ECE276B: Planning & Learning in Robotics
Lecture 6: Search-based Planning

Lecturer:

Nikolay Atanasov: natanasov@ucsd.edu

Teaching Assistants:

Tianyu Wang: tiw161@eng.ucsd.edu

Yongxi Lu: yol070@eng.ucsd.edu

UC San Diego

JACOBS SCHOOL OF ENGINEERING

Electrical and Computer Engineering

Back to the Shortest Path Problem

- Once a graph is constructed (via cell decomposition, skeletonization, lattice, etc.), we need to search it for the least-cost path
- So far we saw that the shortest path problem can be solved via:
  - **Forward DP**: computes the shortest paths from *all* nodes to the goal
  - **Label correcting methods**: not necessary to visit every node of the graph
- **Key Ideas** of LC methods:
  - **Label** $\overset{g_i}{\not{d}_i}$: keeps the lowest cost from s to each visited node $i \in \mathcal{V}$
  - Each time $\overset{g_i}{\not{d}_i}$ is reduced, the labels $\overset{g_j}{\not{d}_j}$ of the **children** of i can be corrected: $\overset{g_j}{\not{d}_j} = \overset{g_i}{\not{d}_i} + c_{ij}$
  - **OPEN**: set of nodes that can potentially be part of the shortest path to τ

Label Correcting Algorithm

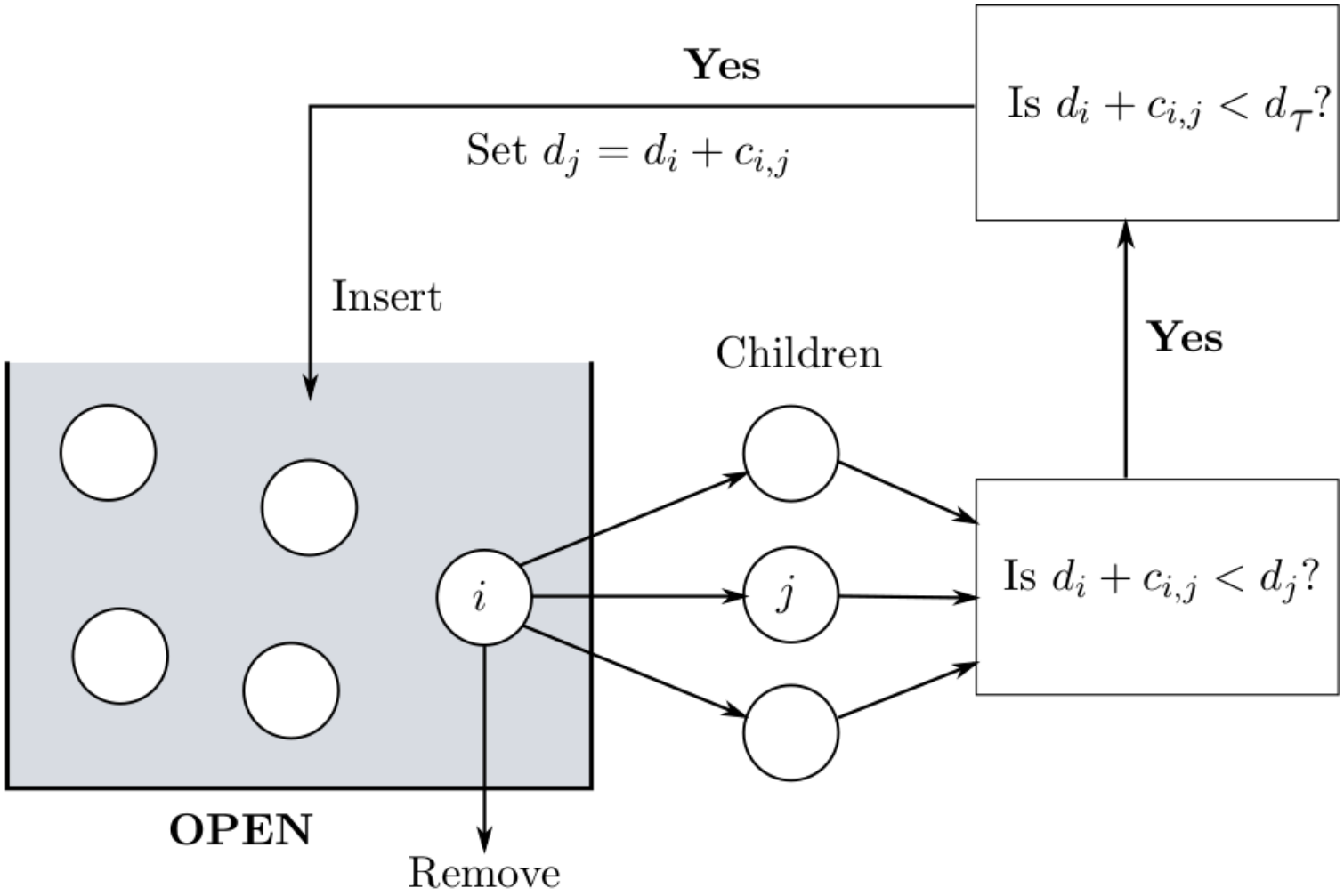
Algorithm 1 Label Correcting Algorithm

OPEN $\leftarrow \{s\}$, $g_s = 0$, $g_i = \infty$ for all $i \in \mathcal{V} \setminus \{s\}$
while OPEN is not empty **do**
    Remove i from OPEN
    for $j \in$ Children(i) **do**
        if $(g_i + c_{ij}) < g_j$ **and** $(g_i + c_{ij}) < g_\tau$ **then**
            $g_j \leftarrow (g_i + c_{ij})$
            Parent$(j) \leftarrow i$
            if $j \neq \tau$ **then**
                OPEN $\leftarrow$ OPEN $\cup \{j\}$

Theorem

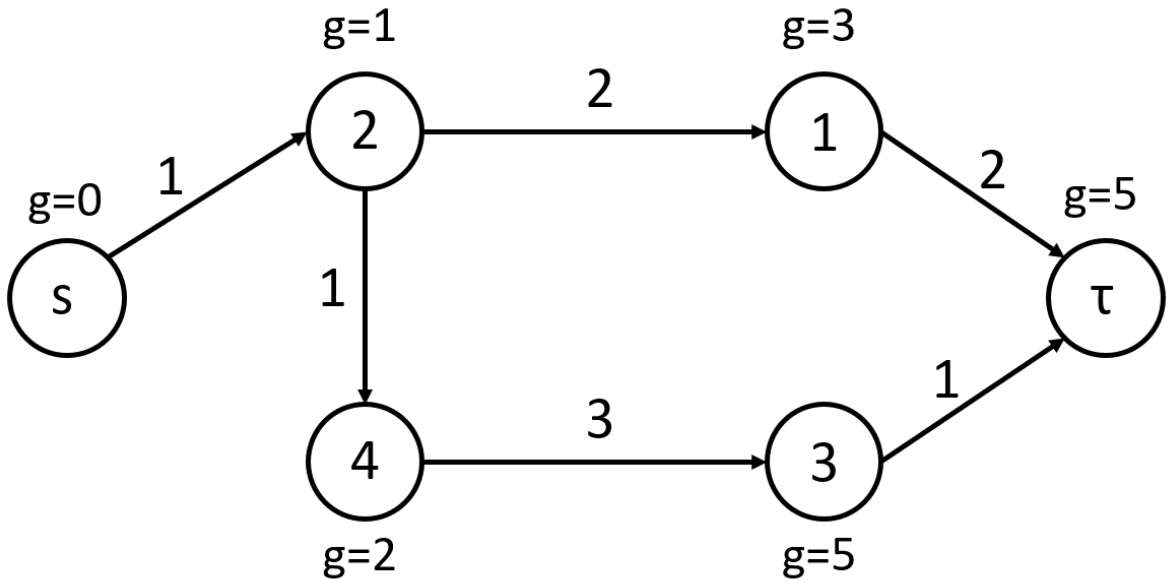
If there exists at least one finite cost path from s to τ, then the Label Correcting (LC) algorithm terminates with $g_\tau = J^{Q^*}$ (the shortest path from s to τ). Otherwise, the LC algorithm terminates with $g_\tau = \infty$.

Label Correcting Algorithm



Properties of LC Algorithms

- LC algorithms compute the optimal labels (g-values) for relevant states
- The optimal g-values satisfy: $\boxed{g_i = \min_{j \in Parents(i)} g_j + c_{j,i}}$



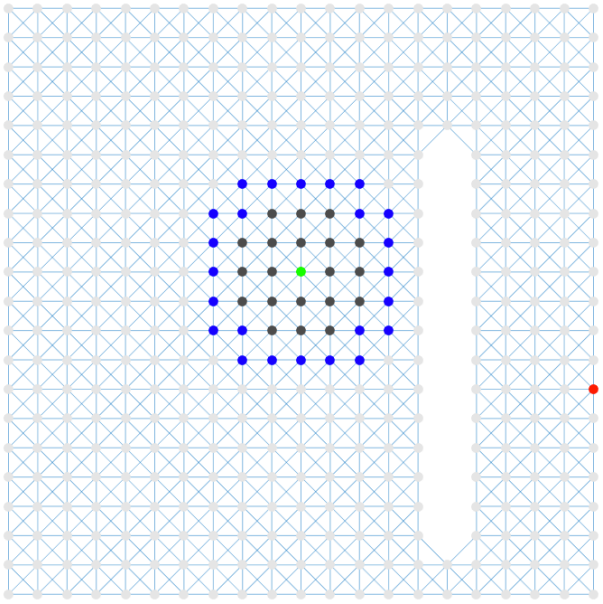
- Once the g-values are available, the least-cost path $Q^* = q_0, \ldots, q_T$ is a greedy path computed starting from $q_T = \tau$ and backtracking:

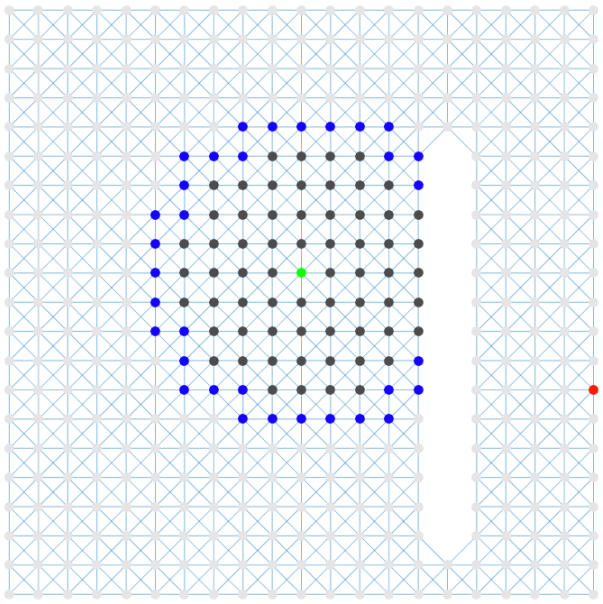
$$q_t = \arg\min_{j \in Parents(q_{t+1})} g_j + c_{j,q_{t+1}} \qquad \text{for } t = T-1, \ldots, 0$$

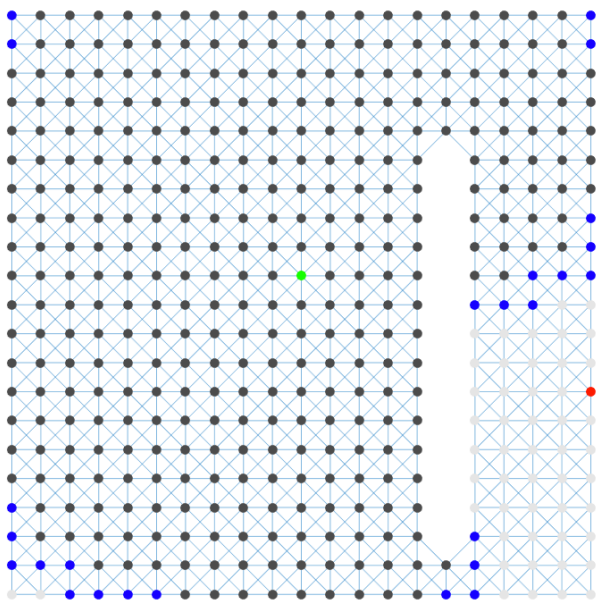
Dijkstra's Algorithm

- **Best-first search**: removes nodes with minimum label $i^* = \arg\min_{j \in OPEN} g_j$ from OPEN (implemented as a **priority queue**)
- If $i \in OPEN$, its label g_i may change as we discover new paths to i
- 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 τ is removed from OPEN, then we cannot discover a shorter path to τ: **Done**!
- 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 τ might be. Can this be estimated and used to improve the search?

OPEN is the Search Frontier







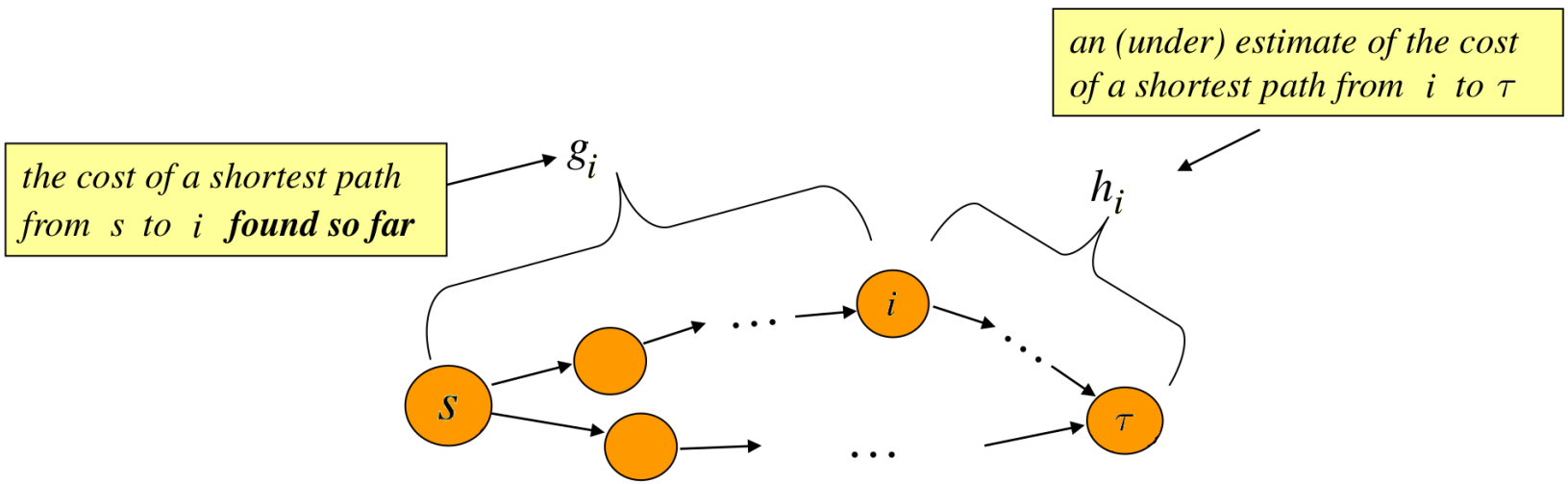
A* Algorithm

- The **A* algorithm** is a modification to the LC algorithm 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 positive lower bound on the optimal cost to get from node j to τ, known as **heuristic**.

- The more stringent criterion can reduce the number of iterations required by the LC algorithm
- The heuristic is constructed depending on special knowledge about the problem. The more accurately h_j estimates the optimal cost from j to τ, the more efficient the A* algorithm becomes!

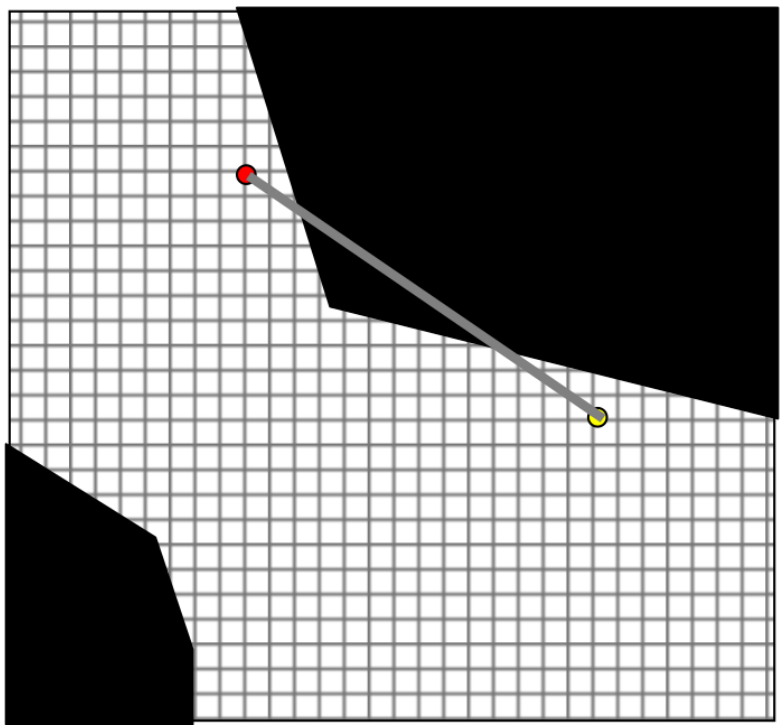


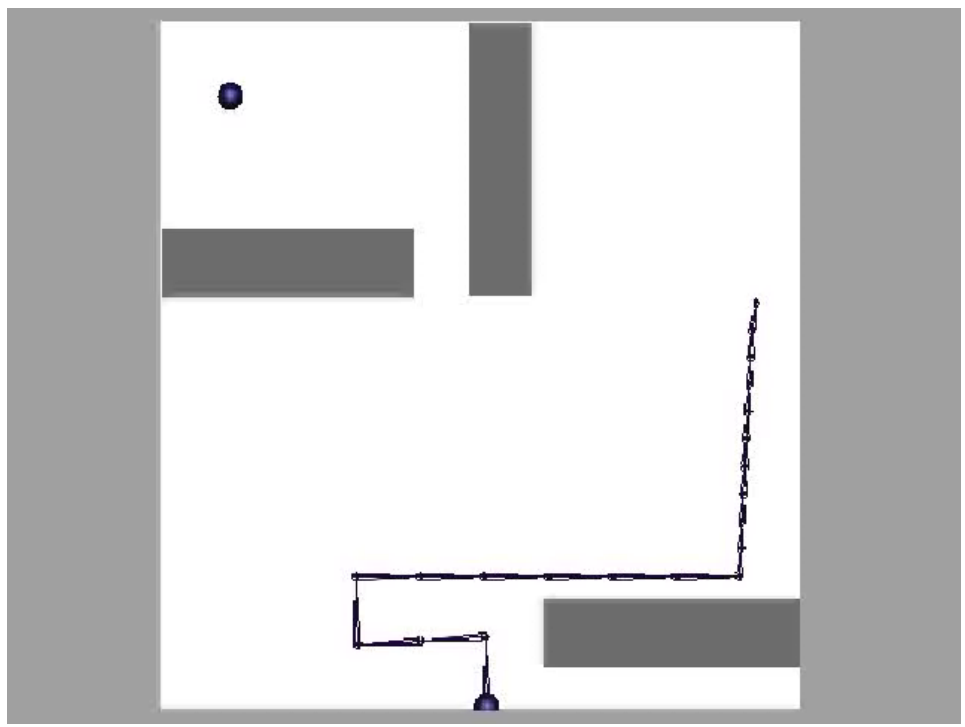
Heuristic Function

- **Admissible**: $h_i \leq J^*_{i,\tau}$ for all $i \in \mathcal{V}$
- **Consistent**: $h_\tau = 0$ and $h_i \leq c_{ij} + h_j$ for all $i \neq \tau$ and $j \in \text{Children}(i)$
  - i.e., h satisfies the **triangle inequality**, which also implies it is 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 ϵ-consistent ($\epsilon = 2$)
- ϵ-**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 $h^{(2)}_i \geq h^{(1)}_i$ for every node $i \in \mathcal{V}$
  - Extreme cases: $h_i \equiv 0$ and $h_i = J^*_{i,\tau}$

