Beyond Classical Search

"Artificial Intelligence: A Modern Approach", Chapter 4

Outline

- Local search algorithms
  - Hill-climbing search
  - Simulated annealing search
  - Local beam search
  - Genetic algorithms
- Searching in more complex environments
  - Non-deterministic environments
  - Partially observable environments
  - Unknown environments

Problem types

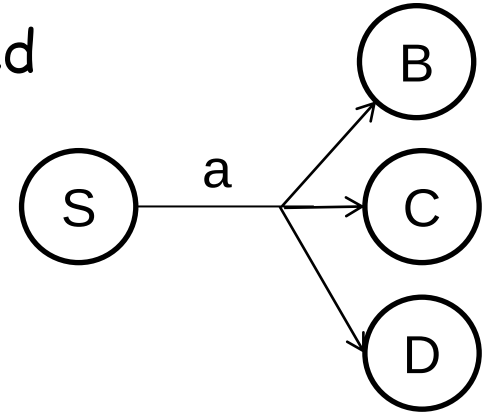
- Deterministic and fully observable (single-state problem)
  - Agent knows exactly its state even after a sequence of actions
  - Solution is a sequence
- Non-observable or sensor-less (conformant problem)
  - Agent's percepts provide no information at all
  - Solution is a sequence
- Nondeterministic and/or partially observable (contingency problem)
  - Percepts provide new information about current state
  - Solution can be a contingency plan (tree or strategy) and not a sequence
  - Often interleave search and execution
- Unknown state space (exploration problem)

Non-deterministic or partially observable

- Perception become useful
  - Partially observable
    - To narrow down the set of possible states for the agent
  - Non-deterministic
    - To show which outcome of the action has occurred
- Future percepts can not be determined in advance
- Solution is a contingency plan
  - A tree composed of nested if-then-else statements
  - What to do depending on what percepts are received

Searching with non-deterministic actions

- In non-deterministic environments, the result of an action can vary
  - Future percepts can specify which outcome has occurred

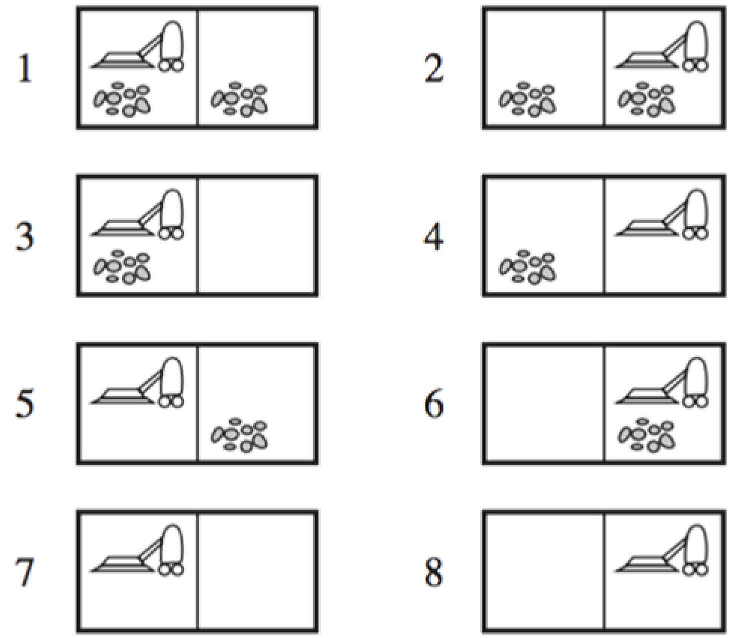


- Generalizing the transition function
  - Results: $S \times A \rightarrow 2^S$ instead of Results: $S \times A \rightarrow S$

- Search tree will be an AND-OR tree
  - Solution will be a sub-tree containing a contingency plan (nested if-then-else statements)

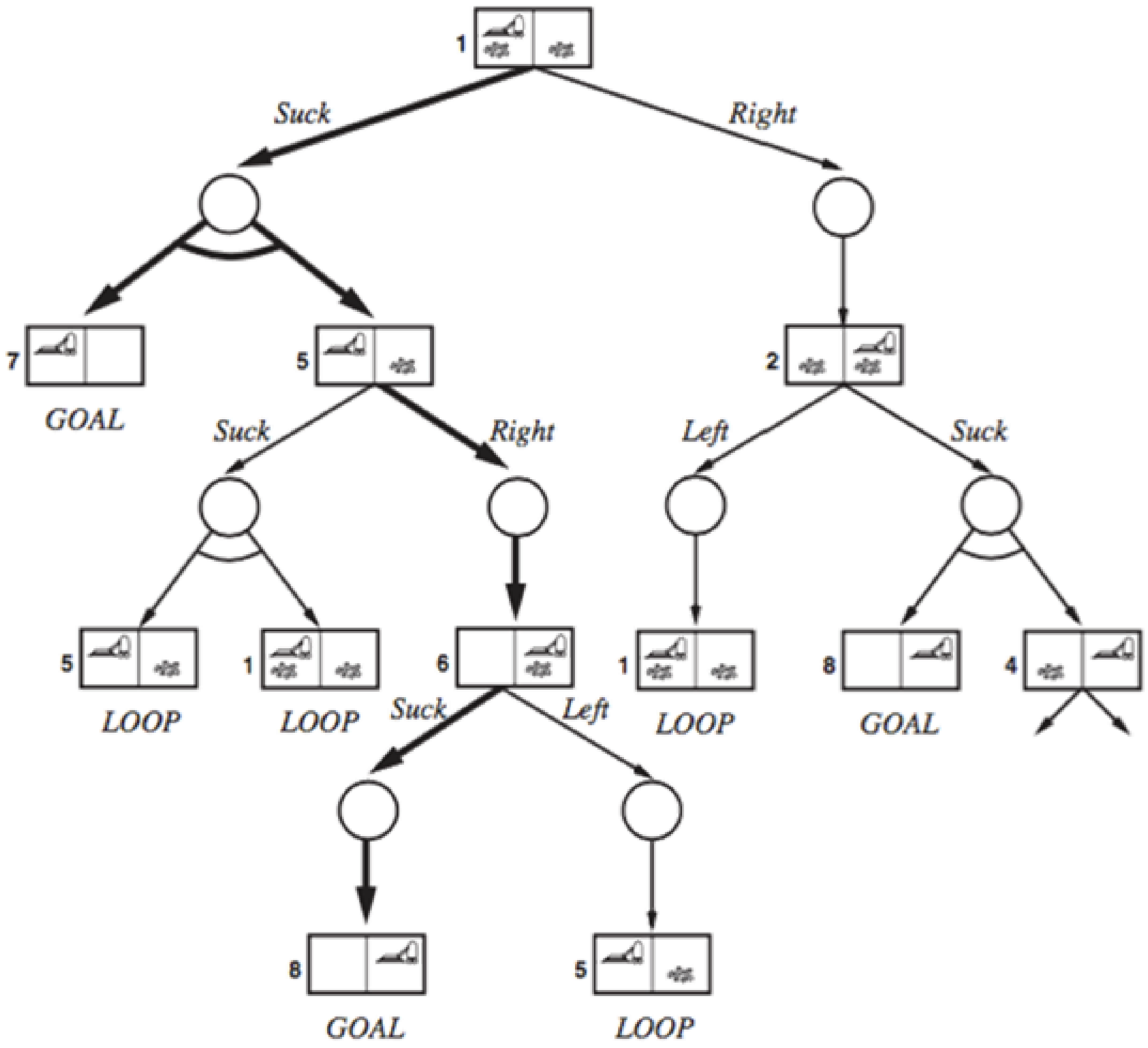
The erratic vacuum world

- States
  - {1,2,…,8}
- Actions
  - {Left, Right, Suck}
- Goal
  - {7} or {8}
- Non-deterministic
  - When sucking a dirty square, it cleans it and sometimes cleans up dirt in an adjacent square
    - Results{1,suck}={5,7}
  - When sucking a clean square, it sometimes deposits dirt on the carpet

