

Incremental Path Planning

April 4, 2016

Joe Leavitt, Ben Ayton, Jessica Noss, Erlend Harbitz, Jake Barnwell & Sam Pinto

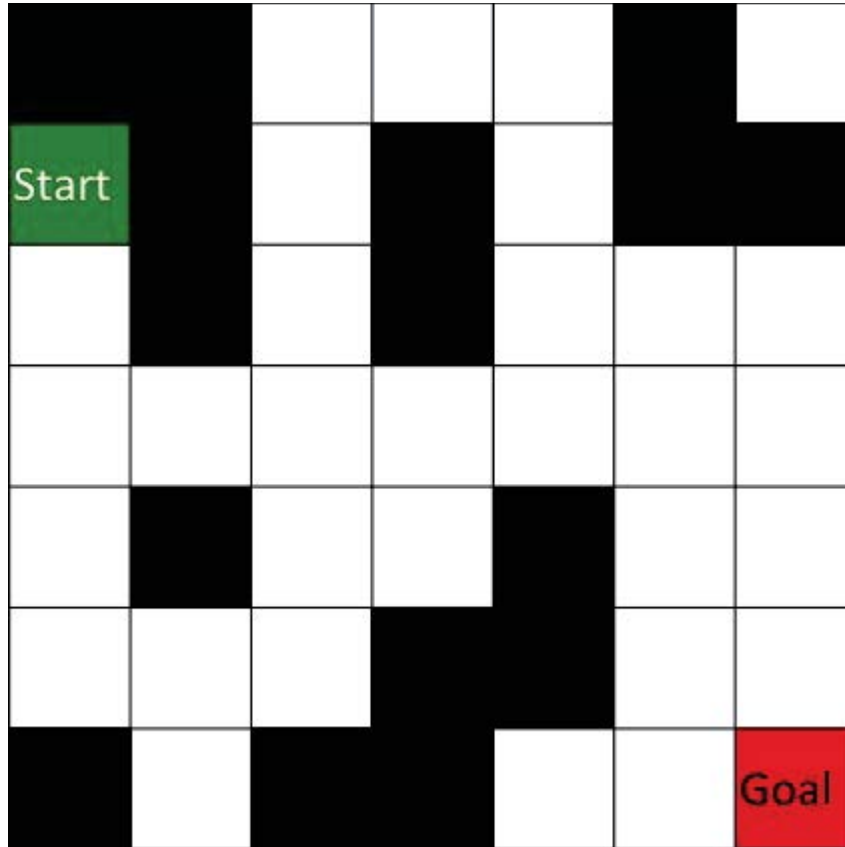
References

- Koenig, S., & Likhachev, M. (2002, July). D* Lite. In *AAAI/IAAI* (pp. 476-483).
- Koenig, S., Likhachev, M., & Furcy, D. (2004). Lifelong planning A*. *Artificial Intelligence*, 155(1), 93-146.
- Stentz, A. (1994, May). Optimal and efficient path planning for partially-known environments. In *Proceedings of the IEEE International Conference on Robotics and Automation*.
- Likhachev, M., Ferguson, D. I., Gordon, G. J., Stentz, A., & Thrun, S. (2005, June). Anytime Dynamic A*: An Anytime, Replanning Algorithm. In *ICAPS* (pp. 262-271).
- Hofmann, A., Fernandez, E., Helbert, J., Smith, S., & Williams, B. (2015). Reactive Integrated Motion Planning and Execution. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence*.

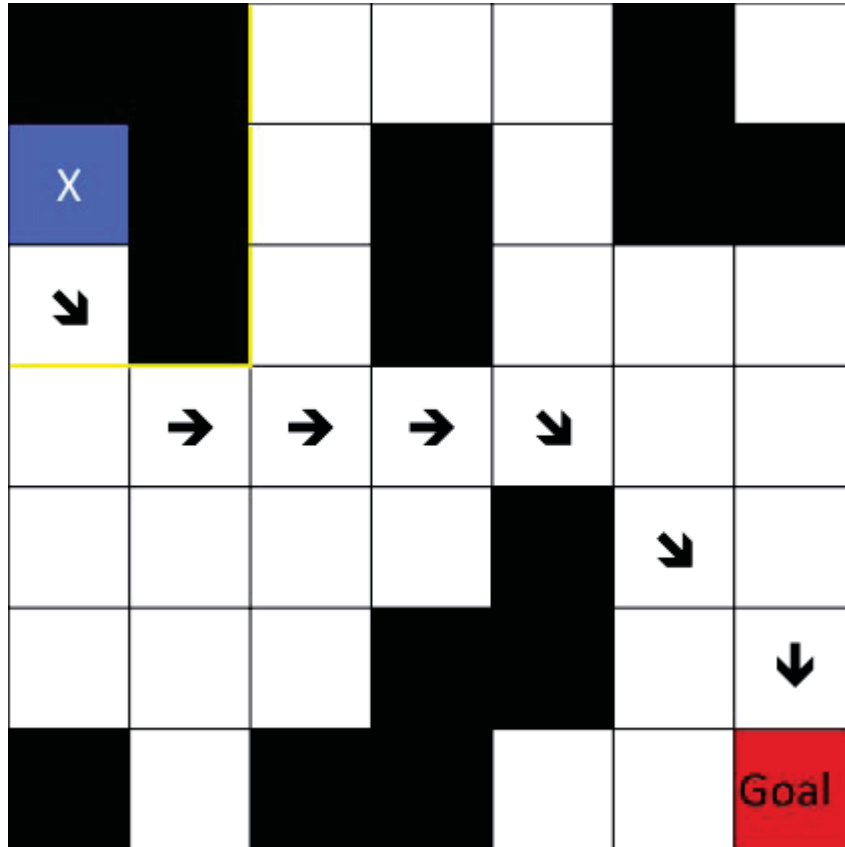
Outline

- Motivation
- Incremental Search
- The D* Lite Algorithm
- D* Lite Example
- When to Use Incremental Path Planning?
- Algorithm Extensions and Related Topics
- Application to Mobile Robotics

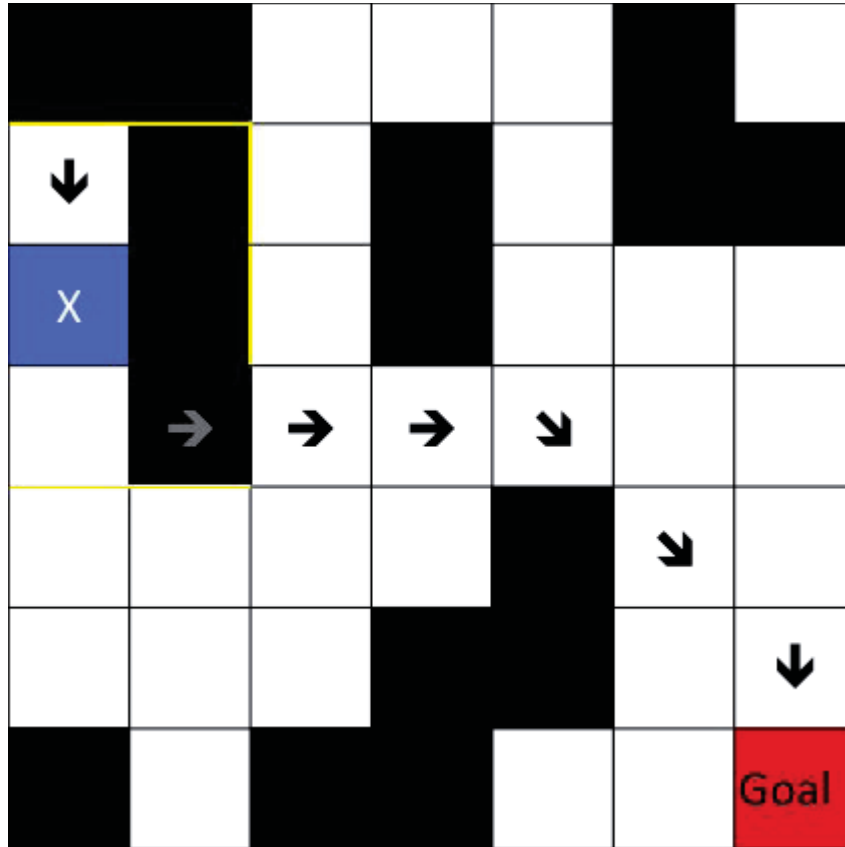
Motivation



Motivation



Motivation

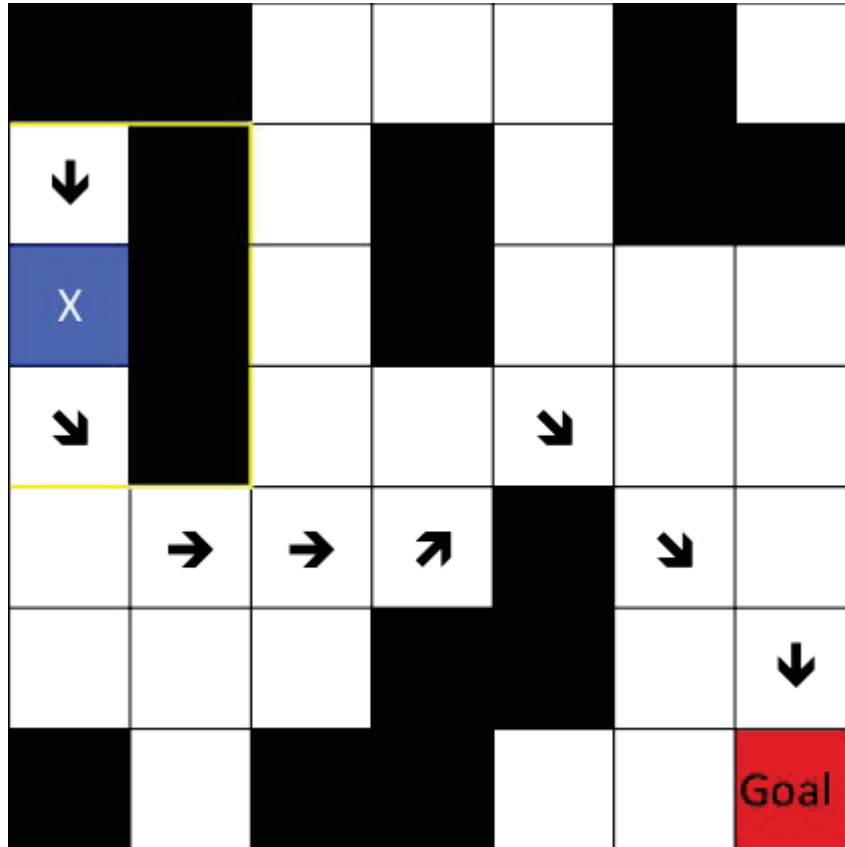


New obstacle detected!



Replan

Motivation

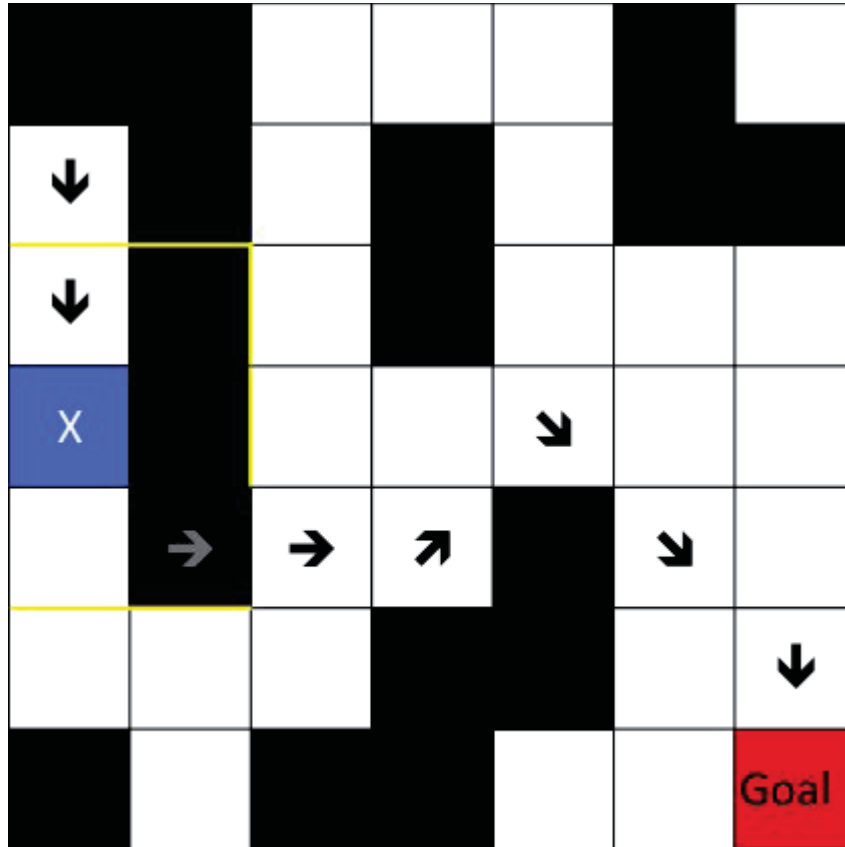


New obstacle detected!



Replan

Motivation

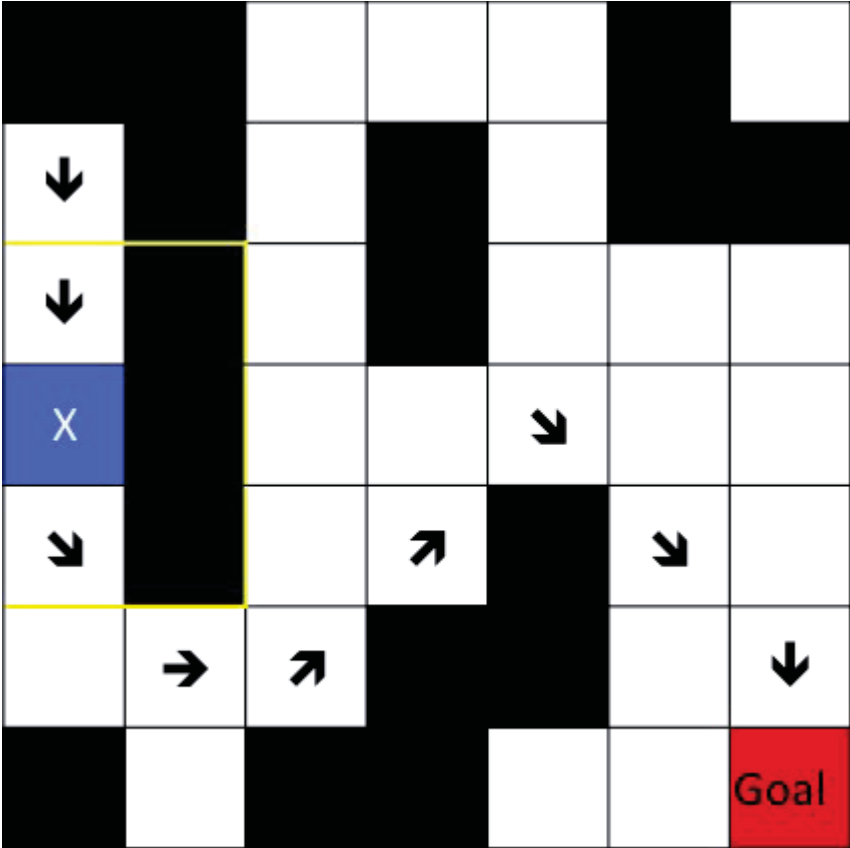


New obstacle detected!



Replan

Motivation

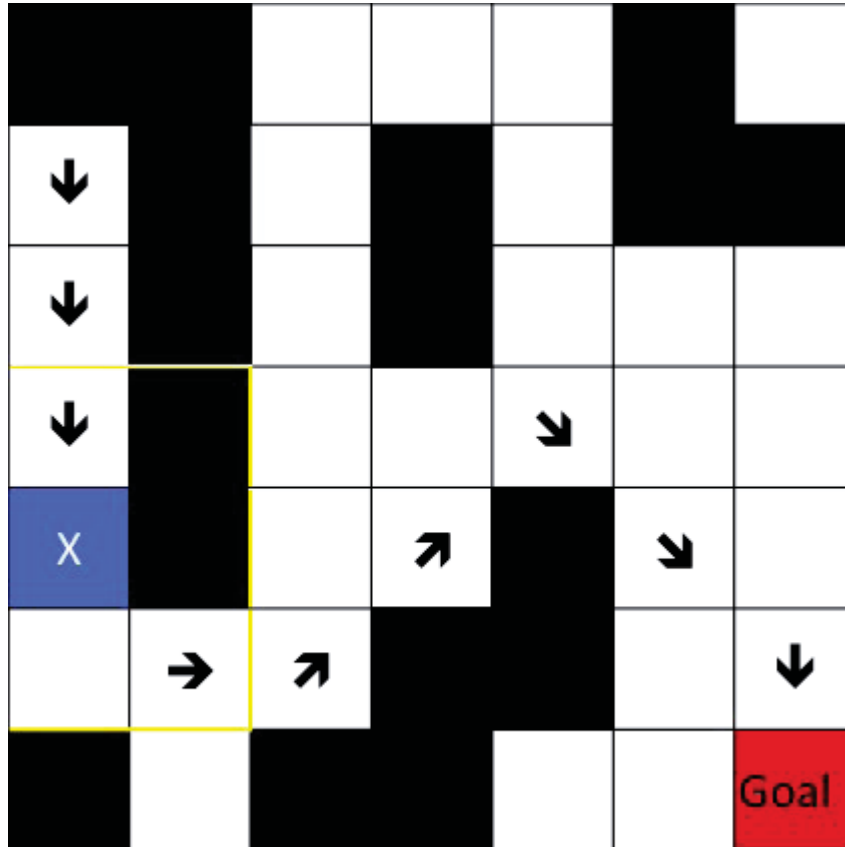


New obstacle detected!

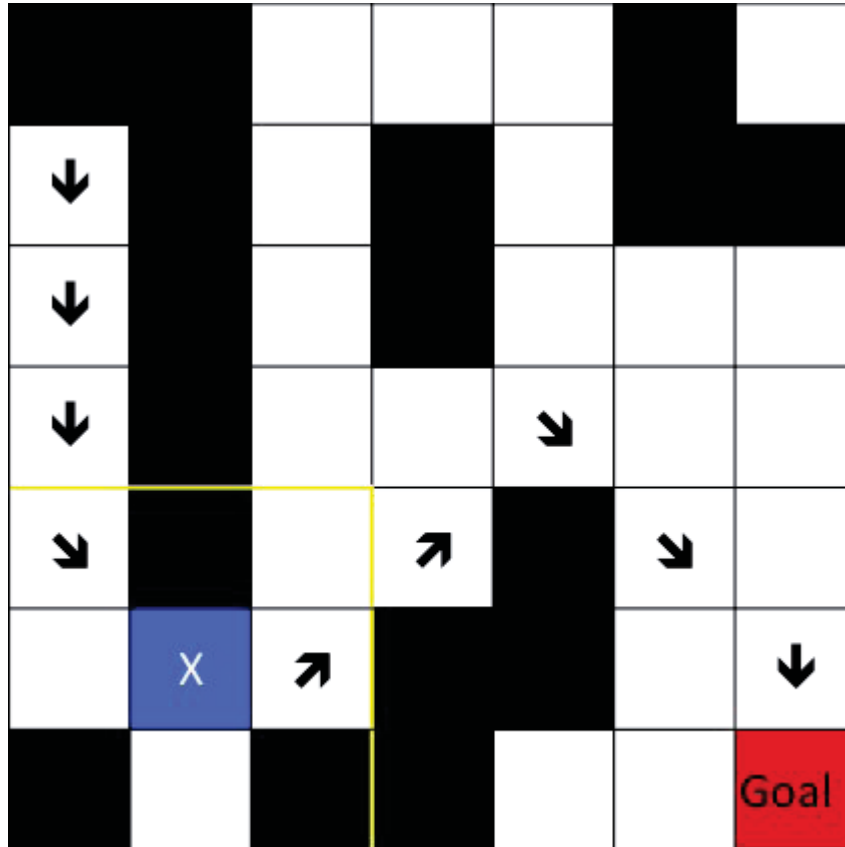


Replan

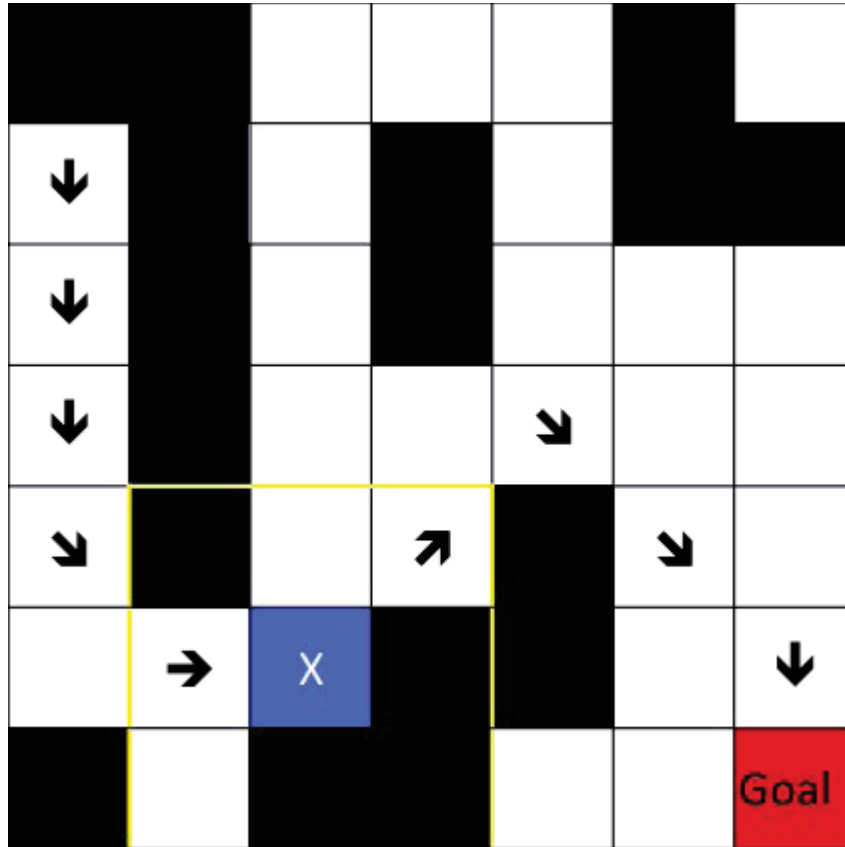
Motivation



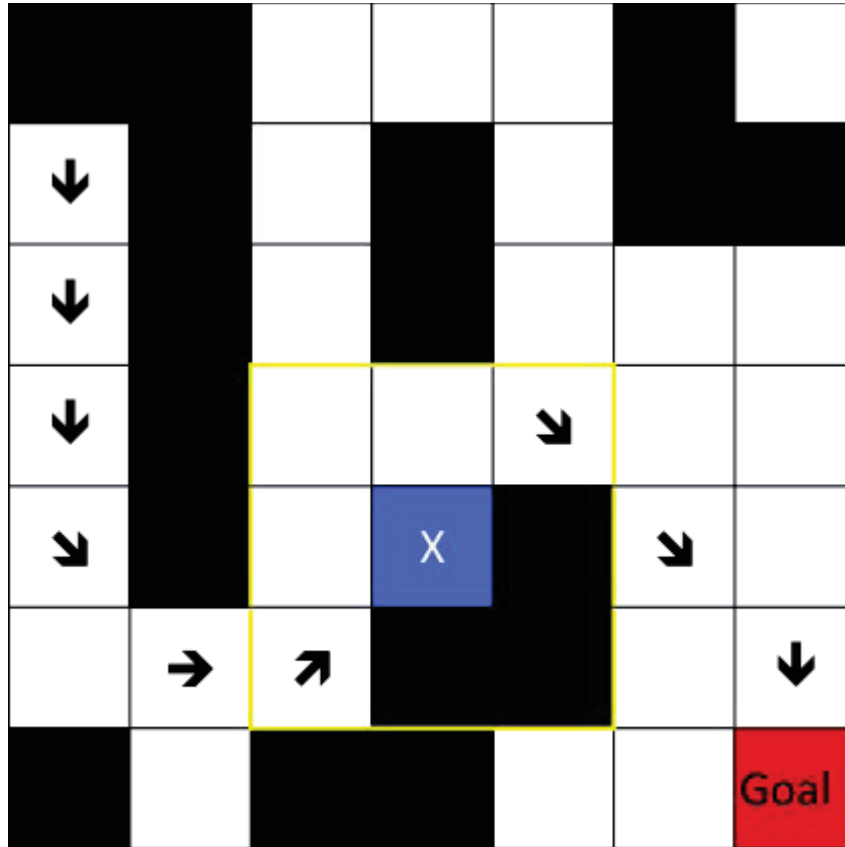
Motivation



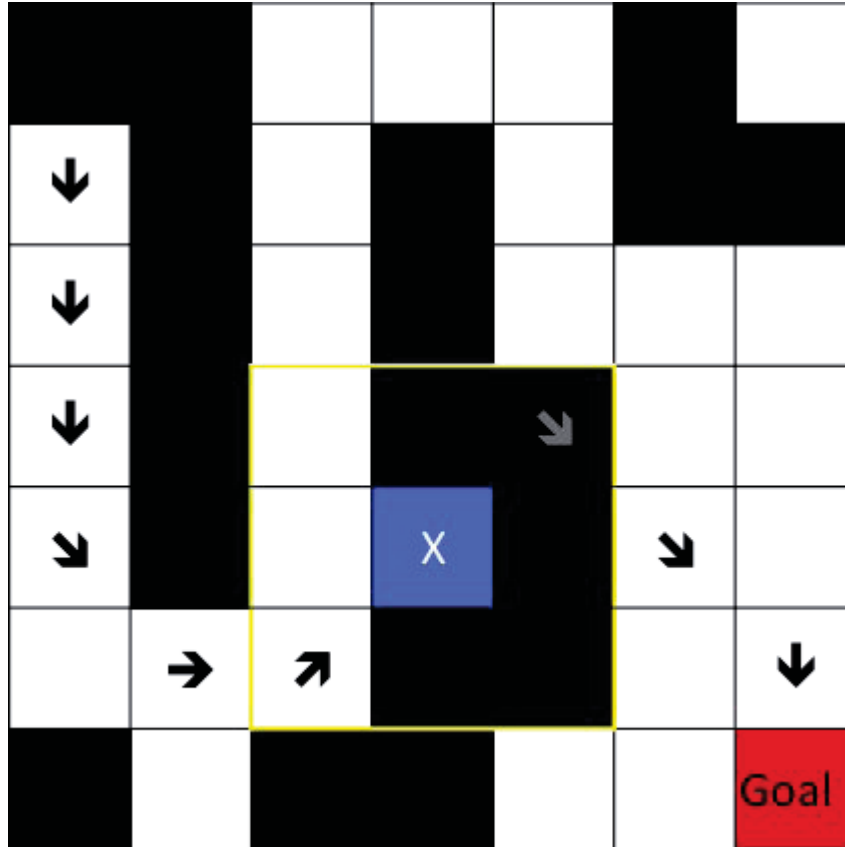
Motivation



Motivation



Motivation



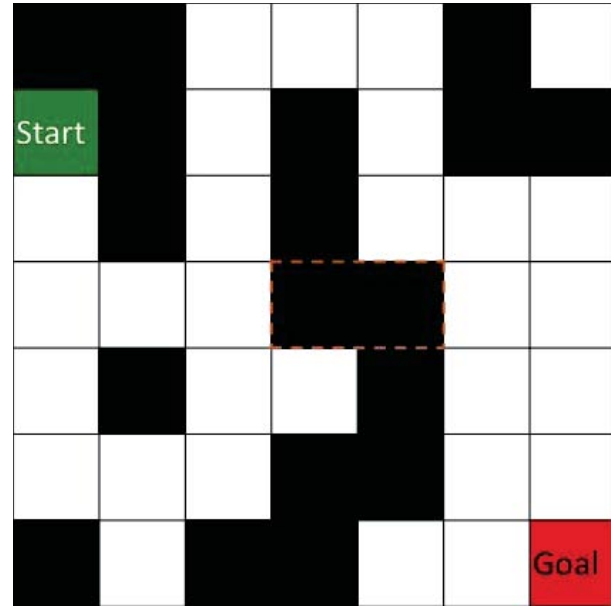
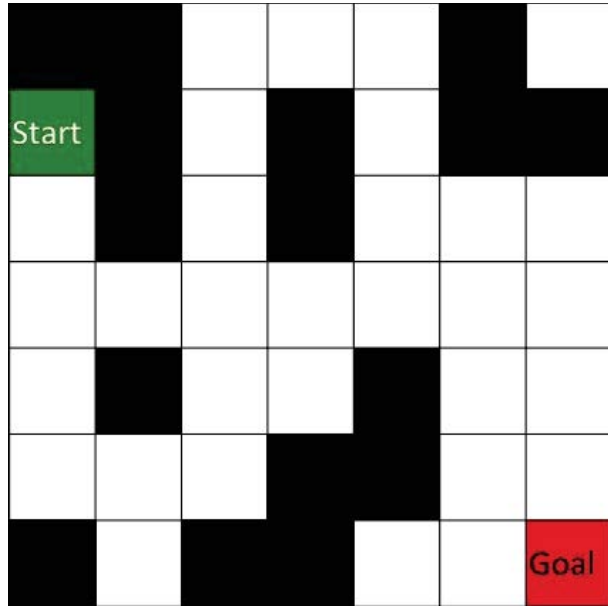
Change detected!



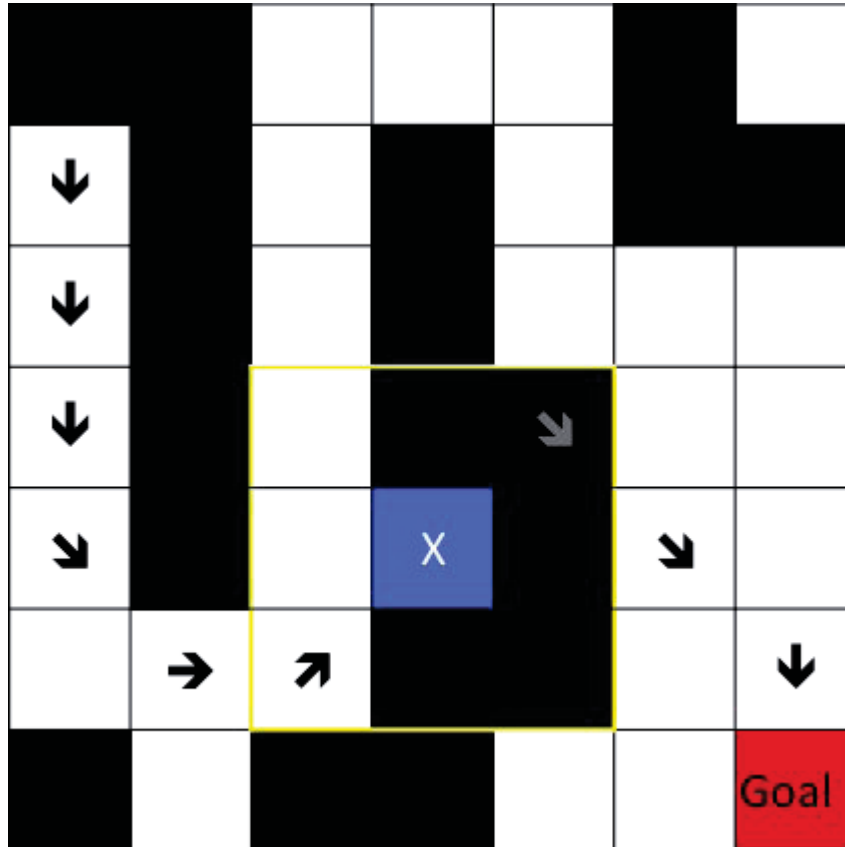
Replan

Motivation

Environmental Change



Motivation

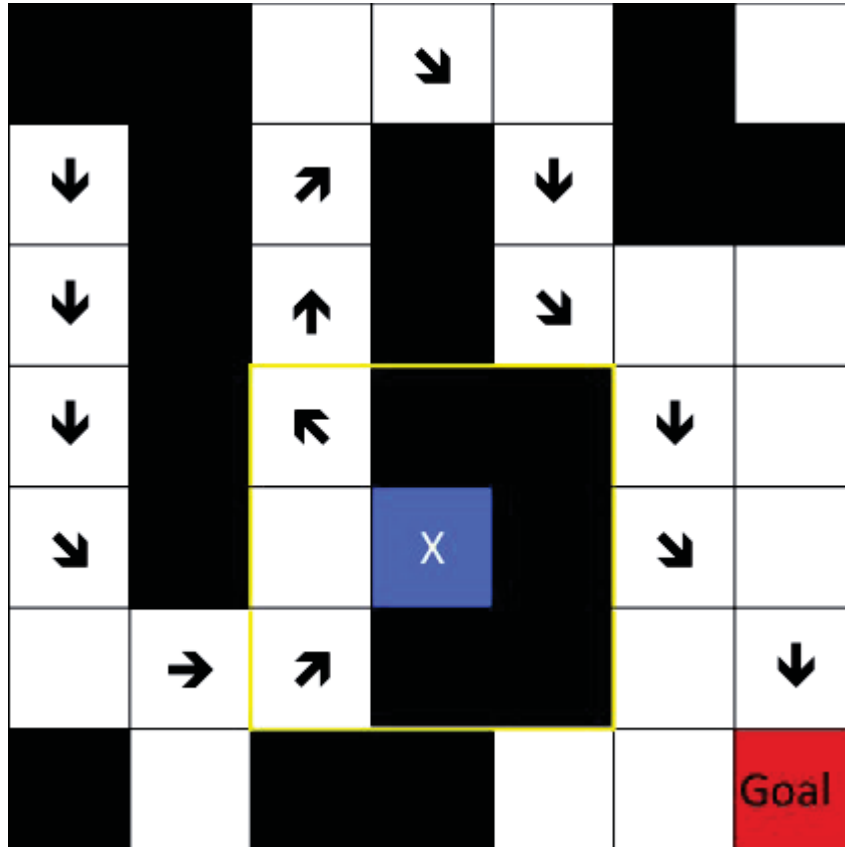


Change detected!



Replan

Motivation

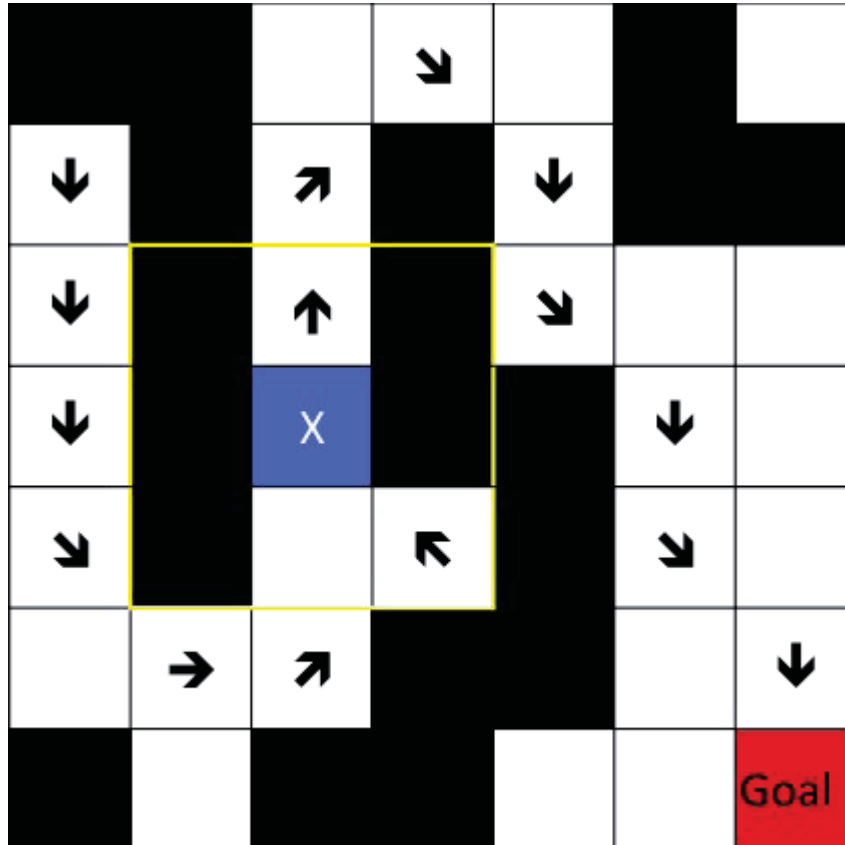


Change detected!

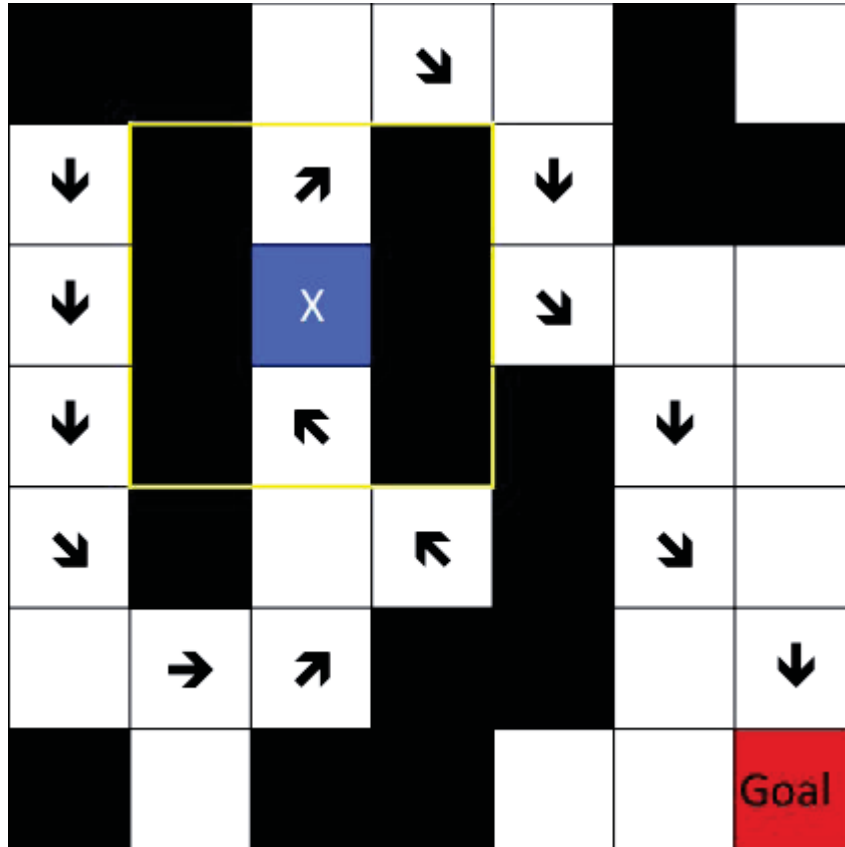


Replan

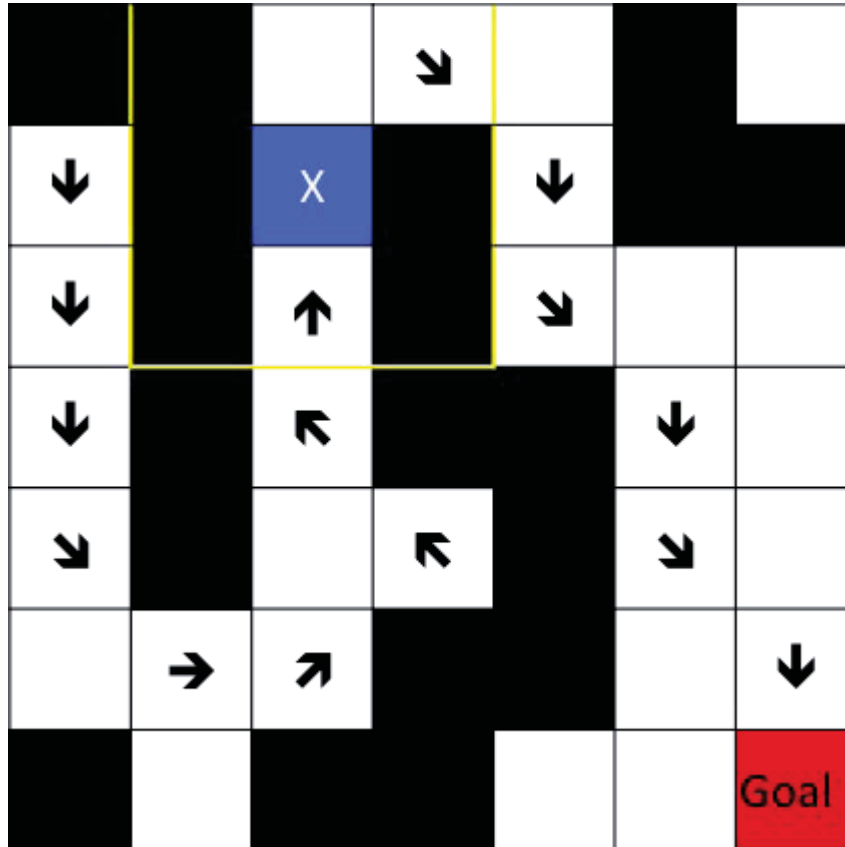
Motivation



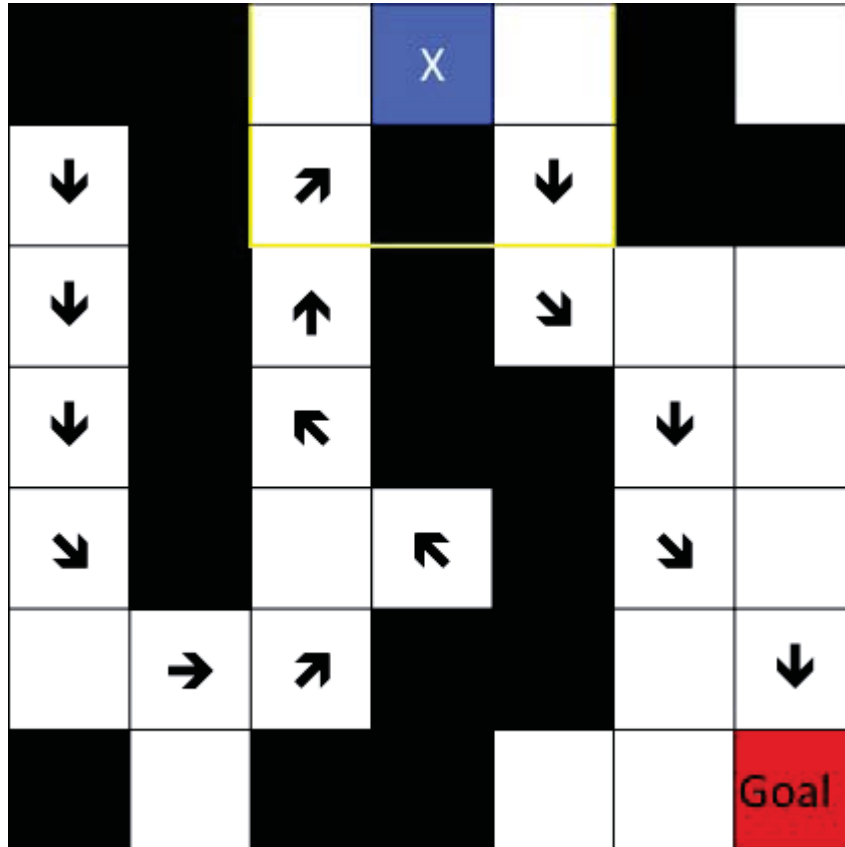
Motivation



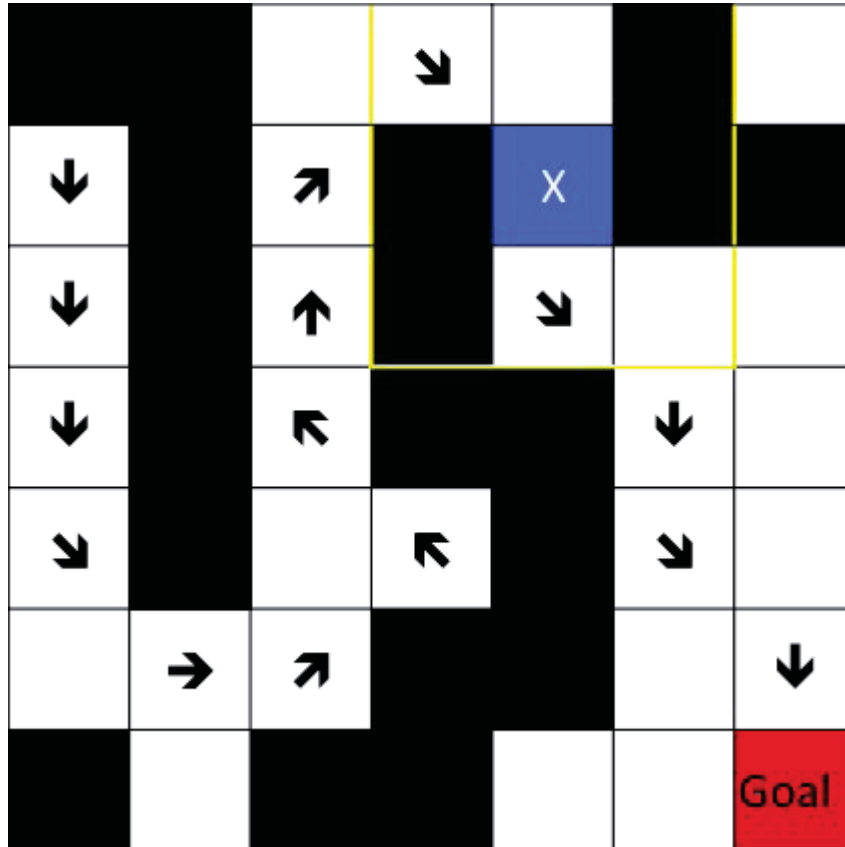
Motivation



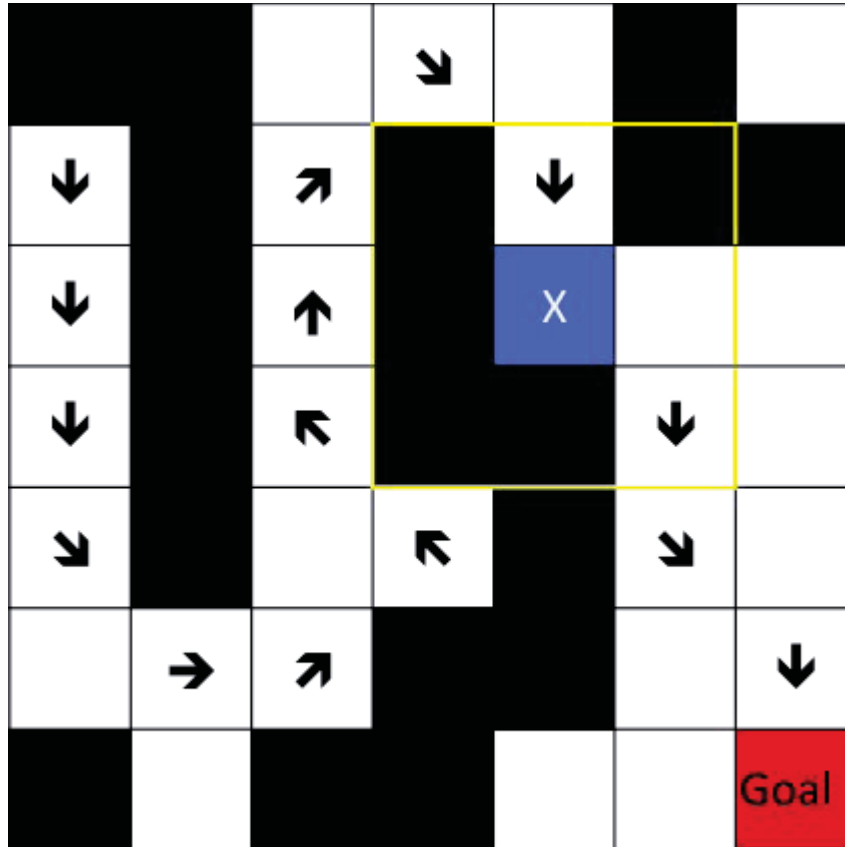
Motivation



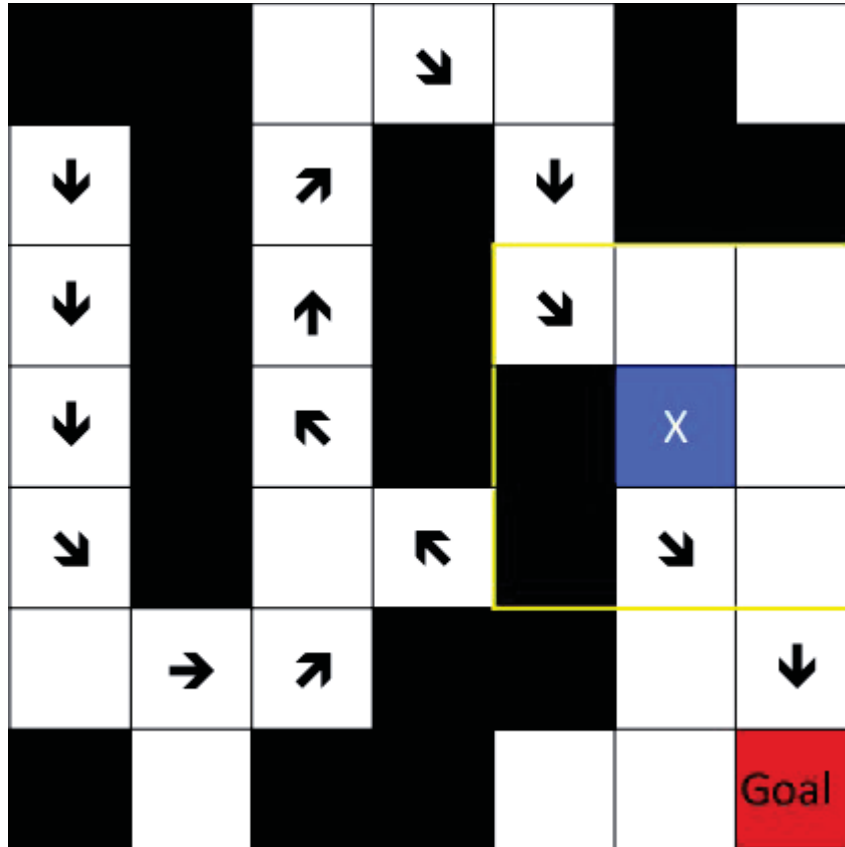
Motivation



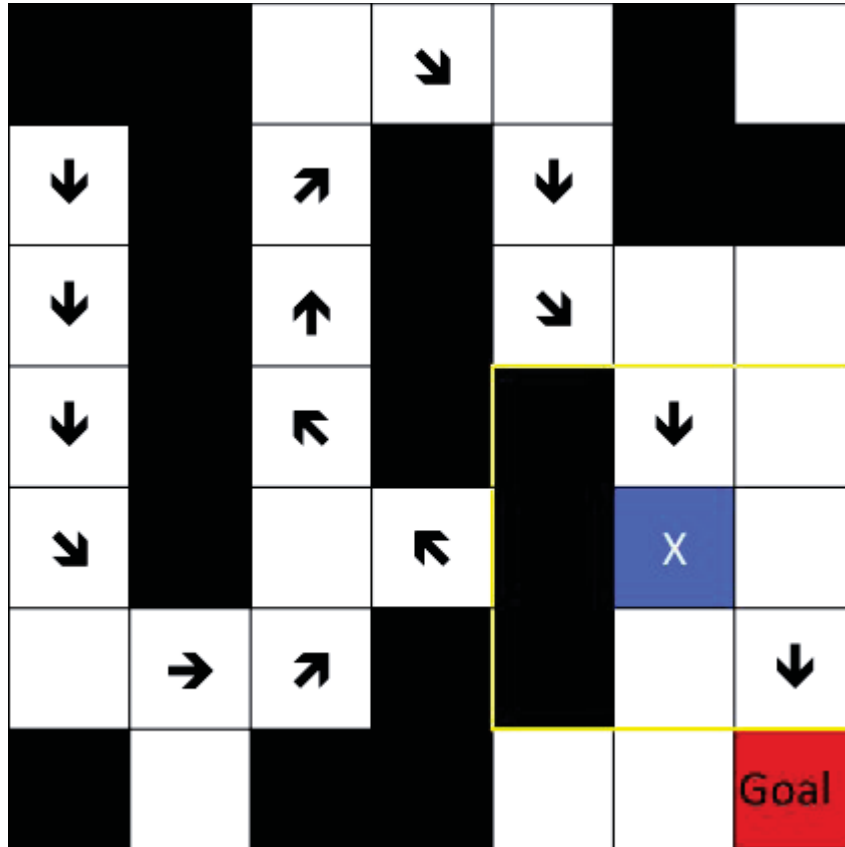
Motivation



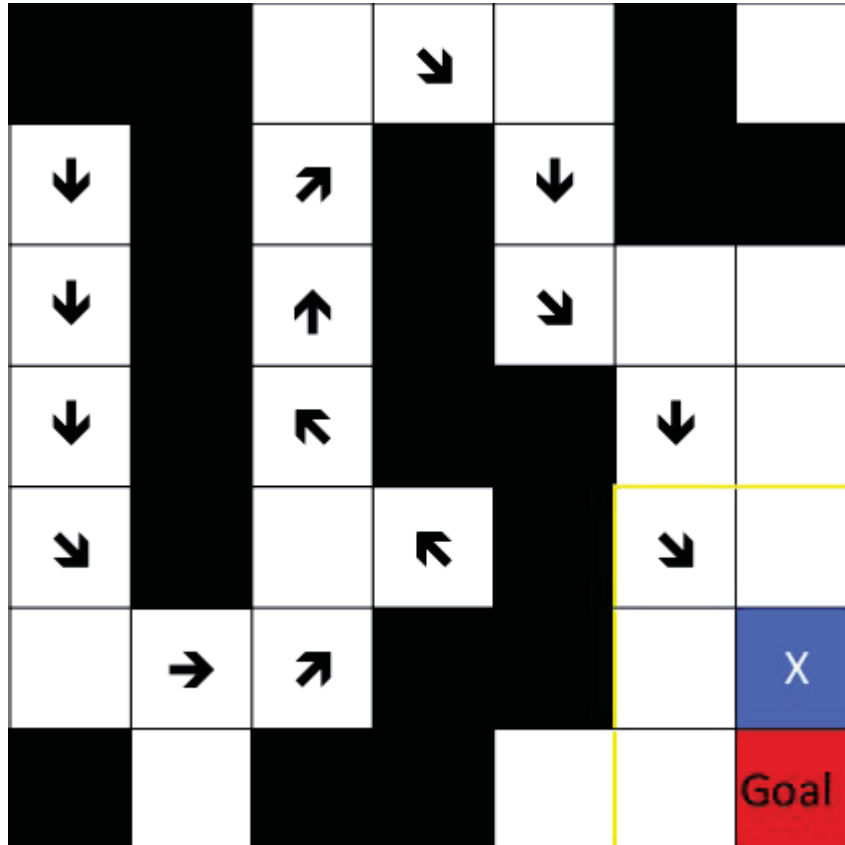
Motivation



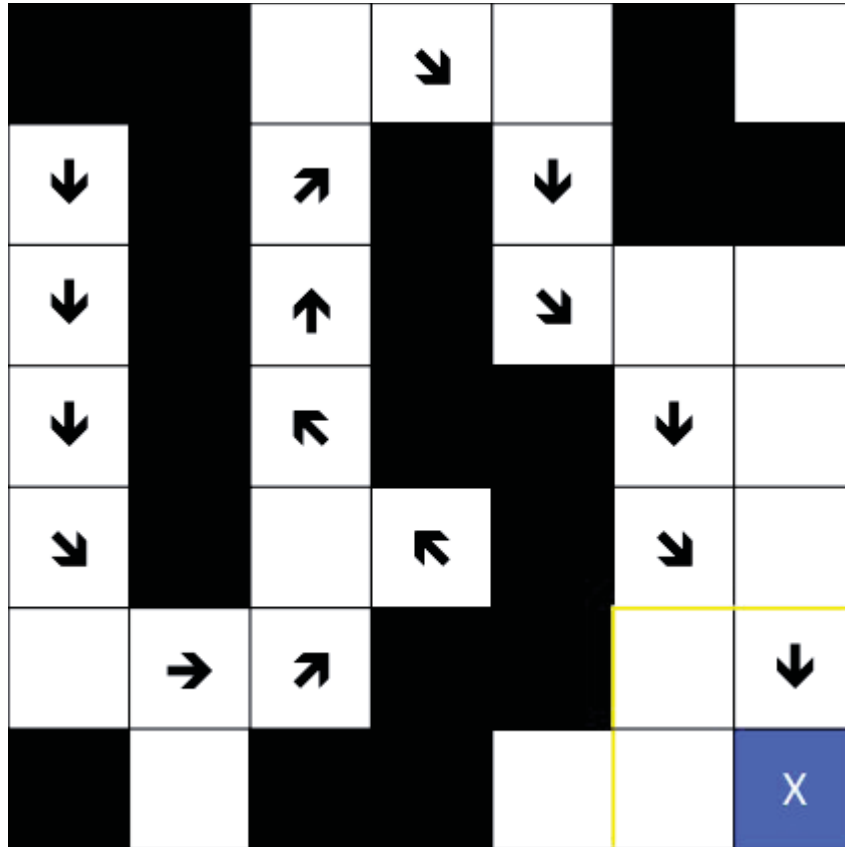
Motivation



Motivation



Motivation



Motivation

RRT



© Sertac Karaman. All rights reserved. This content is excluded from our Creative Commons license. For more information, see <https://ocw.mit.edu/help/faq-fair-use/>.

