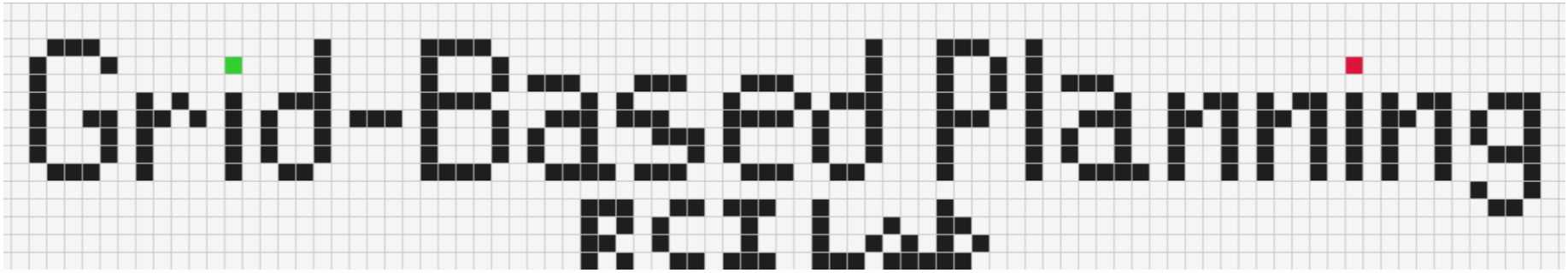
Algorithms – DFS, BFS



Speaker: Taehyun Jung

RCI LAB

KHU

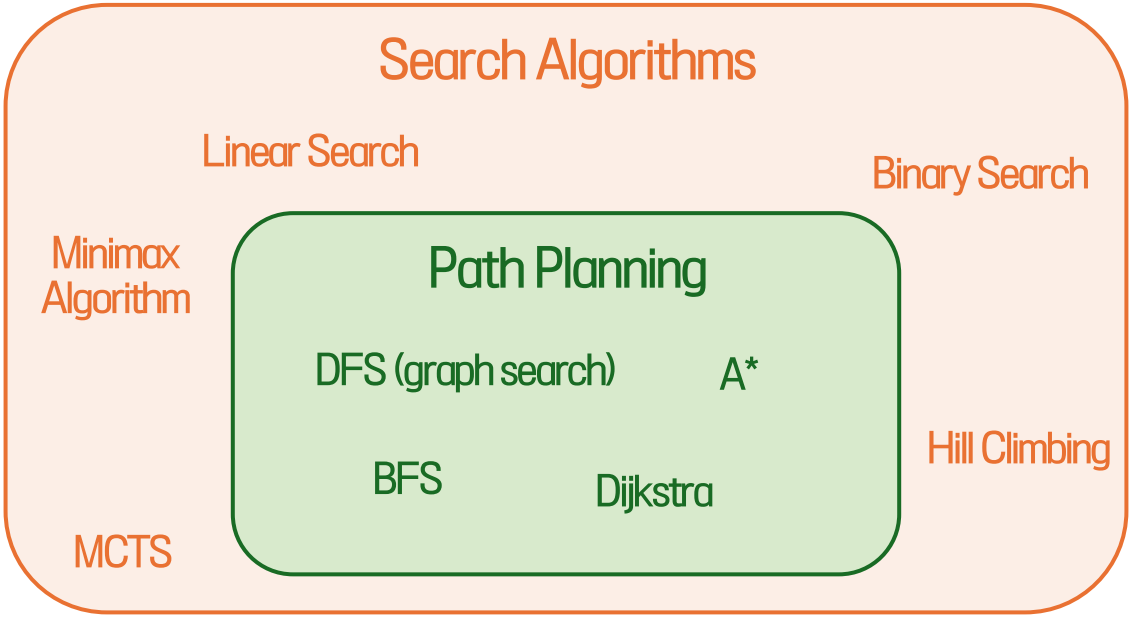
Table of Contents

01 Search Algorithm

01 Search Algorithm

Search Algorithms and Path Planning

- Search Algorithms are systematic state-space exploration to reach a goal or solution.
- Path Planning(finding) is a specialized search task focusing on a path under cost and constraints.

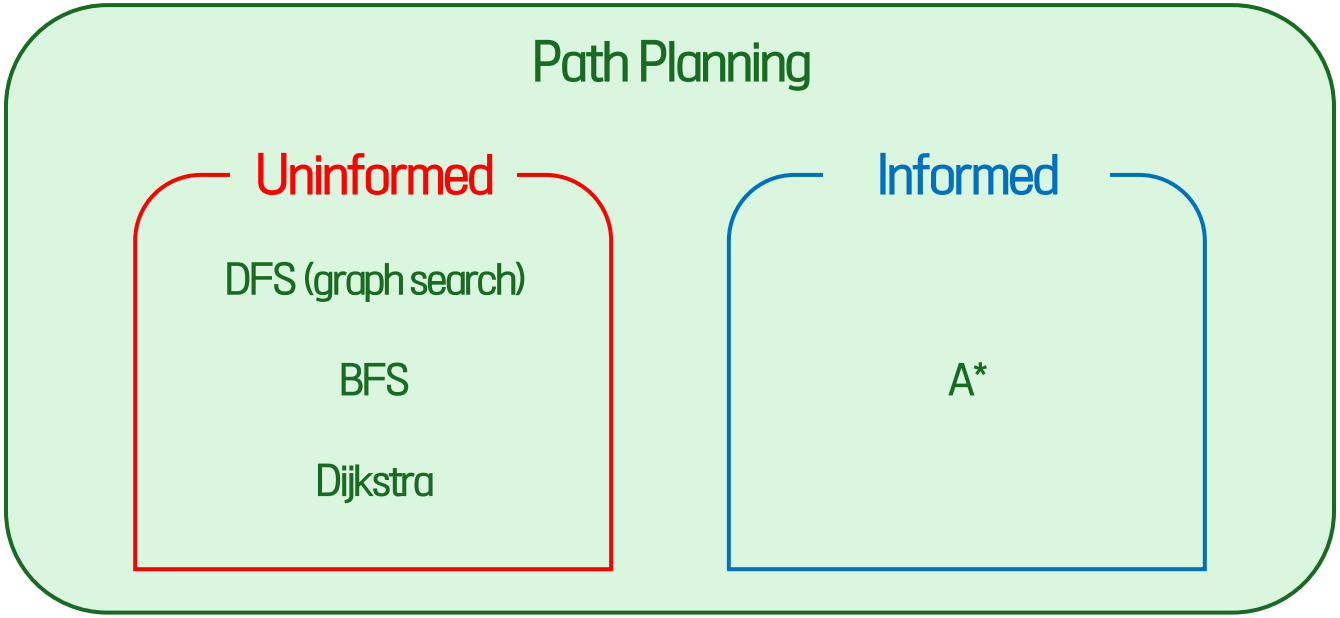


01 Search Algorithm

Uninformed vs. Informed

- Uninformed(Blind) Search :
  No information about the remaining distance to goal. (No heuristic $h(n)$)
  The expansion order is determined solely by the graph's structure.

- Informed(Heuristic) Search :
  Uses a heuristic $h(n)$ to estimate how close a node is to the goal,
  and expands more promising nodes first based on that estimate.

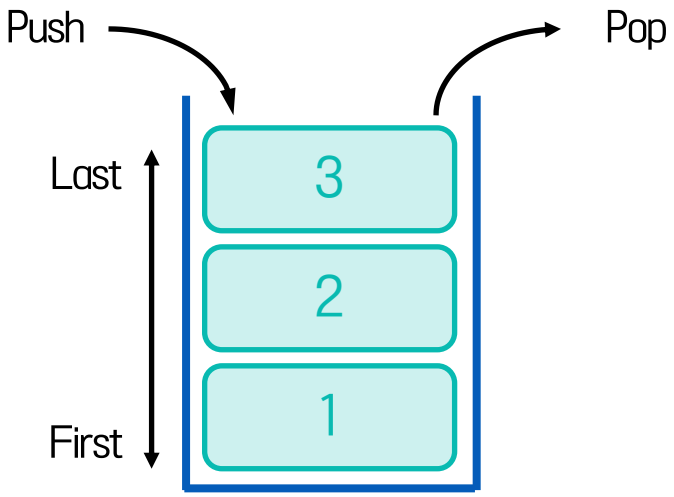


02 Stack vs. Queue

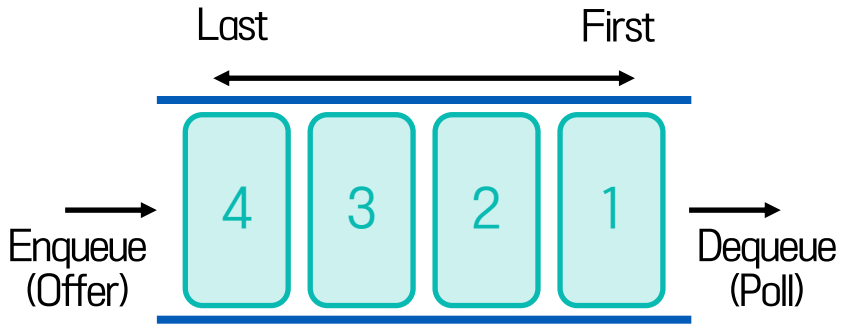
02 Stack vs. Queue

LIFO vs. FIFO

- LIFO(Last In First Out): Data structure where the object you put in later leaves first
- FIFO(First In First Out): Data structure where the object you put in first escapes first



Stack(LIFO)



Queue(FIFO)

→ Logics for path planning varies depending on the data structure that stores information about path.

↳ visited, weight

02 Stack vs. Queue

LIFO vs. FIFO

- LIFO(Last In First Out): Data structure where the object you put in later leaves first
- FIFO(First In First Out): Data structure where the object you put in first escapes first

Representative Algorithms

Algorithm	Main Data Structure	Order Type
DFS	Stack(or recursion stack)	LIFO
BFS	Queue	FIFO

Algorithm	Pros	Cons
DFS	• Simple to implement • Uses relatively little memory	• Not complete (Can go infinitely deep in very large graphs) • Does not guarantee the shortest path
BFS	• On an unweighted graph(unit-cost), it guarantees the shortest path • Complete for finite graphs	• Needs to store many nodes at the same depth → higher memory usage than DFS

03 DFS Algorithm

03 DFS Algorithm

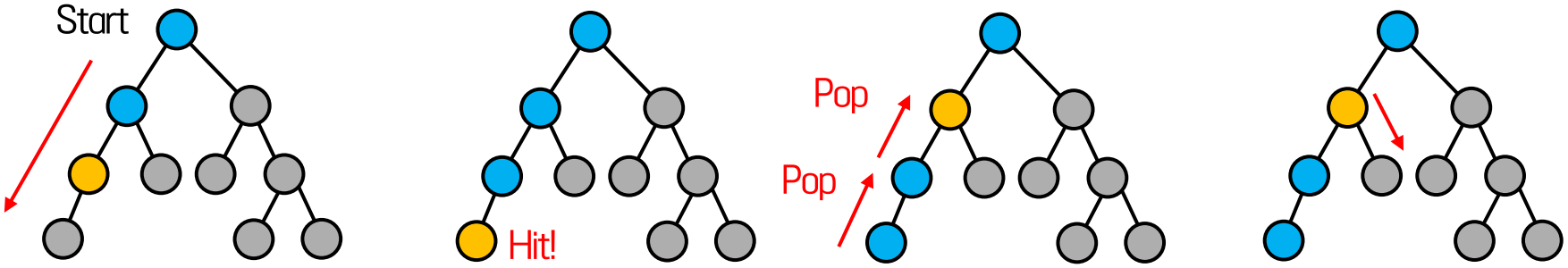
DFS (Depth First Search)

- DFS goes as deeply as possible along a branch. On dead-ends, it backtracks to try another.
- DFS is a basic graph analysis like detection of connected components, cycle, maze generation/solving, etc.

