Lecture 09
Planning - I - Path Planning & PID Control

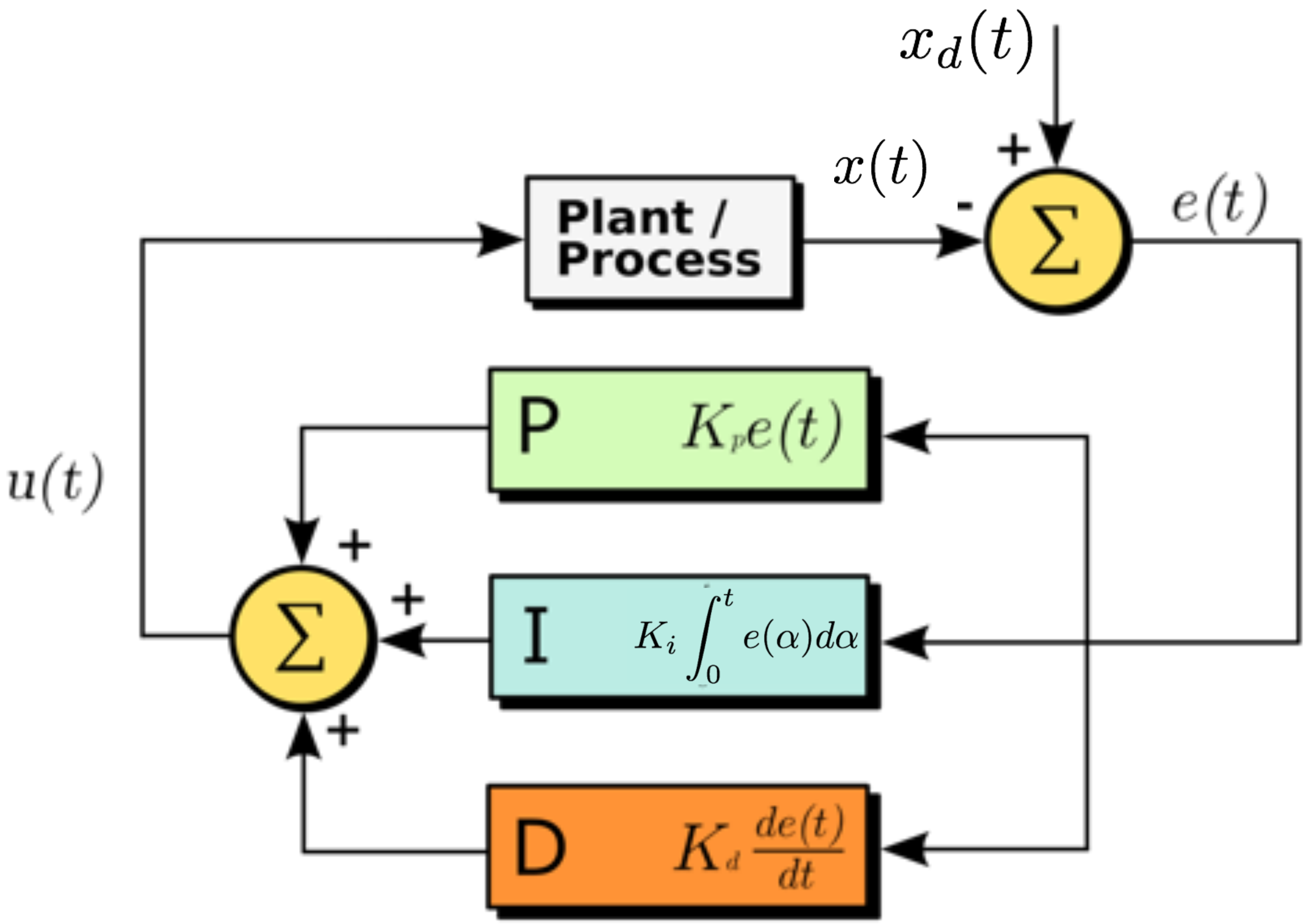
Course Logistics

- **Quiz 7 was posted today and was due before the lecture.**
- Project 2 is posted on 10/02 and will be due 10/11
  - Start early!

PID Control

PID Control

- Proportional-Integral-Derivative Control
- Sum of different responses to error
- Based on the mass spring and damper system
- Feedback correction based on the current error, past error, and predicted future error



Slide borrowed from Michigan Robotics autorob.org

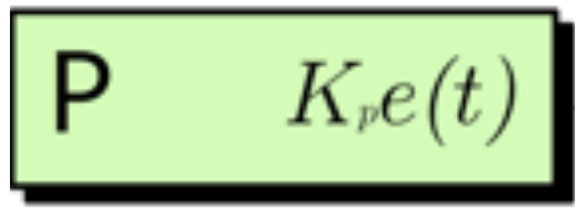
PID Control

Error signal:

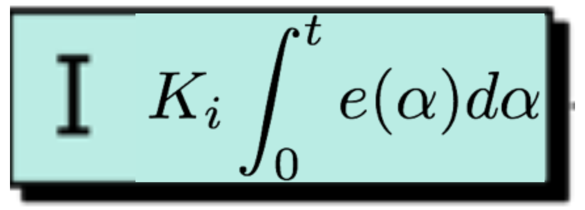
$$e(t) = x_{desired}(t) - x(t)$$

Control signal:

$$u(t) = K_p e(t) + K_i \int_0^t e(\alpha)d\alpha + K_d \frac{d}{dt} e(t)$$

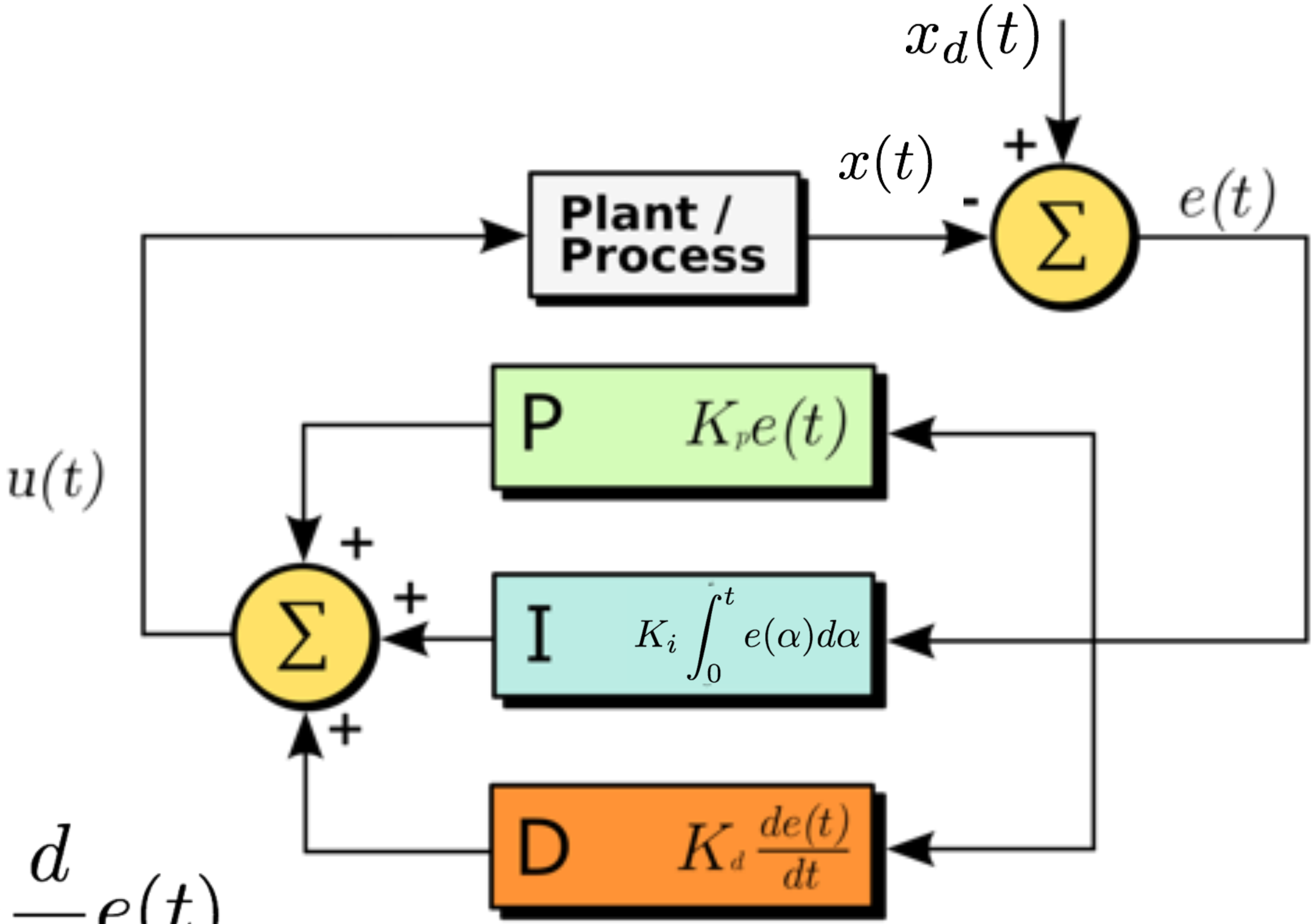


Current

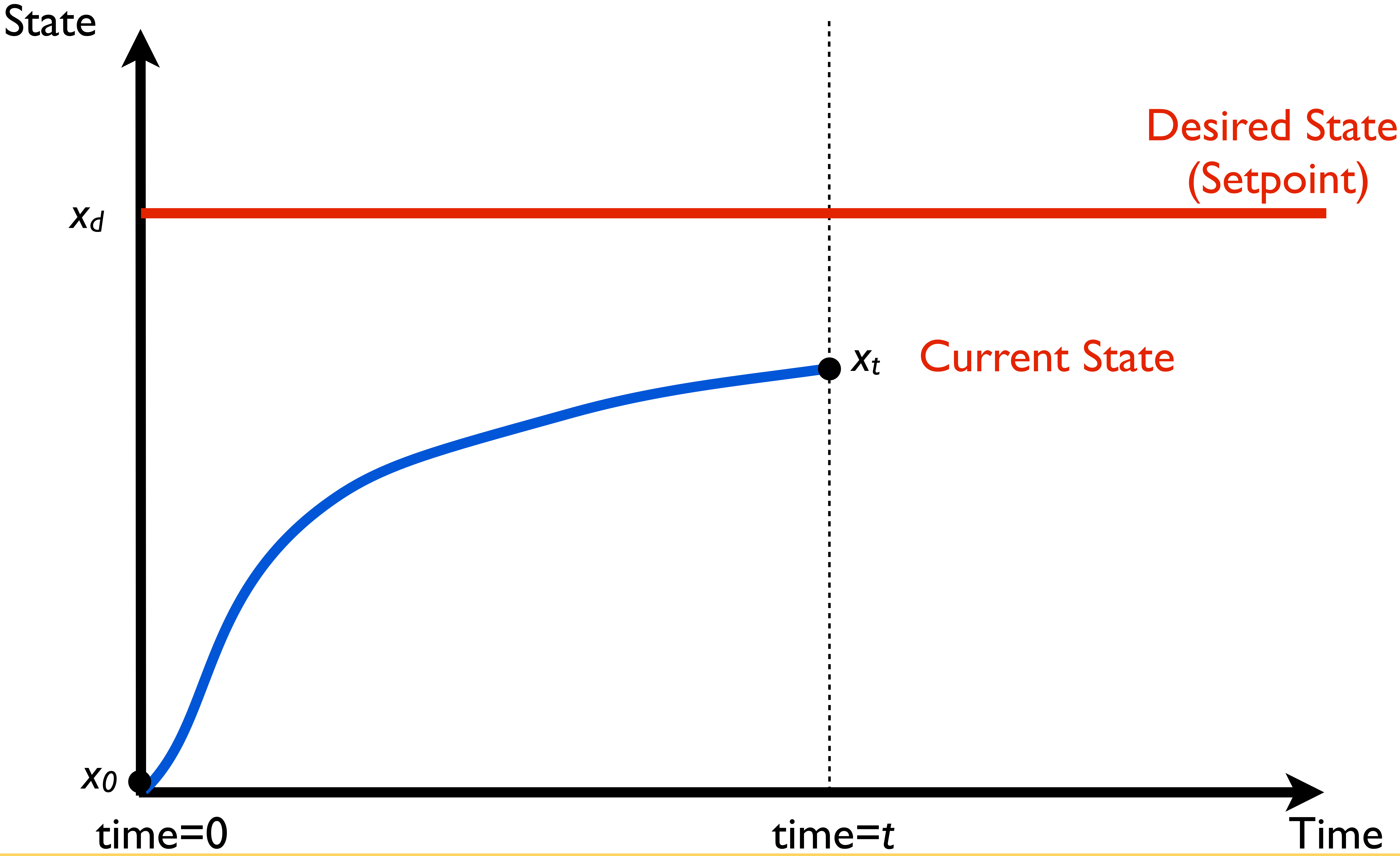


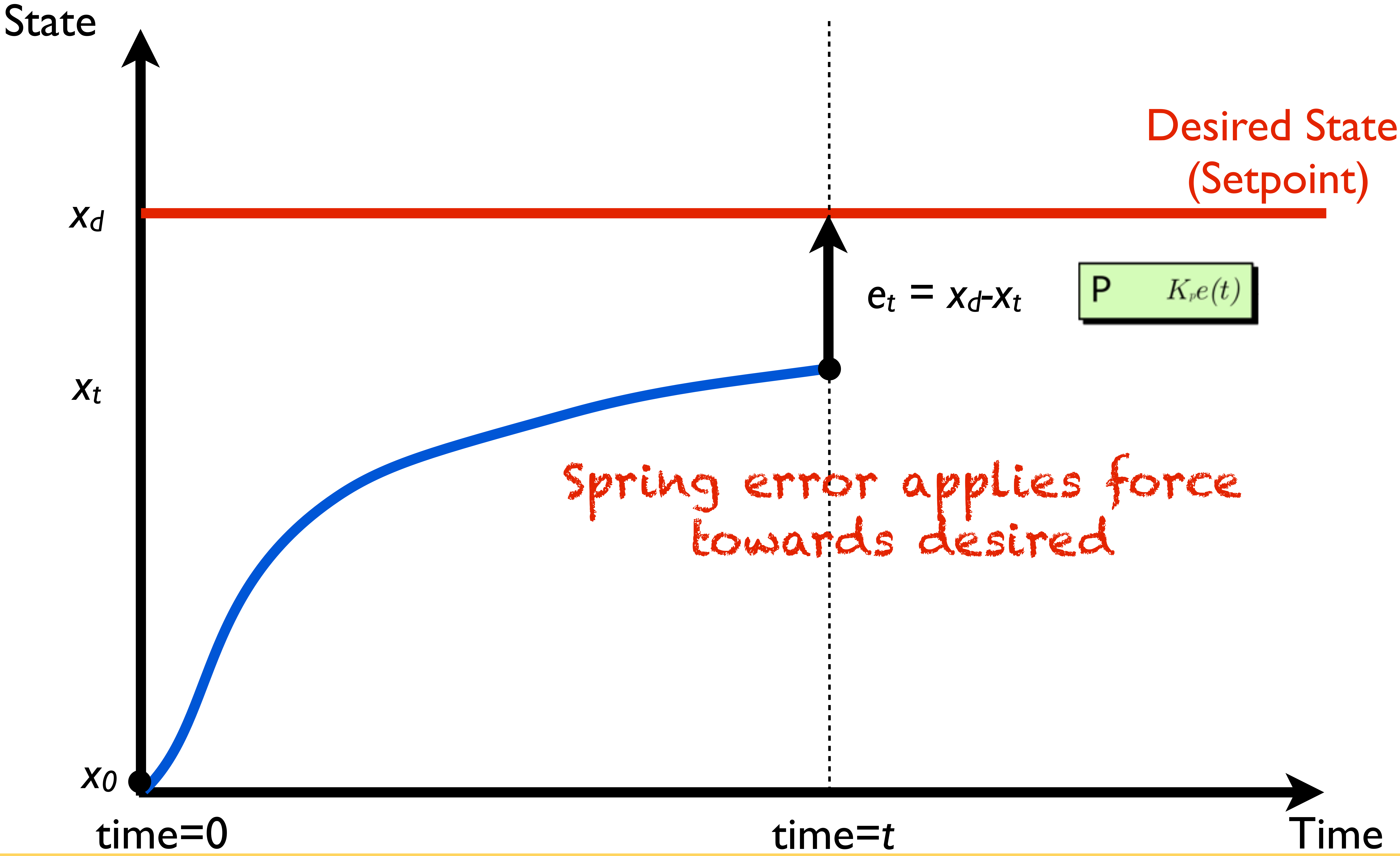
Past

Future

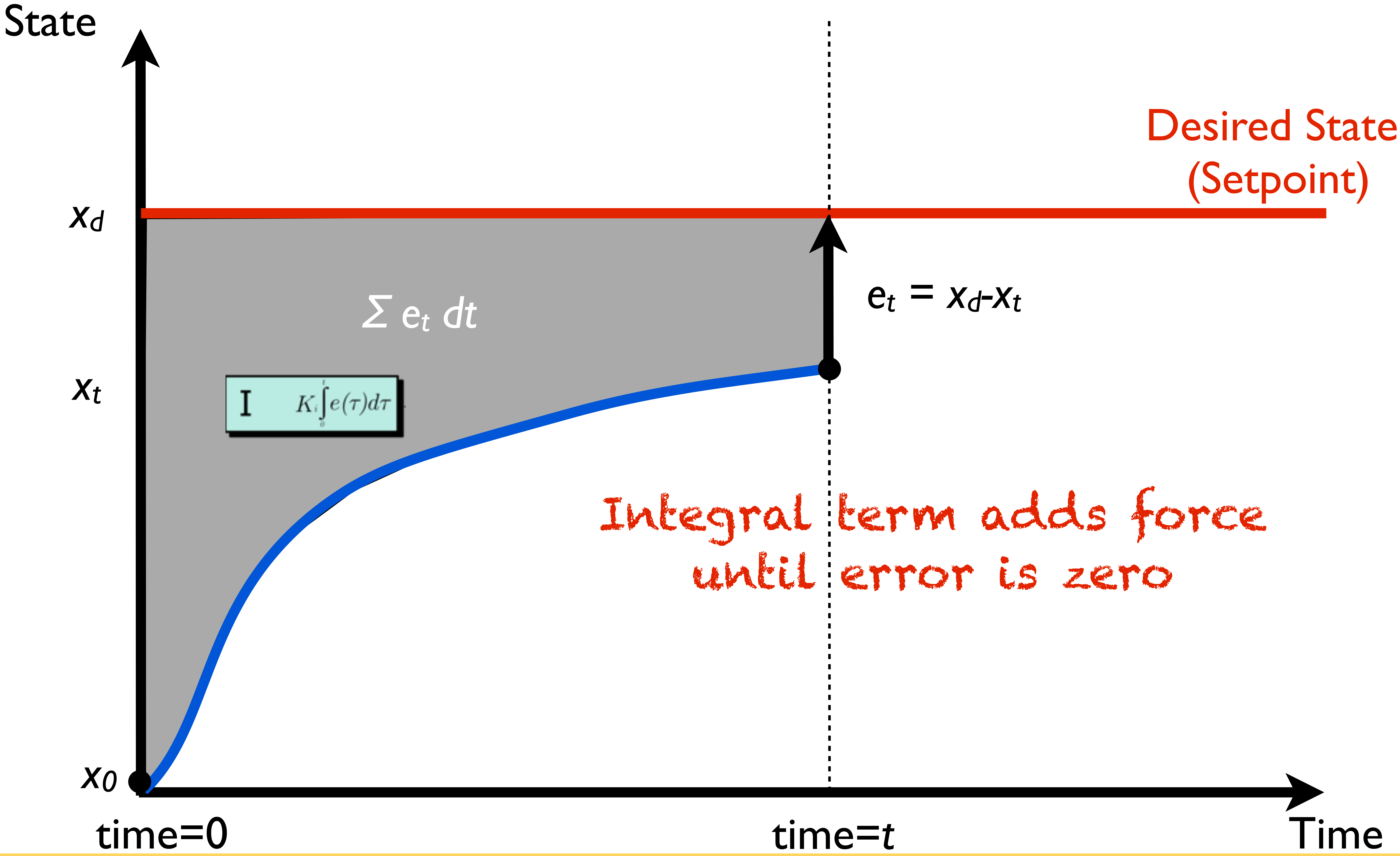


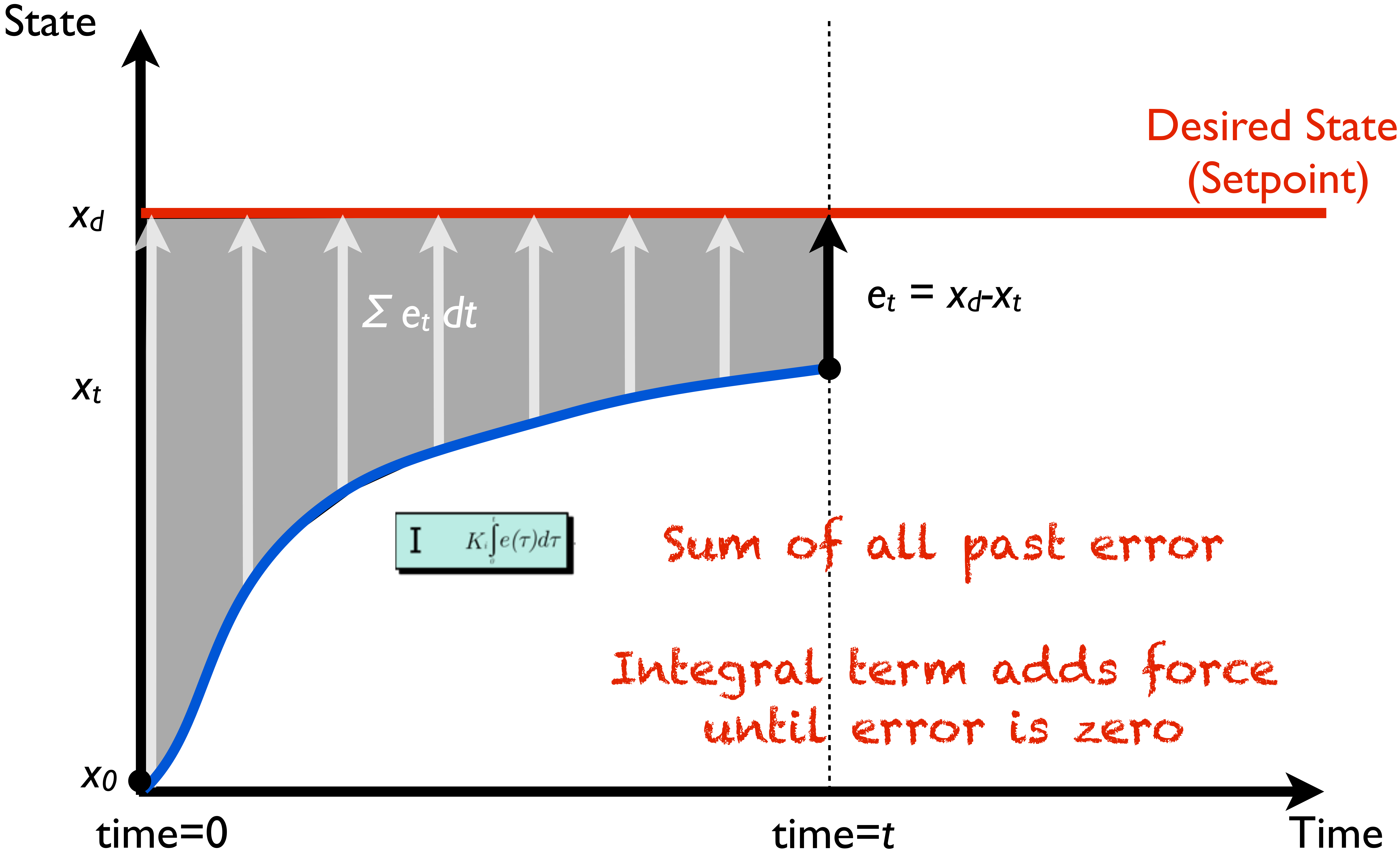
Consider PID wrt. state over time

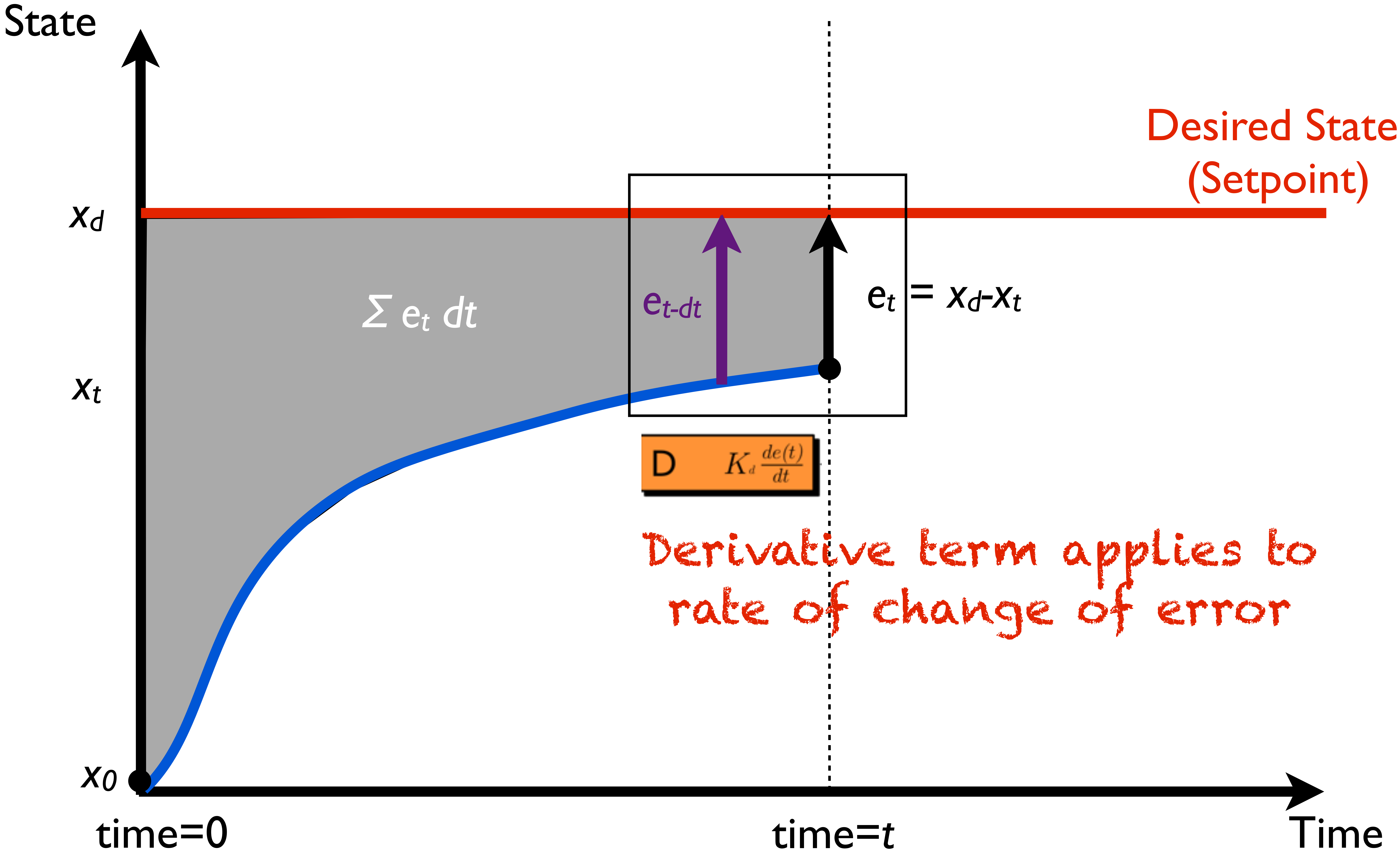

State
Desired State
(Setpoint)
x_d
x_t
Current State
x_0
time=0
time=t
Time



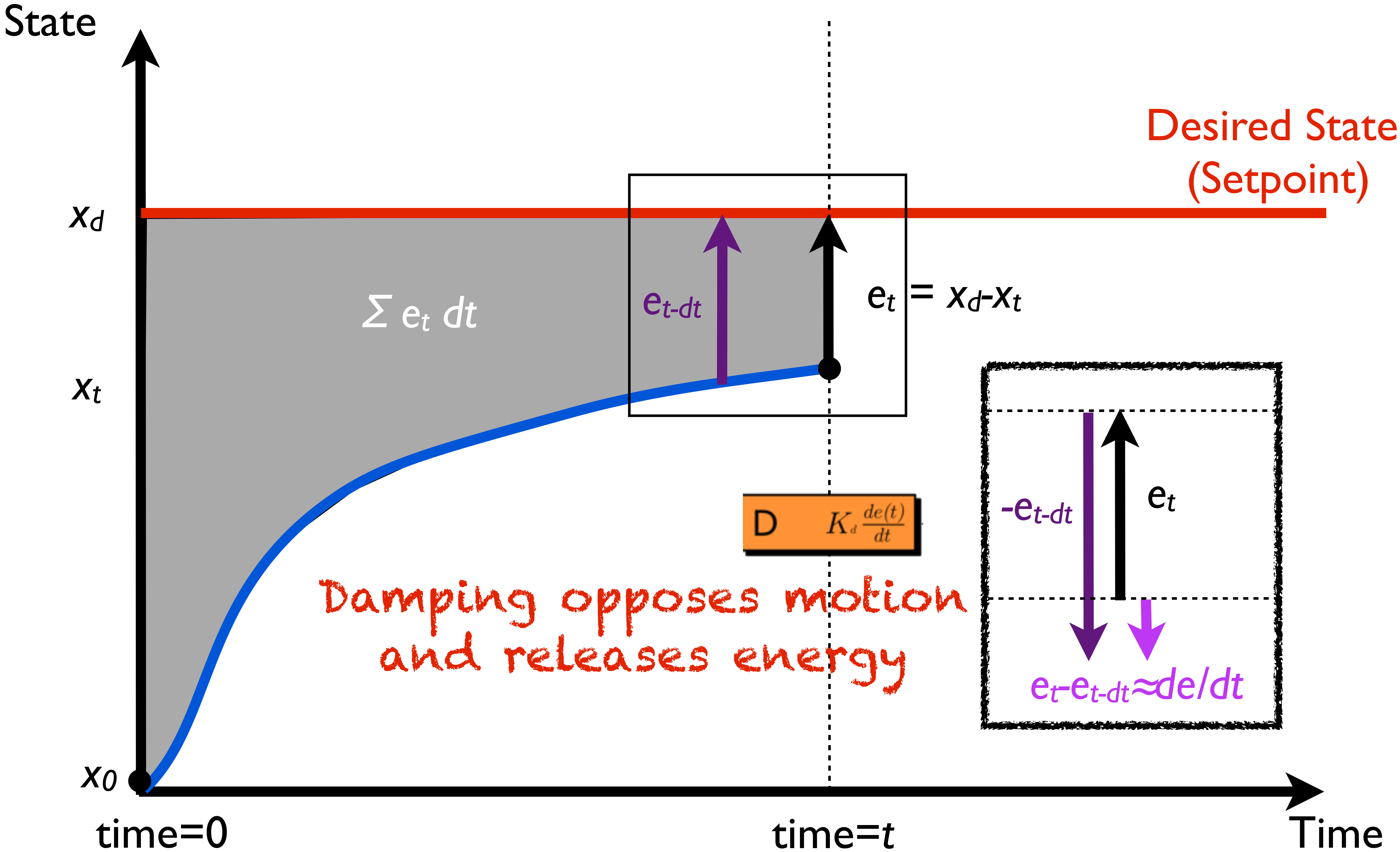
Slide borrowed from Michigan Robotics autorob.org

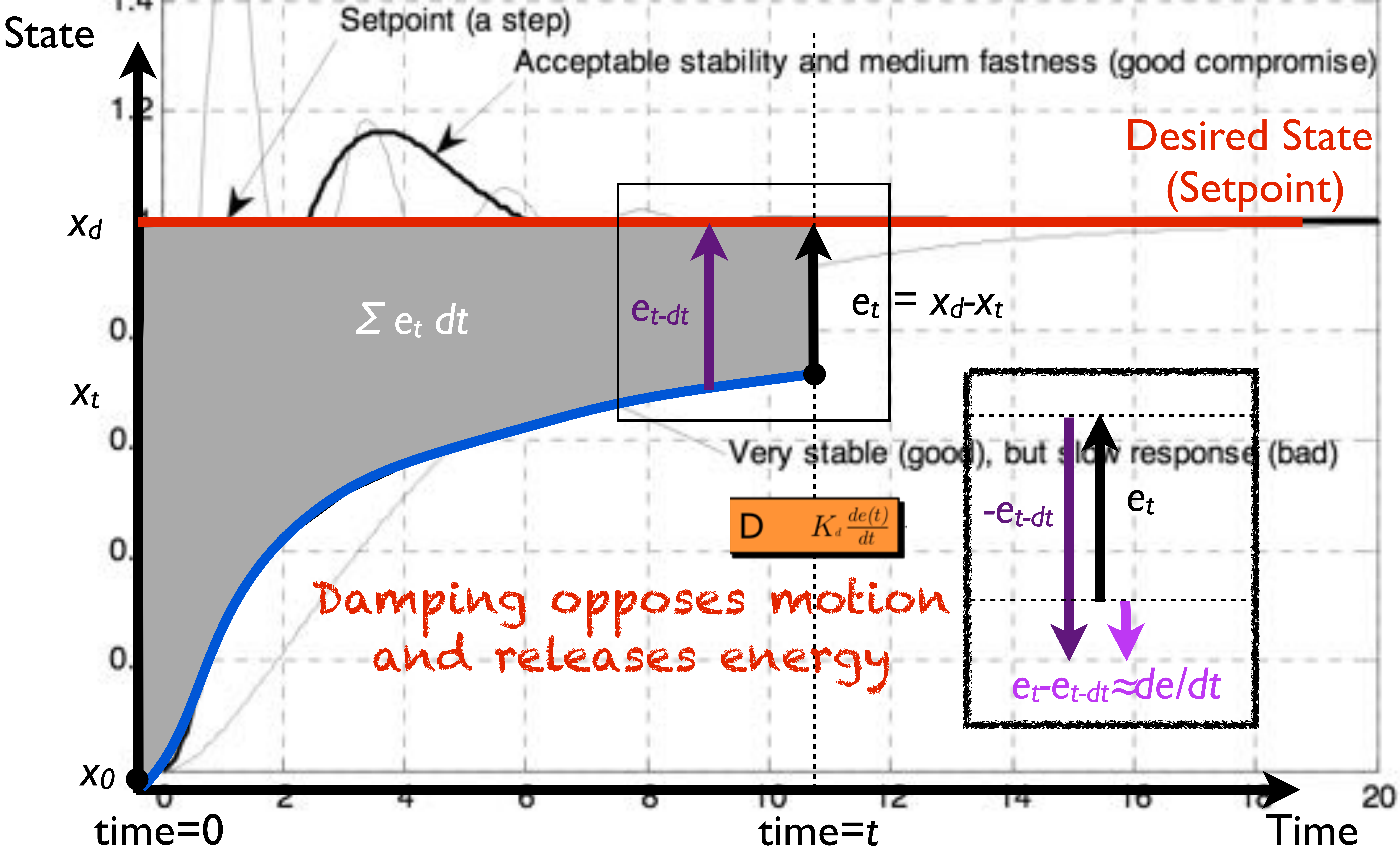

State
Desired State
(Setpoint)
x_d
$\Sigma\ e_t\ dt$
$e_t = x_d - x_t$
x_t
I $K_i \int_0^t e(\tau)d\tau$
Integral term adds force
until error is zero
x_0
time=0
time=t
Time


State
Desired State
(Setpoint)
x_d
x_t
x_0
$\Sigma\, e_t\, dt$
$e_t = x_d - x_t$
I $K_i \int_0^t e(\tau)d\tau$
Sum of all past error
Integral term adds force
until error is zero
time=0
time=t
Time

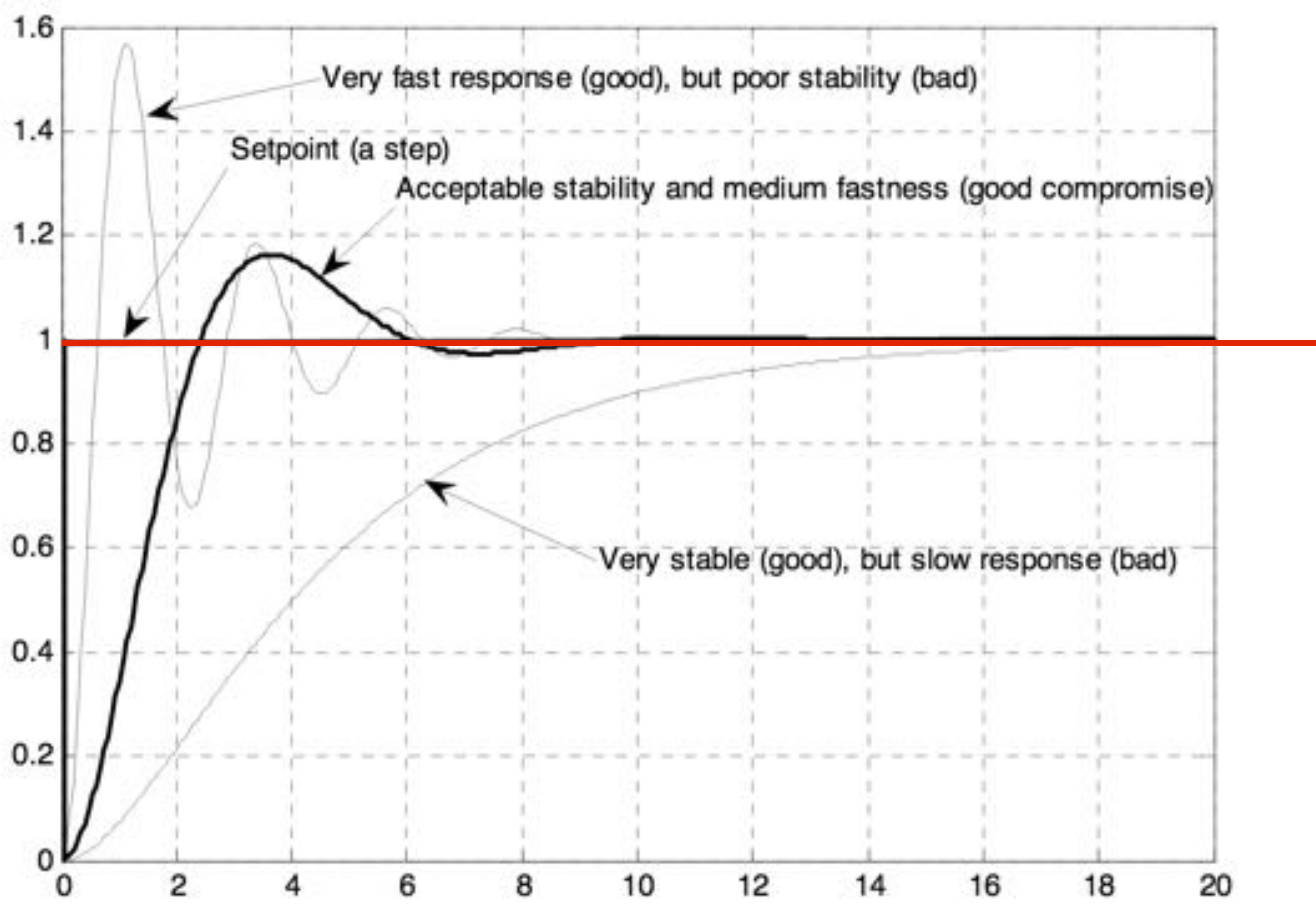


Slide borrowed from Michigan Robotics autorob.org


State
Desired State
(Setpoint)
x_d
x_t
x_0
$\Sigma\ e_t\ dt$
e_{t-dt}
$e_t = x_d - x_t$
D $K_d \frac{de(t)}{dt}$
Damping opposes motion
and releases energy
$-e_{t-dt}$
e_t
$e_t - e_{t-dt} \approx de/dt$
time=0
time=t
Time


State
Setpoint (a step)
Acceptable stability and medium fastness (good compromise)
Desired State
(Setpoint)
x_d
x_t
x_0
$\Sigma\ e_t\ dt$
$e_{t\text{-}dt}$
$e_t = x_d\text{-}x_t$
Very stable (good), but slow response (bad)
D $K_d \frac{de(t)}{dt}$
$-e_{t\text{-}dt}$
e_t
$e_t\text{-}e_{t\text{-}dt} \approx de/dt$
Damping opposes motion
and releases energy
time=0
time=t
Time

PID Converenge



Slide borrowed from Michigan Robotics autorob.org

PID as a spring and damper model

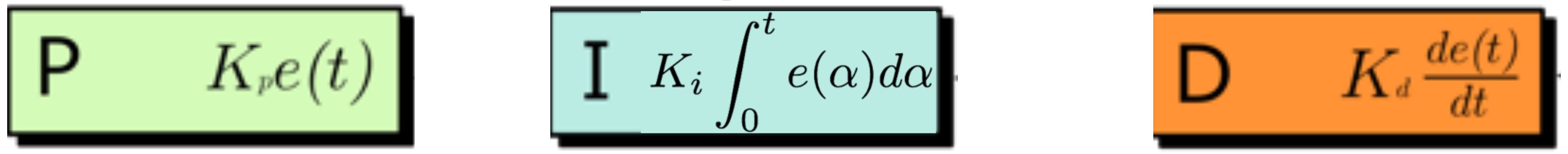
PID Control

Error signal:

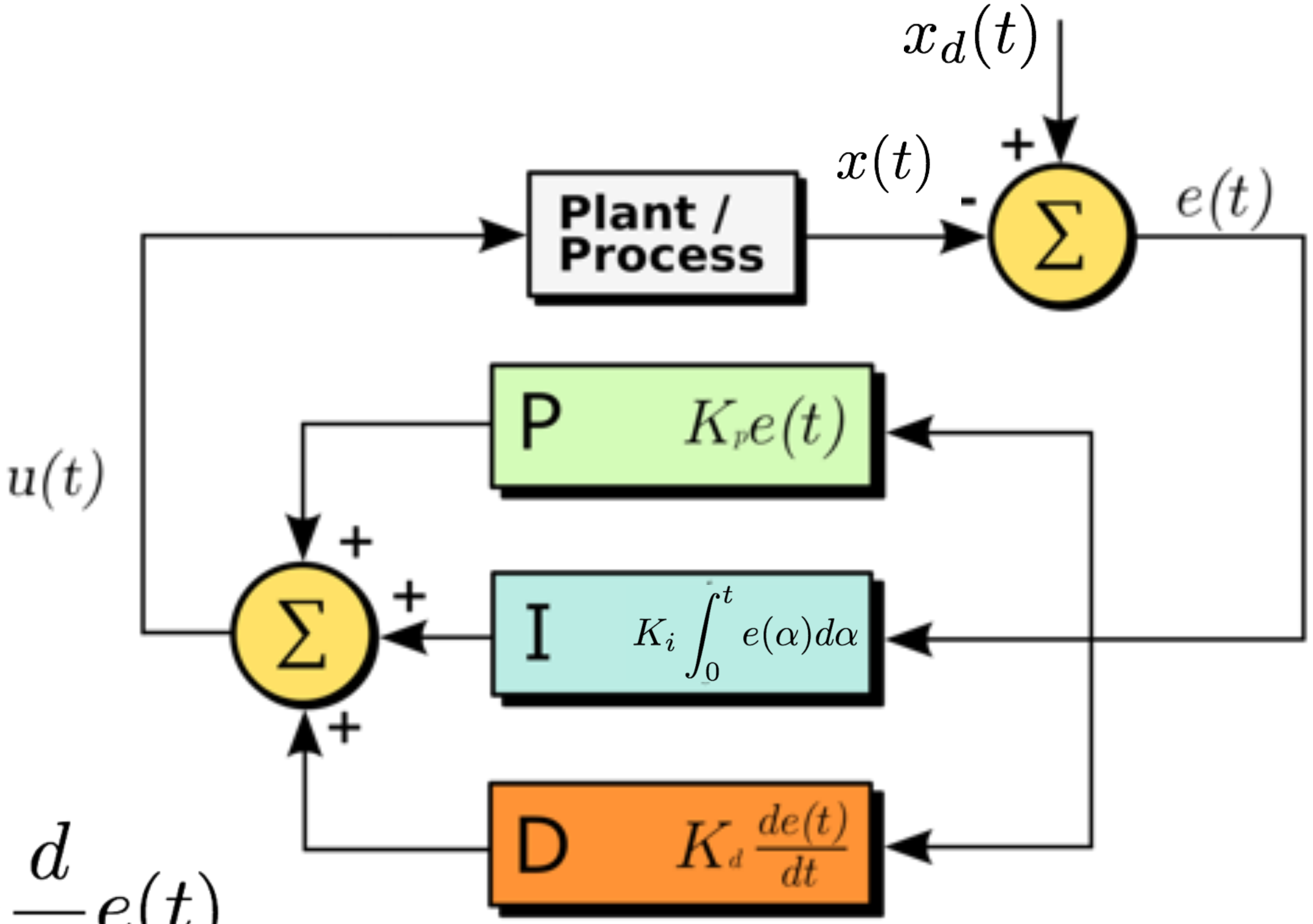
$$e(t) = x_{desired}(t) - x(t)$$

Control signal:

$$u(t) = K_p e(t) + K_i \int_0^t e(\alpha)d\alpha + K_d \frac{d}{dt} e(t)$$



Current Past Future



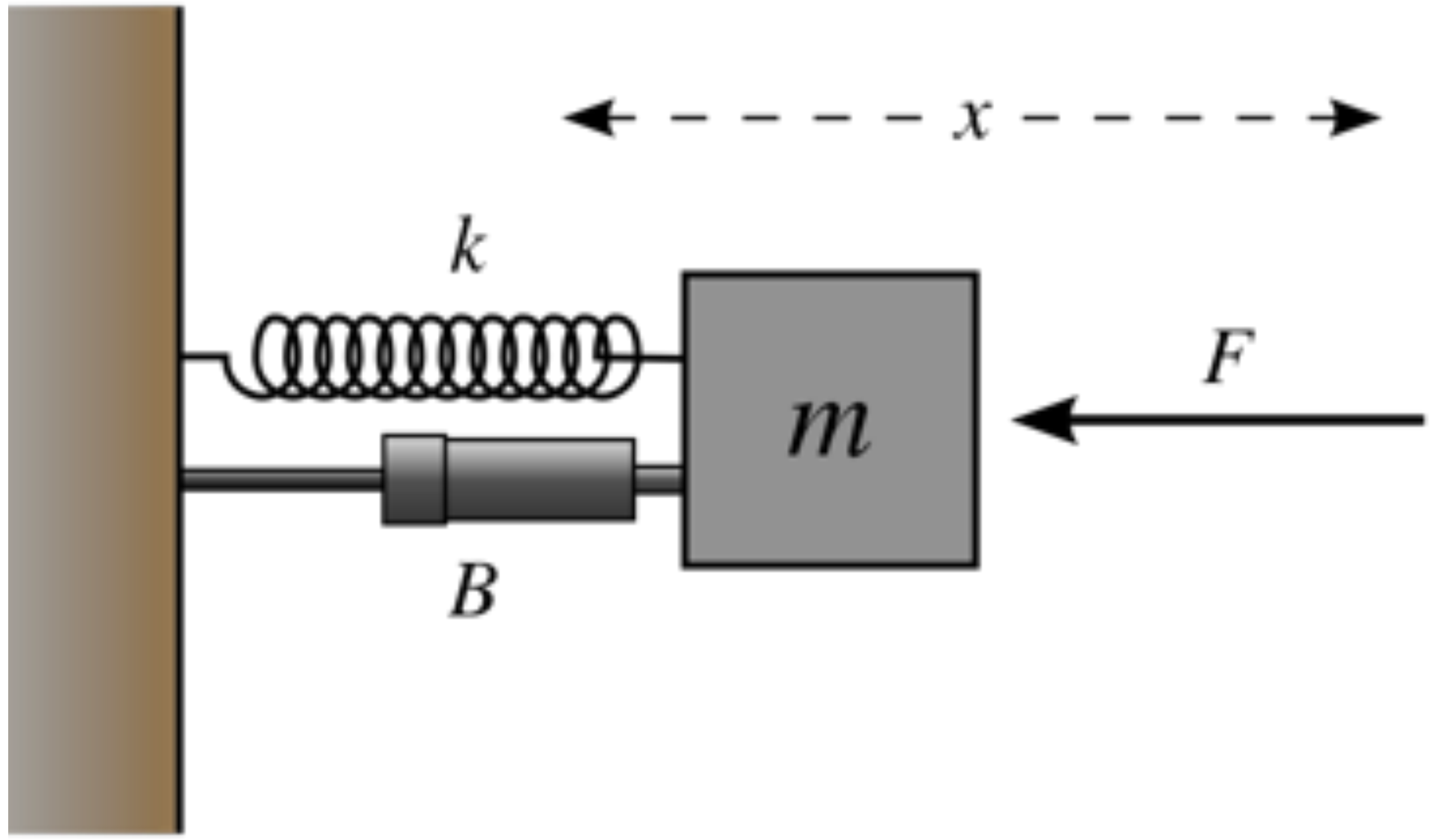
Hooke's Law

P $K_p e(t)$

- Describes motion of mass spring damper system as

$$F = -kx$$

Robert Hooke
(1635-1703)

