Search-based Path Finding

Hongkai YE

2021.10.05

Outline

1. Graph Search Basis
2. Dijkstra and A*
3. Jump Point Search

Graph Search Basis

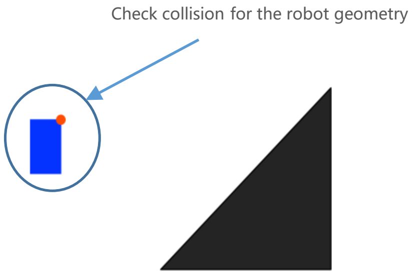
Configuration Space

Configuration Space

- **Robot configuration:** a specification of the positions of all points of the robot
- **Robot degree of freedom (DOF):** The minimum number n of real-valued coordinates needed to represent the robot configuration
- **Robot configuration space:** a n-dim space containing all possible robot configurations, denoted as C-space
- **Each robot pose is a point in the C-space**

Configuration Space Obstacle

- Planning in workspace
  - Robot has different shape and size
  - Collision detection requires knowing the robot geometry - time consuming and hard

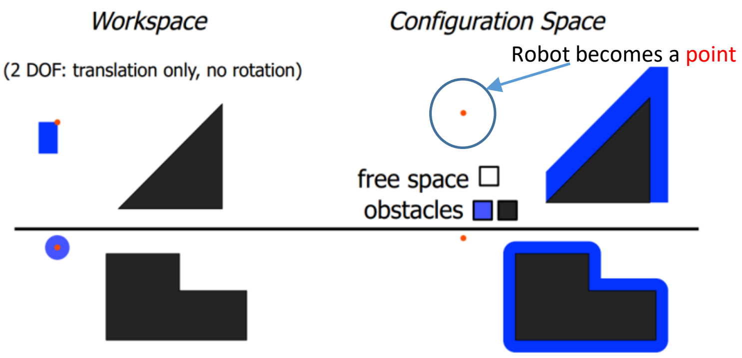


(1) Rectangular mobile robot

(2) Circular mobile robot

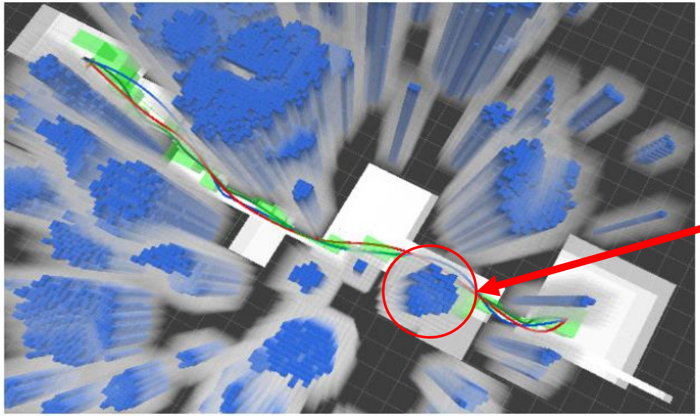
Configuration Space Obstacle

- Planning in configuration space
  - Robot is represented by a point in C-space, e.g. position (a point in R^3), pose (a point in $SO(3)$),etc.
  - Obstacles need to be represented in configuration space (one-time work prior to motion planning), called configuration space obstacle, or C-obstacle
  - C-space = (C-obstacle) ∪ (C-free)
  - The path planning is finding a path between start point q_{start} and goal point q_{goal} within C-free



Workspace and Configuration Space Obstacle

- In workspace
  - Robot has shape and size (i.e. hard for motion planning)
- In configuration space: C-space
  - Robot is a point (i.e. easy for motion planning)
  - Obstacle are represented in C-space prior to motion planning
- Representing an obstacle in C-space can be extremely complicated. So approximated (but more conservative) representations are used in practice.



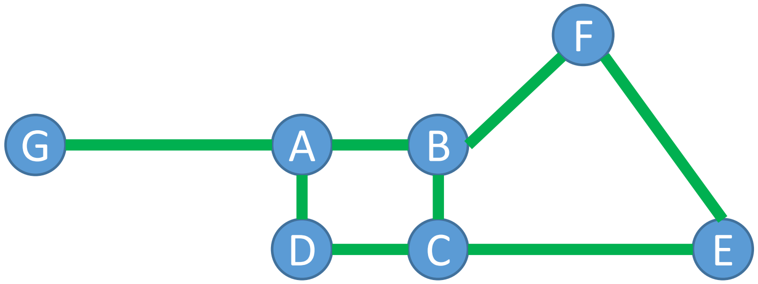
If we model the robot conservatively as a ball with radius δ_r, then the C-space can be constructed by inflating obstacle at all directions by δ_r.

Graph and Search Method

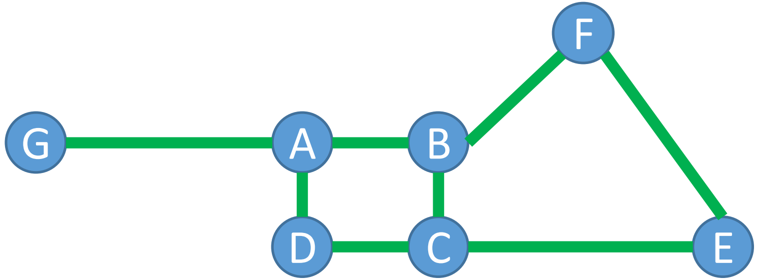
Search-based Method

Graphs

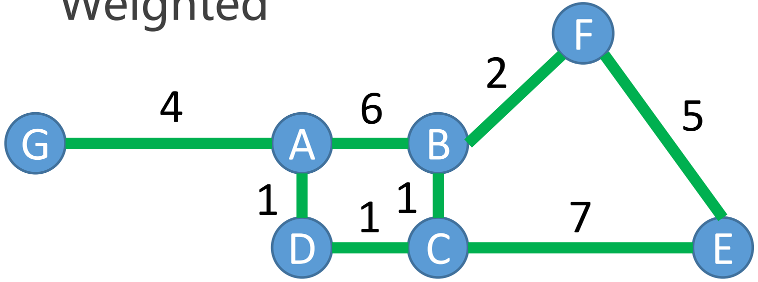
Graphs have nodes and edges



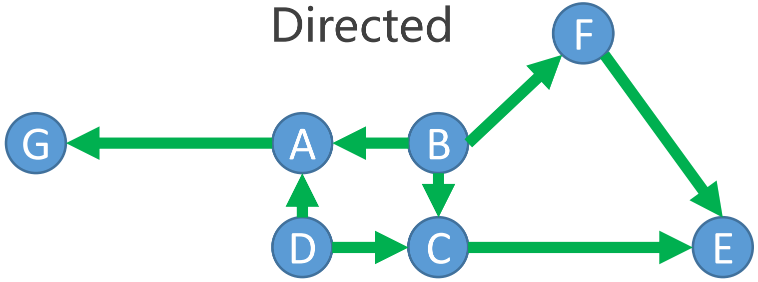
Undirected



Weighted

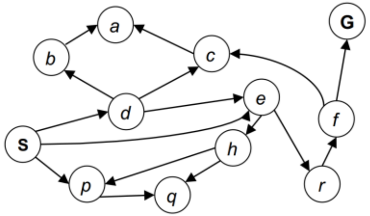


Directed

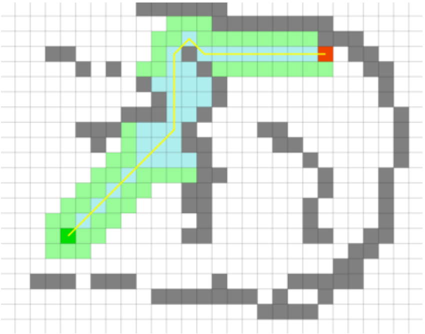


Search-based Method

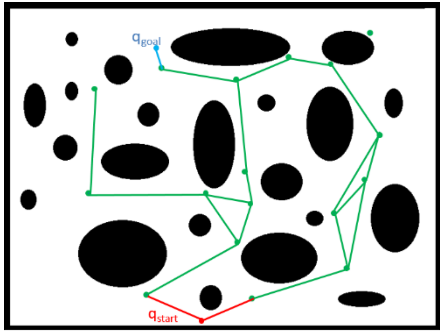
- State space graph: a mathematical representation of a search algorithm
  - For every search problem, there's a corresponding state space graph
  - Connectivity between nodes in the graph is represented by (directed or undirected) edges



Ridiculously tiny search graph for a tiny search problem



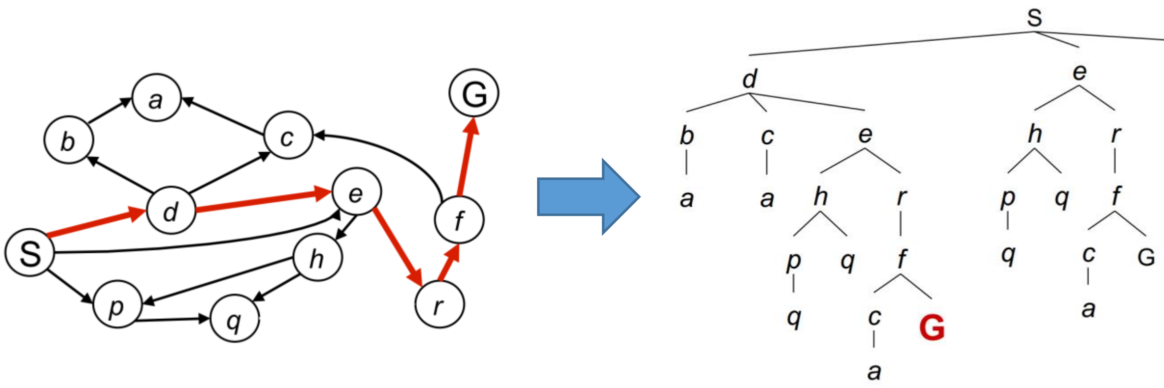
Grid-based graph: use grid as vertices and grid connections as edges



The graph generated by probabilistic roadmap (PRM)

Graph Search Overview

- The search always start from start state X_S
  - Searching the graph produces a search tree
  - Back-tracing a node in the search tree gives us a path from the start state to that node
  - For many problems we can never actually build the whole tree, too large or inefficient – we only want to reach the goal node asap.



Graph Search Overview

- Maintain a container to store all the nodes to be visited
- The container is initialized with the start state X_S
- Loop
  - Remove a node from the container according to some pre-defined score function
    - Visit a node
  - Expansion: Obtain all neighbors of the node
    - Discover all its neighbors
  - Push them (neighbors) into the container
- End Loop

Graph Search Overview

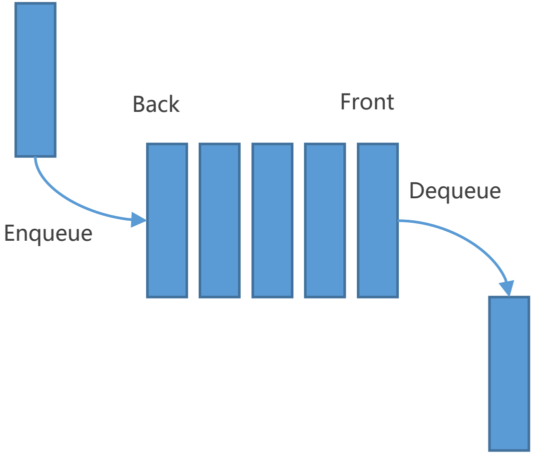
- Question 1: When to end the loop?
  - Possible option: End the loop when the container is empty
- Question 2: What if the graph is cyclic?
  - When a node is removed from the container (expanded / visited), it should never be added back to the container again
- Question 3: In what way to remove the right node such that the goal state can be reached as soon as possible, which results in less expansion of the graph node.