Main Idea

: unvisited : visited : frontier

Dive deeply as possible › Hit a dead-end › Backtrack by popping › Try unvisited branches

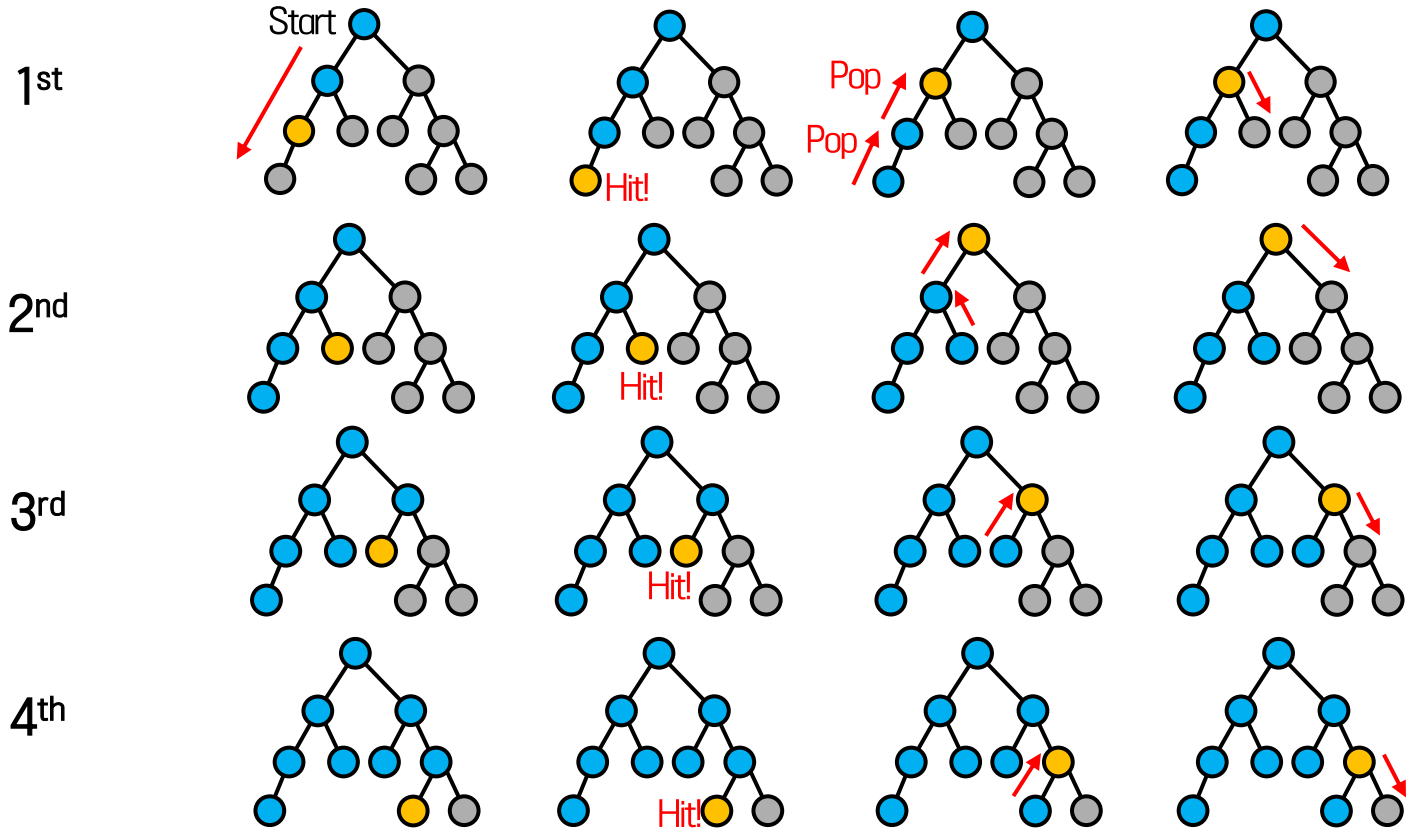


03 DFS Algorithm

Main Idea

: unvisited : visited : frontier

Dive deeply as possible › Hit a dead-end › Backtrack by popping › Try unvisited branches › Repeat until finding the goal

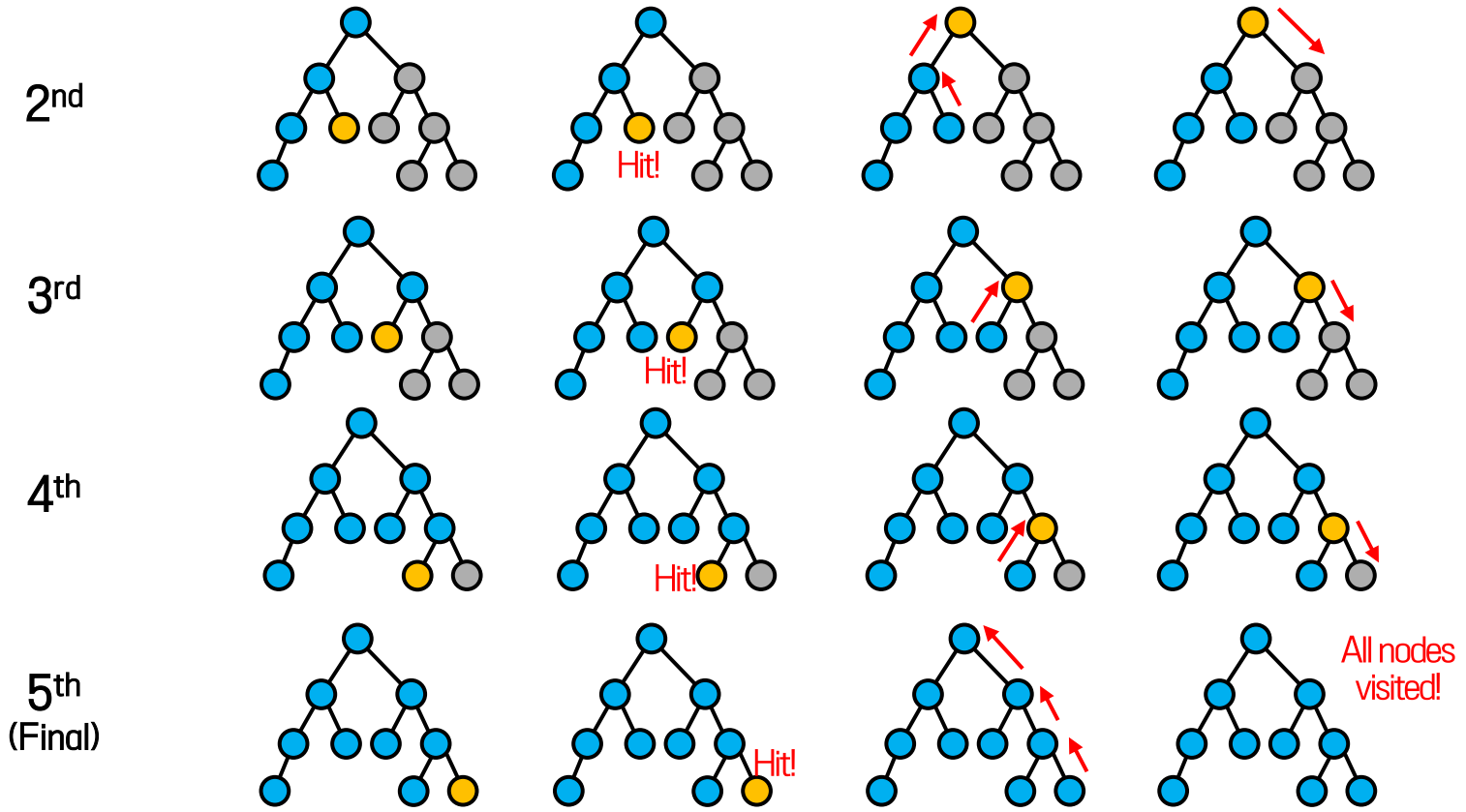


03 DFS Algorithm

Main Idea

: unvisited : visited : frontier

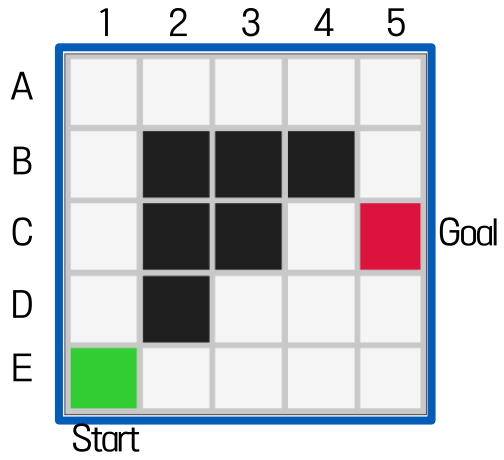
Dive deeply as possible › Hit a dead-end › Backtrack by popping › Try unvisited branches › Repeat until finding the goal



03 DFS Algorithm

Application DFS to Path Planning

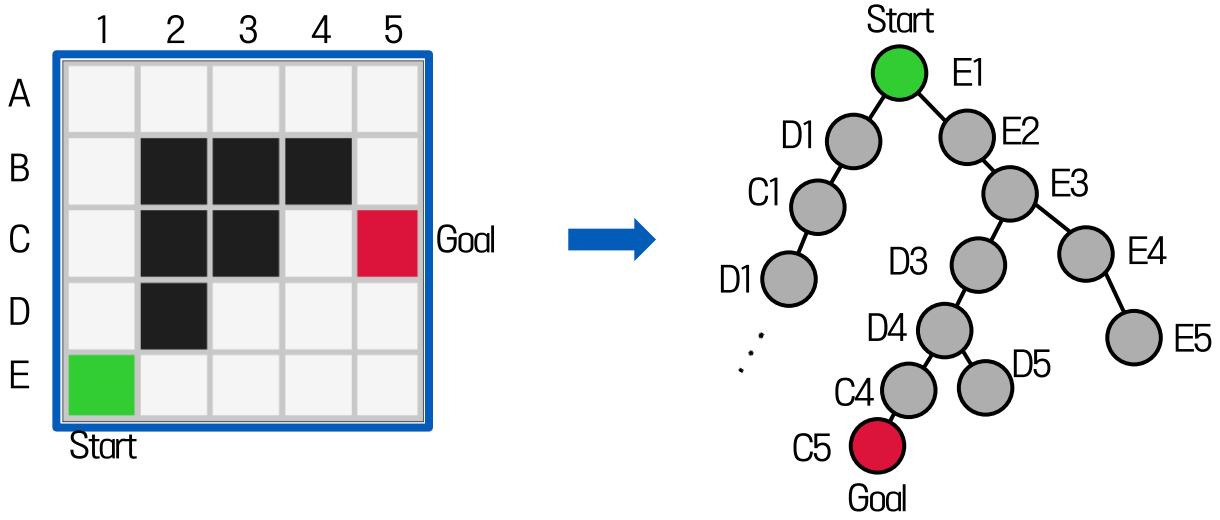
- Grid as a graph:
  1. Nodes → Cells
  2. Edges → Connectivity
  3. Dead-end → Obstacle (or boundaries of map)
  4. Rule of visiting: Left first → N, E, S, W (4-connectivity)
- Obstacles block transitions.
- Note that DFS finds a path, not a shortest one.



03 DFS Algorithm

Application DFS to Path Planning

- Grid as a graph:
  1. Nodes → Cells
  2. Edges → Connectivity
  3. Dead-end → Obstacle (or boundaries of map)
  4. Rule of visiting: Left first → N, E, S, W (4-connectivity)
- Obstacles block transitions.
- Note that DFS finds a path, not a shortest one.



