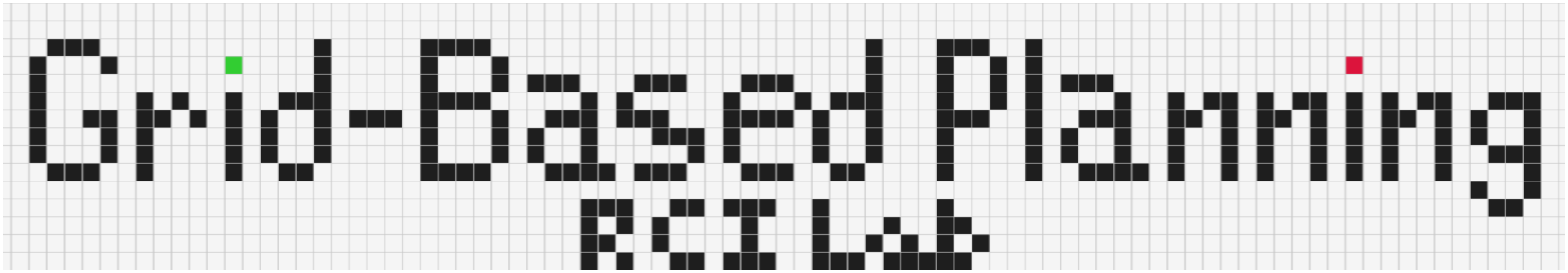
Algorithm – A*



Speaker: Taehyun Jung

RCI LAB

KHU

Table of Contents

01 Limitation of Dijkstra Algorithm

01 Limitation of Dijkstra Algorithm

Main Idea of Dijkstra Algorithm

- Goal: finding a cost-optimal path.
- Principle: expanding the node with smallest cumulative cost g first.
- Container: Priority Queue($\min\_g$) using heapq (item = $(g, \mathrm{row}, \mathrm{column})$).
- Finalization: a node is optimal when popped.
- Stopping rule: stop when the goal is popped.
- Assumption: non-negative edge/cell costs.

※ If all cost of the cells is 1, Dijkstra ≡ BFS (special case).

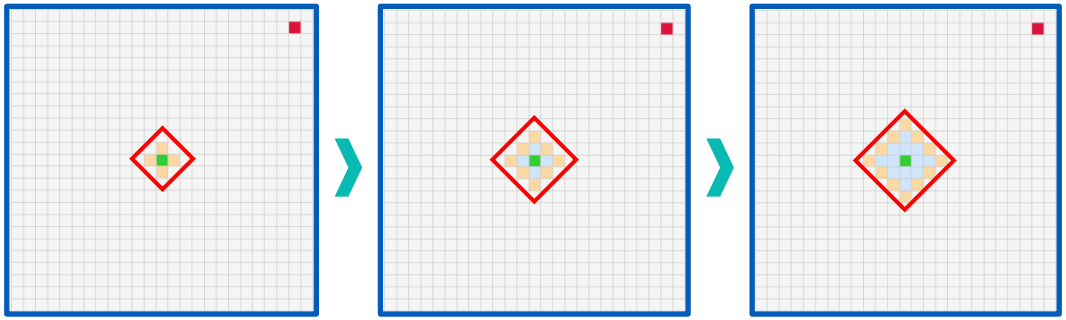
→ In which situations do the limitations of Dijkstra algorithm appear?

01 Limitation of Dijkstra Algorithm

How Dijkstra Algorithm Expands on a Grid?

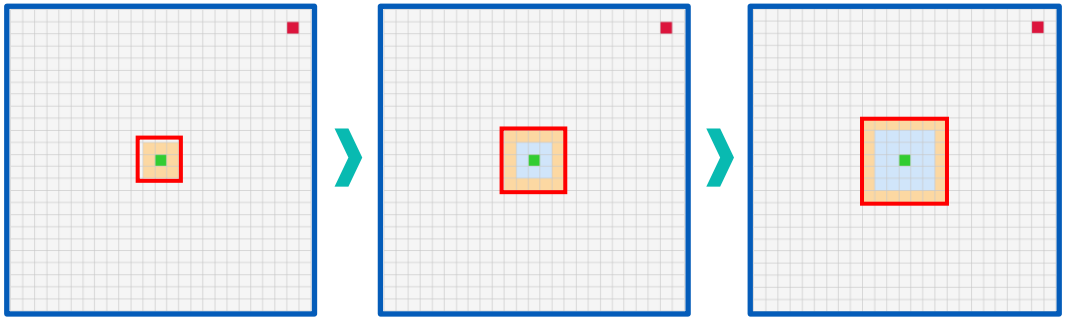
Dijkstra expands along iso-cost shells around the start, using only the accumulated cost $g(n)$.

4-connectivity



→ Rhombus

8-connectivity



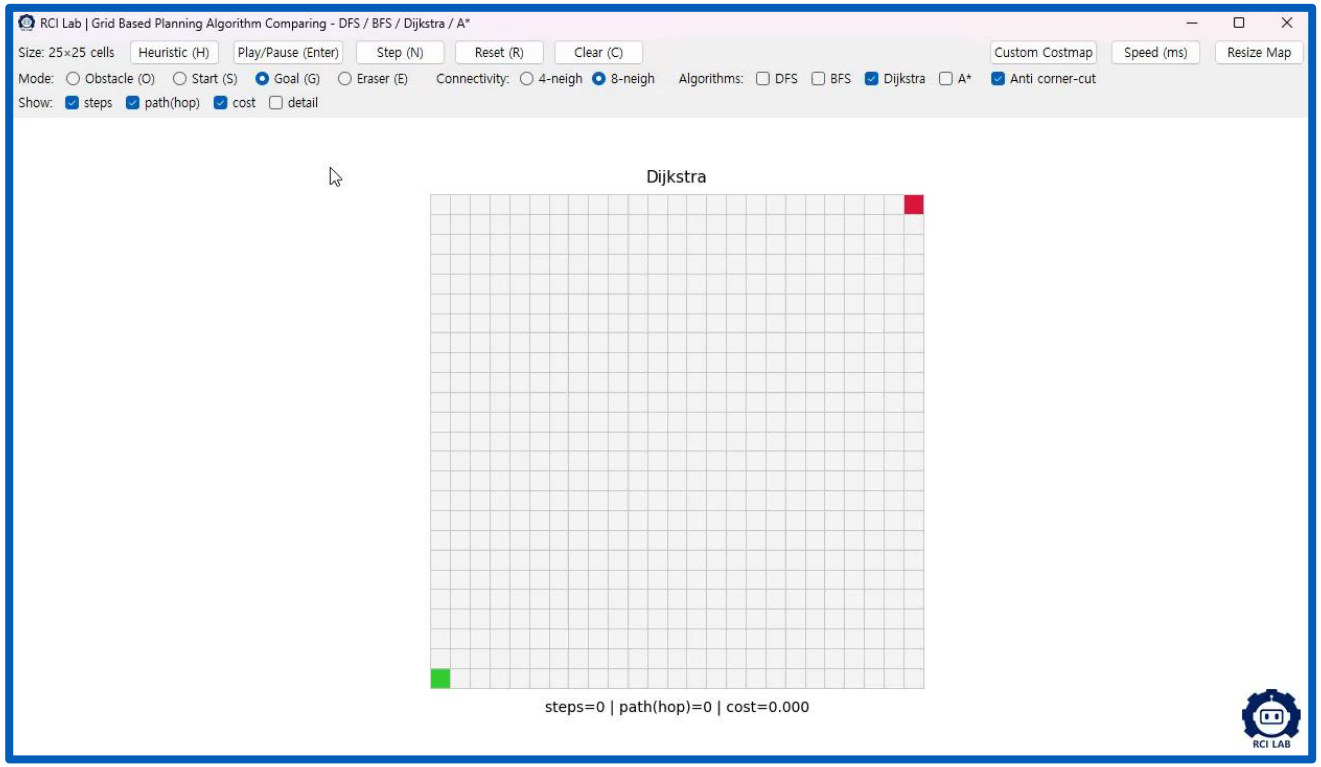
→ Square

- It has no information about how close a node is to the goal. → No directionality toward the goal

01 Limitation of Dijkstra Algorithm

Limitations of Dijkstra Algorithm

Dijkstra expands along iso-cost shells around the start, using only the accumulated cost $g(n)$.

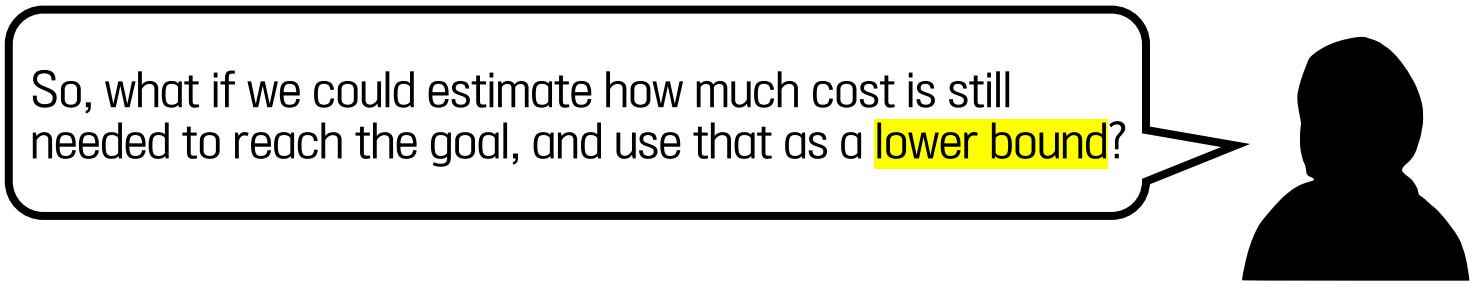


- With nearly uniform costs, Dijkstra follows cost contours, which leads to a lot of unnecessary expansions in large environments.

01 Limitation of Dijkstra Algorithm

Limitations of Dijkstra Algorithm – Summary

- No directionality
  → Dijkstra expands purely by g, without knowing where the goal is.
- Exploring widely
  → Dijkstra follows cost contours and explores many nodes with similar cost.
- No heuristic
  → Dijkstra does not use any information that how much the cost spend at least to reach goal.