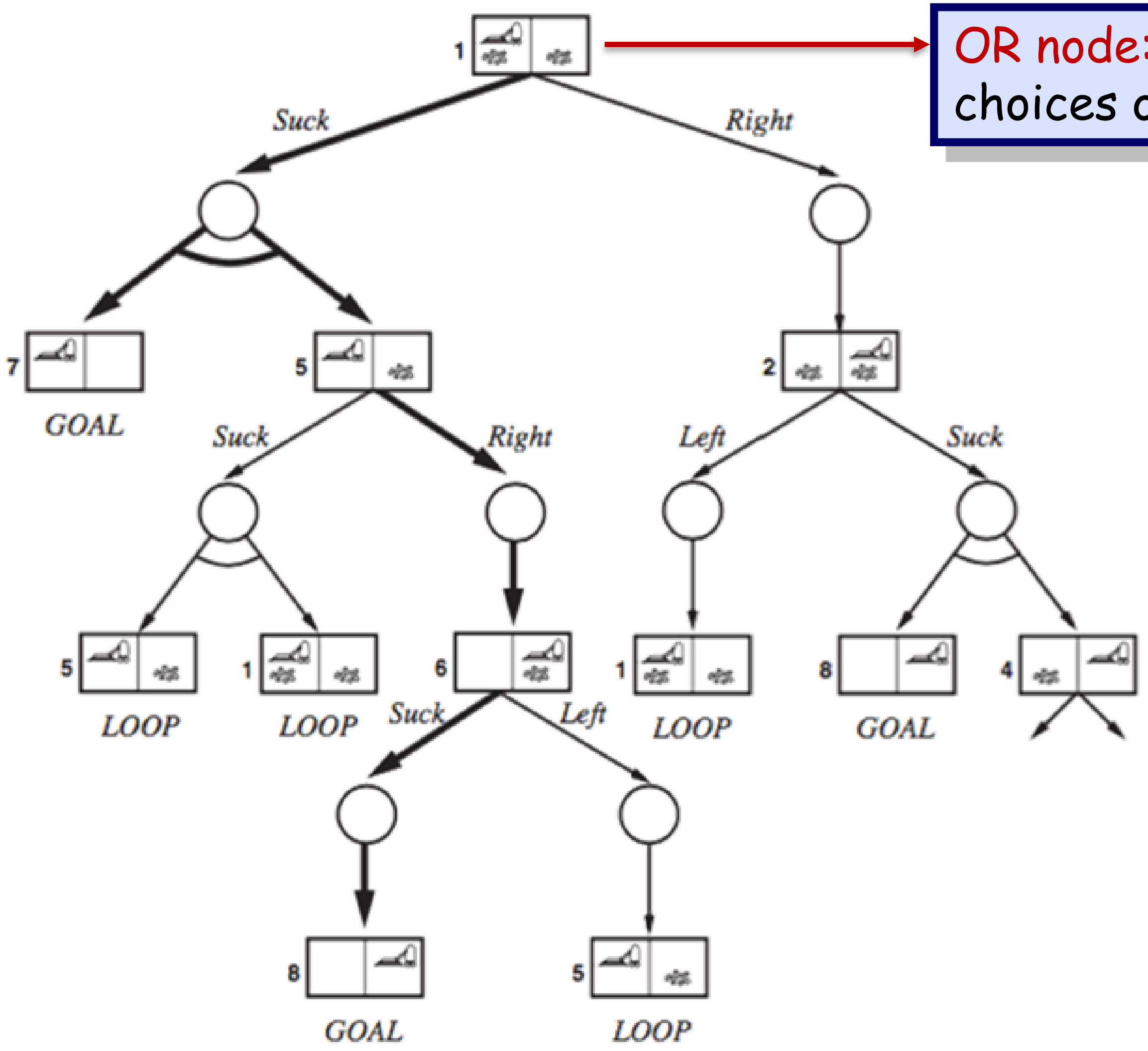


State=1
Suck
if State=5 **then** [Right, Suck]
else []