03 DFS Algorithm

Application DFS to Path Planning

Pseudocode

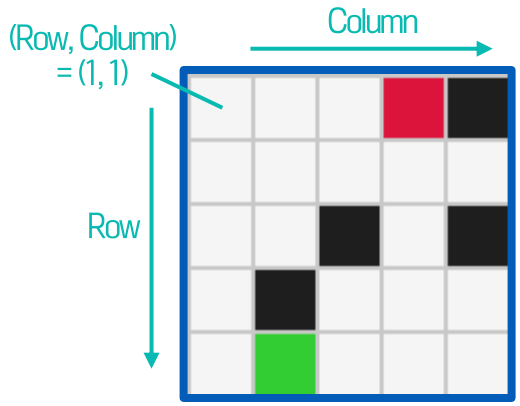
```
1)  S ← [start], parent[start] ← None                ▶ init stack & parent
2)  while S not empty:
3)          u ← pop(S)                               ▶ expand stack(LIFO)
4)          mark u as visited
5)          if u == goal:                            ▶ Check goal when popping
6)                  return reconstruct(parent, u)
7)          for v in neighbors(u) in reverse(N, E, S, W):   ▶ push in reverse of NESW
8)                  if valid(v) and v not visited:
9)                          parent[v] ← u
10)                         push(S, v)               ▶ treat v as visited
11) //                      if v == goal:            ▶ Check goal when pushing
12) //                              return reconstruct(parent, v)
13) return failure
```

In this lecture, we're going to check the goal when popping.

※ You may check the goal either on push or on pop. Pick one and keep it consistent. On-push ends earlier, on-pop aligns with Dijkstra / A*.

03 DFS Algorithm

Application DFS to Path Planning – Example(4-connectivity)



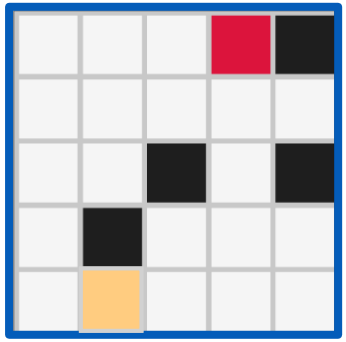
: unvisited
: visited
: obstacle
: frontier
: start
: goal

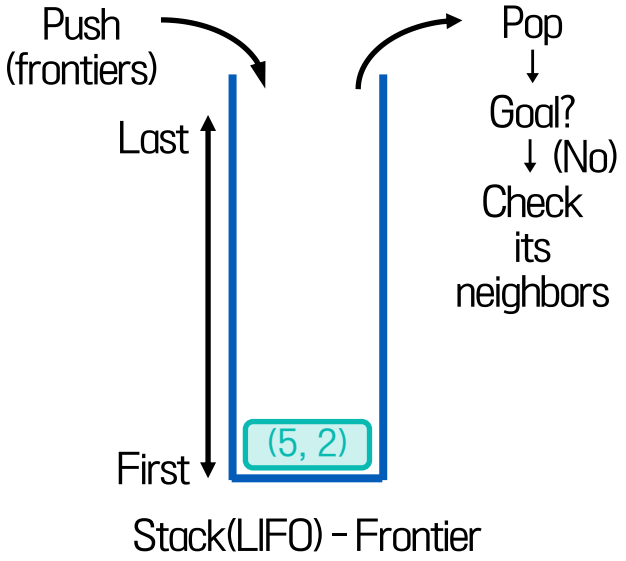
※ Rule of Transition : N → E → S → W

Step 1

- push Start(5,2)
- frontier: {(5,2)}
- parent[(5, 2)] = None

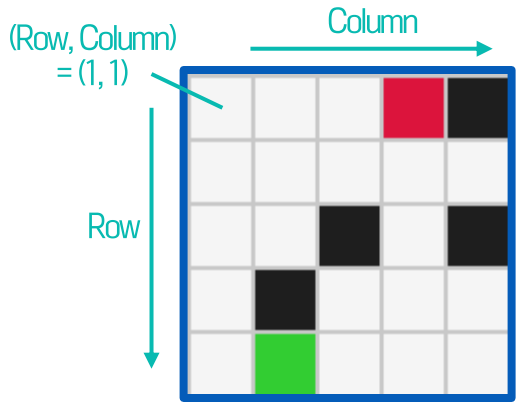
// initialize





03 DFS Algorithm

Application DFS to Path Planning – Example(4-connectivity)



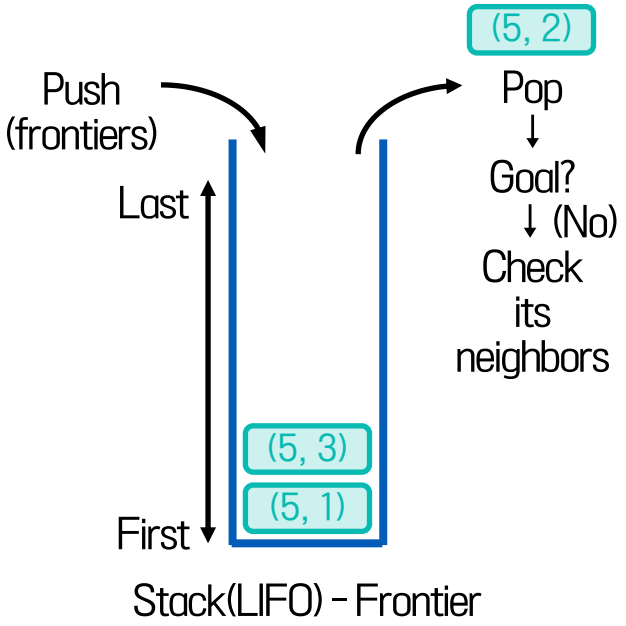
: unvisited
: visited
: obstacle
: frontier
: start
: goal

※ Rule of Transition : N → E → S → W

Step 2

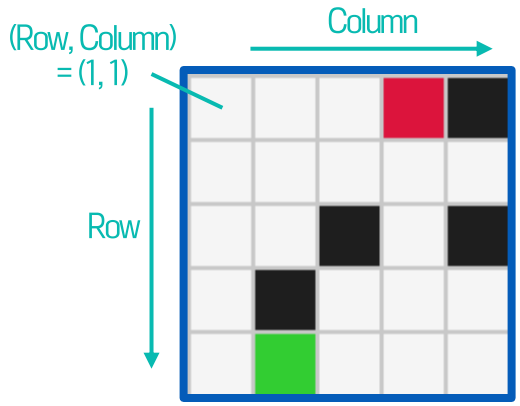
- pop (5, 2) & Check
- W(5, 1): O / E(5, 3): O / N(4, 2): X
  → push W(5, 1) → push E(5, 3)

- frontier: {(5, 3), (5, 1)}
- parent[(5, 1)] = (5, 2)
  parent[(5, 3)] = (5, 2)



03 DFS Algorithm

Application DFS to Path Planning – Example(4-connectivity)



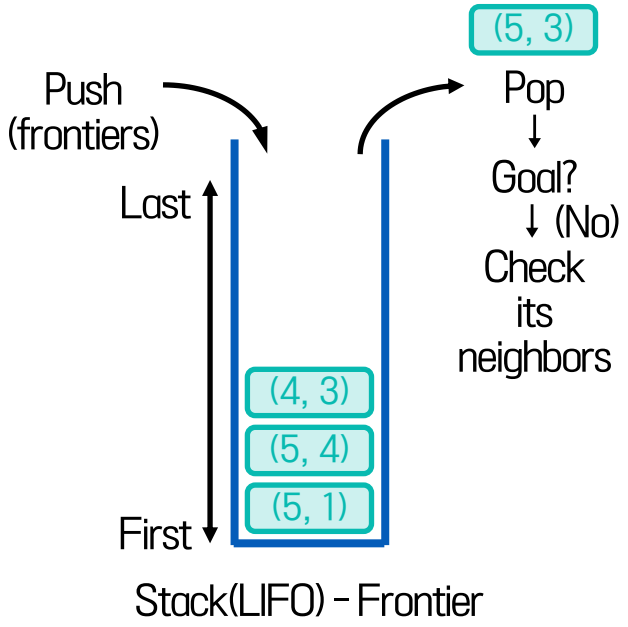
: unvisited
: visited
: obstacle
: frontier
: start
: goal

※ Rule of Transition : N → E → S → W

Step 3

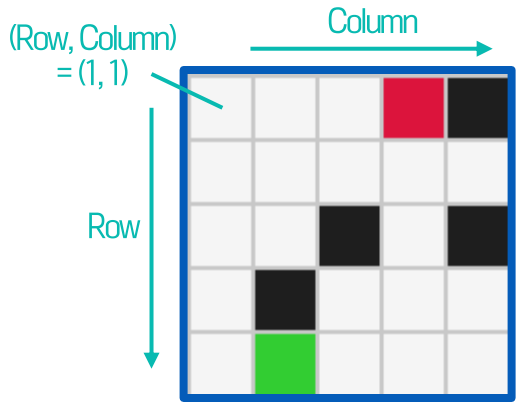
- pop (5, 3) & Check
- E(5, 4): O / N(4, 3): O
  → push E(5, 4) → push N(4, 3)

- frontier: {(4, 3), (5, 4), (5, 1)}
- parent[(5, 4)] = (5, 3)
  parent[(4, 3)] = (5, 3)



03 DFS Algorithm

Application DFS to Path Planning – Example(4-connectivity)



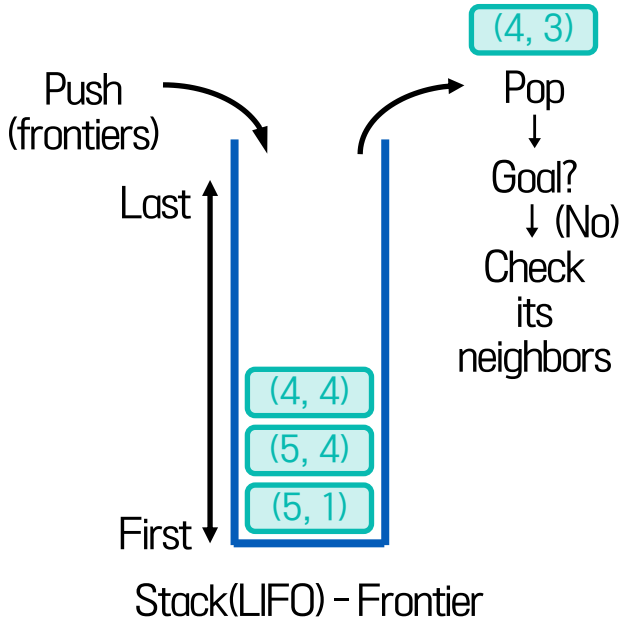
: unvisited
: visited
: obstacle
: frontier
: start
: goal

※ Rule of Transition : N → E → S → W

Step 4

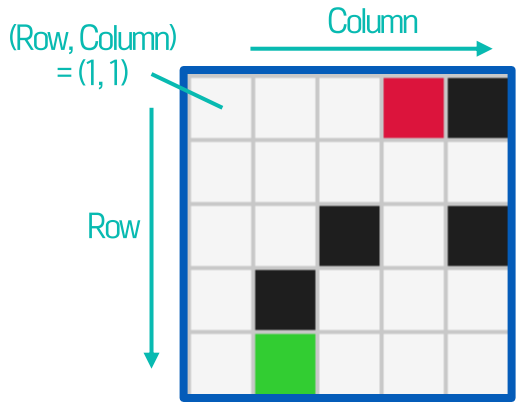
- pop (4, 3) & Check
- S(4, 2): X / E(4, 4): O / N(3, 3): X
  → push E(4, 4)

- frontier: {(4, 4), (5, 4), (5, 1)}
- parent[(4, 4)] = (4, 3)



03 DFS Algorithm

Application DFS to Path Planning – Example(4-connectivity)