Graph Traversal

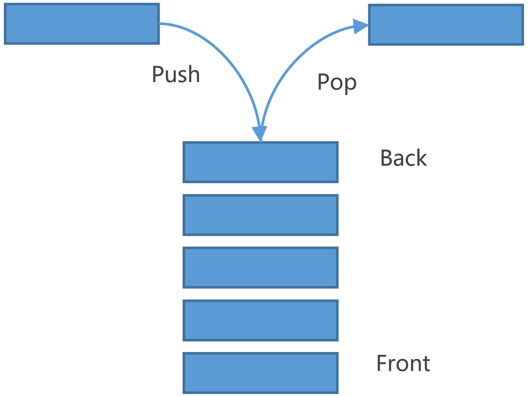
- Breadth First Search (BFS) vs. Depth First Search (DFS)

BFS uses "first in first out"

DFS uses "last in first out"



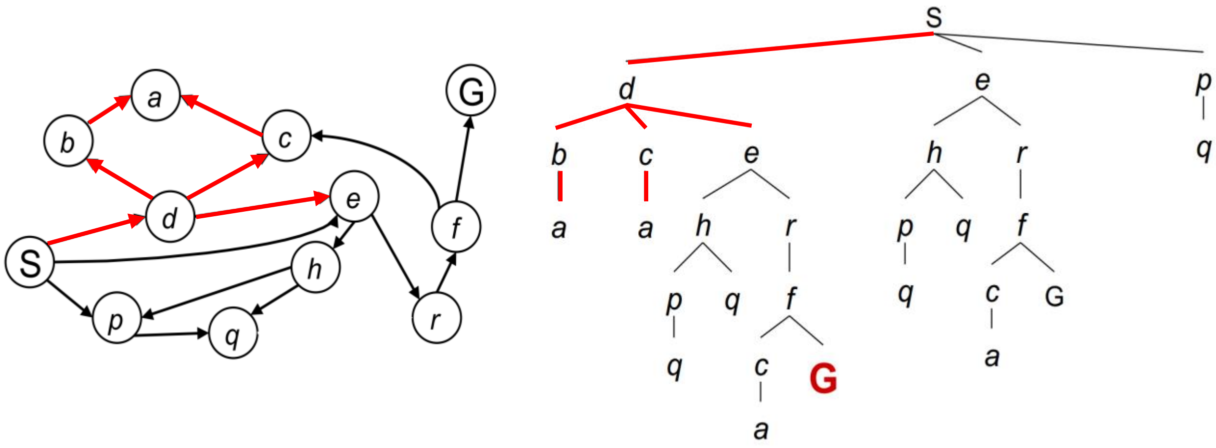
This is a queue



This is a stack

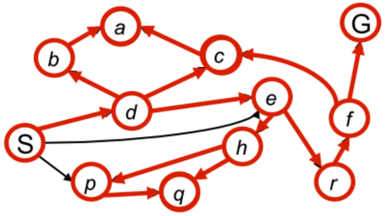
Depth First Search (DFS)

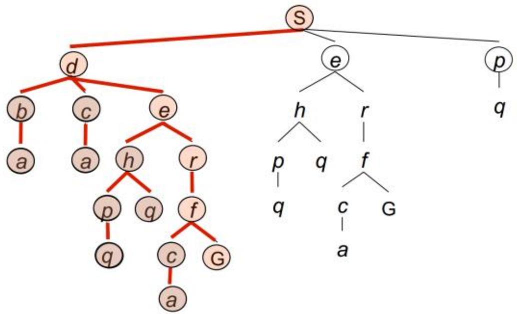
- Strategy: remove / expand the deepest node in the container

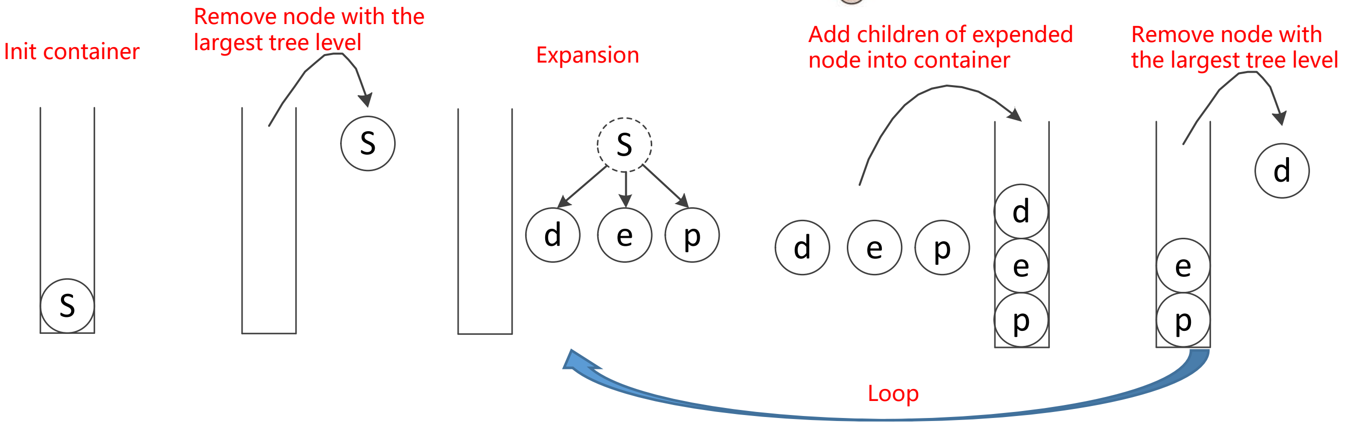


Depth First Search (DFS)

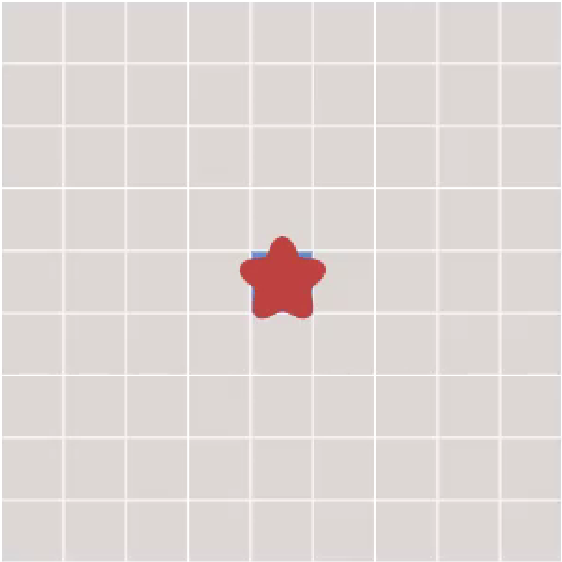
- Implementation: maintain a last in first out (LIFO) container (i.e. stack)







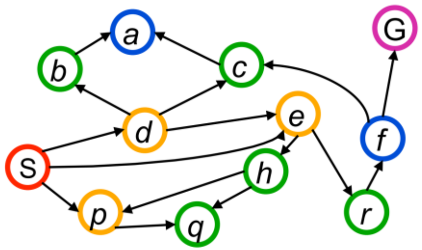
Depth First Search (DFS)

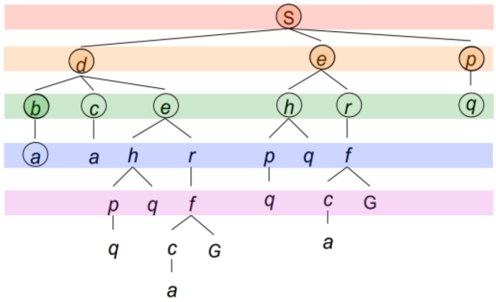


Courtesy: Amit Patel's Introduction to A*, Stanford

Breadth First Search (BFS)

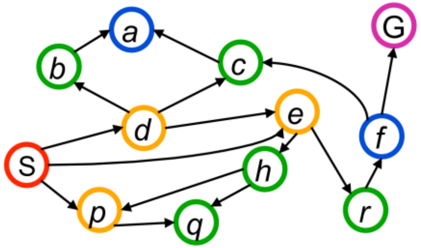
- Strategy: remove / expand the shallowest node in the container

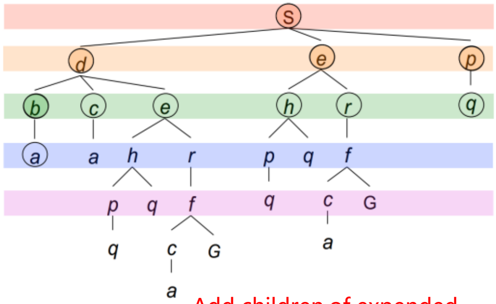


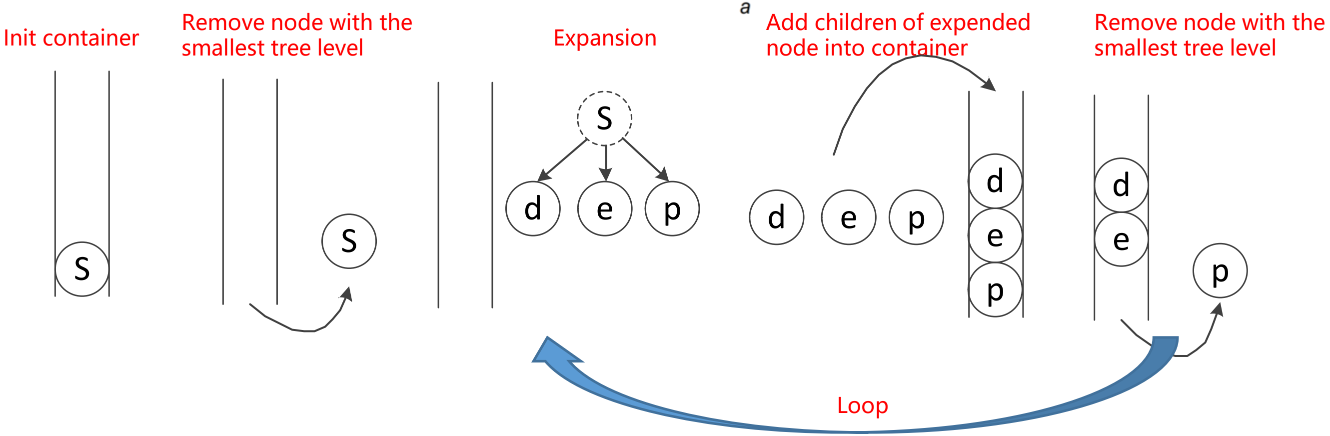


Breadth First Search (BFS)

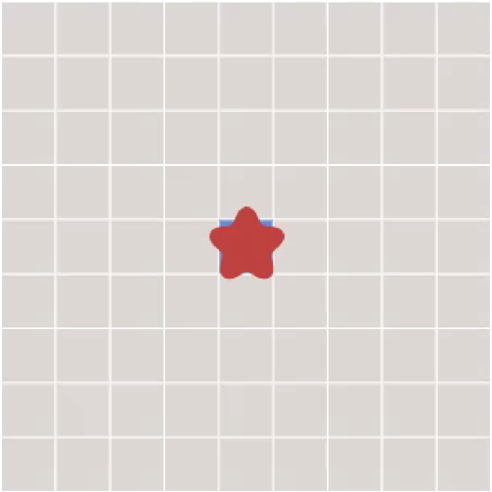
- Implementation: maintain a first in first out (FIFO) container (i.e. queue)





