# ECE276B: Planning & Learning in Robotics
# Lecture 7: Search-Based Motion Planning

Nikolay Atanasov

natanasov@ucsd.edu

UC San Diego

JACOBS SCHOOL OF ENGINEERING

Electrical and Computer Engineering

# Outline

# Motion Planning as Deterministic Shortest Path

- Construct a graph (via cell decomposition, skeletonization, as a lattice, etc.) and search it for a least-cost path
- **Assumption**: there are no negative cycles in the graph, i.e., $J^{i_{1:q}} \geq 0$ for all $i_{1:q} \in \mathcal{P}_{i,i}$ and all $i \in \mathcal{V}$
- The deterministic shortest path problem can be solved via:
  - **Dynamic Programming**: computes shortest paths from *all* nodes to the goal
  - **Forward Dynamic Programming**: computes shortest paths from the start to *all* nodes
  - **Label correcting methods**: visit only promising nodes
- **Key ideas** of label correcting methods:
  - **Label** $g_i$: lowest cost discovered so far from $s$ to node $i \in \mathcal{V}$
  - **Label correction**: each time $g_i$ is reduced, the labels $g_j$ of the **children** of $i$ can be corrected: $g_j = g_i + c_{ij}$
  - **OPEN**: set of nodes that can potentially be part of the shortest path to $\tau$

# Label Correcting Algorithm

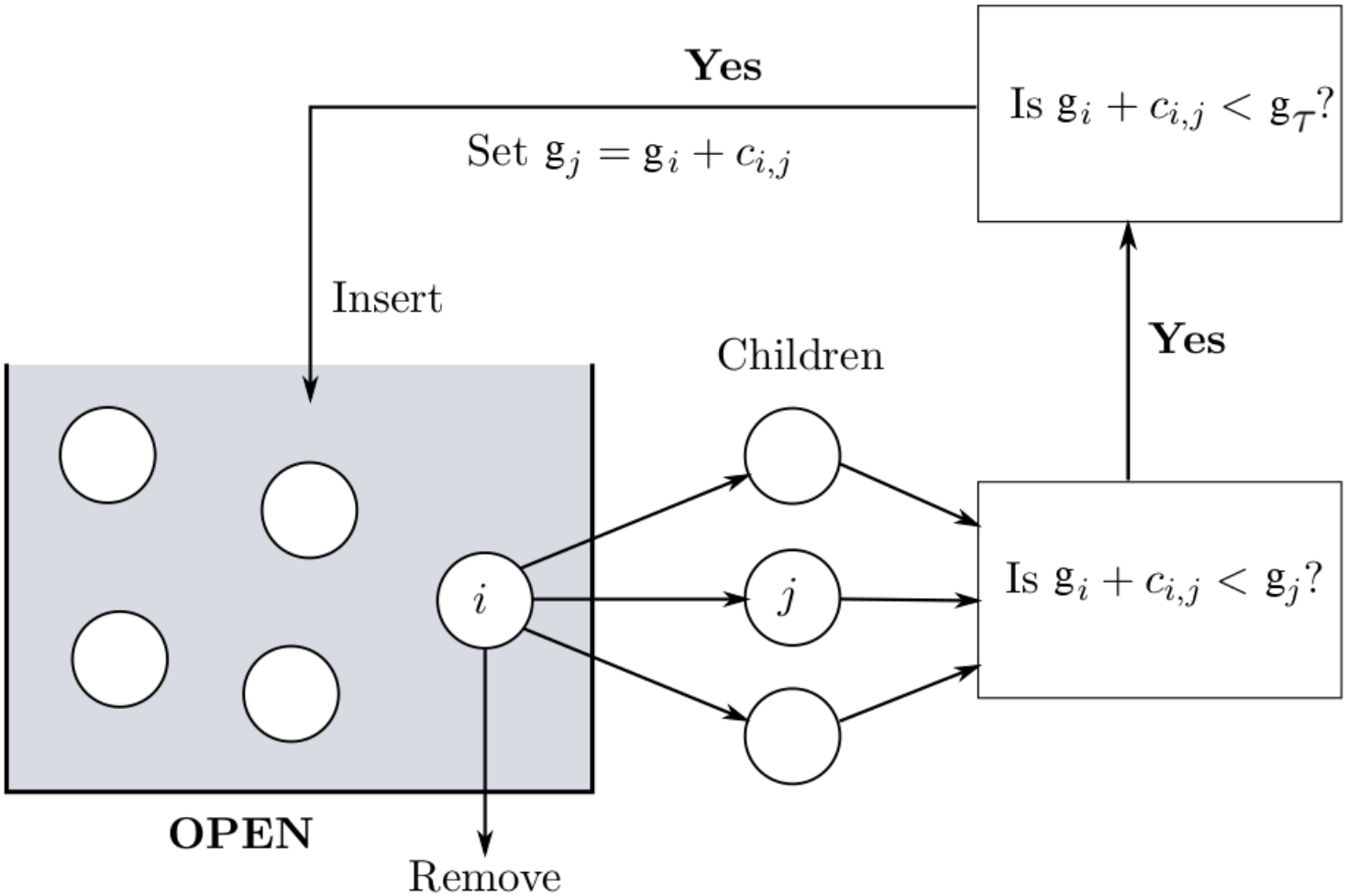
**Algorithm** Label Correcting Algorithm

1: OPEN $\leftarrow \{s\}$, $g_s = 0$, $g_i = \infty$ for all $i \in \mathcal{V} \setminus \{s\}$
2: **while** OPEN is not empty **do**
3: Remove $i$ from OPEN
4: **for** $j \in$ Children$(i)$ **do**
5: **if** $(g_i + c_{ij}) < g_j$ **and** $(g_i + c_{ij}) < g_\tau$ **then** ▷ Only when $c_{ij} \geq 0$ for all $i, j \in \mathcal{V}$
6: $g_j \leftarrow (g_i + c_{ij})$
7: Parent$(j) \leftarrow i$
8: **if** $j \neq \tau$ **then**
9: OPEN $\leftarrow$ OPEN $\cup \{j\}$

## Theorem

If there exists at least one finite cost path from $s$ to $\tau$, then the Label Correcting (LC) algorithm terminates with $g_\tau = \textbf{dist}(s, \tau)$, the shortest path length from $s$ to $\tau$. Otherwise, the LC algorithm terminates with $g_\tau = \infty$.

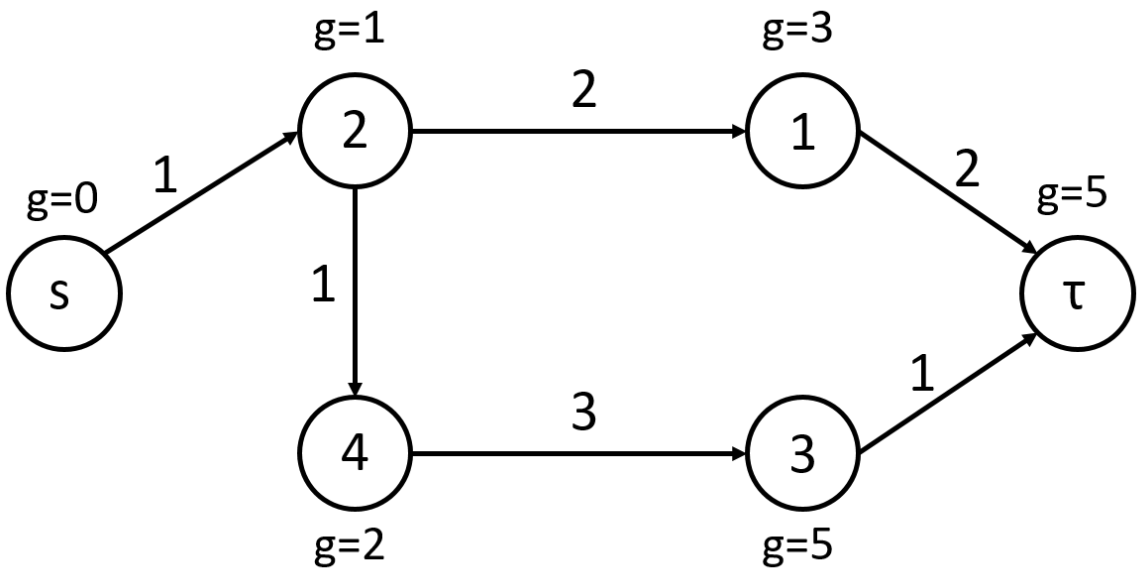
# Label Correcting Algorithm



# Label Correcting Algorithm

- Label correcting algorithms update the labels ($g$-values) of relevant states in OPEN until they satisfy the Bellman equation in dynamic programming:

$$g_i = \min_{j \in Parents(i)} g_j + c_{ji}$$




- **Path recovery**: once correct $g$-values are available, the least-cost path $i_q^*, \ldots, i_1^*$ is a greedy path computed starting from $i_1^* = \tau$ and backtracking:

$$i_{k+1}^* = \arg\min_{j \in Parents(i_k^*)} g_j + c_{j,i_k^*} \qquad \text{until } i_{k+1}^* = s$$

# Outline

# Dijkstra's Algorithm

- **Best-first search**: label correcting algorithm that removes nodes with minimum label $g_i$ from OPEN (implemented as a **priority queue**):

$$i \in \arg\min_{j \in OPEN} g_j$$

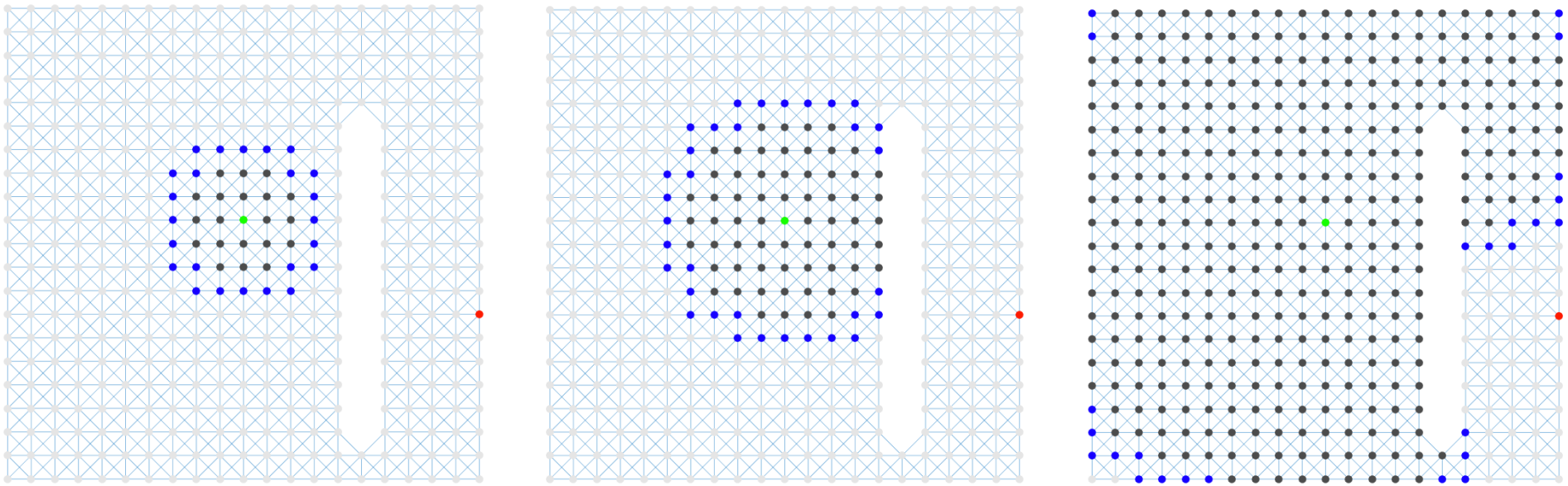
- **Node expansion**: once removed from OPEN, node $i$ is expanded by correcting the labels of its children $j \in \mathit{Children}(i)$
- **Upon termination**: $g_i$ equals the true cost $\mathbf{dist}(s, i)$ of the shortest path from $s$ to $i$ for all expanded nodes