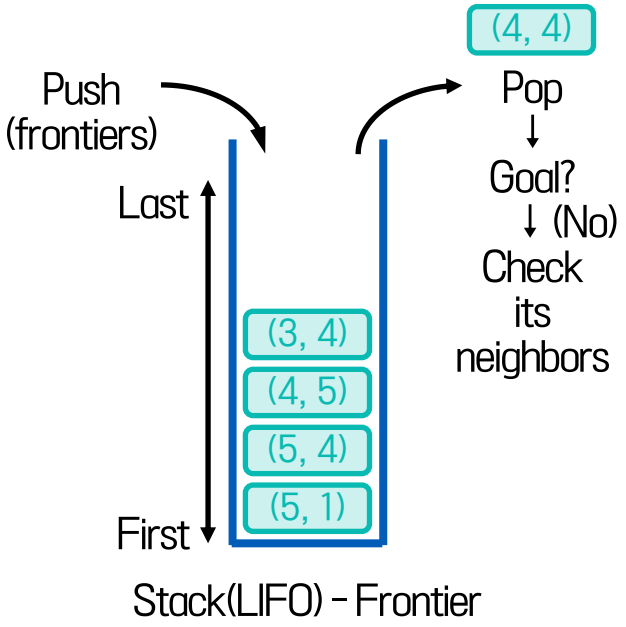
: unvisited
: visited
: obstacle
: frontier
: start
: goal

※ Rule of Transition : N → E → S → W

Step 5

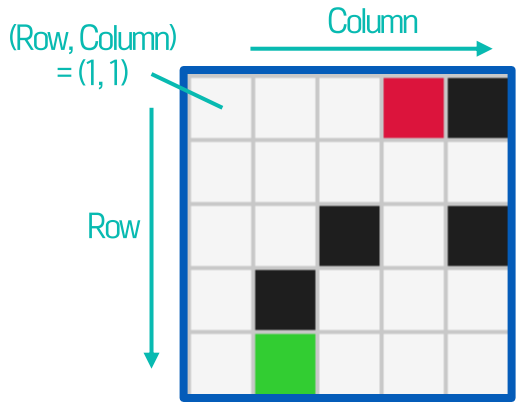
- pop (4, 4) & Check
- E(4, 5): O / N(3, 4): O
  → push E(4, 5) → push N(3, 4)

- frontier:
  {(3, 4), (4, 5), (5, 4), (5, 1)}
- parent[(3, 4)] = (4, 4)
  parent[(4, 5)] = (4, 4)



03 DFS Algorithm

Application DFS to Path Planning – Example(4-connectivity)



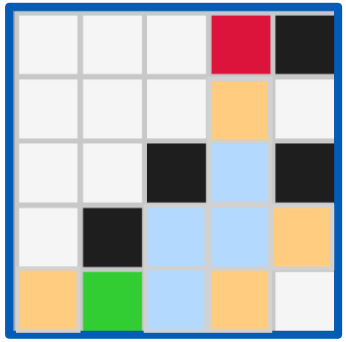
: unvisited
: visited
: obstacle
: frontier
: start
: goal

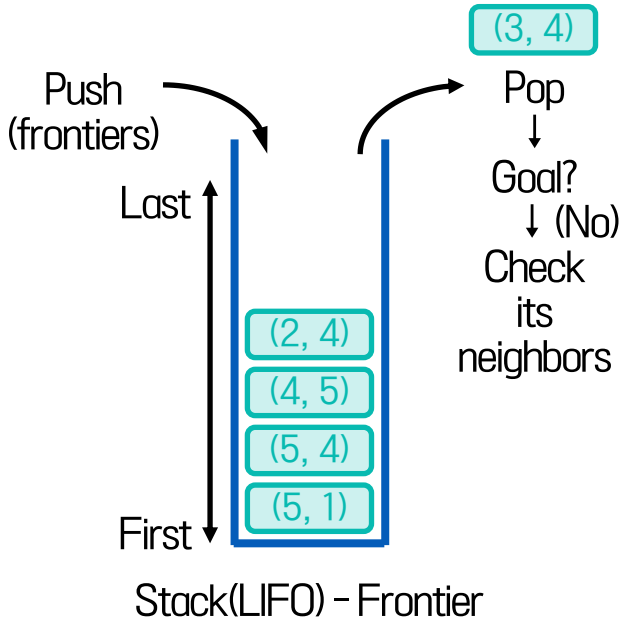
※ Rule of Transition : N → E → S → W

Step 6

- pop (3, 4) & Check
- W(3, 3): X / E(3, 5): X / N(2, 4): O
  → push N(2, 4)

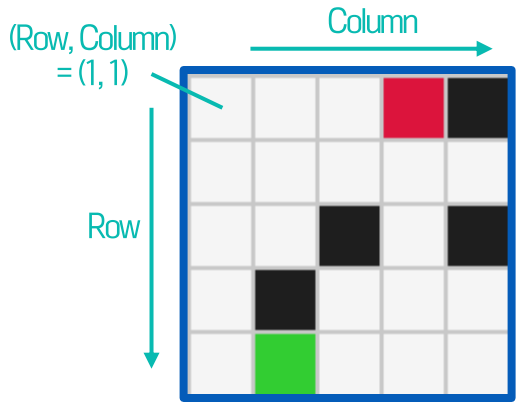
- frontier:
  {(2, 4), (4, 5), (5, 4), (5, 1)}
- parent[(2, 4)] = (3, 4)





03 DFS Algorithm

Application DFS to Path Planning – Example(4-connectivity)



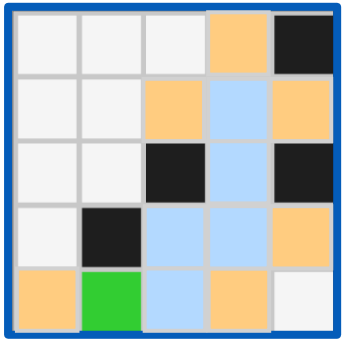
: unvisited
: visited
: obstacle
: frontier
: start
: goal

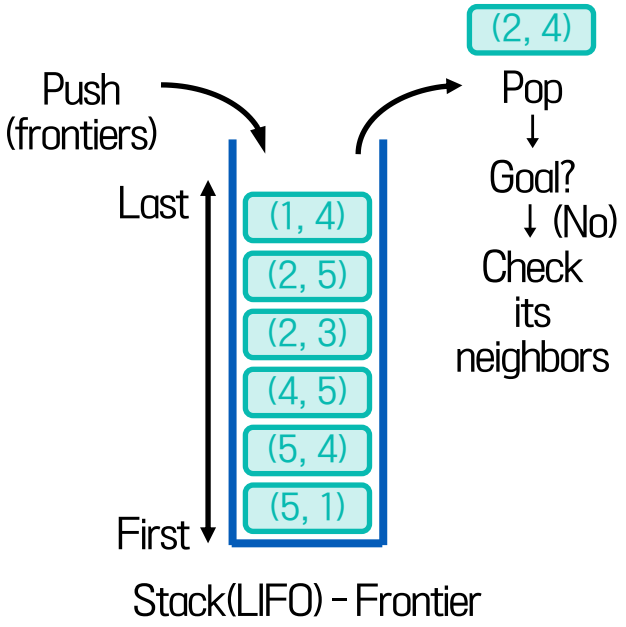
※ Rule of Transition : N → E → S → W

Step 7

- pop (2, 4) & Check
- W(2, 3): O / E(2, 5): O / N(1, 4): O
  → push W(2, 3) → push E(2, 5) → push N(1, 4)

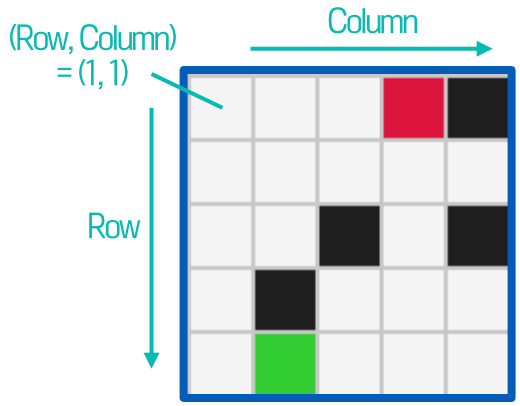
- frontier:
  {(1, 4), (2, 5), (2, 3), (4, 5), (5, 4), (5, 1)}
- parent[(1, 4)] = (2, 4)
  parent[(2, 5)] = (2, 4)
  parent[(2, 3)] = (2, 4)





03 DFS Algorithm

Application DFS to Path Planning – Example(4-connectivity)

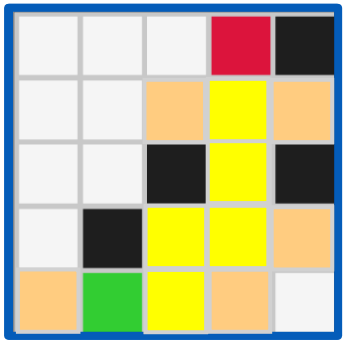


: unvisited
: visited
: obstacle
: frontier
: start
: goal

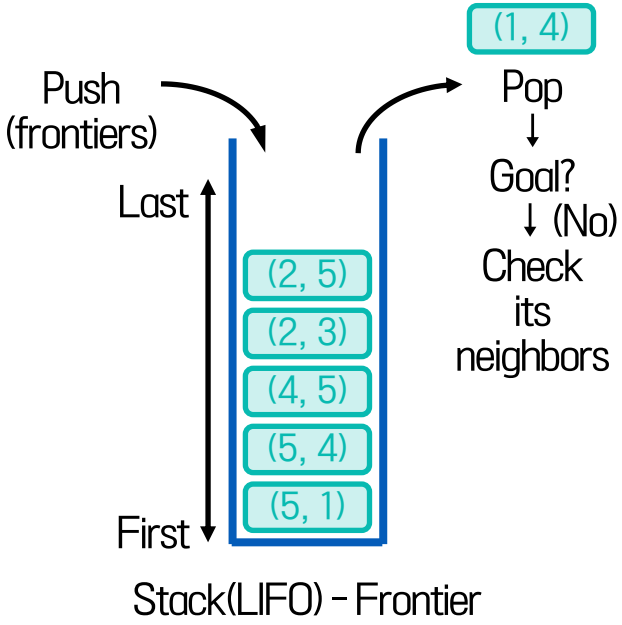
※ Rule of Transition : N → E → S → W

Step 8

- pop (1, 4) & Check ▶ (1, 4) == Goal
- Path Reconstruction from Parent:
  (※ prt == parent) // recursive
  prt[(1, 4)] → prt[(2, 4)] → prt[(3, 4)] →
  prt[(4, 4)] → prt[(4, 3)] → prt[(5, 3)] →
  prt[(5, 2)] // == None



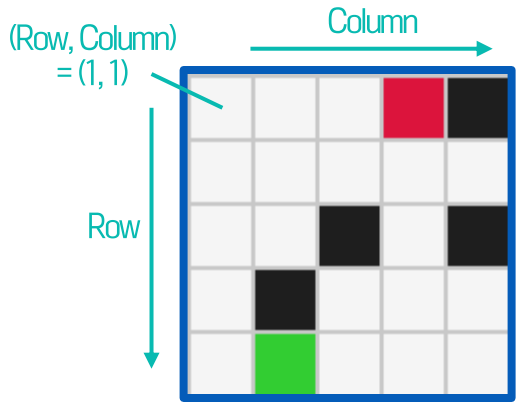
- Reverse:
  Start(5, 2) → (5, 3) → (4, 3) → (4, 4) → (3, 4)
  → (2, 4) → Goal(1, 4)



