MEAM 620

GRAPH SEARCH

Penn Engineering | GRASP Laboratory
General Robotics, Automation, Sensing & Perception Lab

What we'll Cover Today

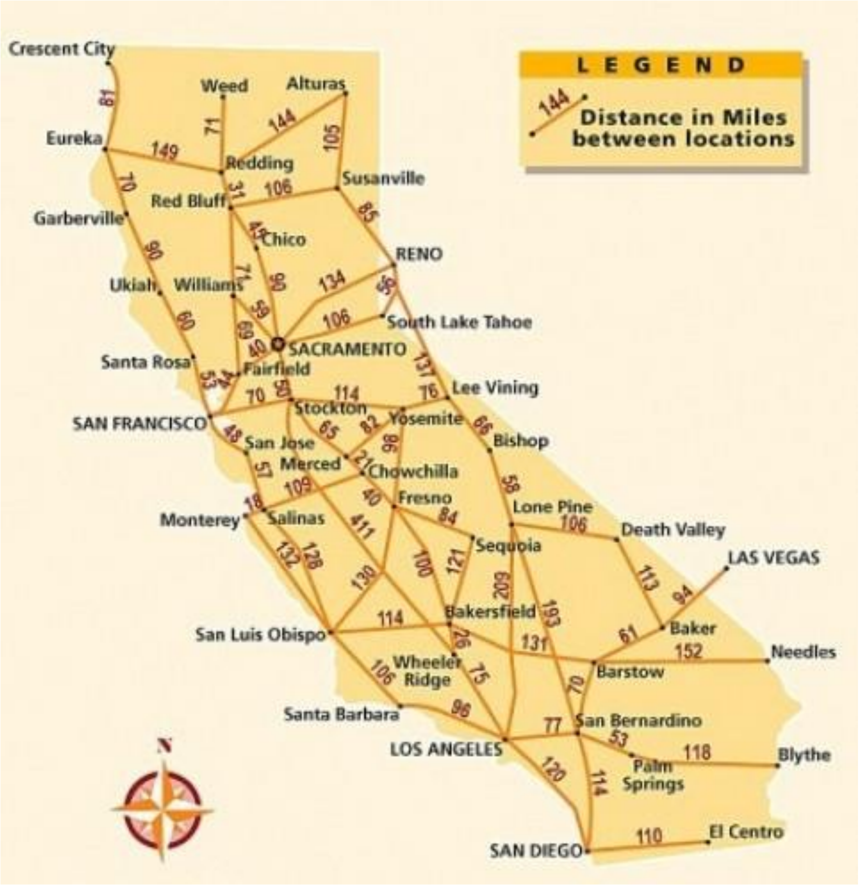
- Search-based motion planning.
- Graph search algorithms
  - Dijkstra
  - A*
  - Jump-point search

Continuous Motion Planning

Goal: Plan collision-free trajectories through cluttered space.

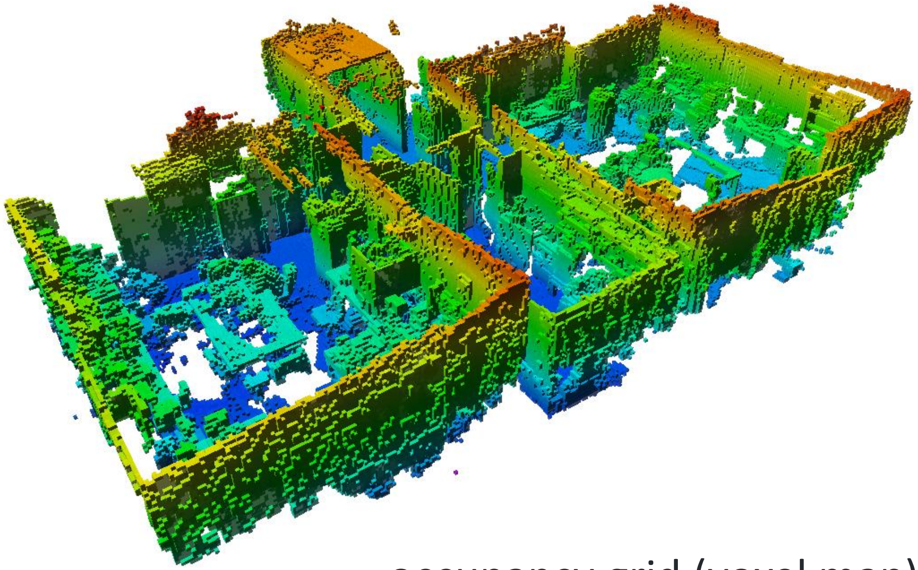
Planning on Graphs

Goal: Find shortest path between two nodes on the graph.



Discrete Approximations of Free Space

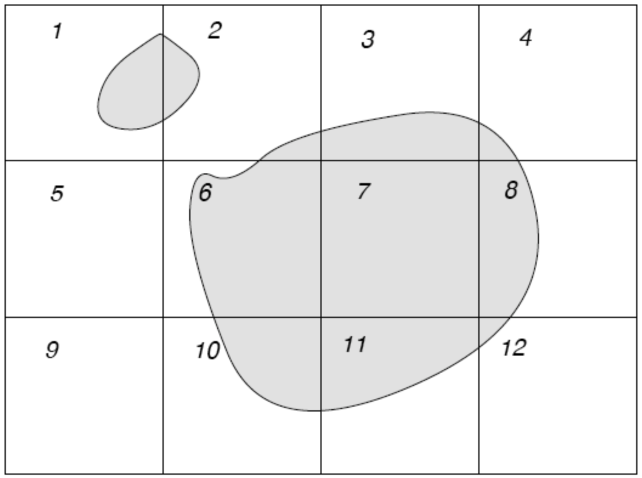
point cloud (sensor data)

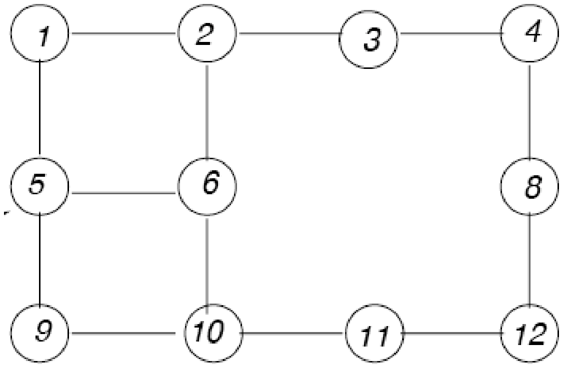


occupancy grid (voxel map)

City University of New York

Approximate Cellular Decompositions

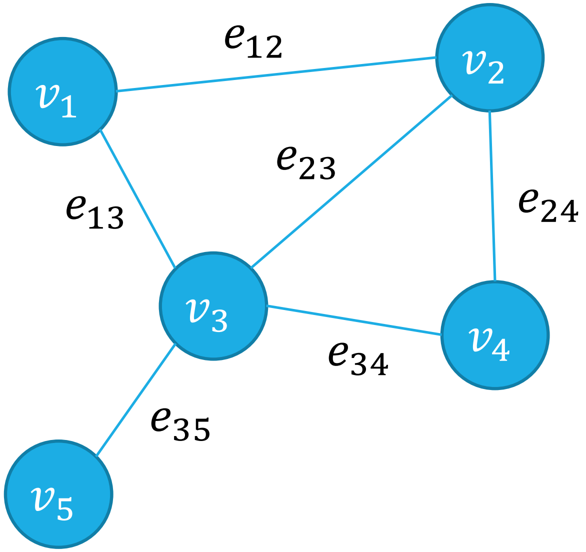




Graph

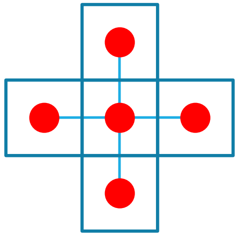
- A **graph** is an ordered pair $G = (V, E)$, where V is a set of vertices or nodes and E is a set of edges