AND-OR search tree



AND-OR search tree

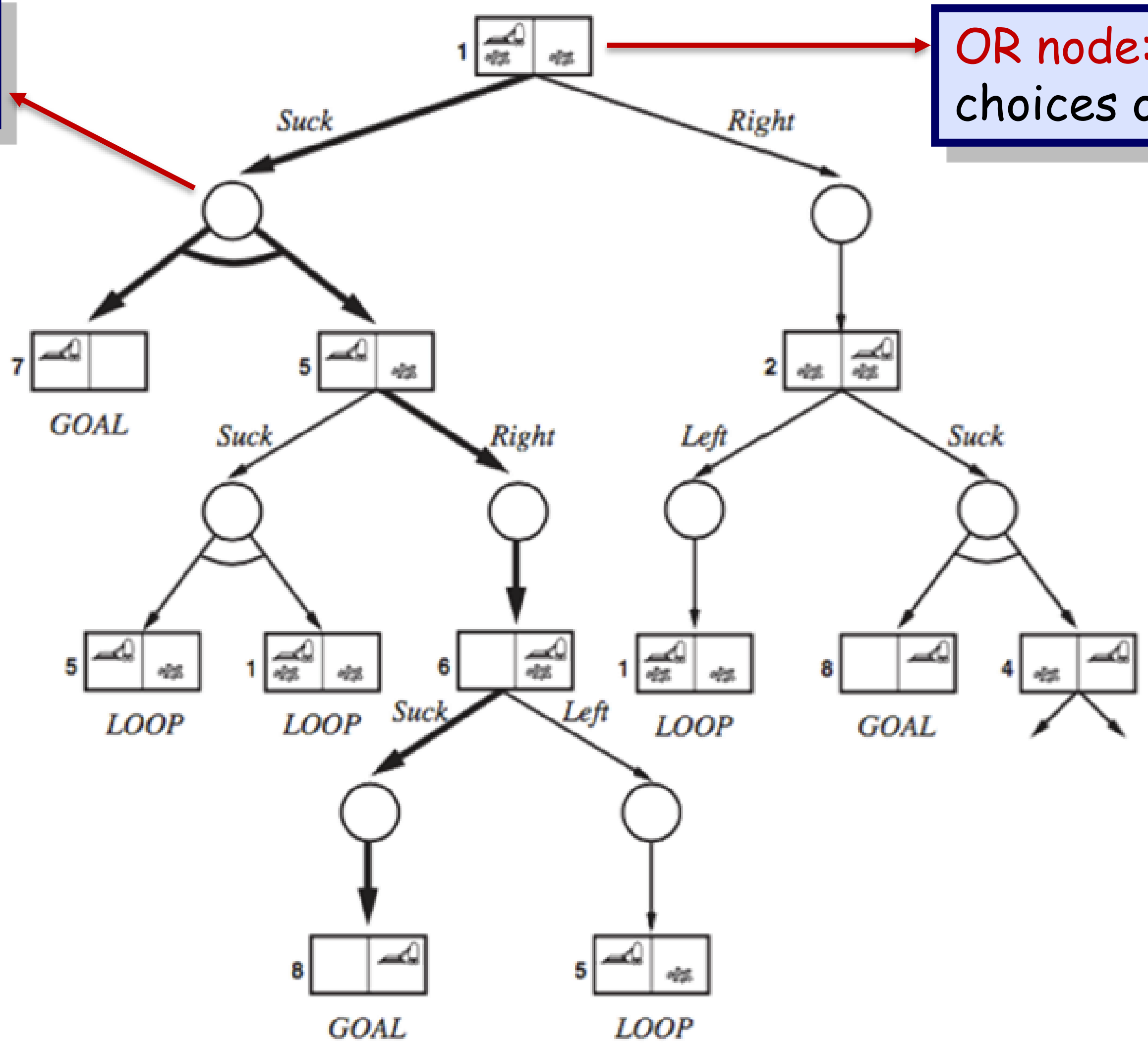


OR node: agent's choices of actions

AND-OR search tree

AND node: environment's choice of outcome

OR node: agent's choices of actions



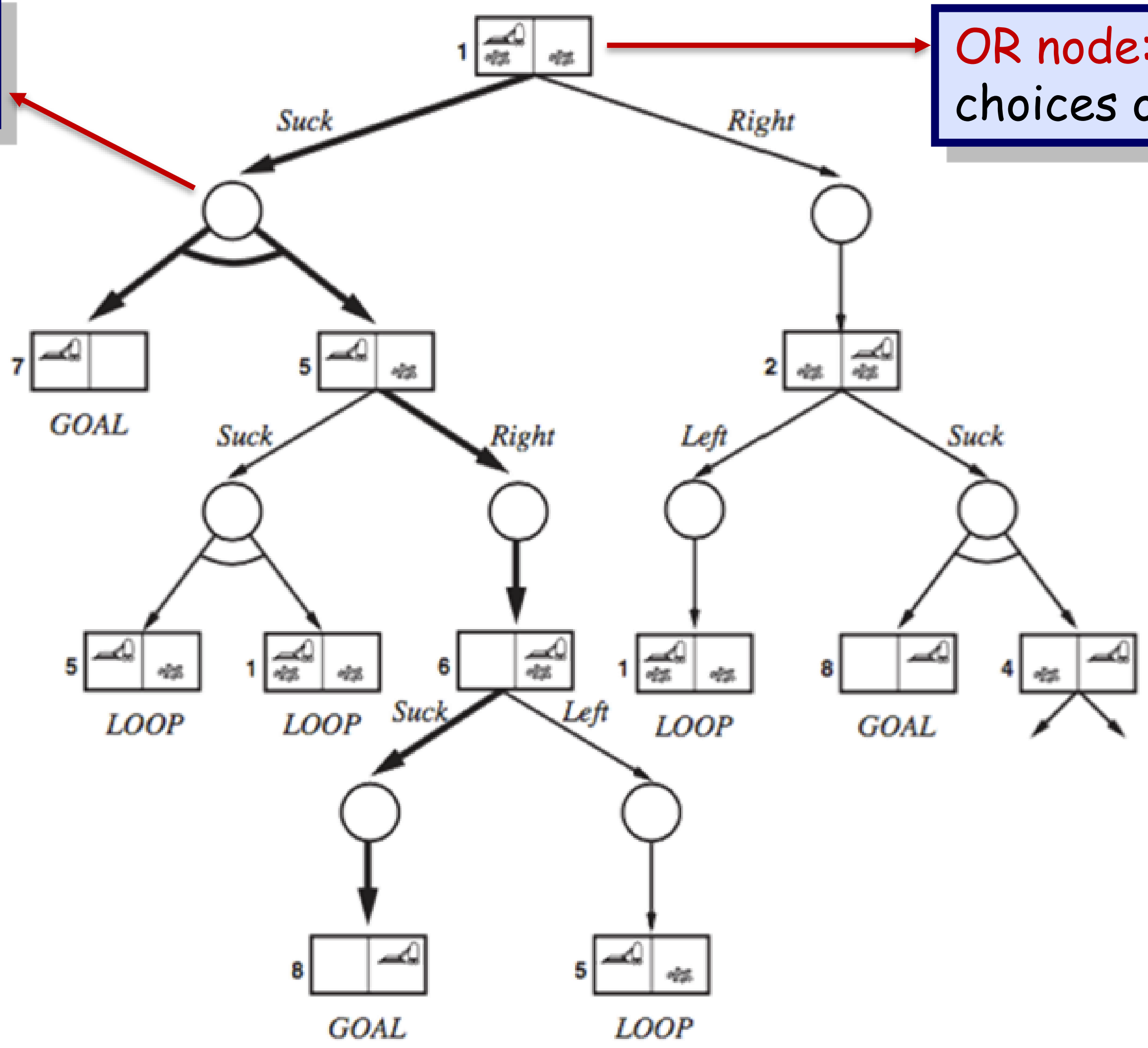
AND-OR search tree

AND node: environment's choice of outcome

OR node: agent's choices of actions

Solution for AND-OR search problem is a sub-tree that:

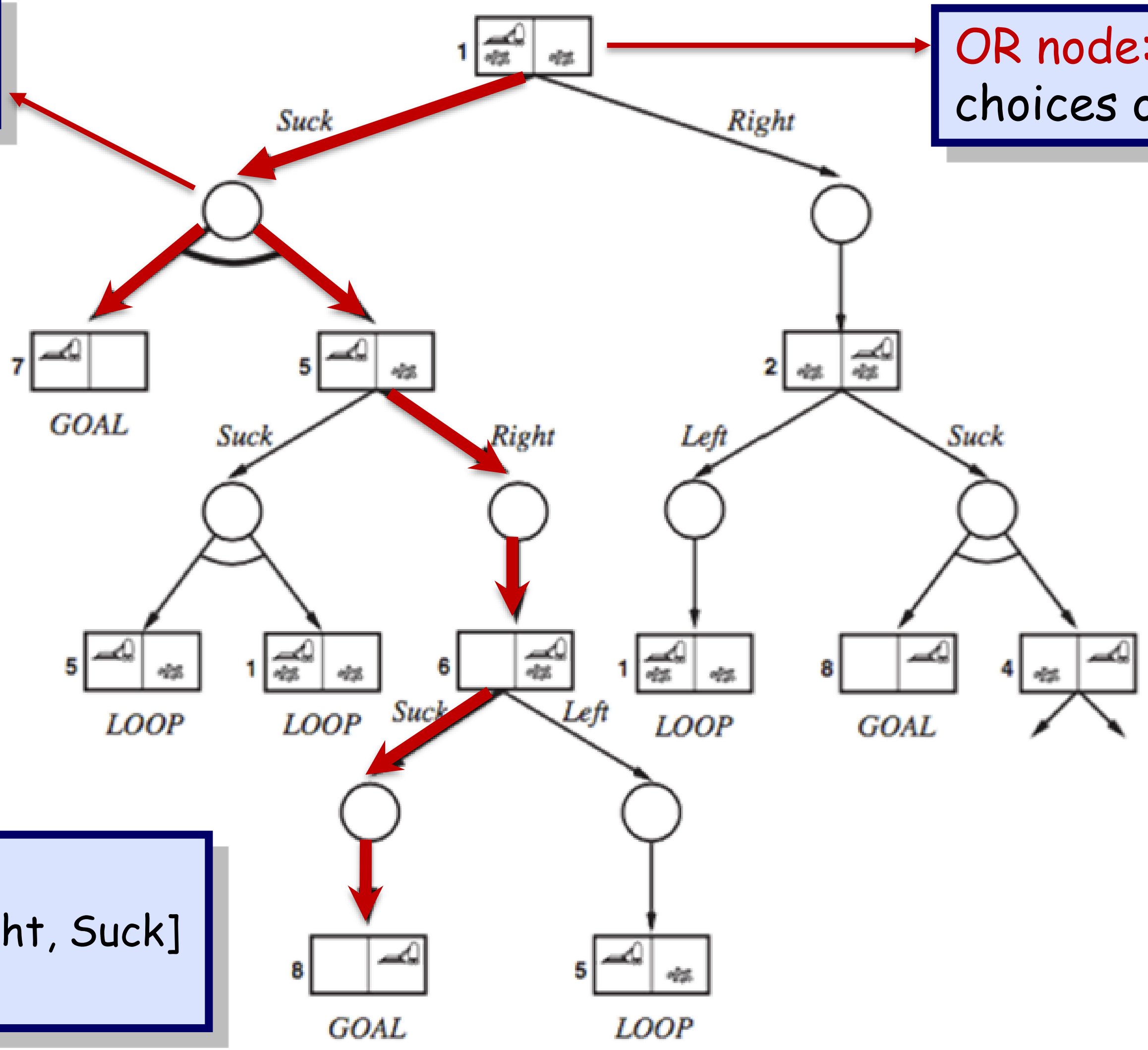
- specifies one action at each OR node
- includes every outcome at each AND node
- **has a goal node at every leaf**



AND-OR search tree

AND node: environment's choice of outcome

OR node: agent's choices of actions



Solution for AND-OR search problem is a sub-tree that:

- specifies one action at each OR node
- includes every outcome at each AND node
- **has a goal node at every leaf**

Suck
if State=5 **then** [Right, Suck]
else []

AND-OR depth first graph search

```
function AND-OR-GRAPH-SEARCH(problem) returns a conditional plan or failure
   OR-SEARCH(problem.INITIAL-STATE, problem, [])
```

```
function OR-SEARCH(state, problem, path) returns a conditional plan or failure
    if problem.GOAL-TEST(state) then return the empty plan
    if state is on path then return failure
    for each action in problem.ACTIONS(state) do
        plan=AND-SEARCH(RESULTS(state, action), problem, [state | path])
        if plan ≠ failure then return [action | plan]
    return failure
```