Stack(LIFO) - Frontier

03 DFS Algorithm

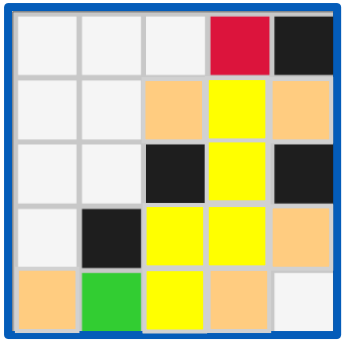
Application DFS to Path Planning – Example(4-connectivity)

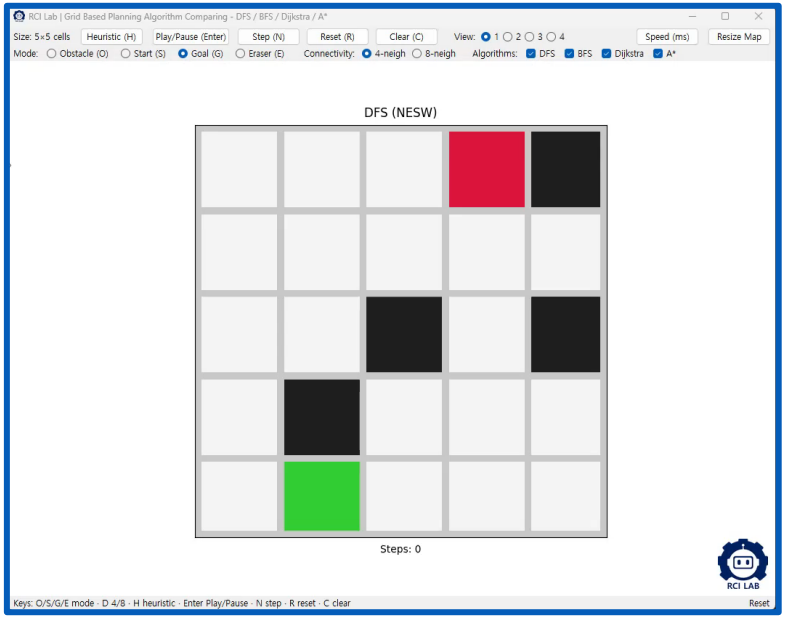


: unvisited
: visited
: obstacle
: frontier
: start
: goal

※ Rule of Transition : N → E → S → W

Demonstration in code

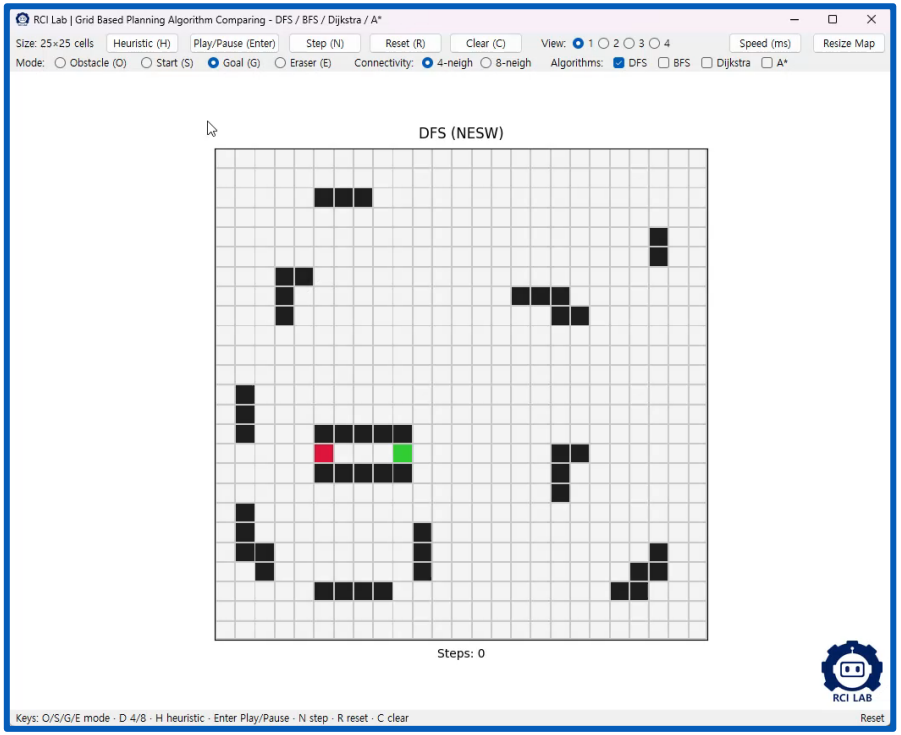




03 DFS Algorithm

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.

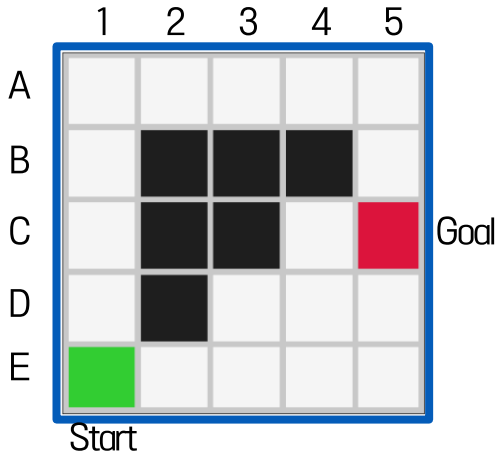


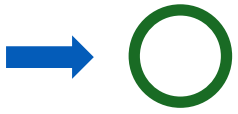
04 BFS Algorithm

04 BFS Algorithm

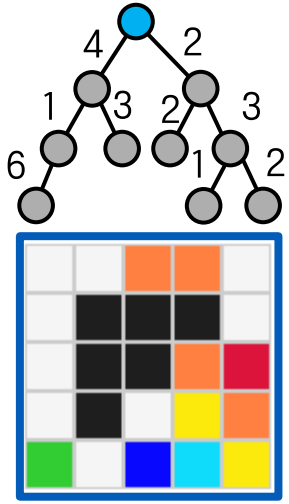
BFS (Breadth First Search)

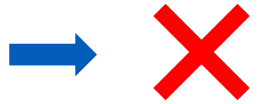
- BFS expands nodes by increasing distance(levels) from the start, using a Queue(FIFO) as the frontier.
- BFS marks visited and records parents at enqueue time.
- BFS guarantees a hop(edge)-shortest path on unit-cost graphs.





BFS can guarantee the shortest path.





BFS can't guarantee the shortest path.

Grid (= unit-cost graph)

- 4-connectivity
- 8-connectivity (Chebyshev dist)

Weighted-edge graph

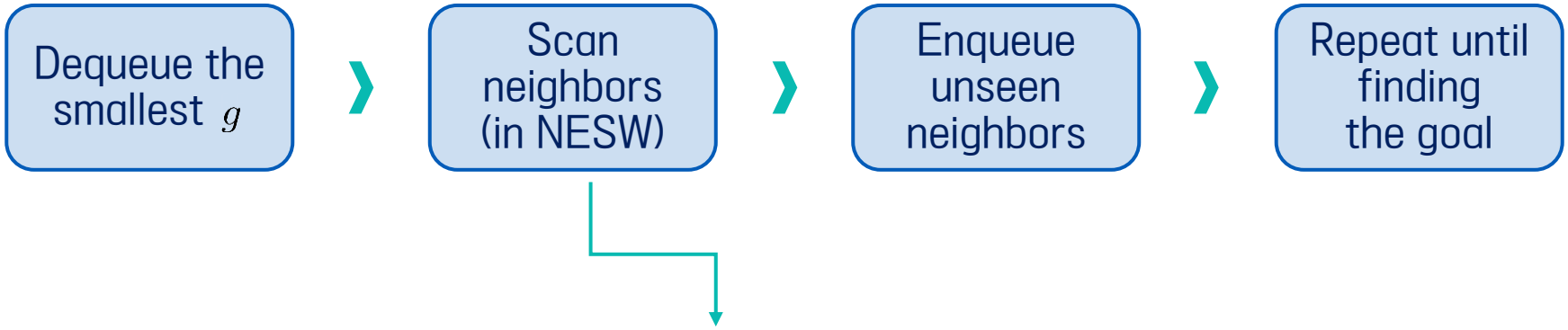
- 8-connectivity (Euclidean dist)
- Grid which is applied non-uniform costmap

04 BFS Algorithm

Main Idea

BFS grows the frontier in non-decreasing path cost(g), recording parents to reconstruct the path.

- Frontier(container of candidates) expands smallest g first. → Key Idea
- Record parent[v] = u on first valid discovery.
- Stop when the goal is selected for expansion. Then, reconstruct via parents



- When several neighbors have same cost, check/enqueue them in NESW order.
- This only decides which one goes first, it does not change shortest-path guarantees(unit-cost) or optimality(Dijkstra)
- A fixed order makes the run deterministic (stable animation, easy debugging, reproducible results).

04 BFS Algorithm

Application BFS to Path Planning

Pseudocode

```
1)  Q ← [start], parent[start] ← None
2)  while Q not empty:
3)          u ← dequeue(S)
4)  //      if u == goal:
5)  //              return reconstruct(parent, goal)
6)          for v in neighbors(u) in (N, E, S, W):
7)                  if valid(v) and v not visited:
8)                          mark v as visited
9)                          parent[v] ← u
10)                         enqueue(Q, v)
11)                         if v == goal:
12)                                 return reconstruct(parent, v)
13) return failure
```

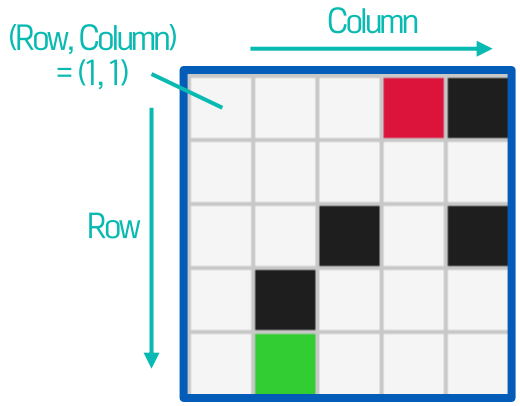
- ▶ init queue & parent
- ▶ expand stack(LIFO)
- ▶ check goal when dequeuing
- ▶ fixed order NESW
- ▶ treat v as visited
- ▶ check goal when enqueuing

In this lecture, we're going to check the goal when enqueuing.

※ Stopping on dequeuing can delay to find the goal.