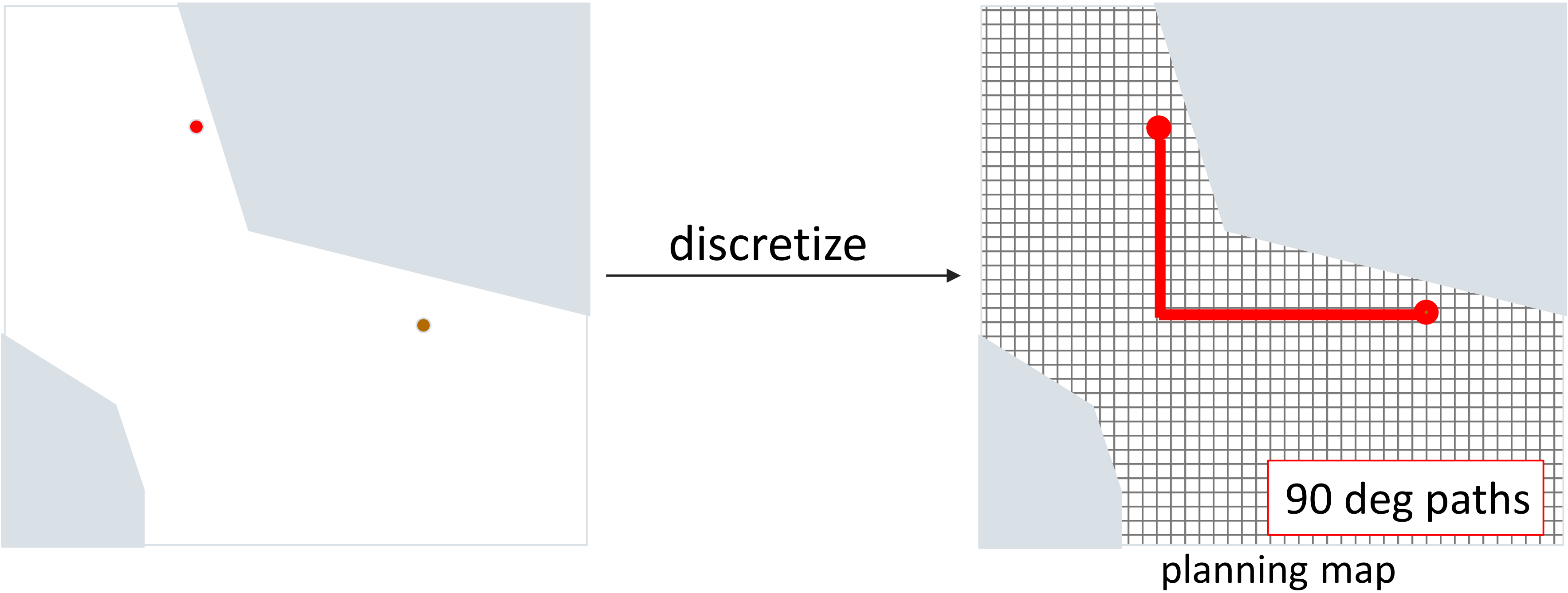
- An **edge** is a 2-tuple of vertices
  - Edges can be directed or undirected
  - <u>Edges can be weighted</u>
    - Edge e_{ij} has weight w_{ij}



4-Connected Graphs

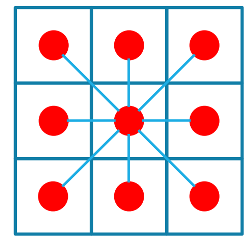
Each node is connected to 4 neighbors

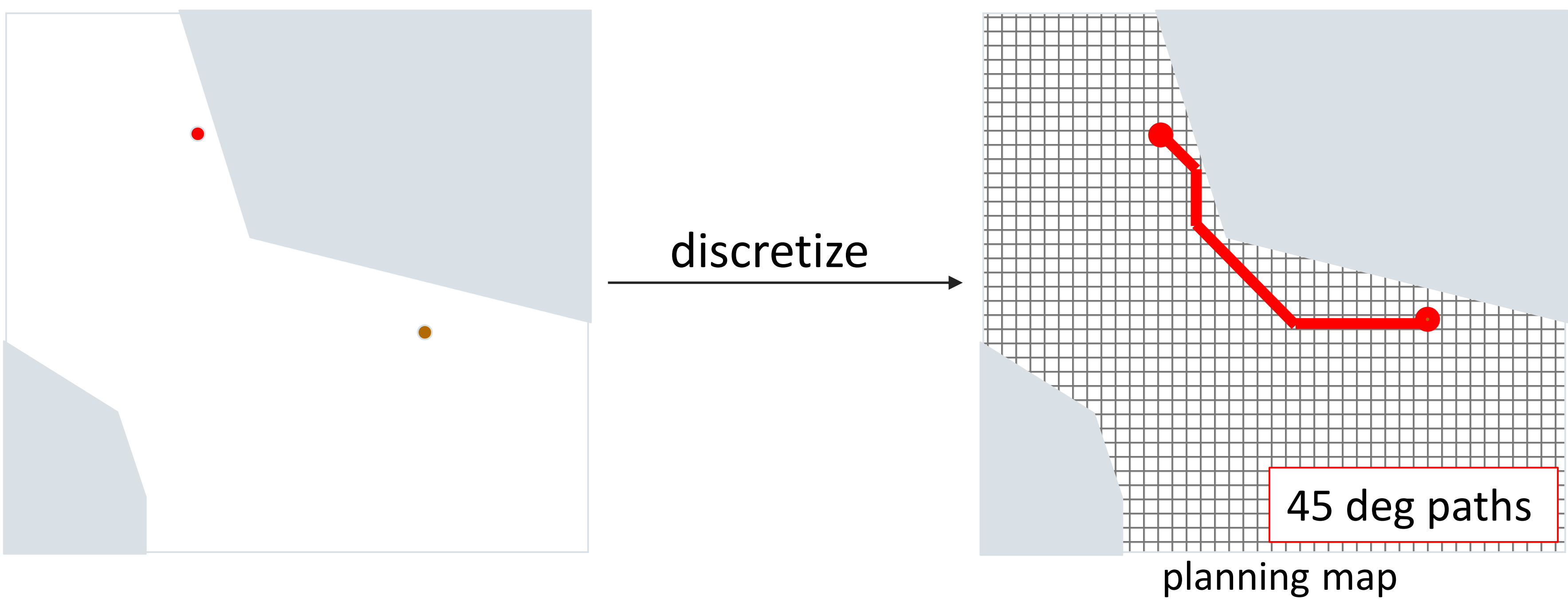




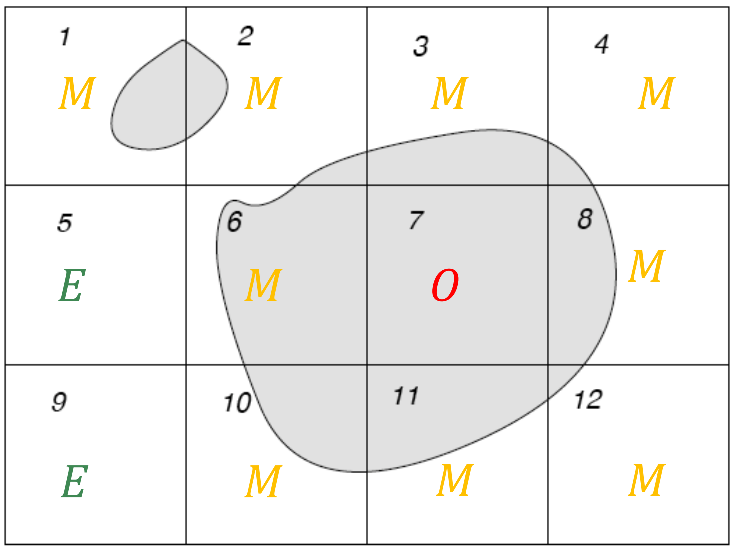
8-Connected Graphs

Each node is connected to 8 neighbors





Partially Blocked Cells

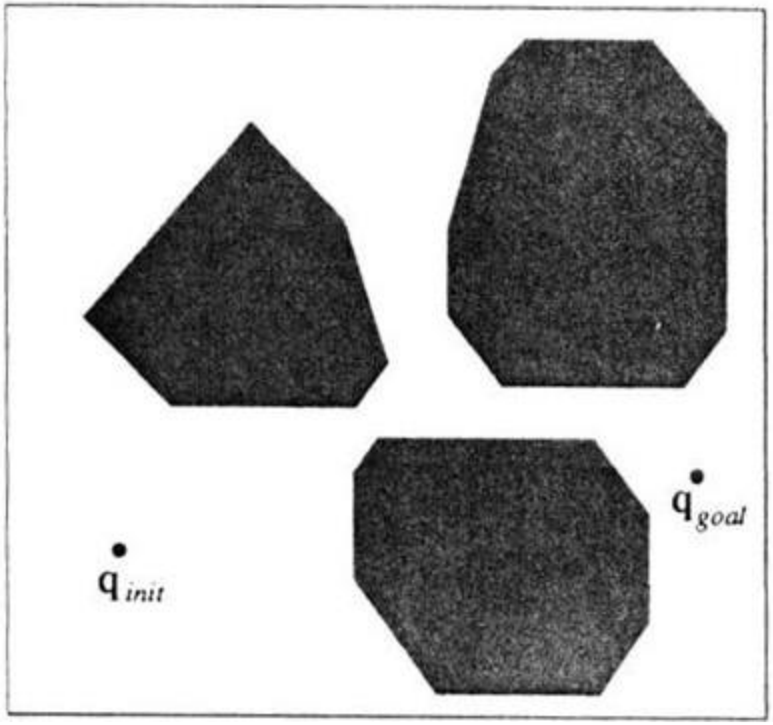


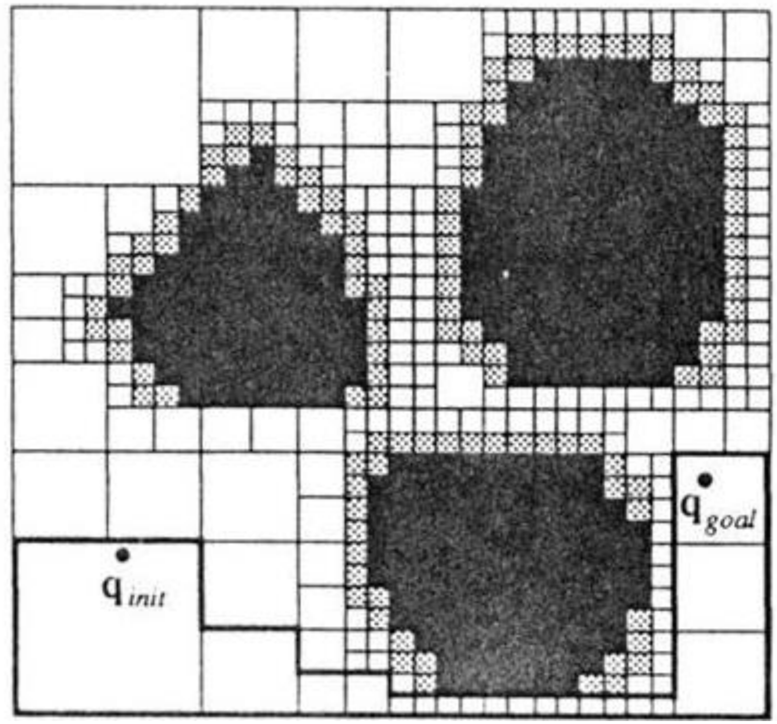
E – empty
M – mixed
O – occupied

1. Make it untraversable
   - Incomplete – may not find a path that exists
2. Make it traversable
   - Incorrect – may return a valid path where there is none
3. Increase grid resolution
   - Expensive – especially in high dimensions
4. Make discretization adaptive
   - Lose uniform grid size