# Dijkstra's Algorithm Properties

- When $c_{ij} \geq 0$
  - The algorithm expands nodes in the order of distance from $s$
  - Each node is expanded at most once
  - If $i \in \mathit{OPEN}$, its label $g_i$ may decrease as we discover new paths to $i$
  - $g_i \geq \mathbf{dist}(s, i)$ always with equality once $i$ is expanded
  - Once we remove $i$ from OPEN, its label $g_i$ can no longer change because all other nodes in OPEN have higher g-values. We cannot hope to find a shorter path to $i$ passing through a node in OPEN.
  - OPEN is the "search frontier" and separates expanded from unexplored nodes. Hence, once a node is removed from OPEN, we cannot hope to find a better path to it. **A node will enter OPEN at most once**.
  - Once $\tau$ is removed from OPEN, we cannot discover a shorter path to $\tau$: **Done**!
- When $c_{ij}$ may be negative but there are no negative cycles and $\mathbf{dist}(i, \tau) \geq 0$ for all $i \in \mathcal{V}$
  - Nodes may be expanded more than once, i.e., may re-enter OPEN
  - No guarantee that $g_i \geq \mathbf{dist}(s, i)$ throughout the execution
  - The algorithm terminates with $g_i = \mathbf{dist}(s, i)$

# OPEN is the Search Frontier

- Dijkstra's algorithm may be thought of as fluid flow starting from $s$ and expanding out
- The costs $c_{ij}$ specify the time for the fluid to traverse edge $i \to j$
- When the fluid arrives at a node $i$, we update the ETA $g_j$ of its neighbors $j$
- Some ETA estimates may be too large since the fluid may find shortcuts



- The order of node expansions in Dijkstra only considers $g_i$, the cost from $s$ to $i$ but does **not** consider how costly the path from $i$ to $\tau$ might be. Can this be estimated and used to improve the search?

# Outline

# A* Algorithm

- The **A* algorithm** is a modification of the label correcting algorithm with $c_{ij} \geq 0$ in which the requirement for admission to OPEN is strengthened:

$$\textbf{from} \quad g_i + c_{ij} < g_\tau \quad \textbf{to} \quad g_i + c_{ij} + h_j < g_\tau$$

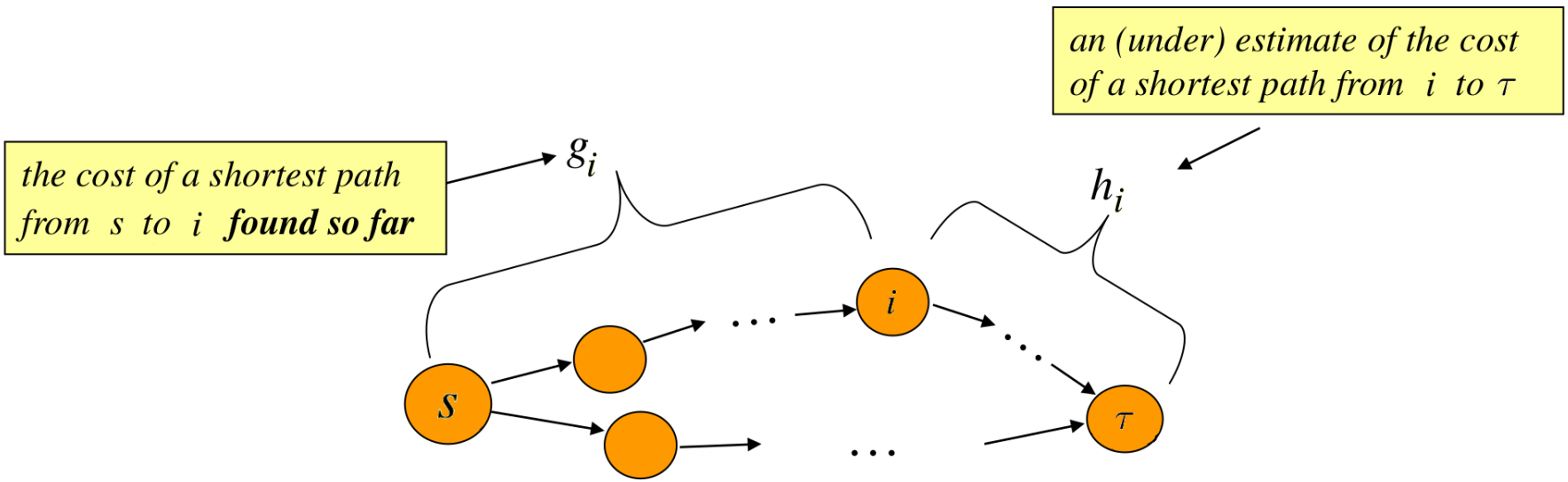
  where $h_j$ is a non-negative lower bound on the optimal cost-to-go from node $j$ to $\tau$ known as a **heuristic function**:

$$0 \leq h_j \leq \textbf{dist}(j, \tau)$$

- The more stringent criterion for admission to OPEN can reduce the number of iterations required by the label correcting algorithm to find an optimal path
- The more accurately $h_j$ estimates $\textbf{dist}(j, \tau)$, the more efficient the A* algorithm becomes!

# Heuristic Function

- A heuristic function $h_i$ is constructed using special knowledge about the problem

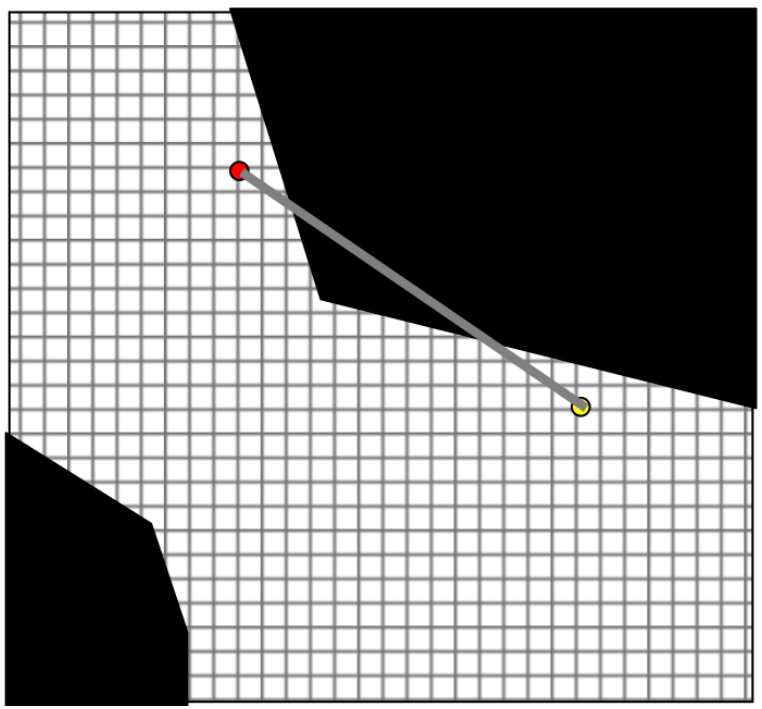


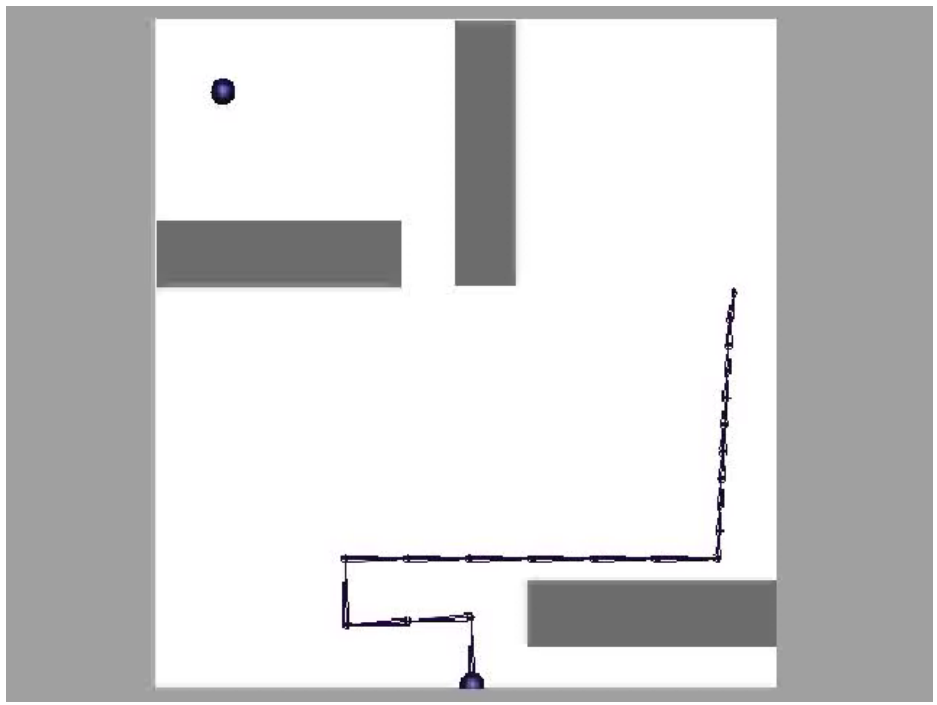
# Heuristic Function

- A heuristic must be admissible for the A* algorithm to work correctly
- A heuristic may be consistent to make the A* algorithm more efficient
- **Admissible**: $h_i \leq \textbf{dist}(i, \tau)$ for all $i \in \mathcal{V}$
- **Consistent**: $h_\tau = 0$ and $h_i \leq c_{ij} + h_j$ for all $i \neq \tau$ and $j \in \text{Children}(i)$
  - $h$ satisfies the **triangle inequality**, which implies it is also admissible
  - If $h^{(1)}$ and $h^{(2)}$ are consistent, then $h := \max\{h^{(1)}, h^{(2)}\}$ is consistent
  - If $h^{(1)}$ and $h^{(2)}$ are consistent, then $h := h^{(1)} + h^{(2)}$ is $\epsilon$-consistent ($\epsilon = 2$)
- $\epsilon$-**Consistent**: $h_\tau = 0$ and $h_i \leq \epsilon c_{ij} + h_j$ for all $i \neq \tau$, $j \in \text{Children}(i)$, and $\epsilon \geq 1$
- A heuristic function $h^{(2)}$ **dominates** $h^{(1)}$ if both are admissible and $h_i^{(2)} \geq h_i^{(1)}$ for every node $i \in \mathcal{V}$
  - Extreme cases: $h_i = 0$ and $h_i = \textbf{dist}(i, \tau)$