04 BFS Algorithm

Application BFS to Path Planning – Example(4-connectivity)

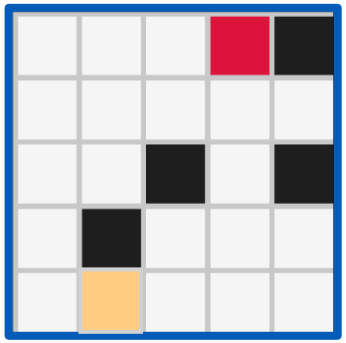


: unvisited
: visited
: obstacle
: frontier
: start
: goal

※ Rule of Transition : N → E → S → W

Step 1

- dequeue X
- enqueue (5, 2) // initialize
- frontier: {(5, 2)}
- parent[(5, 2)] = None

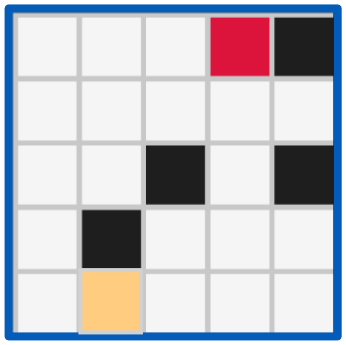


04 BFS Algorithm

Application BFS to Path Planning – Example(4-connectivity)

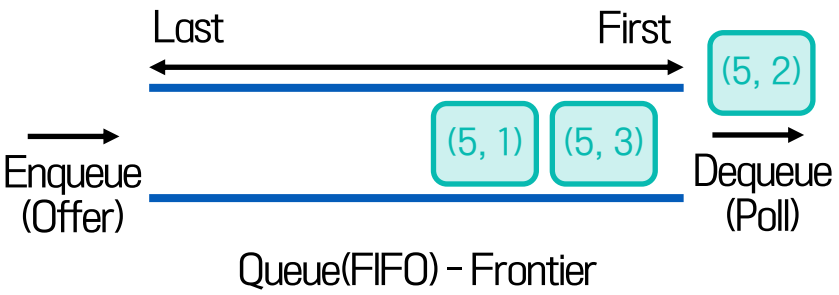
Step 1

- dequeue X
- enqueue (5, 2) // initialize
- frontier: {(5, 2)}
- parent[(5, 2)] = None



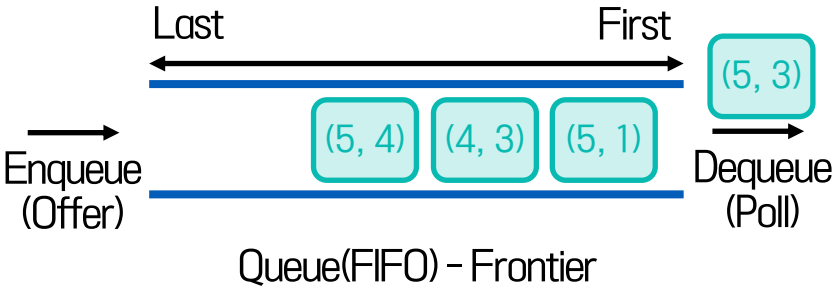
Step 2

- dequeue (5, 2)
- enqueue (5, 3), (5, 1) // NESW
- frontier: {(5, 3), (5, 1)}
- parent[(5, 3)] = (5, 2)
- parent[(5, 1)] = (5, 2)



Step 3

- dequeue (5, 3)
- enqueue (4, 3), (5, 4) // NESW
- frontier: {(5, 1), (4, 3), (5, 4)}
- parent[(4, 3)] = (5, 3)
- parent[(5, 4)] = (5, 3)

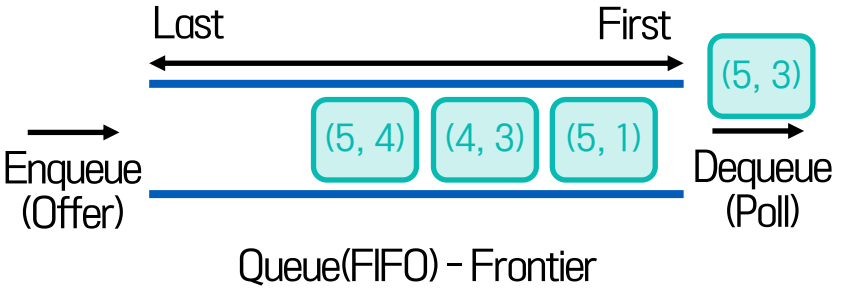


04 BFS Algorithm

Application BFS to Path Planning – Example(4-connectivity)

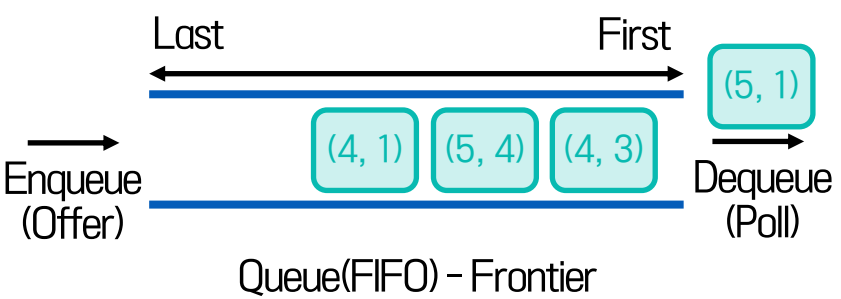
Step 3

- dequeue (5, 3)
- enqueue (4, 3), (5, 4) // NESW
- frontier: {(5, 1), (4, 3), (5, 4)}
- parent[(4, 3)] = (5, 3)
- parent[(5, 4)] = (5, 3)



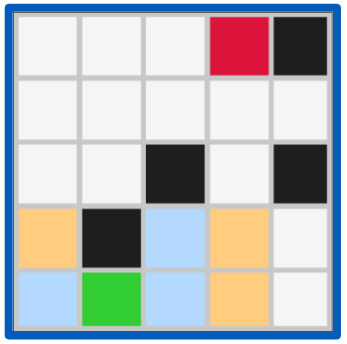
Step 4

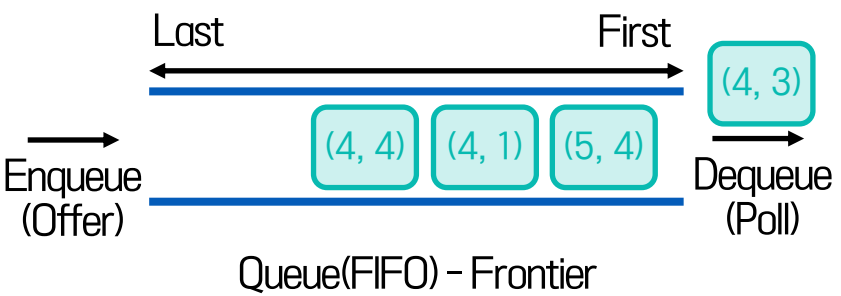
- dequeue (5, 1)
- enqueue (4, 1) // NESW
- frontier: {(4, 3), (5, 4), (4, 1)}
- parent[(4, 1)] = (5, 1)



Step 5

- dequeue (4, 3)
- enqueue (4, 4) // NESW
- frontier: {(5, 4), (4, 1), (4, 4)}
- parent[(4, 4)] = (4, 3)



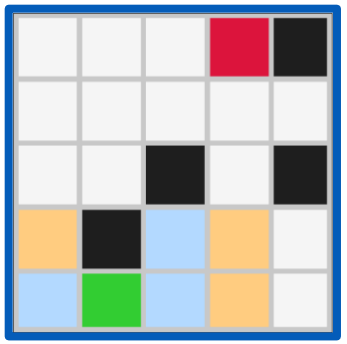


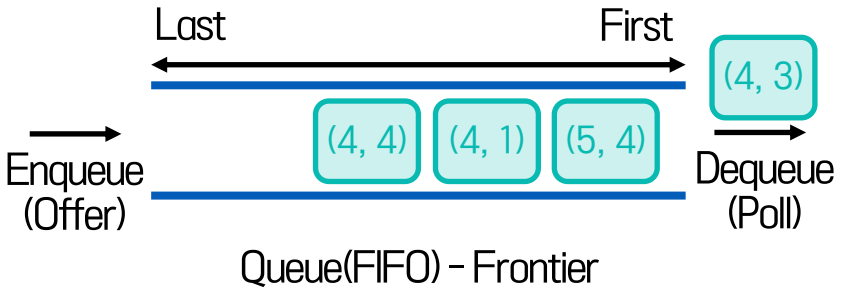
04 BFS Algorithm

Application BFS to Path Planning – Example(4-connectivity)

Step 5

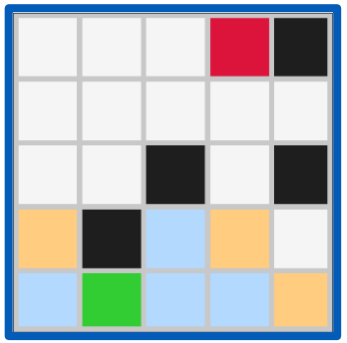
- dequeue (4, 3)
- enqueue (4, 4) // NESW
- frontier: {(5, 4), (4, 1), (4, 4)}
- parent[(4, 4)] = (4, 3)

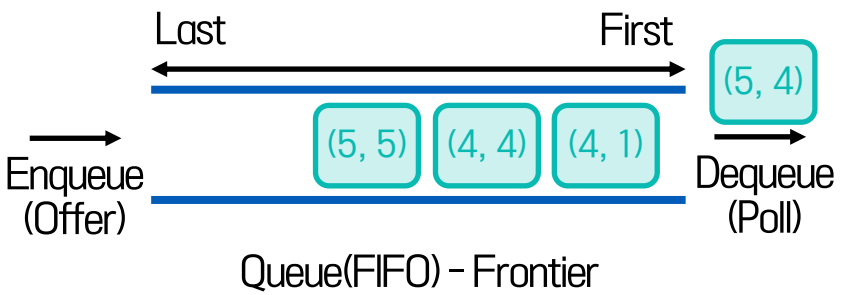




Step 6

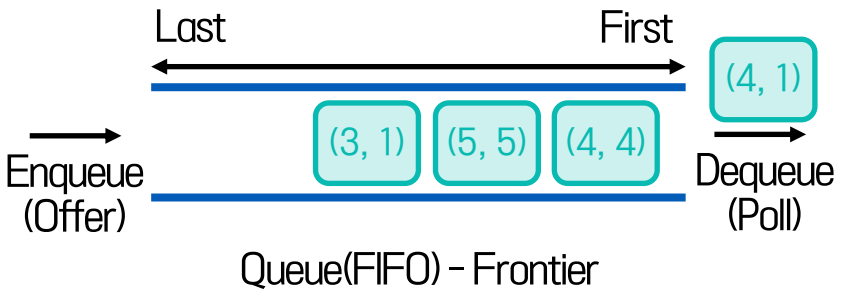
- dequeue (5, 4)
- enqueue (5, 5) // NESW
- frontier: {(4, 1), (4, 4), (5, 5)}
- parent[(5, 5)] = (5, 4)





Step 7

- dequeue (4, 1)
- enqueue (3, 1) // NESW
- frontier: {(4, 4), (5, 5), (3, 1)}
- parent[(3, 1)] = (4, 1)

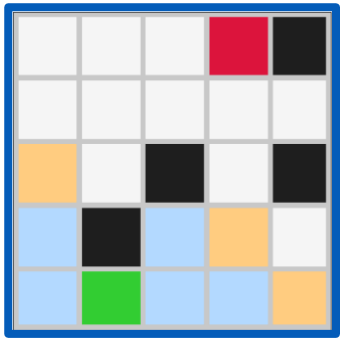


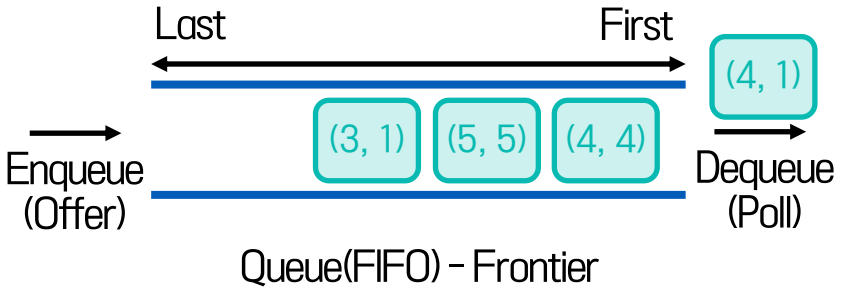
04 BFS Algorithm

Application BFS to Path Planning – Example(4-connectivity)

Step 7

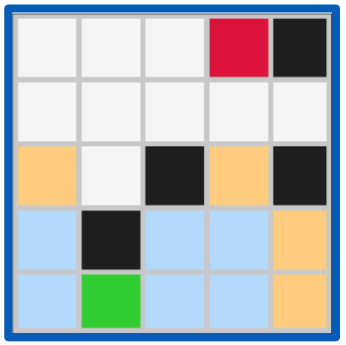
- dequeue (4, 1)
- enqueue (3, 1) // NESW
- frontier: {(4, 4), (5, 5), (3, 1)}
- parent[(3, 1)] = (4, 1)