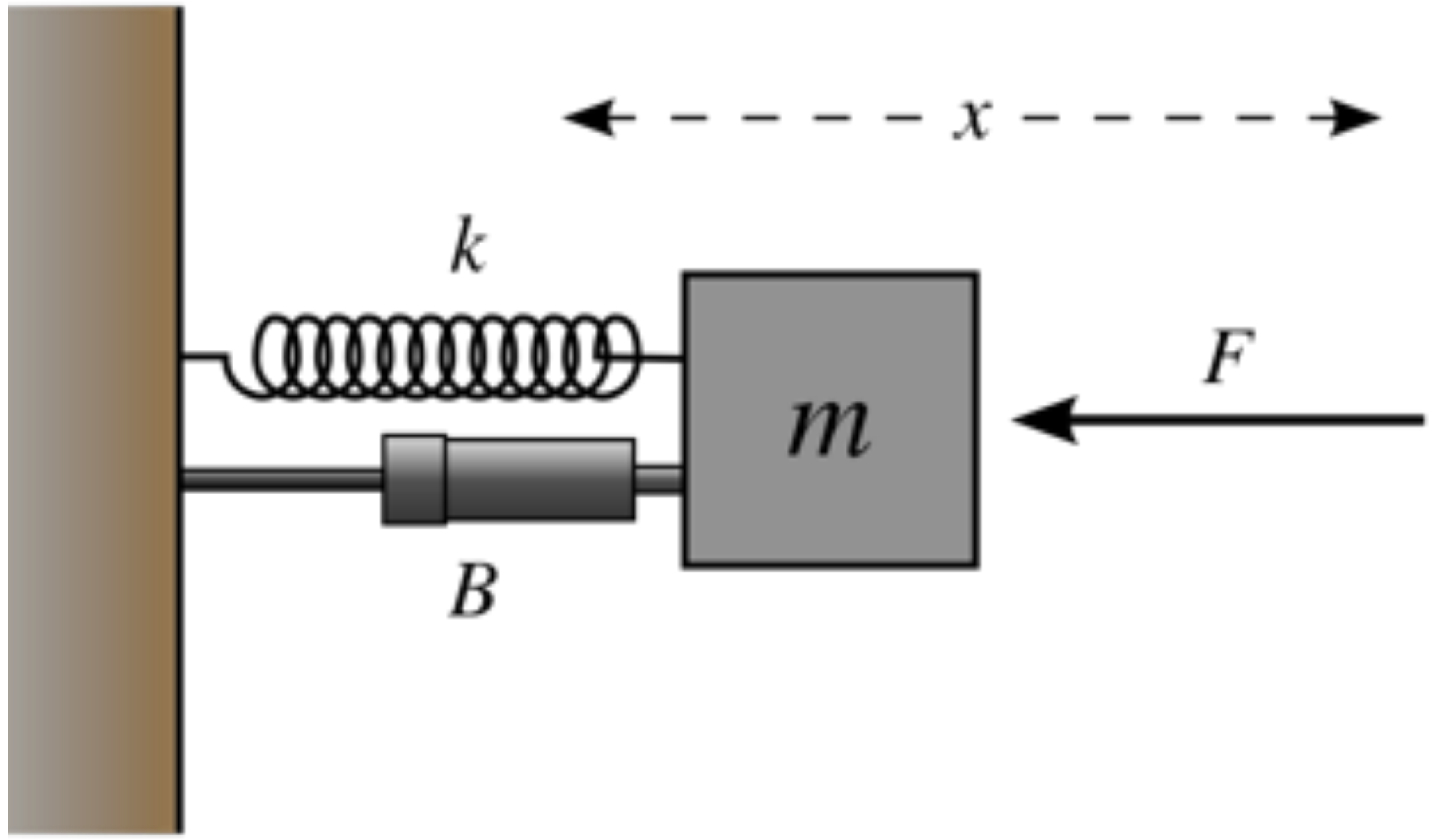


Slide borrowed from Michigan Robotics autorob.org

Hooke's Law

P $K_p e(t)$

- Describes motion of mass spring damper system as



$$F = -kx$$

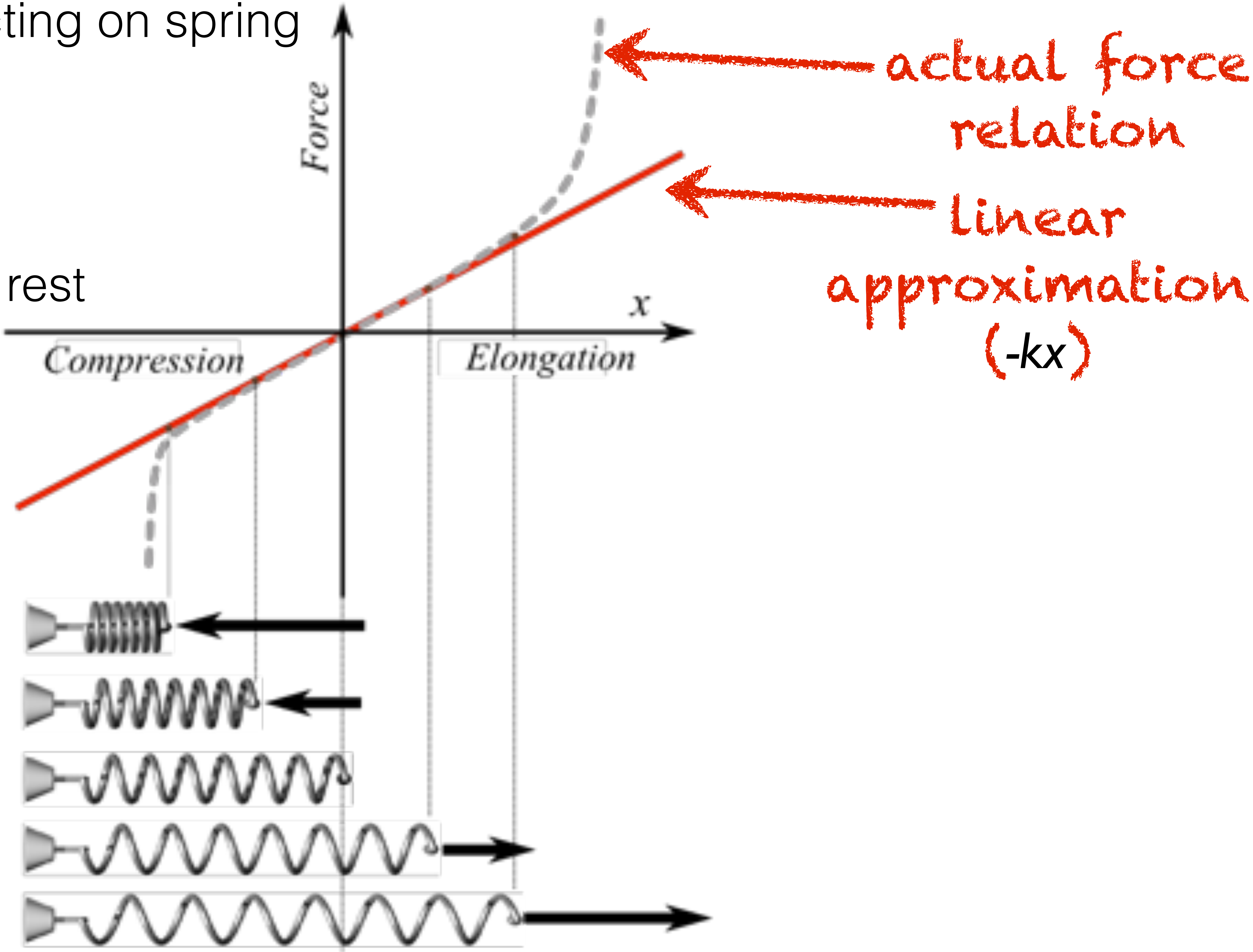
force moving spring towards rest

spring stiffness

distance from rest displacement

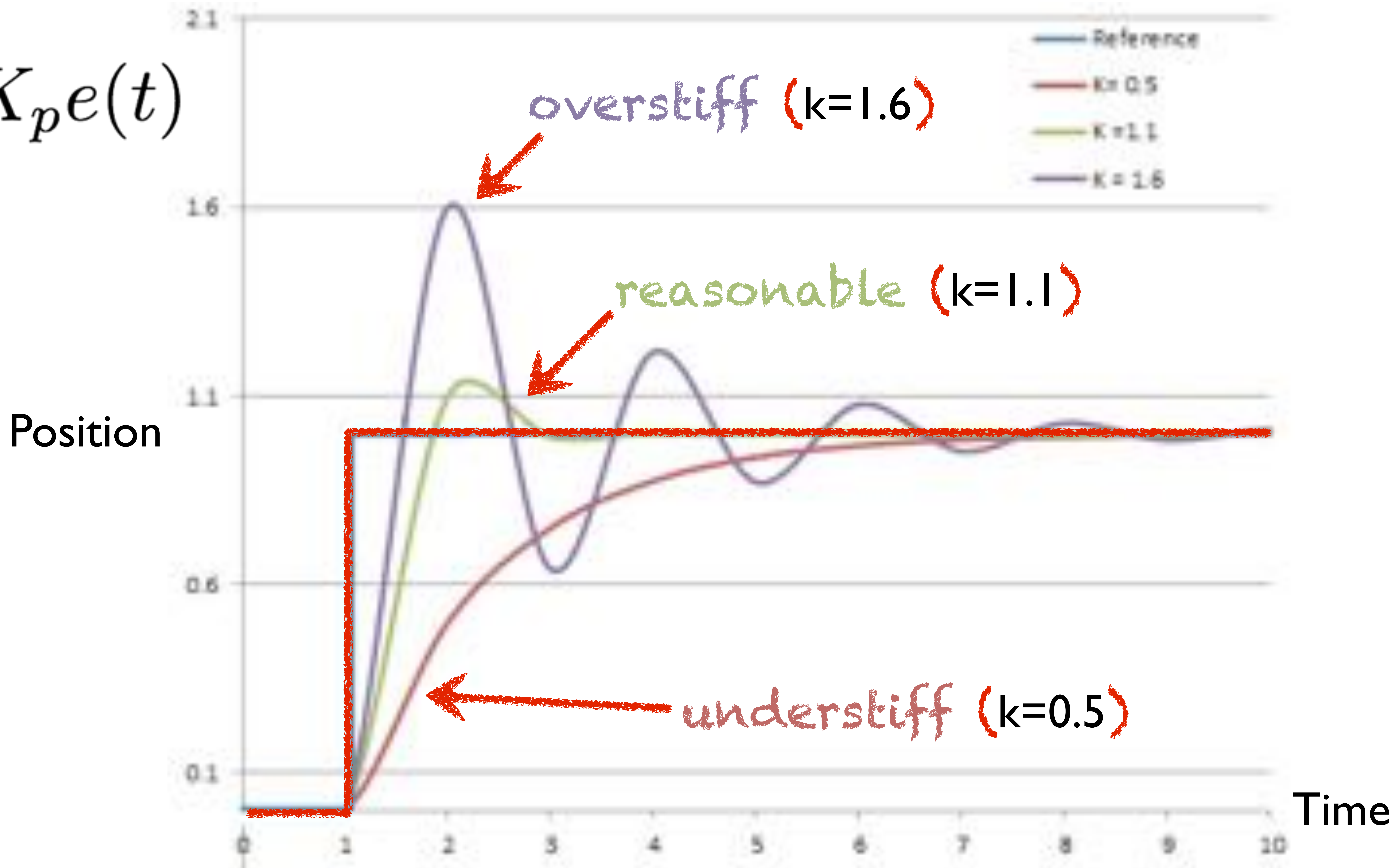
Vertical: Force acting on spring

Horizontal: Position from rest



Slide borrowed from Michigan Robotics autorob.org

$K_p e(t)$



Slide borrowed from Michigan Robotics autorob.org

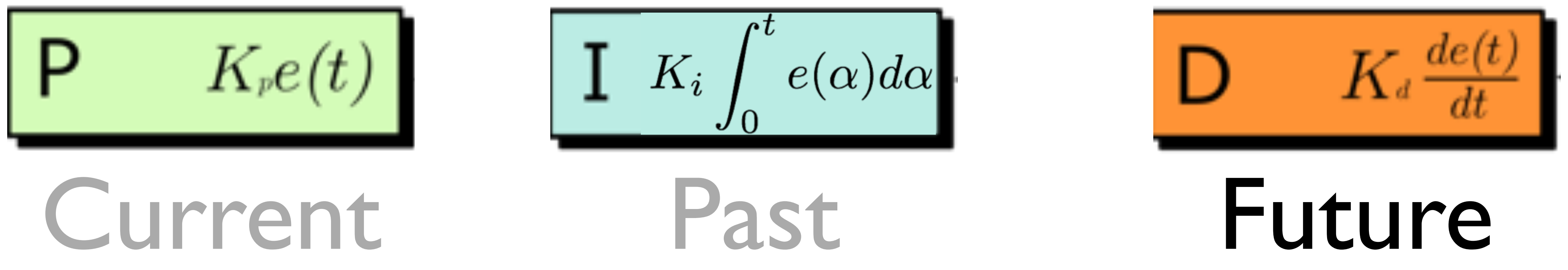
PID Control

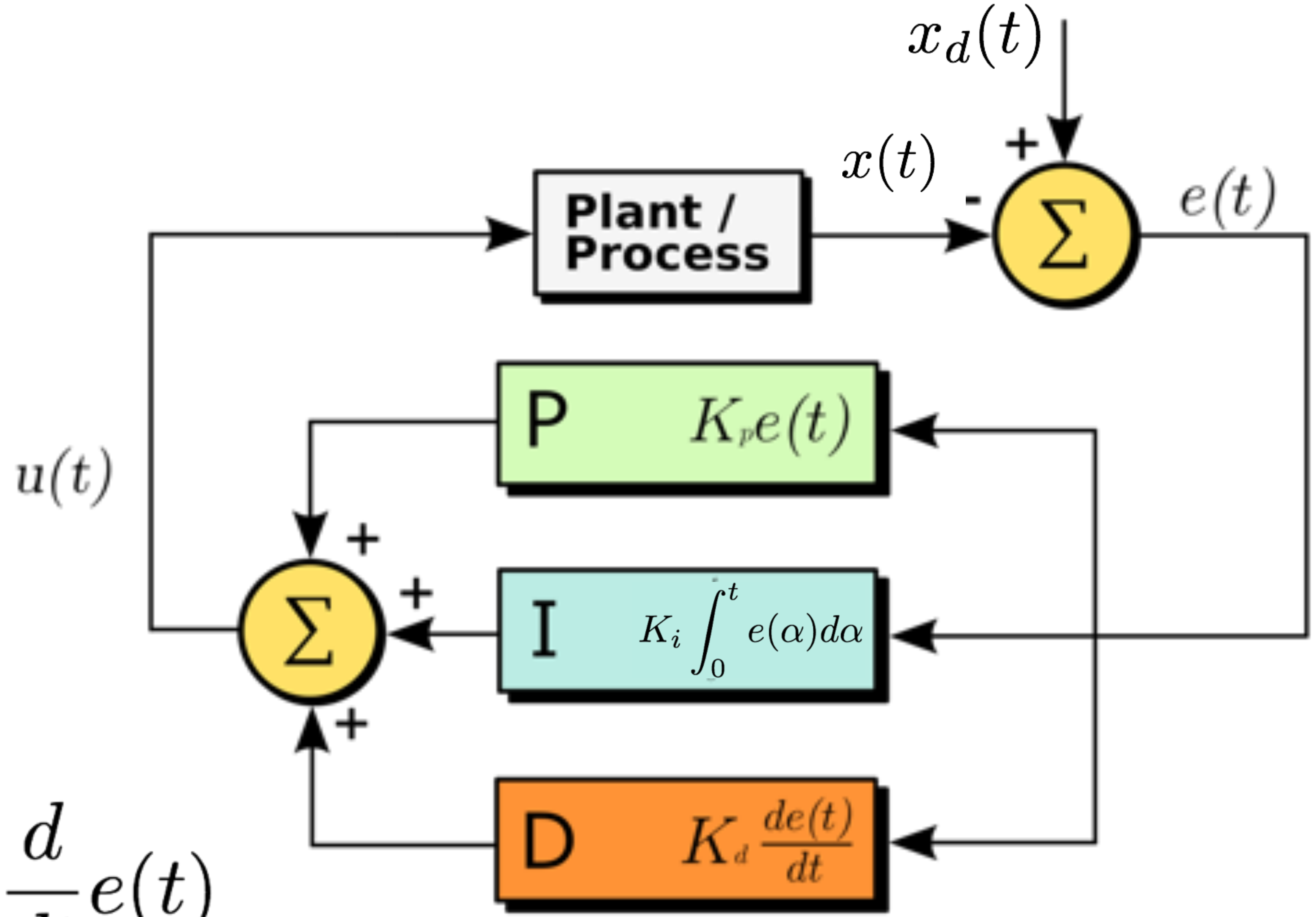
Error signal:

$$e(t) = x_{desired}(t) - x(t)$$

Control signal:

$$u(t) = K_p e(t) + K_i \int_0^t e(\alpha) d\alpha + K_d \frac{d}{dt} e(t)$$



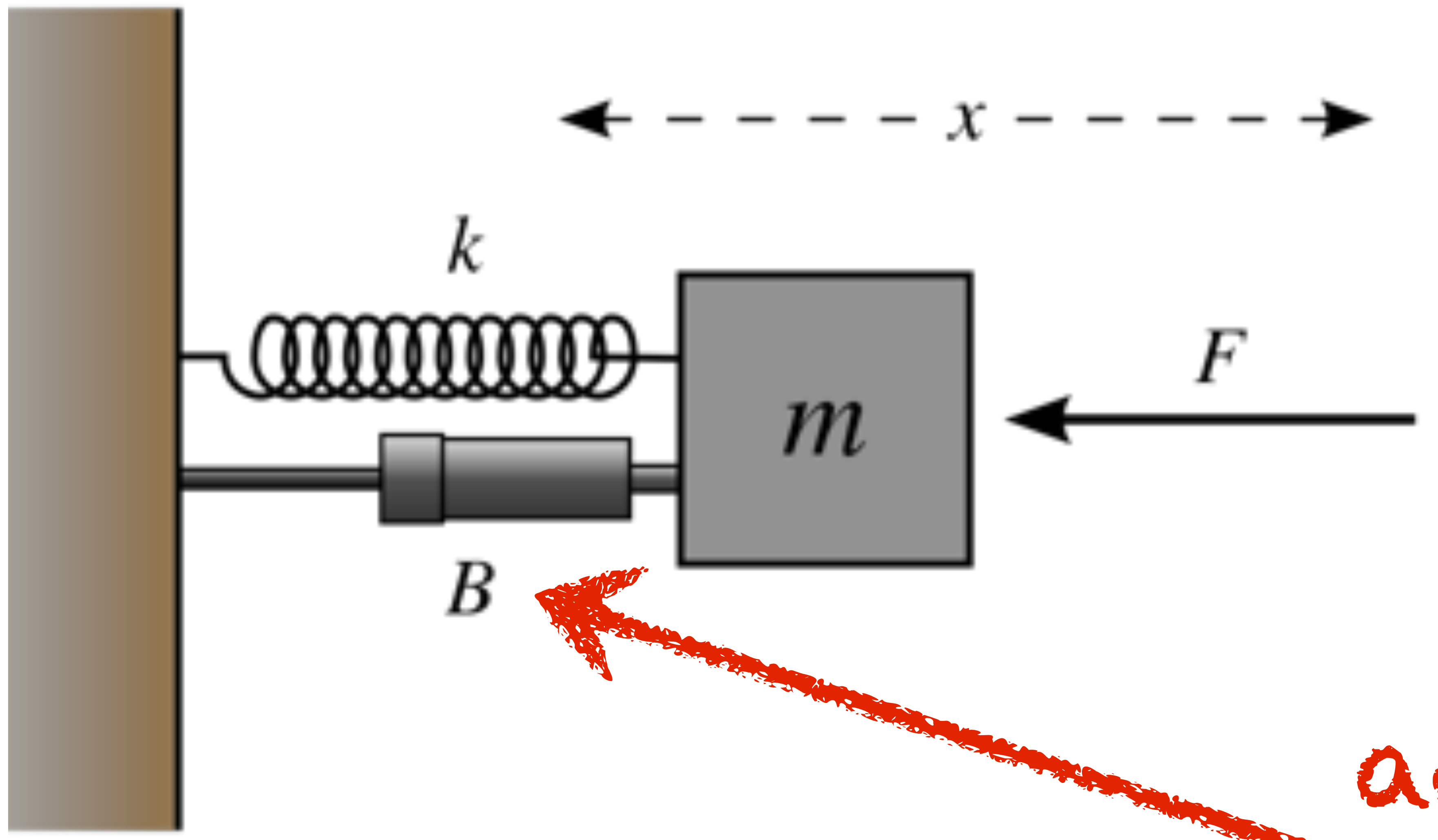


Spring and Damper

P $K_p e(t)$

D $K_d \frac{de(t)}{dt}$

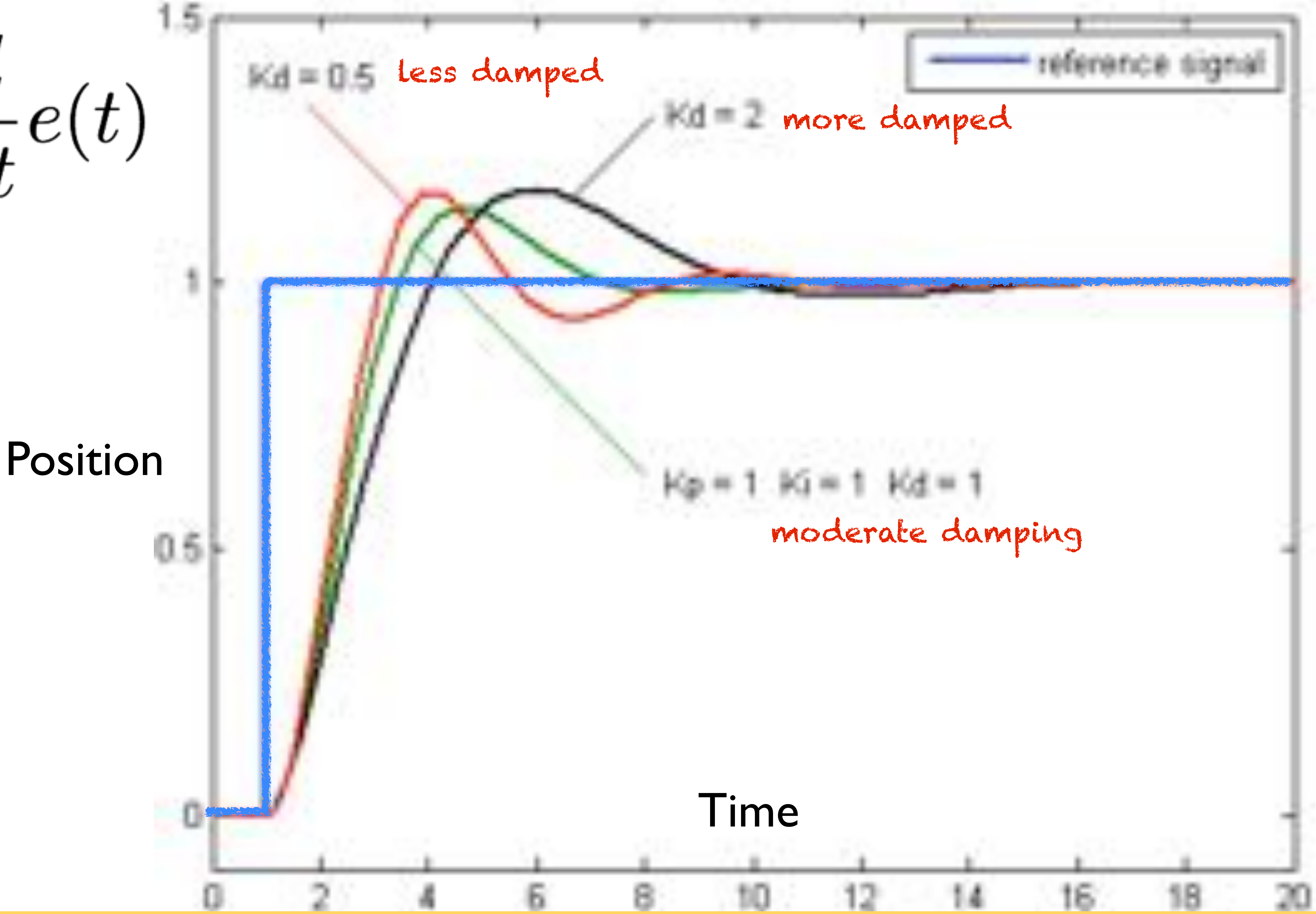
$$F = -kx + -b\dot{x}$$



assuming constant set point, velocity is derivative of error

add damper to release energy

$$K_d \frac{d}{dt} e(t)$$



Slide borrowed from Michigan Robotics autorob.org

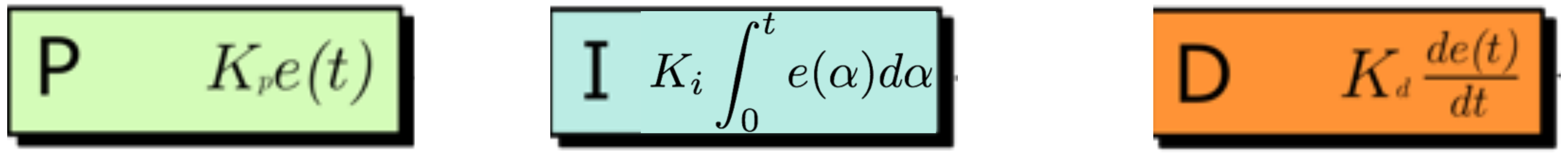
PID Control

Error signal:

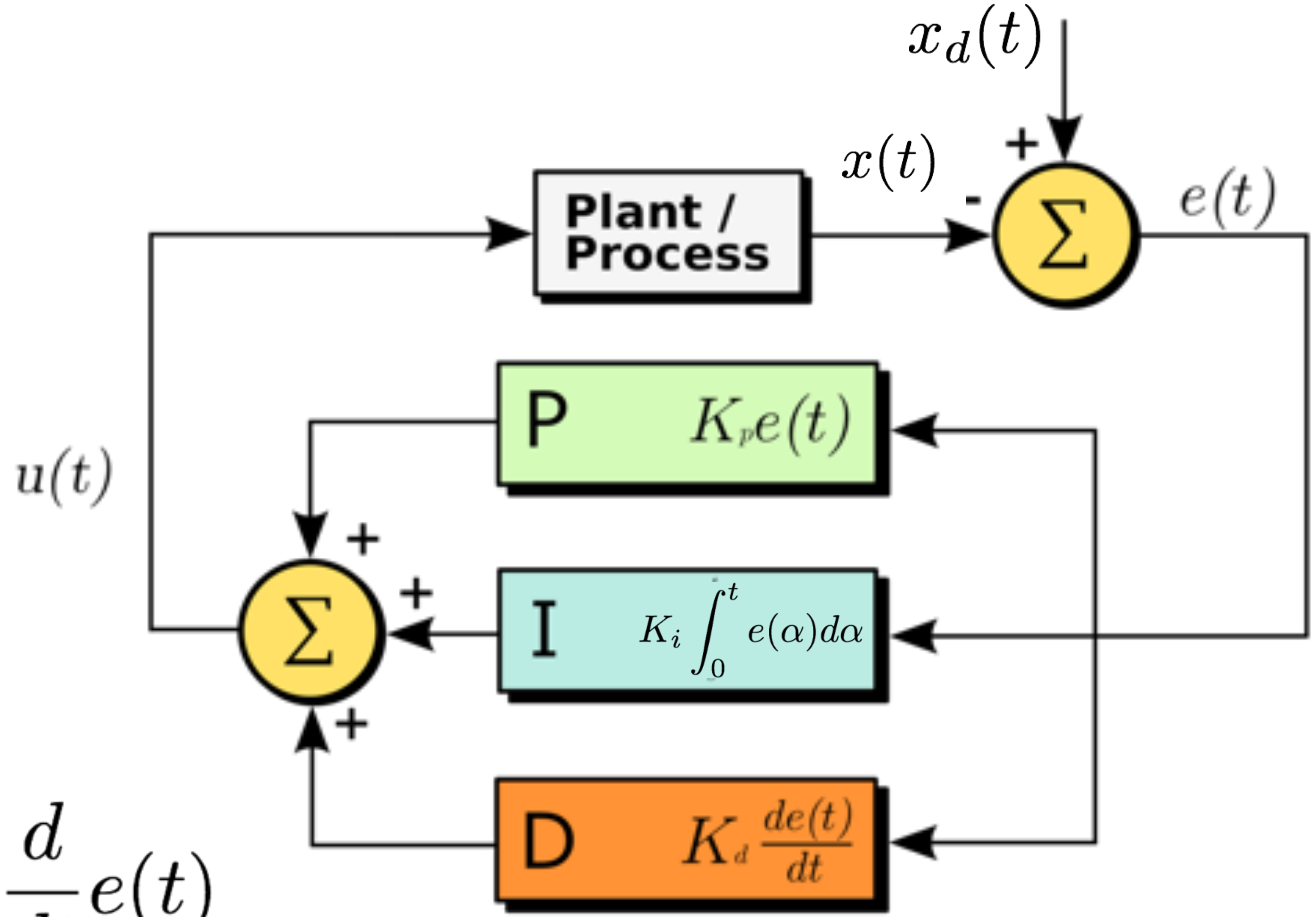
$$e(t) = x_{desired}(t) - x(t)$$

Control signal:

$$u(t) = K_p e(t) + K_i \int_0^t e(\alpha) d\alpha + K_d \frac{d}{dt} e(t)$$

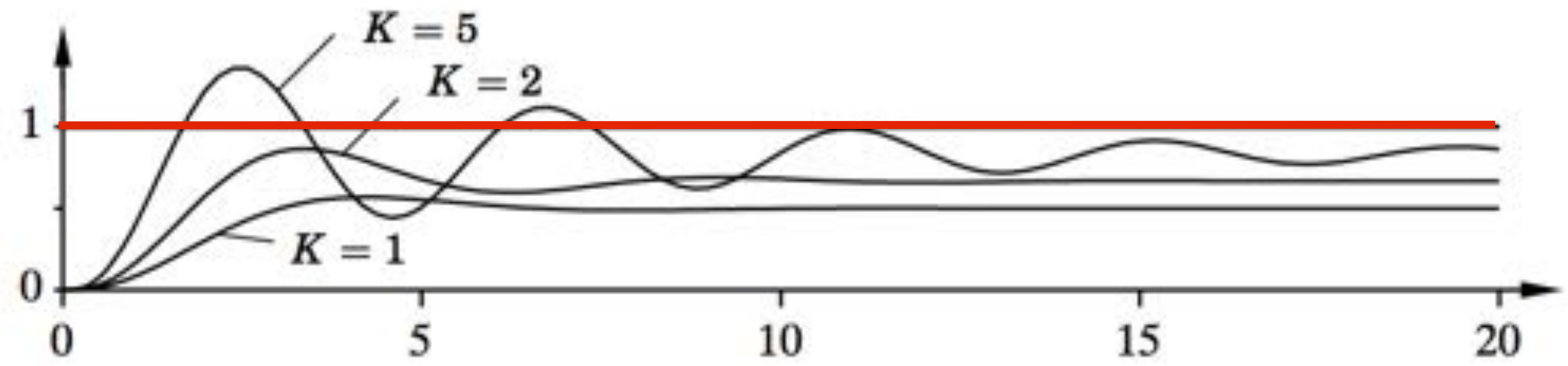


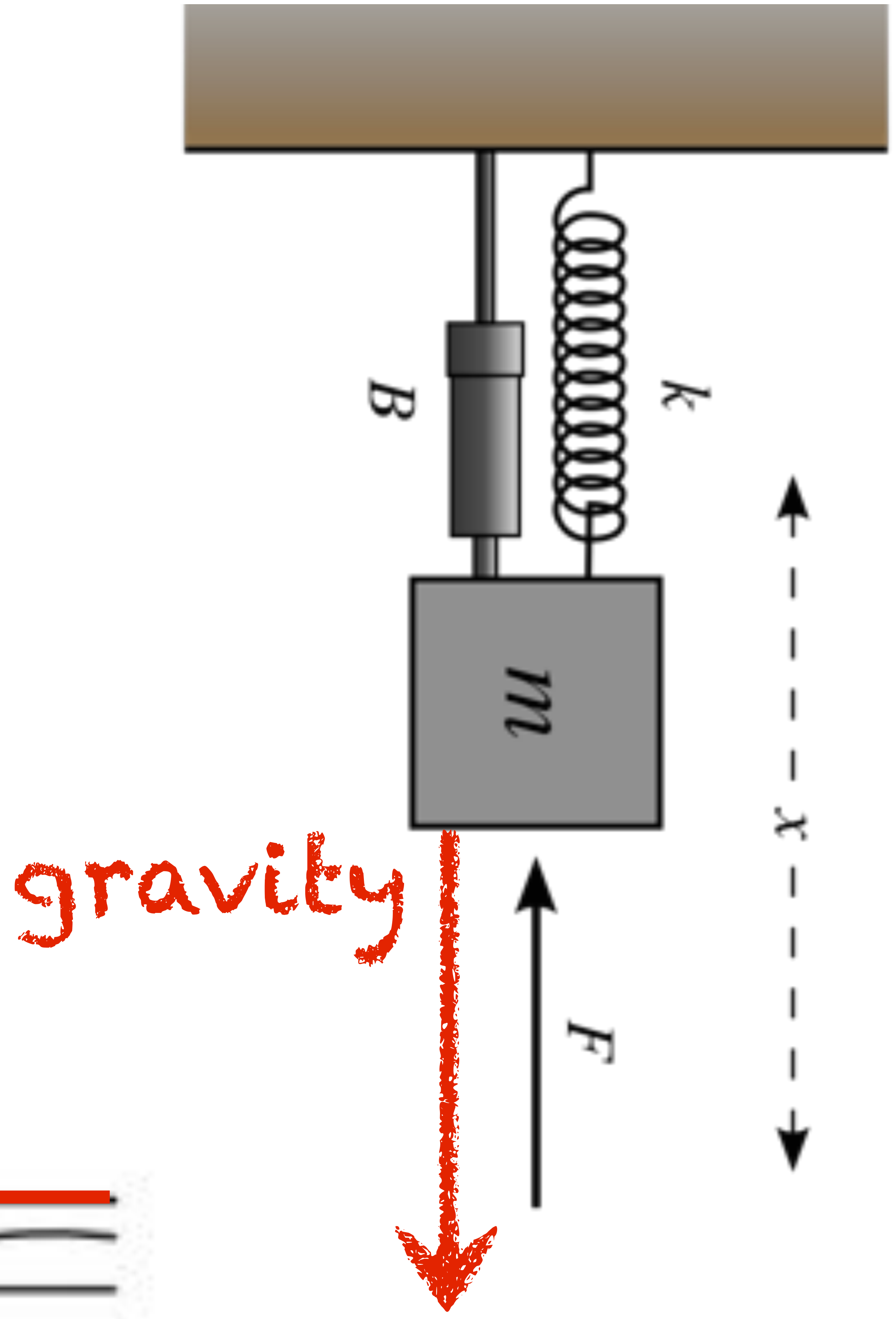
Current Past Future



Steady state error

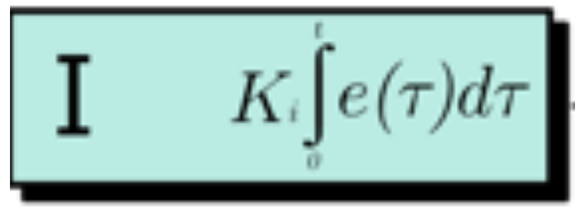
- Steady state error occurs when the system rests at equilibrium before reaching desired state
- Cause could be an significant external force, weak motor, low proportional gain, etc.
- PID integral term compensates by accumulating and acting against error toward convergence

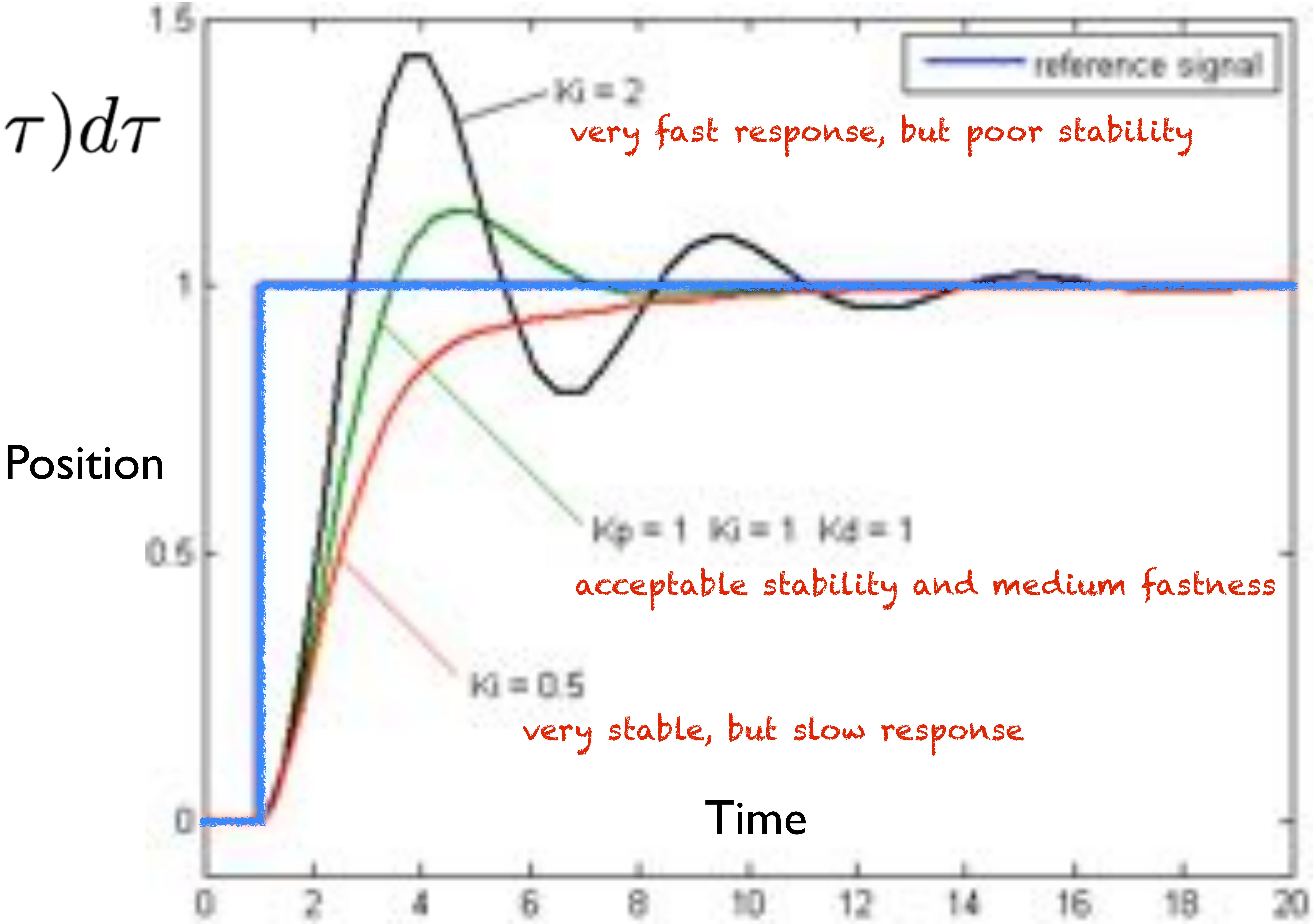




Slide borrowed from Michigan Robotics autorob.org

$$K_i \int_0^t e(\tau)d\tau$$





Gain tuning

- Implementing PID algorithm will not necessarily produce a good controller
- Selection of the gains will greatly affect the performance of the controller
- PID gain tuning is more of an art than a science. Choose carefully.

$$u(t) = \boxed{K_p} e(t) + \boxed{K_i} \int_0^t e(\alpha) d\alpha + \boxed{K_d} \frac{d}{dt} e(t)$$

P $K_p e(t)$

I $K_i \int_0^t e(\alpha) d\alpha$

D $K_d \frac{de(t)}{dt}$

Some tips to PID tuning

(take it or leave it)

- Start all gains at zero : $K_i = K_d = K_p = 0$
- Increase spring gain K_p until system roughly meets desired state
  - overshooting and oscillation about the desired state can be expected
- Increase damping gain K_d until the system is consistently stable
  - damping stabilizes motion, but system will have steady state error
- Increase integral gain K_i until the system consistently reaches desired
- Refine gains as needed to improve performance; Test from different states