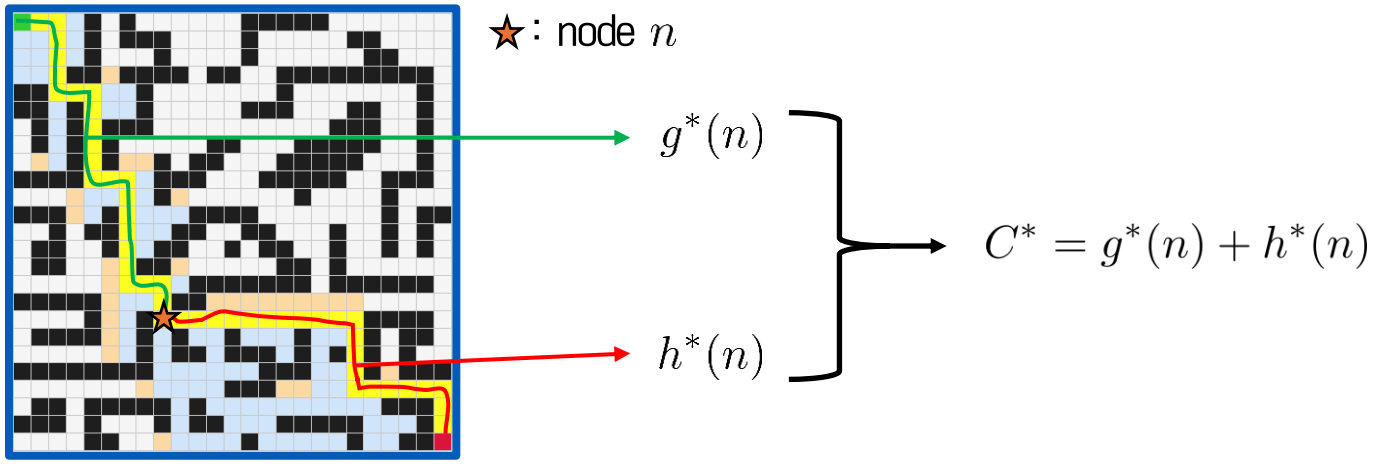
→ This idea leads to heuristics and lower bounds, and eventually to the A* algorithm.

02 Heuristic and Lower Bound

02 Heuristic and Lower Bound

Theoretical Optimal Quantities

- Definitions :
  - C^* : optimal cost from start to goal.
  - $g^*(n)$: optimal prefix cost from start to node n.
  - $h^*(n)$: optimal remaining cost from node n to the goal.



- All three are optimal quantities, so their true values are known only after the search is finished.

02 Heuristic and Lower Bound

Relationship on the Optimal Path

- If node n lies on an optimal path, then the path start → n → goal is part of the optimal path.

$$C^* = g^*(n) + h^*(n)$$
$$\Rightarrow h^*(n) = C^* - g^*(n) \quad \cdots \quad (1)$$

- However, during the search, we do not know C^* (the optimal total cost), $g^*(n)$ (the optimal prefix cost for each node).

- So, formula (1) is only for theoretical analysis, we don't compute $h^*(n)$ this way in an actual algorithm.

Known vs. Unknown During the Search

- Unknown (theoretical quantities):
  - C^* : unknown before the optimal path is found.
  - $g^*(n)$: we don't know yet whether our current path to n is optimal.
  - $h^*(n)$: requires full knowledge of the graph and the optimal solution.
- Known (practical quantities in A*):
  - $g(n)$: cost from start to n along the current path found by the search. ($g(n) \geq g^*(n)$ always satisfies)
  - Graph structure

➢ Instead of $h^*(n)$, we design a function $h(n)$ that lower-bounds $h^*(n)$ and use that.

02 Heuristic and Lower Bound

Heuristics Used in A*

- In grid-based planning, the most common heuristics are distance-based functions to the goal.
- For 4-connected grids,
  - Heuristic : Manhattan distance
  - $h(n) = |x_n - x_{\text{goal}}| + |y_n - y_{\text{goal}}|$
- For 8-connected grids,
  - Heuristic : Octile distance
  - $h(n) = (|\Delta x| + |\Delta y|) + (\sqrt{2} - 2) \cdot \min(|\Delta x|, |\Delta y|)$ or $h(n) = \max(\Delta x, \Delta y) + (\sqrt{2} - 1) \cdot \min(\Delta x, \Delta y)$

    where $\Delta x = x_n - x_{\text{goal}}, \Delta y = y_n - y_{\text{goal}}$
- If straight-line motion(the robot's motion is close to moving in continuous space),
  - Heuristic : Euclidean distance
  - $h(n) = \sqrt{\Delta x^2 + \Delta y^2}$
- In this lecture:
  - We mainly use these distance-based heuristics for A*.
  - They satisfy the properties of admissible and consistent heuristics.
    (Other heuristics may fail to satisfy these properties, which can cause A* to lose optimality.)

02 Heuristic and Lower Bound

Heuristic $h(n)$ - Admissible

- Definition : a function that estimates the remaining cost from node n to the goal.
- Design goal :
  - Ideally, for all node n, we want $0 \leq h(n) \leq h^*(n)$.
  - That is, the heuristic $h(n)$ should be an optimistic prediction, never greater than the true optimal remaining cost $h^*(n)$.

→ In this sense, $h(n)$ is a lower-bounding approximation of $h^*(n)$.

→ This means 'at least this much cost is needed', not 'it will definitely spend at least this much.'

  - A heuristic that satisfies this condition is called an admissible heuristic.

※ This lower-bound property is crucial for A* to preserve Dijkstra-style optimality (stop on pop).

→ The g of the node is the optimal when popping

※ The distance-based heuristics introduced are typical examples of admissible heuristics, since they are lower bounds on the true shortest path cost.

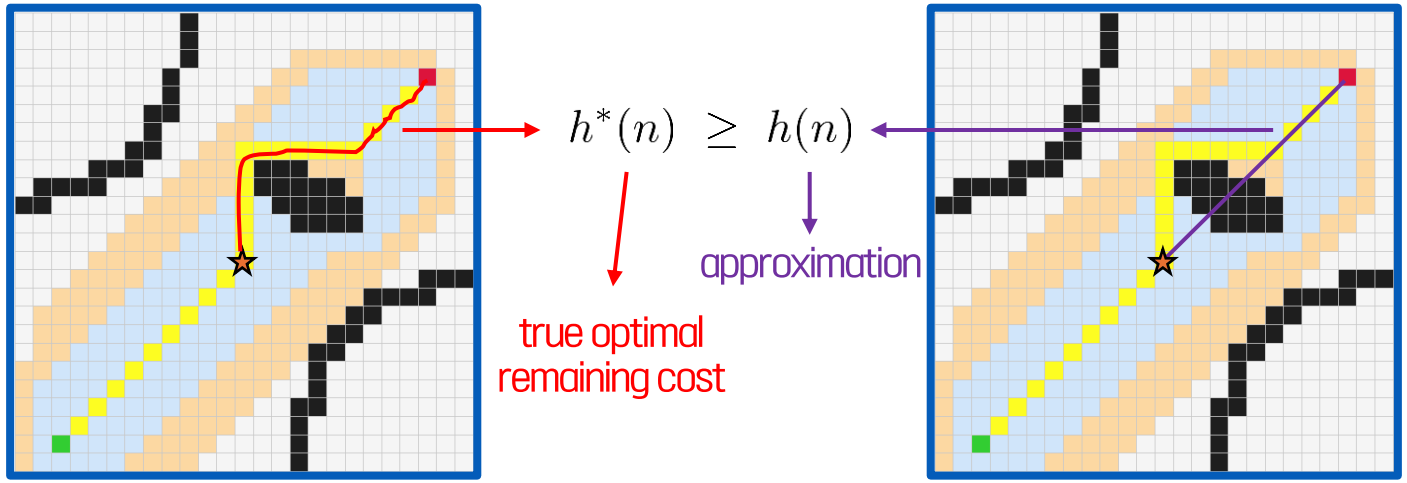
02 Heuristic and Lower Bound

Heuristic $h(n)$ - Admissible

- Definition : a function that estimates the remaining cost from node n to the goal.
- Design goal :
  - Ideally, for all node n, we want $0 \leq h(n) \leq h^*(n)$.
  - That is, the heuristic $h(n)$ should be an optimistic prediction, never greater than the true optimal remaining cost $h^*(n)$.

→ In this sense, $h(n)$ is a lower-bounding approximation of $h^*(n)$.

→ This means 'at least this much cost is needed', not 'it will definitely spend at least this much.'



02 Heuristic and Lower Bound

Heuristic $h(n)$ - Consistent

- Definition :
  - For every edge $u \to v$, $h(u) \leq w(u, v) + h(v)$.
  - A heuristic satisfying this is called consistent (or monotone).
- Meaning :
  - Along an edge, the heuristic value decreases gradually.
  - The drop in heuristic is never larger than the edge cost.
- Result :
  - With non-negative edge costs and a consistent heuristic, g of a node is optimal when it is popped.
  - Therefore, if we stop when the goal is popped, the resulting path is cost-optimal.

※ The distance-based heuristics are also consistent when the edge costs are non-negative and the heuristic is aligned with the movement model(connectivity).

02 Heuristic and Lower Bound

Lemma – Consistent Heuristic

- Lemma :
  - At the moment v is updated via u, $f(u) \leq f(v)$.
    ⇒ Along a path generated by A*, f-values are non-decreasing.
- Assumptions :
  - Edge costs(move_cost + others) : $w(u, v) \geq 0$
  - Heuristic h is consistent : $h(u) \leq w(u, v) + h(v) \quad \forall (u \rightarrow v)$
  - When A* updates v via u : $g(v) = g(u) + w(u, v)$
- Proof :
  1. Consistency : $h(u) \leq w(u, v) + h(v)$
  2. Add $g(u)$ to both sides : $g(u) + h(u) \leq g(u) + w(u, v) + h(v)$
  3. At the update moment : $g(v) = g(u) + w(u, v)$
  4. Hence : $f(u) = g(u) + h(u) \leq g(v) + h(v) = f(v) \quad \Rightarrow \quad f(u) \leq f(v)$
  5. So, parent' f $\leq$ child' f ■

02 Heuristic and Lower Bound

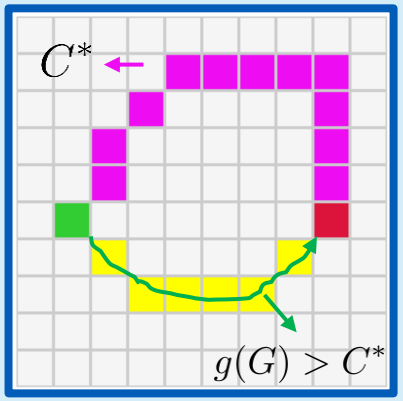
Theorem – Stop-on-pop Optimality of A*

- Assumptions :
  - All edge costs $w(u, v) \geq 0$.
  - Heuristic h is consistent : $h(\text{goal}) = 0 \quad \equiv \quad f(\text{goal}) = g(\text{goal}) = C^* = g^*(\text{goal})$
  - A* is a graph search with a closed set(visited), always popping the node with smallest $f(n) = g(n) + h(n)$.
- Theorem :
  - Whenever a node n is popped from the PQ, its cost $g(n)$ equals the optimal cost $g^*(n)$.
  - In particular, when the goal is popped, $g(\text{goal}) = C^*$, the optimal cost from start to goal.
  - So stopping when the goal is popped yields an optimal path.

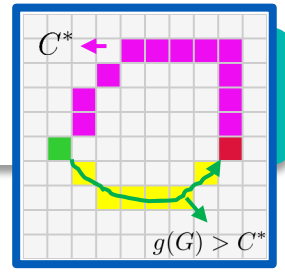
02 Heuristic and Lower Bound

Theorem – Stop-on-pop Optimality of A*

- Proof : Show that when the goal is popped, $g(\text{goal})$ equals the optimal cost C^* ($g(\text{goal}) = C^*$).
  - Assumptions :
    - Suppose A* has just popped the goal node $G = \text{goal}$.
    - Let $g(G)$ be the cost of the path that A* has found to G. → Assume this path is not optimal.
    - Optimal cost from start to G : C^*
    - Cost found by A* : $g(G)$
    - Assumption : $C^* < g(G)$.