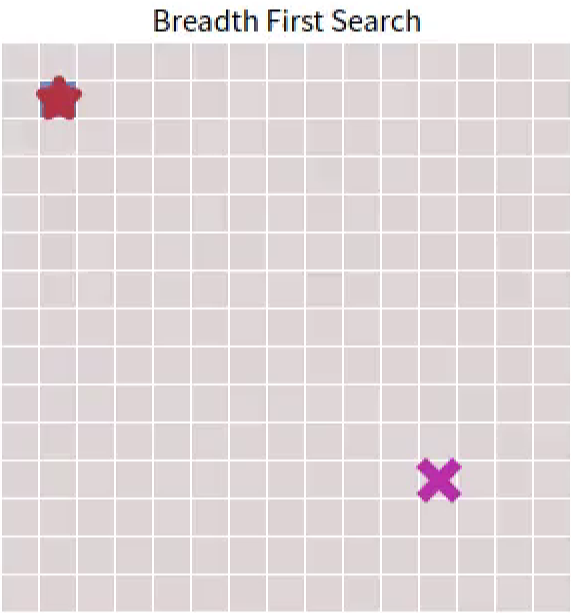


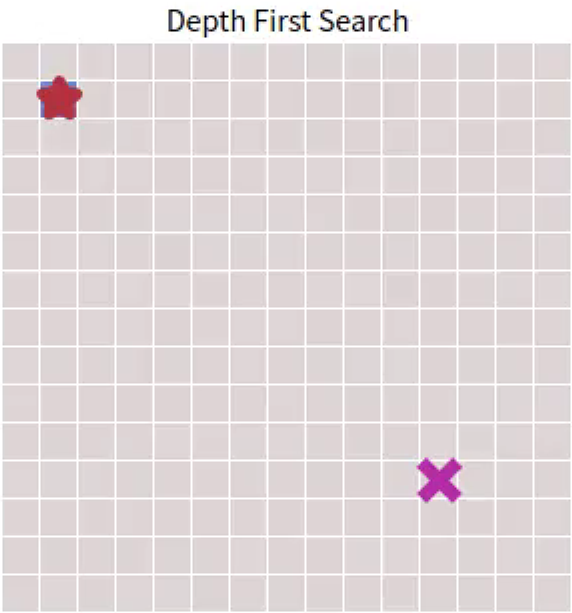
Breadth First Search (BFS)



Courtesy: Amit Patel's Introduction to A*, Stanford

BFS vs. DFS: which one is useful?





Remember BFS.

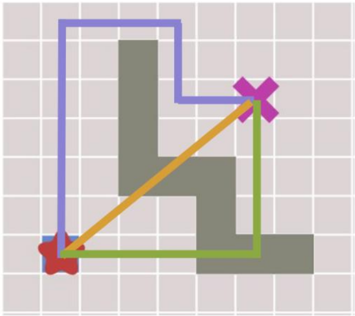
Heuristic search

Greedy Best First Search

- BFS and DFS pick the next node off the frontiers based on which was “first in” or “last in”.
- Greedy Best First picks the “best” node according to some rule, called a heuristic.

- Definition: A heuristic is a guess of how close you are to the target.

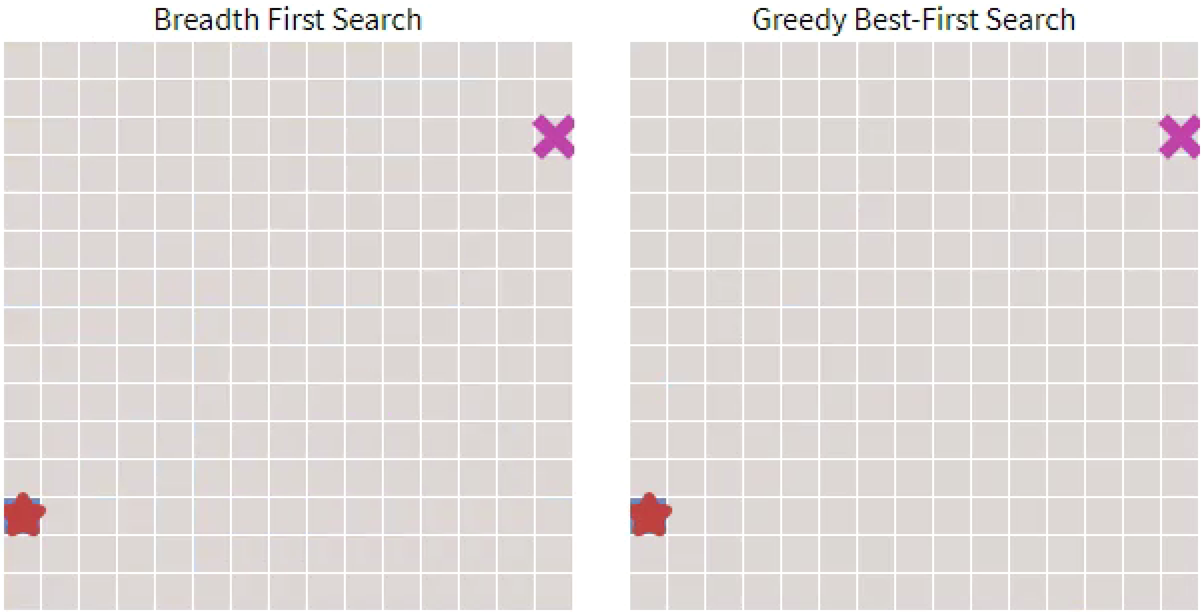
- A heuristic guides you in the right direction.
- A heuristic should be easy to compute.



- Euclidean Distance
- Manhattan Distance

Both are approximations for the actual shortest path.

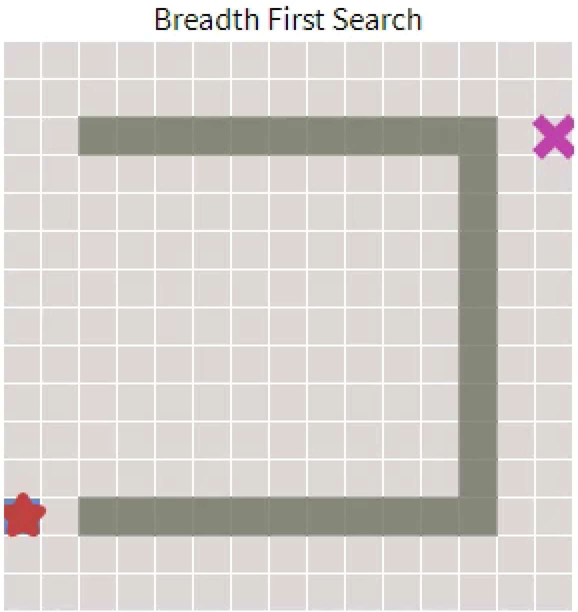
Greedy Best First Search

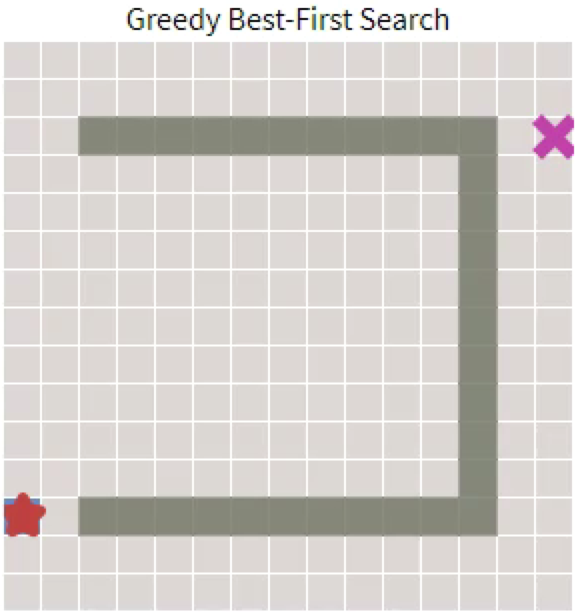


Looks pretty good.

Courtesy: Amit Patel's Introduction to A*, Stanford

Greedy Best First Search



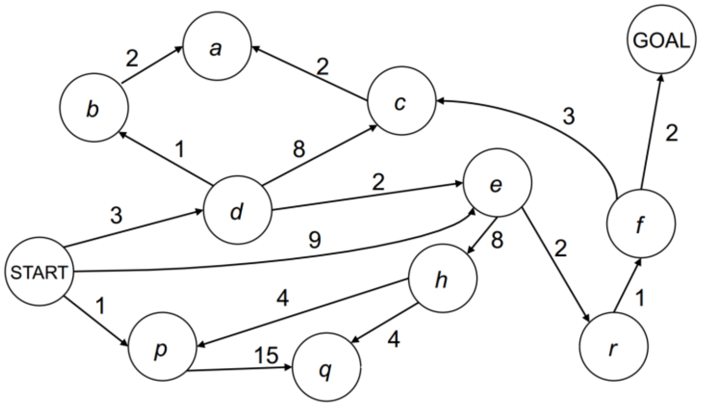


But with obstacles …

Courtesy: Amit Patel's Introduction to A*, Stanford

Costs on Actions

- A practical search problem has a cost "C" from a node to its neighbor
  - Length, time, energy, etc.
- When all weight are 1, BFS finds the optimal solution
- For general cases, how to find the least-cost path as soon as possible?

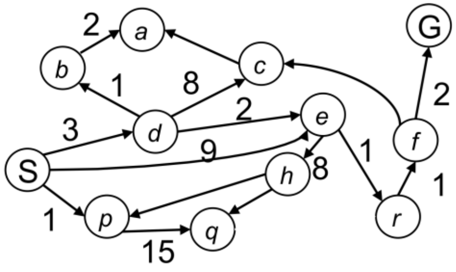


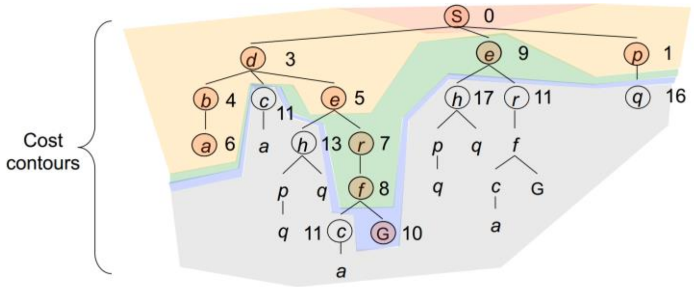
Dijkstra and A*

Algorithm Workflow

Dijkstra’s Algorithm

- Strategy: expand/visit the node with cheapest accumulated cost g(n)
  - g(n): The current best estimates of the accumulated cost from the start state to node “n”
  - Update the accumulated costs g(m) for all unexpanded neighbors “m” of node “n”
  - A node that has been expanded/visited is guaranteed to have the smallest cost from the start state