Watch the video on YouTube (<https://www.youtube.com/watch?v=vW74bC-Ygb4>)

Motivation

After significant offline computation time...

Motivation

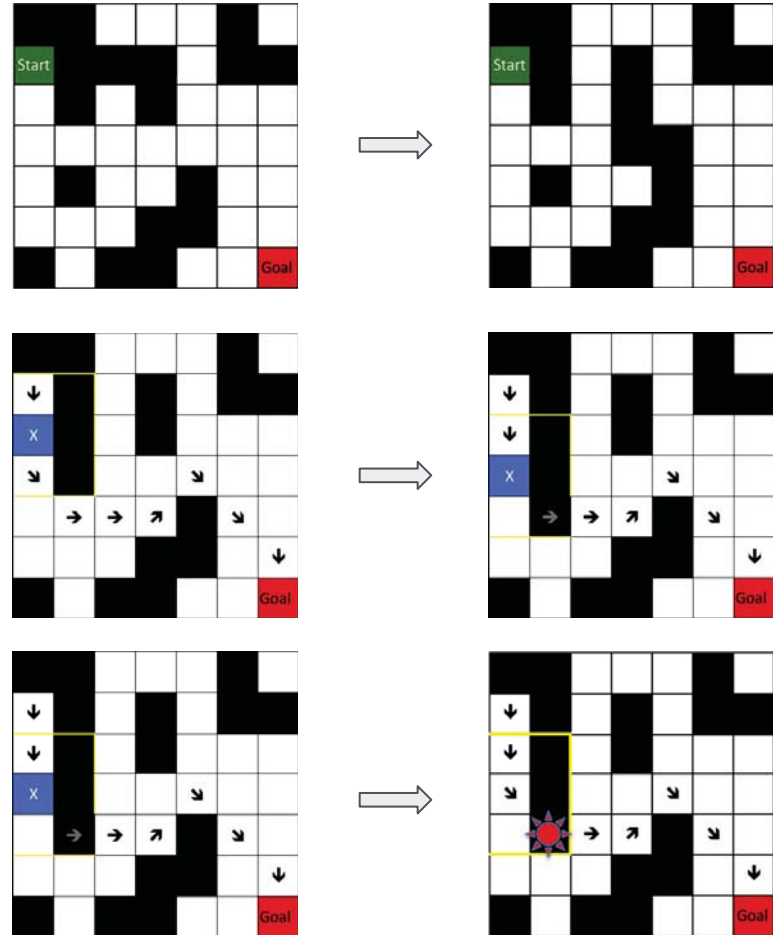
RRT*



© Sertac Karaman. All rights reserved. This content is excluded from our Creative Commons license. For more information, see <https://ocw.mit.edu/help/faq-fair-use/>.

Problems

- Changing environmental conditions:
 - Obstacles
 - Utility or cost
- Sensor limitations:
 - Partial observability
- Computation time:
 - Complete, optimal replanning is slow
 - Stay put or move in wrong direction?

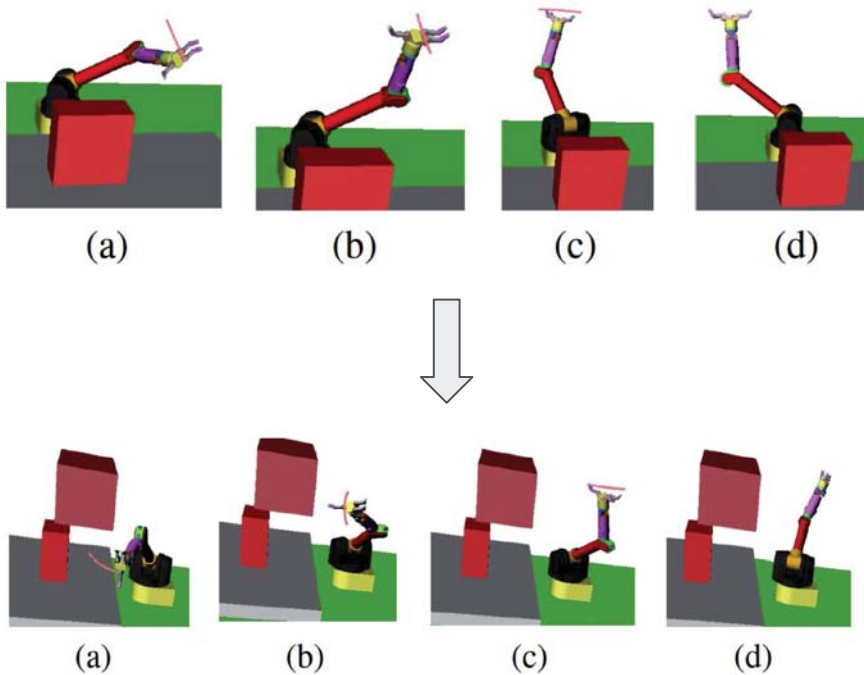


Reuse data from previous search!



Incremental Search Methods

Incremental APSP



Courtesy of Hofmann, Andreas et al. License: cc by-nc-sa.

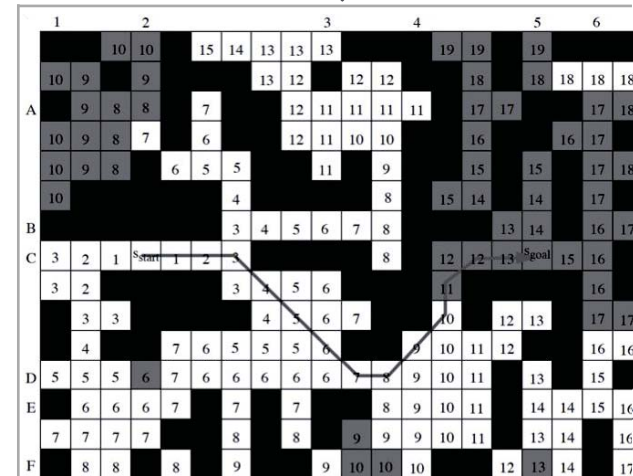
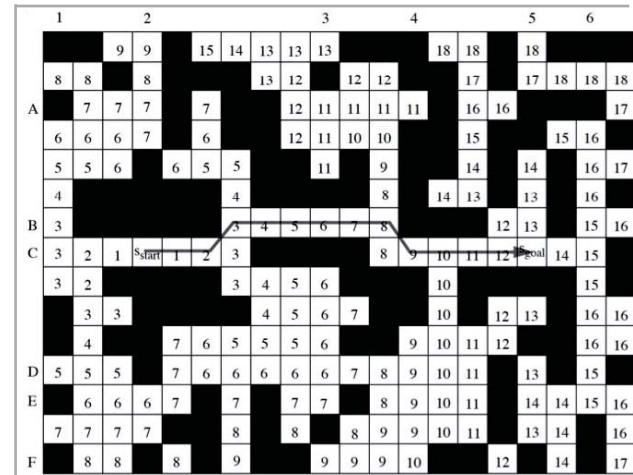
Alg.	Mean	Std	Comp.	Opt.
Chekhov (APSP)	0.0425	0.0213	Yes	Yes
RRT	0.188	0.12	Prob.	No
RRT connect	0.042	0.04	Prob.	No
PRM	0.12	0.08	Prob.	Asym.

Outline

- Motivation
- **Incremental Search**
- The D* Lite Algorithm
- D* Lite Example
- When to Use Incremental Path Planning?
- Algorithm Extensions and Related Topics
- Application to Mobile Robotics

Incremental Search

- Perform graph search
- Repeat:
 - Execute path/plan
 - Receive graph changes
 - Update previous search results



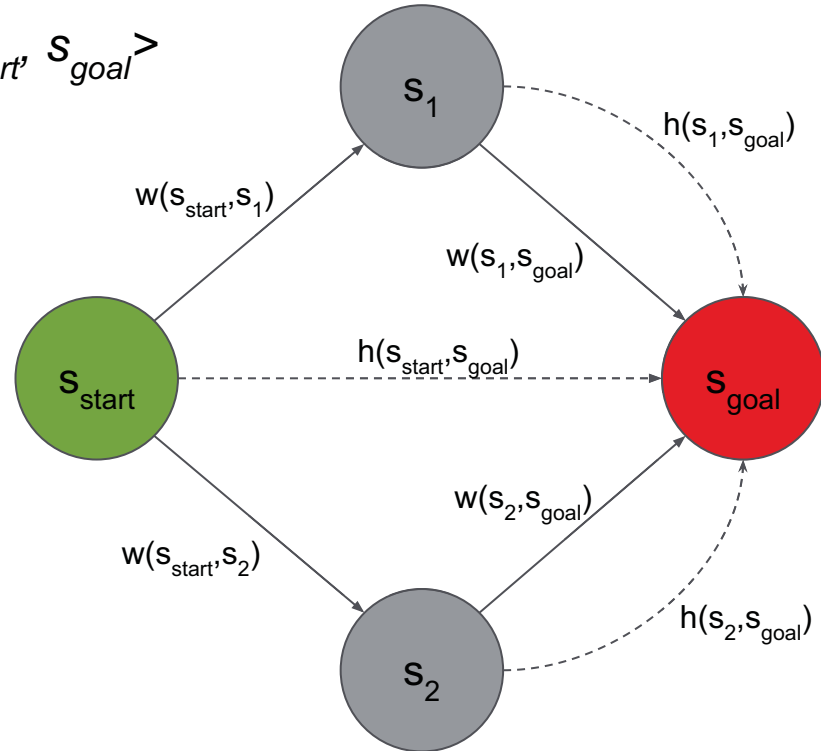
From Koenig, Likhachev, & Furcy (2004)

Review of Graph Search Problem

Input: graph search problem $S = \langle gr, w, h, s_{start}, s_{goal} \rangle$

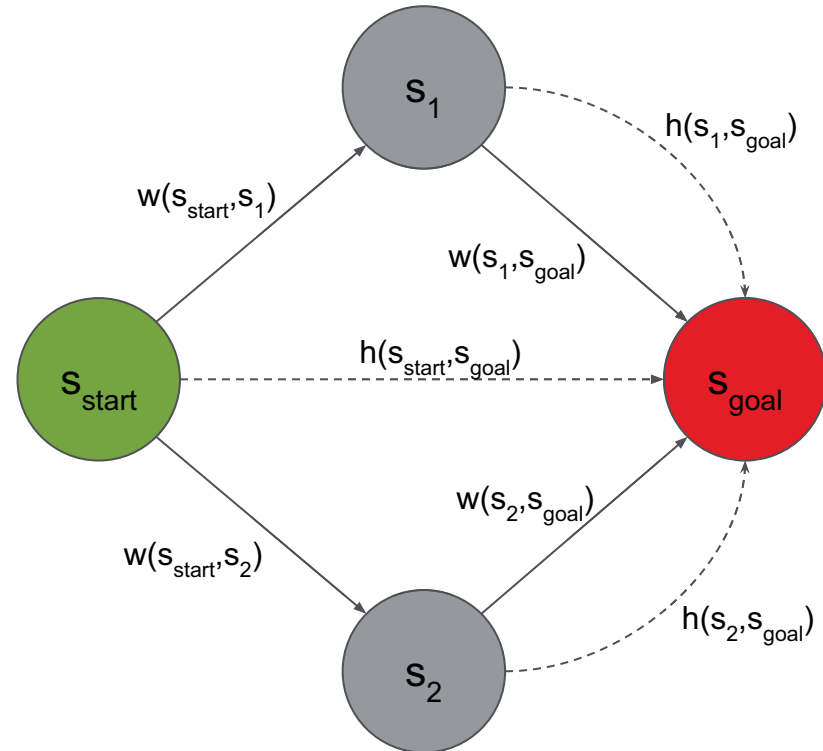
- directed graph: $gr = \langle V, E \rangle$
- edge weighting function: $w: s \times s \rightarrow \mathbb{R}$
- heuristic function: $h: s \times s_{goal} \rightarrow \mathbb{R}$
- start vertex: $s_{start} \in V$
- goal vertex: $s_{goal} \in V$

Output: simple path $P = \langle s_{start}, s_2, \dots, s_{goal} \rangle$



Review of Graph Search Problem

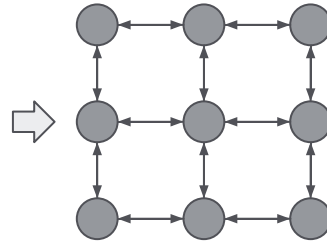
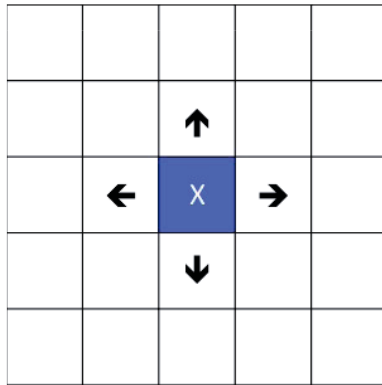
- g: cost-so-far
 - $g(s) = g(s_{\text{predecessor}}) + w(s_{\text{predecessor}}, s)$
- h: heuristic value, cost-to-go
- $f(s) = g(s) + h(s)$



Graph Representations

- Incremental algorithms generalize to any graph (usually non-negative edges).
- Grids used for ease of representation.

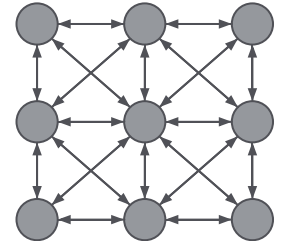
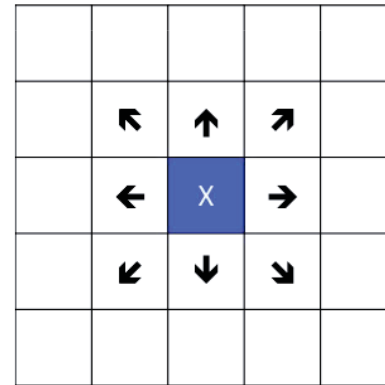
4-Connected Graph



Admissible
Heuristic:

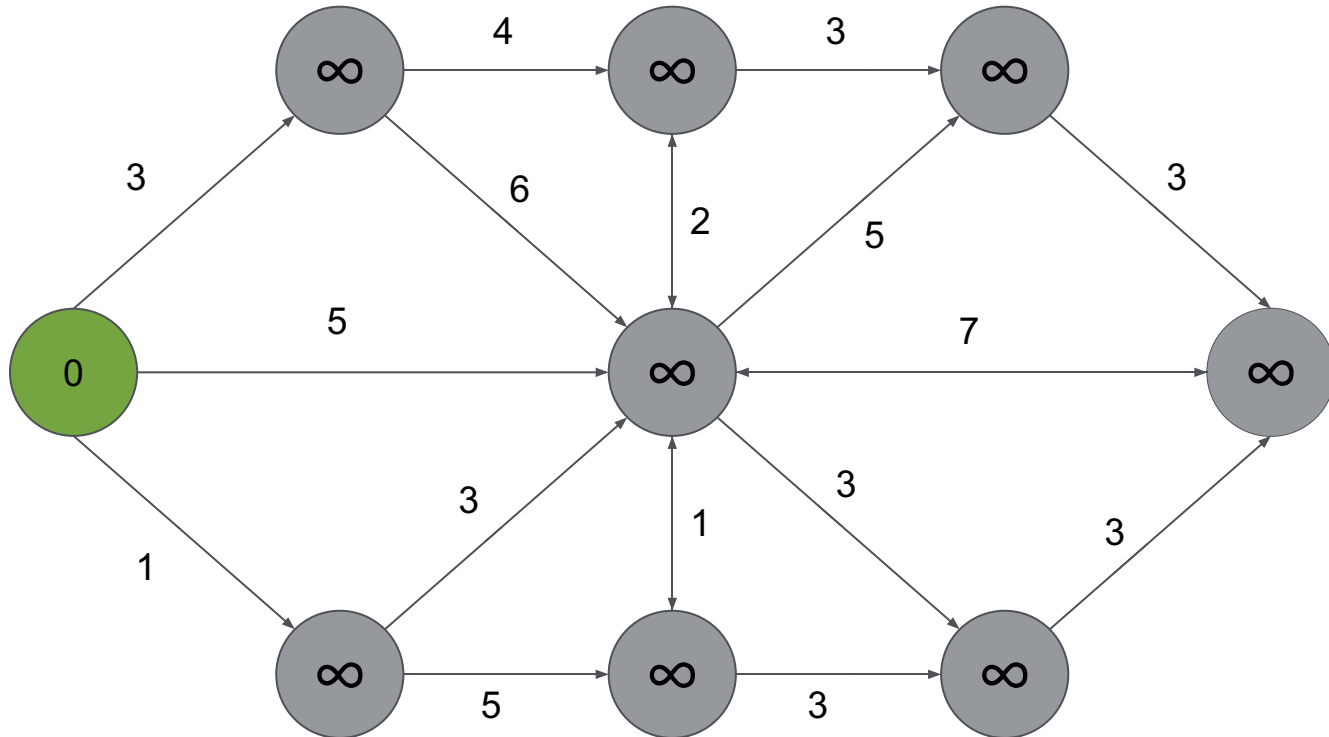
Manhattan Distance
 $= \Delta x + \Delta y$

8-Connected Graph

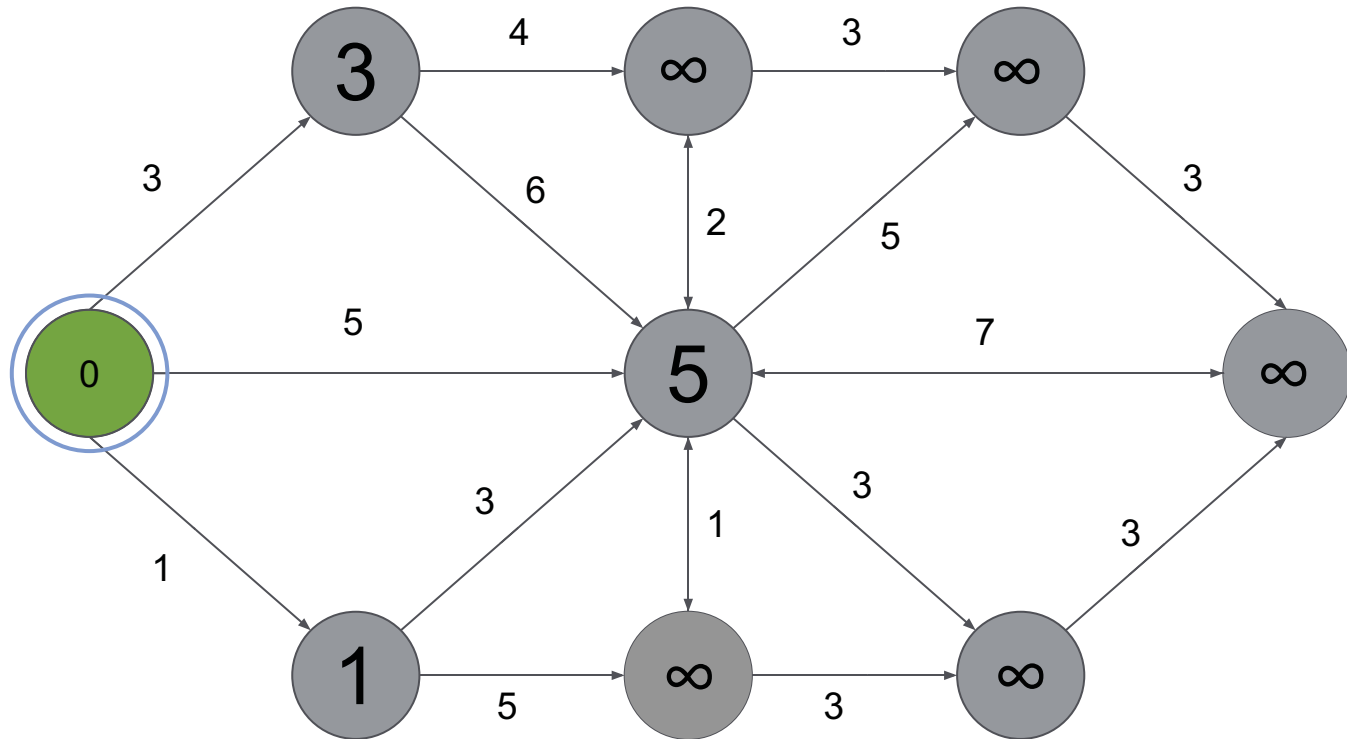


$\max(\Delta x, \Delta y)$
(with unit edge weights)

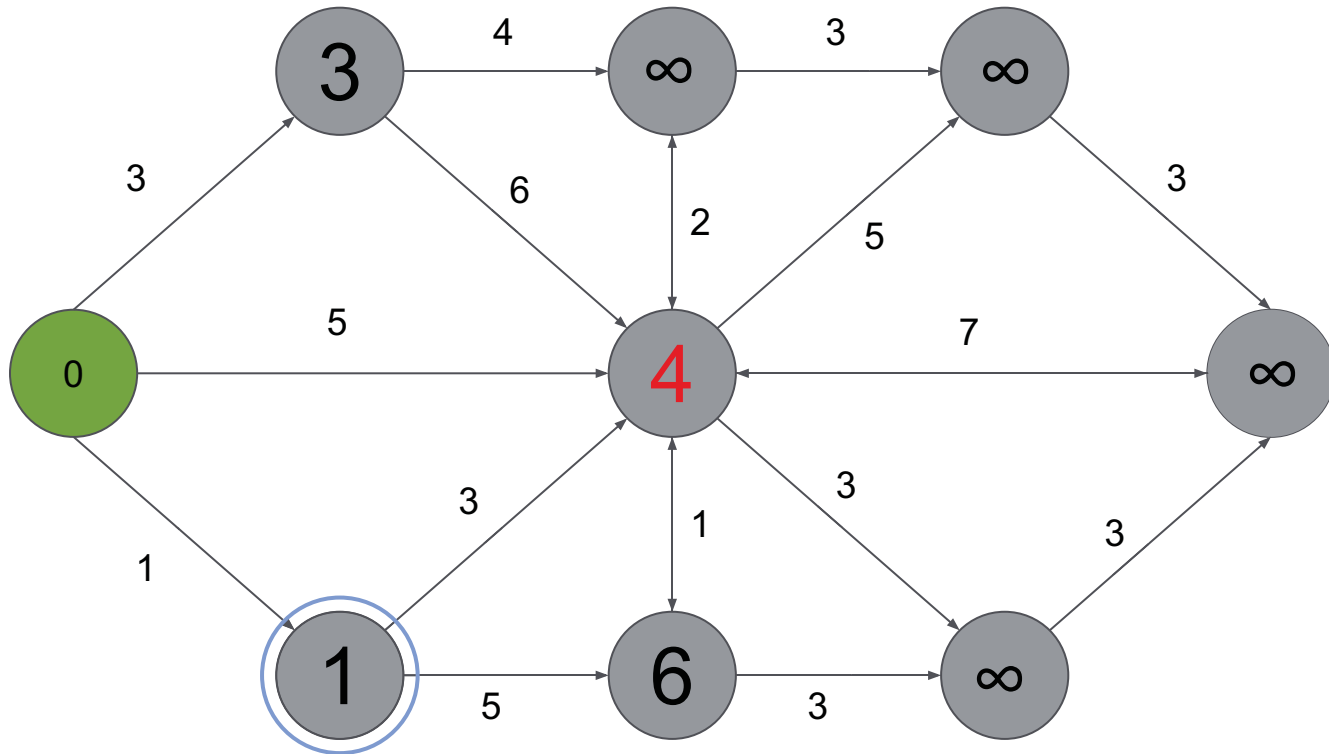
Relaxation - Dijkstra's Algorithm



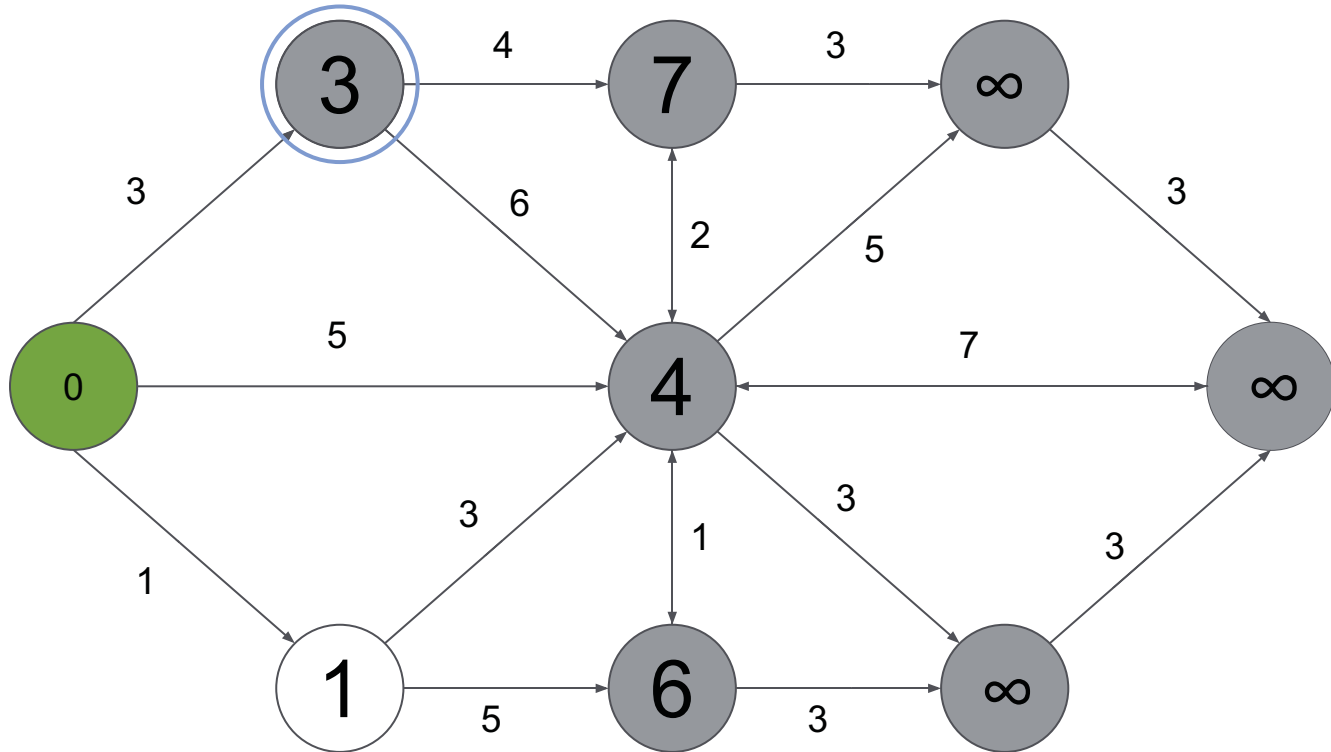
Relaxation - Dijkstra's Algorithm



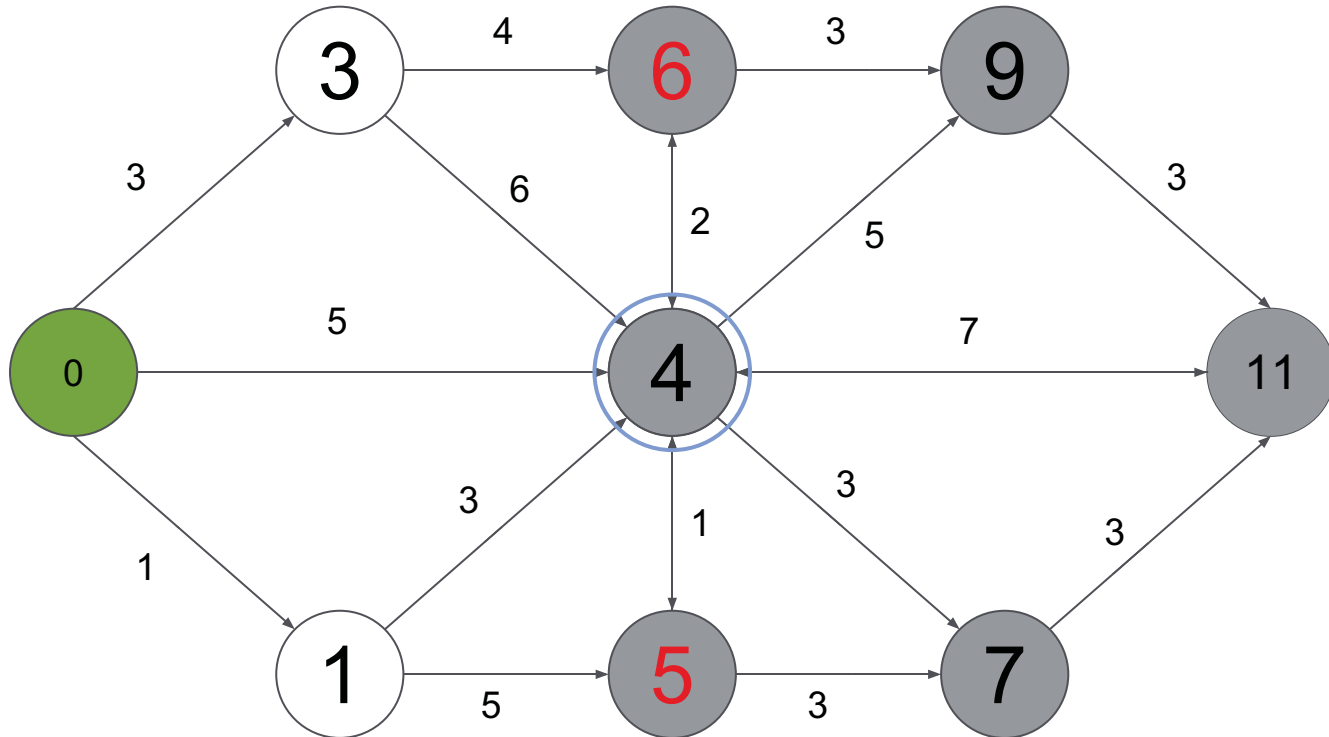
Relaxation - Dijkstra's Algorithm



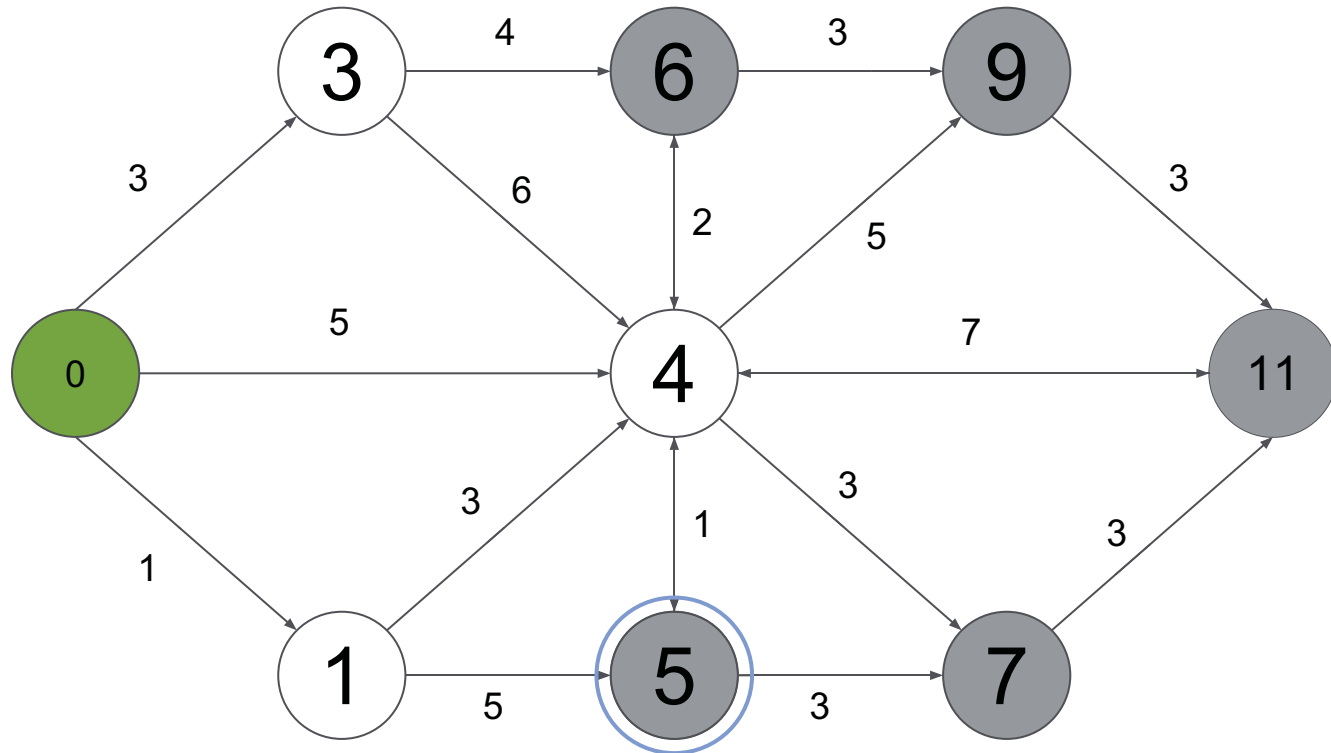
Relaxation - Dijkstra's Algorithm



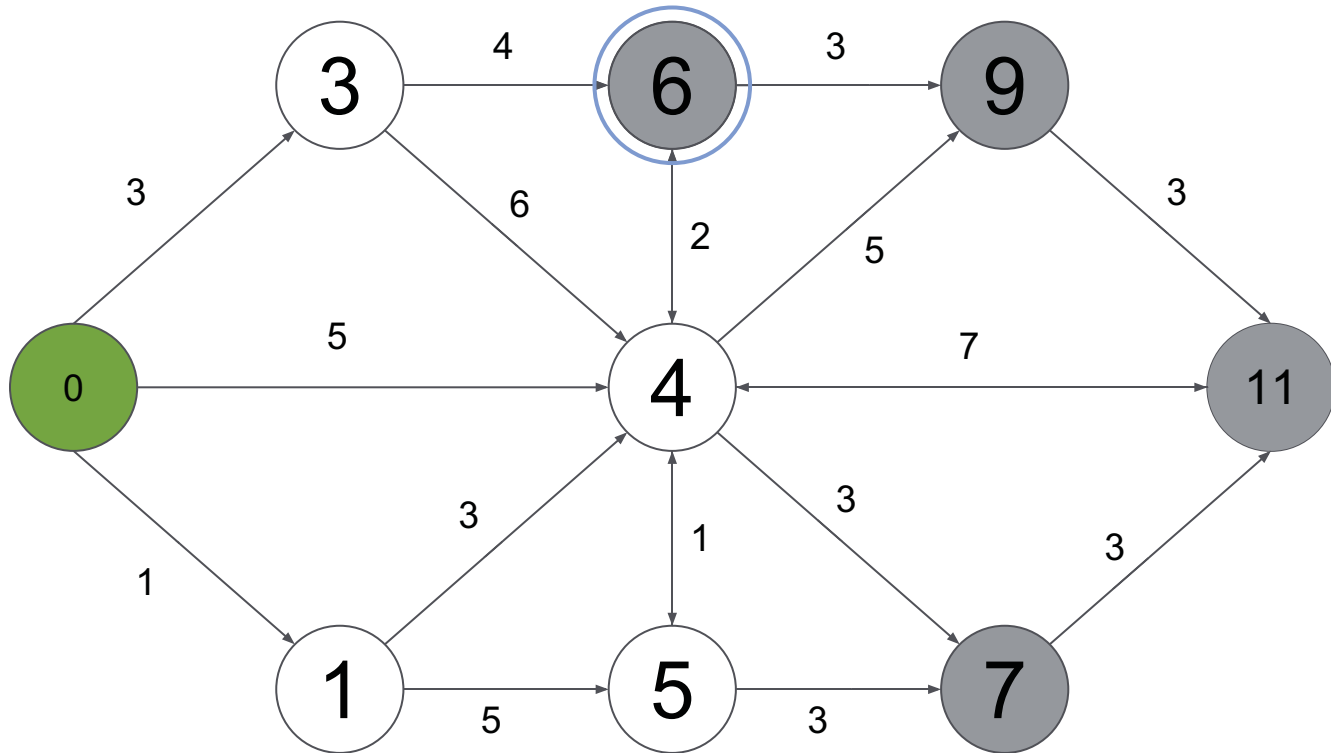
Relaxation - Dijkstra's Algorithm



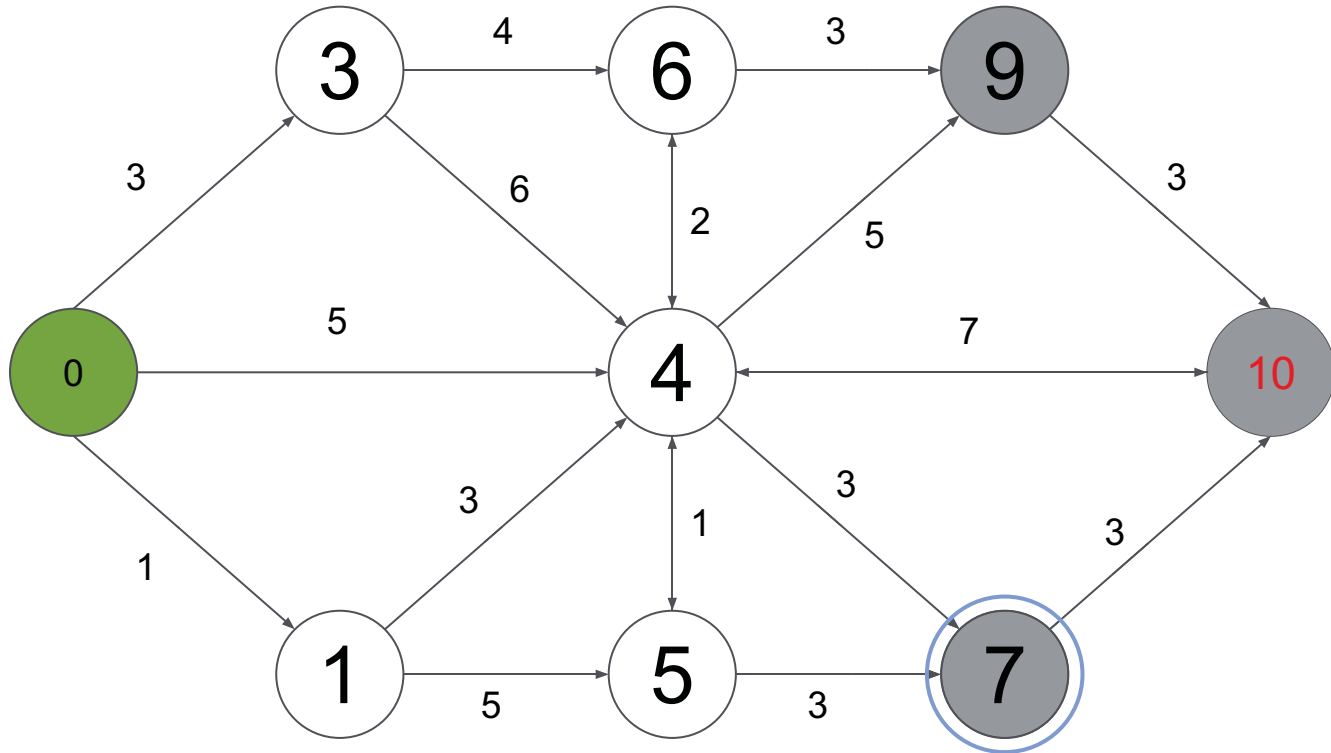
Relaxation - Dijkstra's Algorithm



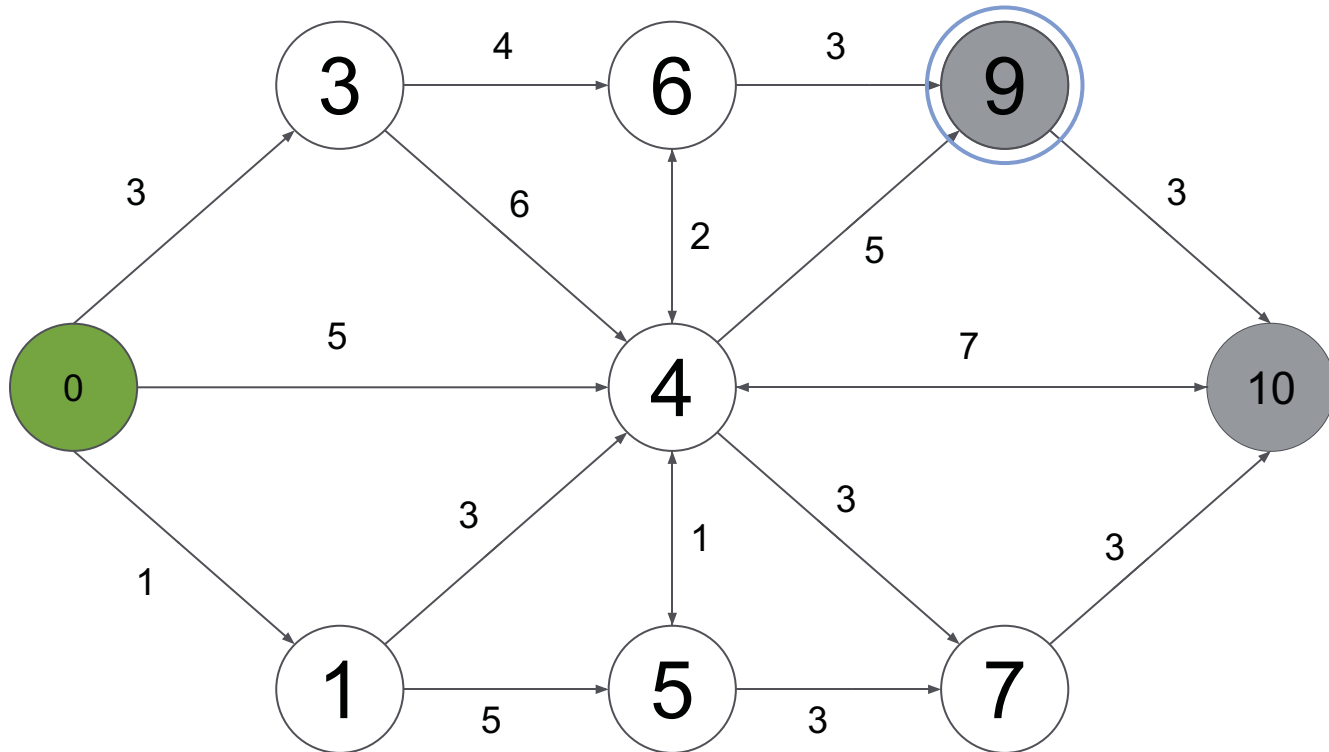
Relaxation - Dijkstra's Algorithm



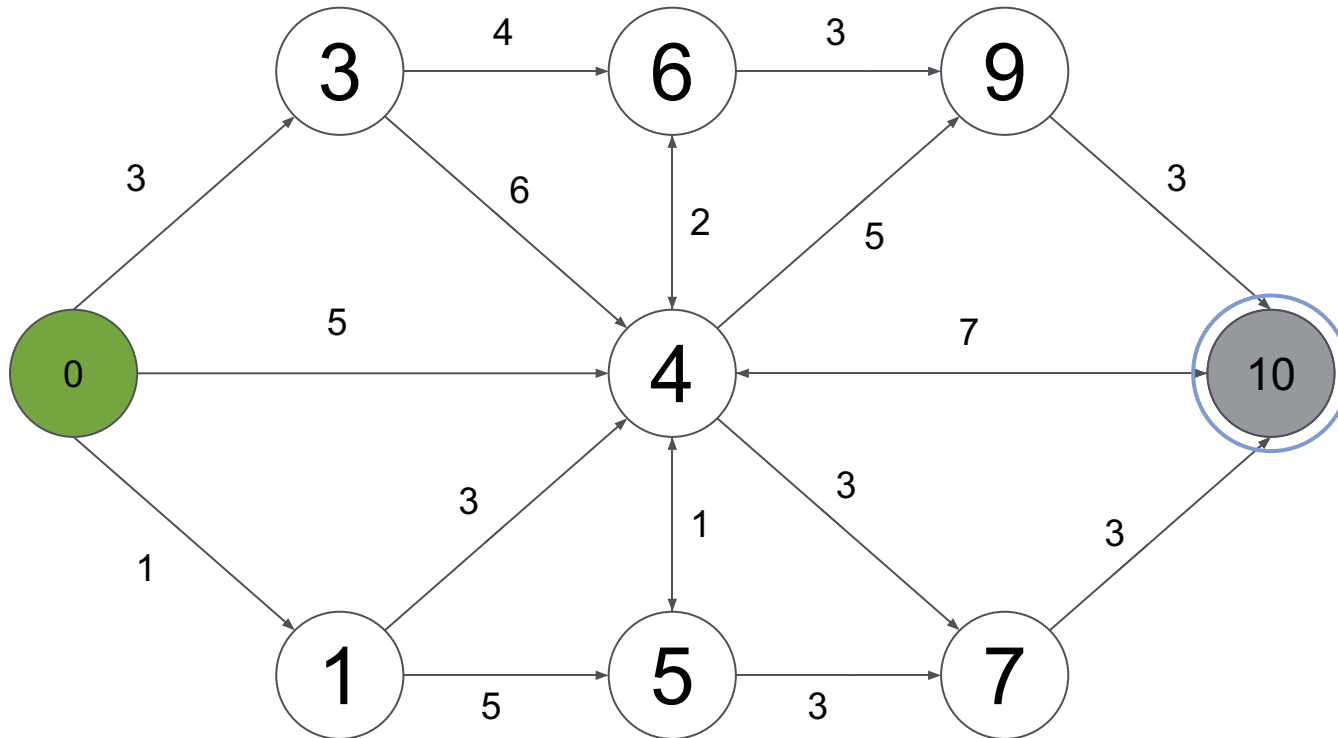
Relaxation - Dijkstra's Algorithm



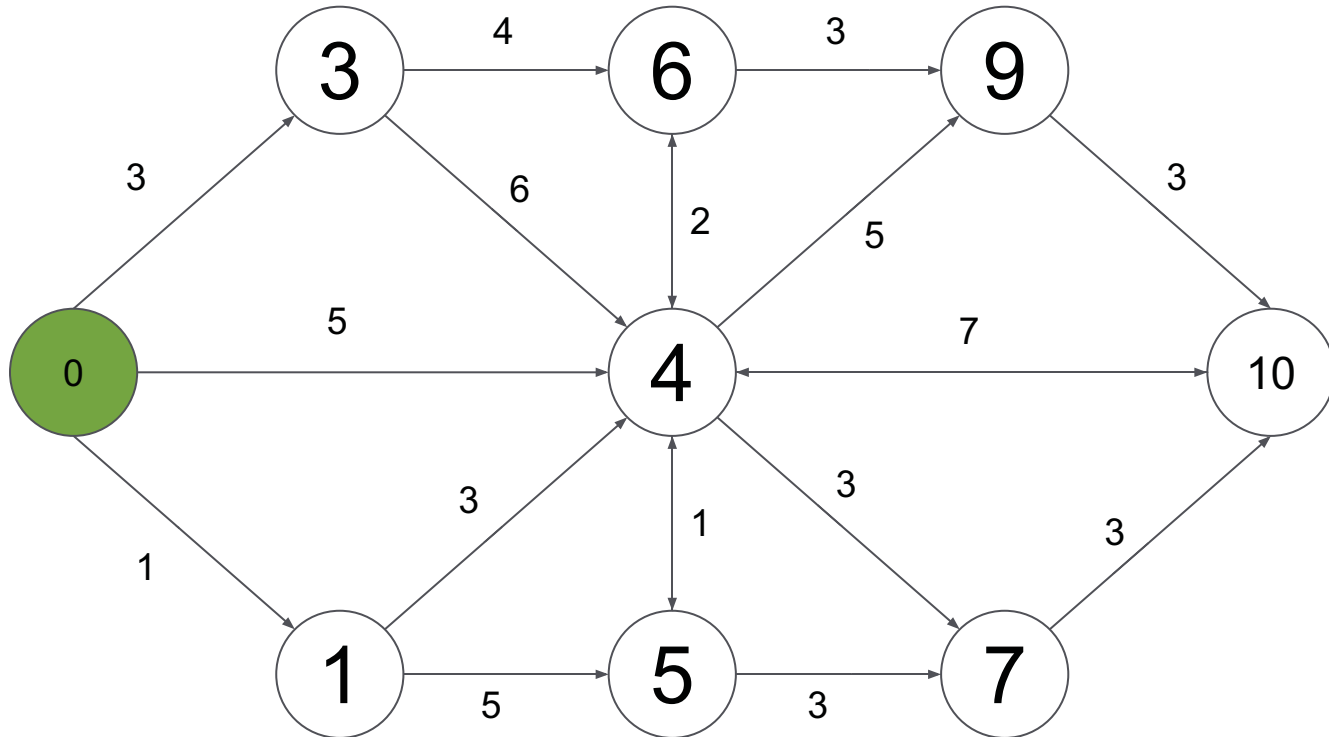
Relaxation - Dijkstra's Algorithm



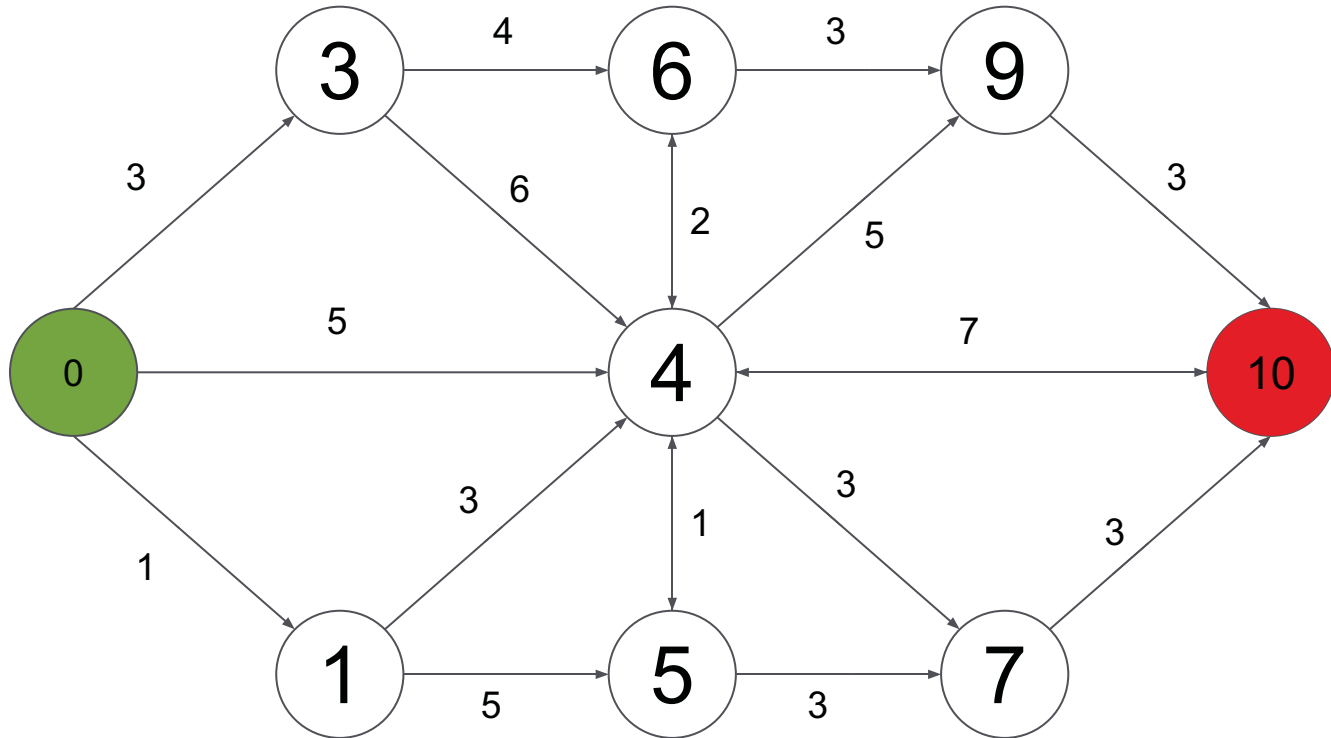
Relaxation - Dijkstra's Algorithm



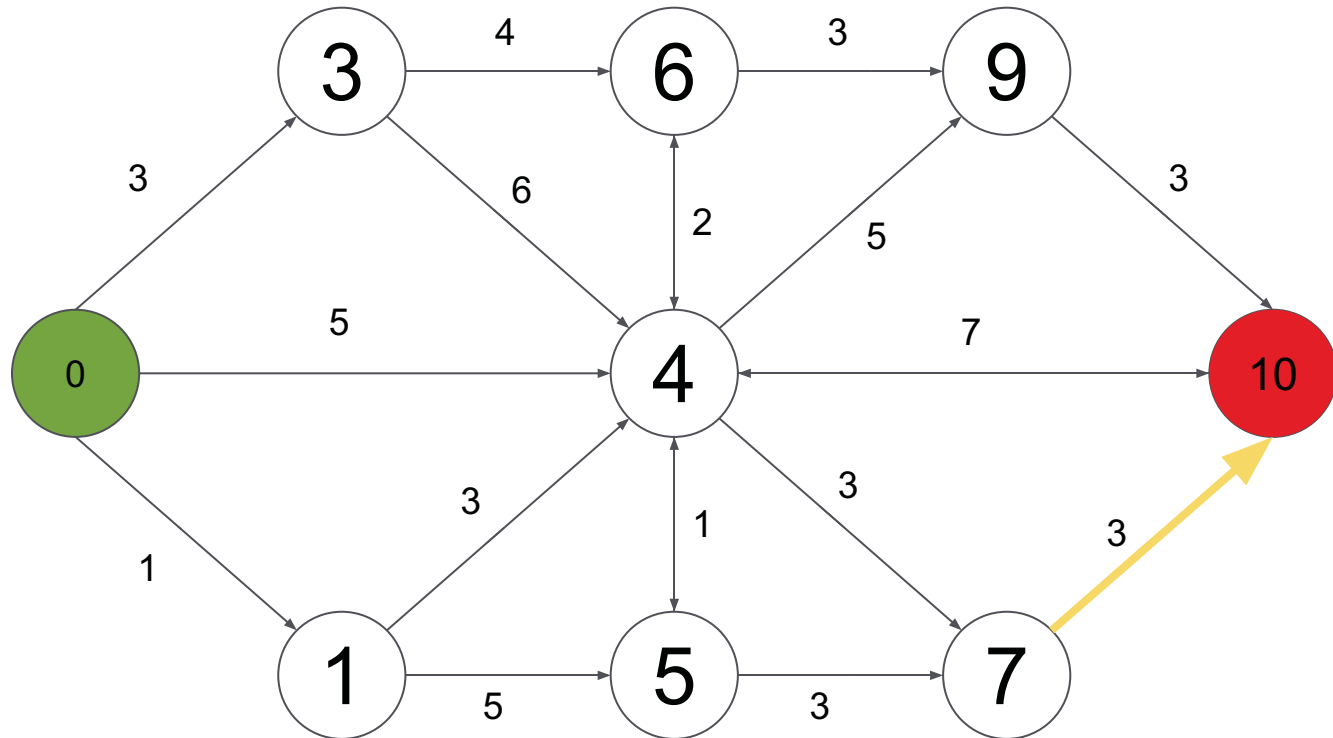
Relaxation - Dijkstra's Algorithm



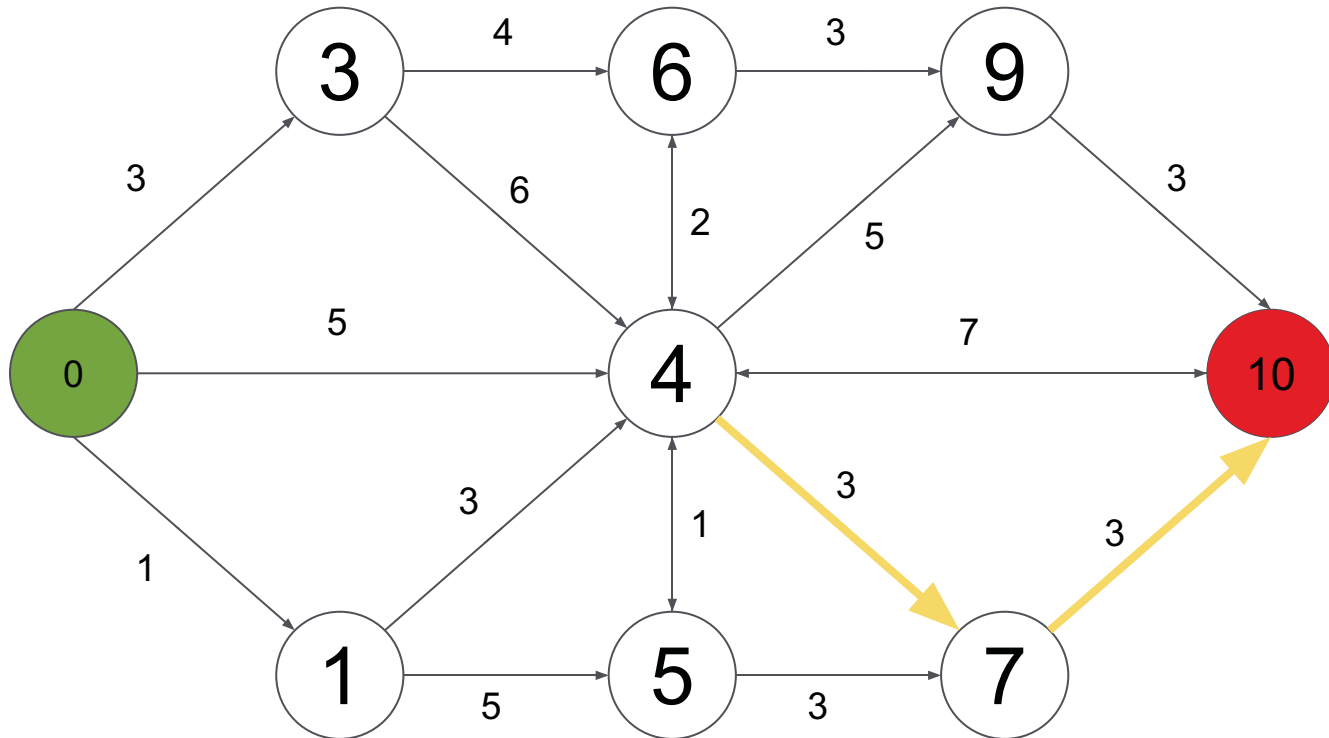
Relaxation - Dijkstra's Algorithm



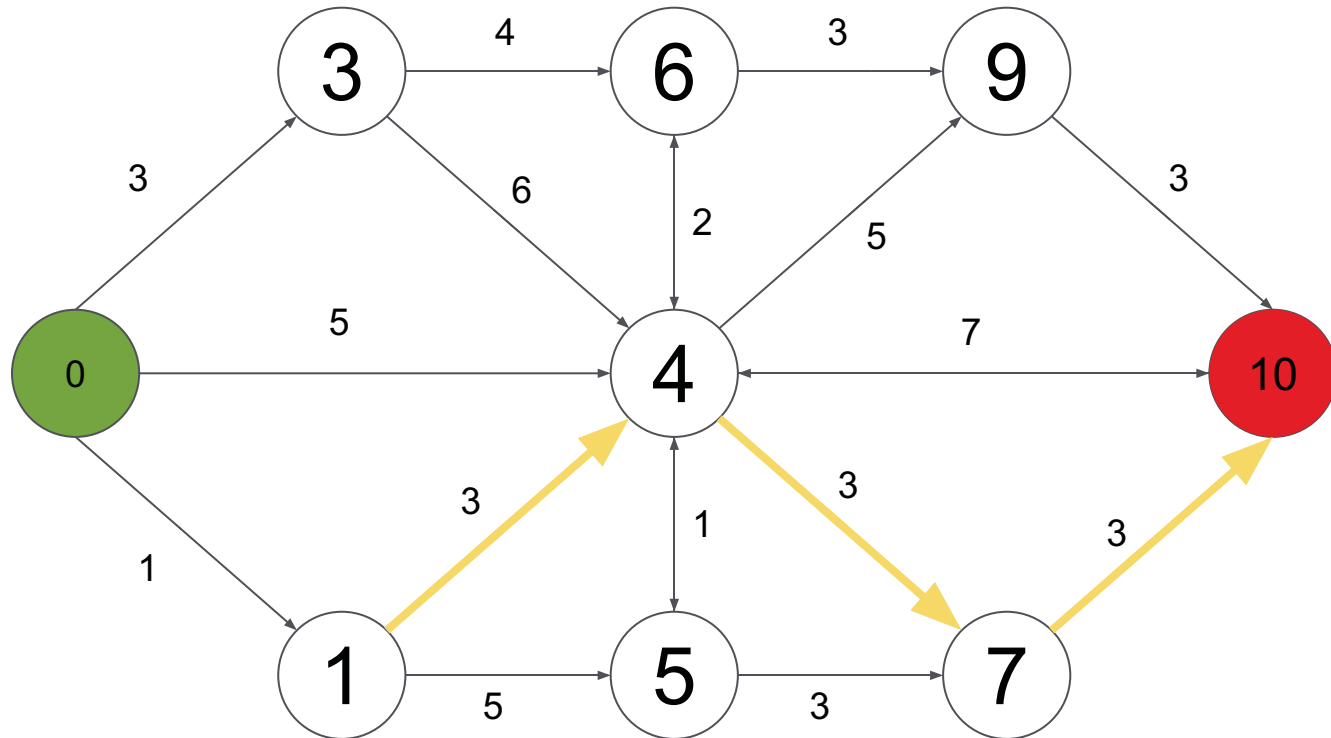
Relaxation - Dijkstra's Algorithm



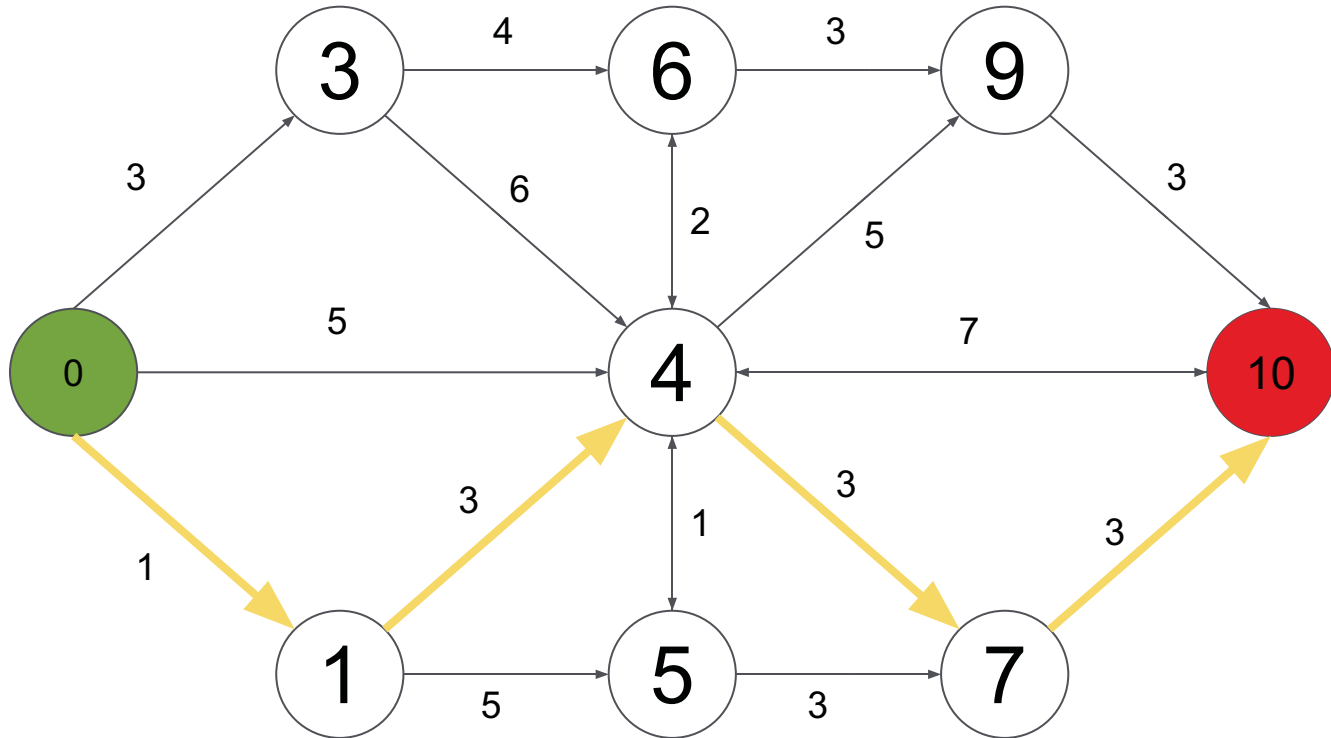
Relaxation - Dijkstra's Algorithm



Relaxation - Dijkstra's Algorithm

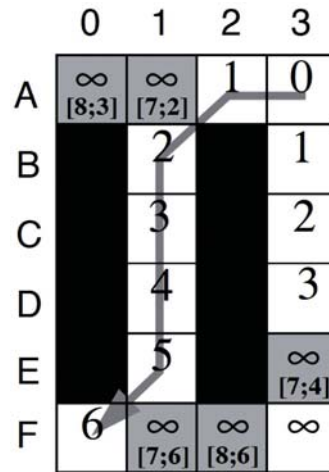


Relaxation - Dijkstra's Algorithm

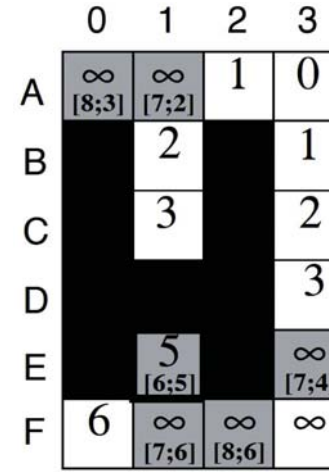


How to Reuse Previous Search Results?