# Examples of Heuristic Functions

- Grid-based planning: let $\mathbf{x}_i \in \mathbb{R}^d$ be the position of node $i$
  - Euclidean distance: $h_i := \|\mathbf{x}_\tau - \mathbf{x}_i\|_2$
  - Manhattan distance: $h_i := \|\mathbf{x}_\tau - \mathbf{x}_i\|_1 := \sum_k |x_{\tau,k} - x_{i,k}|$
  - Diagonal distance: $h_i := \|\mathbf{x}_\tau - \mathbf{x}_i\|_\infty := \max_k |x_{\tau,k} - x_{i,k}|$
  - Octile distance: $h_i := \max_k |x_{\tau,k} - x_{i,k}| + (\sqrt{d} - 1) \min_k |x_{\tau,k} - x_{i,k}|$
- Robot arm planning:
  - End-effector distance: run 2-D Dijkstra for the end effector and use it as a heuristic in the $n$-dimensional search for a joint angle path





# A* Algorithm with an $\epsilon$-consistent Heuristic

**Algorithm** Weighted A* Algorithm

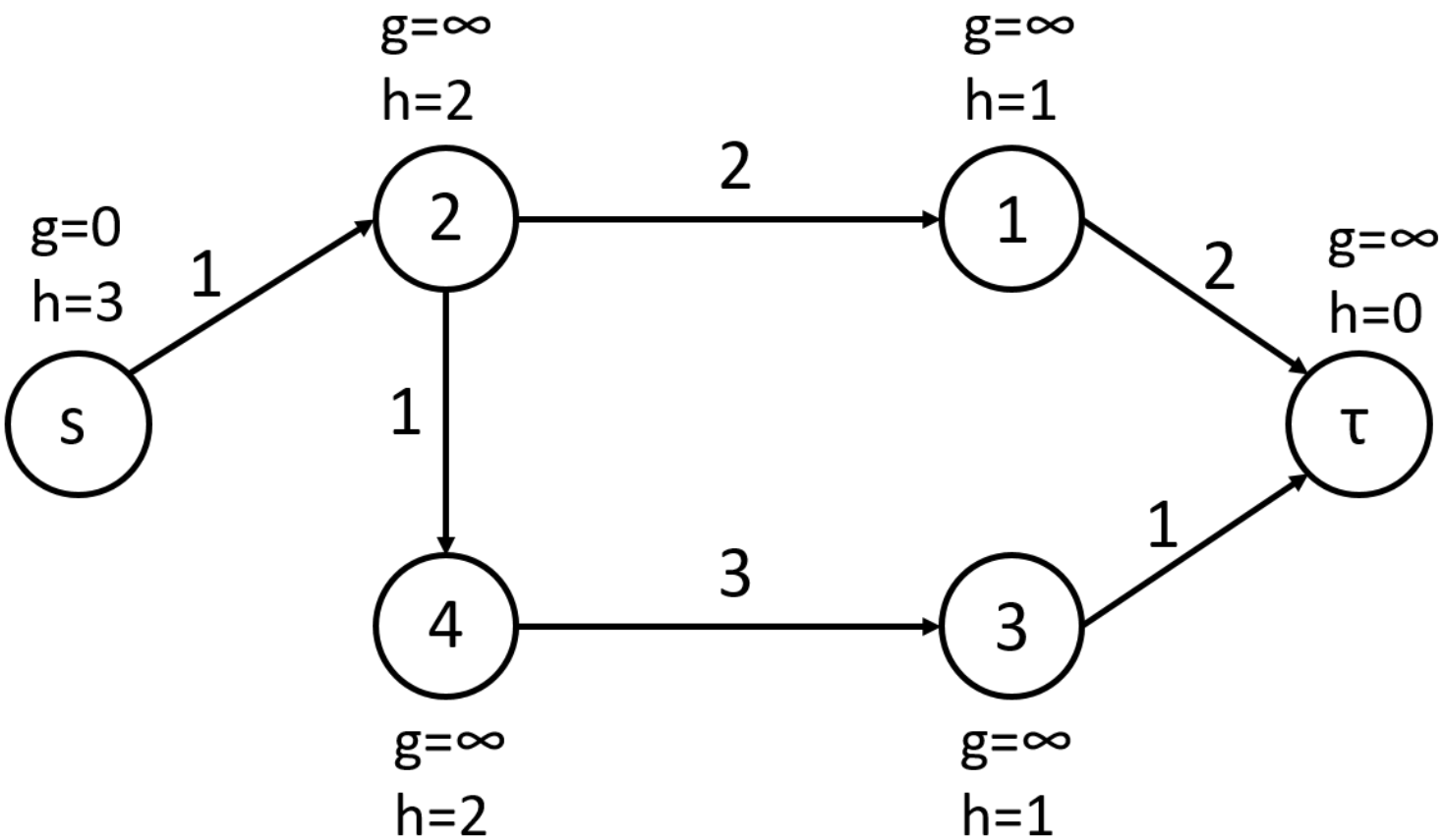
1: OPEN $\leftarrow \{s\}$, CLOSED $\leftarrow \{\}$, $\epsilon \geq 1$
2: $g_s = 0$, $g_i = \infty$ for all $i \in \mathcal{V} \setminus \{s\}$
3: **while** $\tau \notin$ CLOSED **do** ▷ $\tau$ not expanded yet
4: Remove $i$ with smallest $f_i := g_i + \epsilon h_i$ from OPEN ▷ means $g_i + \epsilon h_i < g_\tau$
5: Insert $i$ into CLOSED
6: **for** $j \in$ Children($i$) and $j \notin$ CLOSED **do**
7: **if** $g_j > (g_i + c_{ij})$ **then**
8: $g_j \leftarrow (g_i + c_{ij})$
9: Parent($j$) $\leftarrow i$
10: **if** $j \in$ OPEN **then**
11: Update priority of $j$
12: **else**
13: OPEN $\leftarrow$ OPEN $\cup\{j\}$

expand state $i$:
- try to decrease $g_j$ using path from $s$ to $i$

- There are 3 kinds of states:
  - **CLOSED**: set of states that have already been expanded
  - **OPEN**: set of candidates for expansion (frontier)
  - Unexplored: the rest of the states