Example by Nancy Amato, Texas A&M

Partially Blocked Cells





Search Algorithms

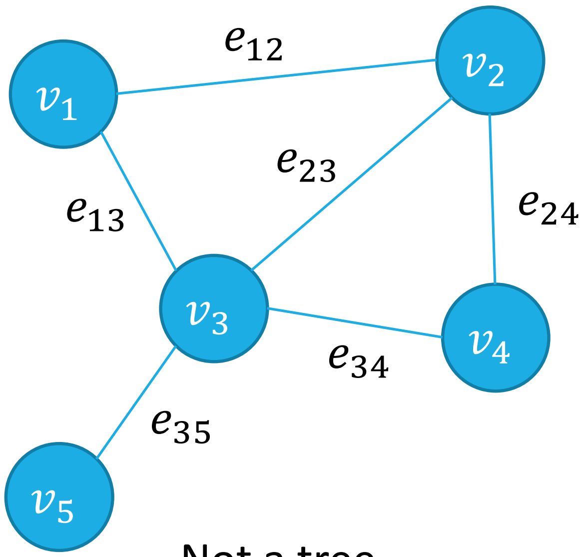
- Cellular decompositions yield a discrete representation of the configuration space in the form of a **weighted graph**.

- We need efficient graph search techniques that allow us to compute online motion/path plans of **least cost** from an initial to a final node

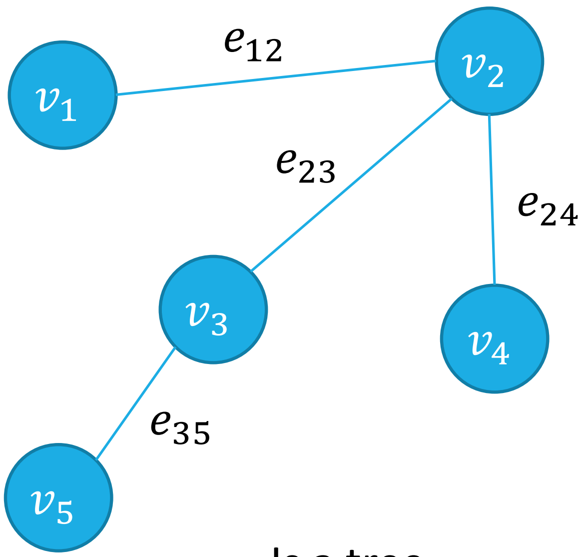
Trees

A **tree** is an undirected graph where any 2 vertices are connected by *exactly* 1 path

- The graph does not have cycles



Not a tree

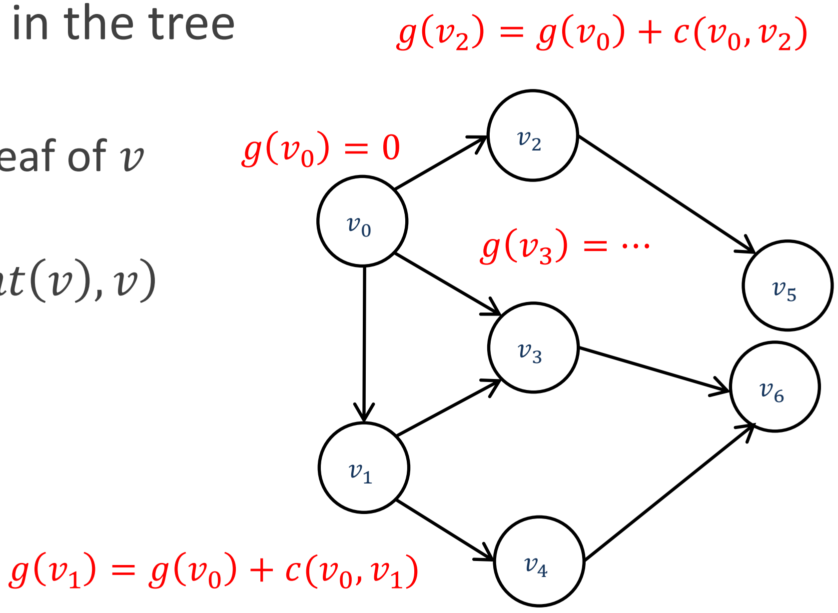


Is a tree

Tree-Based Search

General search strategy for finding a path from start to goal, and keeping track of it's length given edge costs $c(v_1, v_2)$.

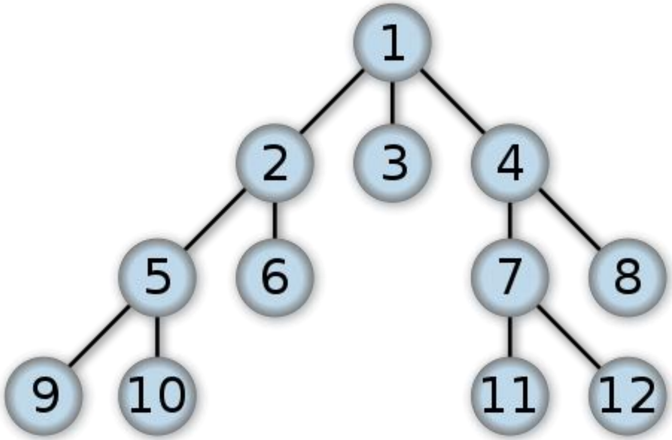
1. Set the root of the tree as the start state and give it a value of 0
2. While there are unexpanded nodes in the tree
   1. Choose a leaf v to expand
   2. For each action, create a new child leaf of v
   3. Set the value of each child leaf as
      $g(v) = g\big(parent(v)\big) + c(parent(v), v)$



What Action to Choose?

Breadth First

- Choose shallowest node
- Guaranteed optimality
  - (if uniform edge costs)
- Storage intensive

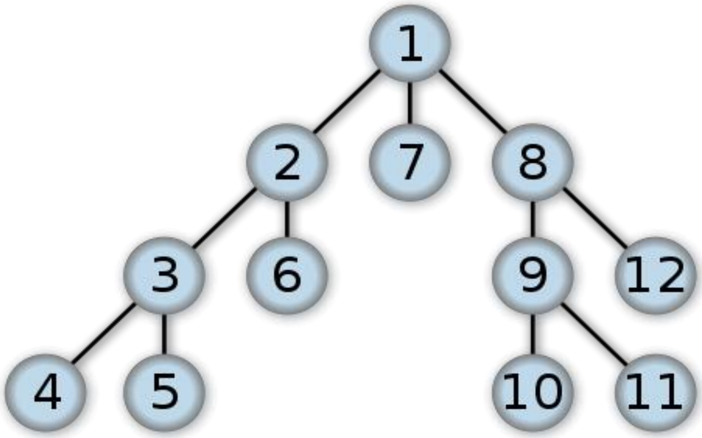


Ref: Wikipedia

What Action to Choose?

Depth First

- Choose deepest next
- No optimality
- Potentially storage cheap
  - (if graph is tree-like or very constrained)

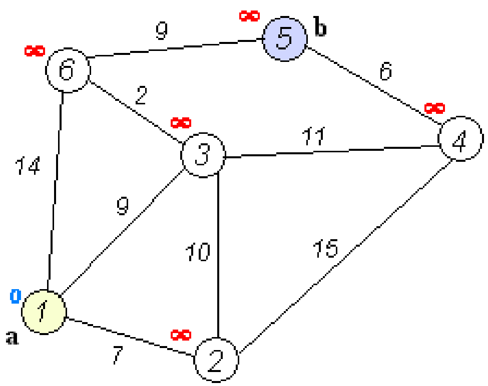


Ref: Wikipedia

What Action to Choose?

