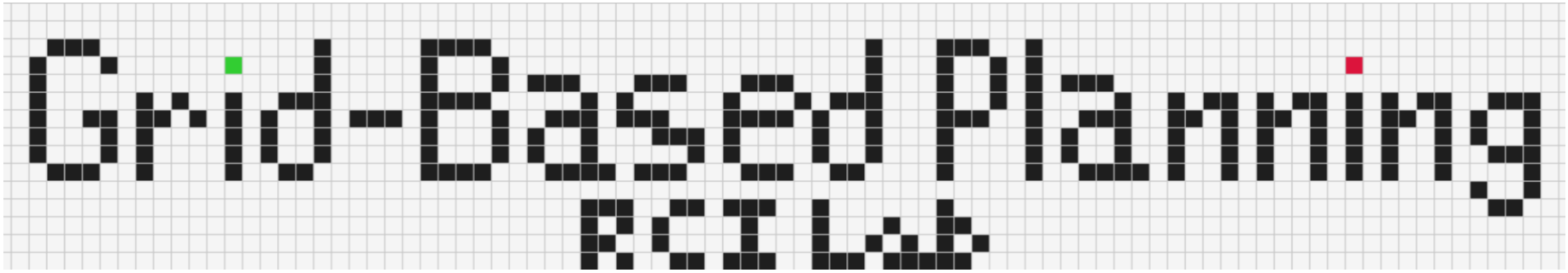
Algorithms - Dijkstra



Speaker: Taehyun Jung

RCI LAB

KHU

Table of Contents

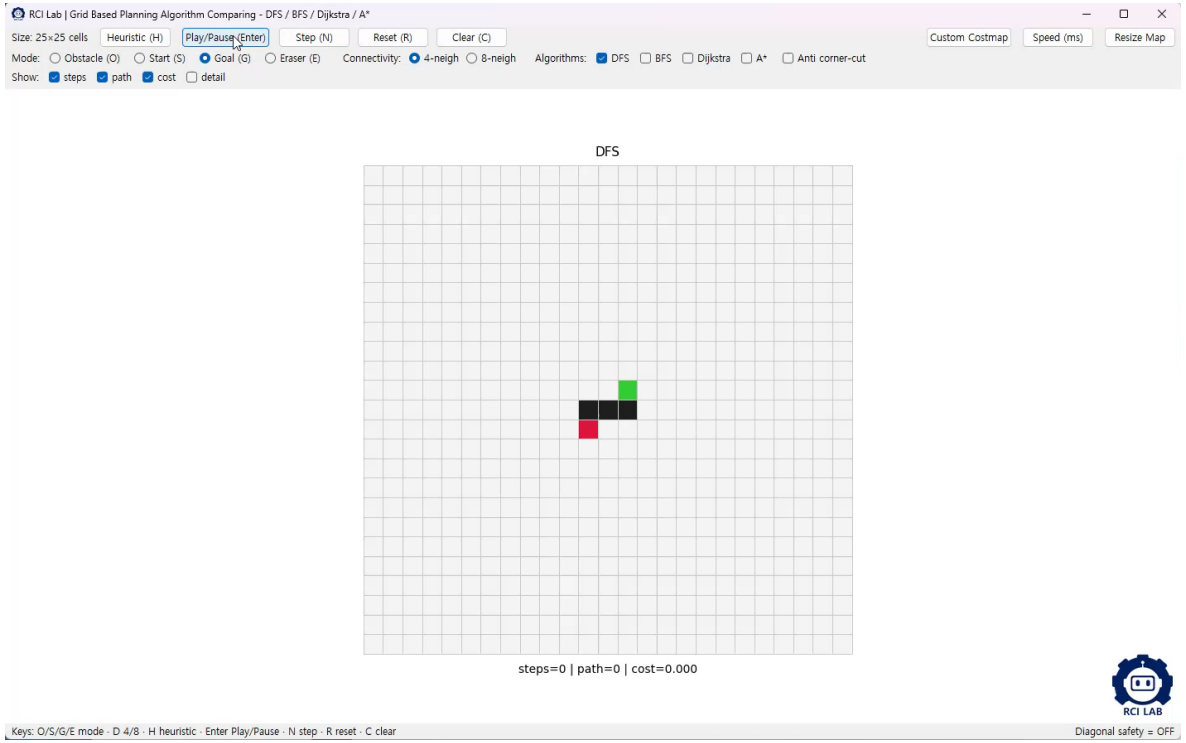
1. Limitations of DFS, BFS Algorithms
2. Priority Queue
3. Dijkstra Algorithm

01 Limitations of DFS, BFS Algorithms

01 Limitations of DFS, BFS Algorithms

Limitations of DFS Algorithm

- DFS often misses shorter routes. It can return a path, but not the shortest. → Not optimal!
- Path heavily depends on neighbor/push(LIFO) order.
- DFS ignores costs. Decisions are driven solely by neighbor/push(LIFO) order.

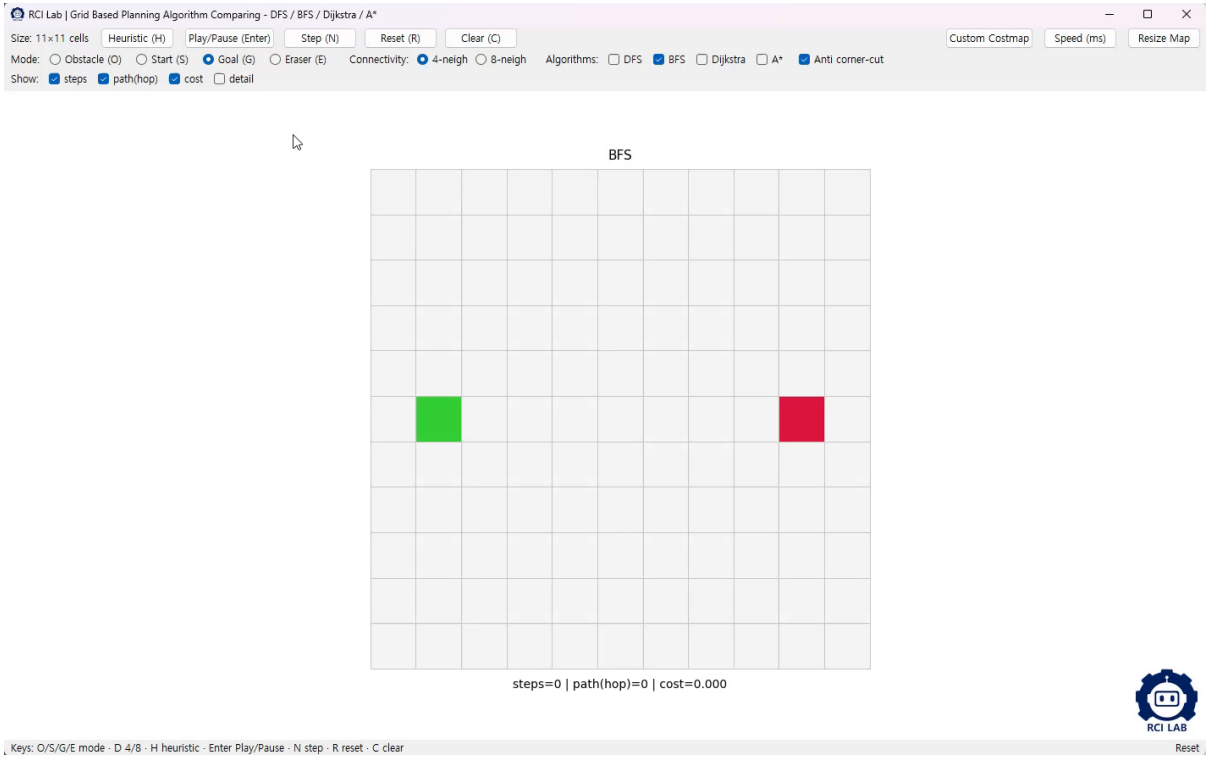


→ DFS can find a path, but does not guaranteed shortest or optimal.

01 Limitations of DFS, BFS Algorithms

Limitations of BFS Algorithm

- With $\sqrt{2}$ diagonals or costmaps, fewer steps can still cost more. → Not cost-optimal!
- Level expansion can create very large frontiers in open spaces. → Memory blow-up
- BFS must traverse all shallower(low-level) layers first. → Inefficient for distant goals



BFS guarantees hop-shortest path only under unit costs.

BFS can't avoid the higher cost cells.

01 Limitations of DFS, BFS Algorithms

Problem

- A Planner must consider cell/edge costs to find a cost-optimal path.
  → This requires using not only the visit order but also the cumulative path cost.
- However, Stack(LIFO) and Queue(FIFO) are order-only containers.
- They cannot prioritize by cost, so DFS/BFS fail to find a cost-optimal path under non-uniform costs.

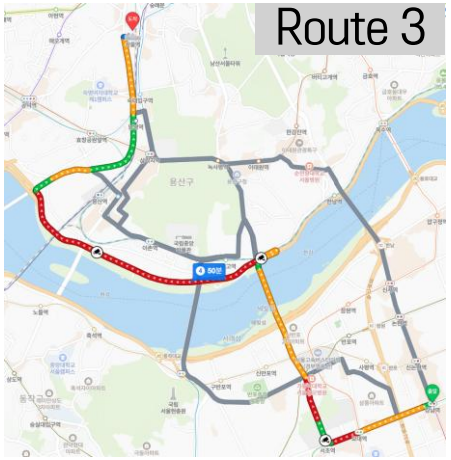
01 Limitations of DFS, BFS Algorithms

Problem

- Even with the same start and goal, there can be many routes, each with a different cost.

(e.g., time, distance, tolls)

Gangnam Station → Seoul Station

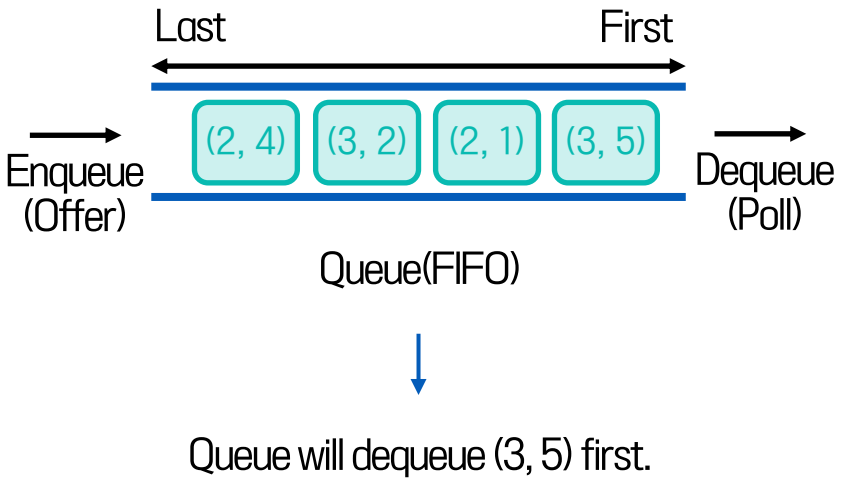


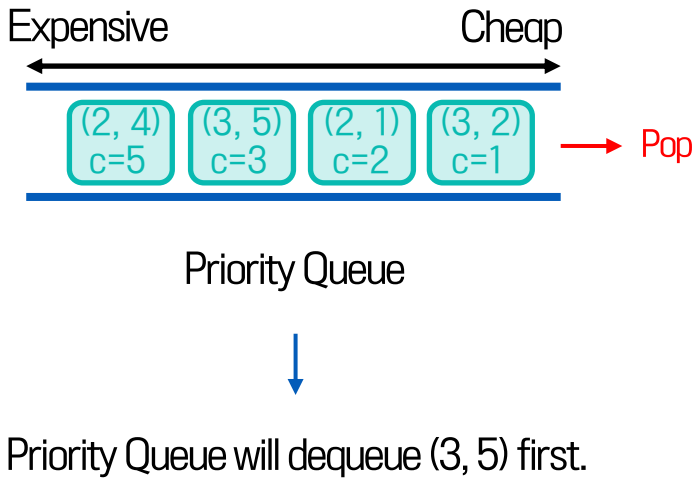
→ Route 1 is a cost-optimal path!