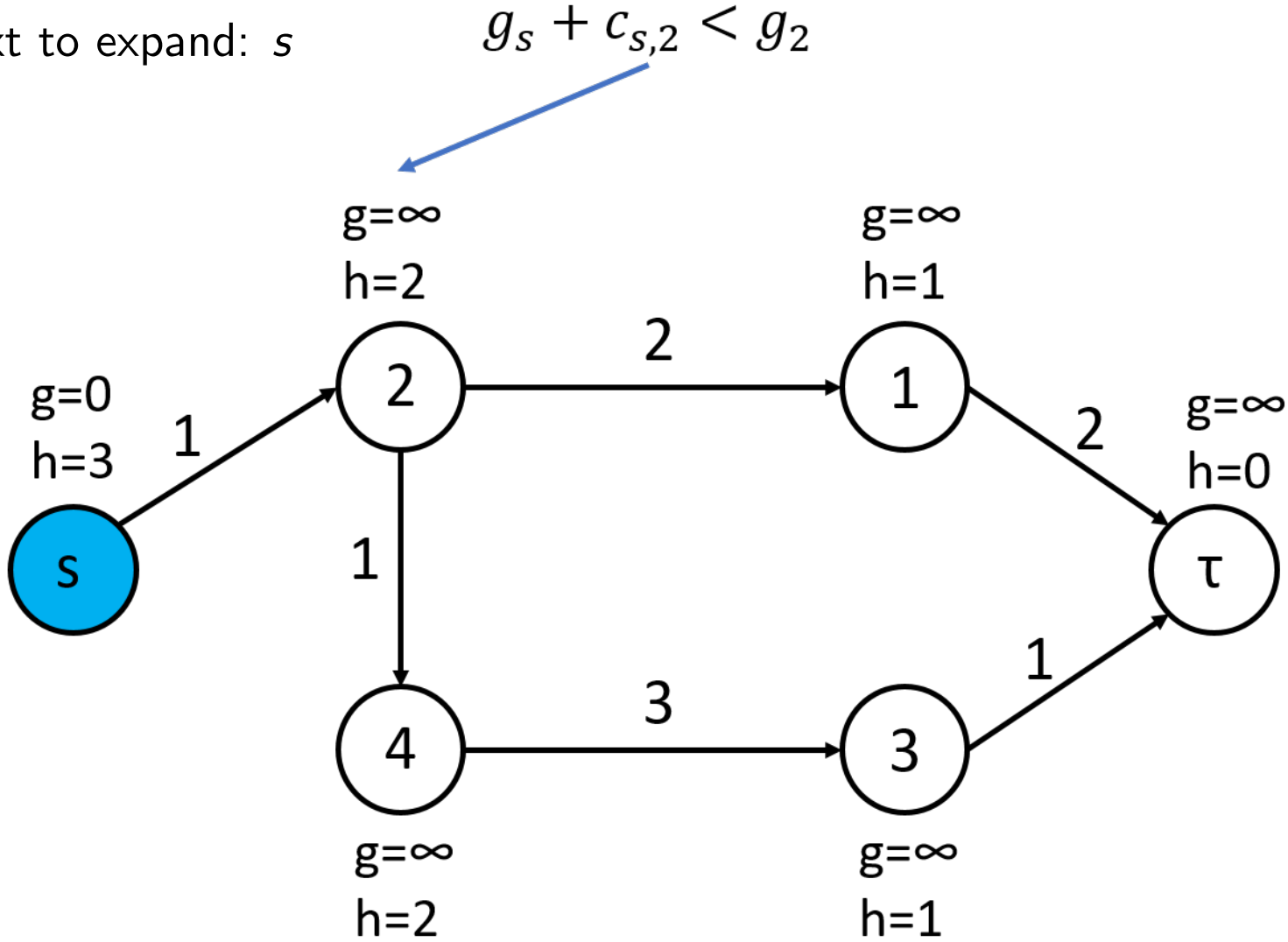
# Example: A* Algorithm

- $OPEN = \{s\}$
- $CLOSED = \{\}$
- Next to expand: $s$



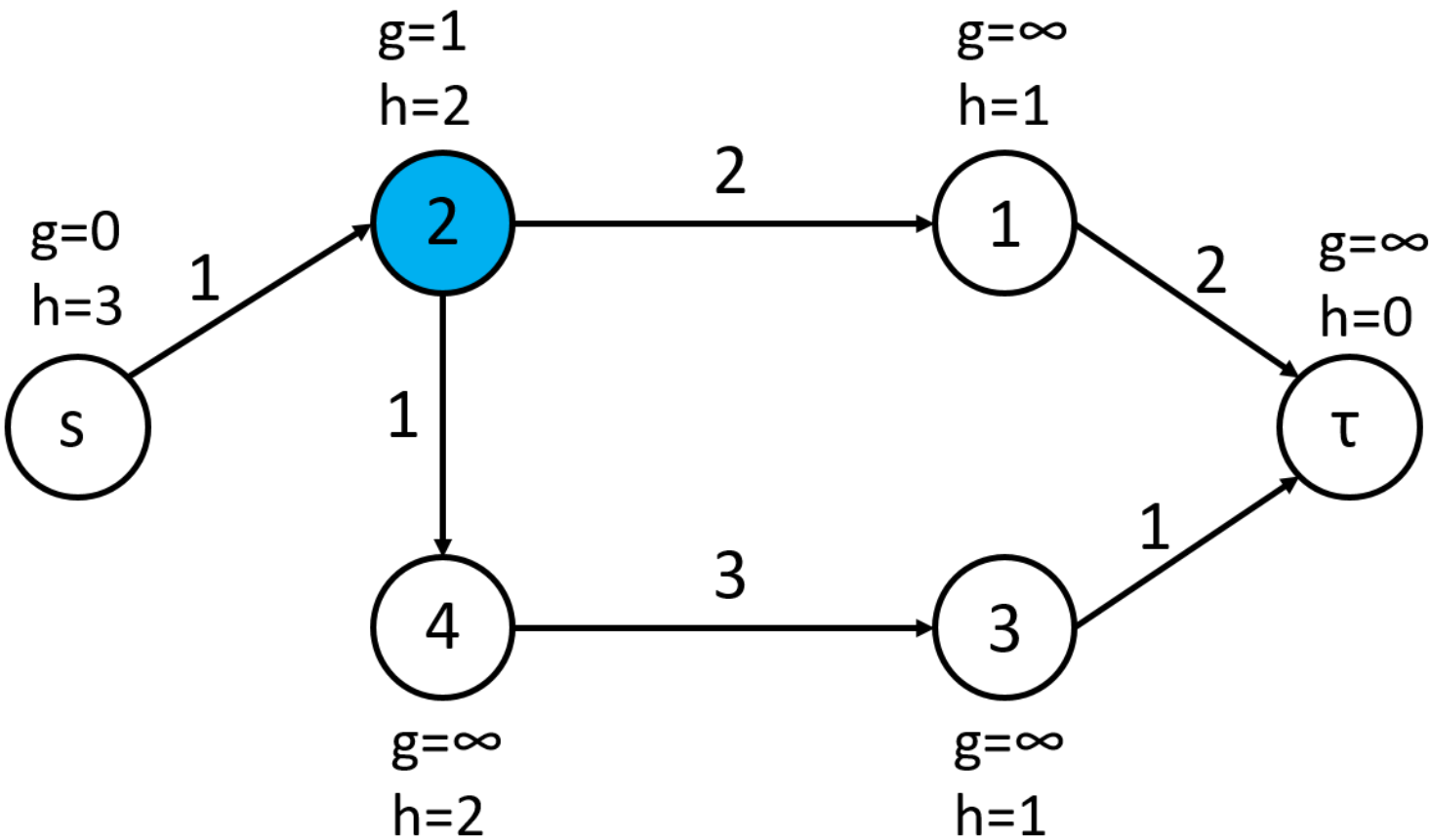
# Example: A* Algorithm

- $OPEN = \{s\}$
- $CLOSED = \{\}$
- Next to expand: $s$



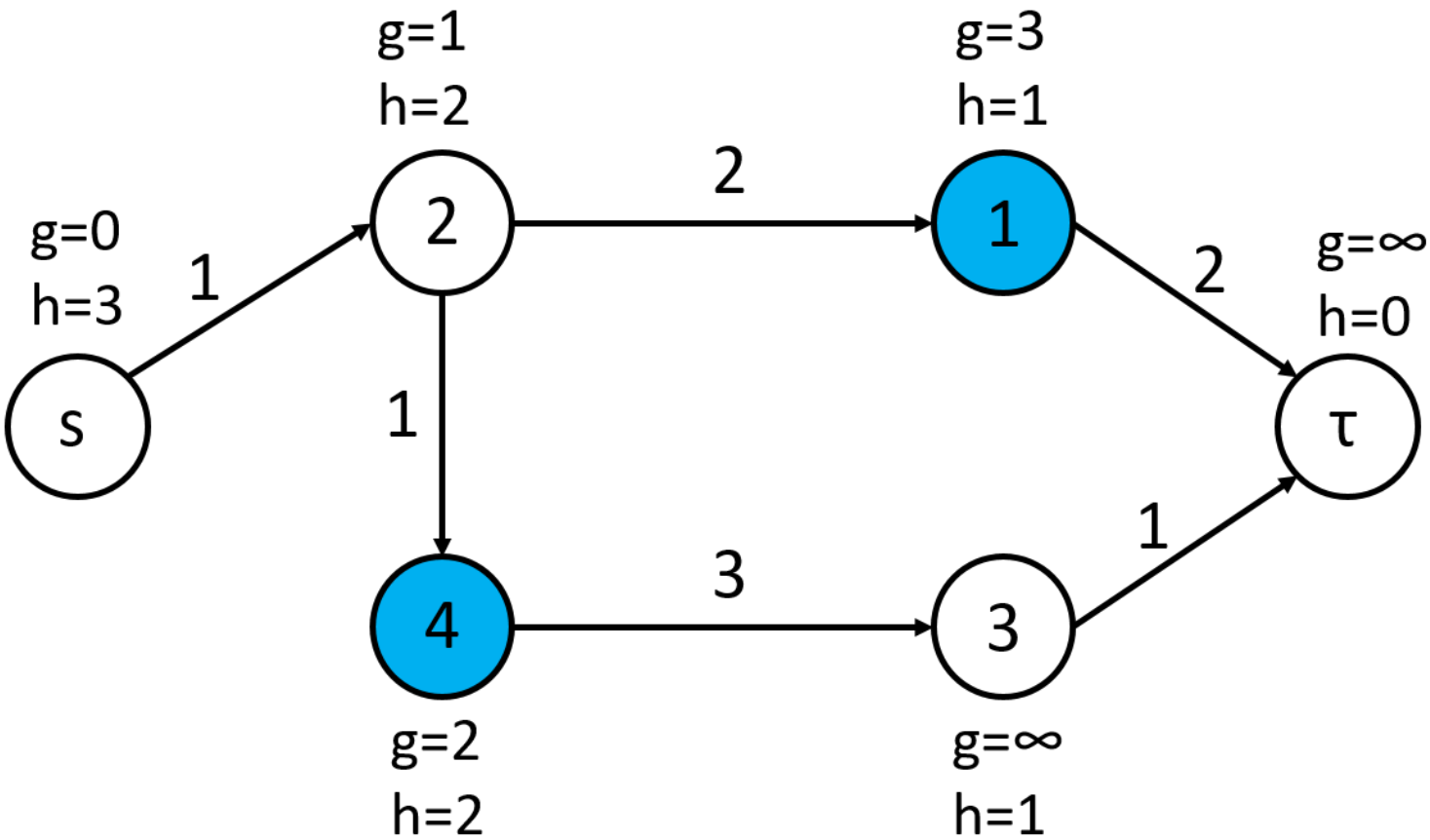
# Example: A* Algorithm

- $OPEN = \{2\}$
- $CLOSED = \{s\}$
- Next to expand: 2



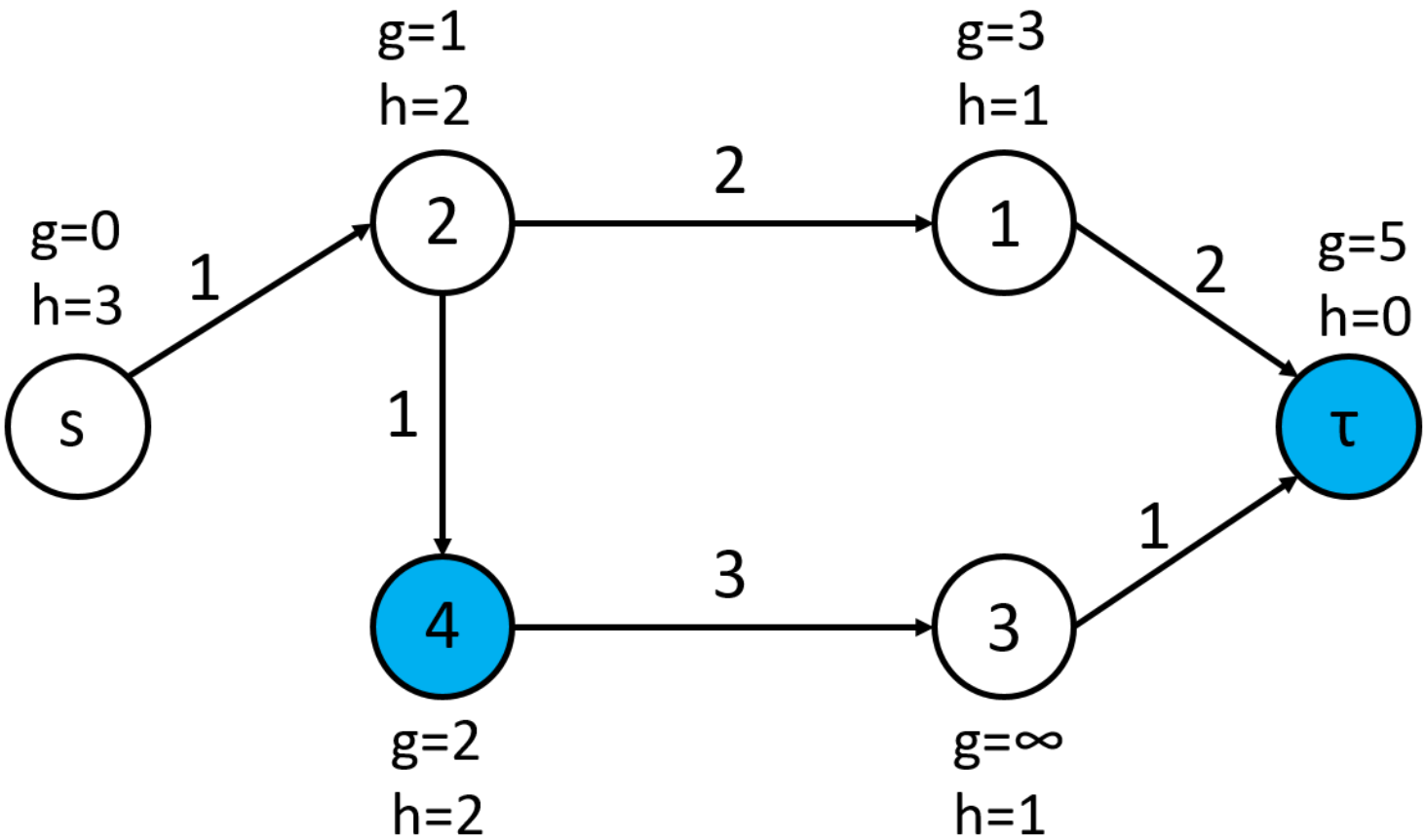
# Example: A* Algorithm

- $OPEN = \{1, 4\}$
- $CLOSED = \{s, 2\}$
- Next to expand: 1



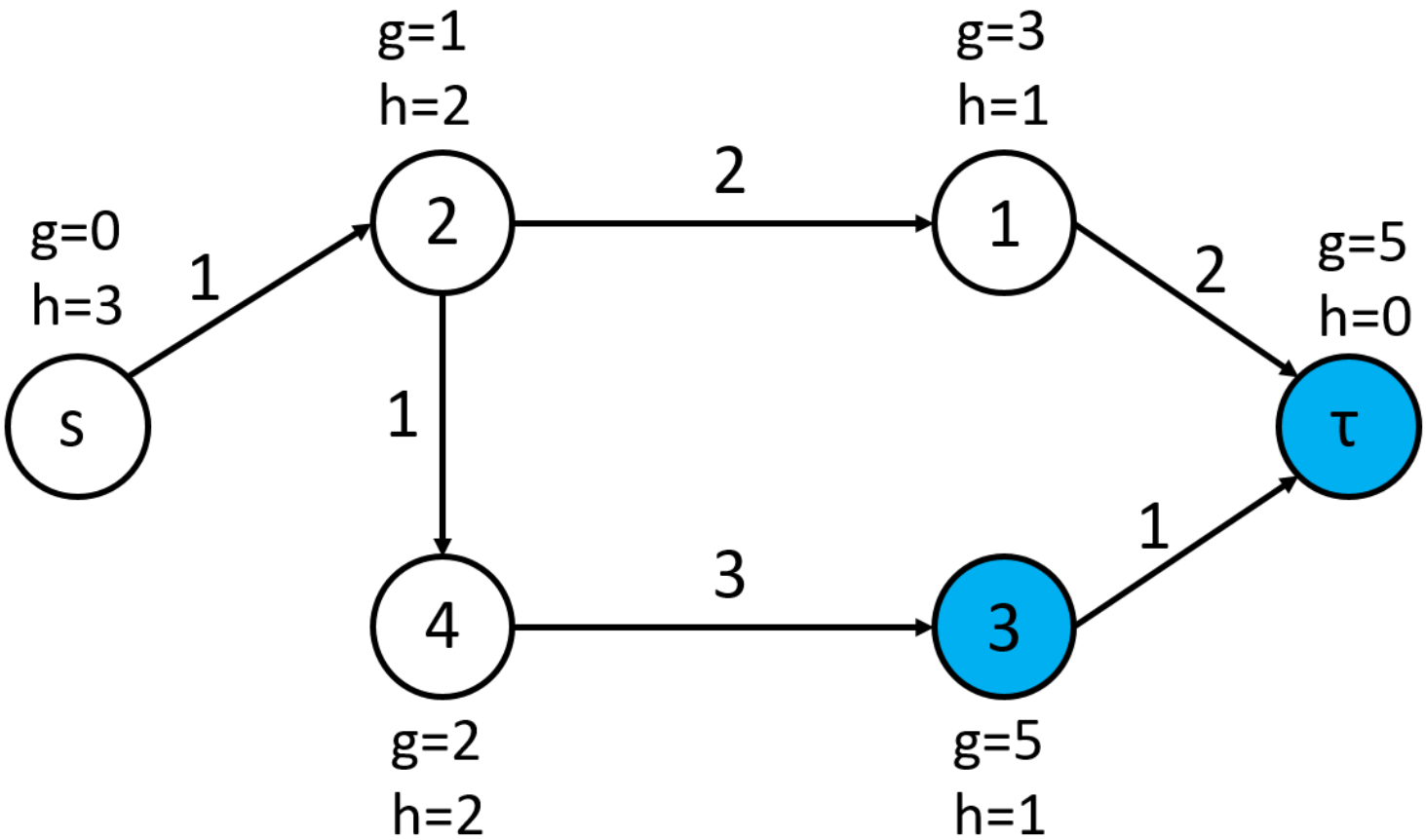
# Example: A* Algorithm

- $OPEN = \{4, \tau\}$
- $CLOSED = \{s, 2, 1\}$
- Next to expand: 4



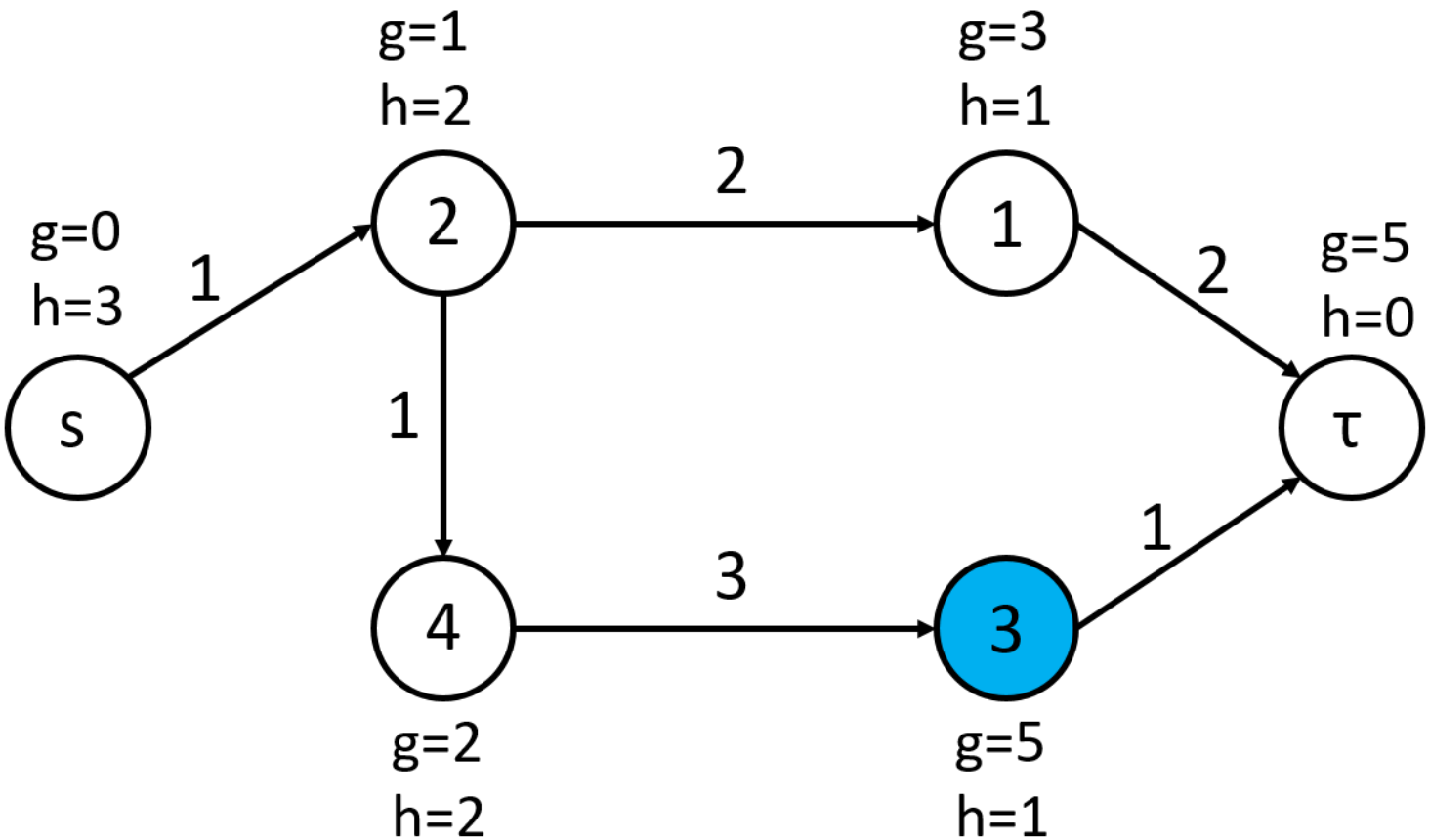
# Example: A* Algorithm

- $OPEN = \{3, \tau\}$
- $CLOSED = \{s, 2, 1, 4\}$
- Next to expand: $\tau$



# Example: A* Algorithm

- $OPEN = \{3\}$
- $CLOSED = \{s, 2, 1, 4, \tau\}$
- Done



# Theoretical Properties of A*

## Theorem: Finite Termination

A* terminates in a finite number of iterations if $\mathcal{V}$ is finite or if $c_{ij} \geq \delta > 0$ for $i, j \in \mathcal{V}$ and the degree of each node $i \in \mathcal{V}$ is finite.