Best First

- Dijkstra (1959)
- A^* (Hart 1968)
- Jump Point Search (Harabor and Grastien 2011)



Ref: Wikipedia

Best first (generic)

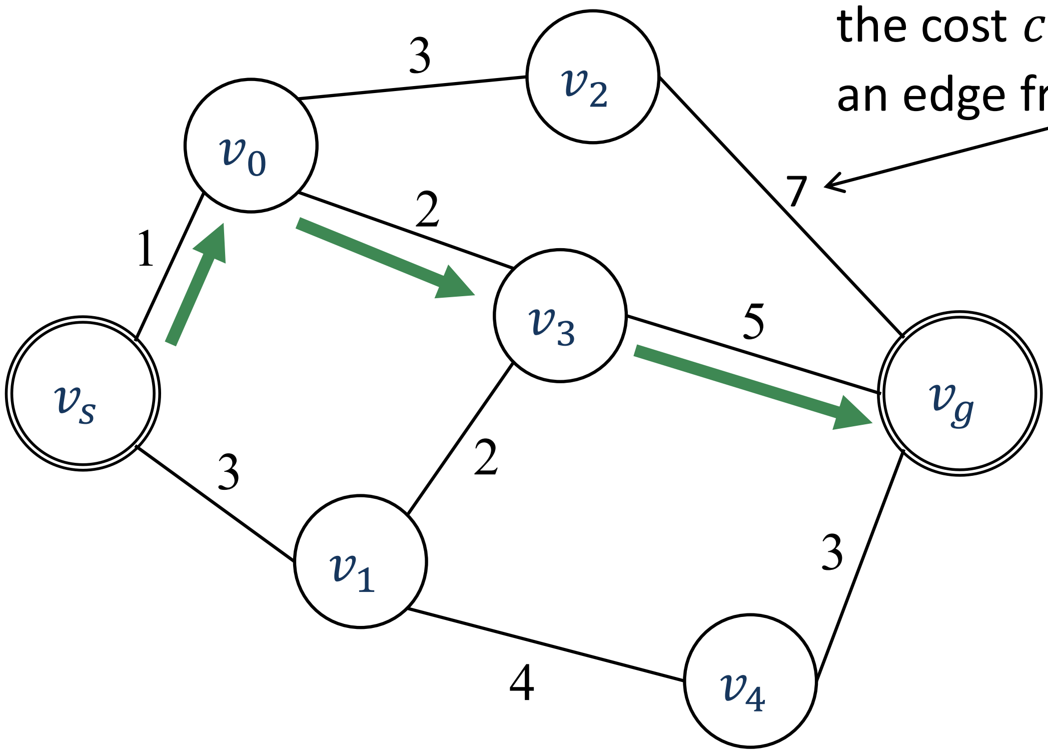
- Choose “most promising” node next according to some rule.
- Depending on the rule, may be
  - Optimal or not
  - Complete or not
  - Efficient or not

Today: Sampling of optimal and complete search algorithms with a “best first” flavor.

Dijkstra's Algorithm

A best-first inspired search using the "cost-to-come."

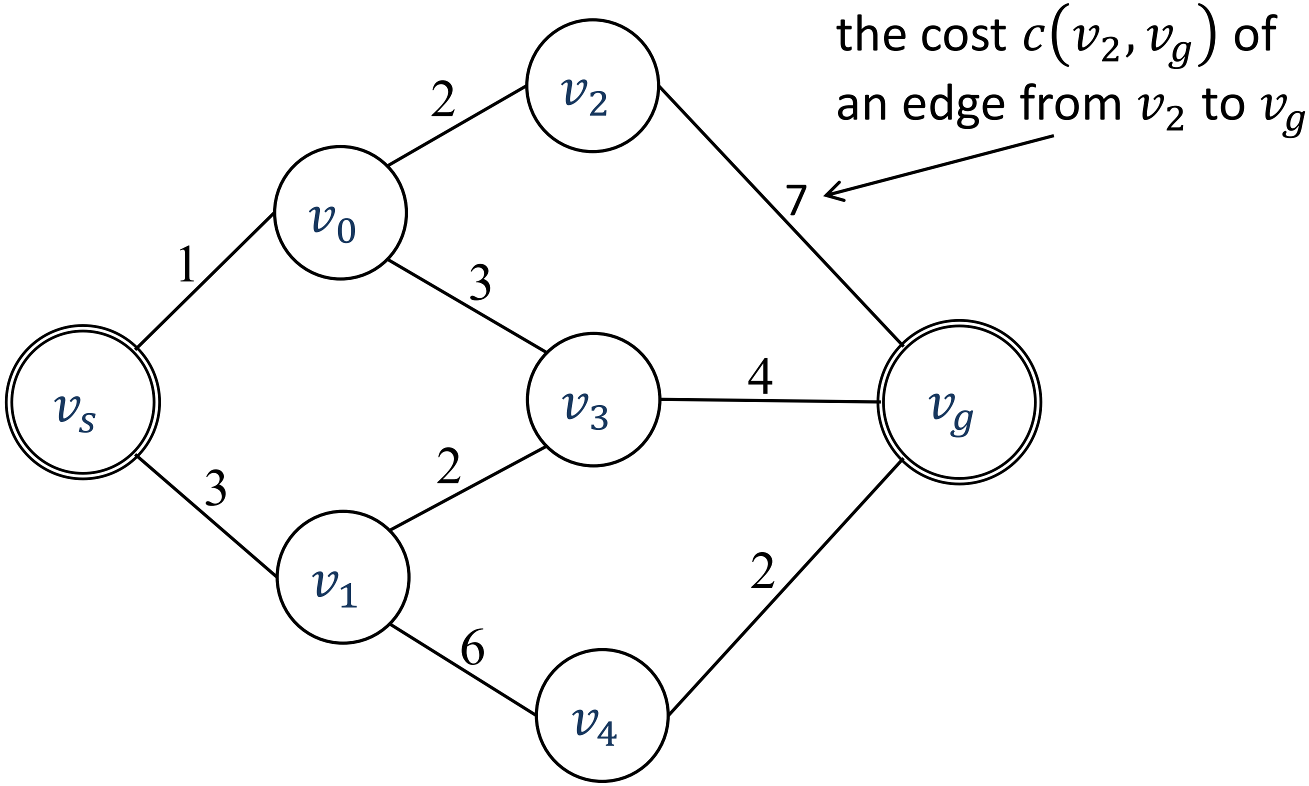
Problem Structure



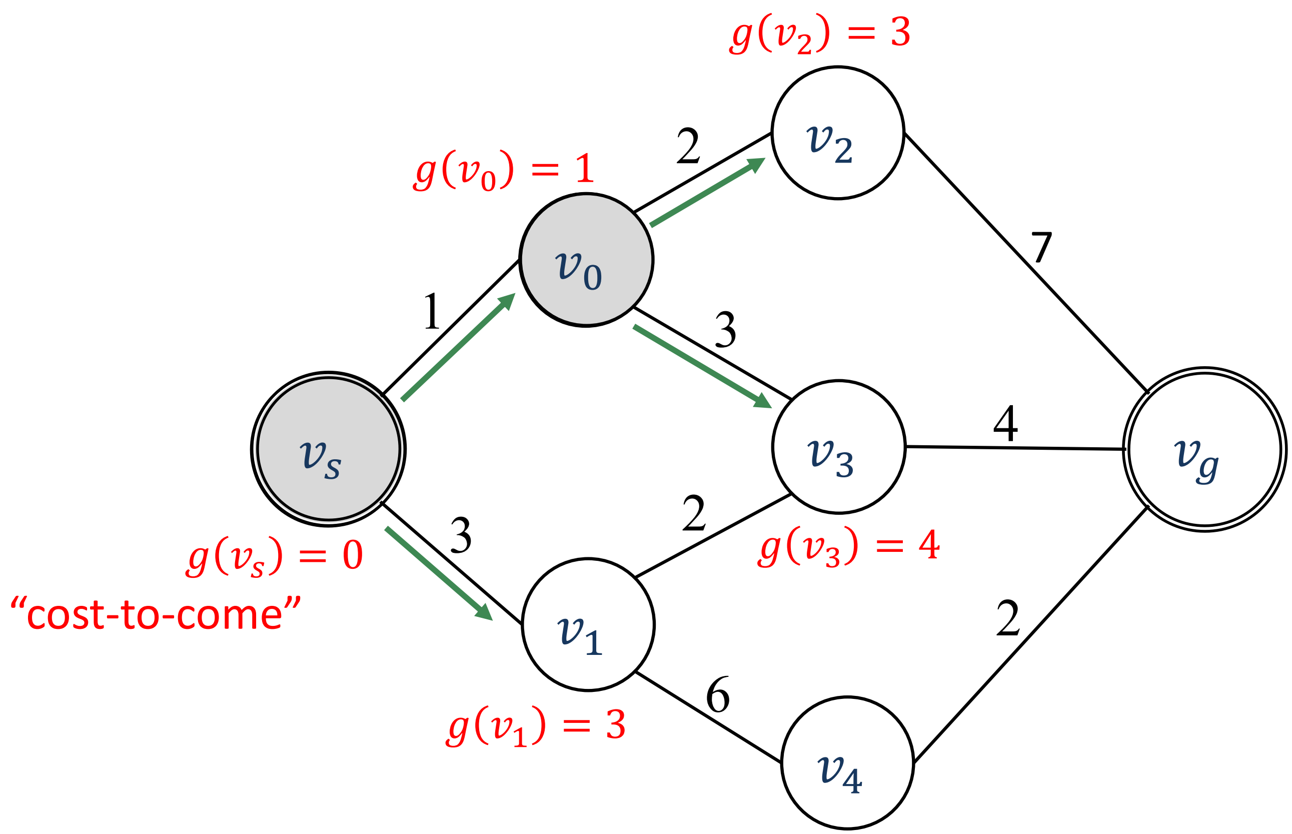
the cost $c(v_2, v_g)$ of an edge from v_2 to v_g

Notice: A subpath of a shortest path is a shortest path.