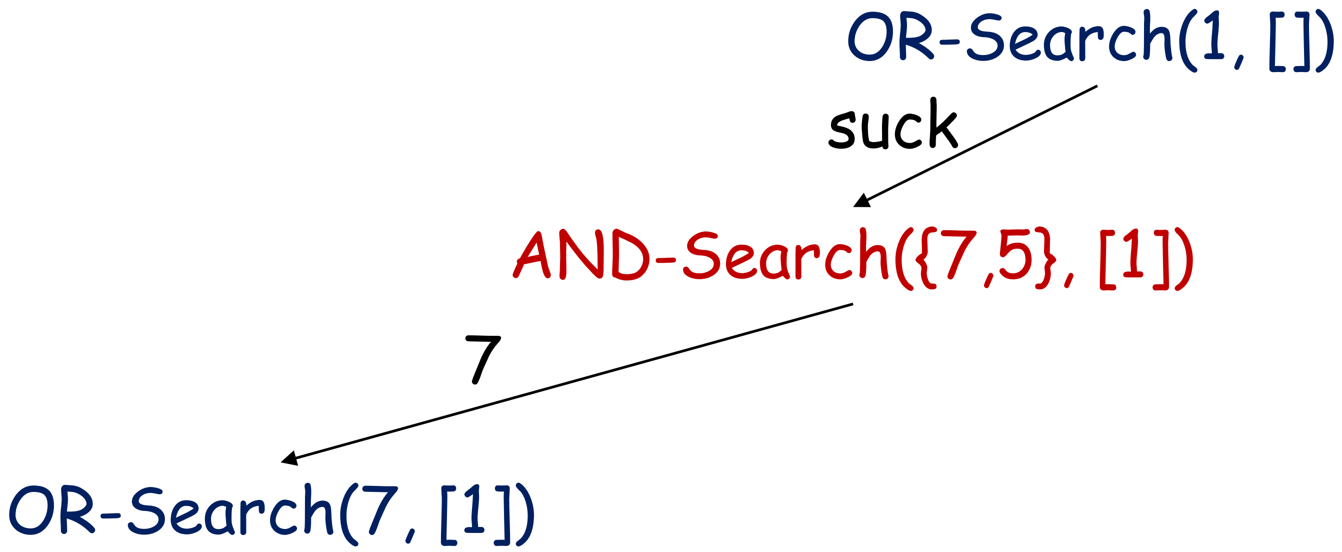
```
function AND-SEARCH(state, problem, path) returns a conditional plan or failure
    for each s_i in states do
        plan_i = OR-SEARCH(s_i, problem, path)
        if plan_i = failure then return failure
    return [if s_1 then plan_1 else if s_2 then plan_2 else … if s_{n-1} then plan_{n-1} else plan_n]
```

OR-Search(1, [])

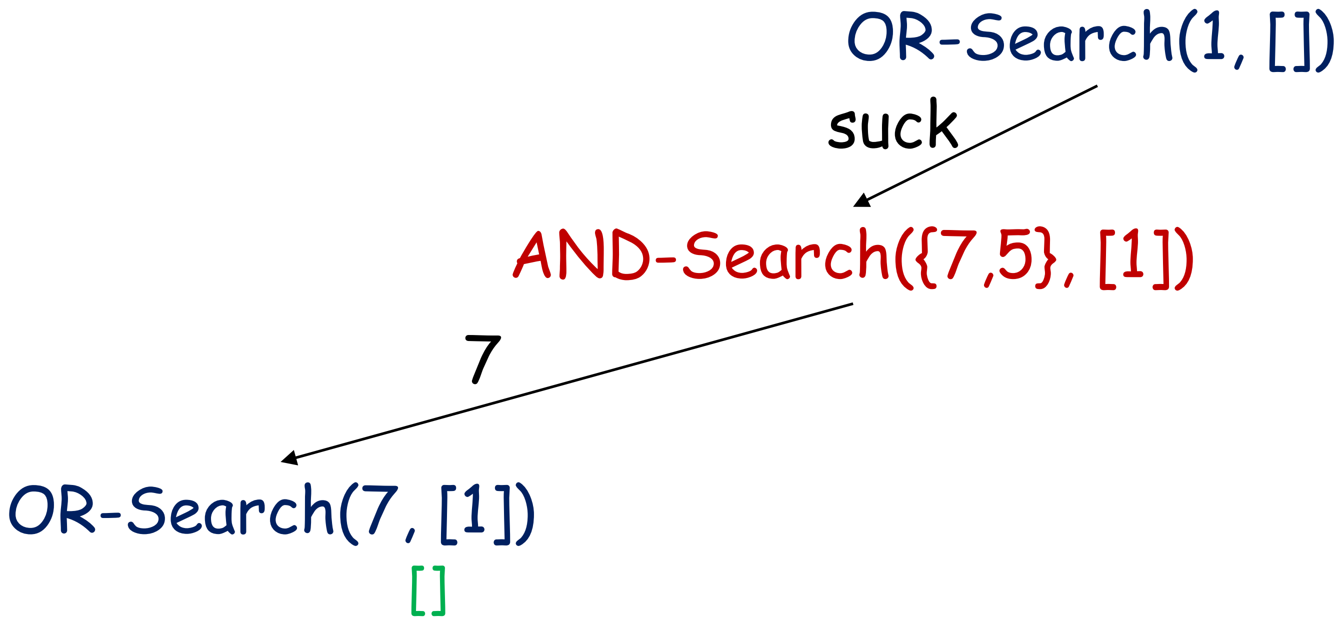
OR-Search(1, [])