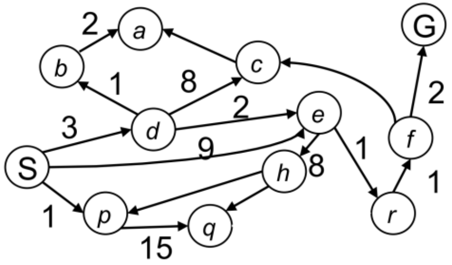


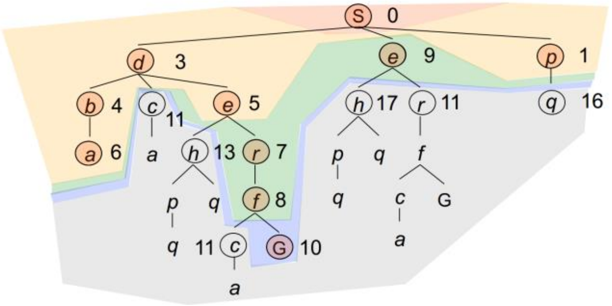


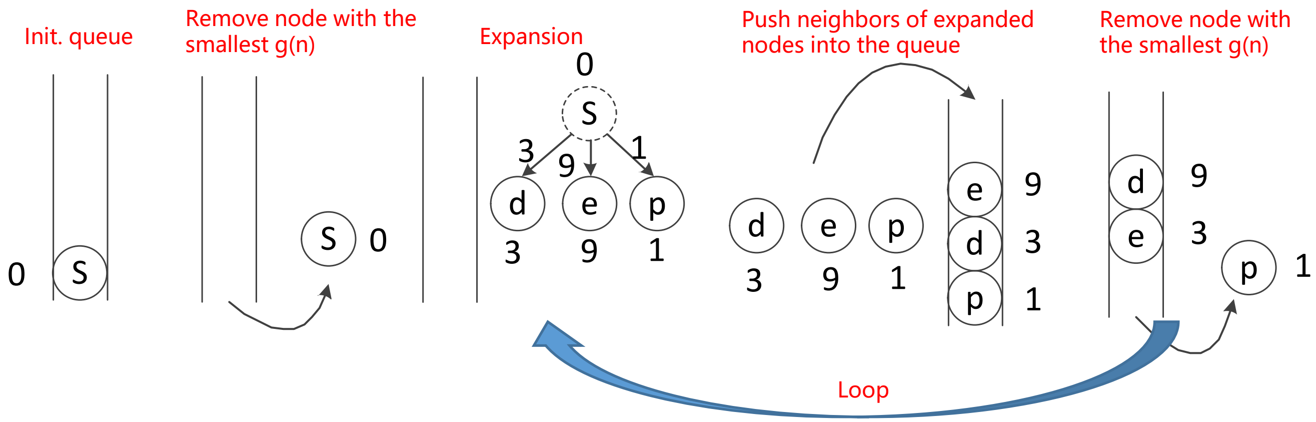
Dijkstra’s Algorithm

- Maintain a priority queue to store all the nodes to be expanded
- The priority queue is initialized with the start state X_S
- Assign $g(X_S)=0$, and $g(n)$=infinite for all other nodes in the graph
- Loop
  - If the queue is empty, return FALSE; break;
  - Remove the node “n” with the lowest g(n) from the priority queue
  - Mark node “n” as expanded
  - If the node “n” is the goal state, return TRUE; break;
  - For all unexpanded neighbors “m” of node “n”
    - If g(m) = infinite
      - g(m)= g(n) + Cnm
      - Push node “m” into the queue
    - If $g(m) > g(n) + C_{nm}$
      - $g(m)= g(n) + C_{nm}$
  - end
- End Loop

Dijkstra’s Algorithm

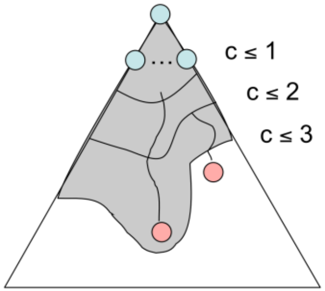


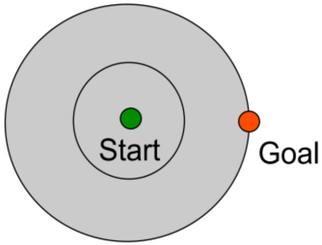




Pros and Cons of Dijkstra’s Algorithm

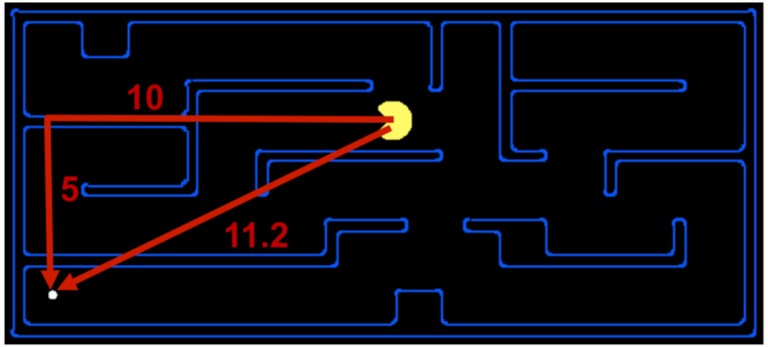
- The good:
  - Complete and optimal
- The bad:
  - Can only see the cost accumulated so far (i.e. the uniform cost), thus exploring next state in every “direction”
  - No information about goal location





Search Heuristics

- Recall the heuristic introduced in **Greedy Best First Search**
- Overcome the shortcomings of uniform cost search by inferring the least cost to goal (i.e. goal cost)
- Designed for particular search problem
- Examples: Manhattan distance VS. Euclidean distance



A*: Dijkstra with a Heuristic

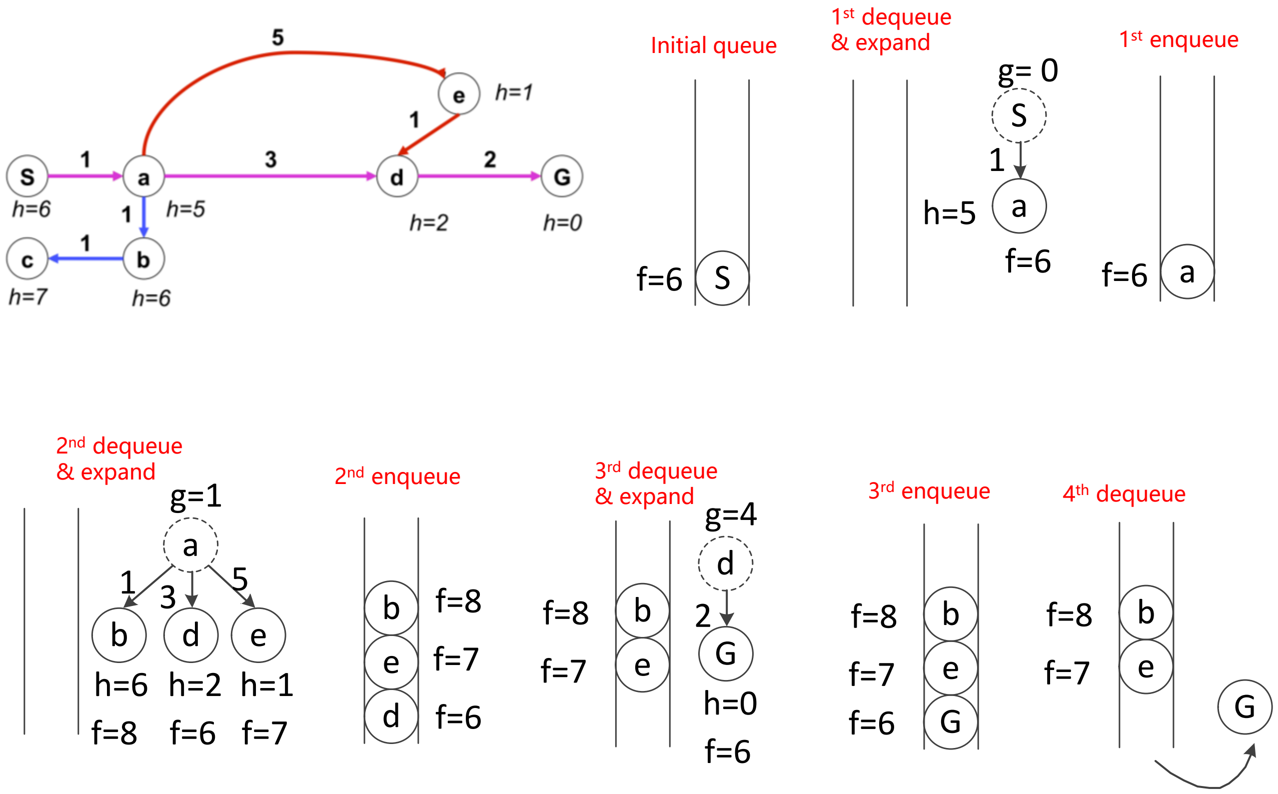
- Accumulated cost
  - $g(n)$: The current best estimates of the accumulated cost from the start state to node "n"
- Heuristic
  - $h(n)$: The **estimated least cost** from node n to goal state (i.e. goal cost)
- The least estimated cost from start state to goal state passing through node "n" is $f(n) = g(n) + h(n)$
- Strategy: expand the node with cheapest $f(n) = g(n) + h(n)$
  - Update the accumulated costs $g(m)$ for all unexpanded neighbors "m" of node "n"
  - A node that has been expanded is guaranteed to have the smallest cost from the start state

A* Algorithm

- Maintain a priority queue to store all the nodes to be expanded
- The heuristic function h(n) for all nodes are pre-defined
- The priority queue is initialized with the start state X_S
- Assign $g(X_S)=0$, and $g(n)$=infinite for all other nodes in the graph
- Loop
  - If the queue is empty, return FALSE; break;
  - Remove the node "n" with the lowest $f(n)=g(n)+h(n)$ from the priority queue
  - Mark node "n" as expanded
  - If the node "n" is the goal state, return TRUE; break;
  - For all unexpanded neighbors "m" of node "n"
    - If g(m) = infinite
      - g(m)= g(n) + Cnm
      - Push node "m" into the queue
    - If $g(m) > g(n) + C_{nm}$
      - g(m)= g(n) + Cnm
  - end
- End Loop

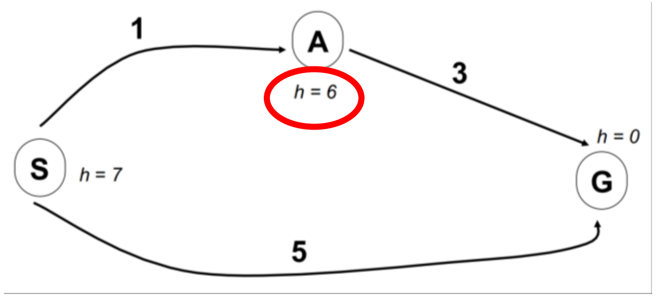
Only difference comparing to Dijkstra's algorithm

A* Example



5
e
h=1
1
S
1
a
3
d
2
G
h=6
1
h=5
h=2
h=0
c
1
b
h=7
h=6
Initial queue
f=6
S
1st dequeue & expand
g= 0
S
1
h=5
a
f=6
1st enqueue
f=6
a
2nd dequeue & expand
g=1
a
1
3
5
b
d
e
h=6
h=2
h=1
f=8
f=6
f=7
2nd enqueue
b
f=8
e
f=7
d
f=6
3rd dequeue & expand
g=4
d
f=8
b
2
f=7
e
G
h=0
f=6
3rd enqueue
f=8
b
f=7
e
f=6
G
4th dequeue
f=8
b
f=7
e
G

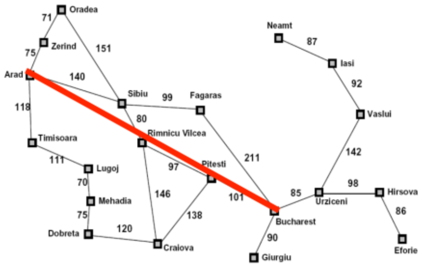
A* Optimality

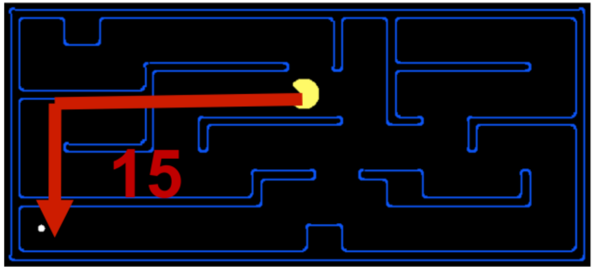


- What went wrong?
- For node A: actual least cost to goal (i.e. goal cost) < estimated least cost to goal (i.e. heuristic)
- We need the estimate to be less than actual least cost to goal (i.e. goal cost) for all nodes!