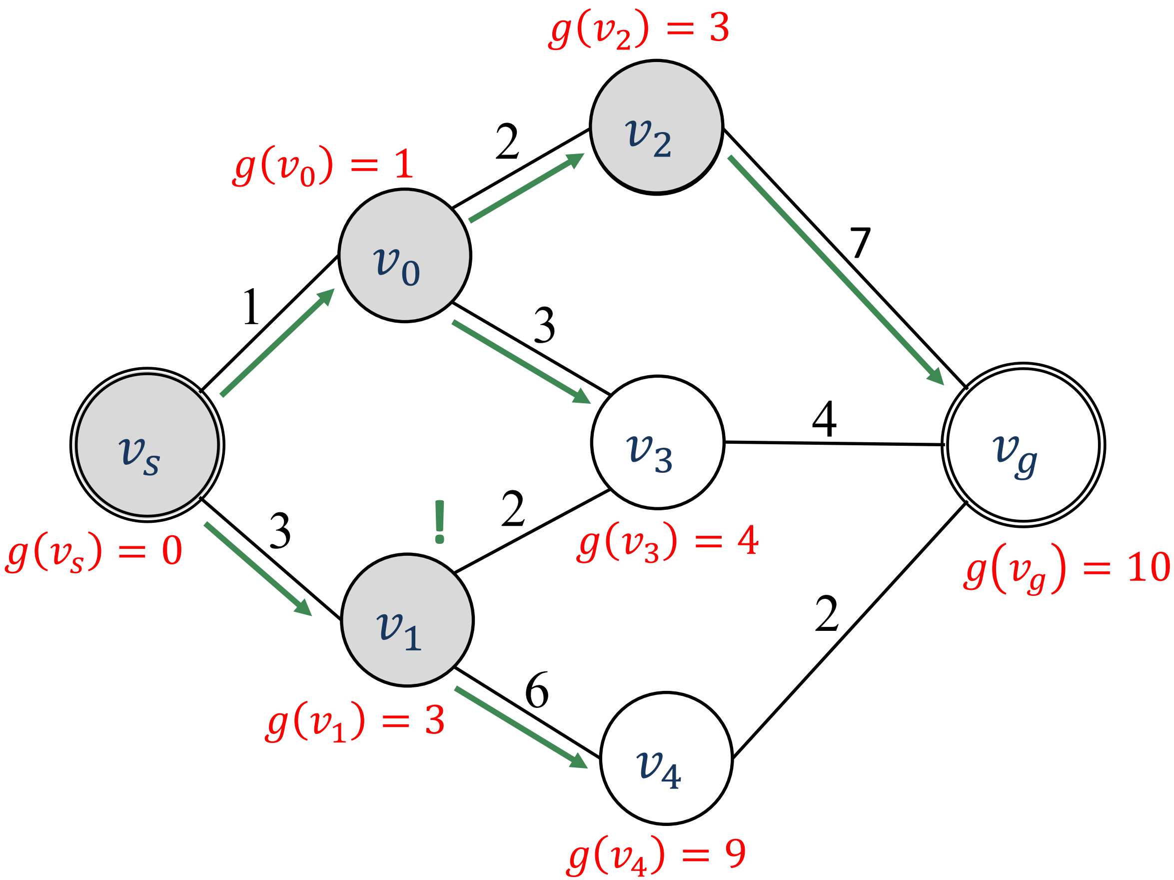
Dijkstra’s Algorithm



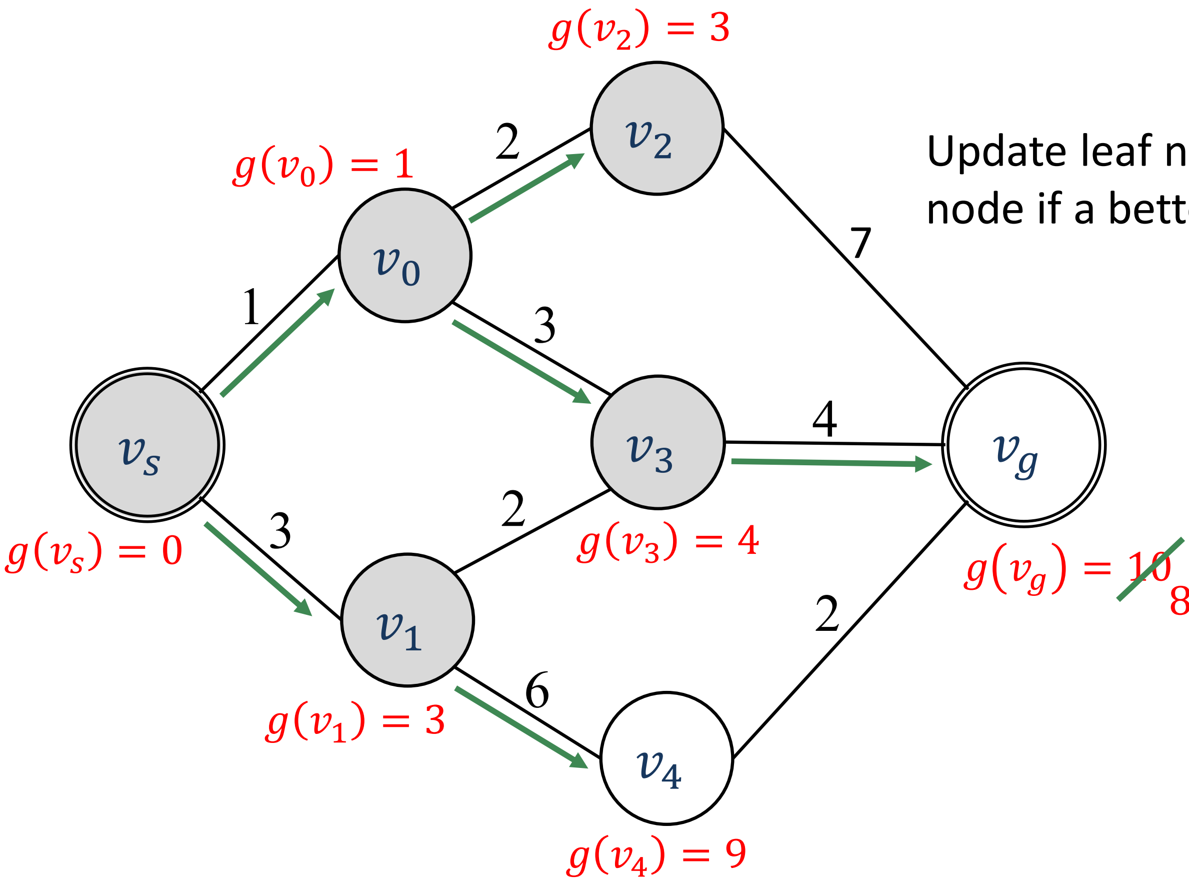
Dijkstra's Algorithm



Dijkstra's Algorithm

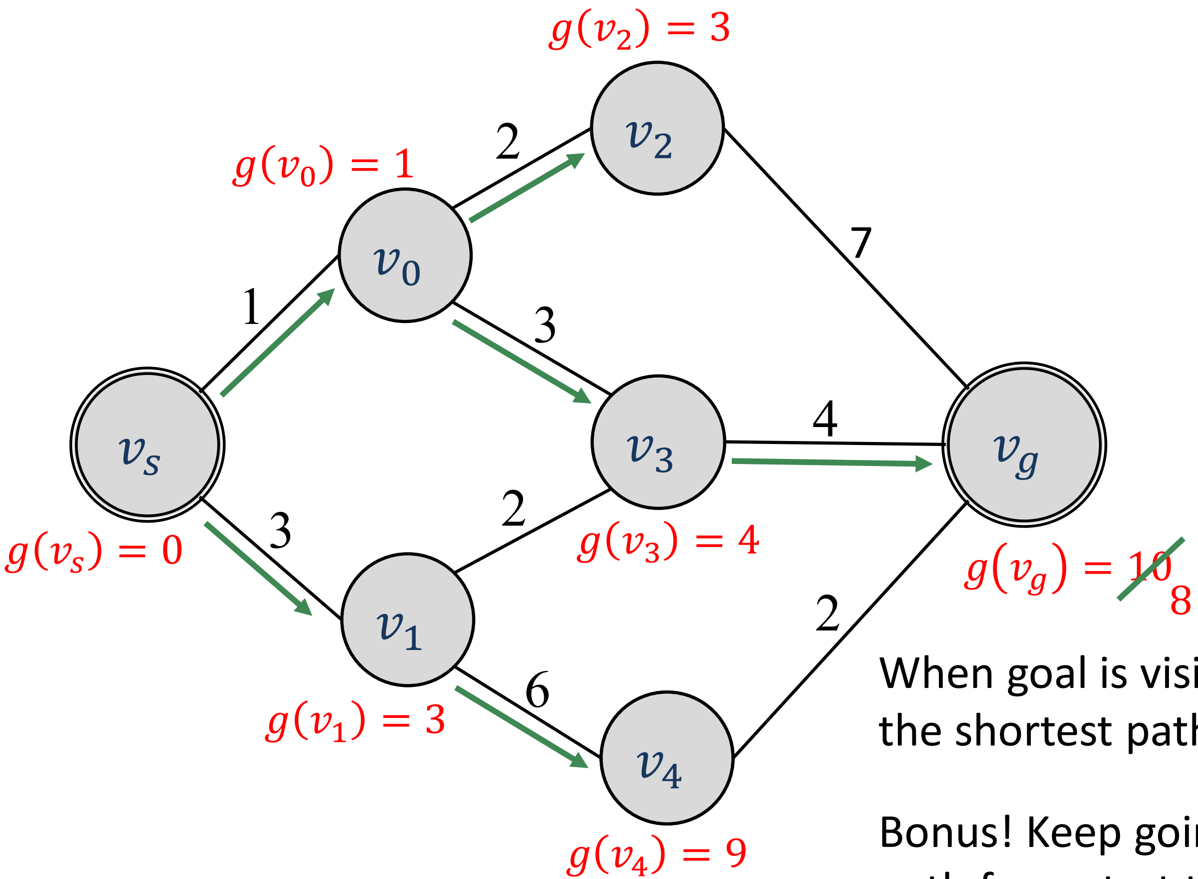


Dijkstra's Algorithm



Update leaf node's cost and parent node if a better path is found.