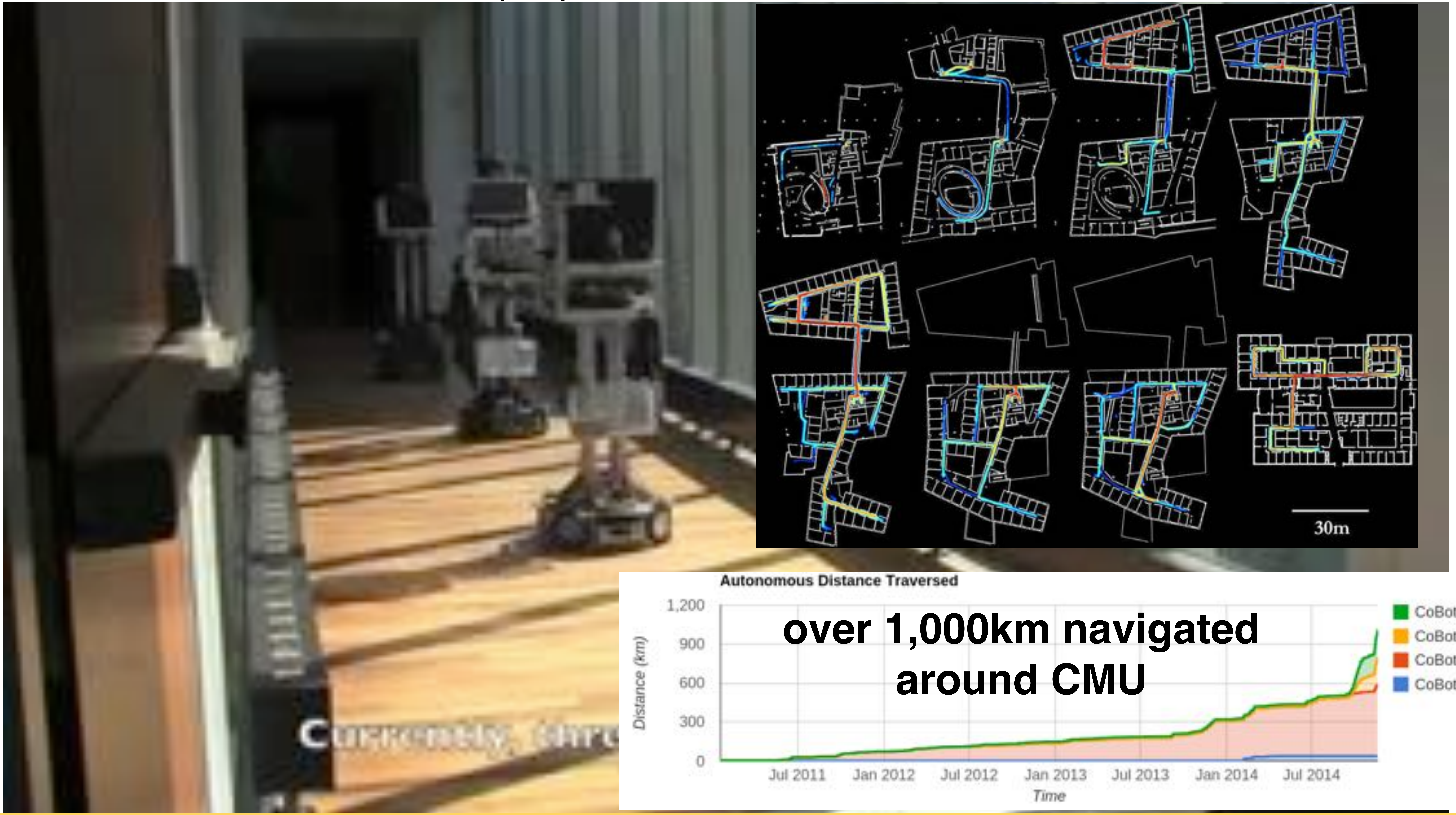
Path Planning

CMDragons 2015 Pass-ahead Goal

CMDragons 2015 slow-motion multi-pass goal

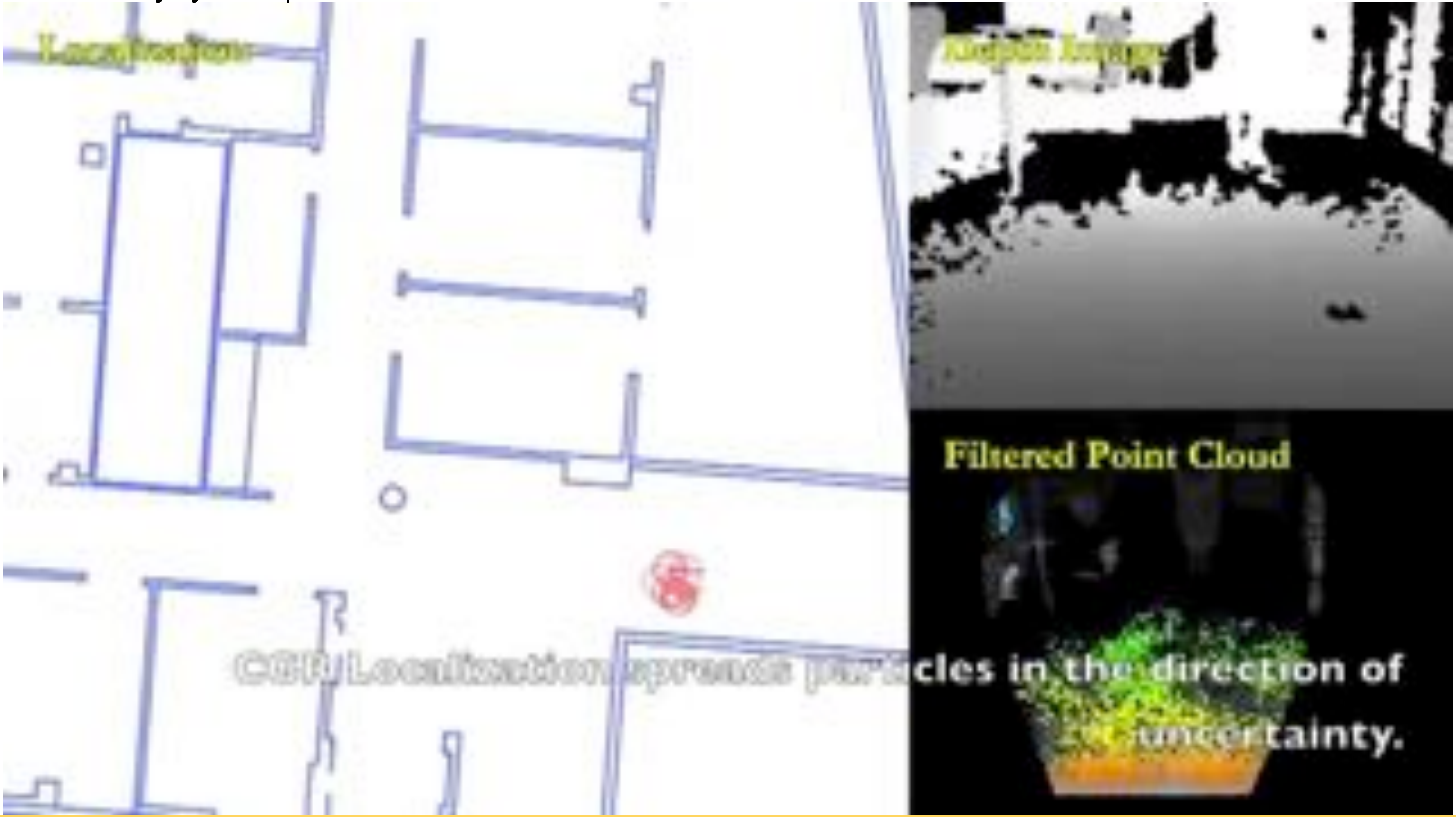
CMDragons 2015 slow-motion multi-pass goal http://www.cs.cmu.edu/~coral/projects/cobot/

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

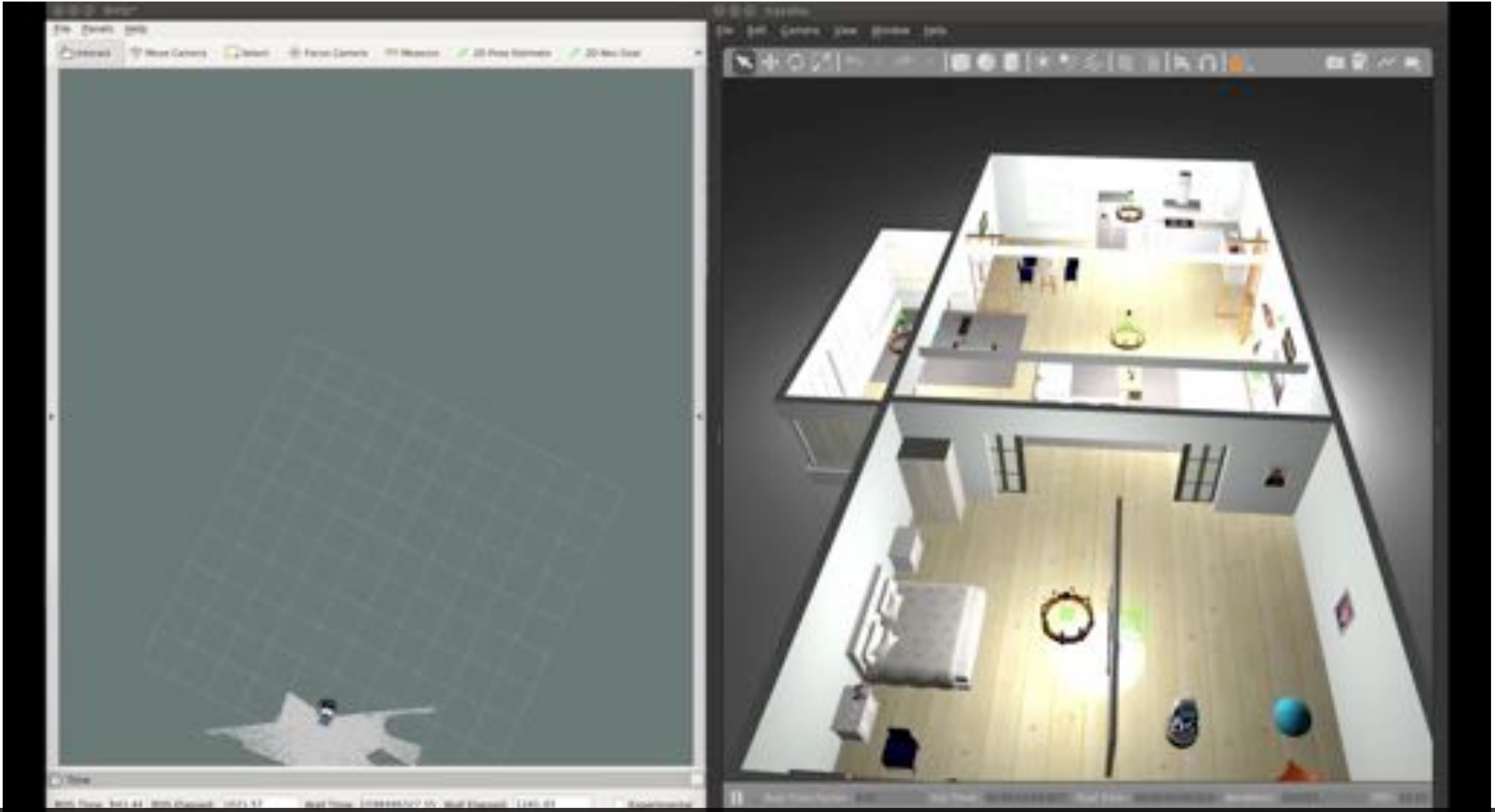


over 1,000km navigated around CMU

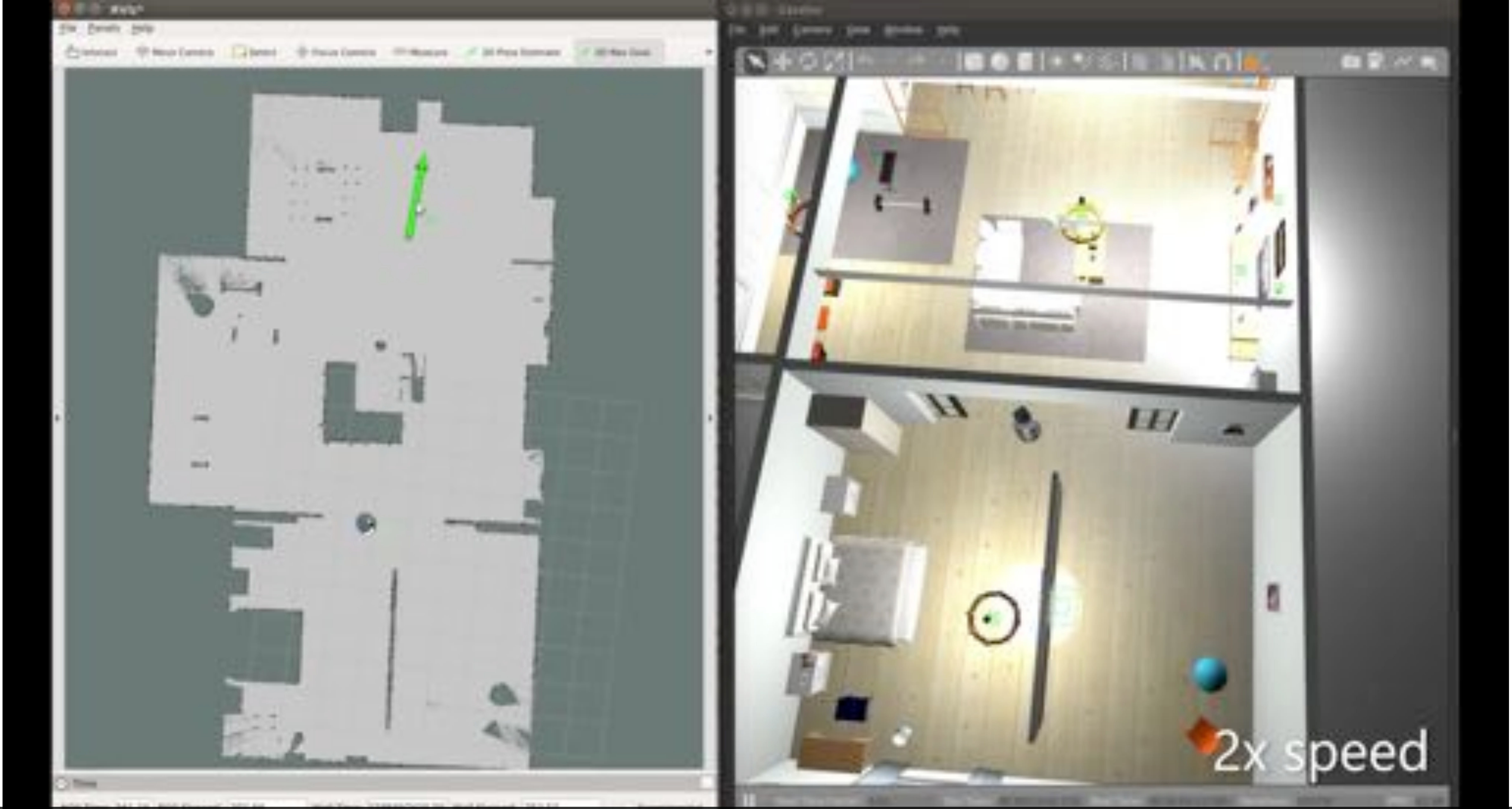
https://www.joydeepb.com/research.html



Slide borrowed from Michigan Robotics autorob.org



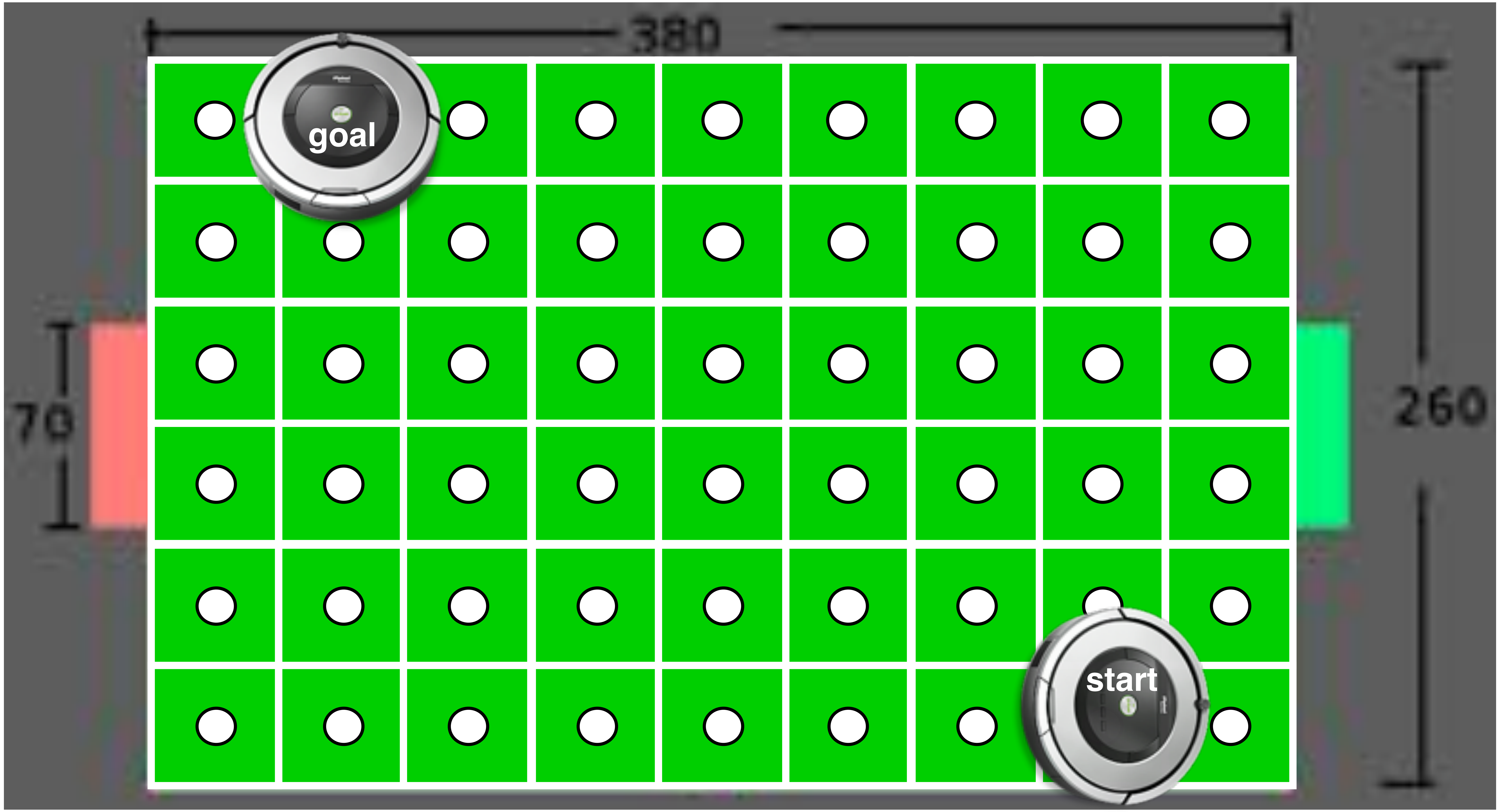
Localization and Mapping - Alphonsus Adu-Bredu - https://youtu.be/wH0QhWgtmuA



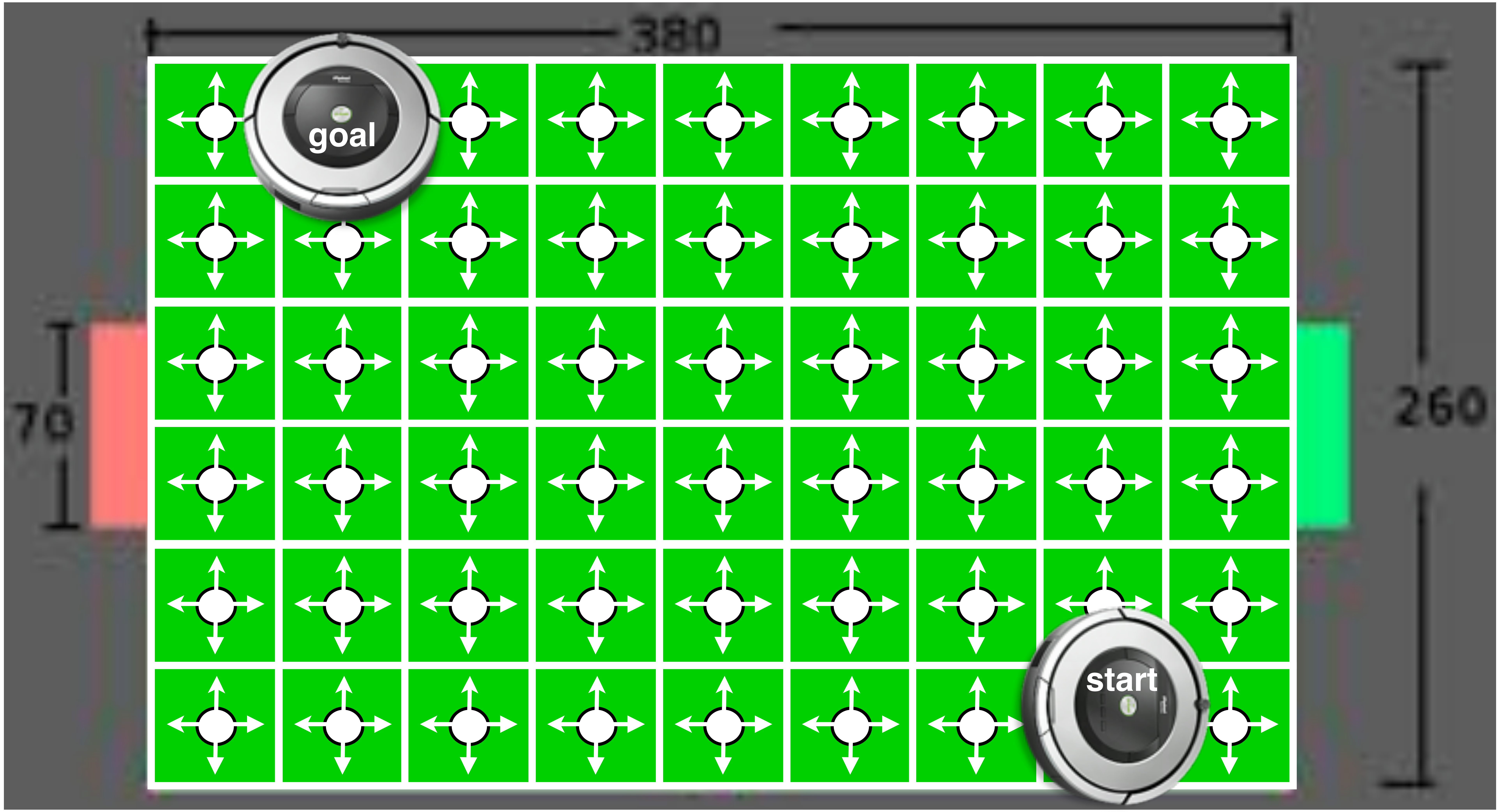
Autonomous Navigation - Alphonsus Adu-Bredu - https://youtu.be/wH0QhWgtmuA

How do we get from A to B?

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



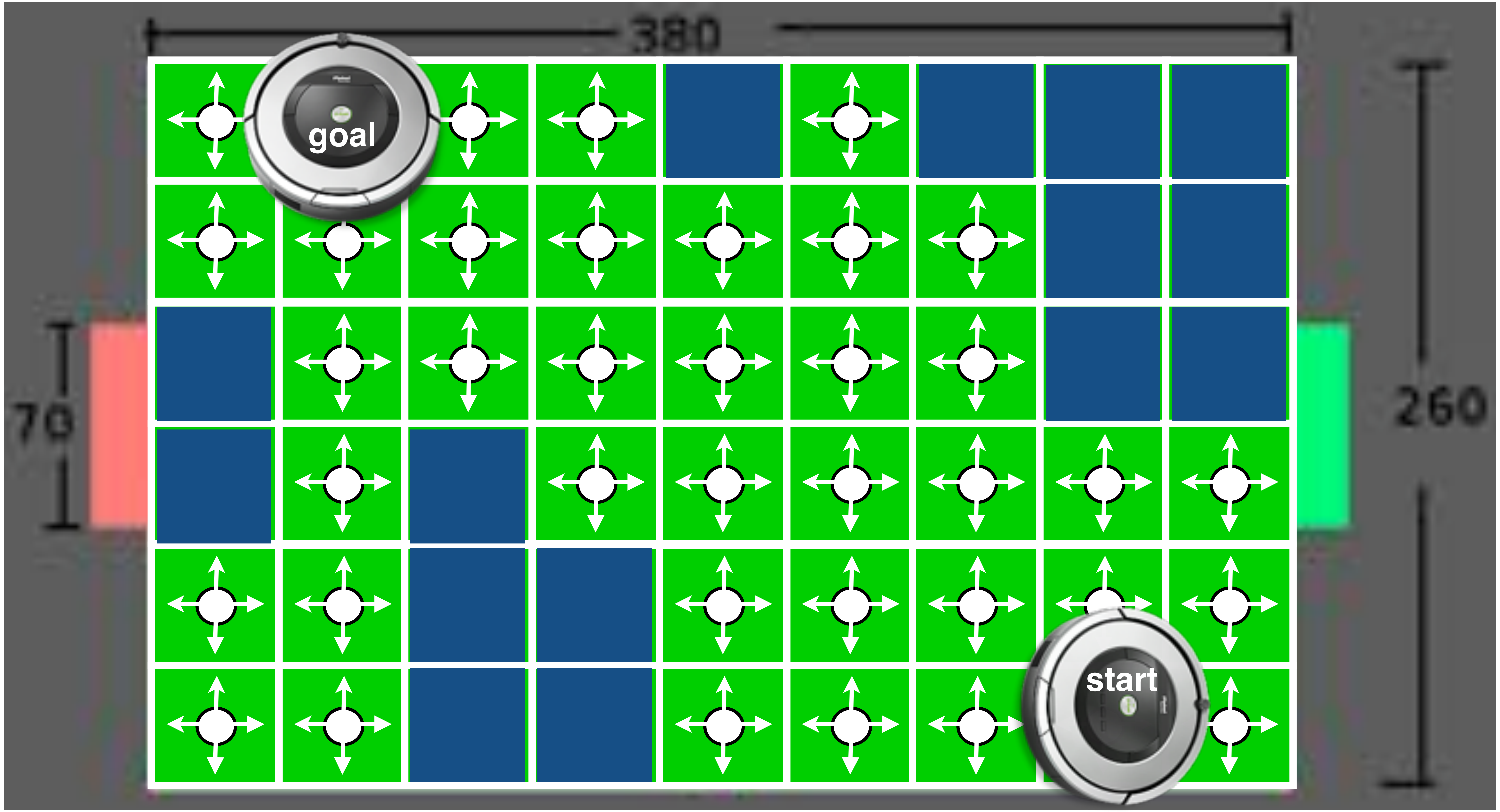
Consider all possible poses as uniformly distributed array of cells in a graph
Edges connect adjacent cells, weighted by distance



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

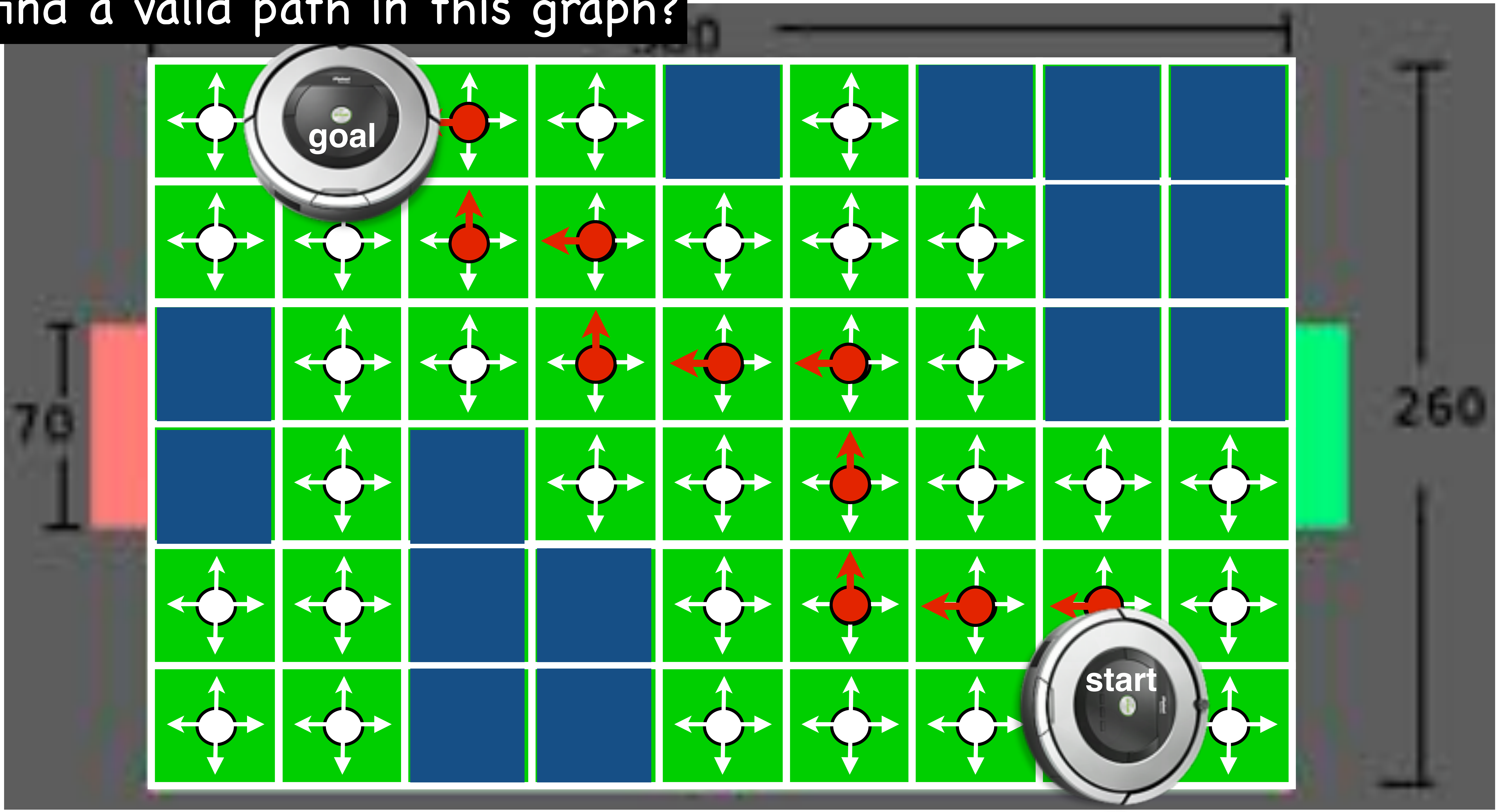


Consider all possible poses as uniformly distributed array of cells in a graph

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?



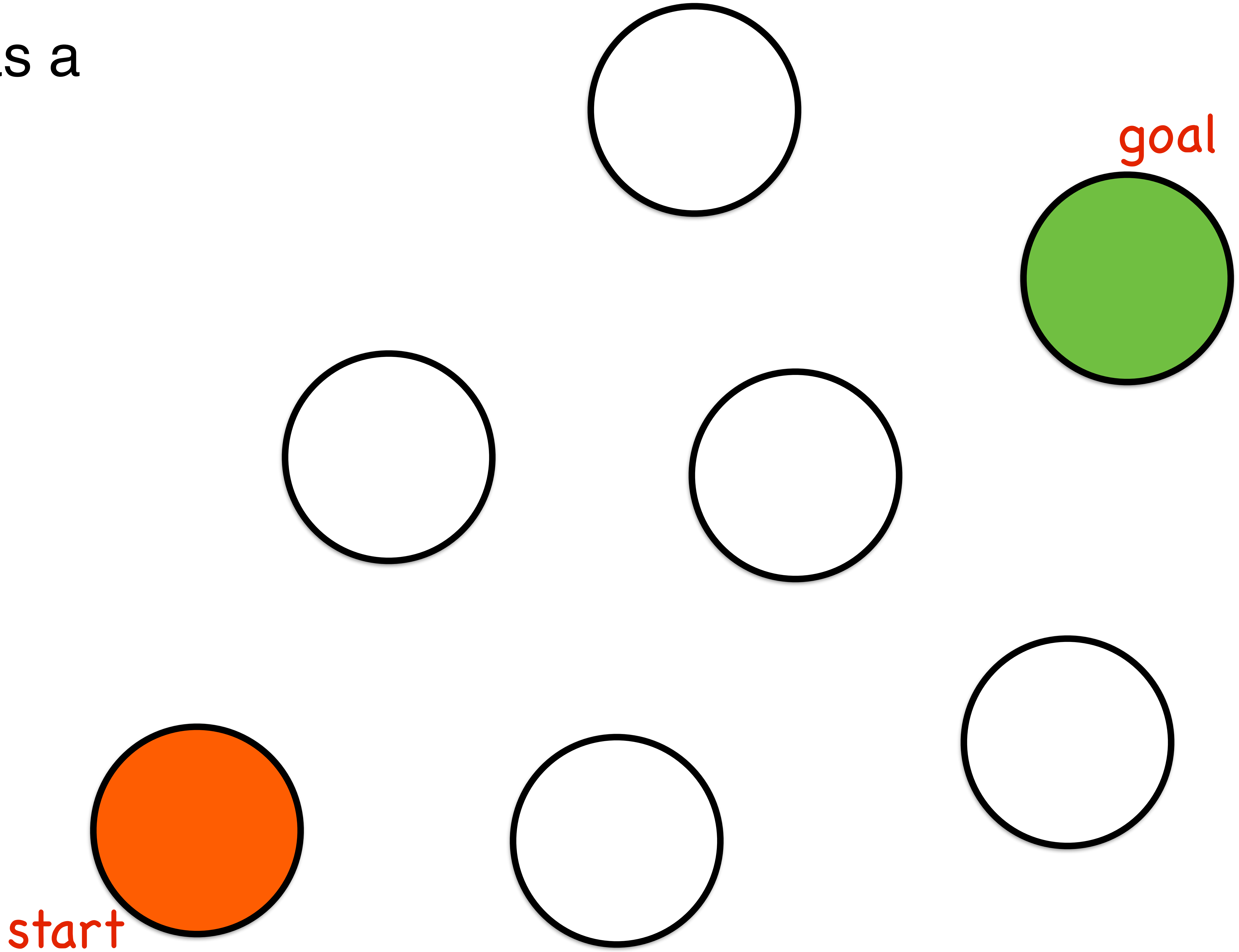
Approaches to motion planning

- 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

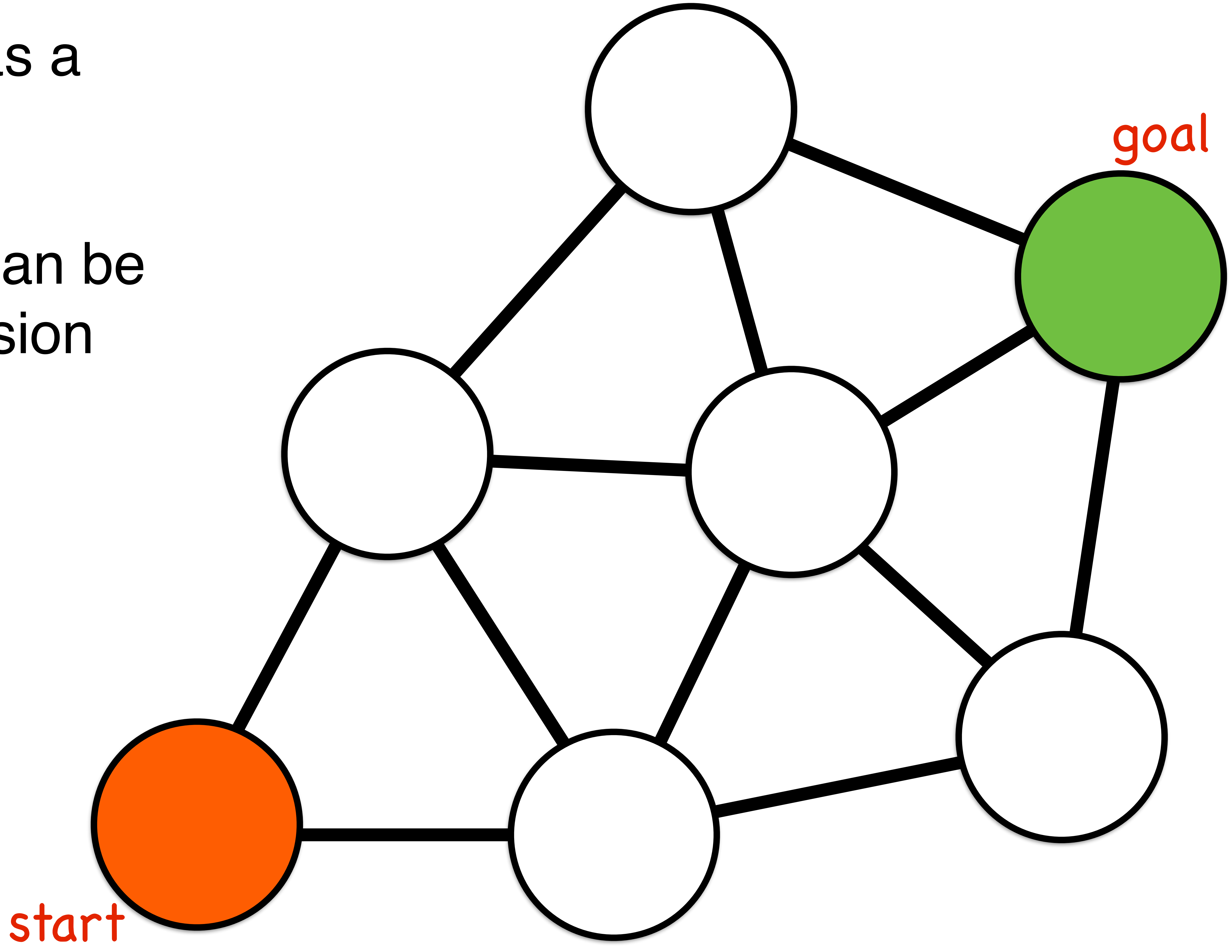
Consider each possible robot pose as a node V_i in a graph $G(V,E)$



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

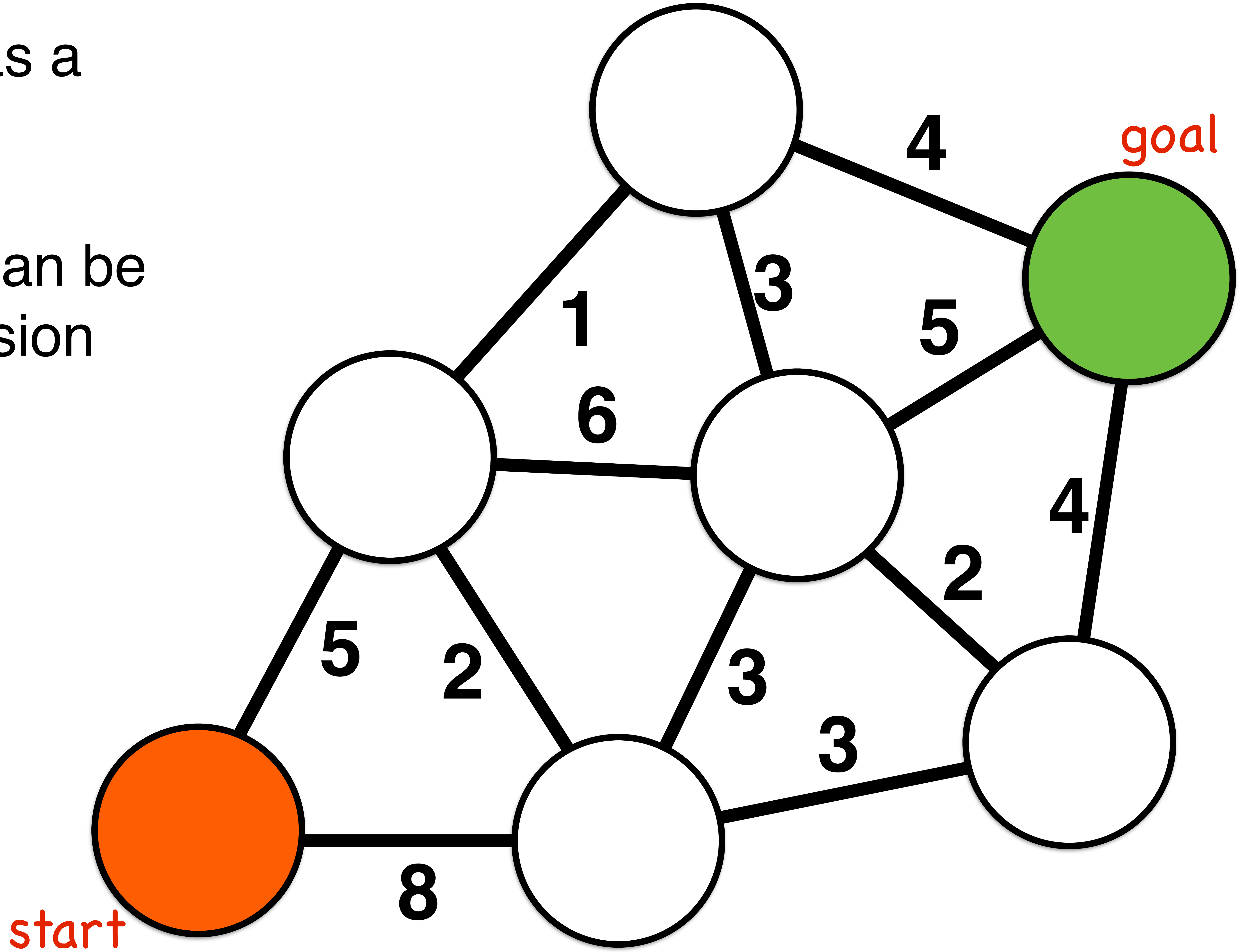


Consider a simple search graph

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



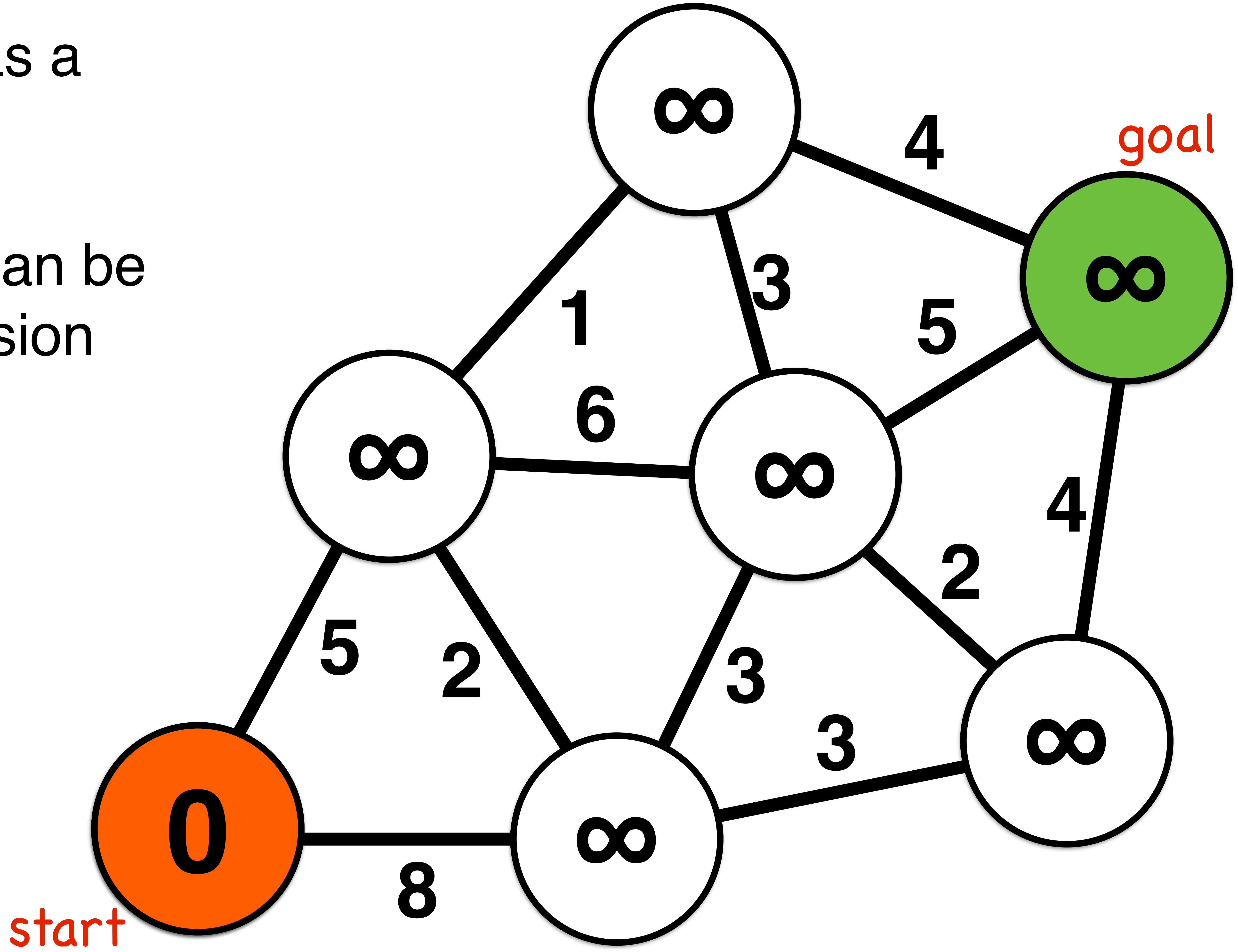
Consider a simple search graph

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

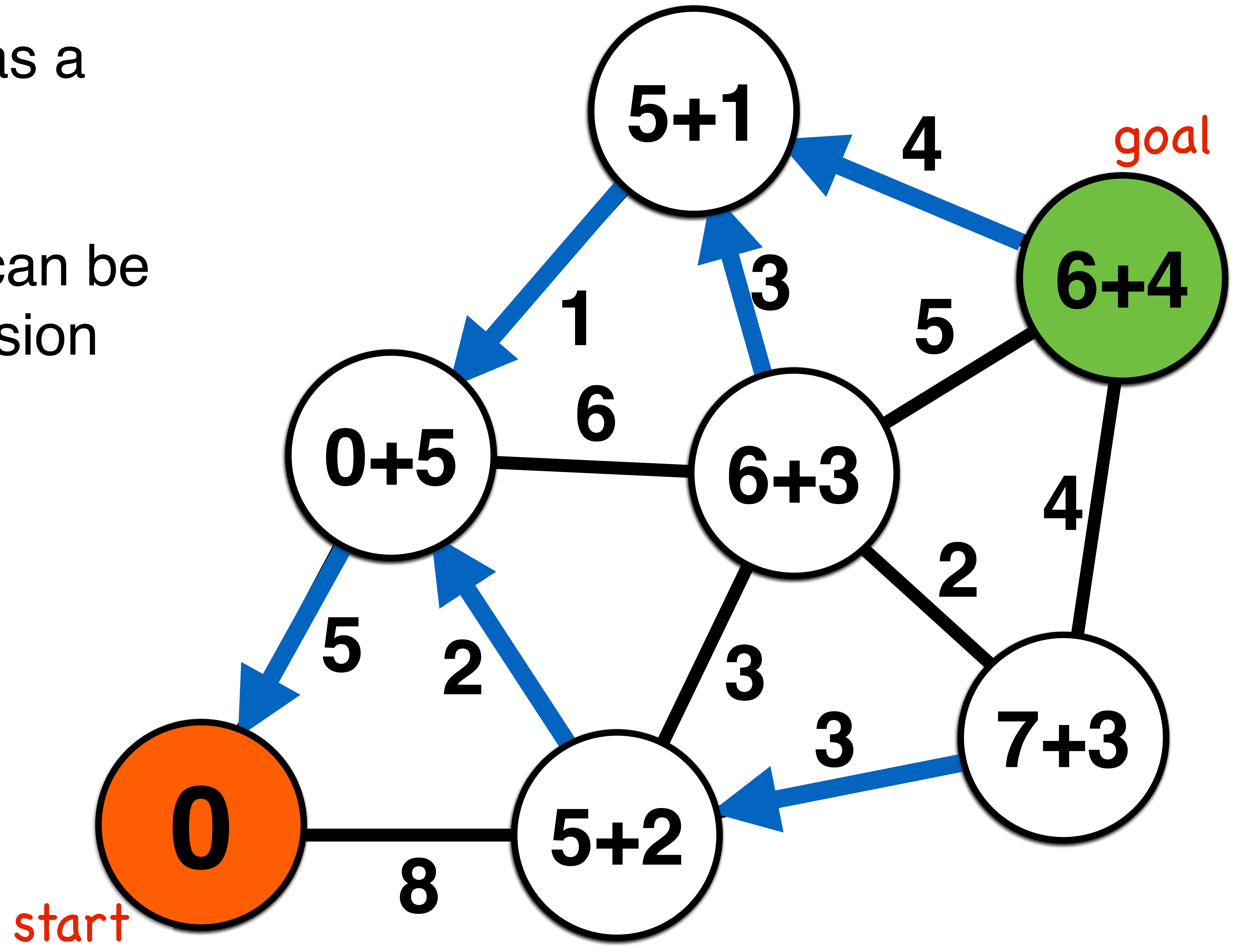
Consider each possible robot pose as a node V_i in a graph $G(V,E)$