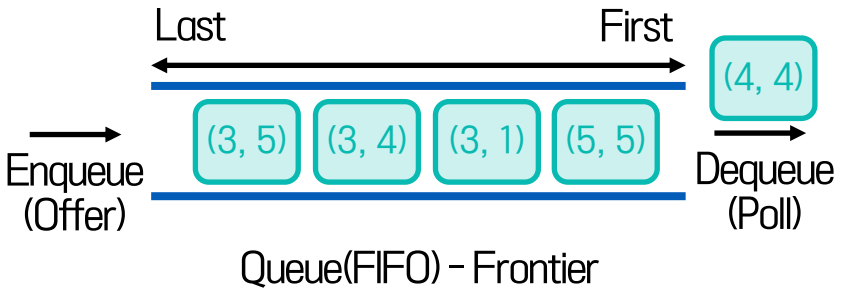




Step 8

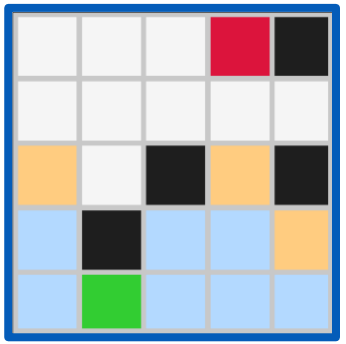
- dequeue (4, 4)
- enqueue (3, 4), (4, 5) // NESW
- frontier: {(5, 5), (3, 1), (3, 4), (3, 5)}
- parent[(3, 4)] = (4, 4)
- parent[(4, 5)] = (4, 4)

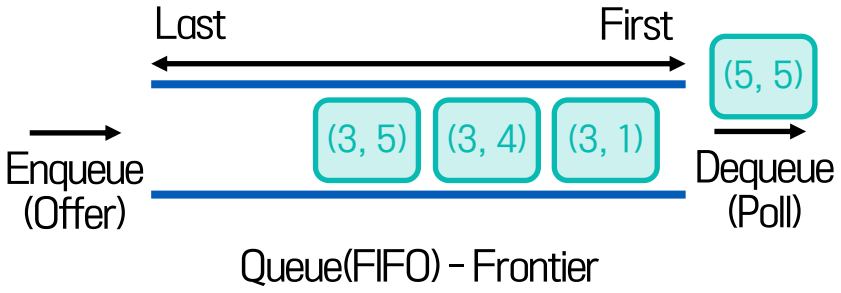




Step 9

- dequeue (5, 5)
- enqueue X // there isn't non-visited.
- frontier: {(3, 1), (3, 4), (3, 5)}



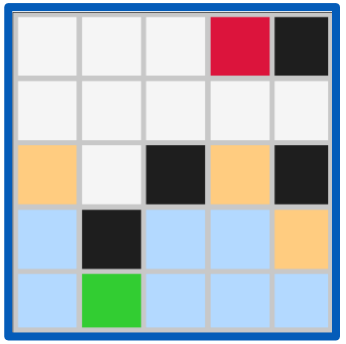


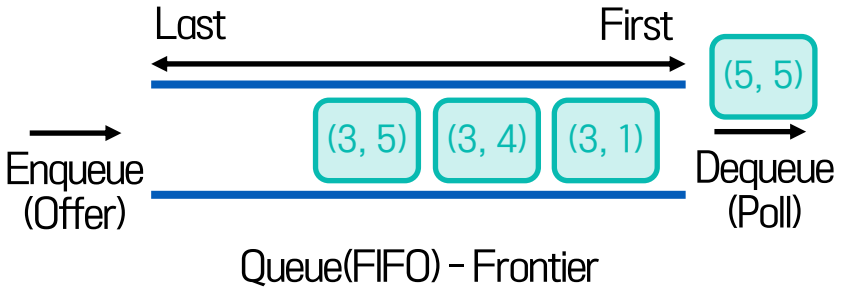
04 BFS Algorithm

Application BFS to Path Planning – Example(4-connectivity)

Step 9

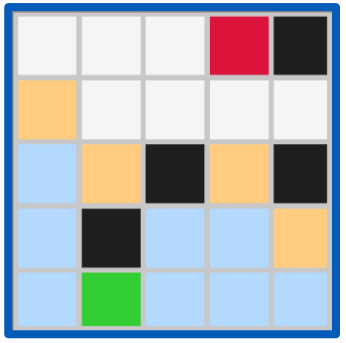
- dequeue (5, 5)
- enqueue X // there isn't non-visited.
- frontier: {(3, 1), (3, 4), (3, 5)}

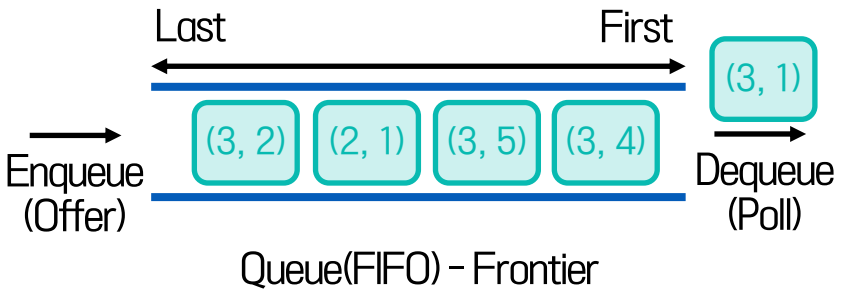




Step 10

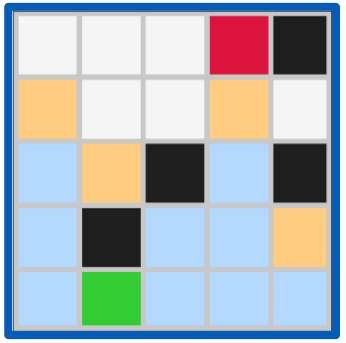
- dequeue (3, 1)
- enqueue (2, 1), (3, 2) // NESW
- frontier: {(3, 4), (3, 5), (2, 1), (3, 2)}
- parent[(2, 1)] = (3, 1)
- parent[(3, 2)] = (3, 1)

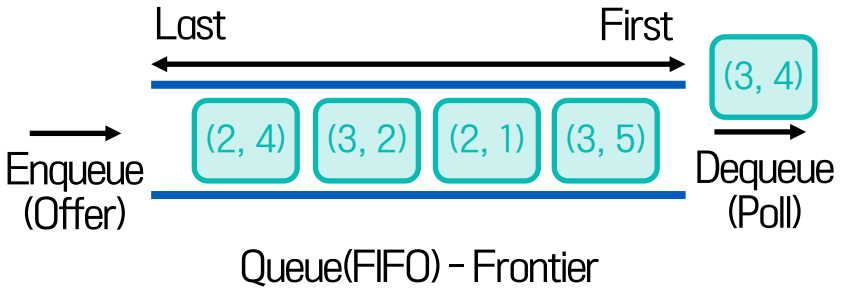




Step 11

- dequeue (3, 4)
- enqueue (2, 4) // NESW
- frontier: {(3, 5), (2, 1), (3, 2), (2, 4)}
- parent[(2, 4)] = (3, 4)



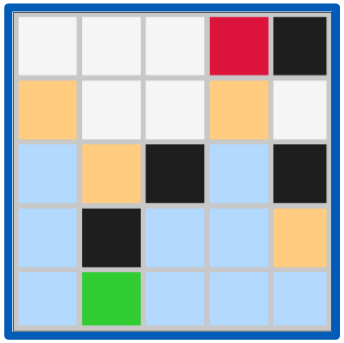


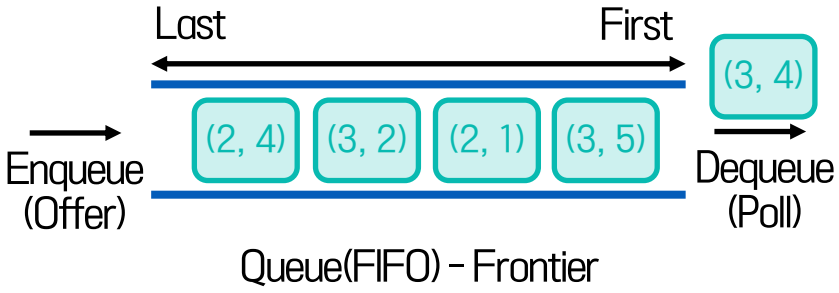
04 BFS Algorithm

Application BFS to Path Planning – Example(4-connectivity)

Step 11

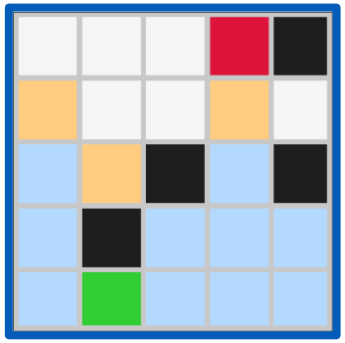
- dequeue (3, 4)
- enqueue (2, 4) // NESW
- frontier: {(3, 5), (2, 1), (3, 2), (2, 4)}
- parent[(2, 4)] = (3, 4)

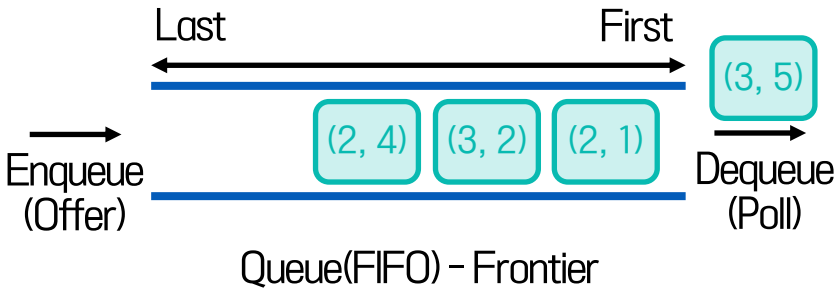




Step 12

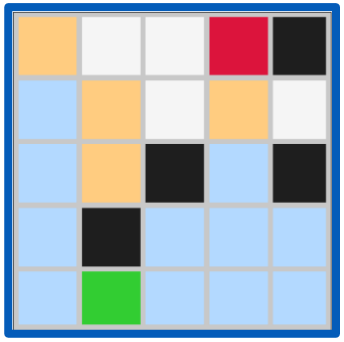
- dequeue (3, 5)
- enqueue X // there isn't non-visited.
- frontier: {(2, 1), (3, 2), (2, 4)}

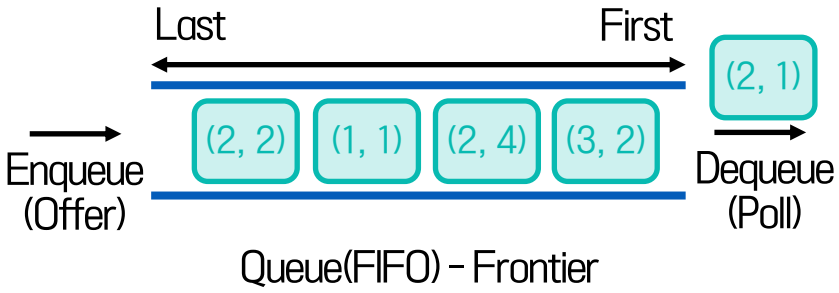




Step 13

- dequeue (2, 1)
- enqueue (1, 1), (2, 2) // NESW
- frontier: {(3, 2), (2, 4), (1, 1), (2, 2)}
- parent[(1, 1)] = (2, 1)
- parent[(2, 2)] = (2, 1)