## Lemma: Consistent Heuristic Implies Correct Labels

If $c_{ij} \geq 0$ for $i, j \in \mathcal{V}$ and A* uses a consistent heuristic, then:

- $g_i$ equals the least-cost from $s$ to $i$ for every expanded state $i \in CLOSED$
- $g_i$ is an upper bound on the least-cost from $s$ to $i$ for every $i \notin CLOSED$

# Proof of Lemma

- Proceed by induction:
  1. Assume all previously expanded states (in CLOSED) have correct $g$-values
  2. Let the next state to expand be $i$ with $f_i := g_i + h_i \leq f_j$ for all $j \in \mathit{OPEN}$
  3. Suppose that $g_i$ is incorrect, i.e., $g_i > \mathbf{dist}(s, i)$
  4. Then, there must exist at least one state $j$ on an optimal path from $s$ to $i$ such that $j \in \mathit{OPEN}$ but $j \notin \mathit{CLOSED}$ so that $f_j \geq f_i$
  5. Let $j$ be the shallowest OPEN node on the optimal path from $s$ to $i$, i.e., $\exists k \in \mathit{CLOSED}$ such that $g_j = g_k + c_{kj} = \mathbf{dist}(s, j)$
  6. However, this leads to a contradiction:

$$f_i = g_i + h_i > \mathbf{dist}(s, i) + h_i = g_j + \mathbf{dist}(j, i) + h_i \underset{\text{consistent}}{\overset{h \text{ is}}{\geq}} g_j + h_j = f_j$$