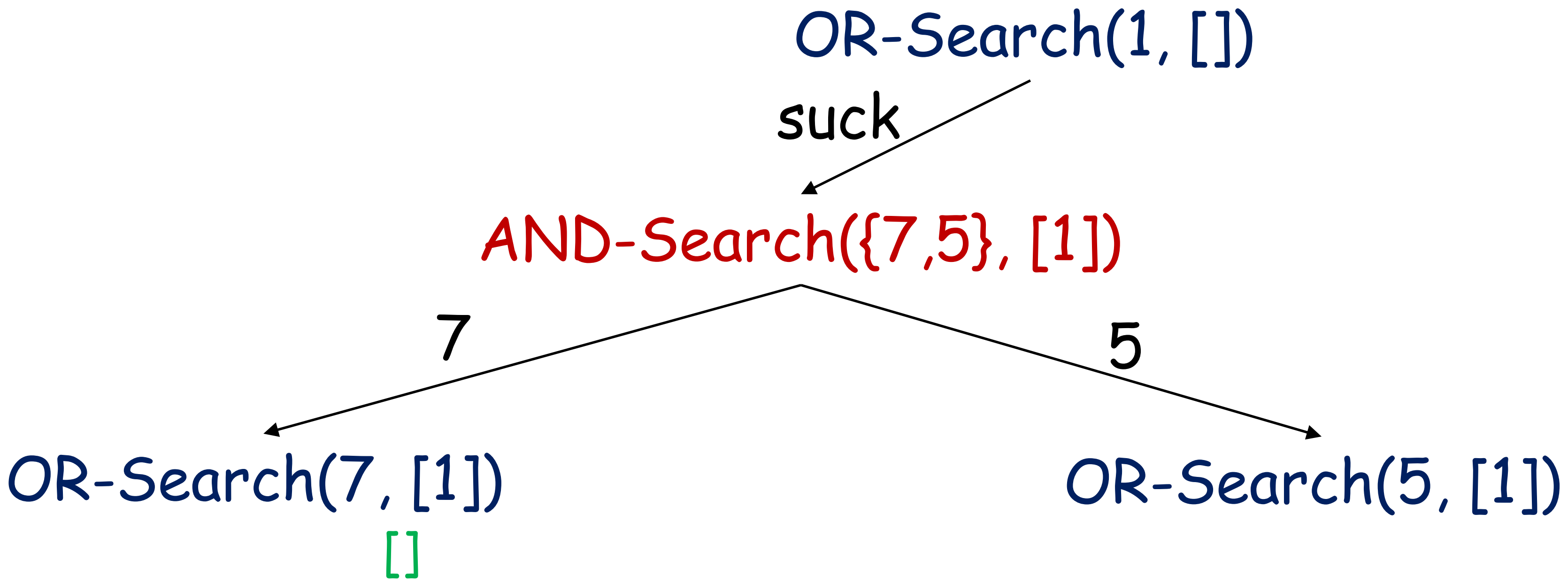
suck

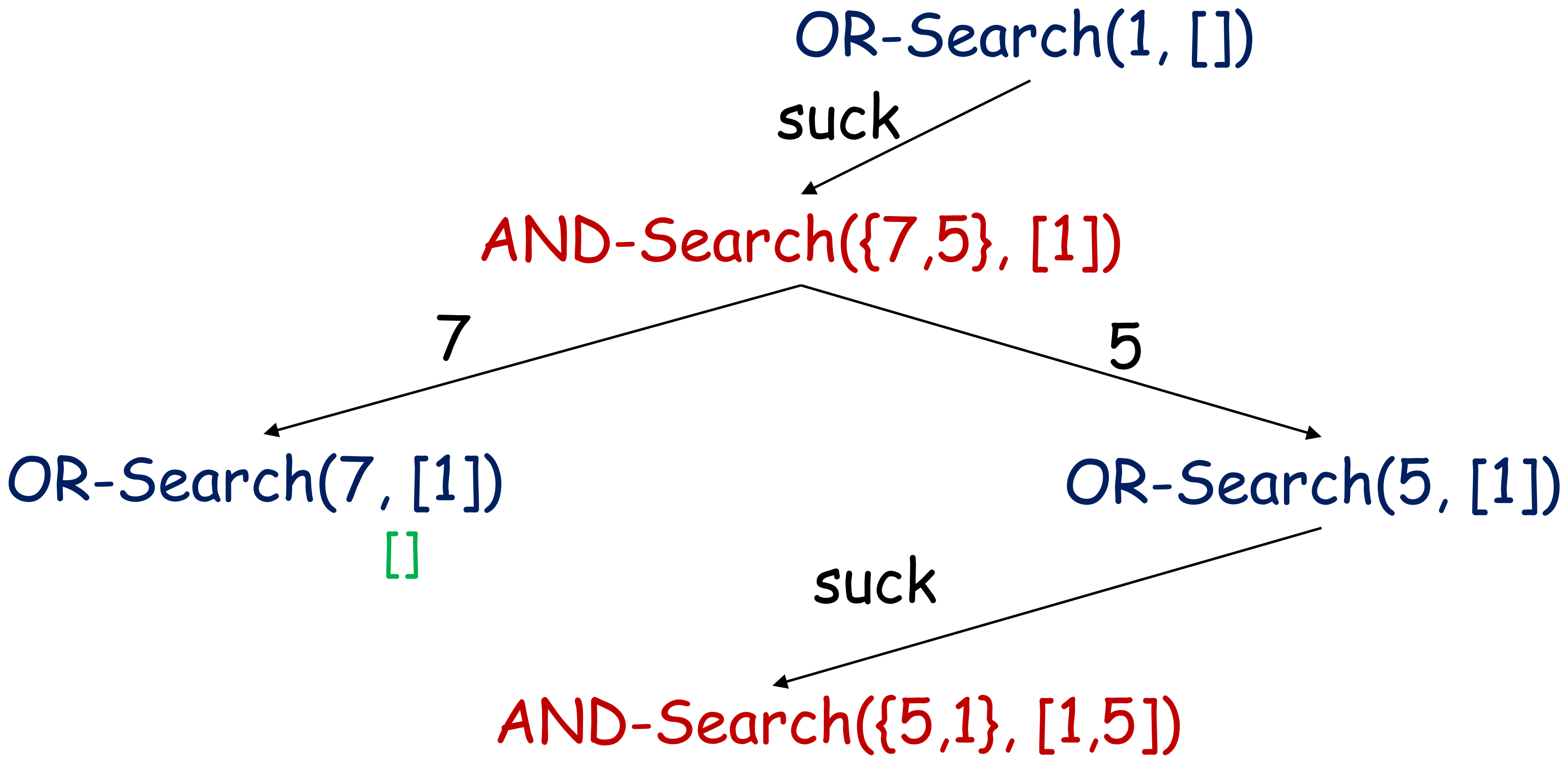
AND-Search({7,5}, [1])