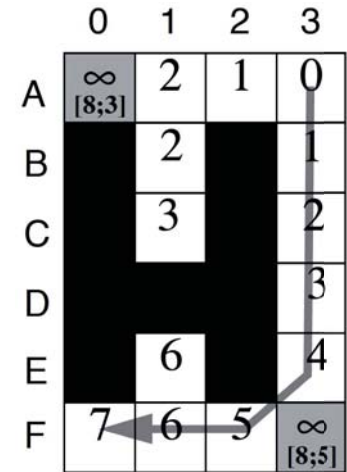
- Store optimal results:
 - shortest paths
 - **g-values**
- Find inconsistencies:
 - **local consistency**
- Make consistent:
 - relaxations



(a)



(b)



(c)

Courtesy of Elsevier, Inc., <http://www.sciencedirect.com>. Used with permission.

Source: Figures 3 and 4 in Keonig, Sven, M. Likhachev, and D. Furcy.

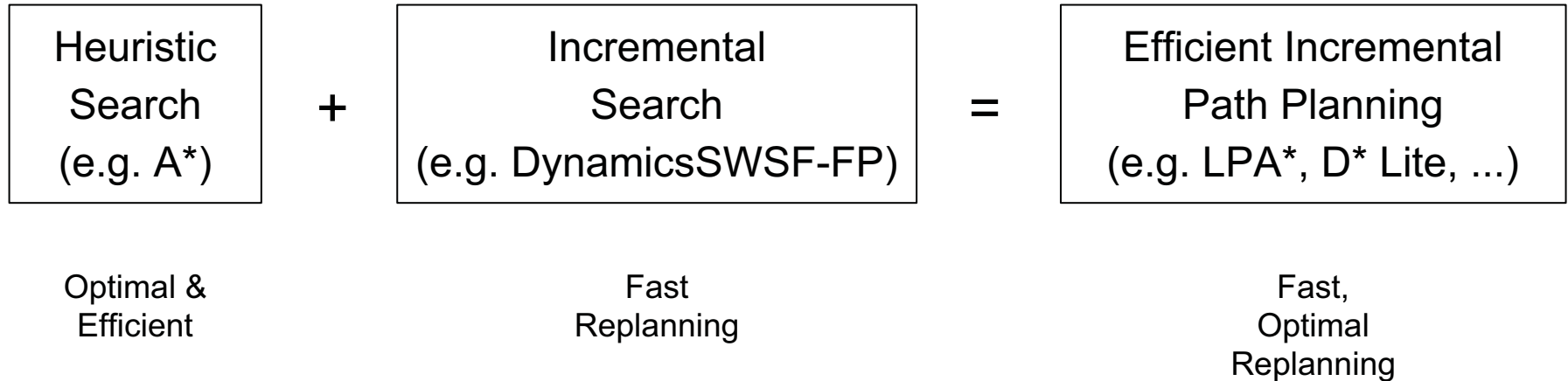
"Lifelong Planning A*." In Artificial Intelligence, 155 (1-2): 93-146.

From Koenig,
Likhachev, &
Furcy (2004)

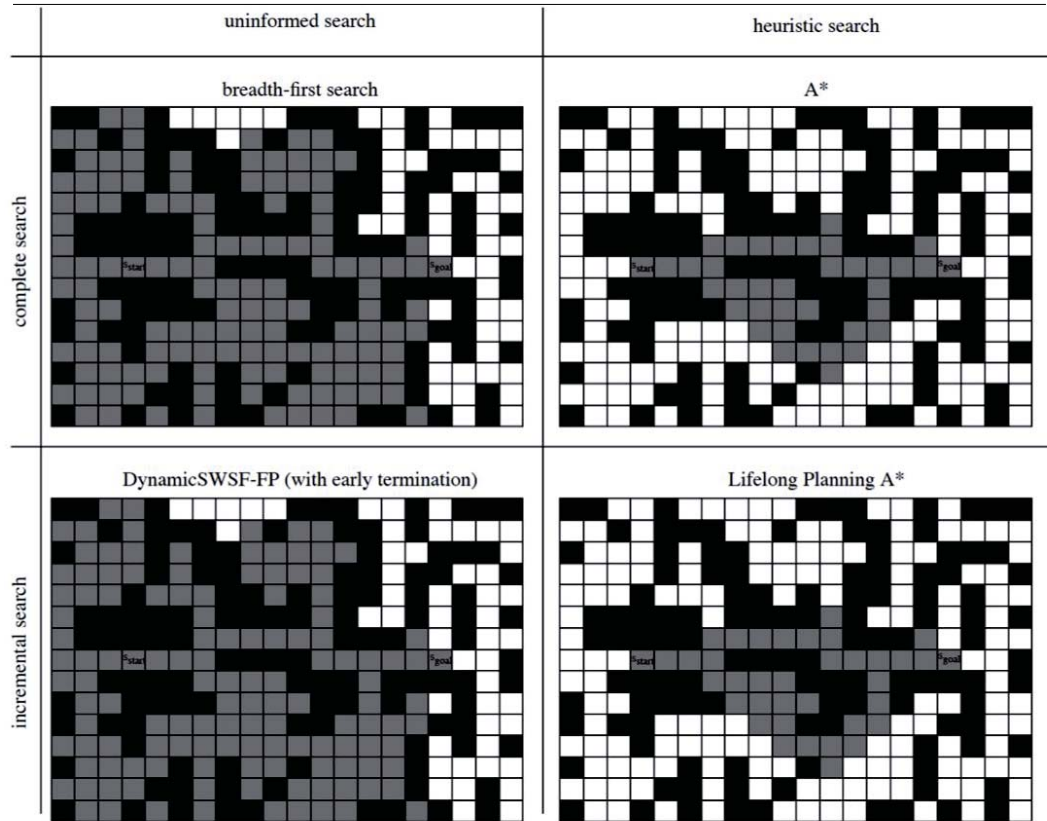
Incremental Search Methods - Examples

- General
 - Incremental APSP
 - Lifelong Planning A* (LPA*)
 - Dynamics Strictly Weakly Superior Function - Fixed Point (DynamicsSWSF-FP)
 - ...
- Mobile Robots
 - D*
 - D* Lite
- Temporal Planning
 - Incremental Temporal Consistency (ITC)
- Propositional Satisfiability
 - Incremental Unit Propagation

D* Incremental Path Planning Approach



Initial Search



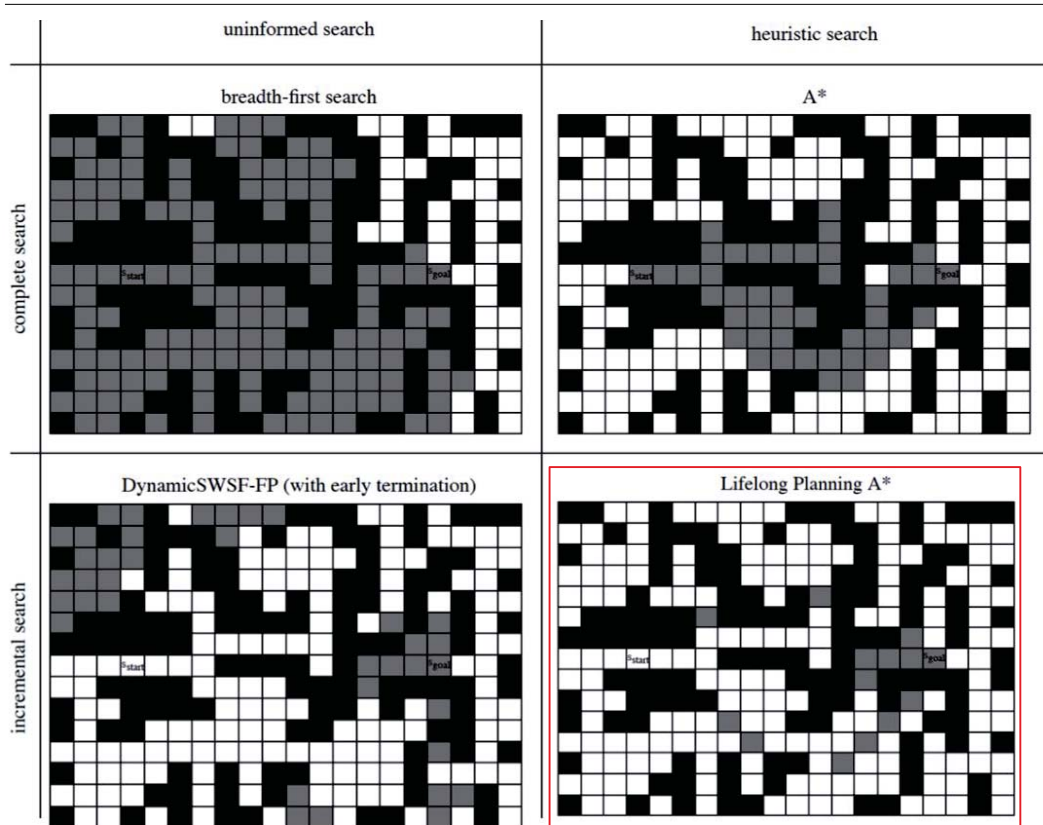
From Koenig,
Likhachev, &
Furcy (2004)

Courtesy of Elsevier, Inc., <http://www.sciencedirect.com>. Used with permission.

Source: Figures 3 and 4 in Koenig, Sven, M. Likhachev, and D. Furcy.

"Lifelong Planning A*." In Artificial Intelligence, 155 (1-2): 93-146.

Follow-on Search



From Koenig,
Likhachev, &
Furcy (2004)

Courtesy of Elsevier, Inc., <http://www.sciencedirect.com>. Used with permission.

Source: Figures 3 and 4 in Keonig, Sven, M. Likhachev, and D. Furcy.

"Lifelong Planning A*." In Artificial Intelligence, 155 (1-2): 93-146.

Outline

- Motivation
- Incremental Search
- **The D* Lite Algorithm**
- D* Lite Example
- When to Use Incremental Path Planning?
- Algorithm Extensions and Related Topics
- Application to Mobile Robotics

A* Reminder

A* is best-first search from start to goal sorted by a cost $f(s)$:

$$f(s) = g(s) + h(s, s_{\text{goal}})$$

$f(s)$ = total node cost

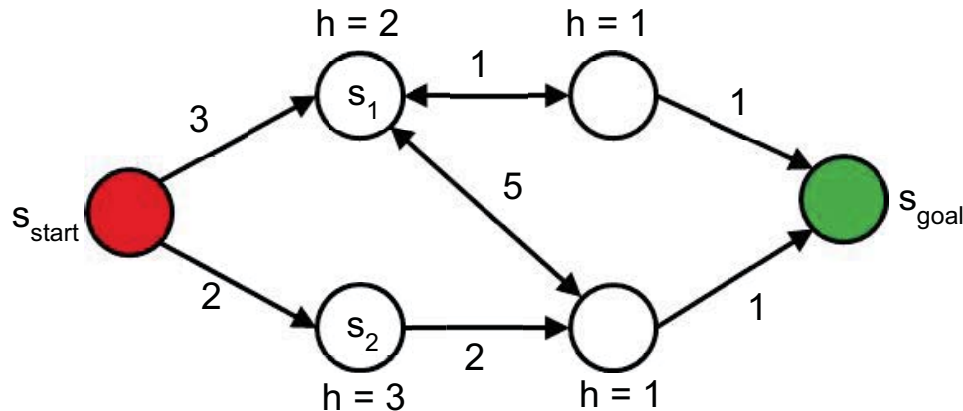
$g(s)$ = path cost to reach vertex from s_{start}

$h(s, s_{\text{goal}})$ = heuristic for cost to reach vertex s_{goal} from vertex s

A*: Choosing Between Ties

Introduce a new notation: $f(s) = \langle f_1(s), f_2(s) \rangle$

$$f(s) = \langle g(s) + h(s, s_{\text{goal}}), g(s) \rangle$$



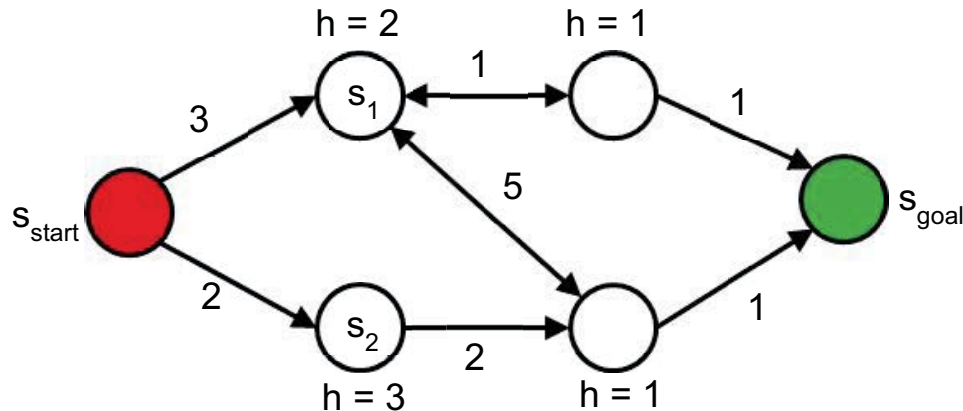
$$f(s_1) = ?$$

$$f(s_2) = ?$$

A*: Choosing Between Ties

Introduce a new notation: $f(s) = \langle f_1(s), f_2(s) \rangle$

$$f(s) = \langle g(s) + h(s, s_{\text{goal}}), g(s) \rangle$$



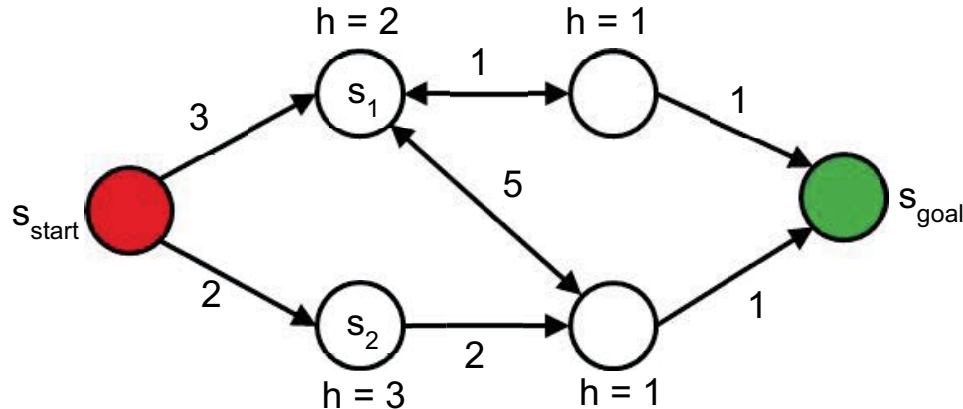
$$f(s_1) = \langle 5, 3 \rangle$$

$$f(s_2) = ?$$

A*: Choosing Between Ties

Introduce a new notation: $f(s) = \langle f_1(s), f_2(s) \rangle$

$$f(s) = \langle g(s) + h(s, s_{\text{goal}}), g(s) \rangle$$



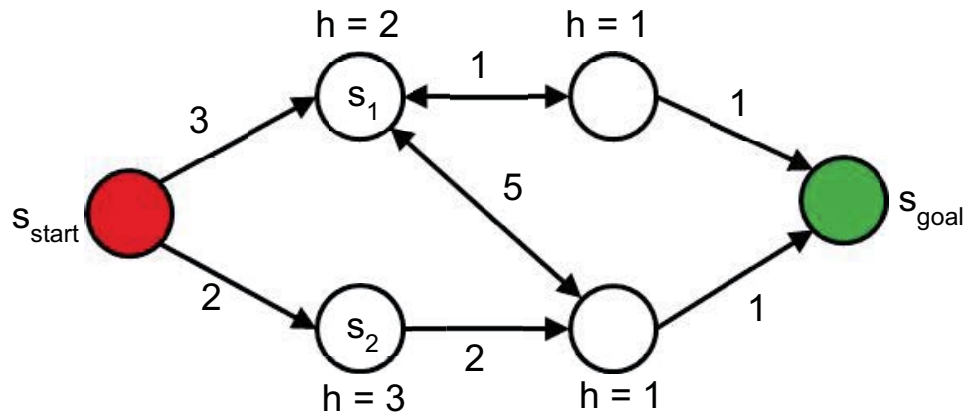
$$f(s_1) = \langle 5, 3 \rangle$$

$$f(s_2) = \langle 5, 2 \rangle$$

A*: Choosing Between Ties

Introduce a new notation: $f(s) = \langle f_1(s), f_2(s) \rangle$

$$f(s) = \langle g(s) + h(s, s_{\text{goal}}), g(s) \rangle$$



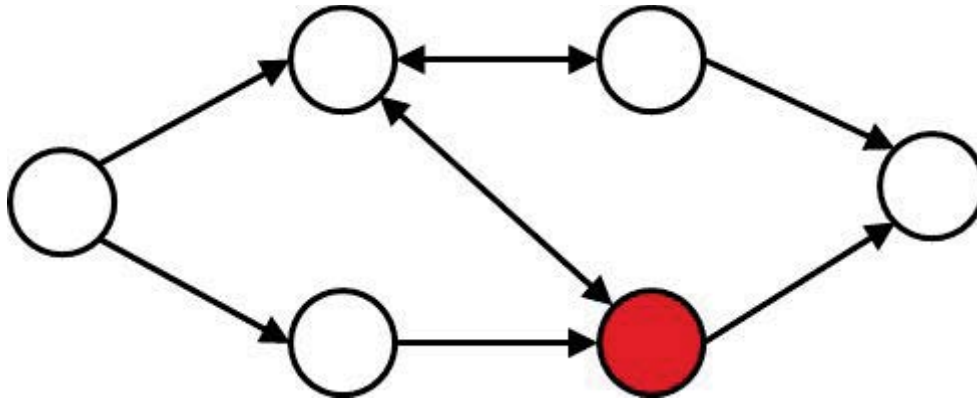
$$f(s_1) = \langle 5, 3 \rangle$$

$$f(s_2) = \langle 5, 2 \rangle$$

Successors and Predecessors

Successors of node s : Every node that can be reached from s , Succ(s)

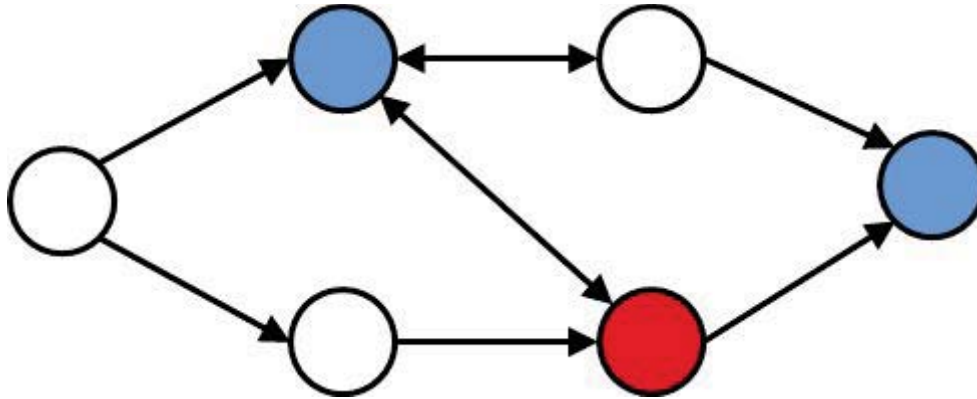
Predecessors of vertex s : Every node from which s can be reached, Pred(s)



Successors and Predecessors

Successors of node s : Every node that can be reached from s , Succ(s)

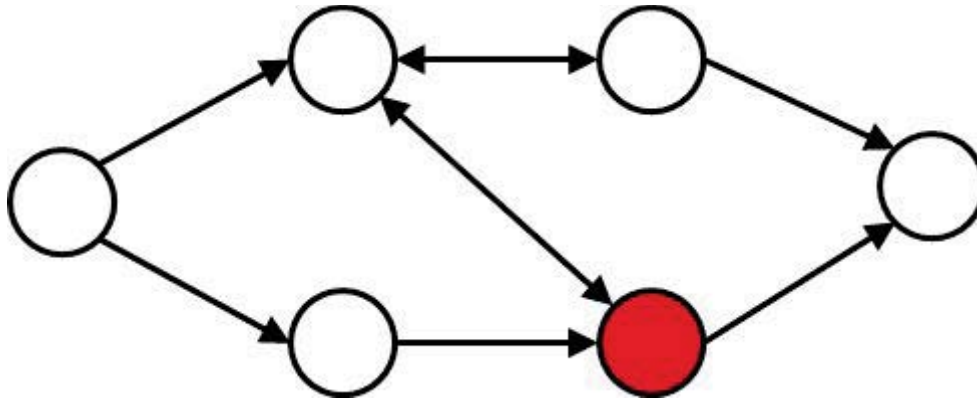
Predecessors of vertex s : Every node from which s can be reached, Pred(s)



Successors and Predecessors

Successors of node s : Every node that can be reached from s , Succ(s)

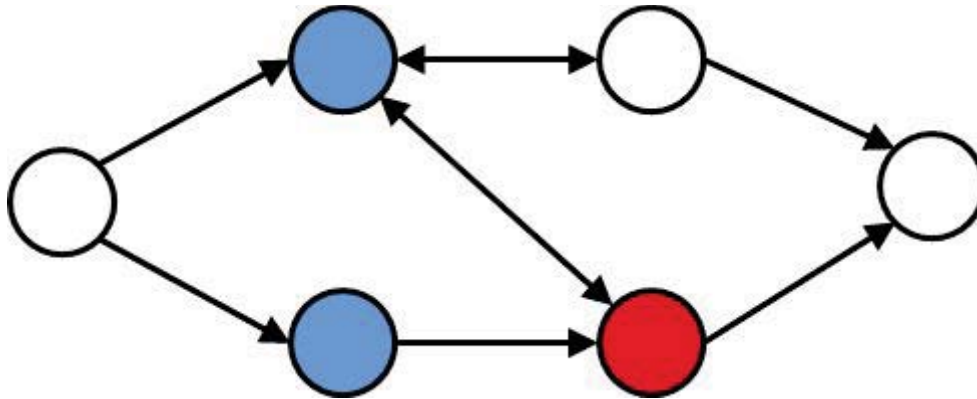
Predecessors of vertex s : Every node from which s can be reached, Pred(s)



Successors and Predecessors

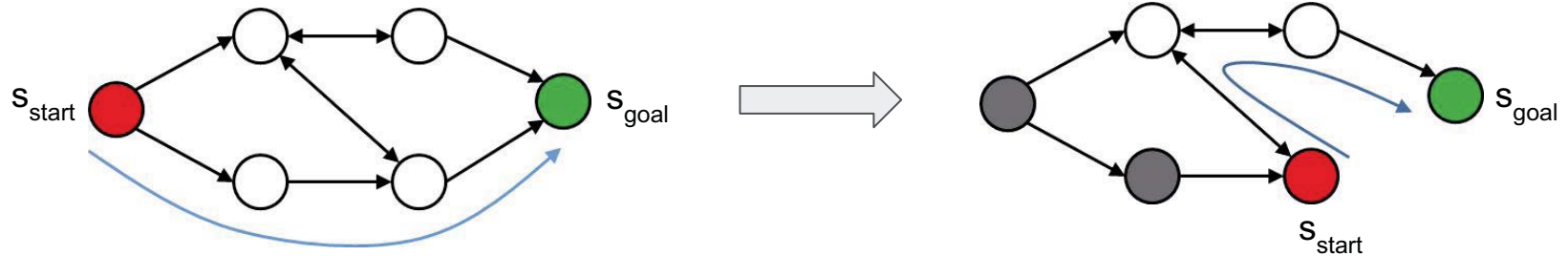
Successors of node s : Every node that can be reached from s , Succ(s)

Predecessors of vertex s : Every node from which s can be reached, Pred(s)



What is D* Lite?

Efficient repeated best-first search through a graph with changing edge weights as the graph is traversed.

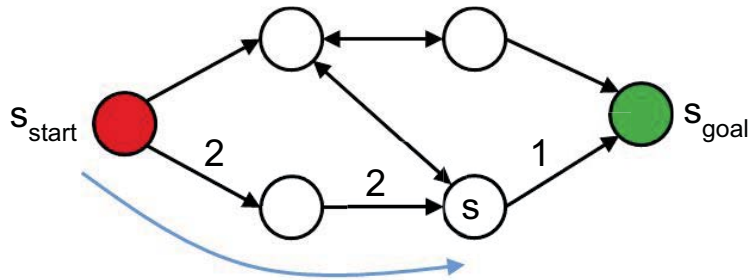


Can be viewed as replanning through relaxation of path costs.

Reformulation: Minimize Recomputation

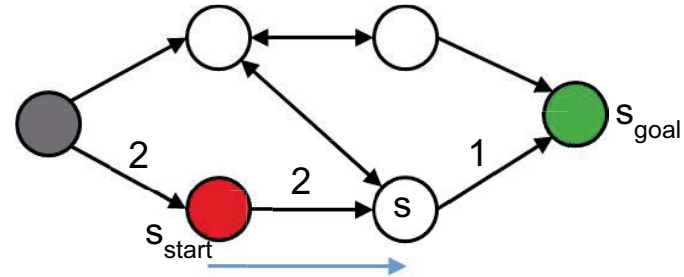
When the start changes, the path cost from start to s is not preserved.

Case 1:



$$g_1(s) = 2 + 2 = 4$$

Case 2:



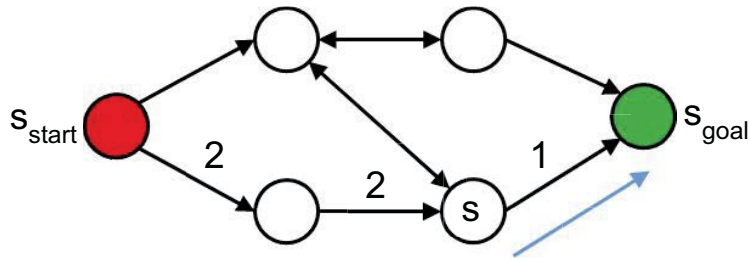
$$g_2(s) = 2$$

$$g_1(s) \neq g_2(s)$$

Reformulation: Minimize Recomputation

Reformulate search from goal to start. Path cost from goal to s is preserved.

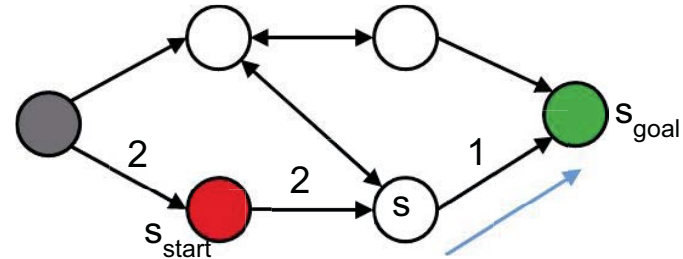
Case 1:



$$g_1(s) = 1$$

$$g_1(s) = g_2(s)$$

Case 2:




$$g_2(s) = 1$$

Overall D* Lite:

1. Initialize all nodes as unexpanded.
2. Best-first search until s_{start} is consistent with neighbors and expanded.
3. Move to next best vertex.
4. If any edge costs change:
 - a. Track how heuristics have changed.
 - b. Update source nodes of changed edges.
5. Repeat from 2.


Overall D* Lite:

1. Initialize all nodes as unexpanded.
 2. Best-first search until s_{start} is consistent with neighbors and expanded.
 3. Move to next best vertex.
 4. If any edge costs change:
 - a. Track how heuristics have changed.
 - b. Update source nodes of changed edges.
 5. Repeat from 2.
- 
- Most computation

Overall D* Lite:

1. Initialize all nodes as unexpanded.
2. Best-first search until s_{start} is consistent with neighbors and expanded.
3. Move to next best vertex.
4. If any edge costs change:
 - a. Track how heuristics have changed.
 - b. Update source nodes of changed edges.
5. Repeat from 2.

Incremental component



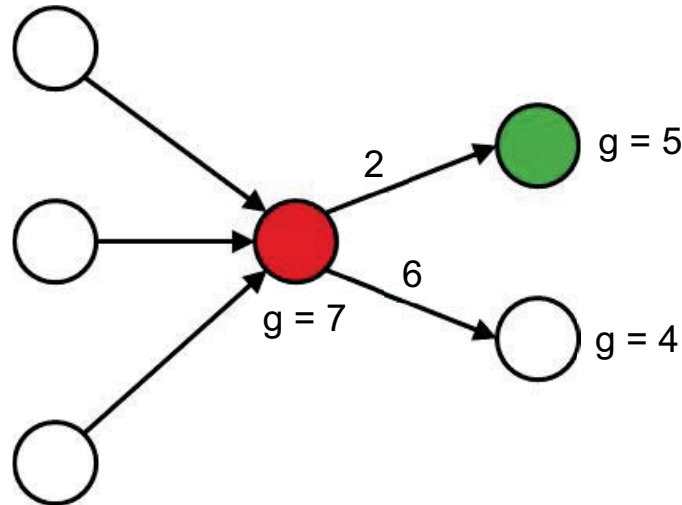
Overall D* Lite:

1. Initialize all nodes as unexpanded.
2. Best-first search until s_{start} is consistent with neighbors and expanded.
3. Move to next best vertex.
4. If any edge costs change:
 - a. Track how heuristics have changed.
 - b. Update source nodes of changed edges.
5. Repeat from 2.

Extracting a Path Given Path Cost

Move from s_{start} to the successor which gives s_{start} the lowest path cost.

$$s_{\text{start}} \leftarrow \operatorname{argmin}_{s' \in \operatorname{Succ}(s_{\text{start}})} (c(s_{\text{start}}, s') + g(s'))$$



Overall D* Lite:

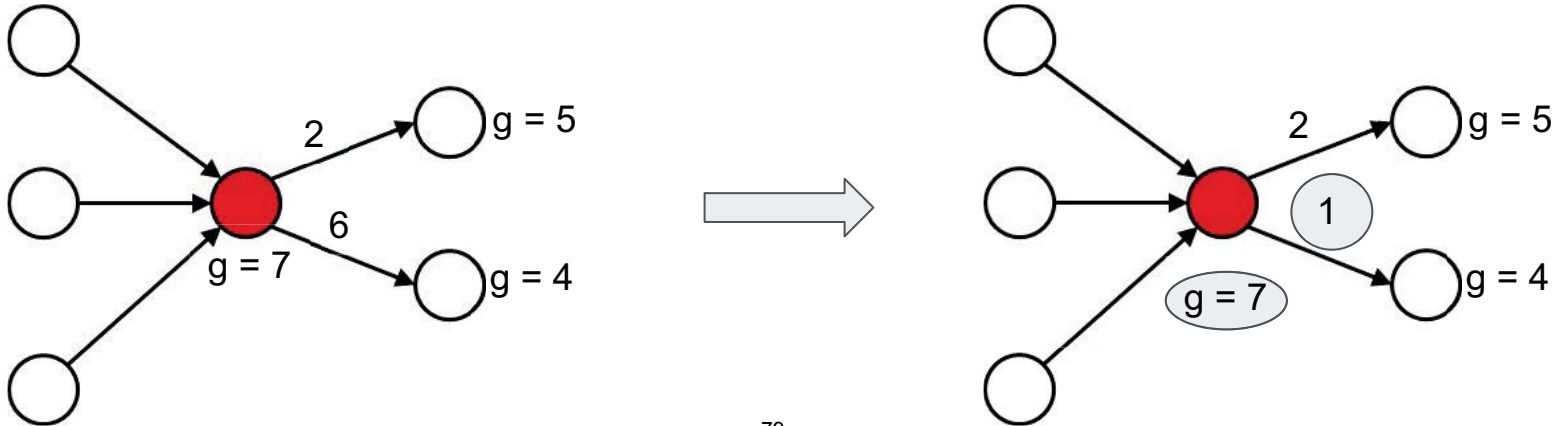
1. Initialize all nodes as unexpanded.
2. Best-first search until s_{start} is consistent with neighbors and expanded.
3. Move to next best vertex.
4. If any edge costs change:
 - a. Track how heuristics have changed.
 - b. Update source nodes of changed edges.
5. Repeat from 2.

Handling Weight Changes Locally

Self-consistent graph:

$$g(s) = \min_{s' \in \text{Succ}(s)} (g(s') + c(s, s'))$$

May no longer be true when edge weights change!

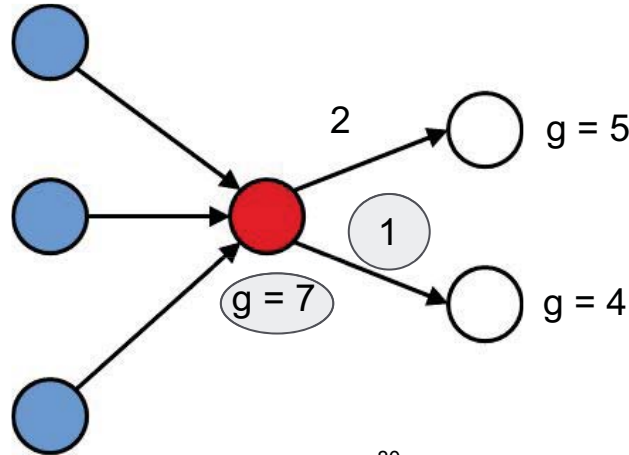


Handling Weight Changes Locally

Changes propagate to predecessors.

For efficient search, update lowest cost nodes first.

→ Use a priority queue like A*. Update nodes until the goal is first expanded.

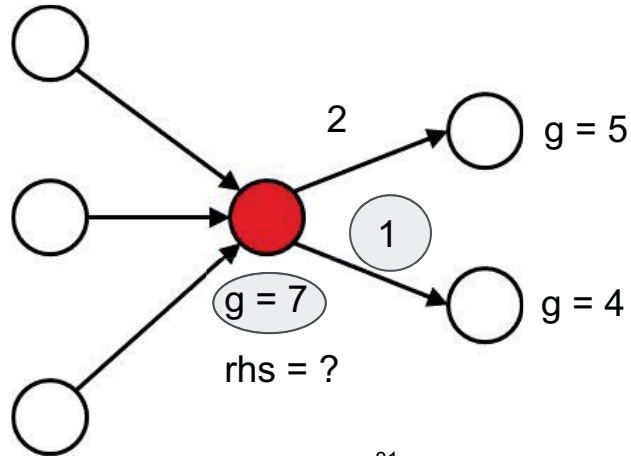


Handling Weight Changes Locally

Store an additional value:

$$\text{rhs}(s) = \min_{s' \in \text{Succ}(s)} (g(s') + c(s, s'))$$

Local inconsistency: $\text{rhs}(s) \neq g(s)$

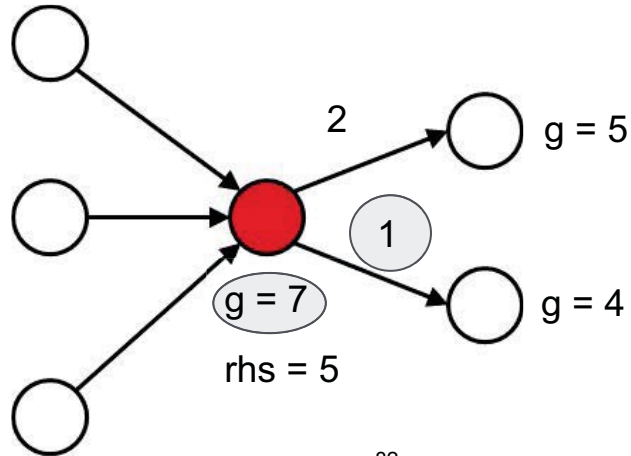


Handling Weight Changes Locally

Store an additional value:

$$\text{rhs}(s) = \min_{s' \in \text{Succ}(s)} (g(s') + c(s, s'))$$

Local inconsistency $\text{rhs}(s) \neq g(s)$



Local Inconsistencies

Signal recomputation is necessary for node and predecessors

1. Locally overconsistent:

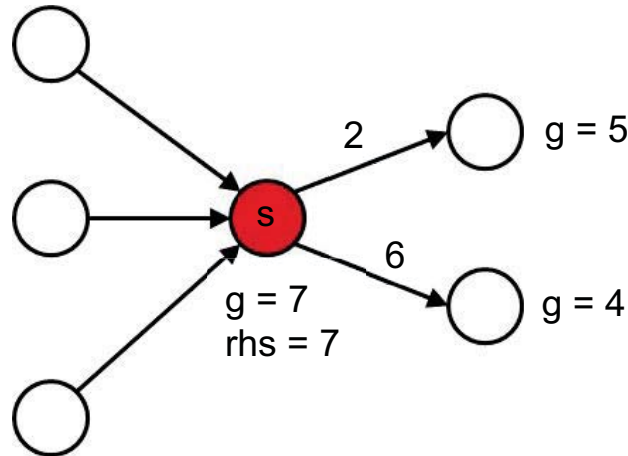
$$g(s) > \text{rhs}(s)$$

2. Locally underconsistent:

$$g(s) < \text{rhs}(s)$$

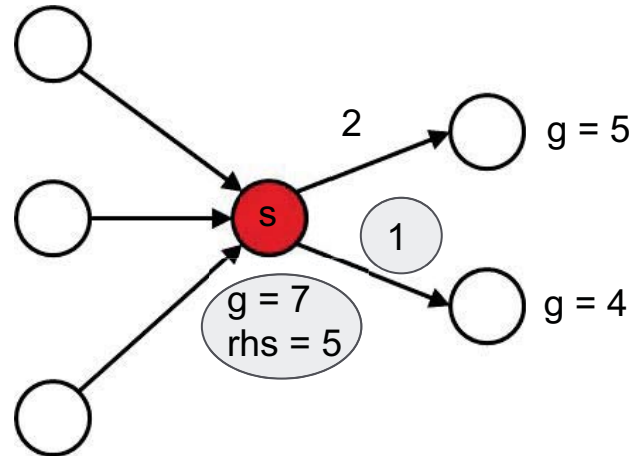
Updating and Expanding Nodes

Update by recomputing rhs and placing the node on the priority queue if locally inconsistent.



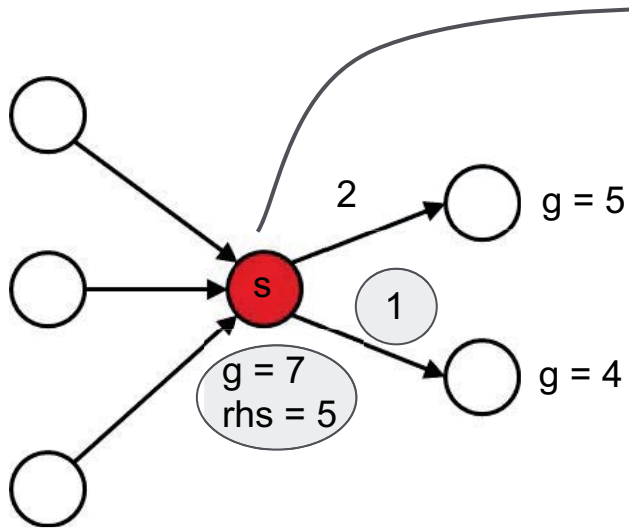
Updating and Expanding Nodes

Update by recomputing rhs and placing the node on the priority queue if locally inconsistent.



Updating and Expanding Nodes

Update by recomputing rhs and placing the node on the priority queue if locally inconsistent.

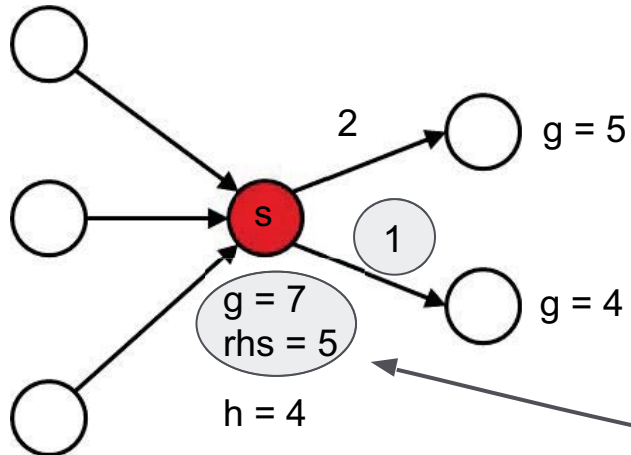


$$Q = [\dots, s, \dots]$$

Updating and Expanding Nodes

Expand by taking it off the priority queue and changing g . Expand in order of total cost:

$$\langle \min[g(s), \text{rhs}(s)] + h(s, s_{\text{start}}), \min[g(s), \text{rhs}(s)] \rangle$$

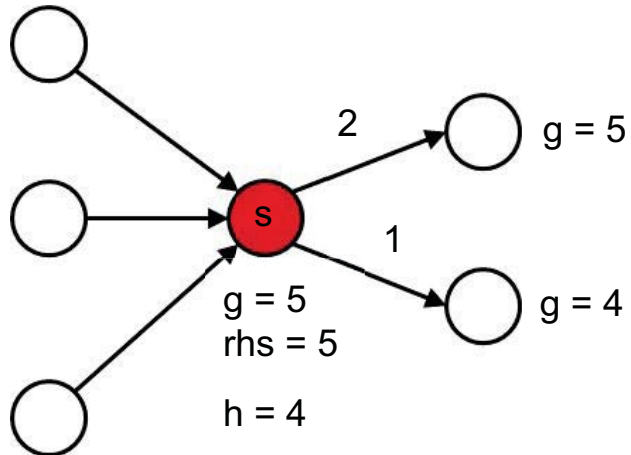


$$Q = [(s, \langle 9, 5 \rangle), \dots]$$

Updating and Expanding Nodes

Expand by taking it off the priority queue and changing g . Expand in order of total cost:

$$\langle \min[g(s), \text{rhs}(s)] + h(s, s_{\text{start}}), \min[g(s), \text{rhs}(s)] \rangle$$



$$Q = [\dots]$$

Updating and Expanding Nodes

Why not recompute $g(s)$ at the same time as $rhs(s)$?

- Make sure that we have updated all successors that could lower the total cost $f(s)$ first.
- s can be updated multiple times before expansion, so $rhs(s)$ can change on the queue.

$g(s) > rhs(s)$ (Overconsistent):

New path cost $rhs(s)$ is better than the old path cost $g(s)$.

Immediately update $g(s)$ to $rhs(s)$ and propagate to all predecessors.

- Set $g(s) = rhs(s)$
- Update all predecessors of s

Vertex is now locally consistent and will remain that way.

$g(s) < rhs(s)$ (Underconsistent):

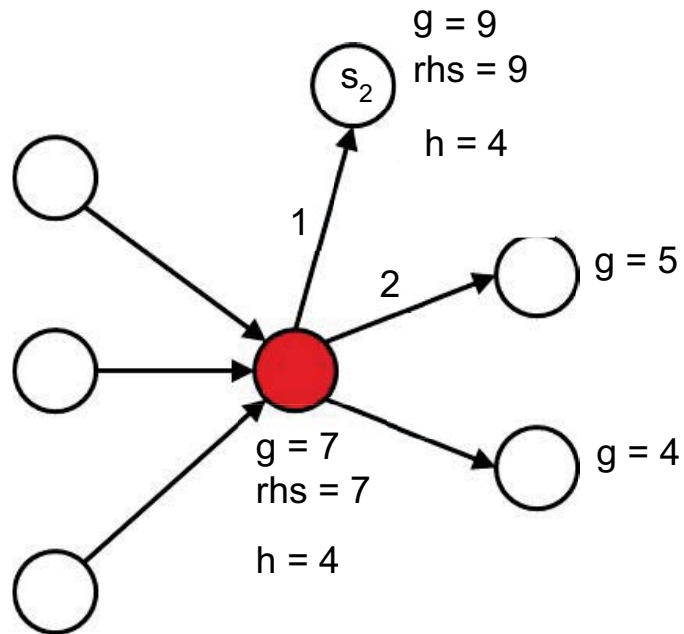
Old path cost $g(s)$ is better than new path cost $rhs(s)$.

Delay vertex expansion and propagate to all predecessors.

- Set $g(s) = \infty$
- Update all predecessors of s and s itself

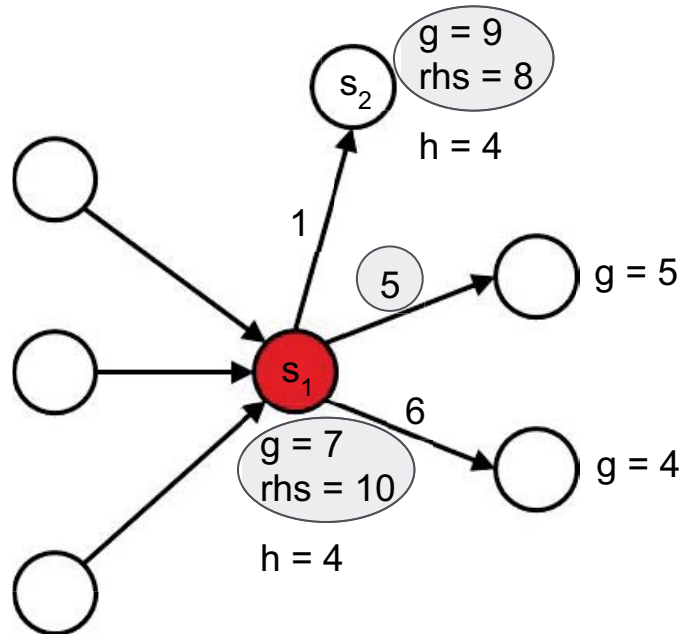
Vertex is now locally consistent or locally overconsistent. It will remain on the queue until $rhs(s)$ is the next best cost.