# Theoretical Properties of A*

## Theorem: Optimality

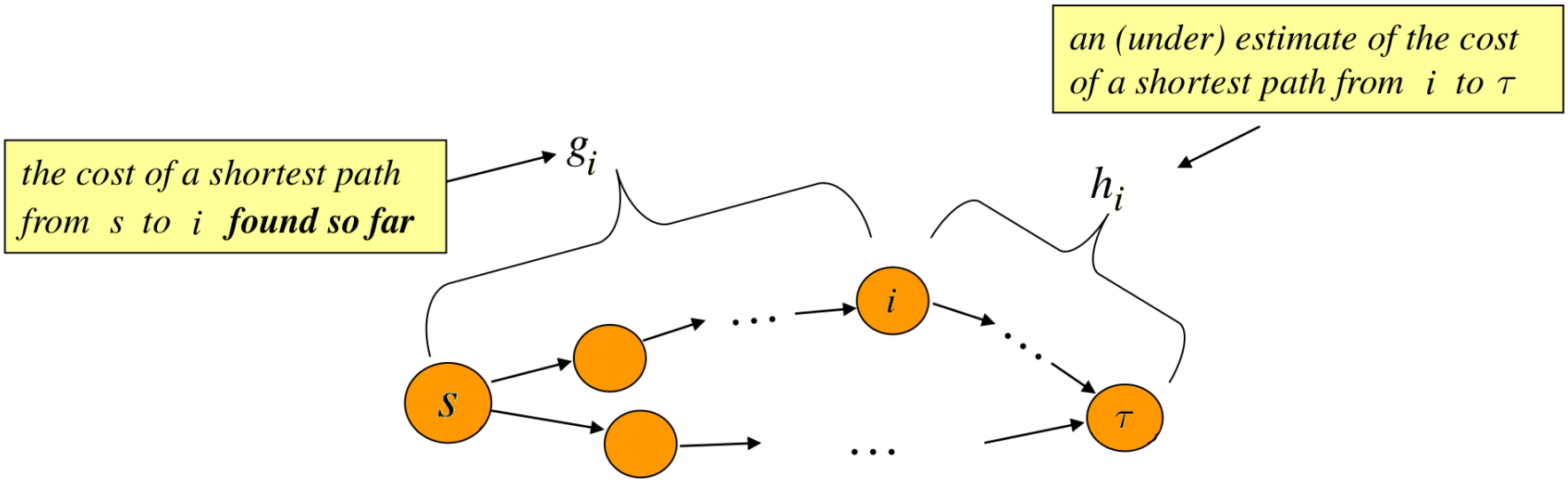
- If A* uses a consistent heuristic, then it is guaranteed to return an optimal path to $\tau$ (and, in fact, to every expanded node)
- If A* uses an admissible but inconsistent heuristic, then it is guaranteed to return an optimal path as long as closed states are re-opened (i.e., remove $j \notin$ CLOSED on Line 6 of the A* algorithm on Slide 16)
- If A* uses an $\epsilon$-consistent heuristic, then it is guaranteed to return an $\epsilon$-suboptimal path with cost $\textbf{dist}(s, \tau) \leq g_\tau \leq \epsilon \, \textbf{dist}(s, \tau)$ for $\epsilon \geq 1$

## Theorem: Efficiency

A* performs the minimal number of state expansions to guarantee optimality

# Effect of the Heuristic Function

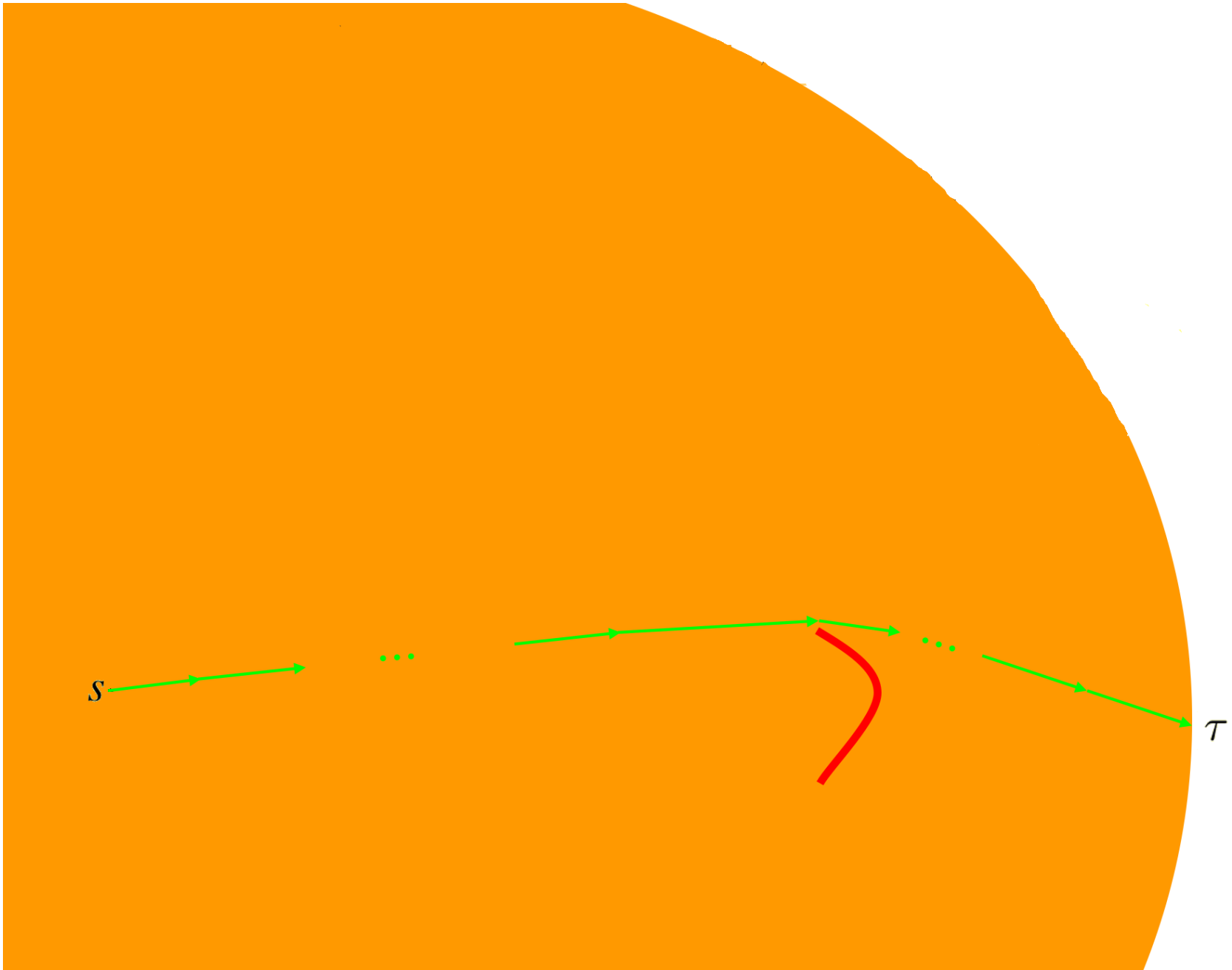
- $f_i$ is an estimate of the cost of a least cost path from $s$ to $\tau$ via $i$




- **Dijkstra**: expands states in the order of $f_i = g_i$
- **A***: expands states in the order of $f_i = g_i + h_i$
  - all nodes with $f_i < \textbf{dist}(s, \tau)$ are expanded
  - some nodes with $f_i = \textbf{dist}(s, \tau)$ are expanded
  - no nodes with $f_i > \textbf{dist}(s, \tau)$ are expanded
- **Weighted A***: expands states in the order of $f_i = g_i + \epsilon h_i$ with $\epsilon > 1$, i.e., biased towards states closer to the goal

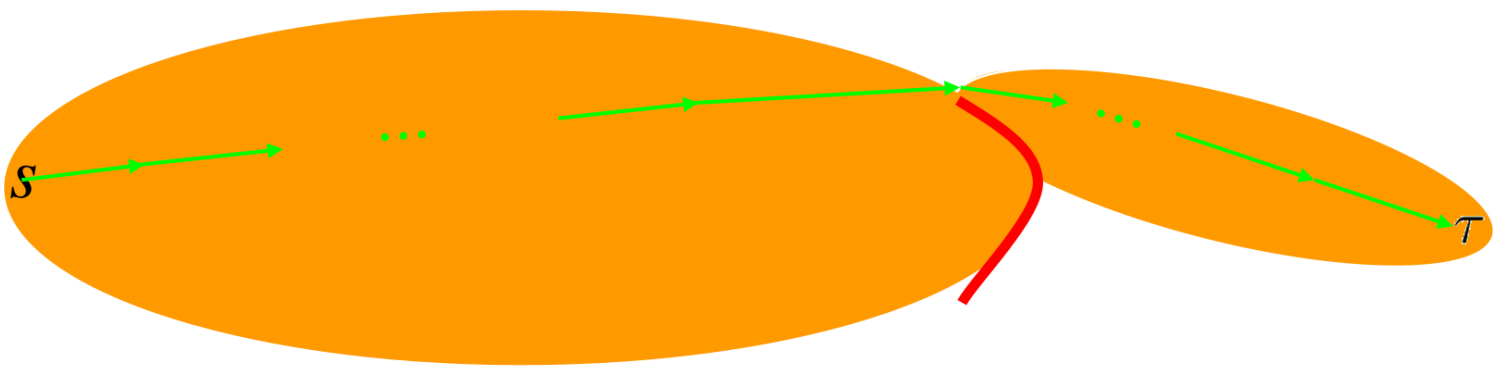
# Effect of the Heuristic Function: Dijkstra

- **Dijkstra**: expands states in the order of $f_i = g_i$



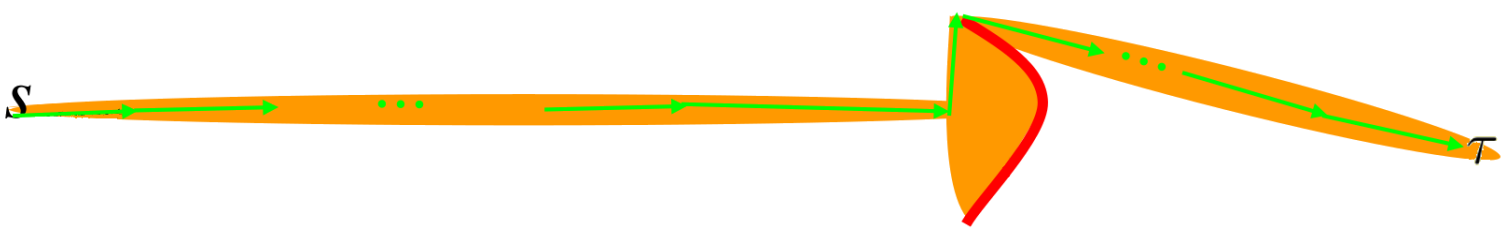
# Effect of the Heuristic Function: A*

- **A***: expands states in the order of $f_i = g_i + h_i$
- The closer $h_i$ is to **dist**$(i, \tau)$, the fewer expansions needed (fast search)
- The closer $h_i$ is to 0, the more expansions needed (slow search)



# Effect of the Heuristic Function: Weighted A*

- **Weighted A***: expands states in the order of $f_i = g_i + \epsilon h_i$ with $\epsilon > 1$, i.e., biased towards states closer to the goal
- Weighted A* is $\epsilon$-suboptimal ($g_\tau \leq \epsilon$ **dist**$(s, \tau)$) but trades optimality for speed. It is orders of magnitude faster than A* in many domains.
- The key to finding solutions fast is to have a heuristic function with shallow local minima!
- Is weighted A* guaranteed to expand no more states than A*?



