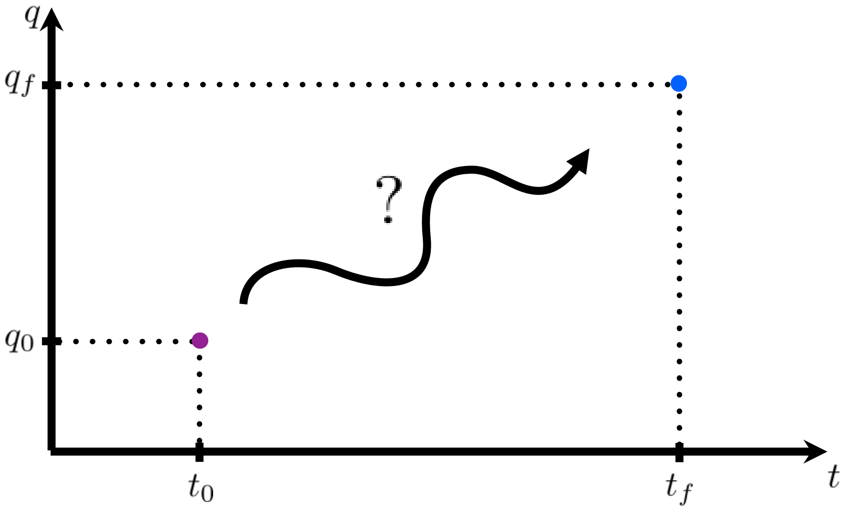
MEAM 520
Lecture 11: Trajectory Planning in Configuration Space

Cynthia Sung, Ph.D.
Mechanical Engineering & Applied Mechanics
University of Pennsylvania

Last Time: Trajectory Planning



First-Order Polynomial (Line)

$$q(t) = a_0 + a_1 t$$

Third-Order Polynomial (Cubic)

$$q(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3$$

Fifth-Order Polynomial (Quintic)

$$q(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5$$

Linear Segment with Parabolic Blends (LSPB, 1 Line + 2 Quadratics)

$$q(t) = b_0 + b_1 t + b_2 t^2 \quad q(t) = a_0 + a_1 t \quad q(t) = c_0 + c_1 t + c_2 t^2$$

Minimum Time Trajectory (Bang-Bang, 2 Quadratics)

$$q(t) = b_0 + b_1 t + b_2 t^2 \quad q(t) = c_0 + c_1 t + c_2 t^2$$

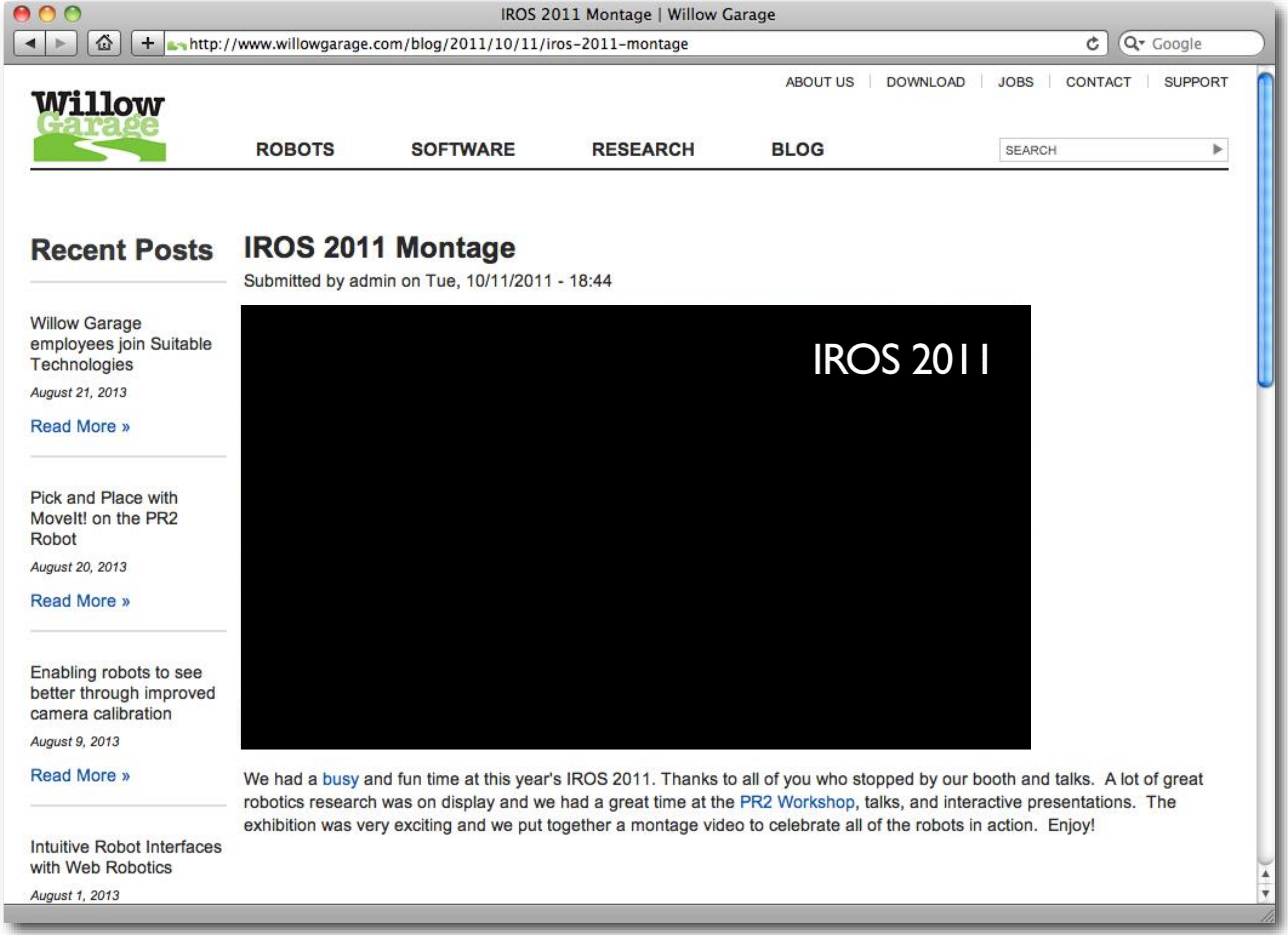
	Initial Conditions	Final Conditions
Position	$q(t_0) = q_0$	$q(t_f) = q_f$
Velocity	$\dot{q}(t_0) = v_0$	$\dot{q}(t_f) = v_f$
Acceleration	$\ddot{q}(t_0) = \alpha_0$	$\ddot{q}(t_f) = \alpha_f$
Jerk	$\dddot{q}(t_0) \neq \infty$	$\dddot{q}(t_f) \neq \infty$

Solving for Coefficients

$$\begin{bmatrix} q_0 \\ v_0 \\ q_f \\ v_f \end{bmatrix} = \begin{bmatrix} 1 & t_0 & {t_0}^2 & {t_0}^3 \\ 0 & 1 & 2t_0 & 3{t_0}^2 \\ 1 & t_f & {t_f}^2 & {t_f}^3 \\ 0 & 1 & 2t_f & 3{t_f}^2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

Outline of the next 2 weeks

- Last Time: Trajectory planning between two points in the absence of obstacles
- Today: Configuration space planning using grid-based methods (1960s)
- 10/8: Configuration space planning using sampling (1980s)
- 10/21: Lab 3 due (implement a planner)



http://www.willowgarage.com/blog/2011/10/11/iros-2011-montage

This Time: How do we find waypoints?

Path planning sounds simple, but it's among the most difficult problems in CS.

We want a complete algorithm: one that finds a solution whenever one exists and signals failure in finite time when no solution exists.

This is a **search** problem.

q

Configuration

complete specification of the location of every point on the robot (via joint variables)

$\mathcal{Q}$

Configuration Space

set of all possible configurations considering only joint limits

$\mathcal{W}$

Workspace

Cartesian space in which robot moves

$$\mathcal{O}_i$$

Obstacles
areas of the workspace that the robot should not occupy (physical objects or hazards)

Collision
when any part of the robot contacts an obstacle in the workspace

$$\mathcal{A}(q)$$

Robot
subset of the workspace occupied by the robot at configuration *q*

$$\mathcal{O} = \cup \mathcal{O}_i$$

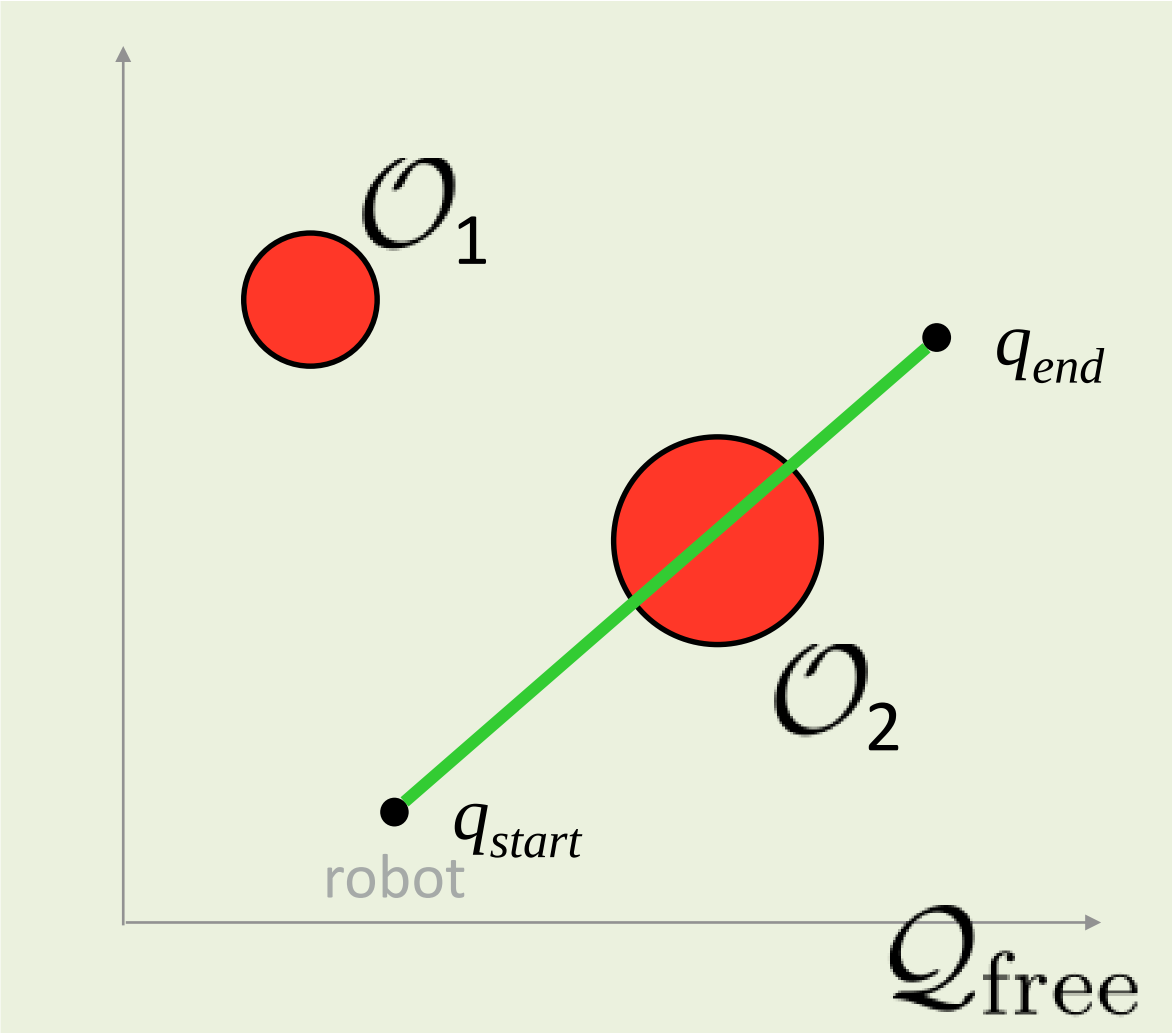
Configuration Space Obstacle
set of configurations for which the robot collides with an obstacle

$$\mathcal{QO} = \{q \in \mathcal{Q} \,|\, \mathcal{A}(q) \cap \mathcal{O} \neq \emptyset\}$$

Free Configuration Space
set of all collision-free configurations

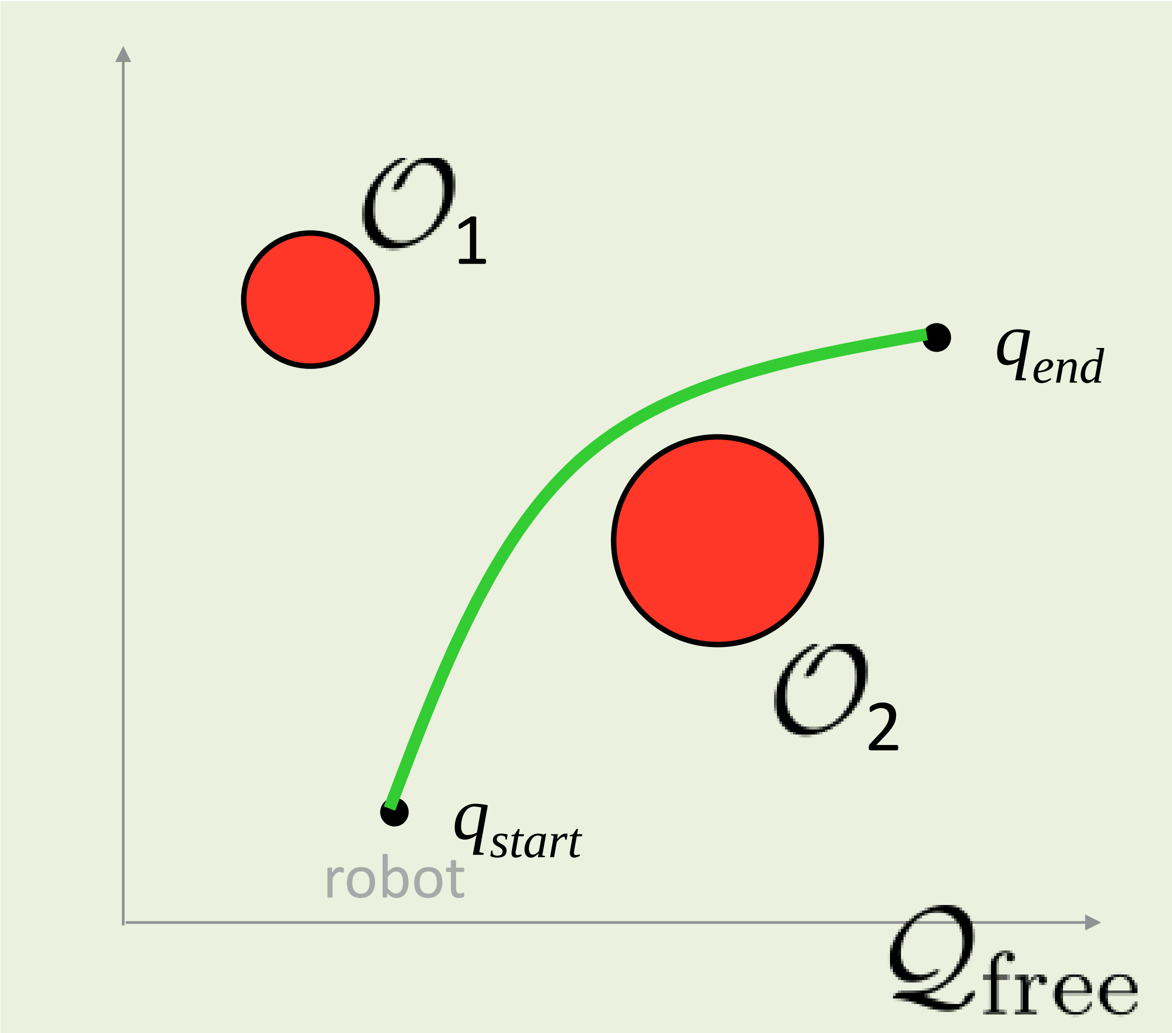
$$\mathcal{Q}_{\text{free}} = \mathcal{Q} \setminus \mathcal{QO}$$

Point Robot in 2D



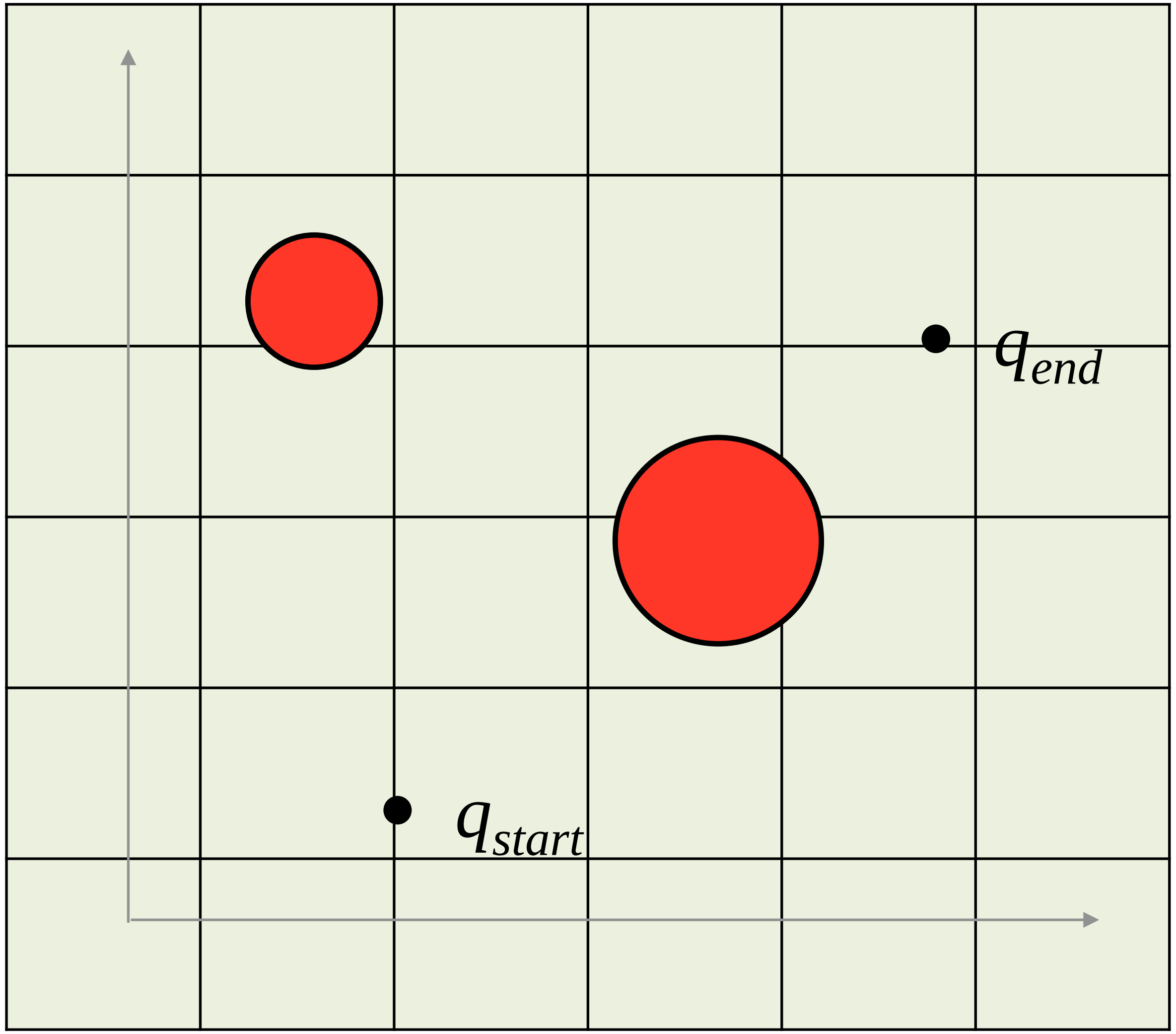
$$\mathcal{Q} = \mathcal{W} = \mathbb{R}^2$$

Point Robot in 2D



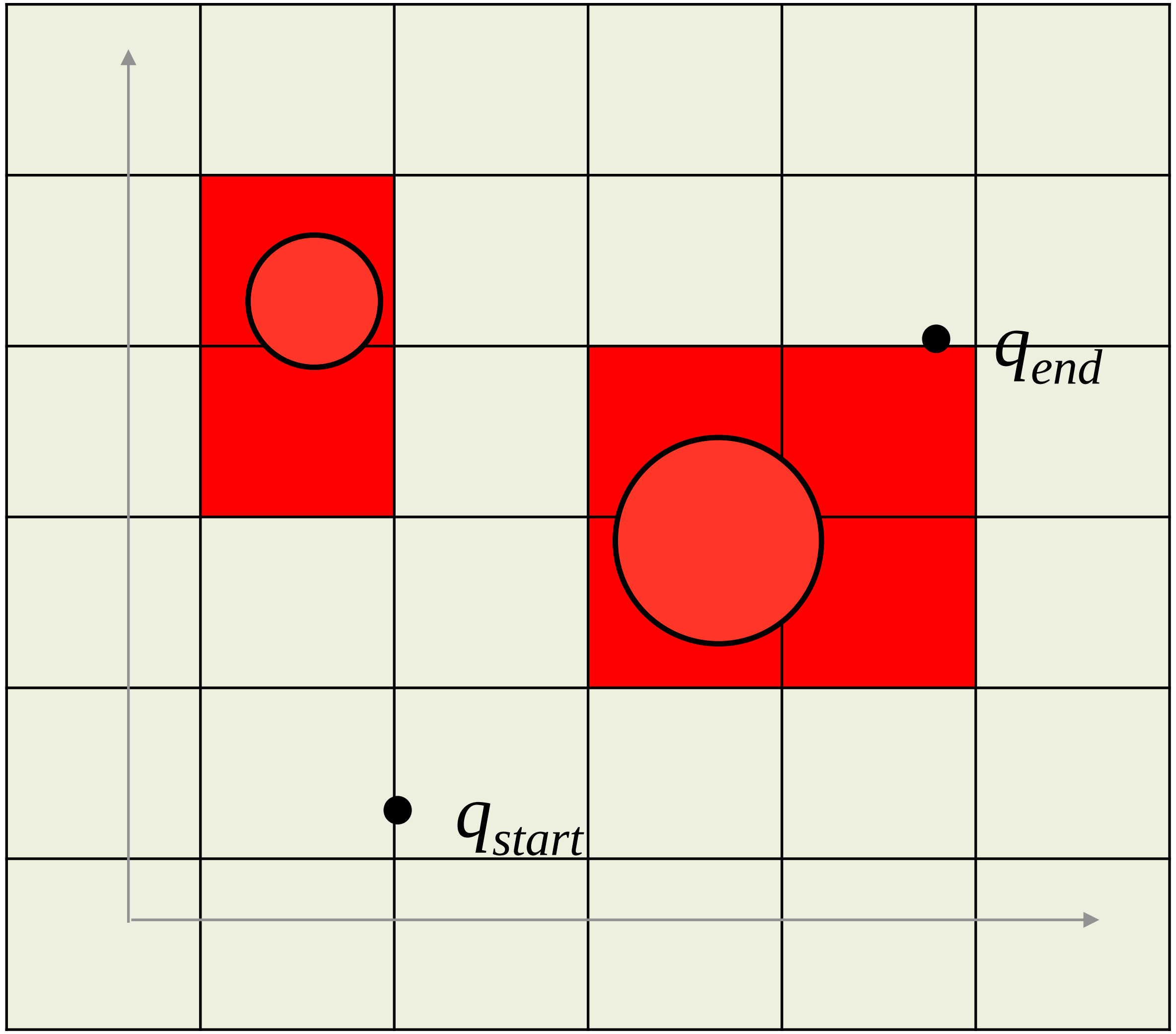
$$\mathcal{Q} = \mathcal{W} = \mathbb{R}^2$$

Discretize Space



n x n grid

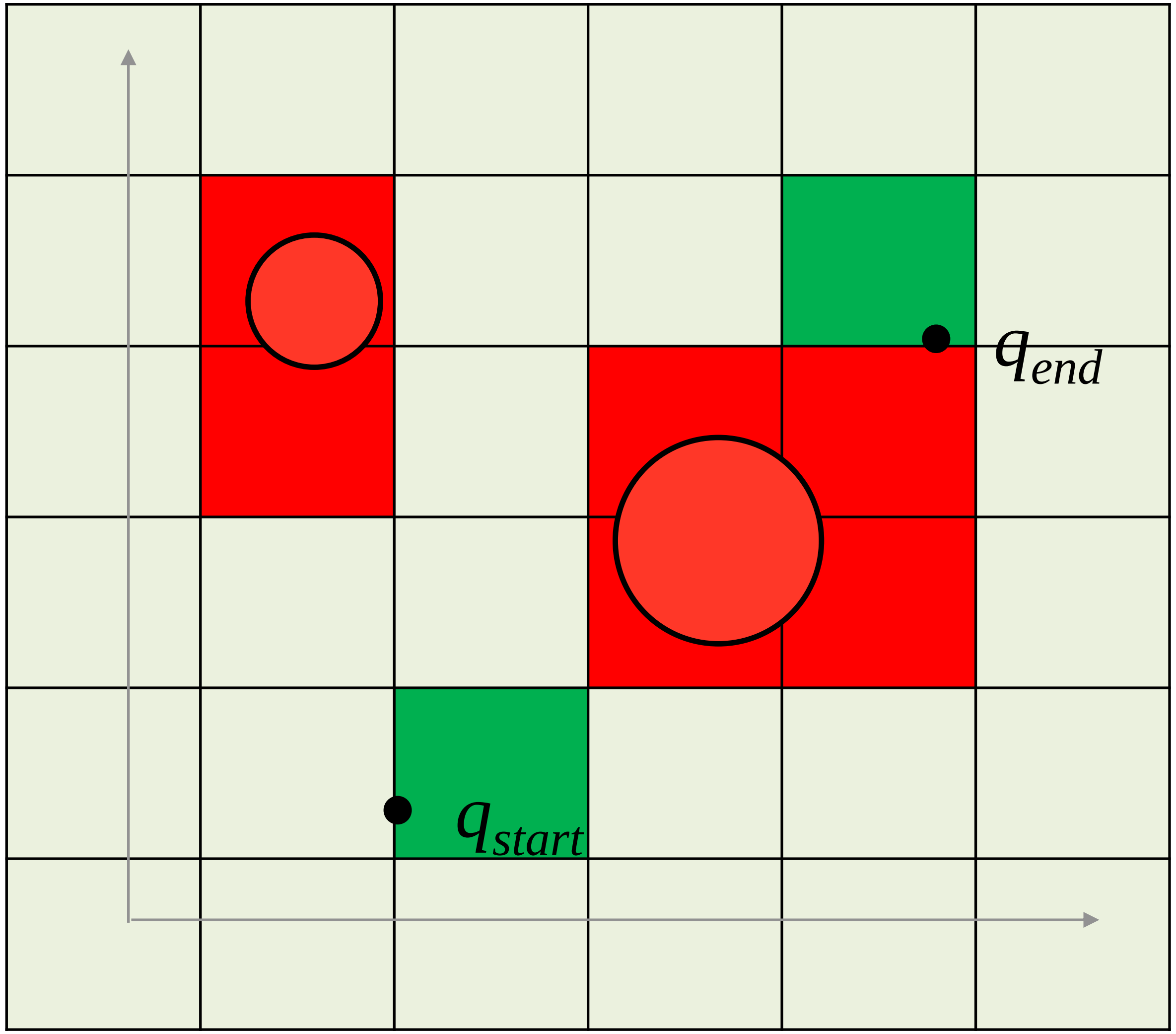
Discretize Space



n x n grid

Remove obstacles

Discretize Space

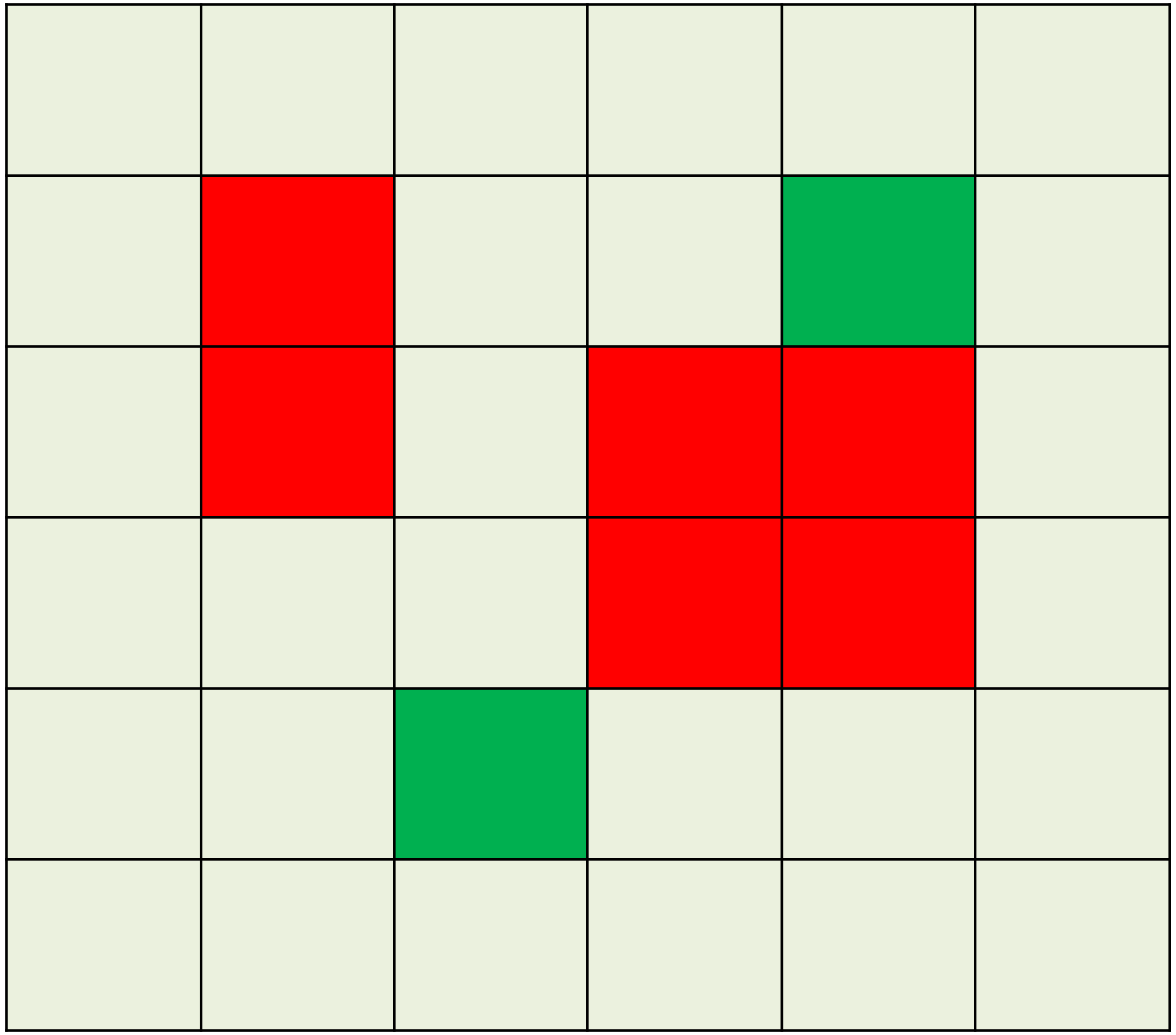


n x n grid

Remove obstacles

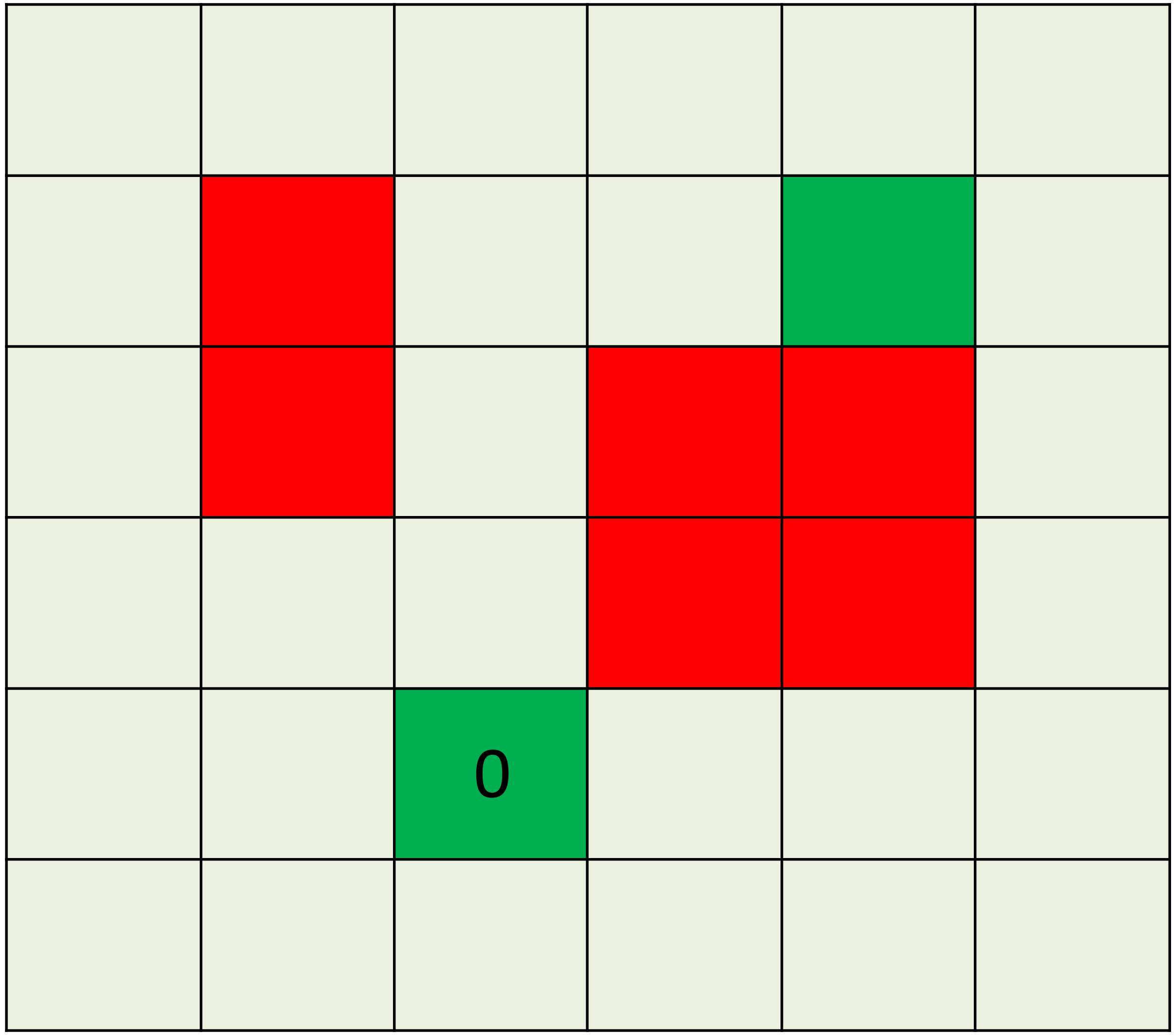
Find start and end cells

Wildfire



Pseudocode:

Wildfire

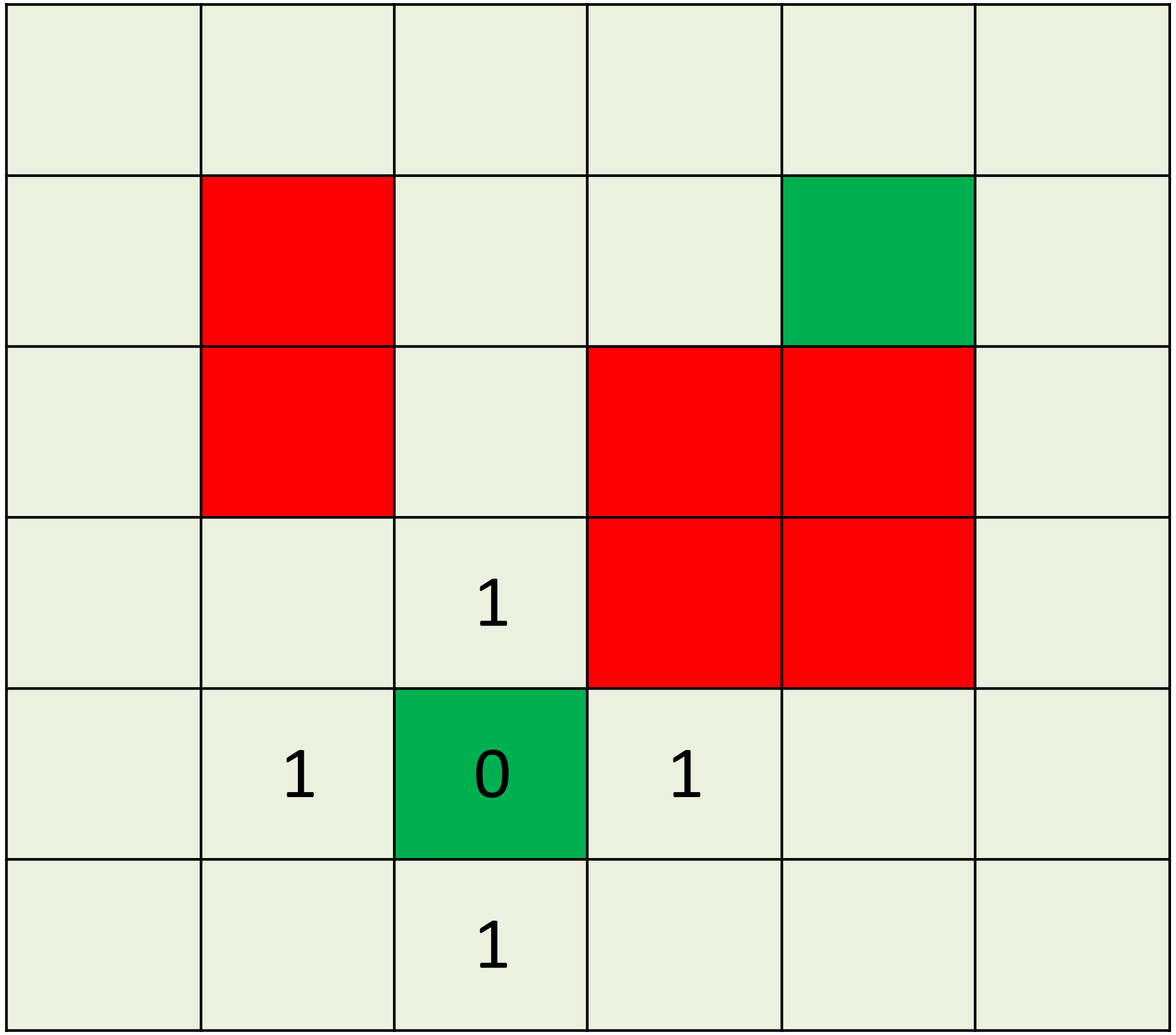


Pseudocode:

Start with i = 0 steps at q_{start}

While exist(empty cells)

Wildfire



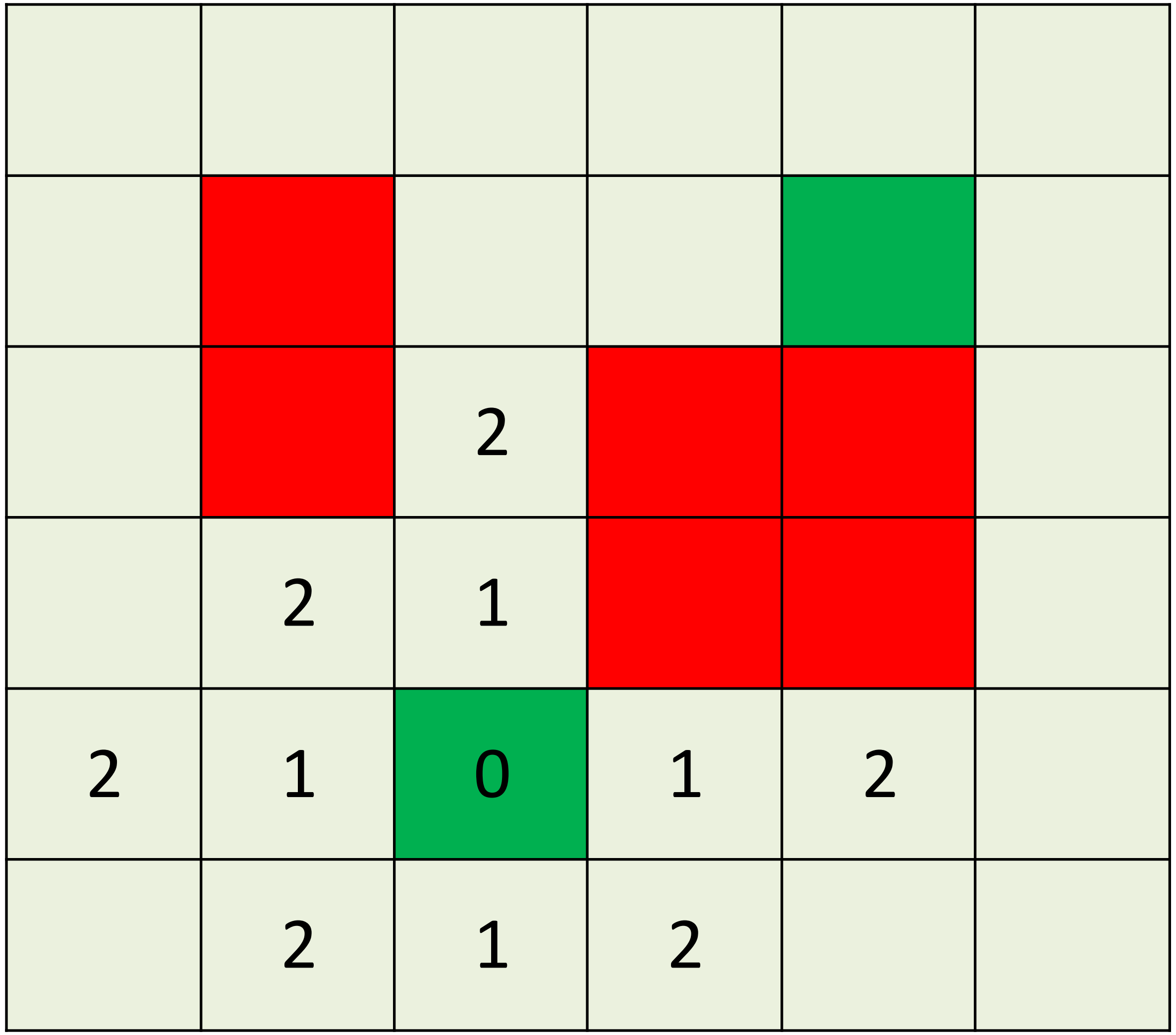
Pseudocode:

Start with i = 0 steps at q_{start}

While exist(empty cells)

  All neighbors have i+1 steps

Wildfire



Pseudocode:

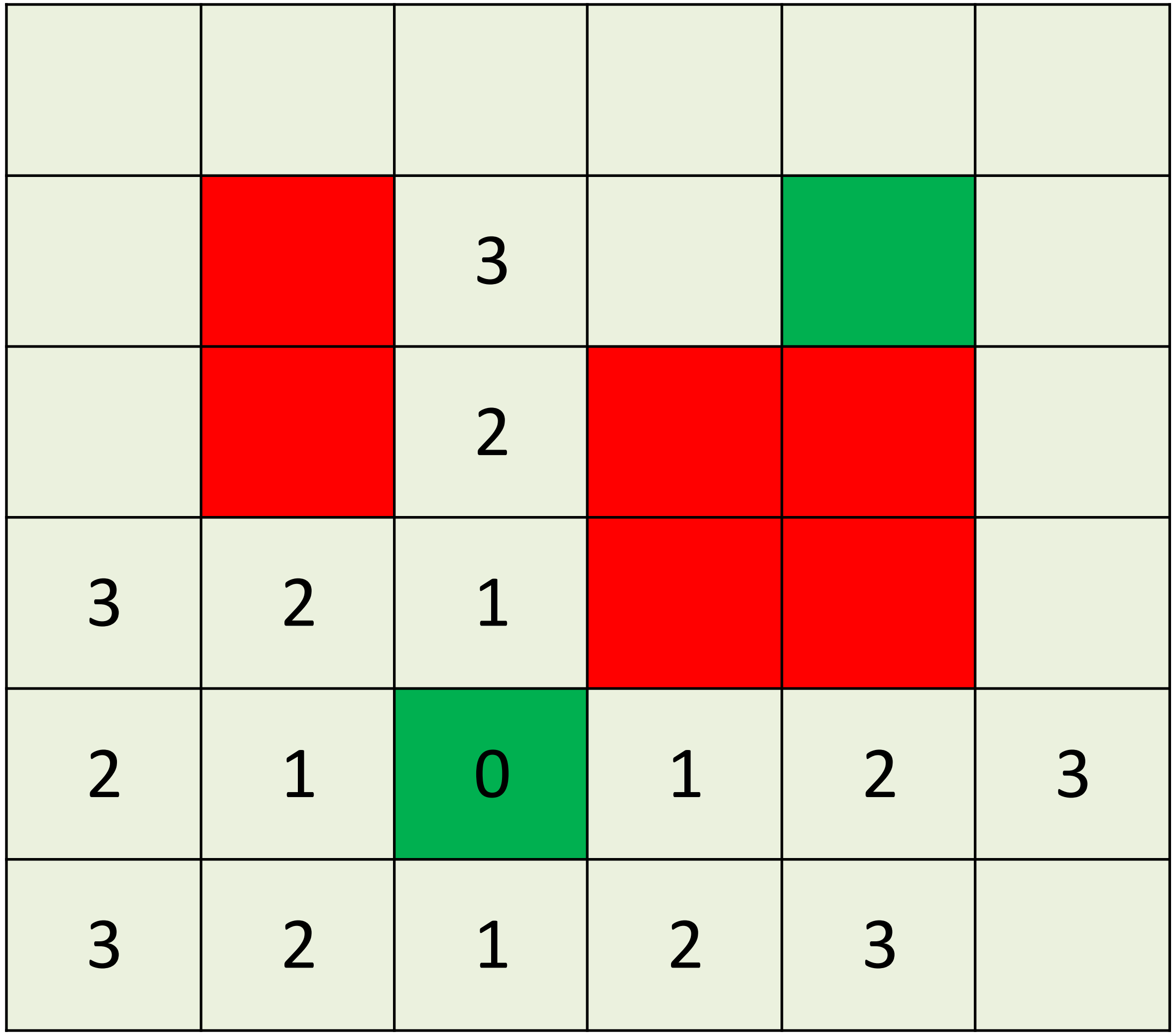
Start with i = 0 steps at q_{start}

While exist(empty cells)

  All neighbors have $i+1$ steps

  Ignore obstacle cells

Wildfire



Pseudocode:

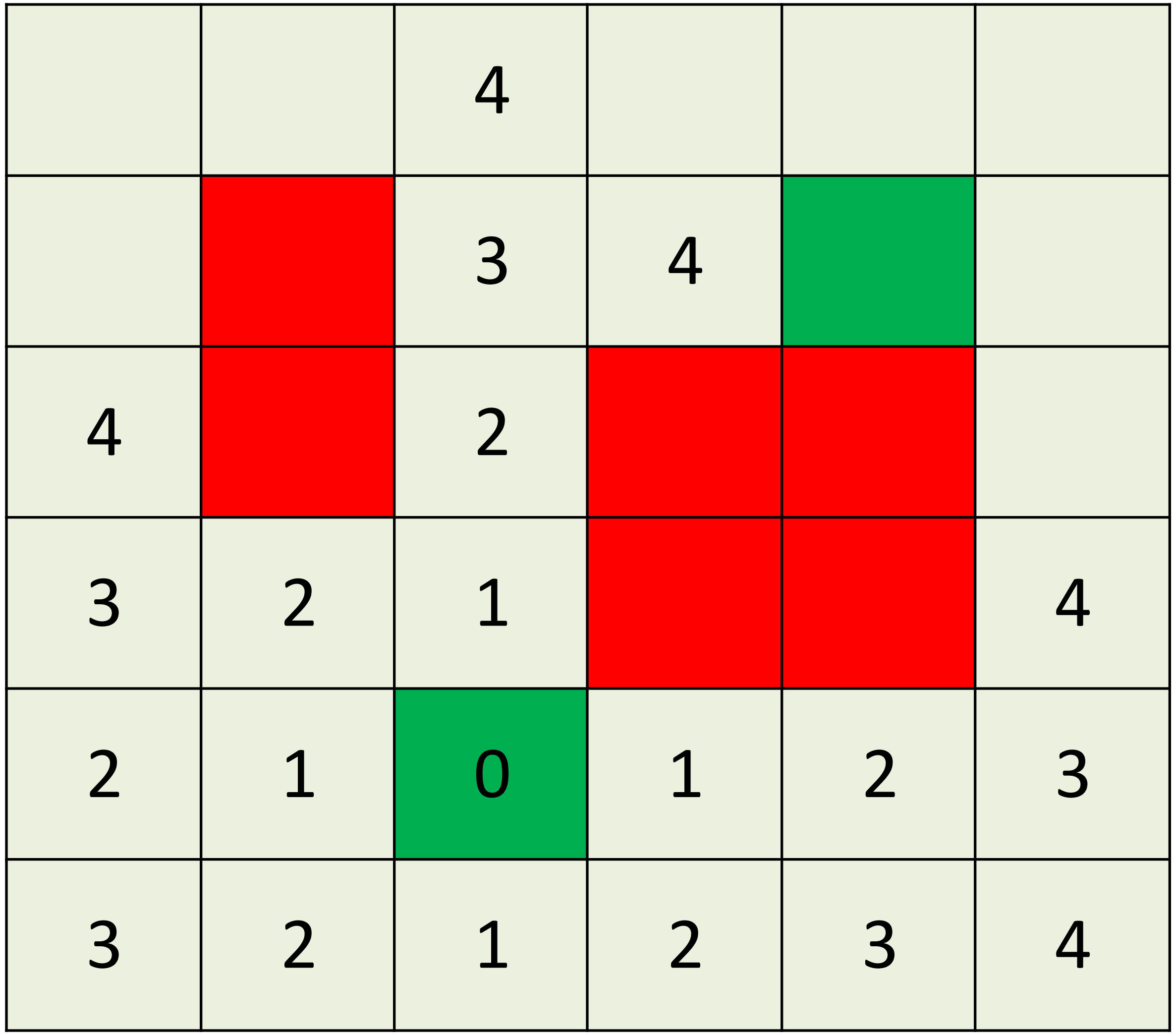
Start with i = 0 steps at q_{start}

While exist(empty cells)

  All neighbors have $i+1$ steps

  Ignore obstacle cells

Wildfire



Pseudocode:

Start with i = 0 steps at q_{start}

While exist(empty cells)

  All neighbors have $i+1$ steps

  Ignore obstacle cells

Wildfire

	5	4	5		
5		3	4	5	
4		2			5
3	2	1			4
2	1	0	1	2	3
3	2	1	2	3	4

Pseudocode:

Start with i = 0 steps at q_{start}

While exist(empty cells)

All neighbors have $i+1$ steps

Ignore obstacle cells

Wildfire

6	5	4	5	6	
5		3	4	5	6
4		2			5
3	2	1			4
2	1	0	1	2	3
3	2	1	2	3	4

Pseudocode:

Start with i = 0 steps at q_{start}

While exist(empty cells)

  All neighbors have $i+1$ steps

  Ignore obstacle cells

Wildfire

6	5	4	5	6	7
5		3	4	5	6
4		2			5
3	2	1			4
2	1	0	1	2	3
3	2	1	2	3	4

Pseudocode:

Start with i = 0 steps at q_{start}

While exist(empty cells)

  All neighbors have $i+1$ steps

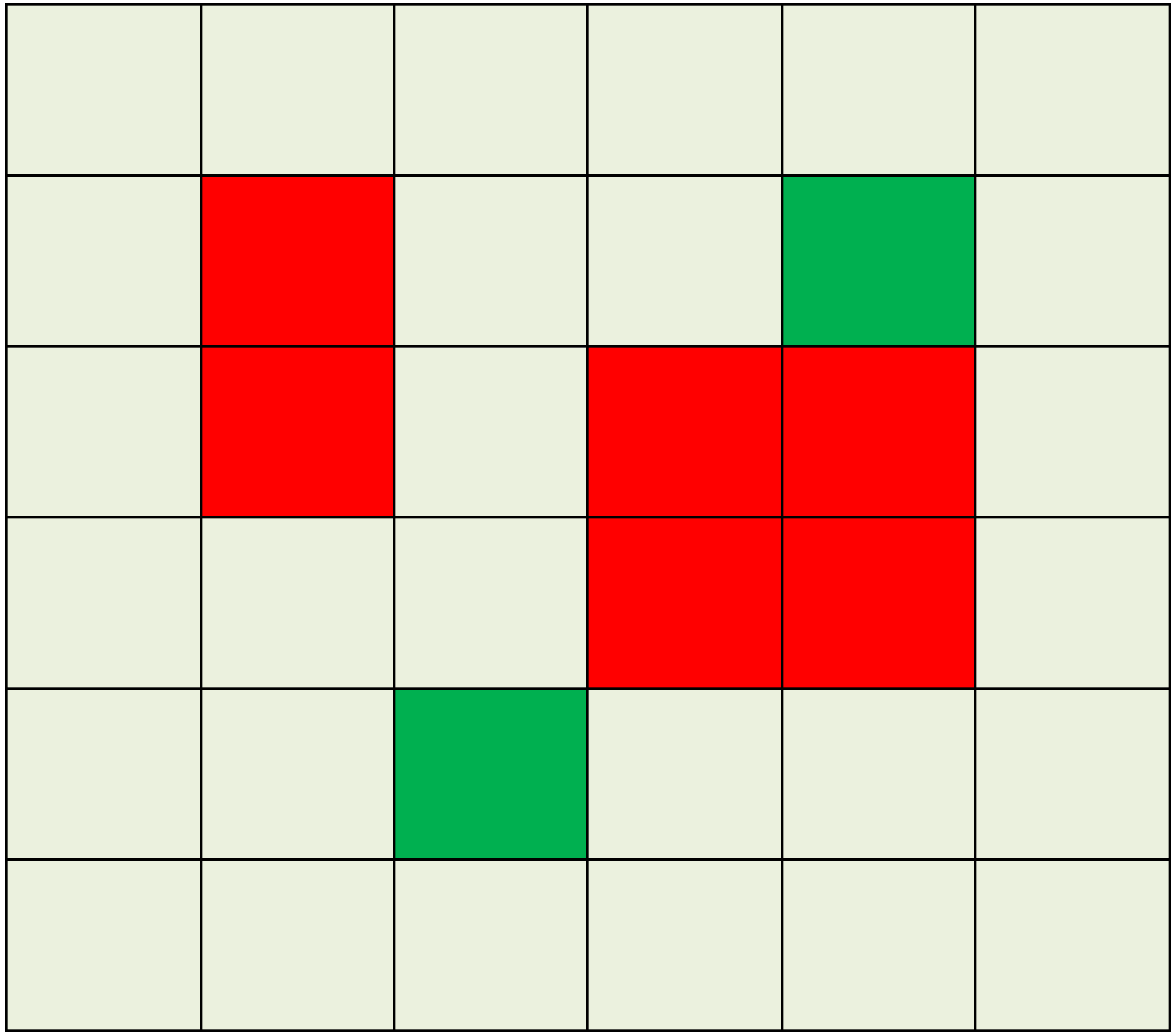
  Ignore obstacle cells

Search all cells:

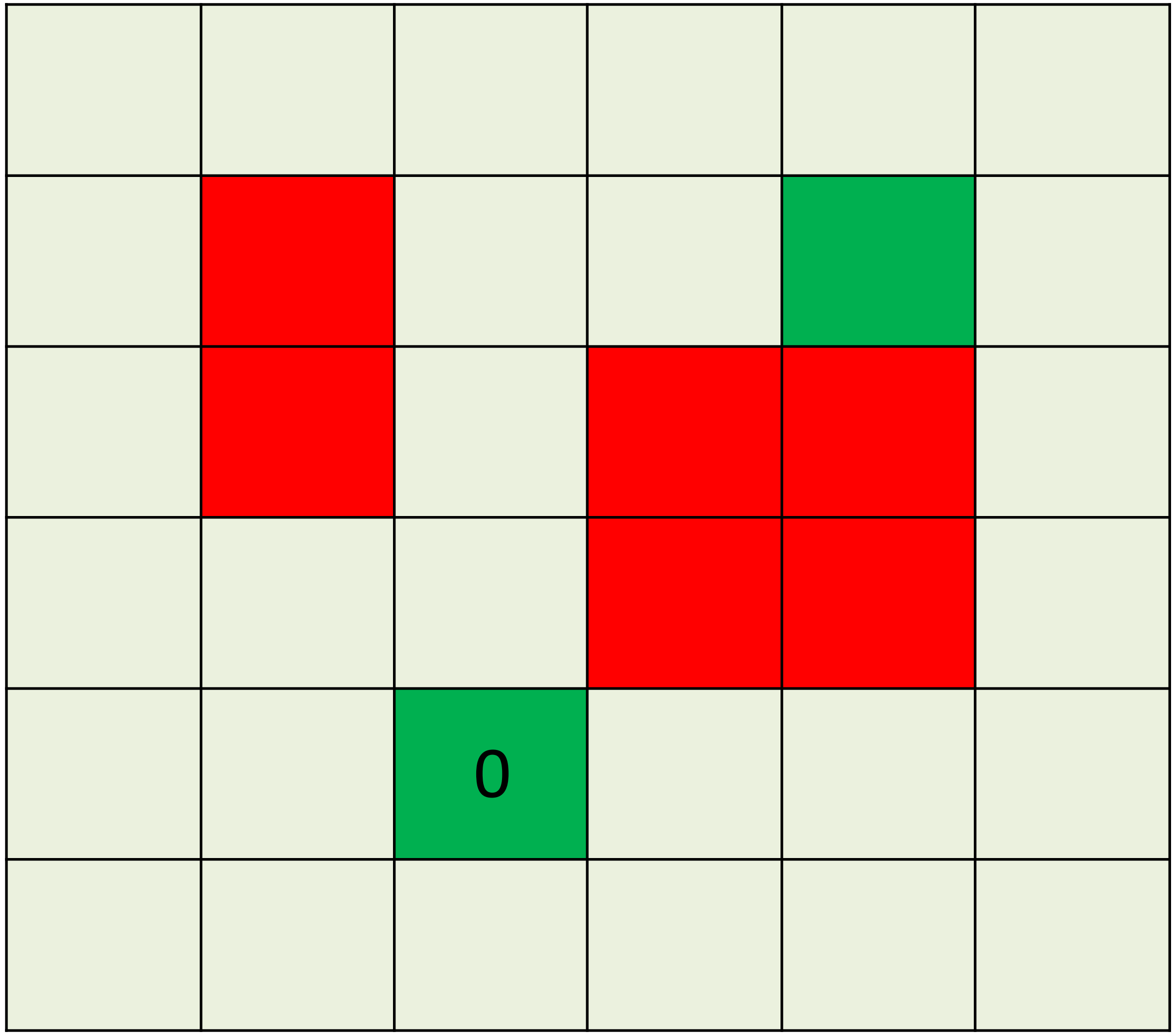
Computation is N_{cell}

Breadth First Search

Pseudocode:



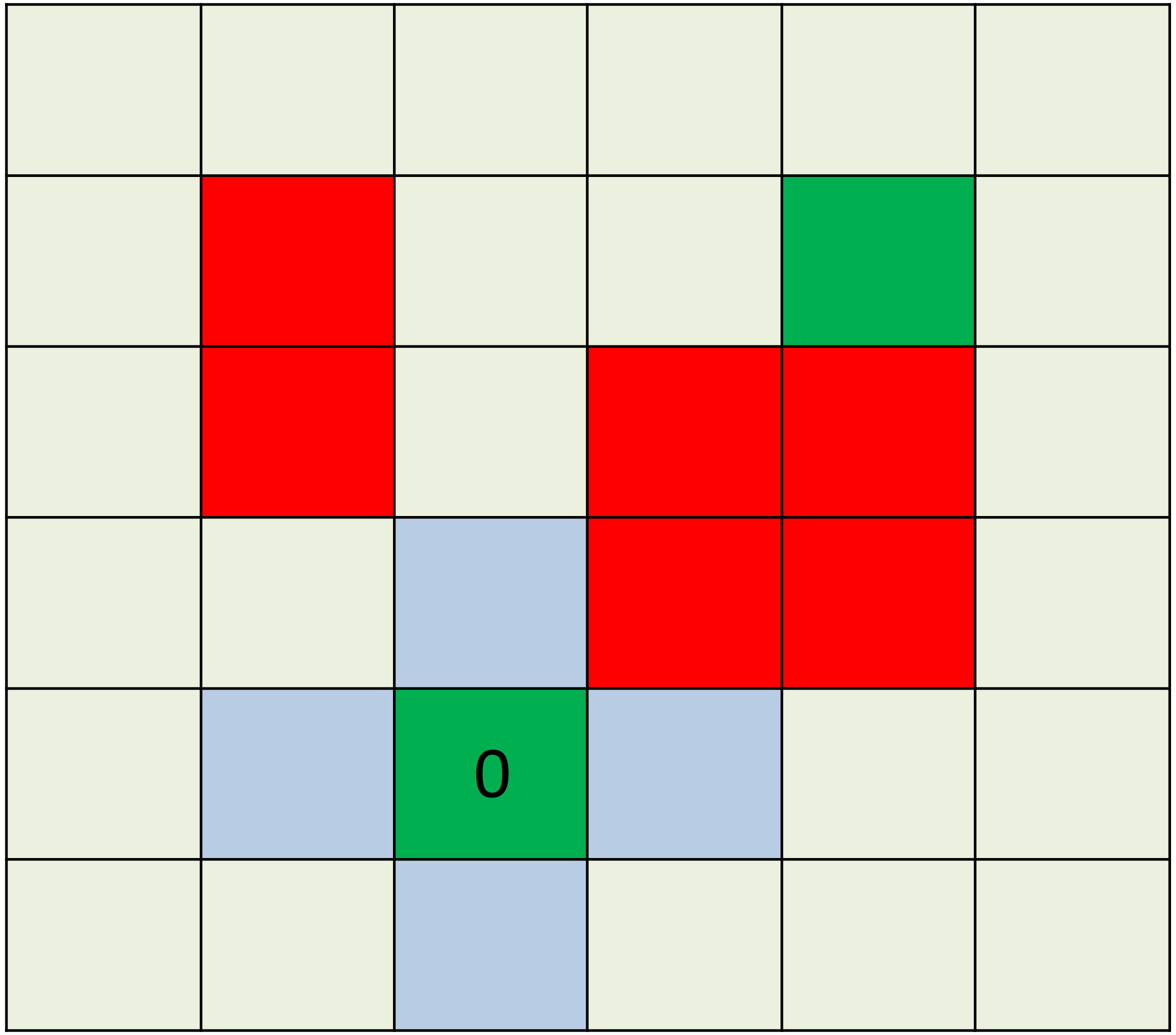
Breadth First Search



Pseudocode:

Start with i = 0 steps at q_{start}

Breadth First Search

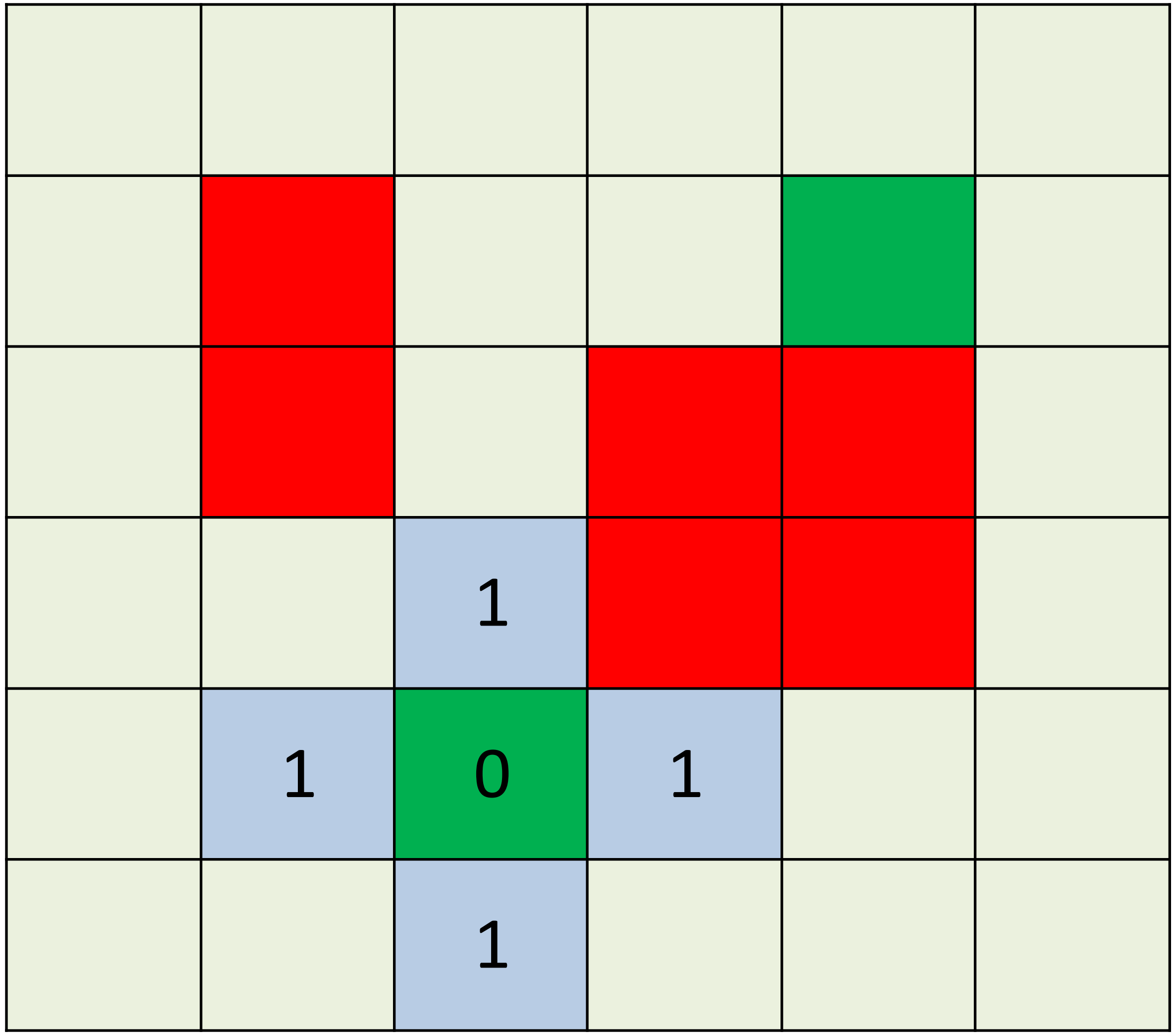


Pseudocode:

Start with i = 0 steps at q_{start}

$Queue$ = neighbors of q_{start}

Breadth First Search



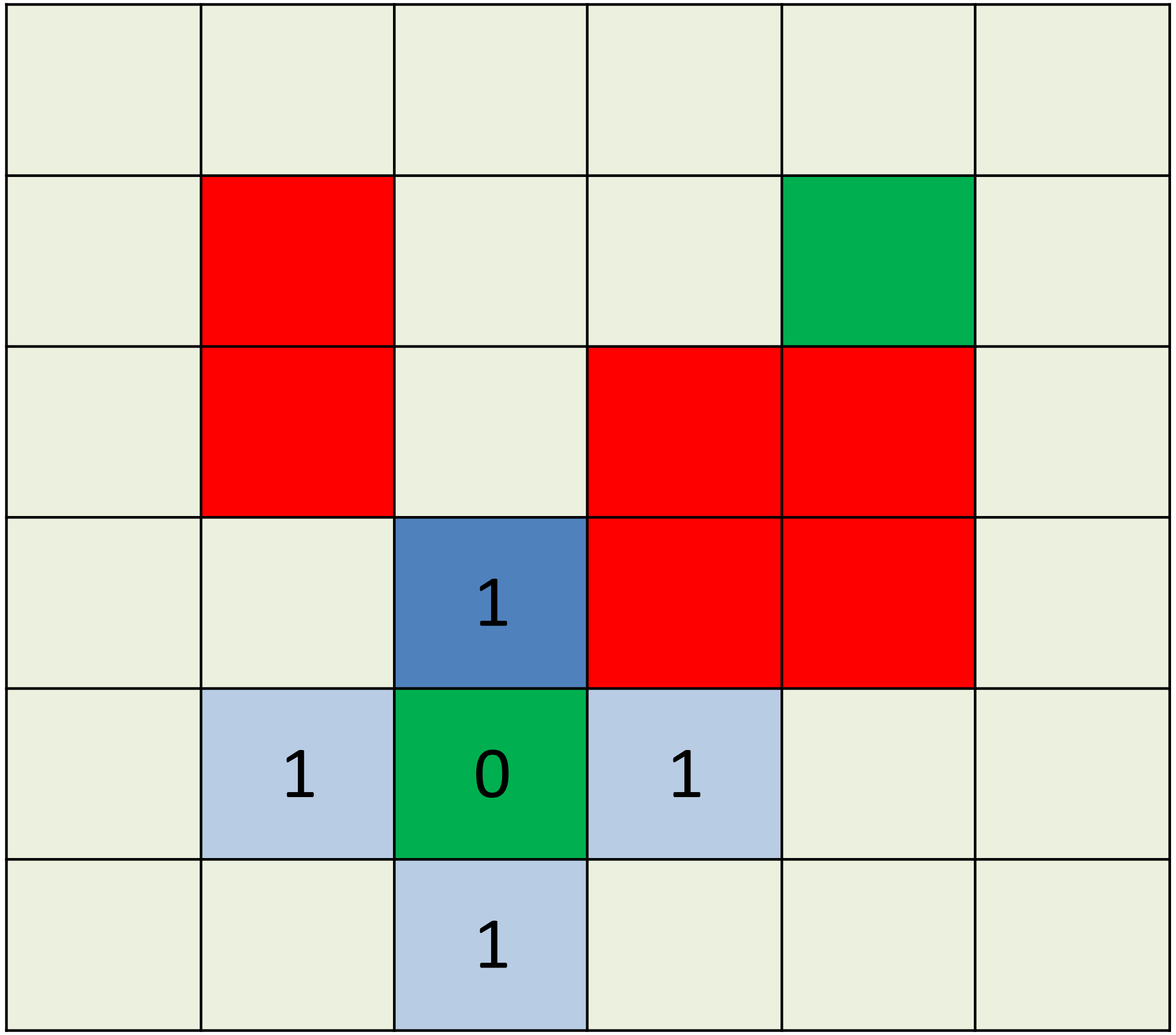
Pseudocode:

Start with i = 0 steps at q_{start}

Queue = neighbors of q_{start}

All neighbors have 1 step

Breadth First Search



Pseudocode:

Start with i = 0 steps at q_{start}

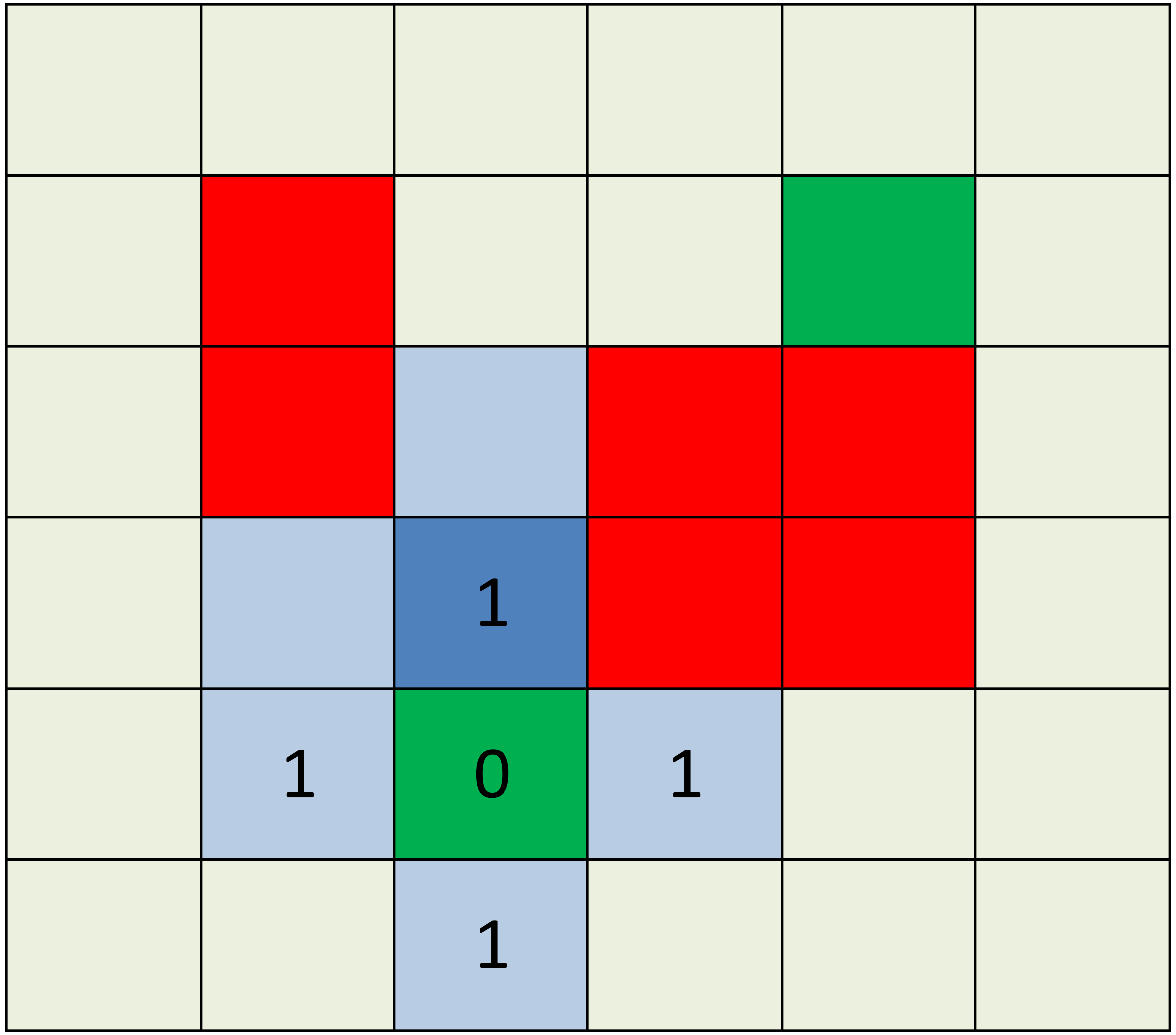
$Queue$ = neighbors of q_{start}

All neighbors have 1 step

While ~empty($Queue$)

  q = next cell in $Queue$

Breadth First Search



Pseudocode:

Start with i = 0 steps at q_{start}

$Queue$ = neighbors of q_{start}

All neighbors have 1 step

While ~empty($Queue$)

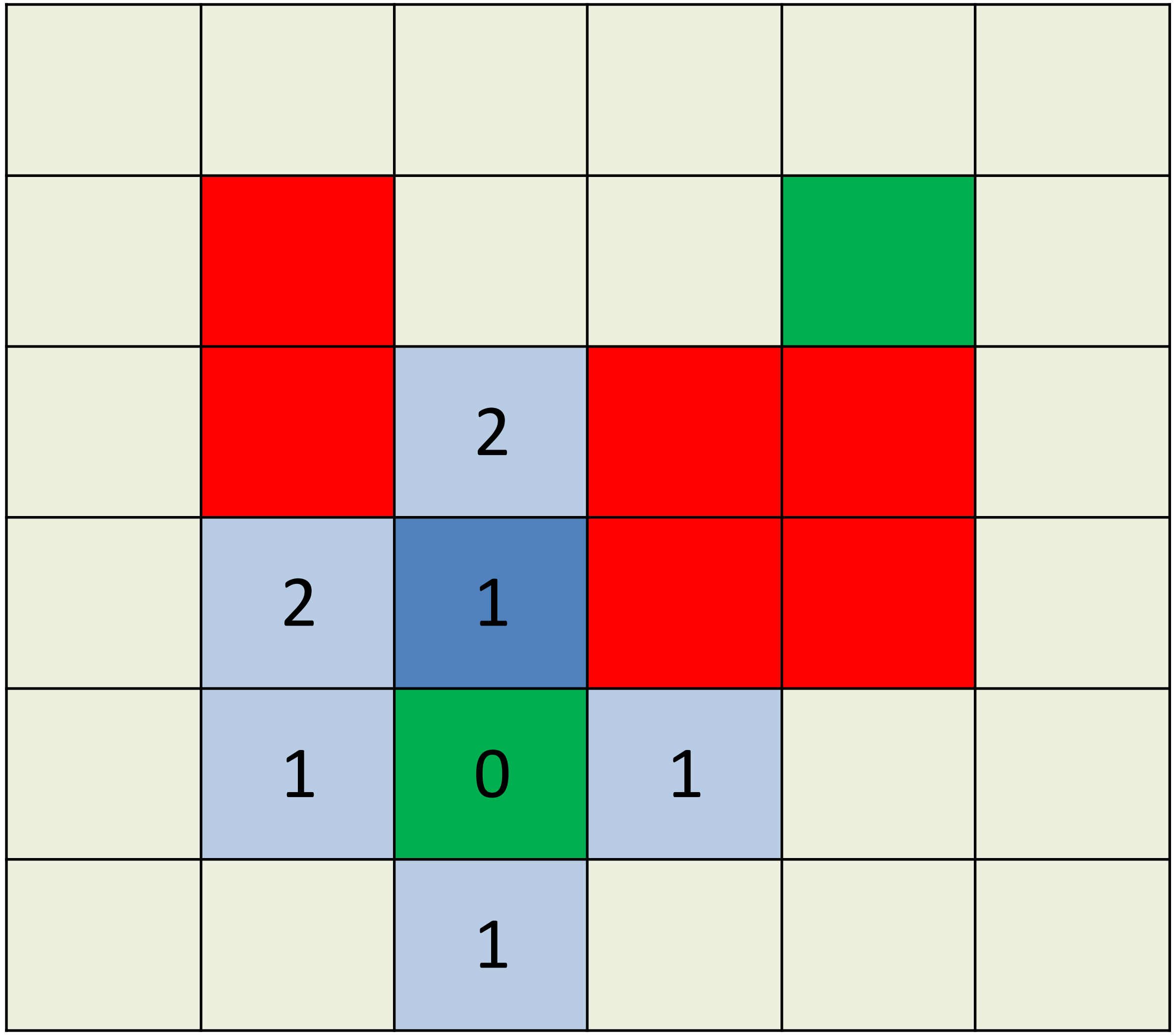
  q = next cell in $Queue$

  i = steps to q

  if a neighbor is q_{end}, STOP

  Add all new neighbors to $Queue$

Breadth First Search



Pseudocode:

Start with i = 0 steps at q_{start}

Queue = neighbors of q_{start}

All neighbors have 1 step

While ~empty(*Queue*)

    q = next cell in *Queue*

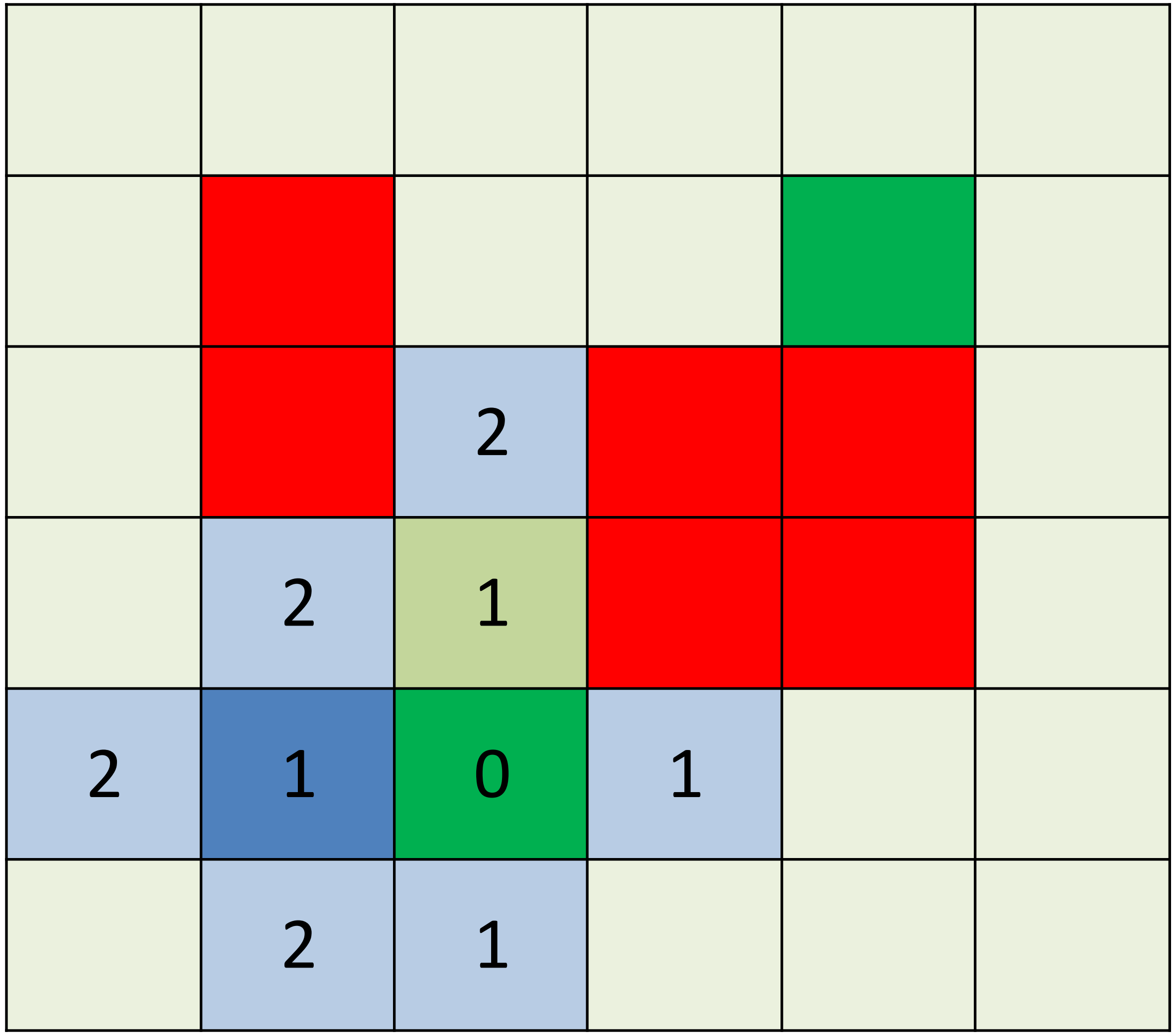
    i = steps to q

    if a neighbor is q_{end}, STOP

    Add all new neighbors to *Queue*

    All neighbors have $i+1$ steps

Breadth First Search



Pseudocode:

Start with i = 0 steps at q_{start}

$Queue$ = neighbors of q_{start}

All neighbors have 1 step

While ~empty($Queue$)

  q = next cell in $Queue$

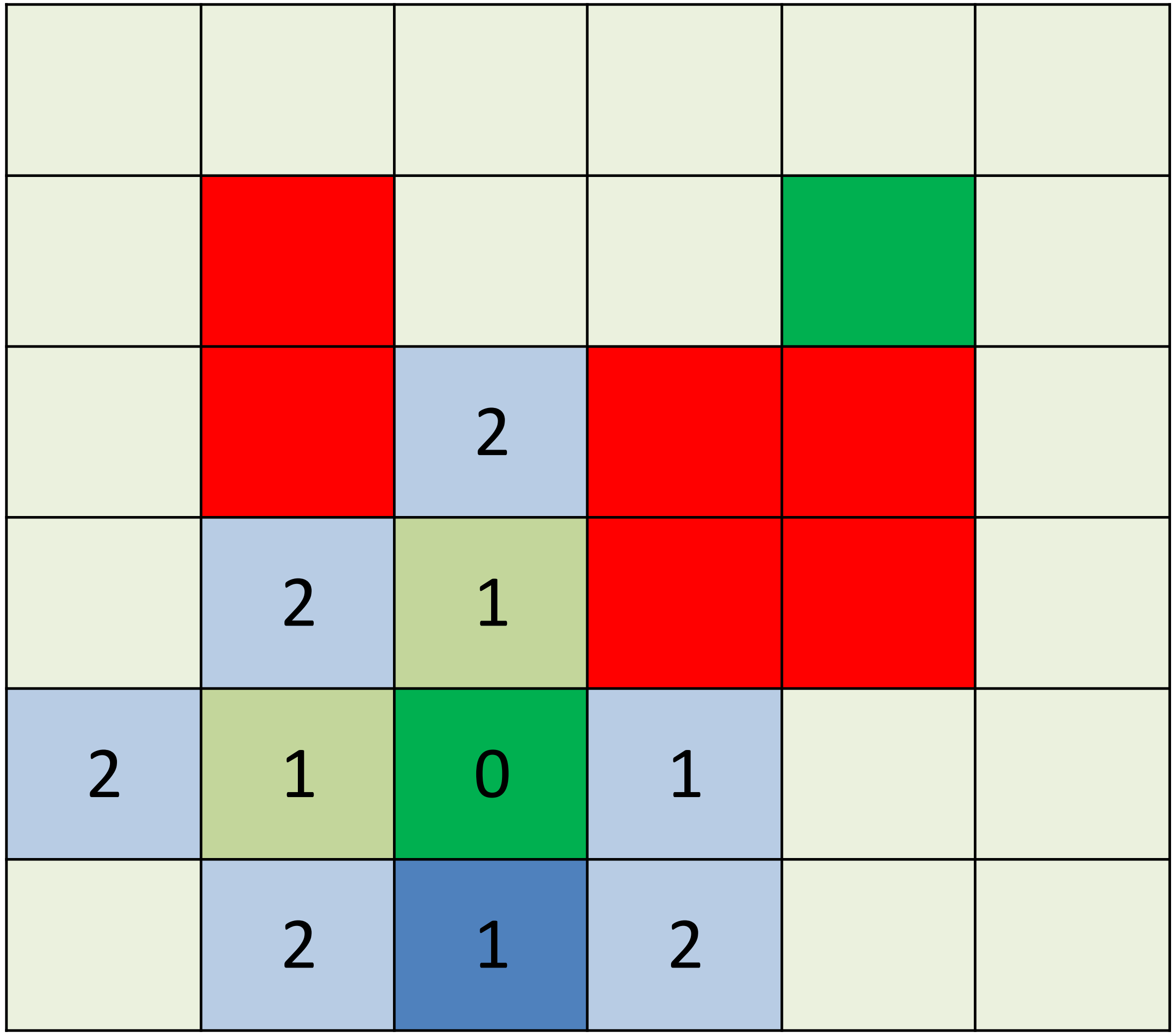
  i = steps to q

  if a neighbor is q_{end}, STOP

  Add all new neighbors to $Queue$

  All neighbors have i+1 steps

Breadth First Search



Pseudocode:

Start with i = 0 steps at q_{start}

Queue = neighbors of q_{start}

All neighbors have 1 step

While ~empty(*Queue*)

  q = next cell in *Queue*

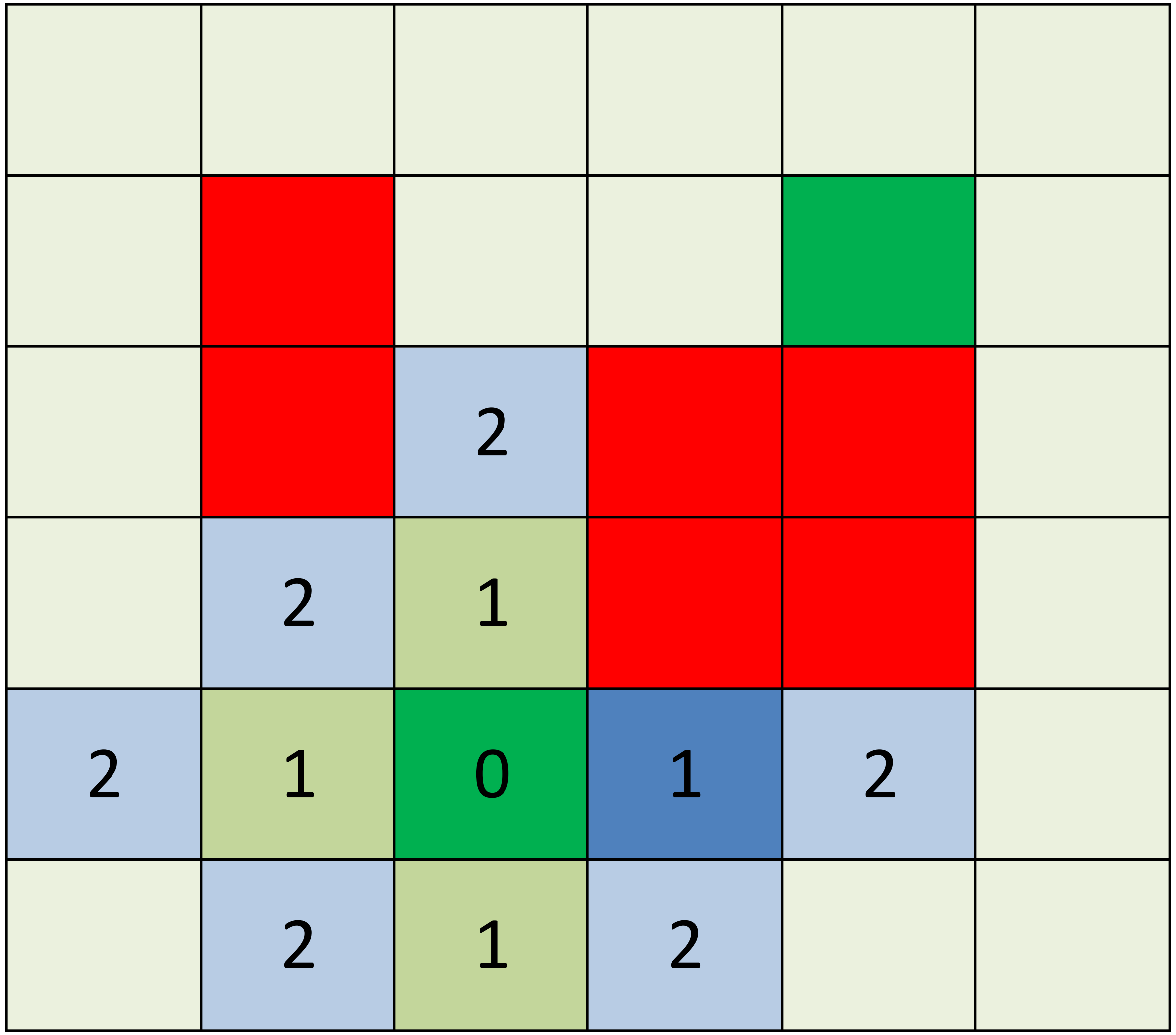
  i = steps to q

  if a neighbor is q_{end}, STOP

  Add all new neighbors to *Queue*

  All neighbors have i+1 steps

Breadth First Search



Pseudocode:

Start with i = 0 steps at q_{start}

Queue = neighbors of q_{start}

All neighbors have 1 step

While ~empty(*Queue*)

  q = next cell in *Queue*

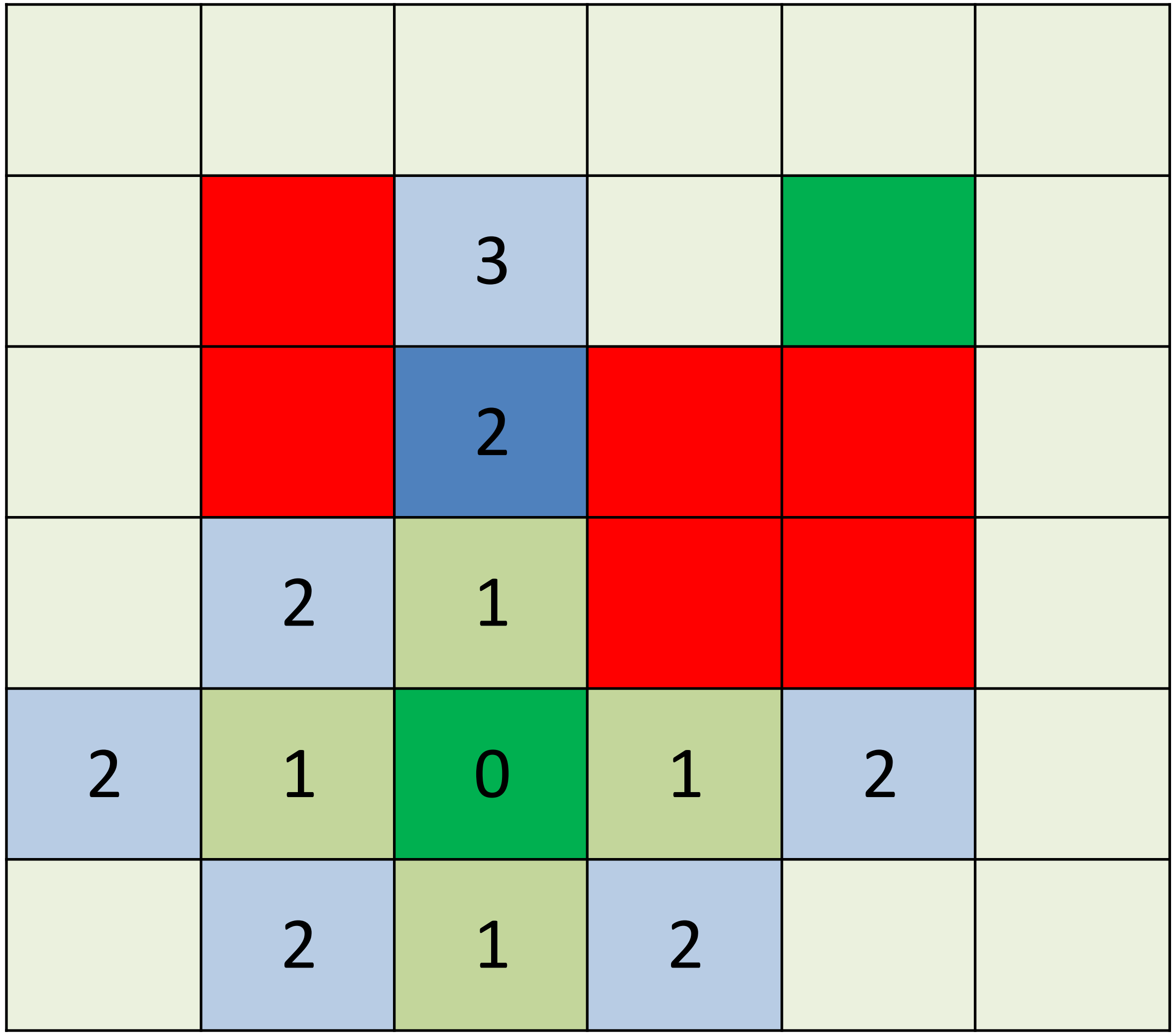
  i = steps to q

  if a neighbor is q_{end}, STOP

  Add all new neighbors to *Queue*

  All neighbors have $i+1$ steps

Breadth First Search



Pseudocode:

Start with i = 0 steps at q_{start}

$Queue$ = neighbors of q_{start}

All neighbors have 1 step

While ~empty($Queue$)

  q = next cell in $Queue$

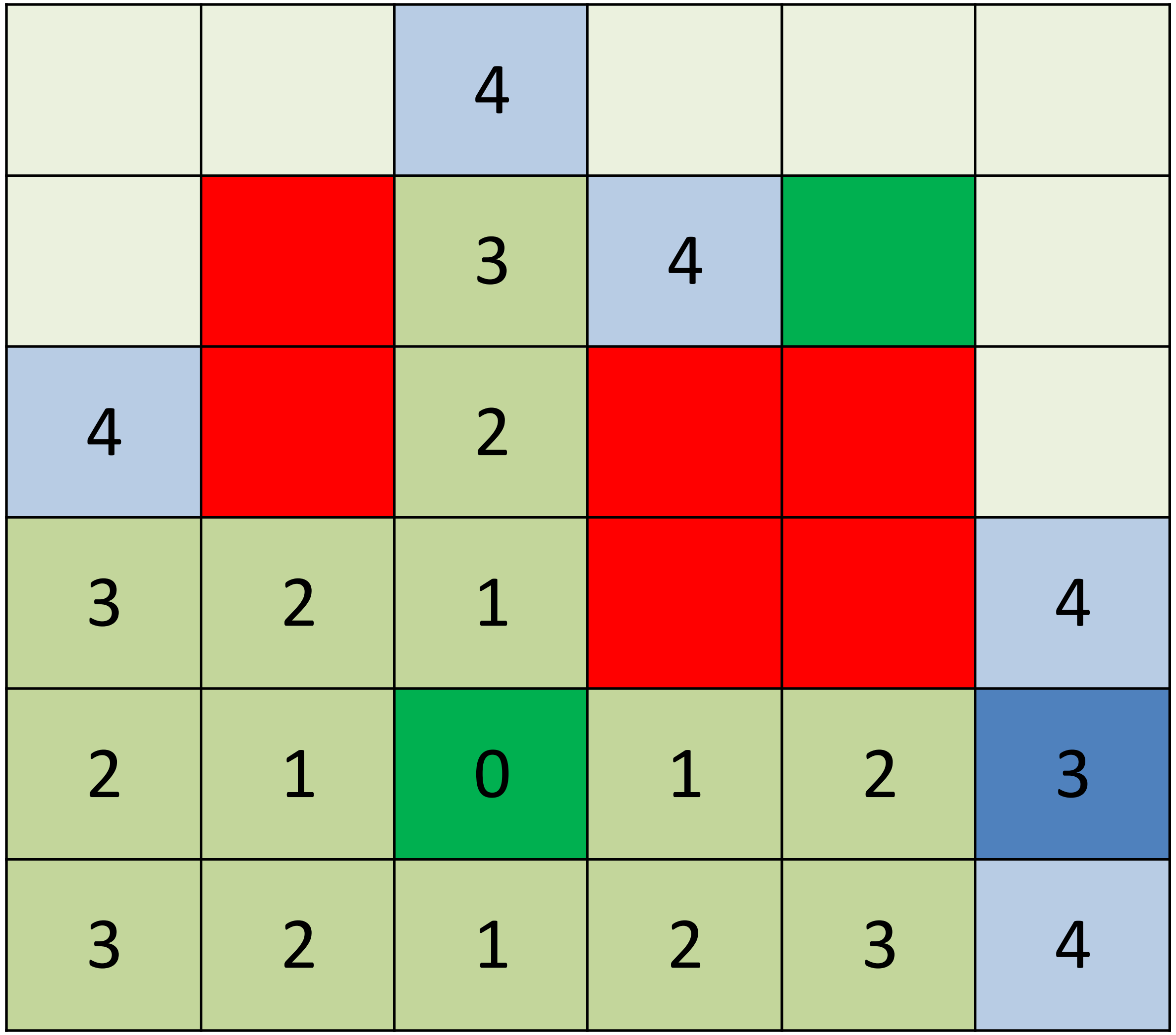
  i = steps to q

  if a neighbor is q_{end}, STOP

  Add all new neighbors to $Queue$

  All neighbors have $i+1$ steps

Breadth First Search



Pseudocode:

Start with i = 0 steps at q_{start}

$Queue$ = neighbors of q_{start}

All neighbors have 1 step

While ~empty($Queue$)

  q = next cell in $Queue$

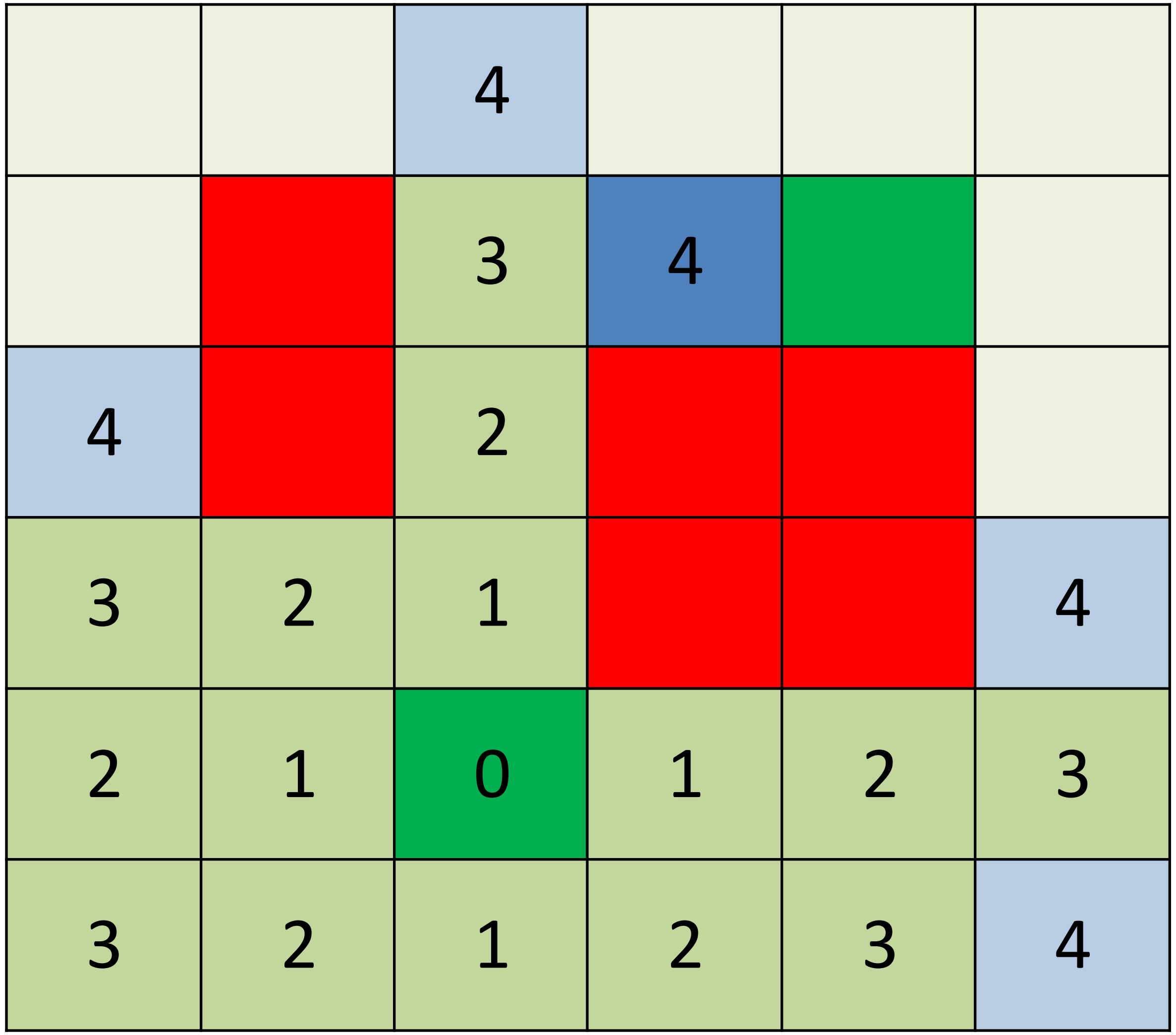
  i = steps to q

  if a neighbor is q_{end}, STOP

  Add all new neighbors to $Queue$

  All neighbors have i+1 steps

Breadth First Search



Pseudocode:

Start with i = 0 steps at q_{start}

Queue = neighbors of q_{start}

All neighbors have 1 step

While ~empty(*Queue*)

    q = next cell in *Queue*

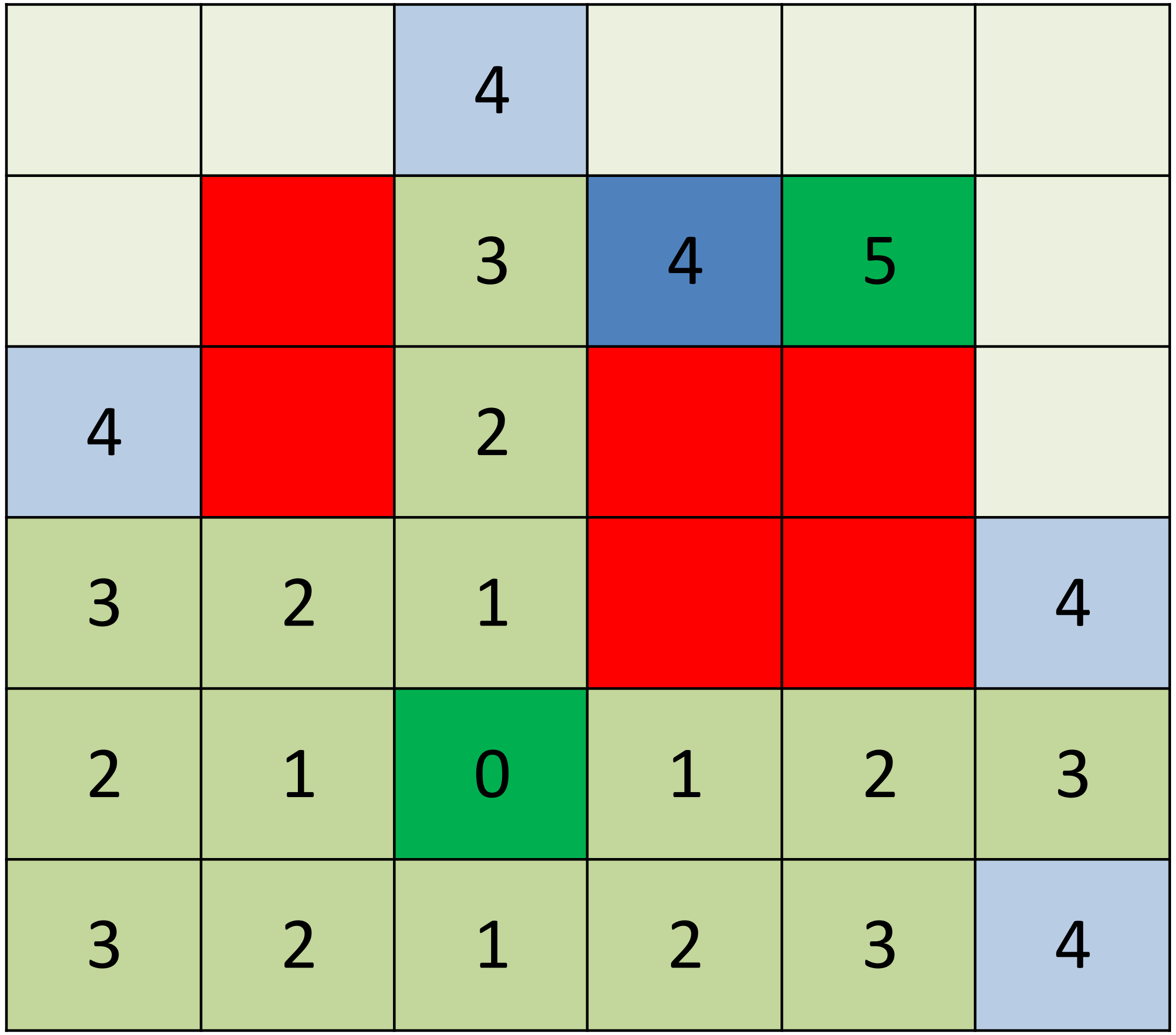
    i = steps to q

    if a neighbor is q_{end}, STOP

    Add all new neighbors to *Queue*

    All neighbors have i+1 steps

Breadth First Search



Pseudocode:

Start with i = 0 steps at q_{start}

$Queue$ = neighbors of q_{start}

All neighbors have 1 step

While ~empty($Queue$)

  q = next cell in $Queue$

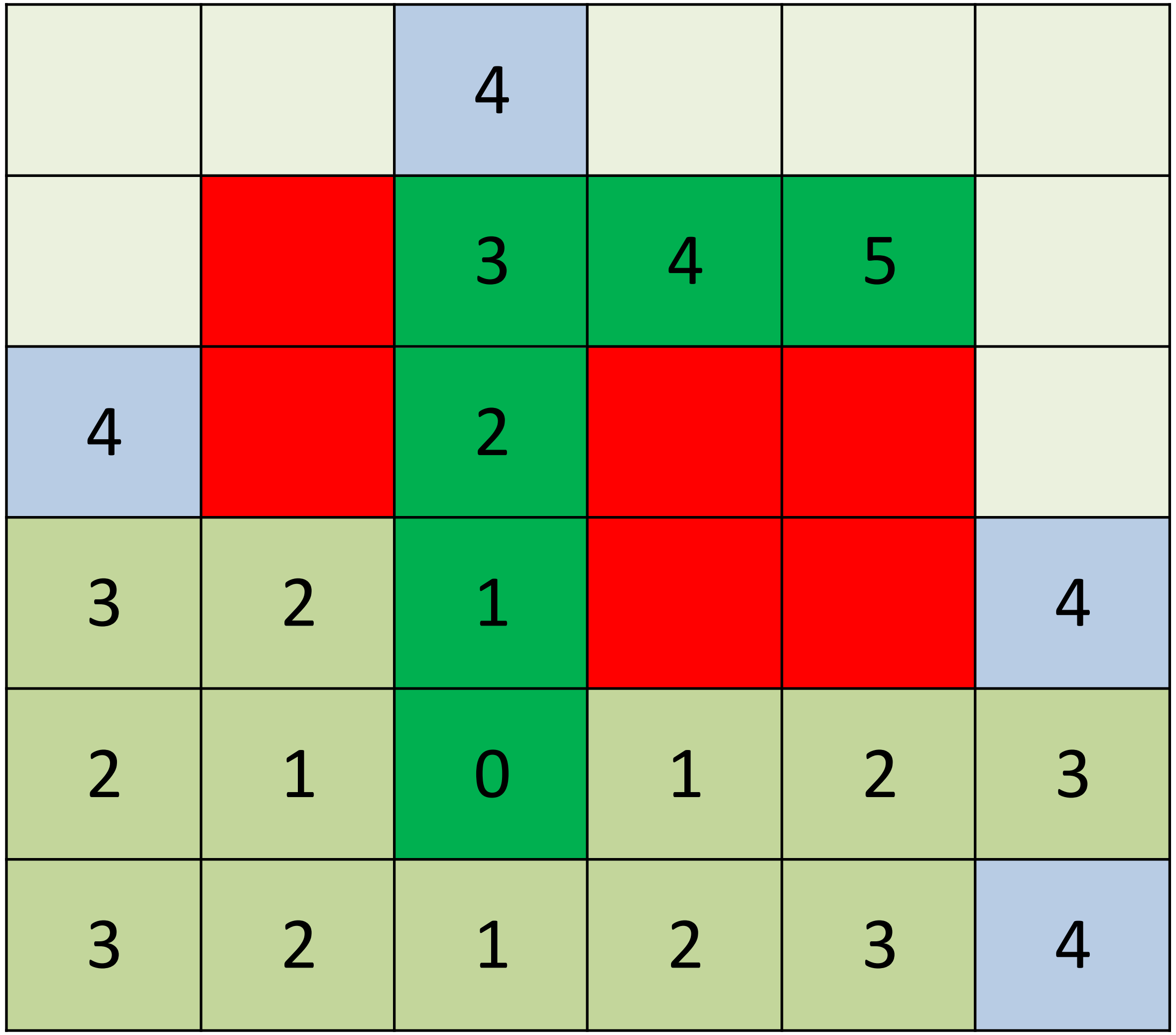
  i = steps to q

  if a neighbor is q_{end}, STOP

  Add all new neighbors to $Queue$

  All neighbors have i+1 steps

Breadth First Search



Pseudocode:

Start with i = 0 steps at q_{start}

$Queue$ = neighbors of q_{start}

All neighbors have 1 step

While ~empty($Queue$)

  q = next cell in $Queue$

  i = steps to q

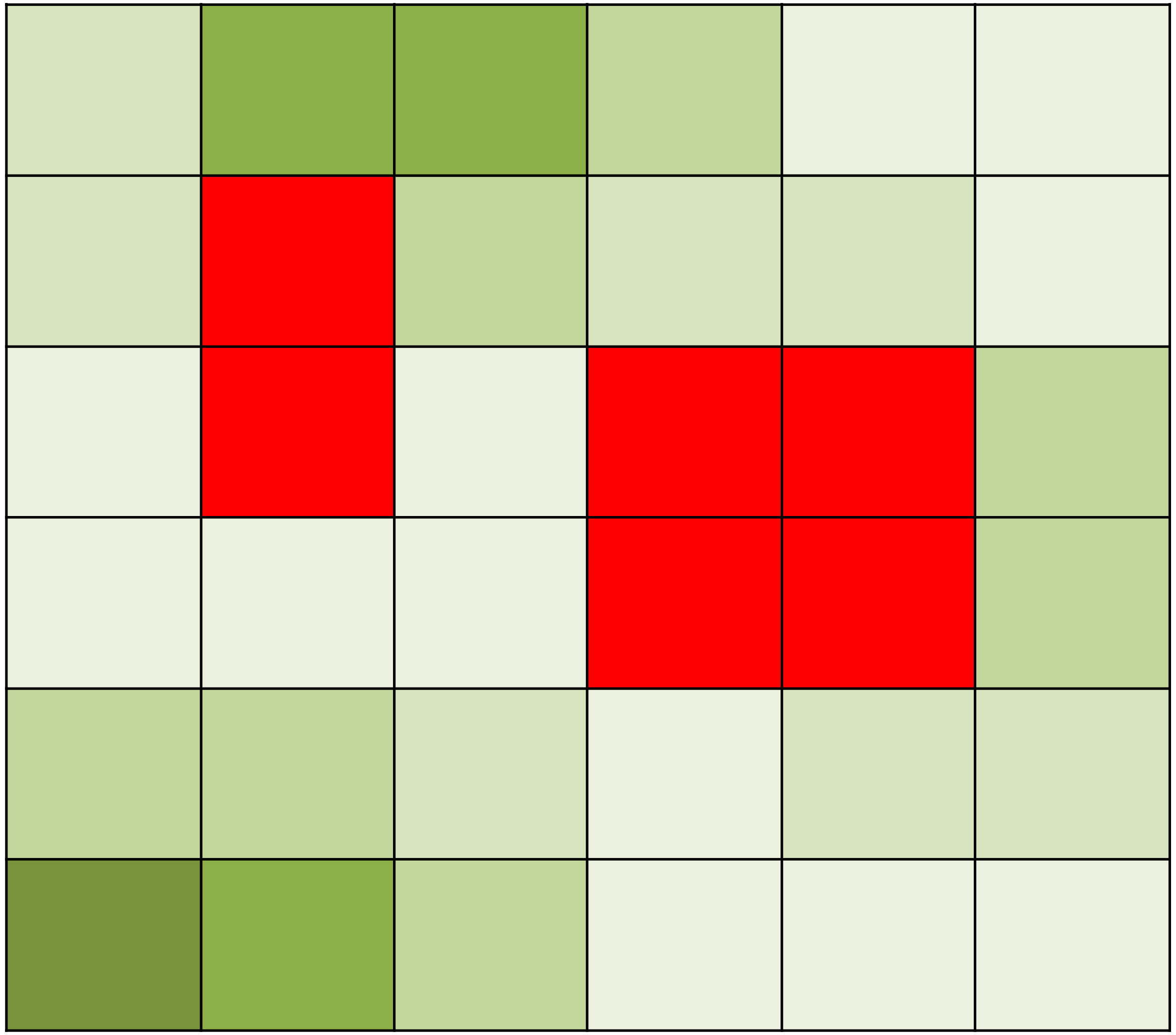
  if a neighbor is q_{end}, STOP

  Add all new neighbors to $Queue$

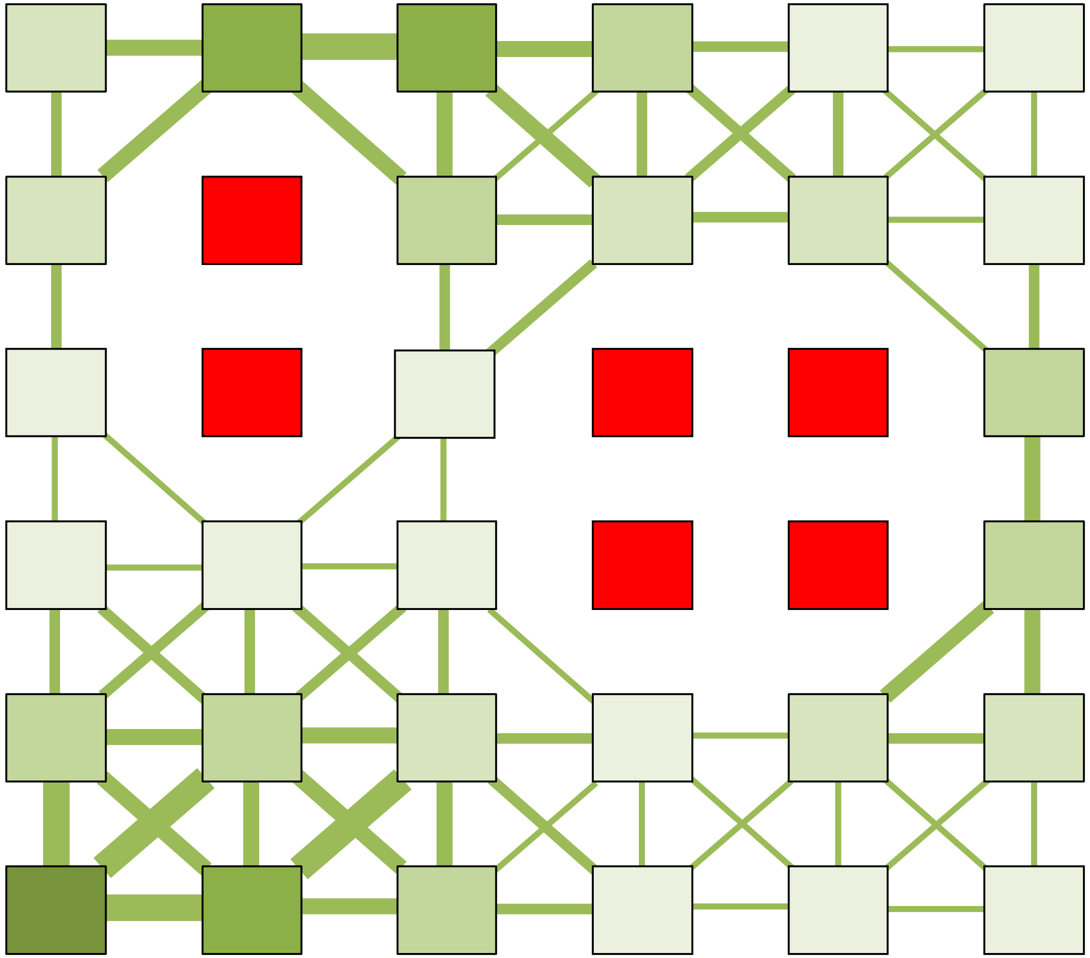
  All neighbors have i+1 steps

Potentially search all cells:
Computation is $O(N_{cell})$

Nonuniform costs



Graph Representation of the Configuration Space

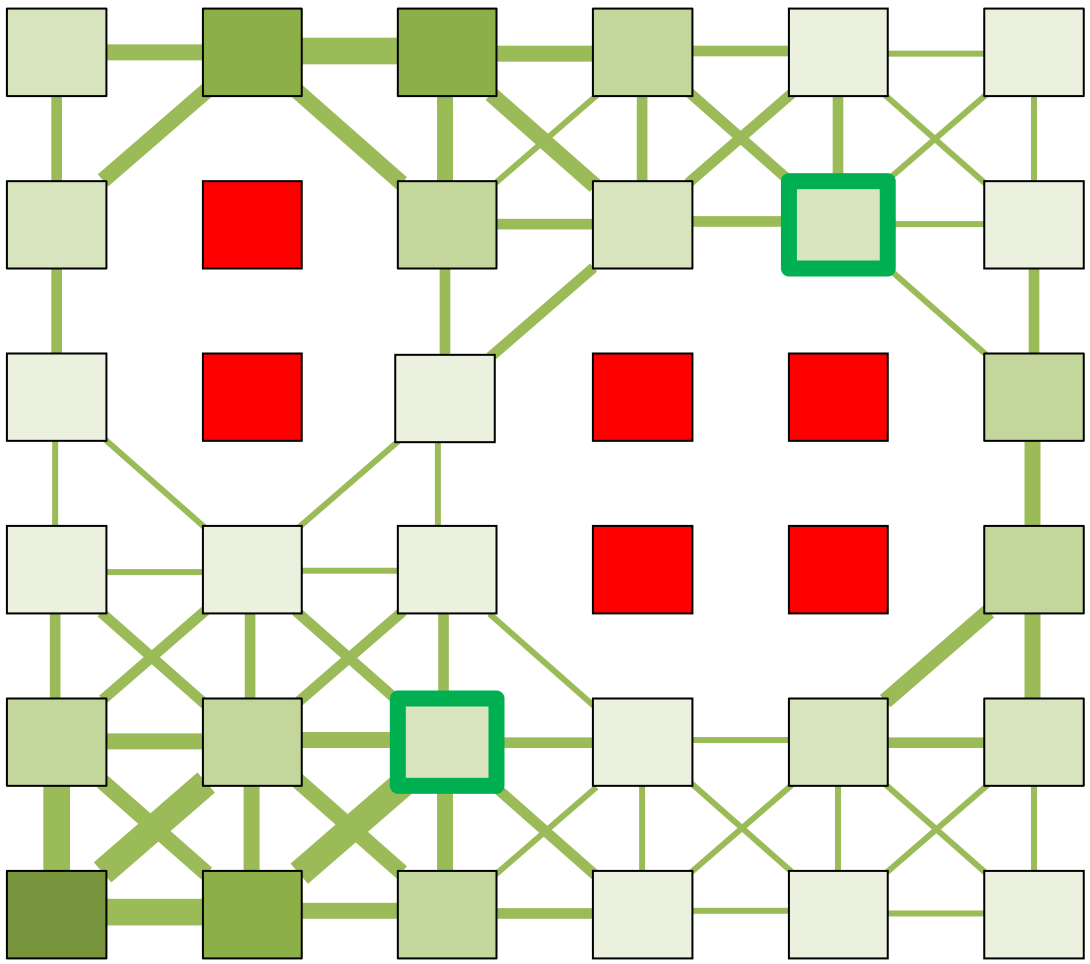


Graph: vertices connected by edges

Assign costs

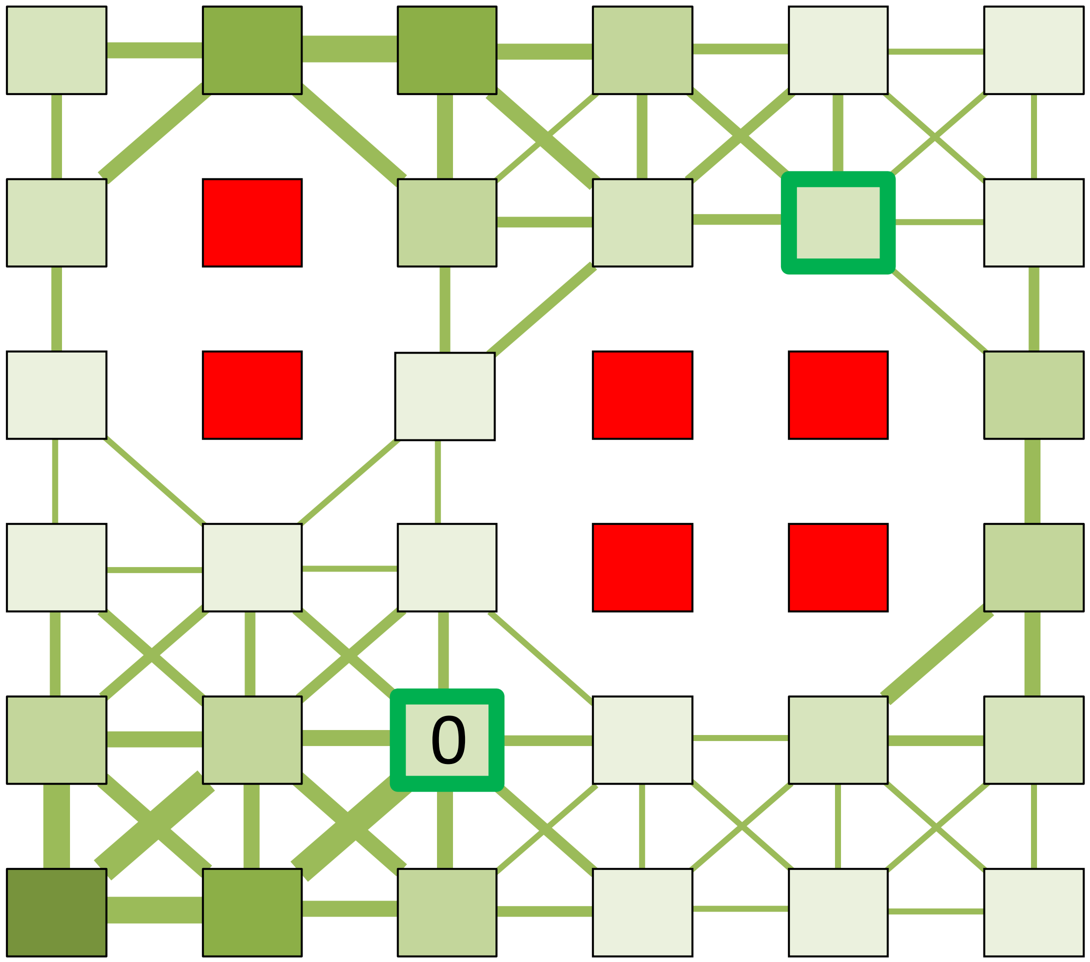
Remove edges to obstacles

Dijkstra's Algorithm



Pseudocode:

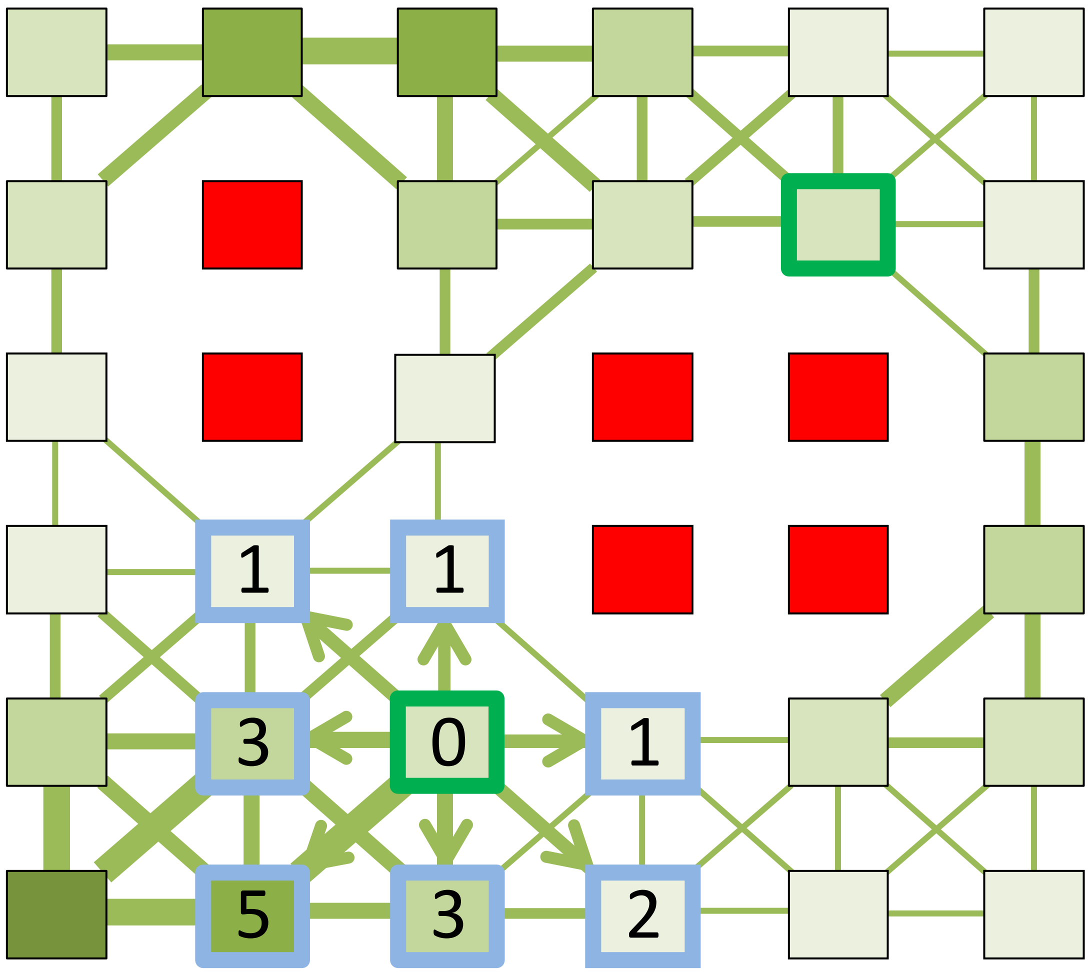
Dijkstra's Algorithm



Pseudocode:

Start with i = 0 steps at q_{start}

Dijkstra's Algorithm



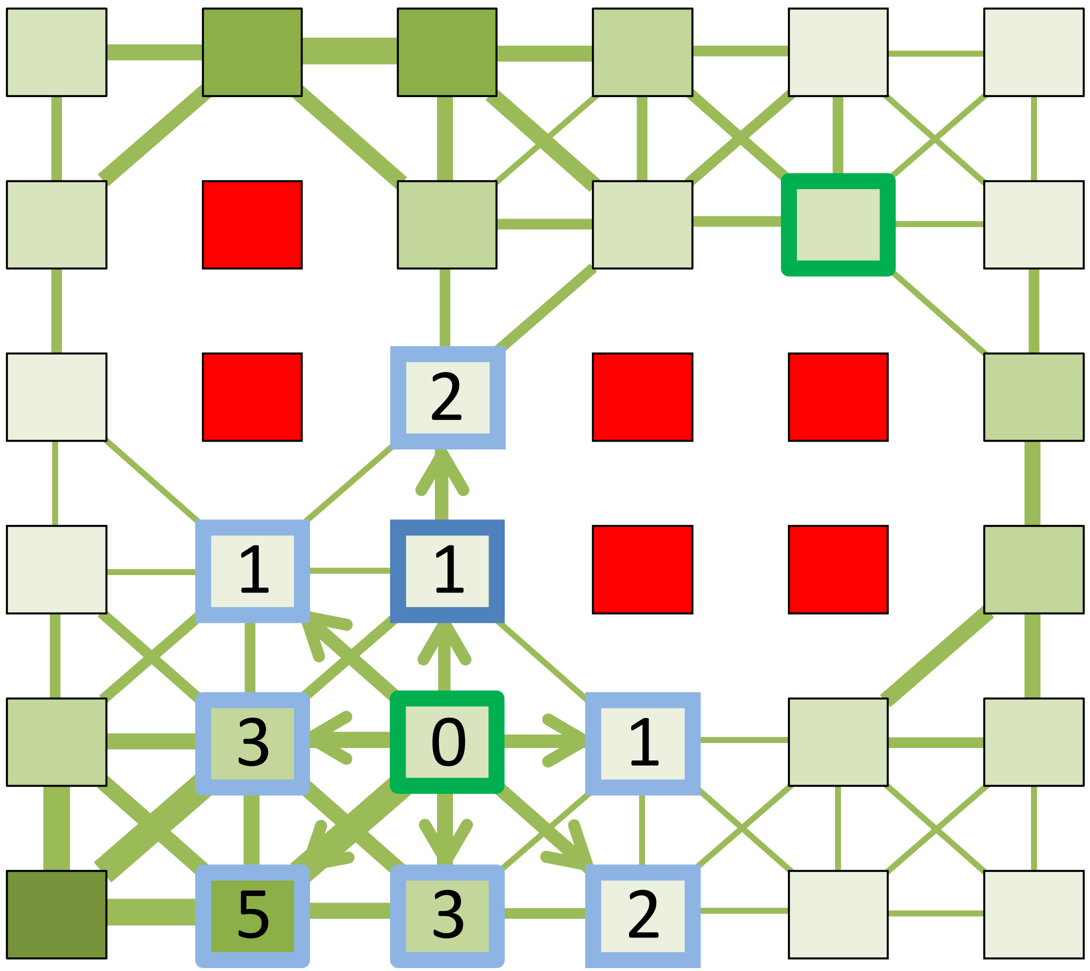
Pseudocode:

Start with i = 0 steps at q_{start}

Add neighbors of q_{start} to *boundary*

Update costs of neighbors

Dijkstra's Algorithm



Pseudocode:

Start with i = 0 steps at q_{start}

Add neighbors of q_{start} to *boundary*

Update costs of neighbors

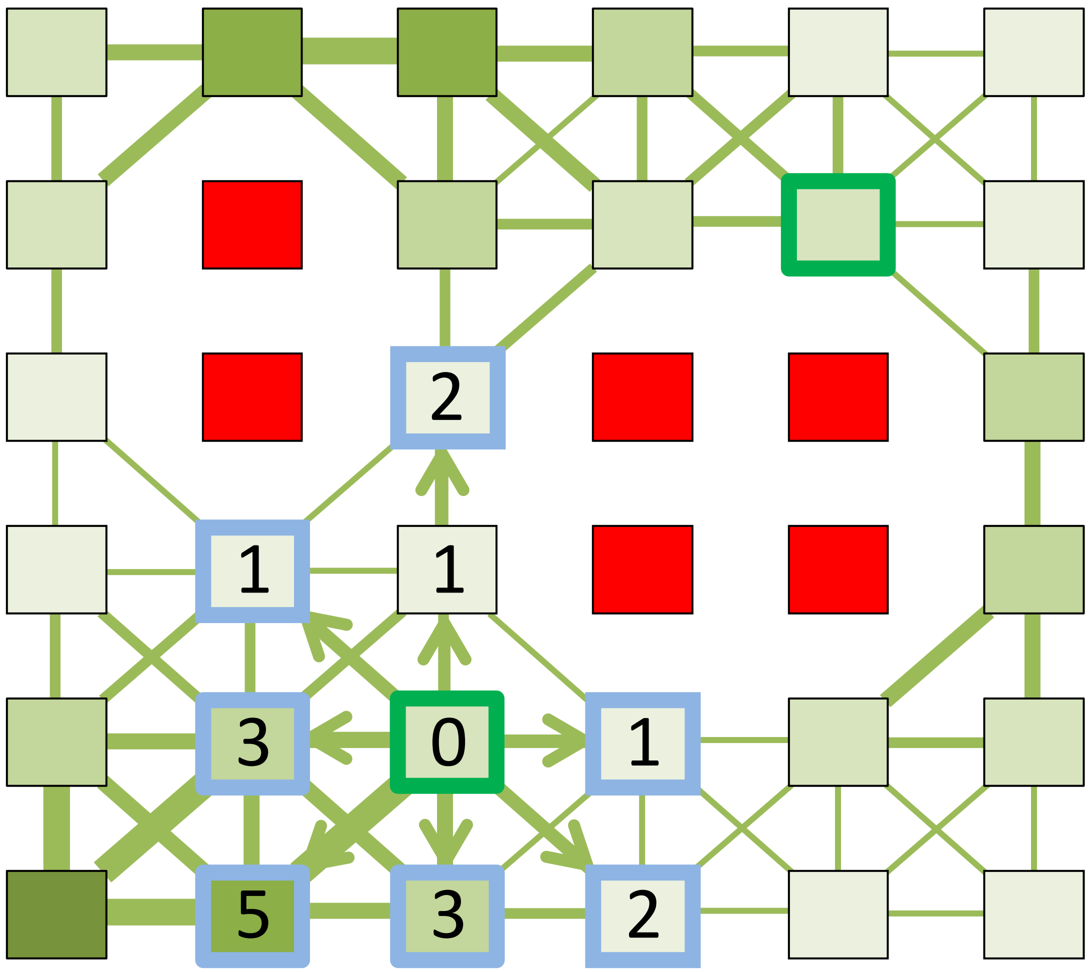
While ~empty(*boundary*)

  q = *boundary* cell with min cost

  Add all new neighbors to *boundary*

  Update costs of new neighbors

Dijkstra’s Algorithm



Pseudocode:

Start with i = 0 steps at q_{start}

Add neighbors of q_{start} to *boundary*

Update costs of neighbors

While ~empty(*boundary*)

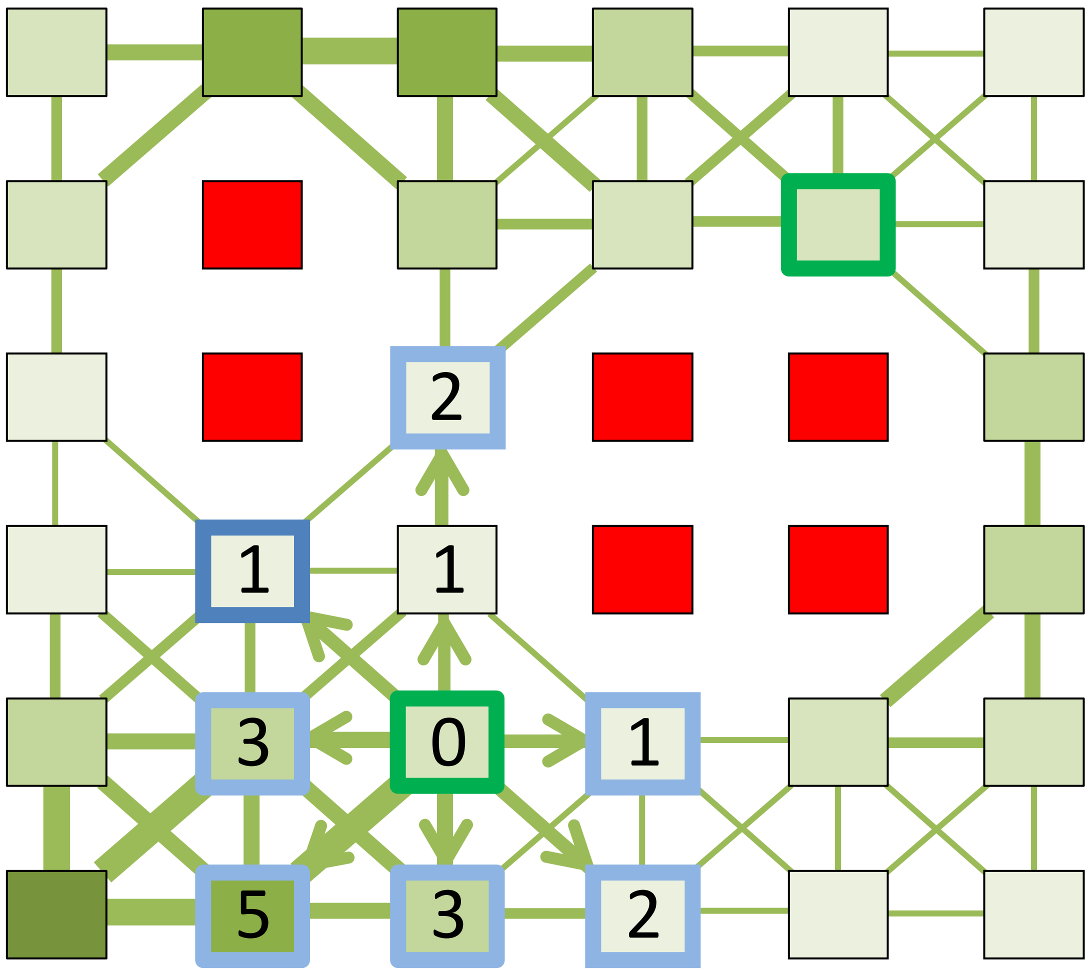
  q = *boundary* cell with min cost

  Add all new neighbors to *boundary*

  Update costs of new neighbors

  Remove q from *boundary*

Dijkstra's Algorithm



Pseudocode:

Start with i = 0 steps at q_{start}

Add neighbors of q_{start} to *boundary*

Update costs of neighbors

While ~empty(*boundary*)

  q = *boundary* cell with min cost

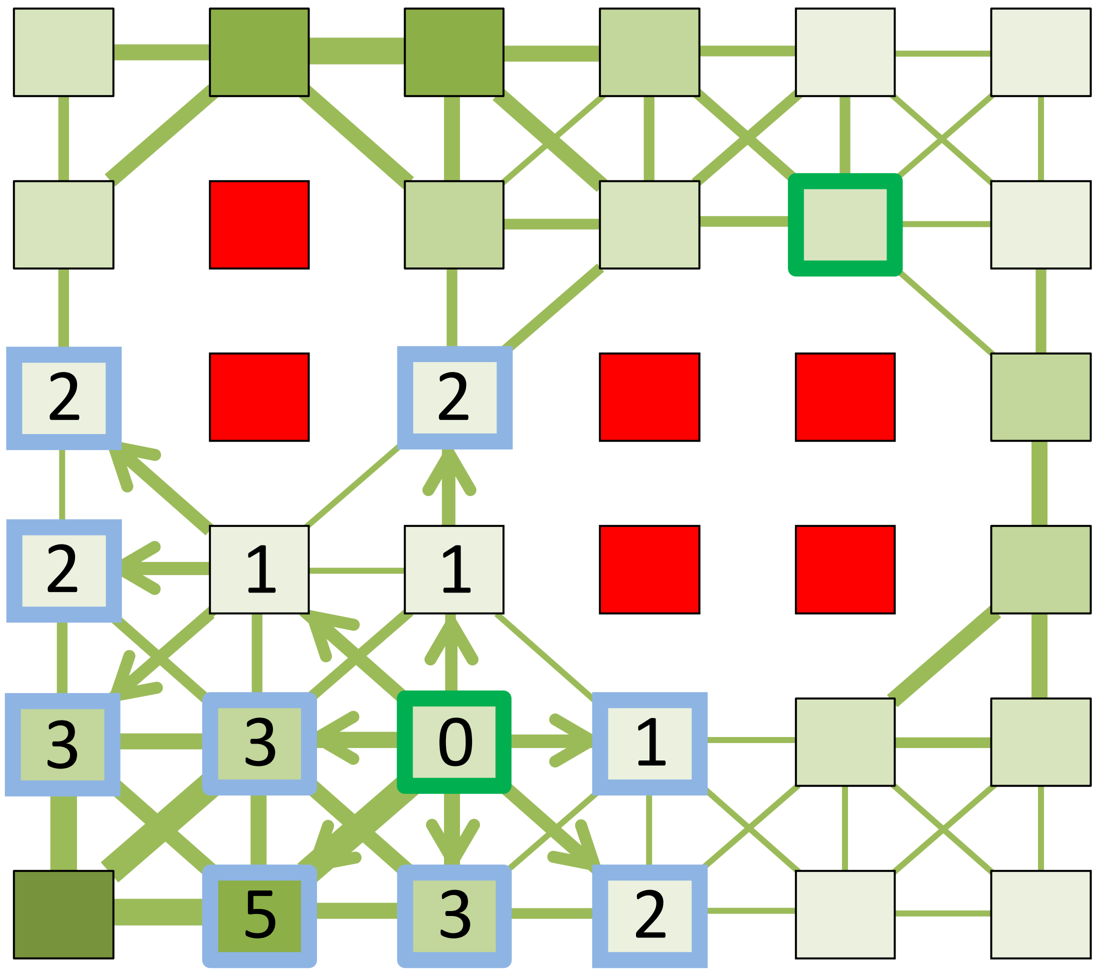
  Add all new neighbors to *boundary*

  Update costs of new neighbors

  Remove q from *boundary*

  If a neighbor is q_{end}, STORE

Dijkstra's Algorithm



Pseudocode:

Start with i = 0 steps at q_{start}

Add neighbors of q_{start} to *boundary*

Update costs of neighbors

While ~empty(*boundary*)

  q = *boundary* cell with min cost

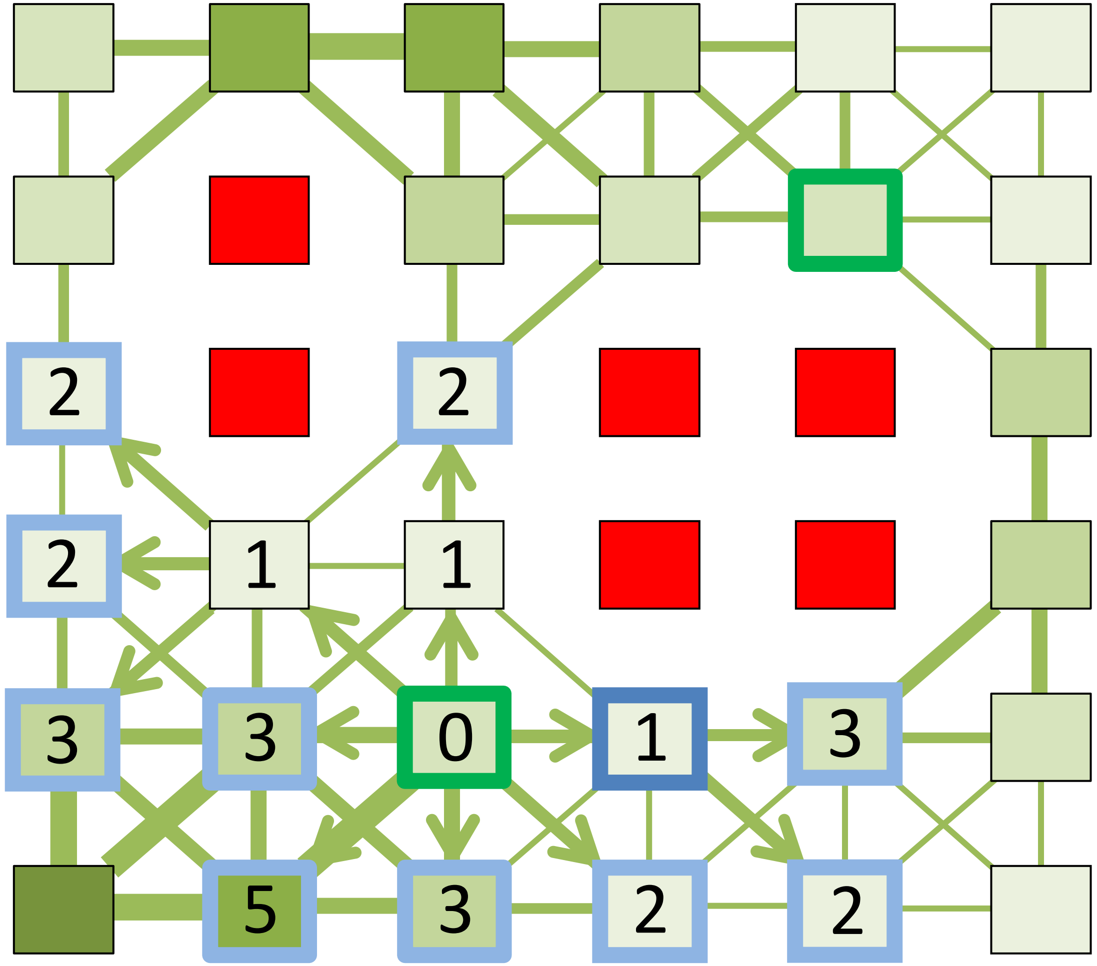
  Add all new neighbors to *boundary*

  Update costs of new neighbors

  Remove q from *boundary*

  If a neighbor is q_{end}, STORE

Dijkstra's Algorithm



Pseudocode:

Start with i = 0 steps at q_{start}

Add neighbors of q_{start} to *boundary*

Update costs of neighbors

While ~empty(*boundary*)

  q = *boundary* cell with min cost

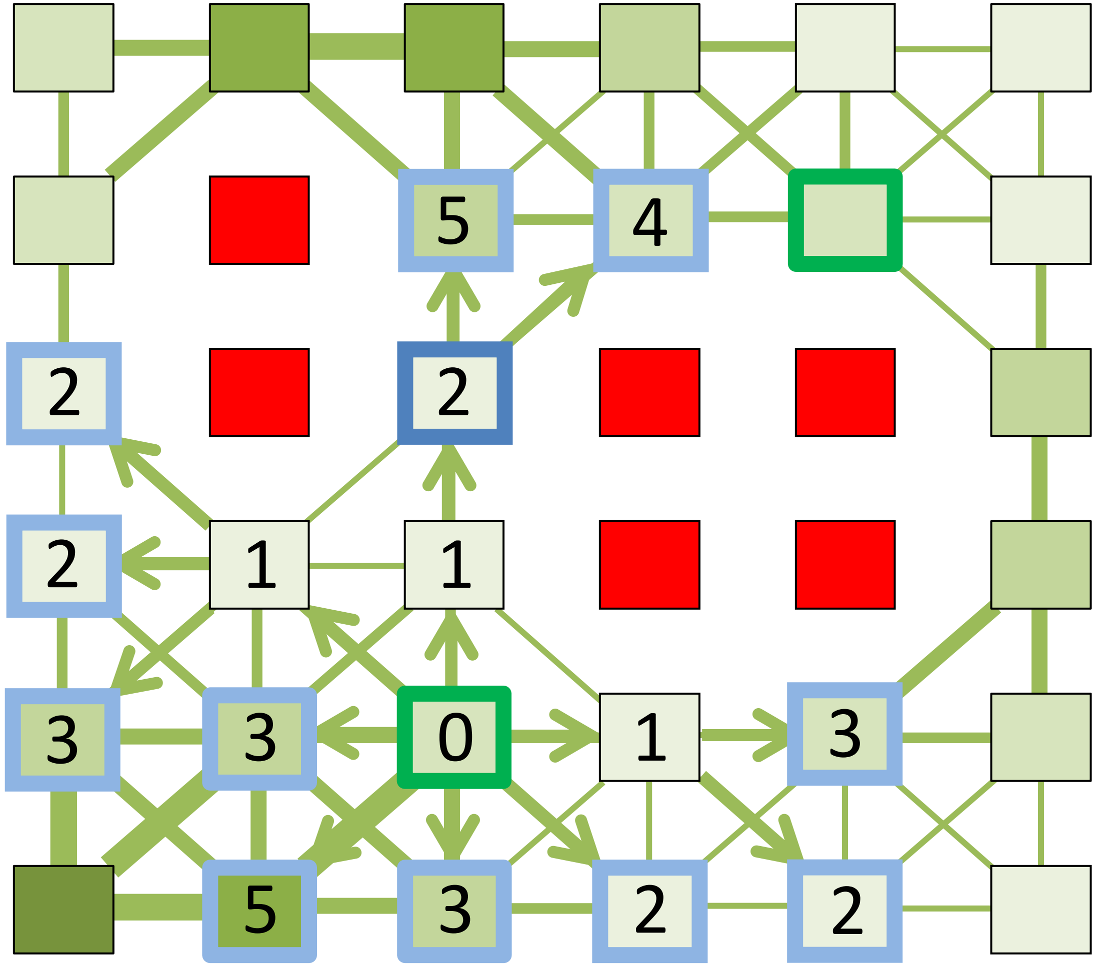
  Add all new neighbors to *boundary*

  Update costs of new neighbors

  Remove q from *boundary*

  If a neighbor is q_{end}, STORE

Dijkstra's Algorithm



Pseudocode:

Start with i = 0 steps at q_{start}

Add neighbors of q_{start} to *boundary*

Update costs of neighbors

While ~empty(*boundary*)

  q = *boundary* cell with min cost

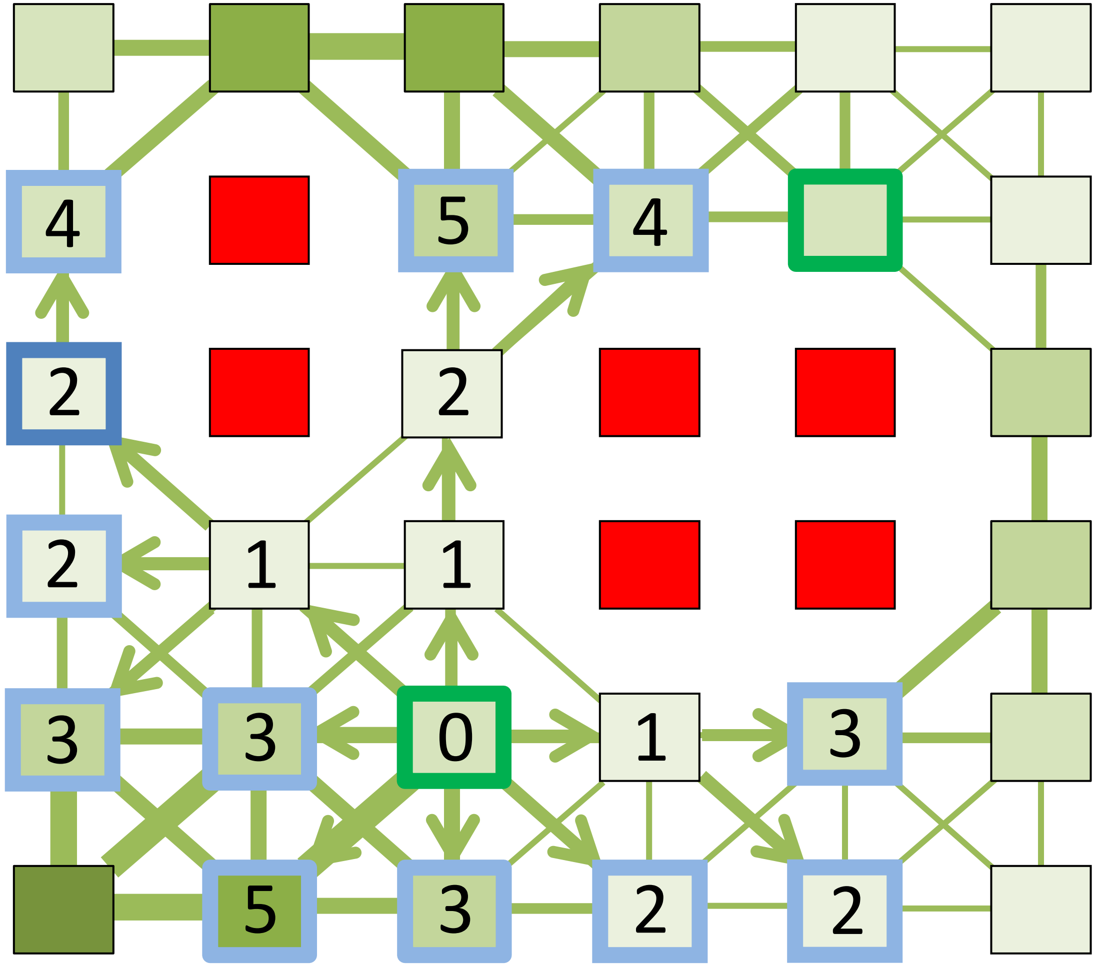
  Add all new neighbors to *boundary*

  Update costs of new neighbors

  Remove q from *boundary*

  If a neighbor is q_{end}, STORE

Dijkstra's Algorithm



Pseudocode:

Start with i = 0 steps at q_{start}

Add neighbors of q_{start} to *boundary*

Update costs of neighbors

While ~empty(*boundary*)

  q = *boundary* cell with min cost

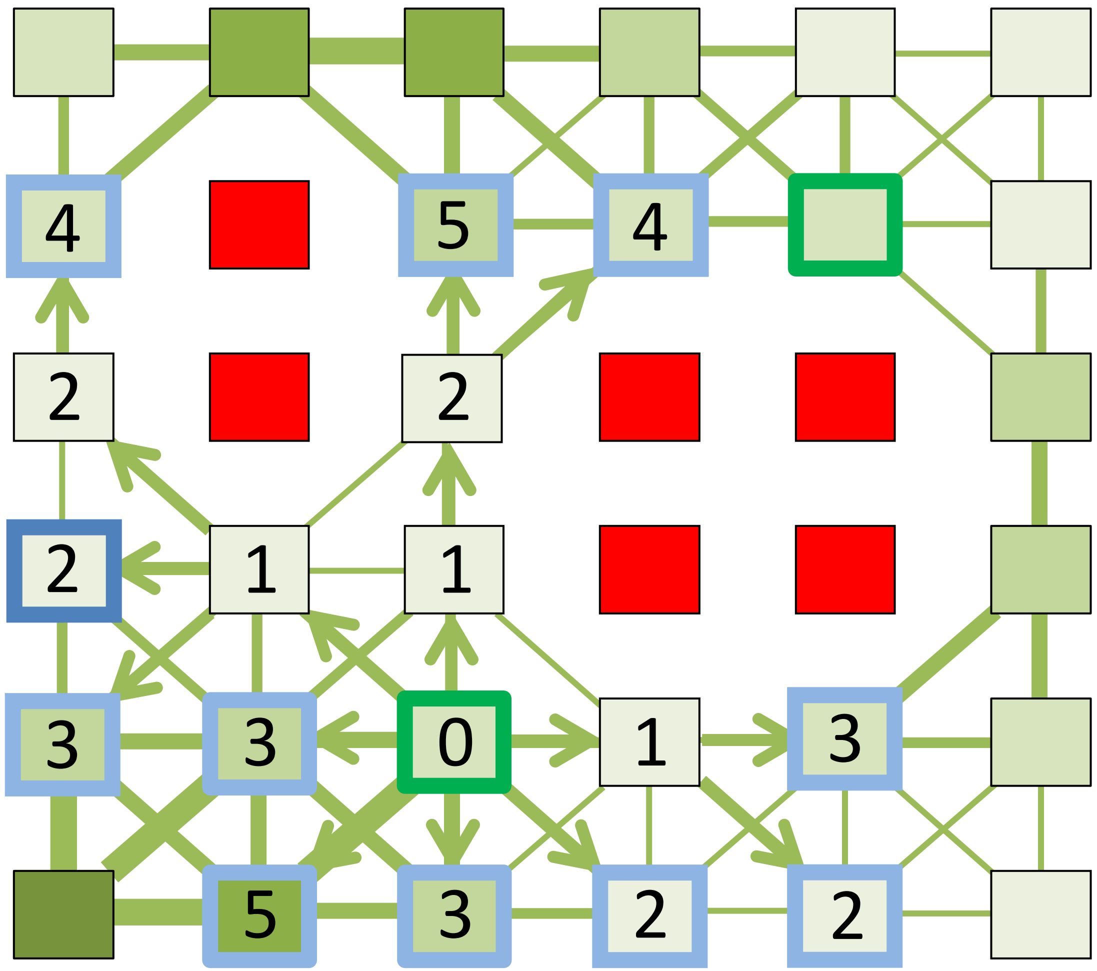
  Add all new neighbors to *boundary*

  Update costs of new neighbors

  Remove q from *boundary*

  If a neighbor is q_{end}, STORE

Dijkstra's Algorithm



Pseudocode:

Start with i = 0 steps at q_{start}

Add neighbors of q_{start} to *boundary*

Update costs of neighbors

While ~empty(*boundary*)

  q = *boundary* cell with min cost

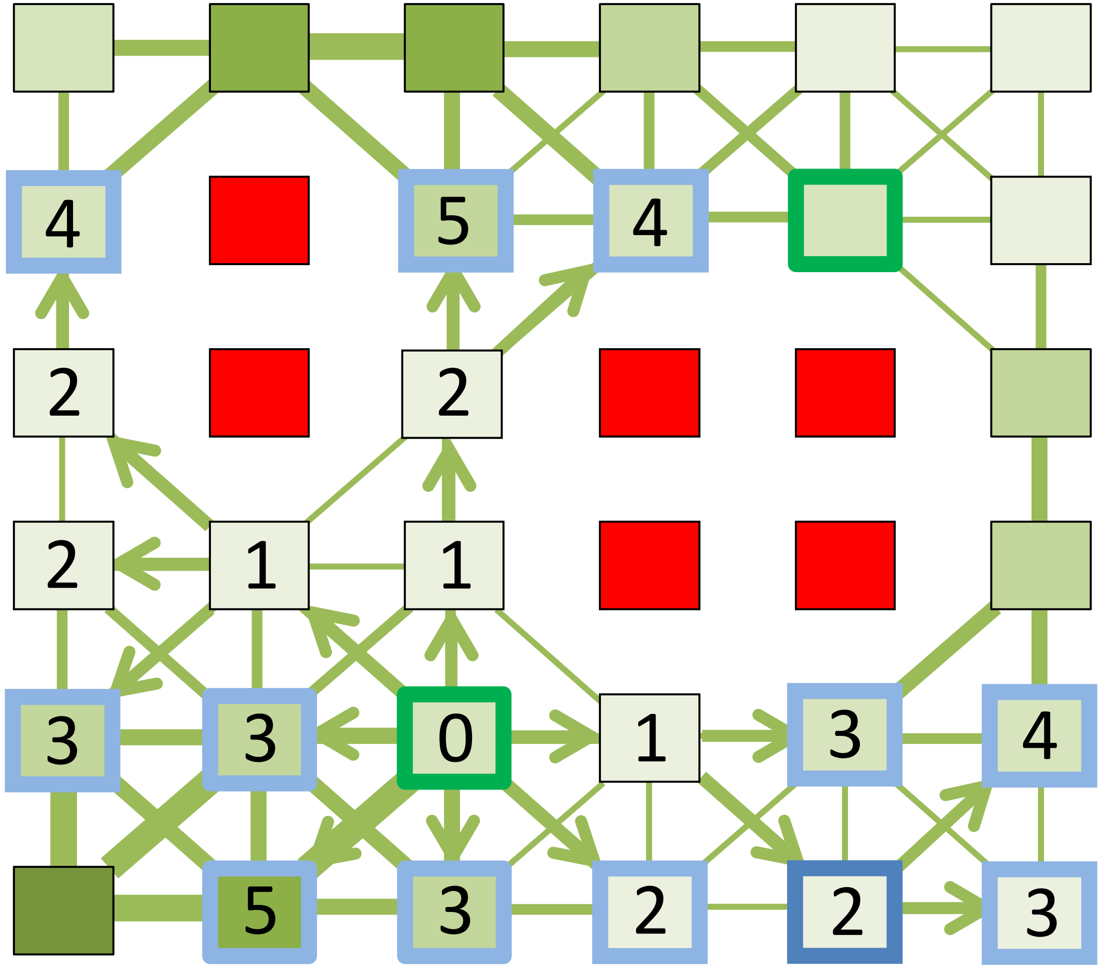
  Add all new neighbors to *boundary*

  Update costs of new neighbors

  Remove q from *boundary*

  If a neighbor is q_{end}, STORE

Dijkstra's Algorithm



Pseudocode:

Start with i = 0 steps at q_{start}

Add neighbors of q_{start} to *boundary*

Update costs of neighbors

While ~empty(*boundary*)

  q = *boundary* cell with min cost

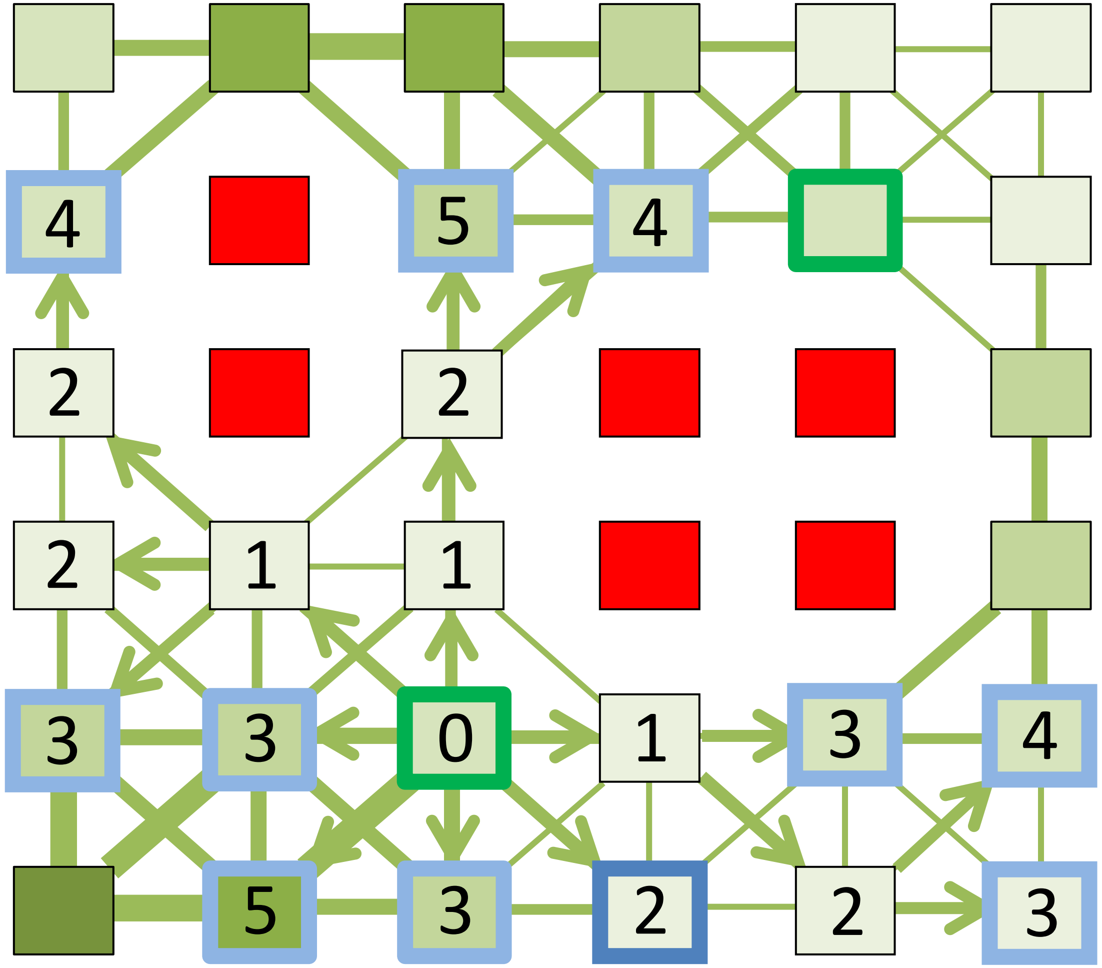
  Add all new neighbors to *boundary*

  Update costs of new neighbors

  Remove q from *boundary*

  If a neighbor is q_{end}, STORE

Dijkstra's Algorithm



Pseudocode:

Start with i = 0 steps at q_{start}

Add neighbors of q_{start} to *boundary*

Update costs of neighbors

While ~empty(*boundary*)

  q = *boundary* cell with min cost

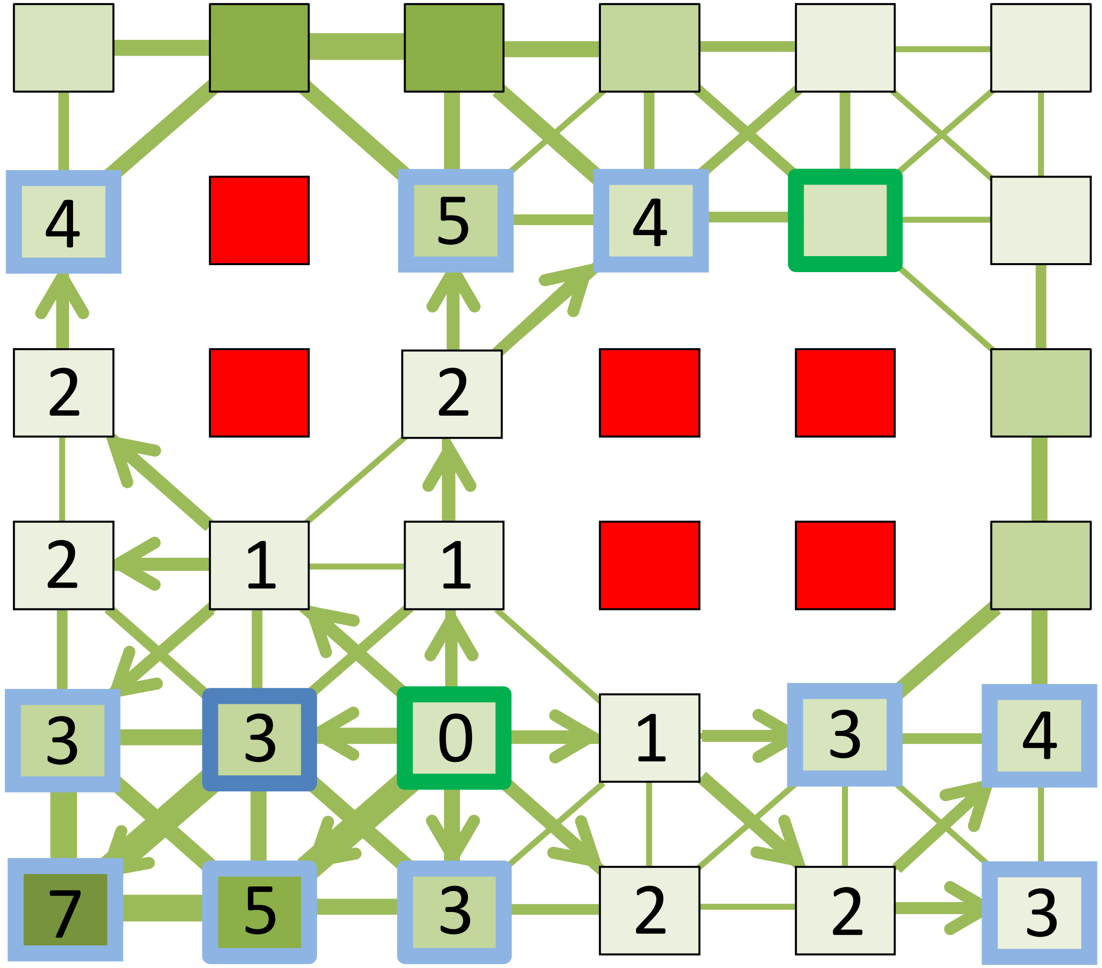
  Add all new neighbors to *boundary*

  Update costs of new neighbors

  Remove q from *boundary*

  If a neighbor is q_{end}, STORE

Dijkstra's Algorithm



Pseudocode:

Start with i = 0 steps at q_{start}

Add neighbors of q_{start} to *boundary*

Update costs of neighbors

While ~empty(*boundary*)

  q = *boundary* cell with min cost

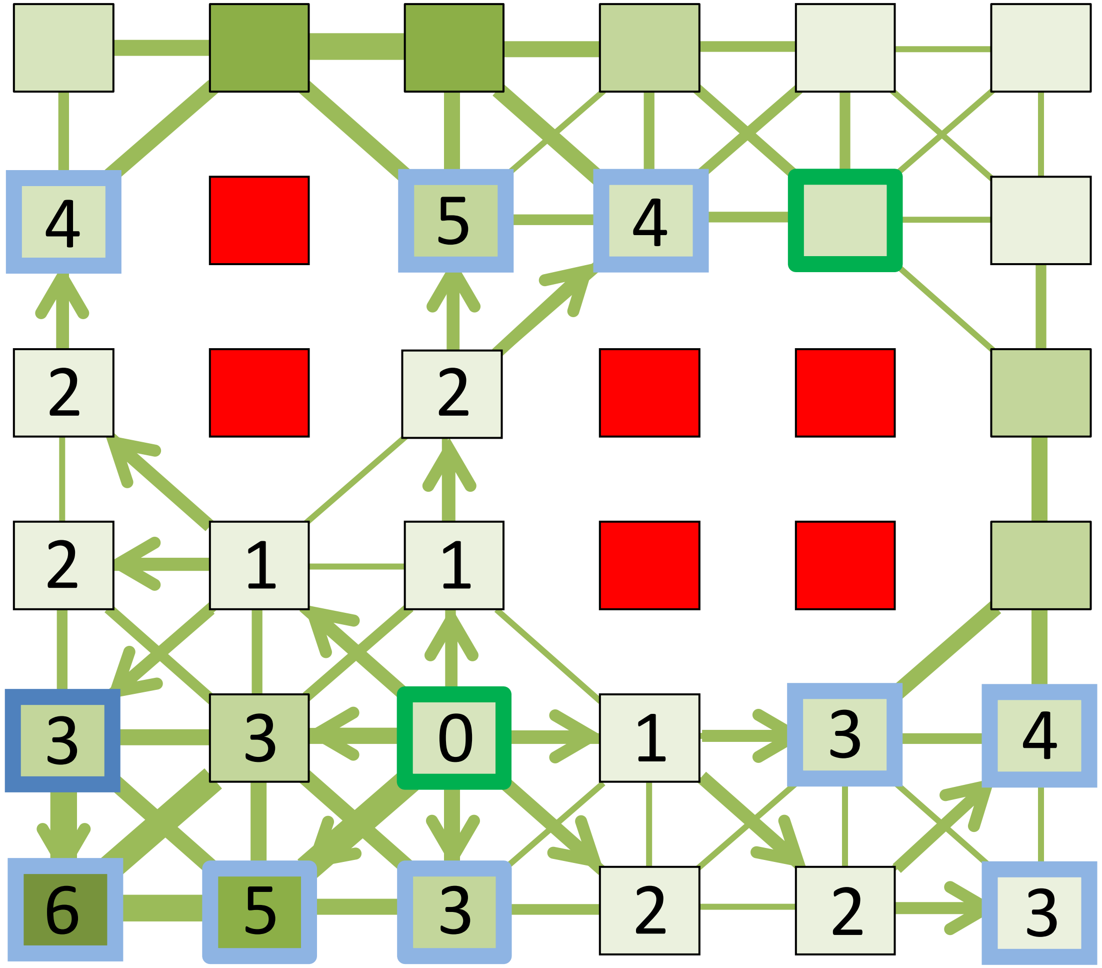
  Add all new neighbors to *boundary*

  Update costs of new neighbors

  Remove q from *boundary*

  If a neighbor is q_{end}, STORE

Dijkstra's Algorithm



Pseudocode:

Start with i = 0 steps at q_{start}

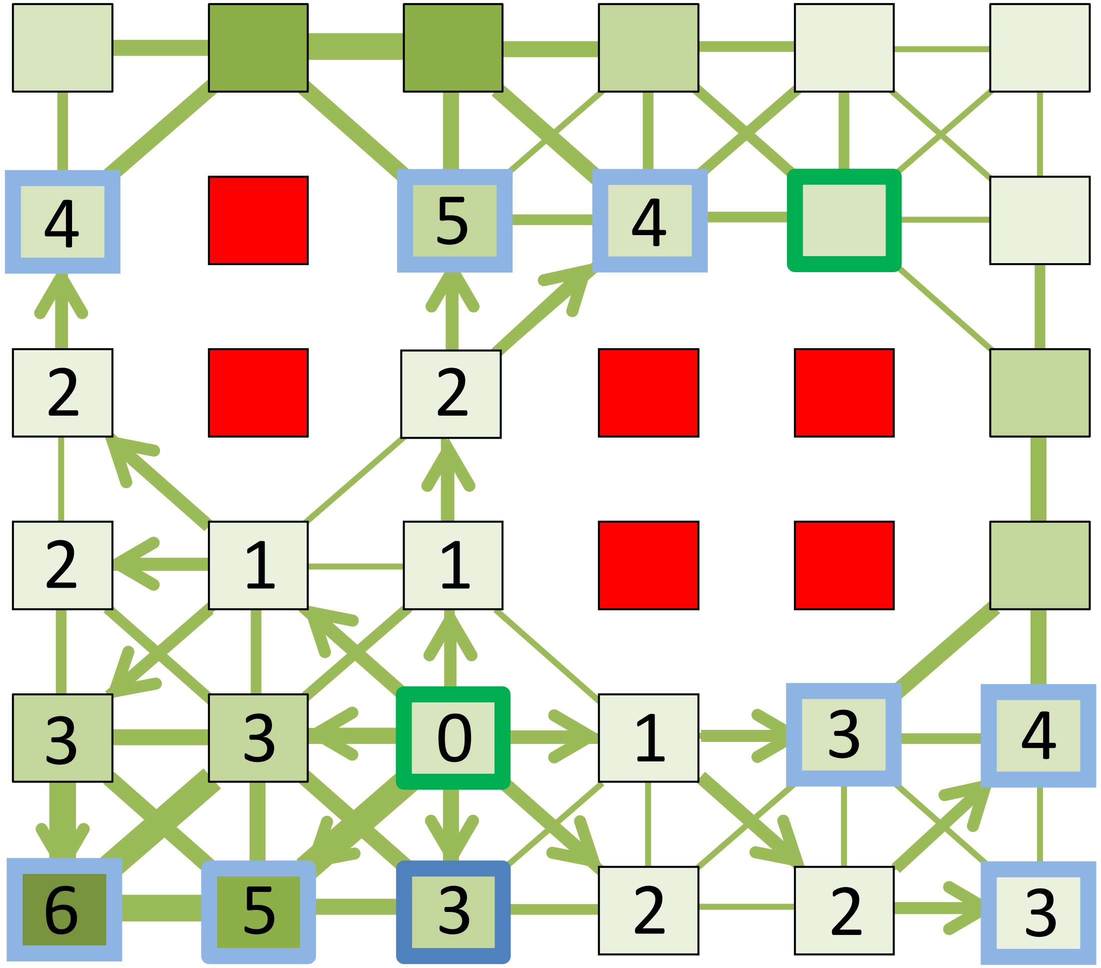
Add neighbors of q_{start} to *boundary*

Update costs of neighbors

While ~empty(*boundary*)

- q = *boundary* cell with min cost
- Add all new neighbors to *boundary*
- Update costs of new neighbors
- Remove q from *boundary*
- If a neighbor is q_{end}, STORE

Dijkstra's Algorithm



Pseudocode:

Start with i = 0 steps at q_{start}

Add neighbors of q_{start} to *boundary*

Update costs of neighbors

While ~empty(*boundary*)

  q = *boundary* cell with min cost

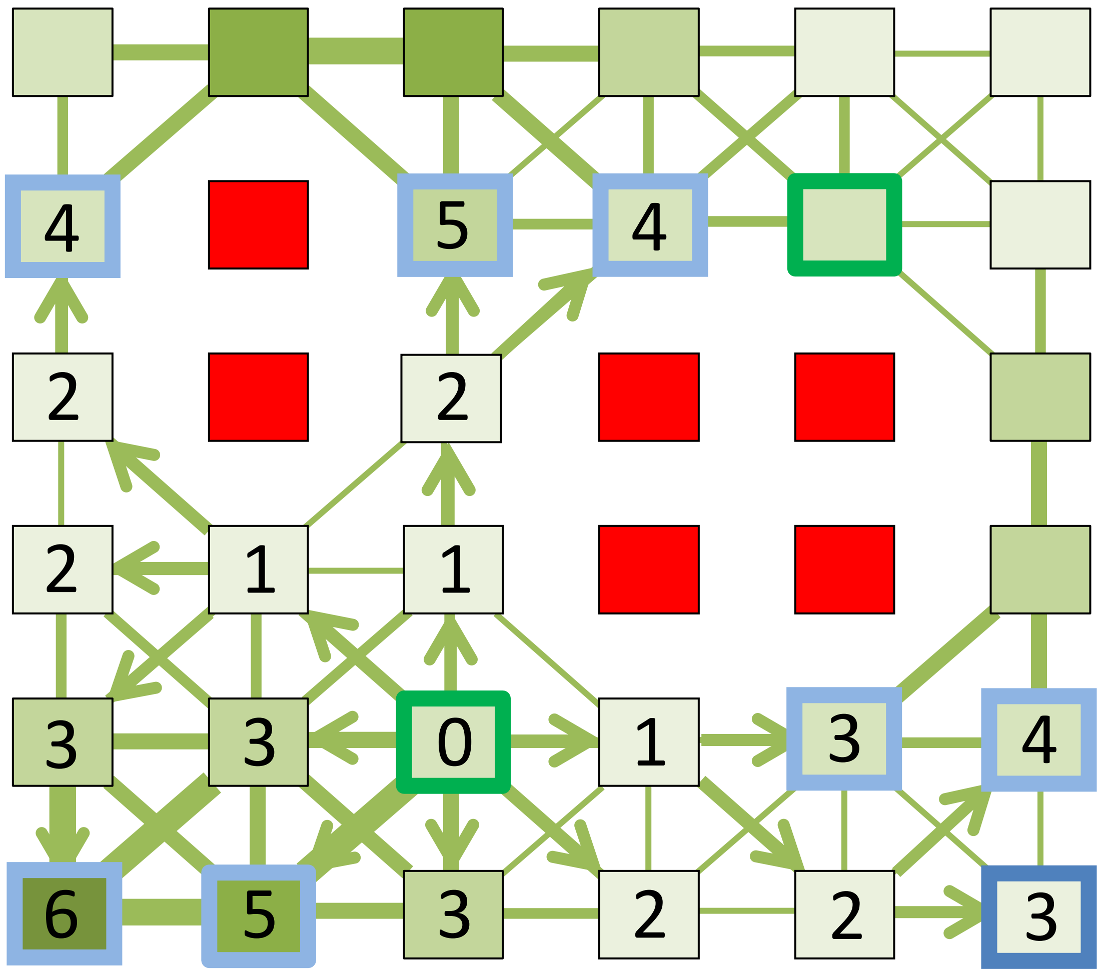
  Add all new neighbors to *boundary*

  Update costs of new neighbors

  Remove q from *boundary*

  If a neighbor is q_{end}, STORE

Dijkstra's Algorithm



Pseudocode:

Start with i = 0 steps at q_{start}

Add neighbors of q_{start} to *boundary*

Update costs of neighbors

While ~empty(*boundary*)

  q = *boundary* cell with min cost

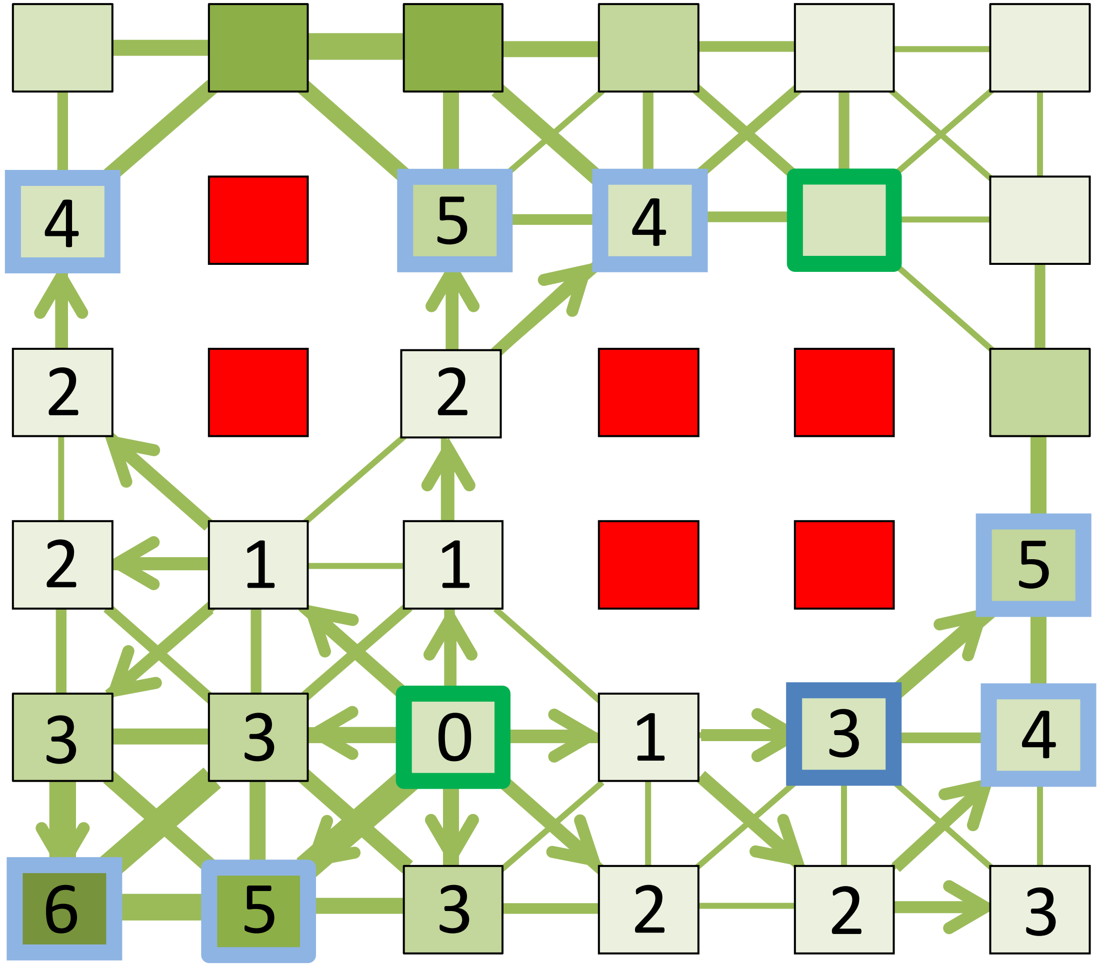
  Add all new neighbors to *boundary*

  Update costs of new neighbors

  Remove q from *boundary*

  If a neighbor is q_{end}, STORE

Dijkstra's Algorithm



Pseudocode:

Start with i = 0 steps at q_{start}

Add neighbors of q_{start} to *boundary*

Update costs of neighbors

While ~empty(*boundary*)

  q = *boundary* cell with min cost

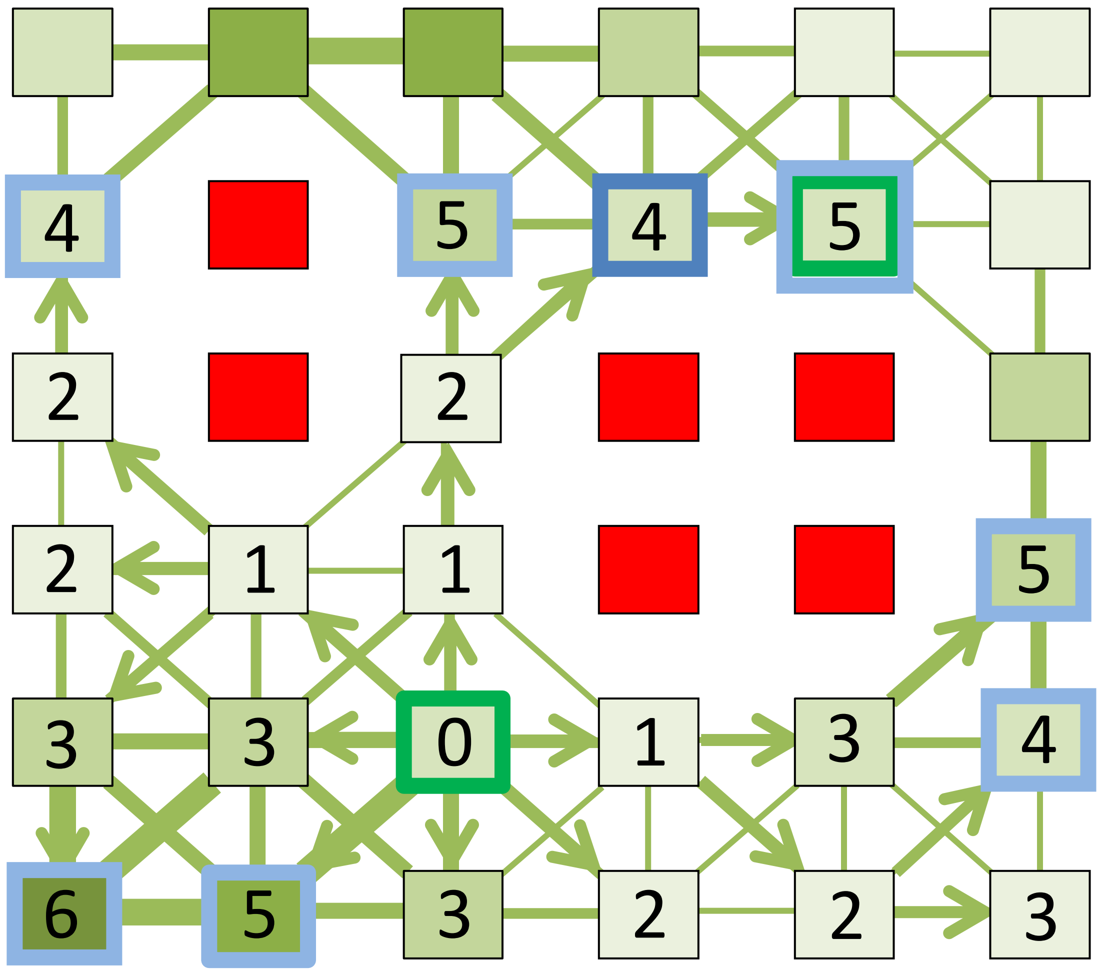
  Add all new neighbors to *boundary*

  Update costs of new neighbors

  Remove q from *boundary*

  If a neighbor is q_{end}, STORE

Dijkstra's Algorithm



Pseudocode:

Start with i = 0 steps at q_{start}

Add neighbors of q_{start} to *boundary*

Update costs of neighbors

While ~empty(*boundary*)

  q = *boundary* cell with min cost

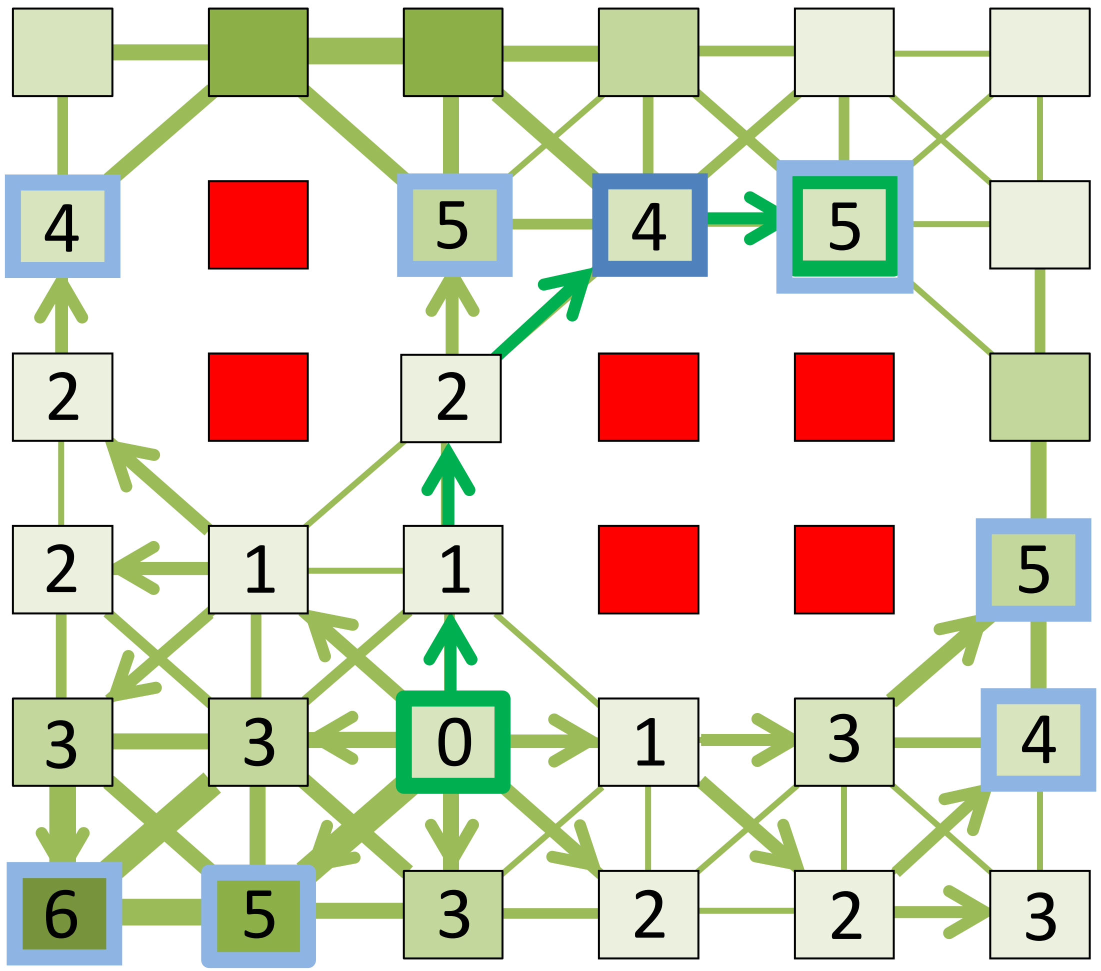
  Add all new neighbors to *boundary*

  Update costs of new neighbors

  Remove q from *boundary*

  If a neighbor is q_{end}, STORE

Dijkstra's Algorithm



Pseudocode:

Start with i = 0 steps at q_{start}

Add neighbors of q_{start} to *boundary*

Update costs of neighbors

While ~empty(*boundary*)

  q = *boundary* cell with min cost

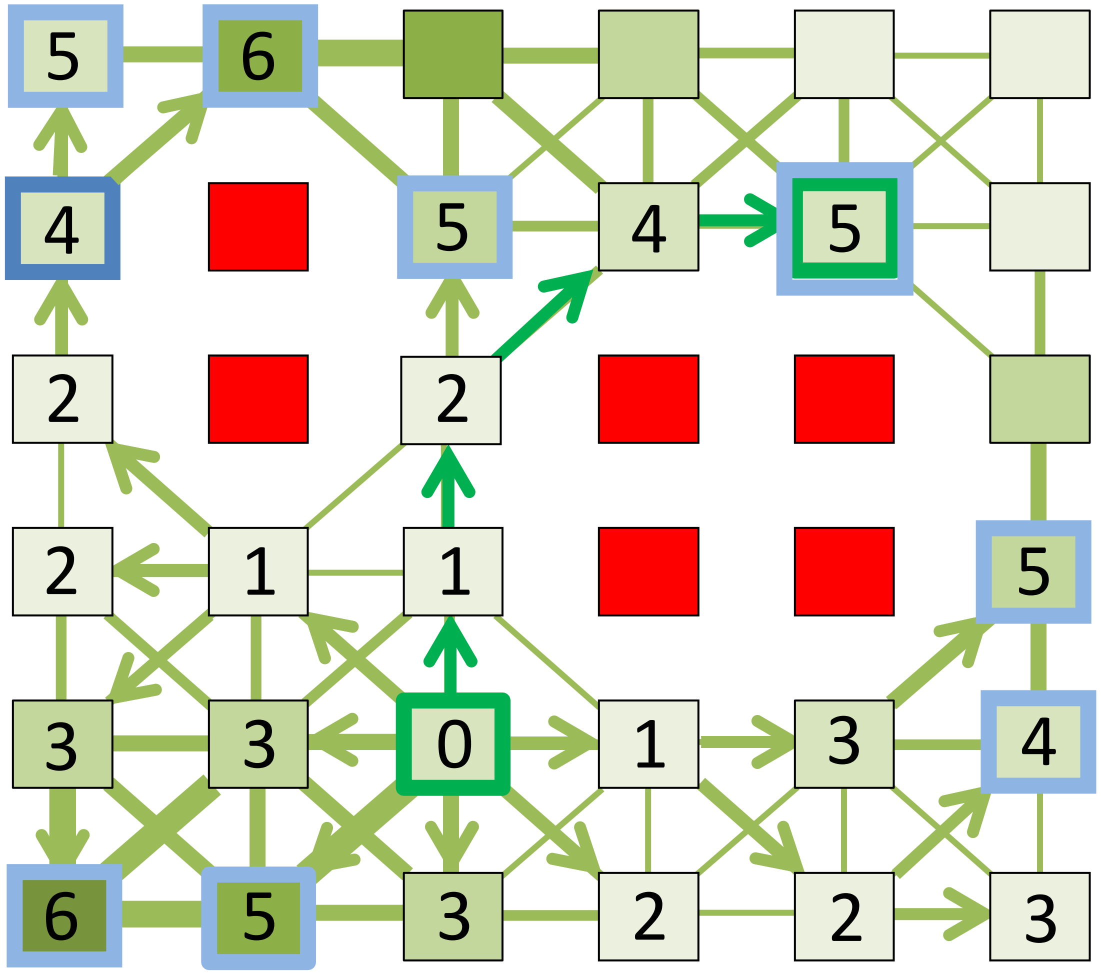
  Add all new neighbors to *boundary*

  Update costs of new neighbors

  Remove q from *boundary*

  If a neighbor is q_{end}, STORE

Dijkstra's Algorithm



Pseudocode:

Start with i = 0 steps at q_{start}

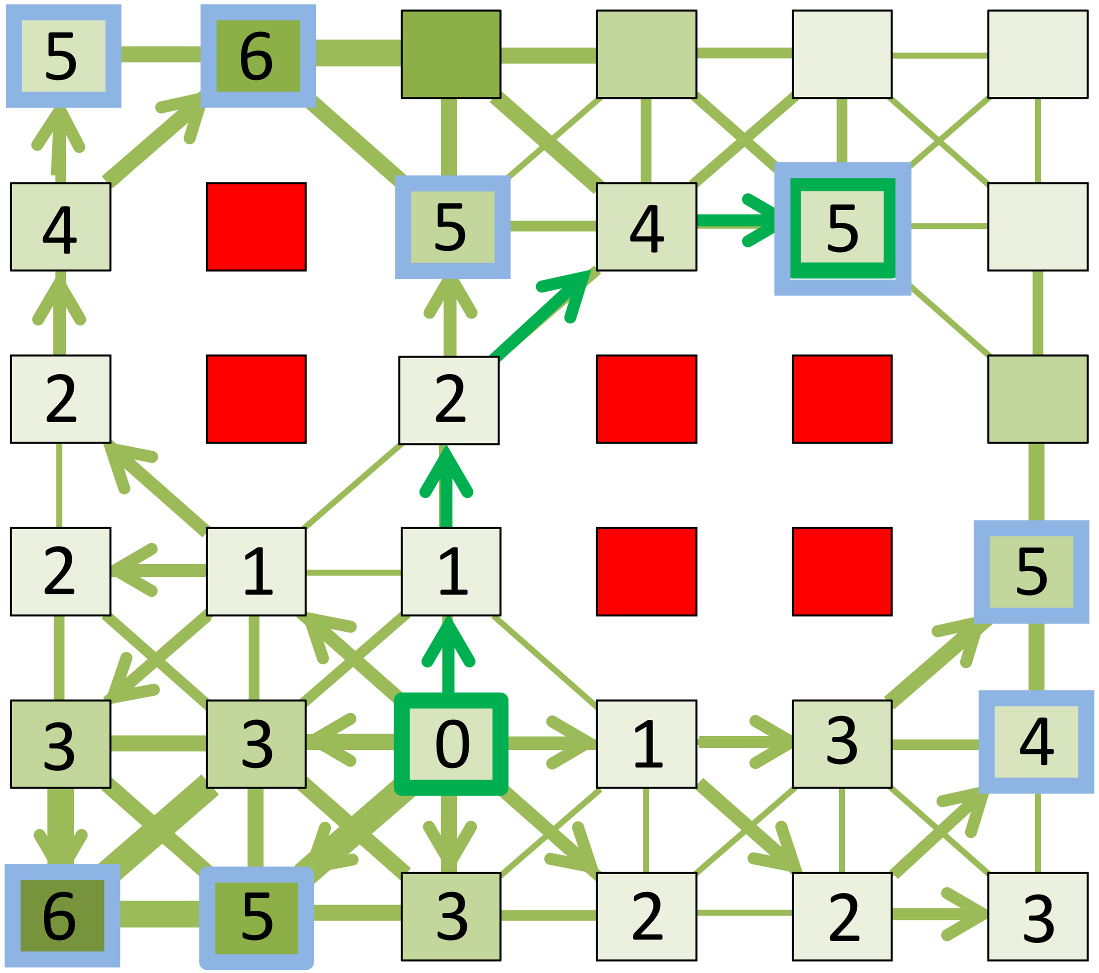
Add neighbors of q_{start} to *boundary*

Update costs of neighbors

While ~empty(*boundary*)

- q = *boundary* cell with min cost
- Add all new neighbors to *boundary*
- Update costs of new neighbors
- Remove q from *boundary*
- If a neighbor is q_{end}, STORE

Dijkstra's Algorithm



Pseudocode:

Start with i = 0 steps at q_{start}

Add neighbors of q_{start} to *boundary*

Update costs of neighbors

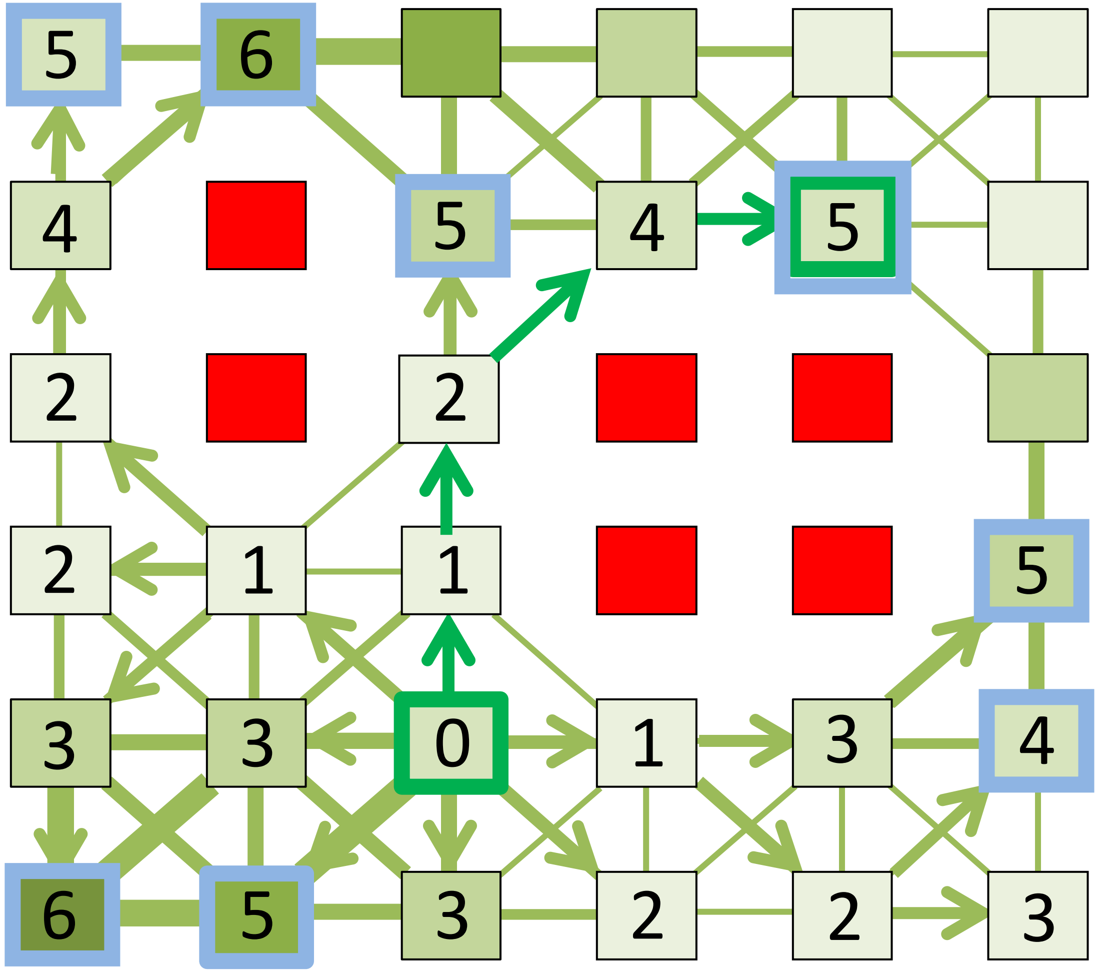
While ~empty(*boundary*)

- q = *boundary* cell with min cost
- Add all new neighbors to *boundary*
- Update costs of new neighbors
- Remove q from *boundary*
- If a neighbor is q_{end}, STORE
- If mincost(*boundary*) ≥ cost(q_{end}), STOP

Potentially search all cells:

Computation is $O(N_{cell})$

Dijkstra's Algorithm



Can we make this more efficient?

Pseudocode:

Start with i = 0 steps at q_{start}

Add neighbors of q_{start} to *boundary*

Update costs of neighbors

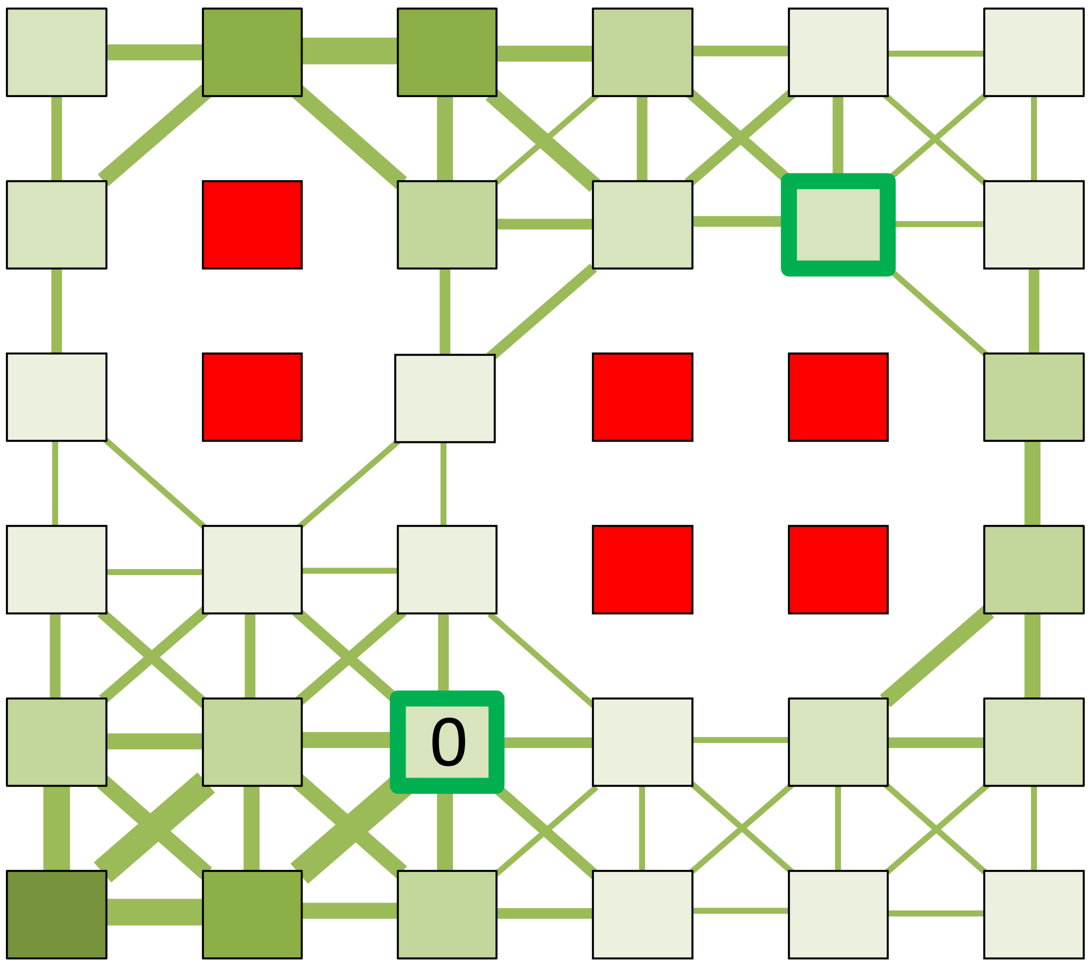
While ~empty(*boundary*)

- q = *boundary* cell with min cost
- Add all new neighbors to *boundary*
- Update costs of new neighbors
- Remove q from *boundary*
- If a neighbor is q_{end}, STORE
- If mincost(*boundary*) ≥ cost(q_{end}), STOP

Potentially search all cells:

Computation is $O(N_{cell})$

A* Search



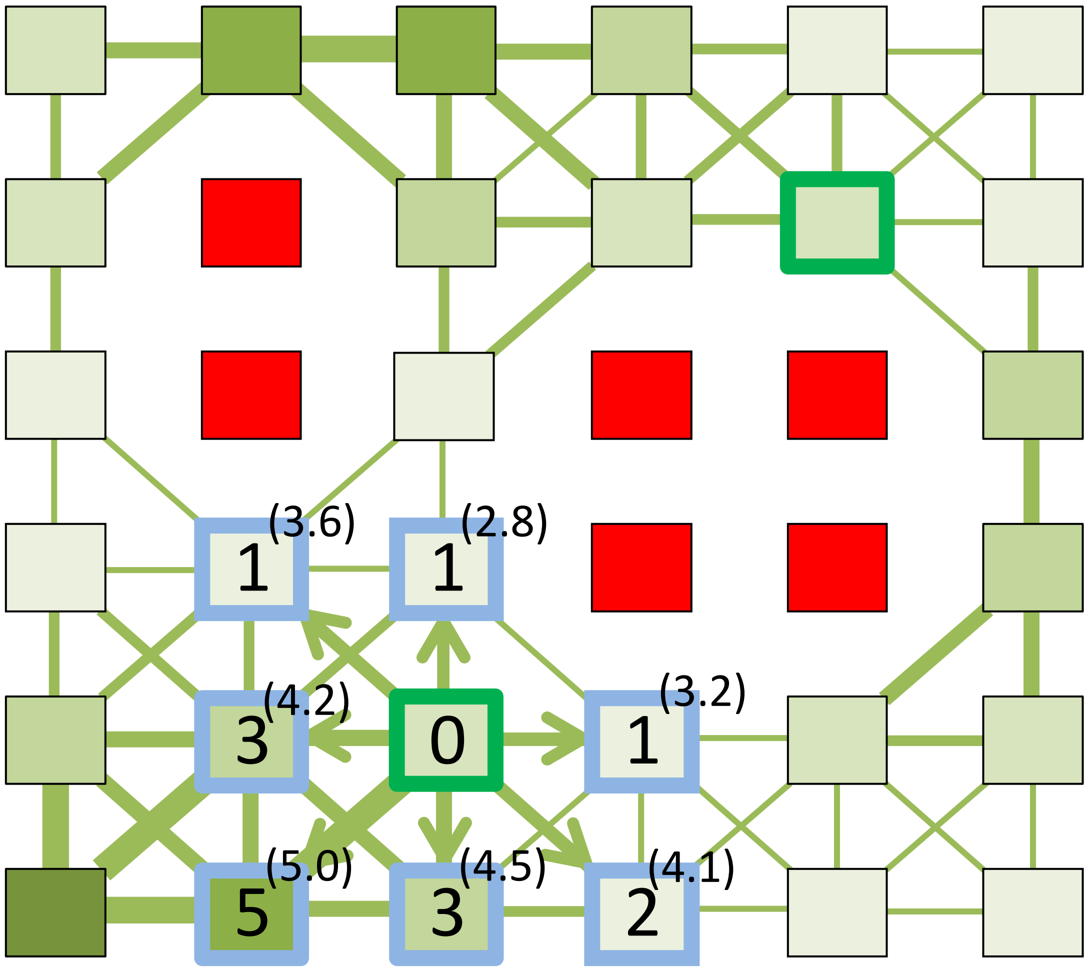
Idea: estimate remaining distance to the goal

Order vertices based on estimated distance

$$f(i) = \underbrace{g(i)}_{\text{cost from start}} + \underbrace{h(i)}_{\text{heuristic: estimated cost to goal}}$$

Let's try $h(i)$ = Euclidean distance to goal

A* Search



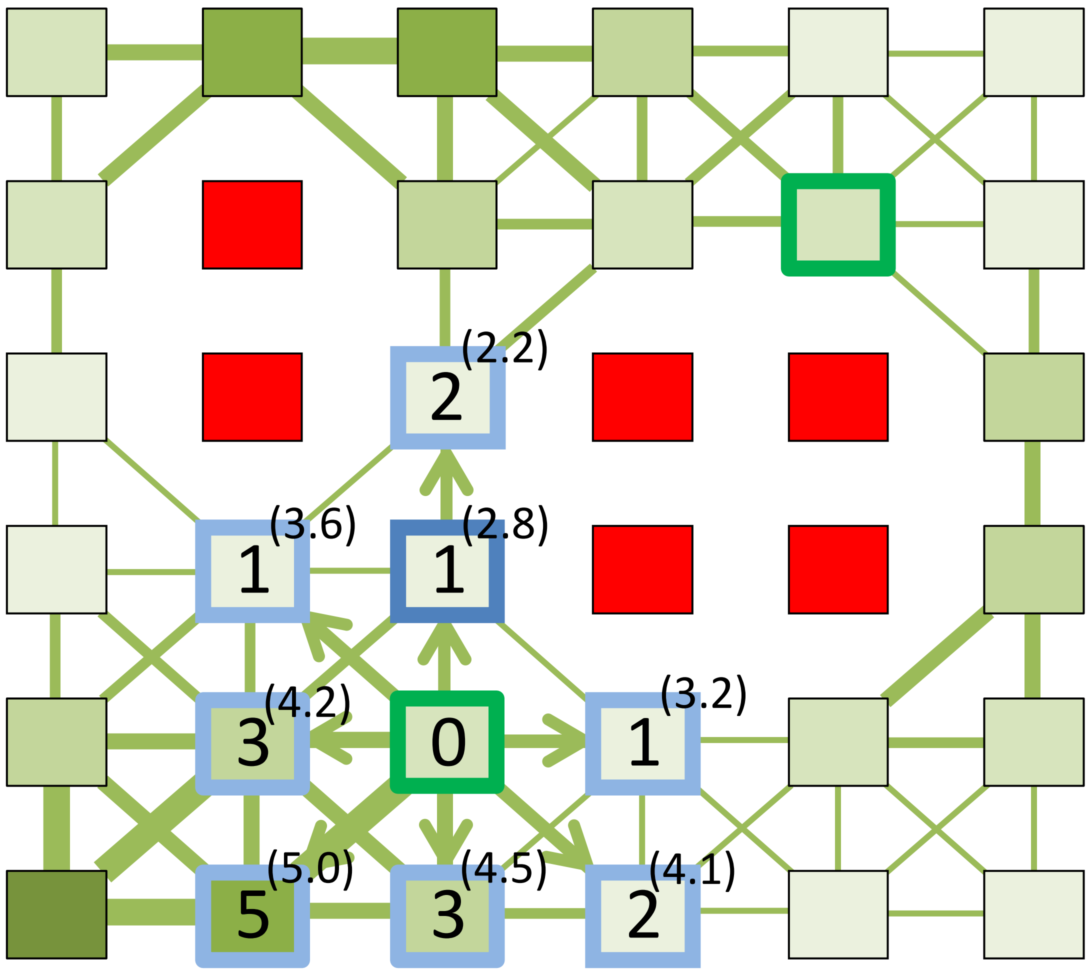
Idea: estimate remaining distance to the goal

Order vertices based on estimated distance

$$f(i) = \underbrace{g(i)}_{\text{cost from start}} + \underbrace{h(i)}_{\text{heuristic: estimated cost to goal}}$$