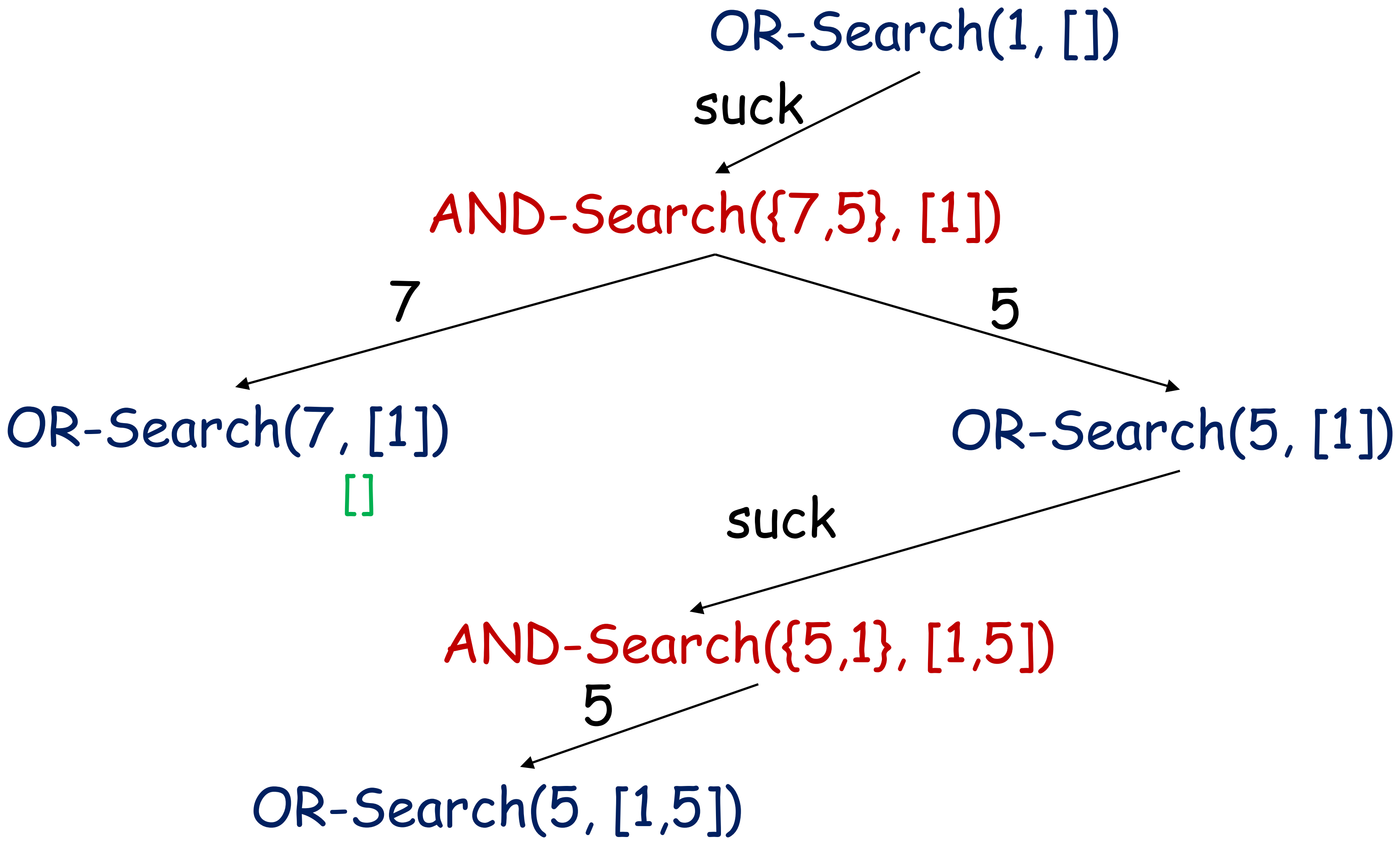


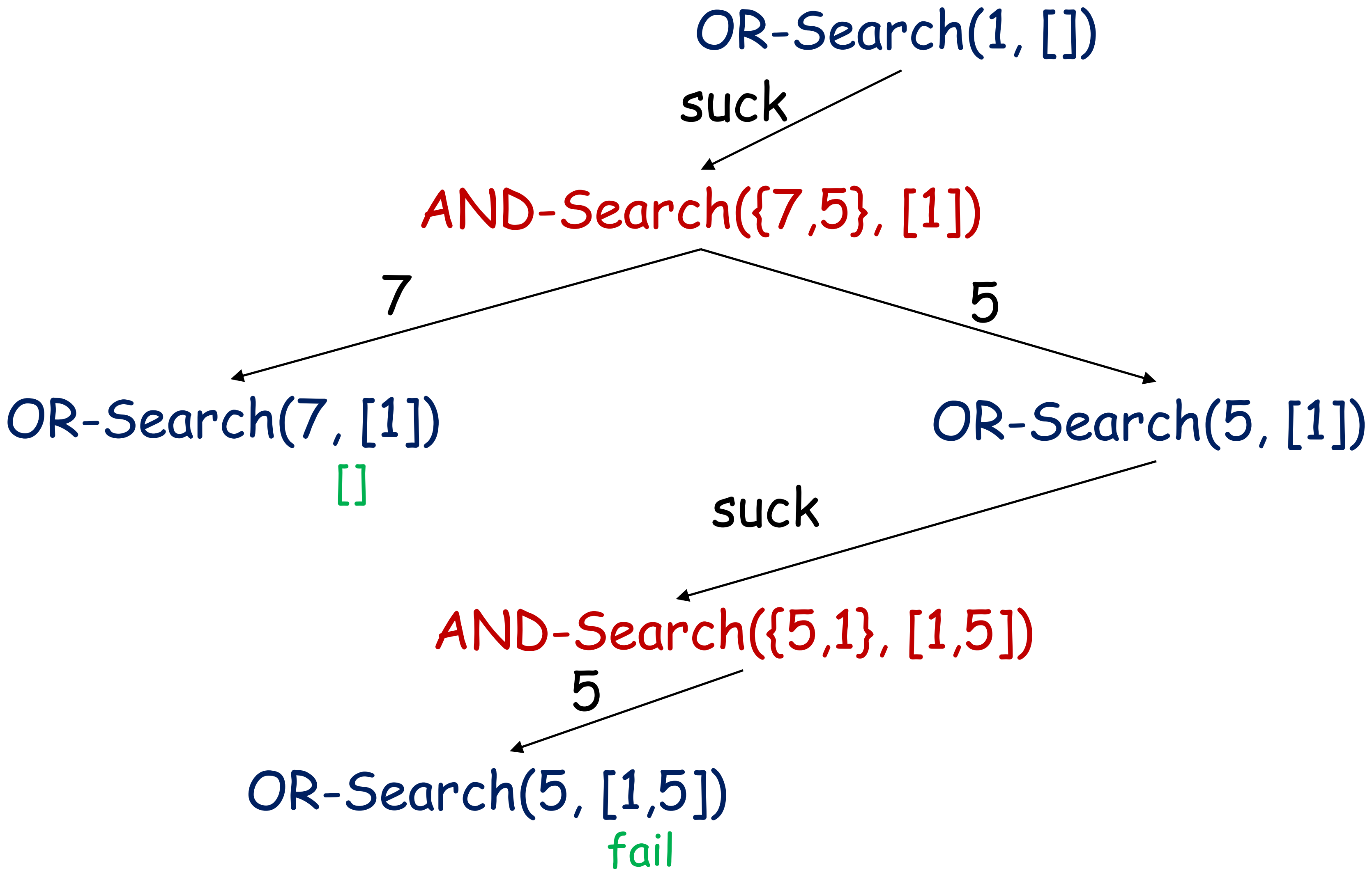
OR-Search(1, [])
suck
AND-Search({7,5}, [1])
7
OR-Search(7, [1])

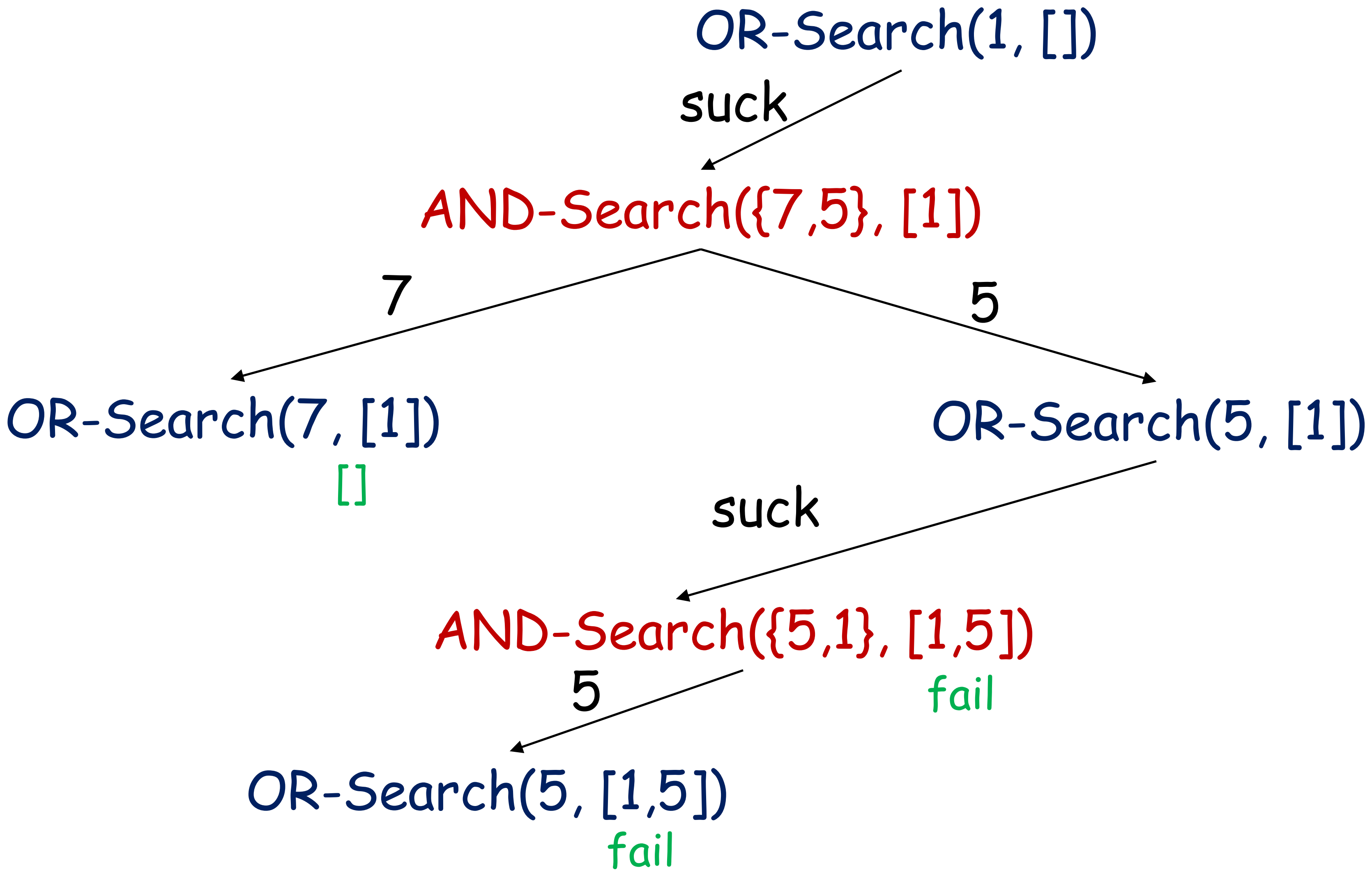

OR-Search(1, [])
suck
AND-Search({7,5}, [1])
7
OR-Search(7, [1])
[]