02 Heuristic and Lower Bound



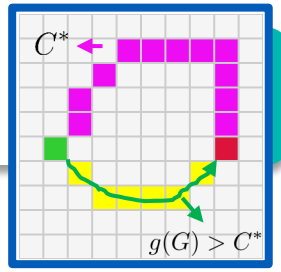
Theorem – Stop-on-pop Optimality of A*

- Proof : Show that when the goal is popped, $g(\text{goal})$ equals the optimal cost C^* ($g(\text{goal}) = C^*$).
  - Let the optimal path from start to goal be $S = n_0, n_1, \cdots, n_k = G$.
  - At the moment when A* pops G,
    let n_i be the first node on this optimal path that has not been popped yet.
    - Then n_{i-1} has already been popped.
    - When A* expanded n_{i-1}, it generated n_i and inserted it into OPEN list(PQ).
    - Since the search can follow the optimal path up to n_i,

$$g(n_i) = g^*(n_i) \qquad (\because n_i \in \text{optimal\_path})$$

  - Now use the admissible heuristic property.
    - Admissible : $h(n_i) \le h^*(n_i)$
    - On the optimal path : $g^*(n_i) + h^*(n_i) = C^*$
    - Therefore, $f(n_i) = g(n_i) + h(n_i) = g^*(n_i) + h(n_i) \le g^*(n_i) + h^*(n_i) = C^* \quad \Rightarrow \quad f(n_i) \le C^*$

02 Heuristic and Lower Bound



Theorem – Stop-on-pop Optimality of A*

- Proof : Show that when the goal is popped, $g(\text{goal})$ equals the optimal cost C^* ($g(\text{goal}) = C^*$).
  - Now use the admissible heuristic property.
    - Admissible : $h(n_i) \le h^*(n_i)$
    - On the optimal path : $g^*(n_i) + h^*(n_i) = C^*$
    - Therefore, $f(n_i) = g(n_i) + h(n_i) = g^*(n_i) + h(n_i) \le g^*(n_i) + h^*(n_i) = C^* \quad \Rightarrow \quad f(n_i) \le C^*$
  - On the other hand, by our assumption $C^* < g(G)$, and since $h(G) = 0$,

$$f(G) = g(G) + h(G) = g(G).$$

    Hence,

$$f(n_i) \le C^* < g(G) = f(G),$$

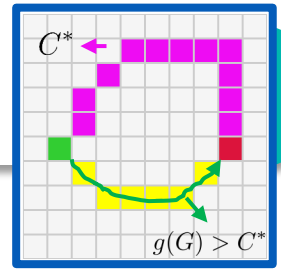
    i.e.,

$$f(n_i) < f(G).$$

  - However, A* always pops the node with the smallest f-value.
  - If there is a node n_i in OPEN(PQ) such that $f(n_i) < f(G)$, A* should have popped n_i before G.

➢ This contradicts the assumption that G is popped now.

02 Heuristic and Lower Bound



Theorem – Stop-on-pop Optimality of A*

- **Proof :** Show that when the goal is popped, $g(\text{goal})$ equals the optimal cost C^* ($g(\text{goal}) = C^*$).
  - On the other hand, by our assumption $C^* < g(G)$, and since $h(G) = 0$,

$$f(G) = g(G) + h(G) = g(G).$$

    Hence,

$$f(n_i) \leq C^* < g(G) = f(G),$$

    i.e.,

$$f(n_i) < f(G).$$

  - However, A* always pops the node with the smallest f -value.
  - If there is a node n_i in OPEN(PQ) such that $f(n_i) < f(G)$, A* should have popped n_i before G.

  ➢ This contradicts the assumption that G is popped now.

  - Therefore, the assumption $C^* < g(G)$ is false, and we must have $C^* \geq g(G)$. $\cdots(1)$
  - From the definition of $g(n)$, $g(n) \geq g^*(n) \forall n \quad \Rightarrow \quad g(G) \geq g^*(G) = C^*$ $\cdots(2)$
  - From (1) and (2) , $C^* \leq g(G) \leq C^*$.
  - $\therefore \; g(G) = C^*$ ■ → This means, when the goal is popped, the path cost $g(G)$ found by A* exactly equals the minimum possible cost C^*.

03 A* Algorithm

03 A* Algorithm

A* Algorithm

- Goal : find a cost-optimal path from start to goal.
- Key idea : $f(n) = g(n) + h(n)$ and always expand the node with the smallest $f(n)$.
- Roles
  - $g(n)$: cost already spent from the start (past cost).
  - $h(n)$: heuristic lower bound on the remaining cost to the goal.
- Relation to Dijkstra
  - If $h(n) \equiv 0$, A* is exactly Dijkstra.
  - So A* can be seen as Dijkstra with heuristic.
- Optimality

  → With an admissible & consistent heuristic, A* preserves stop-on-pop optimality just like Dijkstra.
- Search Pattern
  - Heuristic gives directionality toward the goal.
  - A* has significantly fewer expansions than Dijkstra.

03 A* Algorithm

Pseudocode

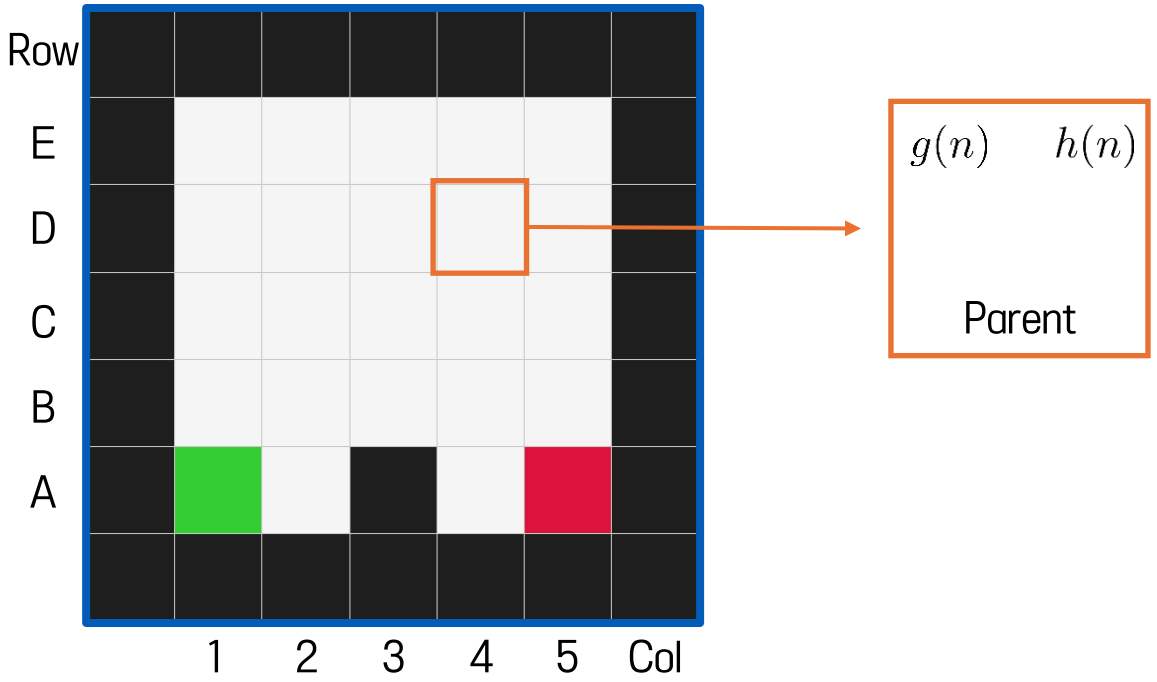
```
1)  S ← start
2)  OPEN ← PQ ordered by f(n) = g(n) + h(n)
3)  CLOSED ← ∅                                  ▶ empty set

4)  for all node n :
5)          g(n) ← ∞
6)          parent[n] ← None

7)  g(S) ← 0
8)  f(S) ← g(S) + h(S)
9)  push(OPEN, (f(S), g(S) S))

11) while OPEN is not empty:
12)         g(u), u ← pop(PQ)                   ▶ pop minimum f
13)
14)         if u == goal:                       ▶ check goal after popping
15)                 return reconstruct(parent, goal)
16)
17)         CLOSED.add(u)                       ▶ CLOSED ~= visited
18)
19)         for each (u→v) in neighbors(u):
20)                 if v is in CLOSED :     continue
21)                 new_g ← g(u) + w(u, v)
22)                 if new_g < g(v):
23)                         g(v) ← new_g
24)                         f(v) ← g(v) + h(v)      ▶ update f
25)                         parent[v] ← u
26)                         if v not in OPEN :
27)                                 push(OPEN, v)
28)
29) return failure                              ▶ no path from S to Goal
```

03 A* Algorithm

Example of A* Algorithm

- Start cell : , Goal cell : , Obstacle cell :
- Each edge(move_cost) : cost = 1 → only straight(4-connectivity)
- For equal f, use FIFO(the order in which they came into OPEN).
- If some child nodes have same cost, the order of pushing follows NESW.

tie tie-breaker



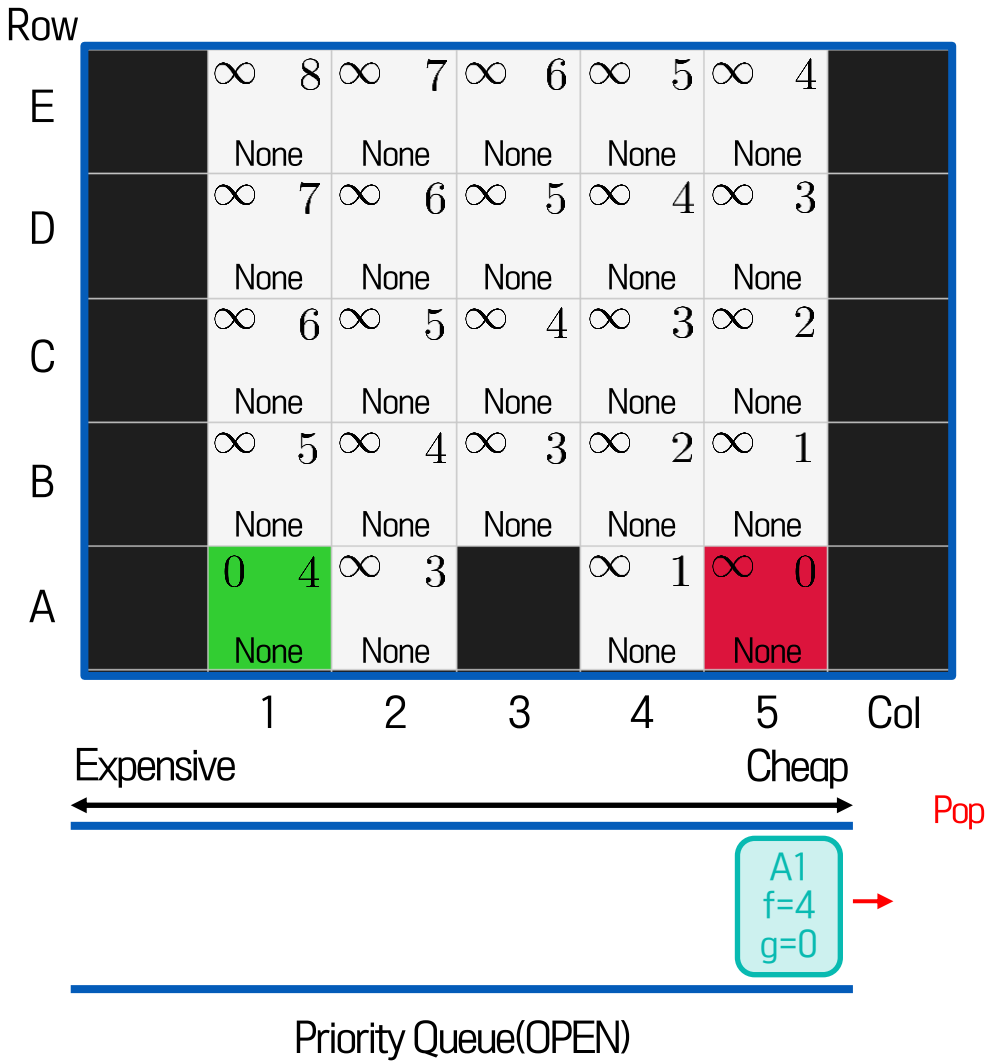
03 A* Algorithm

Example of A* Algorithm

Step 0 - Initialize

- OPEN ← A1 ▶ A1 == start
- V ← all cells
- for n in all node:
  - g(n) ← ∞
  - parent[n] ← None
- g[A1] ← 0
- push(OPEN, (f[A1], g[A1], A1)) ▶ g[A1] == 0, h[A1] == 4