$g(s) < rhs(s)$ (Underconsistent):



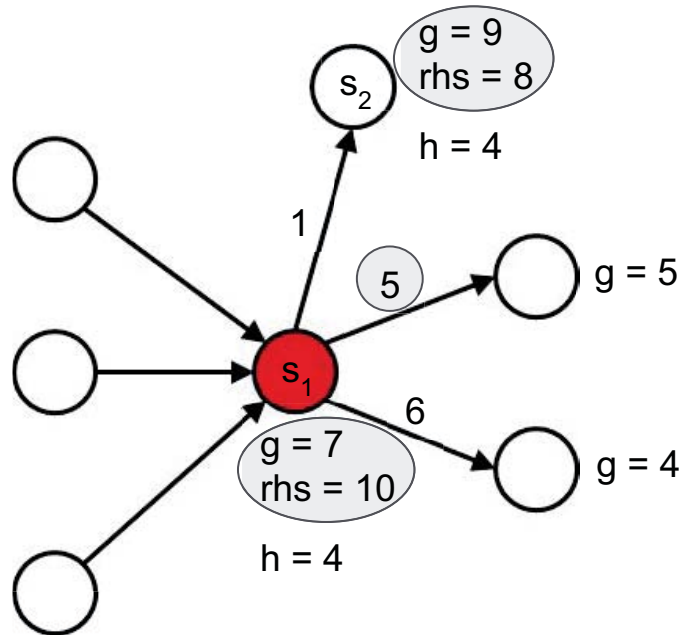
$g(s) < rhs(s)$ (Underconsistent):

Assume after update:



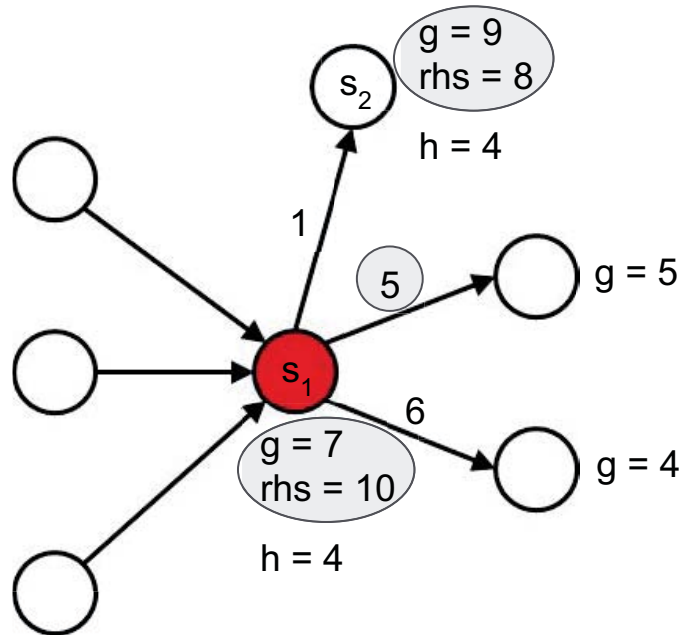
$g(s) < rhs(s)$ (Underconsistent):

Assume after update:



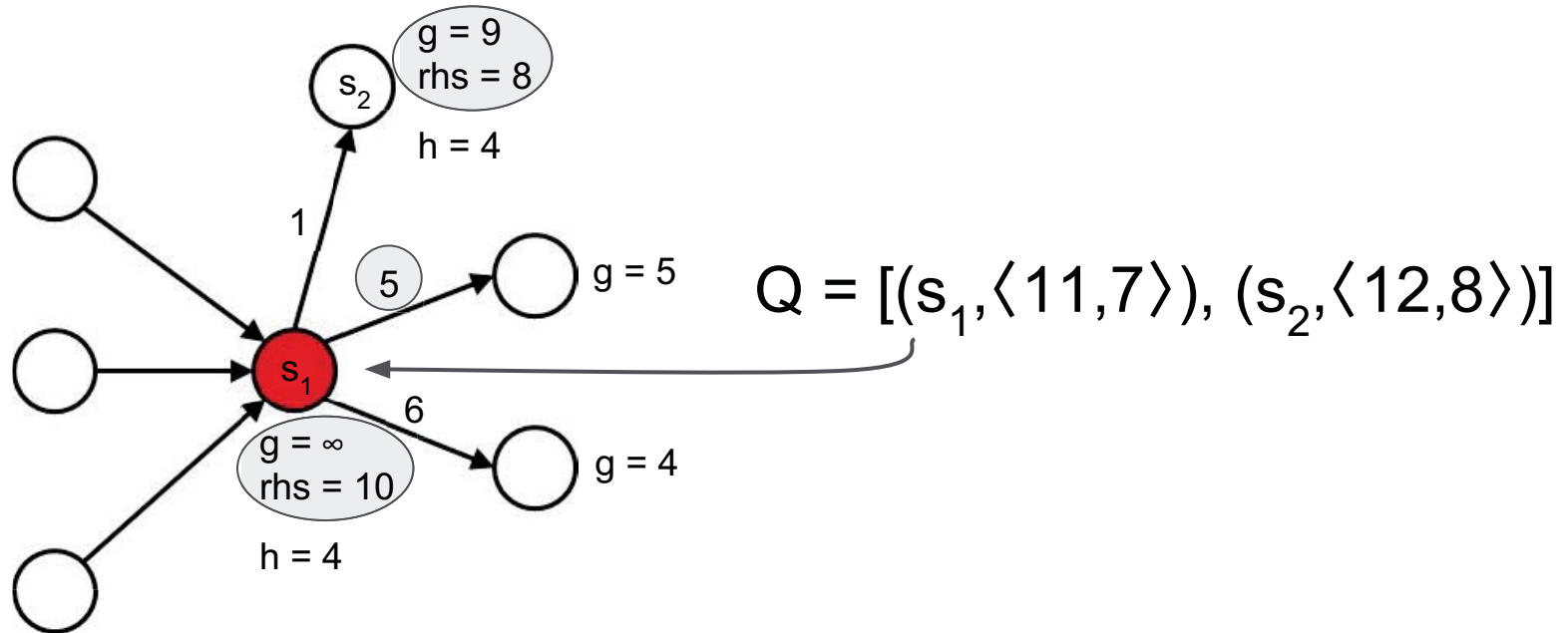
$$Q = [(s_1, \langle 11, 7 \rangle), (s_2, \langle 12, 8 \rangle)]$$

$g(s) < rhs(s)$ (Underconsistent):

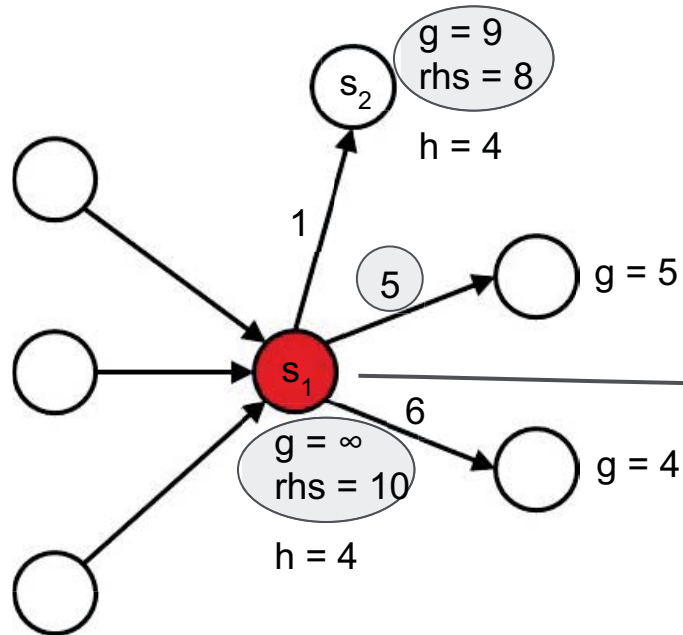


$$Q = [(s_1, \langle 11, 7 \rangle), (s_2, \langle 12, 8 \rangle)]$$

$g(s) < rhs(s)$ (Underconsistent):

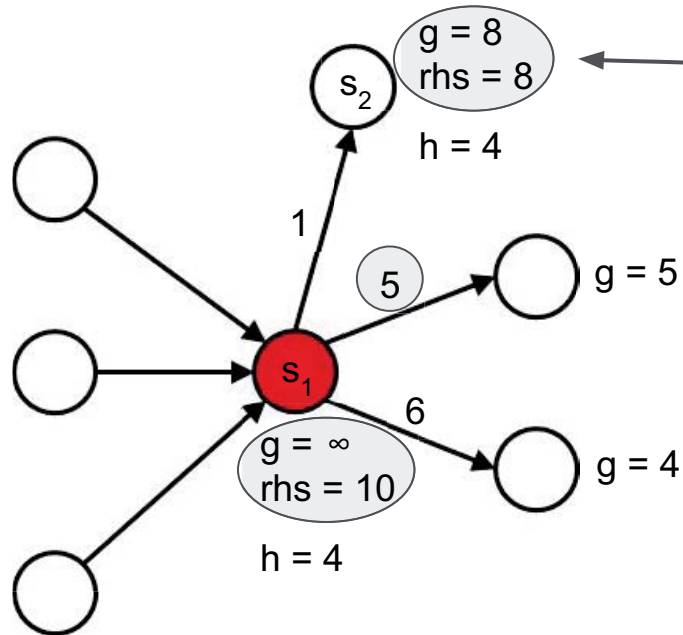


$g(s) < rhs(s)$ (Underconsistent):



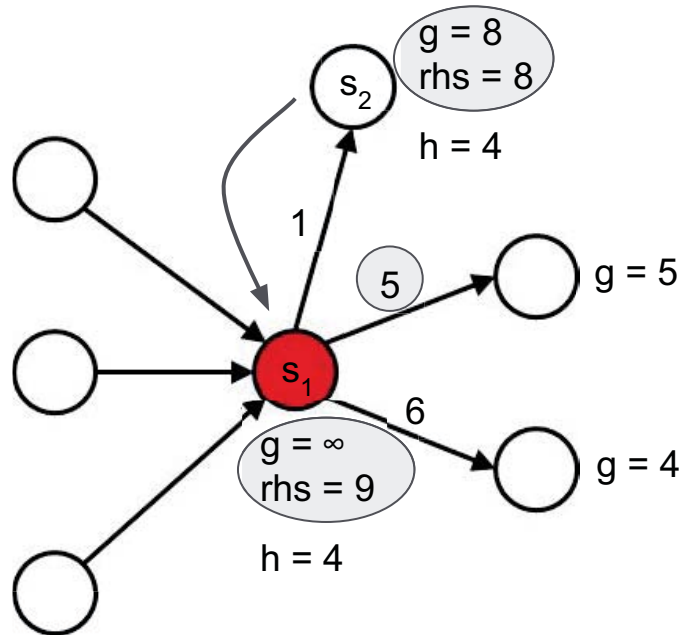
$Q = [(s_2, \langle 12, 8 \rangle), (s_1, \langle 14, 10 \rangle)]$

$g(s) < rhs(s)$ (Underconsistent):



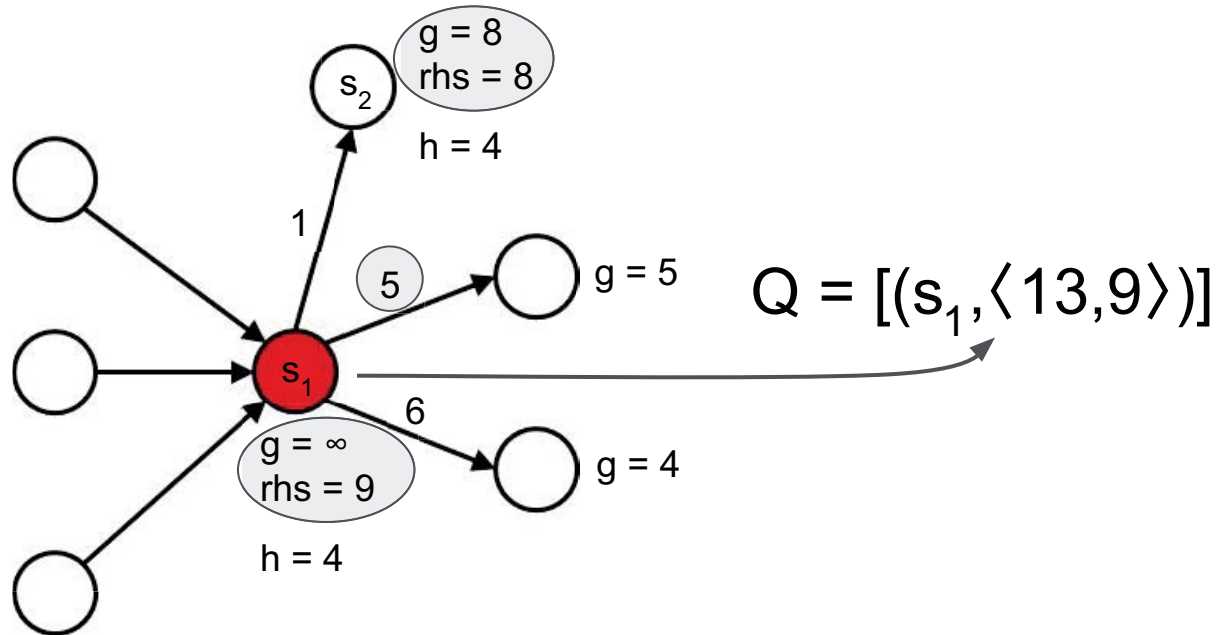
$Q = [(s_2, \langle 12, 8 \rangle), (s_1, \langle 14, 10 \rangle)]$

$g(s) < rhs(s)$ (Underconsistent):

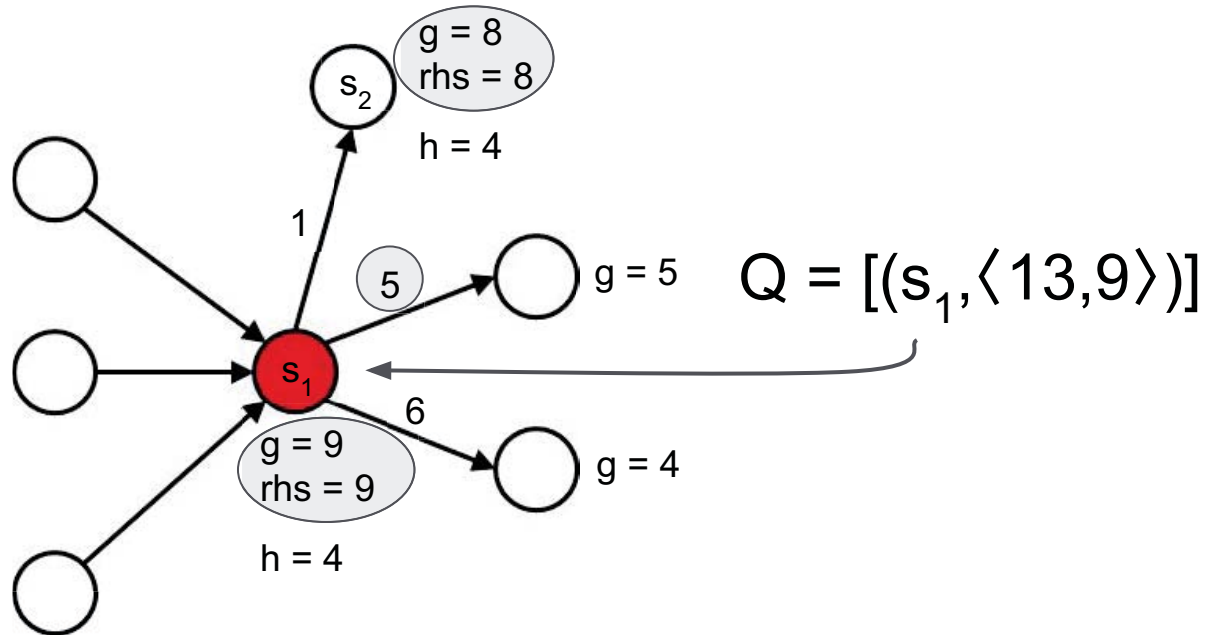


$$Q = [(s_1, \langle 14, 10 \rangle)]$$

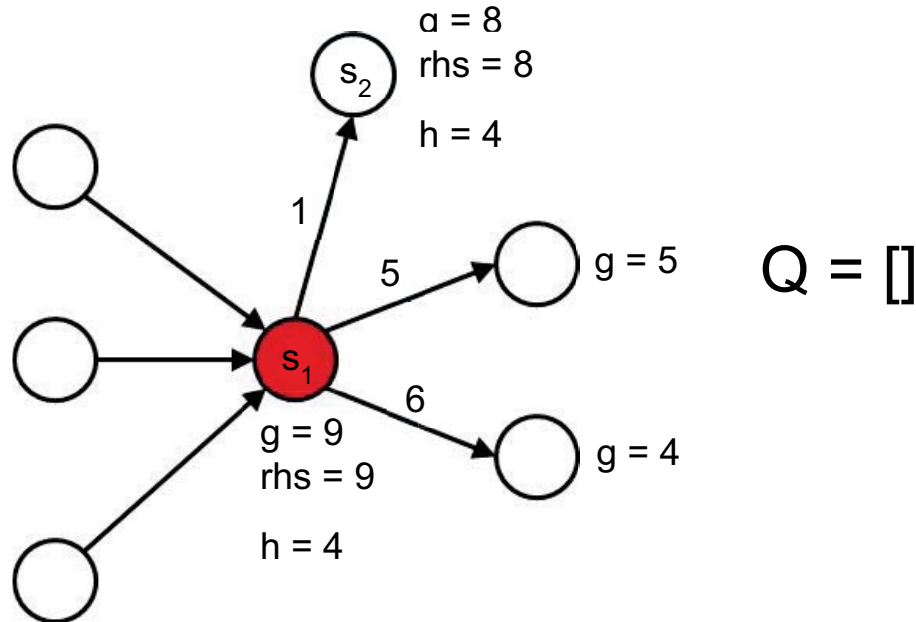
$g(s) < rhs(s)$ (Underconsistent):



$g(s) < rhs(s)$ (Underconsistent):



$g(s) < rhs(s)$ (Underconsistent):



Overall D* Lite:

1. Initialize all nodes as unexpanded.
2. Best-first search until s_{start} is consistent with neighbors and expanded.
3. Move to next best vertex.
4. If any edge costs change:
 - a. Track how heuristics have changed.
 - b. Update source nodes of changed edges.
5. Repeat from 2.

Carrying Over the Priority Queue

After a path is found, the priority queue is not empty.

The value on the priority queue is:

$$\langle \min[g(s), \text{rhs}(s)] + h(s, s_{\text{start}}), \min[g(s), \text{rhs}(s)] \rangle$$

When s_{start} is different, the heuristics are different!

Carrying Over the Priority Queue

$s_{last} \rightarrow s_{start}$: All heuristics lowered by at most $h(s_{last}, s_{start})$.

When adding new nodes to the queue, increase total cost by $h(s_{last}, s_{start})$.

Increase $k_m = k_m + h(s_{last}, s_{start})$:

$$\langle \min[g(s), \text{rhs}(s)] + h(s, s_{start}) + k_m, \min[g(s), \text{rhs}(s)] \rangle$$

Overall D* Lite:

1. Initialize all $g(s) = \infty$, $rhs(s \neq s_{goal}) = \infty$, $rhs(s_{goal}) = 0$, $k_m = 0$, $s_{last} = s_{start}$.
2. Best-first search until s_{start} is locally consistent and expanded.
3. Move so $s_{start} = \operatorname{argmin}_{s' \in \operatorname{Succ}(s_{start})} (c(s_{start}, s') + g(s'))$.
4. If any edge costs change:
 - a. $k_m = k_m + h(s_{last}, s_{start})$.
 - b. Update rhs and queue position for source nodes of changed edges.
5. Repeat from 2.

Always sort by $\langle \min[g(s), rhs(s)] + h(s, s_{start}) + k_m, \min[g(s), rhs(s)] \rangle$

Outline

- Motivation
- Incremental Search
- The D* Lite Algorithm
- **D* Lite Example**
- When to Use Incremental Path Planning?
- Algorithm Extensions and Related Topics
- Application to Mobile Robotics

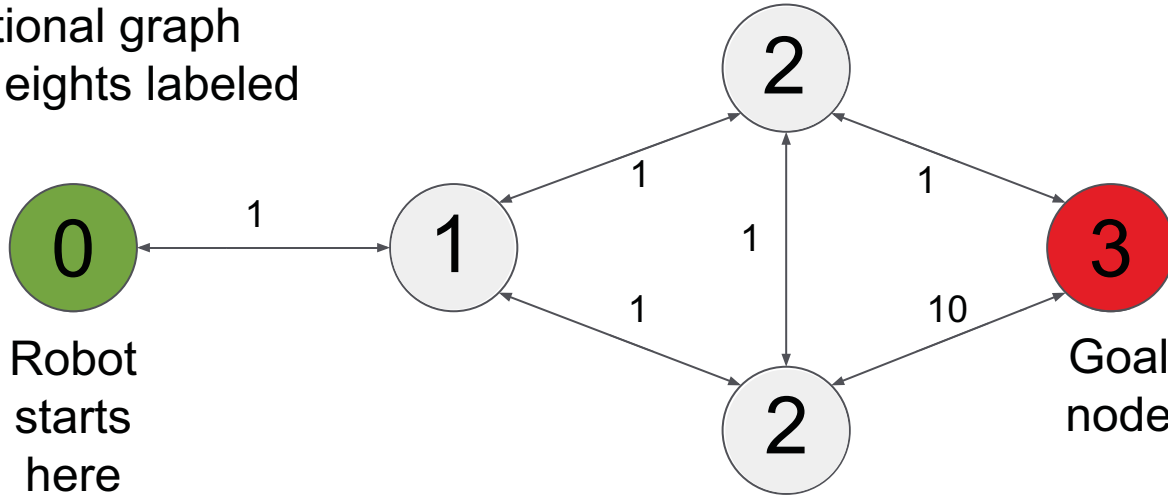
Overall D* Lite:

1. Initialize all $g(s) = \infty$, $\text{rhs}(s \neq s_{\text{goal}}) = \infty$, $\text{rhs}(s_{\text{goal}}) = 0$, $k_m = 0$, $s_{\text{last}} = s_{\text{start}}$.
2. Best-first search until s_{start} is locally consistent and expanded.
3. Move so $s_{\text{start}} = \text{argmin}_{s' \in \text{Succ}(s_{\text{start}})} (c(s_{\text{start}}, s') + g(s'))$.
4. If any edge costs change:
 - a. $k_m = k_m + h(s_{\text{last}}, s_{\text{start}})$.
 - b. Update rhs and queue position for source nodes of changed edges.
5. Repeat from 2.

Always sort by $\langle \min[g(s), \text{rhs}(s)] + h(s, s_{\text{start}}) + k_m, \min[g(s), \text{rhs}(s)] \rangle$

D* Lite Example

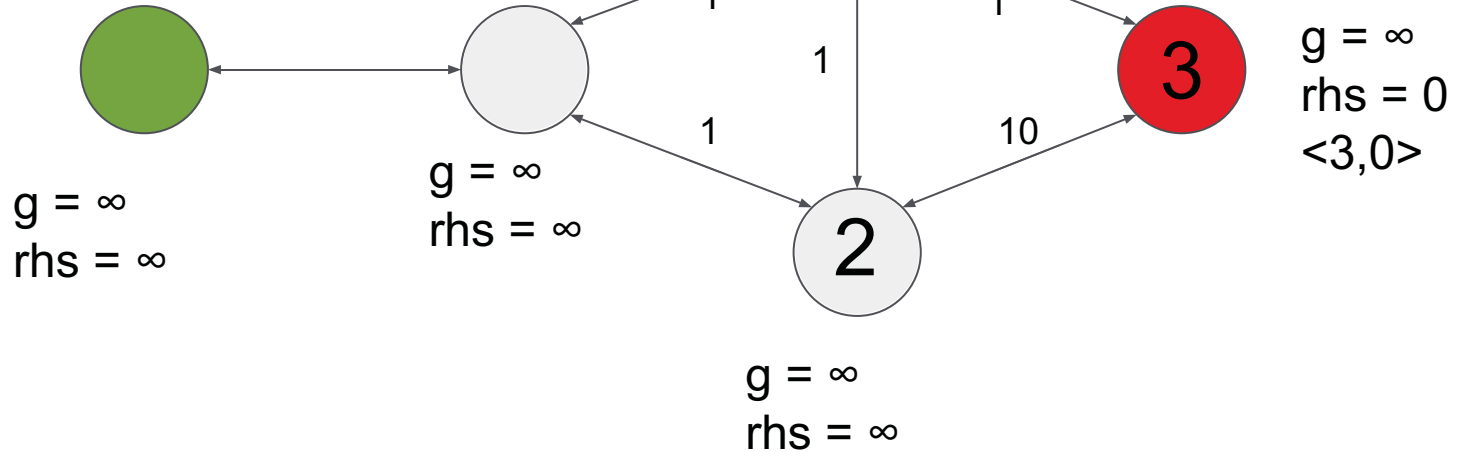
- Heuristic: $h(A, B) = \text{minimum number of nodes to get from A to B}$
 - $h(\text{node}, \text{start})$ written in each node
- Bidirectional graph
- Edge weights labeled



D* Lite Example

1. Initialize

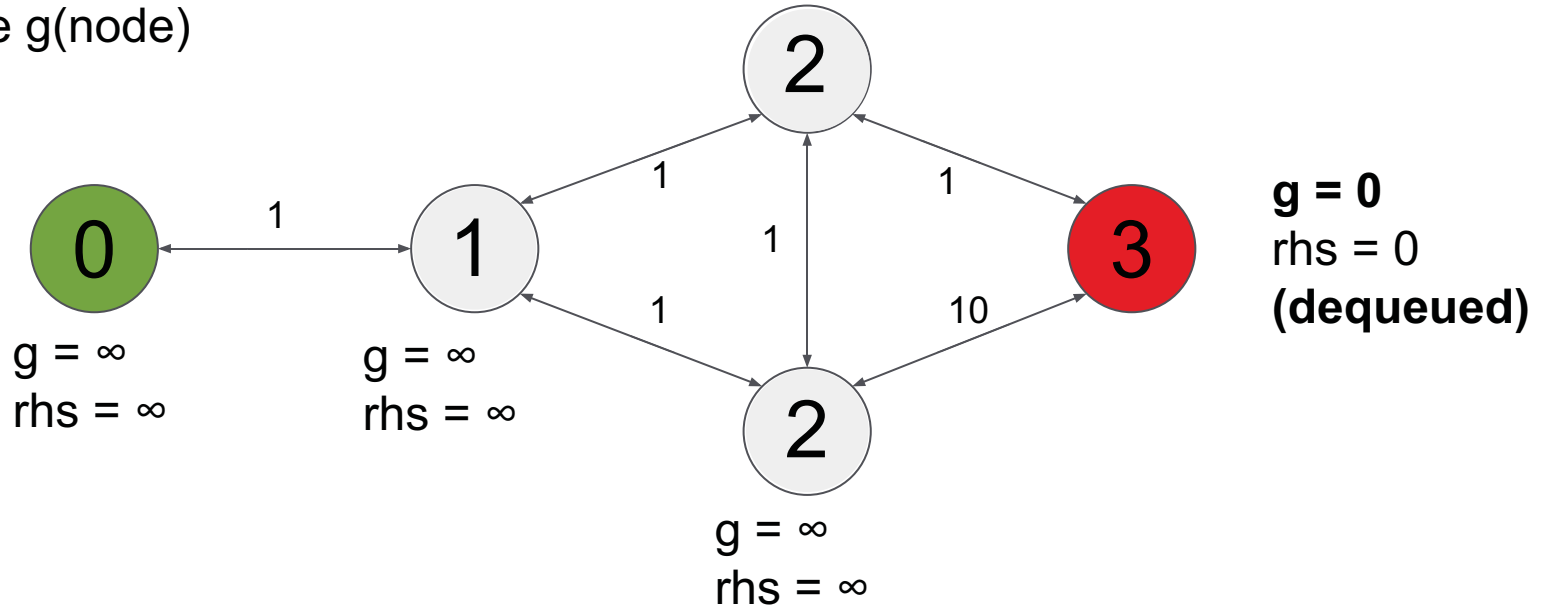
- all $g = rhs = \infty$, except goal
- add goal to queue with key $\langle 3, 0 \rangle$
- key modifier = 0



D* Lite Example

2. Plan initial path (A*)

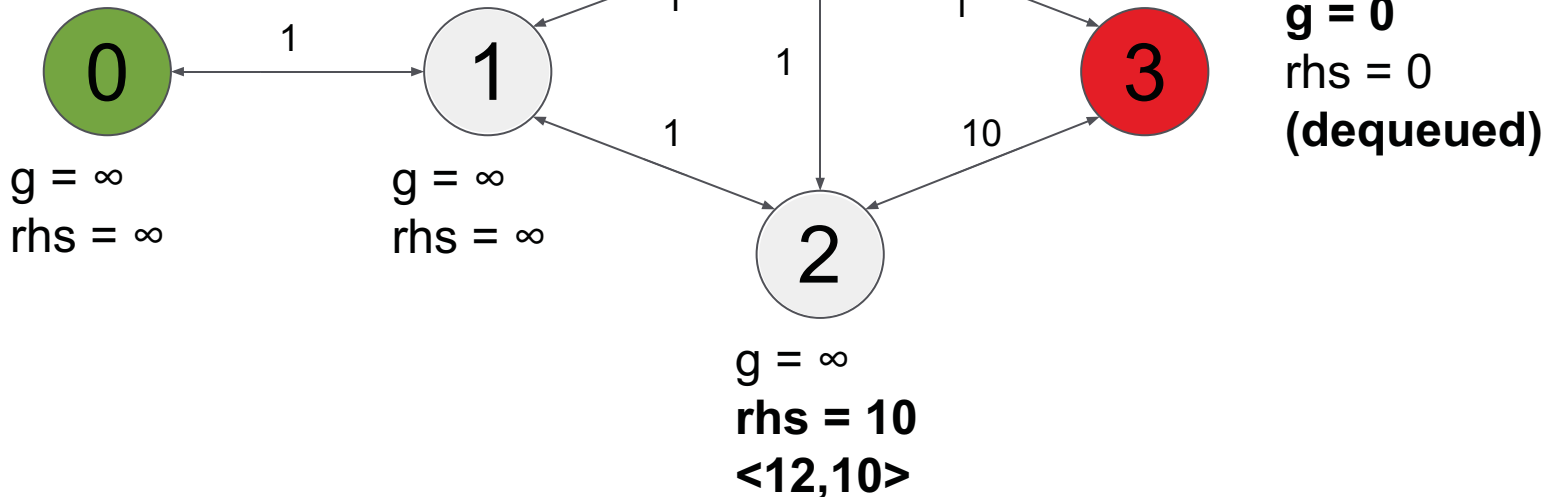
- dequeue node
- update $g(\text{node})$



D* Lite Example

2. Plan initial path (A*)

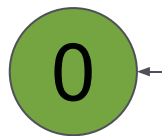
- dequeue node
- update $g(\text{node})$
- update $\text{rhs}(\text{neighbor})$
- queue inconsistent neighbors



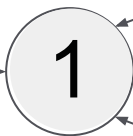
D* Lite Example

2. Plan initial path (A*)

- dequeue node
- update $g(\text{node})$
- update $\text{rhs}(\text{neighbor})$
- queue inconsistent neighbors

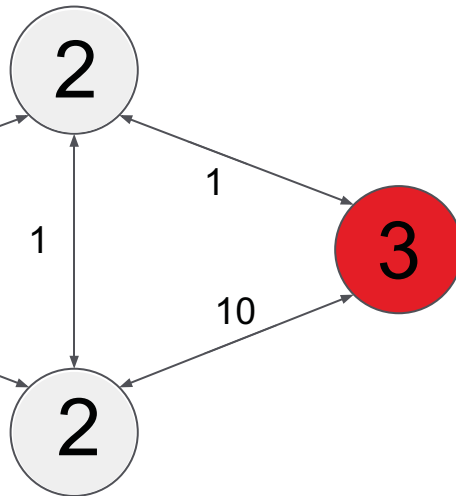


$g = \infty$
 $\text{rhs} = \infty$



$g = \infty$
 $\text{rhs} = \infty$

$g = 1$
 $\text{rhs} = 1$
(dequeued)



$g = \infty$
 $\text{rhs} = 10$
 $\langle 12, 10 \rangle$

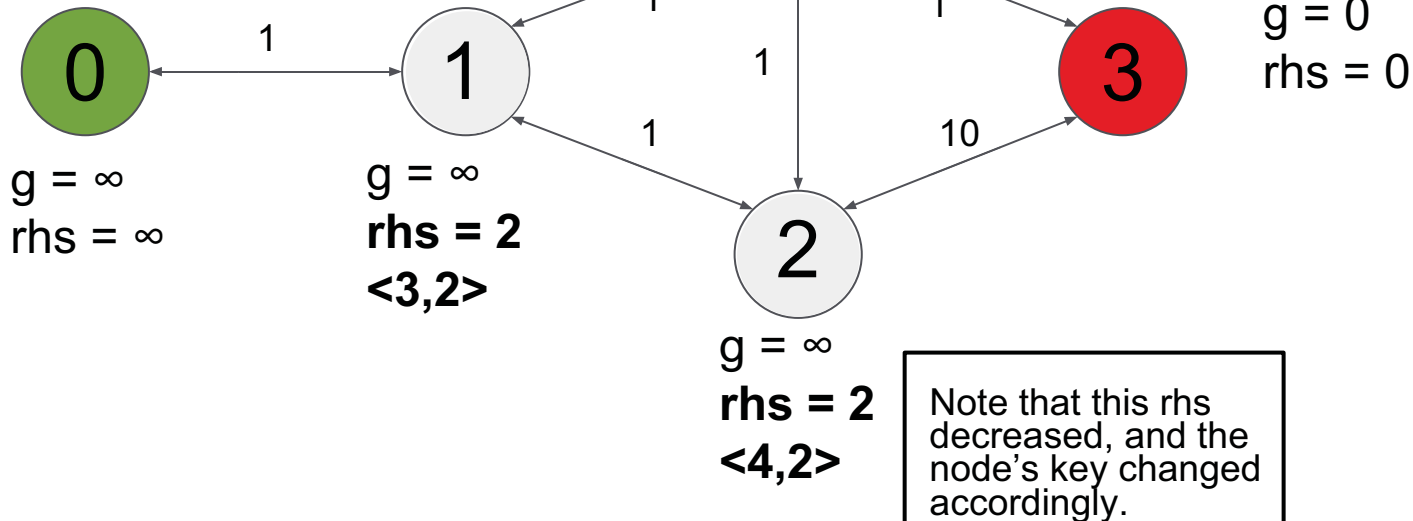
$k_m = 0$

$g = 0$
 $\text{rhs} = 0$

D* Lite Example

2. Plan initial path (A*)

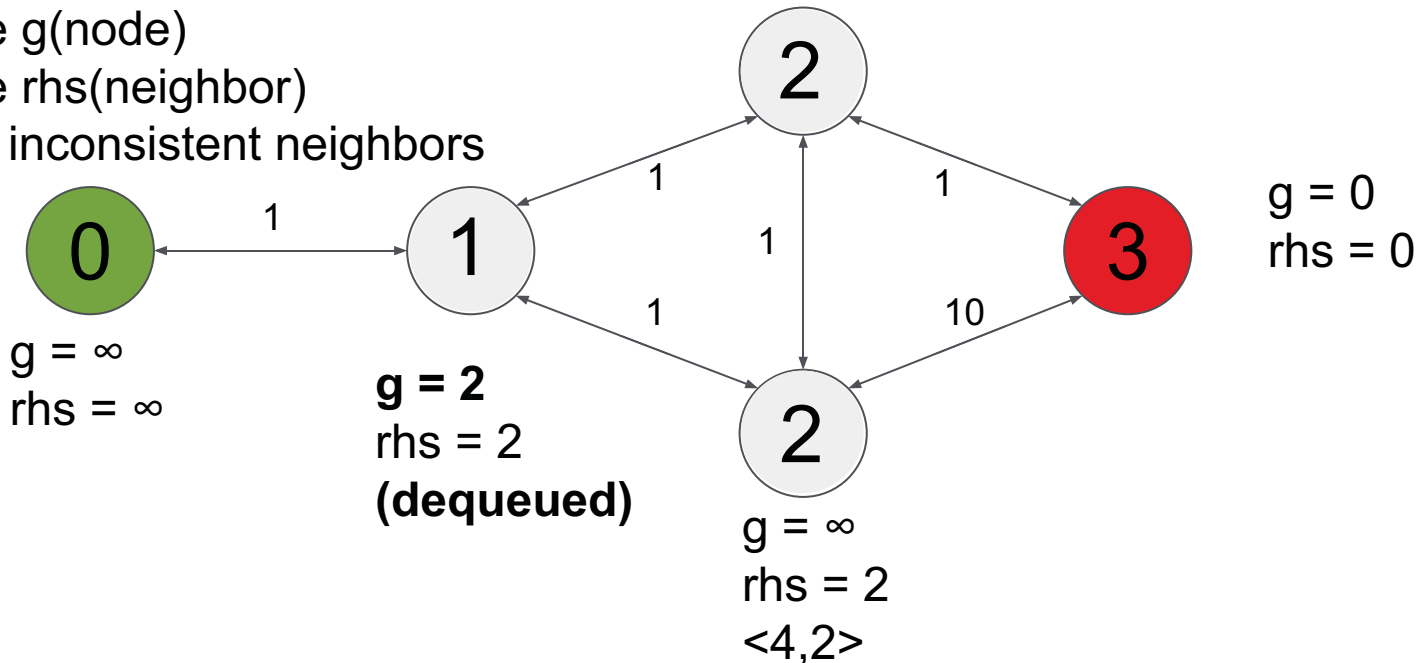
- dequeue node
- update $g(\text{node})$
- update $\text{rhs}(\text{neighbor})$
- queue inconsistent neighbors



D* Lite Example

2. Plan initial path (A*)

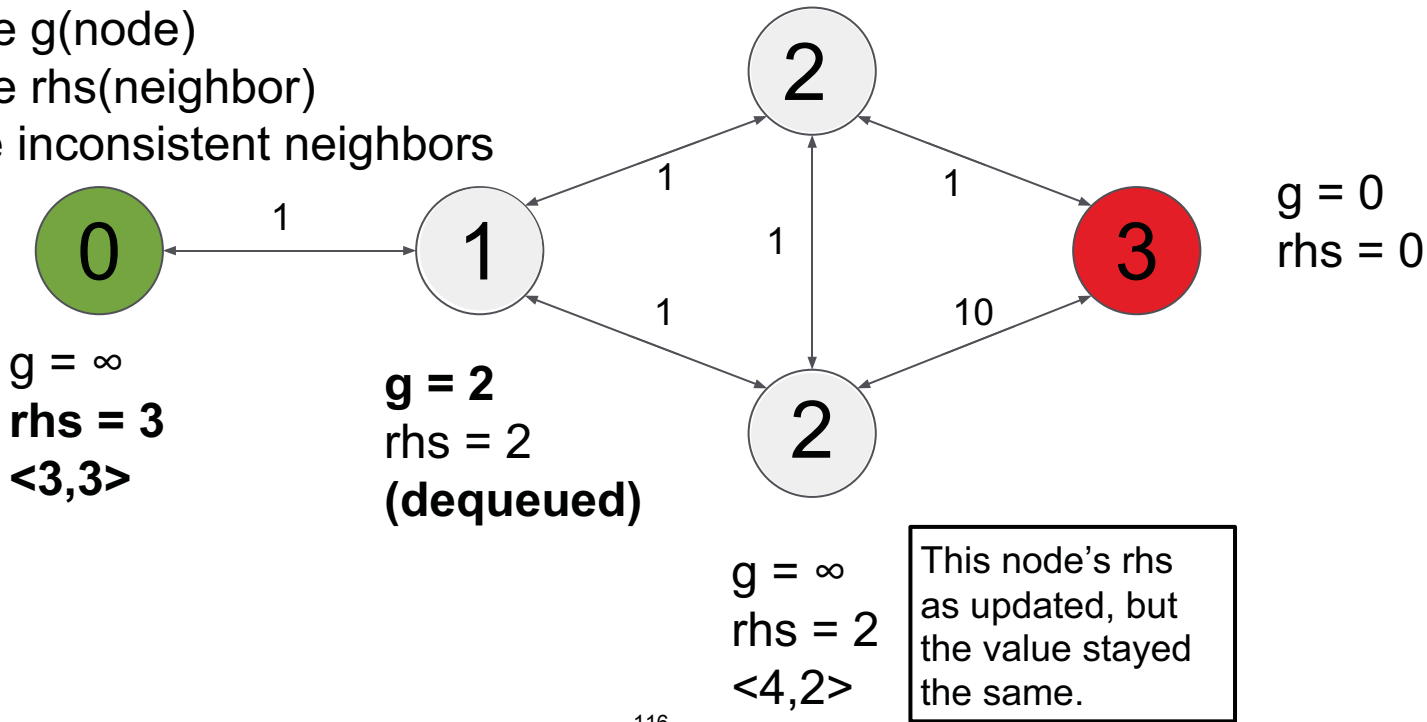
- dequeue node
- update $g(\text{node})$
- update $\text{rhs}(\text{neighbor})$
- queue inconsistent neighbors



D* Lite Example

2. Plan initial path (A*)

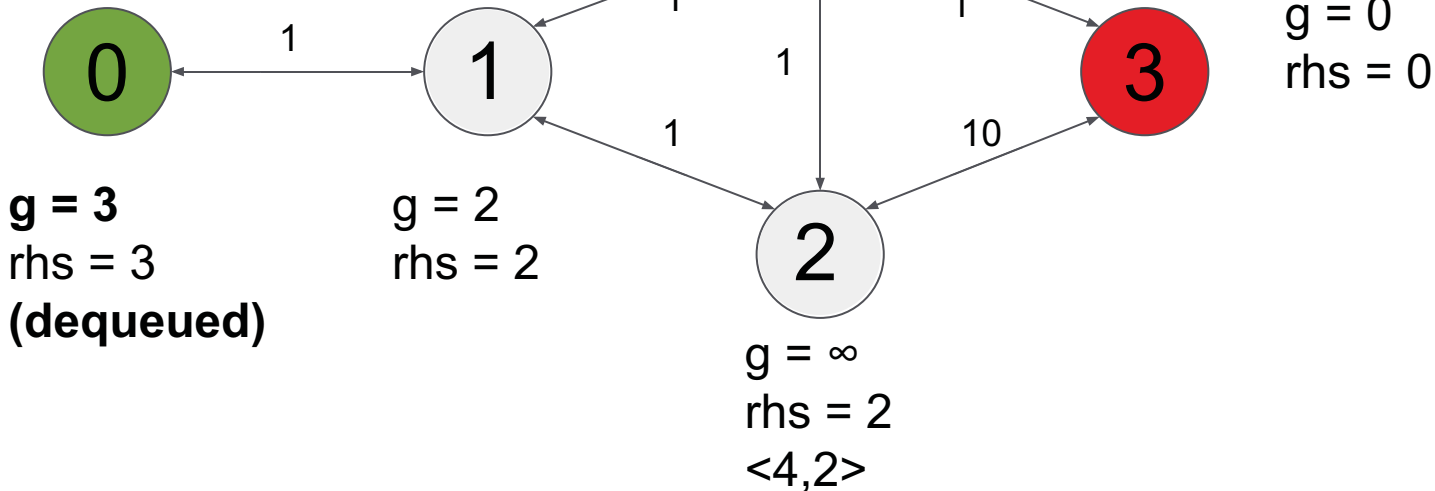
- dequeue node
- update $g(\text{node})$
- update $\text{rhs}(\text{neighbor})$
- queue inconsistent neighbors



D* Lite Example

2. Plan initial path (A*)

- dequeue node
- update $g(\text{node})$
- update $\text{rhs}(\text{neighbor})$
- queue inconsistent neighbors

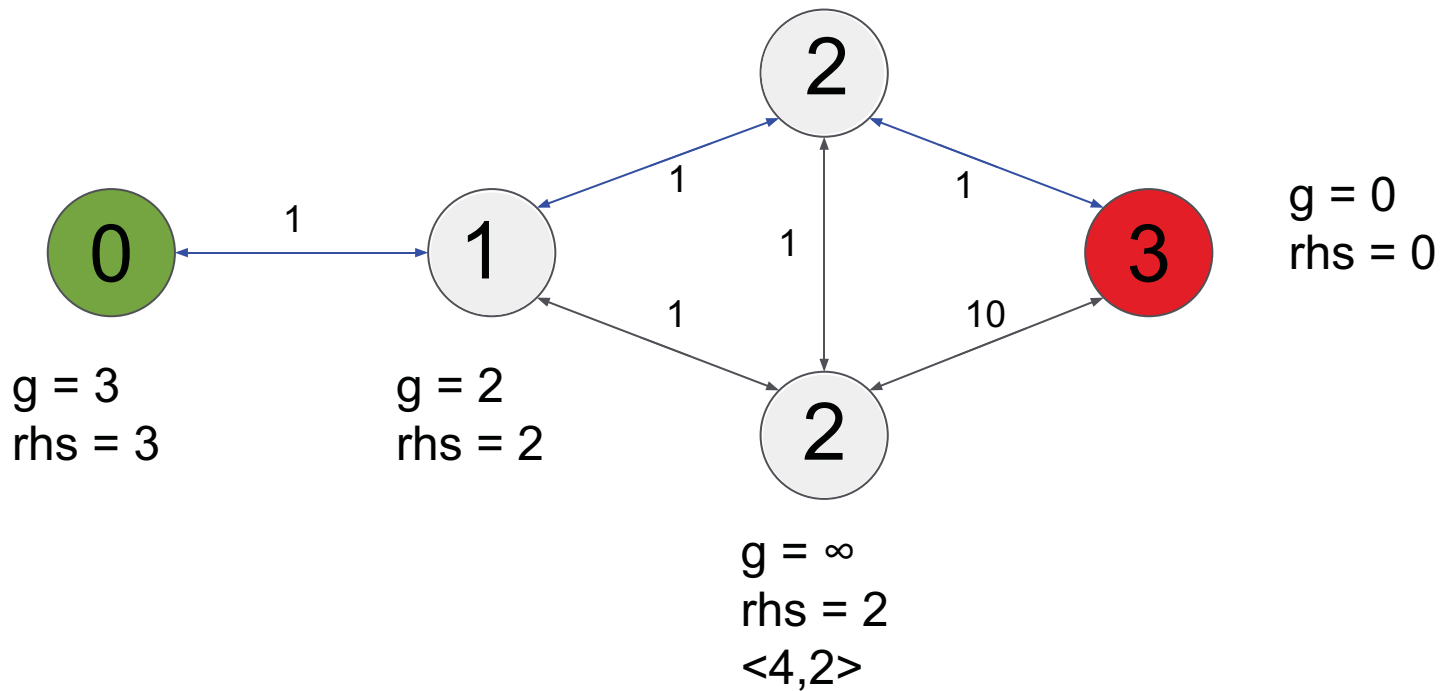


D* Lite Example

Found shortest path to goal!
(edges in blue)

$g = 1$
 $rhs = 1$

$k_m = 0$



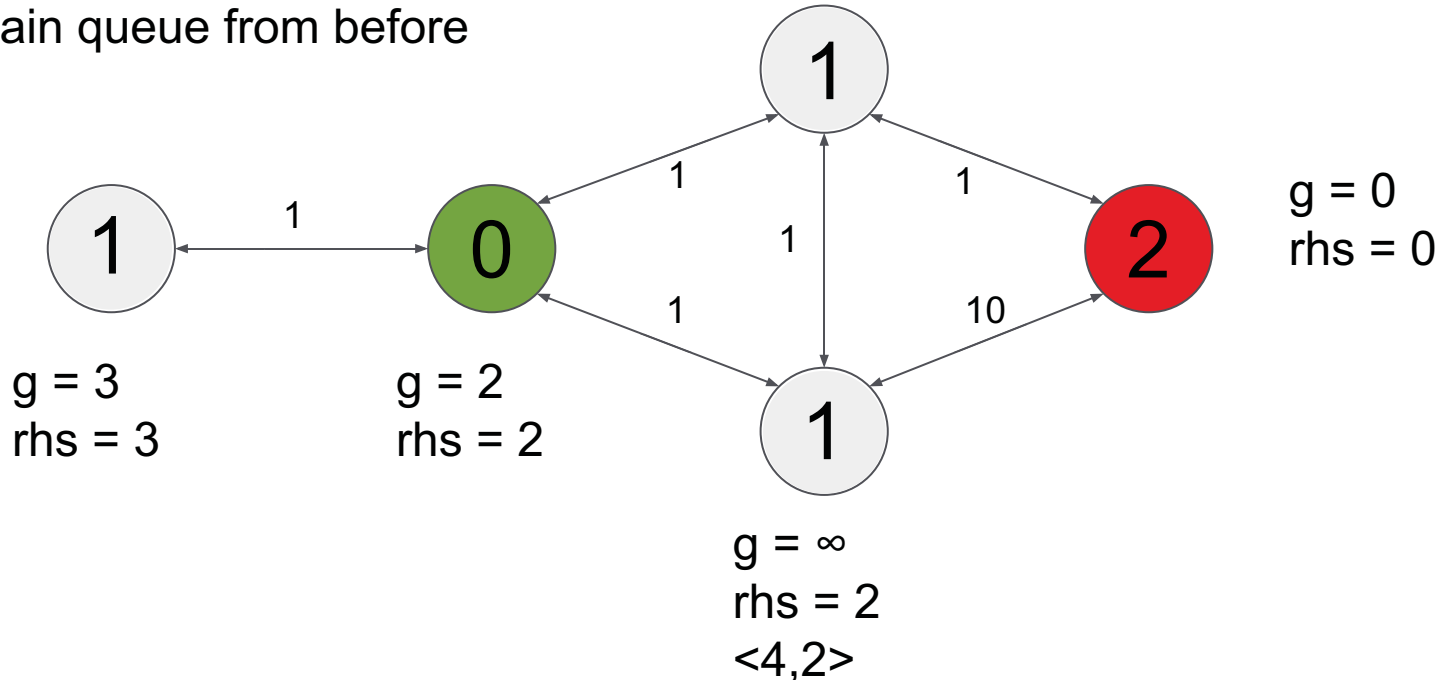
D* Lite Example

3. Robot moves (green)

- update heuristics for new start node
- maintain queue from before

$g = 1$
 $rhs = 1$

$k_m = 0$



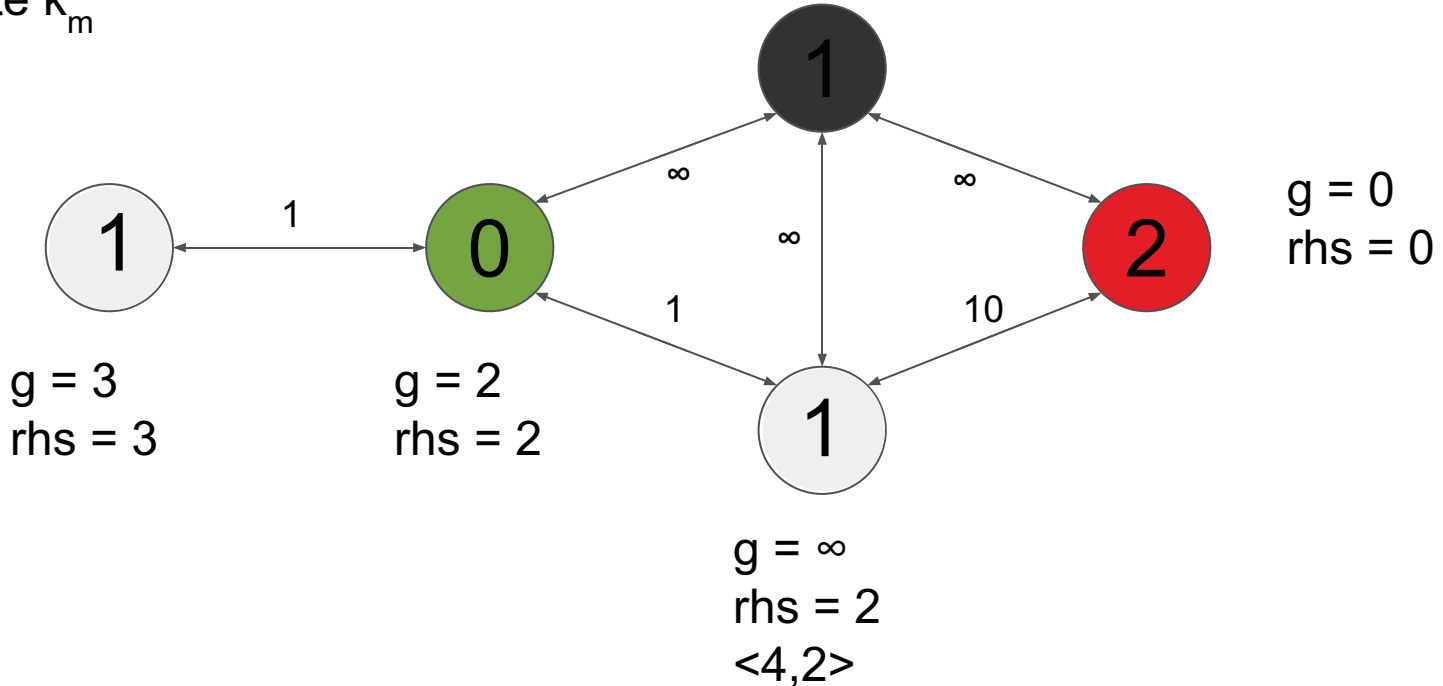
D* Lite Example

4. Obstacle detected (dark gray)

- adjacent edge weights = ∞
- update k_m

$g = 1$
 $rhs = 1$

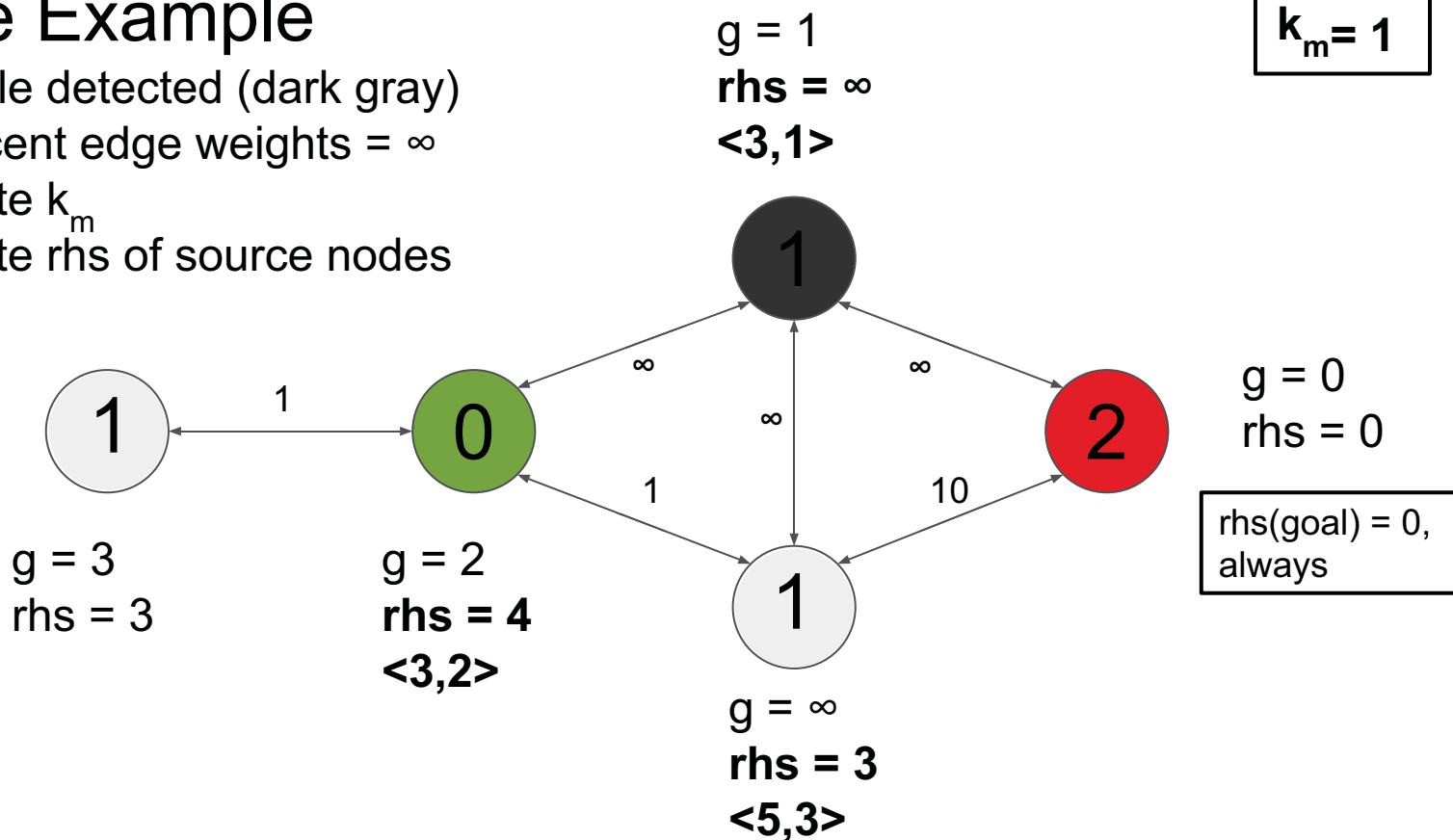
$k_m = 1$



D* Lite Example

4. Obstacle detected (dark gray)

- adjacent edge weights = ∞
- update k_m
- update rhs of source nodes



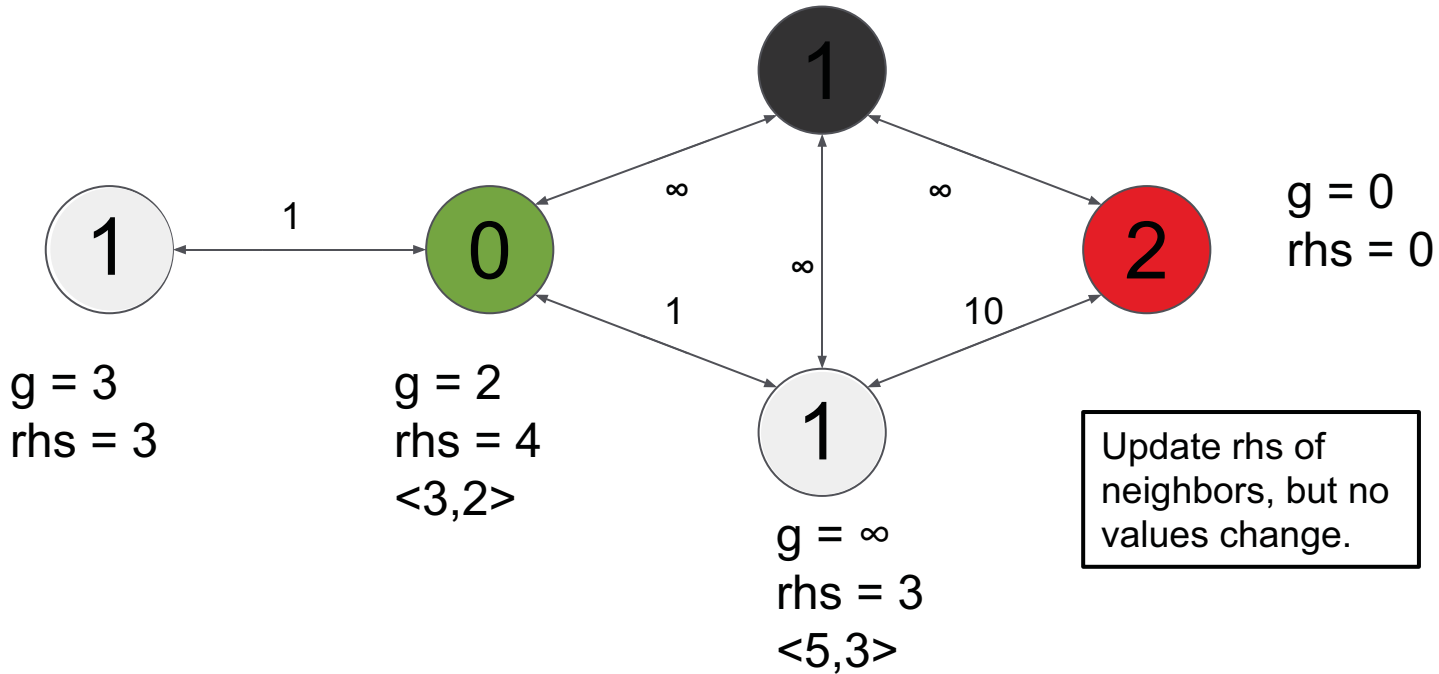
D* Lite Example

- 5. Repeat from 2
- 2. Replan path

$g = \infty$
 $rhs = \infty$
(dequeued)

Underconsistent!