Examples of Heuristic Functions

- Grid-based planning: let $x_i \in \mathbb{R}^d$ be the position of node i
  - Euclidean distance: $h_i := \|x_\tau - x_i\|_2$
  - Manhattan distance: $h_i := \|x_\tau - x_i\|_1 := \sum_k |x_{\tau,k} - x_{i,k}|$
  - Diagonal distance: $h_i := \|x_\tau - x_i\|_\infty := \max_k |x_{\tau,k} - x_{i,k}|$
  - Octal distance: combines $\max_k |x_{\tau,k} - x_{i,k}|$ and $\min_k |x_{\tau,k} - x_{i,k}|$
- 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





A* Algorithm

Algorithm 2 Weighted A* Algorithm

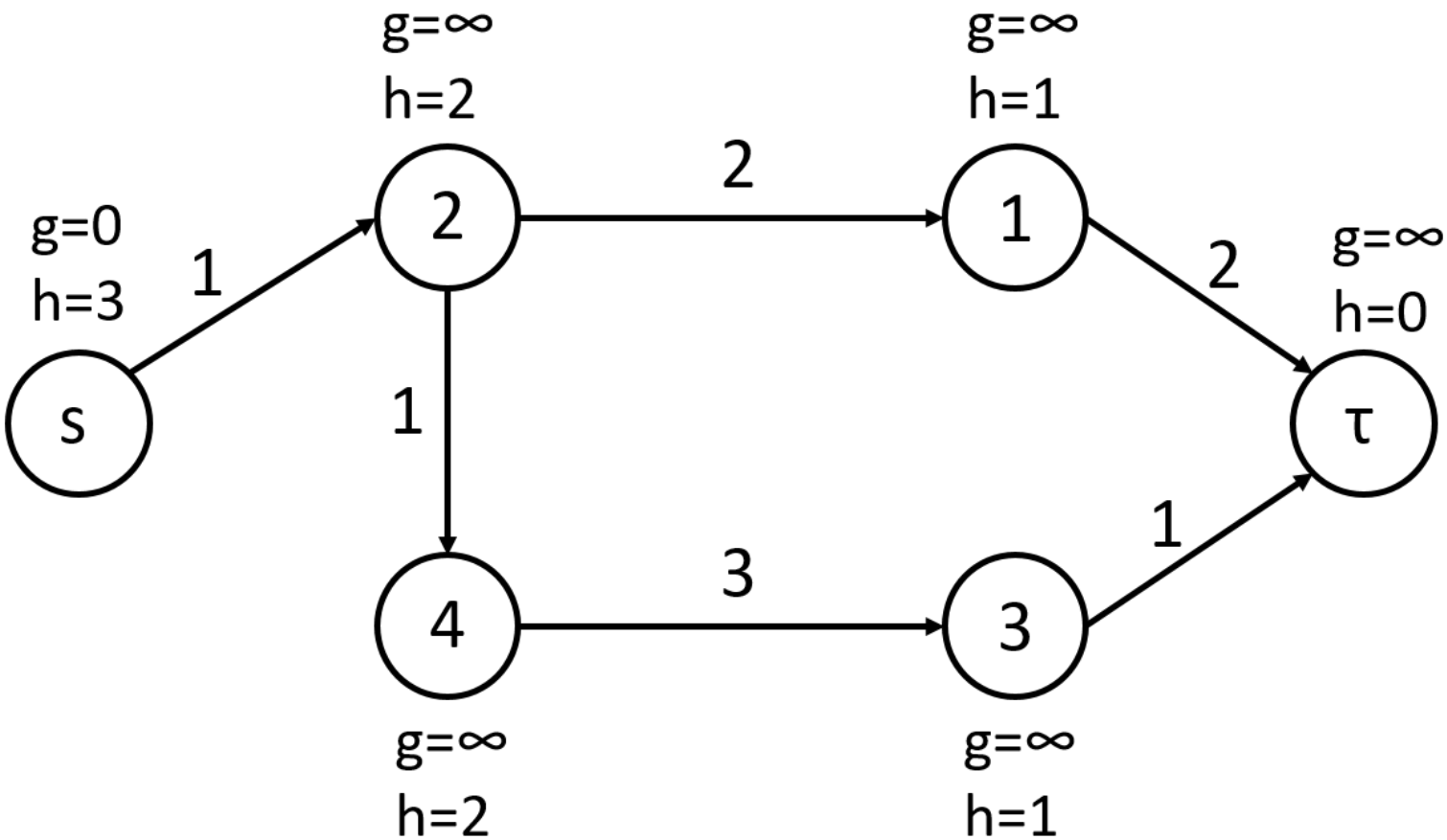
1: OPEN $\leftarrow \{s\}$, CLOSED $\leftarrow \{\}$, $\epsilon \geq 1$
2: $g_s = 0$, $g_i = \infty$ for all $i \in \mathcal{V} \setminus \{s\}$
3: **while** $\tau \notin$ CLOSED **do** ▷ τ not expanded yet
4: Remove i with smallest $f_i := g_i + \epsilon h_i$ from OPEN ▷ means $g_i + \epsilon h_i < g_\tau$
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: Insert j into OPEN

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

- **Weighted A***: A* with an ϵ-consistent heuristic
- 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