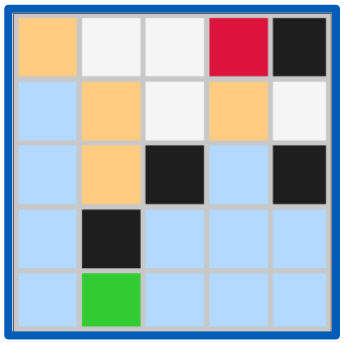


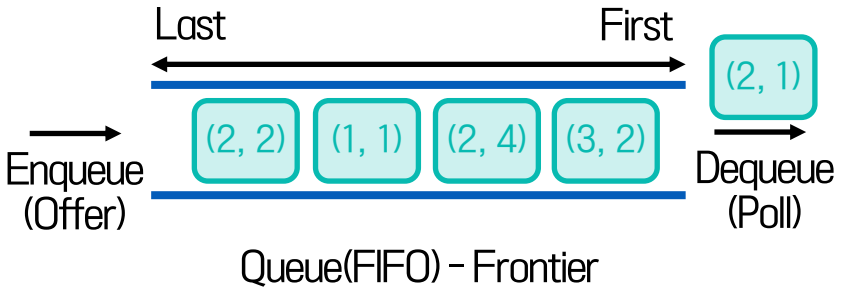
04 BFS Algorithm

Application BFS to Path Planning – Example(4-connectivity)

Step 13

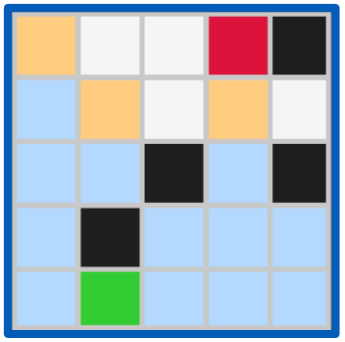
- dequeue (2, 1)
- enqueue (1, 1), (2, 2) // NESW
- frontier: {(3, 2), (2, 4), (1, 1), (2, 2)}
- parent[(1, 1)] = (2, 1)
- parent[(2, 2)] = (2, 1)

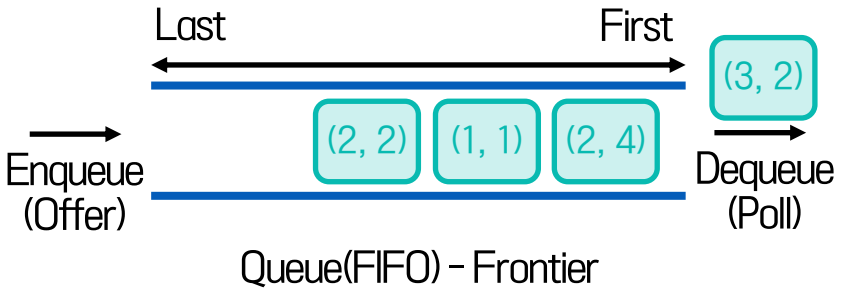




Step 14

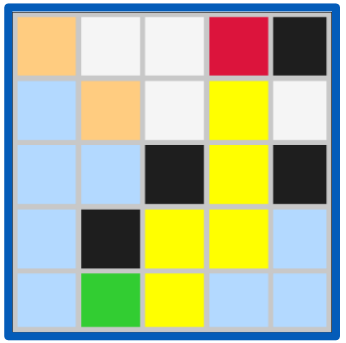
- dequeue (3, 2)
- enqueue X // there isn't non-visited.
- frontier: {(2, 4), (1, 1), (2, 2)}

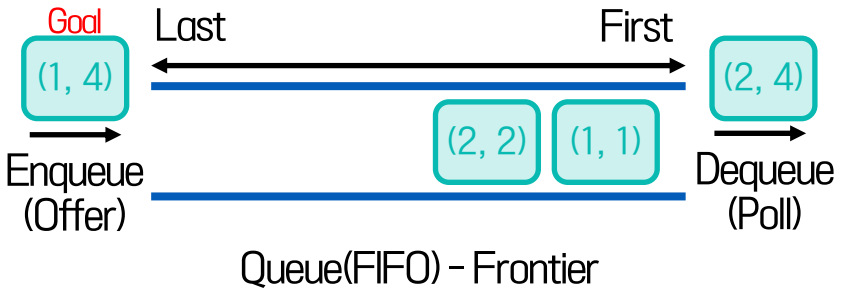




Step 15 - Final

- dequeue (2, 4)
- enqueue (1, 4) // North First & Check Goal
- frontier: {(1, 1), (2, 2), (1, 4)}
- (1, 4) == goal
  → Reconstruct Path from Parent

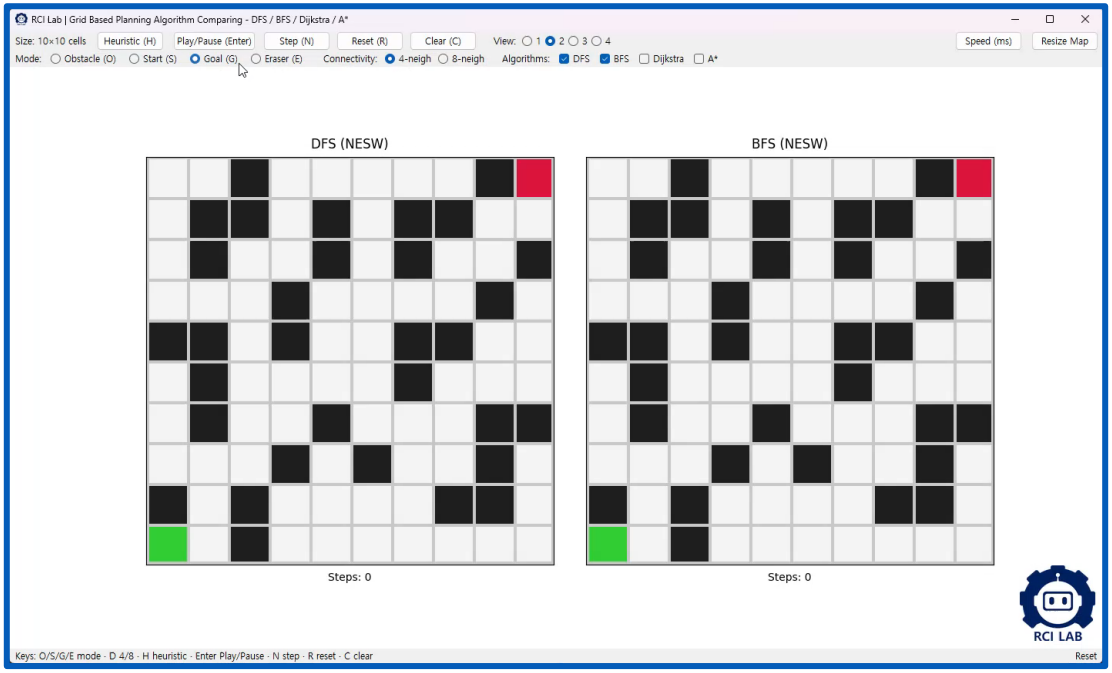




04 BFS Algorithm

DFS vs. BFS

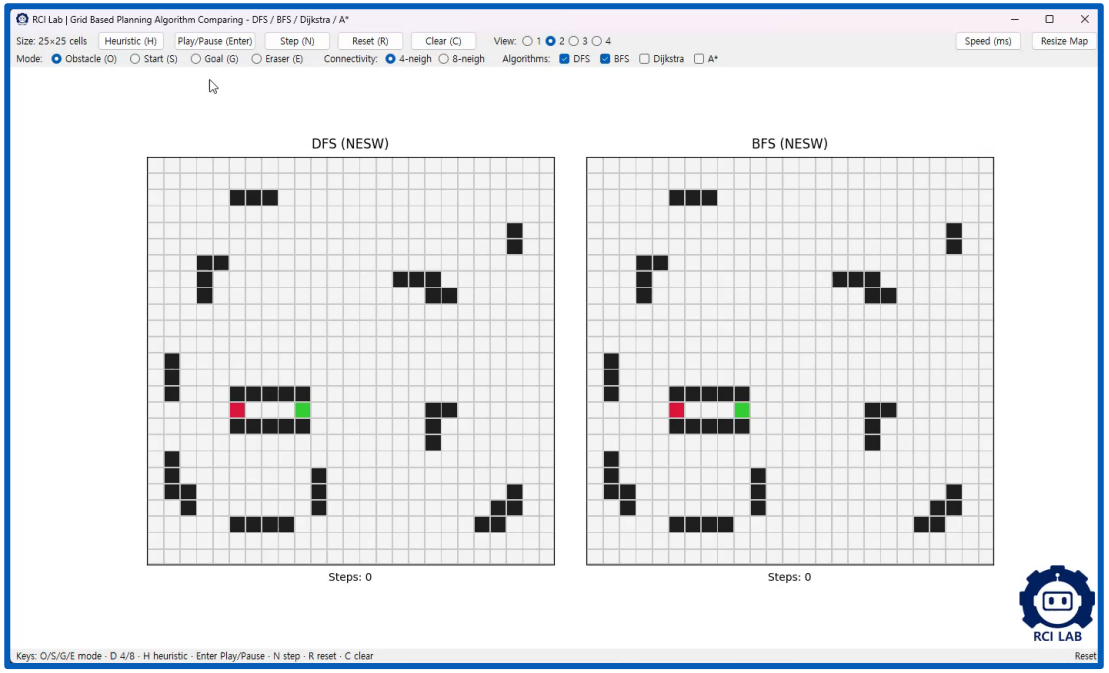
Algorithm	Pros	Cons
DFS	• Simple to implement • Uses relatively little memory	• Not complete (Can go infinitely deep in very large graphs) • Does not guarantee the shortest path
BFS	• On an unweighted graph(unit-cost), it guarantees the shortest path • Complete for finite graphs	• Needs to store many nodes at the same depth → higher memory usage than DFS



04 BFS Algorithm

DFS vs. BFS

Algorithm	Pros	Cons
DFS	• Simple to implement • Uses relatively little memory	• Not complete (Can go infinitely deep in very large graphs) • Does not guarantee the shortest path
BFS	• On an unweighted graph(unit-cost), it guarantees the shortest path • Complete for finite graphs	• Needs to store many nodes at the same depth → higher memory usage that DFS

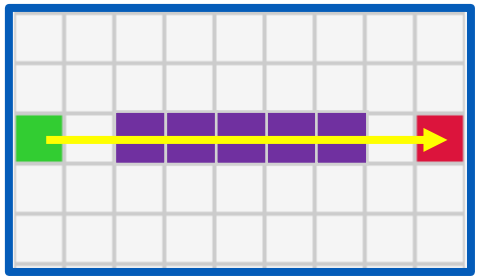


04 BFS Algorithm

Limitations of BFS Algorithm

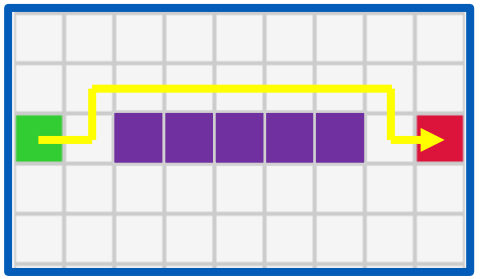
- With $\sqrt{2}$ diagonals or costmaps, fewer steps can still cost more. → Not cost-optimal!
- Level expansion can create very large frontiers in open spaces. → Memory blow-up
- BFS must traverse all shallower(low-level) layers first. → Inefficient for distant goals

: free (cost = 1) : puddle (cost = 5)



BFS

total cost: 28



Dijkstra (4-connectivity)

total cost: 10

↓

cost-optimal

04 BFS Algorithm

Uniform cost vs. Non-uniform cost

Item	BFS (Unit-Cost)	Dijkstra / Non-Unit
Container	Queue (FIFO)	Priority Queue (min g)
Visit Finalization	Mark visited on enqueue	Mark visited on dequeue
Goal Test	Stop on enqueue (on discovery) ※ Stopping on dequeue is also valid, but it can delay to stop.	Stop on dequeue (when selected)
Guarantee	Hop(edge)-shortest	Cost-optimal
Wavefront pattern	Concentric levels	Circular/Octagonal cost contours
NESW role	Order within a level	Order among equal-g candidates

Thank you