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

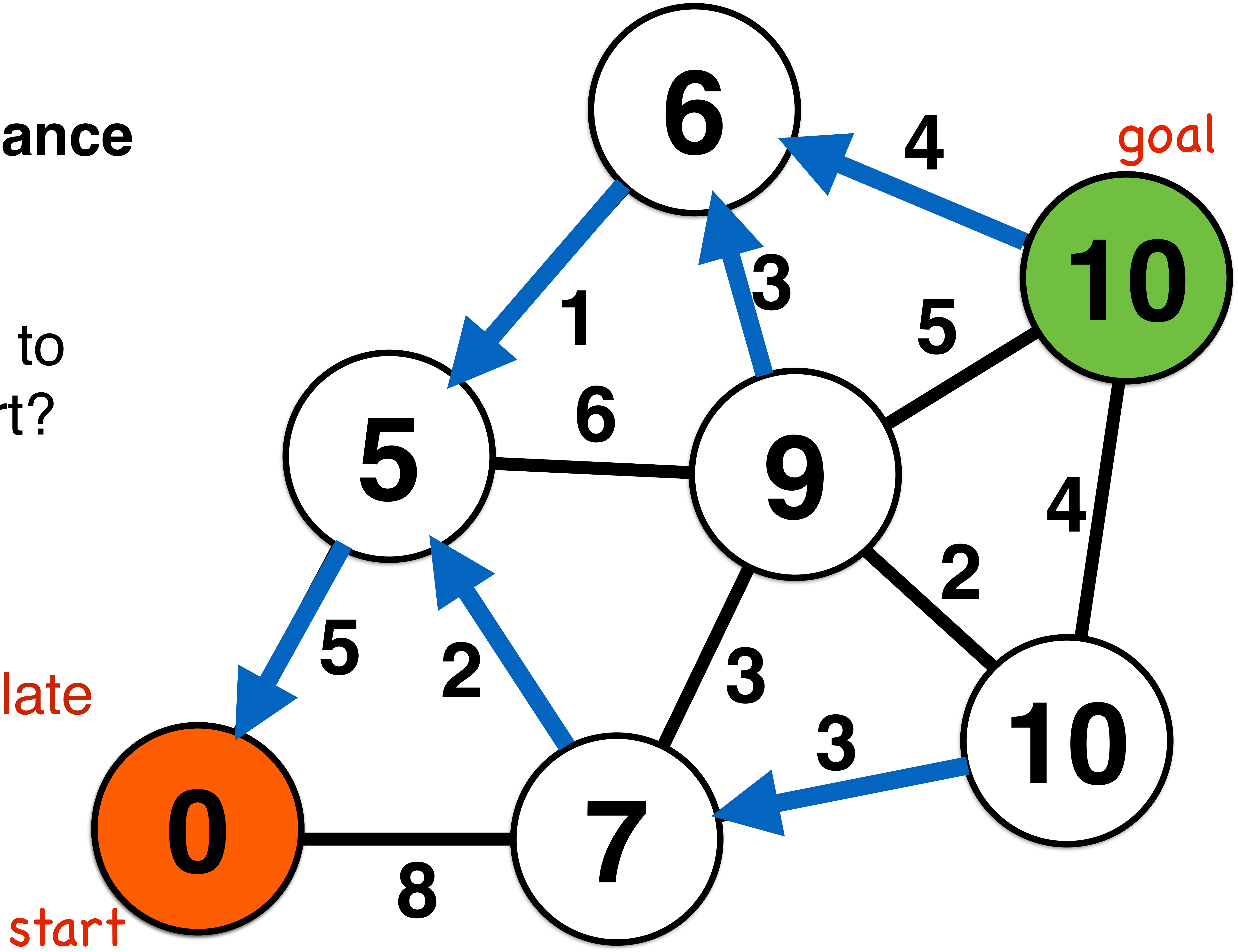


Path Planning as Graph Search

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

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

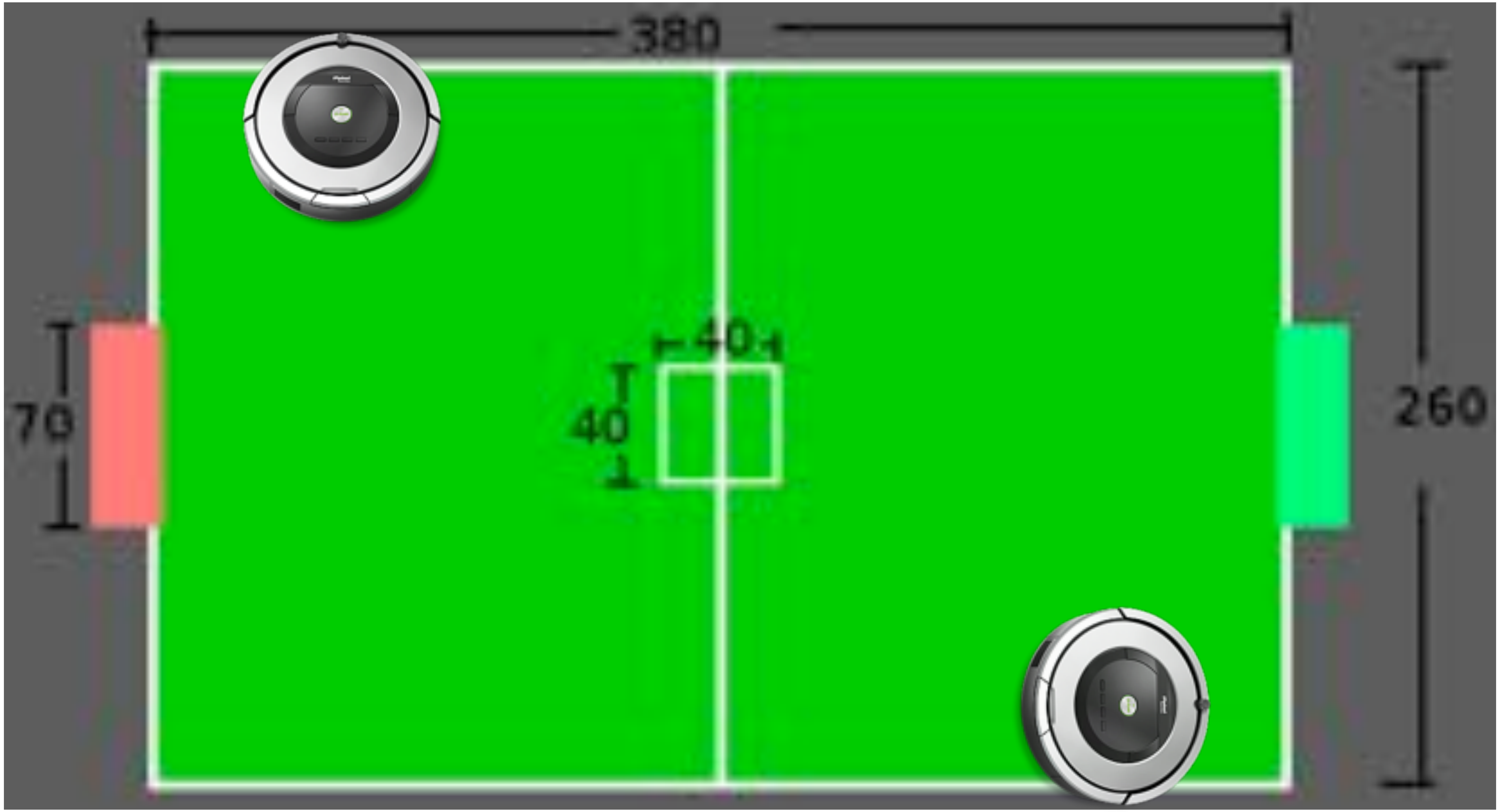
We will use a single algorithm template for our graph search computation



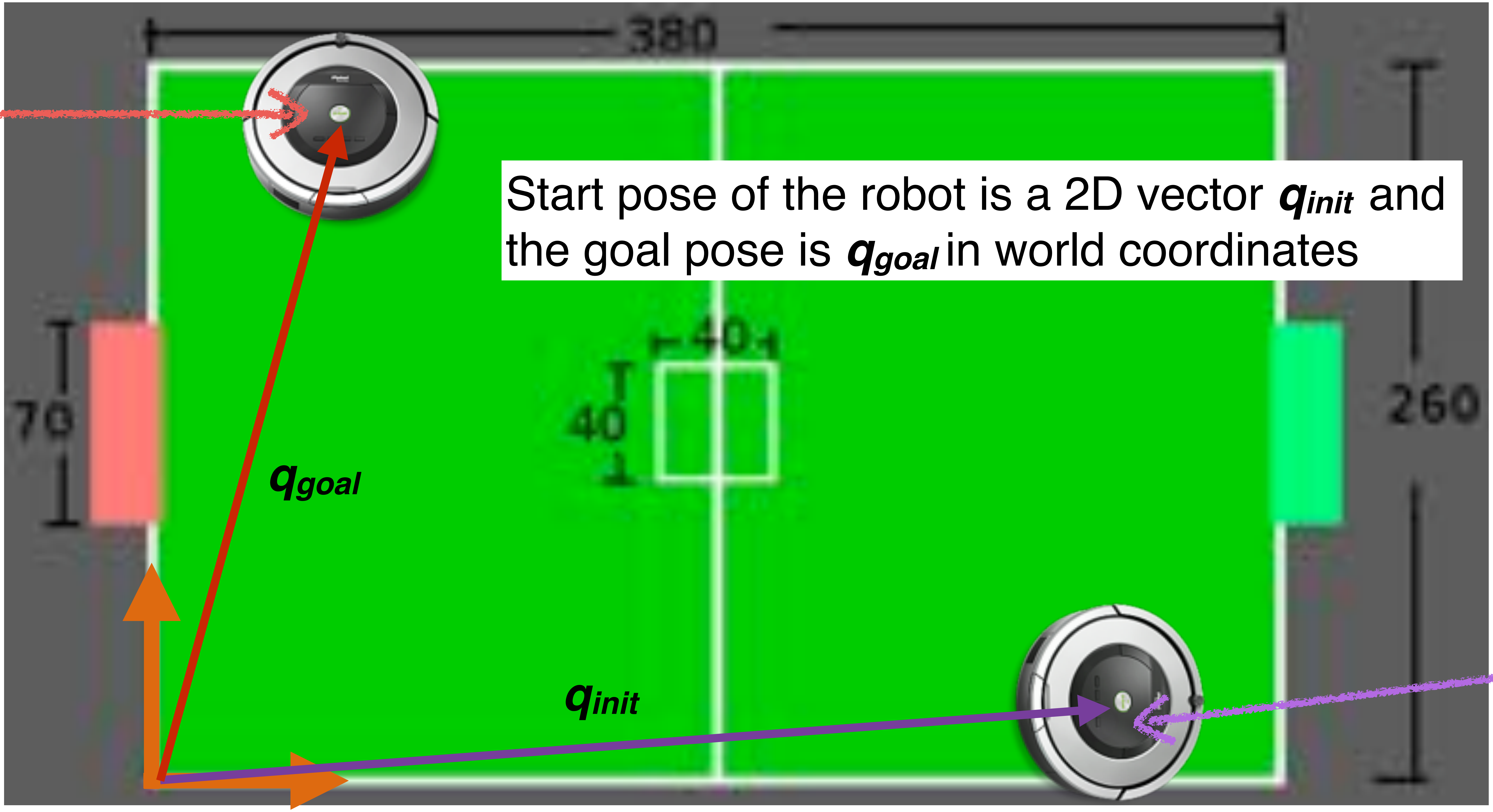
Depth-first search

intuition and walkthrough

Depth-first search

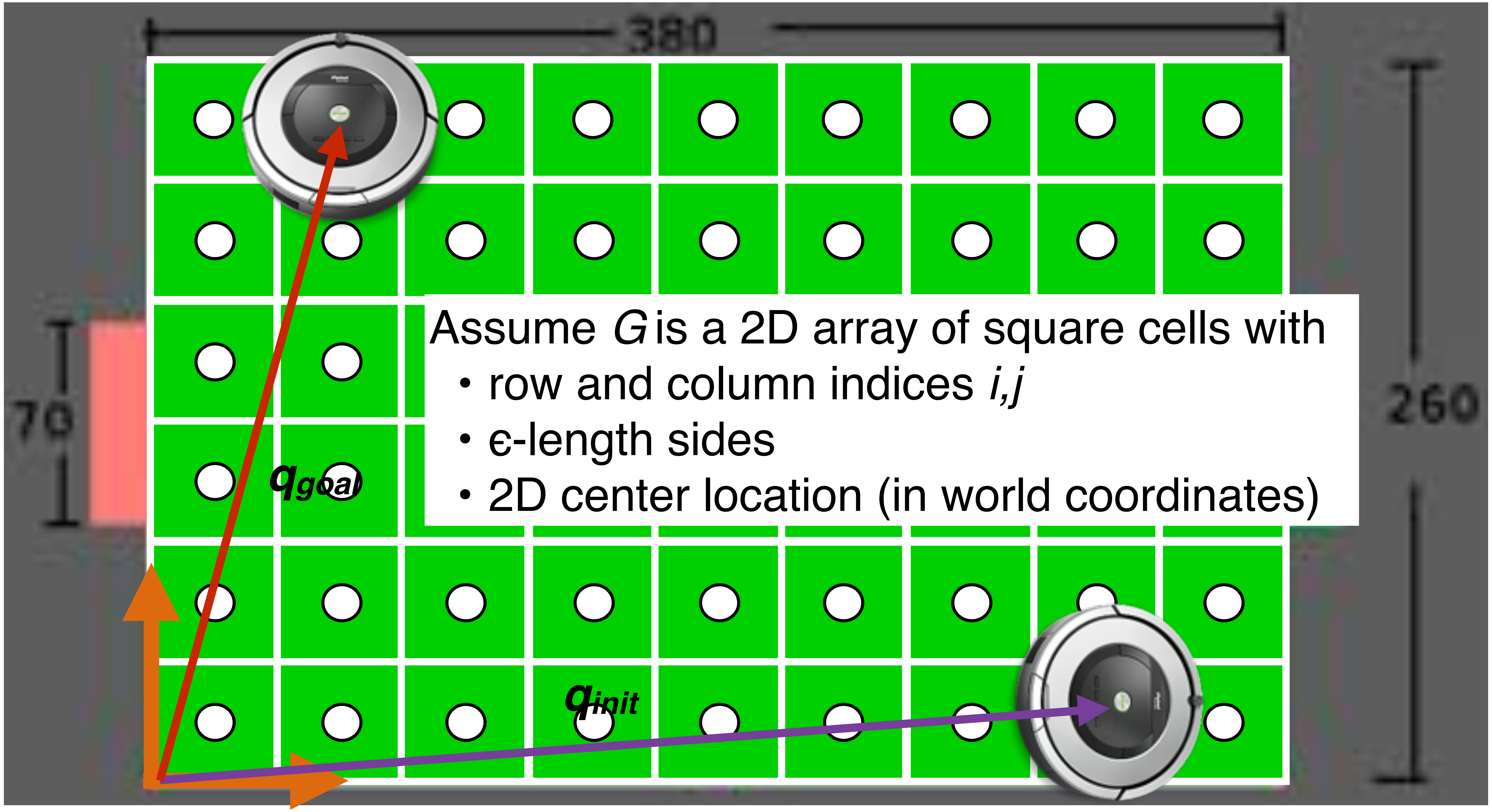


Depth-first search



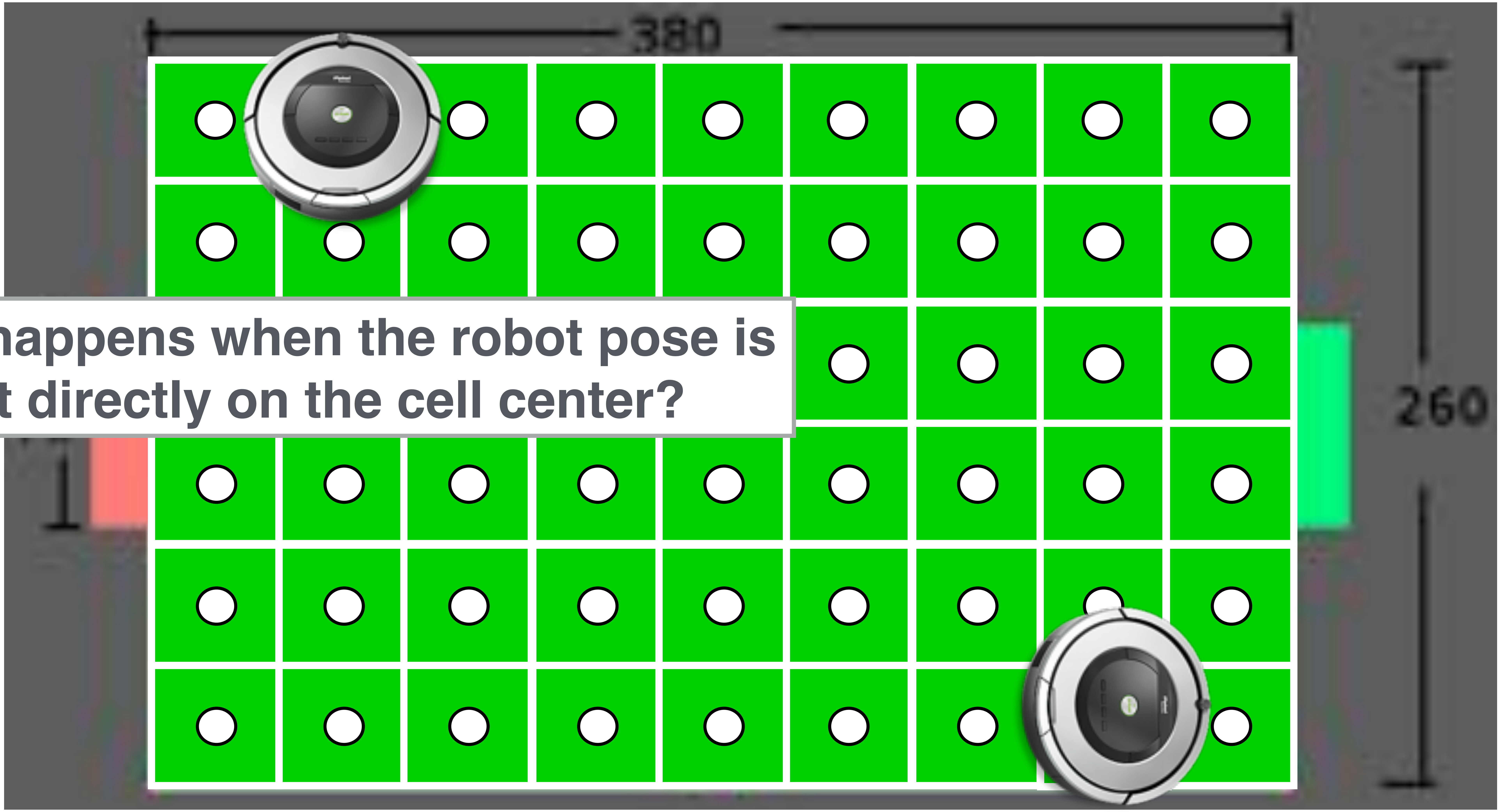
Start pose of the robot is a 2D vector q_{init} and the goal pose is q_{goal} in world coordinates

Depth-first search



Assume *G* is a 2D array of square cells with
- row and column indices *i,j*
- ϵ-length sides
- 2D center location (in world coordinates)

Depth-first search



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

Graph Accessibility

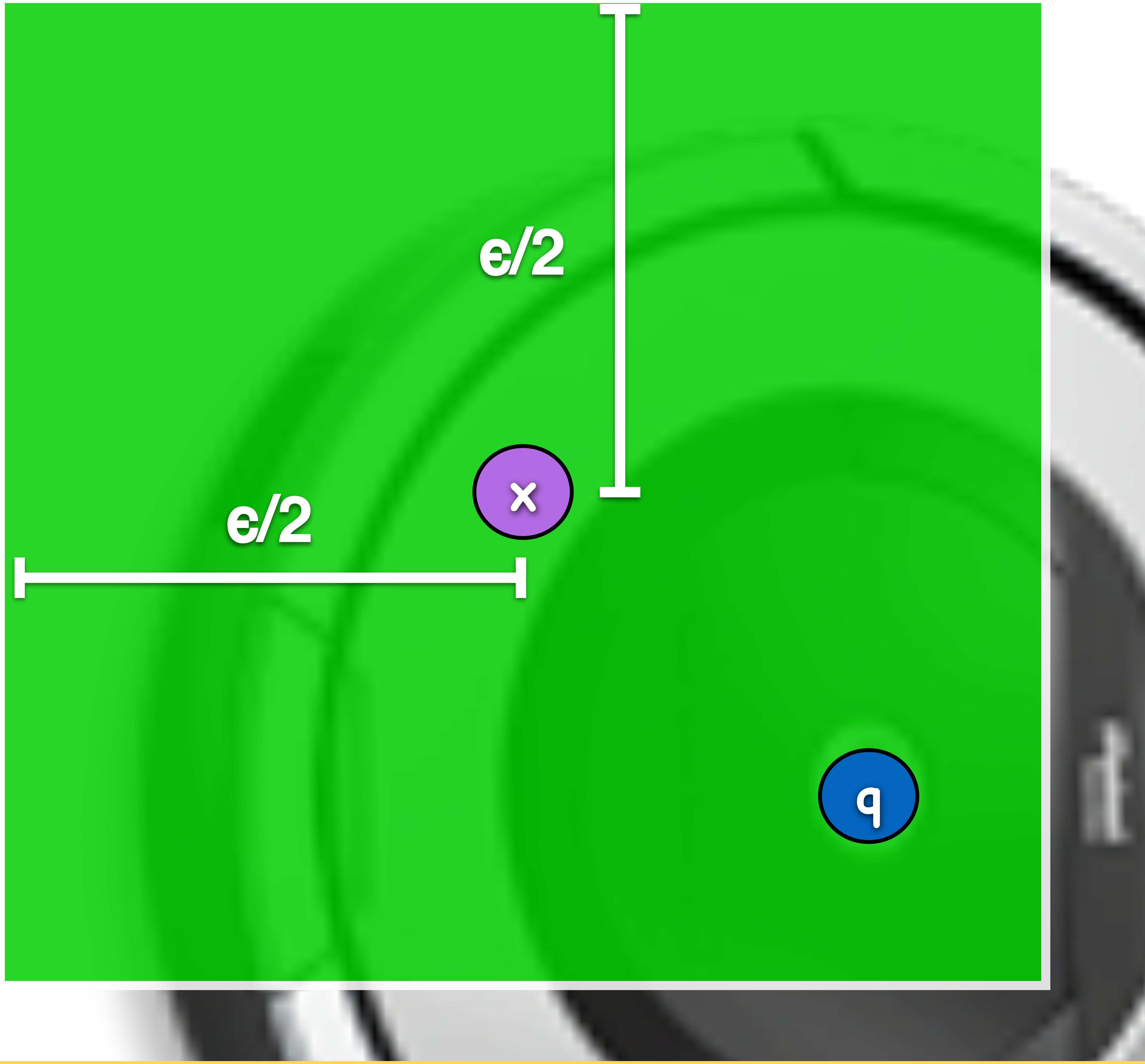
What happens when the robot pose is not directly on the cell center?

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

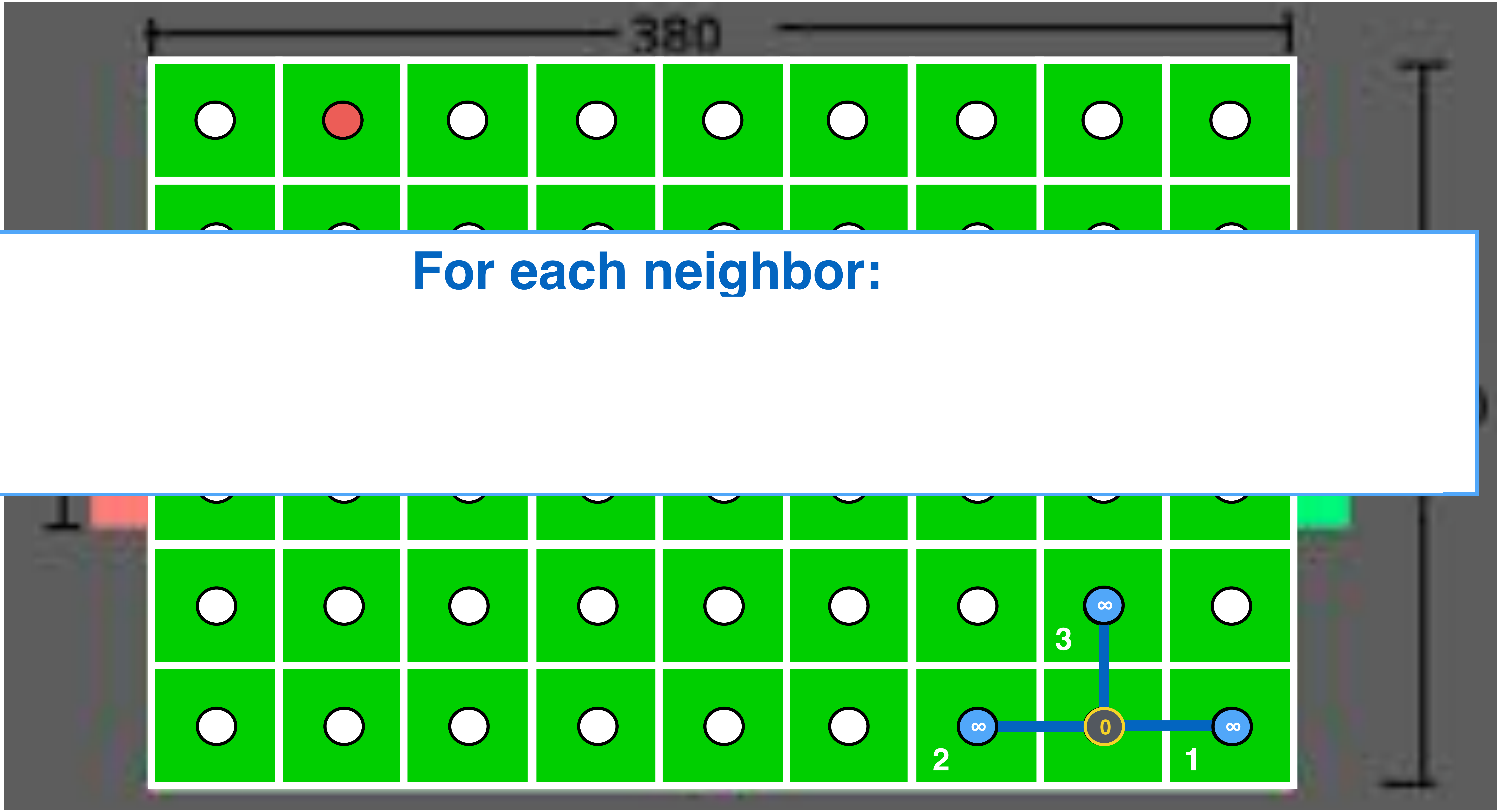
Depth-first search

Depth-first search

Slide borrowed from Michigan Robotics autorob.org

Depth-first search

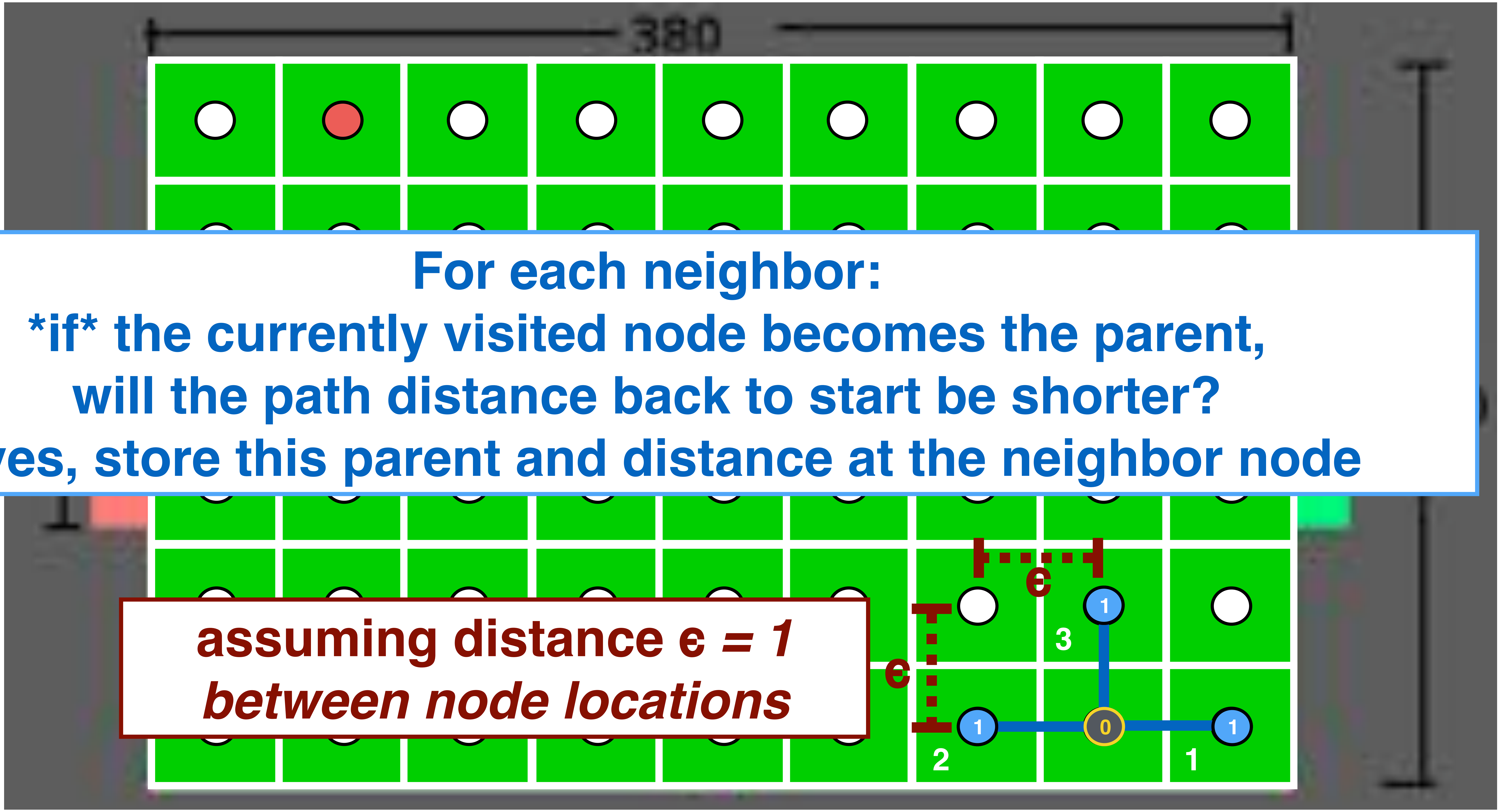
Depth-first search



For each neighbor:

Slide borrowed from Michigan Robotics autorob.org

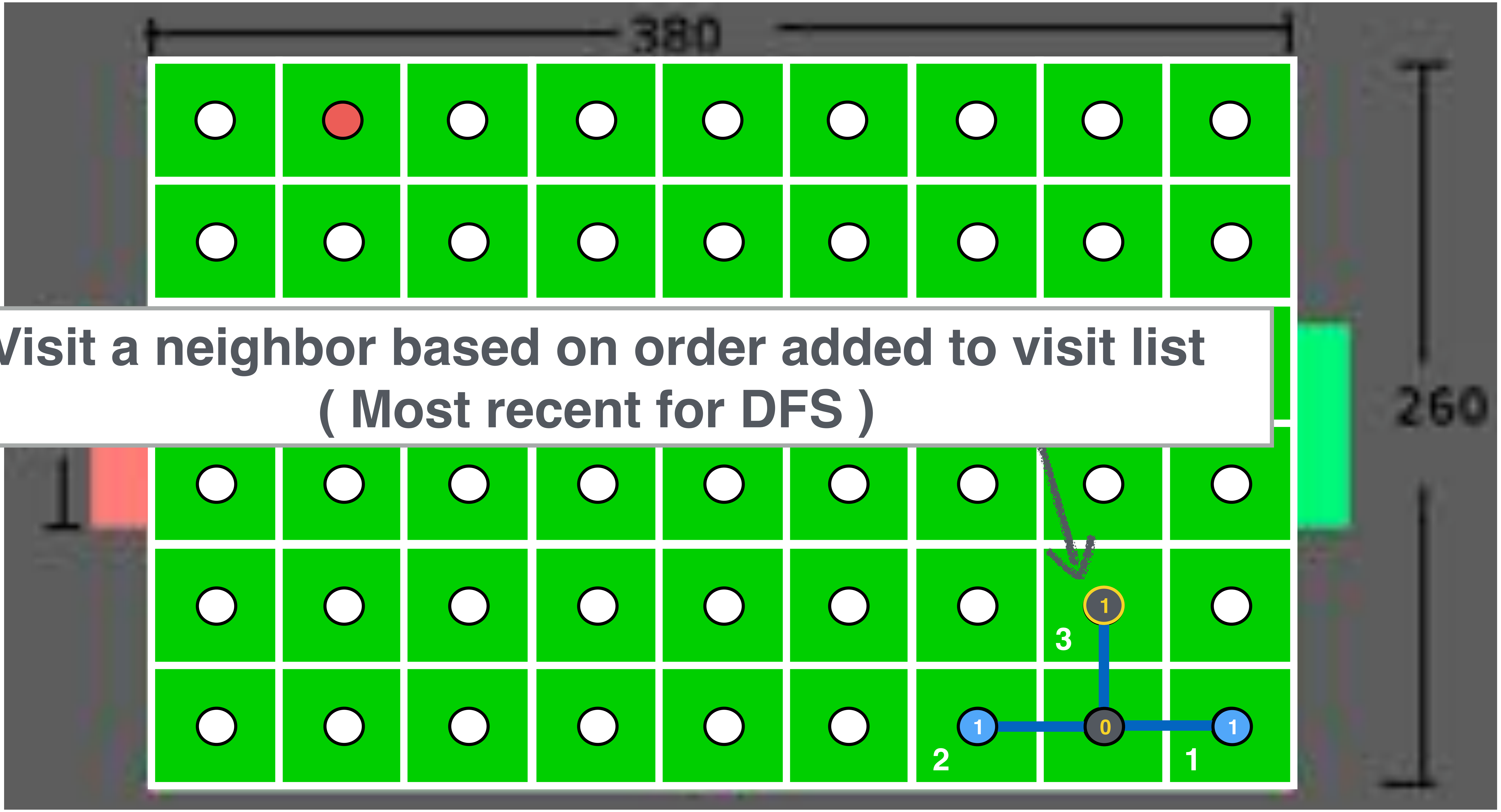
Depth-first search



For each neighbor:
***if* the currently visited node becomes the parent,**
will the path distance back to start be shorter?
if yes, store this parent and distance at the neighbor node

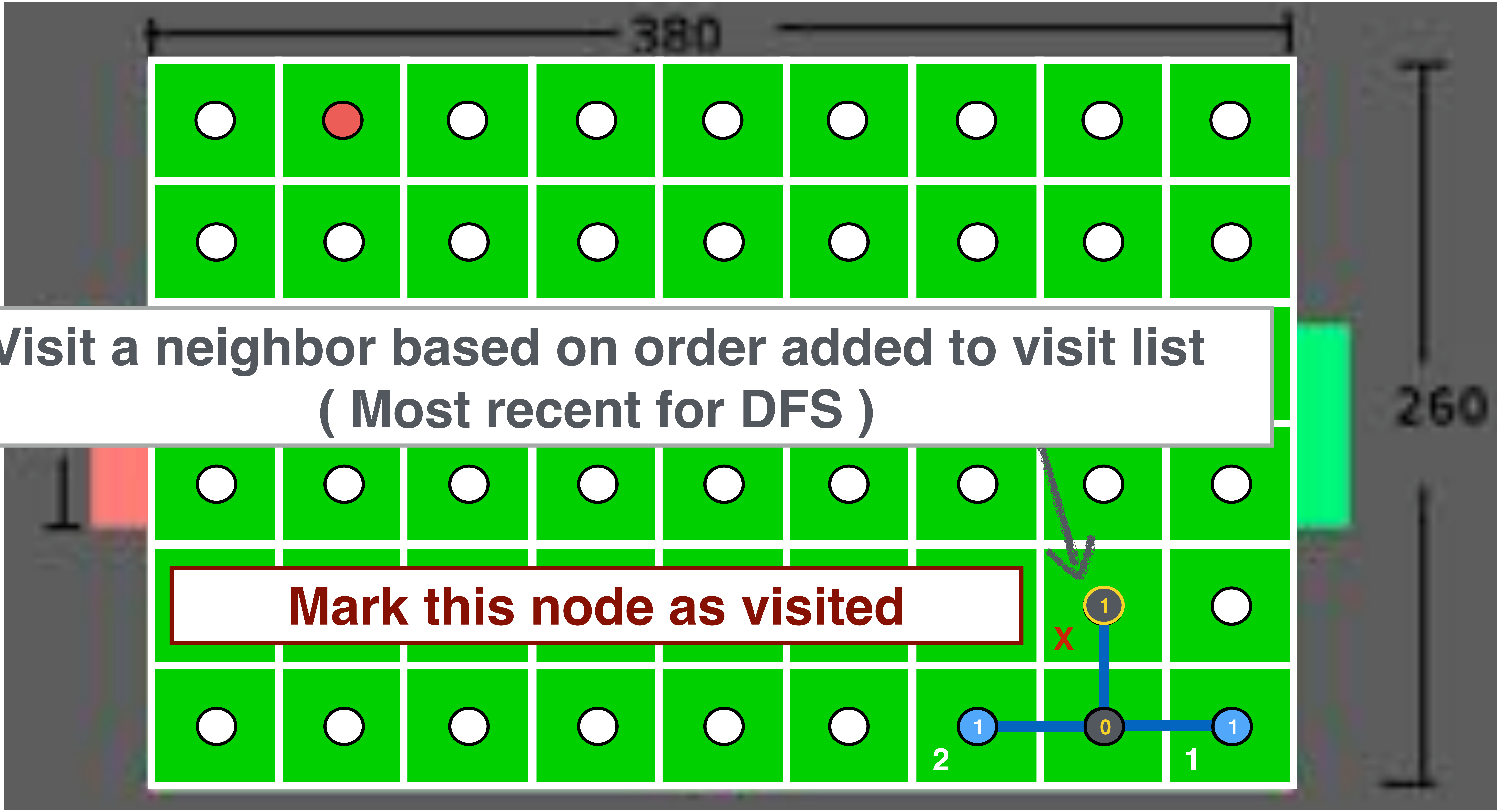
assuming distance ϵ = 1
between node locations

Depth-first search



Visit a neighbor based on order added to visit list
(Most recent for DFS)

Depth-first search

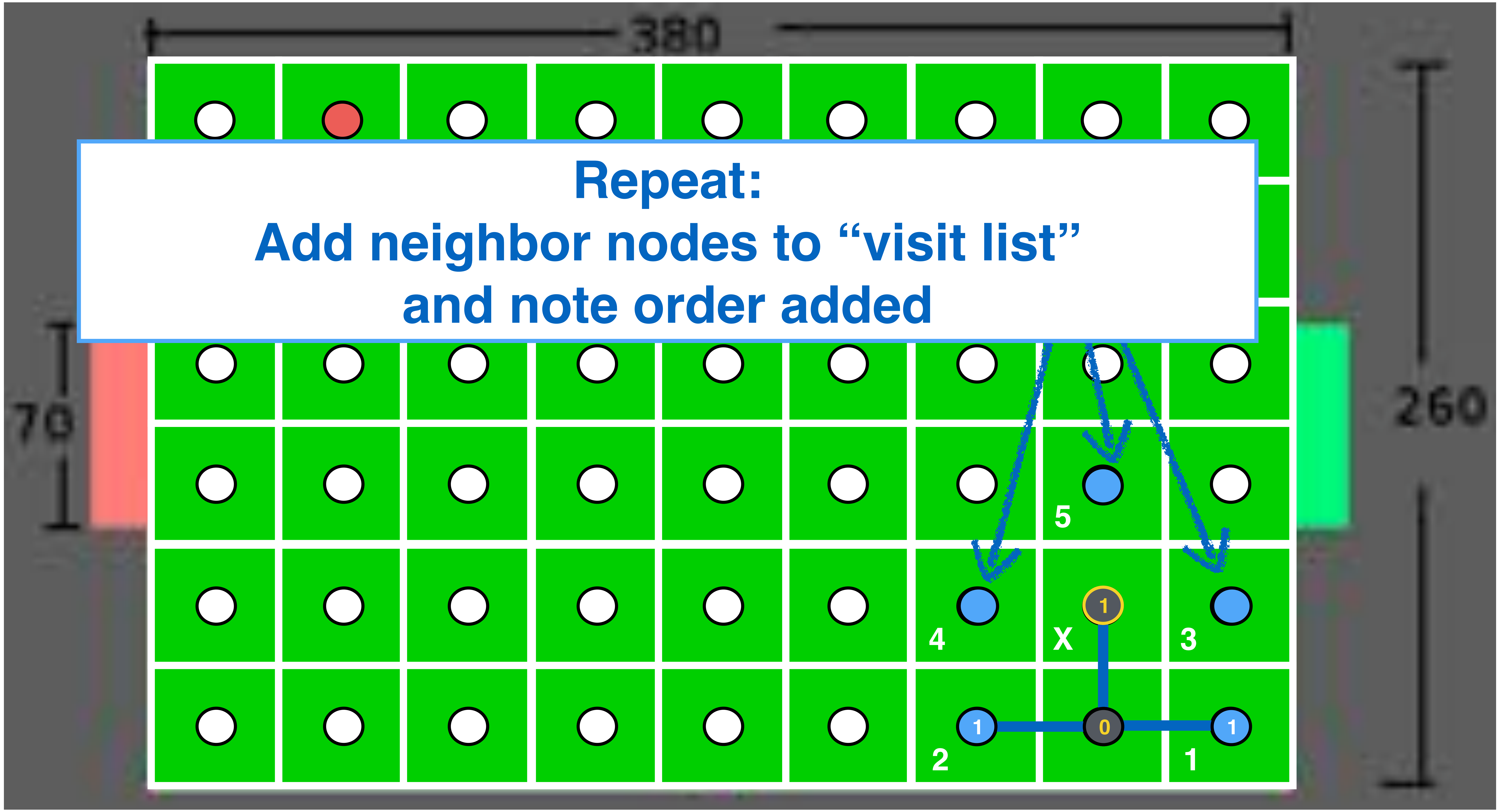


Visit a neighbor based on order added to visit list
(Most recent for DFS)