01 Limitations of DFS, BFS Algorithms

Solution

- We need an algorithm whose frontier is ordered by cumulative cost g, not just by insertion order.
  1. Cheapest-first principle: grow the frontier in non-decreasing cumulative cost g.
  2. Replace Queue(FIFO) with a min-Priority Queue keyed by g
     (Finalize on pop/dequeue and stop when the goal is popped/dequeued.



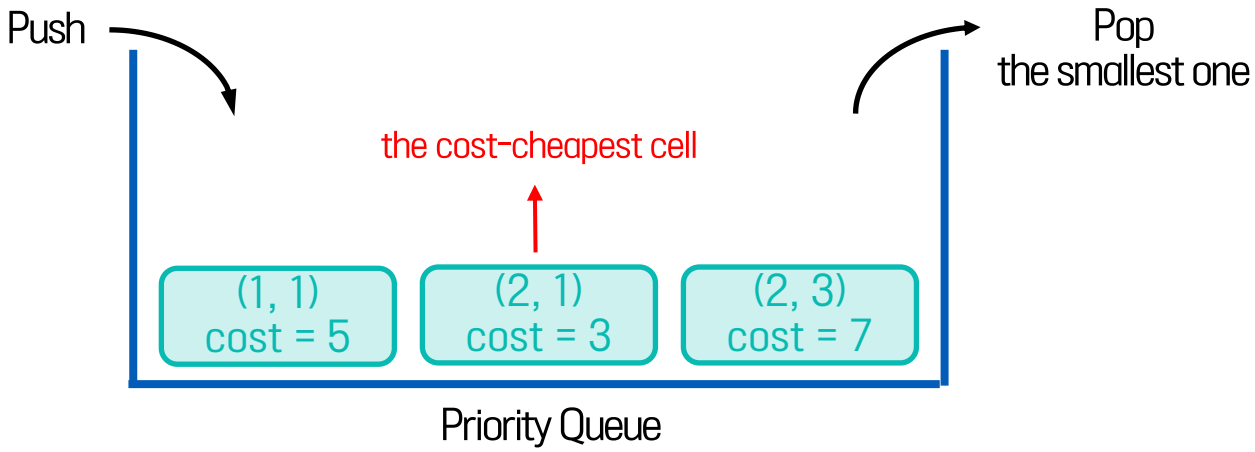


02 Priority Queue

02 Priority Queue

Definition of Priority Queue

A container that keeps items with a priority(key) and always pops the smallest key first.



Pop the cheapest g → Check Goal & Stop → Relax neighbors → Push improved ones

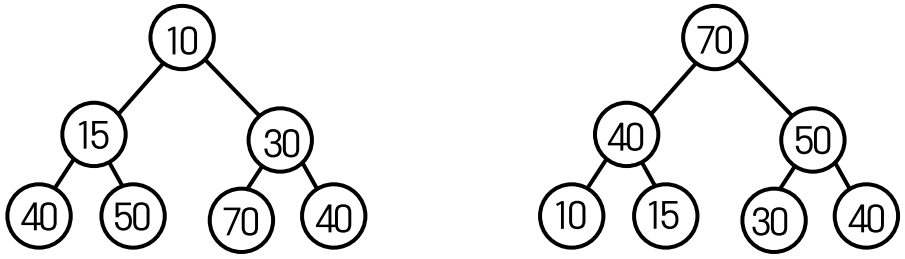
For each neighbor, compute $\mathrm{new\_}g$
If cheaper, update parent & g

02 Priority Queue

Heap

- Heap is a complete binary tree with a specific rule, which is designed to make the operation fast to find the maximum or minimum values.
- The property of Heap:
  1. If node A is a parent of node B, then there is an ordering relation between the key of A and the key of B.
  2. In this case, the ordering of keys is guaranteed only between a parent and its children.
  3. Note that no ordering is defined between sibling nodes.

Heap Data Structure



Min Heap Max Heap

02 Priority Queue

Heap in Python

- In Python, there is no separate heap type. Ordinary lists are used to represent heaps.
- By default, a heap in Python(via `heapq`) is a min-heap.
- To use heap operations, you must import the module with `import heapq`.
- Basic heap operations (`heapq`):
  1. `heapq.heappush(heap, item)` - push a new item onto the heap.
  2. `heapq.heappop(item)` - pop and return the smallest item from the heap.
  3. `heapq.heapify(list)` - transform a list into a heap in-place.
  4. `heapq.heappushpop(heap, item)` - push then pop the smallest item in one step.
  5. `heapq.heapreplace(heap, item)` - pop the smallest item and then push a new item.
- In Python, `heapq` commonly stores items as tuples with the priority(key) first.
- In this lecture, we use items of the form $(g, \text{row}, \text{column})$.

03 Dijkstra Algorithm

03 Dijkstra Algorithm

Dijkstra Algorithm

Dijkstra algorithm is a shortest-path algorithm that efficiently computes the minimum distance from a single source vertex to all other vertices in a graph with non-negative edge weights.

It repeatedly selects the node with the smallest tentative distance and relaxes its outgoing edges using a priority queue.

Main Idea

- 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.

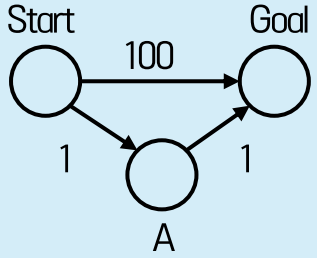
※ If all cost of the cells is 1, Dijkstra ≡ BFS (special case).

03 Dijkstra Algorithm

Main Idea

- 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).



Pop the cheapest g › Check Goal & Stop › Relax neighbors › Push improved ones

For each neighbor, compute $\mathrm{new\_}g$
If cheaper, update parent & g

03 Dijkstra Algorithm

Pseudocode

```
1)  S ← start, V = all nodes
2)  PQ ← min-priority-queue
3)  visited ← ∅
4)
5)  for v in V:
6)          g[v] ← ∞
7)          parent[v] ← None
8)
9)  g[S] ← 0
10) push(PQ, (0, S))
11) while PQ is not empty:
12)         (g_u, u) ← pop(PQ)              ▶ pop minimum g
13)         if u in visited:
14)                 continue
15)         visited.add(u)
16)
17)         if u == goal:                   ▶ check goal after popping
18)                 return recontstruct(parent, goal)
19)
20)         for each (u→v) in outgoing(u):
21)                 new_g ← g_u + w(u, v)
22)                 if new_g < g[v]:        ▶ edge relaxation
23)                         g[v] ← new_g
24)                         parent[v] ← u
25)                         push(PQ, (new_g, v))
26)
27)  return (g, parent)
```

u : the current node popped from the PQ.
v : a neighbor(outgoing) of u reached by an edge u→v.
g[u] : the best-known cumulative cost from the start to u.

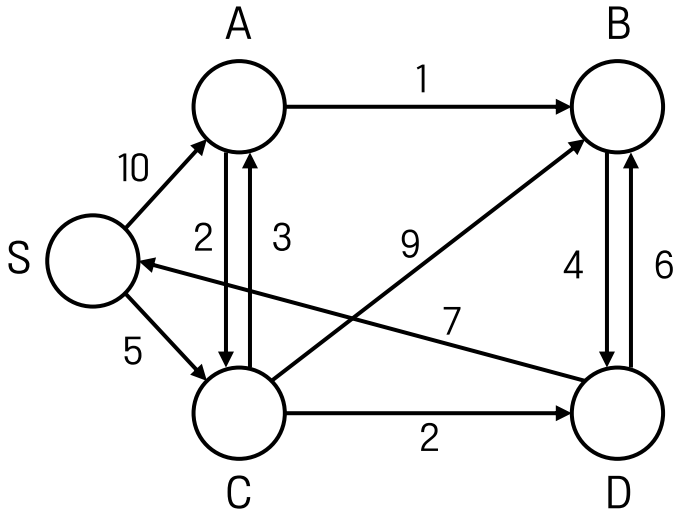
g_u : the popped cost that comes with u from the queue.
new_g : the tentative cost via u to v.
outgoing(u): the set of outgoing edges from u.

03 Dijkstra Algorithm

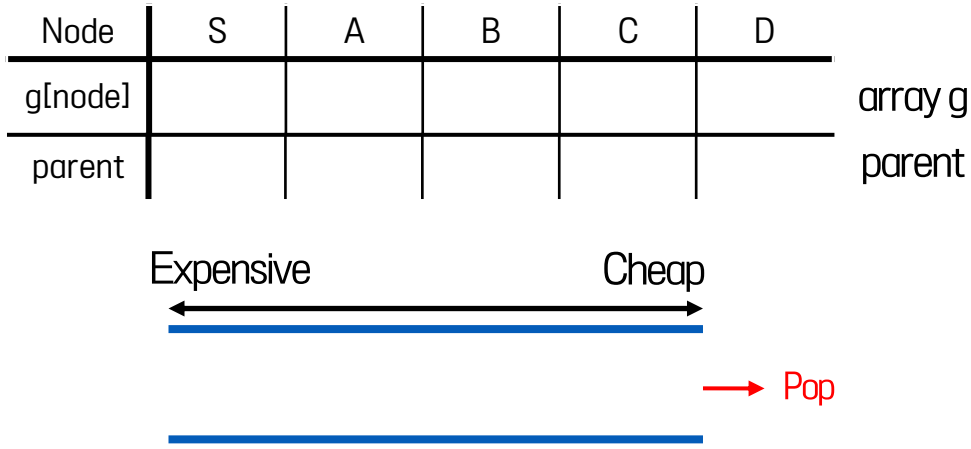
Example of Dijkstra Algorithm – Node Graph (weighted)

- Start node : S
- Goal node : B

- There is only non-negative edge cost.