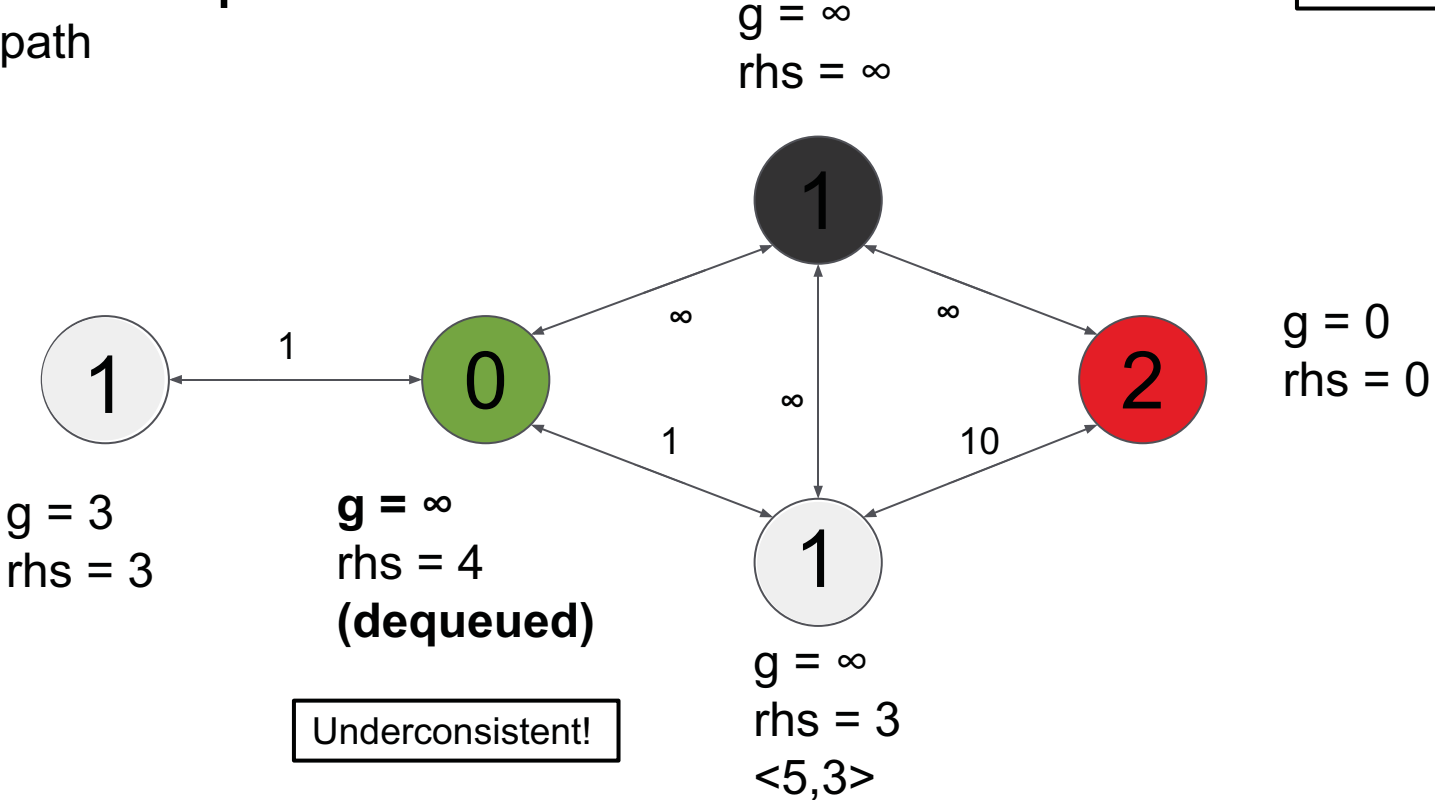
$k_m = 1$



D* Lite Example

2. Replan path

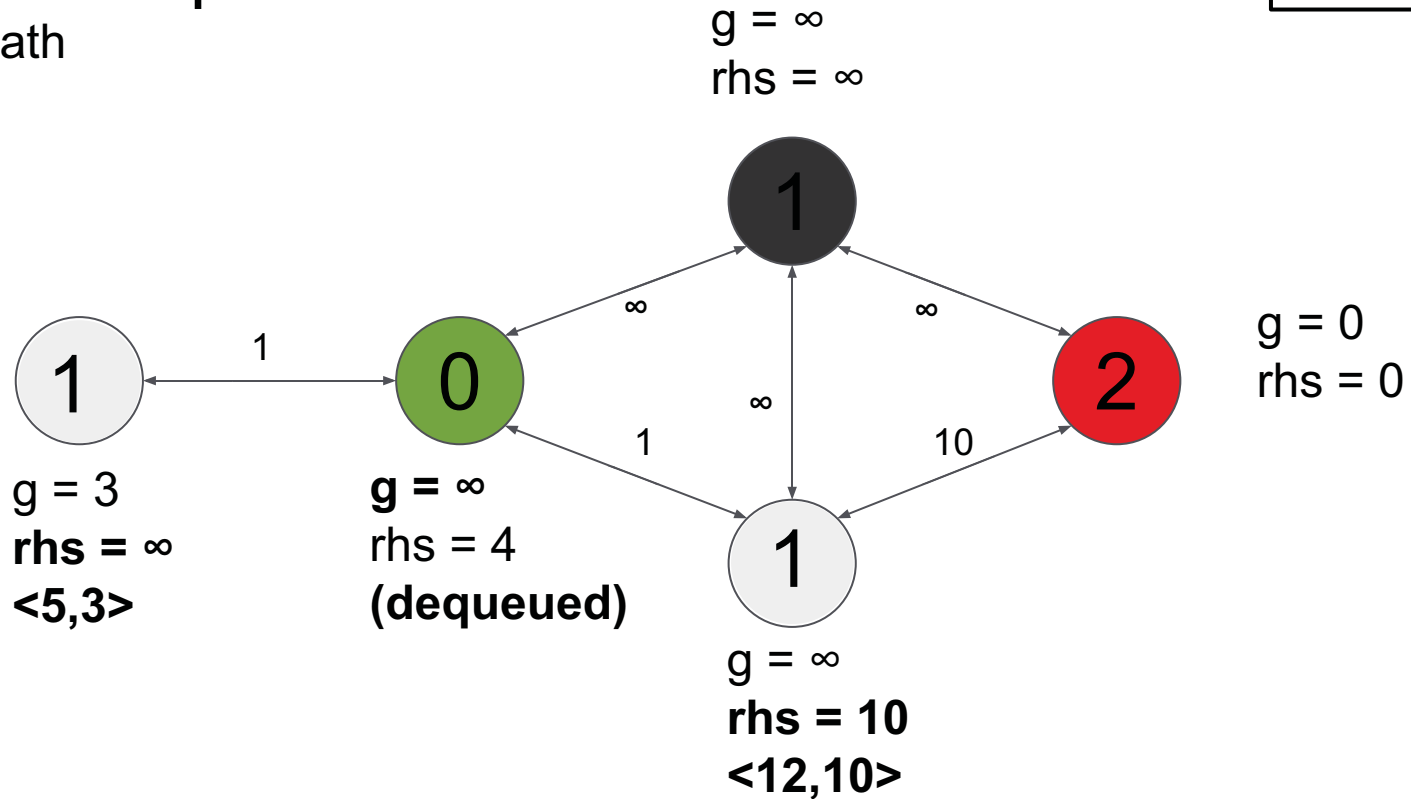
$k_m = 1$



D* Lite Example

2. Replan path

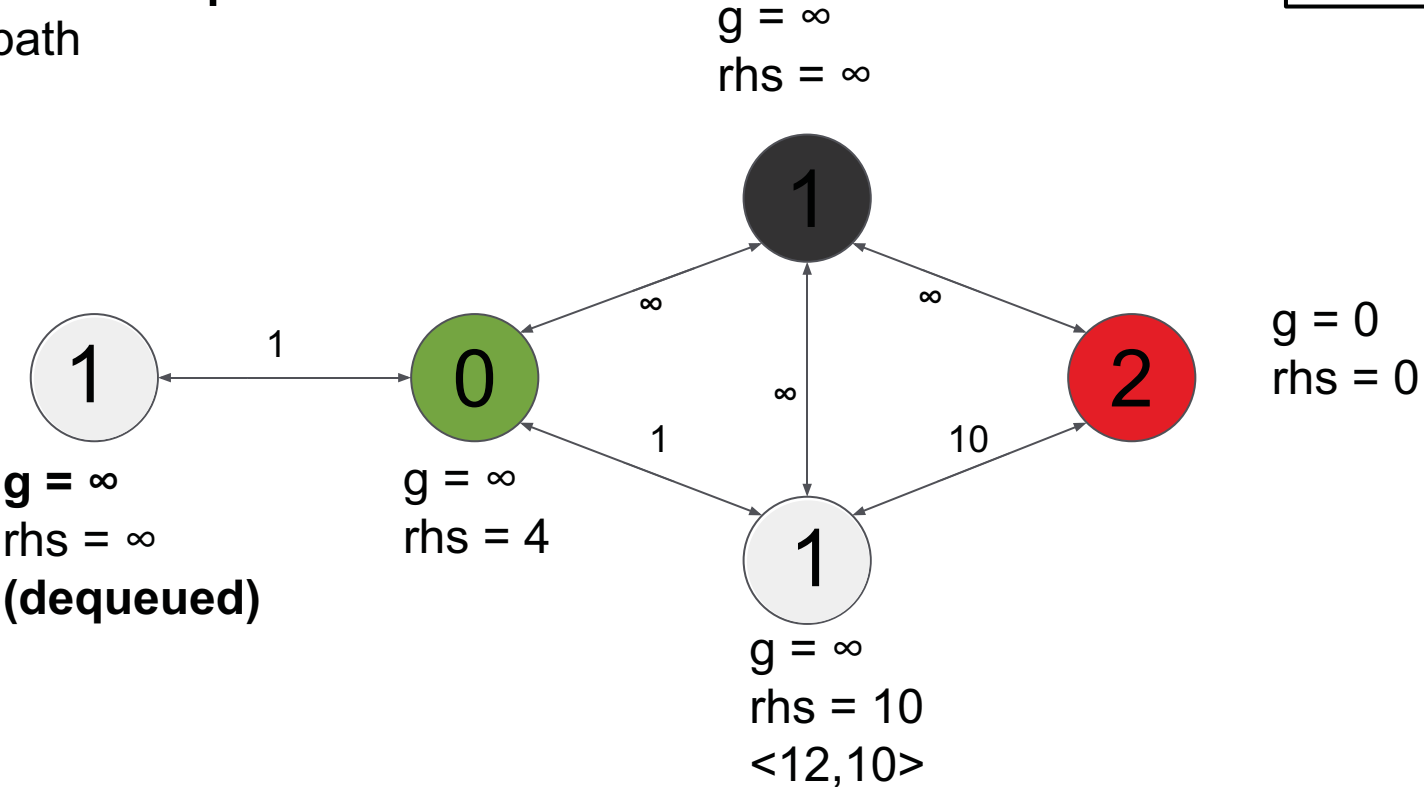
$k_m = 1$



D* Lite Example

2. Replan path

$k_m = 1$

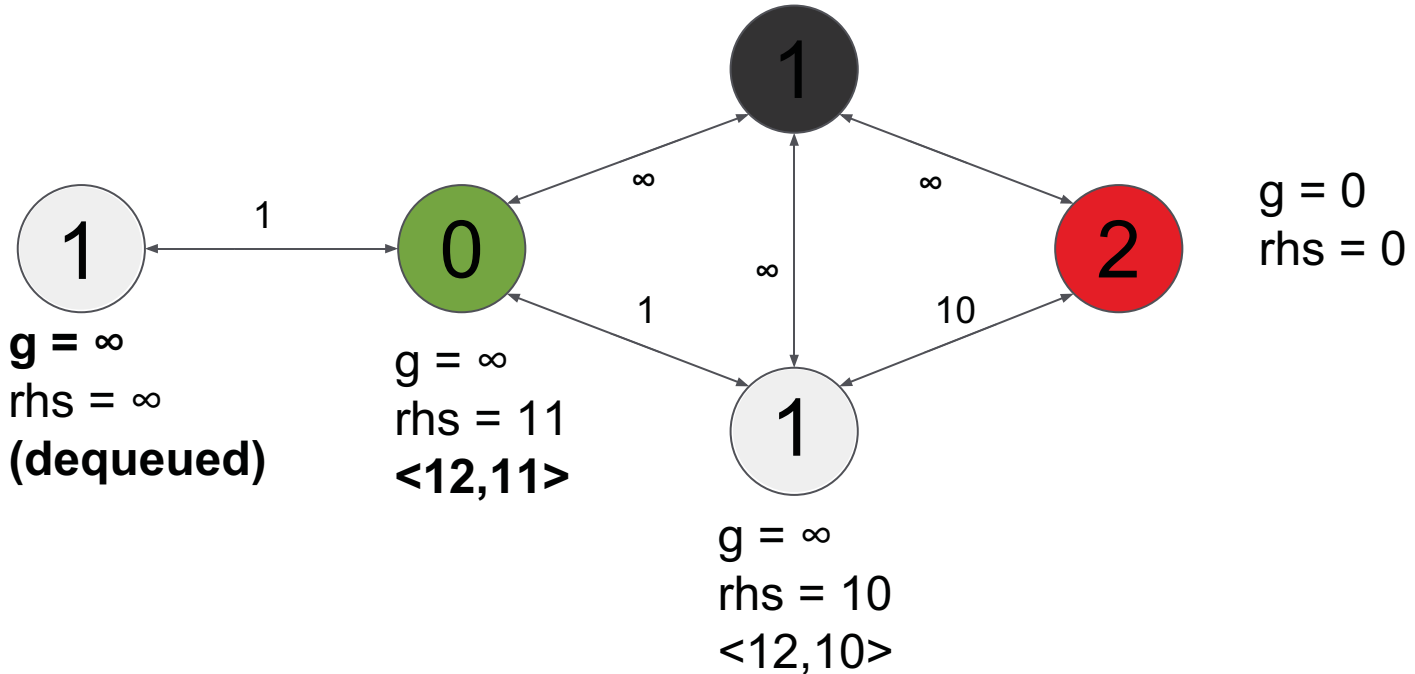


D* Lite Example

2. Replan path

$$k_m = 1$$

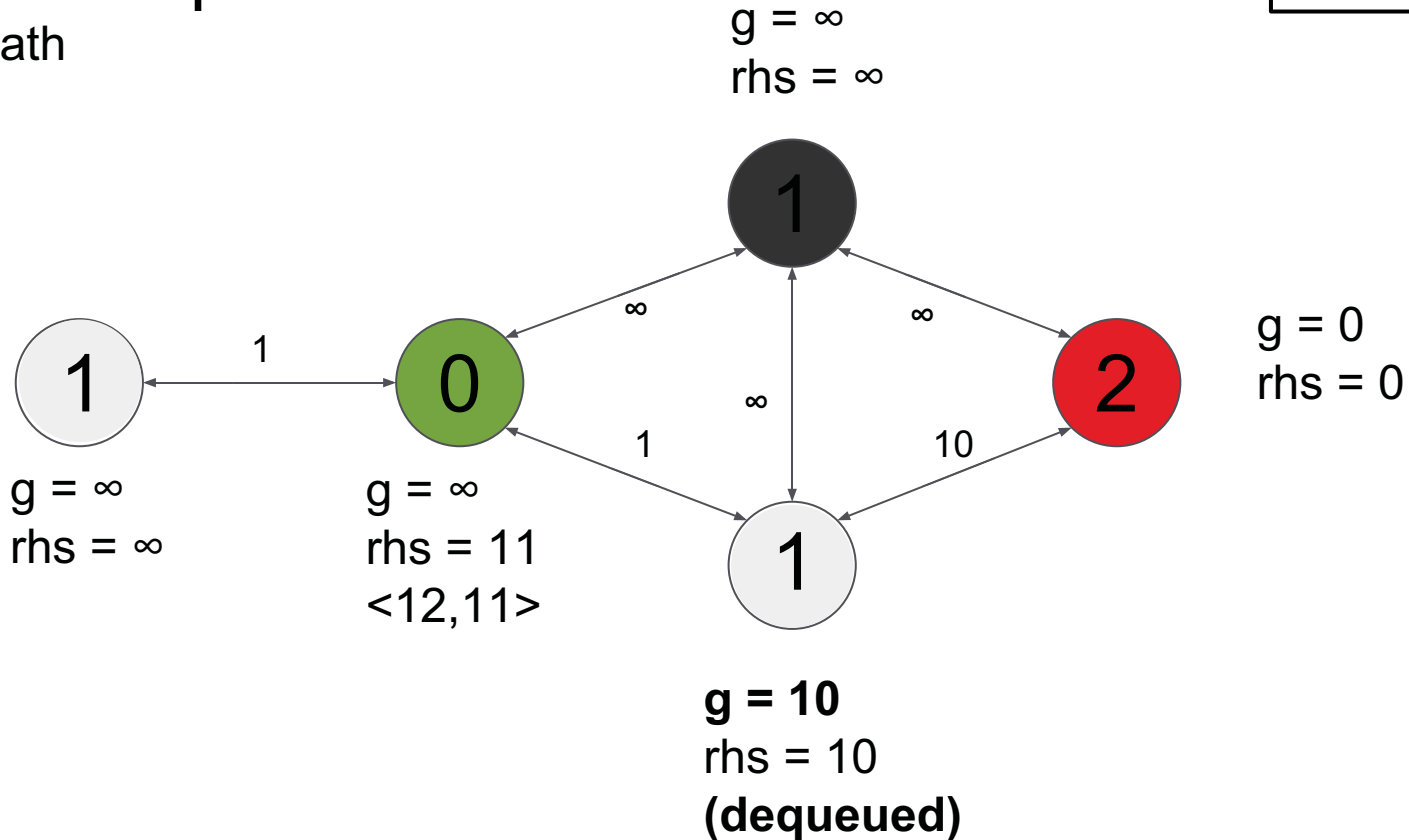
$$g = \infty$$
$$rhs = \infty$$



D* Lite Example

2. Replan path

$$k_m = 1$$

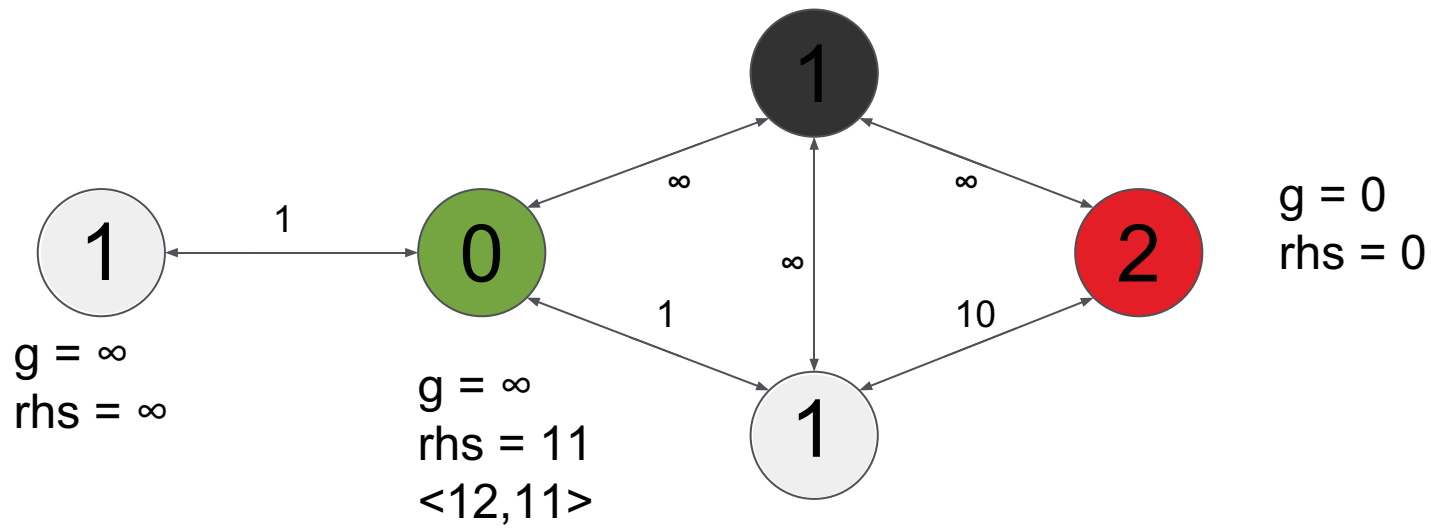


D* Lite Example

2. Replan path

$k_m = 1$

$g = \infty$
 $rhs = \infty$



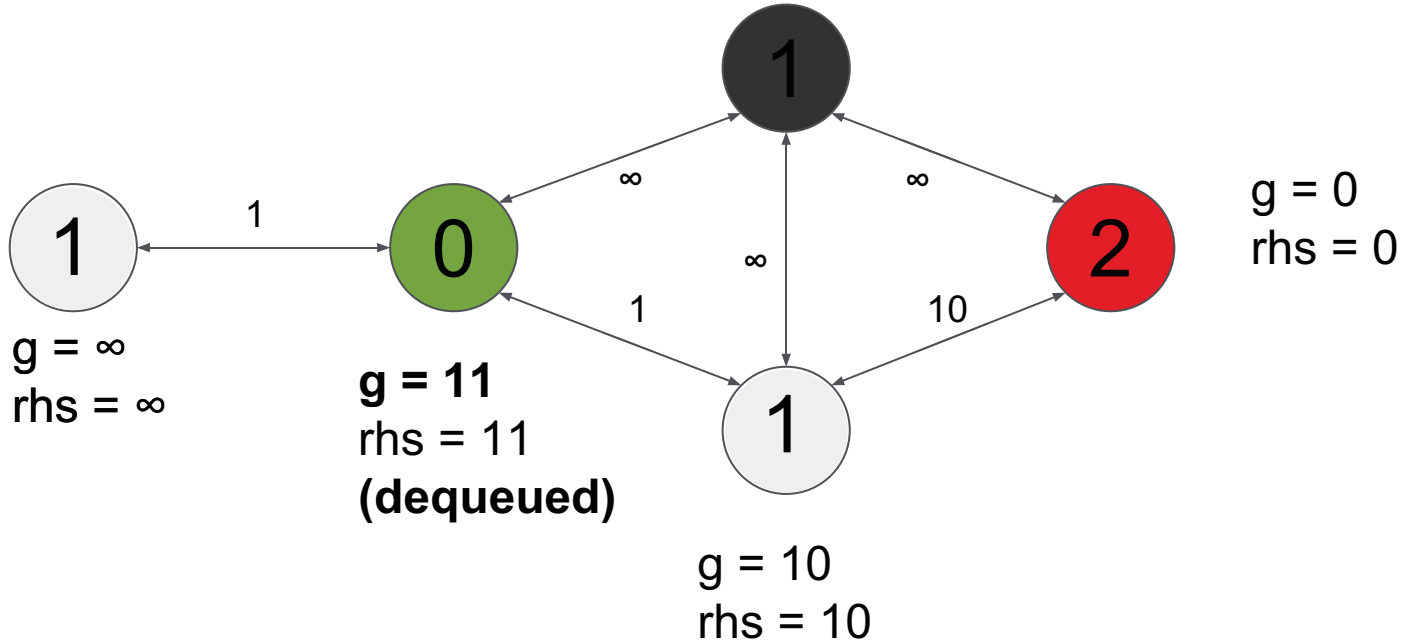
This node's rhs was updated, but the value stayed the same.

D* Lite Example

2. Replan path

$g = \infty$
 $rhs = \infty$

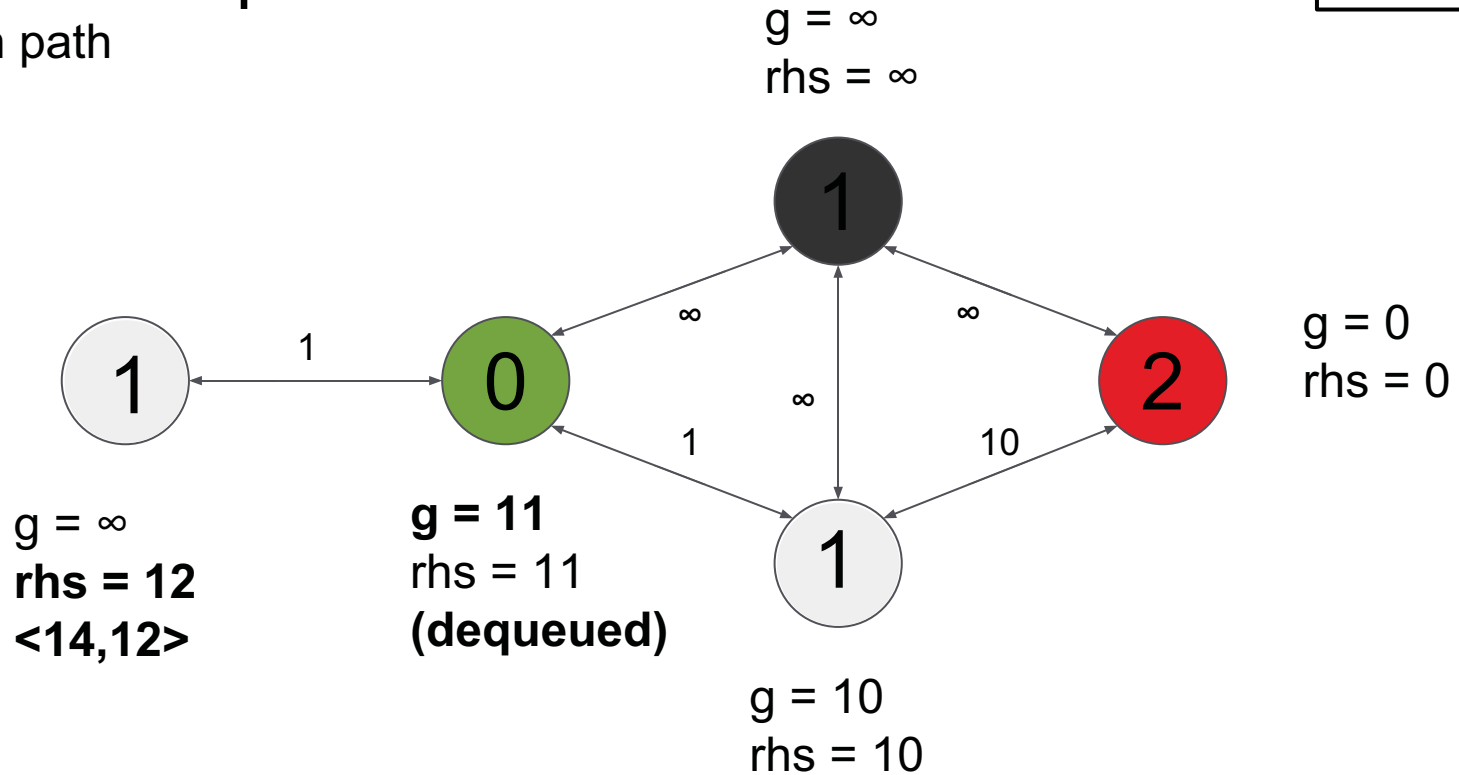
$k_m = 1$



D* Lite Example

2. Replan path

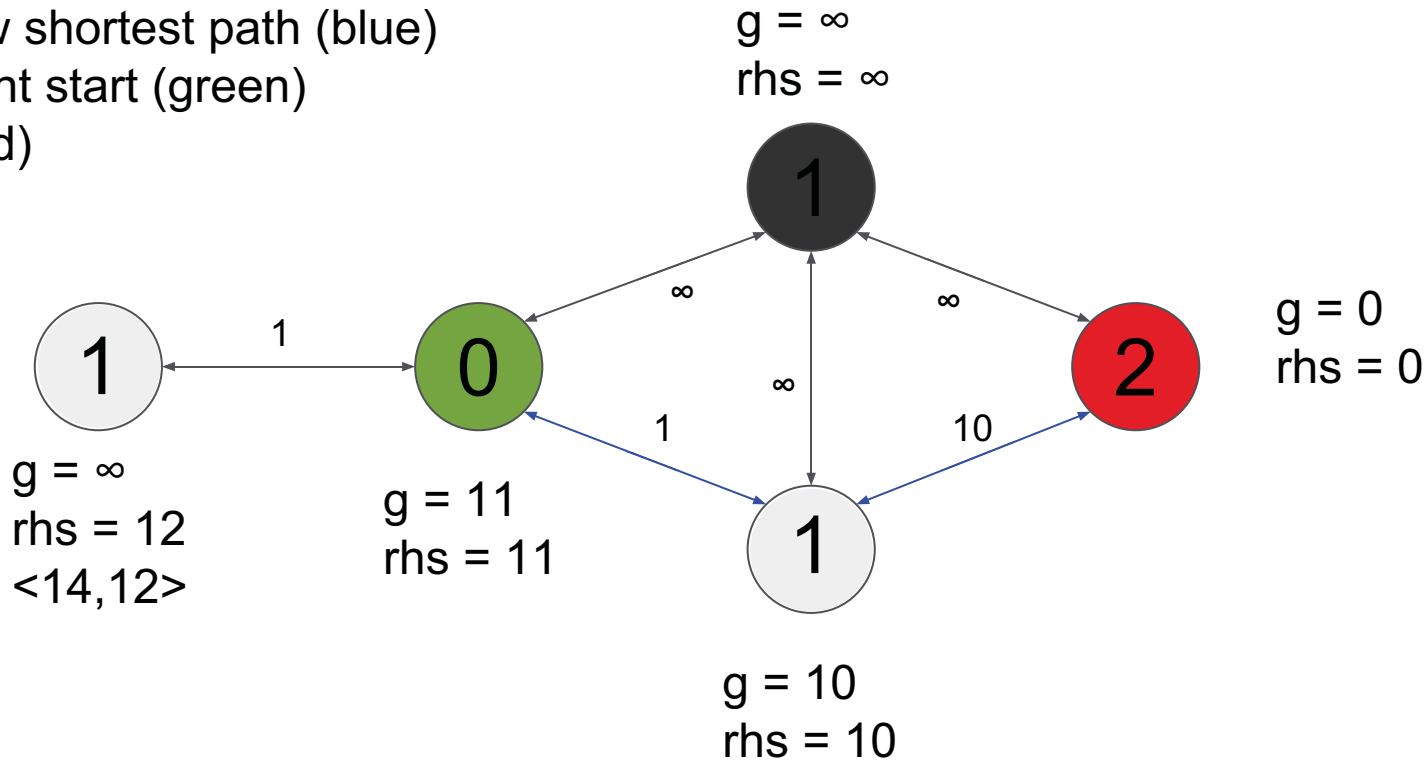
$k_m = 1$



D* Lite Example

Found new shortest path (blue)
from current start (green)
to goal (red)

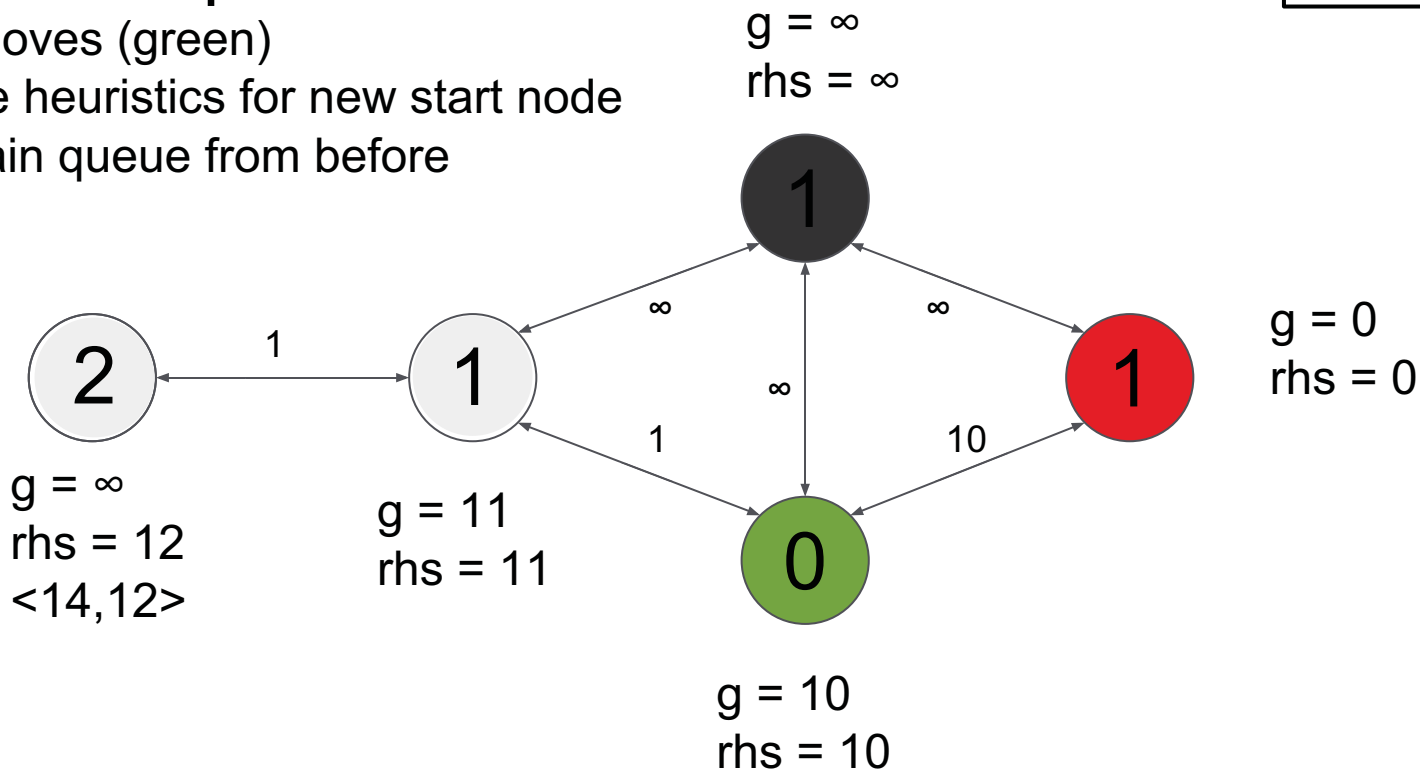
$$k_m = 1$$



D* Lite Example

3. Robot moves (green)

- update heuristics for new start node
- maintain queue from before

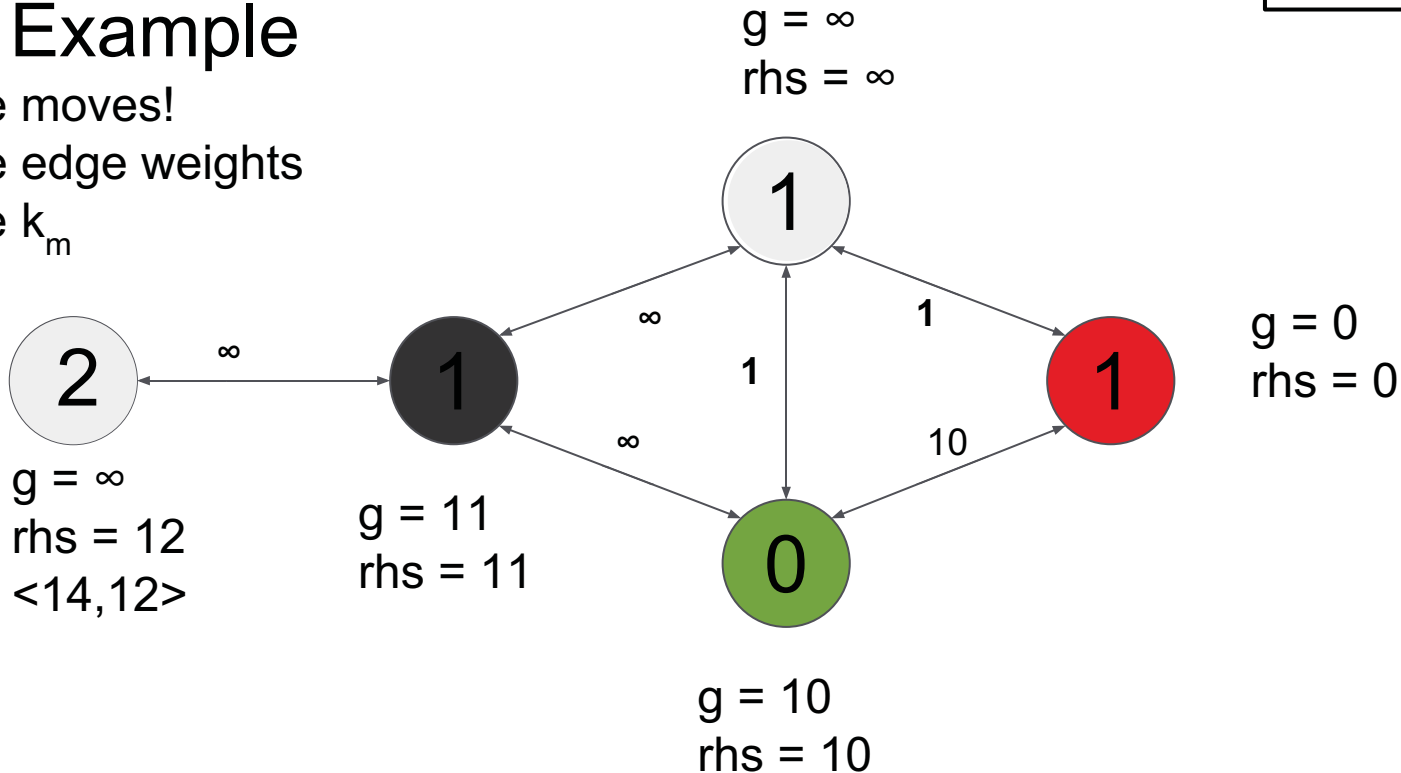


$$k_m = 2$$

D* Lite Example

4. Obstacle moves!

- update edge weights
- update k_m

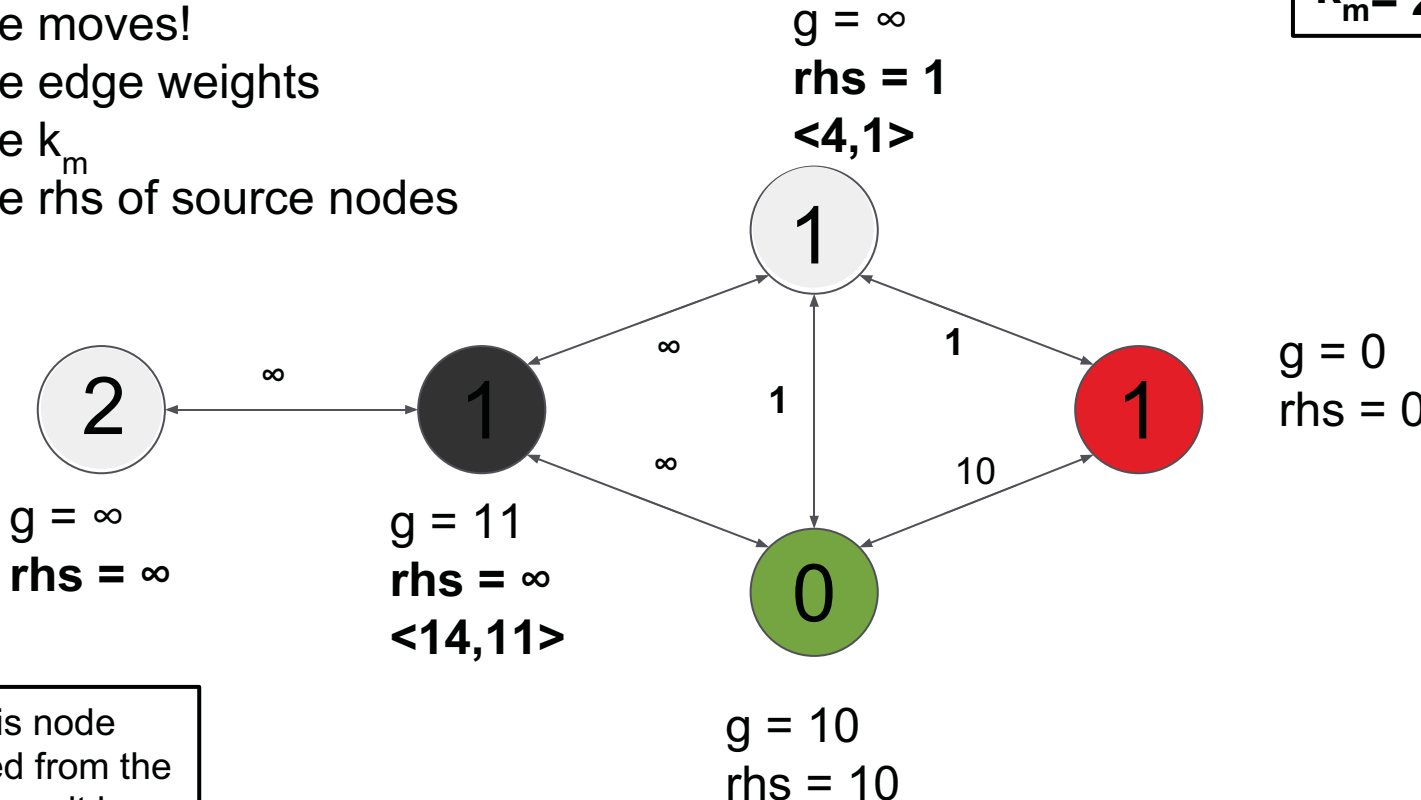


D* Lite Example

4. Obstacle moves!

- update edge weights
- update k_m
- update rhs of source nodes

$$k_m = 2$$



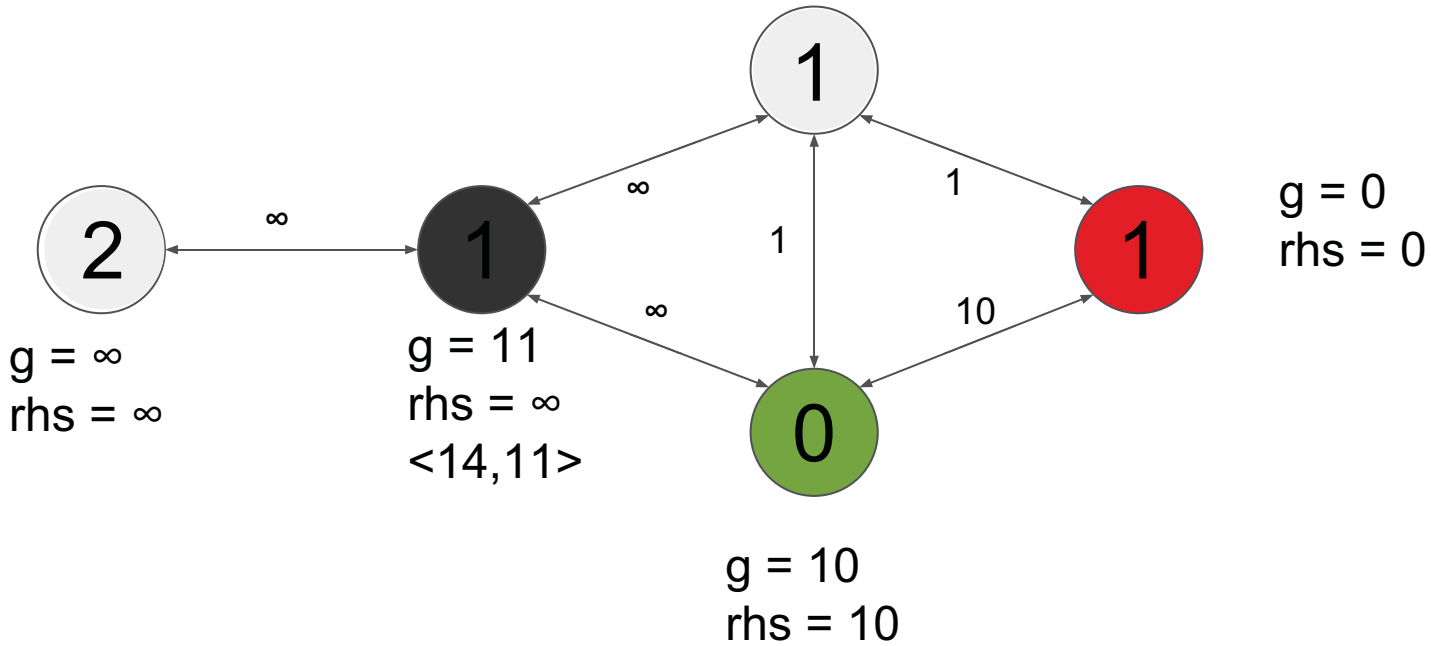
Note that this node was removed from the queue because it is now locally consistent

D* Lite Example

5. Repeat from 2
2. Replan path

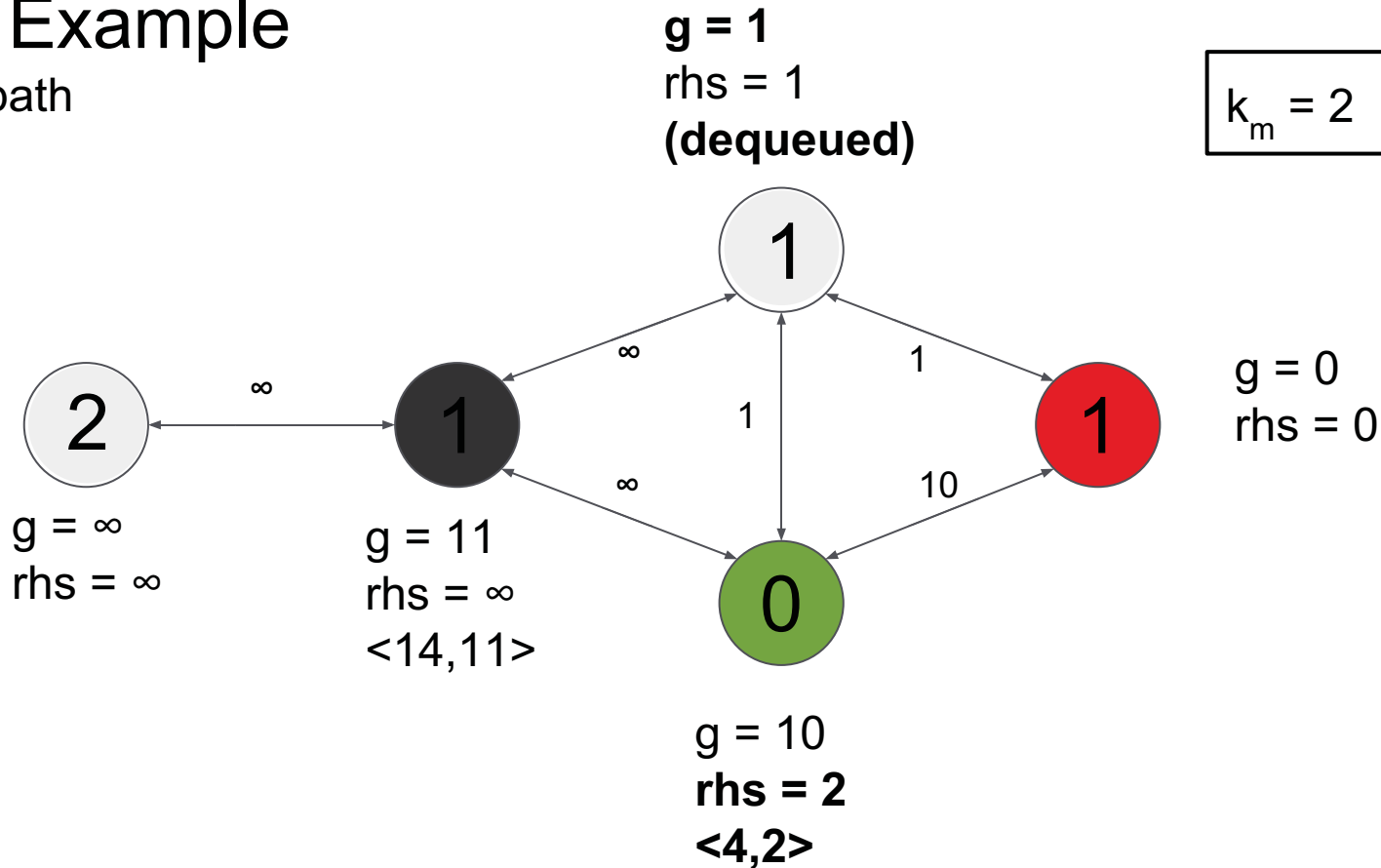
$g = 1$
 $rhs = 1$
(dequeued)

$k_m = 2$



D* Lite Example

2. Replan path

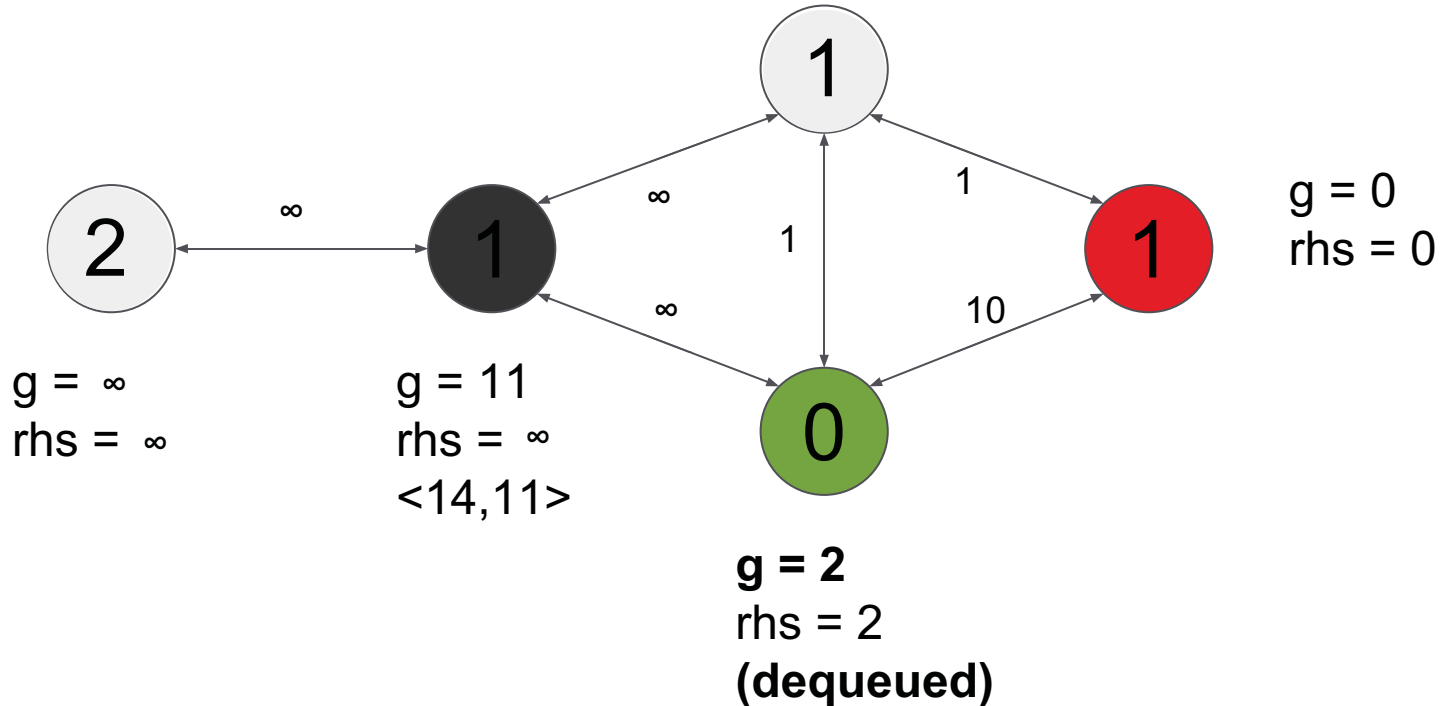


D* Lite Example

2. Replan path

$g = 1$
 $rhs = 1$

$k_m = 2$

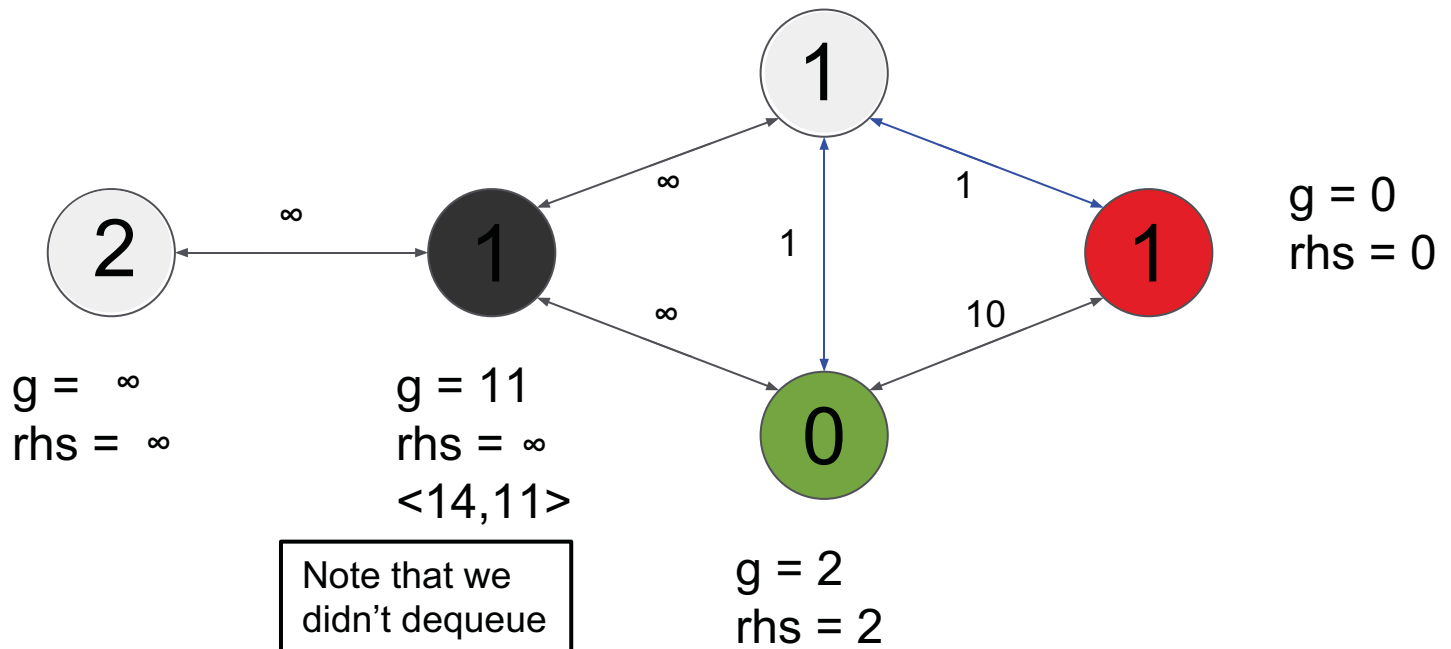


D* Lite Example

Found new shortest path!