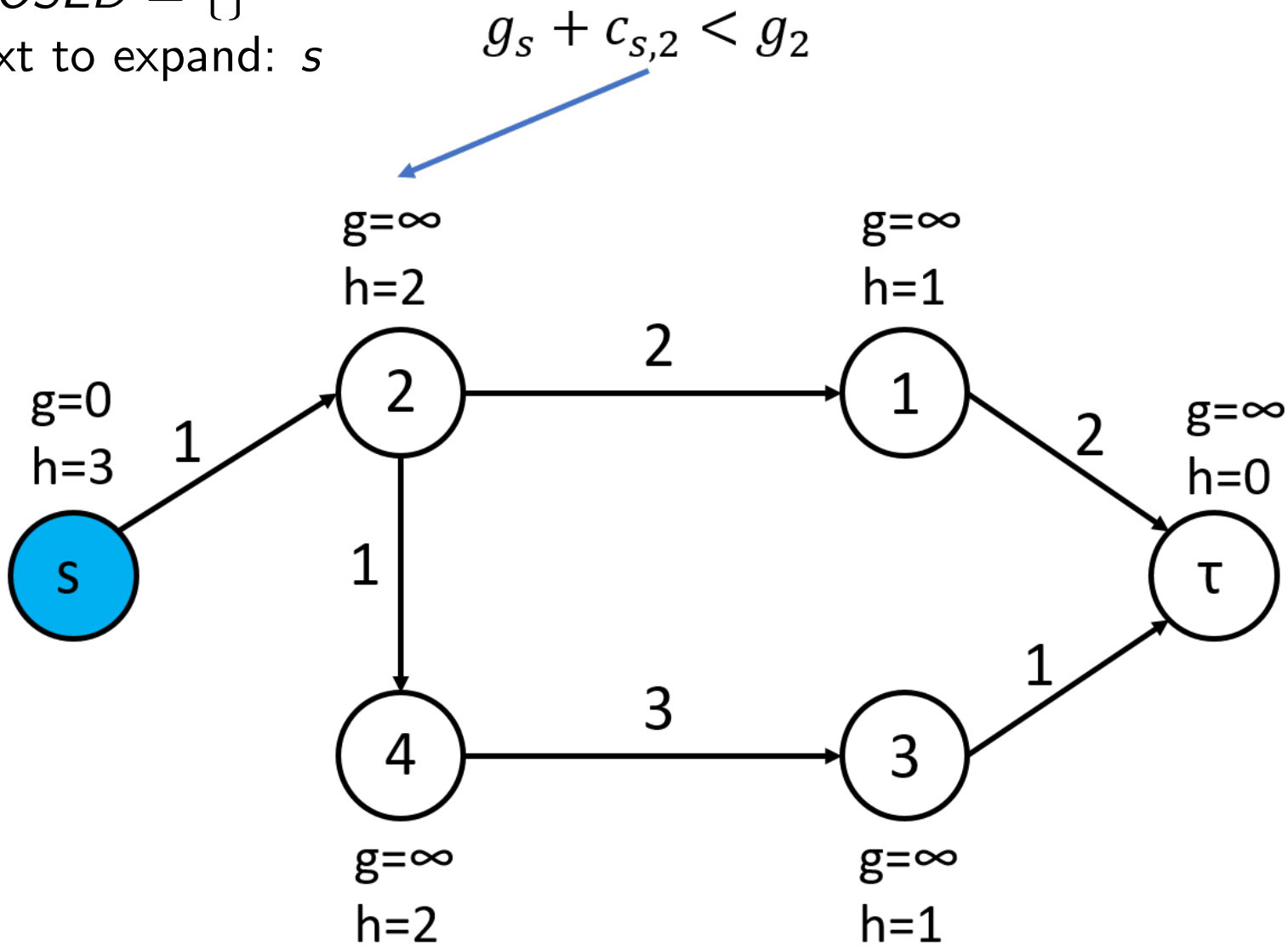
Example: A* Search

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



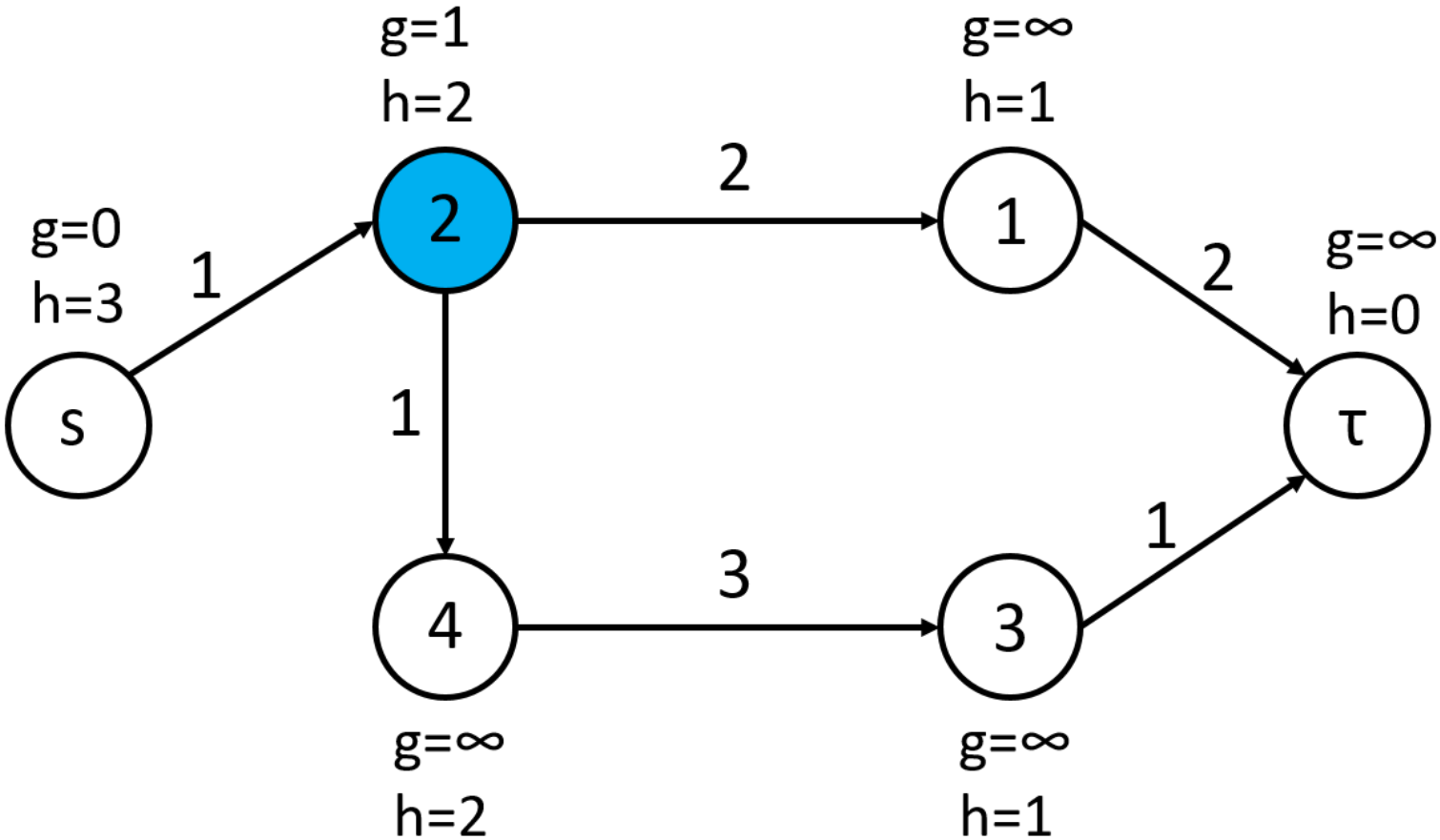
Example: A* Search

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



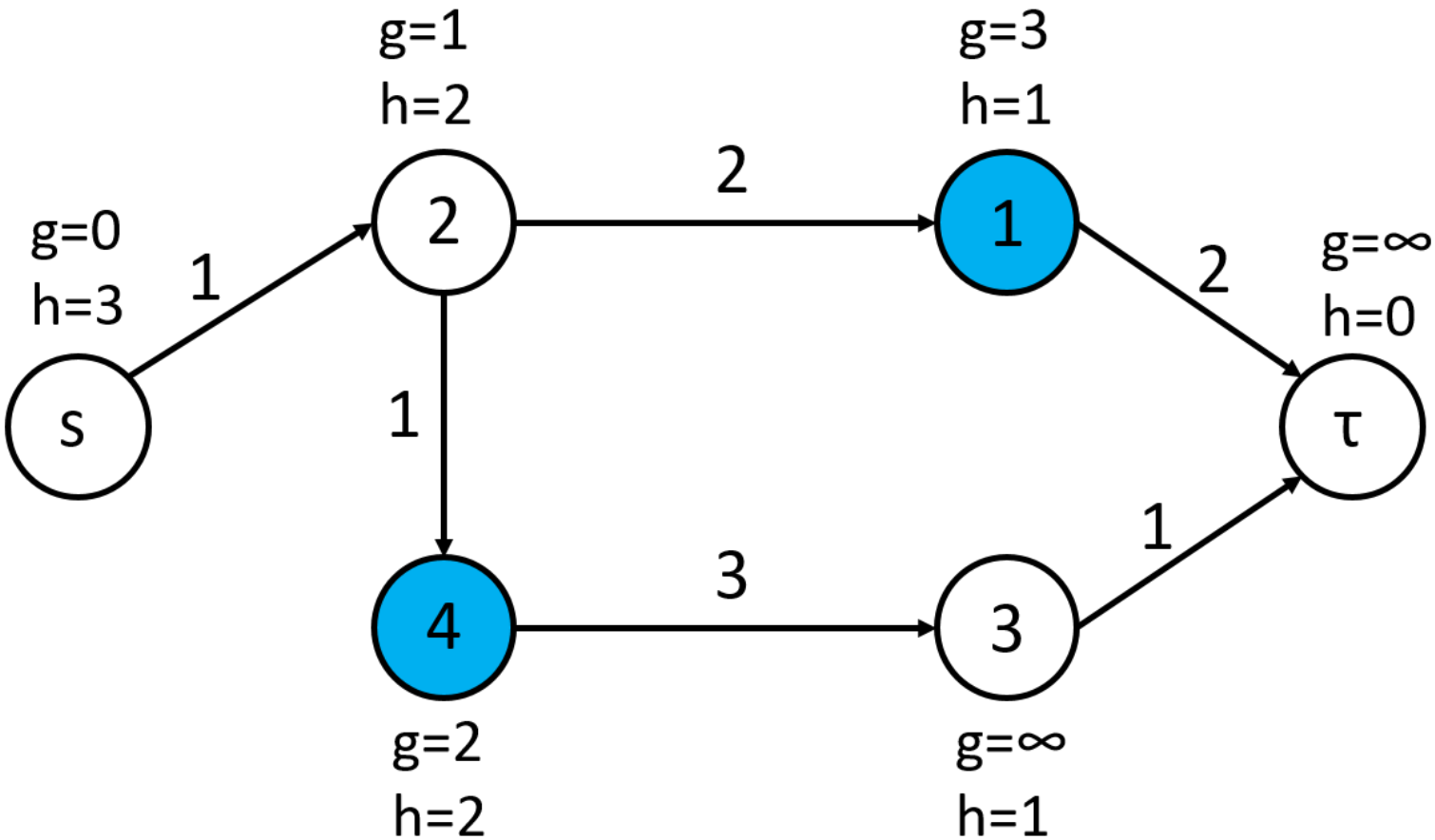
Example: A* Search

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



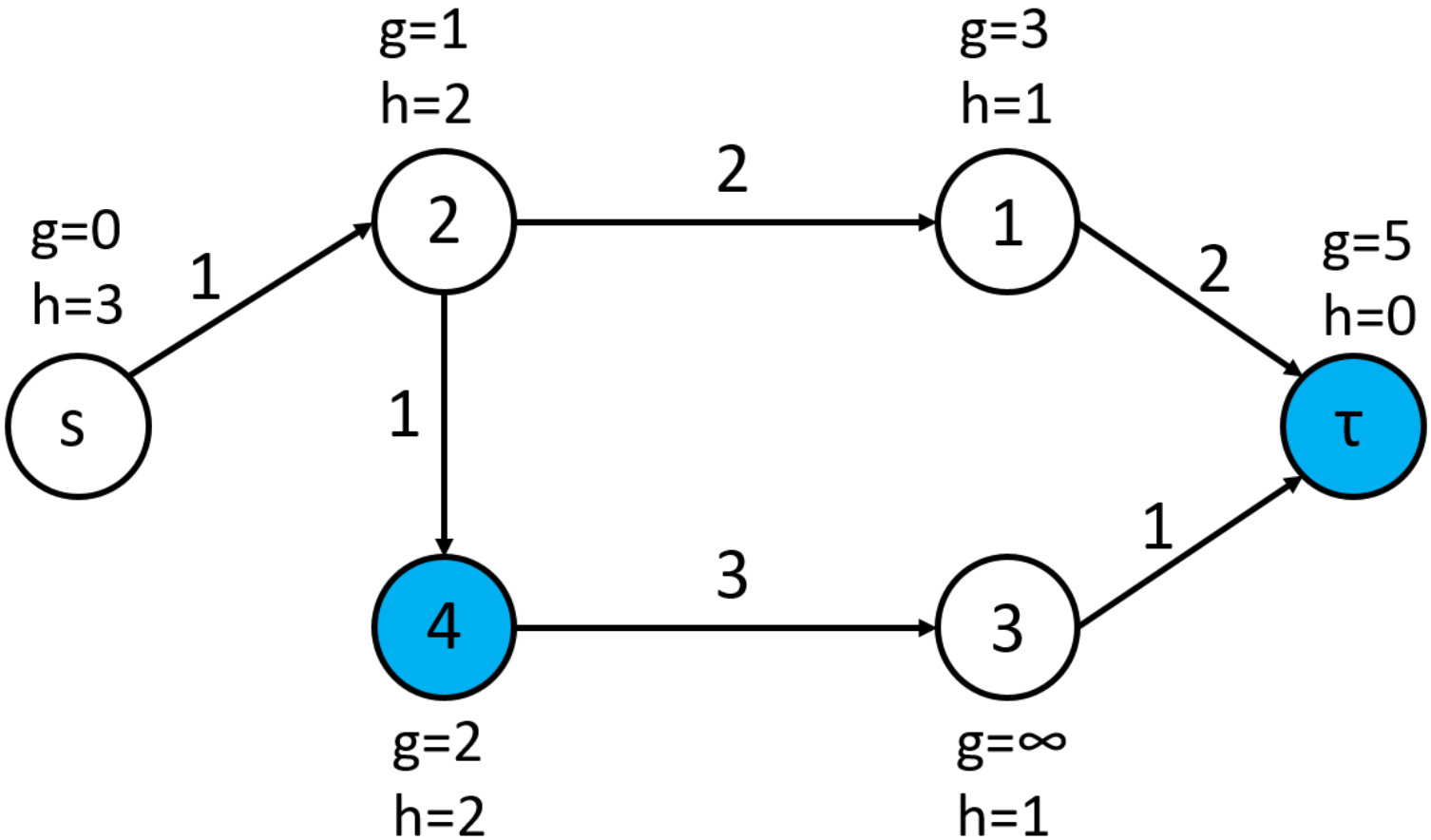
Example: A* Search

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



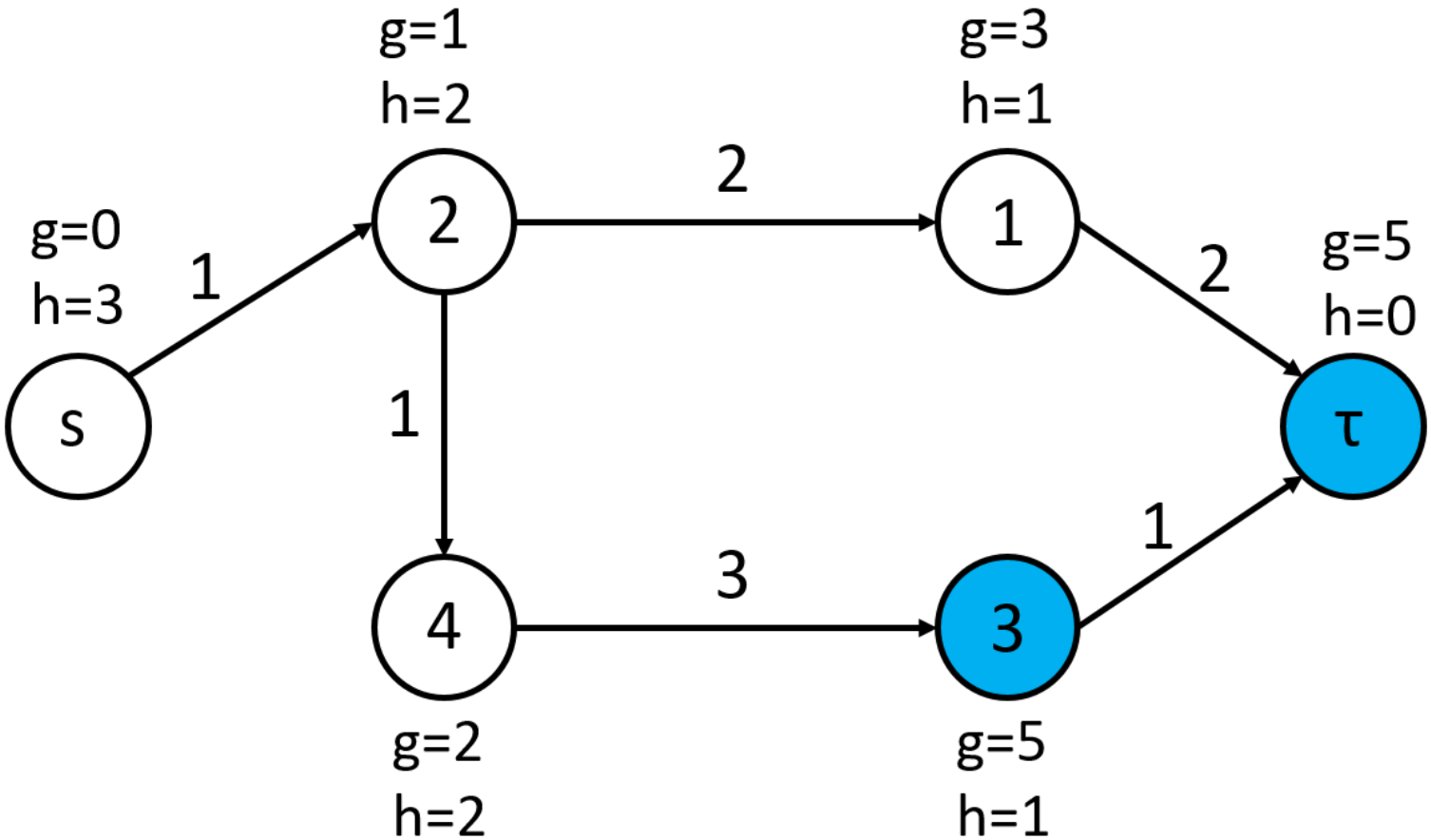
Example: A* Search

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



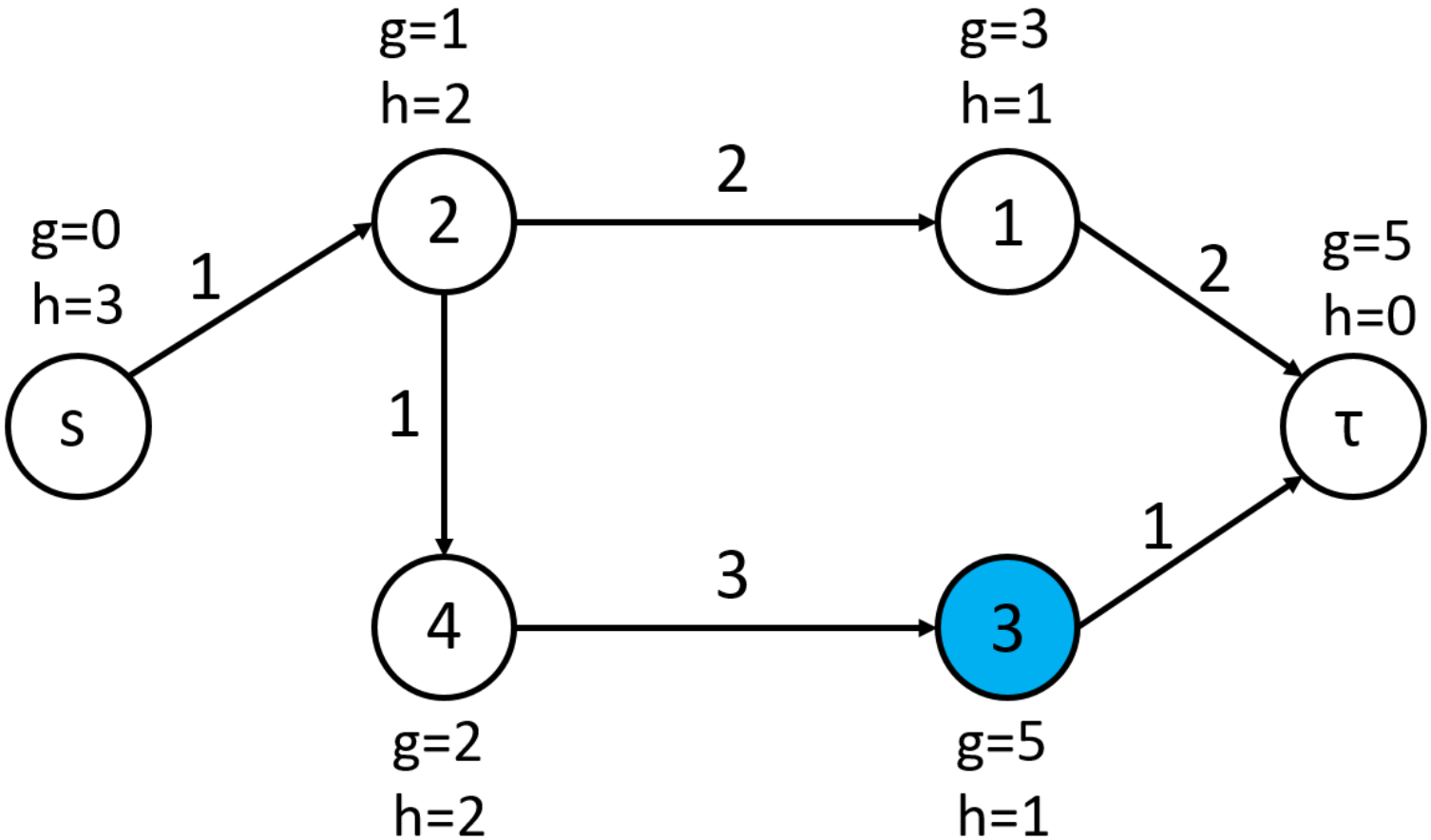
Example: A* Search

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



Example: A* Search

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



Theoretical Properties of A*

Lemma: Consistency Implies Correct Labels

If 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 $i \notin CLOSED$

Theorem: Optimality of A*

- If A* uses a consistent heuristic, then it is guaranteed to return an optimal path (in fact, for every expanded state)
- If A* uses an admissible (not consistent) heuristic, then it is guaranteed to return an optimal path as long as closed states are re-opened
- If A* uses an ϵ-consistent heuristic, then it is guaranteed to return an ϵ-suboptimal path with cost $J \leq \epsilon J^*$ for $\epsilon \geq 1$