Let's try $h(i) =$ Euclidean distance to goal

A* Search



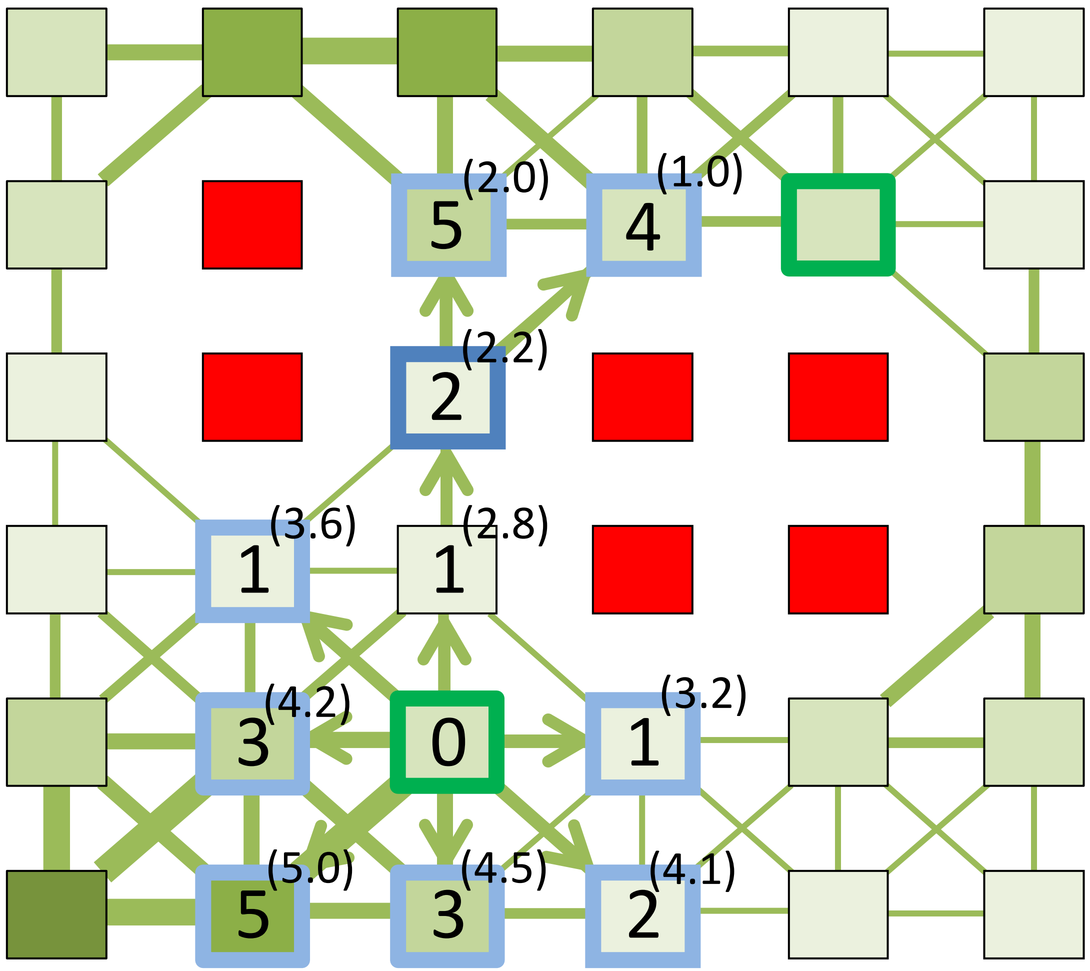
Idea: estimate remaining distance to the goal

Order vertices based on estimated distance

$$f(i) = \underbrace{g(i)}_{\text{cost from start}} + \underbrace{h(i)}_{\text{heuristic: estimated cost to goal}}$$

Let's try $h(i) =$ Euclidean distance to goal

A* Search



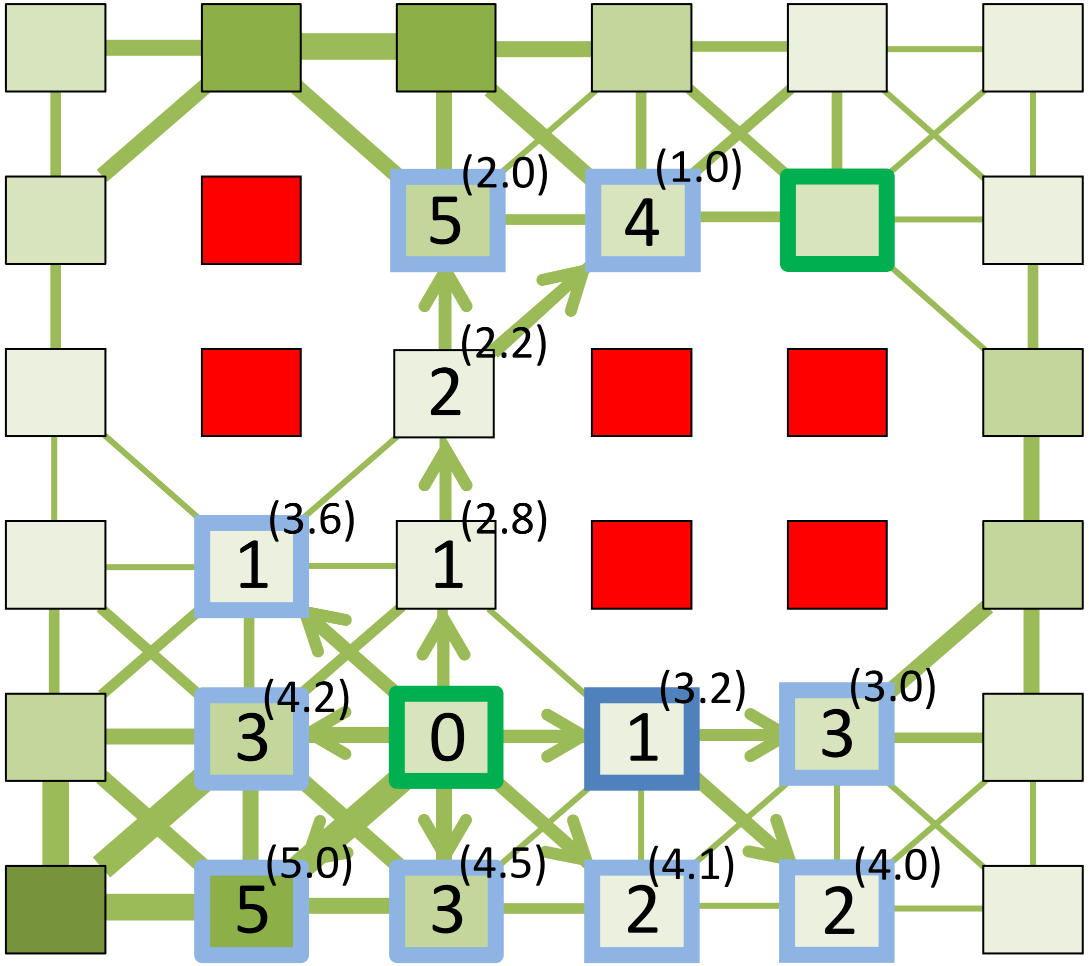
Idea: estimate remaining distance to the goal

Order vertices based on estimated distance

$$f(i) = \underbrace{g(i)}_{\text{cost from start}} + \underbrace{h(i)}_{\text{heuristic: estimated cost to goal}}$$

Let's try $h(i)$ = Euclidean distance to goal

A* Search



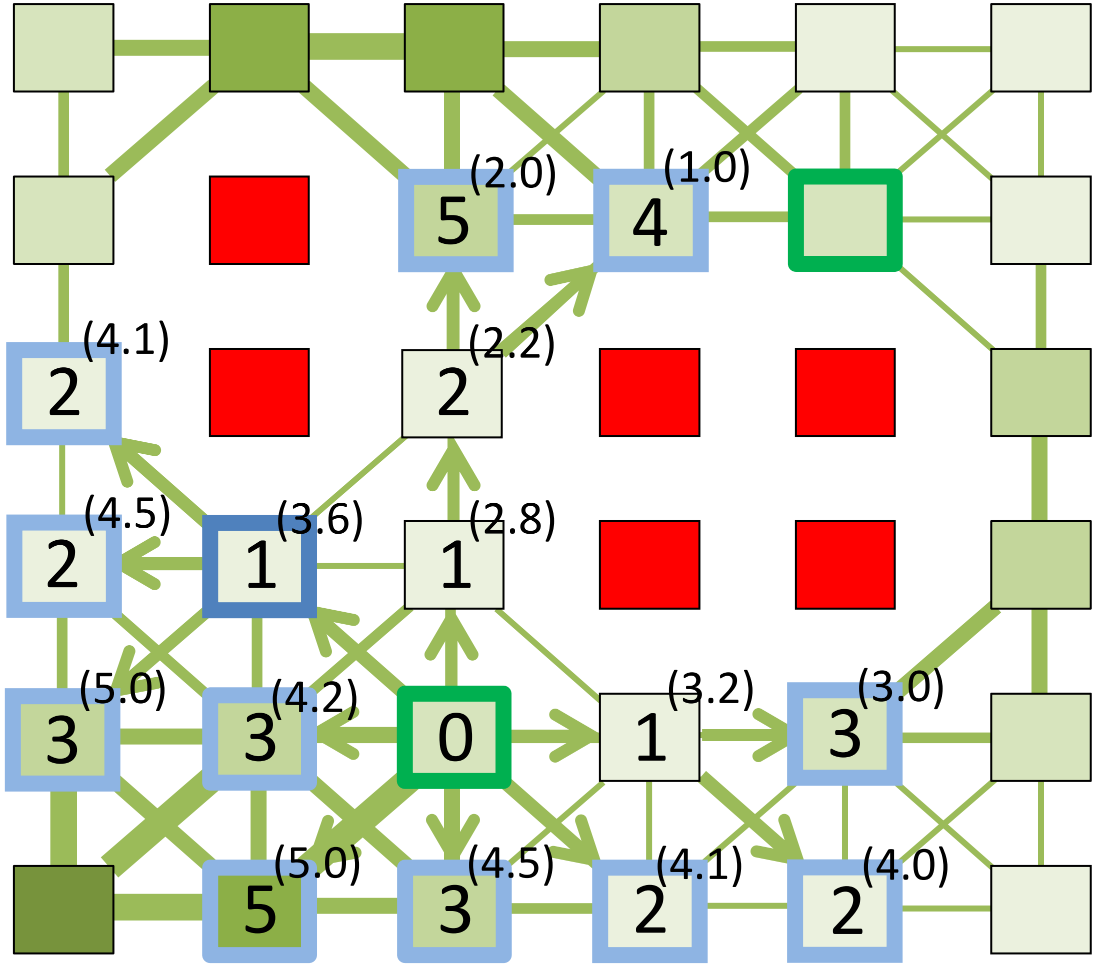
Idea: estimate remaining distance to the goal

Order vertices based on estimated distance

$$f(i) = \underbrace{g(i)}_{\text{cost from start}} + \underbrace{h(i)}_{\text{heuristic: estimated cost to goal}}$$

Let's try $h(i)$ = Euclidean distance to goal

A* Search



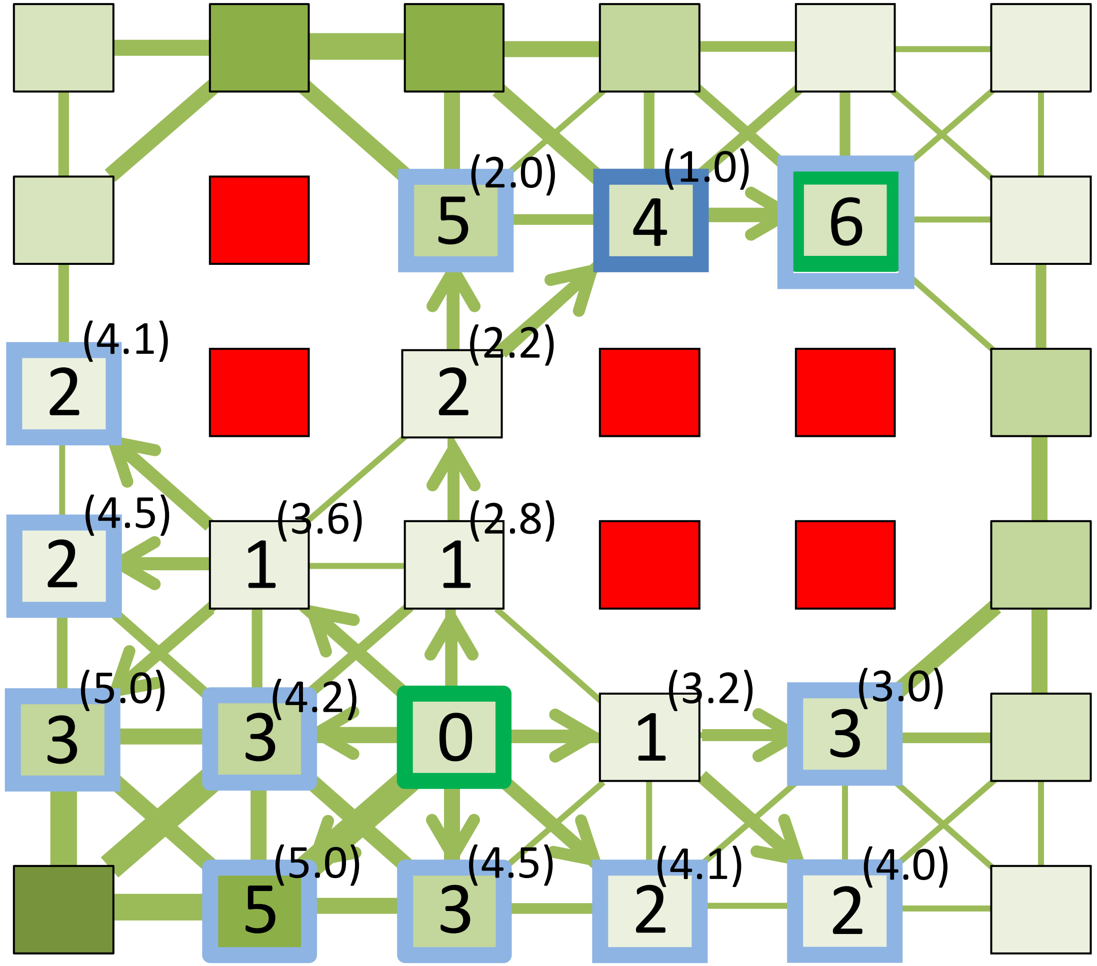
Idea: estimate remaining distance to the goal

Order vertices based on estimated distance

$$f(i) = \underbrace{g(i)}_{\text{cost from start}} + \underbrace{h(i)}_{\text{heuristic: estimated cost to goal}}$$

Let's try $h(i)$ = Euclidean distance to goal

A* Search



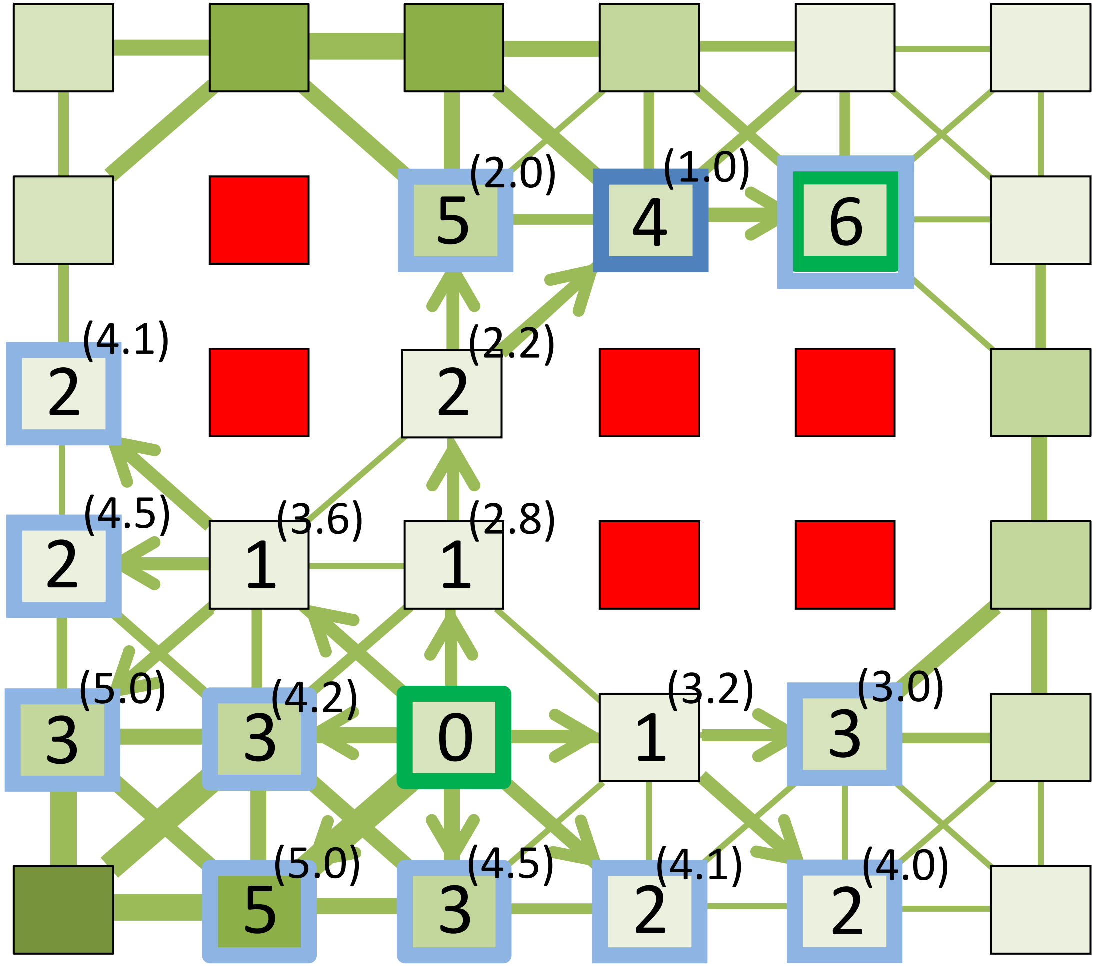
Idea: estimate remaining distance to the goal

Order vertices based on estimated distance

$$f(i) = \underbrace{g(i)}_{\text{cost from start}} + \underbrace{h(i)}_{\text{heuristic: estimated cost to goal}}$$

Let's try $h(i) =$ Euclidean distance to goal

A* Search



Idea: estimate remaining distance to the goal

Order vertices based on estimated distance

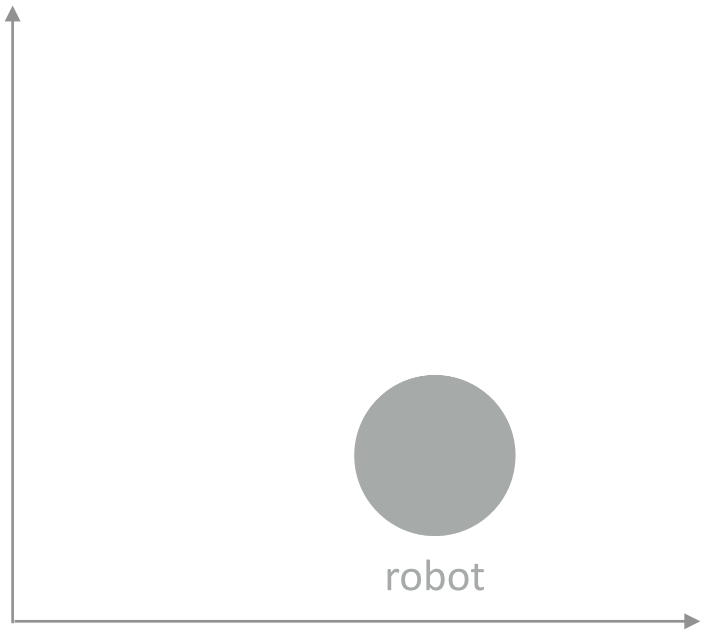
$$f(i) = \underbrace{g(i)}_{\text{cost from start}} + \underbrace{h(i)}_{\substack{\text{heuristic:} \\ \text{estimated cost to goal}}}$$

Let's try $h(i)$ = Euclidean distance to goal

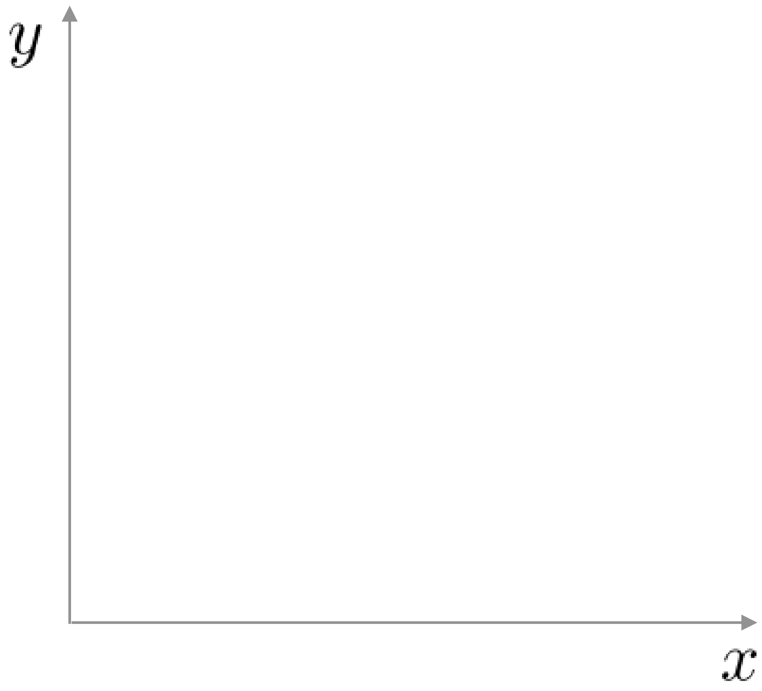
$h(i)$ must be **admissible**

Worst case computational cost?

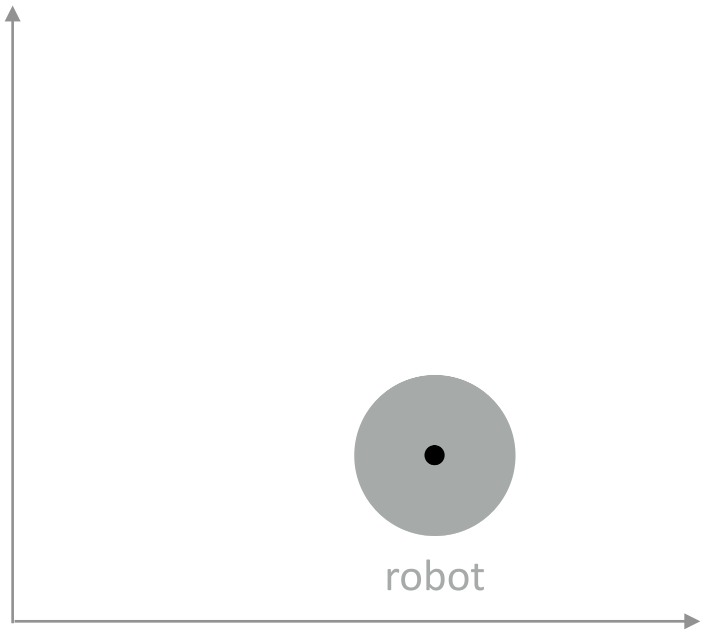
Non-Point/Non-Line Robots



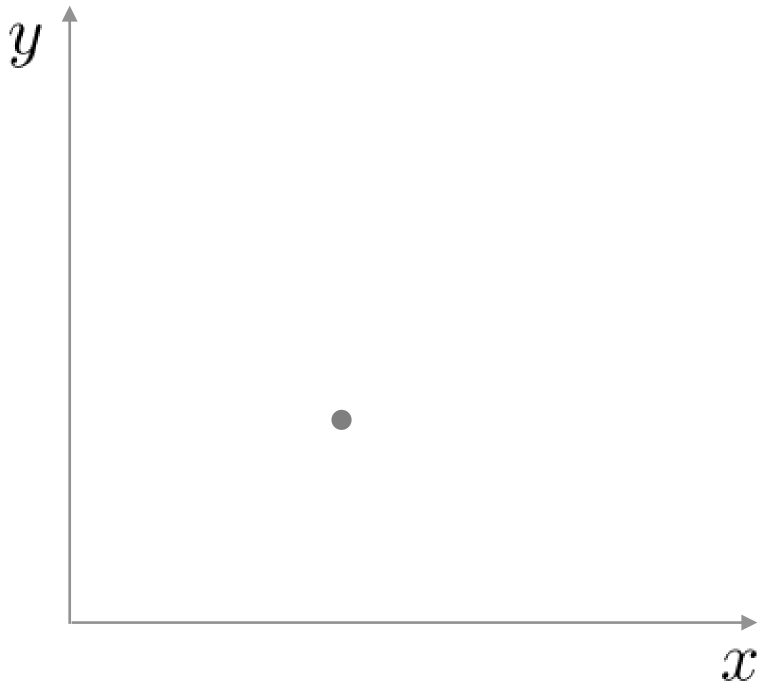
What does the configuration space look like for this round mobile robot (planar PP)?



Non-Point/Non-Line Robots

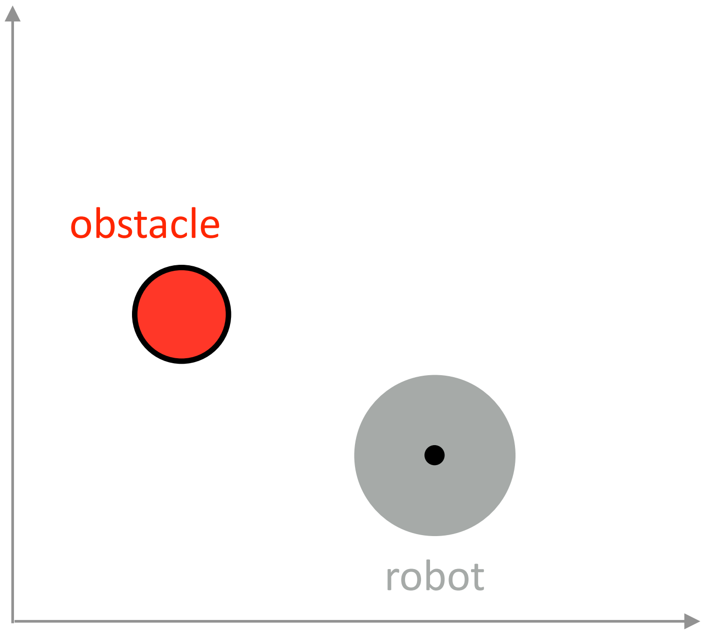


What does the robot look like in this configuration space?

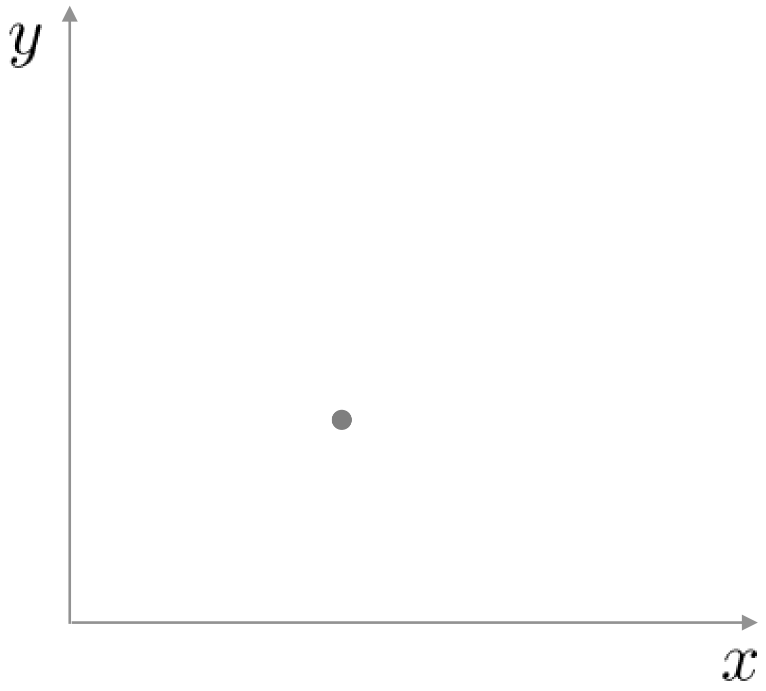


All robots are points in their configuration space!

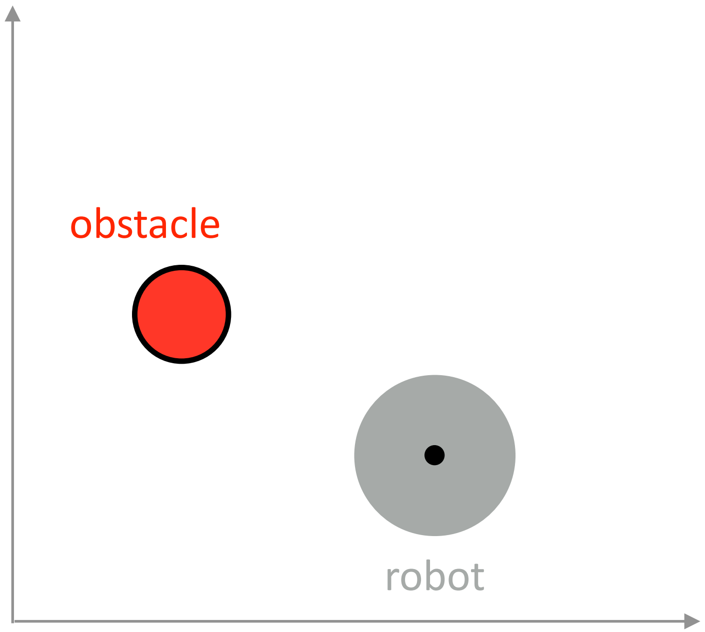
Non-Point/Non-Line Robots



What does the free configuration space look like for this round mobile robot (planar PP) with one small round obstacle in the workspace?

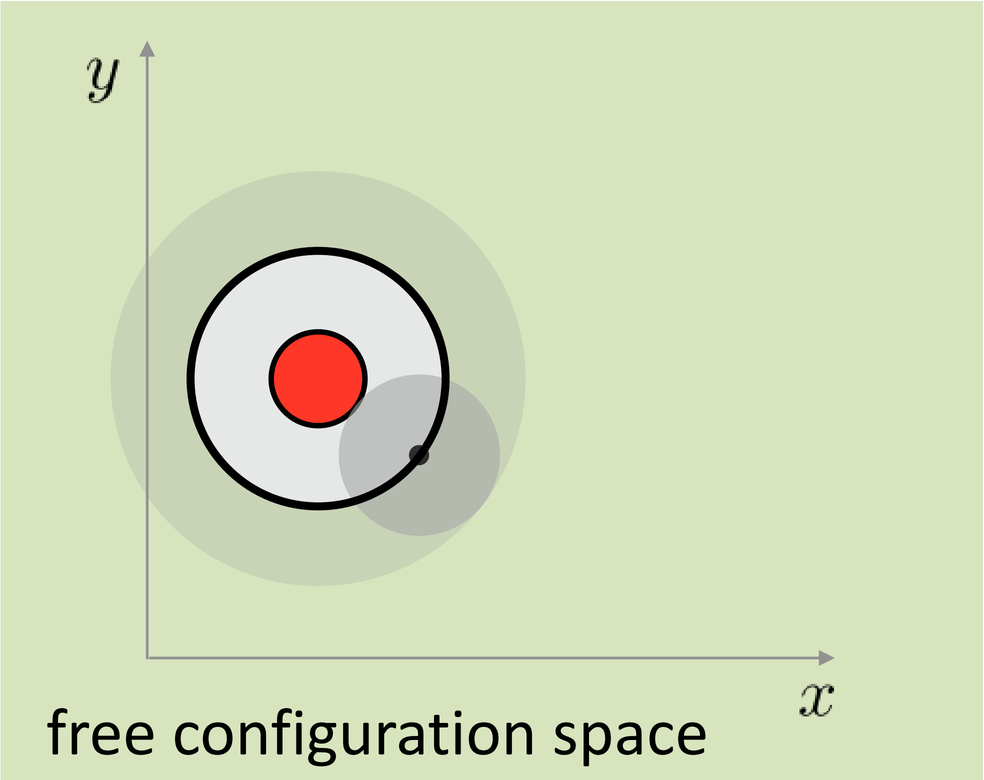


Non-Point/Non-Line Robots

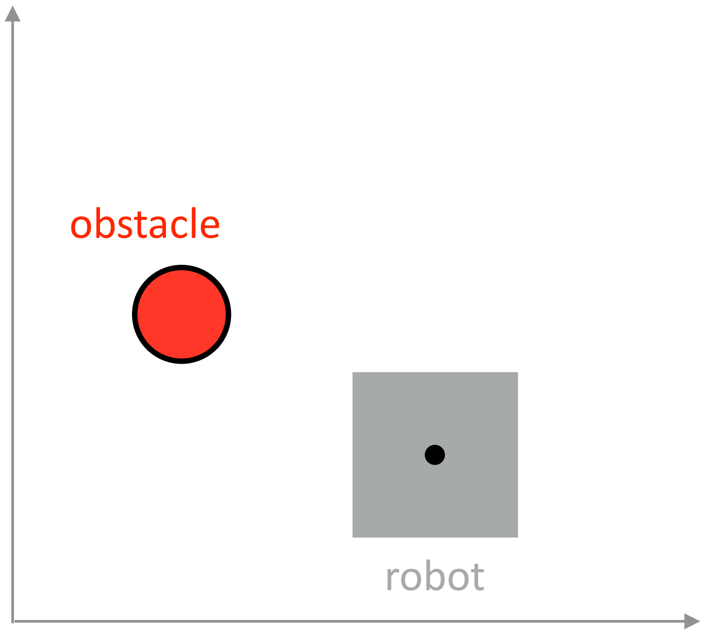


What does the free configuration space look like for this round mobile robot (planar PP) with one small round obstacle in the workspace?

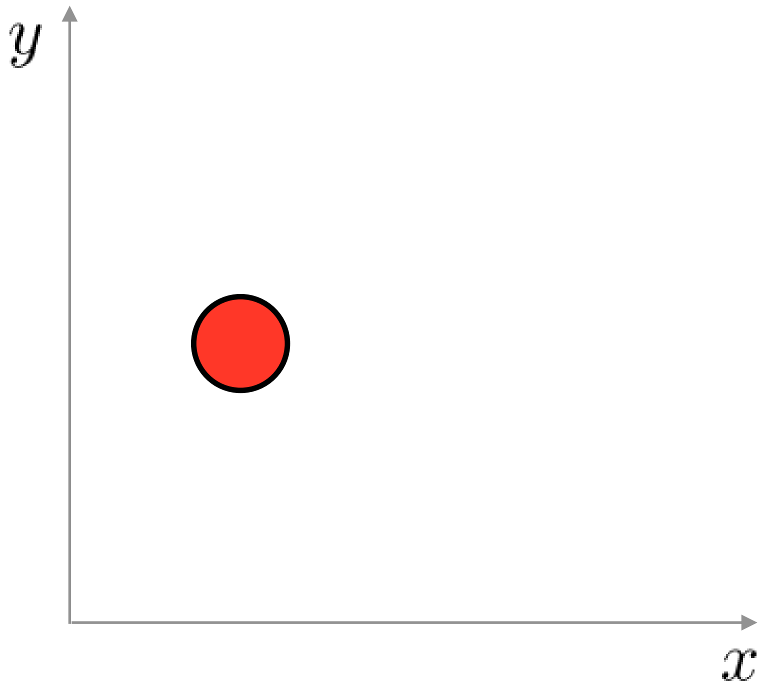
For a round robot, the obstacles simply grow by the robot's radius.