Mark this node as visited

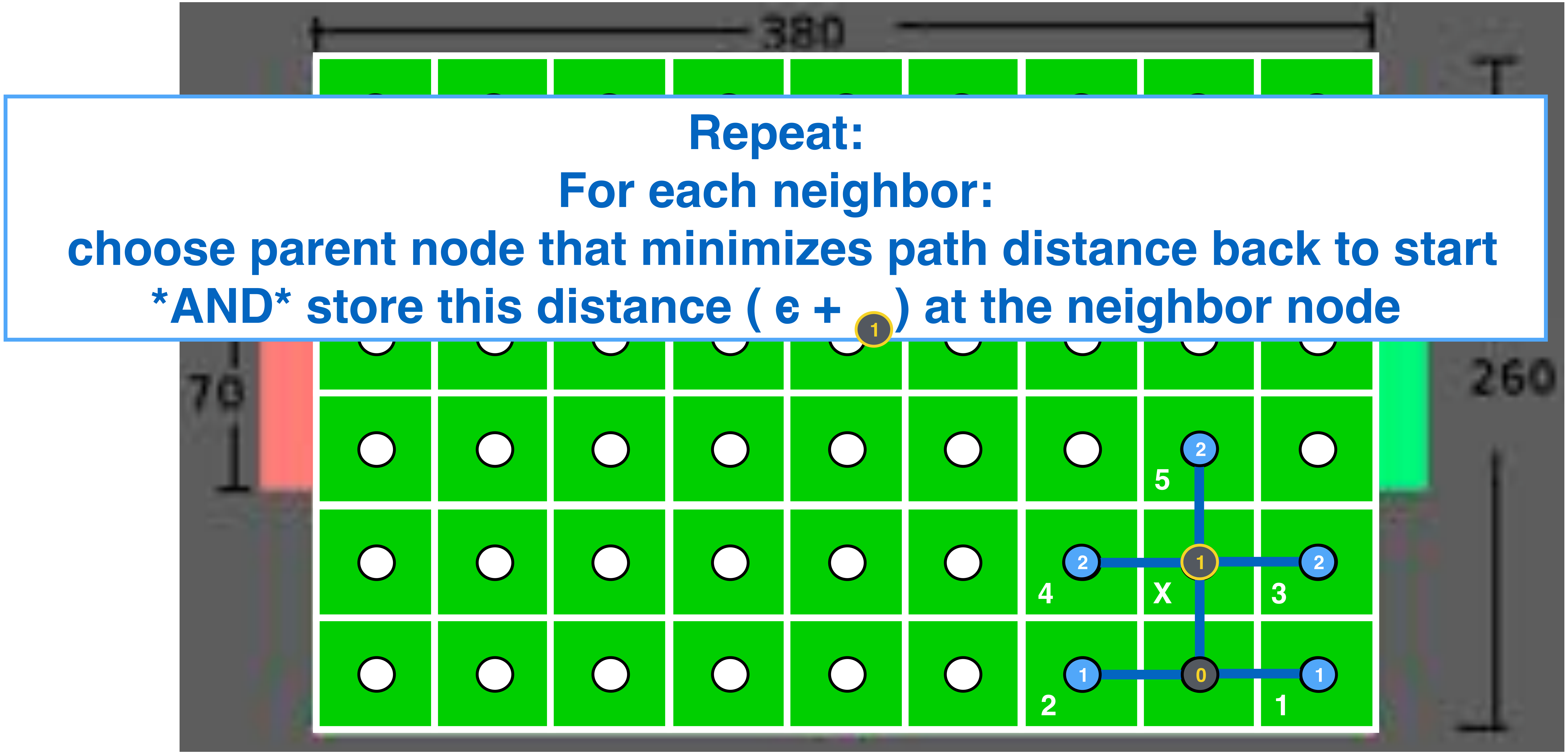
Slide borrowed from Michigan Robotics autorob.org

Depth-first search

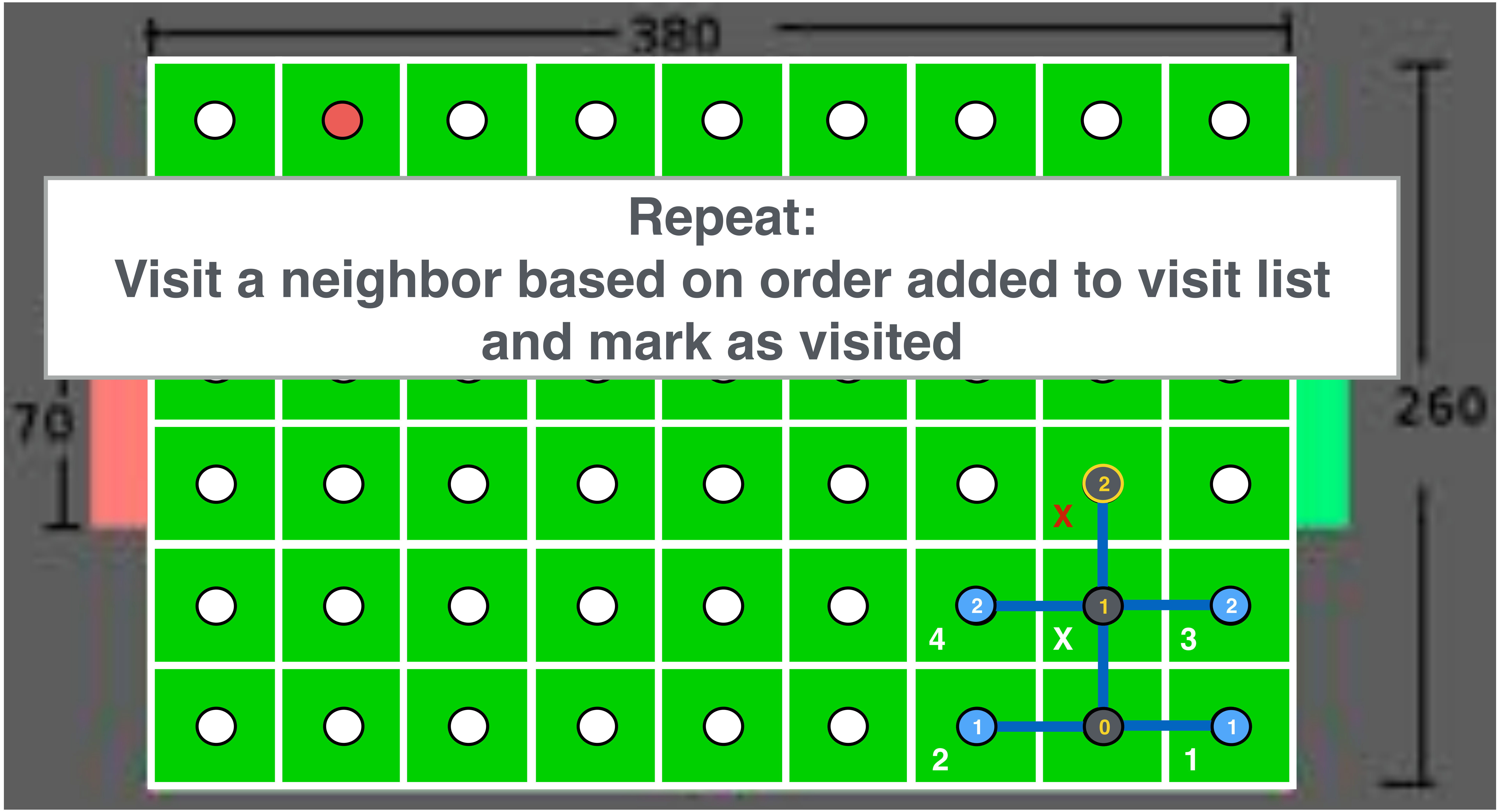


Repeat:
Add neighbor nodes to "visit list" and note order added

Depth-first search

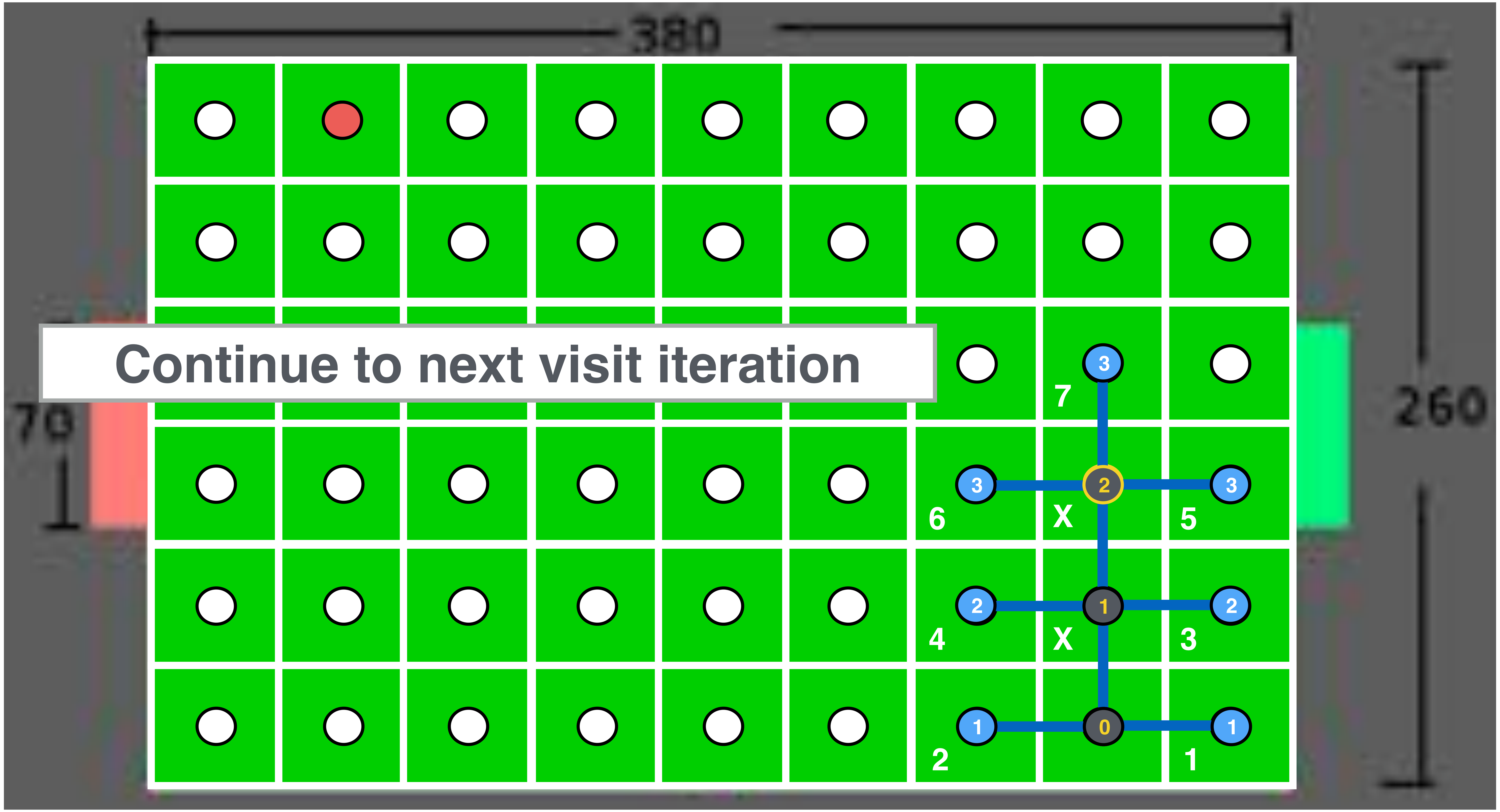


Depth-first search



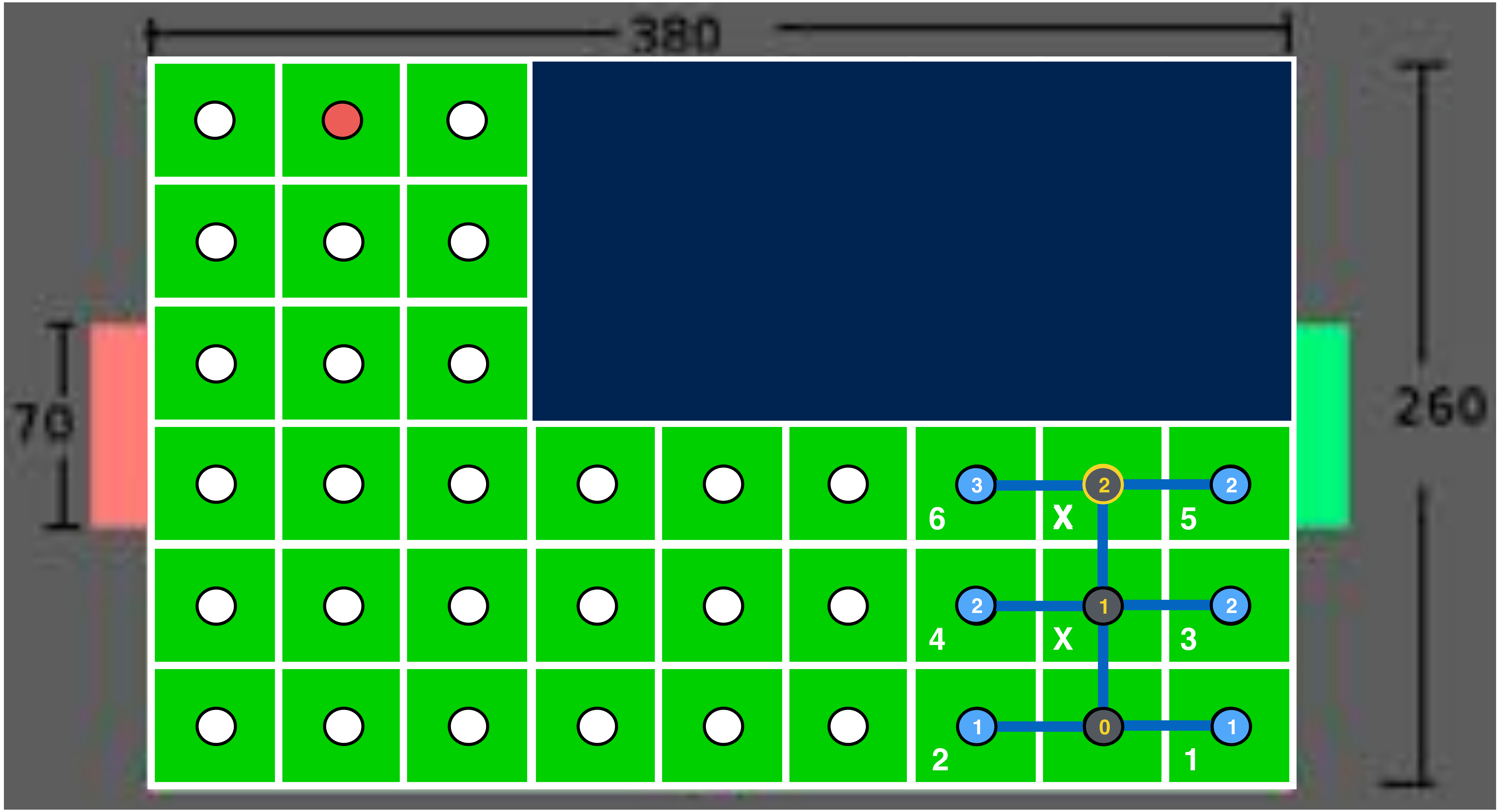
Repeat:
Visit a neighbor based on order added to visit list and mark as visited

Depth-first search

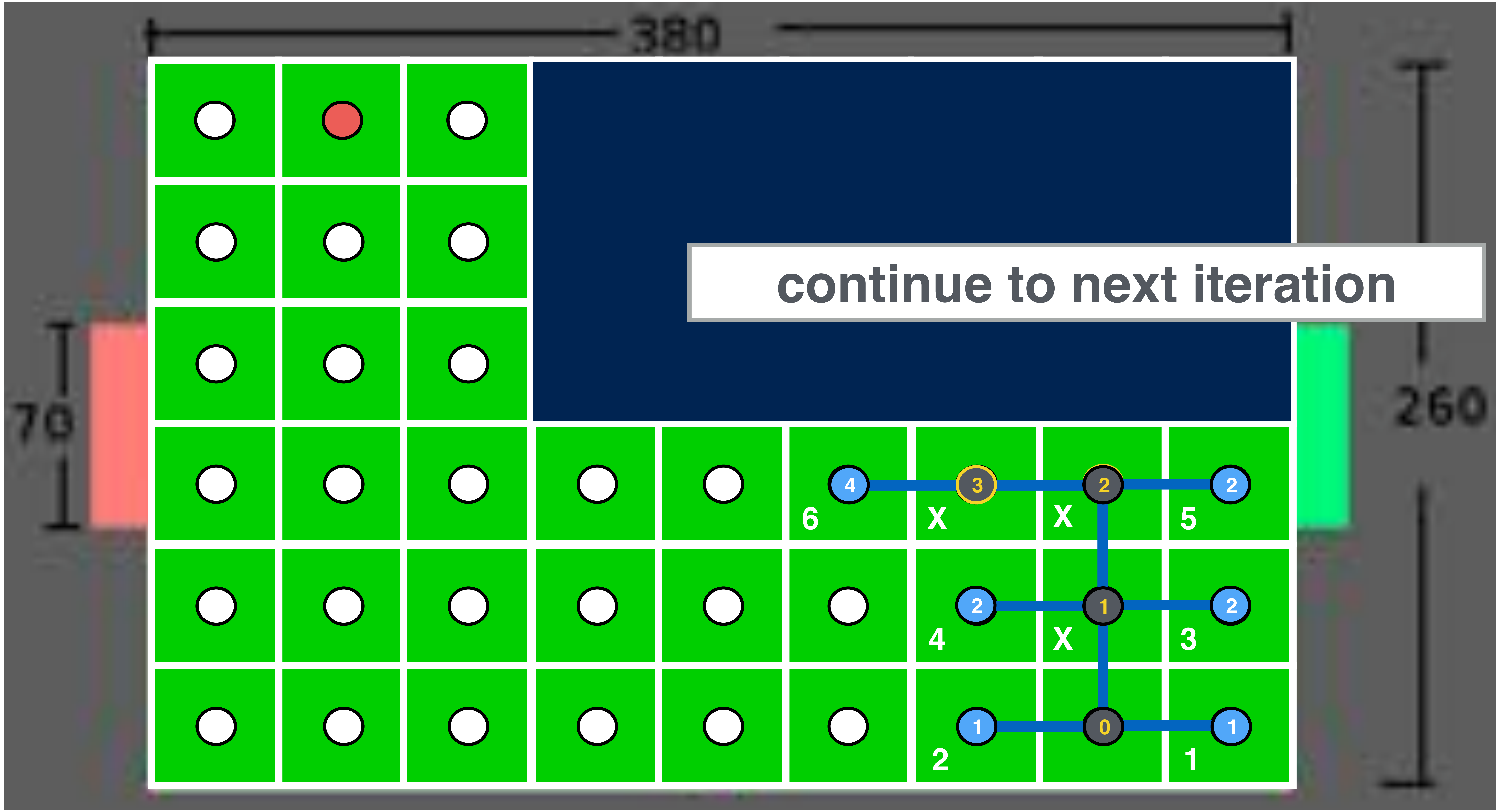


Depth-first search

Depth-first search

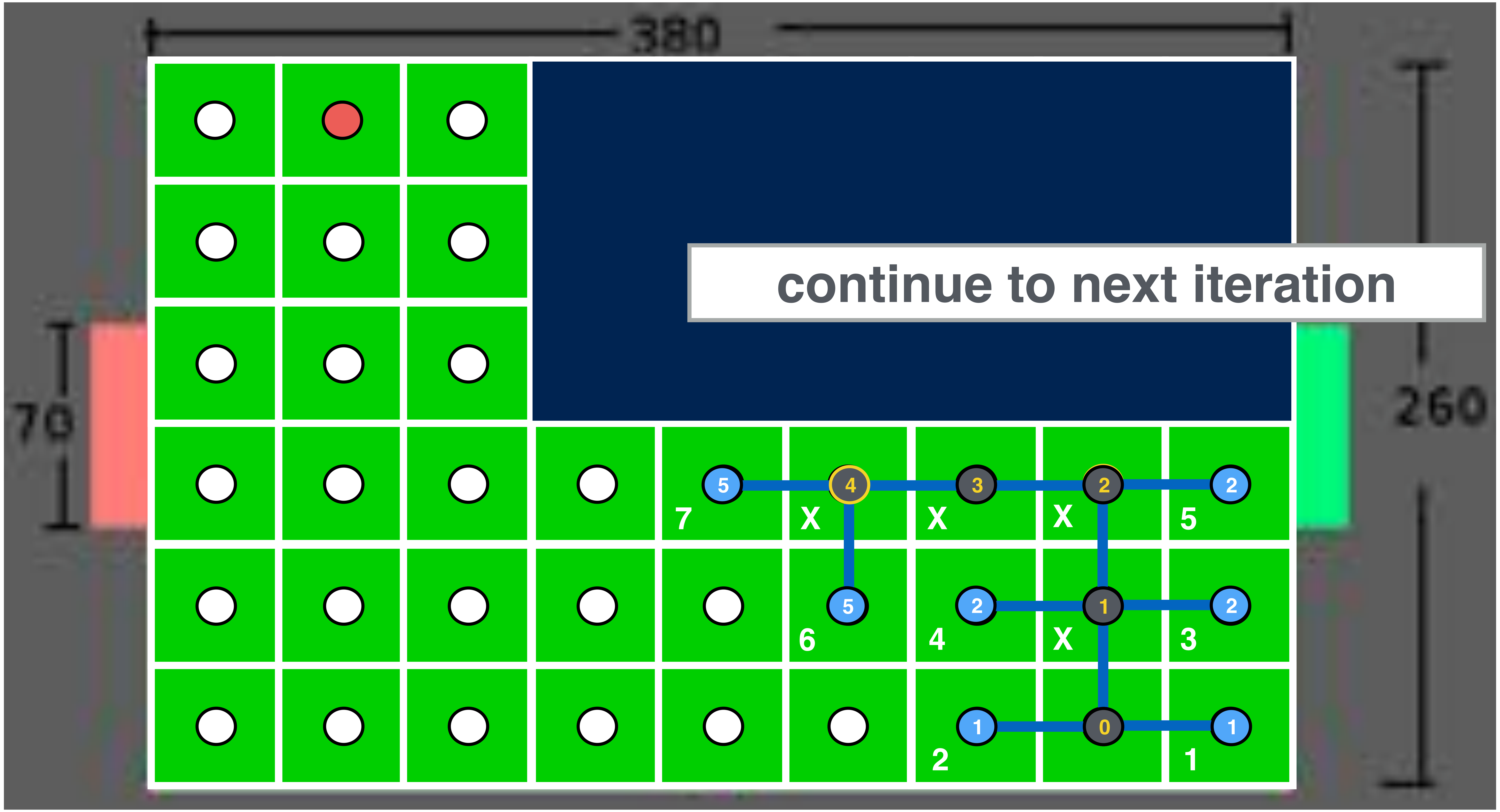


Depth-first search

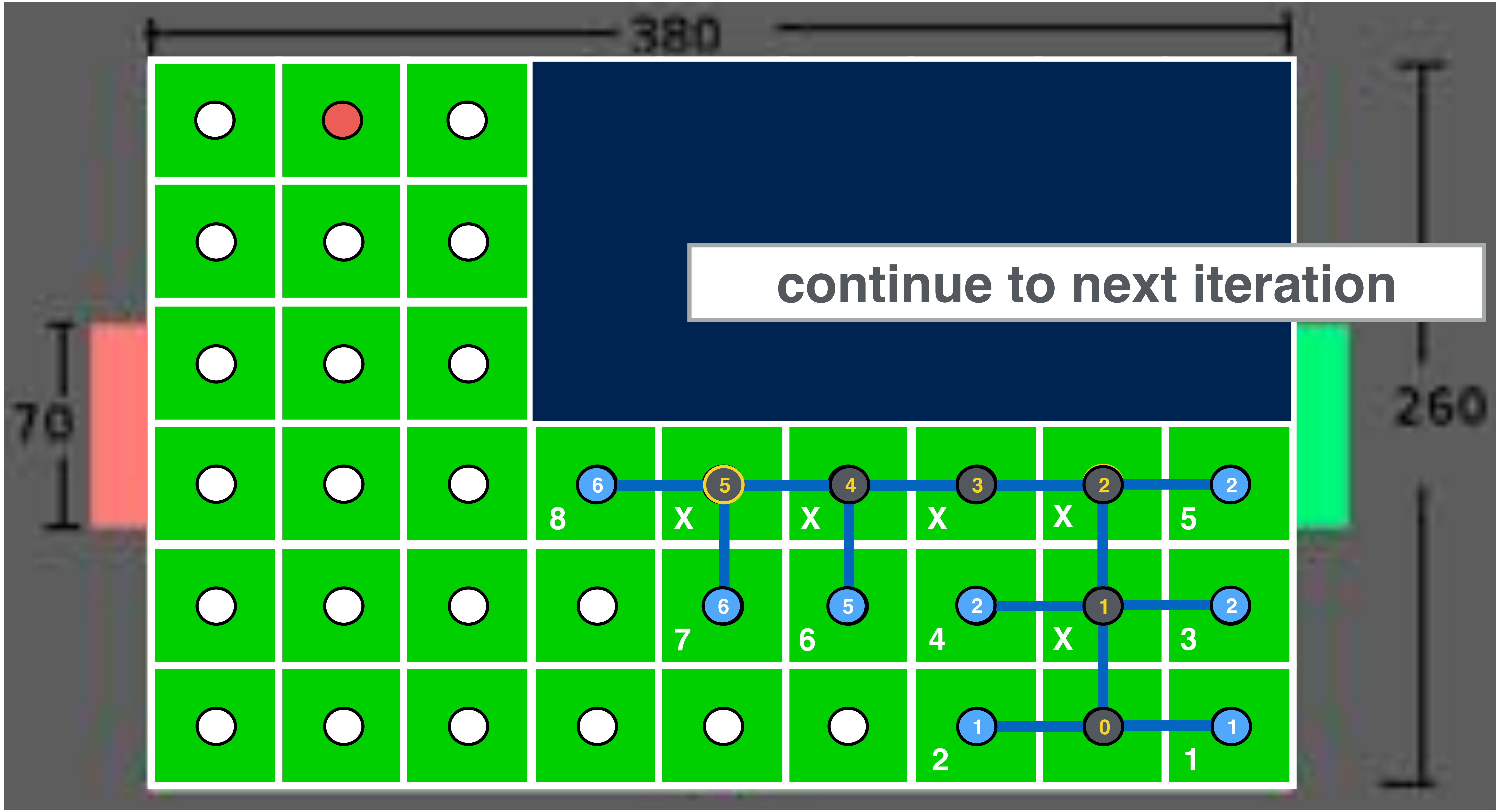


Depth-first search

Depth-first search

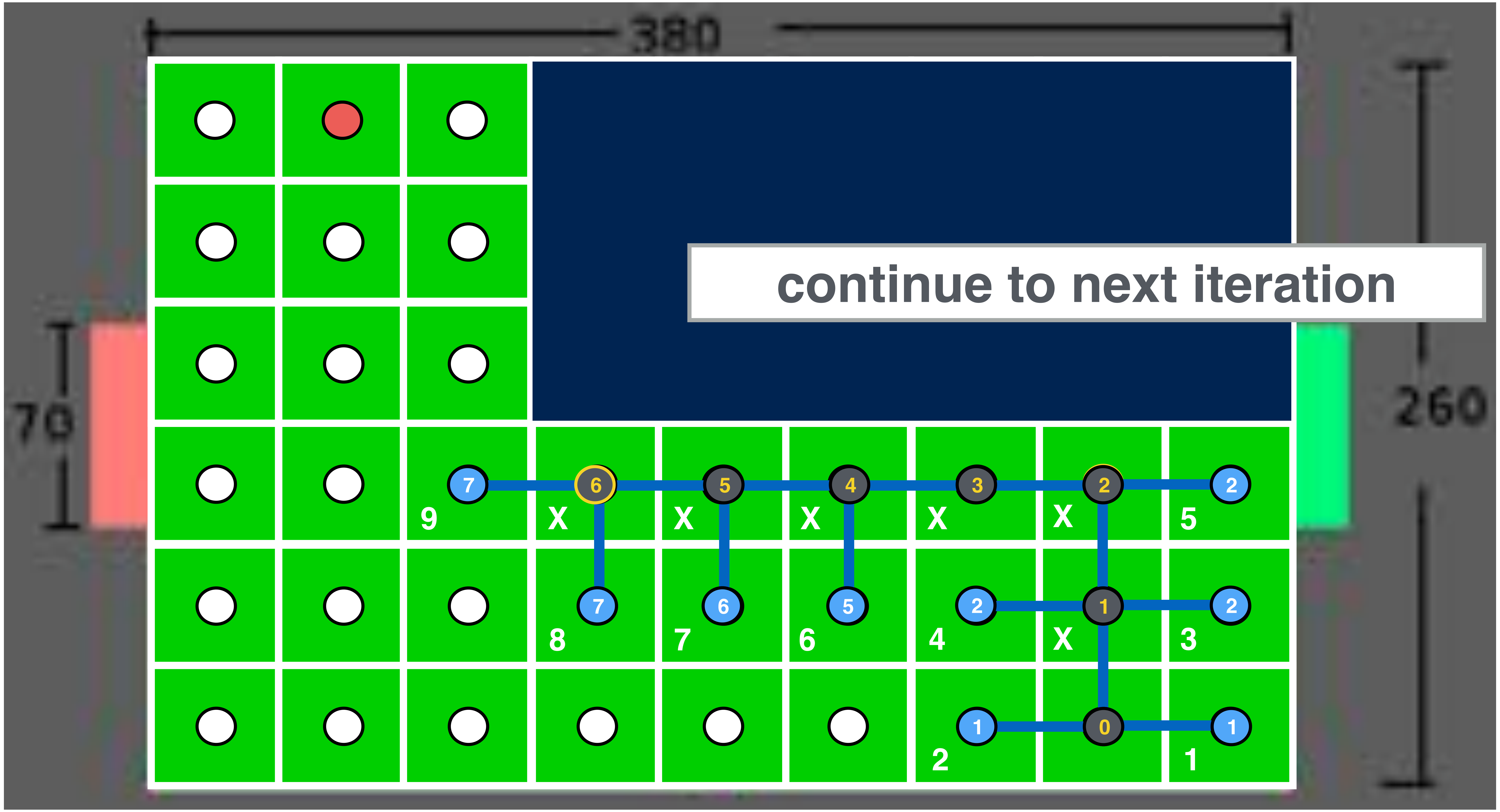


Depth-first search

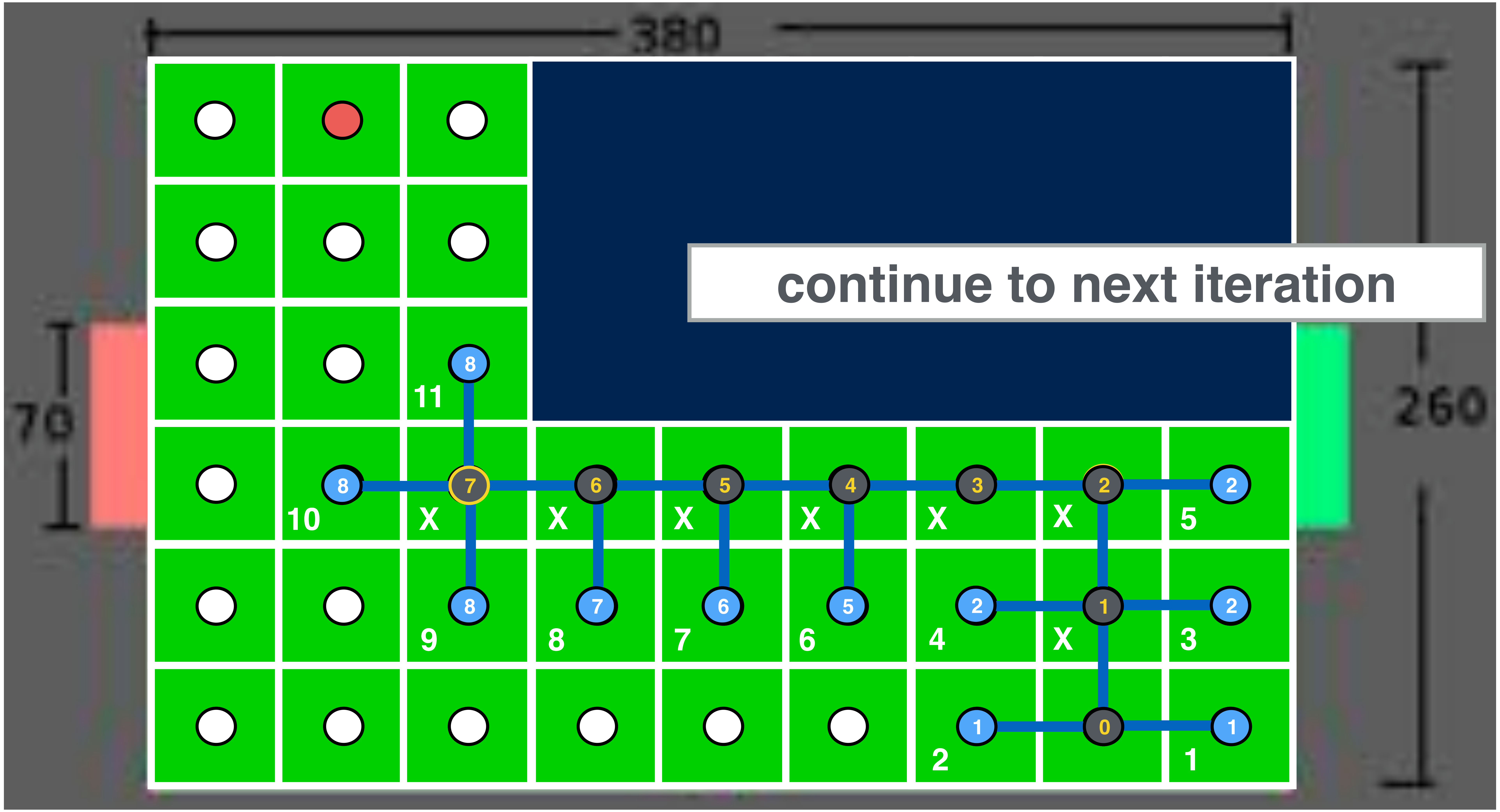


Slide borrowed from Michigan Robotics autorob.org

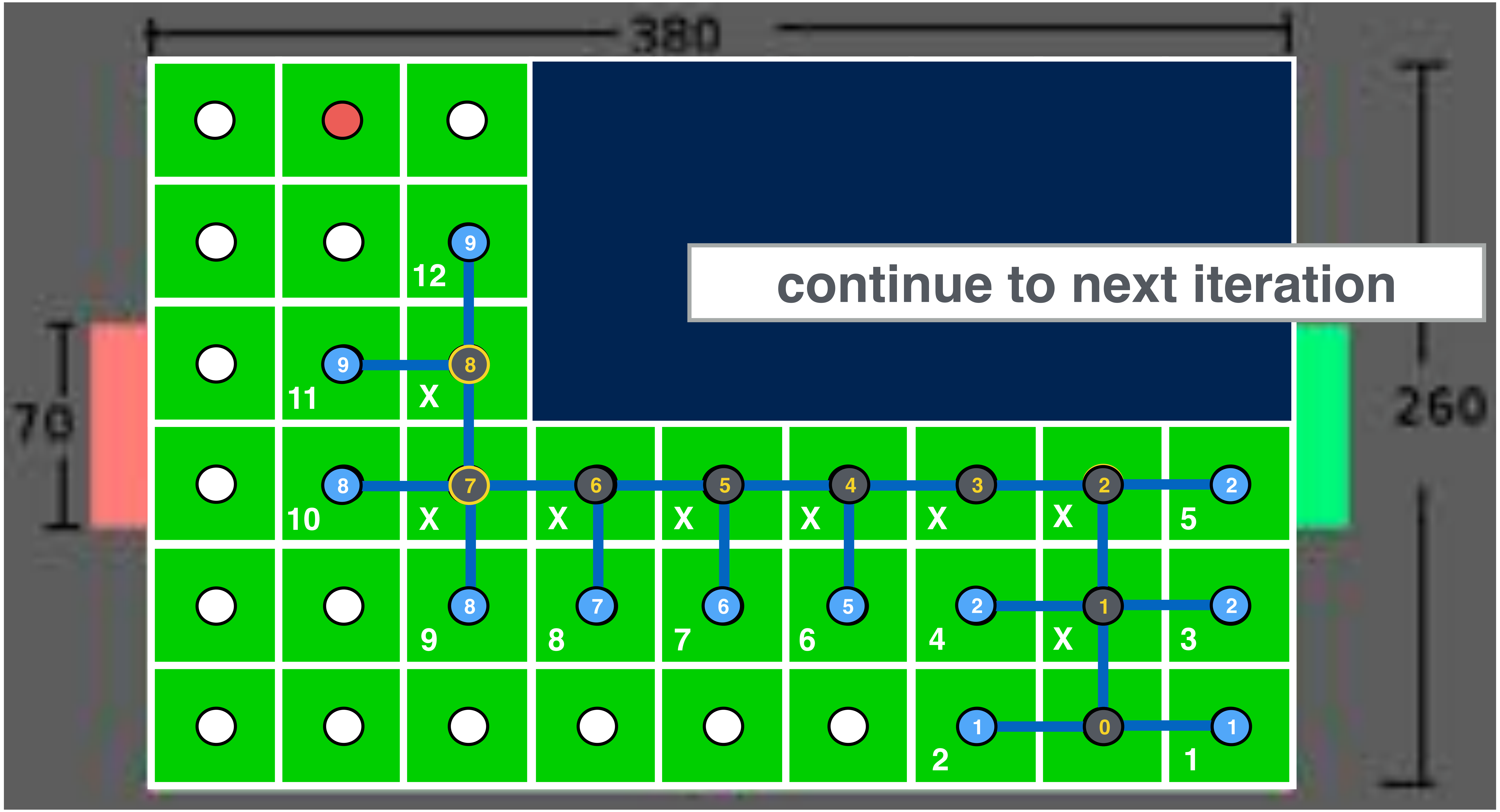
Depth-first search



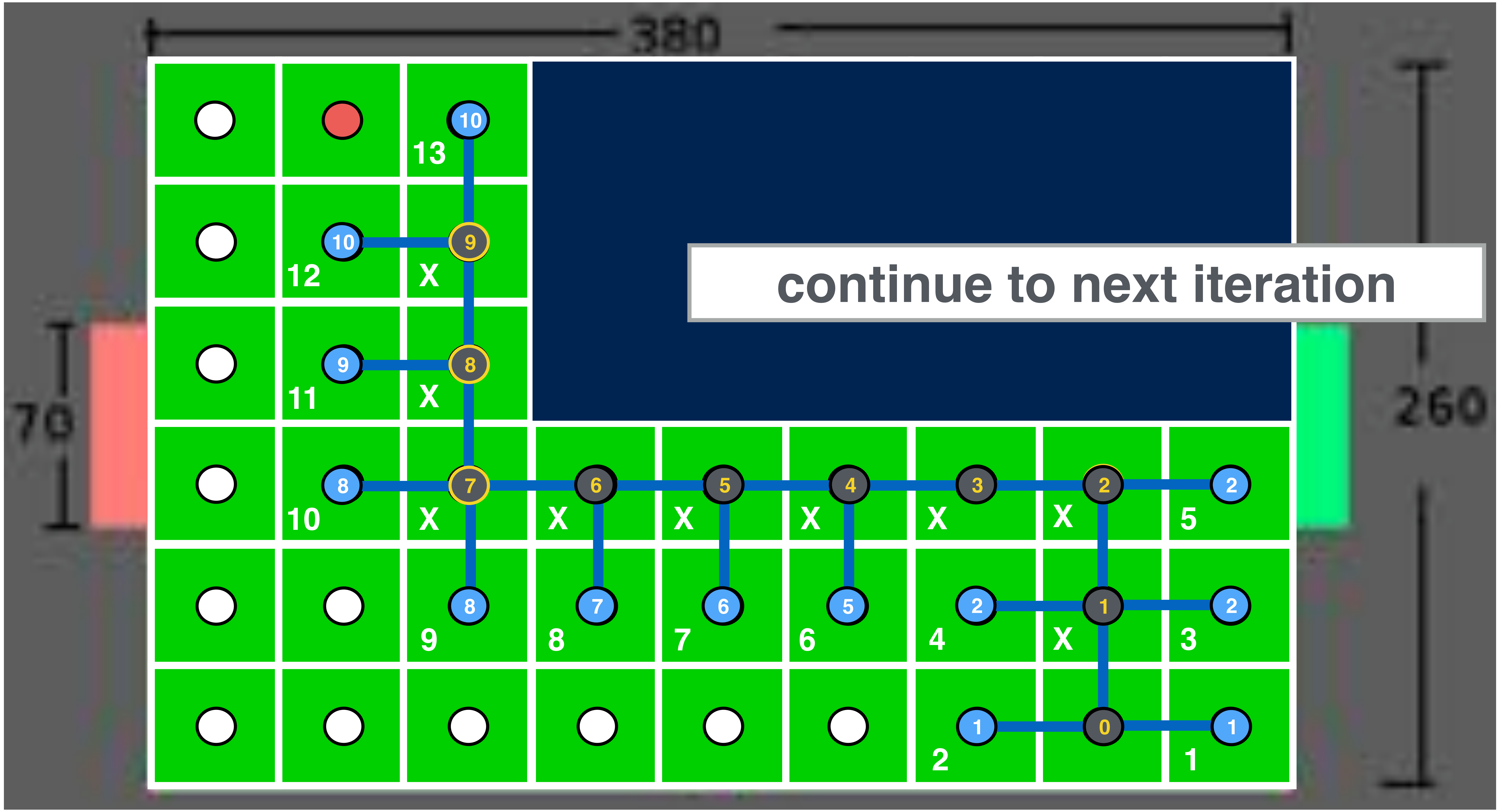
Depth-first search



Depth-first search



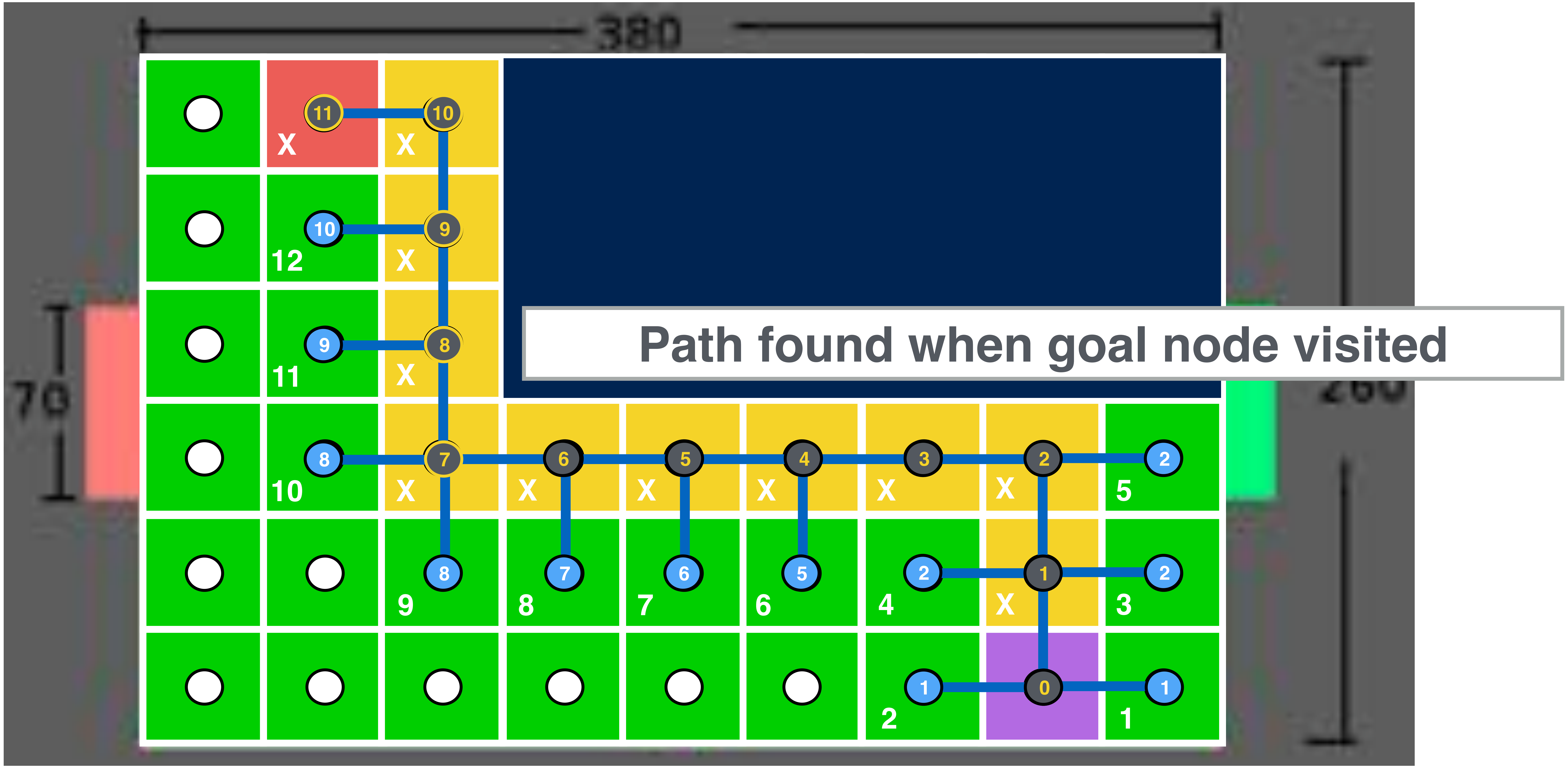
Depth-first search



Slide borrowed from Michigan Robotics autorob.org

Depth-first search

Depth-first search



Slide borrowed from Michigan Robotics autorob.org

Let’s turn this idea into code

Search algorithm template

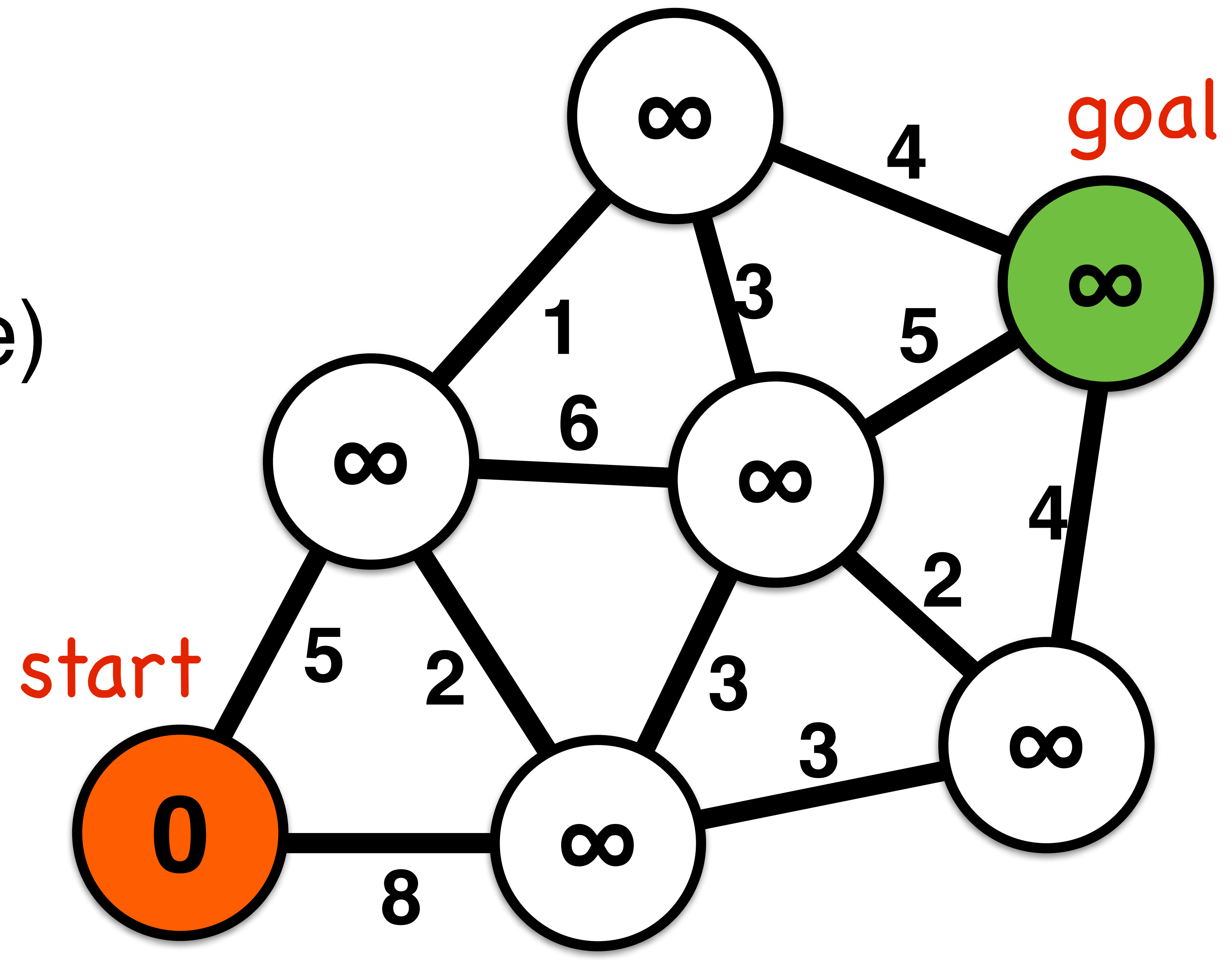
all nodes ← {$dist_{start}$ ← infinity, $parent_{start}$ ← none, $visited_{start}$ ← false}