# Implementation Details

- **Graph**: a **hashmap** data structure (stores key-value pairs) that maps node $i$ to its properties: label $g_i$, heuristic $h_i$, parent, etc.
  - e.g., std::unordered_map in C++ or dictionary in Python
- **Depth-first search**: last-in, first-out (LIFO): *OPEN* is a **stack**
  - e.g., std::stack in C++ or collections.deque in Python
- **Breath-first search**: first-in, first-out (FIFO): *OPEN* is a **queue**
  - e.g., std::queue in C++ or collections.deque in Python
- **Dijkstra and A* search**: *OPEN* is a **priority queue** based on $f_i$
  - e.g., boost::heap::d_ary_heap in C++ or pqdict in Python

# Time Complexity

- **Graph**: number of nodes $|\mathcal{V}|$, number of edges $|\mathcal{E}|$, maximum node degree $\Delta$ (number of outgoing edges)
- **Dynamic Programming**: $O(|\mathcal{V}|^3)$:
  - $|\mathcal{V}| \times |\mathcal{V}|$ entries in the table
  - Each entry requires $\Delta$ comparisons and in the worst case $\Delta = O(|\mathcal{V}|)$
- **Dijkstra and A***: $O(\mathit{makequeue} + \mathit{pop} \times |\mathcal{V}| + \mathit{update} \times |\mathcal{E}|)$
  - Array and make_heap, e.g., std::priority_queue in C++:
    $O(|\mathcal{V}|) + O(|\mathcal{V}|)|\mathcal{V}| + O(1)|\mathcal{E}| = O(|\mathcal{V}|^2)$
  - Binary heap, e.g., boost::heap::d_ary_heap in C++:
    $O(|\mathcal{V}|) + O(\log|\mathcal{V}|)|\mathcal{V}| + O(\log|\mathcal{V}|)|\mathcal{E}| = O((|\mathcal{E}| + |\mathcal{V}|)\log|\mathcal{V}|)$
  - Fibonacci heap, e.g., boost::heap::fibonacci_heap in C++:
    $O(|\mathcal{V}|) + O(\log|\mathcal{V}|)|\mathcal{V}| + O(1)|\mathcal{E}| = O(|\mathcal{E}| + |\mathcal{V}|\log|\mathcal{V}|)$

| | **Sparse graph**: $\lvert\mathcal{E}\rvert = O(\lvert\mathcal{V}\rvert)$ | **Dense graph**: $\lvert\mathcal{E}\rvert = O(\lvert\mathcal{V}\rvert^2)$ |
|---|---|---|
| Array | $O(\lvert\mathcal{V}\rvert^2)$ | $O(\lvert\mathcal{V}\rvert^2)$ |
| Binary heap | $O(\lvert\mathcal{V}\rvert \log \lvert\mathcal{V}\rvert)$ | $O(\lvert\mathcal{V}\rvert^2 \log \lvert\mathcal{V}\rvert)$ |
| Fibonacci heap | $O(\lvert\mathcal{V}\rvert \log \lvert\mathcal{V}\rvert)$ | $O(\lvert\mathcal{V}\rvert^2)$ |

# Memory Complexity

- A* does provably minimum number of expansions, $O(|\mathcal{V}|)$, to find the optimal solution but this might require an infeasible amount of memory
- The memory requirements of weighted A* are often but not always better
- **Depth-first search** (without marking expanded states): explore one possible path at a time and keep only the best path discovered so far in memory:
  - Complete and optimal (assuming a finite graph)
  - Memory: $O(\Delta m)$, where $\Delta$ - max branching factor, $m$ - max pathlength
  - Time: $O(\Delta^m)$, since it will repeatedly re-expand states
- Example: 4-connected 40 by 40 grid with $s$ at the center of the grid
  - A* expands up to 800 states
  - Depth-first search may expand over $4^{20} > 10^{12}$ states

# A* Implementation: Node Class

```
1  class ANode(object):
2    def __init__(self, key, coordinates):
3      self.key = key
4      self.coordinates = coordinates
5      self.g = math.inf
6      self.h = 0.0
7      self.parent = None
8      self.parent_action = None
9      self.is_open = False
10     sefl.is_closed = False
11
12   def __lt__(self, other):
13     return self.g < other.g
```

# A* Implementation: Environment Class

```
1 class Environment:
2   def isGoal(self, node):
3     return True
4
5   def getSuccessors(self, node):
6       return successor_list, cost_list, action_list
7
8   def getHeuristic(self, node):
9       return 0.0
```

# A* Implementation

```
from pqdict import pqdict
def aStar(start_coordinates, Env, epsilon = 1.0):
  current = ANode(tuple(start_coordinates), start_coordinates)
  current.g = 0.0
  current.h = Env.getHeuristic(current)

  Graph[current.key] = current
  OPEN = pqdict()

  while True:
    if Env.isGoal(current.coordinates):
      return recoverPath(current, Env)

    current.is_closed = True
    updateData(current, Graph, OPEN, Env, epsilon)

    if not OPEN:
      return # If OPEN is empty, no path is found

    # remove the element with smallest f value
    current = OPEN.popitem()[1][1]
```

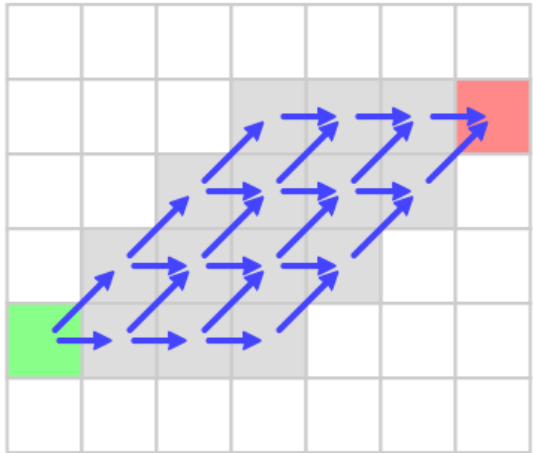
# A* Implementation

```
def updateData(current, Graph, OPEN, Env):
  successor_list, cost_list, action_list = Env.getSuccessors(current)
  for s_coord, s_cost, s_action in zip(successor_list, cost_list, action_list):
    s_key = tuple(s_coord)
    if s_key not in Graph:
      Graph[s_key] = ANode(s_key, s_coord)
      Graph[s_key].h = Env.getHeuristic(s_coord)
    child = Graph[s_key]

    tentative_g = current.g + s_cost
    if( tentative_g < child.g ):
      child.parent, child.parent_action = current, s_action
      child.g = tentative_g  # Correct label

      fval = tentative_g + epsilon*child.h
      if child.is_open: # if OPEN, update priority
        OPEN[s_key] = (fval, child)
        OPEN.heapify(s_key)
      elif child.is_closed and reopen_nodes: # if CLOSED, consider reopening
        OPEN[s_key] = (fval, child)
        child.is_open, child.is_closed = True, False
      else:  # new node, add to heap
        OPEN[s_key] = (fval, child)
        child.is_open = True
```

# Outline

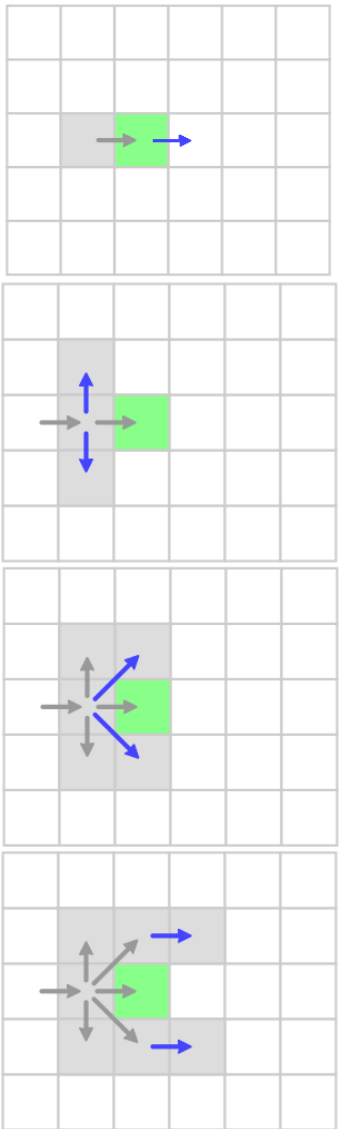
# Jump Point Search

- In a large open space, there are many equal length shortest paths
- A* adds a node's immediate neighbors to the OPEN priority queue, only to pop them soon after
- What if we could look ahead and skip nodes that are not valuable, e.g., lead to symmetric paths?



- **Assumption**: undirected uniform-cost grid, i.e., the same move costs the same amount in every node $i$
- 2-D case:
  - each node has $\leq 8$ neighbors
  - straight moves cost 1
  - diagonal moves cost $\sqrt{2}$

# Straight Moves



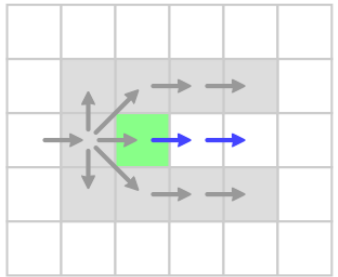
Consider horizontal/vertical movement from node $i$. We can ignore the node we are coming from (parent $p(i)$) since we already visited it

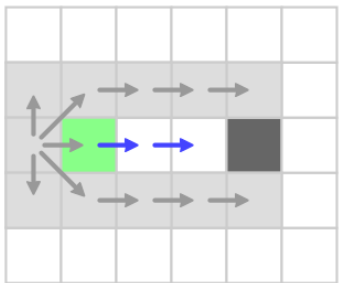
We can assume the two nodes diagonally behind us have been reached via $p(i)$ since those are shorter paths than going through $i$

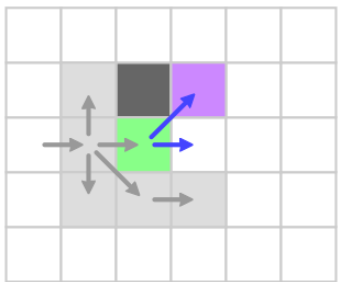
We can assume that the nodes above and below have also been reached via diagonal moves from $p(i)$, which cost $\sqrt{2}$ rather than going through $i$ for a cost of 2

The nodes diagonally in front of us can be reached via the neighbors above and below

# Forced Neighbors for Straight Moves





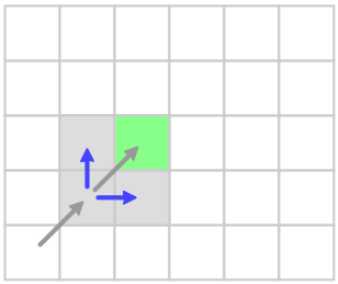


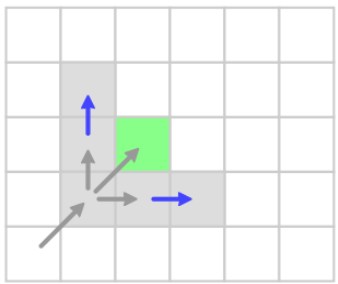
This leaves only a single **natural neighbor** to consider and that is the main idea – as long as the way is clear we can **jump** ahead to the right without adding any nodes to OPEN.