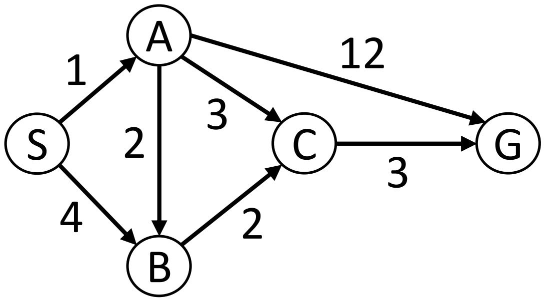
Admissible Heuristics

- A Heuristic h is admissible (optimistic) if:
  - h(n) <= h*(n) for every node “n”, where h*(n) is the true least cost to goal from node “n”
- If the heuristic is admissible, the A* search is optimal
- Coming up with admissible heuristics is most of what’s involved in using A* in practice.
- Example:





A* Optimal Efficiency



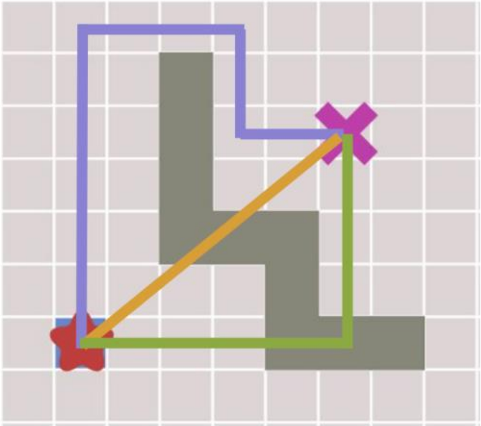
State	H
S	7
A	6
B	2
C	1
G	0

- A Heuristic h is consistent (monotone) if:
  - h(x) <= d(x, y) + h(y) for every edge (x, y)
- Triangle inequality.
- Once a node is expanded, the cost by which it was reached is the lowest possible. If not, a node will need repeated expansion every time a new best (so-far) cost is achieved for it.
- All consistent heuristics are admissible.

Heuristic Design

An admissible heuristic function has to be designed case by case.

- Euclidean Distance
- Manhattan Distance



Is Euclidean distance (L2 norm) admissible? Always

Is Manhattan distance (L1 norm) admissible? Depends

Is L∞ norm distance admissible? Always

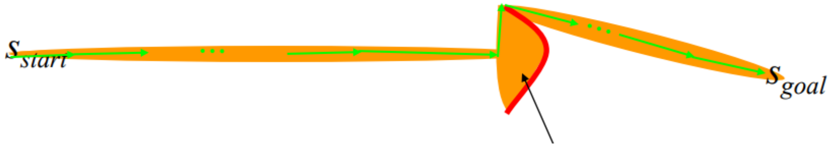
Is 0 distance admissible? Always

Sub-optimal Solution

What if we intend to use an over-estimate heuristic?

- Suboptimal path
- Faster

Weighted A*: Expands states based on $f = g + \varepsilon h, \varepsilon > 1$ =bias towards states that are closer to goal.



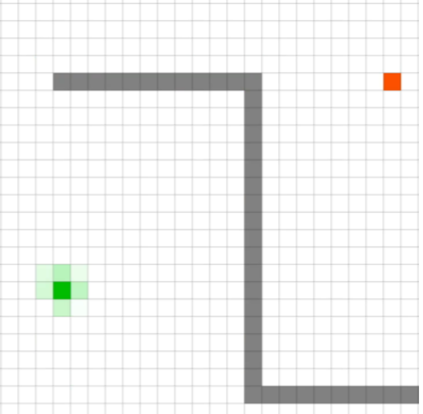
- Weighted A* Search:

➢ Optimality vs. speed
➢ ε-suboptimal:
cost(solution) <= εcost (optimal solution)
➢ It can be orders of magnitude faster than A*

Weighted A*-> $\underbrace{\text{Anytime A*-> ARA*->D*}}_{\text{Beyond the scope of this course}}$

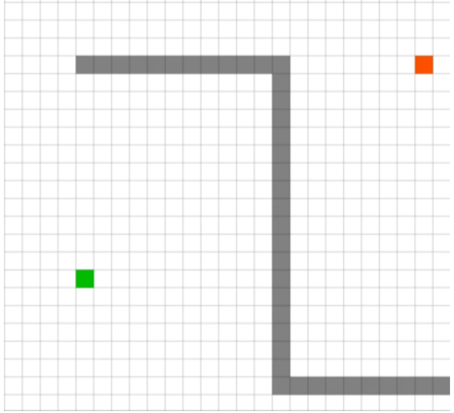
Greedy Best First Search vs. Weighted A* vs. A*

$$f = \color{red}{a} \cdot g + \color{red}{b} \cdot h$$



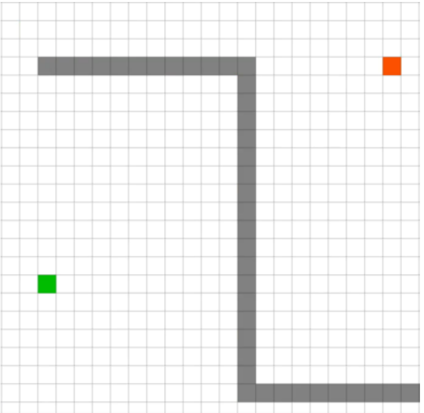
Most Greedy

$$a = 0, b = 1$$



Tunable Greediness

$$a = 1, b = \varepsilon > 1$$



Optimal

$$a = 1, b = 1$$

Animation from: http://qiao.github.io/PathFinding.js/visual/

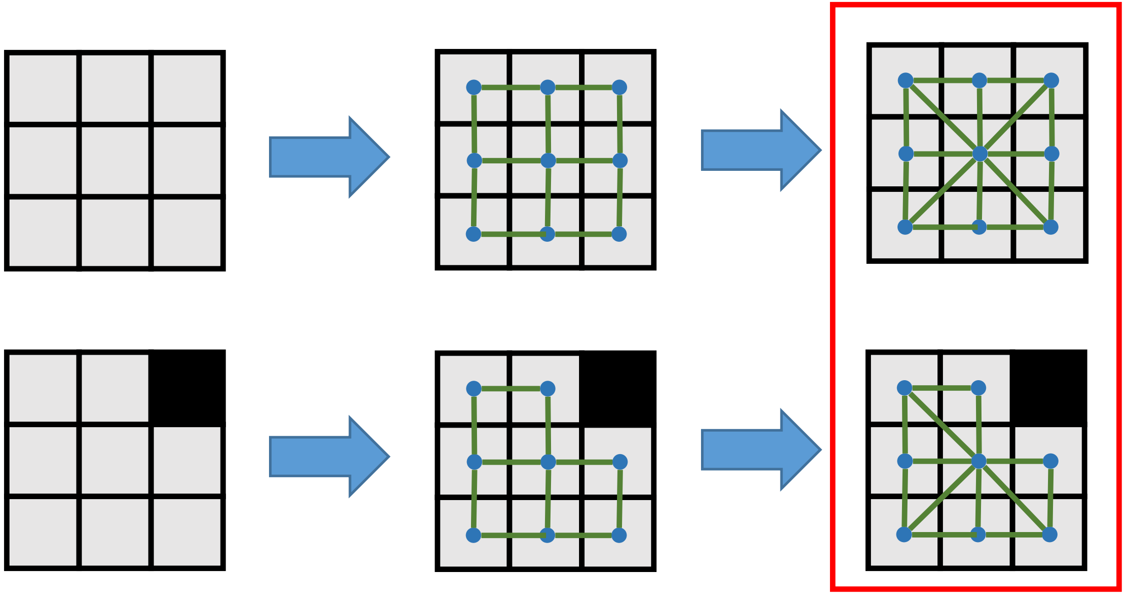
Dijkstra: $a = 1, b = 0$

Engineering Considerations

Example: Grid-based Path Search

How to represent grids as graphs?

Each cell is a node. Edges connect adjacent cells.



Common Choice!

4 connection

8 connection

The Best Heuristic

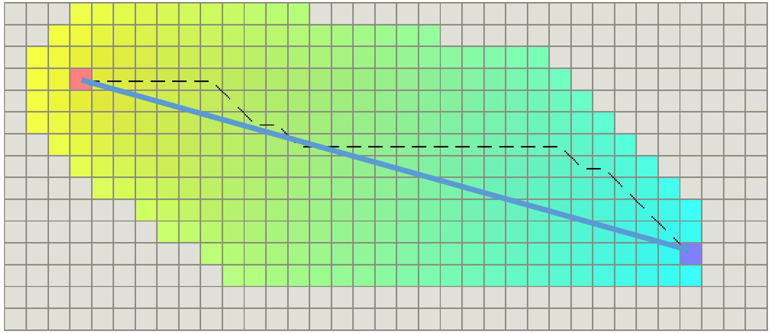
- Recall:
  - Is Euclidean distance (L2 norm) admissible? ✓
  - Is Manhattan distance (L1 norm) admissible? ✗
  - Is L∞ norm distance admissible? ✓
  - Is 0 distance admissible? ✓

They are useful, but none of them is the best choice, why?

Because none of them is tight.

Tight means who close they measure the true shortest distance.

Euclidean Heuristic



Why so many nodes expanded?

Because Euclidean distance is far from the truly theoretical optimal solution.

The Best Heuristic

How to get the truly theoretical optimal solution?

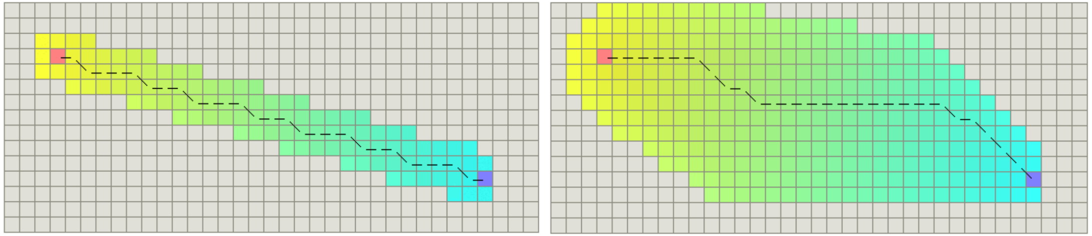
Fortunately, the grid map is highly structural.