Theorem: Efficiency of A*

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

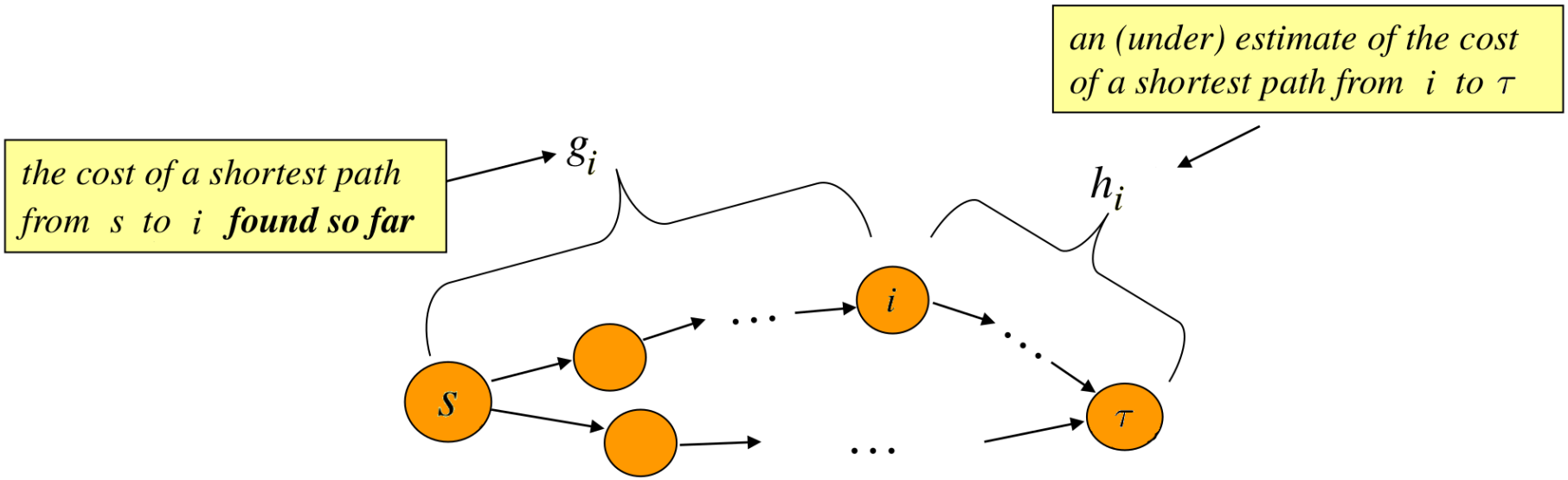
Proof of Lemma

- Let $c_{ij} > 0$ for $i, j \in \mathcal{V}$ and h be a consistent heuristic
- 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., larger than the least cost from s to 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. However, this leads to a contradiction:

$$f_i = g_i + h_i > J^*_{s,i} + h_i = g_j + J^*_{ji} + h_i \overset{\substack{h \text{ is} \\ \text{consistent}}}{\geq} g_j + h_j = f_j$$

Effect of the Heuristic

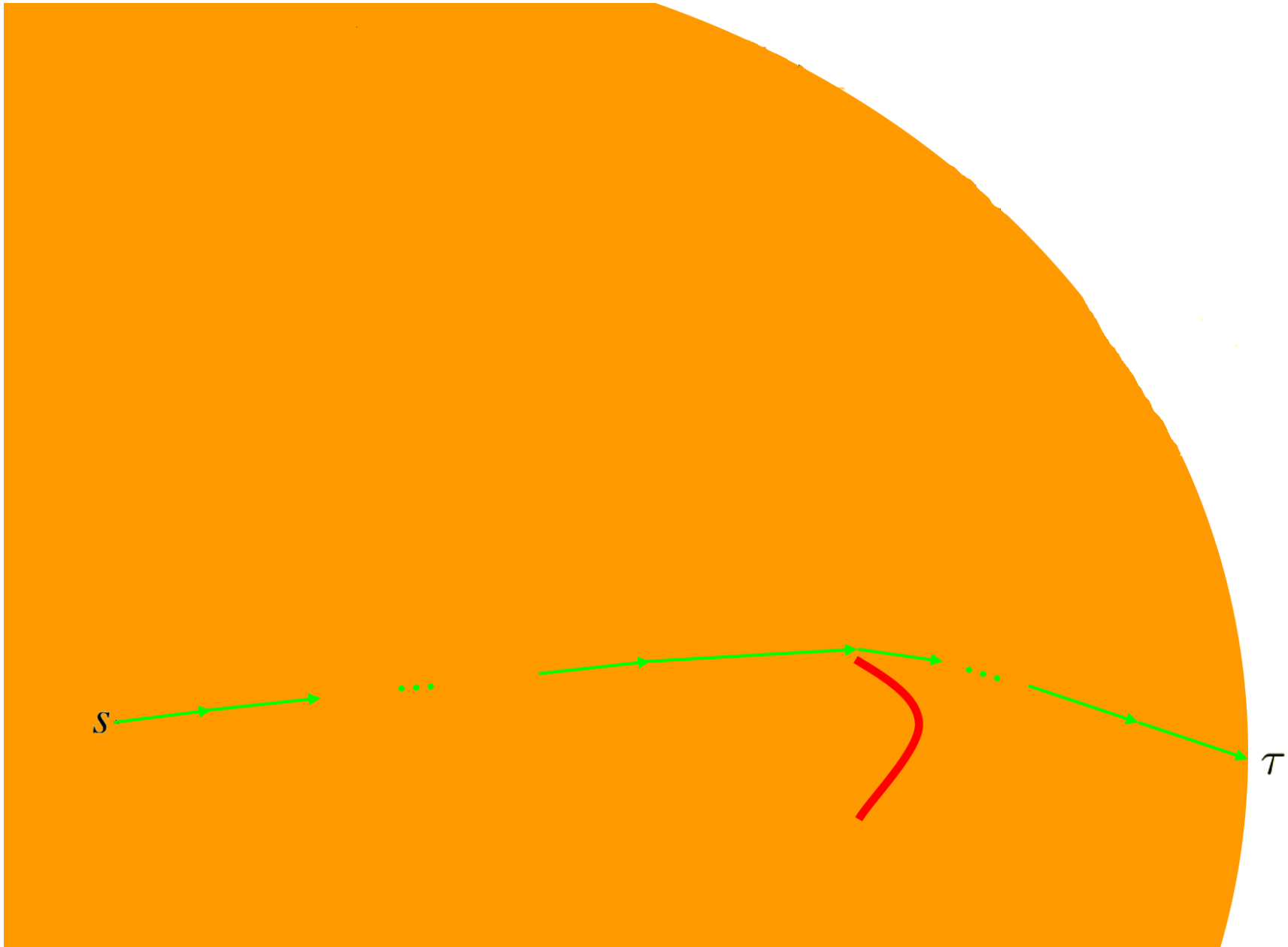
- f_i is an estimate of the cost of a least cost path from s to τ via i



- 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 $f_i < J^*_{s,\tau}$ are expanded
  - some nodes with $f_i = J^*_{s,\tau}$ are expanded
  - no nodes with $f_i > J^*_{s,\tau}$ 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