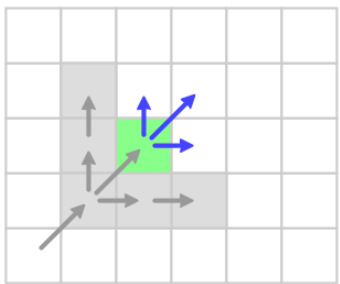
If the way is blocked as we jump to the right, we can safely disregard the entire jump because the paths above and below will be handled via other nodes.

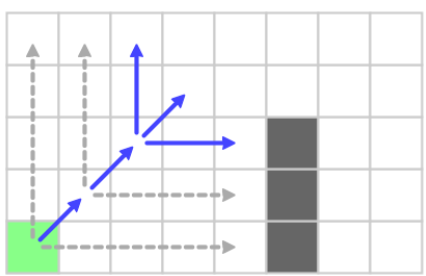
But what happens if one of these neighbors that we assume will cover other paths is blocked? We are **forced** to consider the node that would have otherwise been considered by the blocked path. Such a neighbor is called a **forced neighbor**. When we reach a node with a forced neighbor, we stop jumping right and add the node to the OPEN list for further examination.

# Diagonal Moves









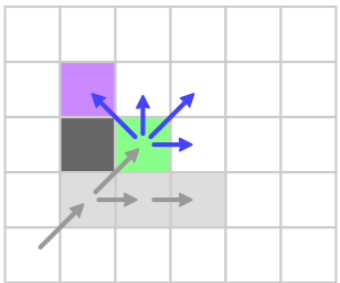
Consider diagonal movement from node $i$. As before, we can ignore the parent $p(i)$ since we already visited it. We can also ignore the left and below neighbors since they can be reached optimally from $p(i)$ via a straight move.

The nodes up and to the left and down and to the right can also be reached more optimally via the neighbors to the left and below.

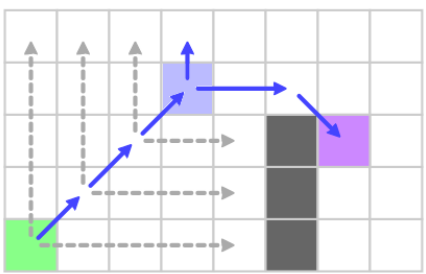
This leaves three **natural neighbors**: two above and to the right, and one diagonally in the original direction of travel.

Two of the natural neighbors require straight moves and since we already know how to jump straight we can look there first for forced neighbors. If neither finds any, we move one more step diagonally and repeat.

# Forced Neighbors for Diagonal Moves



Simiar to forced neighbors during straight movement, when an obstacle is present to our left or below, then the neighbors diagonally up-and-left and down-and-right cannot be reached in any other way but through $i$. These are **forced neighbors** for diagonal moves. When we reach a node with a forced neighbor, we stop jumping diagonally and add the node to OPEN for further examination.



The straight-line jumps initiated from the two natural neighbors might also reach a forced neighbor. In that case, we also need to add the current node $i$ to the OPEN set and continue with the next A* iteration.

# Formal Definitions

- Let $i$ be the current node under evaluation and $p(i)$ be its parent
- **Natural neighbor**: a node $j \in \text{Neib}(i)$ is a natural neighbor if
  - (**Straight Move**): $c_{p(i),i} + c_{i,j} < c_{p(i),k} + c_{k,j}$ for all $k \in \text{Neib}(i)$, including $k = j$ in which case $c_{j,j} = 0$. In other words, $j$ is a natural neighbor of $i$ if **the** shortest path from $p(i)$ to $j$ has to go through $i$.
  - (**Diagonal Move**): $c_{p(i),i} + c_{i,j} \leq c_{p(i),k} + c_{k,j}$ for all $k \in \text{Neib}(i)$, including $k = j$ in which case $c_{j,j} = 0$. In other words, $j$ is a natural neighbor of $i$ if **a** shortest path from $p(i)$ to $j$ has to go through $i$.
- **Forced neighbor**: a node $j \in \text{Neib}(i)$ is a forced neighbor if both:
  1. $j$ is not a natural neighbor of $i$
  2. $c_{p(i),k} + c_{k,j} > c_{p(i),i} + c_{i,j}$ for all $k \in \text{Neib}(i)$
- **Jump point**: node $j$ with coordinates $x_j$ is a **jump point** from node $i$ in direction $d$, if $x_j$ minimizes $\lambda \in \mathbb{N}$ such that $x_j = x_i + \lambda d$ and one of the following holds:
  1. Node $j$ is the goal node $\tau$
  2. Node $j$ has at least one forced neighbor
  3. $\|d\|_1 = 2$ (diagonal move) and $\exists k \in \mathcal{V}$ which lies $\lambda_i \in \mathbb{N}$ steps in a straight direction $d_i \in \{d_1, d_2\}$, i.e., $x_k = x_j + \lambda_i d_i$, and $k$ is a jump point from $j$ by condition 1 or 2 above.

# Putting It All Together

- We apply the A* algorithm as usual, except that when we are expanding a node $i$ from the OPEN list we:
  1. Look at its parent $p(i)$ to determine the direction of travel.
  2. Jump as far as possible (straight first, then diagonally), skipping intermediate nodes using the simplifying rules until we encounter a jump point $j$
  3. We treat $j$ as if it were an immediate child of $i$: try to decrease its $g$-value and then insert it into OPEN
- **Main takeaway**: accessing the contents of many points on a grid in a few iterations of A* is more efficient than maintaining a priority queue over many iterations of A*

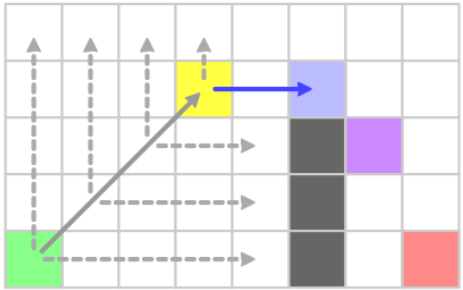
# 2-D Jump Point Search

**Algorithm** 2-D Jump Point Search

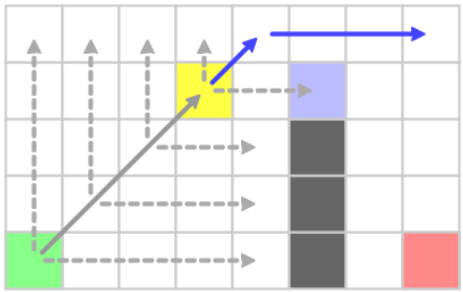
**function** GETSUCCESSORS($i, \tau$)
    Successors($i$) $\leftarrow \emptyset$ ▷ Keep all neighbors of the start node $s$
    Neighbors($i$) $\leftarrow$ PRUNE($i$,Neighbors($i$)) ▷ Keep only natural and forced neighbors
    **for** all $j \in$ Neighbors($i$) **do**
        $j \leftarrow$ JUMP($i$,direction($i$,$j$),$\tau$)
        add $j$ to Successors($i$)
    **return** Successors($i$)

**function** JUMP($i, d, \tau$)
    $j \leftarrow$ STEP($i, d$)
    **if** $j$ is an obstacle or outside the grid **then**
        **return** *null*
    **if** $j = \tau$ **or** $\exists k \in$ Neighbors($j$) such that $k$ is forced **then**
        **return** $j$
    **if** $\|d\|_1 = 2$ (diagonal) **then**
        **for** $k \in \{1, 2\}$ **do**
            **if** JUMP($j, d_k, \tau$) is **not** *null* **then**
                **return** $j$
    **return** JUMP($j, d, \tau$)

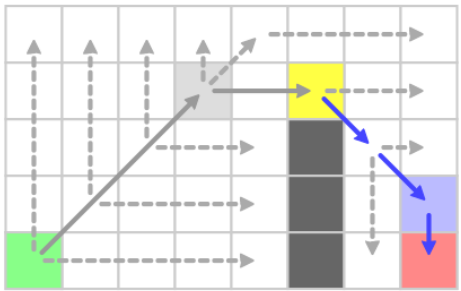
# Example: 2-D JPS



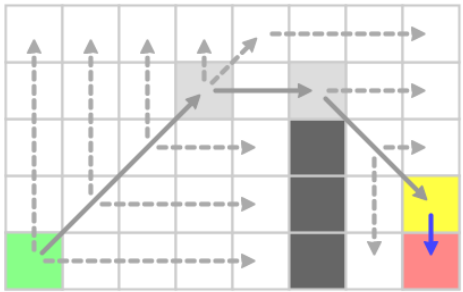
Starting from the green node in OPEN we jump horizontally, then vertically, then diagonally until a jump finds a node (blue) with a forced neighbor (purple). We add the yellow node to OPEN.



We expand the yellow node. Checking diagonally leads to the edge of the map so no new jump points are added. The jump point (blue) is added to the OPEN list.

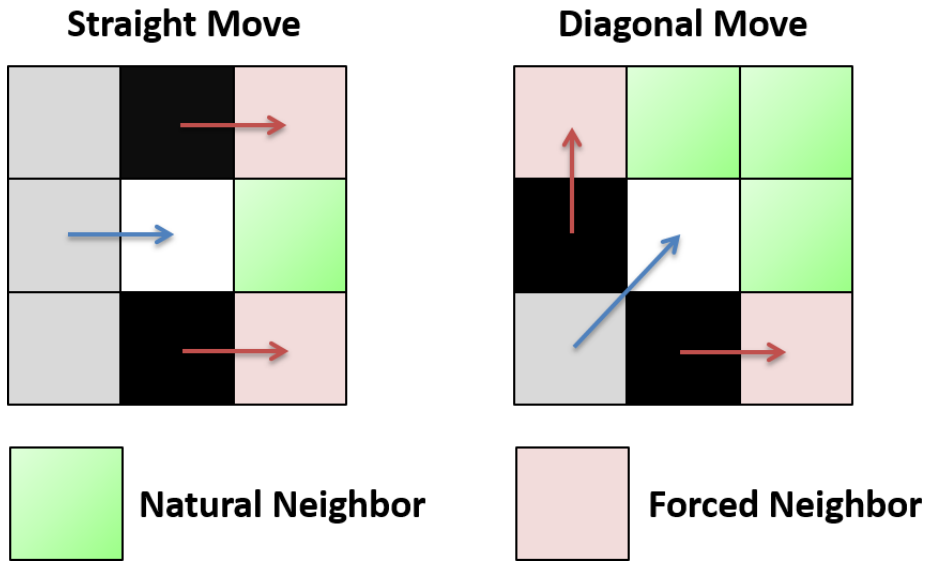


We expand the yellow node from OPEN. Since we were moving diagonally, we first explore the horizontal (leads to map edge) and vertical (blocked) directions and then jump diagonally.



We encounter a node with a forced neighbor (the goal) and add it to OPEN. Expanding this last node reaches the goal.

# 2-D JPS Pruning Rules and Optimality




**Theorem: Optimality of JPS (Harabor and Grastien, AAAI 2011)**

Jump point search in a 2-D undirected uniform-cost grid returns the cost of an optimal path from $s$ to $\tau$ if a feasible path exists and $\infty$ otherwise.


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

# 3-D JPS Pruning Rules