Simply, push(OPEN, A1)

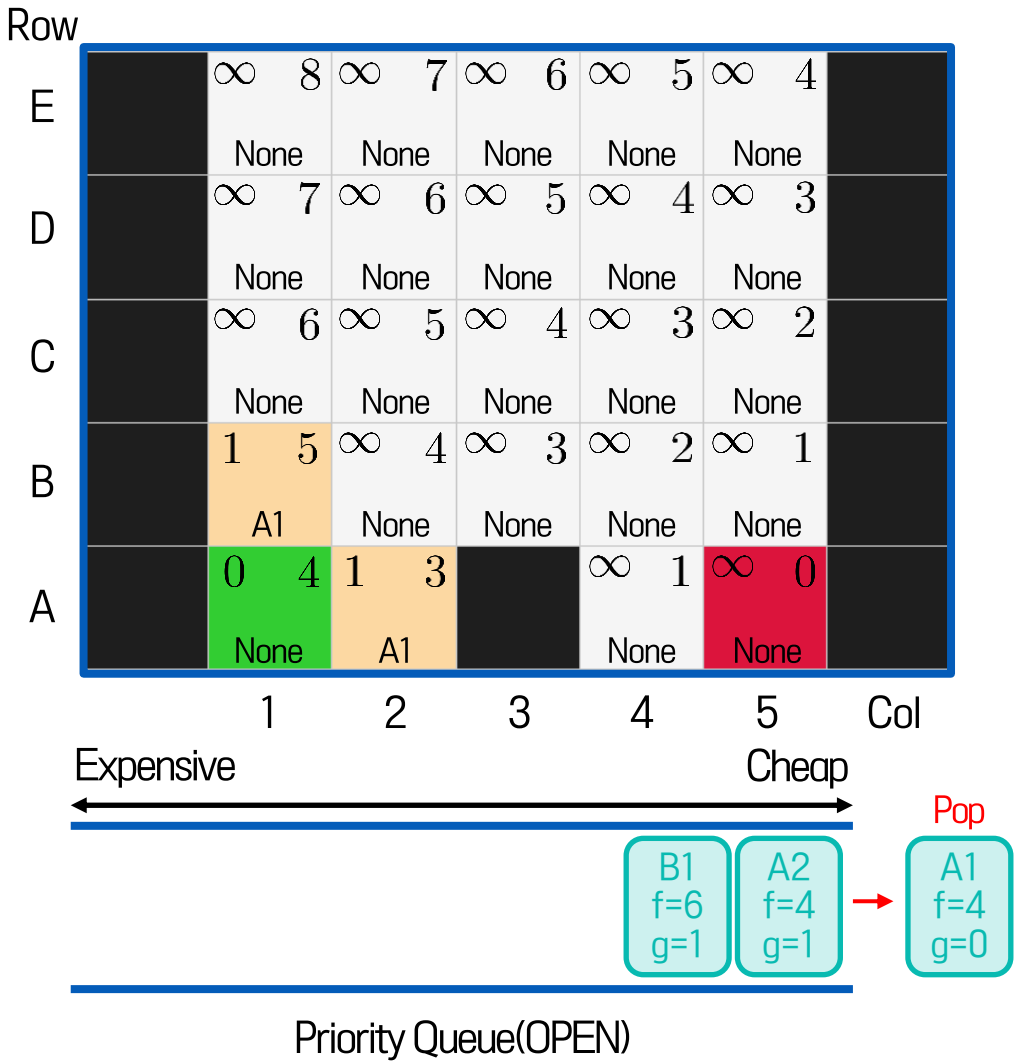


03 A* Algorithm

Example of A* Algorithm

Step 1

```
g(u), u ← pop(OPEN)                    ▶ pop(OPEN) == (0, A1)
if u == goal :                         ▶ X
    return reconstruct(parent, goal)
CLOSED.add(u)

for v in neighbors(u) :                ▶ B1, A2
    if v is in CLOSED : continue       ▶ X
    new_g ← g(u) + w(u, v)             ▶ B1(1), A2(1)
    if new_g < g(v) :
        g(v) ← new_g
        f(v) ← g(v) + h(v)             ▶ B1(4), A2(4)
        parent[v] ← u                  ▶ B1(A1), A2(A1)
        if v not in OPEN :
            push(OPEN, v)              ▶ push A2, B1
```

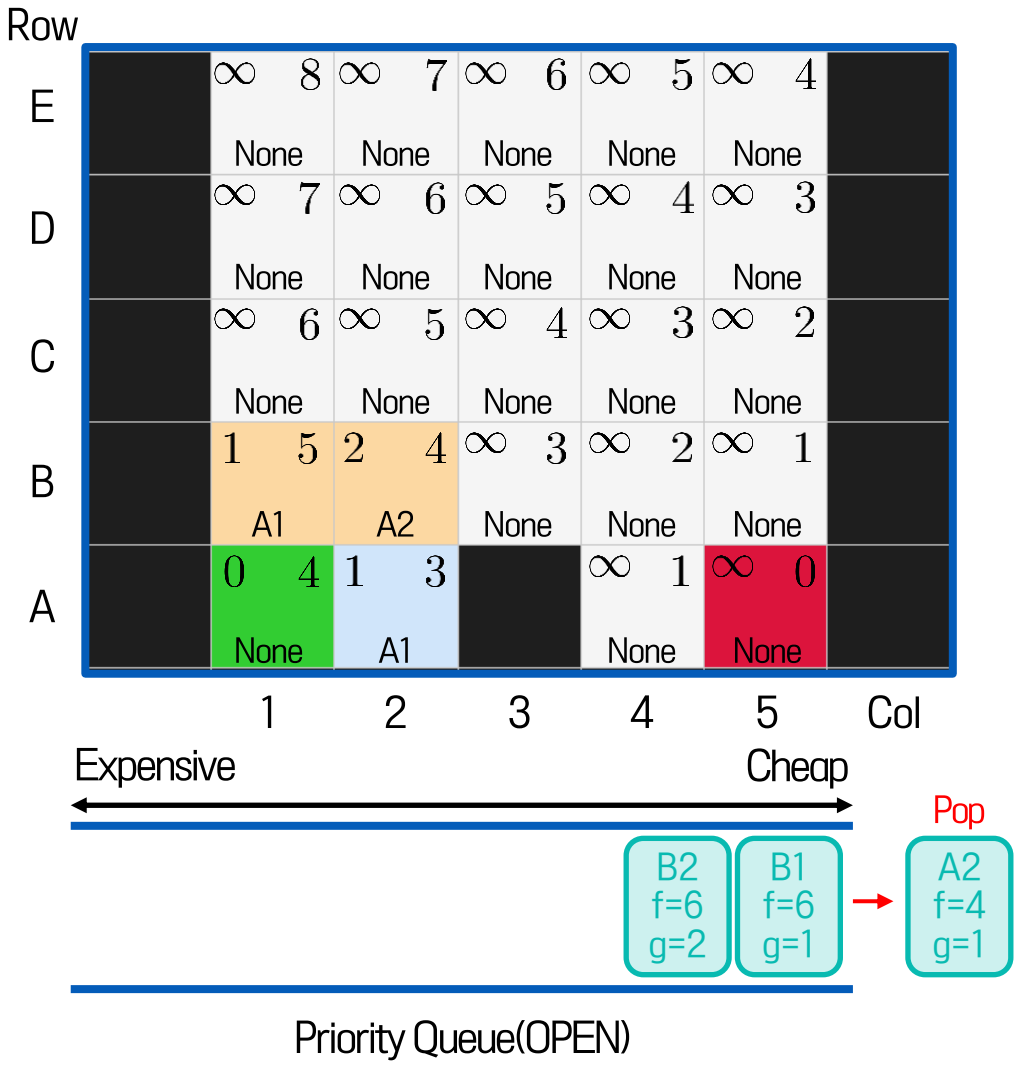


03 A* Algorithm

Example of A* Algorithm

Step 2

- g(u), u ← pop(OPEN) ▶ pop(OPEN) == (1, A2)
- if u == goal : ▶ X
- return reconstruct(parent, goal)
- CLOSED.add(u)
-
- for v in neighbors(u) : ▶ B2, A1
- if v is in CLOSED : continue ▶ A1
- new_g ← g(u) + w(u, v) ▶ B2(2)
- if new_g < g(v) :
- g(v) ← new_g
- f(v) ← g(v) + h(v) ▶ B2(6)
- parent[v] ← u ▶ B2(A2)
- if v not in OPEN :
- push(OPEN, v) ▶ push B2