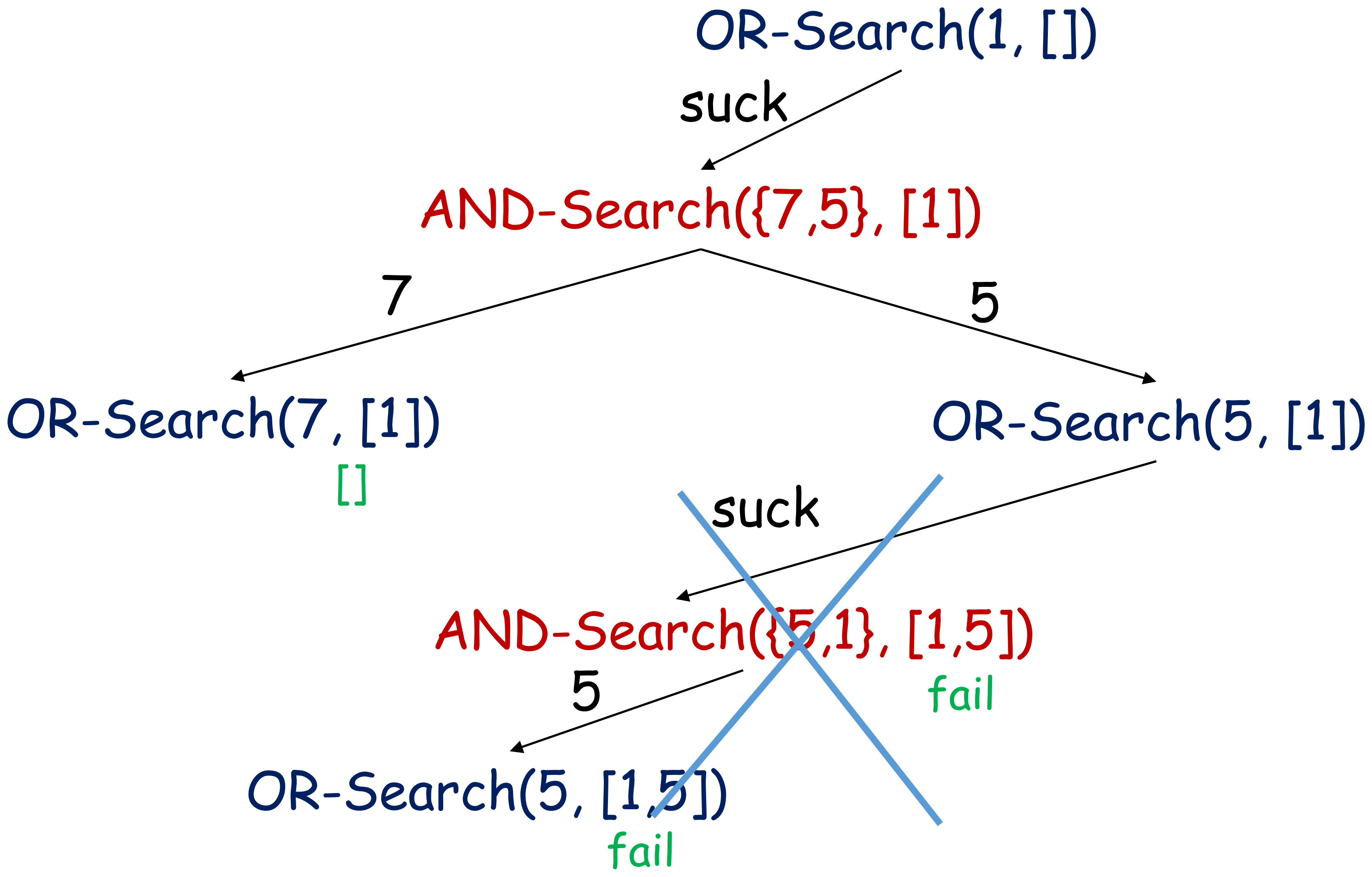

OR-Search(1, [])
suck
AND-Search({7,5}, [1])
7
5
OR-Search(7, [1])
[]
OR-Search(5, [1])

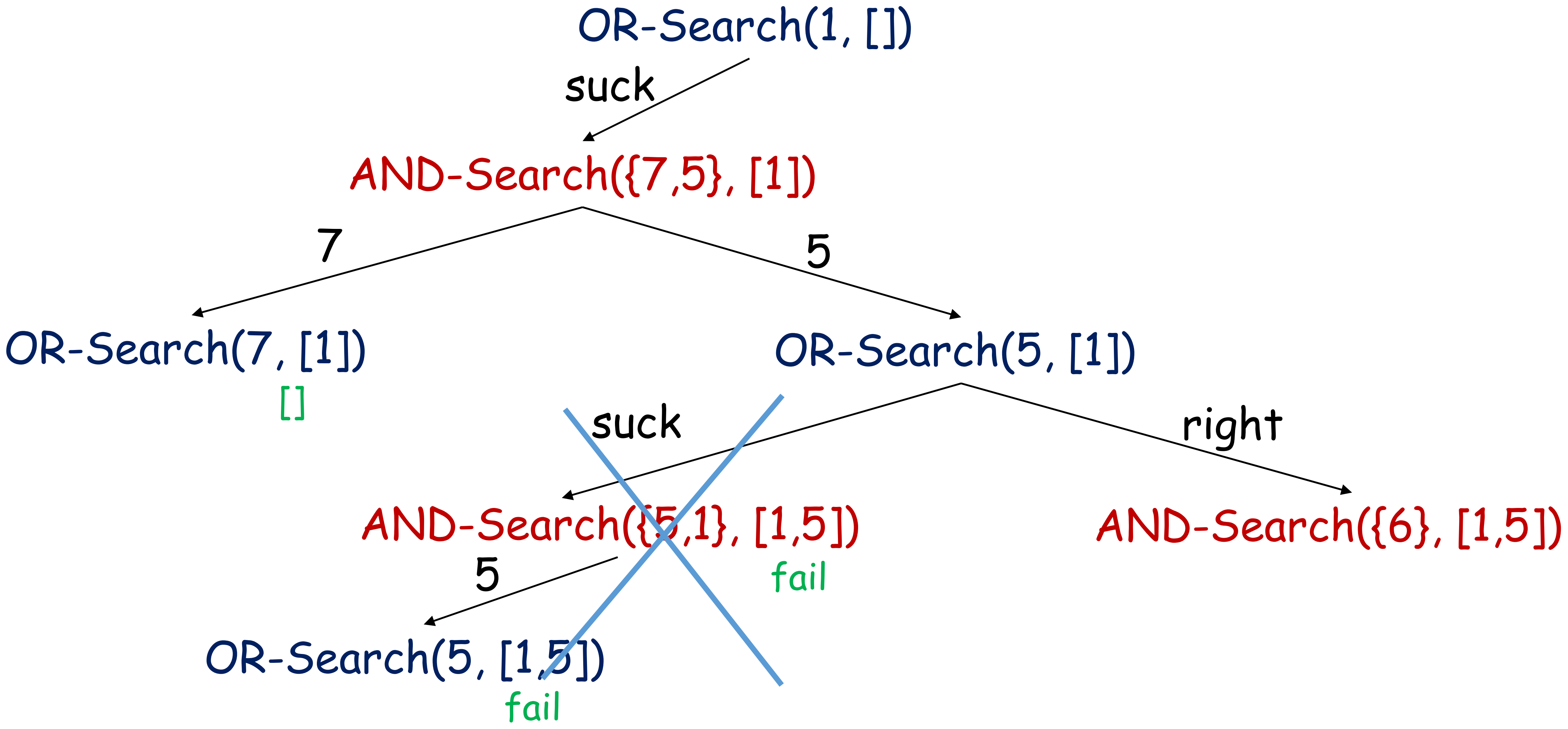

OR-Search(1, [])
suck
AND-Search({7,5}, [1])
7
5
OR-Search(7, [1])
[]
OR-Search(5, [1])
suck
AND-Search({5,1}, [1,5])