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

while visit_list != empty && current_node != goal

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

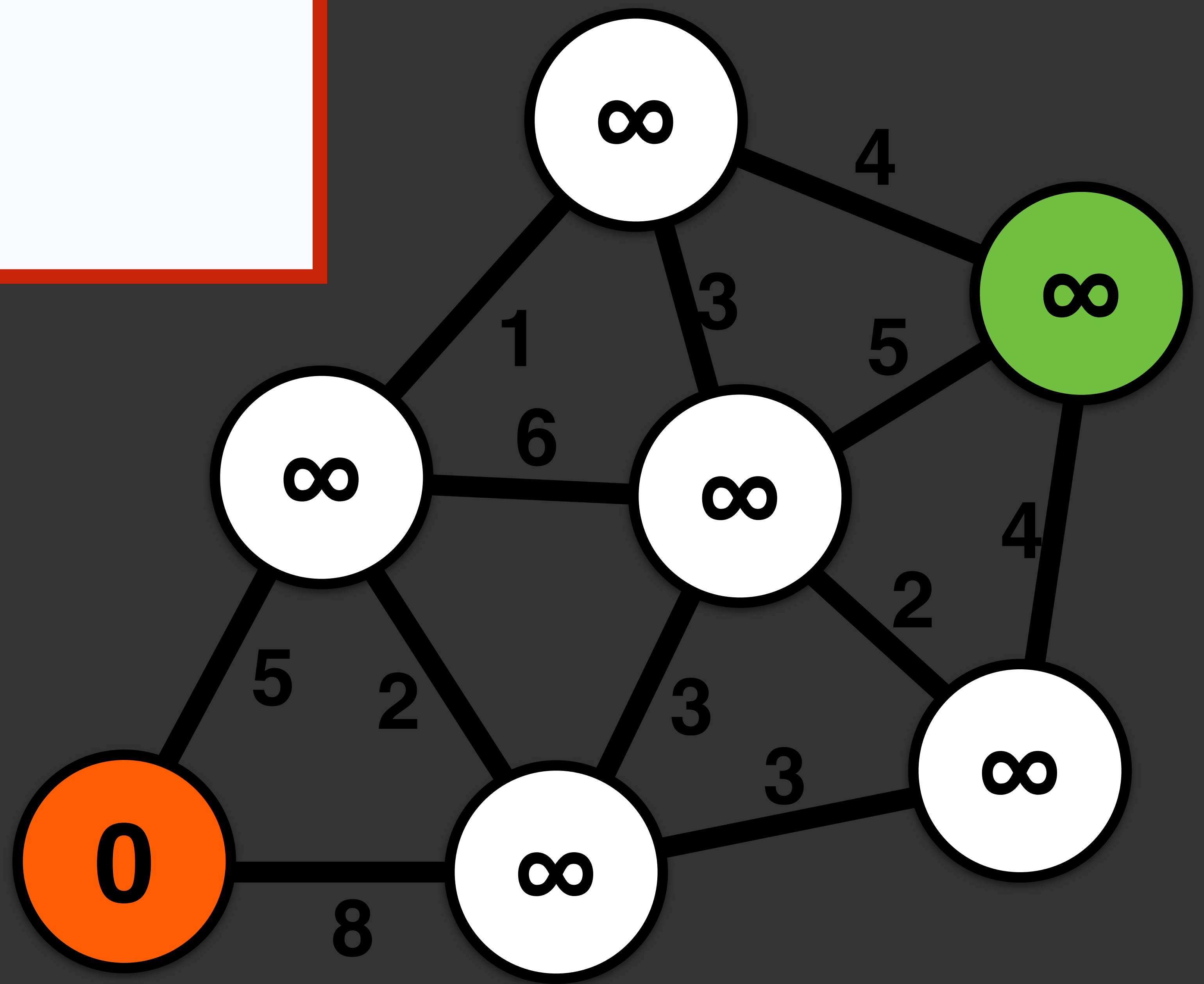
dist_{nbr} ← dist_{cur_node} + distStraightLine(nbr,cur_node)

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

        for each nbr in not_visited(adjacent(cur_node))

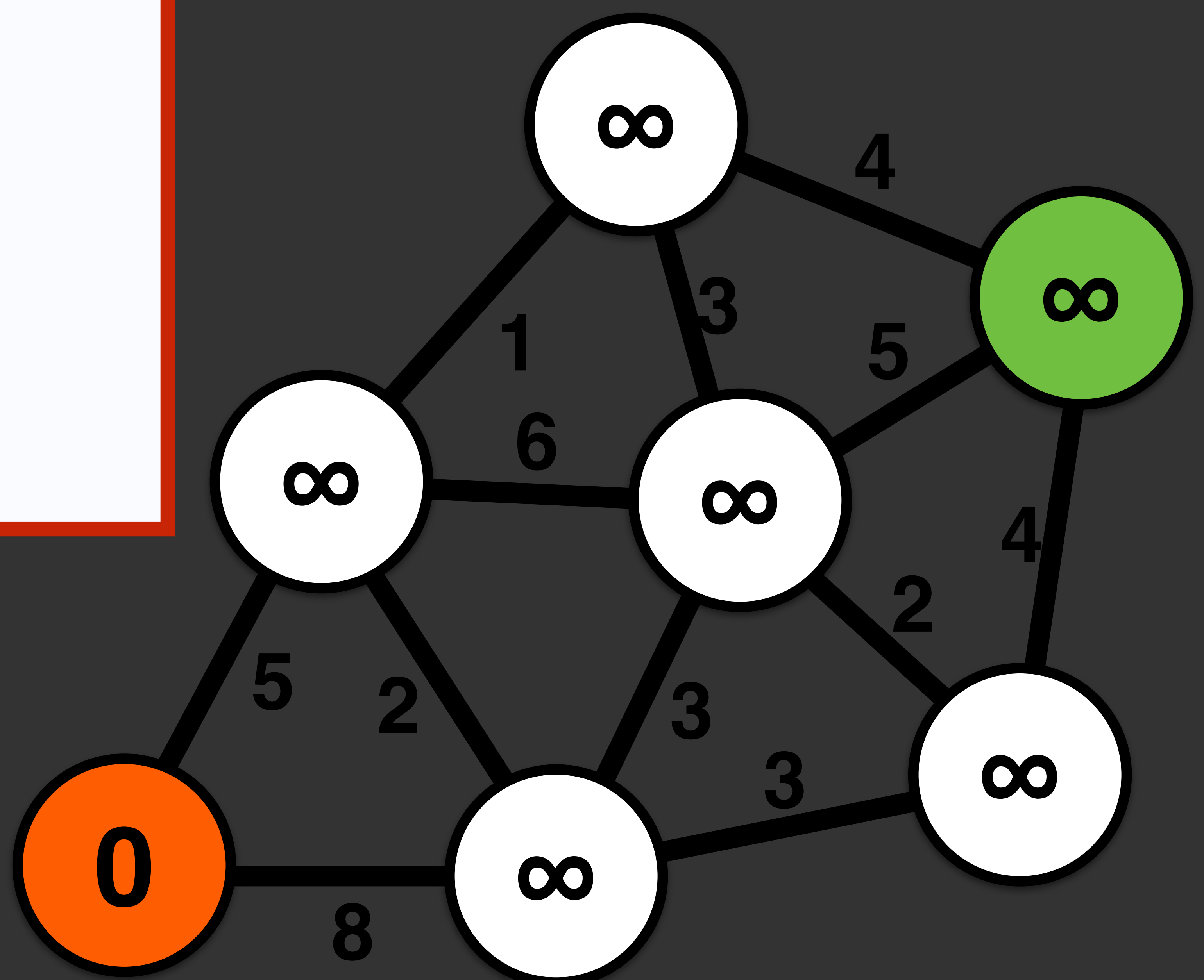
Main Loop

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

        end for loop

    end while loop

output ← parent, distance



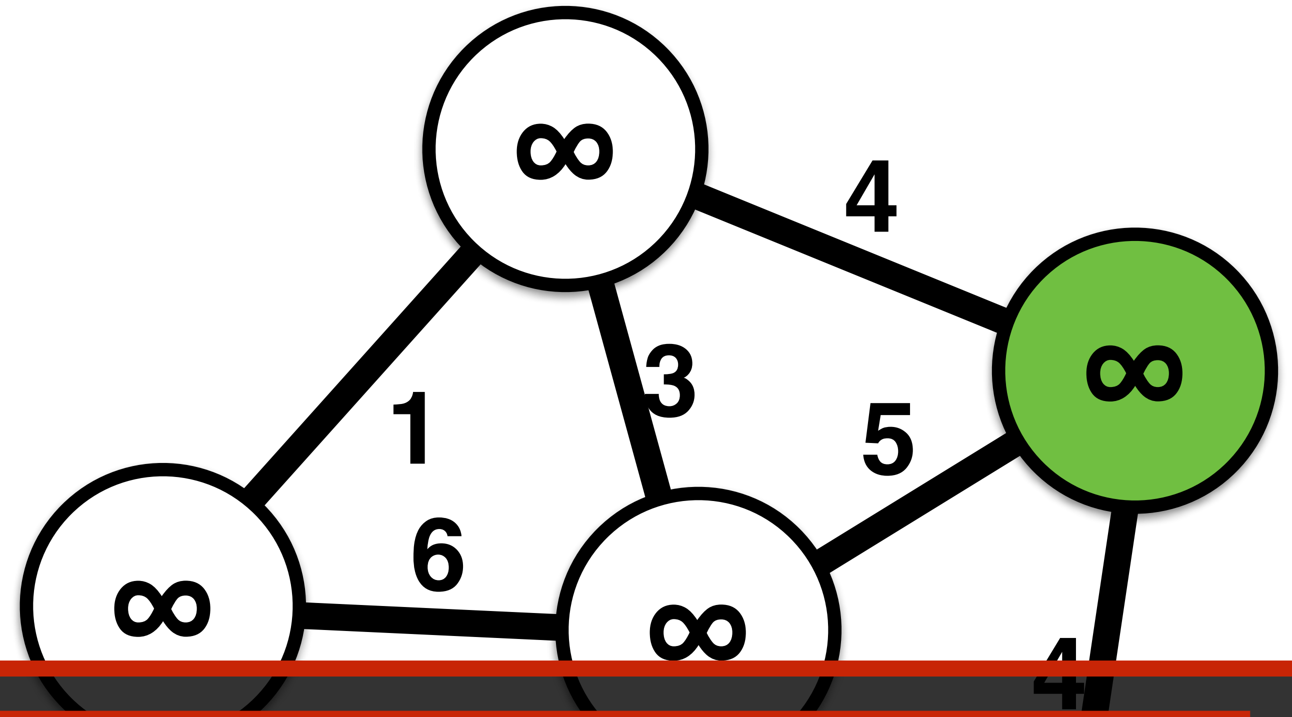
Search algorithm template

all nodes ← {$dist_{start}$ ← 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)

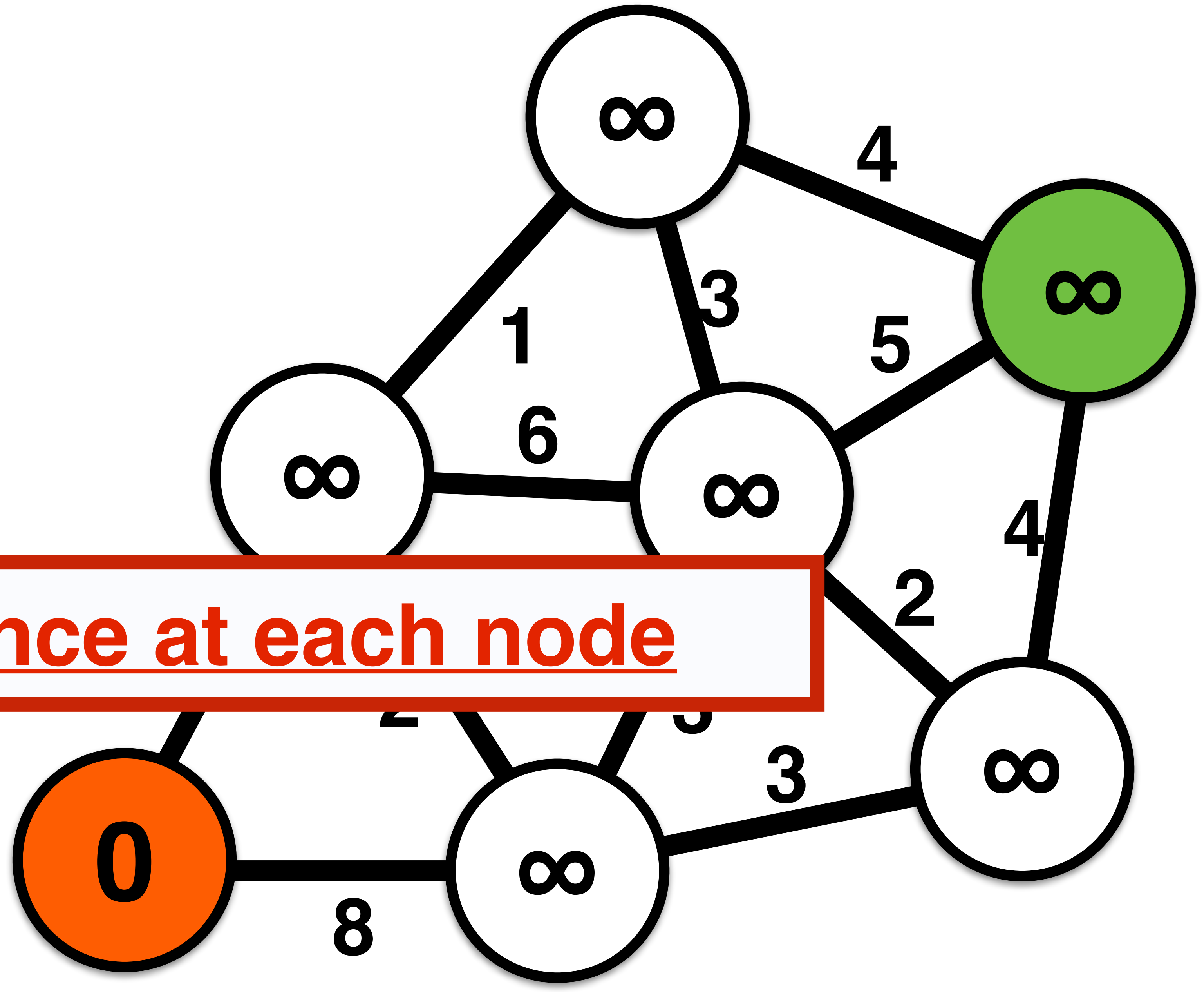
            end if

        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

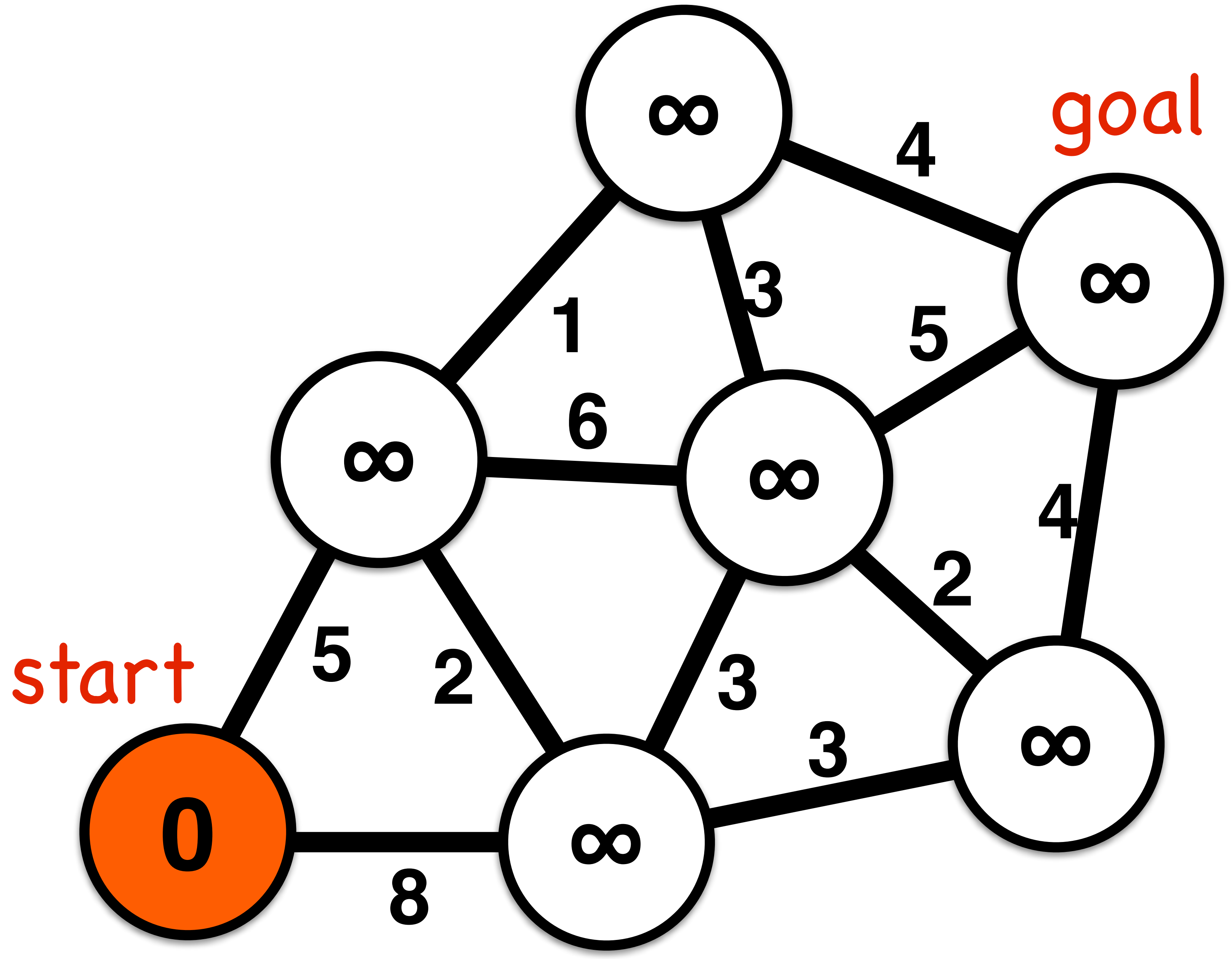
$dist_{nbr}$ ← $dist_{cur\_node}$ + distance(nbr,cur_node)

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}

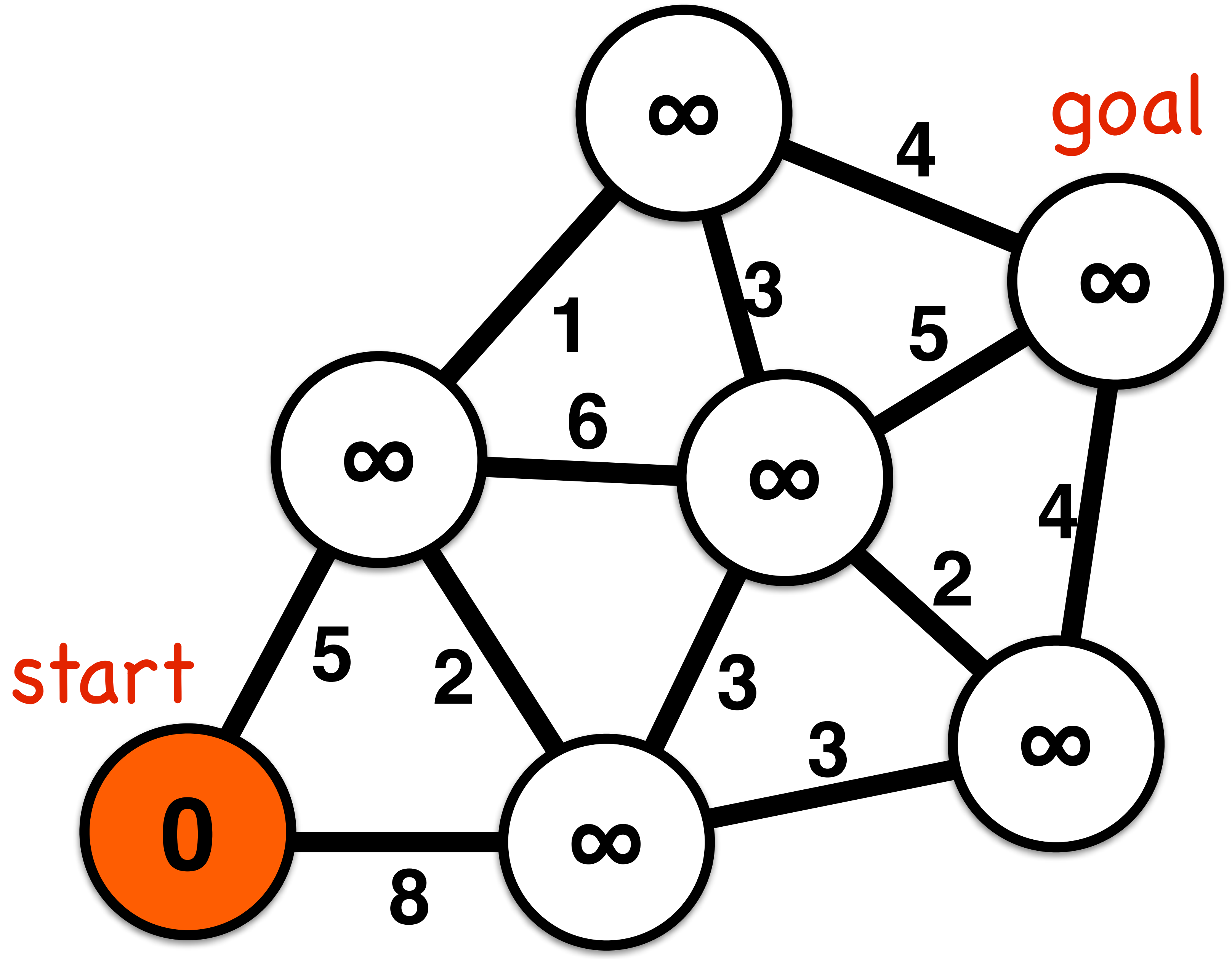
start_node ← {$dist_{start}$ ← 0, $parent_{start}$ ← none, $visited_{start}$ ← true}

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

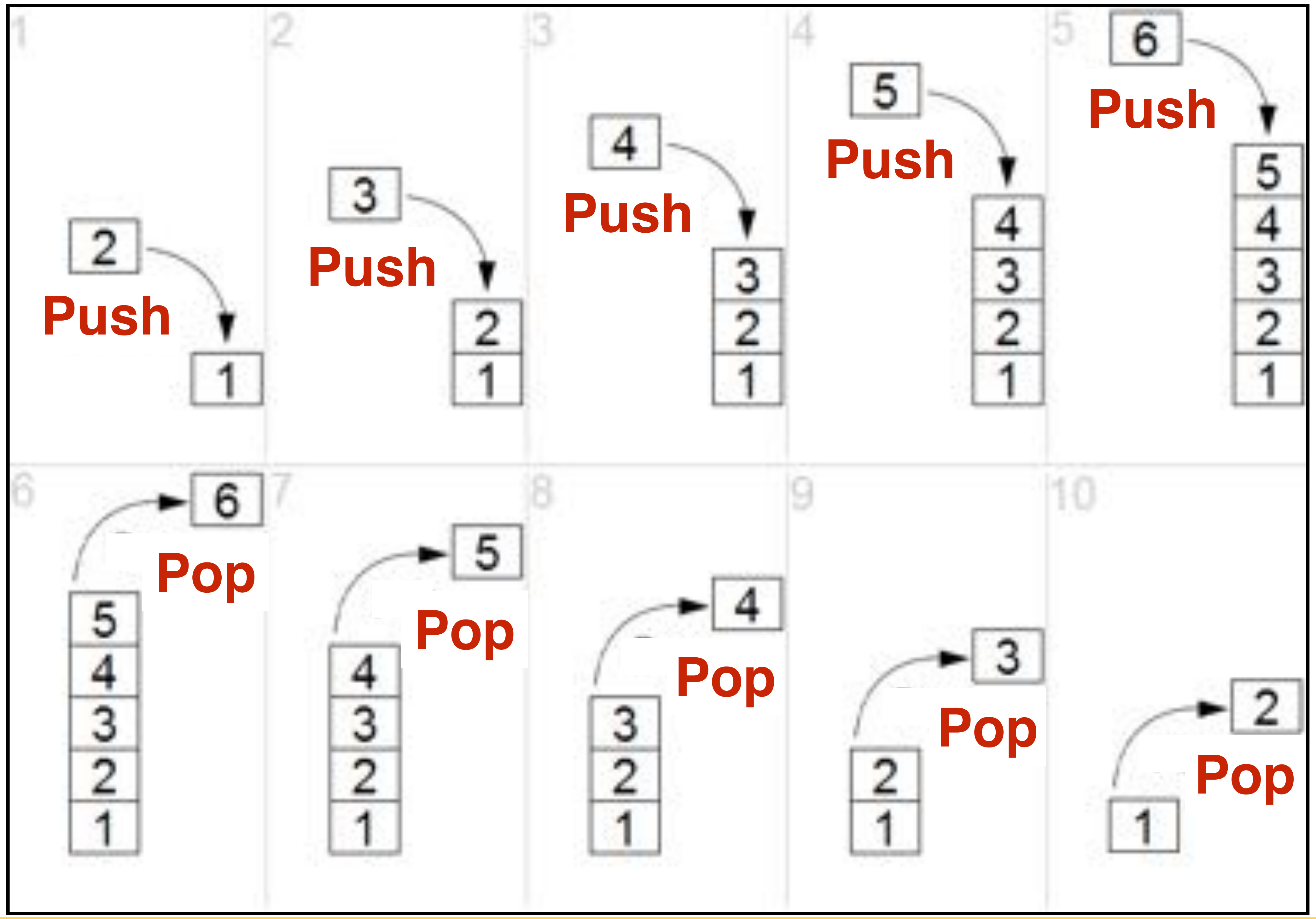
Priority:
Most recent

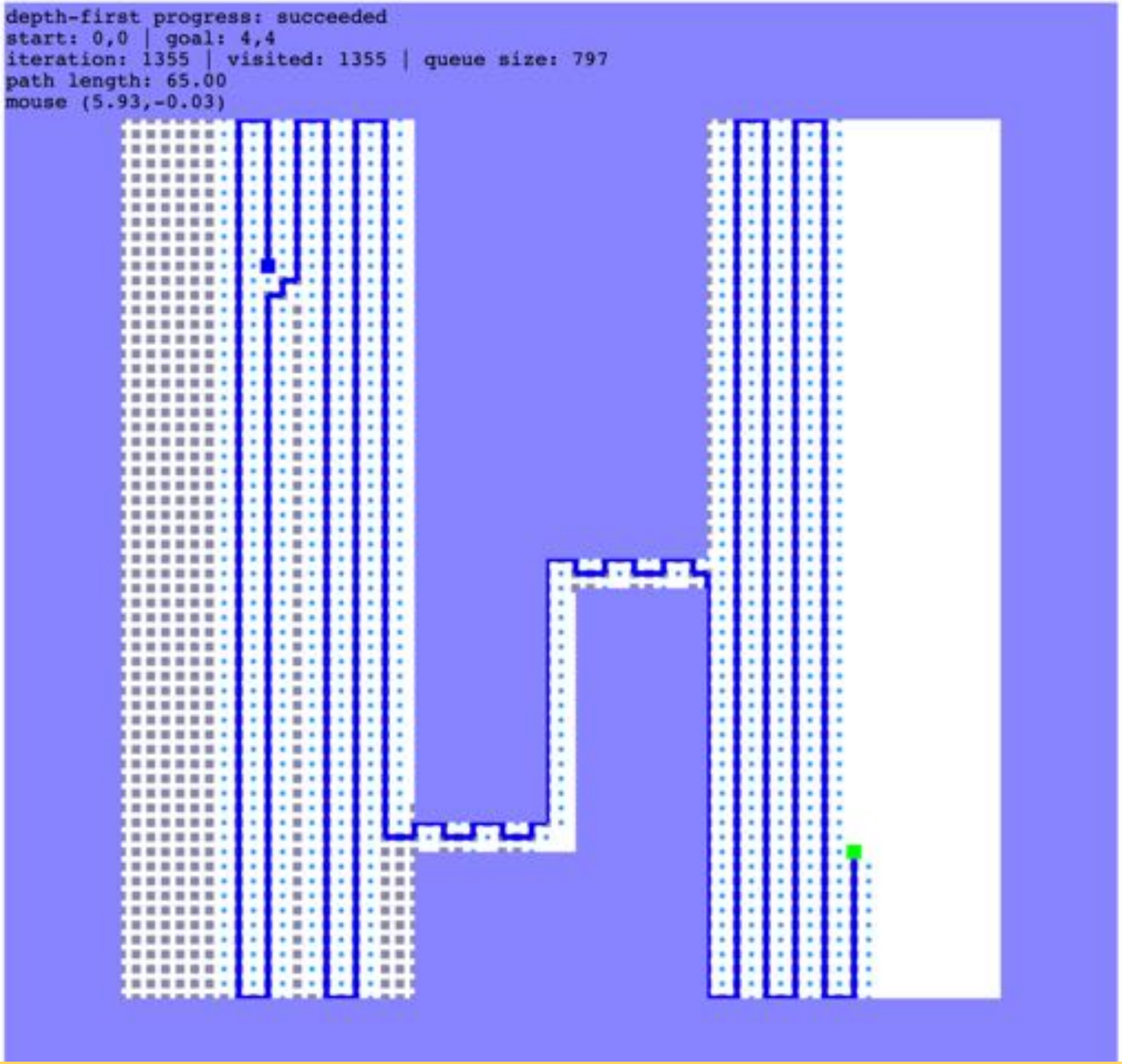


Stack data structure

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

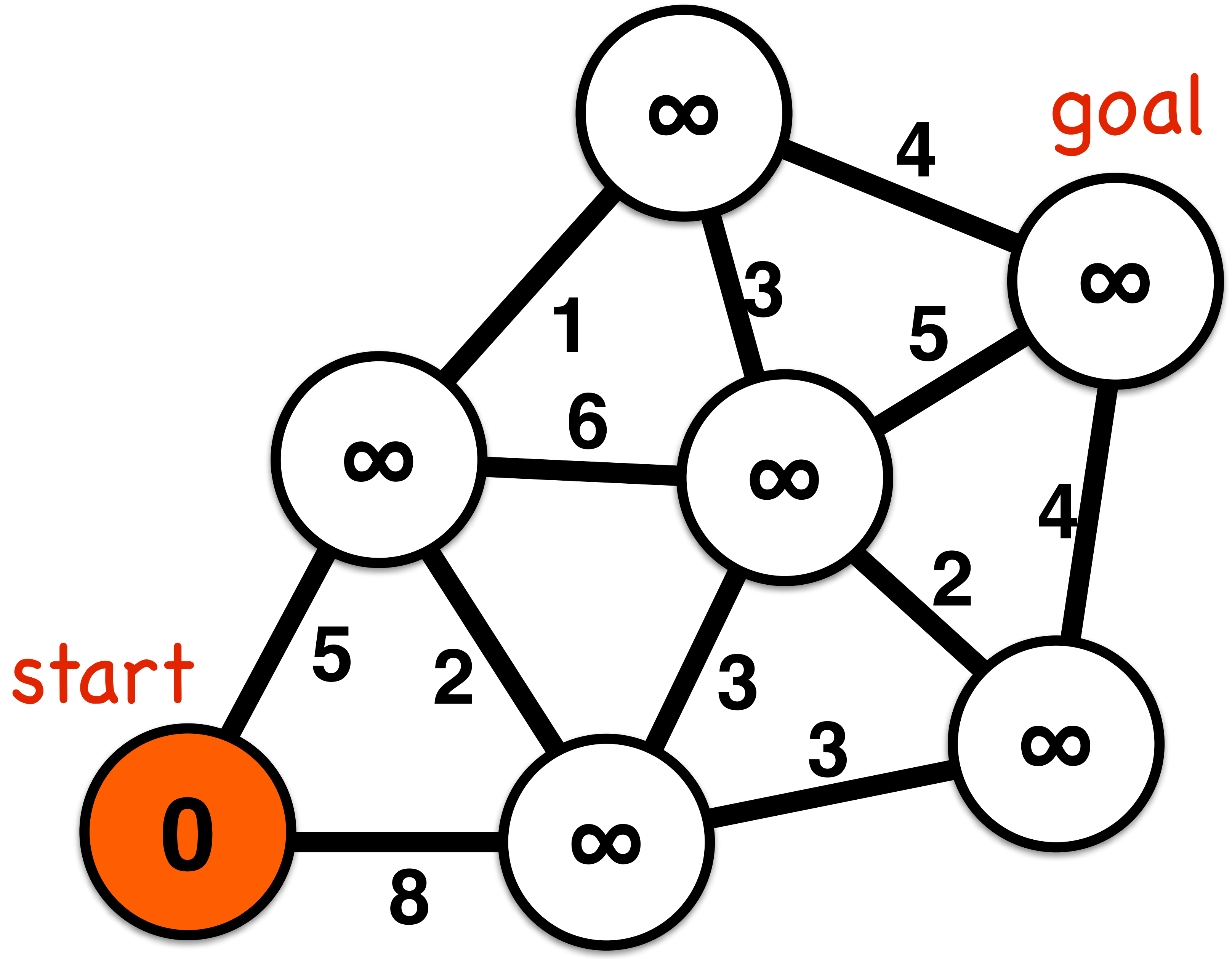
Search algorithm template

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**)

Priority:
Least recent

$visited_{cur\_node}$ ← 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

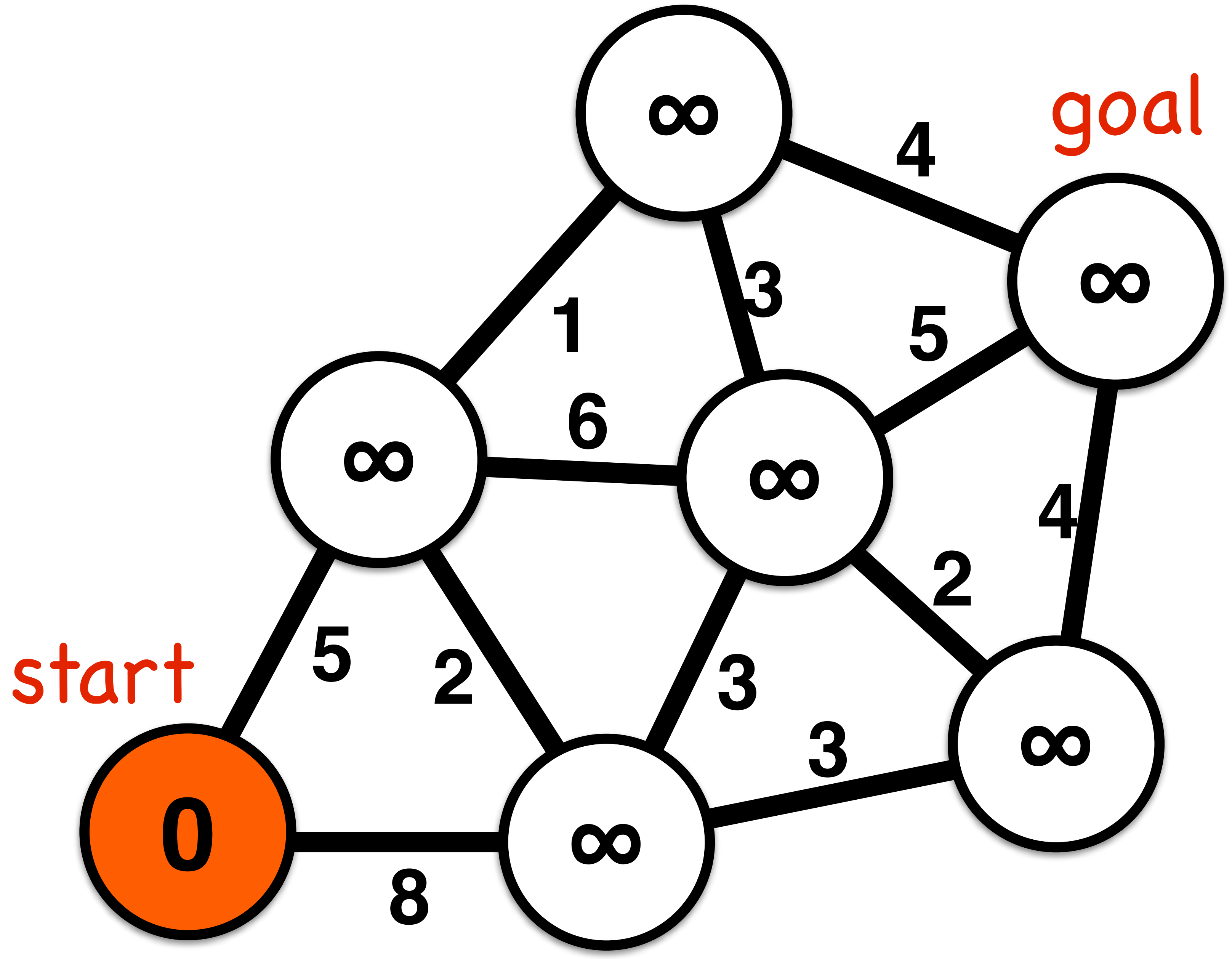
$dist_{nbr}$ ← $dist_{cur\_node}$ + distance(nbr,cur_node)

end if

end for loop

end while loop

output ← parent, distance



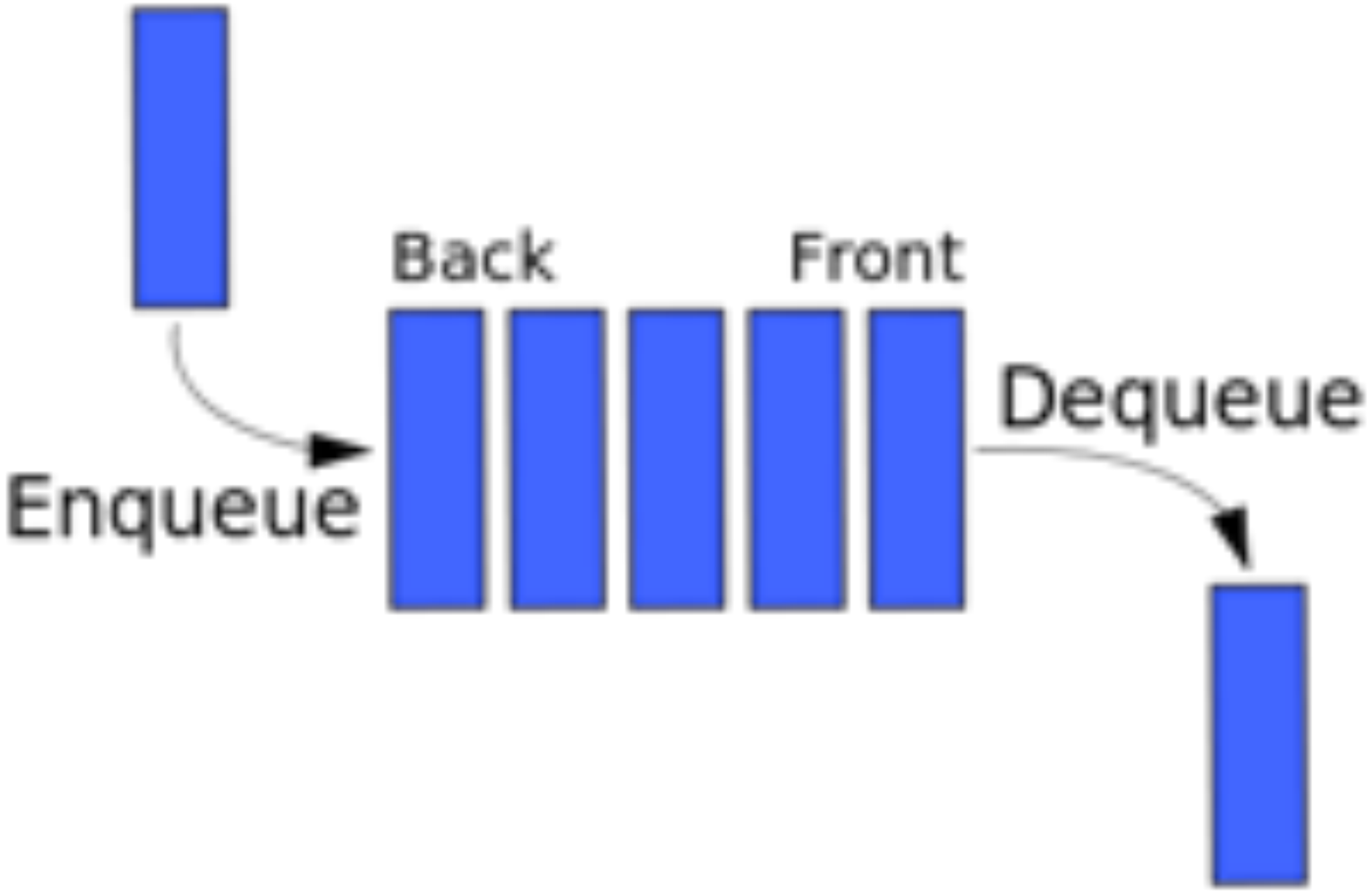
Slide borrowed from Michigan Robotics autorob.org