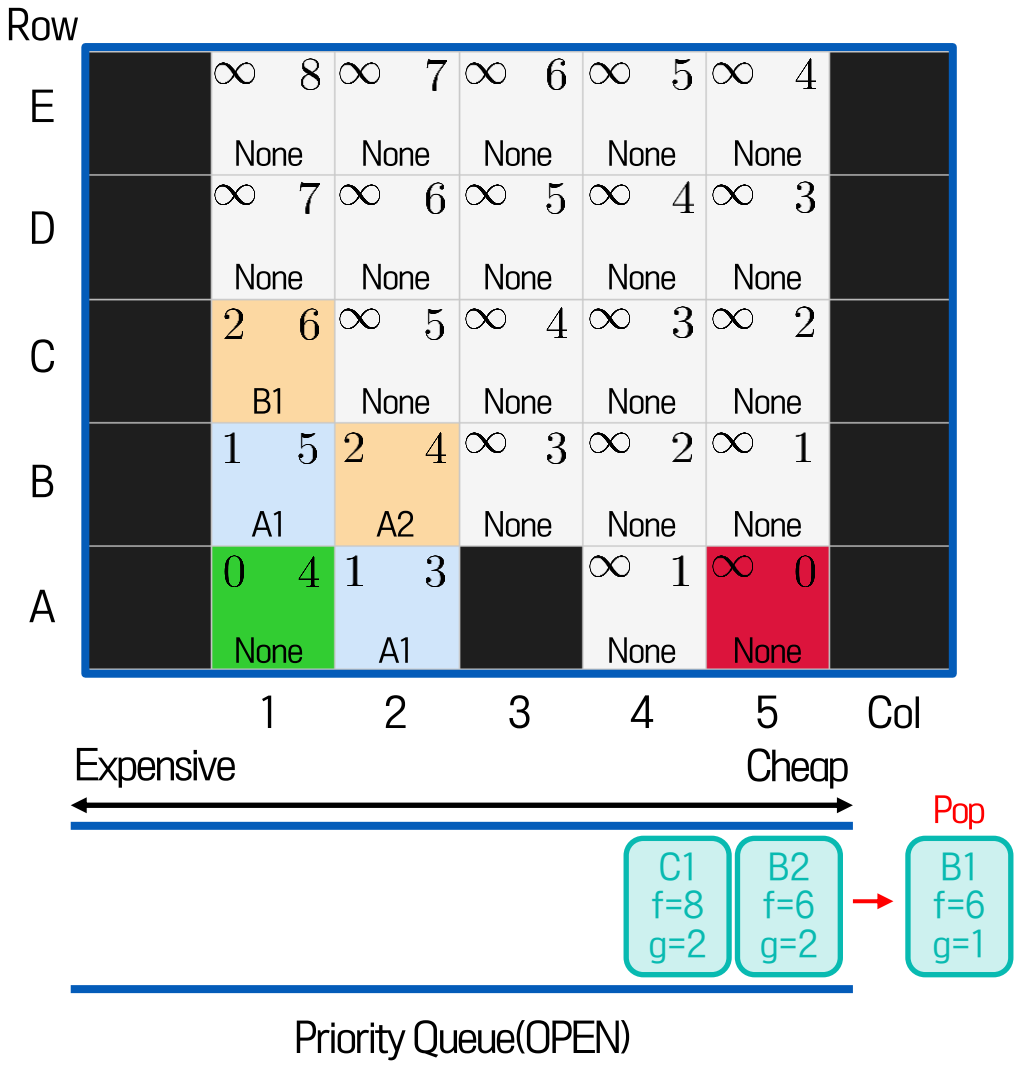


03 A* Algorithm

Example of A* Algorithm

Step 3

- g(u), u ← pop(OPEN) ▶ pop(OPEN) == (1, B1)
- if u == goal : ▶ X
- return reconstruct(parent, goal)
- CLOSED.add(u)
-
- for v in neighbors(u) : ▶ C1, B2, A1
- if v is in CLOSED : continue ▶ A1
- new_g ← g(u) + w(u, v) ▶ C1(2)
- if new_g < g(v) :
- g(v) ← new_g
- f(v) ← g(v) + h(v) ▶ C1(8)
- parent[v] ← u ▶ C1(B1)
- if v not in OPEN : ▶ B2 is already in OPEN
- push(OPEN, v) ▶ push C1

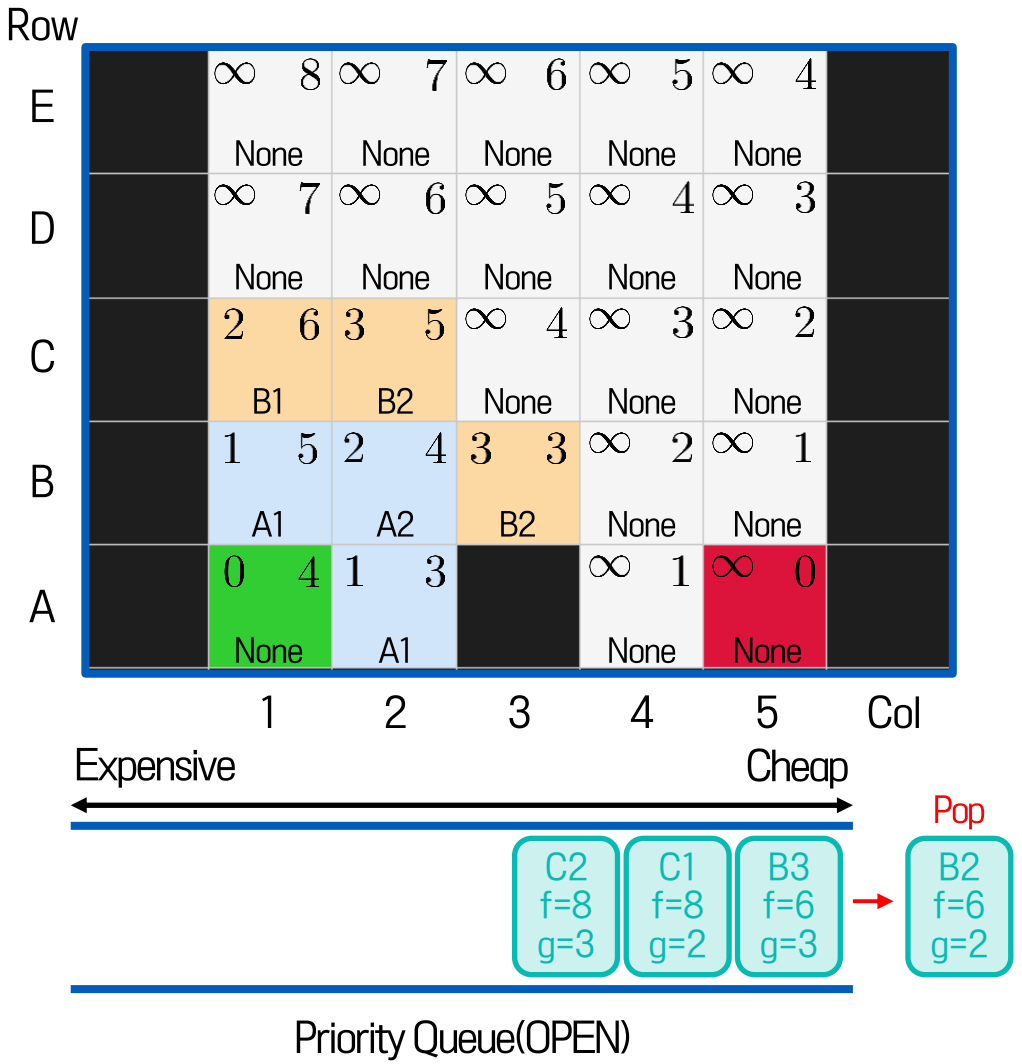


03 A* Algorithm

Example of A* Algorithm

Step 4

- g(u), u ← pop(OPEN) ▶ pop(OPEN) == (2, B2)
- if u == goal : ▶ X
- return reconstruct(parent, goal)
- CLOSED.add(u)
-
- for v in neighbors(u) : ▶ C2, B3, A2, B1
- if v is in CLOSED : continue ▶ A2, B1
- new_g ← g(u) + w(u, v) ▶ C2(3), B3(3)
- if new_g < g(v) :
- g(v) ← new_g
- f(v) ← g(v) + h(v) ▶ C2(8), B3(6)
- parent[v] ← u ▶ C2(B2), B3(B2)
- if v not in OPEN :
- push(OPEN, v) ▶ push B3, C2

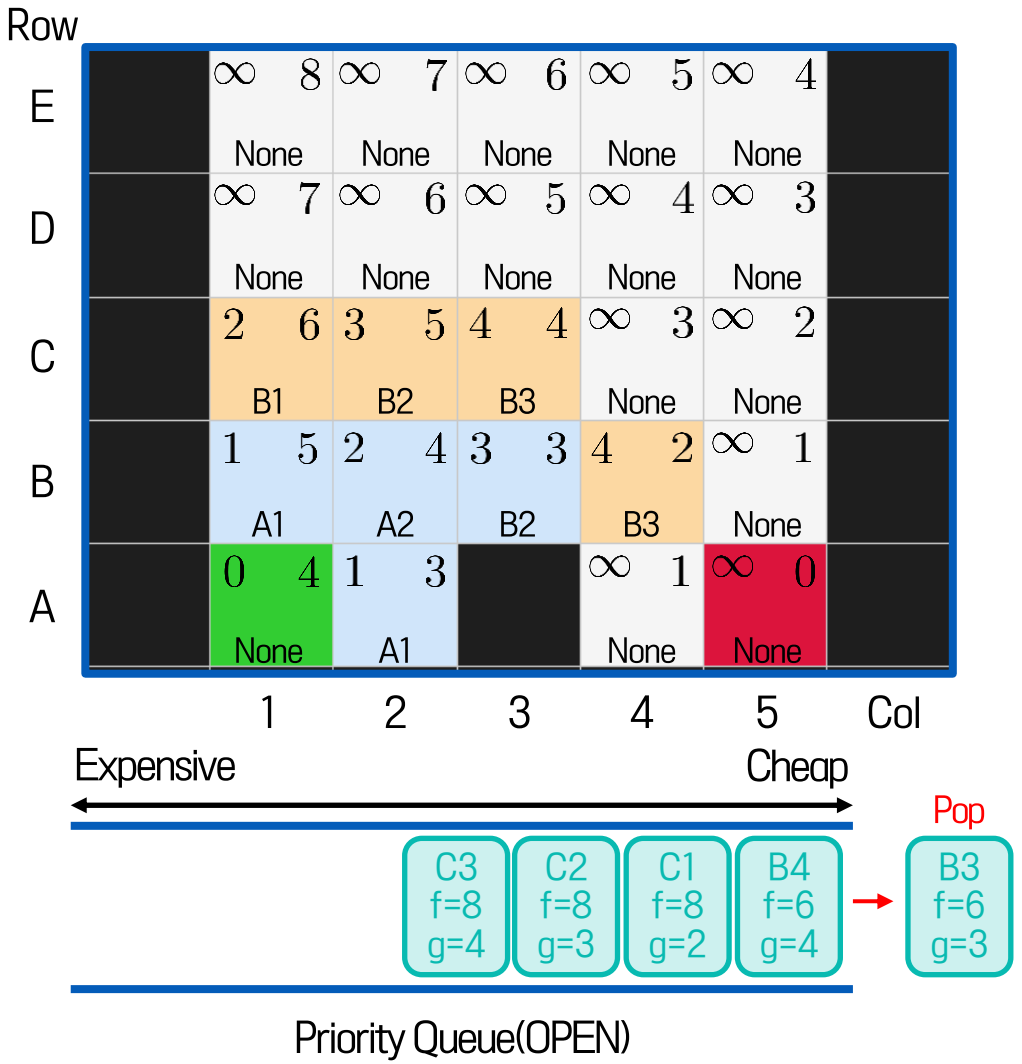


03 A* Algorithm

Example of A* Algorithm

Step 5

- g(u), u ← pop(OPEN) ▶ pop(OPEN) == (3, B3)
- if u == goal : ▶ X
- return reconstruct(parent, goal)
- CLOSED.add(u)
-
- for v in neighbors(u) : ▶ C3, B4, B2
- if v is in CLOSED : continue ▶ B2
- new_g ← g(u) + w(u, v) ▶ C3(4), B4(4)
- if new_g < g(v) :
- g(v) ← new_g
- f(v) ← g(v) + h(v) ▶ C3(8), B4(6)
- parent[v] ← u ▶ C3(B3), B4(B3)
- if v not in OPEN :
- push(OPEN, v) ▶ push B4, C3