Node Graph (weighted)



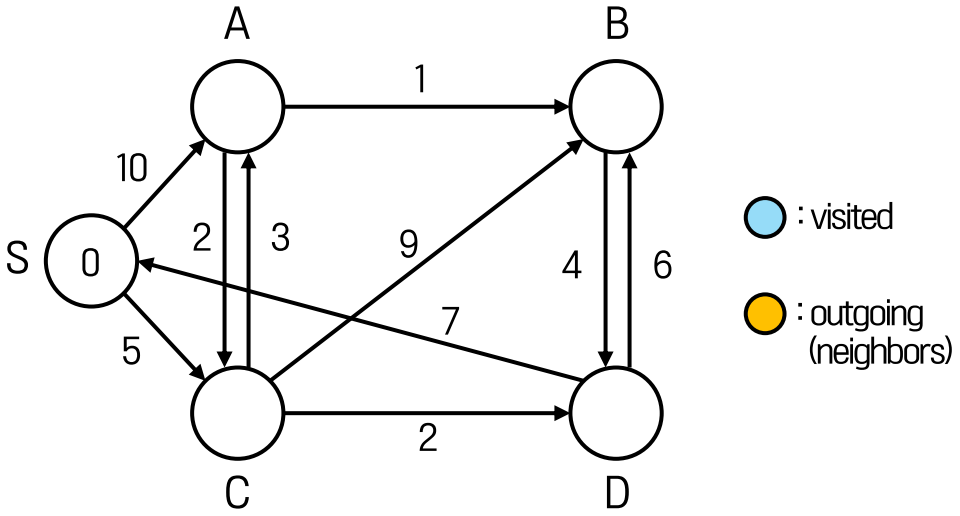
Priority Queue(PQ)

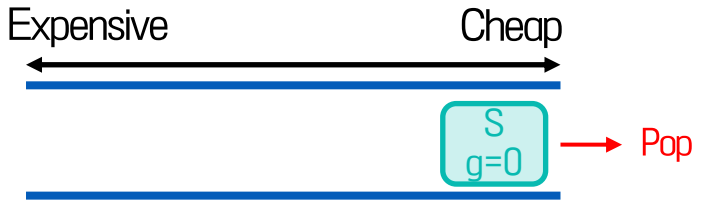
03 Dijkstra Algorithm

Example of Dijkstra Algorithm – Node Graph (weighted)

Step 0 - Initialize

- PQ ← S
- V ← {S, A, B, C, D}
- for v in V:
  - g[v] ← ∞
  - parent[v] ← None
- g[S] ← 0
- push(PQ, (g[S], S)) ▶ g[S] == 0





Node	S	A	B	C	D
g[node]	0	∞	∞	∞	∞
parent	None	None	None	None	None

03 Dijkstra Algorithm

Example of Dijkstra Algorithm – Node Graph (weighted)

Step 1

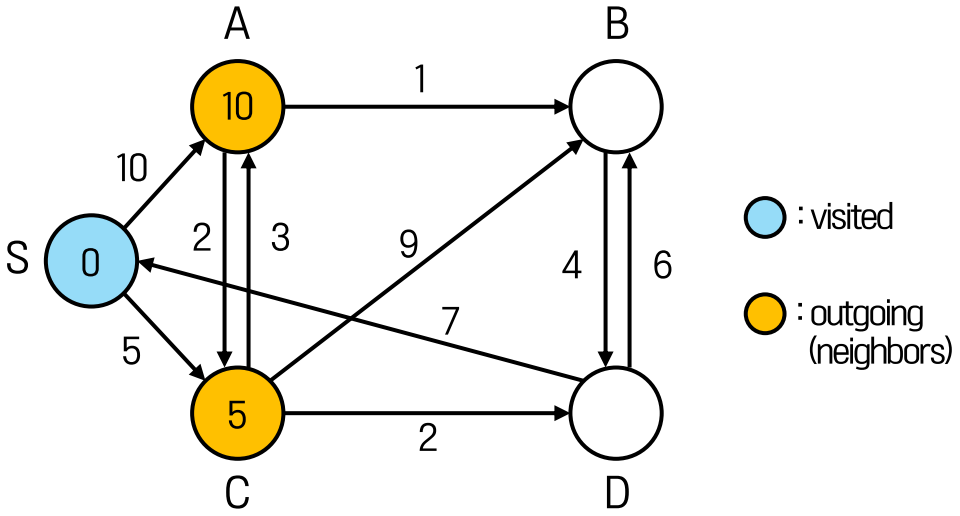
- g_u, u ← pop(PQ) ▶ pop(PQ) == (0, S)
- visited.add(u)
- for each (u→v) in outgoing(u):
  - new_g ← g_u + w(u, v)
  - if new_g < g[v]:
    - g[v] ← new_g
    - parent[v] ← u ▶ S = parent of A, C
    - push(PQ, (new_g, v))

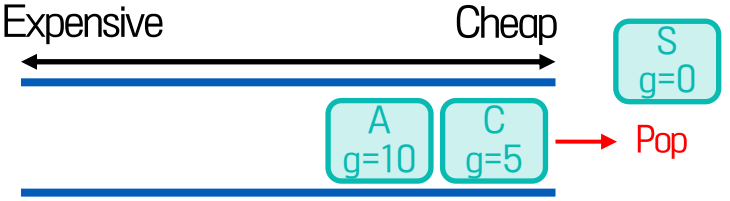
※ outgoing(u) == [A, C]

g[A] : ∞ → 10

g[C] : ∞ → 5

} edge relaxation





Node	S	A	B	C	D
g[node]	0	10	∞	5	∞
parent	None	S	None	S	None

03 Dijkstra Algorithm

Example of Dijkstra Algorithm – Node Graph (weighted)

Step 2

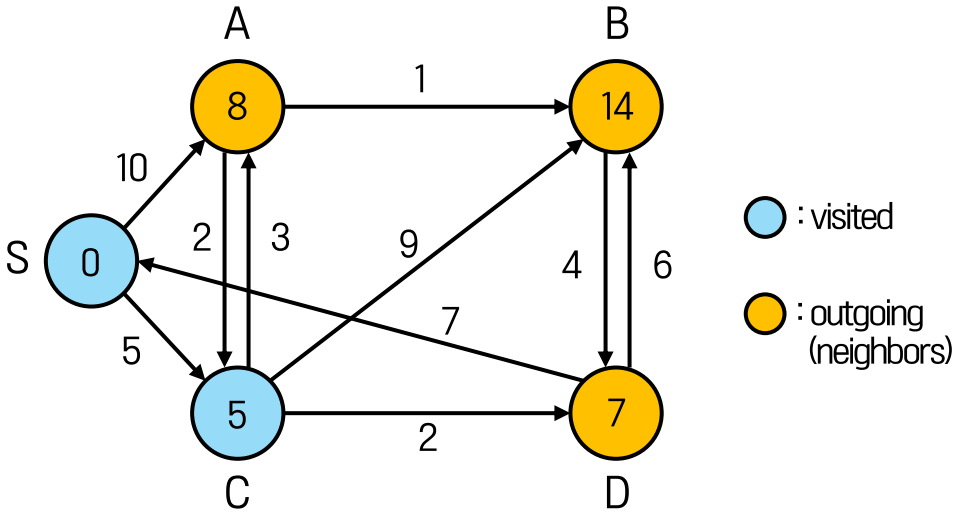
- g_u, u ← pop(PQ) ▶ pop(PQ) == (5, C)
- visited.add(u)
- for each (u→v) in outgoing(u):
  - new_g ← g_u + w(u, v)
  - if new_g < g[v]:
    - g[v] ← new_g
    - parent[v] ← u ▶ C = parent of A, B, D
    - push(PQ, (new_g, v))

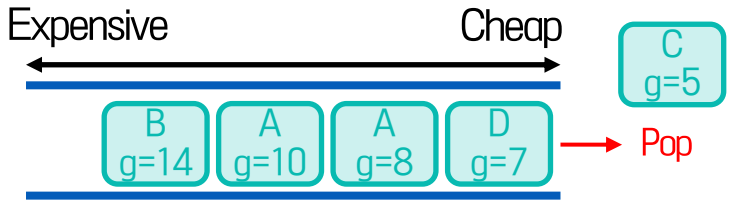
※ outgoing(u) == [A, B, D]

g[A] : 10 → 8

g[B] : ∞ → 14

g[D] : ∞ → 7





Node	S	A	B	C	D
g[node]	0	8	14	5	7
parent	None	C	C	S	C

03 Dijkstra Algorithm

Example of Dijkstra Algorithm – Node Graph (weighted)

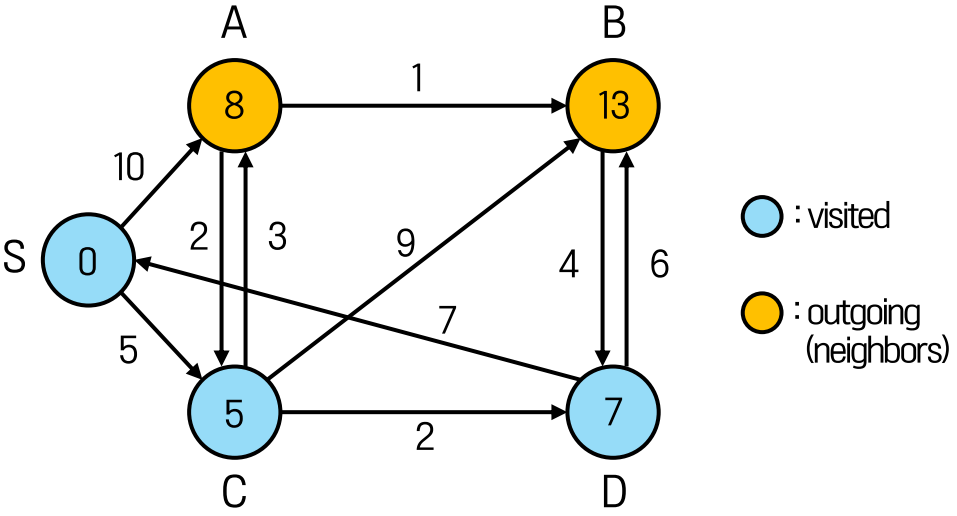
Step 3

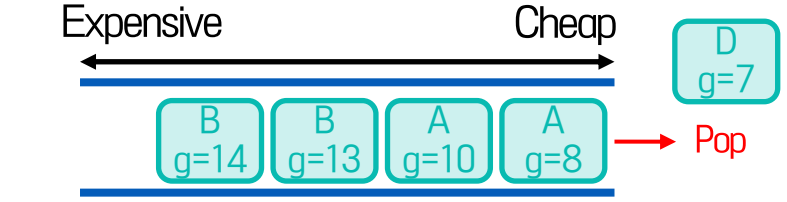
- g_u, u ← pop(PQ) ▶ pop(PQ) == (7, D)
- visited.add(u)
- for each (u→v) in outgoing(u):
  - new_g ← g_u + w(u, v)
  - if new_g < g[v]:
    - g[v] ← new_g
    - parent[v] ← u ▶ D = parent of B
    - push(PQ, (new_g, v))

※ outgoing(u) == [S, B]

g[S] : not changed ▶ new_g == 14 > g[S] == 0

g[B] : 14 → 13





Node	S	A	B	C	D
g[node]	0	8	13	5	7
parent	None	C	D	S	C

03 Dijkstra Algorithm

Example of Dijkstra Algorithm – Node Graph (weighted)

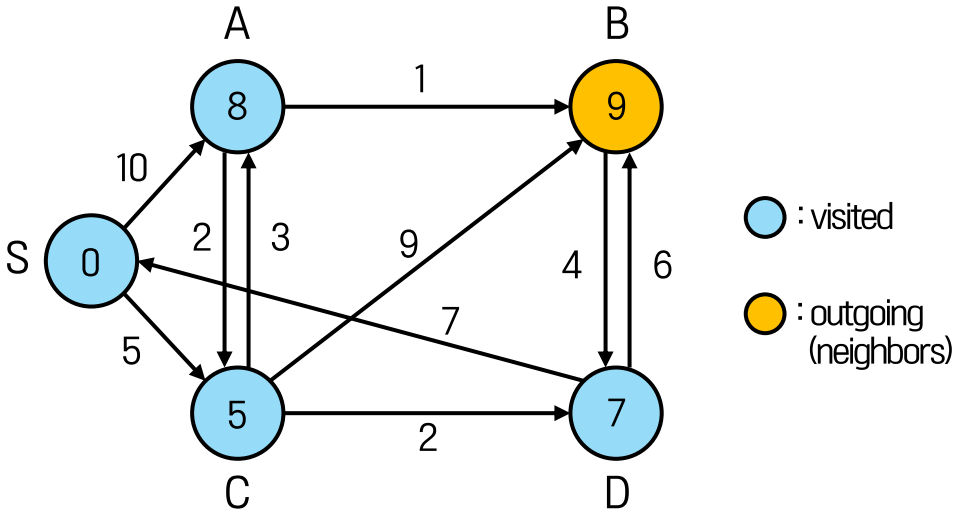
Step 4

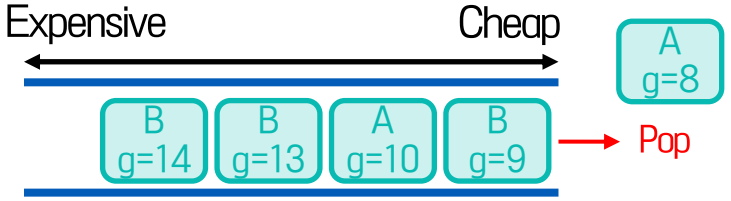
- g_u, u ← pop(PQ) ▶ pop(PQ) == (8, A)
- visited.add(u)
- for each (u→v) in outgoing(u):
  - new_g ← g_u + w(u, v)
  - if new_g < g[v]:
    - g[v] ← new_g
    - parent[v] ← u ▶ A = parent of B
    - push(PQ, (new_g, v))