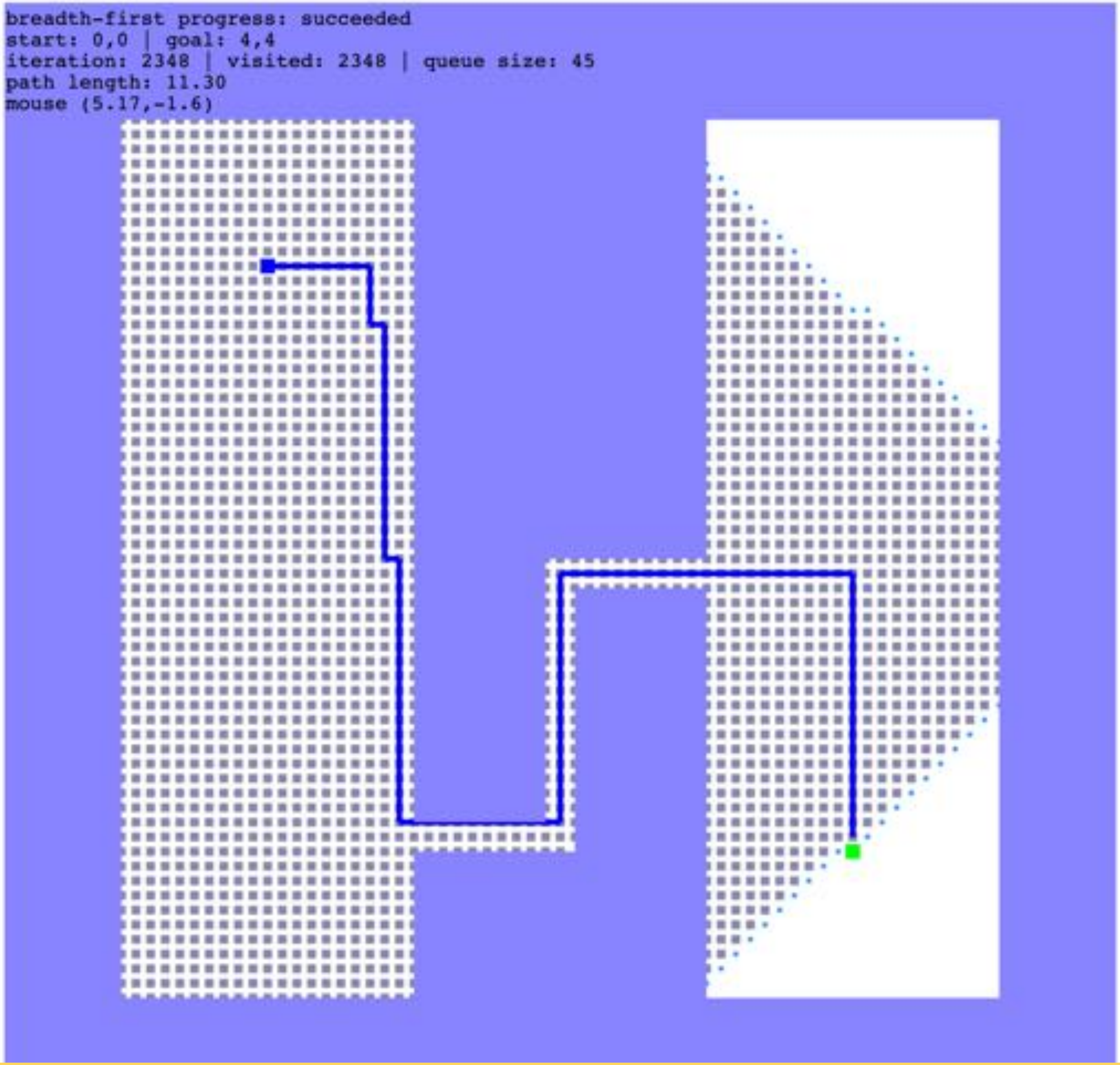
Queue data structure

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





Slide borrowed from Michigan Robotics autorob.org

Dijkstra’s algorithm

Search algorithm template

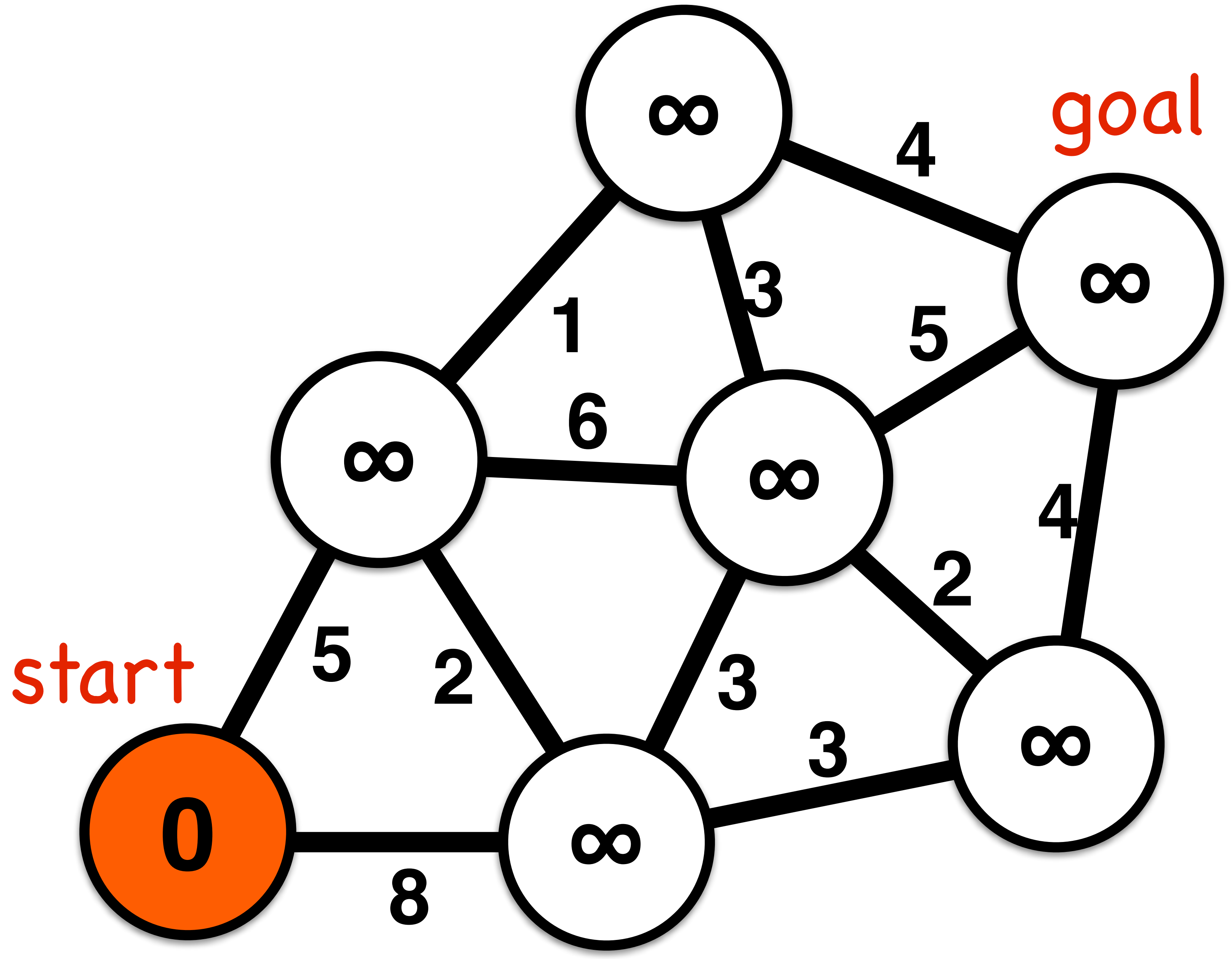
all nodes ← {$dist_{start}$ ← infinity, $parent_{start}$ ← none, $visited_{start}$ ← false}

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)

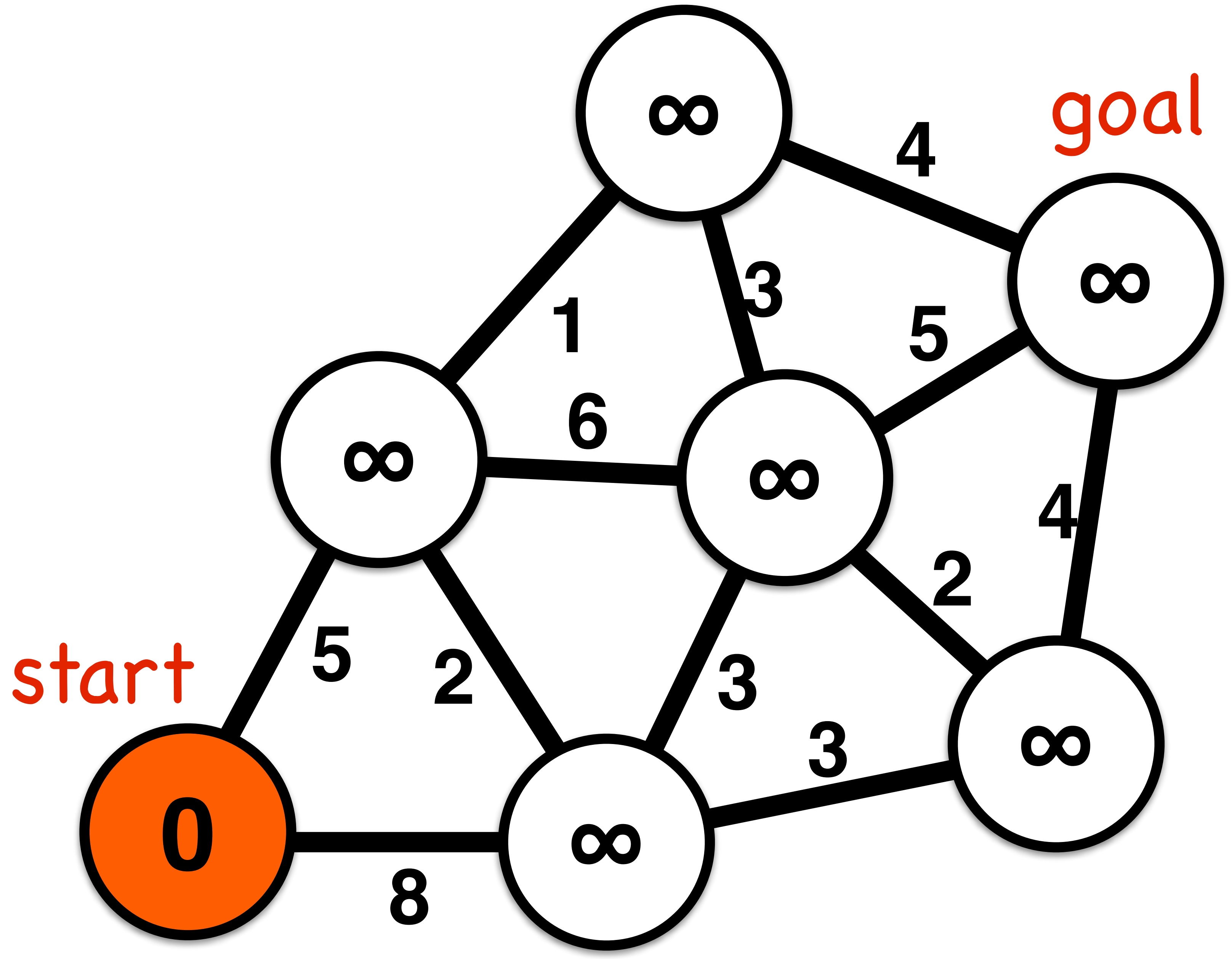
      end if

    end for loop

  end while loop

output ← parent, distance

Priority:
Minimum route distance from start



Dijkstra shortest path algorithm

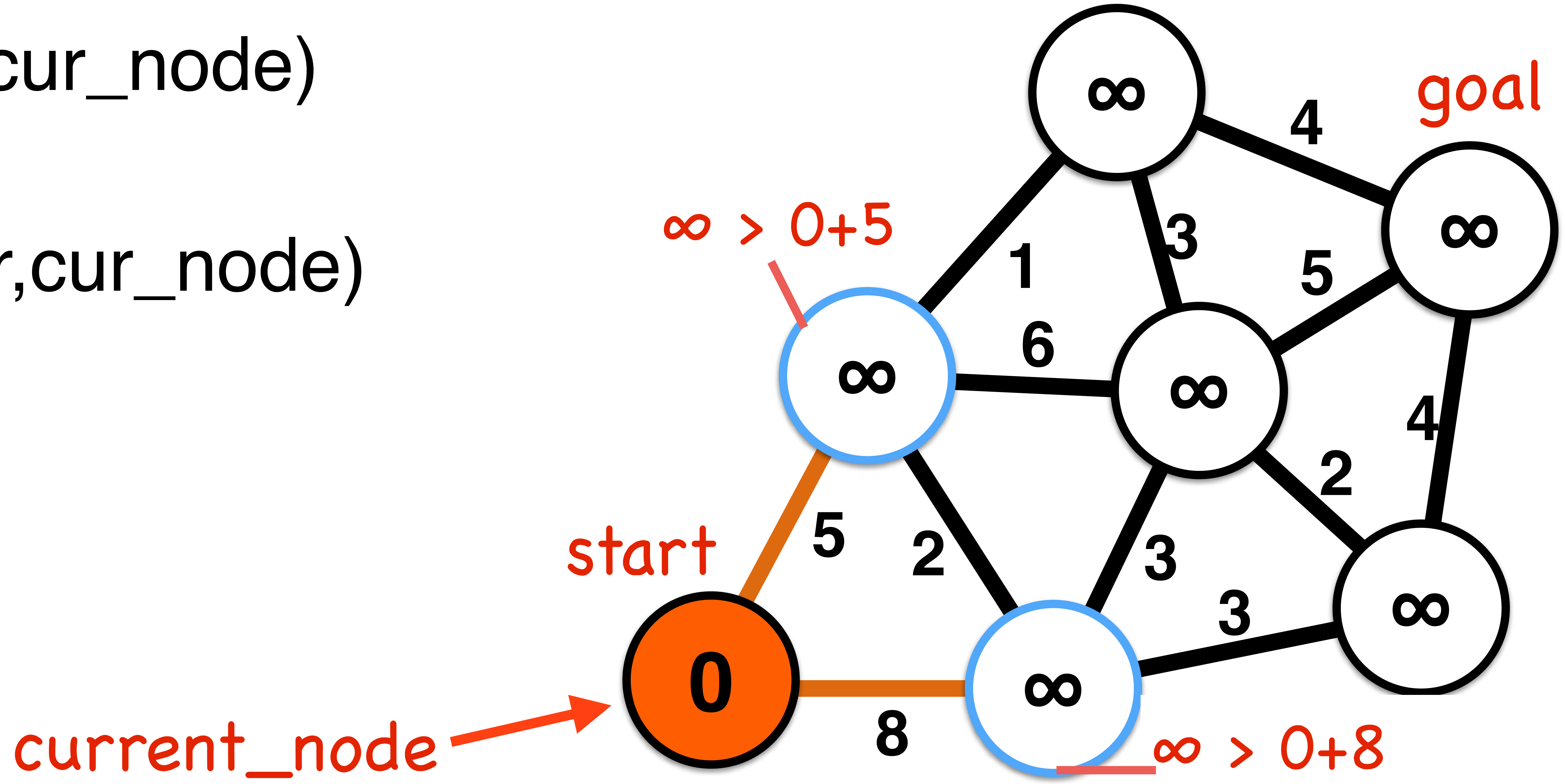
all nodes ← {$\text{dist}_{\text{start}}$ ← infinity, $\text{parent}_{\text{start}}$ ← none, $\text{visited}_{\text{start}}$ ← false}

start_node ← {$\text{dist}_{\text{start}}$ ← 0, $\text{parent}_{\text{start}}$ ← none, $\text{visited}_{\text{start}}$ ← 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)

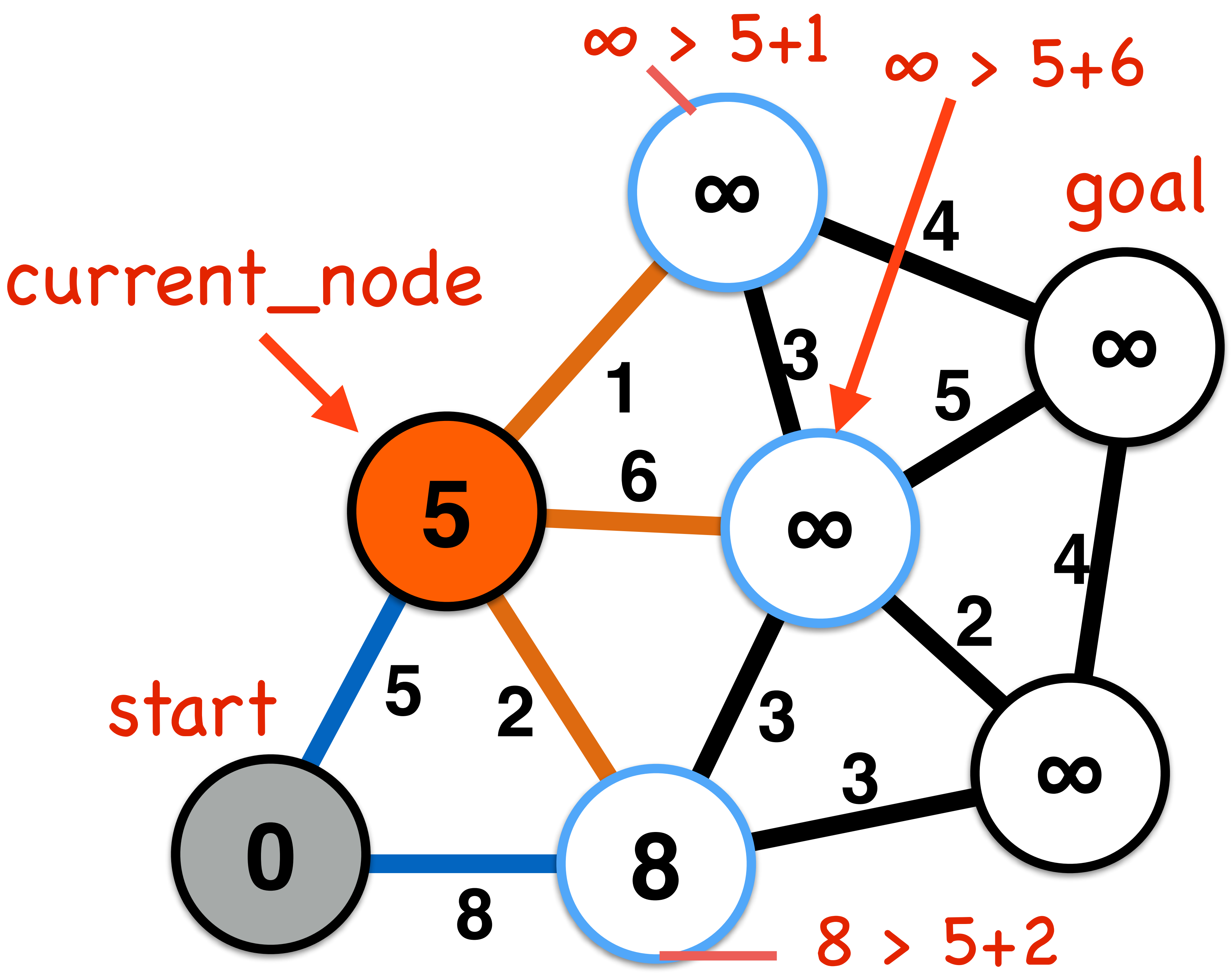
end if

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)

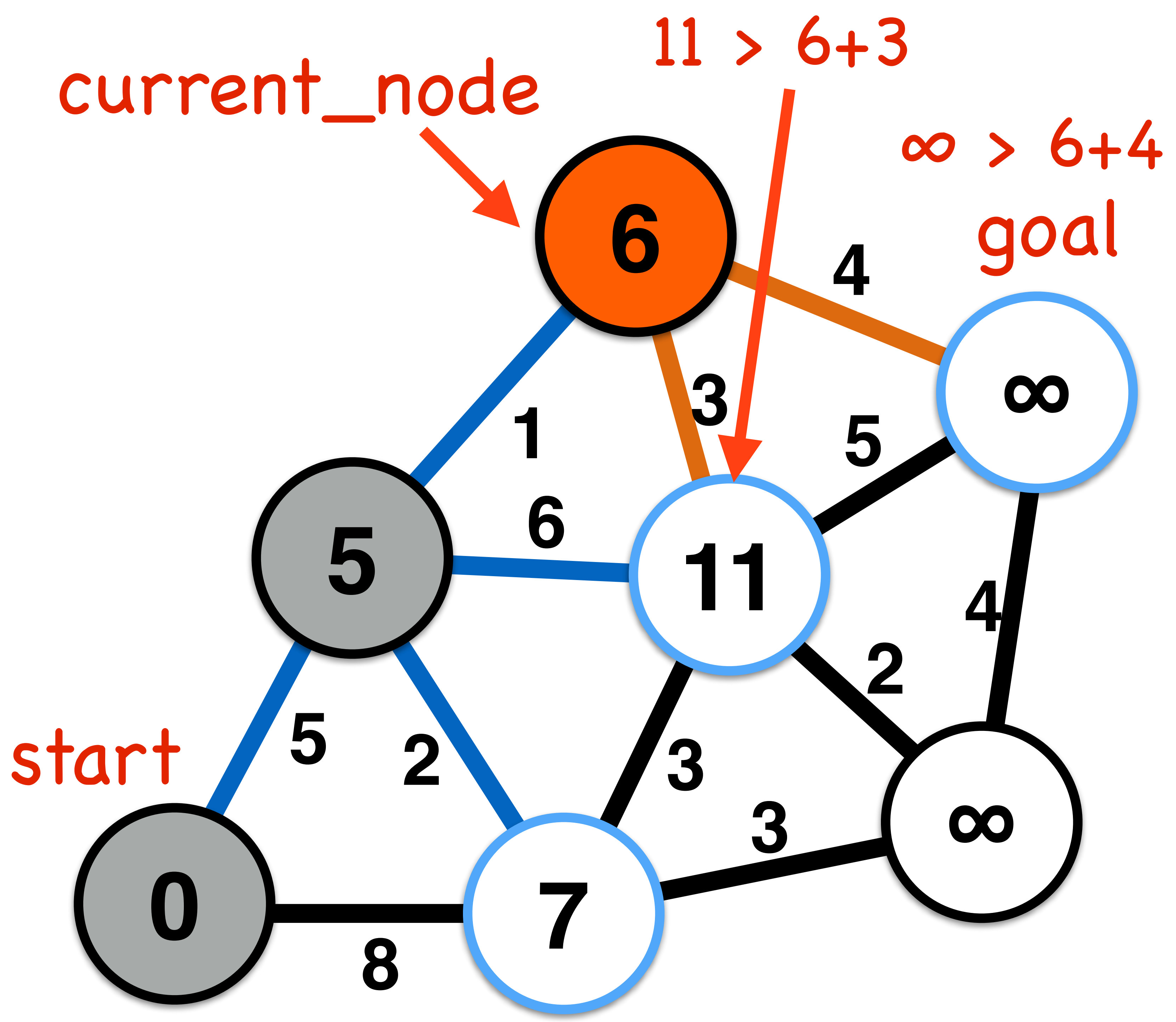
end if

end for loop

end while loop

output ← parent, distance

Dijkstra walkthrough



Slide borrowed from Michigan Robotics autorob.org

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

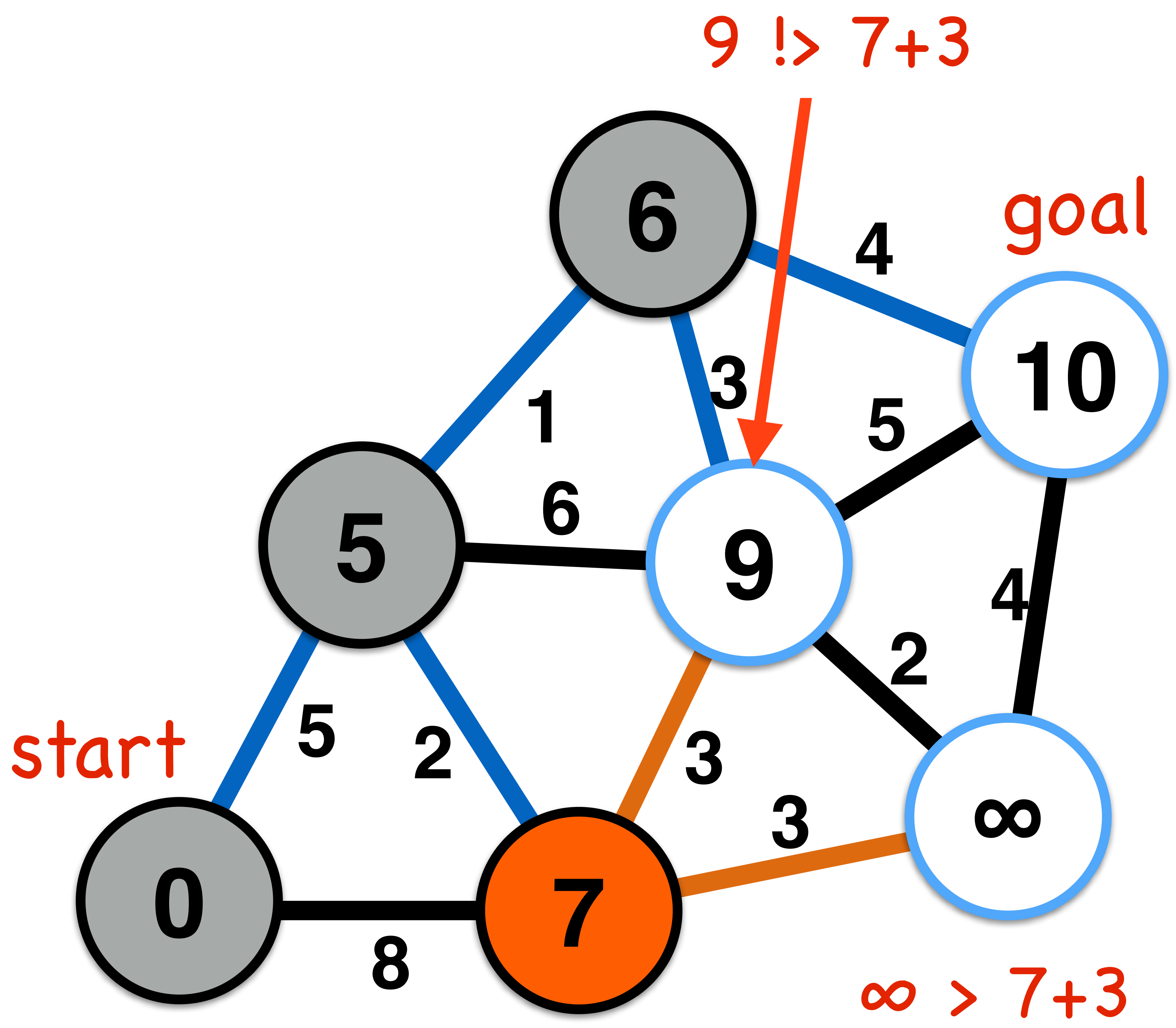
                $dist_{nbr}$ ← $dist_{cur\_node}$ + distance(nbr,cur_node)

            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

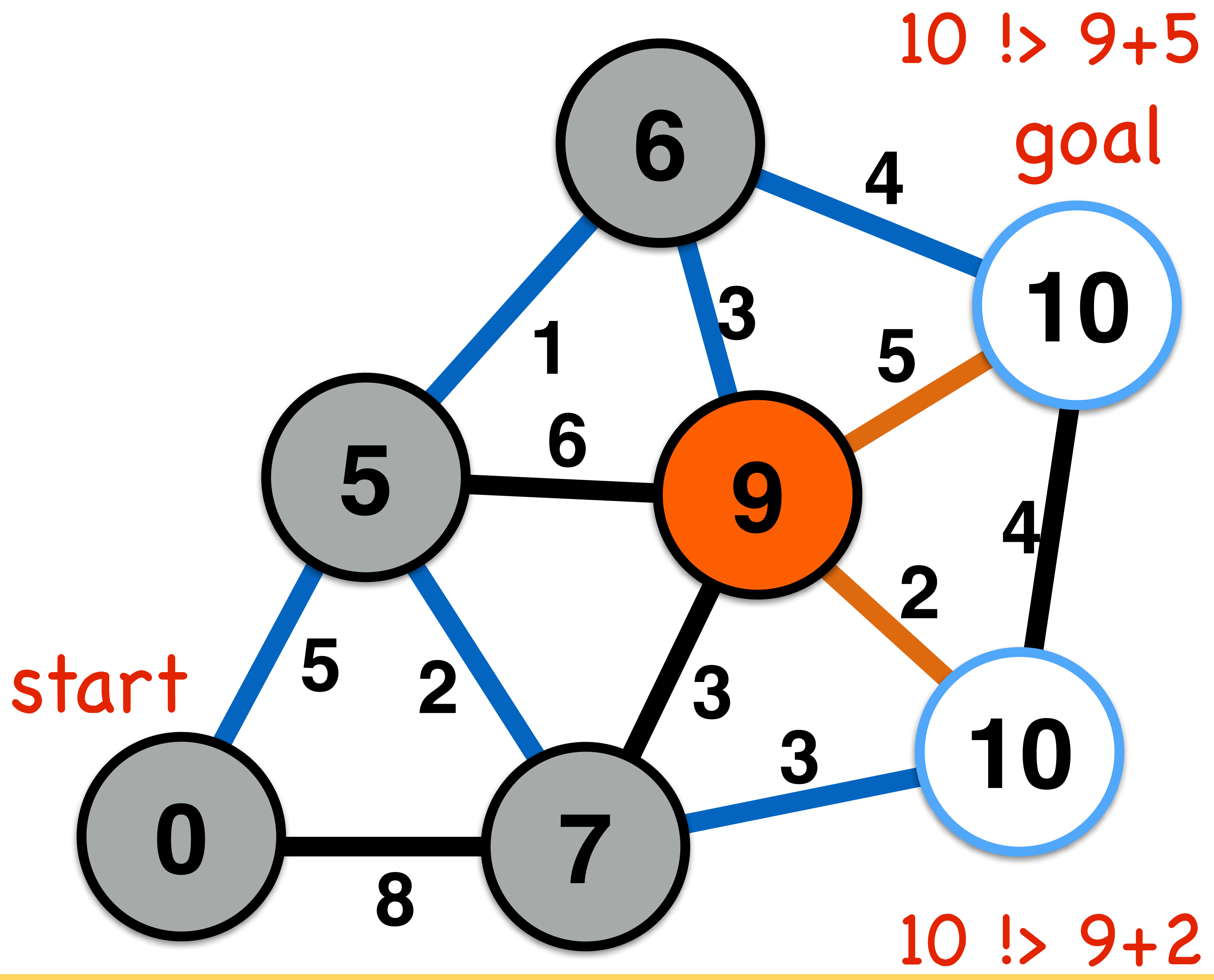
$dist_{nbr}$ ← $dist_{cur\_node}$ + distance(nbr,cur_node)

end if

end for loop

end while loop

output ← parent, distance



Slide borrowed from Michigan Robotics autorob.org

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

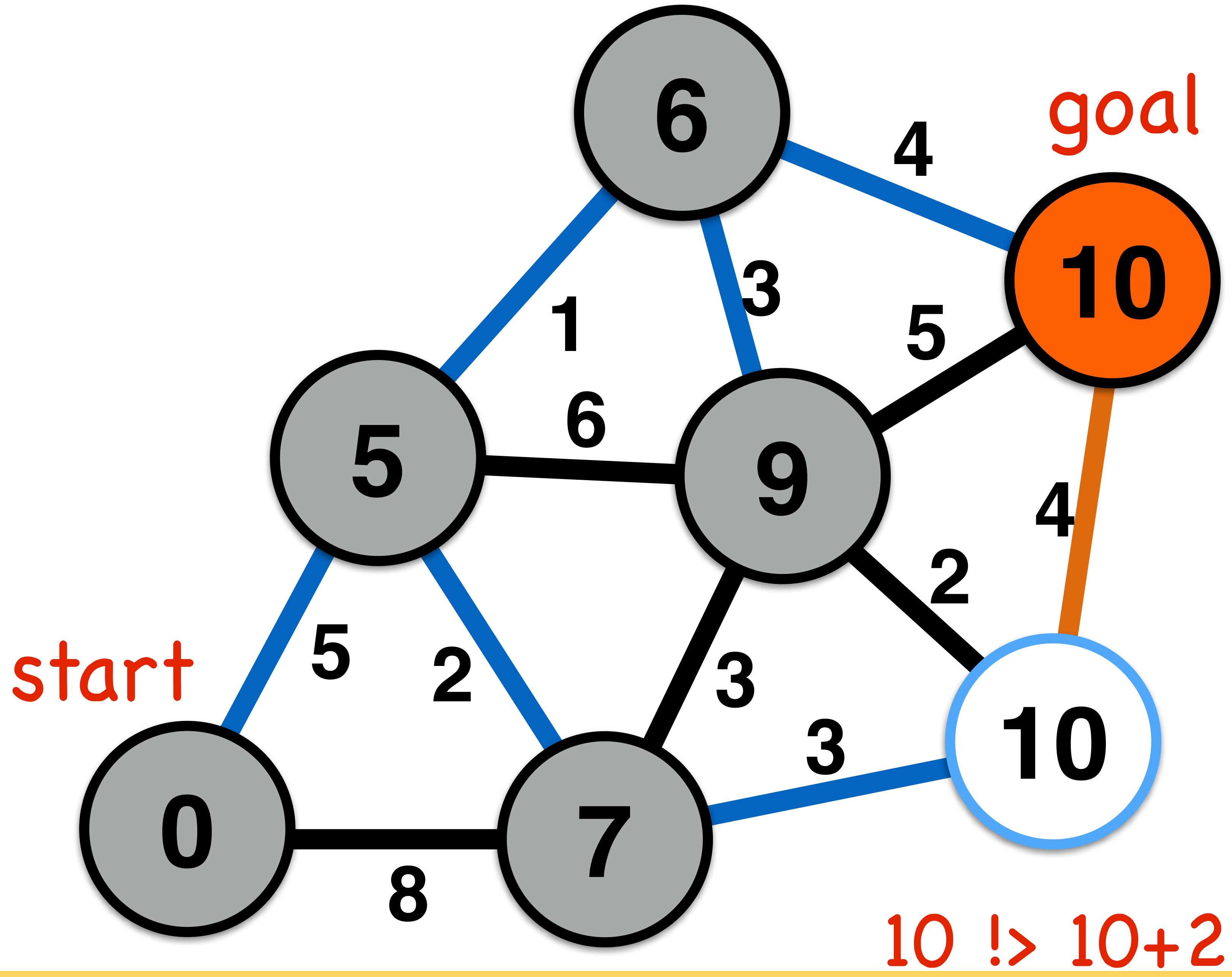
$dist_{nbr}$ ← $dist_{cur\_node}$ + distance(nbr,cur_node)

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

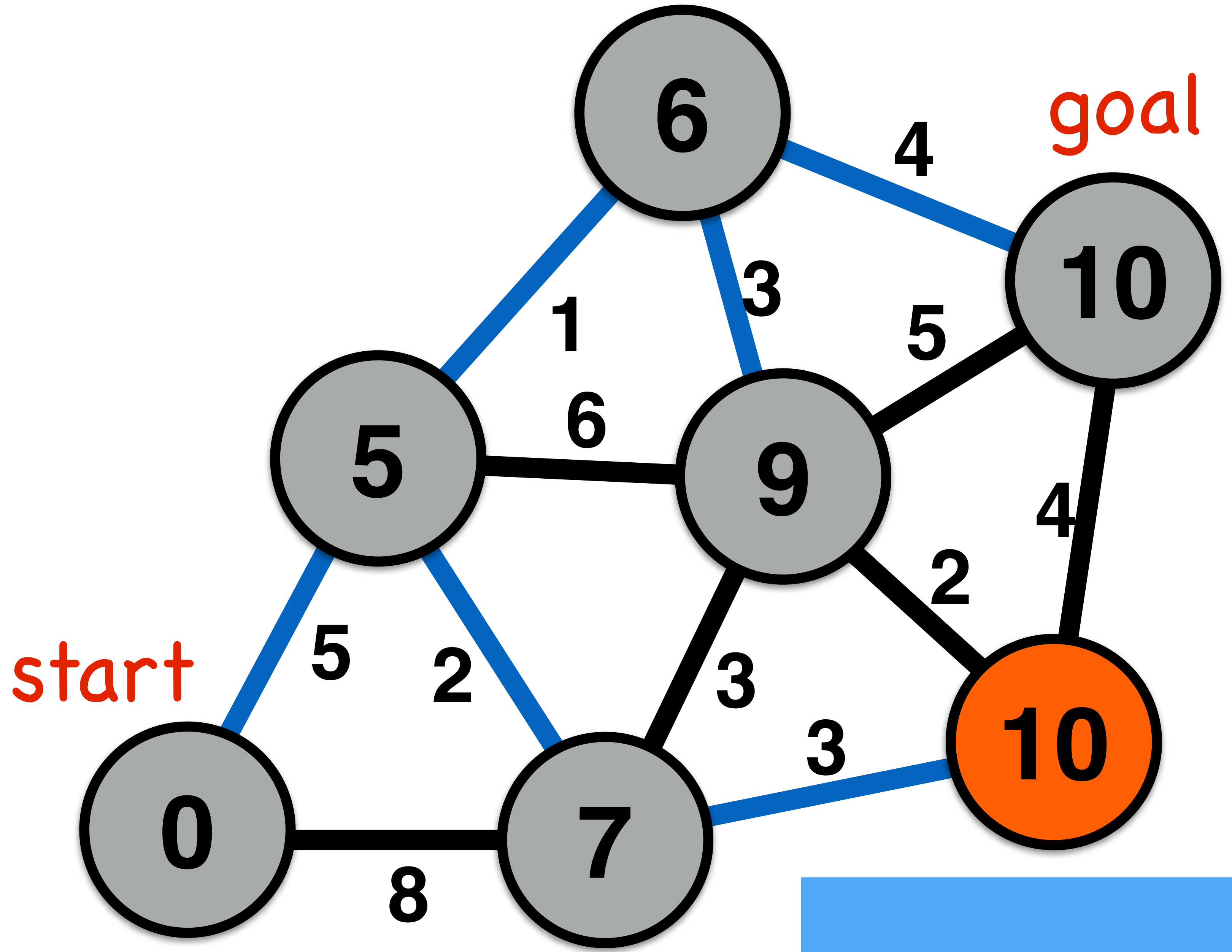
                $dist_{nbr}$ ← $dist_{cur\_node}$ + distance(nbr,cur_node)

            end if

        end for loop

    end while loop

output ← parent, distance



empty queue

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

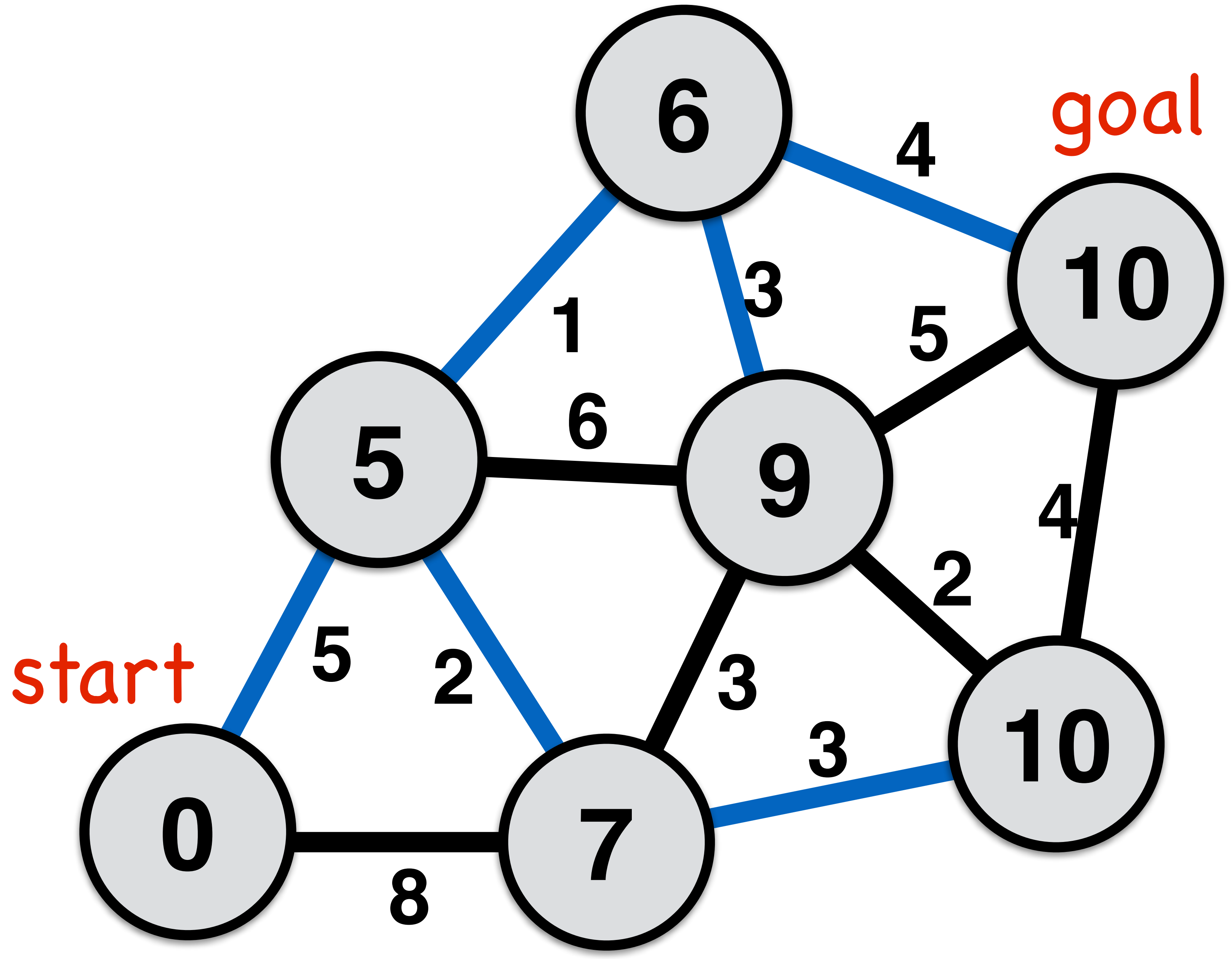
                $dist_{nbr}$ ← $dist_{cur\_node}$ + distance(nbr,cur_node)

            end if

        end for loop

    end while loop

output ← parent, distance



<u>Dijkstra shortest path algorithm</u>

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)

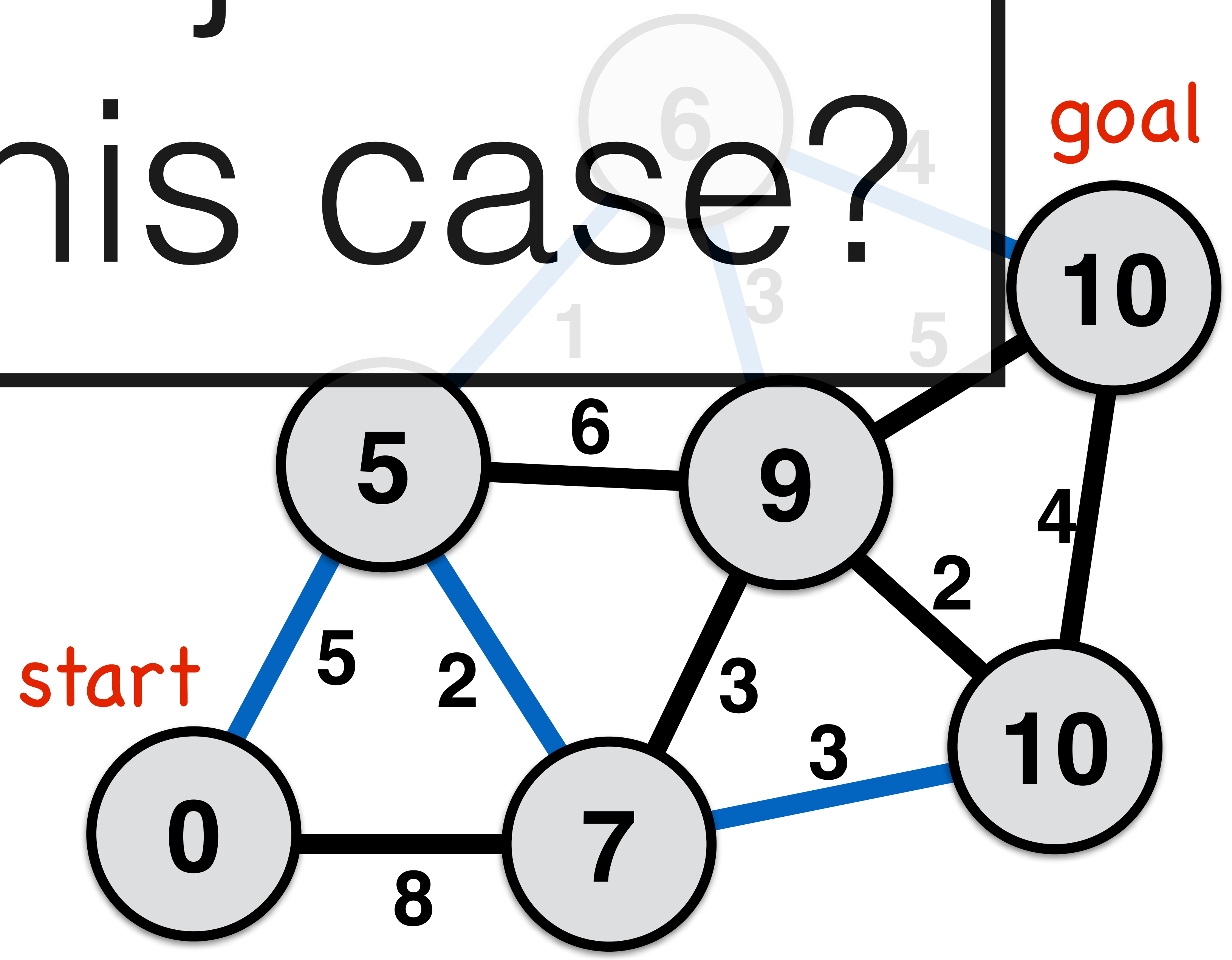
end if

end for loop

end while loop

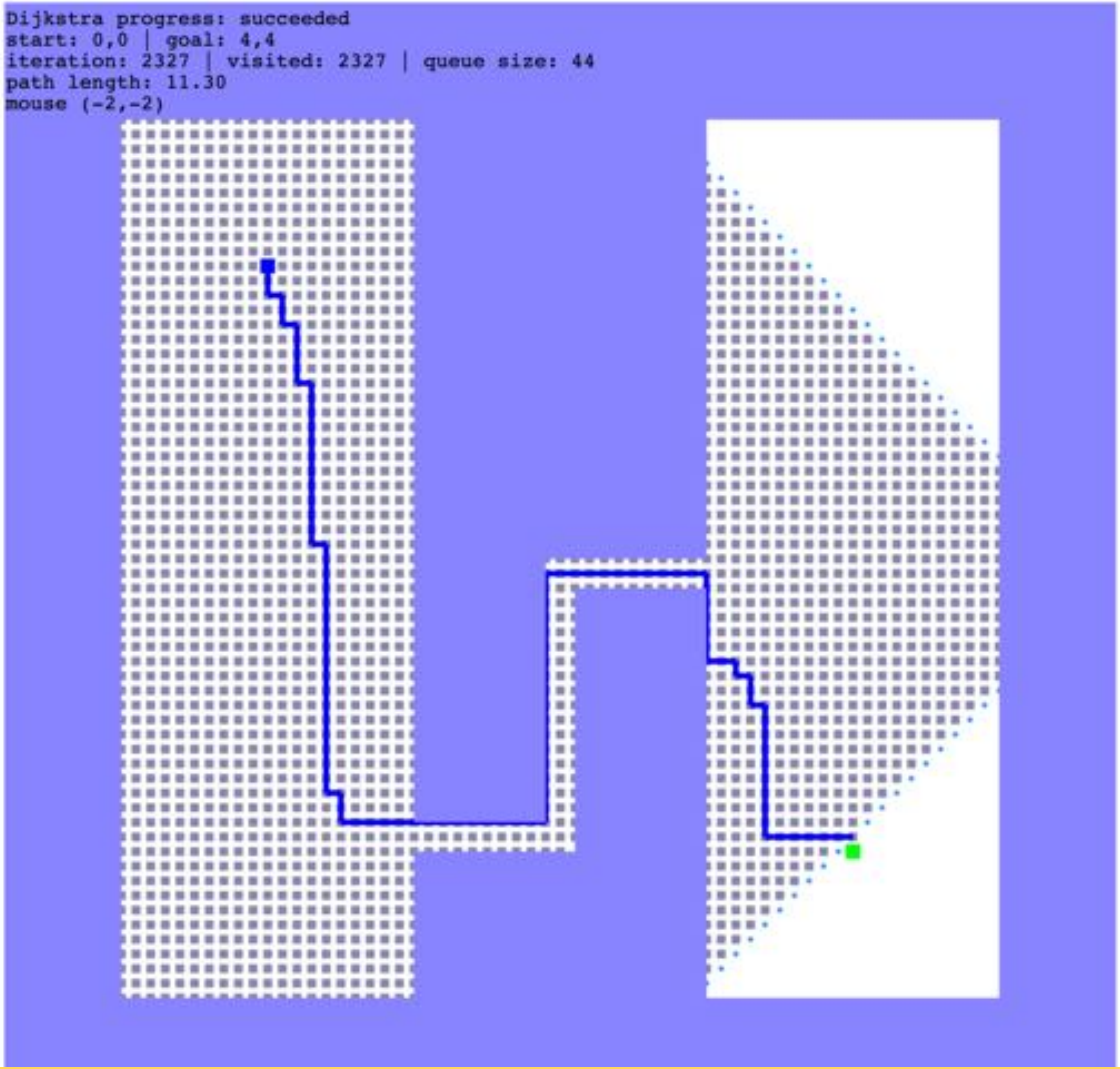
output ← parent, distance

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



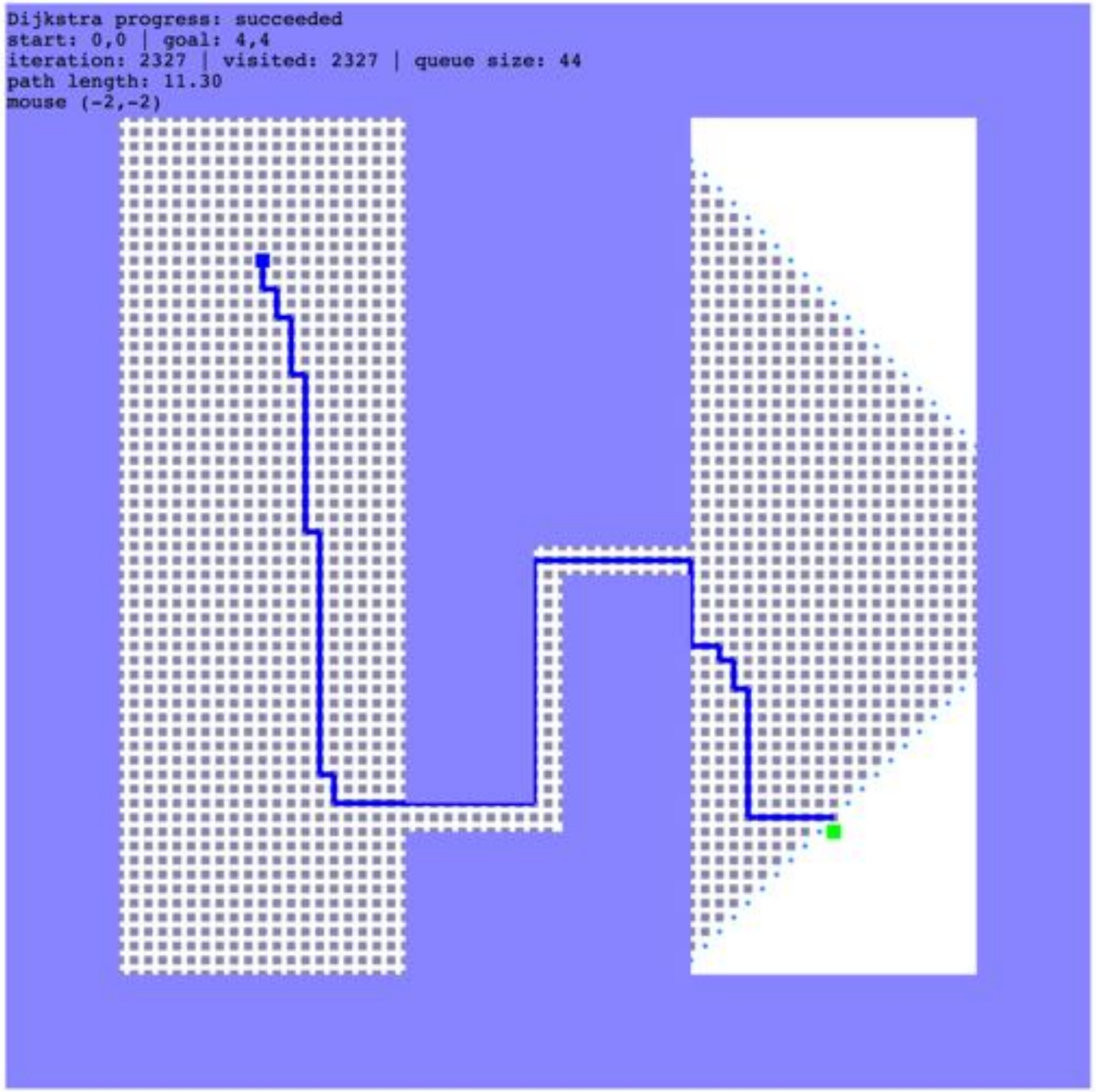
What will search with Dijkstra's algorithm look like in this case?