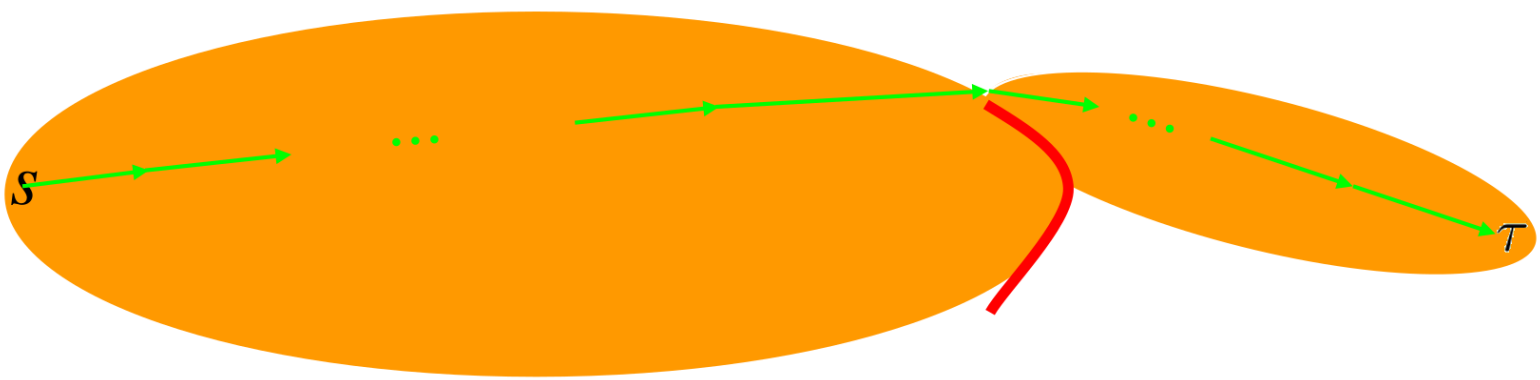
Effect of the Heuristic: Dijkstra

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



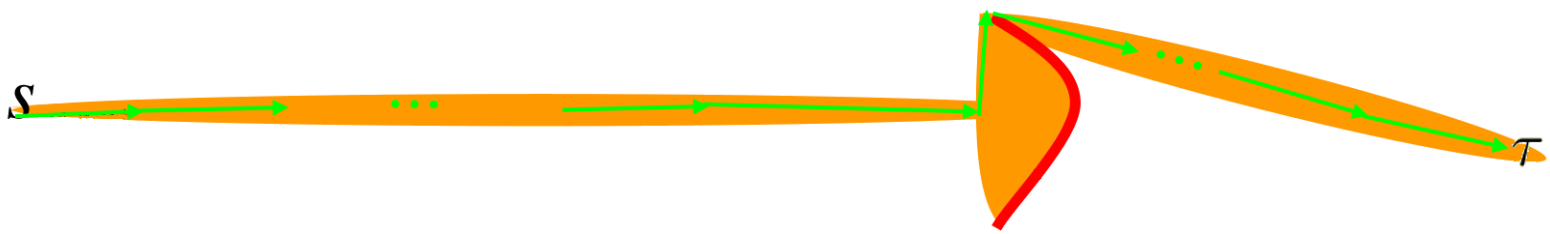
Effect of the Heuristic: A*

- A*: expands states in the order of $f_i = g_i + h_i$
- The closer h_i is to $J^*_{i,\tau}$, the fewer expansions are needed (fast search)
- The closer h_i is to 0, the more expansions are needed (slow search)
- For large problems the number of states that need to be stored, $O(|\mathcal{V}|)$, causes A* to run out of memory!



Effect of the Heuristic: 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 ϵ-suboptimal ($J \leq \epsilon J^*$) 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**: store node-coordinate pairs (i, x_i) in a **map** data structure (stores key-value pairs, e.g., std::unordered_map)
- **Depth-first search**: last-in, first-out (LIFO): *OPEN* is a **stack**
- **Breath-first search**: first-in, first-out (FIFO): *OPEN* is a **queue**
  - Step 0: $OPEN = \{s\}$ with $g_s = 0$
  - Step 1: all Children i of s are in *OPEN*, i.e., all 1-hop neighbors
  - Step t: all t-hop neighbors of s are in *OPEN*
- **Dijkstra search**: *OPEN* is a **priority queue** based on g_i
- **Weighted A* search**: *OPEN* is a **priority queue** based on f_i

Priority Queue

- **Operations**: empty(), top(), push(i,f_i), pop(), update(i,f_i)
- Possible implementations:
  - **Array and make_heap**, e.g., std::priority_queue
    - Complication: How do we know where i is in the queue?
    - Keep a supplemental array indexed by i, and keep pointers in both directions between this array and queue elements
  - **Binary heap**, e.g., boost::heap::d_ary_heap
  - **Fibonacci heap**, e.g., boost::heap::fibonacci_heap

Table 16.1. Comparison of amortized complexity

	`top()`	`push()`	`pop()`	`update()`	`increase()`	`decrease()`	`erase()`	`merge_and_clear()`
`boost::heap::priority_queue`	`O(1)`	O(log(N))	O(log(N))	n/a	n/a	n/a	n/a	O((N+M)*log(N+M))
`boost::heap::d_ary_heap`	`O(1)`	O(log(N))	O(log(N))	O(log(N))	O(log(N))	O(log(N))	O(log(N))	O((N+M)*log(N+M))
`boost::heap::binomial_heap`	`O(1)`	O(log(N))	O(log(N))	O(log(N))	O(log(N))	O(log(N))	O(log(N))	O(log(N+M))
`boost::heap::fibonacci_heap`	`O(1)`	O(1)	O(log(N))	O(log(N)) [a]	O(1)	O(log(N))	O(log(N))	O(1)
`boost::heap::pairing_heap`	`O(1)`	O(2**2*log(log(N)))	O(log(N))	O(2**2*log(log(N)))	O(2**2*log(log(N)))	O(2**2*log(log(N)))	O(2**2*log(log(N)))	O(2**2*log(log(N)))
`boost::heap::skew_heap`	`O(1)`	O(log(N))	O(log(N))	O(log(N))	O(log(N))	O(log(N))	O(log(N))	O(log(N+M))

[a] The fibonacci a `update_lazy()` method, which has O(log(N)) amortized complexity as well, but does not try to consolidate the internal forest

Time Complexity

- **Graph**: number of nodes $|\mathcal{V}|$, number of edges $|\mathcal{E}|$, maximum node degree Δ (number of outgoing edges)
- **Dynamic Programming**: $O(|\mathcal{V}|^3)$:
  - $|\mathcal{V}| \times |\mathcal{V}|$ entries in the table
  - Each entry requires Δ 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:
    $O(|\mathcal{V}|) + O(|\mathcal{V}|)|\mathcal{V}| + O(1)|\mathcal{E}| = O(|\mathcal{V}|^2)$
  - Binary heap:
    $O(|\mathcal{V}|) + O(\log|\mathcal{V}|)|\mathcal{V}| + O(\log|\mathcal{V}|)|\mathcal{E}| = O((|\mathcal{E}| + |\mathcal{V}|)\log|\mathcal{V}|)$
  - Fibonacci heap:
    $O(|\mathcal{V}|) + O(\log|\mathcal{V}|)|\mathcal{V}| + O(1)|\mathcal{E}| = O(|\mathcal{E}| + |\mathcal{V}|\log|\mathcal{V}|)$

	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

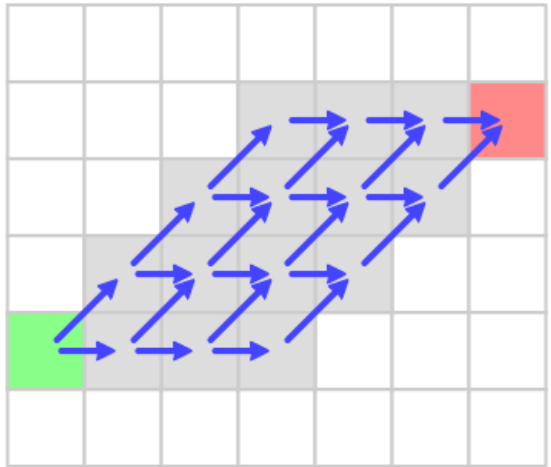
- A* does provably minimum number of expansions, $O(|\mathcal{V}|)$, to find the optimal solution but this might require an infeasible amount of memory
- The memory requirements of weighted A* are often but not always better
- **Depth-first search** (without coloring expanded states): explore each possible path at a time and keep only the best path discovered so far in memory:
  - Complete and optimal (assuming finite state spaces)
  - Memory: $O(\Delta m)$, where Δ - 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
  - DFS may expand over $4^{20} > 10^{12}$ states

IDA*

- What if the goal is only a few steps away in a huge state space?
- **Iterative Deepening A***
  1. Set $f_{max} = 1$ (or some other small value)
  2. Run DFS that expands only states with $f \leq f_{max}$
  3. If DFS returns a path to the goal, **Done!**
  4. Otherwise $f_{max} = f_{max} + 1$ (or larger increment) and go to step 2
- Properties of IDA*
  - Complete and optimal in any state space with positive costs
  - Memory: $O(\Delta m^*)$, where Δ - max. branching factor, m^* - length of optimal path
  - Time: $O(k\Delta^{m^*})$, where k is the number of times DFS is called

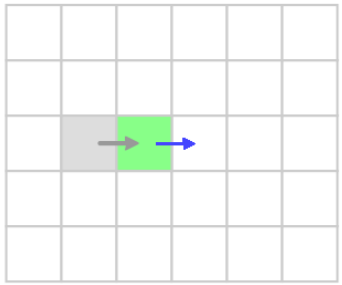
Jump Point Search

- 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 over nodes that are not valuable, e.g., lead to symmetric paths?