※ outgoing(u) == [B, C]

g[B] : 13 → 9

g[C] : not changed ▶ new_g == 10 > g[C] == 5





Node	S	A	B	C	D
g[node]	0	8	9	5	7
parent	None	C	A	S	C

03 Dijkstra Algorithm

Example of Dijkstra Algorithm – Node Graph (weighted)

Step 5

- g_u, u ← pop(PQ) ▶ pop(PQ) == (9, B)
- visited.add(u)
- if u == goal: ▶ B == Goal

  return reconstruct(parent, goal)

parent[B] == A ▶ B == Goal

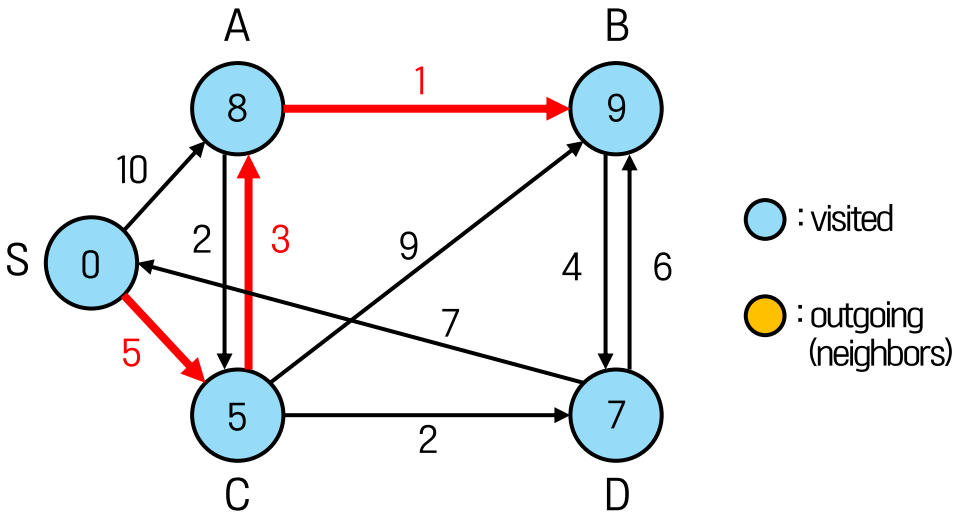
↓

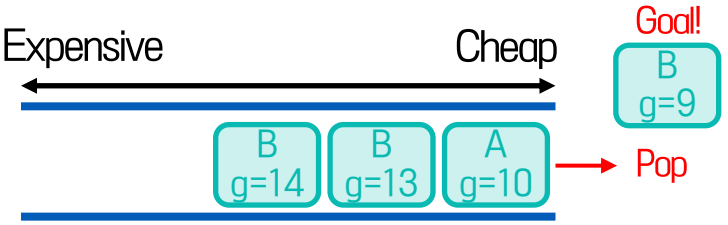
parent[A] == C

↓

parent[C] == S ▶ S == Start

∴ cost-optimal path : S → C → A → B



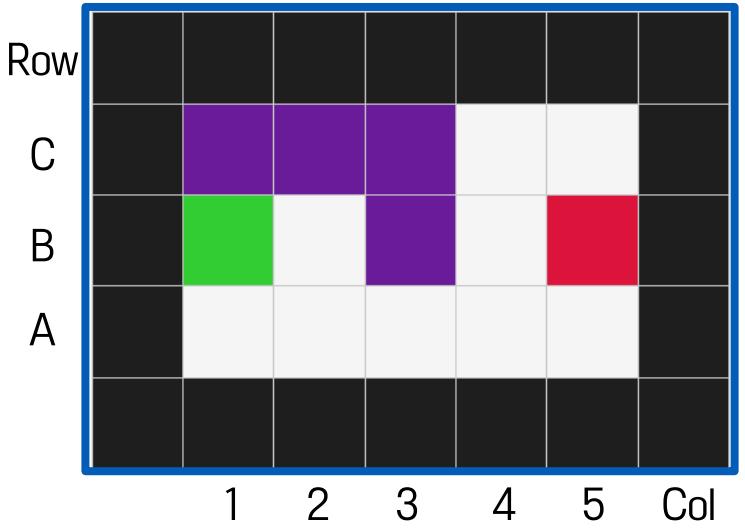


Node	S	A	B	C	D
g[node]	0	8	9	5	7
parent	None	C	A	S	C

03 Dijkstra Algorithm

Example of Dijkstra Algorithm – Gridmap (with costmap)

- Start cell : , Goal cell : , Obstacle cell :
- Each edge(move_cost) : cost = 1 → only straight(4-connectivity)
- Custom cost cell : , cost = 5
- For equal g, use FIFO. → same weight situation ≡ BFS
- If some child nodes have same cost, the order of pushing follows NESW.
  tie tie-breaker

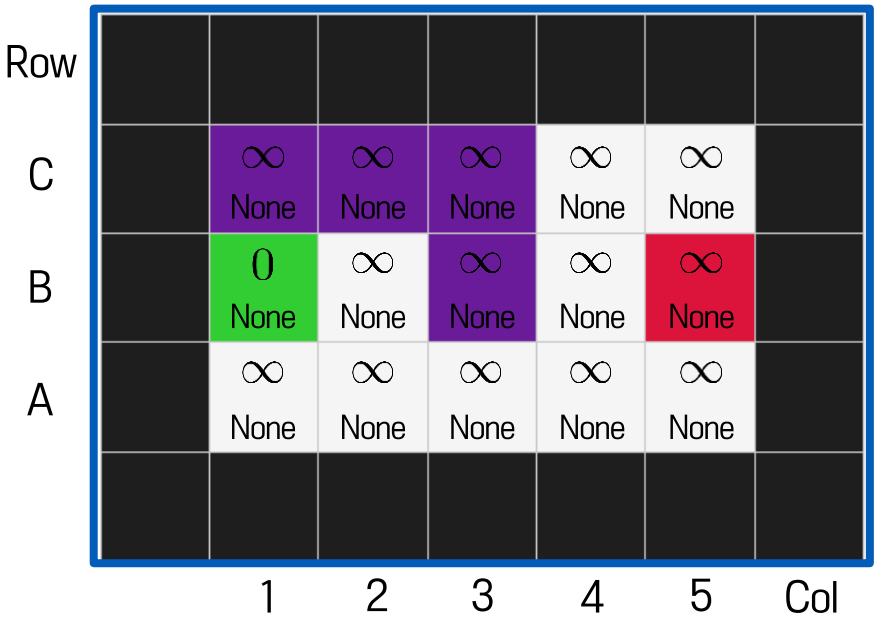


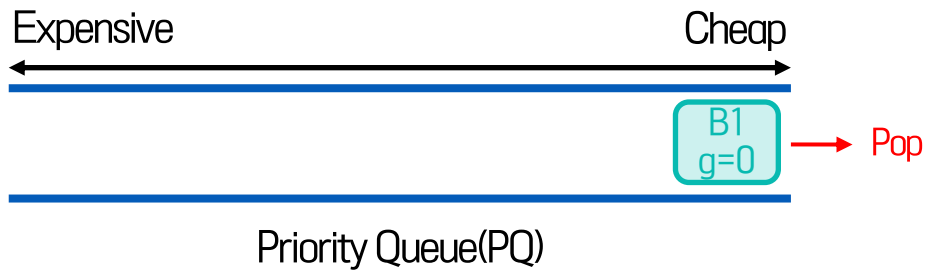
03 Dijkstra Algorithm

Example of Dijkstra Algorithm – Gridmap (with costmap)

Step 0 - Initialize

- PQ ← B1
- V ← all cells
- for v in V:
  - g[v] ← ∞
  - parent[v] ← None
- g[B1] ← 0
- push(PQ, (g[B1], B1)) ▶ g[B1] == 0





03 Dijkstra Algorithm

Example of Dijkstra Algorithm – Gridmap (with costmap)

Step 1

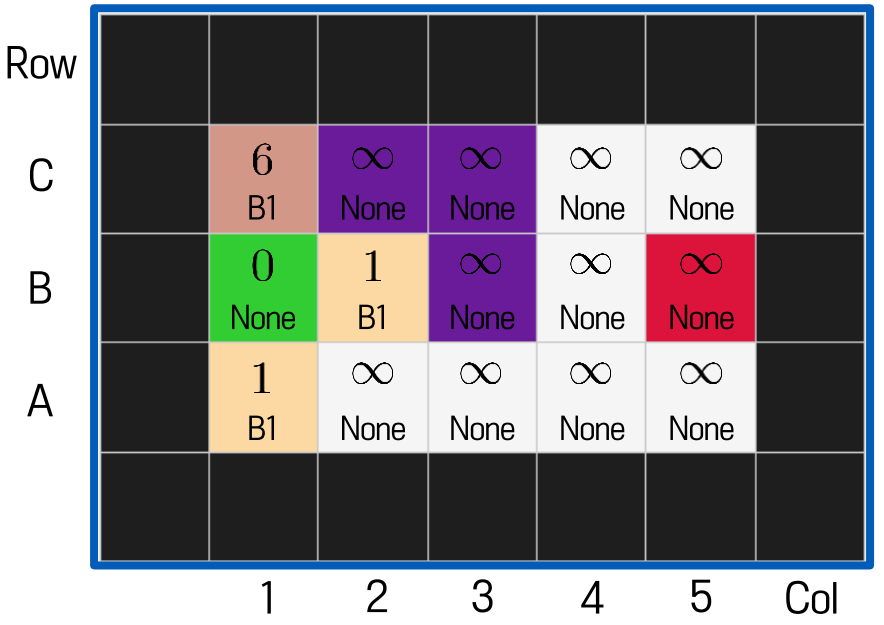
- g_u, u ← pop(PQ) ▶ pop(PQ) == (0, B1)
- visited.add(u)
- for each (u→v) in outgoing(u):
  - new_g ← g_u + w(u, v)
  - if new_g < g[v]:
    - g[v] ← new_g
    - parent[v] ← u ▶ B1 = parent of C1, B2, A1
    - push(PQ, (new_g, v))

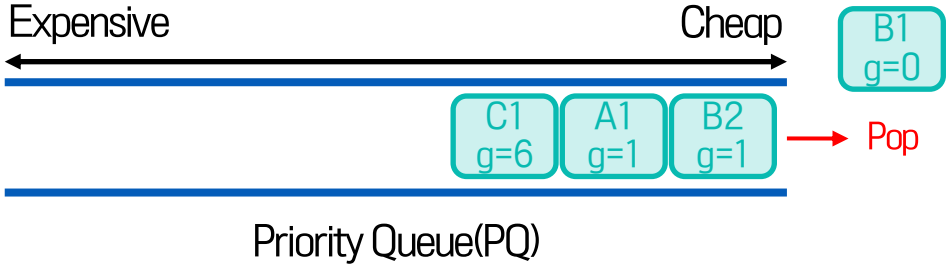
※ outgoing(u) == [C1, B2, A1]

g[C1] : ∞ → 6

g[B2] : ∞ → 1

g[A1] : ∞ → 1





03 Dijkstra Algorithm

Example of Dijkstra Algorithm – Gridmap (with costmap)