$g = 1$
 $rhs = 1$

$k_m = 2$



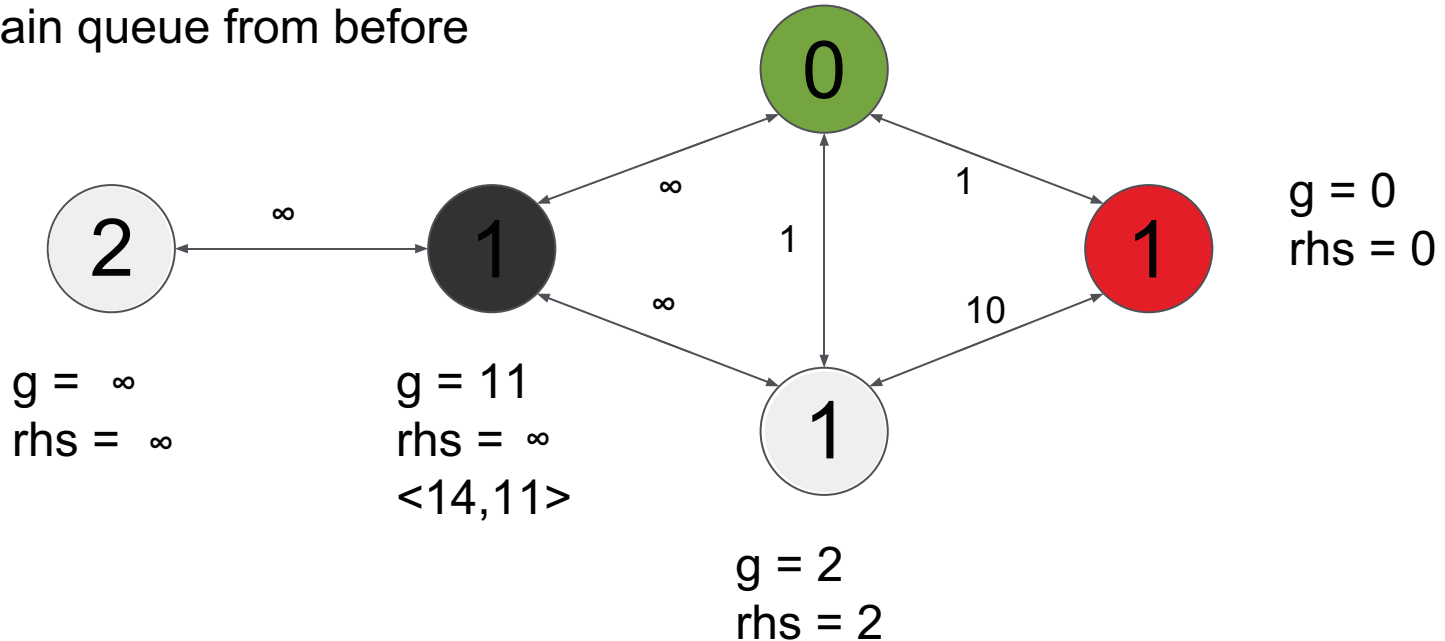
D* Lite Example

3. Robot moves

- update heuristics for new start node
- maintain queue from before

$g = 1$
 $rhs = 1$

$k_m = 2$



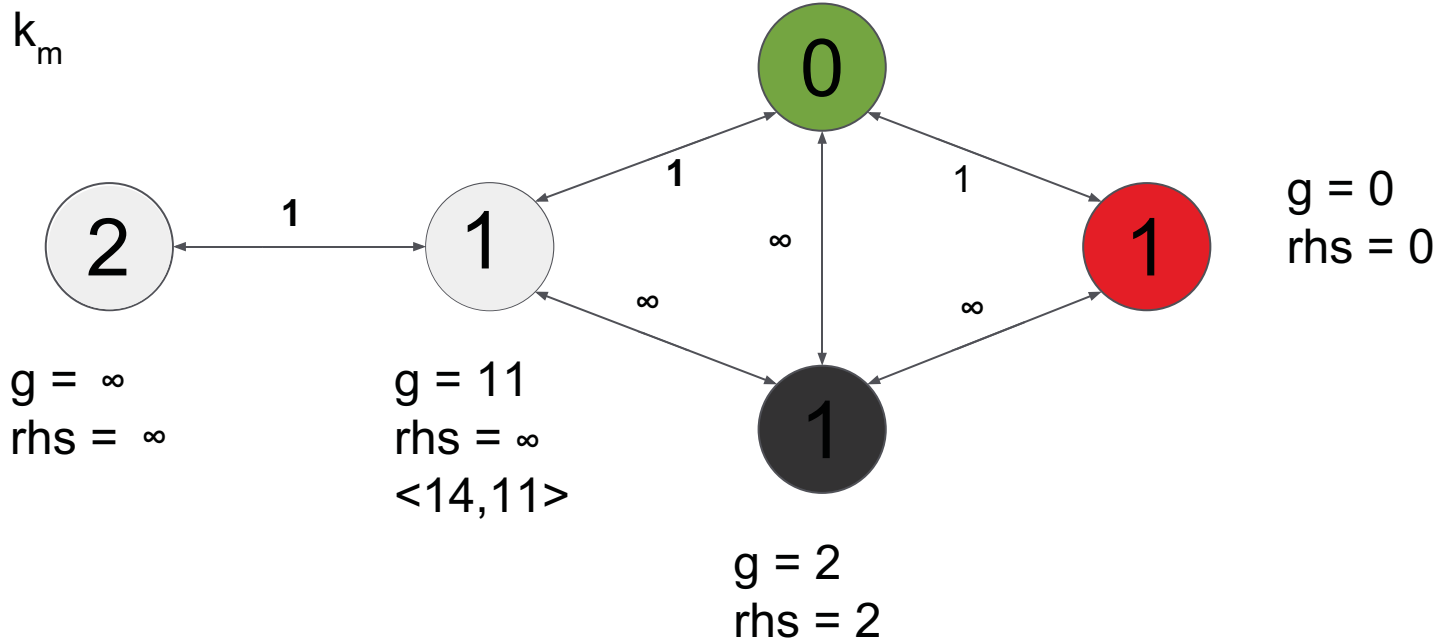
D* Lite Example

4. Obstacle moves, chasing robot!

- update edge weights
- update k_m

$g = 1$
 $rhs = 1$

$k_m = 3$



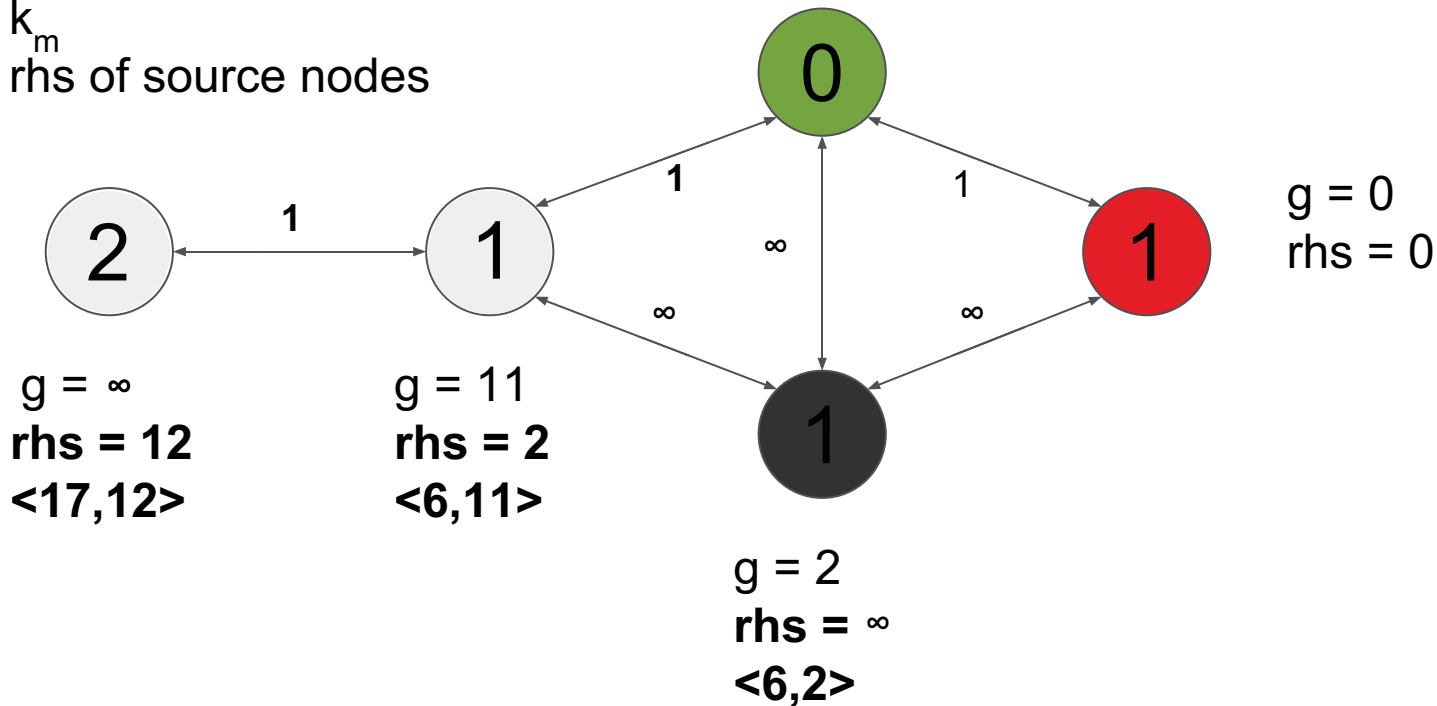
D* Lite Example

4. Obstacle moves, chasing robot!

- update edge weights
- update k_m
- update rhs of source nodes

$g = 1$
 $rhs = 1$

$k_m = 3$

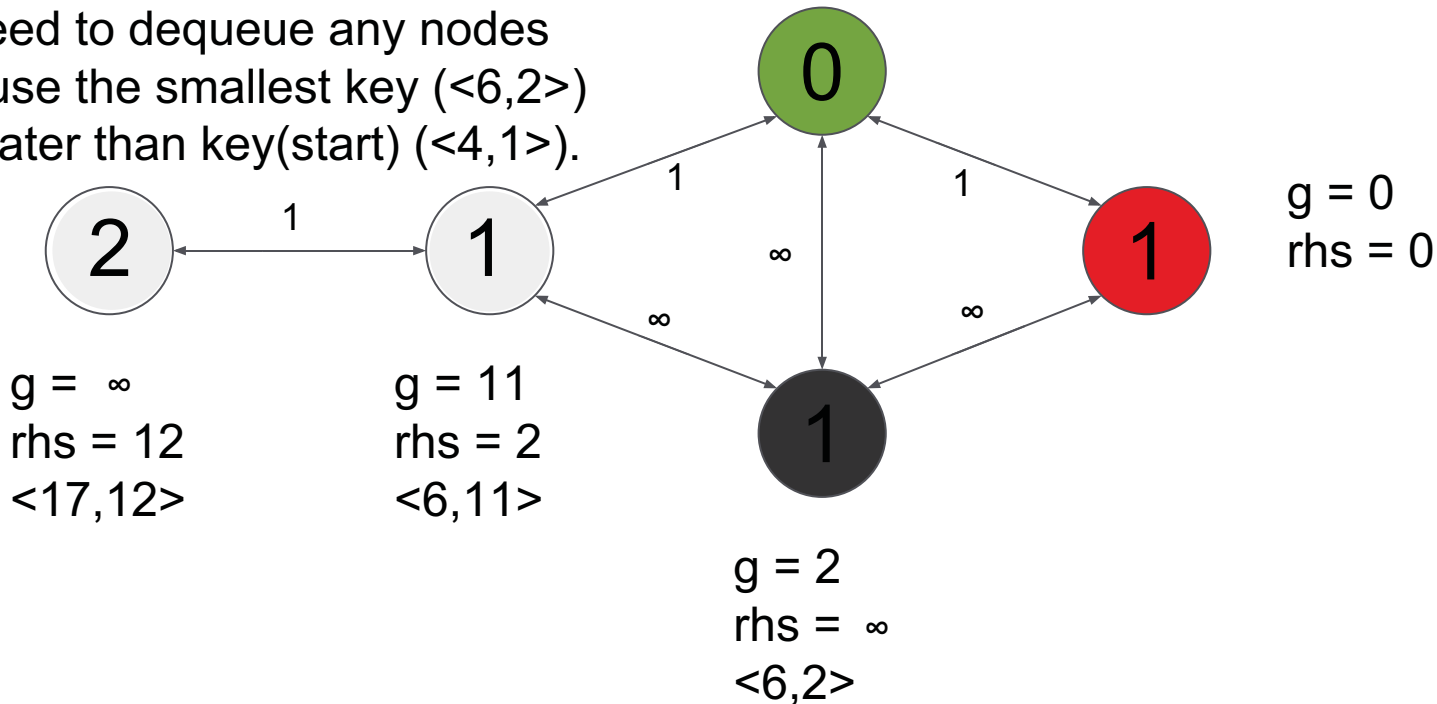


D* Lite Example

5. Repeat from 2

2. Replan path?

- No need to dequeue any nodes because the smallest key $\langle 6, 2 \rangle$ is greater than $\text{key}(\text{start}) \langle 4, 1 \rangle$.



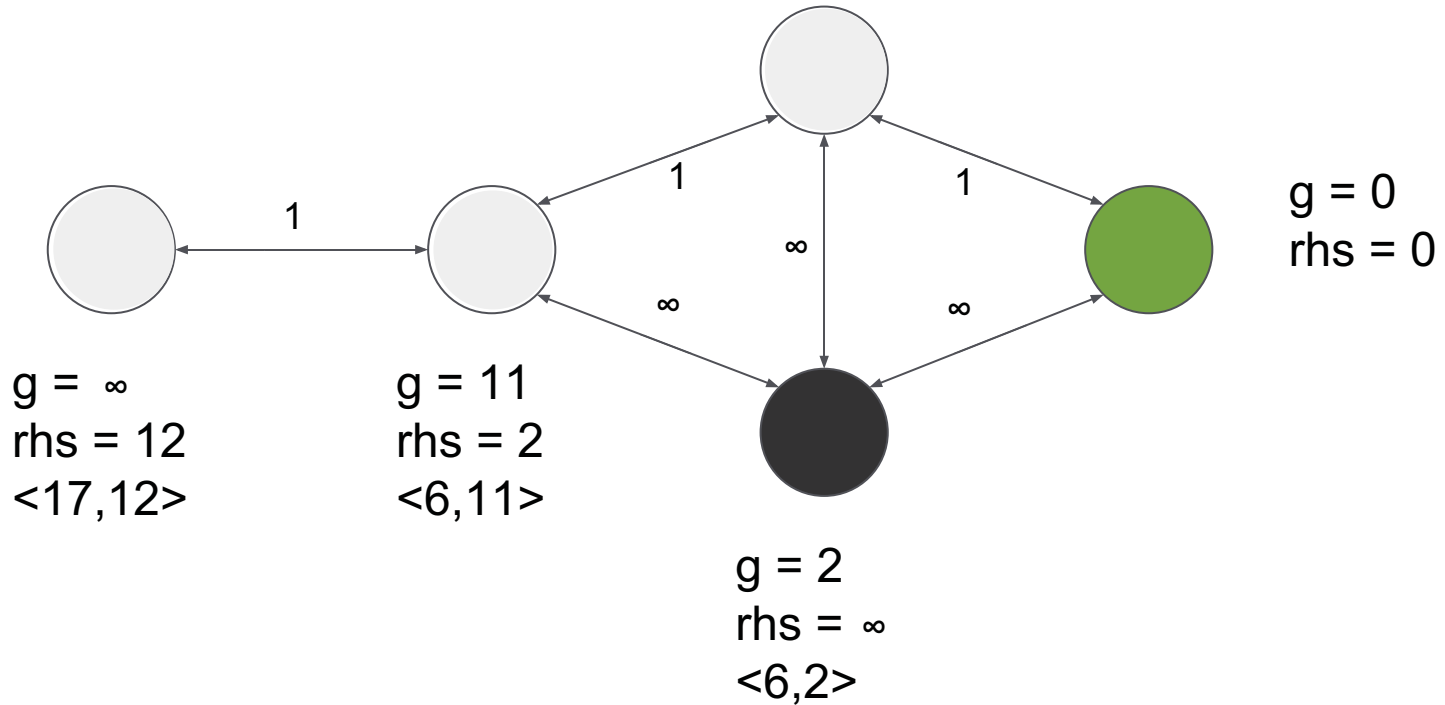
D* Lite Example

3. Robot moves

- Robot reaches goal: Done!

$g = 1$
 $rhs = 1$

$k_m = 3$



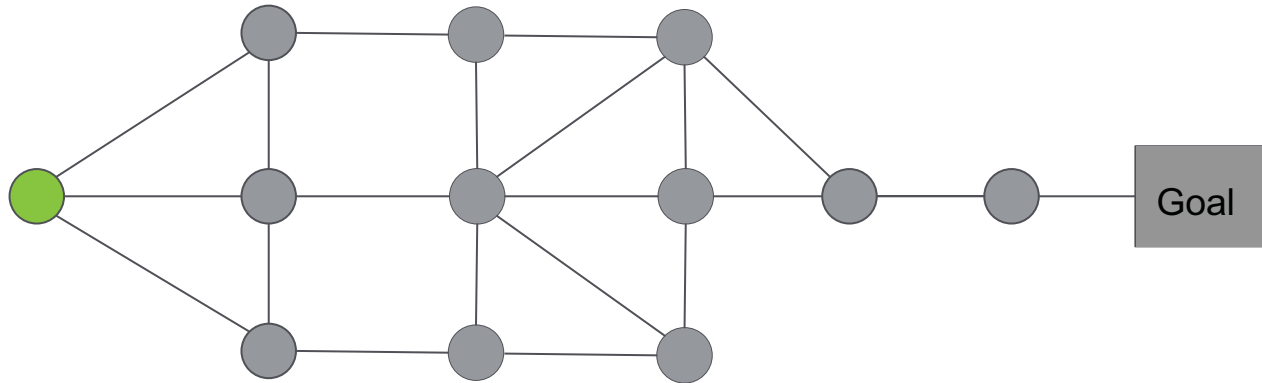
Outline

- Motivation
- Incremental Search
- The D* Lite Algorithm
- D* Lite Example
- **When to Use Incremental Path Planning?**
- Algorithm Extensions and Related Topics
- Application to Mobile Robotics

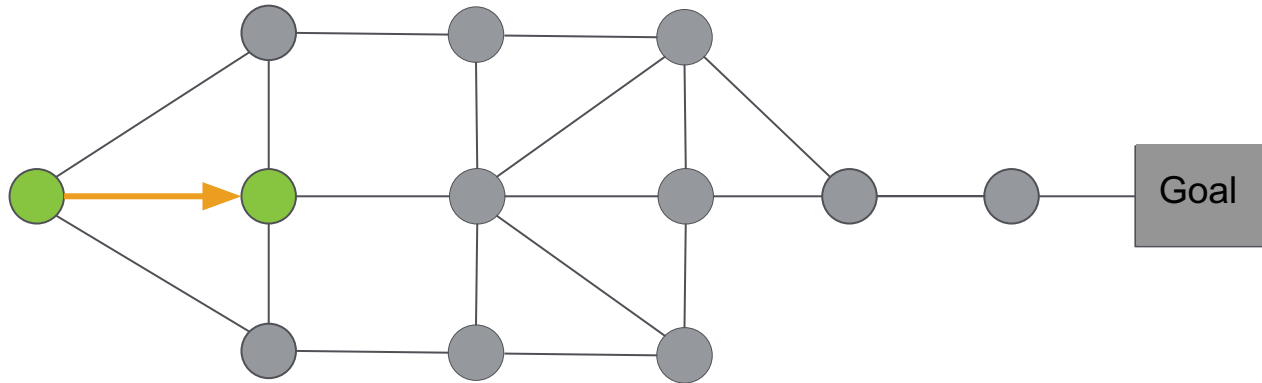
Efficiency

- A* expands each node at most once
- D* Lite (and LPA*) expands each node at most twice
 - But will in most cases expand fewer nodes than A*
- A* might perform better if:
 - Changes are close to start node
 - Large changes to the graph

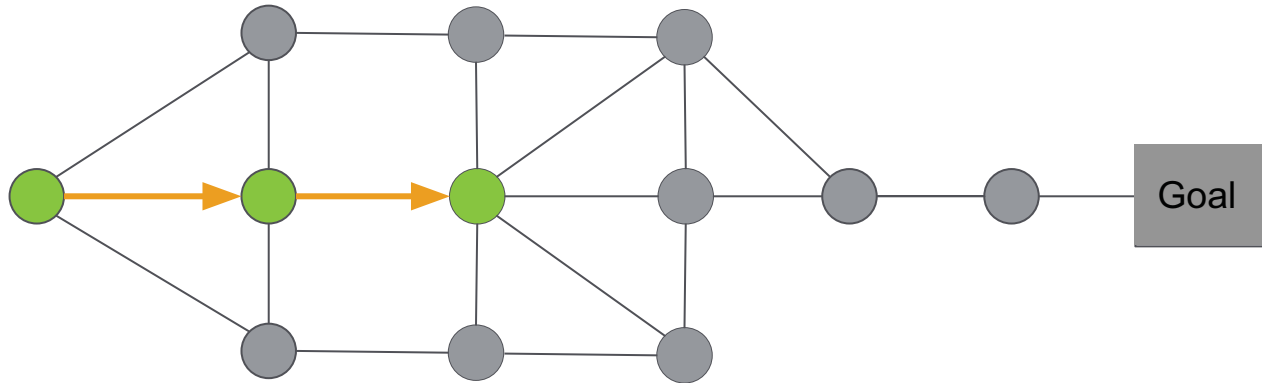
When to use incremental path planning?



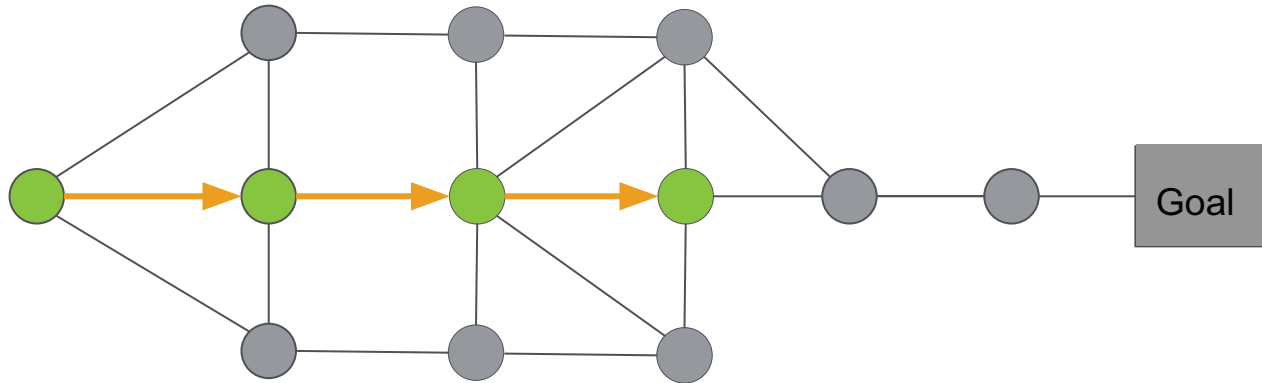
When to use incremental path planning?



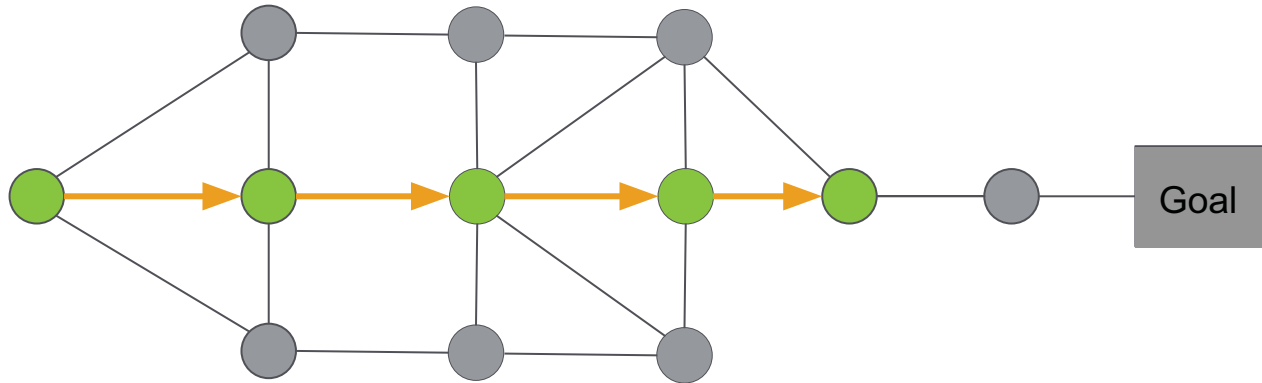
When to use incremental path planning?



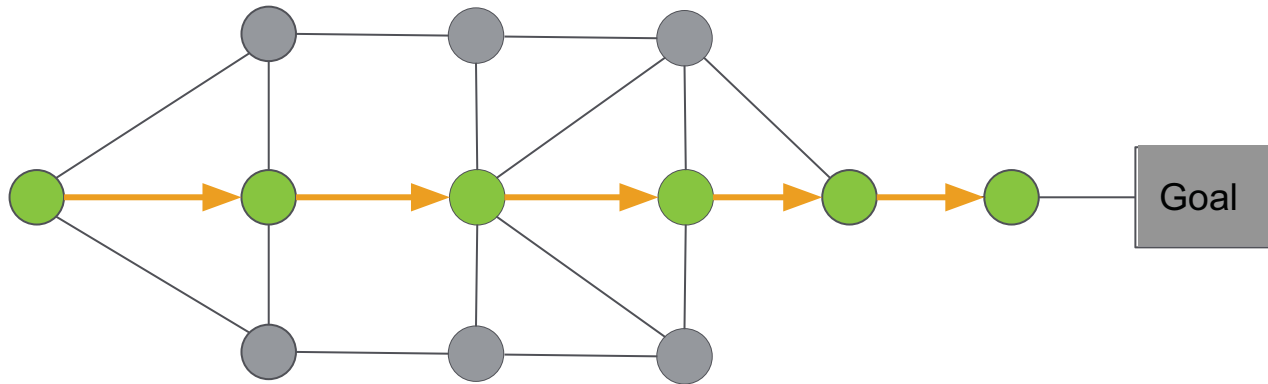
When to use incremental path planning?



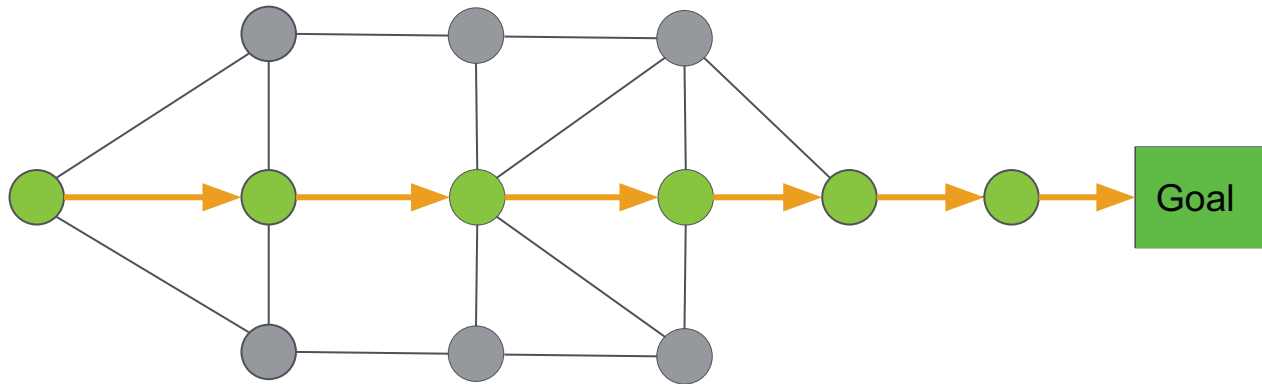
When to use incremental path planning?



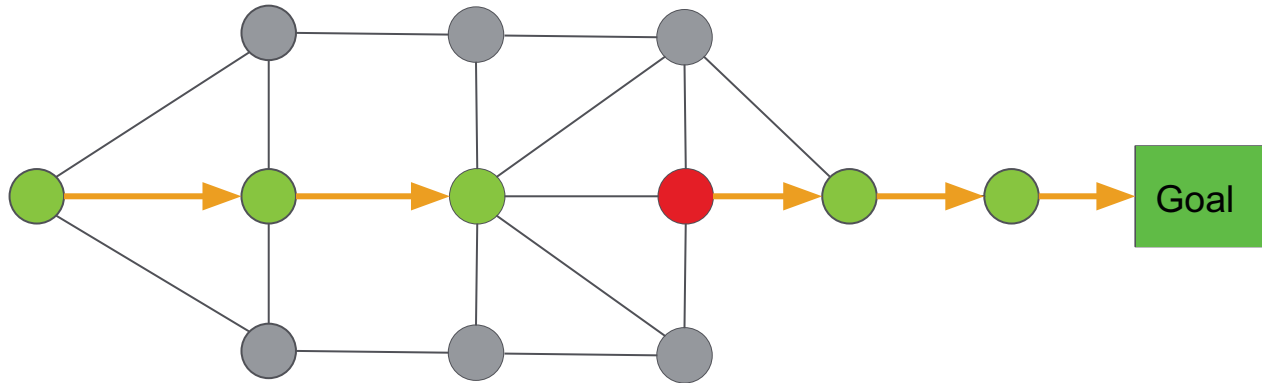
When to use incremental path planning?



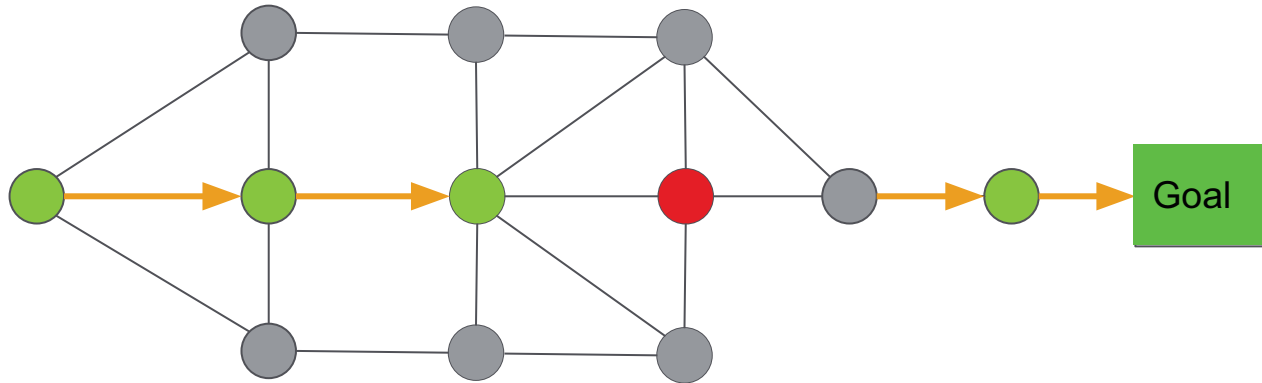
When to use incremental path planning?



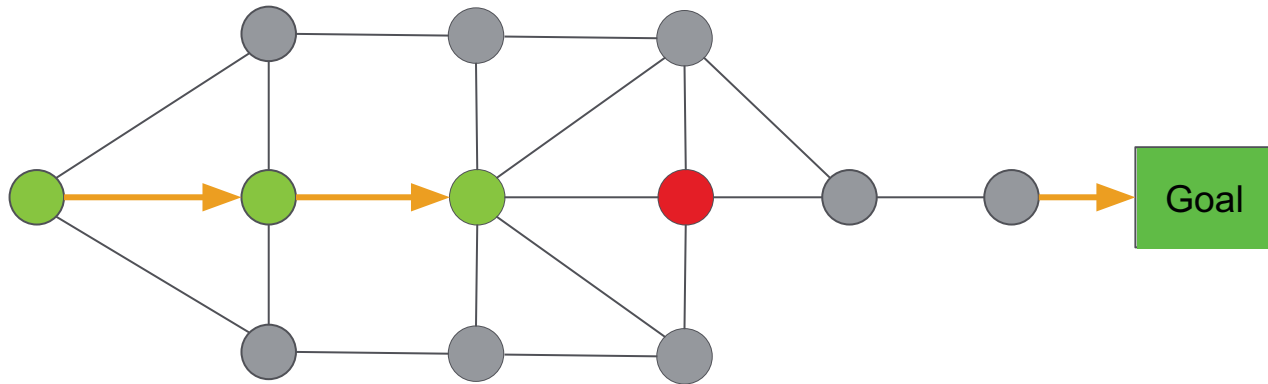
When to use incremental path planning?



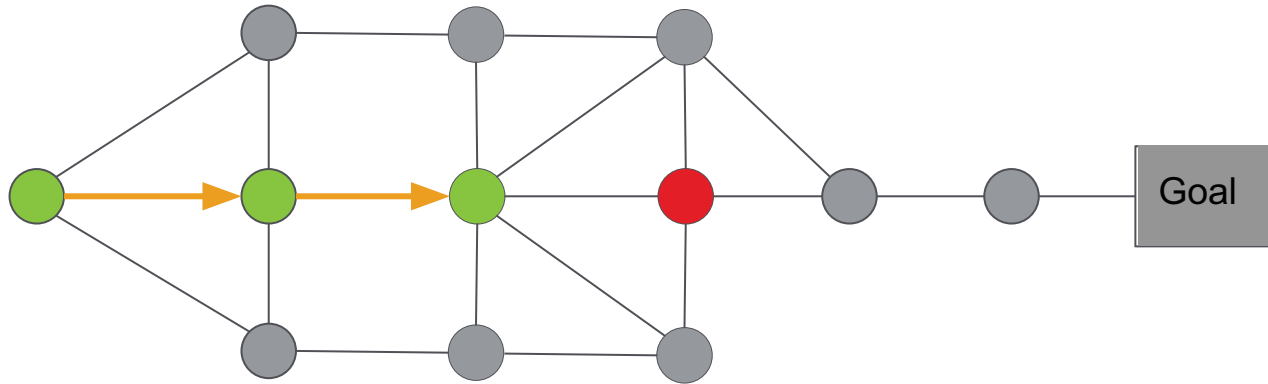
When to use incremental path planning?



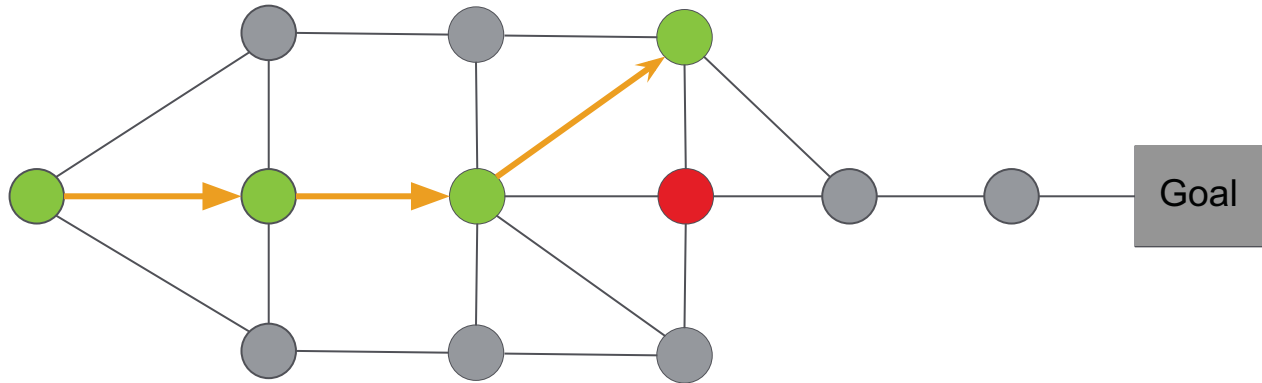
When to use incremental path planning?



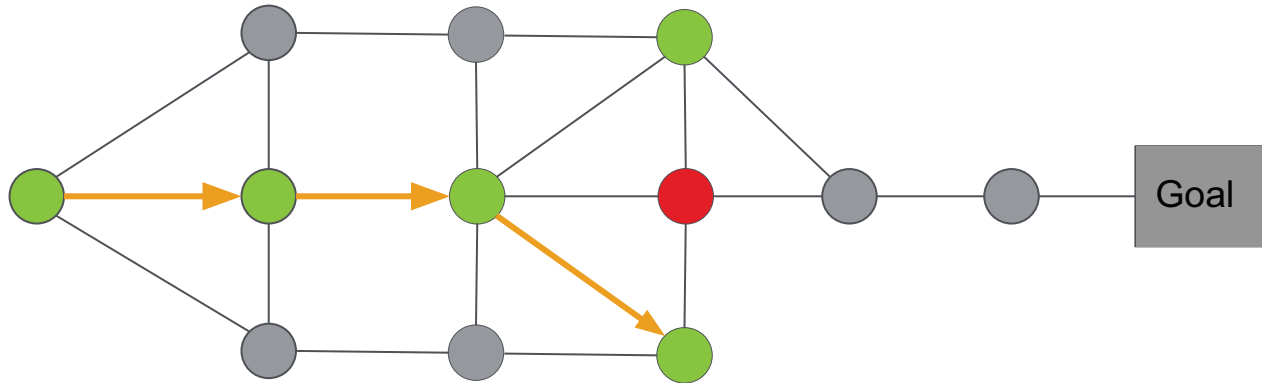
When to use incremental path planning?



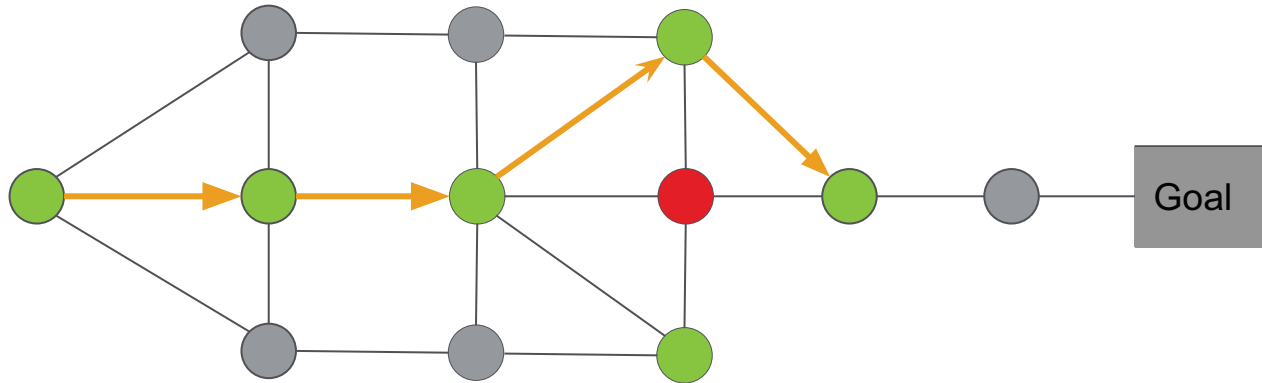
When to use incremental path planning?



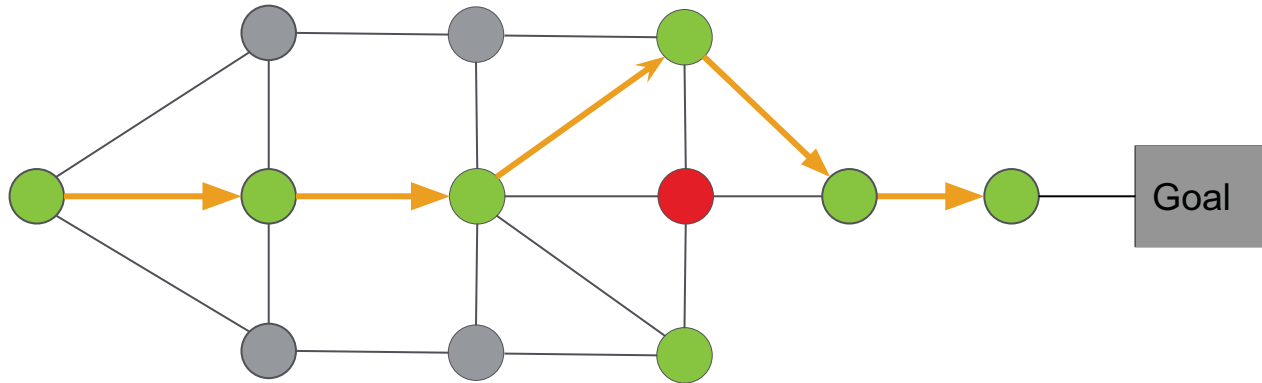
When to use incremental path planning?



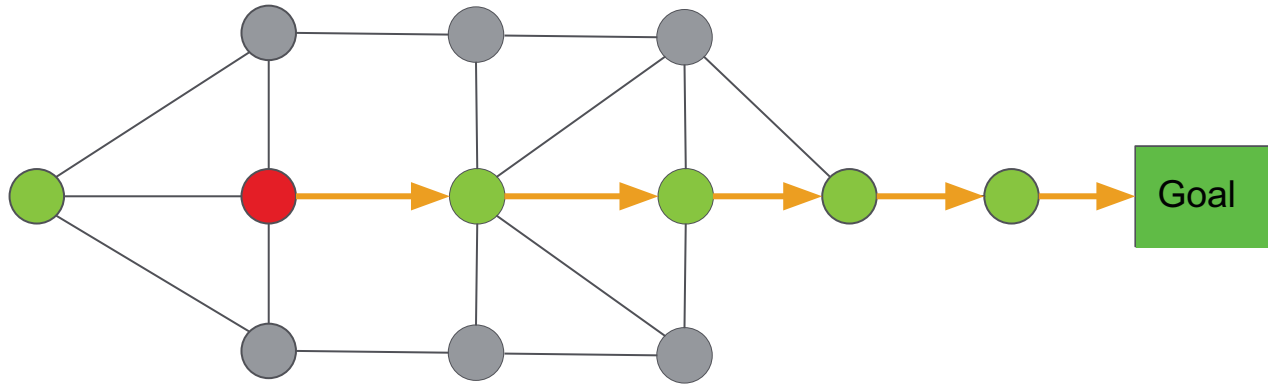
When to use incremental path planning?



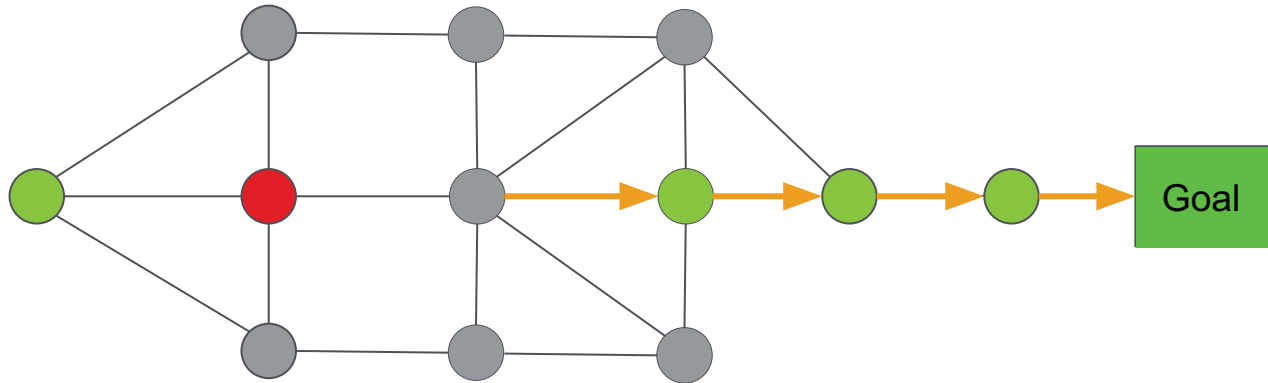
When to use incremental path planning?



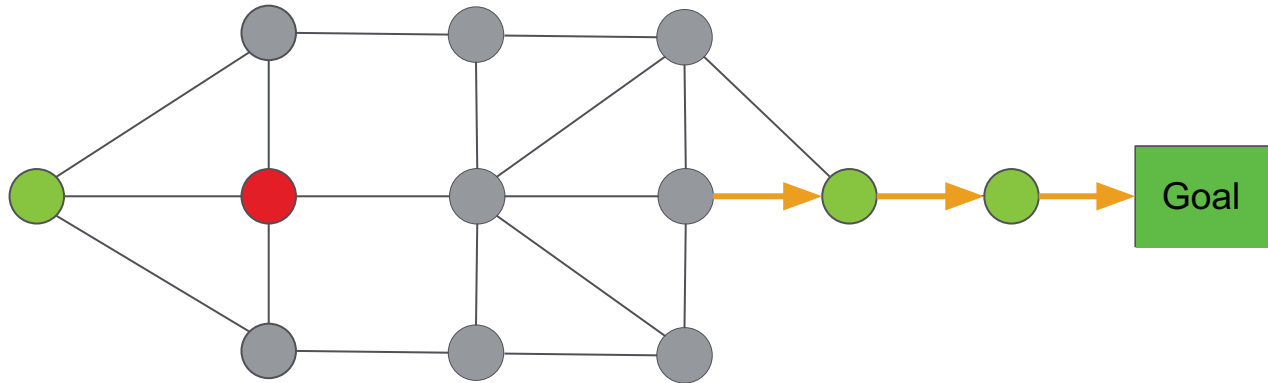
When to use incremental path planning?



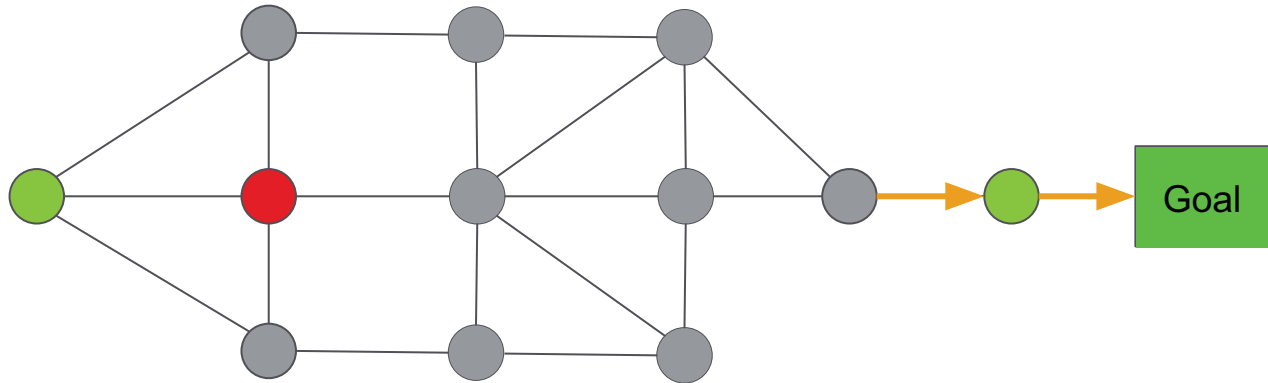
When to use incremental path planning?



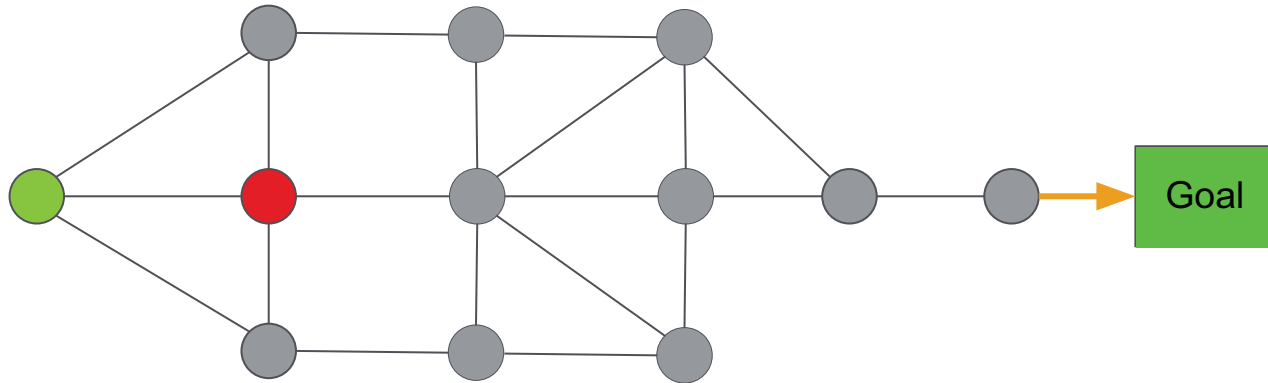
When to use incremental path planning?



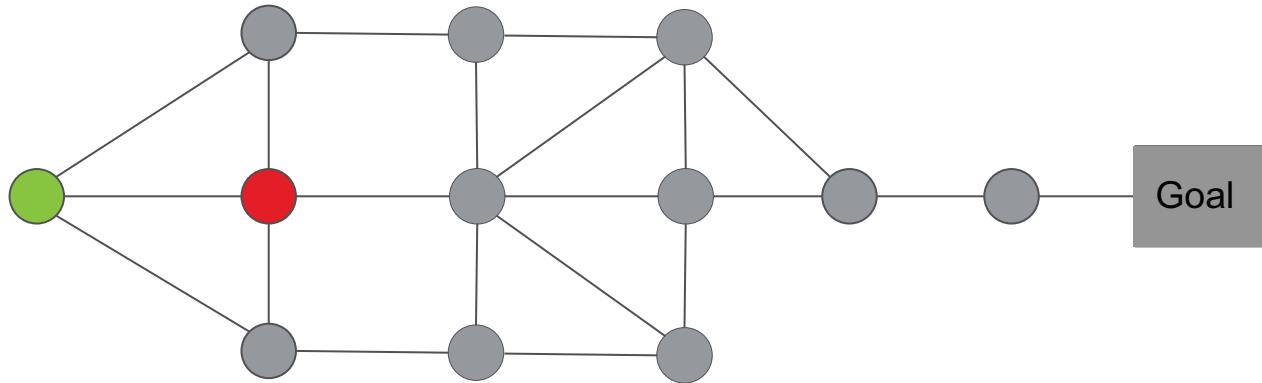
When to use incremental path planning?



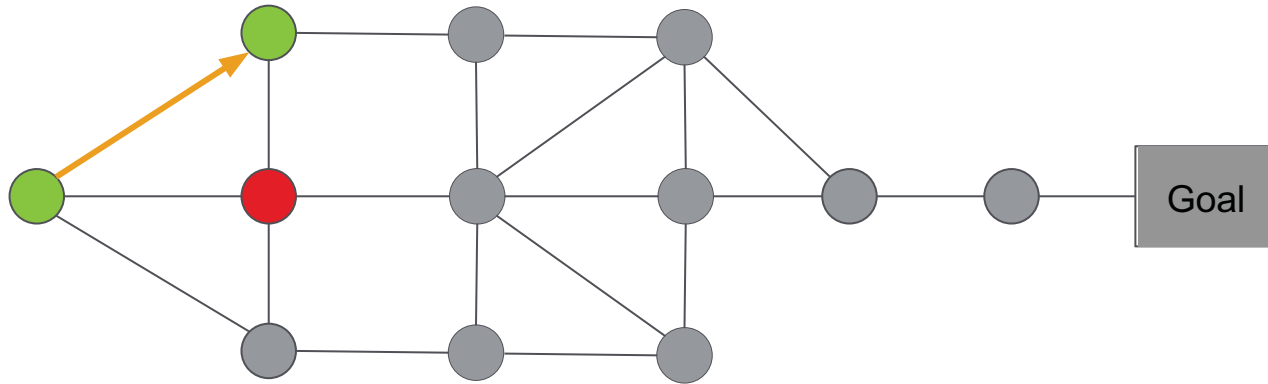
When to use incremental path planning?



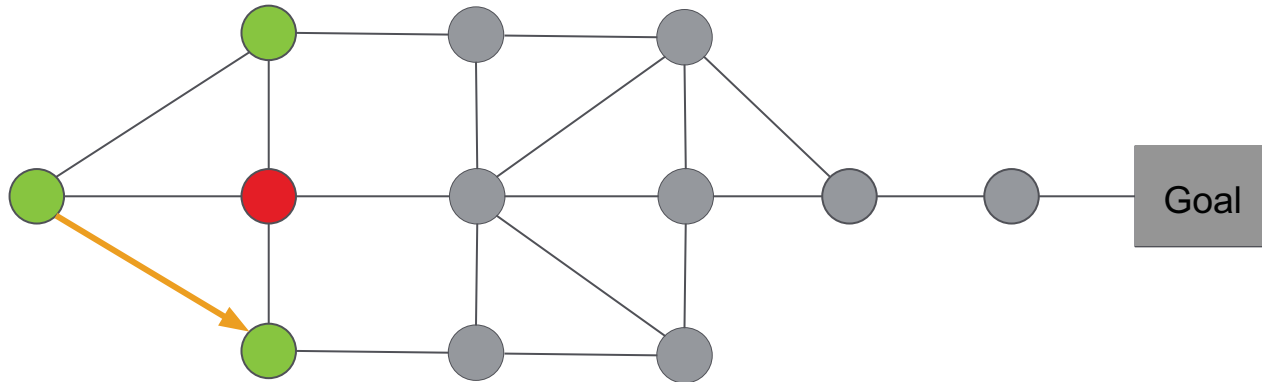
When to use incremental path planning?



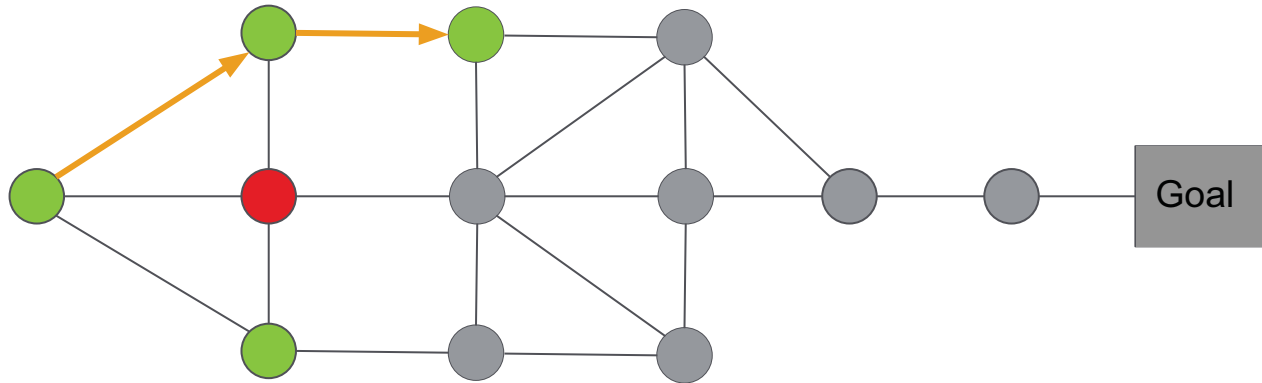
When to use incremental path planning?



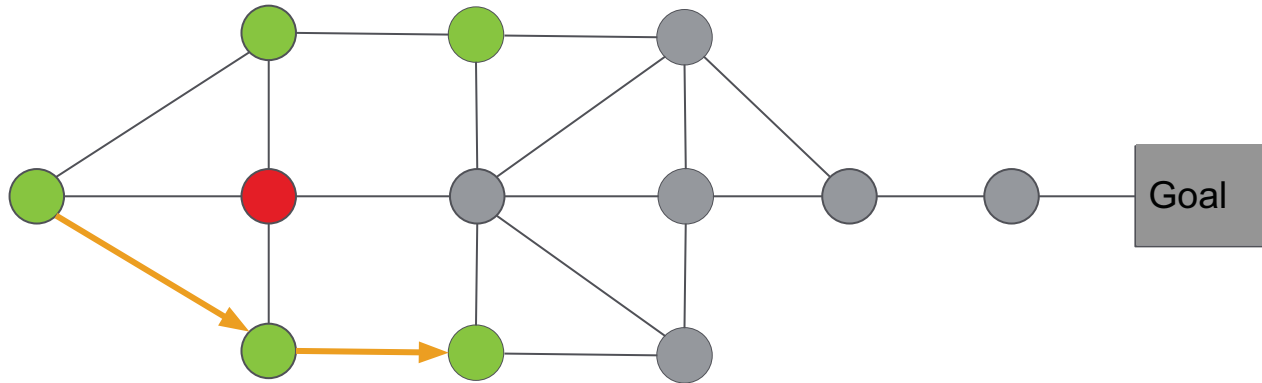
When to use incremental path planning?



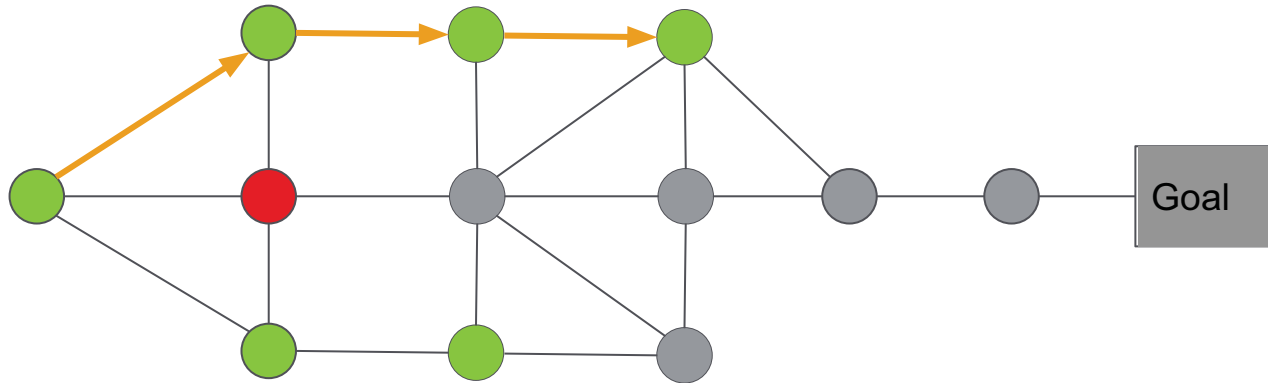
When to use incremental path planning?



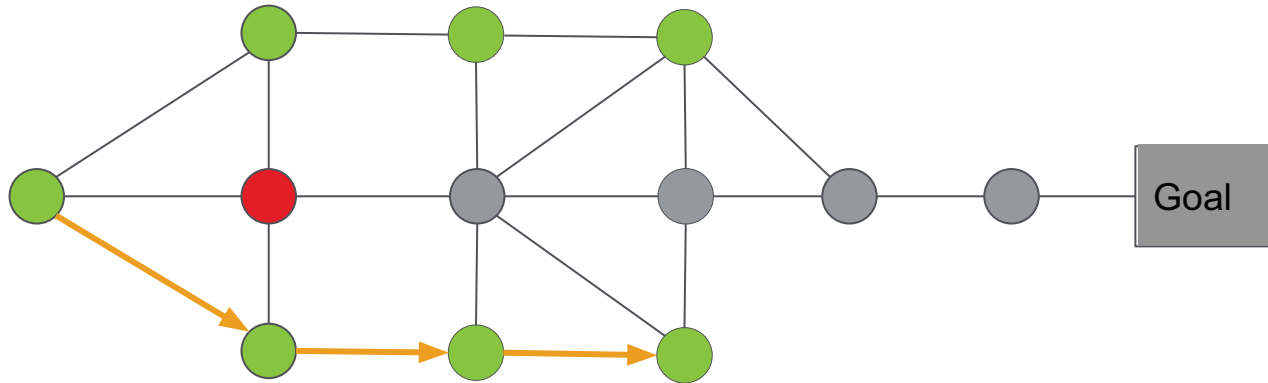
When to use incremental path planning?



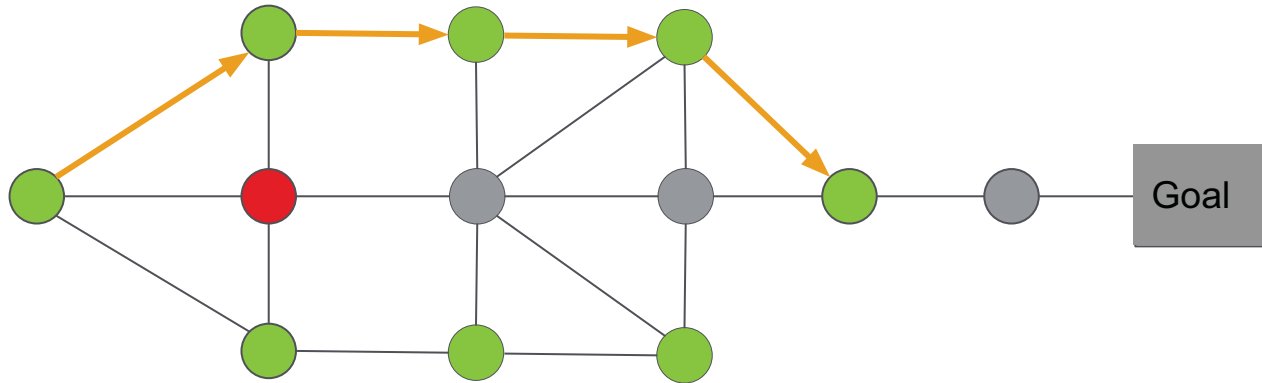
When to use incremental path planning?



