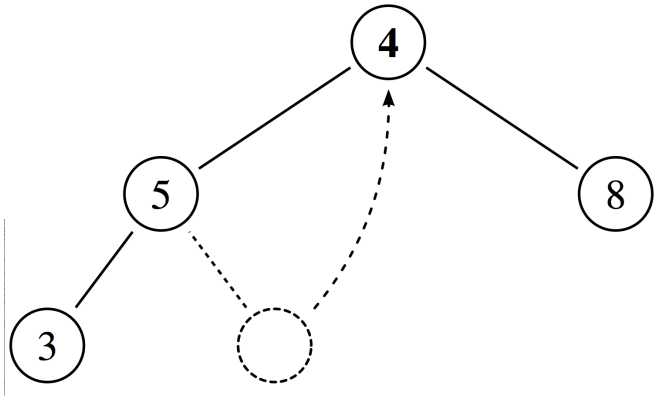


Heap operations: Extract

1) extract root element

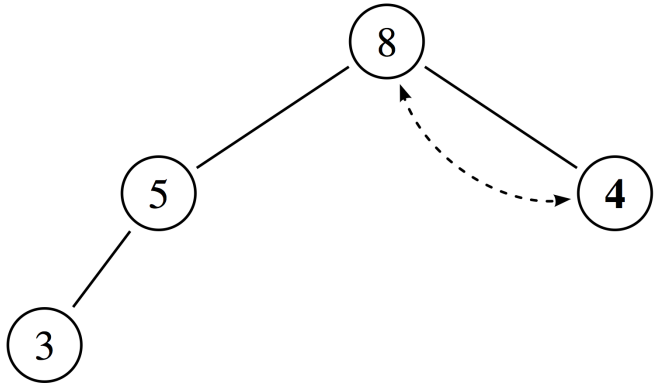


2) put last element at root



3) swap with higher priority child

4) until heaped, do (3)

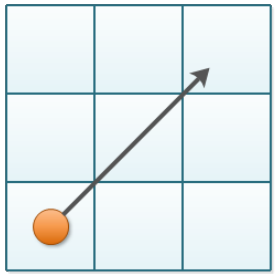


For priority queue, the root of the heap will be the next node to visit

Considerations

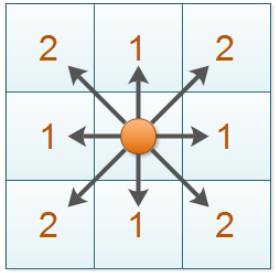
- How many operations are needed for heap insertion and extraction?
- How much better is a min heap than an array wrt. # of operations?
- Can there be duplicate heap elements for the same robot pose?
- How should we measure distance on a uniform grid?
- Is a choice of distance measure both metric and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

Euclidean Distance



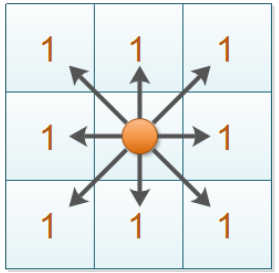
$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Manhattan Distance



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

Chebyshev Distance

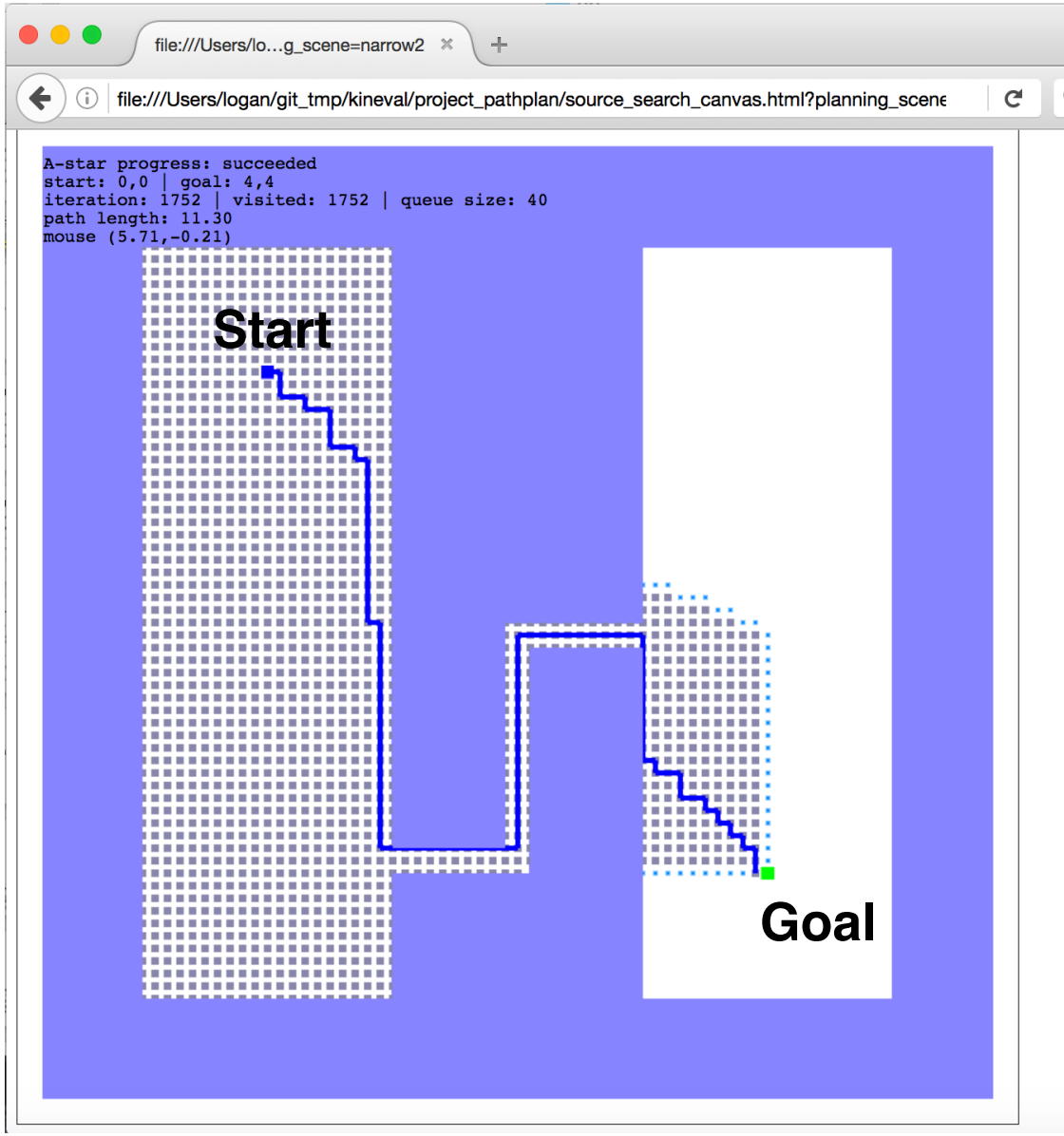


$$\max(|x_1 - x_2|, |y_1 - y_2|)$$

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
- 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.org">
EECS 367 Introduction to Autonomous Robotics <br>
ROB 511 Robot Operating Systems
</a>

</body>
</html>
```

Next lecture:

JavaScript/HTML5 and git Tutorial

EECS 367 Introduction to Autonomous Robotics
ROB 511 Robot Operating Systems