- You don’ t need to search the path.
- It has the closed-form solution!

```
dx=abs(node.x −goal.x)
dy=abs(node.y −goal.y)

h=D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)
                or
h = D * max(dx, dy) + (D2-D) * min(dx, dy)
                or
if (dx > dy)
   h = D * (dx-dy) + D2 * dy
else
   h = D * (dy-dx) + D2 * dx
```

Compare



Diagonal Heuristic　　　Euclidean Heuristic

When D = 1 and D2 = 1, this is called the *Chebyshev distance.*
When D = 1 and D2 = sqrt(2), this is called the *Octile distance*

For 3D case, we also have a similar version of this.

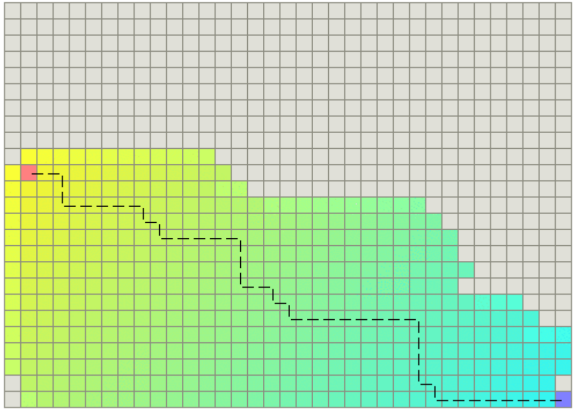
Tie Breaker

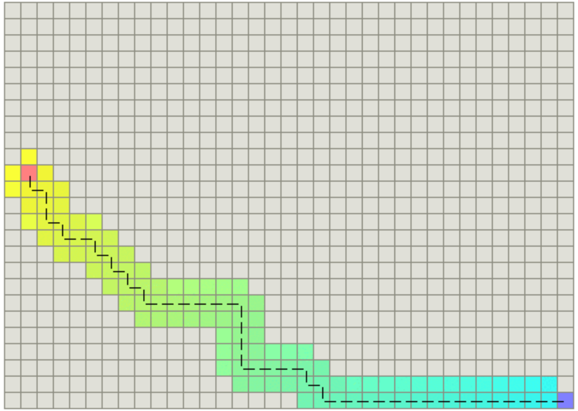
- Many paths have the same f value.
- No differences among them making them explored by A* equally.

- Manipulate the f value breaks the tie.
- Make same f values differ.
- Interfere h slightly.

$$\mathrm{h} = \mathrm{h} \times (1.0 + \mathrm{p})$$

$$p < \frac{minimum\ cost\ of\ one\ step}{expected\ maximum\ path\ cost}$$





Slightly breaks the admissibility of h, does it matter?

Tie Breaker

Core idea of tie breaker:

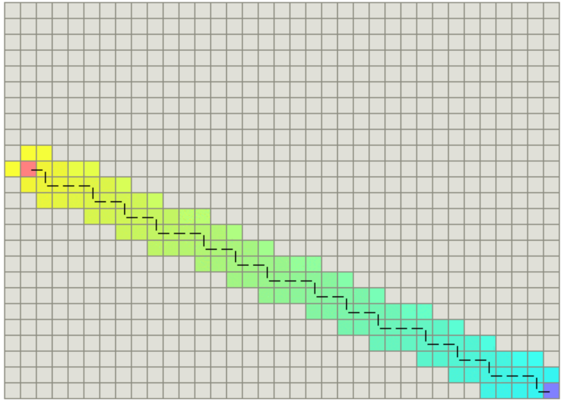
Find a preference among same cost paths

- When nodes having same f, compare their h.
- Add deterministic random numbers to the heuristic or edge costs (A hash of the coordinates).
- Prefer paths that are along the straight line from the starting point to the goal.

$$dx1 = abs(node.x \ - goal.x)$$
$$dy1 = abs(node.y \ - goal.y)$$
$$dx2 = abs(start.x \ - goal.x)$$
$$dy2 = abs(start.y \ - goal.y)$$
$$cross = abs(dx1 \times dy2 - dx2 \times dy1)$$
$$\mathrm{h} = h + cross \times 0.001$$

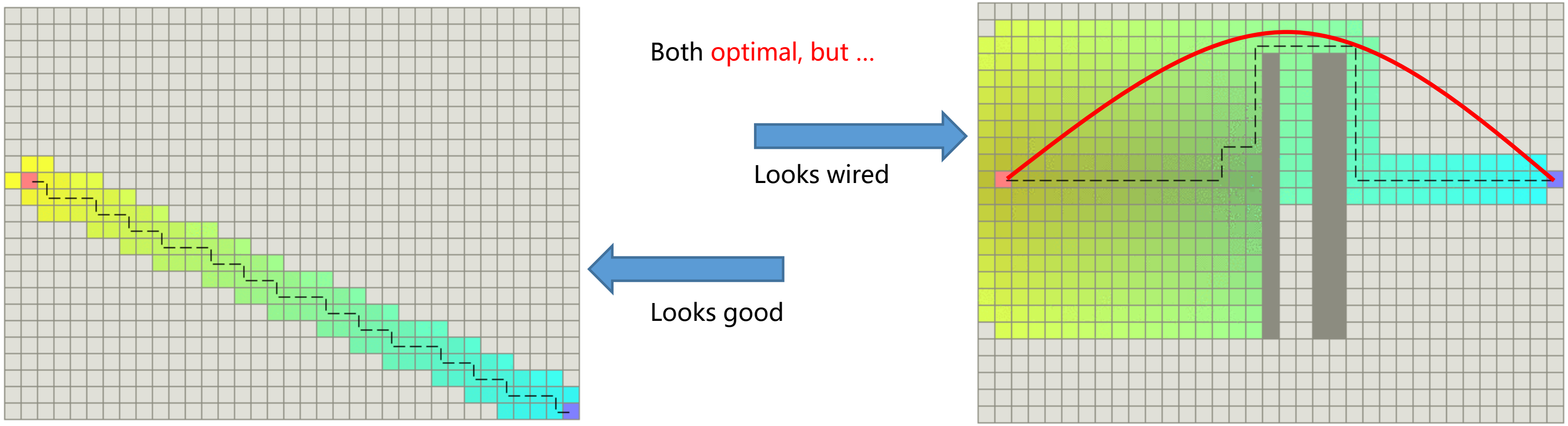
- ... Many customized ways

Better/other **tie breaker**?



Tie Breaker

- Prefer paths that are along the straight line from the starting point to the goal.



It is the shortest path, but harm for trajectory generation (smoothing).

Or a systematic approach: Jump Point Search (JPS)

Jump Point Search

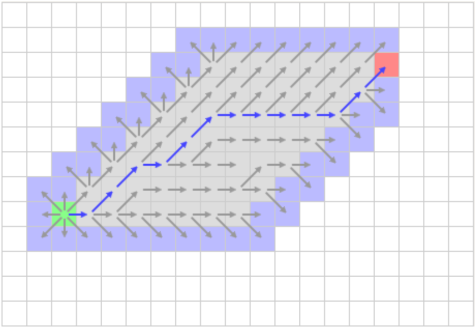
A method to deal with path symmetry for grid-based path search

Jump Point Search

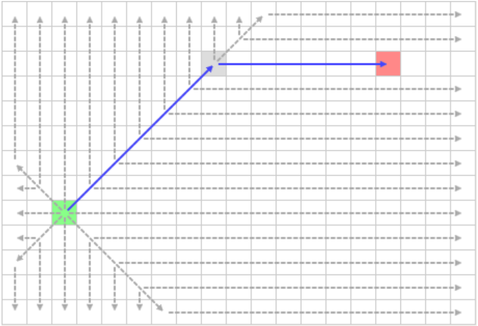
Core idea of JPS:

Find symmetry and break them.

A* explore all symmetric path.



JPS choose one path.

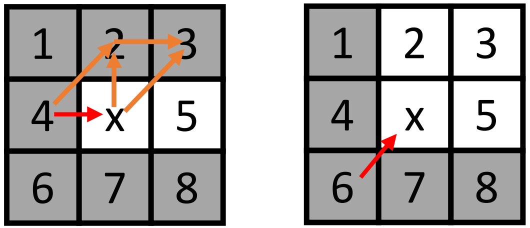


Grey node: Added in the Open List.

JPS explores intelligently, because it always looks ahead based on a rule.

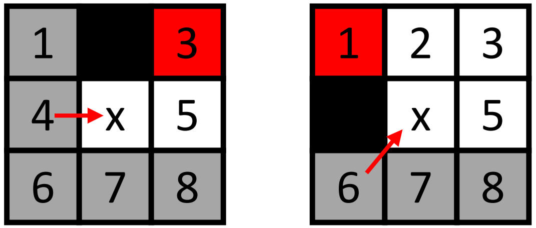
Look Ahead Rule

straight diagonal



Consider:

- current node x
- x's expanded direction



Neighbor Pruning

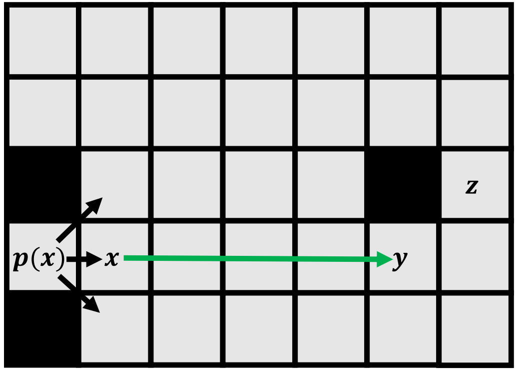
- Gray nodes: inferior neighbors, when going to them, the path without x is cheaper. Discard.
- White nodes: natural neighbors.
- We only need to consider natural neighbors when expand the search.

Forced Neighbors

- There is obstacle adjacent to x
- Red nodes are forced neighbors.
- A cheaper path from x's parent to them is blocked by obstacle.