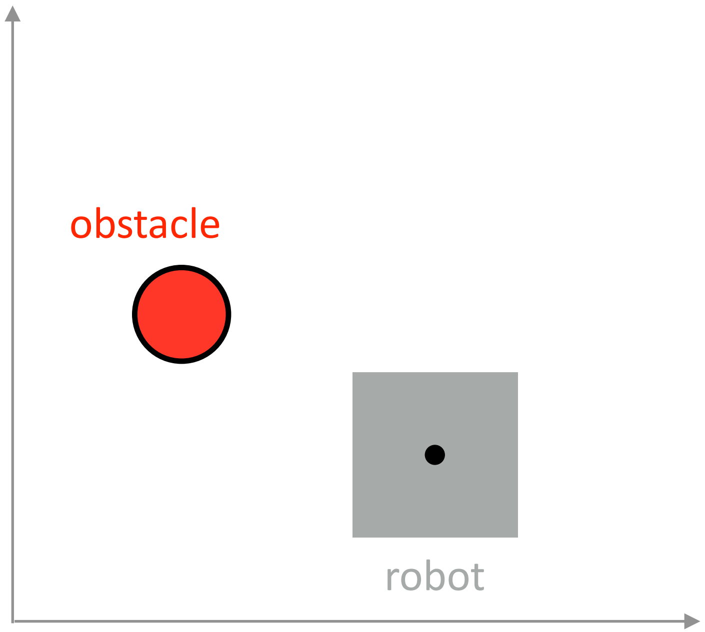
Non-Point/Non-Line Robots



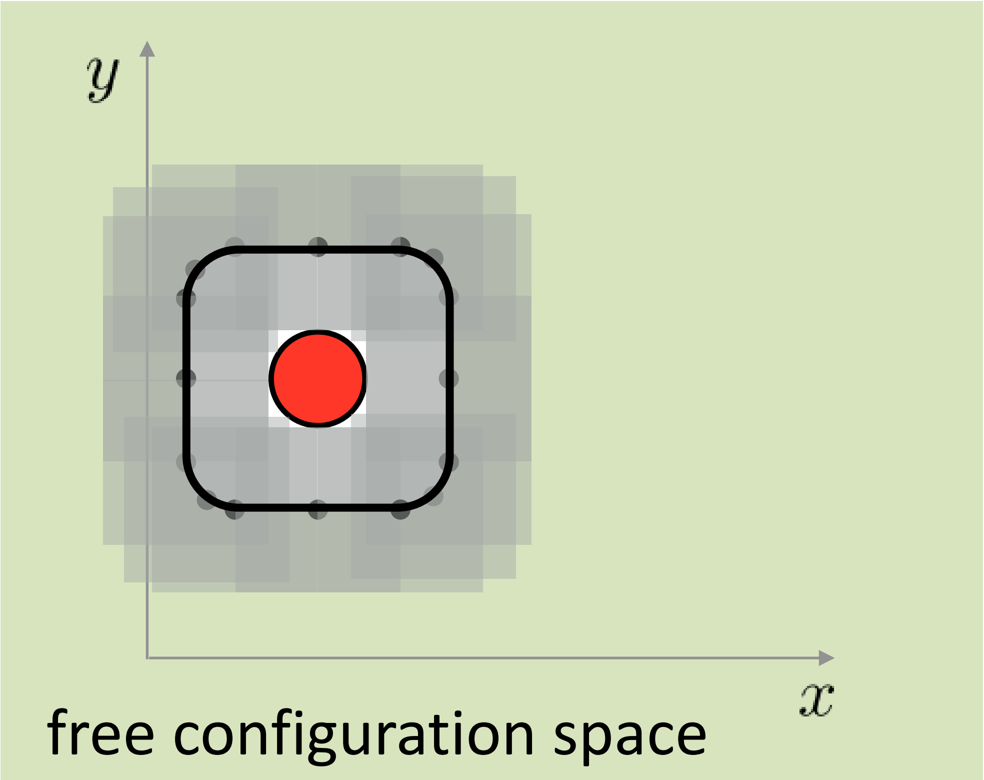
What does the free configuration space look like for this square non-rotating mobile robot with one small round obstacle in the workspace?



Non-Point/Non-Line Robots

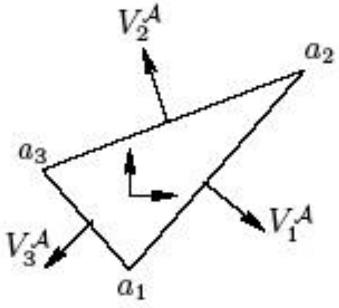


What does the free configuration space look like for this square non-rotating mobile robot with one small round obstacle in the workspace?

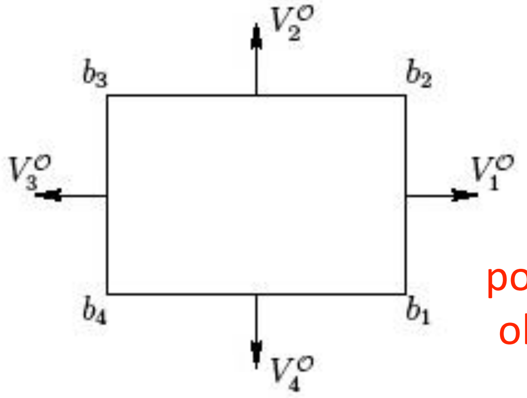


Minkowski Sum

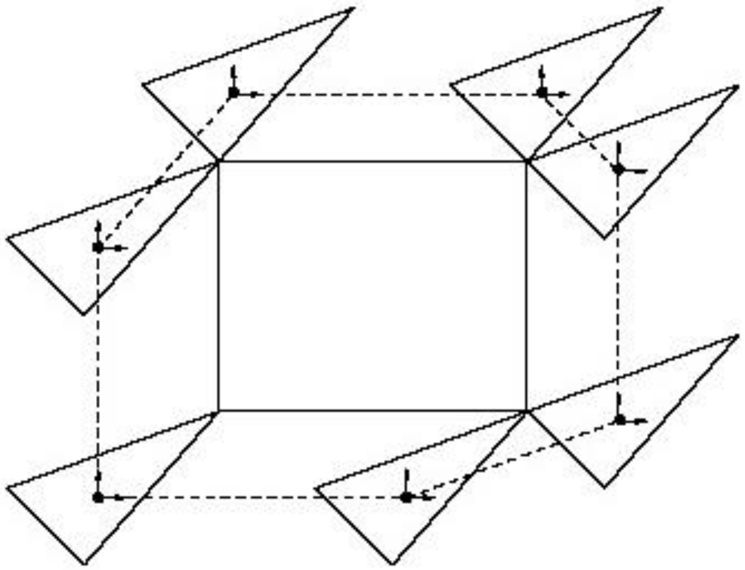
Places the end-effector at all positions around the obstacle that involve vertex-to-vertex contact.



polygonal robot

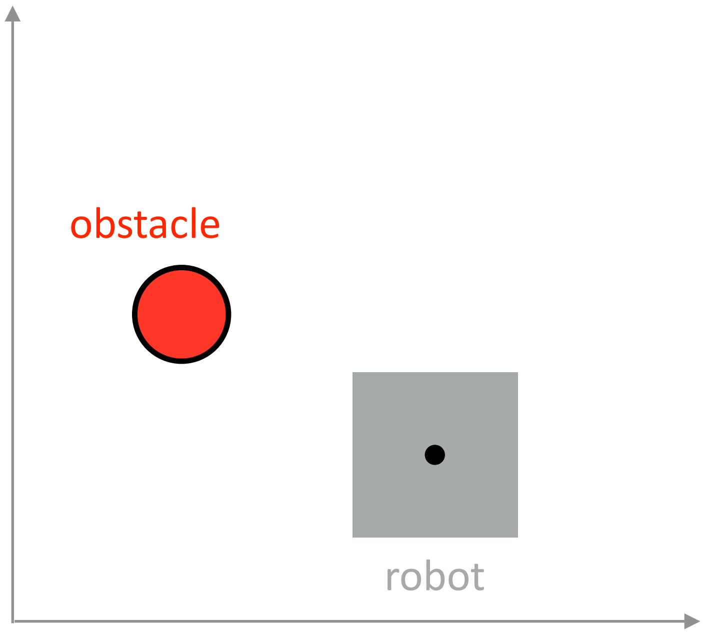


polygonal obstacle

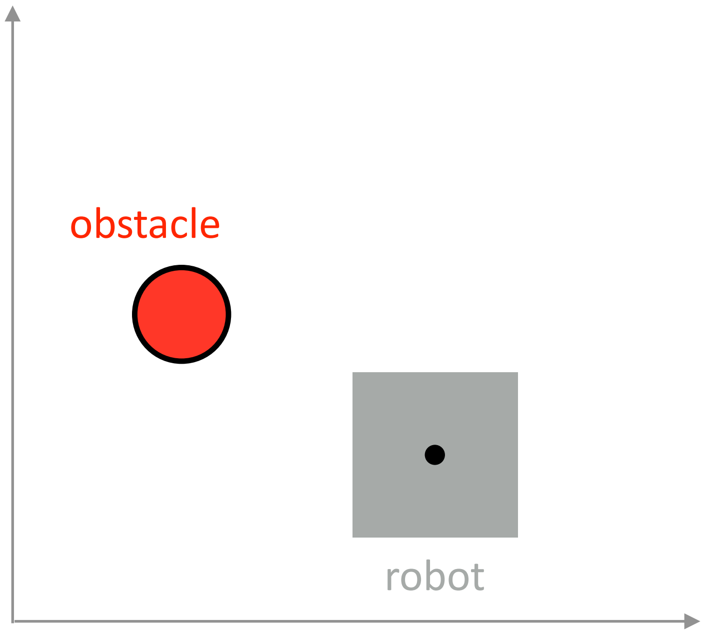


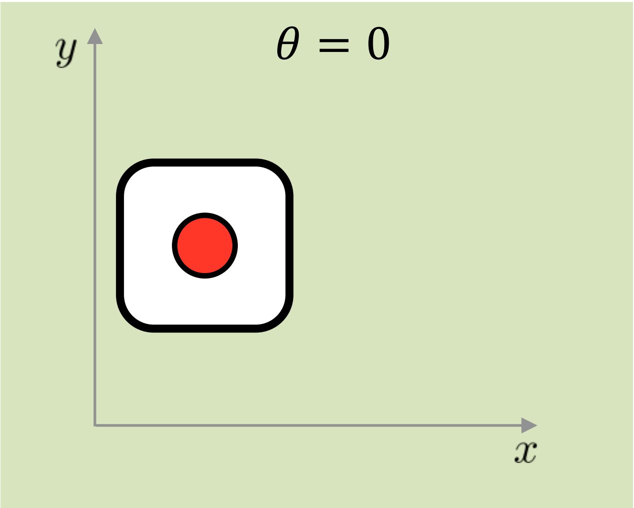
- For each pair V_j^O and V_{j-1}^O, if V_i^A points between $-V_j^O$ and $-V_{j-1}^O$ then add to QO the vertices $b_j - a_i$ and $b_j - a_{i+1}$
- For each pair V_i^A and V_{i-1}^A, if V_j^O points between $-V_i^A$ and $-V_{i-1}^A$ then add to QO the vertices $b_j - a_i$ and $b_{j+1} - a_i$

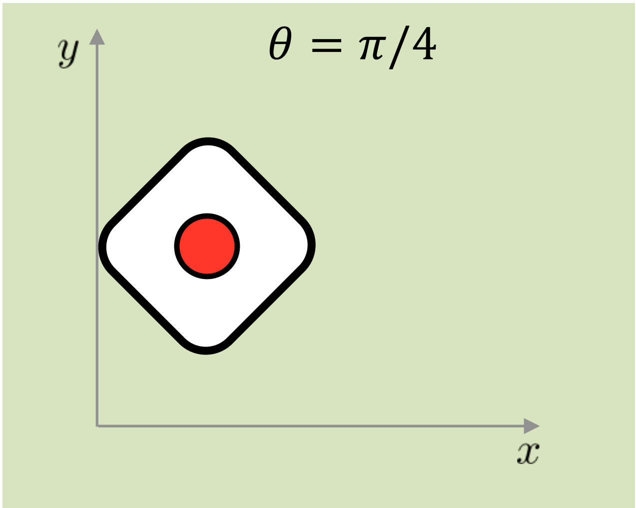
Rotating Non-Point Robots in the Plane

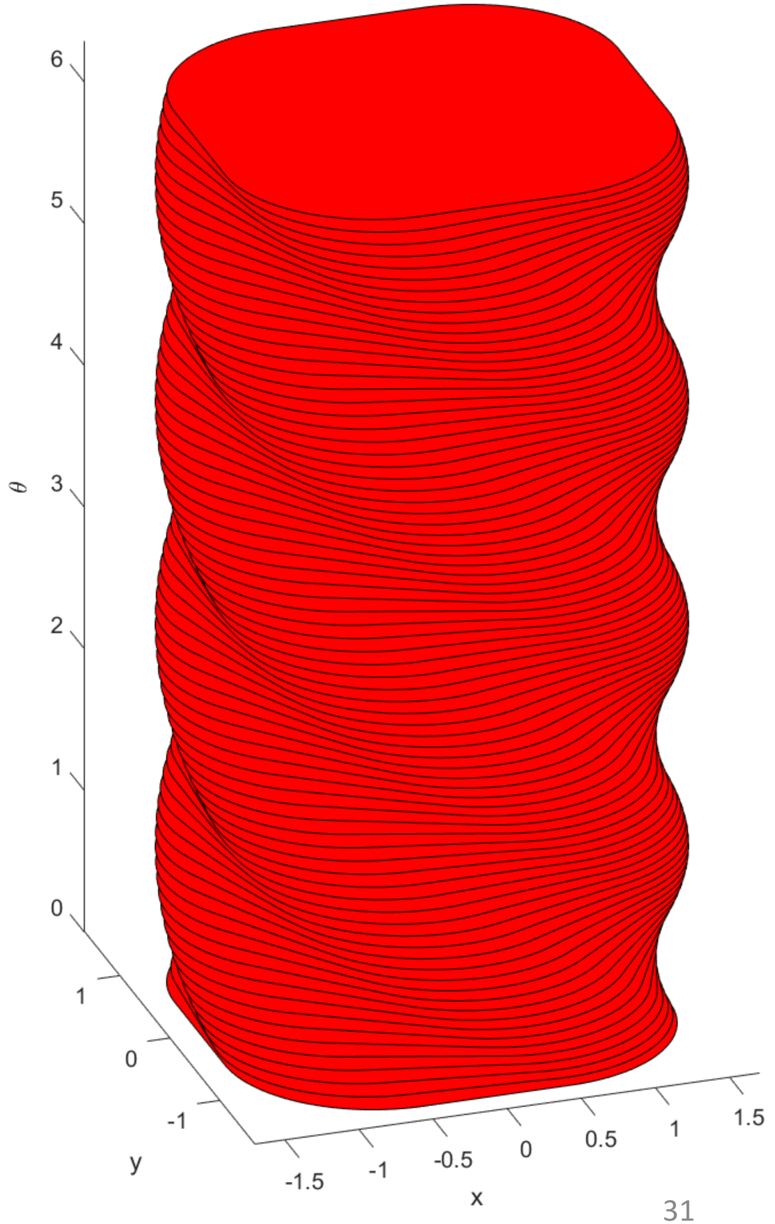


Rotating Non-Point Robots in the Plane

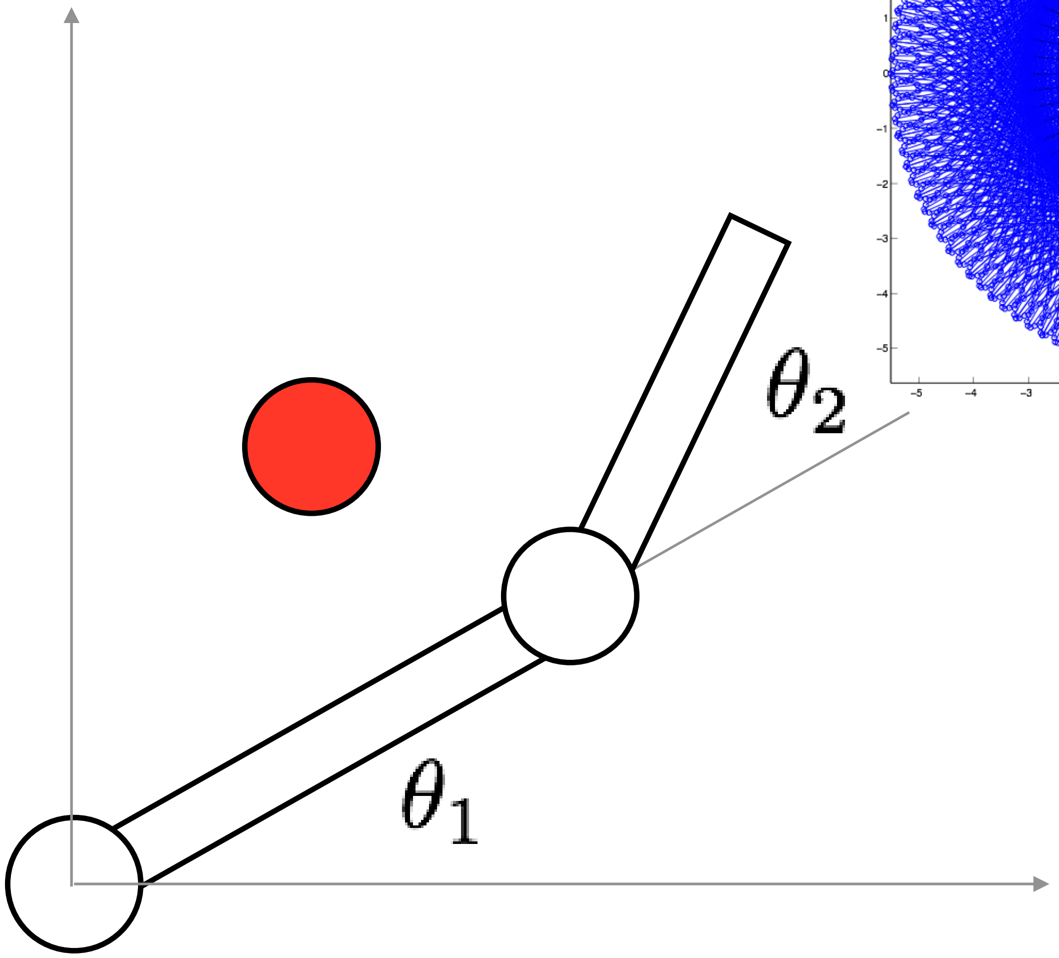




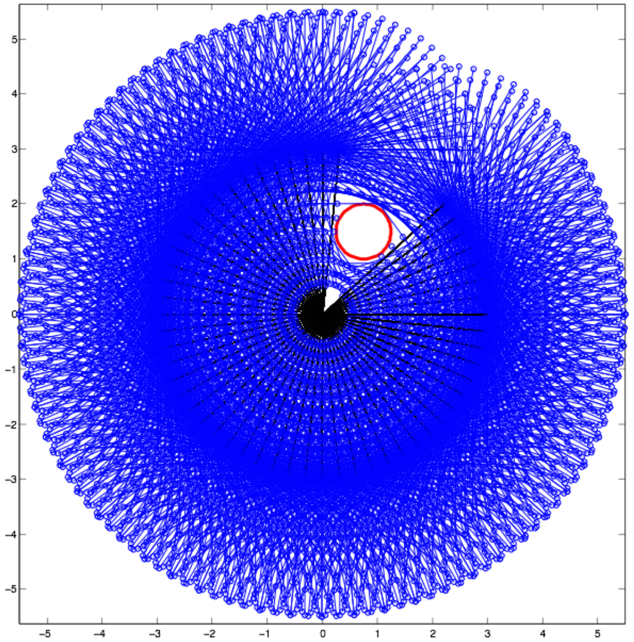




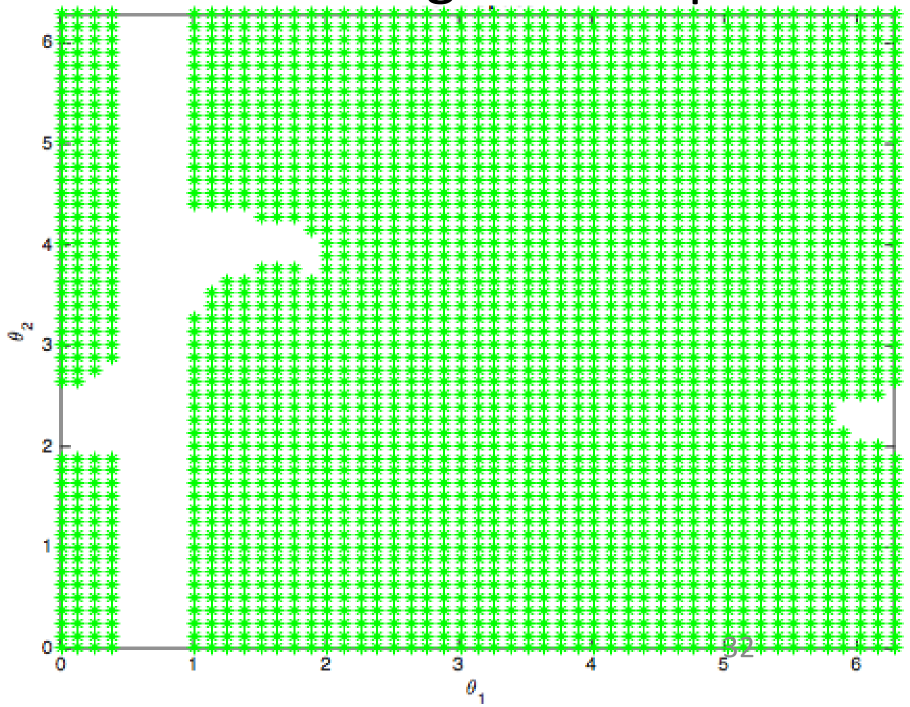
2-Link Manipulator



Workspace



Free Configuration Space

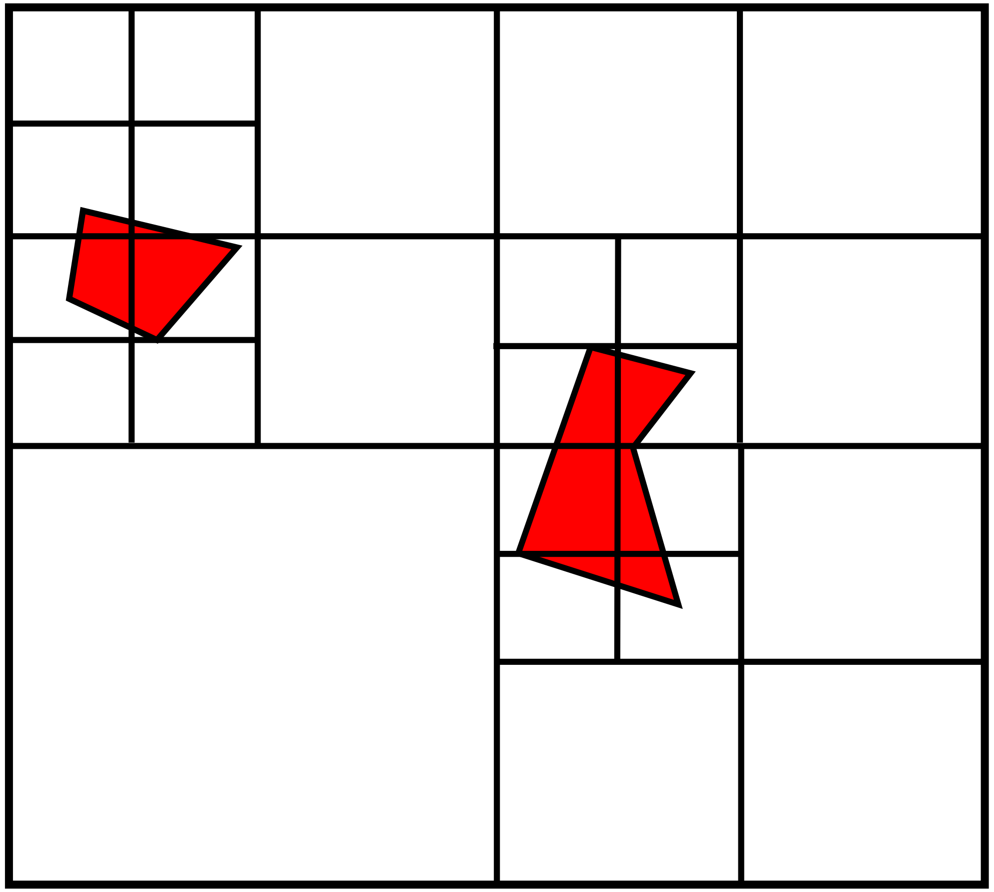


Computational complexity of a trajectory planner grows with the size of the configuration space.

Complete planners have to search every cell of the discretized space in the worst case.

Worst case complexity is **exponential** in the robot dof (number of joints for a manipulator): $O(c^J)$

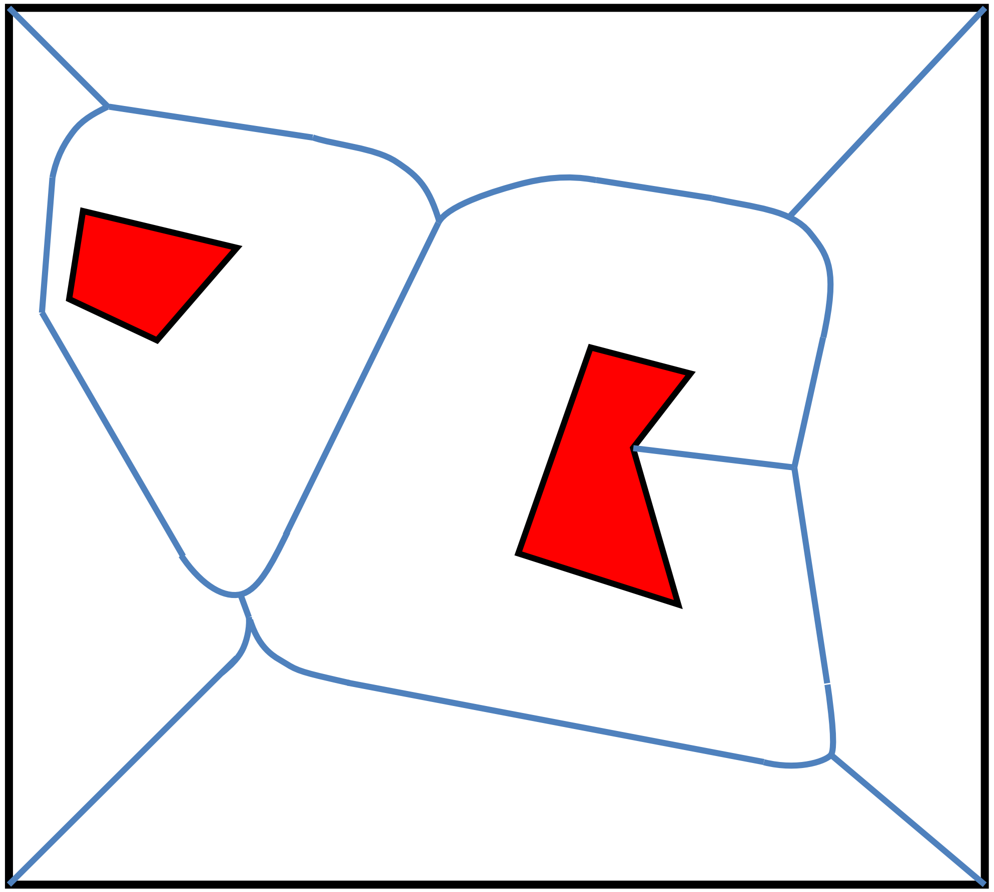
Can we do better?



Idea: Discretize only as much as necessary

This will depend on the number and geometric complexity of your obstacles

Can we do better?



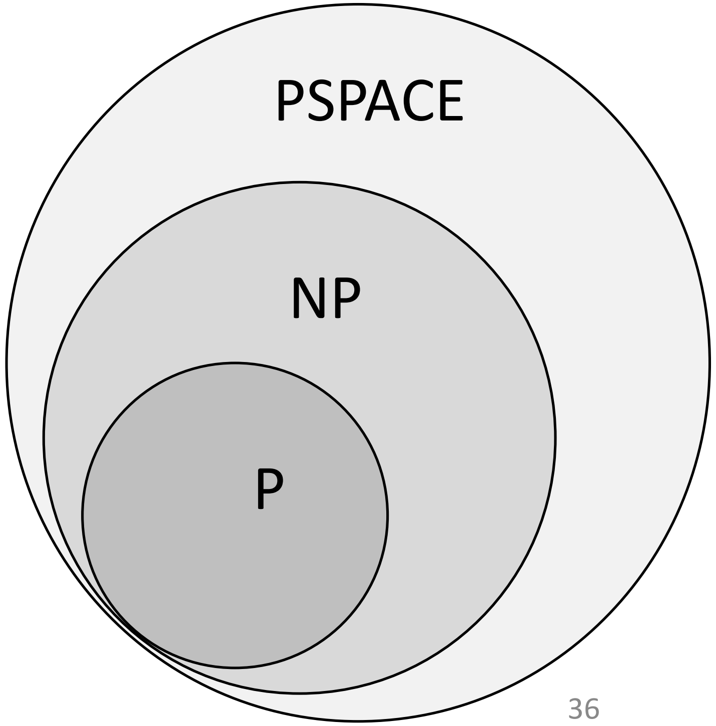
Idea: Map out the free space

This is called the Voronoi Diagram

Can we do better?

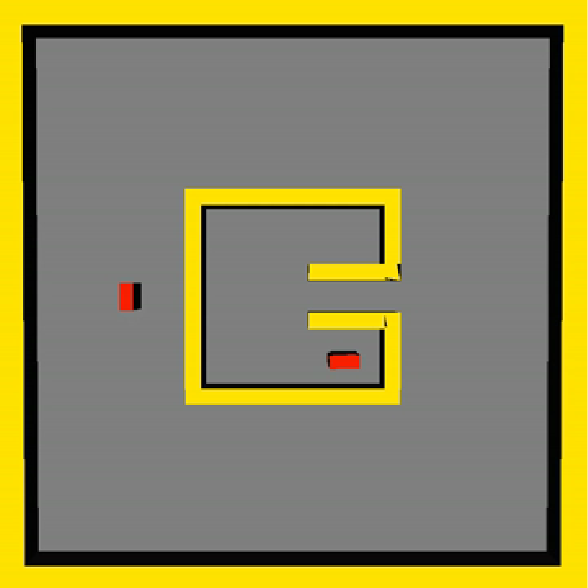
Theoretically, no.

General motion planning is in a class of problems we call PSPACE-complate. These are some of the hardest problems in computer science.



What makes planning hard?

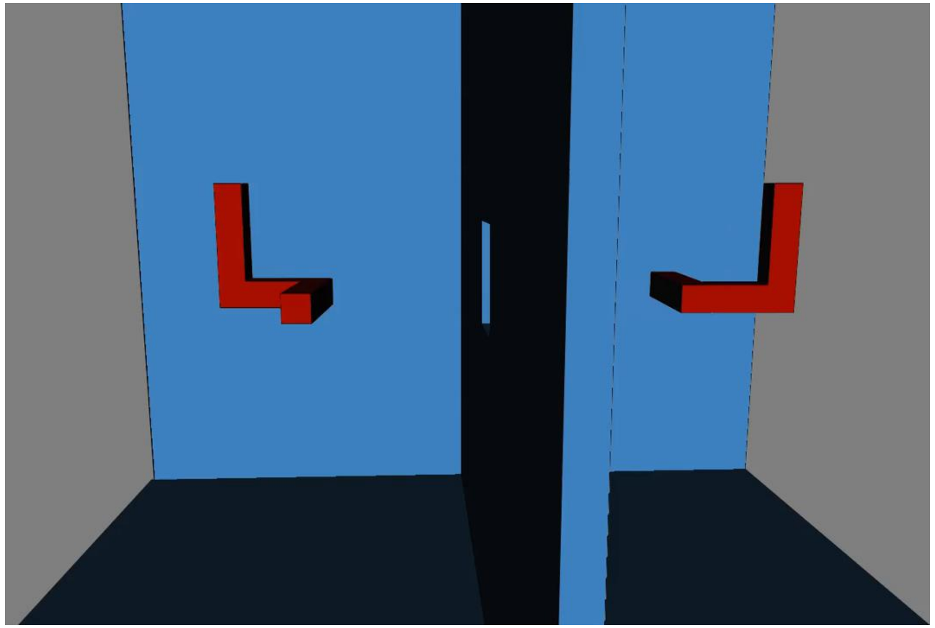
https://www.youtube.com/watch?v=UTbiAu8IXas



https://vimeo.com/58686591

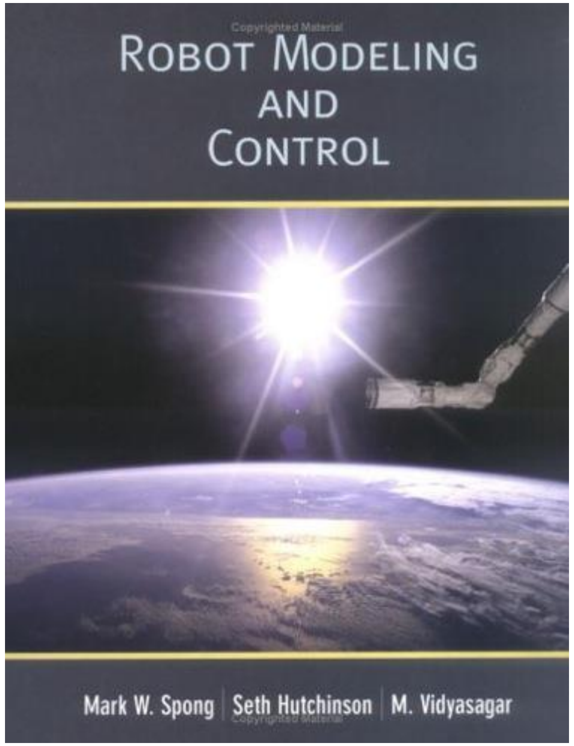
Complex obstacles
Narrow corridors in the free C-space

CHALLENGE: Map out the free C-Space



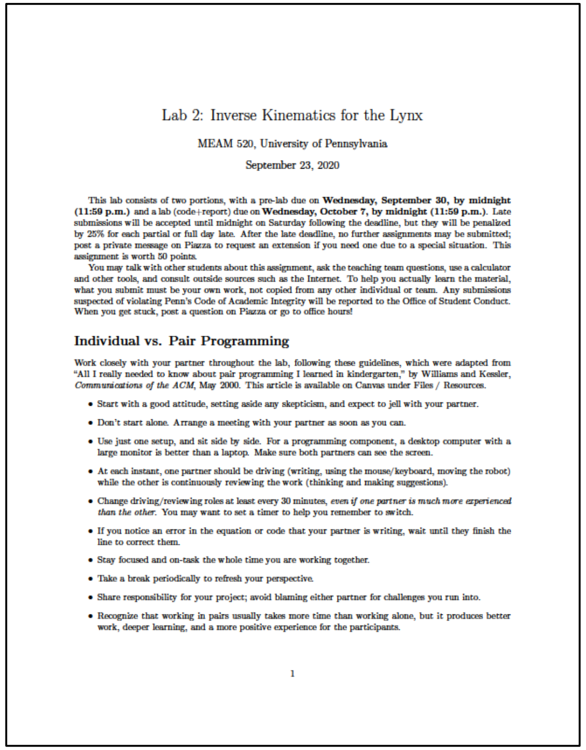
https://vimeo.com/58709589

Next time: Probabilistic Trajectory Planning



Chapter 5: Path and Trajectory Planning

- Read 5.4



Lab 2: Inverse Kinematics for the Lynx

MEAM 520, University of Pennsylvania

September 23, 2020

This lab consists of two portions, with a pre-lab due on **Wednesday, September 30, by midnight (11:59 p.m.)** and a lab (code+report) due on **Wednesday, October 7, by midnight (11:59 p.m.)**. Late submissions will be accepted until midnight on Saturday following the deadline, but they will be penalized by 25% for each partial or full day late. After the late deadline, no further assignments may be submitted; post a private message on Piazza to request an extension if you need one due to a special situation. This assignment is worth 50 points.

You may talk with other students about this assignment, ask the teaching team questions, use a calculator and other tools, and consult outside sources such as the Internet. To help you actually learn the material, what you submit must be your own work, not copied from any other individual or team. Any submissions suspected of violating Penn's Code of Academic Integrity will be reported to the Office of Student Conduct. When you get stuck, post a question on Piazza or go to office hours!

Individual vs. Pair Programming

Work closely with your partner throughout the lab, following these guidelines, which were adapted from "All I really needed to know about pair programming I learned in kindergarten," by Williams and Kessler, *Communications of the ACM*, May 2000. This article is available on Canvas under Files / Resources.

- Start with a good attitude, setting aside any skepticism, and expect to jell with your partner.
- Don't start alone. Arrange a meeting with your partner as soon as you can.
- Use just one setup, and sit side by side. For a programming component, a desktop computer with a large monitor is better than a laptop. Make sure both partners can see the screen.
- At each instant, one partner should be driving (writing, using the mouse/keyboard, moving the robot) while the other is continuously reviewing the work (thinking and making suggestions).
- Change driving/reviewing roles at least every 30 minutes, *even if one partner is much more experienced than the other*. You may want to set a timer to help you remember to switch.
- If you notice an error in the equation or code that your partner is writing, wait until they finish the line to correct them.
- Stay focused and on-task the whole time you are working together.
- Take a break periodically to refresh your perspective.
- Share responsibility for your project; avoid blaming either partner for challenges you run into.
- Recognize that working in pairs usually takes more time than working alone, but it produces better work, deeper learning, and a more positive experience for the participants.

1

Lab 2: Inverse Kinematics due 10/7