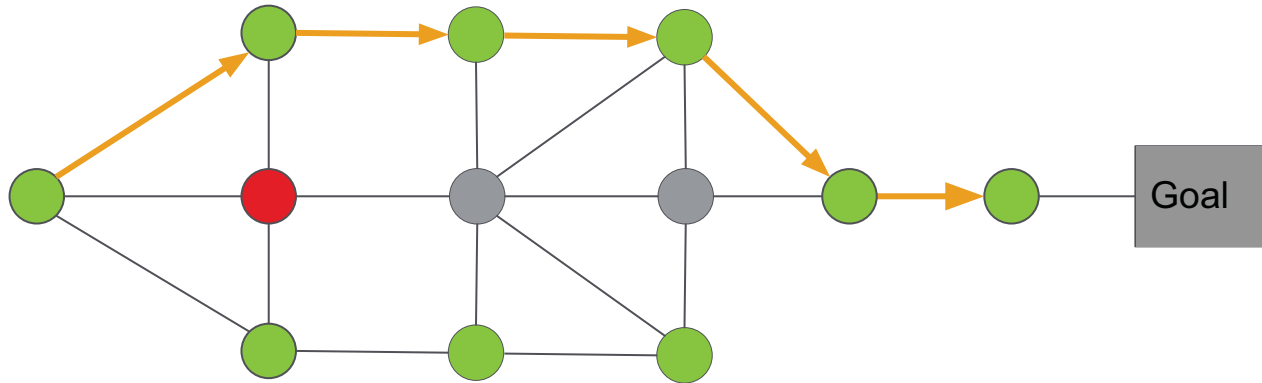
When to use incremental path planning?



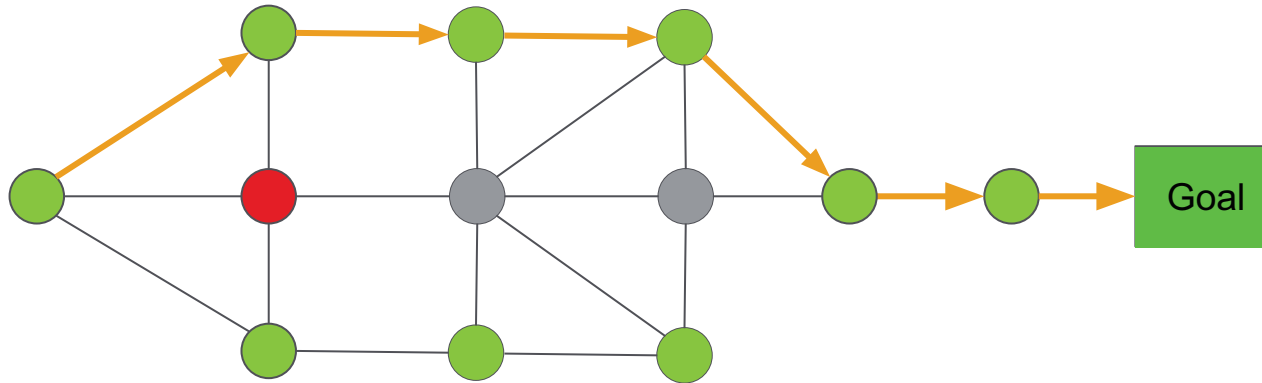
When to use incremental path planning?



When to use incremental path planning?



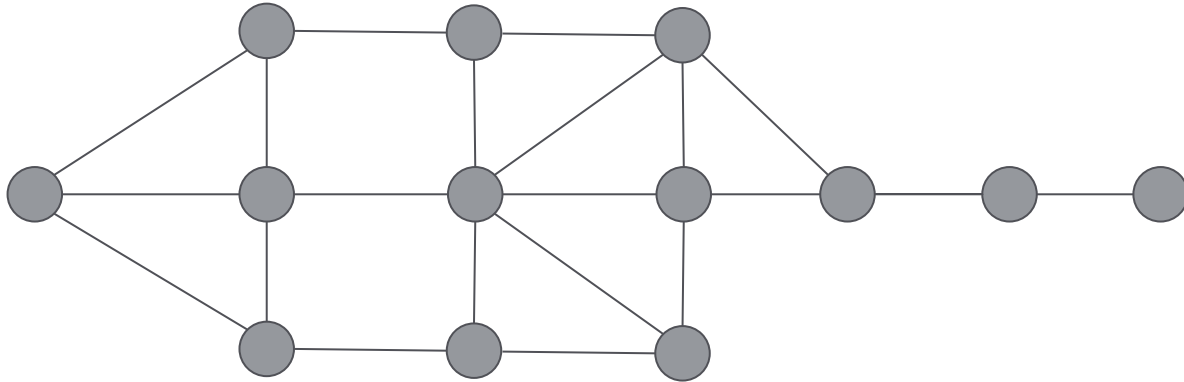
When to use incremental path planning?



Outline

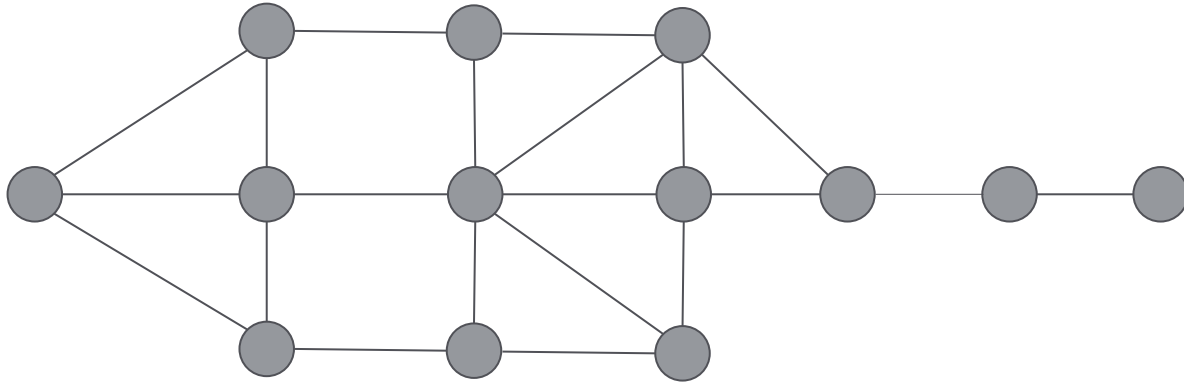
- Motivation
- Incremental Search
- The D* Lite Algorithm
- D* Lite Example
- When to Use Incremental Path Planning?
- **Algorithm Extensions and Related Topics**
- Application to Mobile Robotics

Greedy mapping

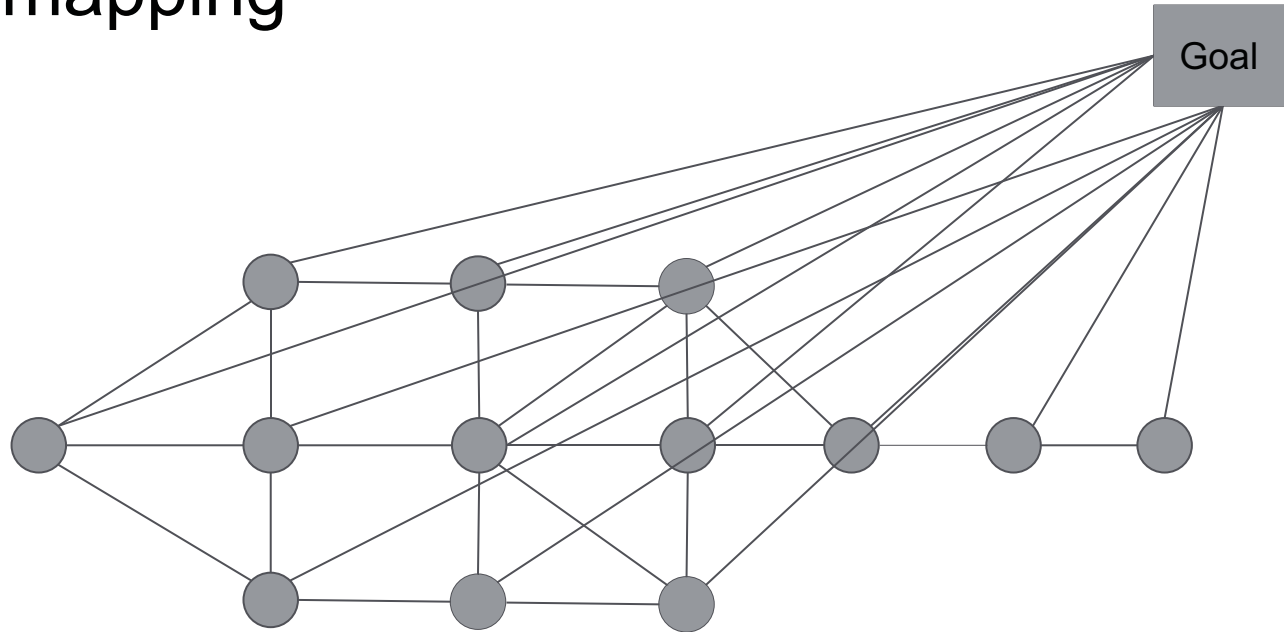


Greedy mapping

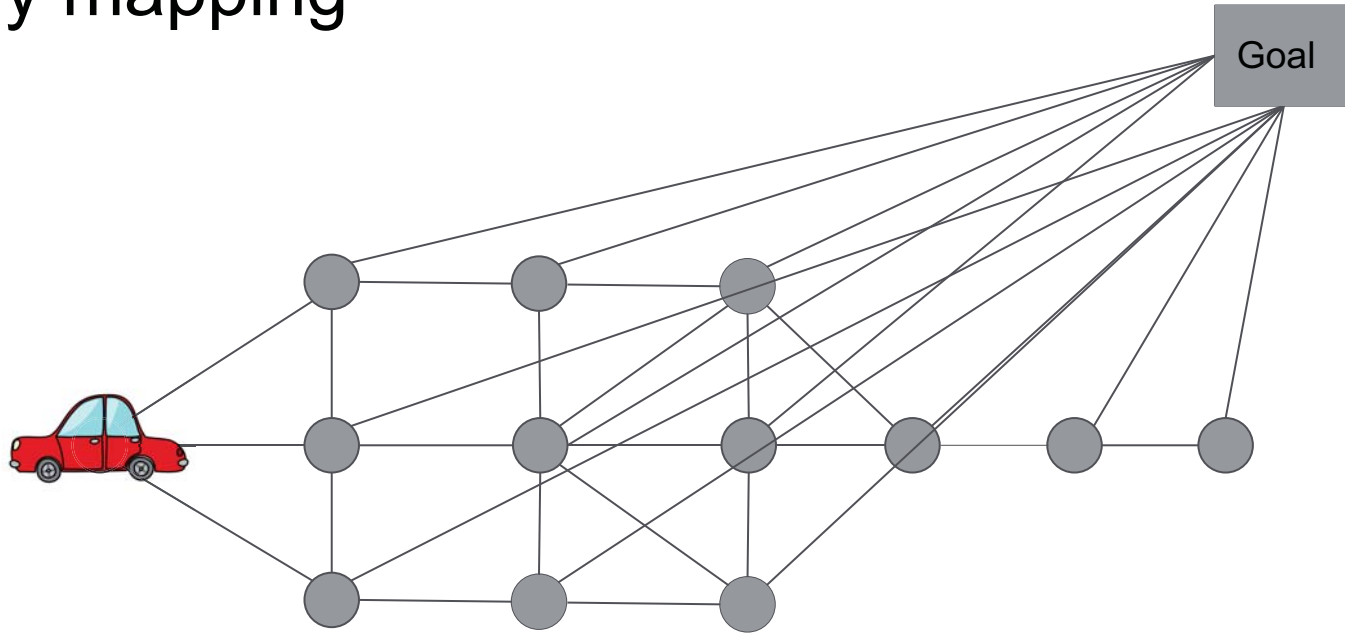
Goal



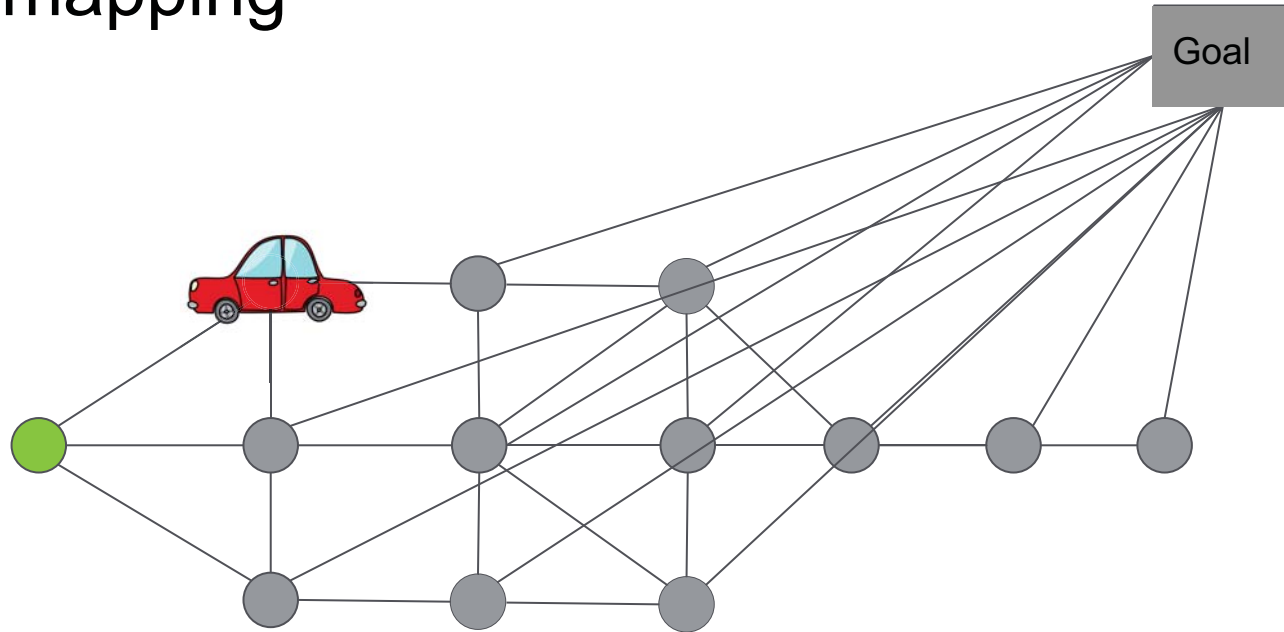
Greedy mapping



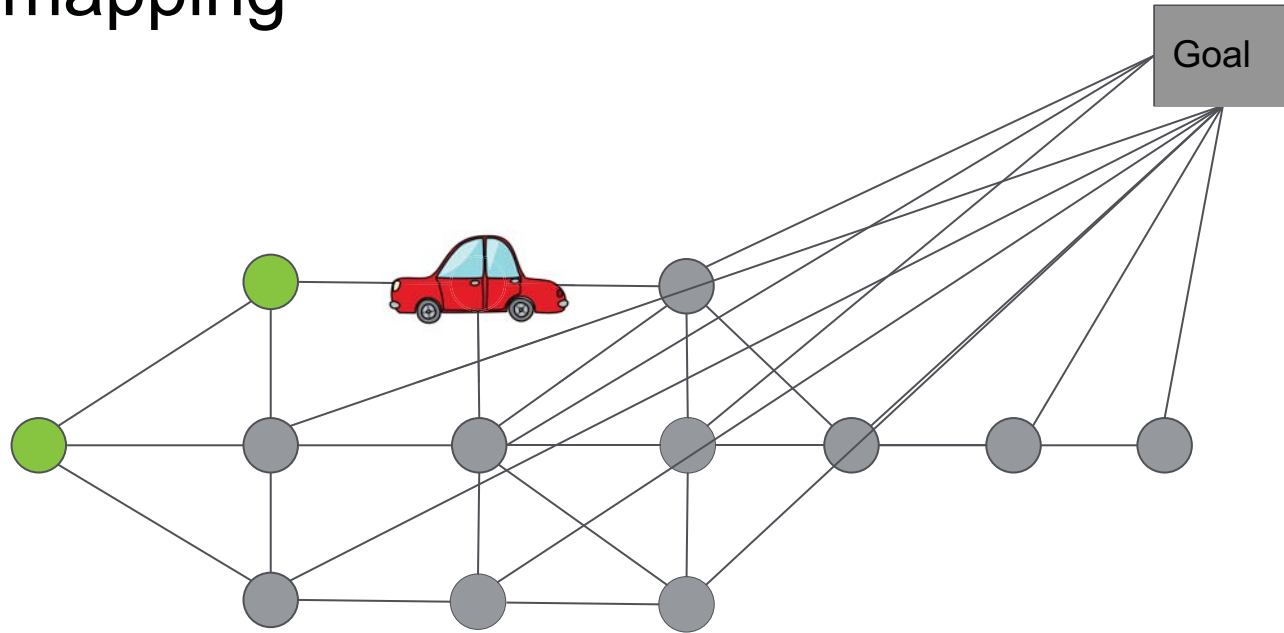
Greedy mapping



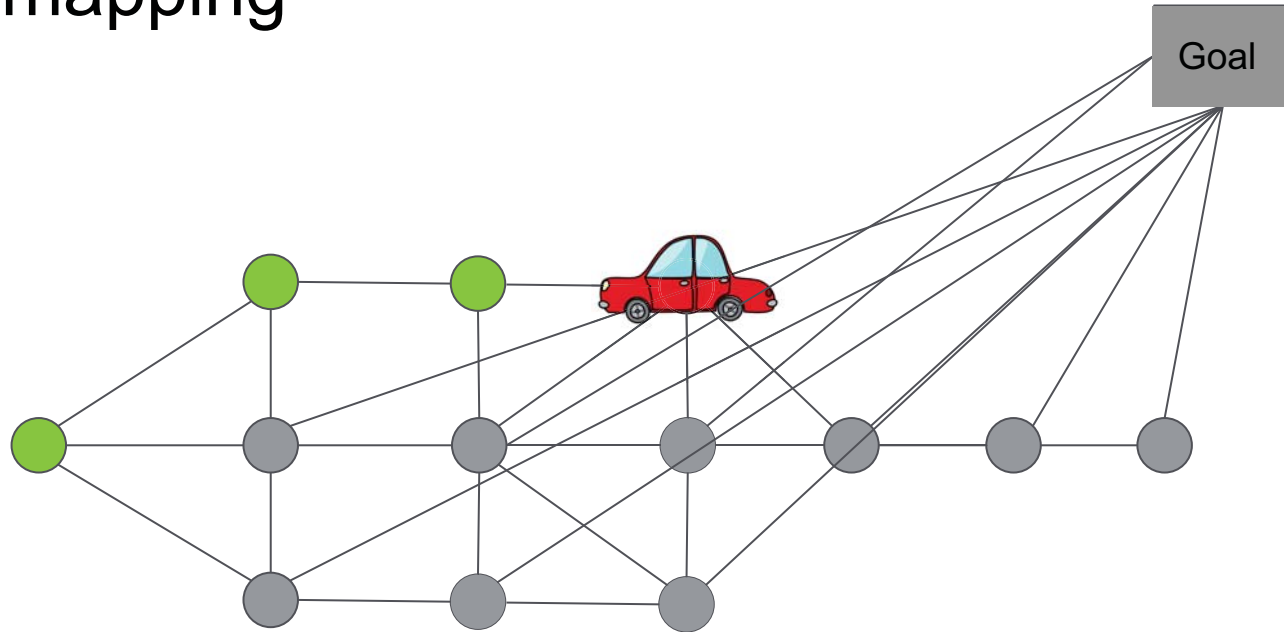
Greedy mapping



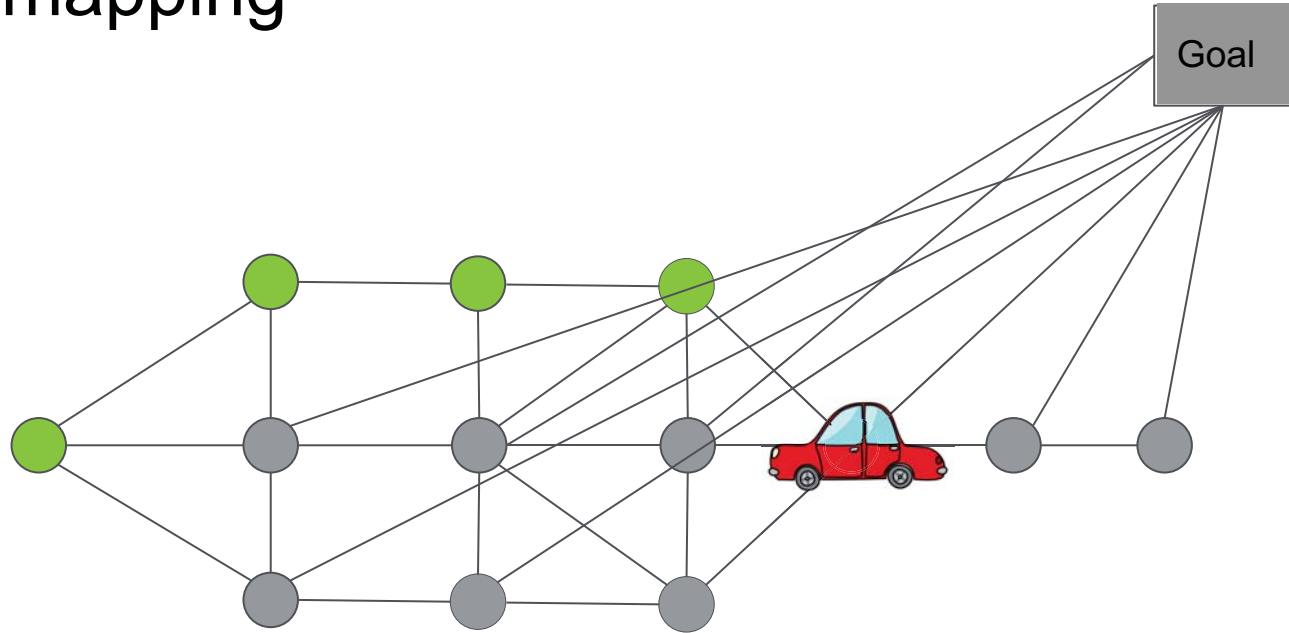
Greedy mapping



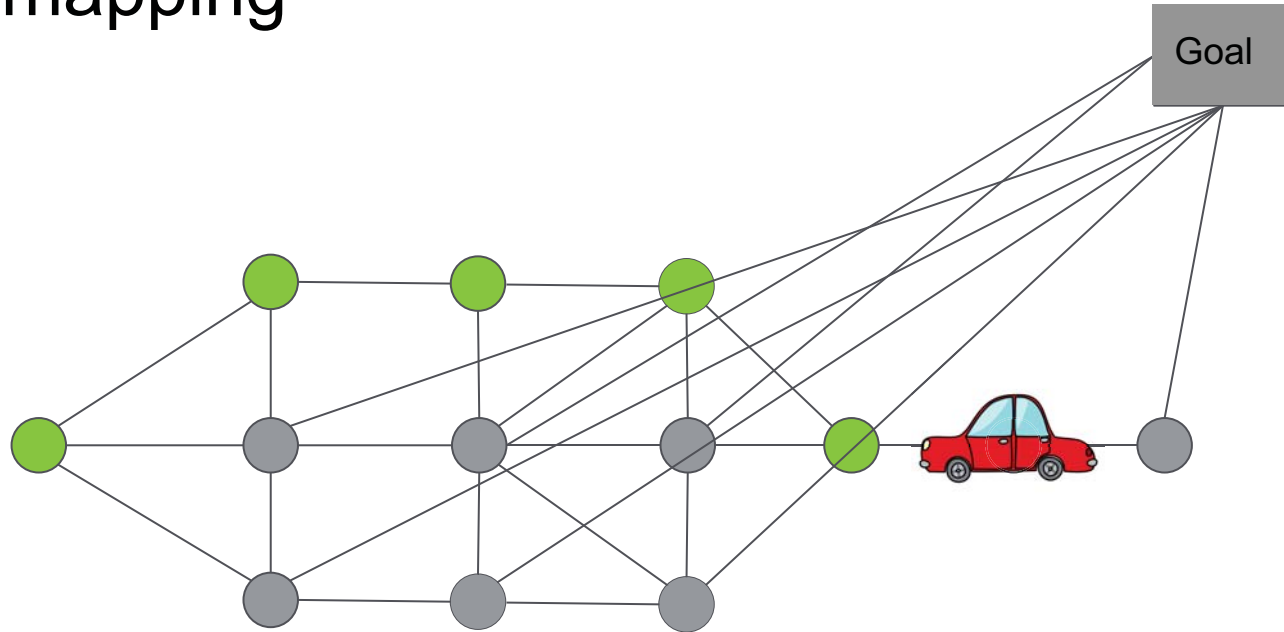
Greedy mapping



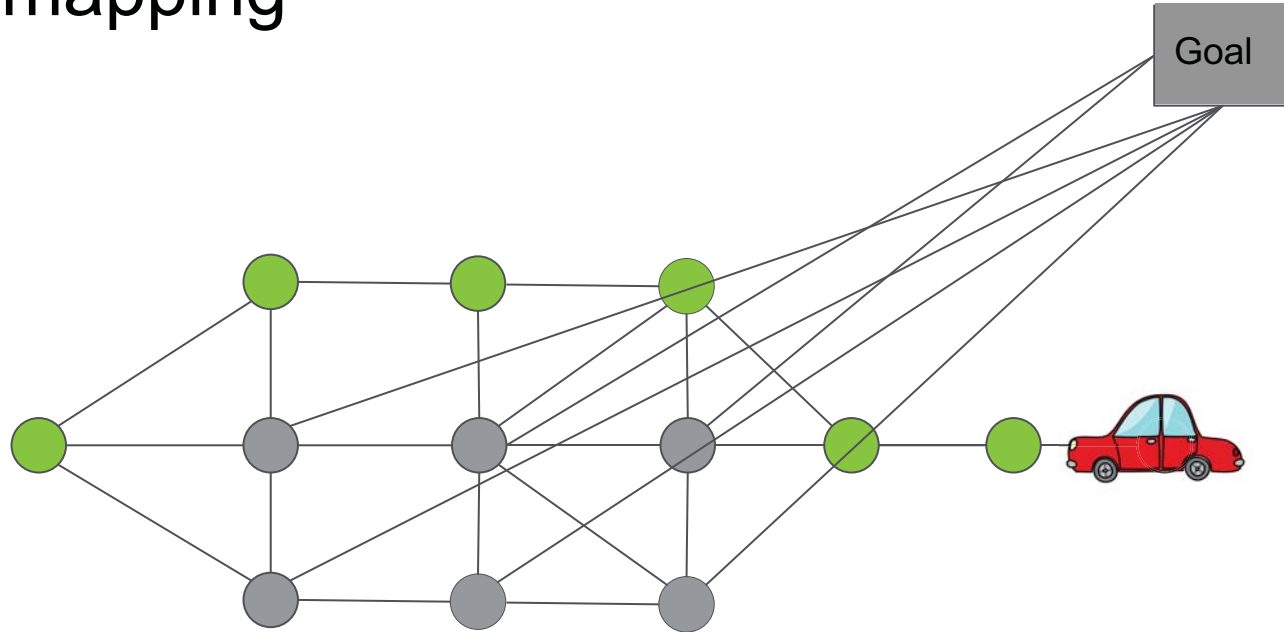
Greedy mapping



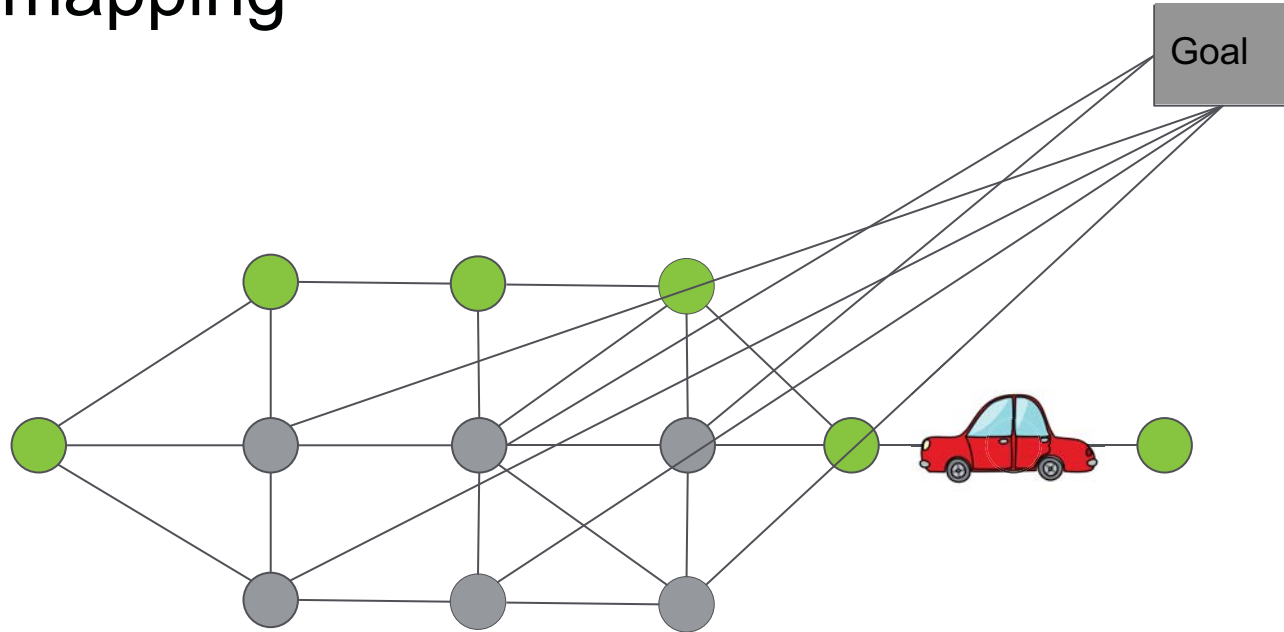
Greedy mapping



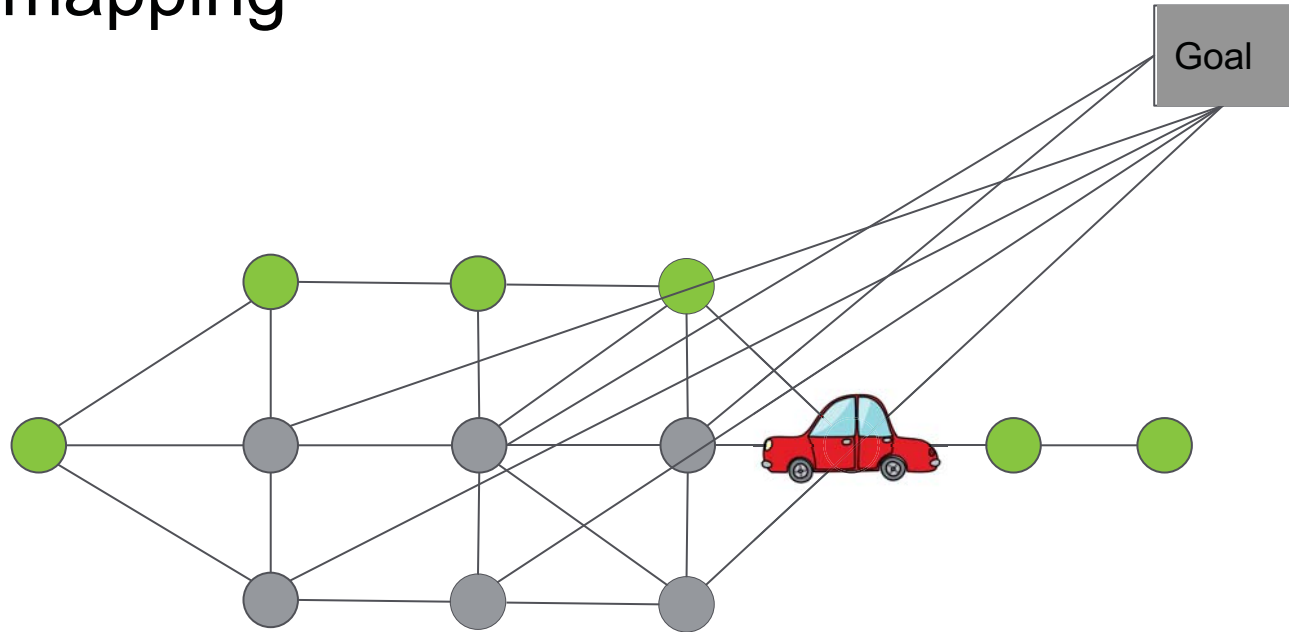
Greedy mapping



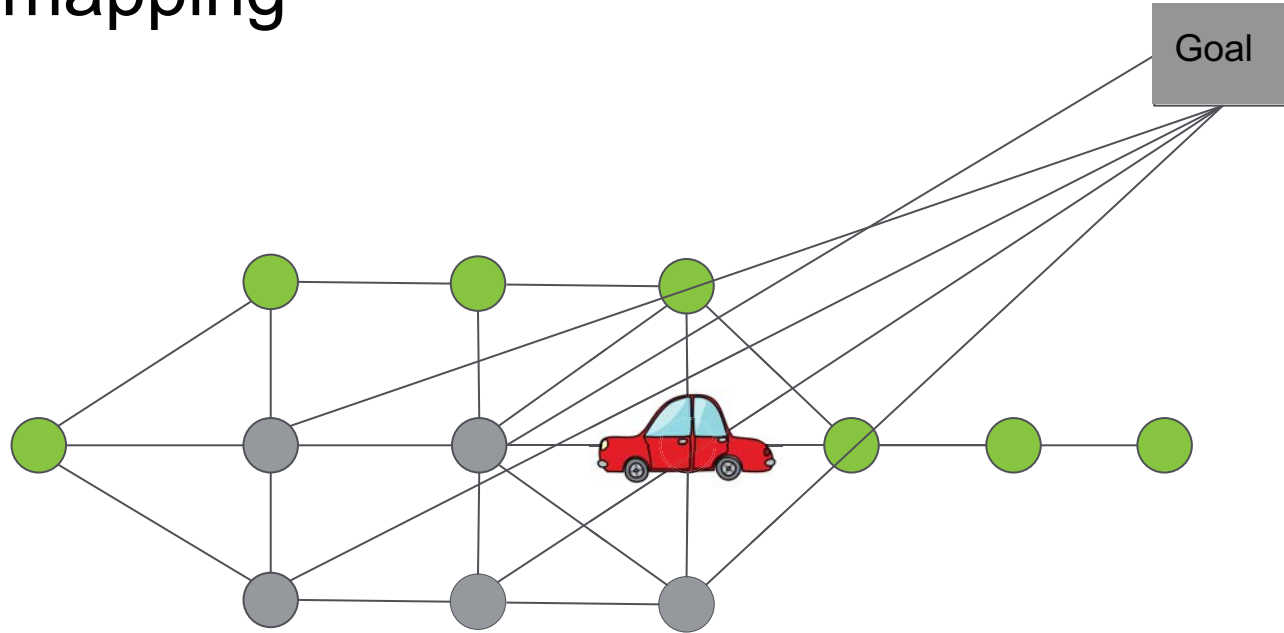
Greedy mapping



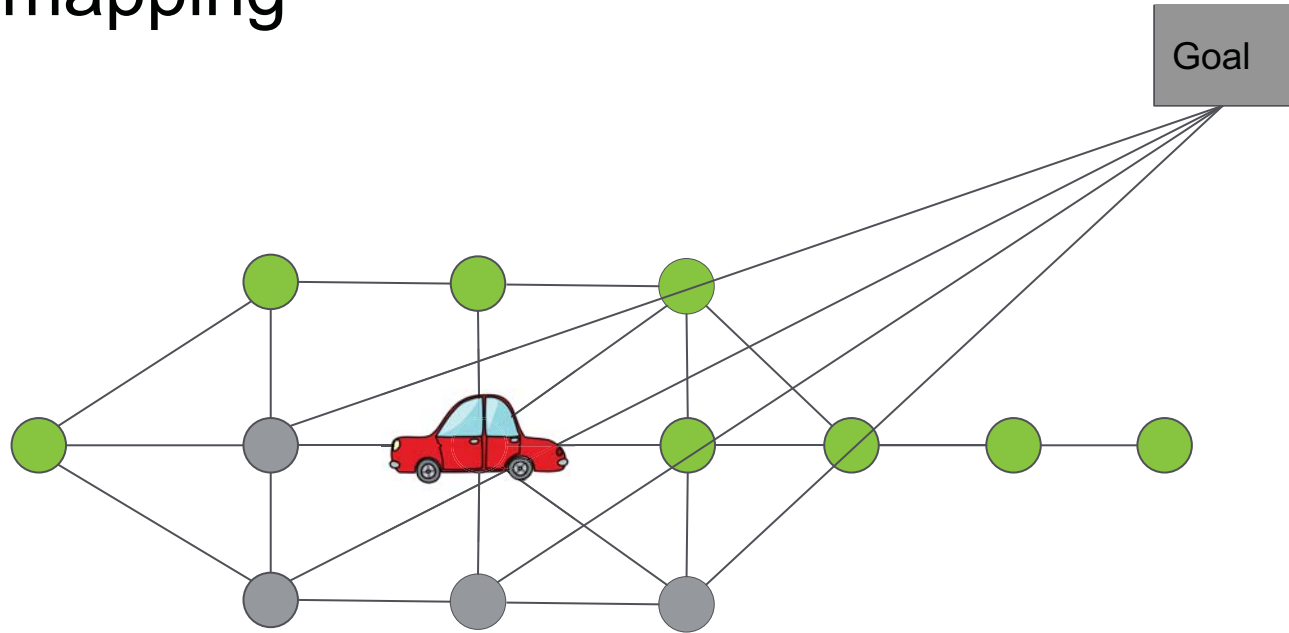
Greedy mapping



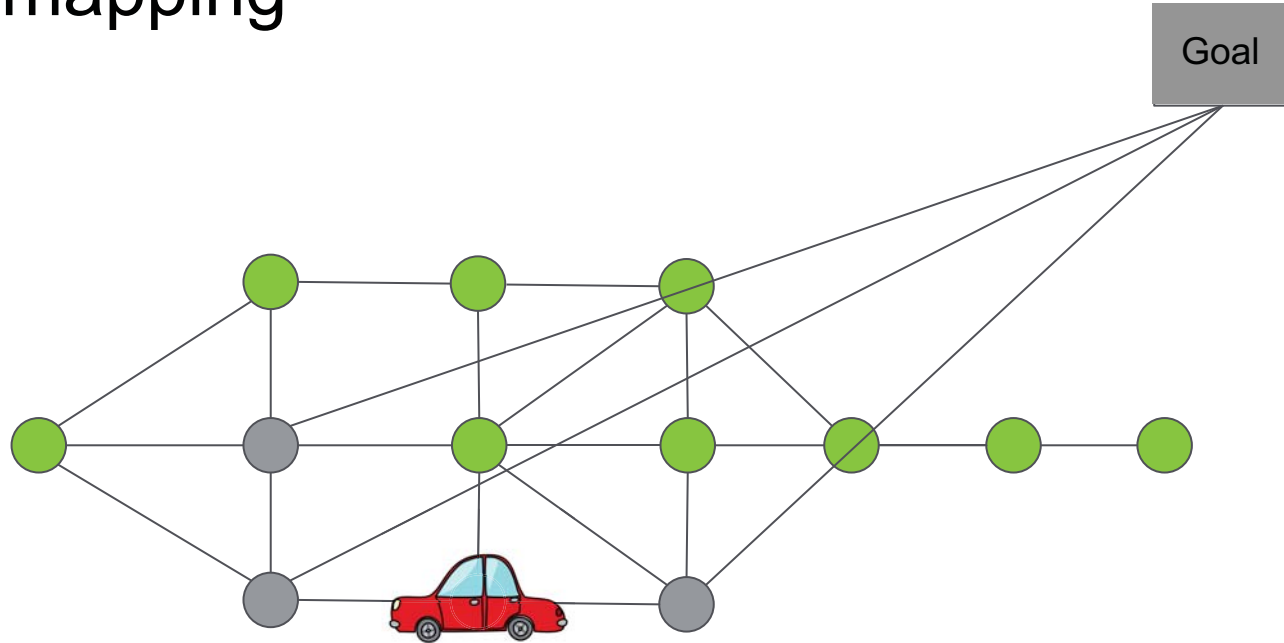
Greedy mapping



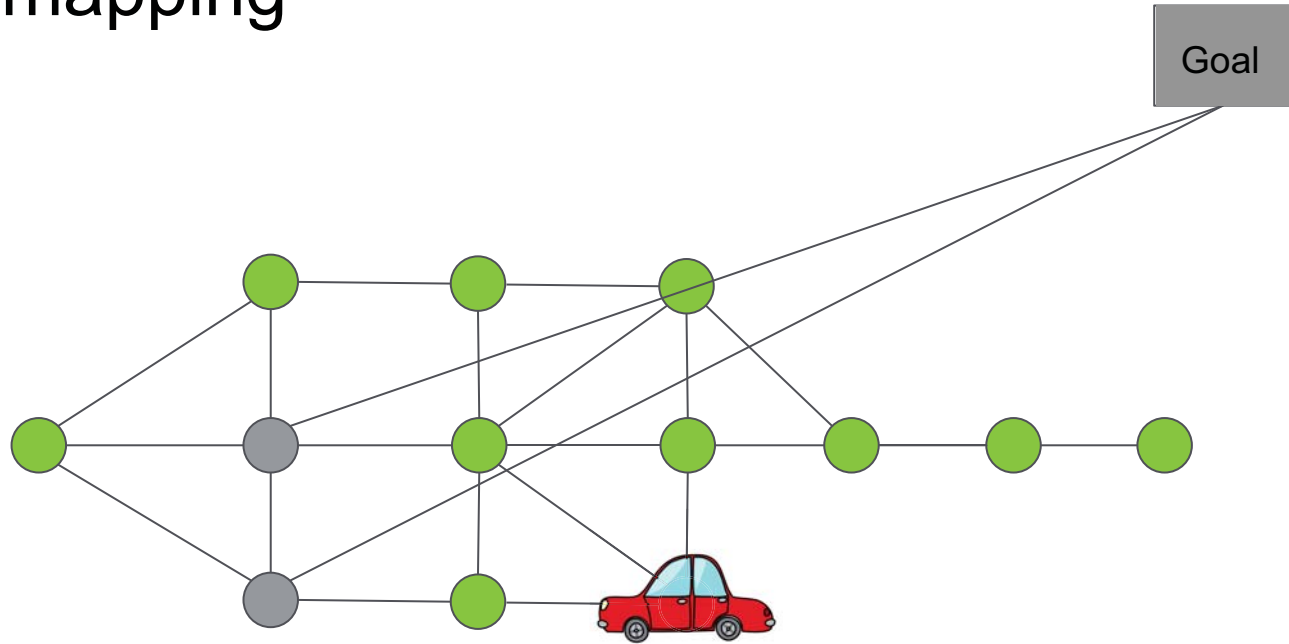
Greedy mapping



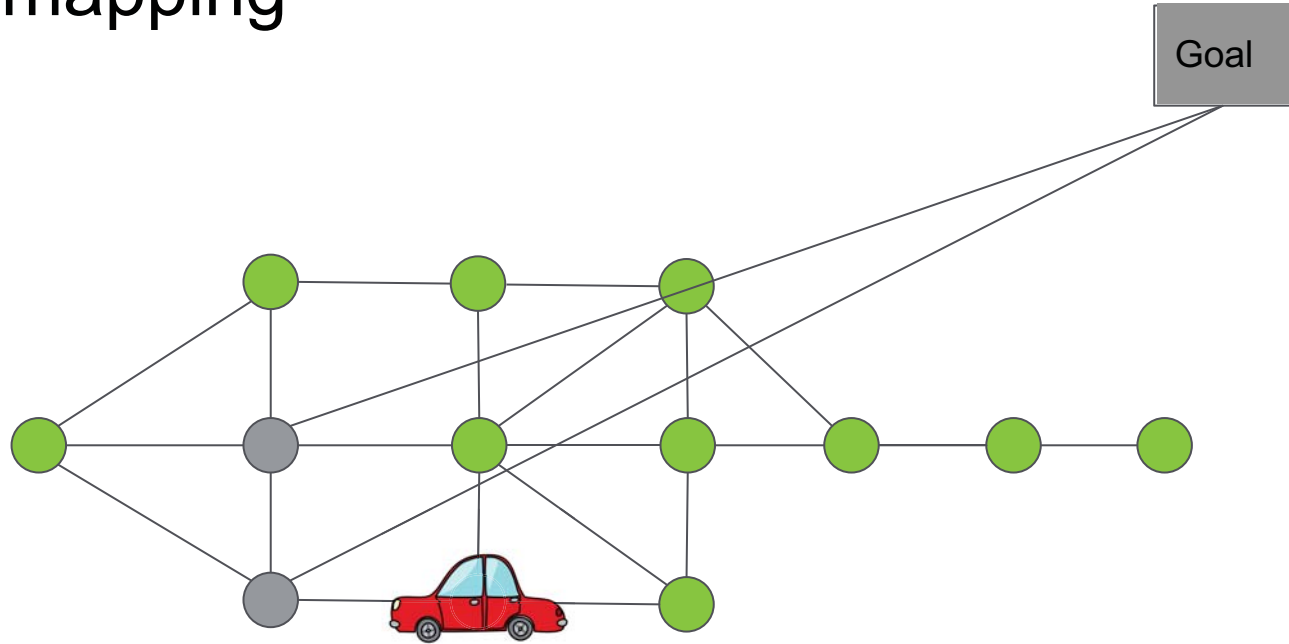
Greedy mapping



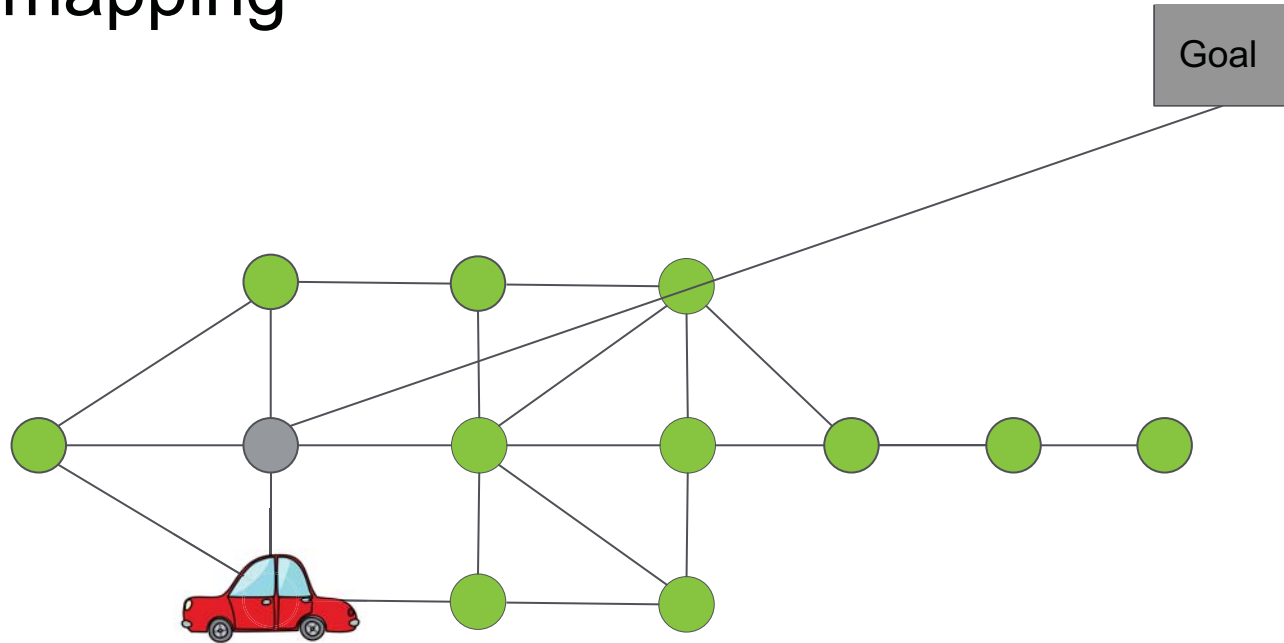
Greedy mapping



Greedy mapping

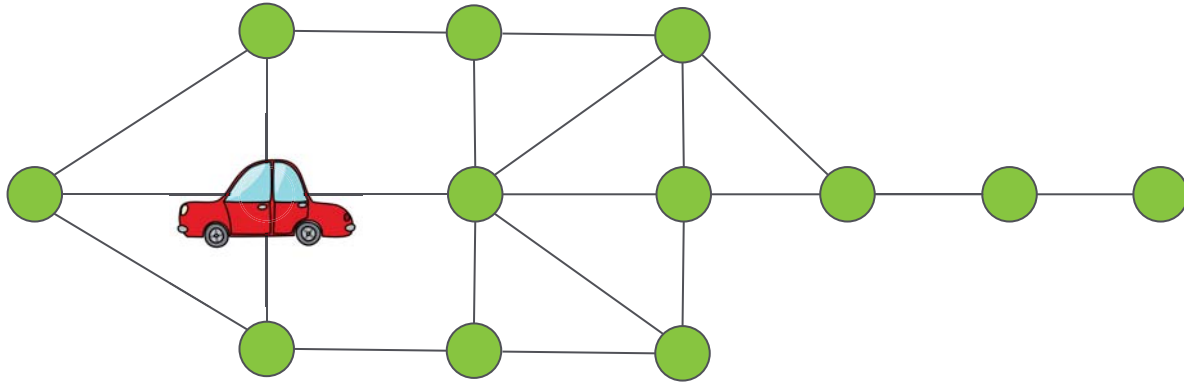


Greedy mapping



Greedy mapping

Goal



Anytime Dynamic A* (AD*)

- Combines the benefits of anytime and incremental planners
- In the real world:
 - Changing environment (incremental planners)
 - Agents need to act upon decisions quickly (anytime planners)



Ref: Likhachev, M., Ferguson, D. I., Gordon, G. J., Stentz, A., & Borrajo, S. (2005, June). Anytime Dynamic A*: An Anytime, Replanning Algorithm. In *ICAPS*(pp. 262 - 271).

Anytime Planners

- Usually start off by computing a highly suboptimal solution
 - then improves the solution over time
- A* with inconsistent heuristics for example
 - Inflated heuristic values give substantial speed-up
 - Inflated heuristics make the algorithm prefer to keep expanding paths
 - Use incrementally less inflated heuristics to approach optimality

Anytime Dynamic A* (AD*)

- Starts off by setting a sufficiently high inflation factor
 - Generates suboptimal plan quickly
- Decreases inflation factor to approach optimality
- When changes to edge costs are detected the current solution is repaired
 - If the changes are substantial the inflation factor is increased to generate a new plan quickly

Outline

- Motivation
- Incremental Search
- The D* Lite Algorithm
- D* Lite Example
- When to Use Incremental Path Planning?
- Algorithm Extensions and Related Topics
- **Application to Mobile Robotics**

Application to mobile robotics

- How do we go from a complete configuration space to a graph?



This image is in the public domain.

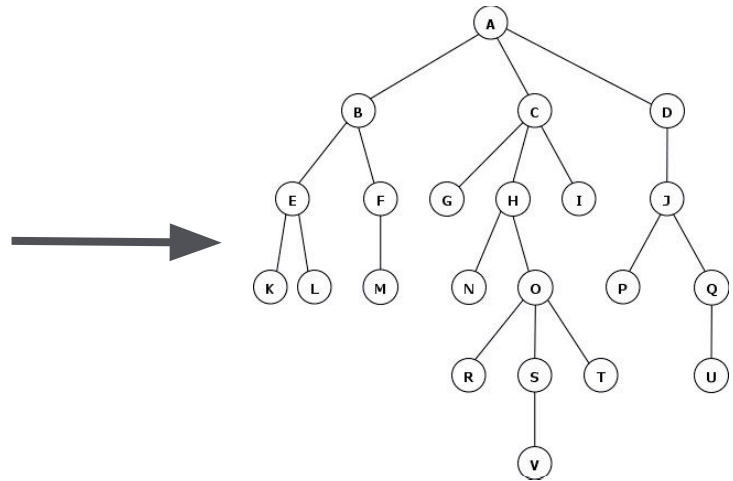
Reference: Lecture given by Michal Čáp in 2.166 and paper currently in review

Application to mobile robotics

- How do we go from a complete configuration space to a graph?



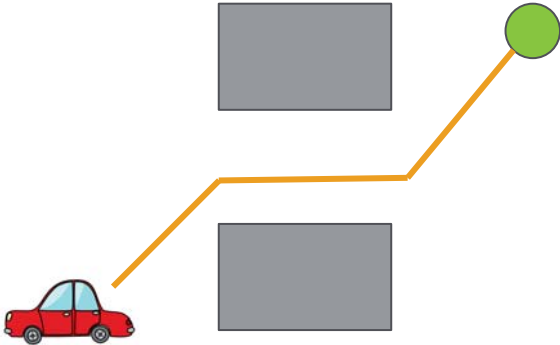
This image in the public domain.