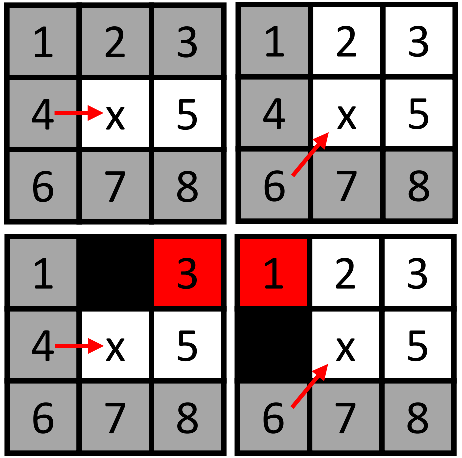
See: http://users.cecs.anu.edu.au/~dharabor/data/papers/harabor-grastien-aaai11.pdf Equation 1/2

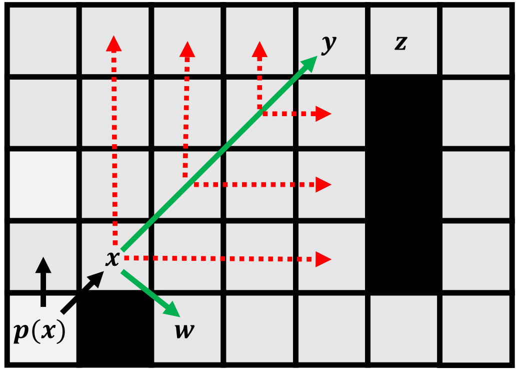
Jumping Rules



Jumping Straight

Look Ahead Rule

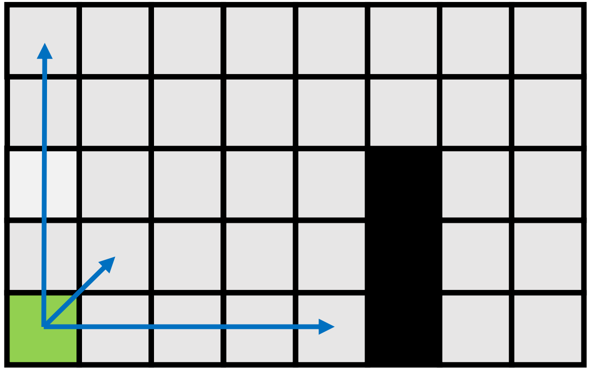




Jumping Diagonally

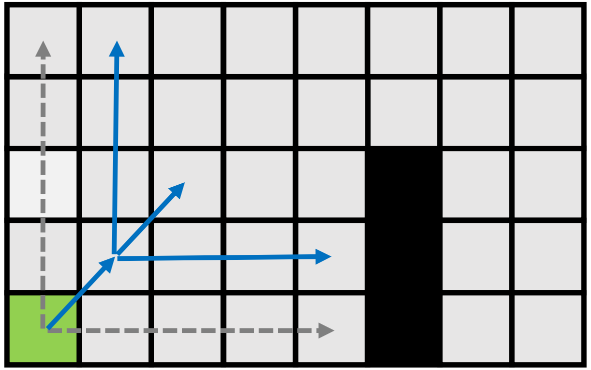
- Recursively apply straight pruning rule and identify y as a jump point successor of x. This node is interesting because it has a neighbor z that cannot be reached optimally except by a path that visits x then y.
- Recursively apply the diagonal pruning rule and identify y as a jump point successor of x.
- Before each diagonal step we first recurse straight. Only if both straight recursions fail to identify a jump point do we step diagonally again.
- Node w, a forced neighbor of x, is expanded as normal. (also push into the open list, the priority queue)

Jump Point Search



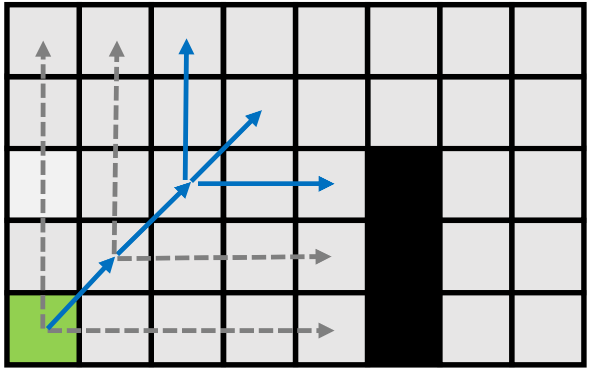
- Expand horizontally and vertically.
- Both jumps end in obstacles.
- Move diagonally.

Jump Point Search



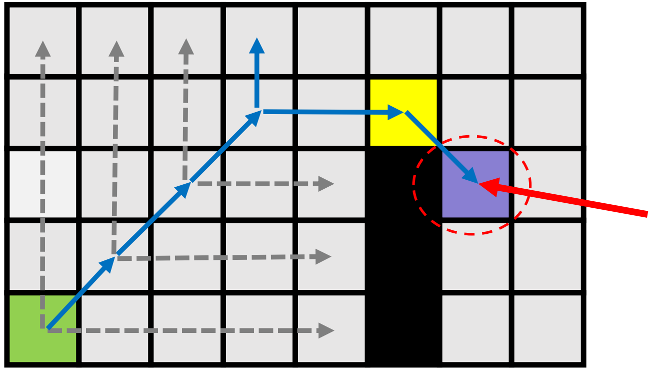
- Expand horizontally and vertically.
- Both jumps end in obstacles.
- Move diagonally.

Jump Point Search



- Expand horizontally and vertically.
- Both expansions end in obstacles.
- Move diagonally.

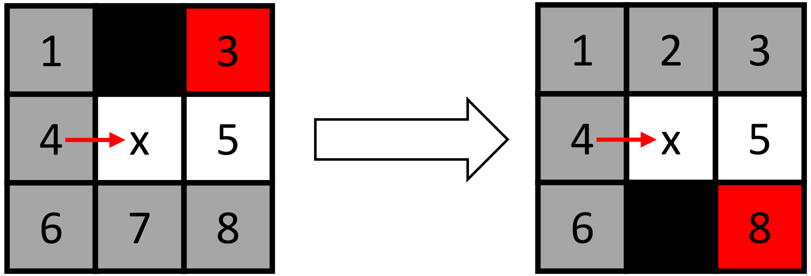
Jump Point Search



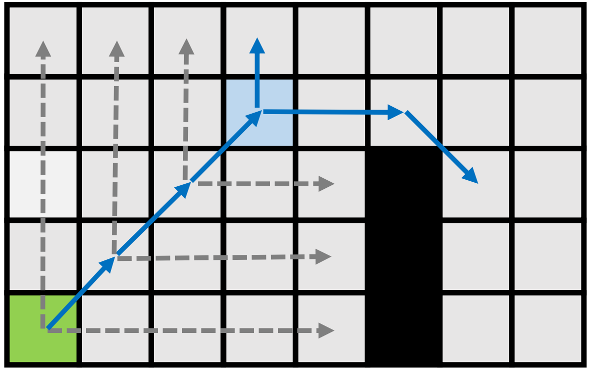
- Vertically expansion end in obstacle.
- Right-ward expansion finds a node with a forced neighbor.

Recall the rule

So we have



Jump Point Search



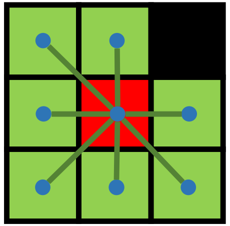
- Now this node is of interested.
- Put it to open list.

Jump Point Search

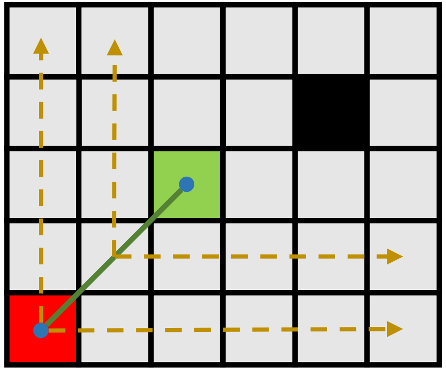
Recall A*’ s pseudo-code, JPS’ s is all the same!

- Maintain a priority queue to store all the nodes to be expanded
- The heuristic function h(n) for all nodes are pre-defined
- The priority queue is initialized with the start state X_S
- Assign $g(X_S)=0$, and g(n)=infinite for all other nodes in the graph
- Loop
  - If the queue is empty, return FALSE; break;
  - Remove the node “n” with the lowest f(n)=g(n)+h(n) from the priority queue
  - Mark node “n” as expanded
  - If the node “n” is the goal state, return TRUE; break;
  - For all unexpanded neighbors “m” of node “n”
    - If g(m) = infinite
      - g(m)= g(n) + C_{nm}
      - Push node “m” into the queue
    - If g(m) > g(n) + C_{nm}
      - g(m)= g(n) + C_{nm}
  - end
- End Loop

A*: “Geometric” neighbors



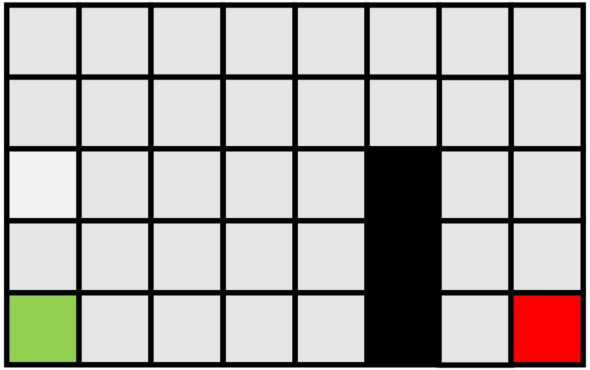
JPS: “Jumping” neighbors



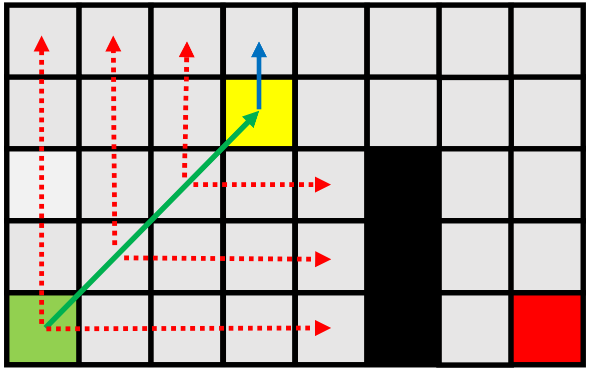
Example

Jumping Example

Planning Case

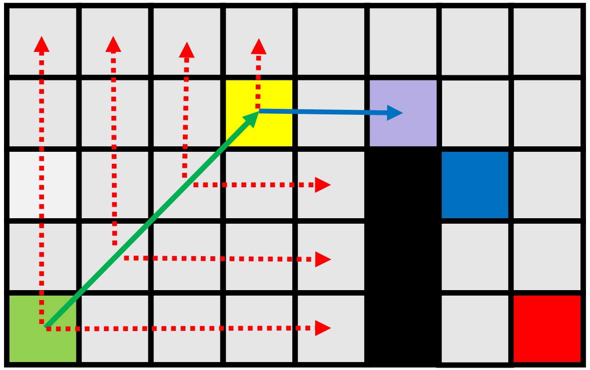


Jumping Example



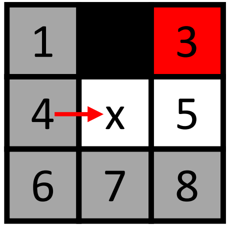
- Expand—>move diagonally
- Find a critical node finally, add it into open list.
- Pop it（the only one） from the open list.
- Expand vertically, end at obstacles.

Jumping Example

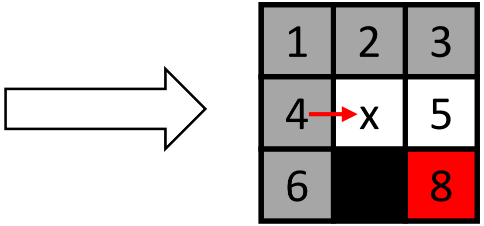


- Expand horizontally, meets a node with a forced neighbor.
- Add it to open list

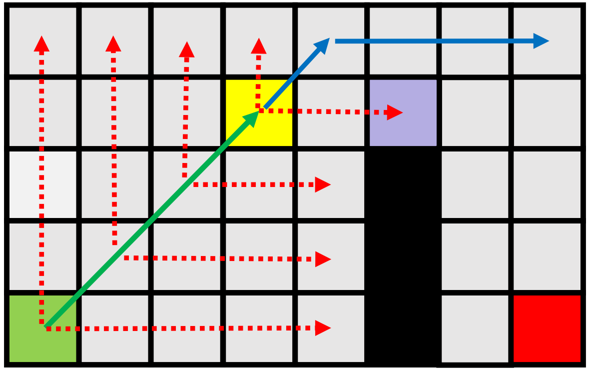
Recall the rule



So we have

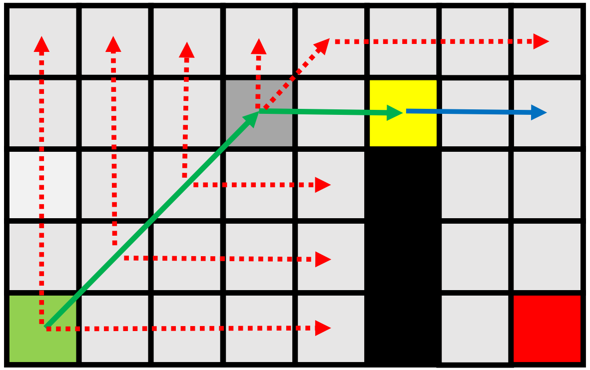


Jumping Example



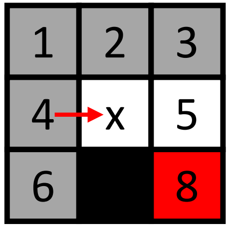
- Expand diagonally, expand, find nothing.
- Finish the expansion of the current node.