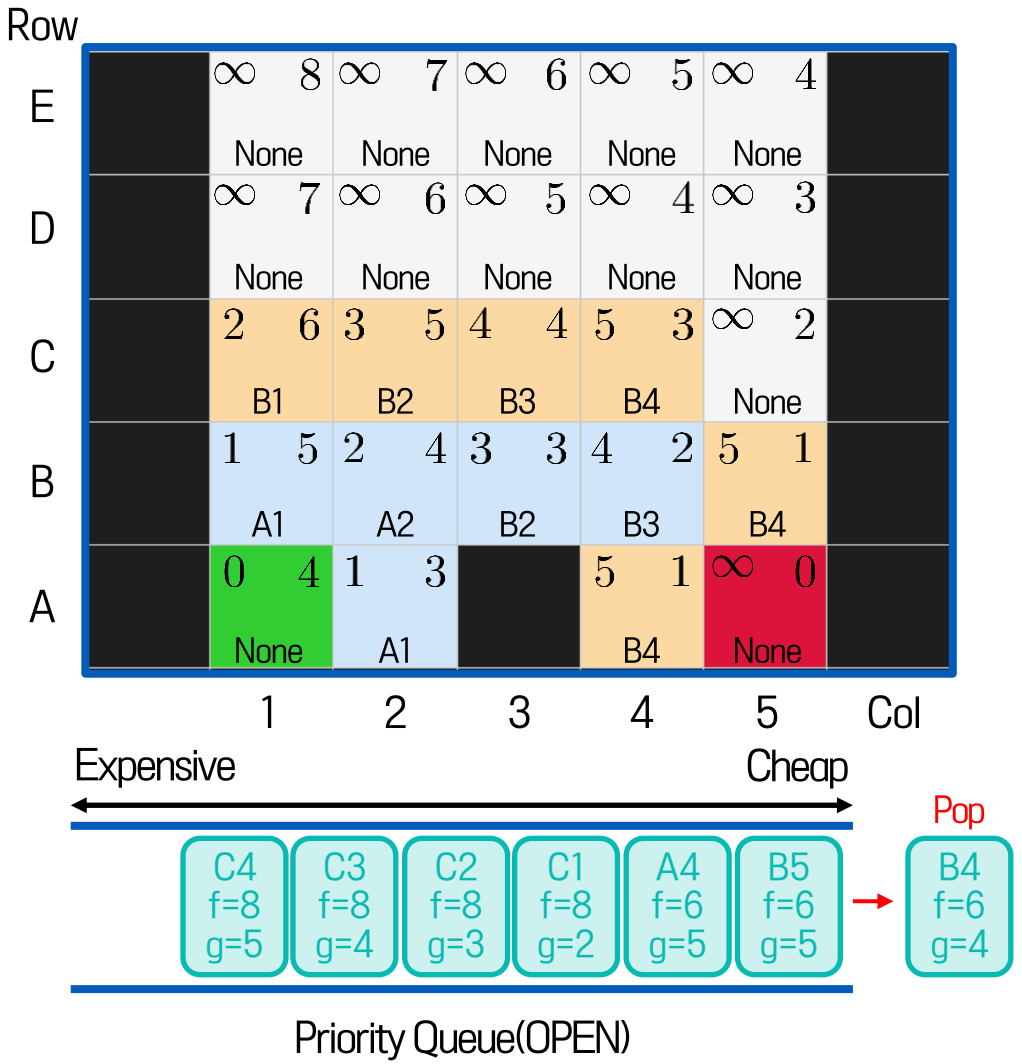


03 A* Algorithm

Example of A* Algorithm

Step 6

- g(u), u ← pop(OPEN) ▶ pop(OPEN) == (4, B4)
- if u == goal : ▶ X
- return reconstruct(parent, goal)
- CLOSED.add(u)
-
- for v in neighbors(u) : ▶ C4, B5, A4, B3
- if v is in CLOSED : continue ▶ B3
- new_g ← g(u) + w(u, v) ▶ C4(5), B5(5), A4(5)
- if new_g < g(v) :
- g(v) ← new_g
- f(v) ← g(v) + h(v) ▶ C4(8), B5(6), A4(6)
- parent[v] ← u ▶ C4(B4), B5(B4), A4(B4)
- if v not in OPEN :
- push(OPEN, v) ▶ push B5, A4, C4

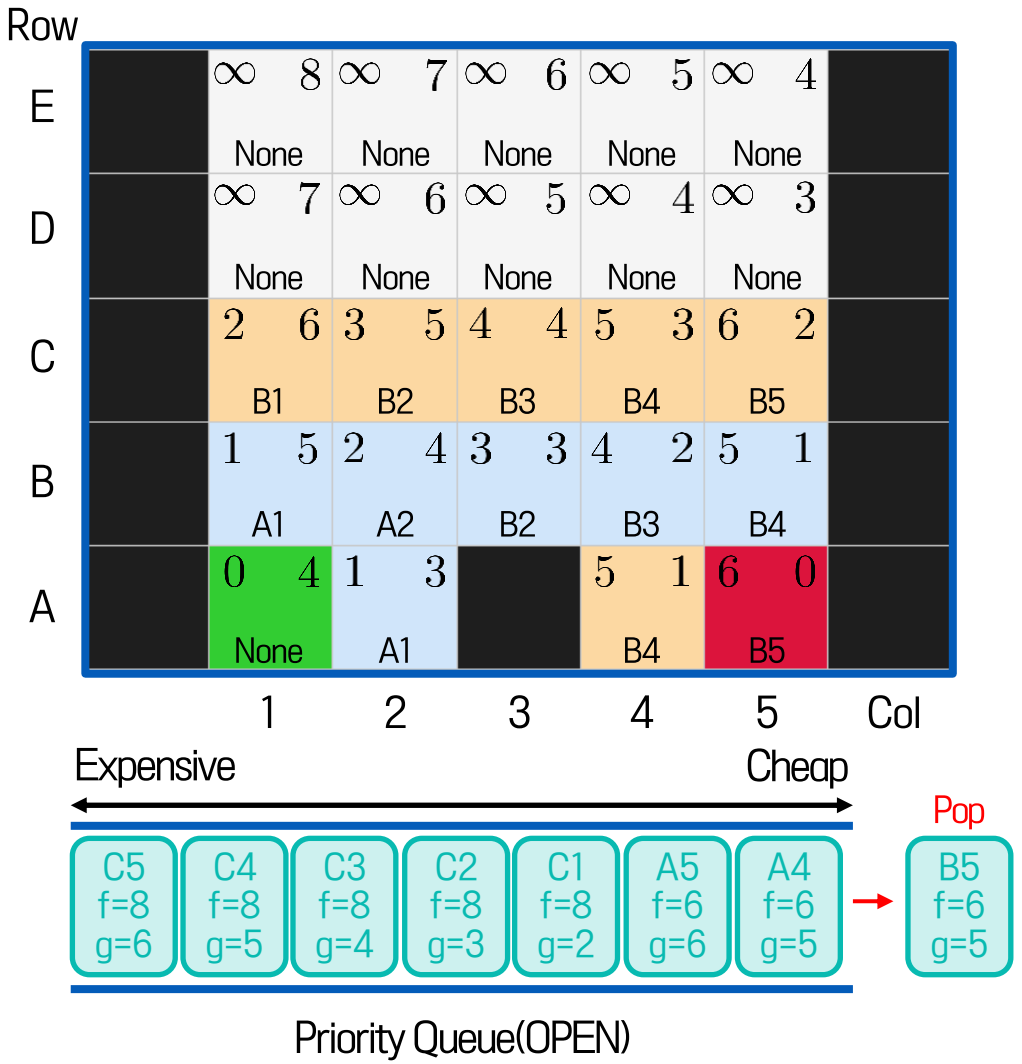


03 A* Algorithm

Example of A* Algorithm

Step 7

- g(u), u ← pop(OPEN) ▶ pop(OPEN) == (5, B5)
- if u == goal : ▶ X
- return reconstruct(parent, goal)
- CLOSED.add(u)
-
- for v in neighbors(u) : ▶ C5, A5, B4
- if v is in CLOSED : continue ▶ B4
- new_g ← g(u) + w(u, v) ▶ C5(6), A5(6)
- if new_g < g(v) :
- g(v) ← new_g
- f(v) ← g(v) + h(v) ▶ C5(8), A5(6)
- parent[v] ← u ▶ C5(B5), A5(B5)
- if v not in OPEN :
- push(OPEN, v) ▶ push C5, A5

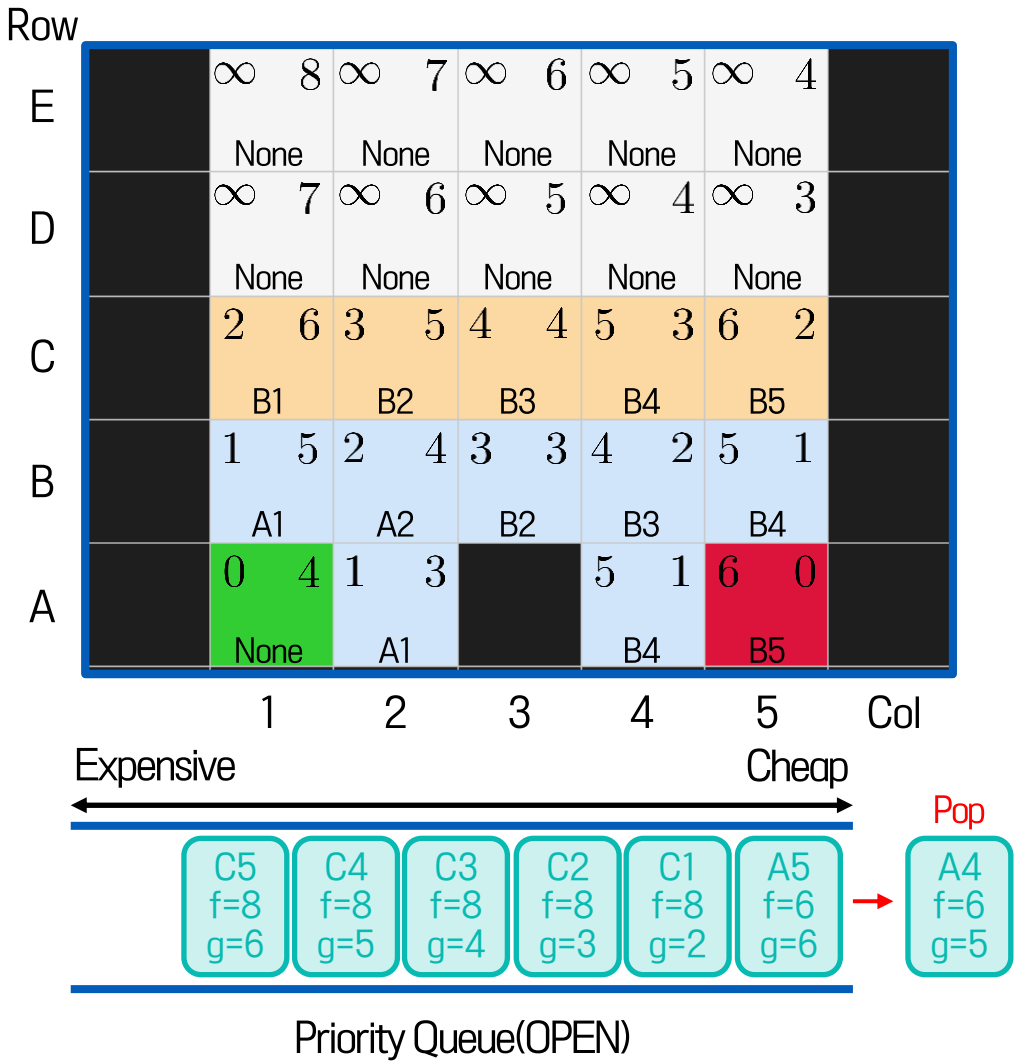


03 A* Algorithm

Example of A* Algorithm

Step 8

- g(u), u ← pop(OPEN) ▶ pop(OPEN) == (5, A4)
- if u == goal : ▶ X
- return reconstruct(parent, goal)
- CLOSED.add(u)
-
- for v in neighbors(u) : ▶ B4, A5
- if v is in CLOSED : continue ▶ B4
- new_g ← g(u) + w(u, v)
- if new_g < g(v) :
- g(v) ← new_g
- f(v) ← g(v) + h(v)
- parent[v] ← u
- if v not in OPEN : ▶ A5 is already in OPEN
- push(OPEN, v)