Slide borrowed from Michigan Robotics autorob.org

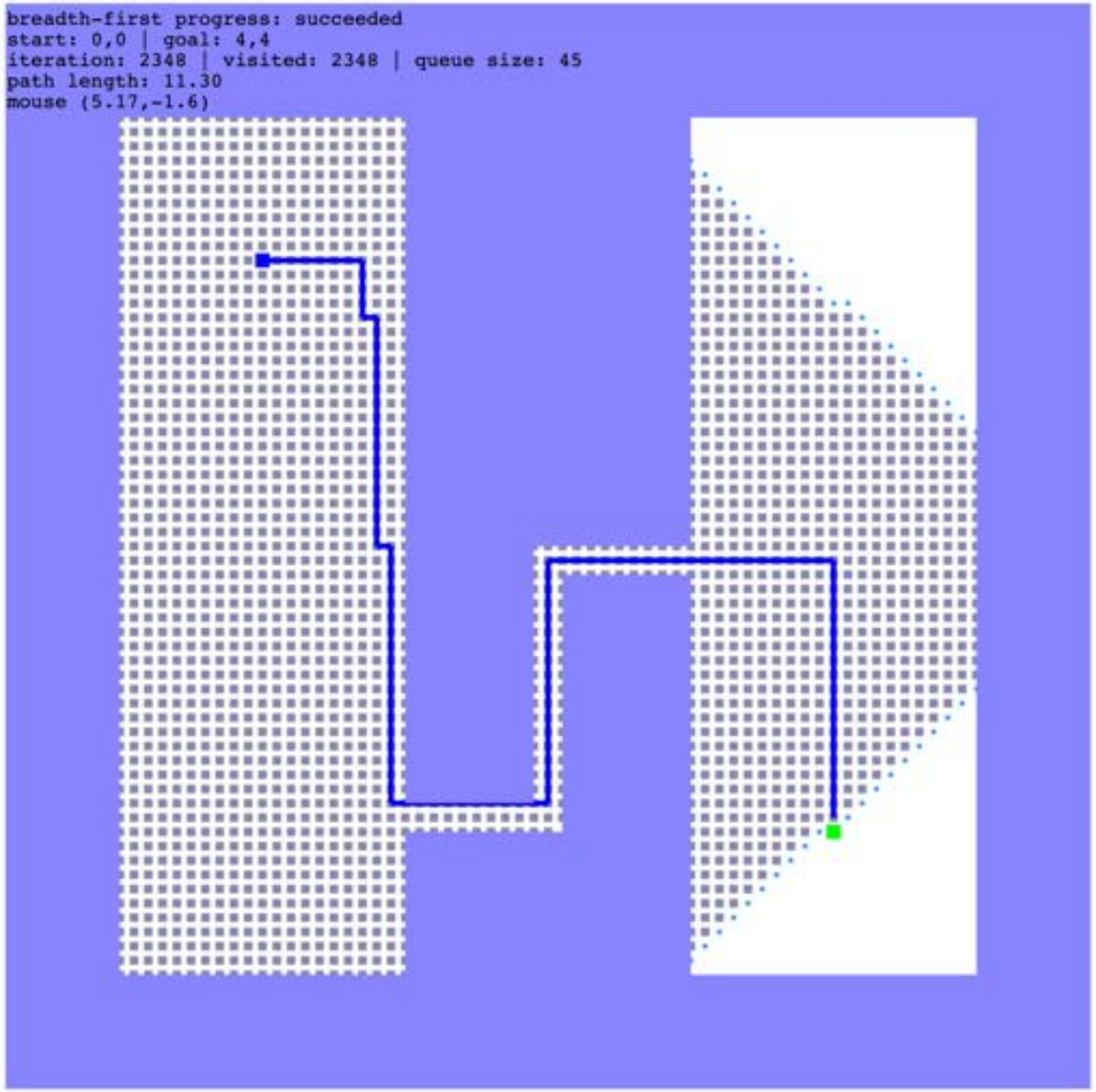


Slide borrowed from Michigan Robotics autorob.org

Dijkstra



BFS



Why does their visit pattern look similar?

A-star Algorithm

A Formal Basis for the Heuristic Determination of Minimum Cost Paths

PETER E. HART, MEMBER, IEEE, NILS J. NILSSON, MEMBER, IEEE, AND BERTRAM RAPHAEL

Abstract—Although the problem of determining the minimum cost path through a graph arises naturally in a number of interesting applications, there has been no underlying theory to guide the development of efficient search procedures. Moreover, there is no adequate conceptual framework within which the various ad hoc search strategies proposed to date can be compared. This paper describes how heuristic information from the problem domain can be incorporated into a formal mathematical theory of graph searching and demonstrates an optimality property of a class of search strategies.

I. Introduction

A. *The Problem of Finding Paths Through Graphs*

MANY PROBLEMS of engineering and scientific importance can be related to the general problem of finding a path through a graph. Examples of such problems include routing of telephone traffic, navigation through a maze, layout of printed circuit boards, and

Manuscript received November 24, 1967.
The authors are with the Artificial Intelligence Group of the Applied Physics Laboratory, Stanford Research Institute, Menlo Park, Calif.

mechanical theorem-proving and problem-solving. These problems have usually been approached in one of two ways, which we shall call the *mathematical approach* and the *heuristic approach.*

1) The mathematical approach typically deals with the properties of abstract graphs and with algorithms that prescribe an orderly examination of nodes of a graph to establish a minimum cost path. For example, Pollock and Wiebenson[1] review several algorithms which are guaranteed to find such a path for any graph. Busacker and Saaty[2] also discuss several algorithms, one of which uses the concept of dynamic programming.[3] The mathematical approach is generally more concerned with the ultimate achievement of solutions than it is with the computational feasibility of the algorithms developed.

2) The heuristic approach typically uses special knowledge about the domain of the problem being represented by a graph to improve the computational efficiency of solutions to particular graph-searching problems. For example, Gelernter's[4] program used Euclidean diagrams to direct the search for geometric proofs. Samuel[5] and others have used ad hoc characteristics of particular games to reduce

Hart, Nilsson, and Raphael
IEEE Transactions of System Science and Cybernetics, 4(2):100-107, 1968

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

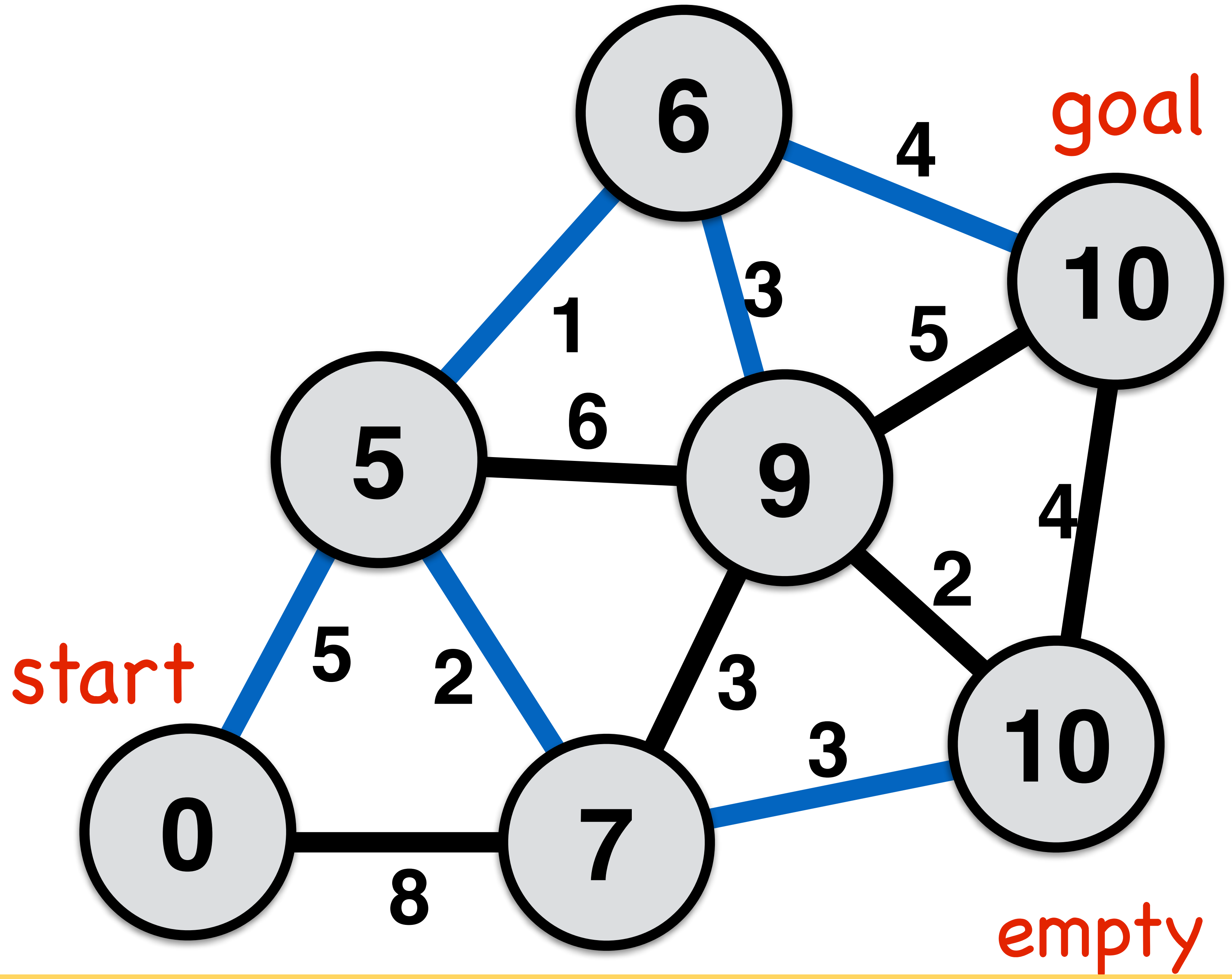
$dist_{nbr}$ ← $dist_{cur\_node}$ + distance(nbr,cur_node)

end if

end for loop

end while loop

output ← parent, distance



Slide borrowed from Michigan Robotics autorob.org

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)

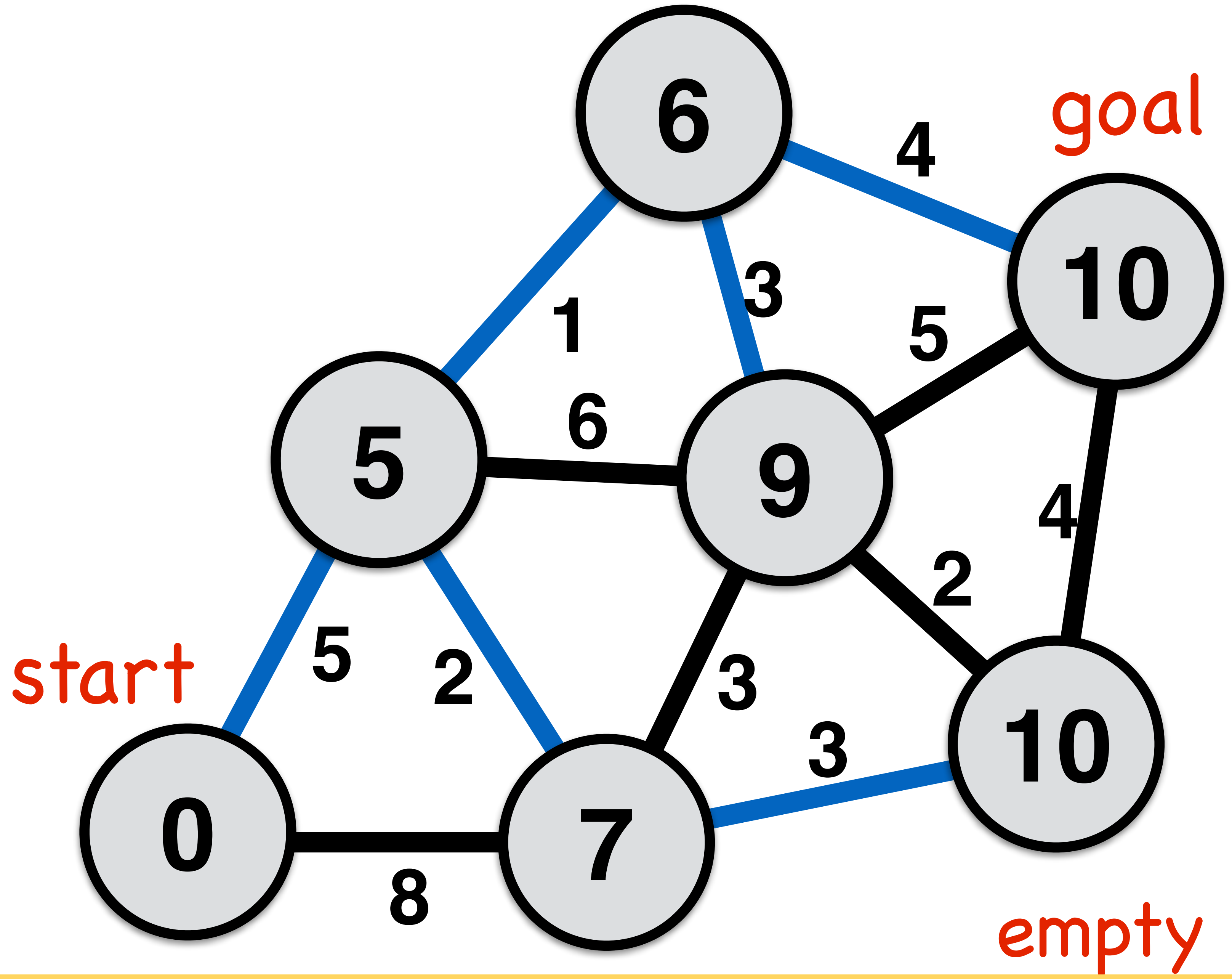
                f_score ← $distance_{nbr}$ + $line\_distance_{nbr,goal}$

            end if

        end for loop

    end while loop

output ← parent, distance



A-star shortest path algorithm

all nodes ← {$\text{dist}_{\text{start}}$ ← infinity, $\text{parent}_{\text{start}}$ ← none, $\text{visited}_{\text{start}}$ ← false}

start_node ← {$\text{dist}_{\text{start}}$ ← 0, $\text{parent}_{\text{start}}$ ← none, $\text{visited}_{\text{start}}$ ← true}

visit_queue ← start_node

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

    cur_node ← **dequeue(visit_queue, f_score)**

    $\text{visited}_{\text{cur\_node}}$ ← true

    for each nbr in not_visited(adjacent(cur_node))

        enqueue(nbr to visit_queue)

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

            $\text{parent}_{\text{nbr}}$ ← current_node

            dist_{nbr} ← $\text{dist}_{\text{cur\_node}}$ + distance(nbr,cur_node)

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

        end if

    end for loop

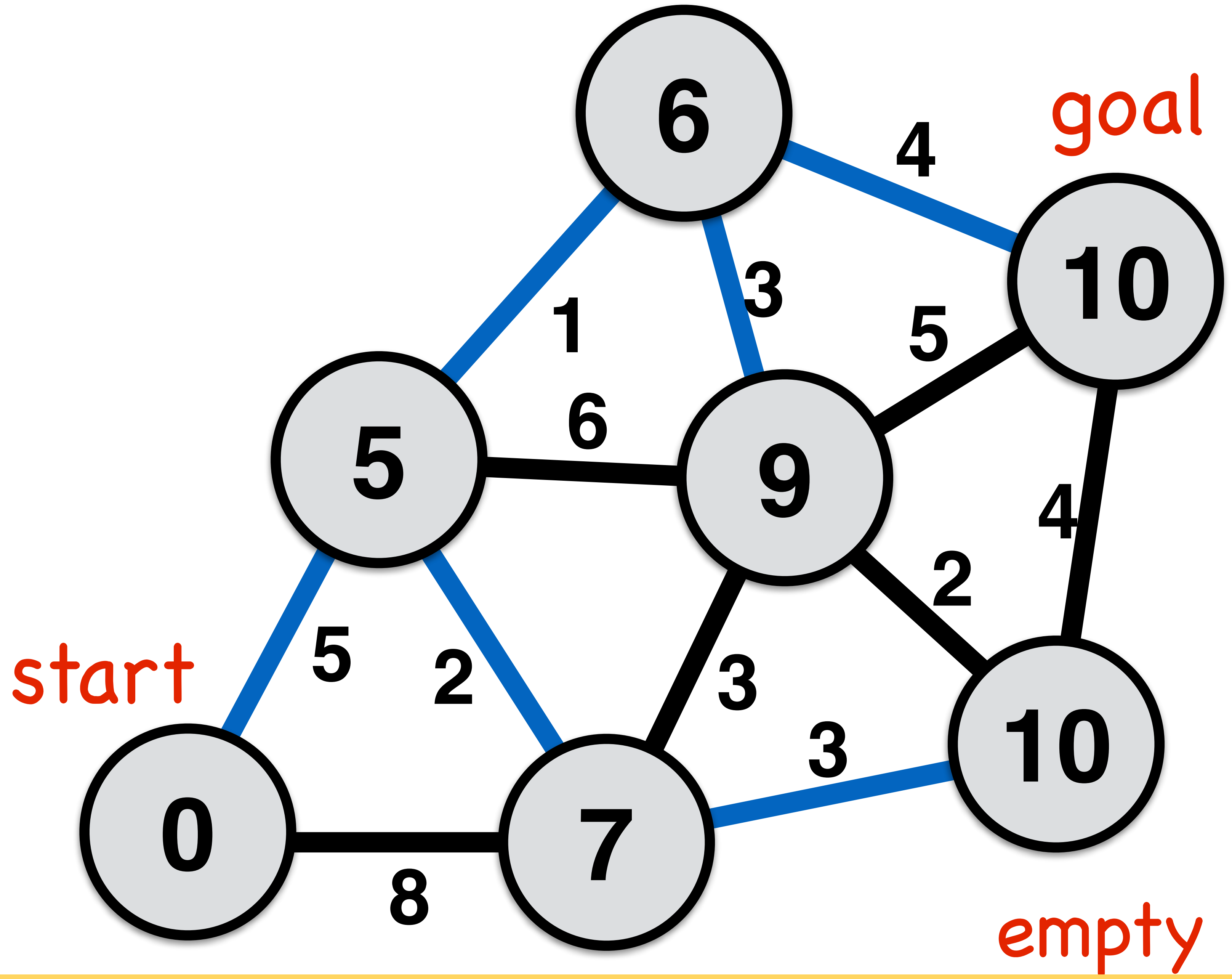
end while loop

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 ← {$\text{dist}_{\text{start}}$ ← infinity, $\text{parent}_{\text{start}}$ ← none, $\text{visited}_{\text{start}}$ ← false}

start_node ← {$\text{dist}_{\text{start}}$ ← 0, $\text{parent}_{\text{start}}$ ← none, $\text{visited}_{\text{start}}$ ← 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)

$\text{visited}_{\text{cur\_node}}$ ← true

for each nbr in not_visited(adjacent(cur_node))

enqueue(nbr to visit_queue)

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

$\text{parent}_{\text{nbr}}$ ← current_node

dist_{nbr} ← $\text{dist}_{\text{cur\_node}}$ + distance(nbr,cur_node)

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

end if

end for loop

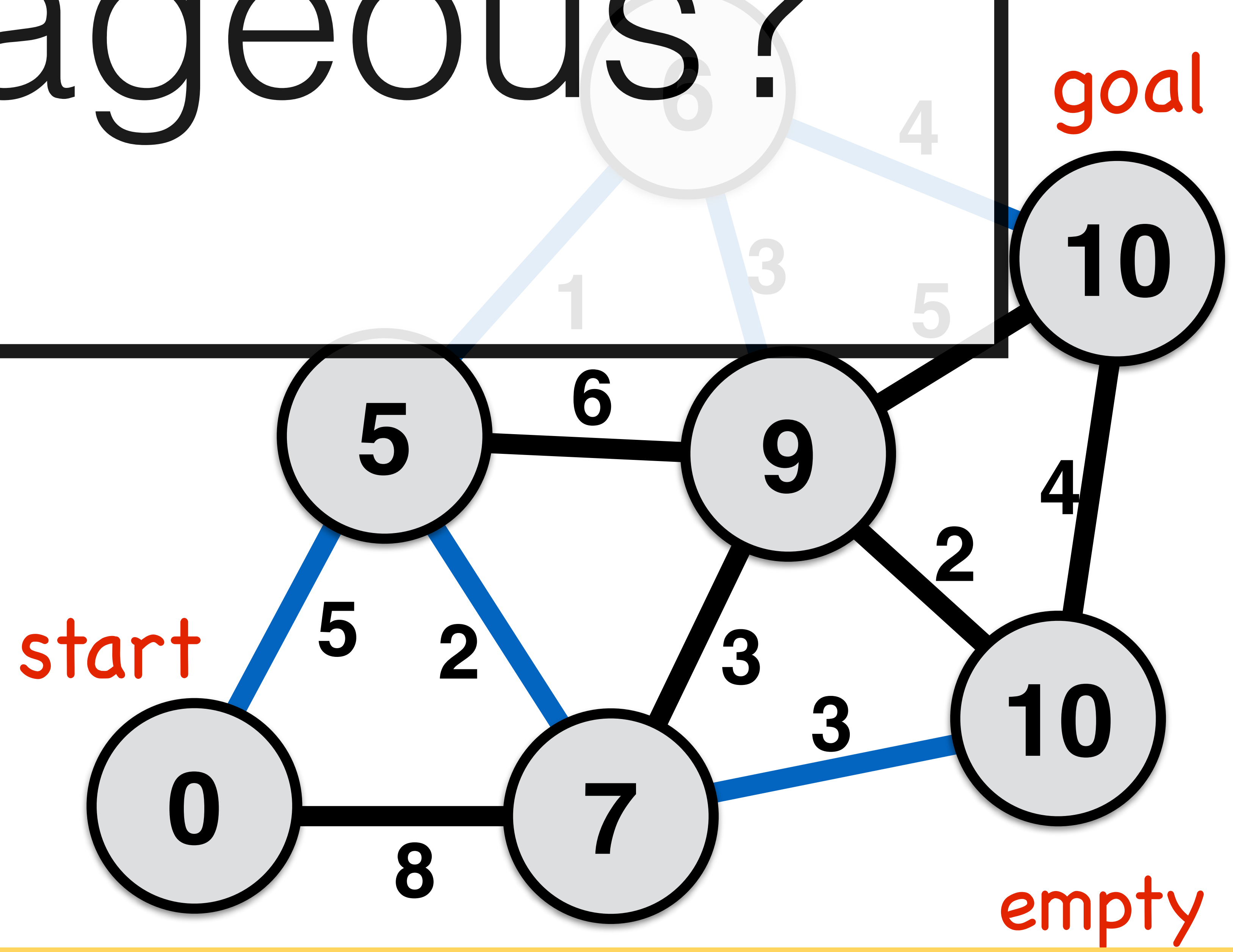
end while loop

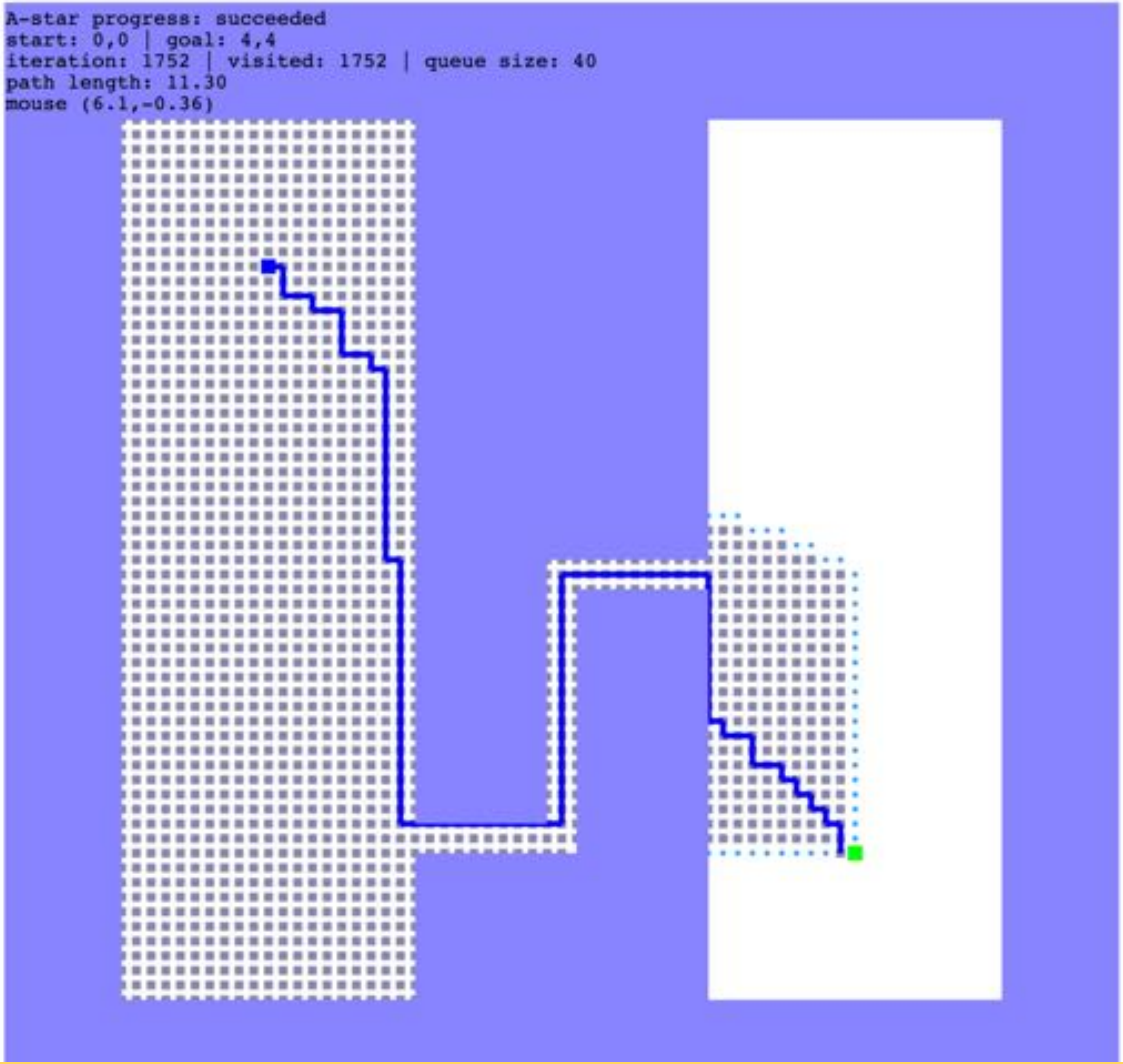
output ← parent, distance

g_score: distance along current path back to start

h_score: best possible distance to goal

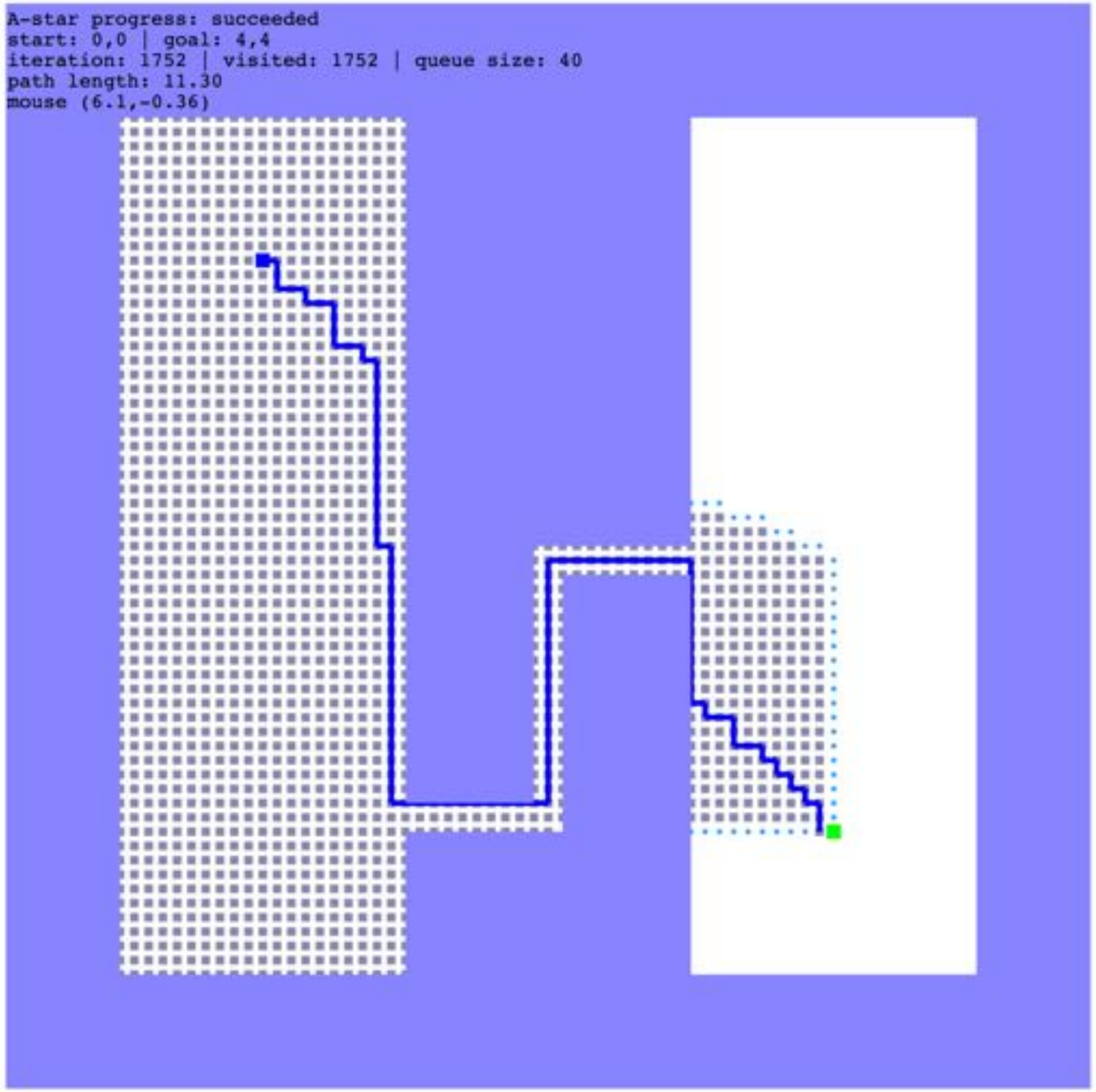
Why is A-star advantageous?



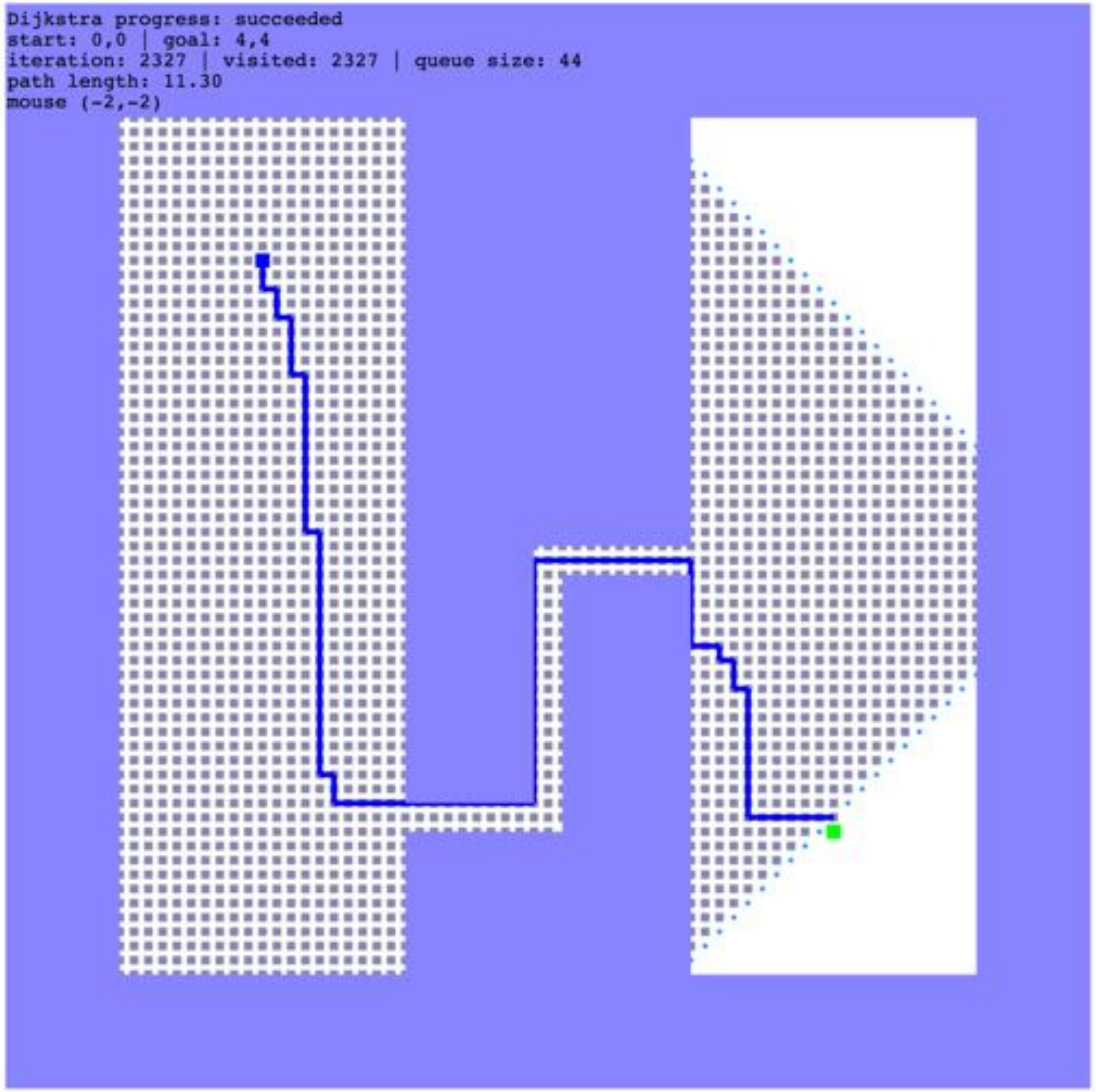


Slide borrowed from Michigan Robotics autorob.org

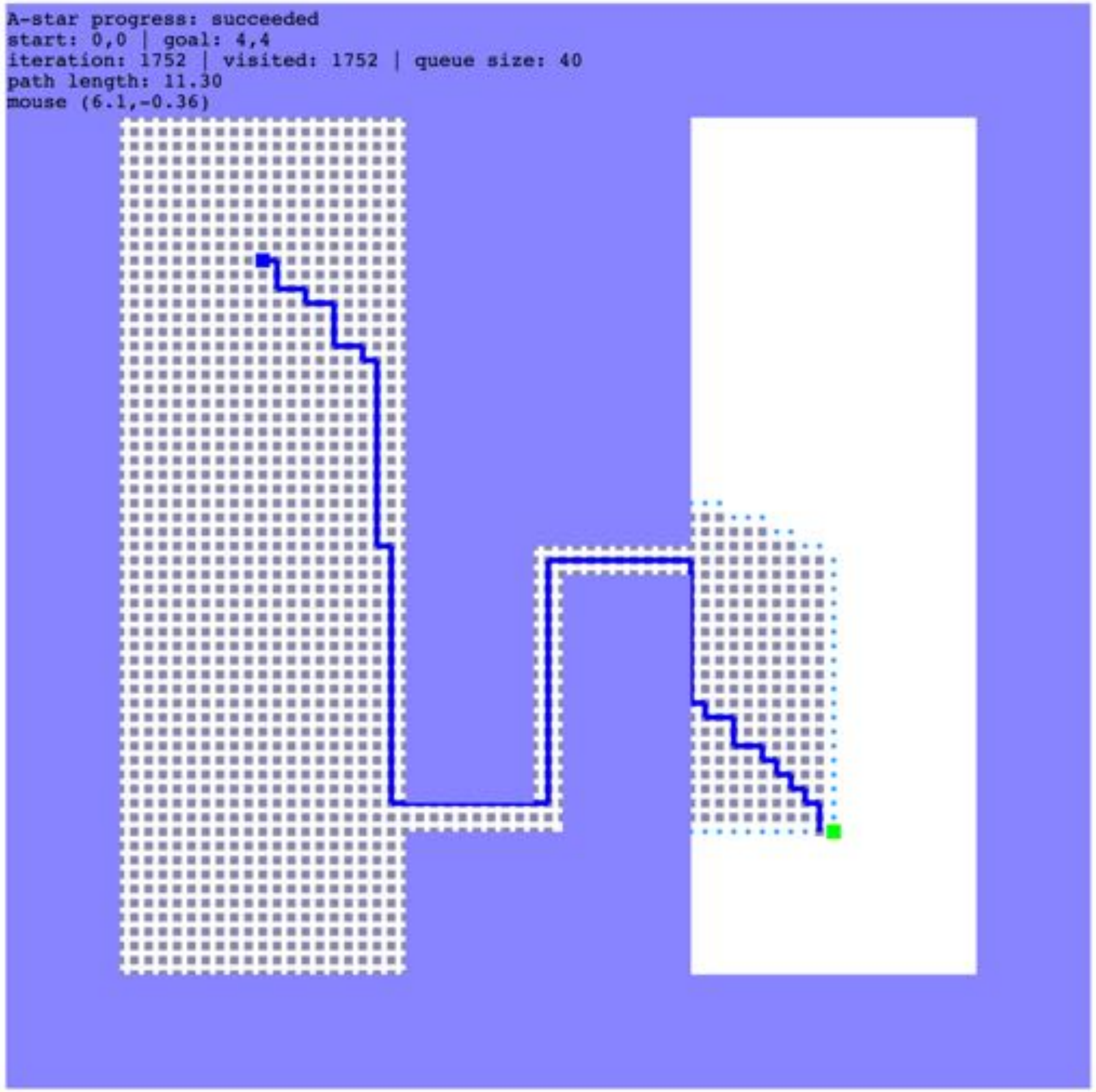
A-Star



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

Slide borrowed from Michigan Robotics autorob.org

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')

https://www.cs.cmu.edu/~./awm/tutorials/astar08.pdf

Proof: A* with Admissible Heuristic Guarantees Optimal Path

- Suppose it finds a suboptimal path, ending in goal state G_1 where $f(G_1) > f^*$ where $f^* = h^*(start)$ = cost of optimal path.
- There must exist a node n which is
  - Unexpanded
  - The path from start to n (stored in the BackPointers(n) values) is the start of a true optimal path

Why must such a node exist? Consider any optimal path s,n1,n2...goal. If all along it were expanded, the goal would've been reached along the shortest path.

- $f(n) >= f(G_1)$ (else search wouldn't have ended)
- Also $f(n) = g(n) + h(n)$

$= g^*(n) + h(n)$ because it's on optimal path

$<= g^*(n) + h^*(n)$ By the admissibility assumption

$= f^*$ Because n is on the optimal path

So $f^* >= f(n) >= f(G_1)$ contradicting top of slide

Slide 21

Next Lecture

Planning - II - Bug Algorithms