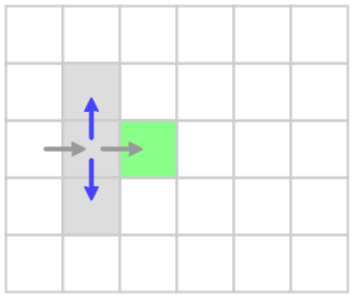


- **Assumption**: undirected uniform-cost grid, i.e., the same move costs the same amount in every node i
- 2-D case:
  - each node has ≤ 8 neighbors
  - straight moves cost 1
  - diagonal moves cost $\sqrt{2}$

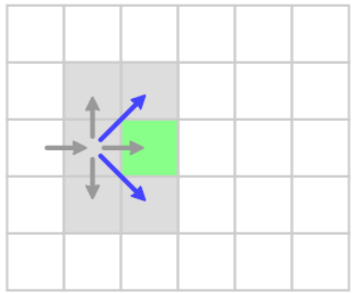
Straight Moves



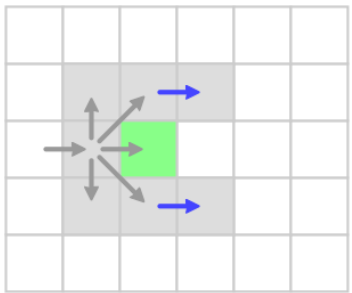
Consider horizontal/vertical movement from node i. We can ignore the node we are coming from (parent $p(i)$) since we already visited it



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

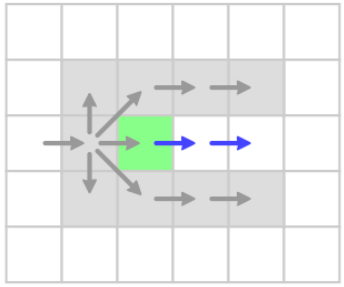


We can assume that the nodes above and below have also been reached via diagonal moves from $p(i)$, which cost $\sqrt{2}$ rather than going through i for a cost of 2

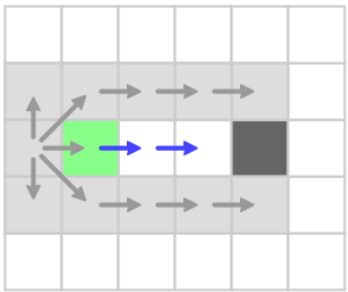


The nodes diagonally in front of us can be reached via the neighbors above and below

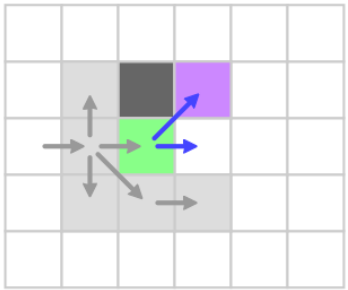
Forced Neighbors for Straight Moves



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.

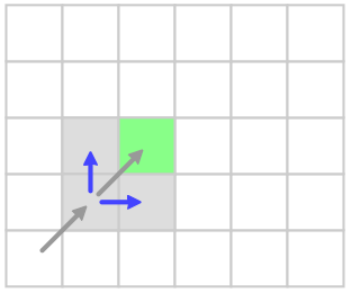


If the way is blocked as we jump to the right, we can safely disregard the entire jump because the paths above and below will be handled via other nodes.

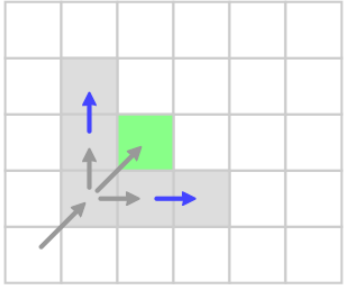


But what happens if one of these neighbors that we assume will cover other paths is blocked? We are **forced** to consider the node that would have otherwise been considered by the blocked path. Such a neighbor is called a **forced neighbor**. When we reach a node with a forced neighbor, we stop jumping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

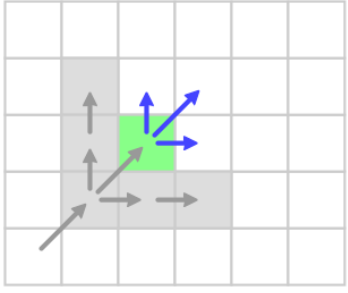
Diagonal Moves



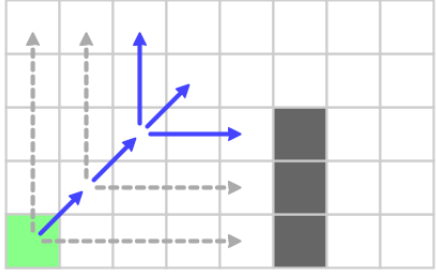
Consider diagonal movement from node i. As before, we can ignore the parent $p(i)$ since we already visited it. We can also ignore the left and below neighbors since they can be reached optimally from $p(i)$ via a straight move.



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.

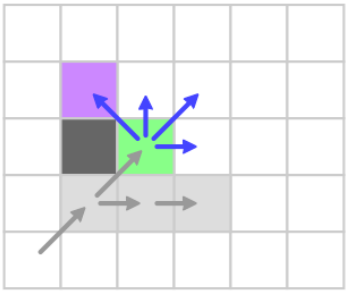


This leaves three **natural neighbors**: two above and to the right, and one diagonally in the original direction of travel.

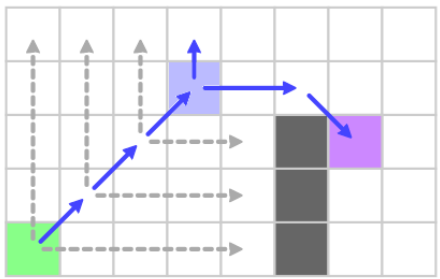


Two of the natural neighbors require straight moves and since we already know how to jump straight we can look there first for forced neighbors. If neither finds any, we move one more step diagonally and repeat.

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 τ
  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

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 ahead 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

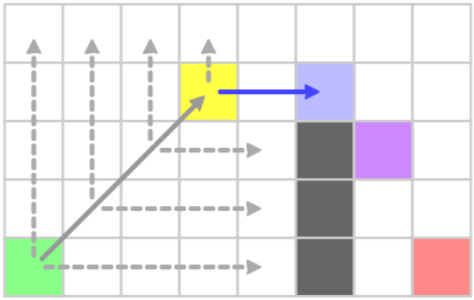
- **Main takeaway**: accessing the contents of many points on a grid in a few iterations of A* is more efficient than maintaining a priority queue over many iterations of A*

2-D Jump Point Search

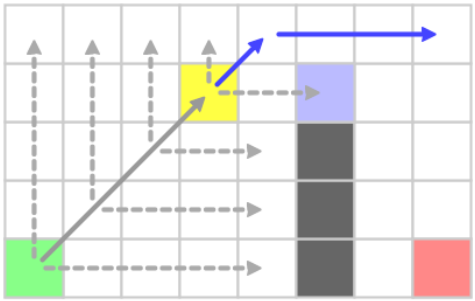
Algorithm 3 2-D Jump Point Search

```
1:  function GETSUCCESSORS(i, τ)
2:      Succ(i) ← ∅
3:      NatNeib(i) ← PRUNE(i,Neib(i))
4:      for all j ∈ NatNeib(i) do
5:          j ← JUMP(i,direction(i,j),τ)
6:          add j to Succ(i)
7:      return Succ(i)
8:
9:  function JUMP(i, d, τ)
10:     j ← STEP(i, d)
11:     if j is an obstacle or outside the grid then
12:         return null
13:     if j = τ or ∃k ∈ Neib(j) such that k is forced then
14:         return j
15:     if ‖d‖1 = 2 (diagonal) then
16:         for k ∈ {1, 2} do
17:             if JUMP(j, dk, τ) is not null then
18:                 return j
19:     return JUMP(j, d, τ)
```

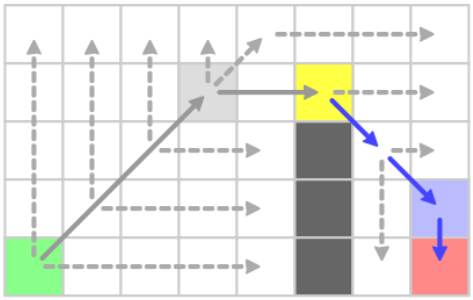
Example: 2-D JPS



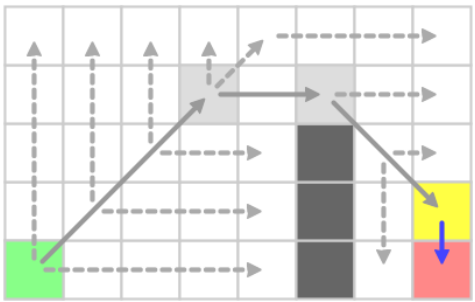
Starting from the green node in OPEN we jump horizontally, then vertically, then diagonally until a jump finds a node (blue) with a forced neighbor (purple). We add the yellow node to OPEN.



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.

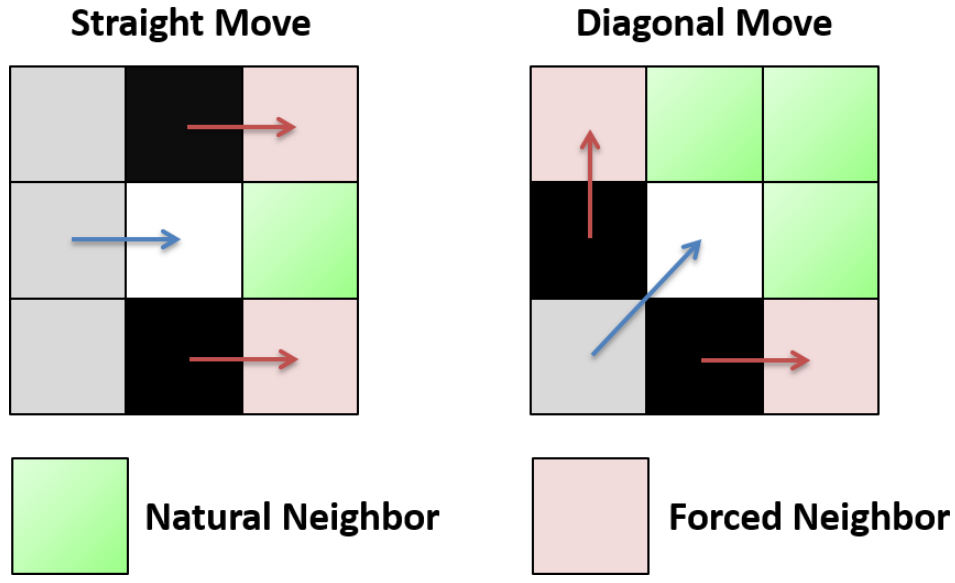


We expand the yellow node from OPEN. Since we were moving diagonally, we first explore the horizontal (leads to map edge) and vertical (blocked) directions and then jump diagonally.



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 2012)

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

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

3-D JPS Pruning Rules