03 A* Algorithm

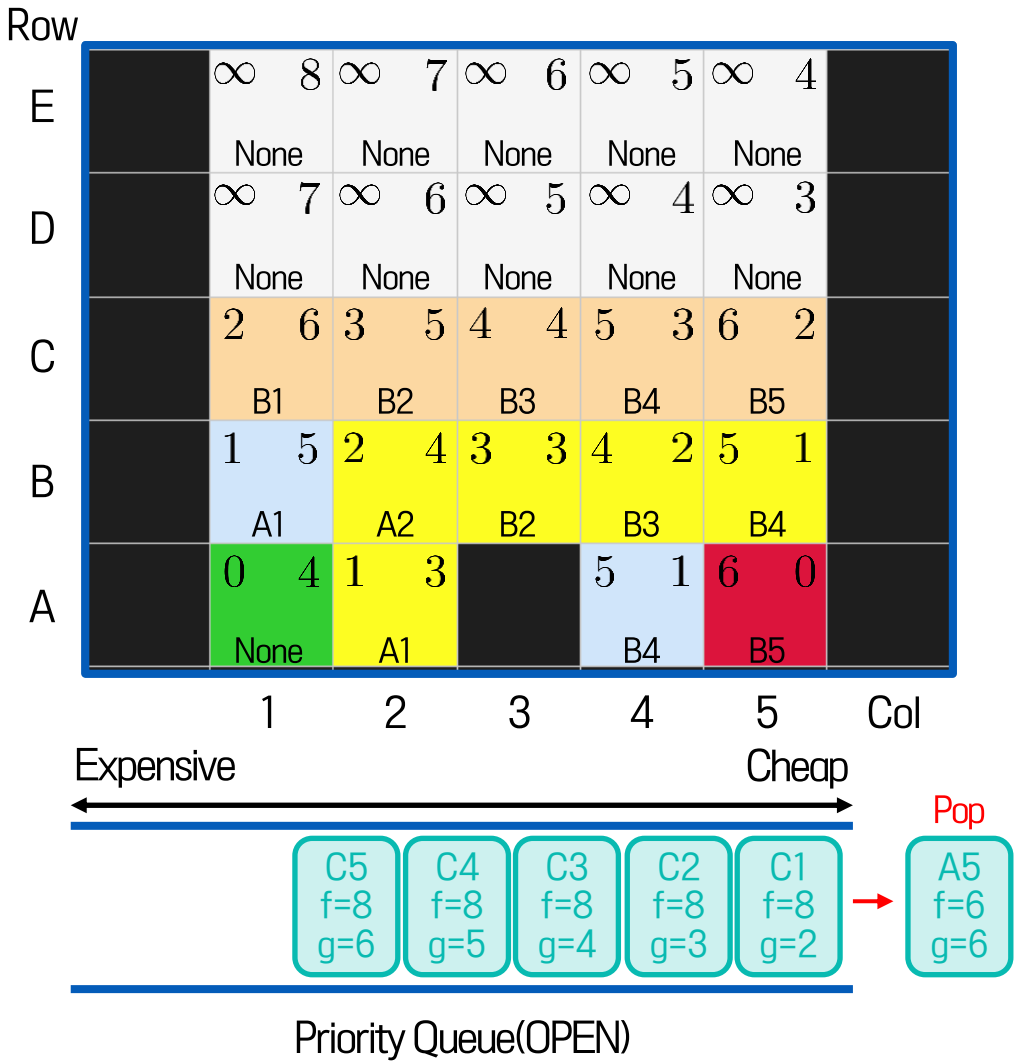
Example of A* Algorithm

Step 9

- g(u), u ← pop(OPEN) ▶ pop(OPEN) == (6, A5)
- if u == goal : ▶ A5 == goal → O
- return reconstruct(parent, goal)

parent[A5] == B5 ▶ A5 == Goal
parent[B5] == B4
parent[B4] == B3
parent[B3] == B2
parent[B2] == A2
parent[A2] == A1 ▶ A1 == Start

∴ cost-optimal path : A1 → A2 → B2 → B3 → B4 → B5 → A5



03 A* Algorithm

Heuristic Quality

- Let two heuristics h_1, h_2 be both admissible.
- If $h_2(n) \geq h_1(n) \quad \forall n$ and strictly larger than for at least one node, we say that h_2 is more informative or h_2 dominates h_1.

  $\Rightarrow$ A* with h_2 will never expand more nodes than A* with h_1 (and usually fewer).

- Example
  - $h_0(n) = 0$ (Dijkstra)
  - $h_1(n) =$ Manhattan distance (A*)

  $\Rightarrow \quad h_1(n) \geq h_0(n) \quad \Rightarrow$ A* with Manhattan will always expand no more nodes than Dijkstra.

- Proof :
  - For same node n, $f_1(n) = g(n) + h_1(n)$, $f_2(n) = g(n) + h_2(n)$, and $f_2(n) \geq f_1(n) \quad \forall n$.
  - Suppose A_2^* pops node n, and let C^* be the optimal path cost. Then, such a node satisfies $f_2(n) \leq C^*$ typically.
  - If $f_2(n) < C^*$, then $f_1(n) \leq f_2(n) < C^*$. By the Lemma, node n with $f_1(n) < C^*$ must be popped by A_1^*.
  - Hence, over the region $f < C^*$, all nodes expanded by A_2^* form a subset of those expanded by A_1^*. ■

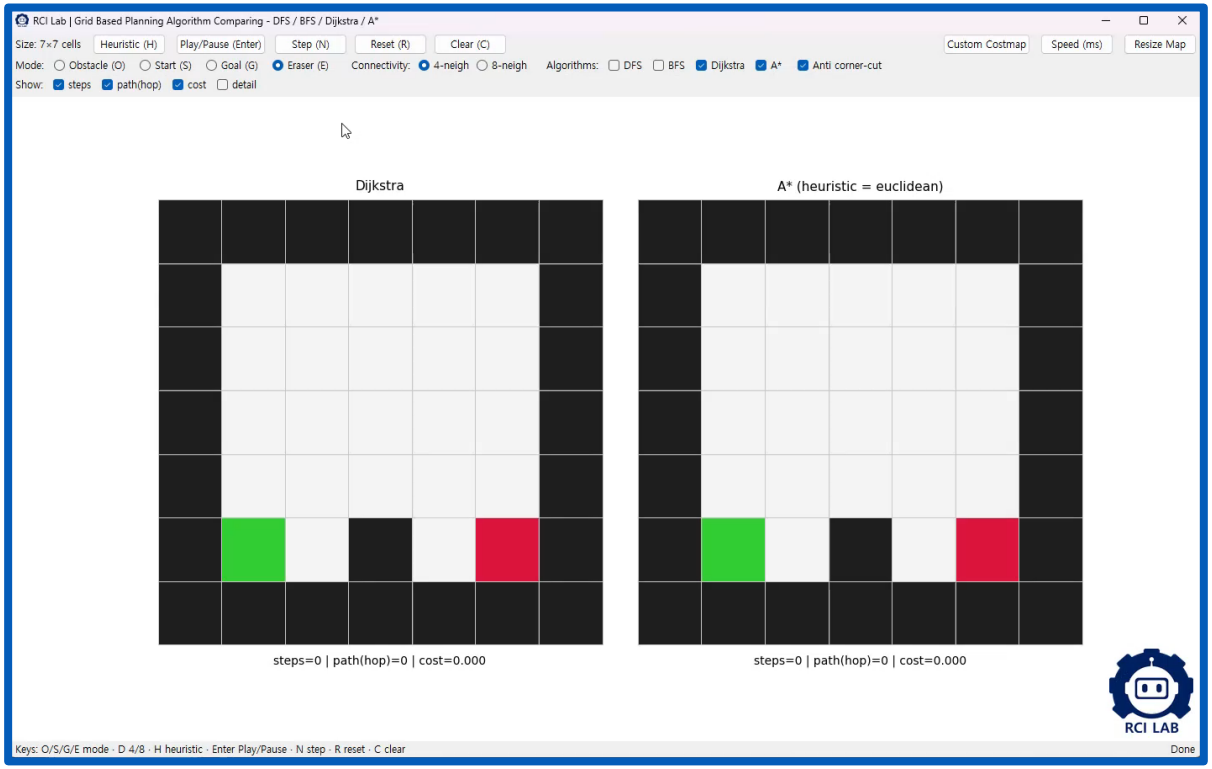
  ※ Nodes with $f > C^*$ are not required for optimality.

03 A* Algorithm

Heuristic Quality

- Example
  - $h_0(n) = 0$ (Dijkstra)
  - $h_1(n) =$ Manhattan distance (A*)

$\Rightarrow \quad h_1(n) \geq h_0(n) \quad \Rightarrow$ A* with Manhattan will always expand no more nodes than Dijkstra.