Step 2

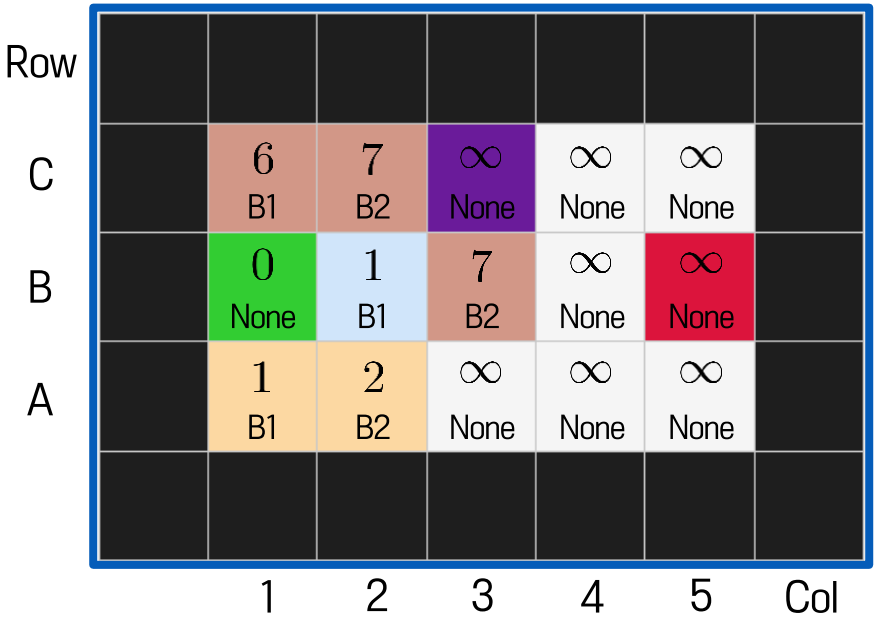
- g_u, u ← pop(PQ) ▶ pop(PQ) == (0, B2)
- visited.add(u)
- for each (u→v) in outgoing(u):
  - new_g ← g_u + w(u, v)
  - if new_g < g[v]:
    - g[v] ← new_g
    - parent[v] ← u ▶ B2 = parent of C2, B3, A2
    - push(PQ, (new_g, v))

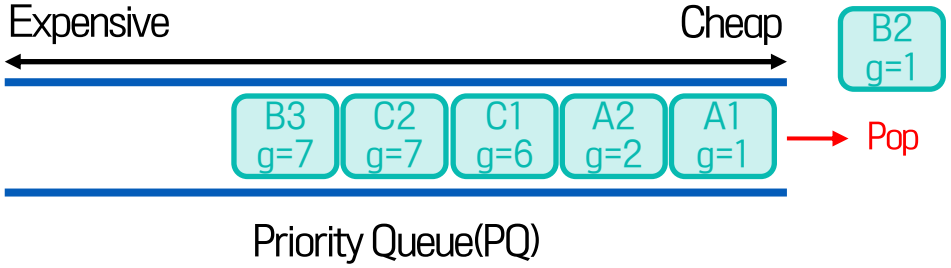
※ outgoing(u) == [C2, B3, A2]

g[C2] : ∞ → 7

g[B3] : ∞ → 7

g[A2] : ∞ → 2





03 Dijkstra Algorithm

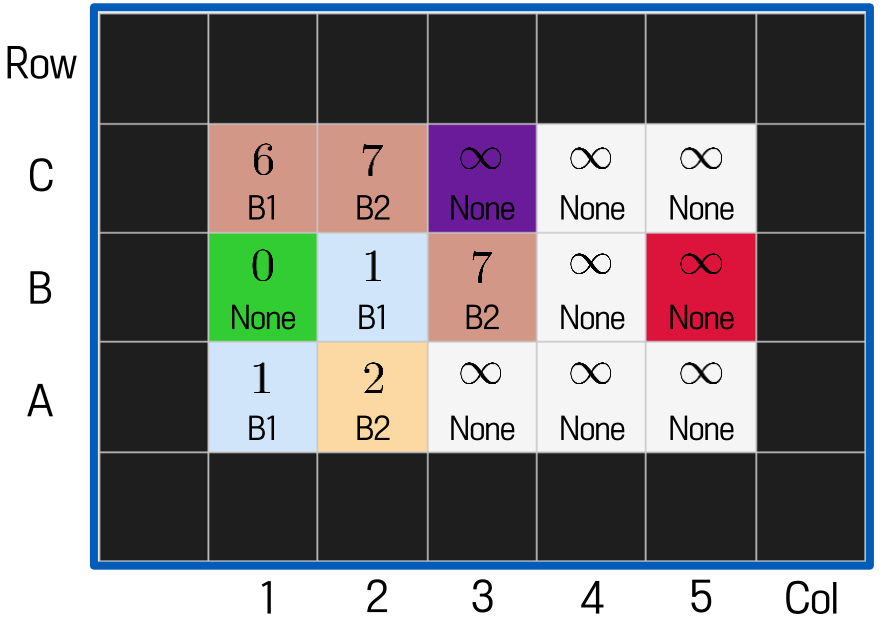
Example of Dijkstra Algorithm – Gridmap (with costmap)

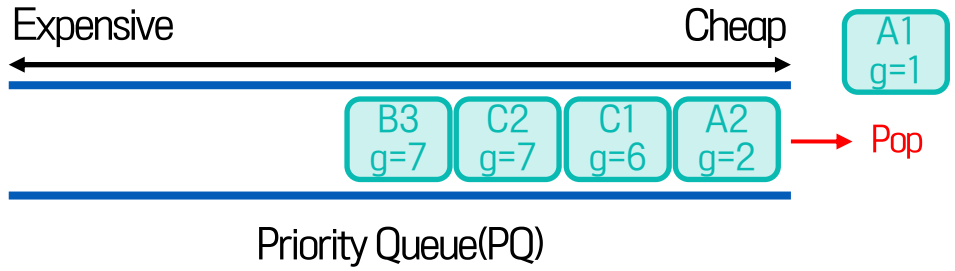
Step 3

- g_u, u ← pop(PQ) ▶ pop(PQ) == (0, A1)
- visited.add(u)
- for each (u→v) in outgoing(u):
  - new_g ← g_u + w(u, v)
  - if new_g < g[v]:
    - g[v] ← new_g
    - parent[v] ← u ▶ X
    - push(PQ, (new_g, v))

※ outgoing(u) == [A2]

g[A2] : not changed ▶ new_g == g[A2] == 2





03 Dijkstra Algorithm

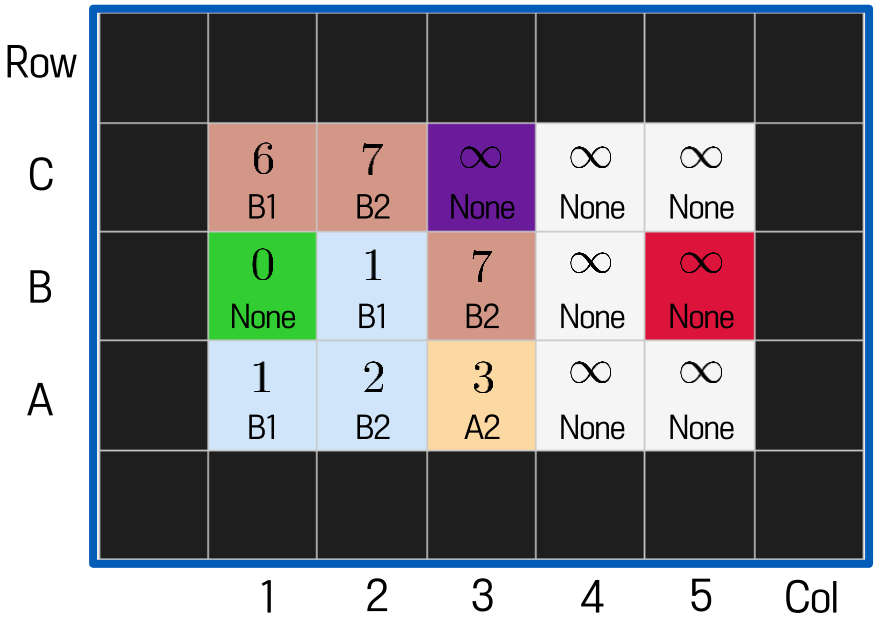
Example of Dijkstra Algorithm – Gridmap (with costmap)

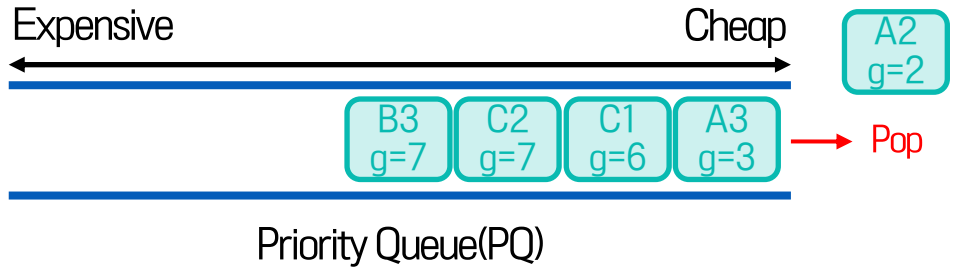
Step 4

- g_u, u ← pop(PQ) ▶ pop(PQ) == (0, A2)
- visited.add(u)
- for each (u→v) in outgoing(u):
  - new_g ← g_u + w(u, v)
  - if new_g < g[v]:
    - g[v] ← new_g
    - parent[v] ← u ▶ A2 = parent of A3
    - push(PQ, (new_g, v))

※ outgoing(u) == [A3]

g[A3] : ∞ → 3





03 Dijkstra Algorithm

Example of Dijkstra Algorithm – Gridmap (with costmap)