Jumping Example

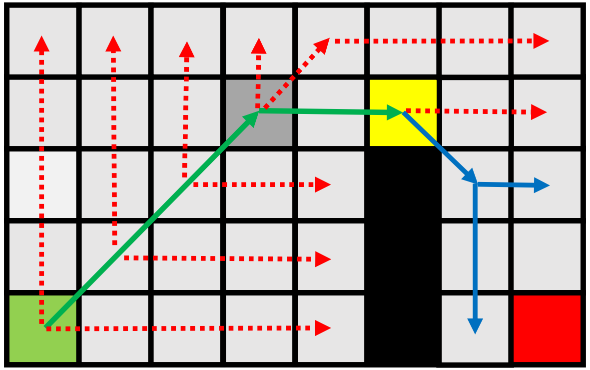


- Examine the "new best" node in the open list.
- Expand horizontally.
- Finds nothing.

Remember the rule

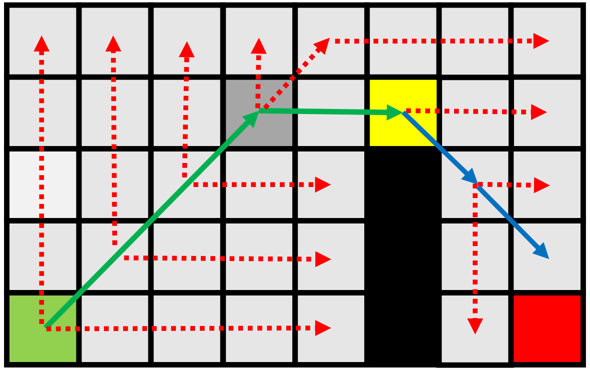


Jumping Example



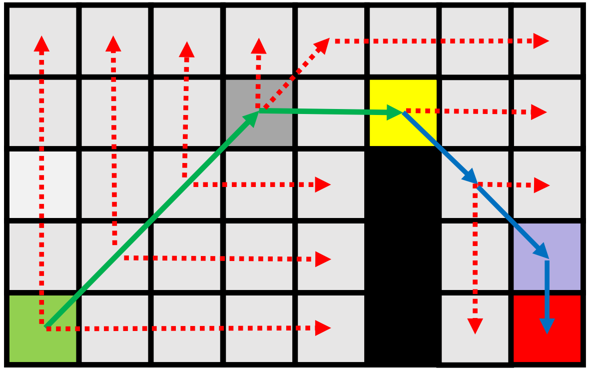
- Move diagonally.
- Expand along vertical and horizontal first.

Jumping Example



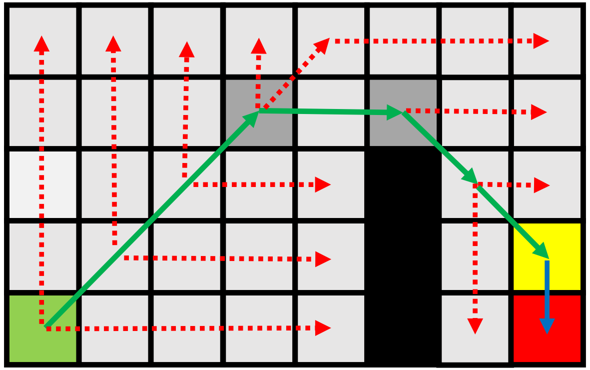
- Finds nothing.
- Move diagonally.

Jumping Example



- Expand horizontally and vertically.
- Finds the goal. Equally interested as finding a node with a forced neighbor.
- Add this node to open list.
- Finish the expand of the current node (No naturally neighbors left).
- Pop it out of the open list.

Jumping Example

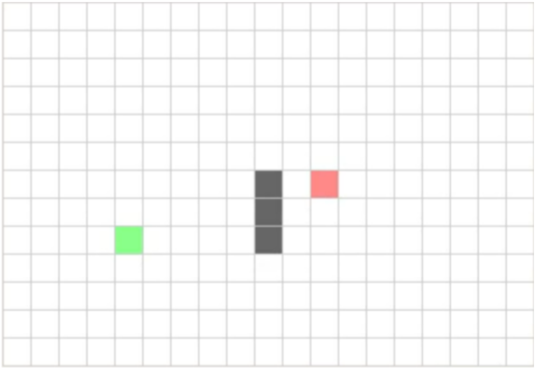


- Examine the “new best” node in the open list.
- Expand horizontally (nowhere), and vertically (finds the goal).
- The end.

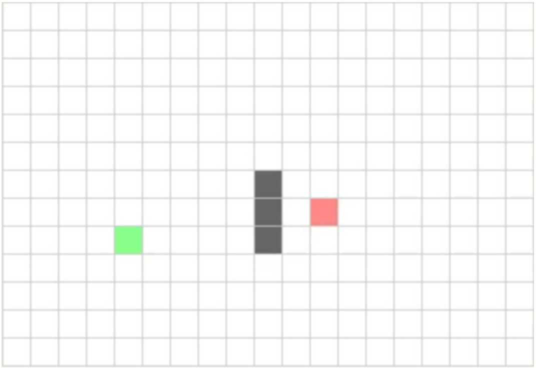
Jumping Example

Final Path

Example



(1)



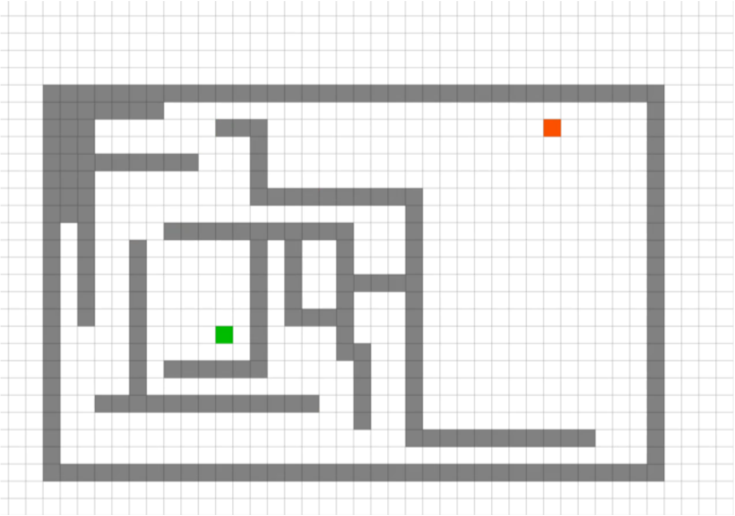
(2)

Thanks:

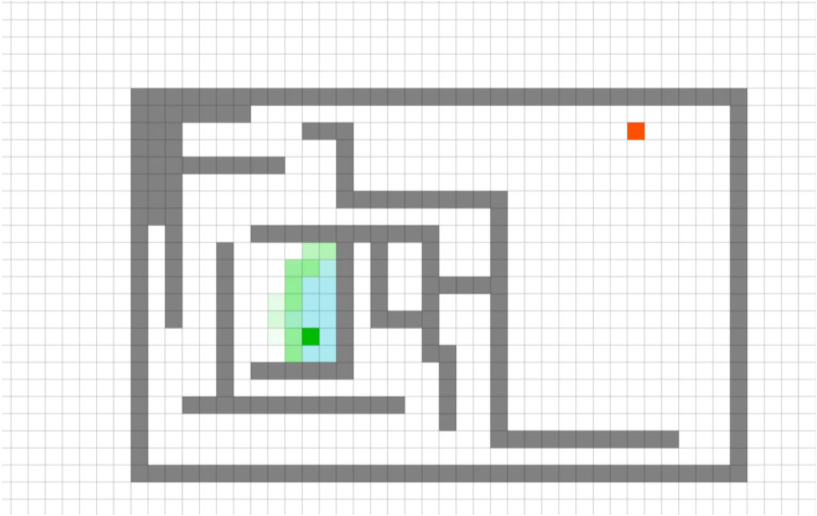
https://zerowidth.com/2013/a-visual-explanation-of-jump-point-search.html

Is JPS always better?

Maze-like environments



JPS

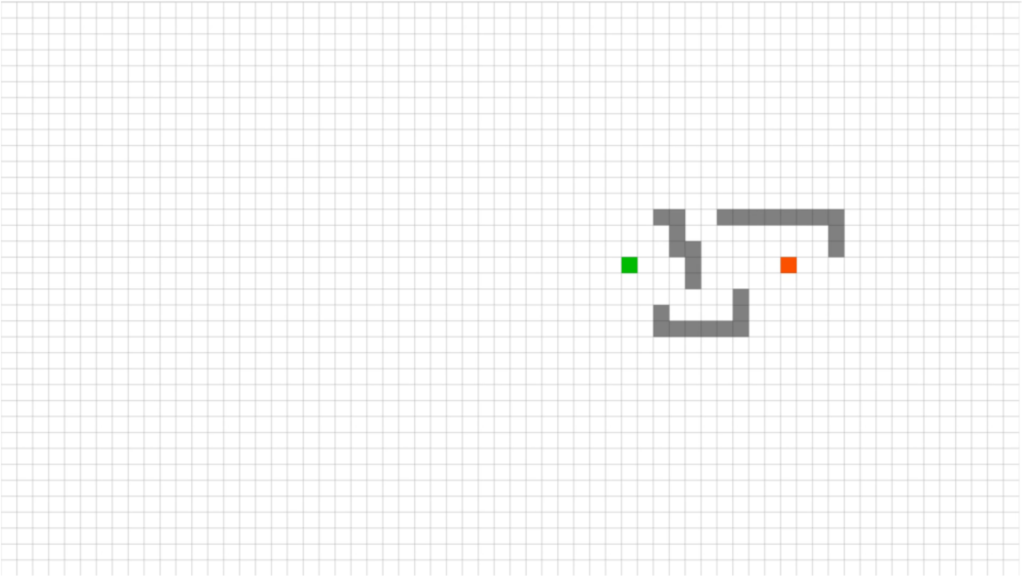


A*

Do more tests by yourself!

Thanks: http://qiao.github.io/PathFinding.js/visual/

Is JPS always better?



- This is a simple example saying "No."
- This case may commonly occur in robot navigation.
- Robot with limited FOV, but a global map/large local map.

Conclusion:

- Most time, especially in complex environments, JPS is better, but far away from "always". Why?
- JPS reduces the number of nodes in Open List, but increases the number of status query.
- JPS's limitation: only applicable to uniform grid map.

Thanks for Listening