Dijkstra’s Algorithm



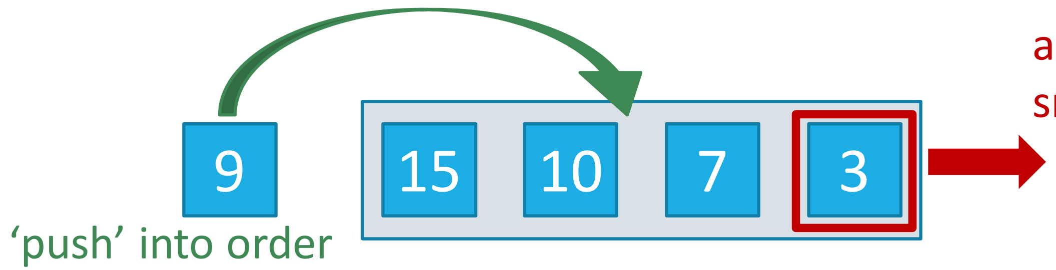
When goal is visited, we have found the shortest path from start to goal.

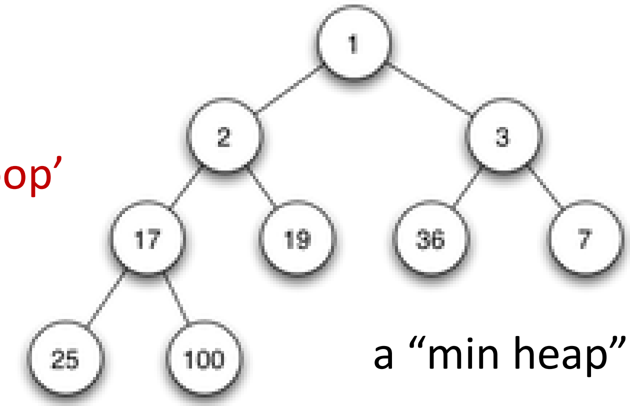
Bonus! Keep going to find the shortest path from start to *every* node.

Priority Queue

To implement Dijkstra's algorithm, we have to repeatedly find the open node with the smallest cost-to-come.

We want a "priority queue."





A priority queue can be implemented with a "heap."

A min heap is a binary tree for which every parent node has a value less than or equal to any of its children.

Python's priority queue is called 'heapq.'
https://docs.python.org/3.6/library/heapq.html

Dijkstra's Algorithm

function Dijkstra(G, v_s, v_g):

$Q = \emptyset$ — priority queue of open/alive nodes

for each $v \in G$:

 $g[v] \leftarrow \infty$ — "cost-to-come" from v_s to v, unknown

 $p[v] \leftarrow \emptyset$ — parent node, unknown

 $Q \leftarrow Q \cup v$ — all nodes initially open

$g[v_s] \leftarrow 0$ — distance from v_s to v_s is 0

no path found yet

a path might exists

while $v_g \in Q$ and $\min_{v \in Q} g[v] < \infty$:

 $u \leftarrow \operatorname{argmin}_{v \in Q} g[v]$ — v_s will be selected first

 $Q \leftarrow Q \setminus u$

 for each $v \in Q$ such that $v \in \text{neighbors}(u)$:

  $d \leftarrow g[u] + c(u, v)$

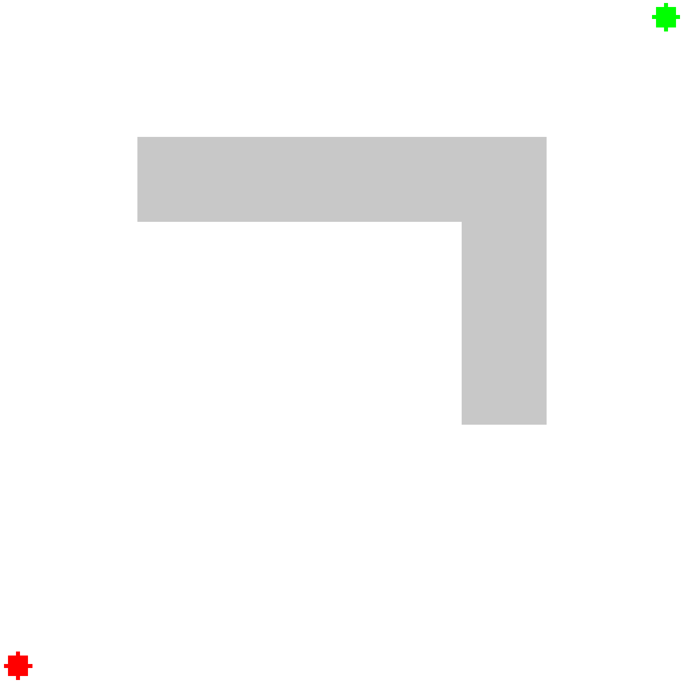
  if $d < g[v]$:

   $g[v] \leftarrow d$ — A shorter path to v has been found

   $p[v] \leftarrow u$

return g, p for all nodes — reconstruct path using parents, starting from v_g

Dijkstra’s Algorithm



Complexity

Naïve implementation has complexity

$$O(|E| + |V|^2)$$ (if finding min cost node with linear search)

For sparse, connected graphs (with $E \ll V^2$), it is possible to attain

$$O(|E| + |V| \log|V|)$$ (using an appropriate priority queue)

A^*

A best-first inspired search using the "cost-to-arrive" and an *estimate* of the "cost-to-go."

Prioritize Nodes

Need a method to prioritize nodes

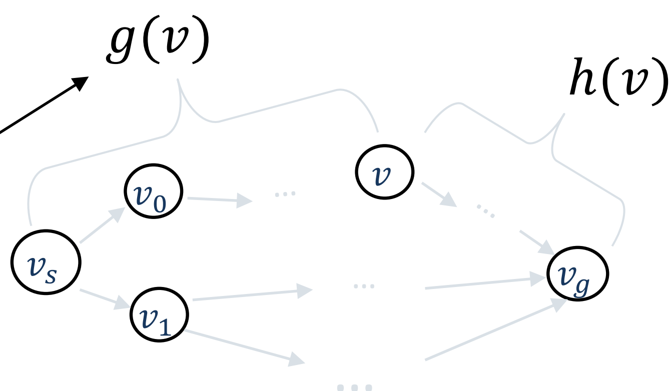
Estimate the running cost $g(v)$

- Optimal values satisfy $g(v) = \min_{p \in p[v]} g(p) + c(p, v)$

The cost for a node $f(v) = g(v) + h(v)$

- $g(v)$ is the running cost
- $h(v)$ is an *under-estimate* of the cost to reach the goal v_g from v
- h is a **heuristic**

the cost of a shortest path from s_{Init} to s **found so far**



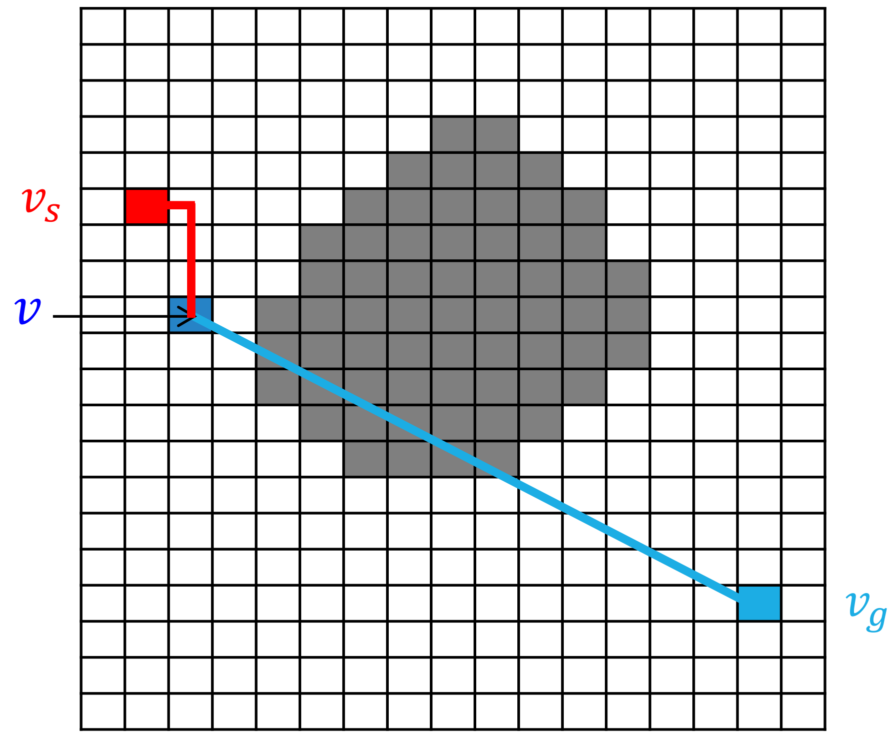
an (under) estimate of the cost of a shortest path from v to v_g