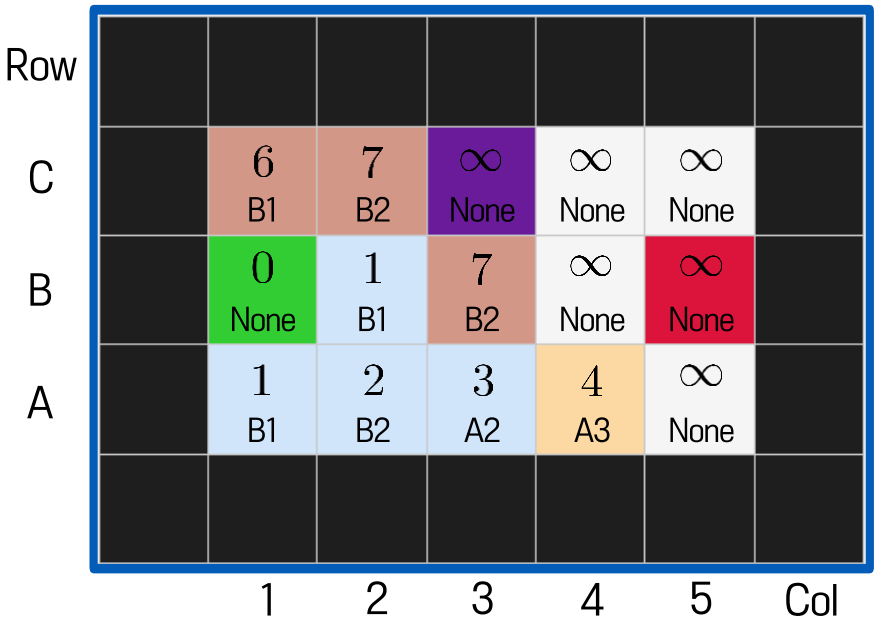
Step 5

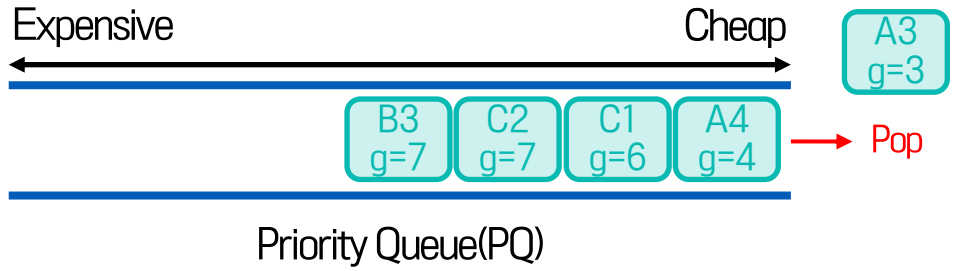
- g_u, u ← pop(PQ) ▶ pop(PQ) == (0, A3)
- visited.add(u)
- for each (u→v) in outgoing(u):
  - new_g ← g_u + w(u, v)
  - if new_g < g[v]:
    - g[v] ← new_g
    - parent[v] ← u ▶ A3 = parent of A4
    - push(PQ, (new_g, v))

※ outgoing(u) == [B3, A4]

g[B3] : not changed ▶ new_g == 9 > g[B3] == 6

g[A4] : ∞ → 4





03 Dijkstra Algorithm

Example of Dijkstra Algorithm – Gridmap (with costmap)

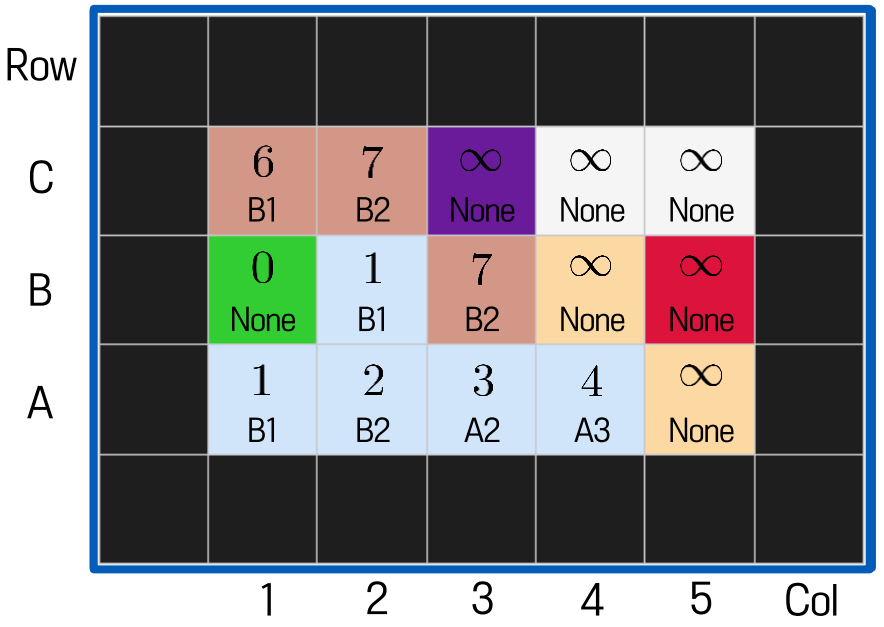
Step 6

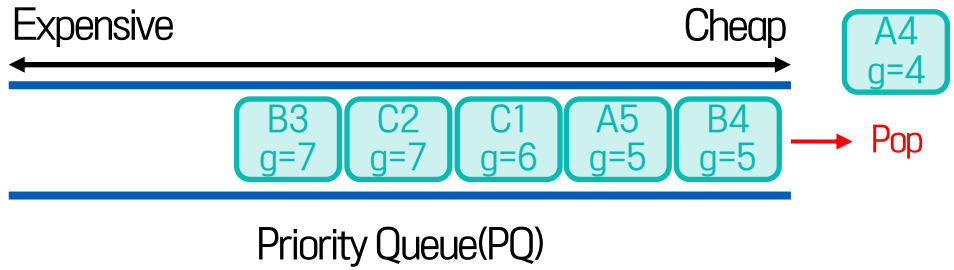
- g_u, u ← pop(PQ) ▶ pop(PQ) == (0, A4)
- visited.add(u)
- for each (u→v) in outgoing(u):
  - new_g ← g_u + w(u, v)
  - if new_g < g[v]:
    - g[v] ← new_g
    - parent[v] ← u ▶ A4 = parent of B4, A5
    - push(PQ, (new_g, v))

※ outgoing(u) == [B4, A5]

g[B4] : ∞ → 5

g[A5] : ∞ → 5





03 Dijkstra Algorithm

Example of Dijkstra Algorithm – Gridmap (with costmap)

Step 7

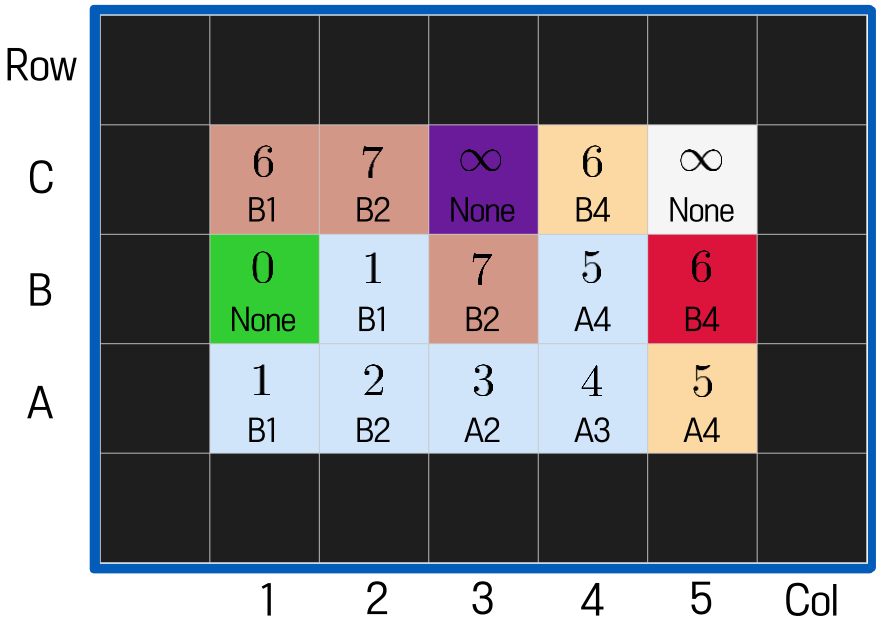
- g_u, u ← pop(PQ) ▶ pop(PQ) == (0, B4)
- visited.add(u)
- for each (u→v) in outgoing(u):
  - new_g ← g_u + w(u, v)
  - if new_g < g[v]:
    - g[v] ← new_g
    - parent[v] ← u ▶ B4 = parent of C4, B5
    - push(PQ, (new_g, v))

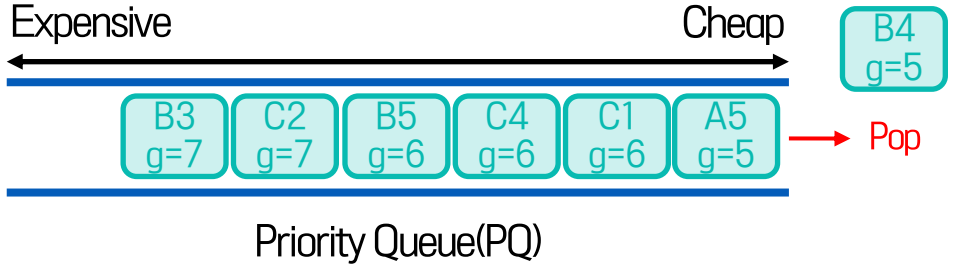
※ outgoing(u) == [C4, B5, B3]

g[C4] : ∞ → 6

g[B5] : ∞ → 6

g[B3] : not changed ▶ new_g == 13 > g[B3] ==7





03 Dijkstra Algorithm

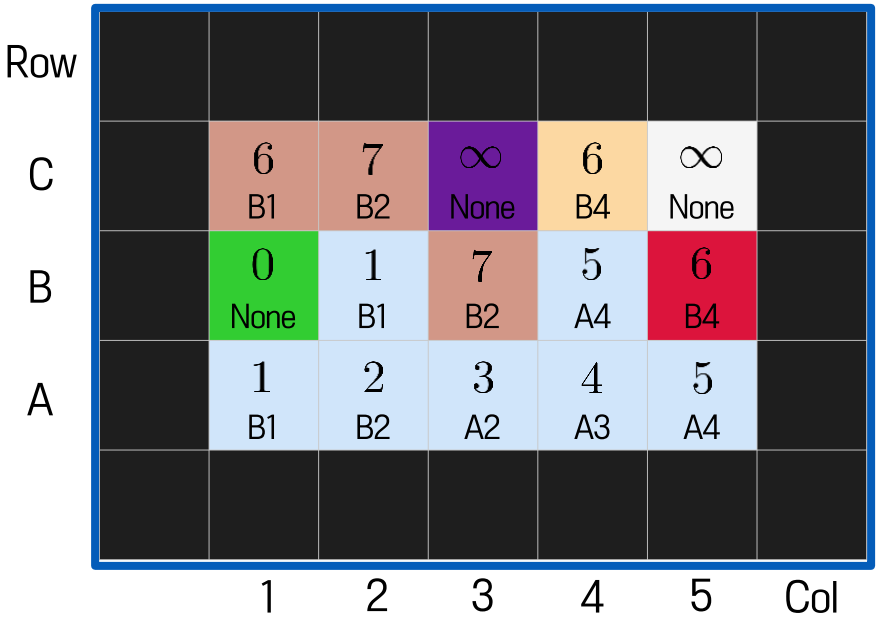
Example of Dijkstra Algorithm – Gridmap (with costmap)

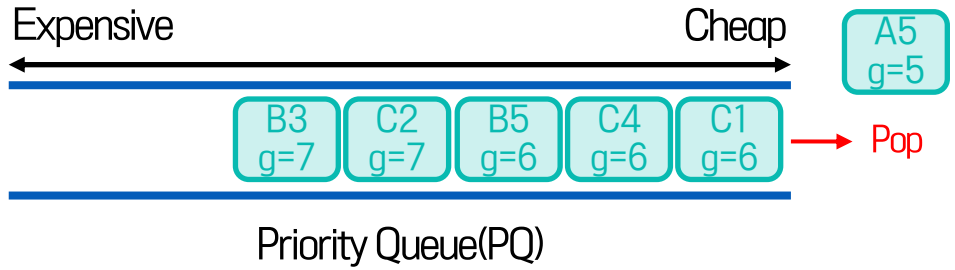
Step 8

- g_u, u ← pop(PQ) ▶ pop(PQ) == (0, A5)
- visited.add(u)
- for each (u→v) in outgoing(u):
  - new_g ← g_u + w(u, v)
  - if new_g < g[v]:
    - g[v] ← new_g
    - parent[v] ← u ▶ X
    - push(PQ, (new_g, v))