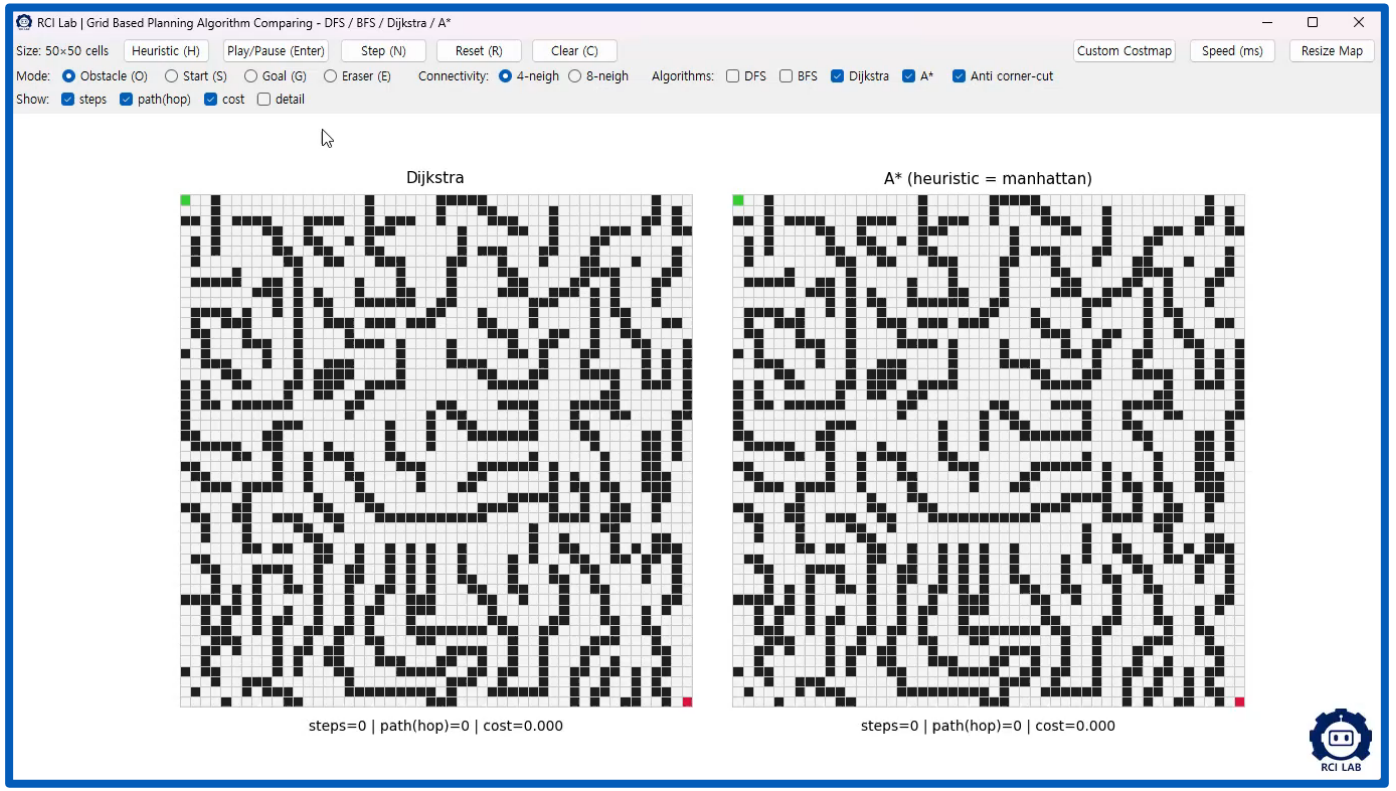


03 A* Algorithm

Heuristic Quality

- Example
  - $h_0(n) = 0$ (Dijkstra)
  - $h_1(n) =$ Manhattan distance (A*)

$\Rightarrow \quad h_1(n) \geq h_0(n) \quad \Rightarrow$ A* with Manhattan will always expand no more nodes than Dijkstra.



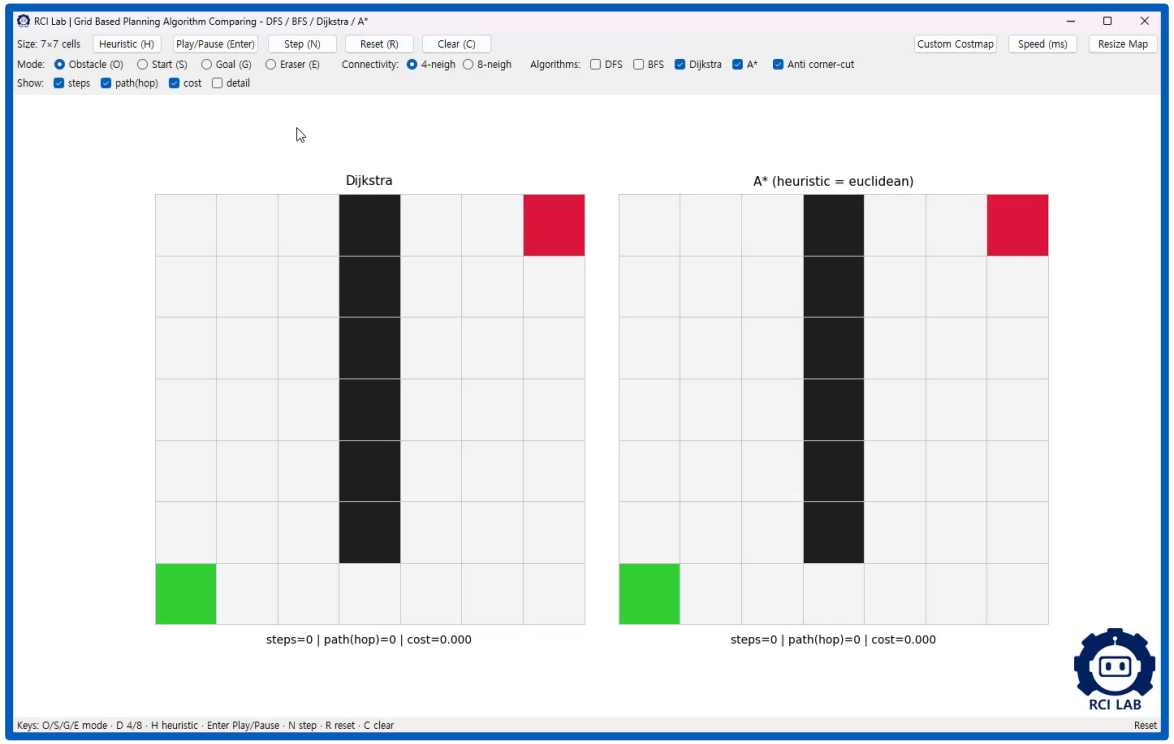
03 A* Algorithm

Heuristic Quality

- Example
  - $h_0(n) = 0$ (Dijkstra)
  - $h_1(n) =$ Manhattan distance (A*)

$\Rightarrow \quad h_1(n) \geq h_0(n) \quad \Rightarrow$ A* with Manhattan will always expand no more nodes than Dijkstra.

not always $h_2 > h_1$



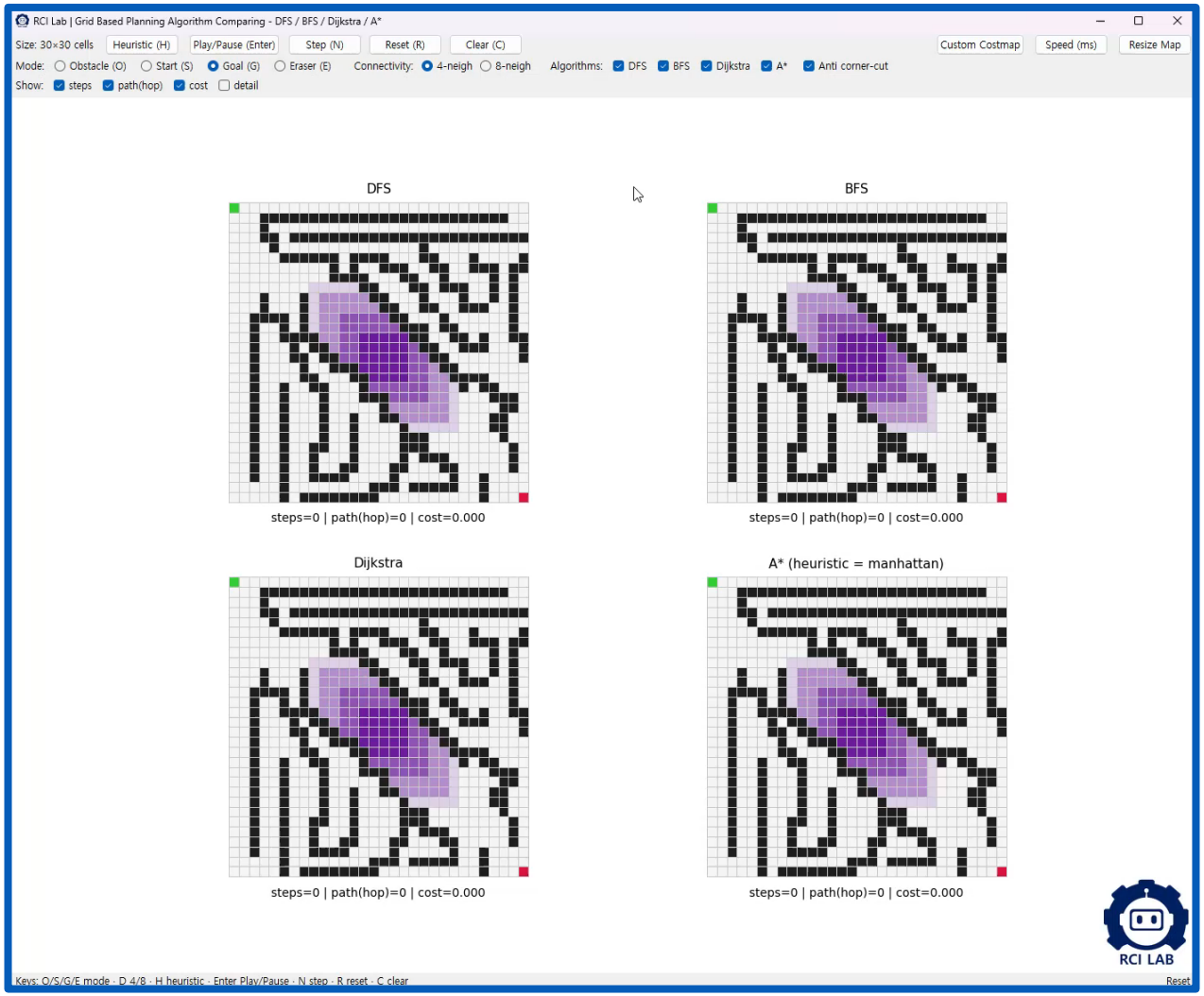
03 A* Algorithm

Summary

Algorithm / Category	DFS	BFS	Dijkstra	A*
Data structure	Stack(LIFO)	Queue(FIFO)	Priority queue on $g(n)$	Priority queue on $f(n) = g(n) + h(n)$
Edge cost conditions	Irrelevant (does not use edge cost)	All edges have identical cost	Non-negative costs	Non-negative costs, admissible & consistent h
Optimality	No optimality (neither shortest hops nor minimum cost)	Shortest path in terms of number of hops (edges)	Cost-optimal shortest path	Same cost-optimal path as Dijkstra
Search pattern	Dives deep, then backtracks when blocked.	Expands node in layers from the start (wavefront)	Grows iso-cost shells around the start	Search is biased toward the goal (with avoiding many unnecessary expansions)
Uses heuristic	No	No	No	Yes (distance-based)
Typical use cases	Reachability, Checking connected components	Unweighted shortest paths(or mazes)	General weighted shortest paths (maps, networks)	Grid-based path planning(nav2), Robot navigation, Game AI pathfinding

03 A* Algorithm

Summary – Comparison of All Introduced Algorithms



Thank you