Heuristics

Node v has cost $f(v) = g(v) + h(v)$

Example

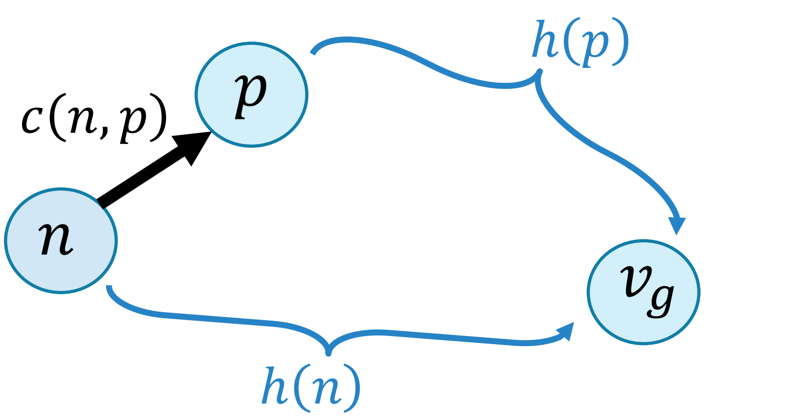
- 4-connected grid
- $g(v) = 4$
- $h(v) = |v - v_g|$
  $= \sqrt{8^2 + 13^2}$
  $= 15.3$
- $f(v) = 19.3$
- Actual minimum cost $c^*(v_s, v_g) = 25$



Heuristics

Heuristic functions must be:

1. Admissible – for every v, $h(v) \le c^*(v, v_g)$ ("optimism")
2. Consistent – satisfy the triangle inequality
   1. $h(v_g) = 0$
   2. for every $n \ne v_g$ and $p = \mathrm{succ}(n)$, $h(n) \le c(n,p) + h(p)$
3. Consistency implies admissibility.
   Not necessarily the other way around.



Choosing a Heuristic

We’re particularly interested in graphs that represent distances in metric space.

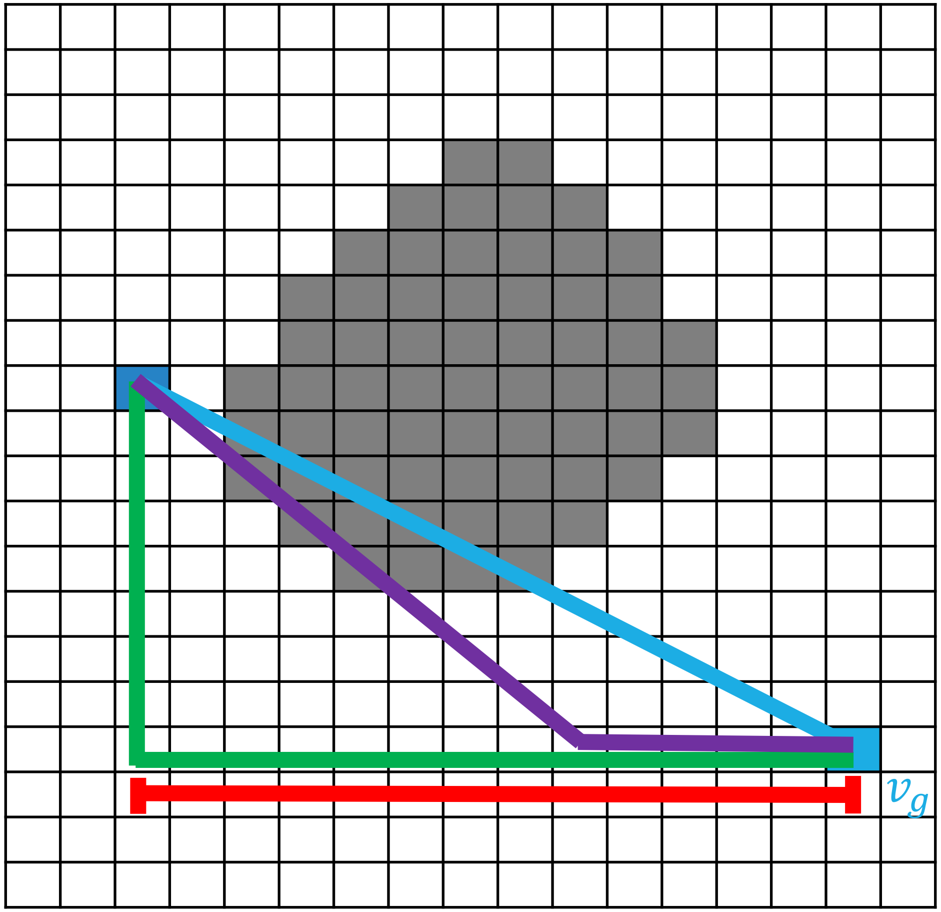
Different kinds of distances or norms are interesting candidates for heuristics.

$\|x\|_2$ (Euclidean distance)

$\|x\|_1$ (Manhattan distance)

$\|x\|_\infty$ (Maximum metric, or Chebyshev distance)

“octile-” or “move-distance”



A^* Algorithm

function $A^*(G, v_s, v_g)$:

$Q = \emptyset$ — priority queue of open/alive nodes

for each $v \in G$:

- $g[v] \leftarrow \infty$ — "cost-to-come" from v to v_s, unknown
- $p[v] \leftarrow \emptyset$ — parent node, unknown
- $Q \leftarrow Q \cup v$ — all nodes initially open

$g[v_s] \leftarrow 0$ — distance from v_s to v_s is 0

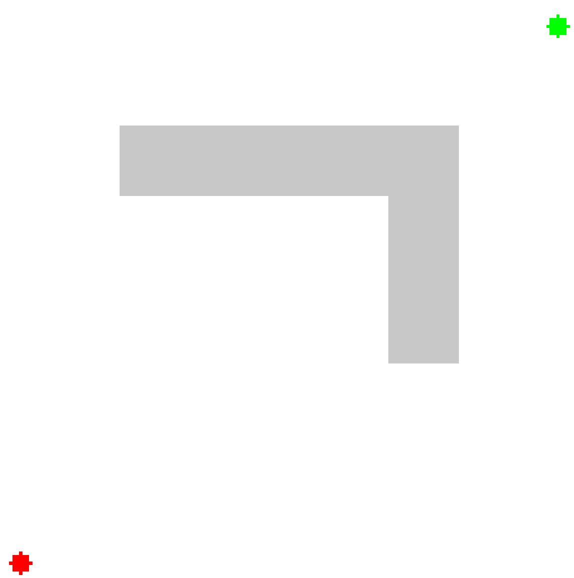
while $v_g \in Q$ and $\min_{v \in Q} f[v] < \infty$: — Replace $g[v]$ with $f[v] = g[v] + h[v]$

- $u \leftarrow \operatorname{argmin}_{v \in Q} f[v]$ — v_s will be selected first
- $Q \leftarrow Q \setminus u$
- **for each** $v \in Q$ such that $v \in \text{neighbors}(u)$:
  - $d \leftarrow g[u] + c(u, v)$
  - **if** $d < g[v]$: — A shorter path to v has been found
    - $g[v] \leftarrow d$
    - $p[v] \leftarrow u$

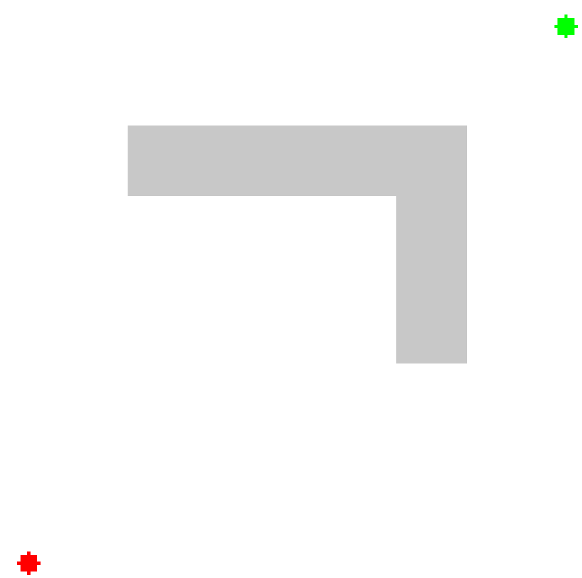
return $d[], p[]$

Dijkstra vs. A^*

Dijkstra



A^*



https://qiao.github.io/PathFinding.js/visual/

Dijkstra / A* Resources

Fun Interactive Visualizations

- https://qiao.github.io/PathFinding.js/visual/

LaValle, Planning Algorithms, Chapter 2

Does A-Star always outperform Dijkstra?

Not if the heuristic is uninformative.

Jump Point Search

Speed up A* by expanding fewer neighbors.