※ outgoing(u) == [B5]

g[B5] : not changed ▶ new_g == g[B5] == 6





03 Dijkstra Algorithm

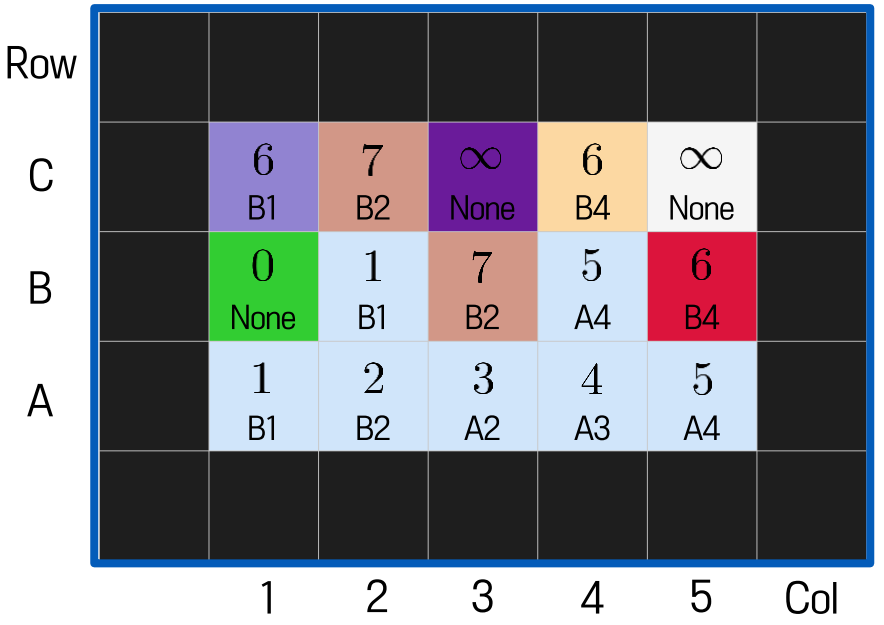
Example of Dijkstra Algorithm – Gridmap (with costmap)

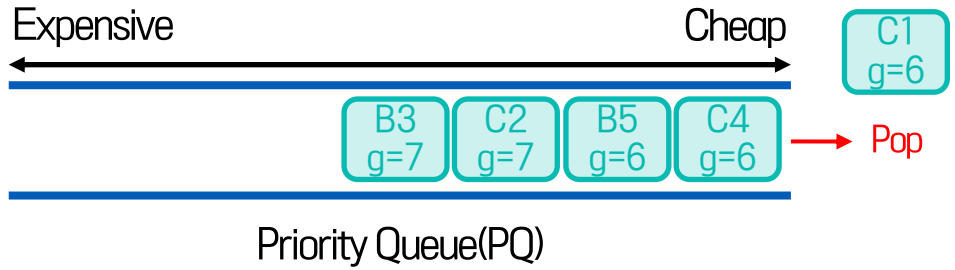
Step 9

- g_u, u ← pop(PQ) ▶ pop(PQ) == (0, C1)
- visited.add(u)
- for each (u→v) in outgoing(u):
  - new_g ← g_u + w(u, v)
  - if new_g < g[v]:
    - g[v] ← new_g
    - parent[v] ← u ▶ X
    - push(PQ, (new_g, v))

※ outgoing(u) == [B2]

g[B5] : not changed ▶ new_g == 14 > g[B2] == 7





03 Dijkstra Algorithm

Example of Dijkstra Algorithm – Gridmap (with costmap)

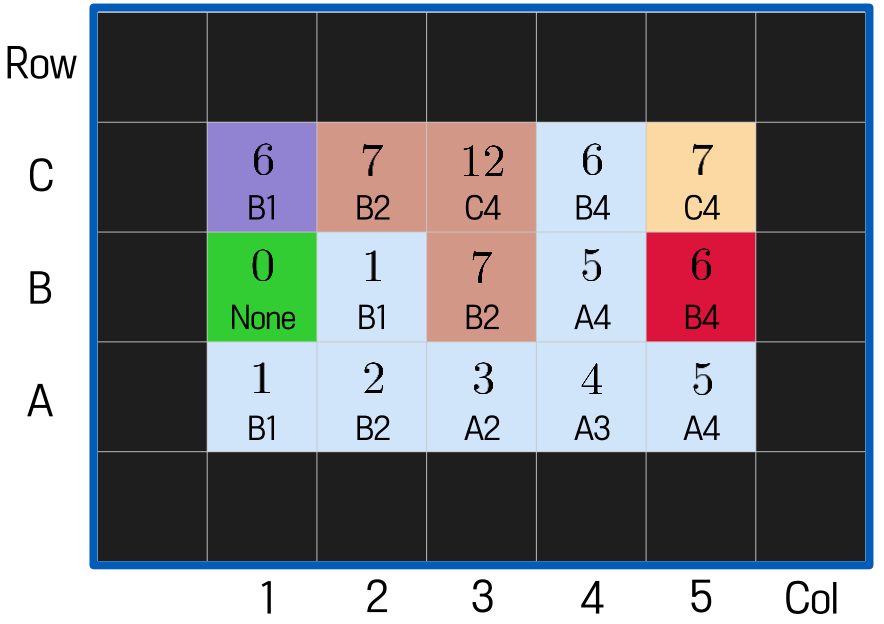
Step 10

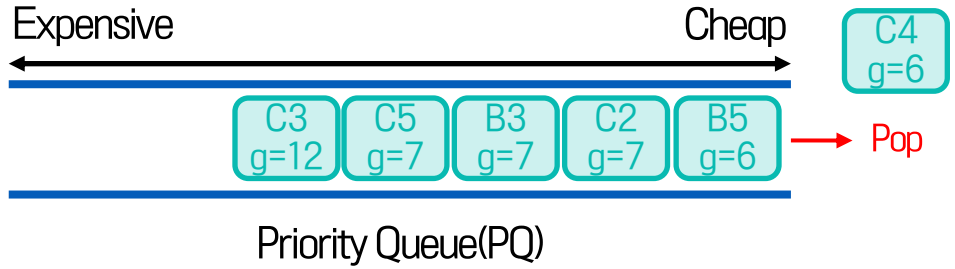
- g_u, u ← pop(PQ) ▶ pop(PQ) == (0, C4)
- visited.add(u)
- for each (u→v) in outgoing(u):
  - new_g ← g_u + w(u, v)
  - if new_g < g[v]:
    - g[v] ← new_g
    - parent[v] ← u ▶ C4 = parent of C5, C3
    - push(PQ, (new_g, v))

※ outgoing(u) == [C5, C3]

g[C5] : ∞ → 7

g[C3] : ∞ → 12





03 Dijkstra Algorithm

Example of Dijkstra Algorithm – Gridmap (with costmap)

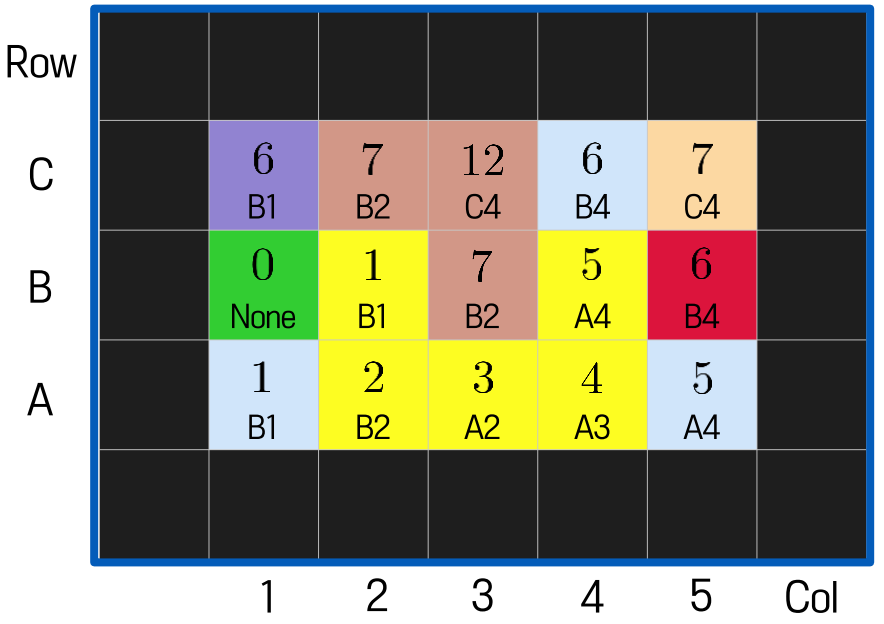
Step 11

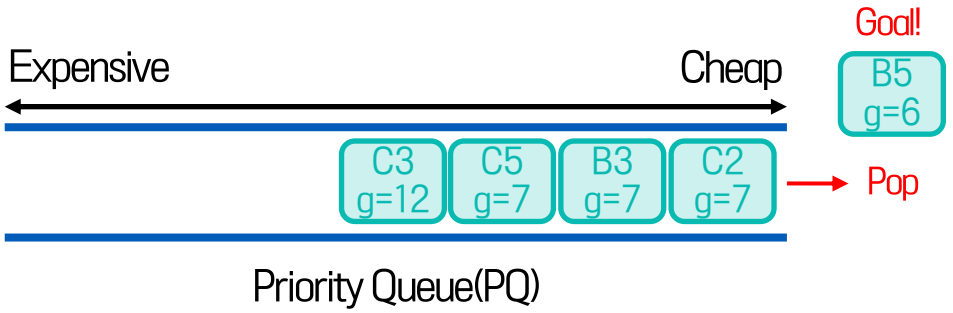
- g_u, u ← pop(PQ) ▶ pop(PQ) == (0, B5)
- visited.add(u)
- if u == goal: ▶ B5 == Goal

  return reconstruct(parent, goal)

parent[B5] == B4 ▶ B5 == Goal
parent[B4] == A4
parent[A4] == A3
parent[A3] == A2
parent[A2] == B2
parent[B2] == B1 ▶ B1 == Start