Reference: Lecture given by Michal Čáp in 2.166 and paper currently in review

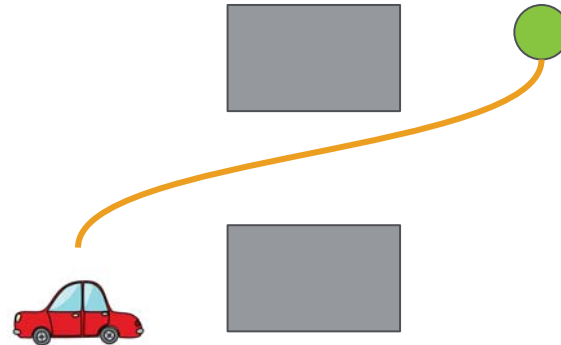
Holonomic System

- No differential constraints



Nonholonomic System

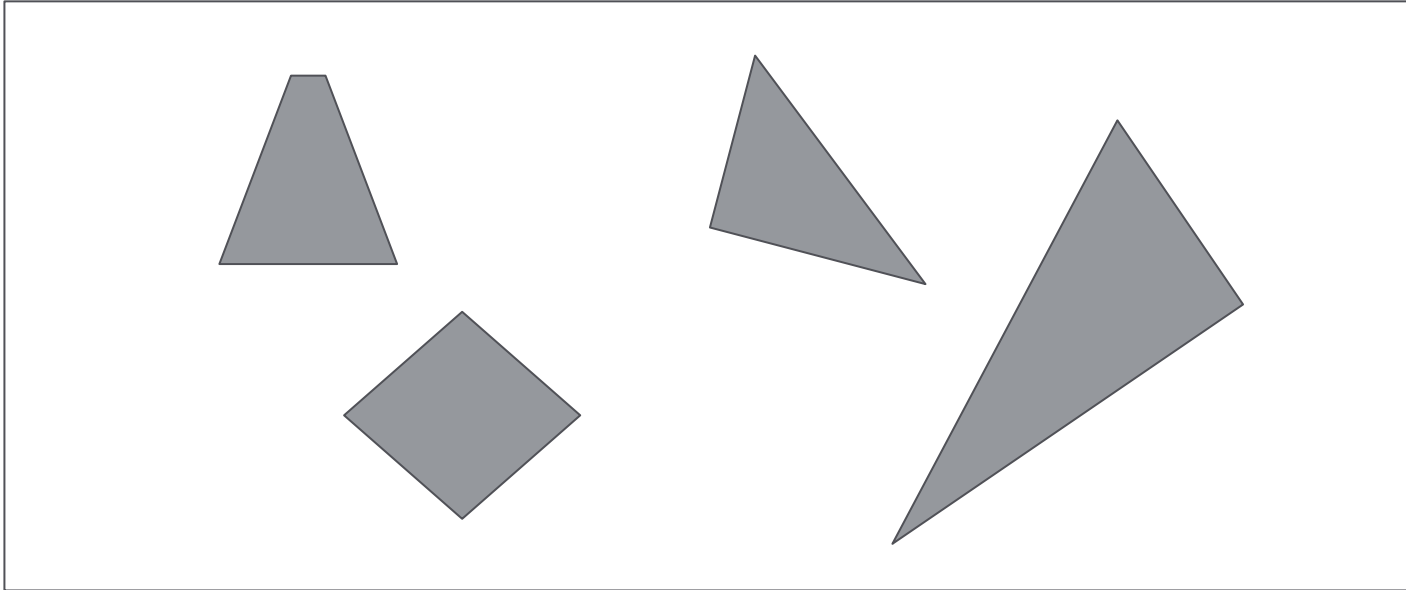
- Differential constraints



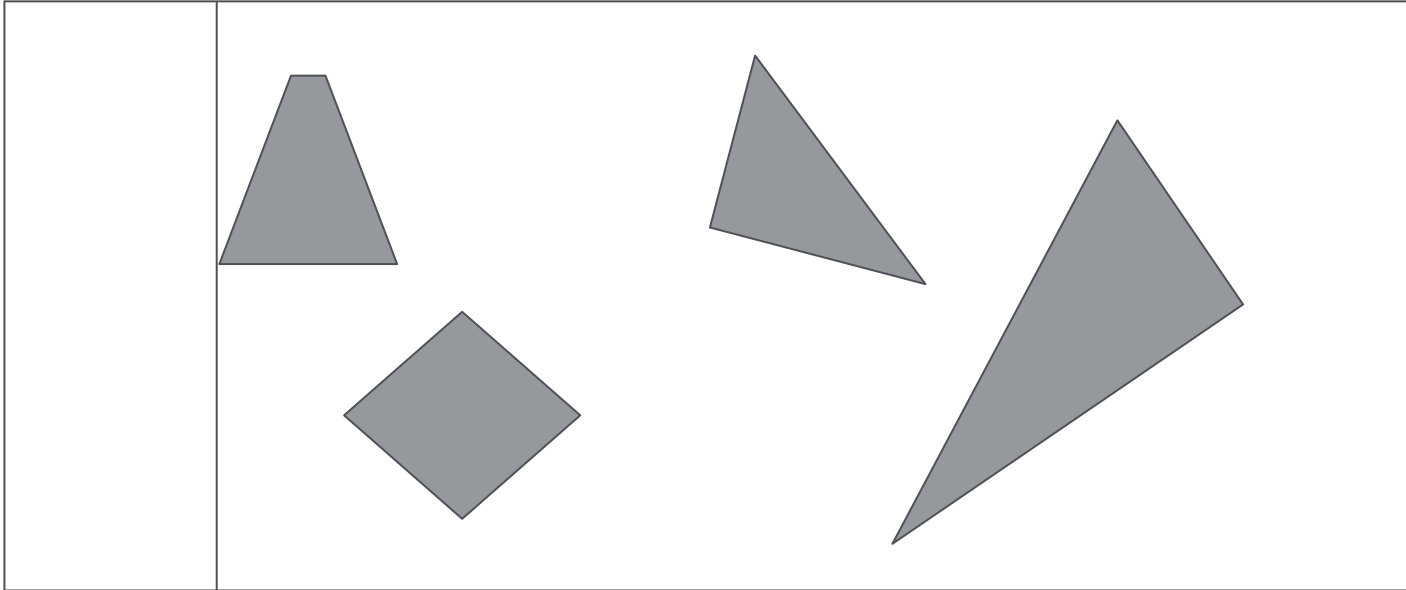
Methods

- Cell decomposition
- Visibility graph
- Sampling-based roadmap construction

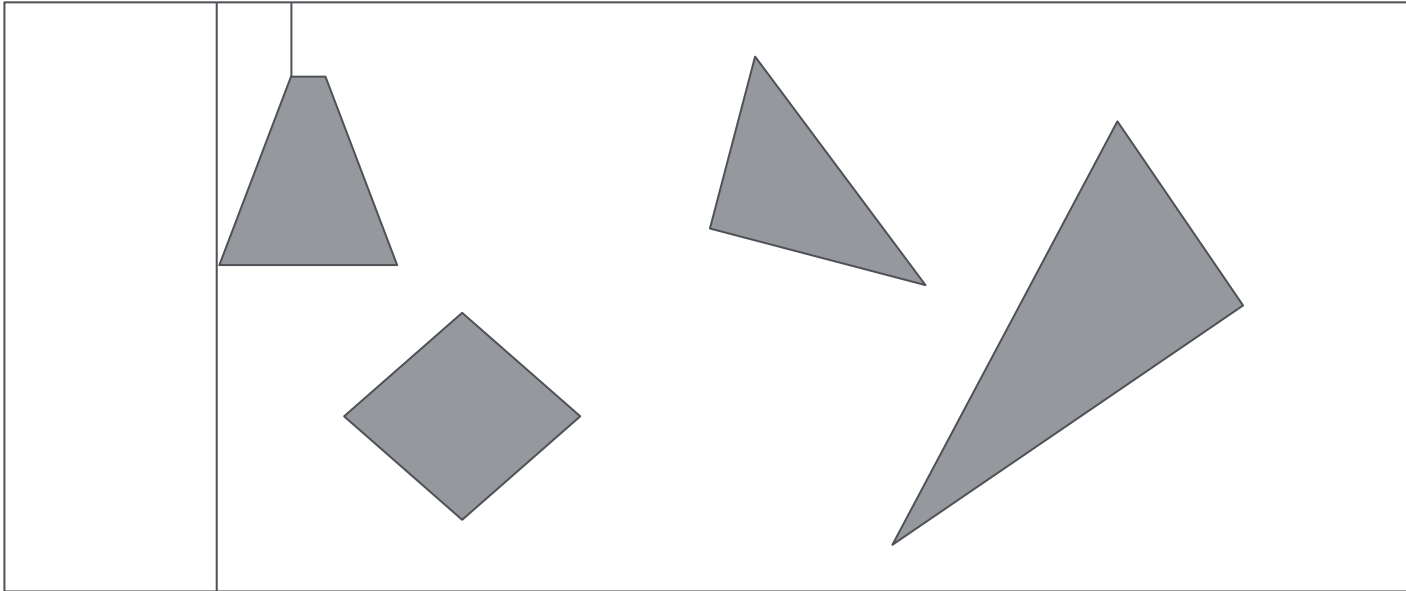
Cell Decomposition



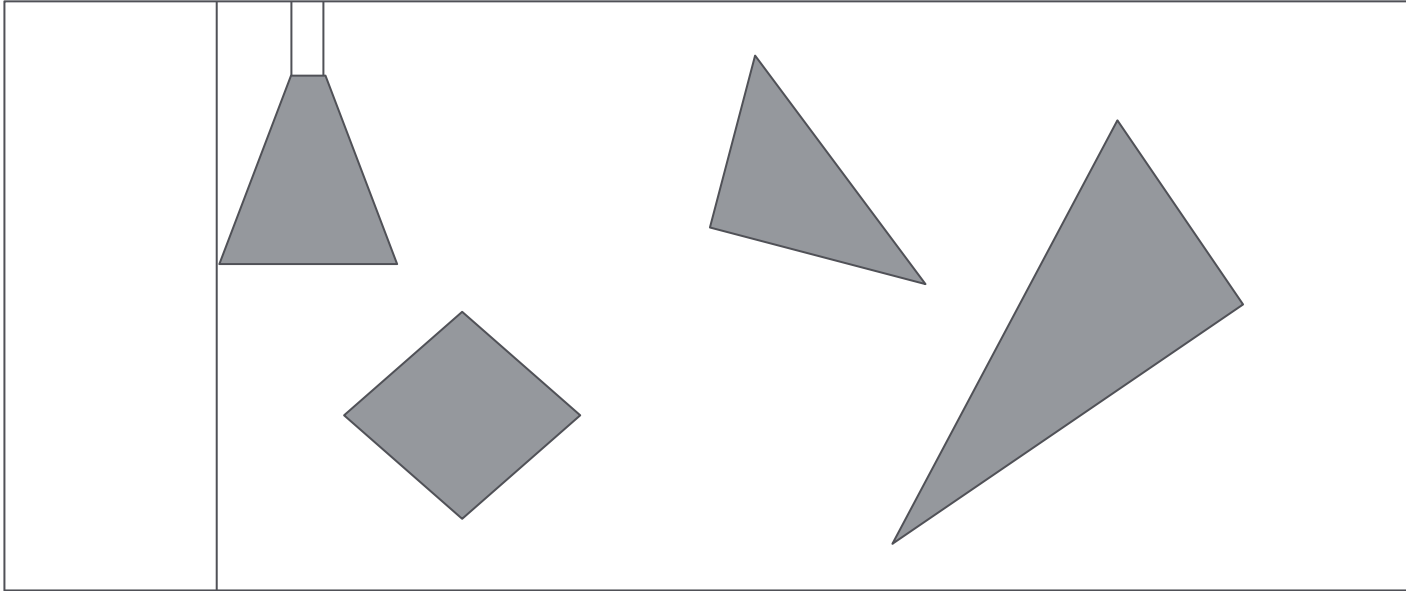
Cell Decomposition



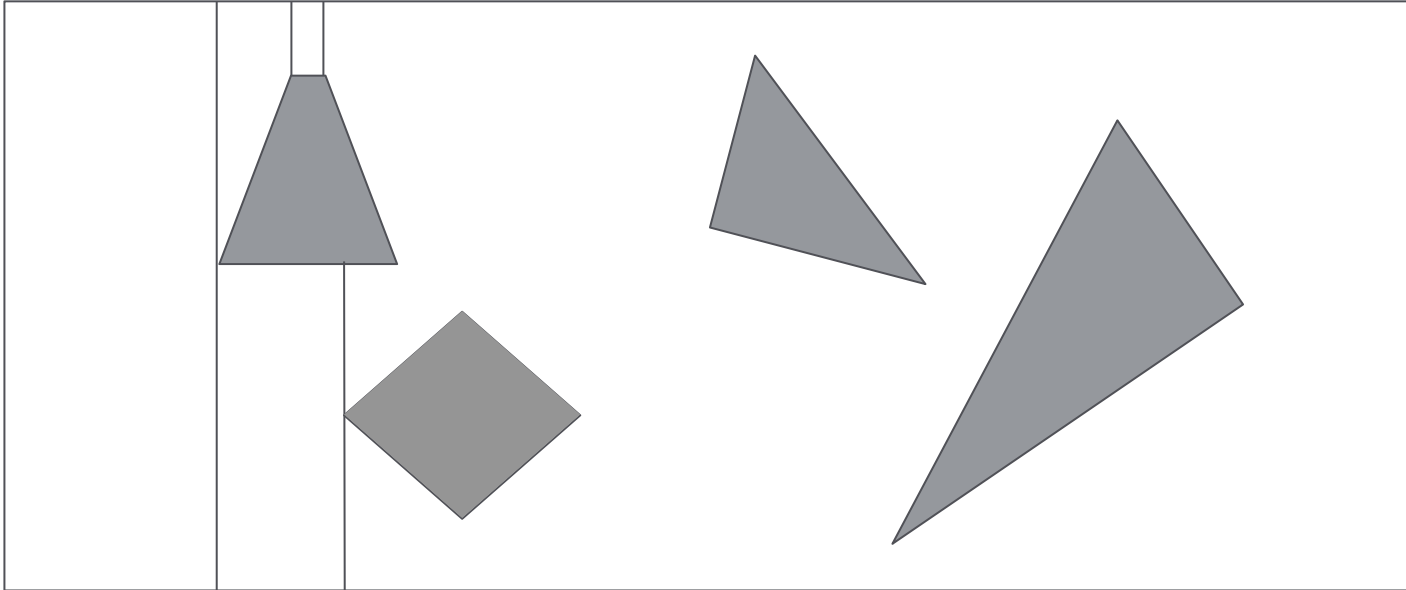
Cell Decomposition



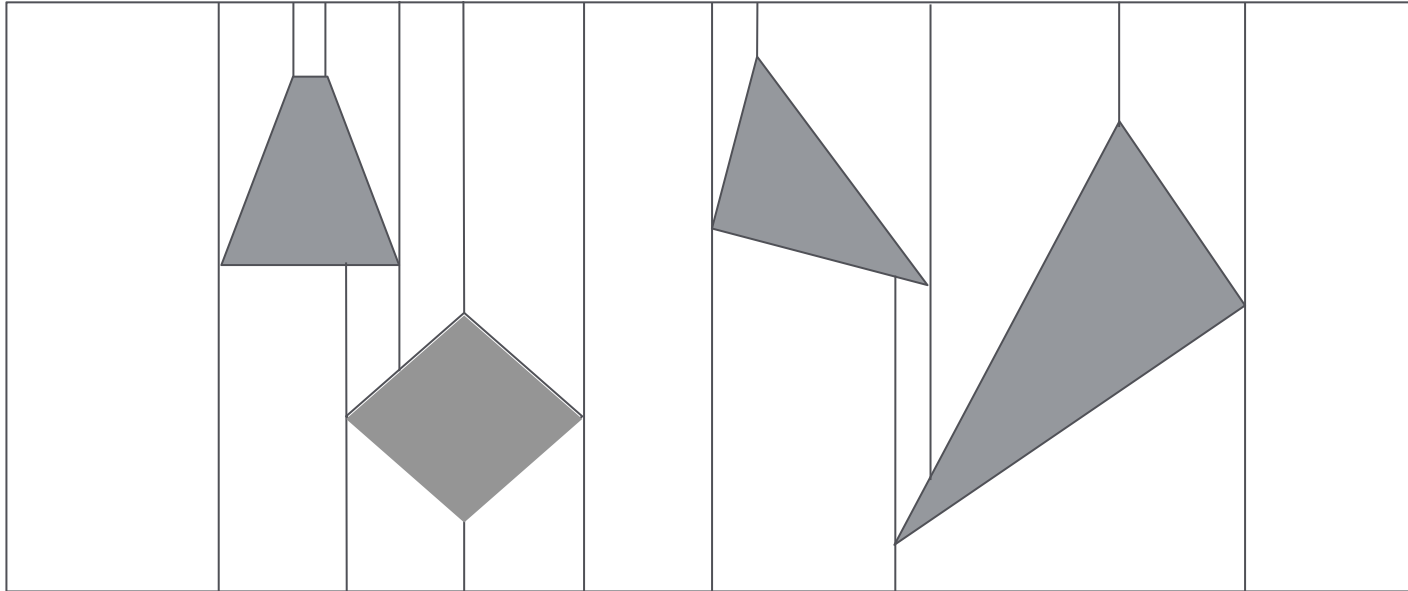
Cell Decomposition



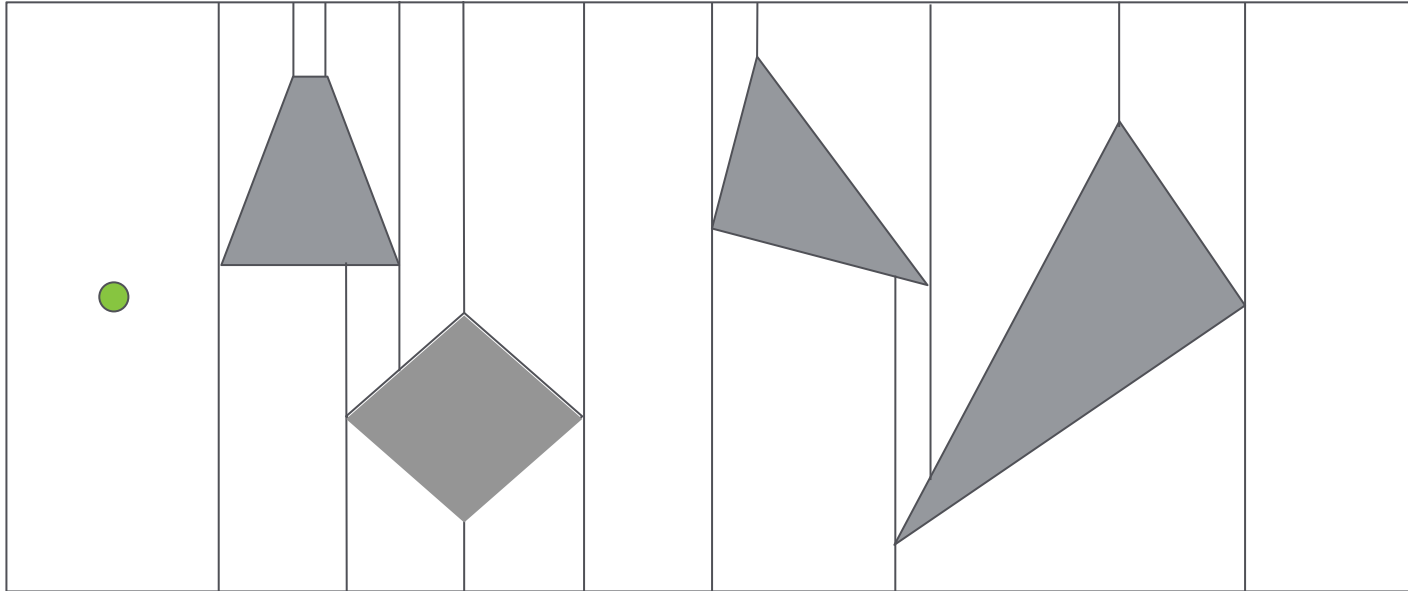
Cell Decomposition



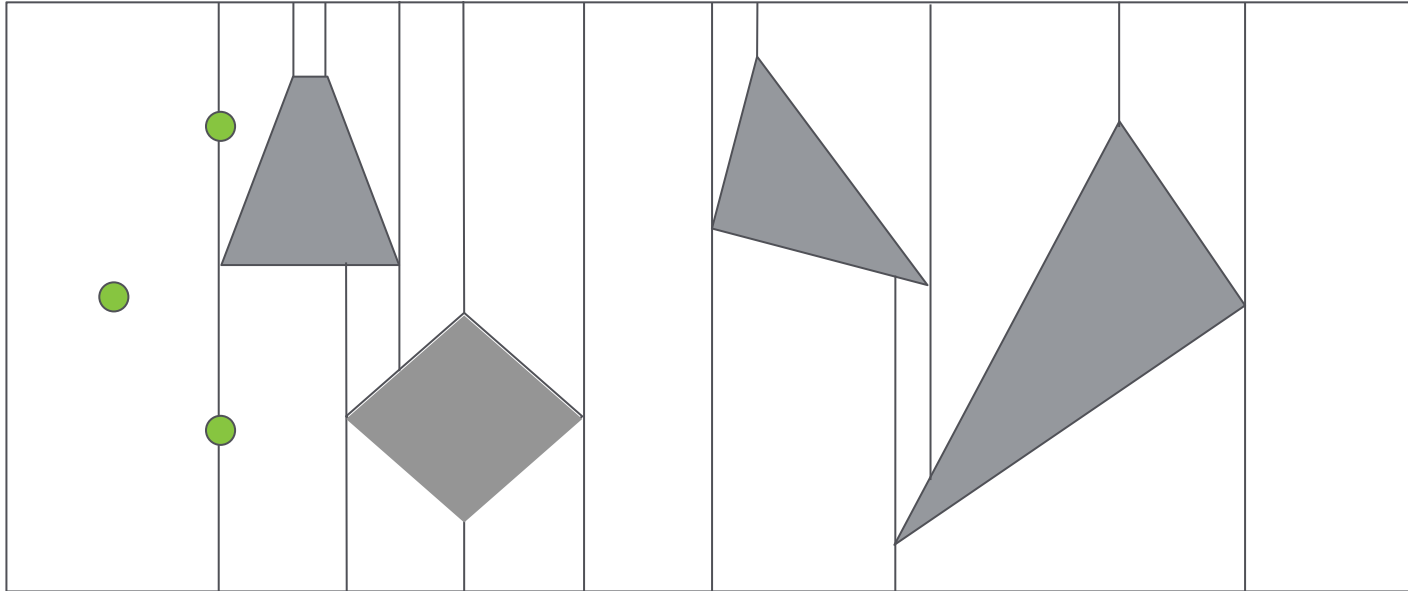
Cell Decomposition



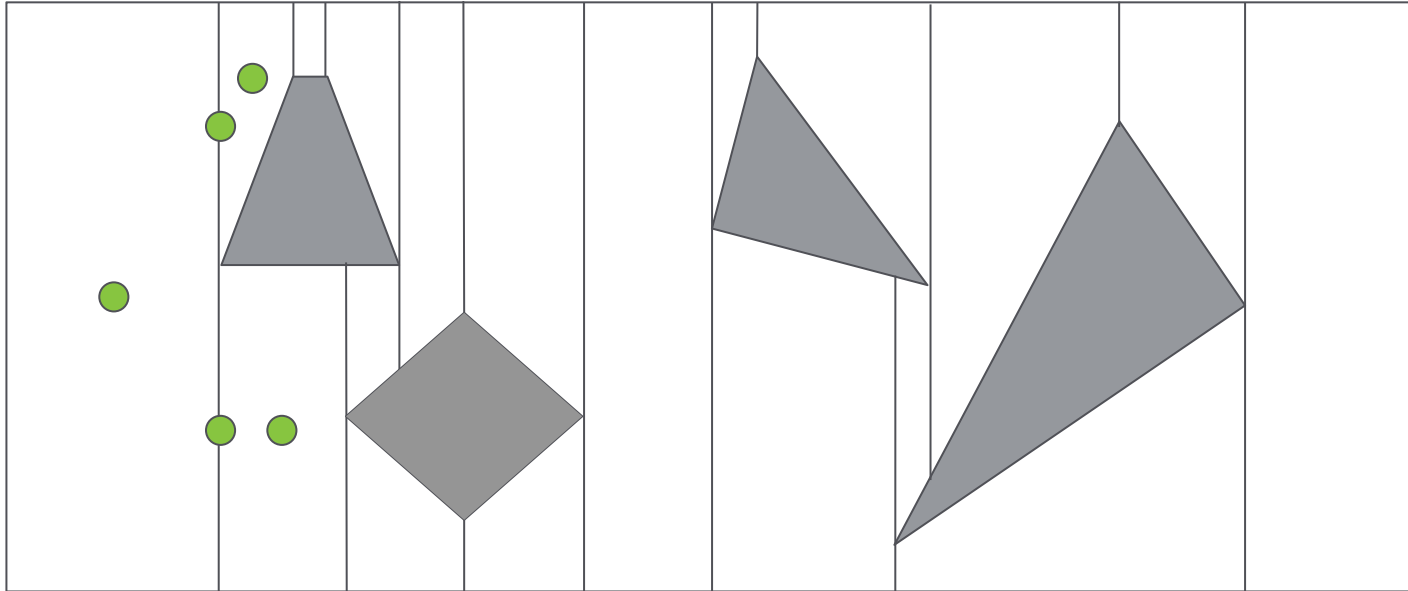
Cell Decomposition



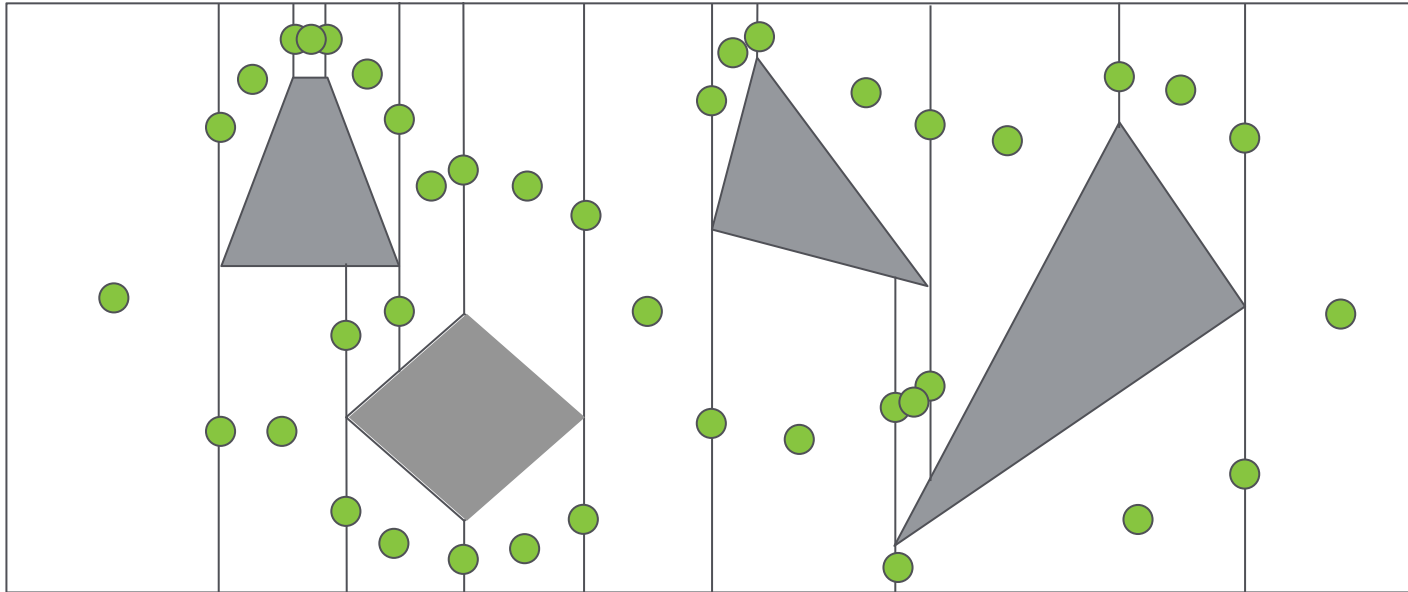
Cell Decomposition



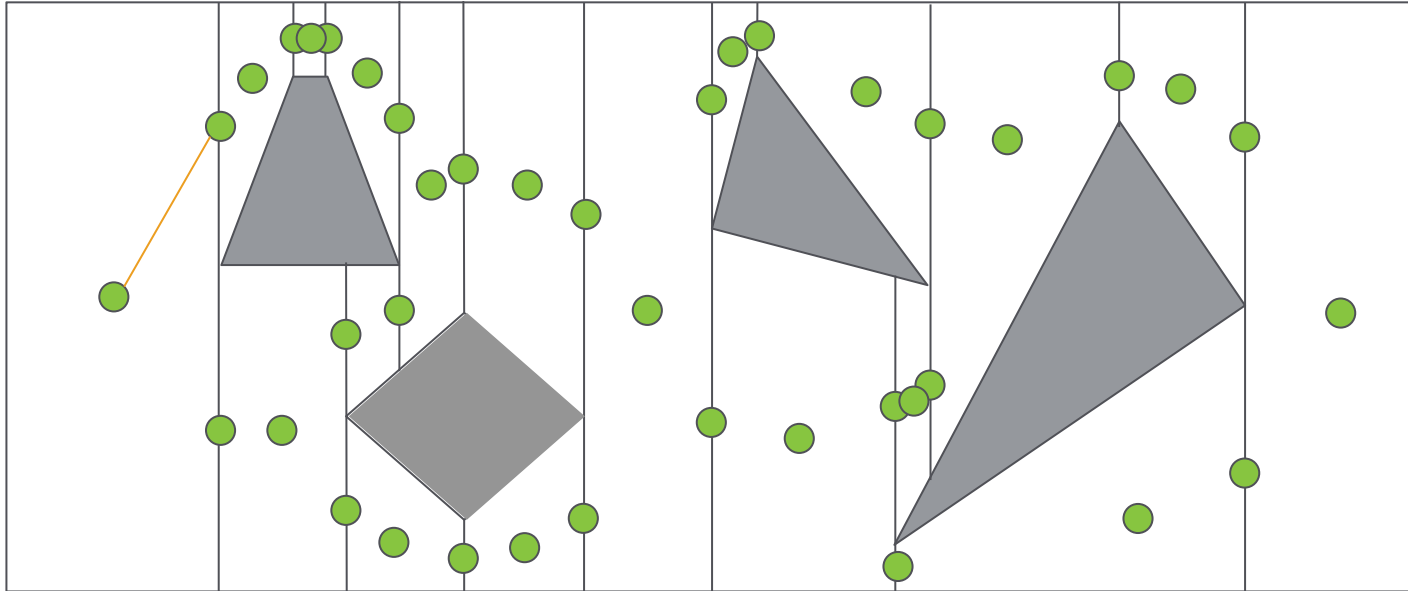
Cell Decomposition



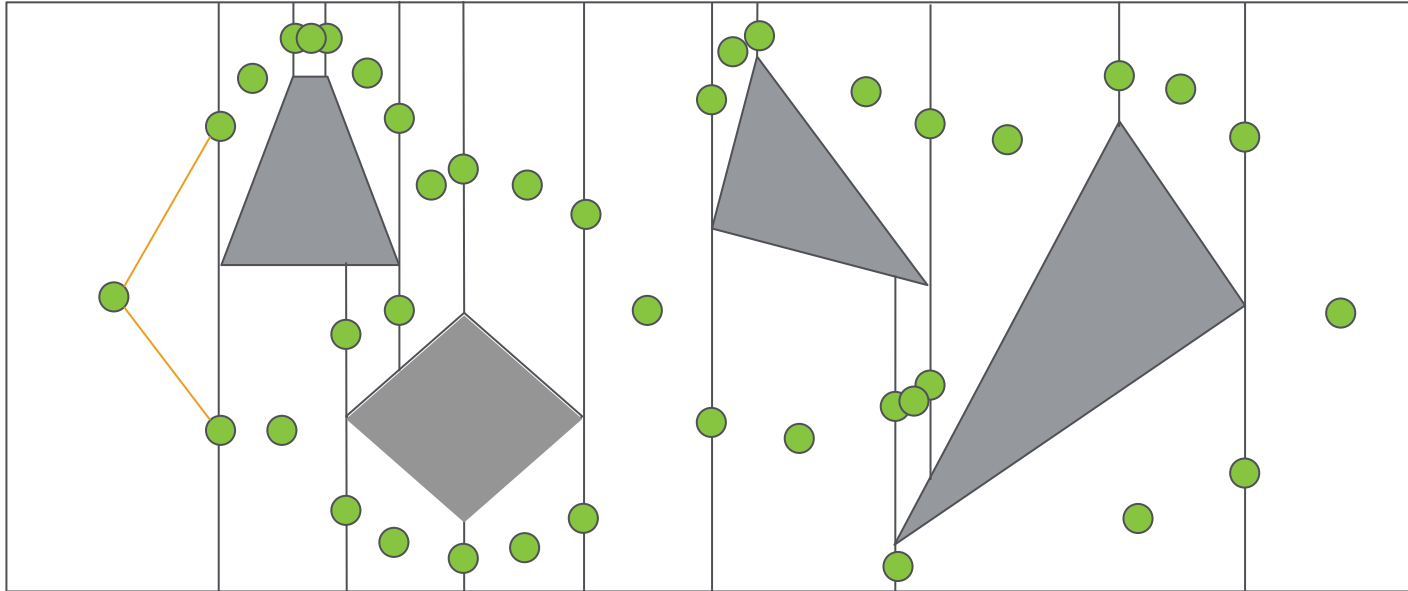
Cell Decomposition



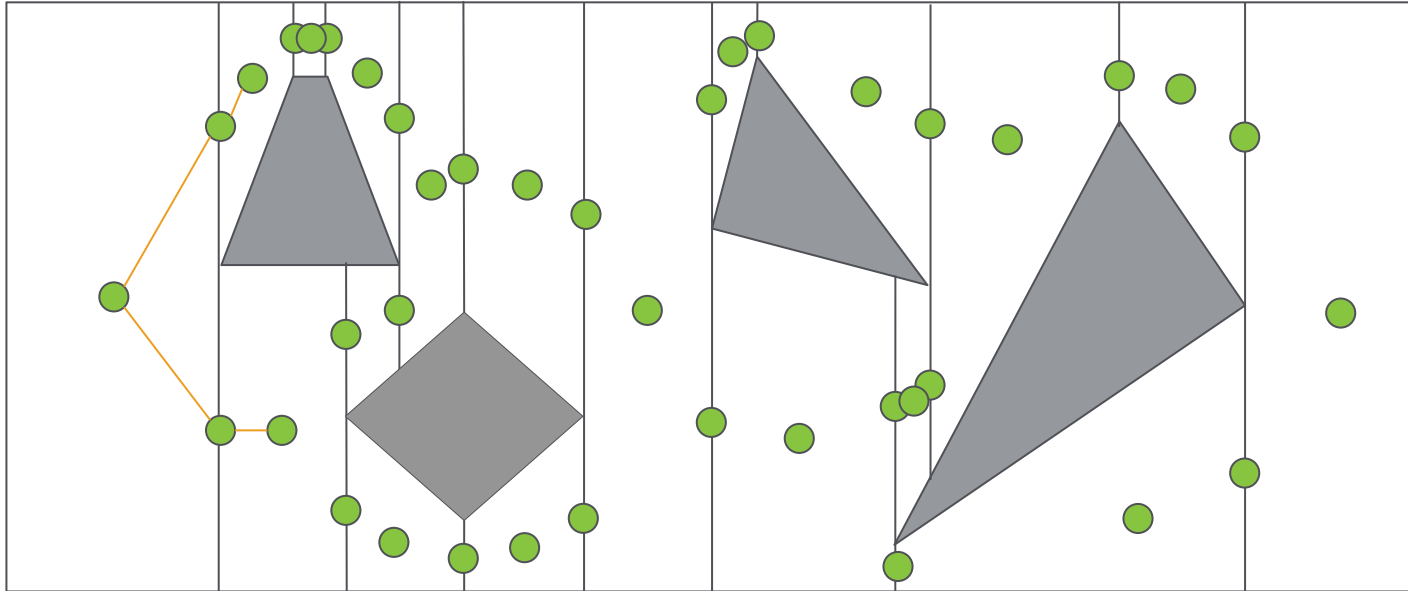
Cell Decomposition



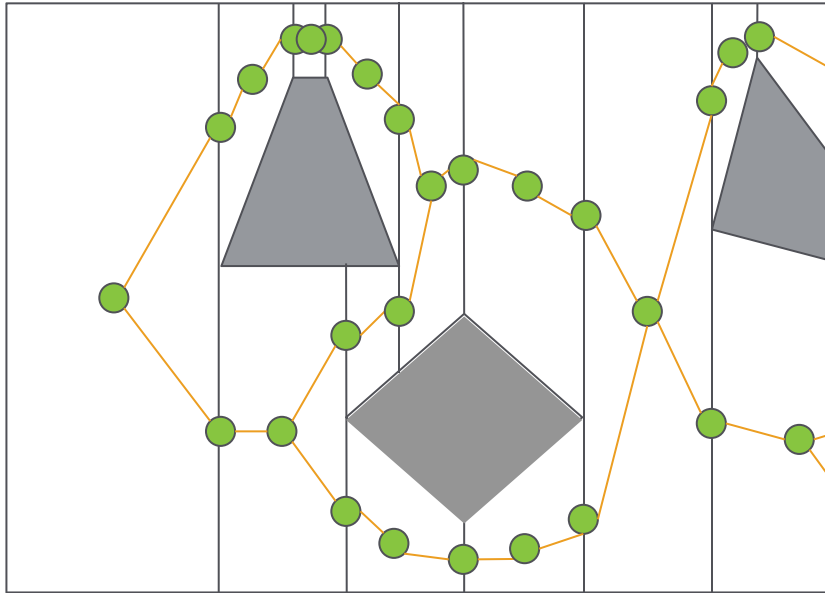
Cell Decomposition



Cell Decomposition

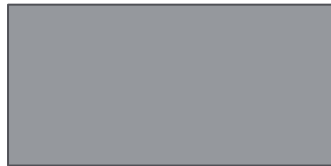


Cell Decomposition

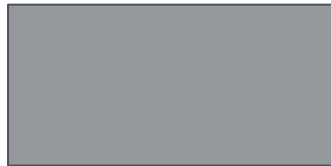
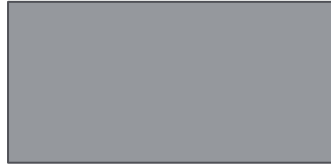


- Works in environments where obstacles are 2D polygons
- Path is not optimal
- Only holonomic systems

Visibility Graph



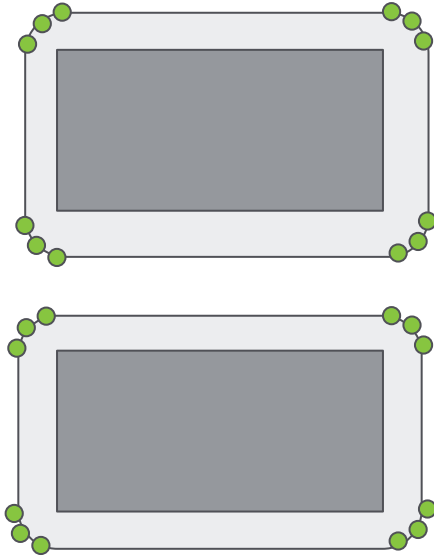
Visibility Graph



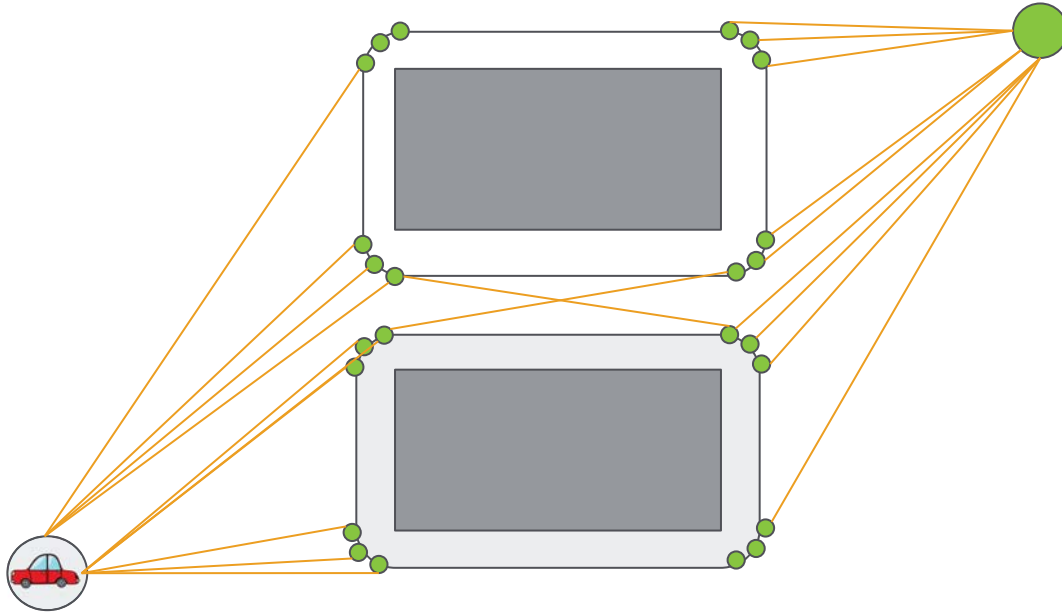
Visibility Graph



Visibility Graph

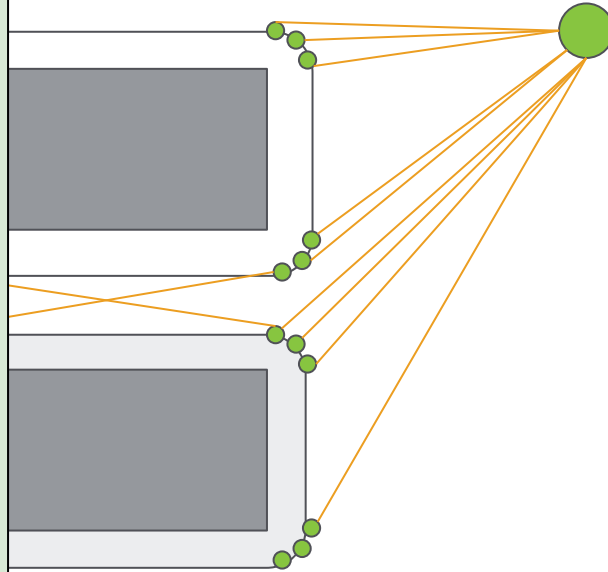


Visibility Graph



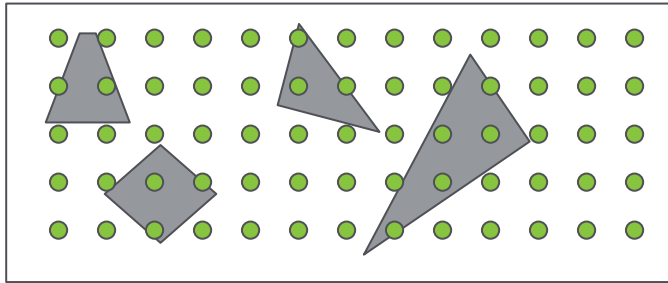
Visibility Graph

- Works in environments where obstacles are 2D polygons
- Only holonomic systems
- Optimal path
- Circular robot

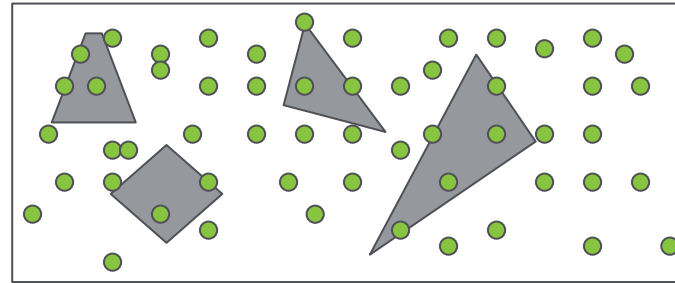


Sampling-based Roadmap Construction

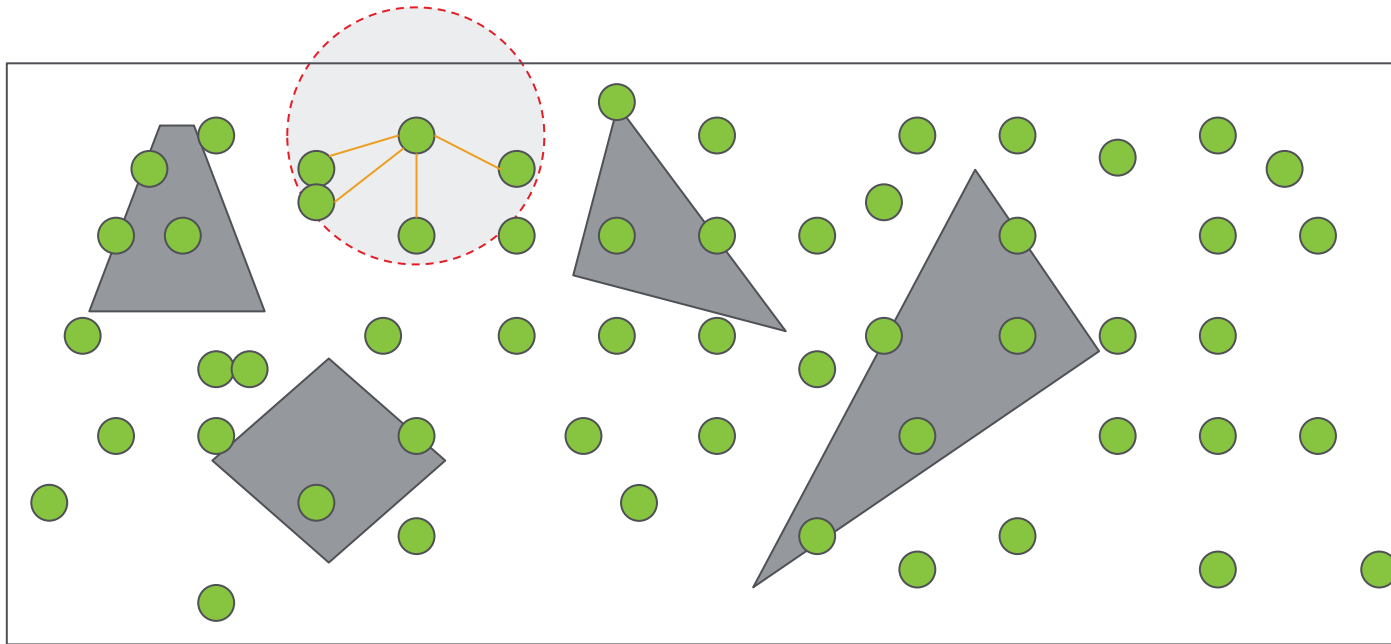
Deterministic sampling



Random sampling

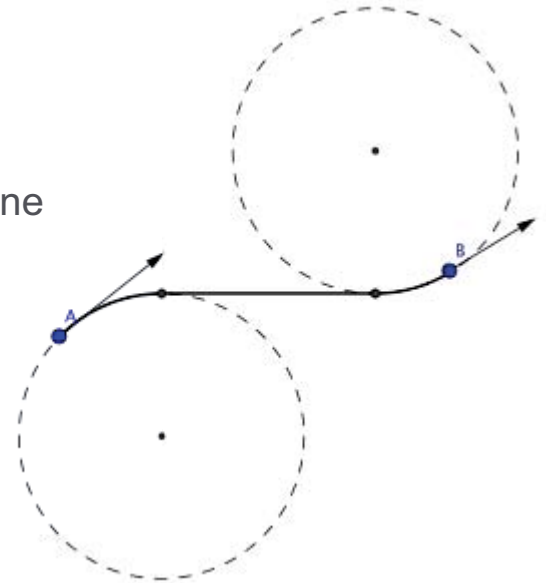


Sampling-based Roadmap Construction



Connecting Samples

- Steering function used
 - Steer(a, b) gives feasible path between samples
 - Nonholonomic
- Dubins path
 - Shortest path between two points
 - Can be constructed of maximum curvature and straight line segments
 - RSR, RSL, LSR, LSL, RLR or LRL



Outline

- Motivation
- Incremental Search
- The D* Lite Algorithm
- D* Lite Example
- When to Use Incremental Path Planning?
- Algorithm Extensions and Related Topics
- Application to Mobile Robotics



Backup

Incremental Path Planning

- Dynamic A* (D*) - Stentz, 1994
 - Initial combination of A* and incremental search for mobile robot path planning.
- DynamicsSWSF-FP – Ramalingam and Reps, 1996
 - Incremental search technique.
- Lifelong Planning A* (LPA*) – Koenig and Likhachev, 2001
 - Generalizes A* and DynamicsSWSF-FP for indefinite re-planning on finite graph.
- D* Lite – Koenig and Likhachev, 2002
 - Extends LPA* to provide D* functionality for mobile robots.
 - Simpler formulation than D*.
- More recent extensions to D* and D* Lite, including Anytime D* (AD*), Field D*, etc.
- Other methods...

D* Algorithm

Function: INSERT (X, h_{new})

```
L1 if  $t(X) = NEW$  then  $k(X) = h_{new}$ 
L2 else
L3   if  $t(X) = OPEN$  then
L4      $k(X) = MIN(k(X), h_{new})$ ;  $DELETE(X)$ 
L5   else  $k(X) = MIN(h(X), h_{new})$ 
L6    $h(X) = h_{new}$ ;  $r(X) = R_{curr}$ 
L7    $f(X) = k(X) + GVAL(X, R_{curr})$ ;  $f_B(X) = f(X) + d_{curr}$ 
L8    $PUT-STATE(X)$ 
```

Function: MIN-STATE ()

```
L1 while  $X = GET-STATE() \neq NULL$ 
L2   if  $r(X) \neq R_{curr}$  then
L3      $h_{new} = h(X)$ ;  $h(X) = k(X)$ 
L4      $DELETE(X)$ ;  $INSERT(X, h_{new})$ 
L5   else return X
L6 return NULL
```

The $MIN-VAL$ function, given below, returns the f^* and k^* values of the state on the $OPEN$ list with minimum f^* value, that is, $\langle f_{min}, k_{val} \rangle$.

Function: MIN-VAL ()

```
L1  $X = MIN-STATE()$ 
L2 if  $X = NULL$  then return  $NO-VAL$ 
L3 else return  $\langle f(X), k(X) \rangle$ 
```

Function: PROCESS-STATE ()

```
L1  $X = MIN-STATE()$ 
L2 if  $X = NULL$  then return  $NO-VAL$ 
L3  $val = \langle f(X), k(X) \rangle$ ;  $k_{val} = k(X)$ ;  $DELETE(X)$ 
L4 if  $k_{val} < h(X)$  then
L5   for each neighbor  $Y$  of  $X$ :
L6     if  $t(Y) \neq NEW$  and  $LESSEQ(COST(Y), val)$  and
L7        $h(X) > h(Y) + c(Y, X)$  then
L8        $b(X) = Y$ ;  $h(X) = h(Y) + c(Y, X)$ 
L9   if  $k_{val} = h(X)$  then
L10  for each neighbor  $Y$  of  $X$ :
L11   if  $t(Y) = NEW$  or
L12     ( $b(Y) = X$  and  $h(Y) \neq h(X) + c(X, Y)$ ) or
L13     ( $b(Y) \neq X$  and  $h(Y) > h(X) + c(X, Y)$ ) then
L14      $b(Y) = X$ ;  $INSERT(Y, h(X) + c(X, Y))$ 
L15 else
L16 for each neighbor  $Y$  of  $X$ :
L17   if  $t(Y) = NEW$  or
L18     ( $b(Y) = X$  and  $h(Y) \neq h(X) + c(X, Y)$ ) then
L19      $b(Y) = X$ ;  $INSERT(Y, h(X) + c(X, Y))$ 
L20   else
L21     if  $b(Y) \neq X$  and  $h(Y) > h(X) + c(X, Y)$  and
L22        $t(X) = CLOSED$  then
L23        $INSERT(X, h(X))$ 
L24   else
L25     if  $b(Y) \neq X$  and  $h(X) > h(Y) + c(Y, X)$  and
L26        $t(Y) = CLOSED$  and
L27        $LESS(val, COST(Y))$  then
L28        $INSERT(Y, h(Y))$ 
L29 return  $MIN-VAL()$ 
```

Function: MODIFY-COST (X, Y, c_{val})

```
L1  $c(X, Y) = c_{val}$ 
L2 if  $t(X) = CLOSED$  then  $INSERT(X, h(X))$ 
L3 return  $MIN-VAL()$ 
```

Function: MOVE-ROBOT (S, G)

```
L1 for each state  $X$  in the graph:
L2    $t(X) = NEW$ 
L3    $d_{curr} = 0$ ;  $R_{curr} = S$ 
L4    $INSERT(G, 0)$ 
L5    $val = \langle 0, 0 \rangle$ 
L6   while  $t(S) \neq CLOSED$  and  $val \neq NO-VAL$ 
L7      $val = PROCESS-STATE()$ 
L8   if  $t(S) = NEW$  then return  $NO-PATH$ 
L9    $R = S$ 
L10  while  $R \neq G$ :
L11   if  $s(X, Y) \neq c(X, Y)$  for some  $(X, Y)$  then
L12     if  $R_{curr} \neq R$  then
L13        $d_{curr} = d_{curr} + GVAL(R, R_{curr}) + \epsilon$ ;  $R_{curr} = R$ 
L14     for each  $(X, Y)$  such that  $s(X, Y) \neq c(X, Y)$ :
L15        $val = MODIFY-COST(X, Y, s(X, Y))$ 
L16     while  $LESS(val, COST(R))$  and  $val \neq NO-VAL$ 
L17        $val = PROCESS-STATE()$ 
L18      $R = b(R)$ 
L19 return  $GOAL-REACHED$ 
```

LPA* Algorithm

procedure CalculateKey(s)

{01} return $[\min(g(s), rhs(s)) + h(s); \min(g(s), rhs(s))]$;

procedure Initialize()

{02} $U = \emptyset$;

{03} for all $s \in S$ $rhs(s) = g(s) = \infty$;

{04} $rhs(s_{start}) = 0$;

{05} $U.Insert(s_{start}, [h(s_{start}); 0])$;

procedure UpdateVertex(u)

{06} if $(u \neq s_{start})$ $rhs(u) = \min_{s' \in pred(u)} (g(s') + c(s', u))$;

{07} if $(u \in U)$ $U.Remove(u)$;

{08} if $(g(u) \neq rhs(u))$ $U.Insert(u, CalculateKey(u))$;

procedure ComputeShortestPath()

{09} while $(U.TopKey() < CalculateKey(s_{goal})$ OR $rhs(s_{goal}) \neq g(s_{goal})$)

{10} $u = U.Pop()$;

{11} if $(g(u) > rhs(u))$

{12} $g(u) = rhs(u)$;

{13} for all $s \in succ(u)$ $UpdateVertex(s)$;

{14} else

{15} $g(u) = \infty$;

{16} for all $s \in succ(u) \cup \{u\}$ $UpdateVertex(s)$;

procedure Main()

{17} Initialize();

{18} forever

{19} ComputeShortestPath();

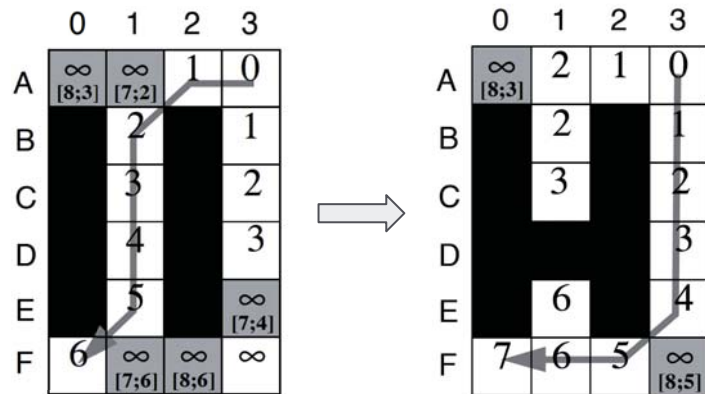
{20} Wait for changes in edge costs;

{21} for all directed edges (u, v) with changed edge costs

{22} Update the edge cost $c(u, v)$;

{23} UpdateVertex(v);

From Koenig, Likhachev, & Furcy (2004)



Courtesy of Elsevier, Inc., <http://www.sciencedirect.com>. Used with permission.

Source: Figures 3 and 4 in Keonig, Sven, M. Likhachev, and D. Furcy.

"Lifelong Planning A*." In Artificial Intelligence, 155 (1-2): 93-146.

Review of A*

- Use admissible heuristic, $h(s) \leq h^*(s)$, to guide search.
- Keep track of total cost/distance from start, $g(s)$.
- Order node expansions using **priority queue**, with priorities $f(s) = g(s) + h(s)$.
- Avoid re-expanding nodes by using expanded list.
- Better heuristics (how closely $h(s)$ approximates of $h^*(s)$) improve search speed.
- Guaranteed to return optimal solution if one exists.

RHS Values

- One-step look-ahead on g-values, $\text{rhs}(s) = 0$ if s is beginning node of search, otherwise:

$$\text{rhs}(s) = \min_{s' \in \text{pred}(s)} (g(s') + c(s', s))$$

- Potentially better informed than g-value after changes to search graph.
- Note: term comes from grammar rules used in DynamicsSWSF-FP algorithm, no other significance.

Local Consistency

- Tells us which nodes may need g-values updated in order to find shortest path.
- Node s is locally consistent iff:
$$g(s) = rhs(s)$$
- Node s is locally overconsistent iff:
$$g(s) > rhs(s)$$
- Node s is locally underconsistent iff:
$$g(s) < rhs(s)$$
- Initially, all nodes are locally consistent with $g(s) = rhs(s) = \infty$, with exception of start node, $rhs(s_{start}) = 0$ and $g(s_{start}) = \infty$

Comparison of Incremental Path Planning to A*

Similarities:

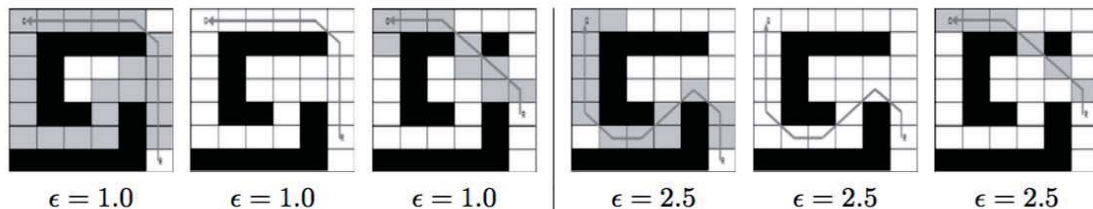
- First search expands same nodes in same order as A*, if A* breaks ties in favor of smaller g-values.

Differences:

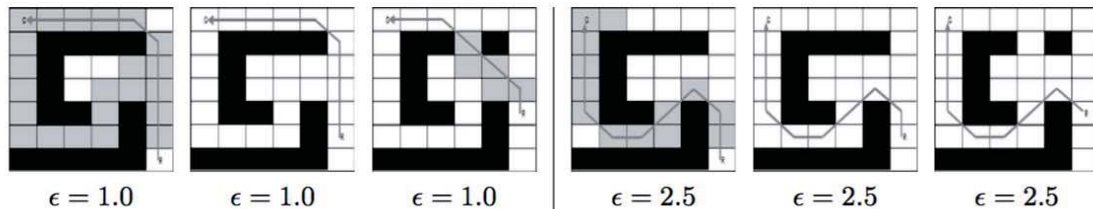
- Priority queue ordered using key, $k(s)$:
 - $k(s) = [k_1(s); k_2(s)]$
 - $k_1(s) = f(s) = \min(g(s), \text{rhs}(s)) + h(s)$
 - $k_2(s) = g(s) = \min(g(s), \text{rhs}(s))$
 - Lexicographic ordering, $k(s) < k'(s)$ iff:
 - $k_1(s) < k_1'(s)$
 - OR $(k_1(s) = k_1'(s) \text{ AND } k_2(s) < k_2'(s))$
- No expanded list, node re-expansion prevented by local consistency checks.
- Nodes may be expanded twice, depending on algorithm specifics, once when underconsistent and once when overconsistent.

Anytime Dynamic A* (AD*)

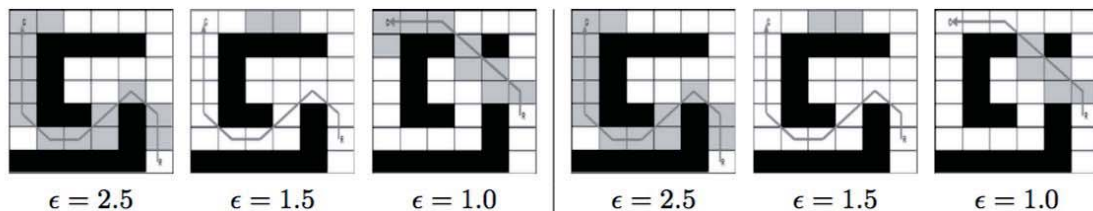
left: A*
right: A* with $\epsilon = 2.5$



left: D* Lite
right: D* Lite with $\epsilon = 2.5$



left: ARA*
right: Anytime Dynamic A*



© American Association for Artificial Intelligence. All rights reserved.
This content is excluded from our Creative Commons license. For more information, see <https://ocw.mit.edu/help/faq-fair-use/>.

Fig: Likhachev, M., Ferguson, D. I., Gordon, G. J., Stentz, A., & Hrun, S. (2005, June). Anytime Dynamic A*: An Anytime, Replanning Algorithm. In *ICAPS*(pp. 262-271).

MIT OpenCourseWare
<https://ocw.mit.edu>

16.412J / 6.834J Cognitive Robotics
Spring 2016

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>.