Harabor and Grastien, “Online Graph Pruning for Pathfinding on Grid Maps,” 2011.

Motivation

There are often many "symmetric" paths of equal cost

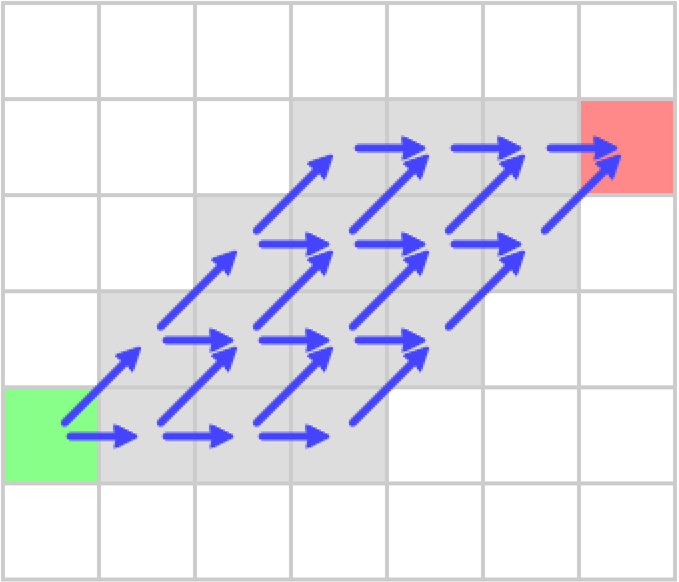
A^* expands the immediate neighbors of the current node

- For straight line paths, this will expand many unnecessary nodes

Speed up A^* by selectively expanding nodes

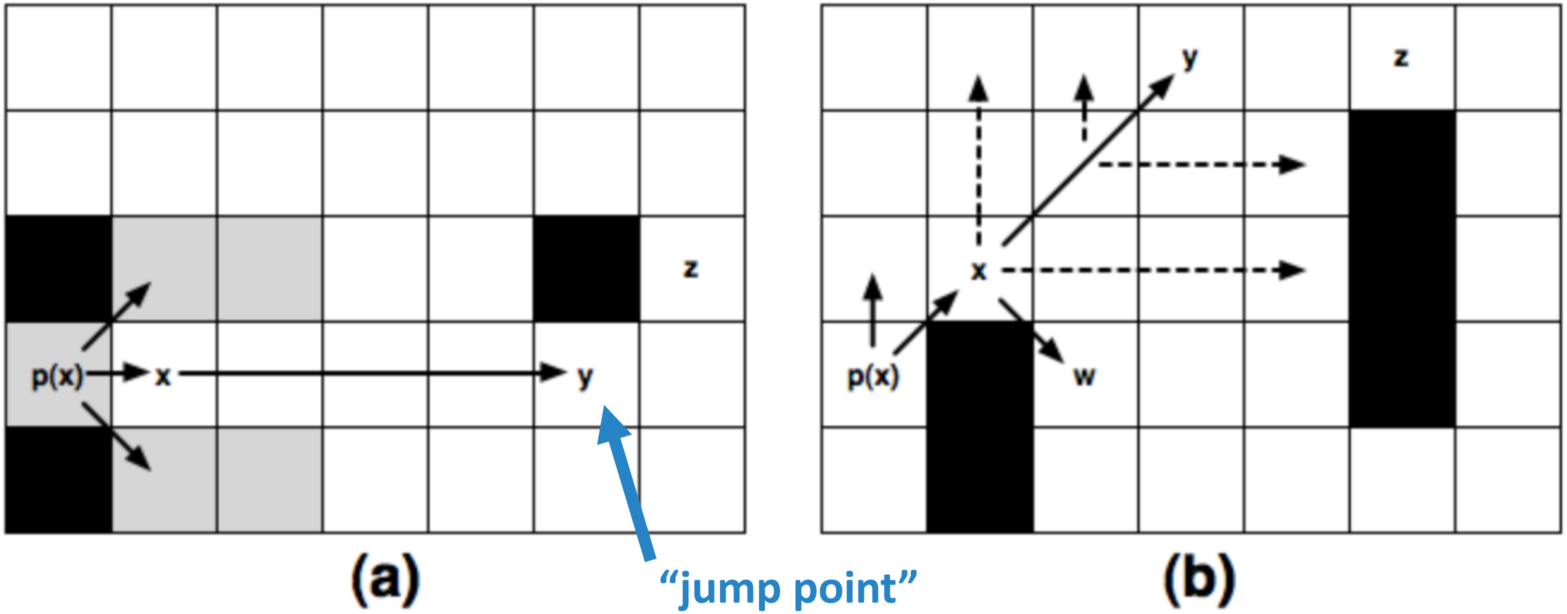
Assumptions

- Uniform, 8-connected grid (in 2D)
- Cost for straight moves is 1 and for diagonal moves is $\sqrt{2}$



Basic Idea

Ignore all grey nodes, because they could have been reached optimally from $p[x]$ without going through x.



(a) (b)

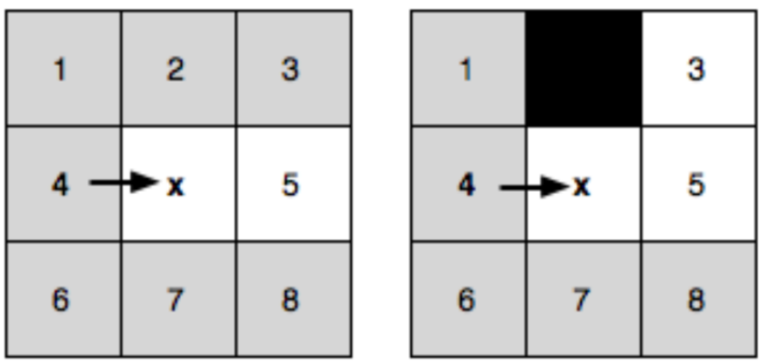
Neighbor Pruning Rules

Let x be the current node and $p[x]$ its predecessor

For each other node n, compare the path from p[x] to n

- If it goes through x?
- If it does not go through x?

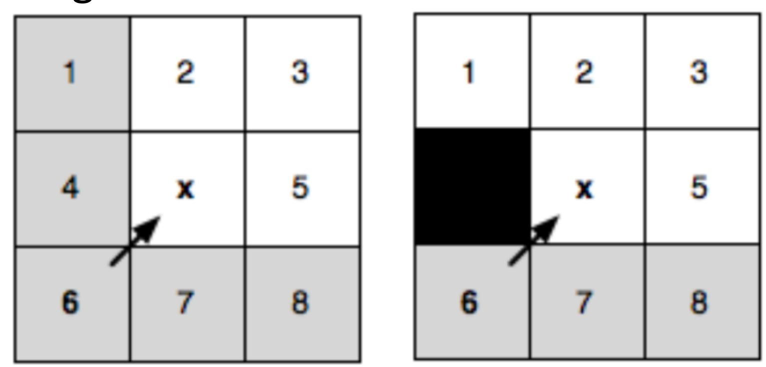
Straight Moves



$$len(\ \langle p(x), \ldots, n \rangle \setminus x\ ) \leq len(\ \langle p(x), x, n \rangle\ )$$

Prune neighbors reached in less or equal time.

Diagonal Moves



$$len(\ \langle p(x), \ldots, n \rangle \setminus x\ ) < len(\ \langle p(x), x, n \rangle\ )$$

Prune neighbors reached in strictly less time.

Properties of JPS

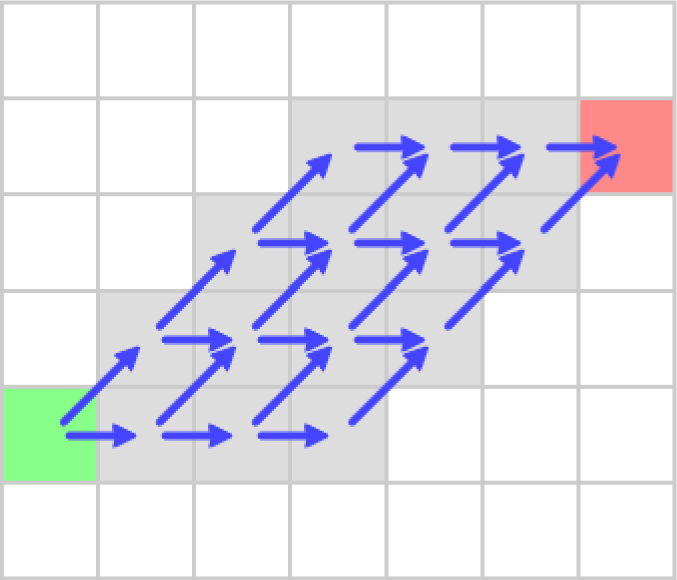
Solutions are optimal

Does not require additional pre-processing

No extra memory overhead

Speeds up A^* search by 10x or more

only works on uniform grids



While you're not going to use JPS in the project, you will see the impact of multiple "equivalent" paths in your results.

Resources on JPS

Original JPS Paper

- Harabor and Grastien, "Online Graph Pruning for Pathfinding on Grid Maps," 2011.

JPS Without Corner Cutting

- Harabor and Grastien, "The JPS Pathfinding System," 2012.

JPS Tutorials

- http://zerowidth.com/2013/05/05/jump-point-search-explained.html
- https://harablog.wordpress.com/2011/09/07/jump-point-search/