∴ cost-optimal path : B1 → B2 → A2 → A3 → A4 → B4 → B5

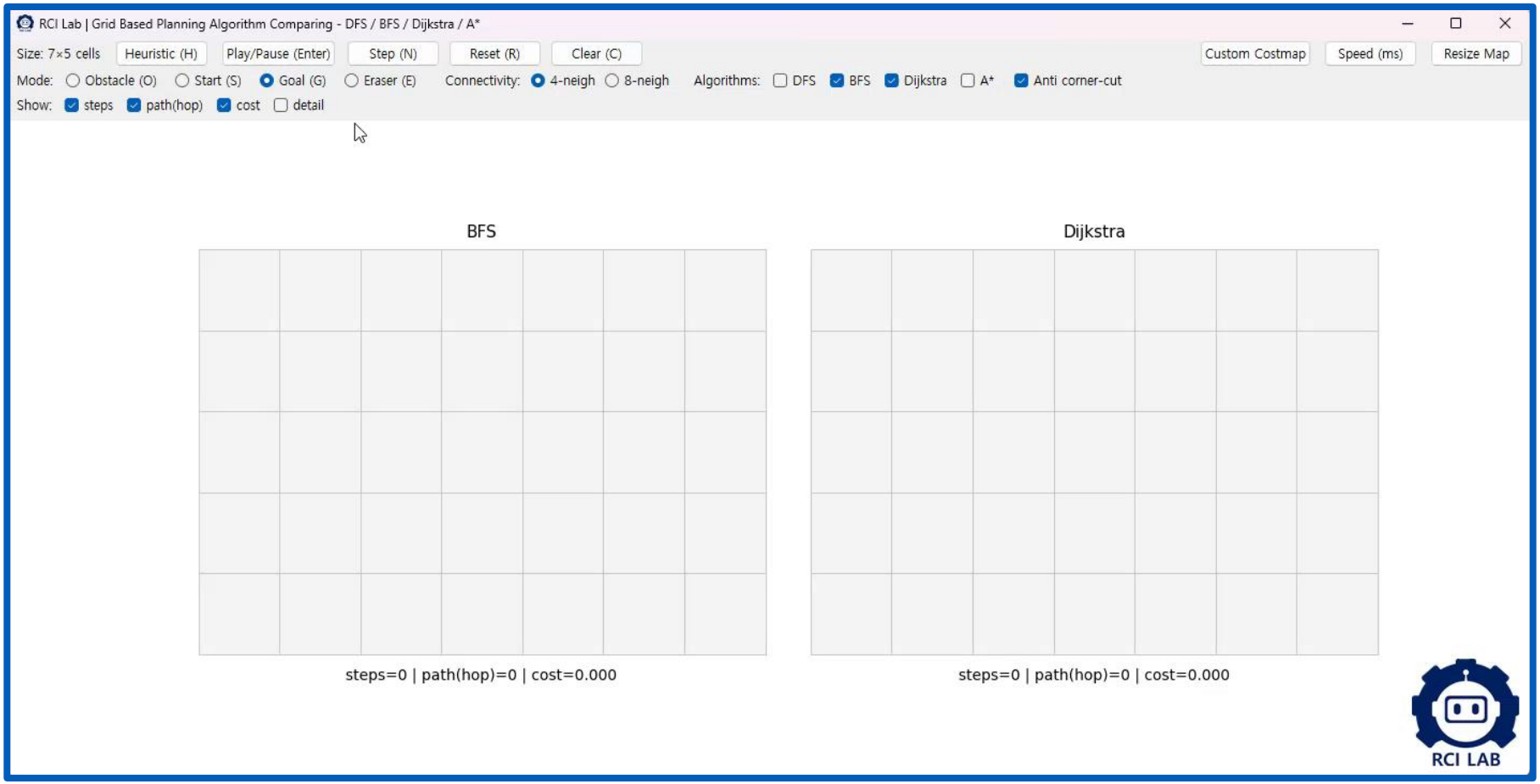




03 Dijkstra Algorithm

Comparison of BFS and Dijkstra

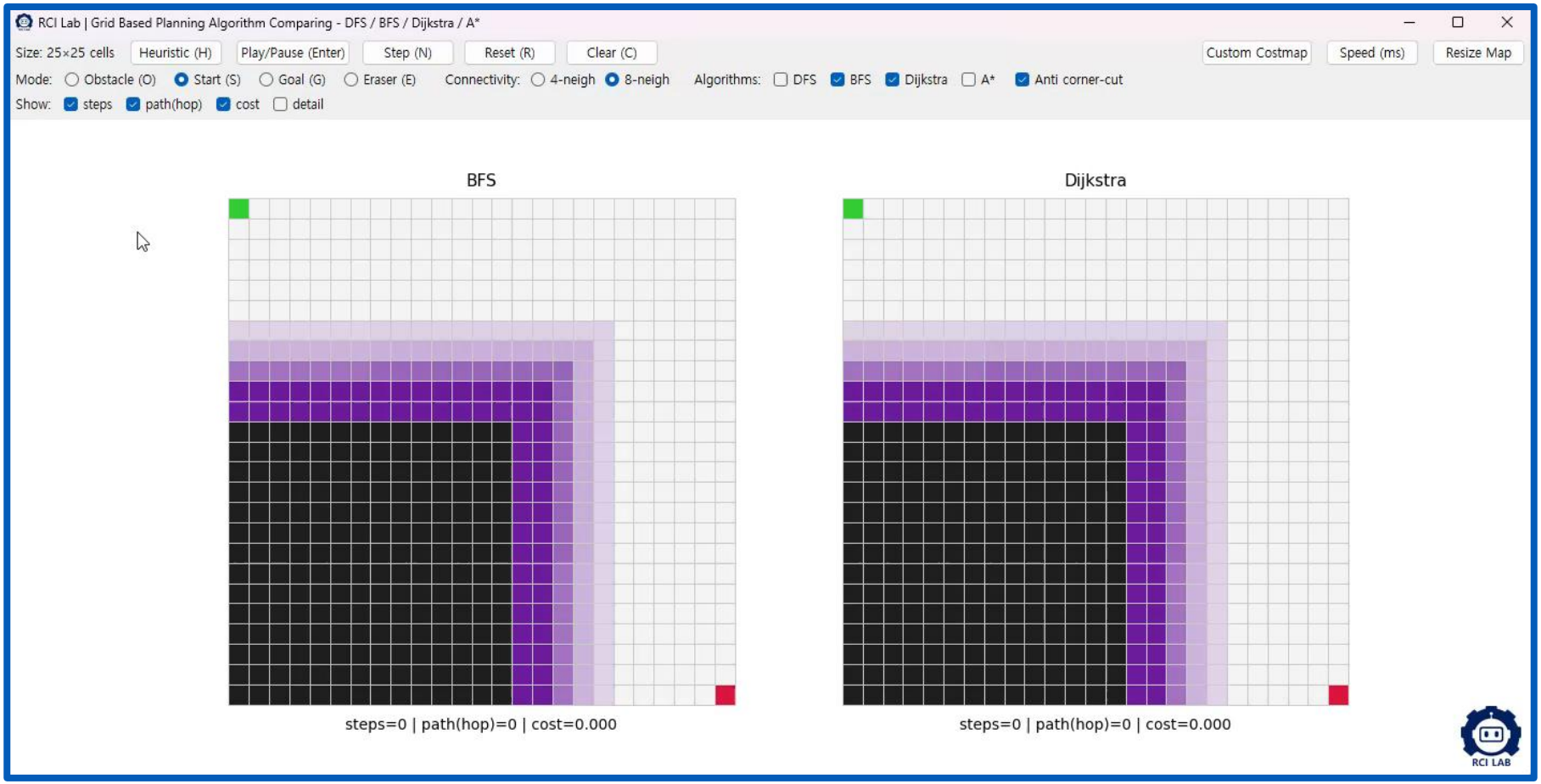
BFS vs. Dijkstra (1)



03 Dijkstra Algorithm

Comparison of BFS and Dijkstra

BFS vs. Dijkstra (2)



03 Dijkstra Algorithm

The Optimality of Dijkstra Algorithm

- The PQ pops nodes(cells) in non-decreasing g.
- With non-negative costs, a popped node's g can never be improved later.

→ Dijkstra Algorithm effectively makes a cheapest-first choice at every step.

Proof by Contradiction

- Assumption : The goal t is popped with g^*, yet there exists a cheaper path $g' < g^*$.
- Let ω be the predecessor of t on that path, then $g[\omega] \leq g' < g^*$.
- ω must have popped earlier and would have relaxed t to g', inserting it into the PQ.
- Then t would pop before g^*.

→ Contradiction.

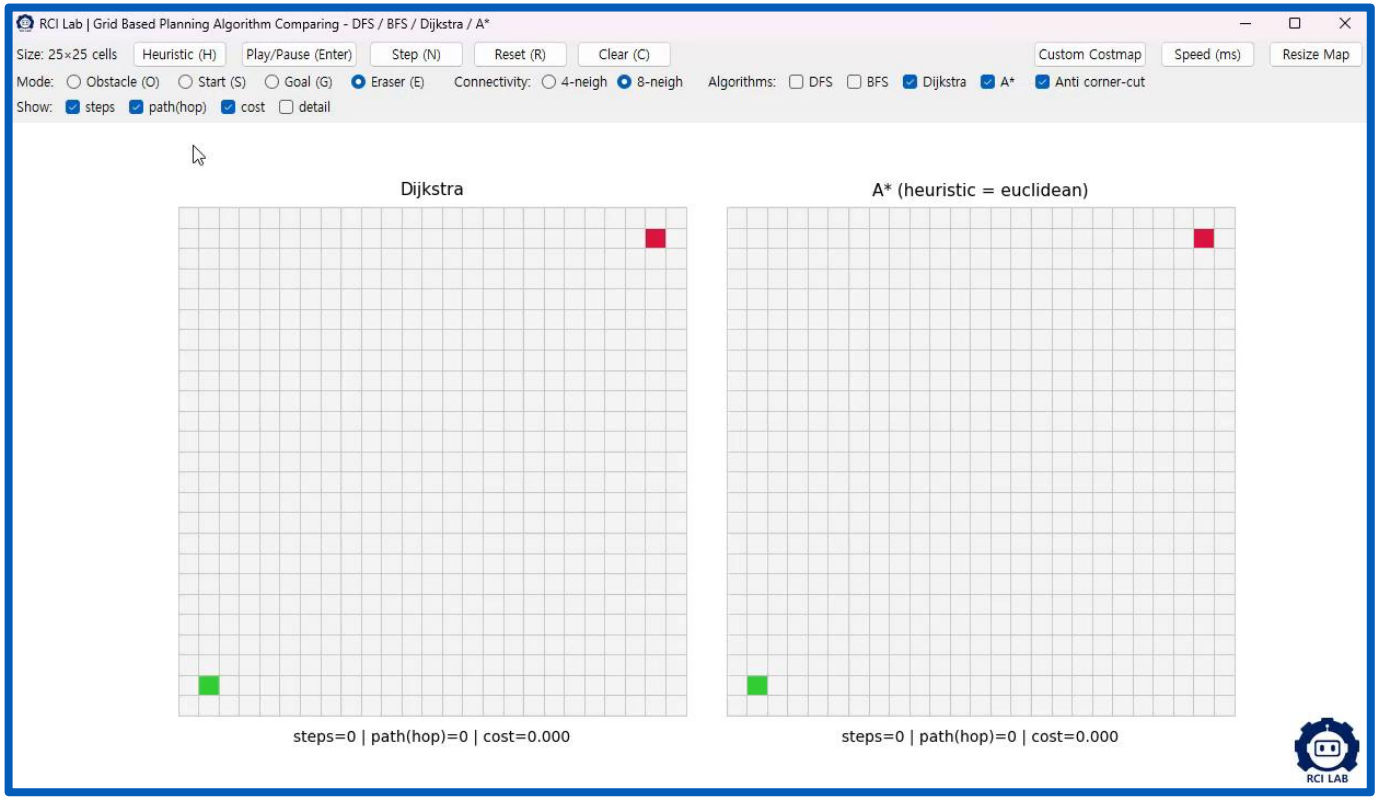
Hence g^* is optimal.

03 Dijkstra Algorithm

Limitations of Dijkstra Algorithm

- No directionality : Dijkstra Algorithm expands by g only.
- Exploring widely : With nearly uniform costs it follows cost contours.
- No heuristic : Lacks a lower bound toward the goal.

It takes too long
to find a optimal path



03 Dijkstra Algorithm

Limitations of Dijkstra Algorithm

- No heuristic : Lacks a lower bound toward the goal.

Let $J^*(n)$ be the optimal remaining cost from node(cell) n to the goal.

A lower bound(heuristic) $h(n)$ is any function that always satisfies $h(n) \leq J^*(n)$.

- Dijkstra : Uses $h \equiv 0$, i.e., $f(n) = g(n)$ only.

→ It has no information about the minimum remaining cost, so it lacks goal directionality and tends to explore wide iso-cost shells of equal g.

- A* : Uses the key $f(n) = g(n) + h(n)$

→ combines cost so far g with a lower bound to go h .

→ If h is admissible and consistent, A* keeps stop-on-pop optimality while reducing the number of expansions.

advantage of Dijkstra

improvement

Thank you