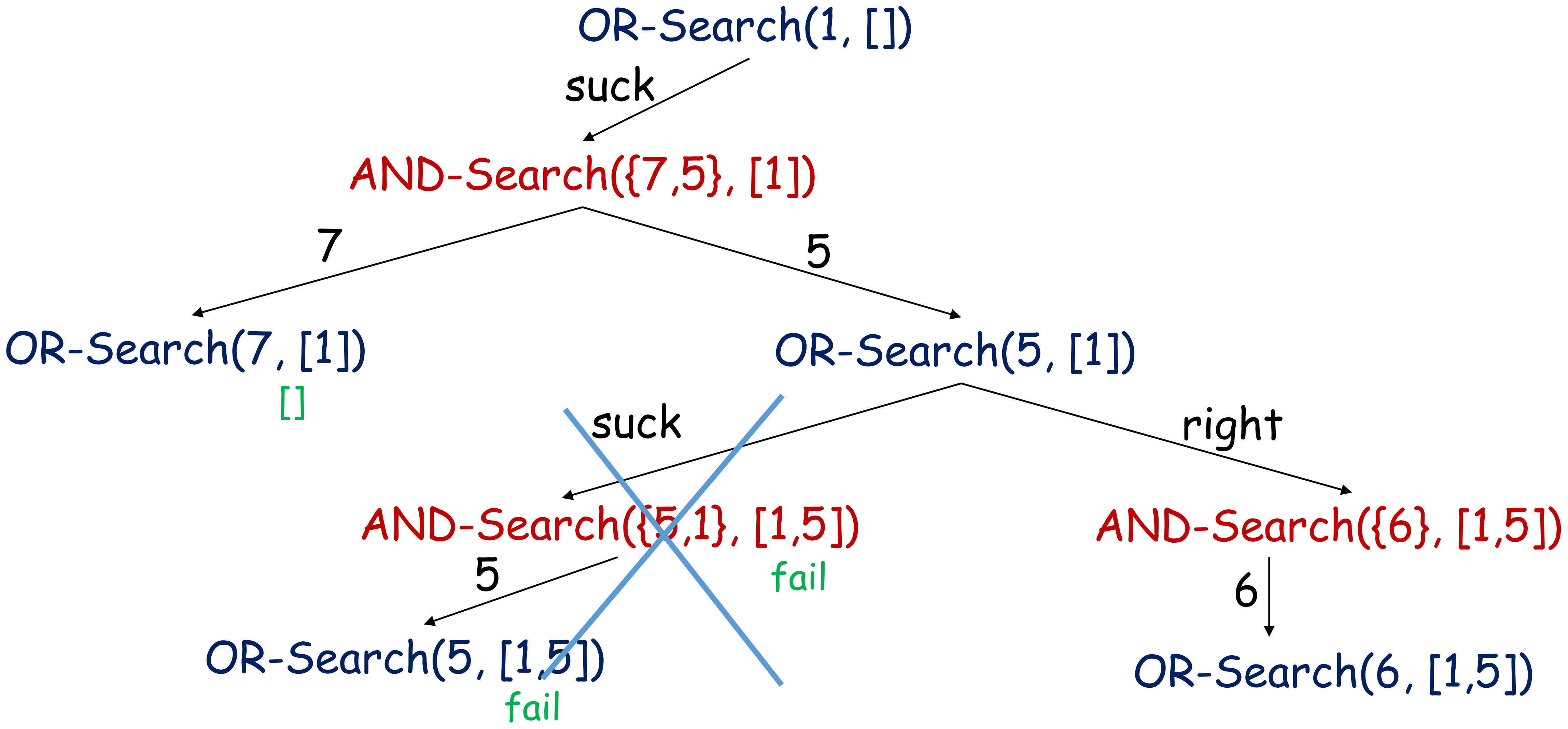

OR-Search(1, [])
suck
AND-Search({7,5}, [1])
7
5
OR-Search(7, [1])
[]
OR-Search(5, [1])
suck
AND-Search({5,1}, [1,5])
5
OR-Search(5, [1,5])

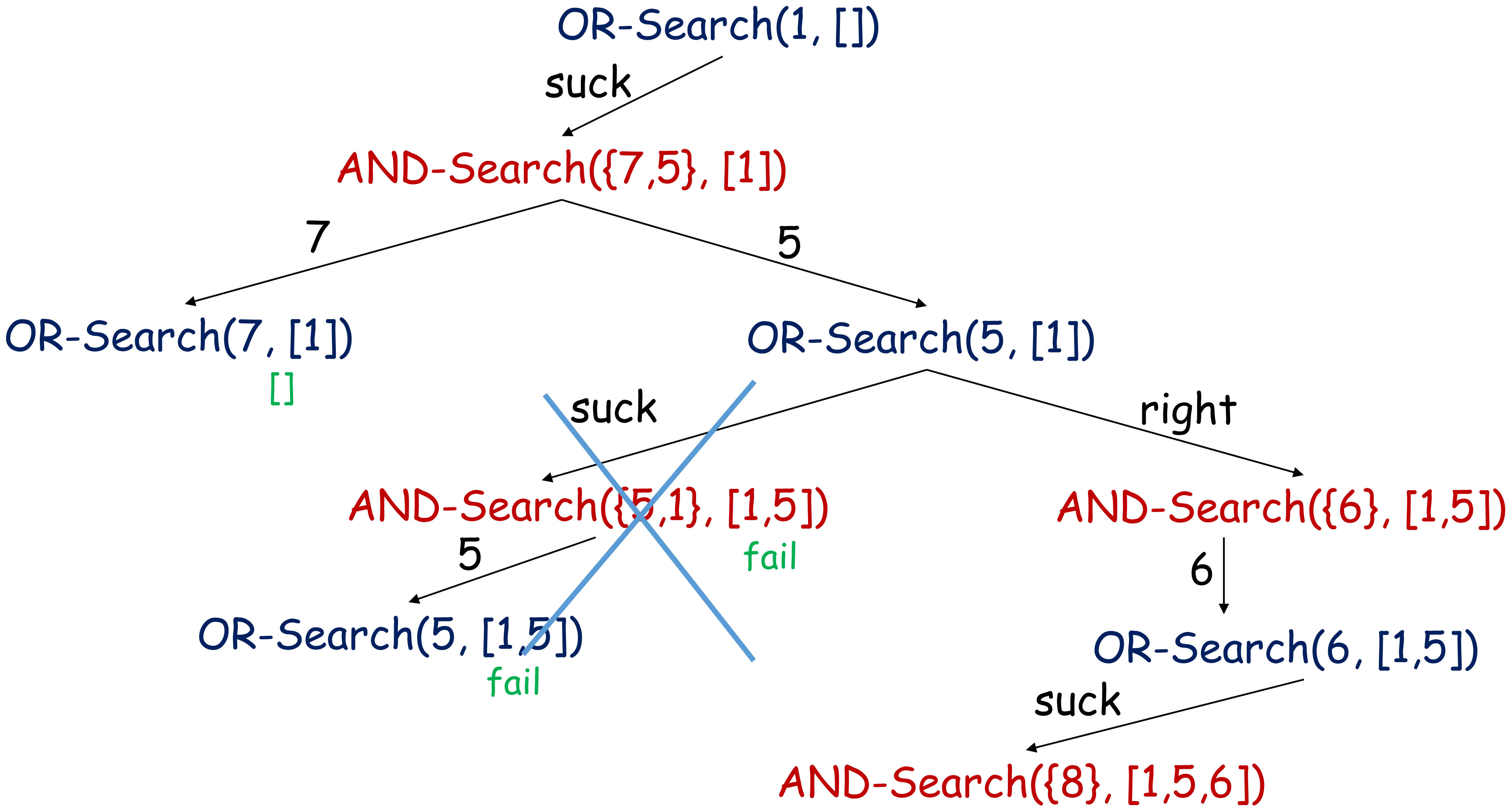

OR-Search(1, [])
suck
AND-Search({7,5}, [1])
7
5
OR-Search(7, [1])
[]
OR-Search(5, [1])
suck
AND-Search({5,1}, [1,5])
5
OR-Search(5, [1,5])
fail


OR-Search(1, [])
suck
AND-Search({7,5}, [1])
7
5
OR-Search(7, [1])
[]
OR-Search(5, [1])
suck
AND-Search({5,1}, [1,5])
5
fail
OR-Search(5, [1,5])
fail

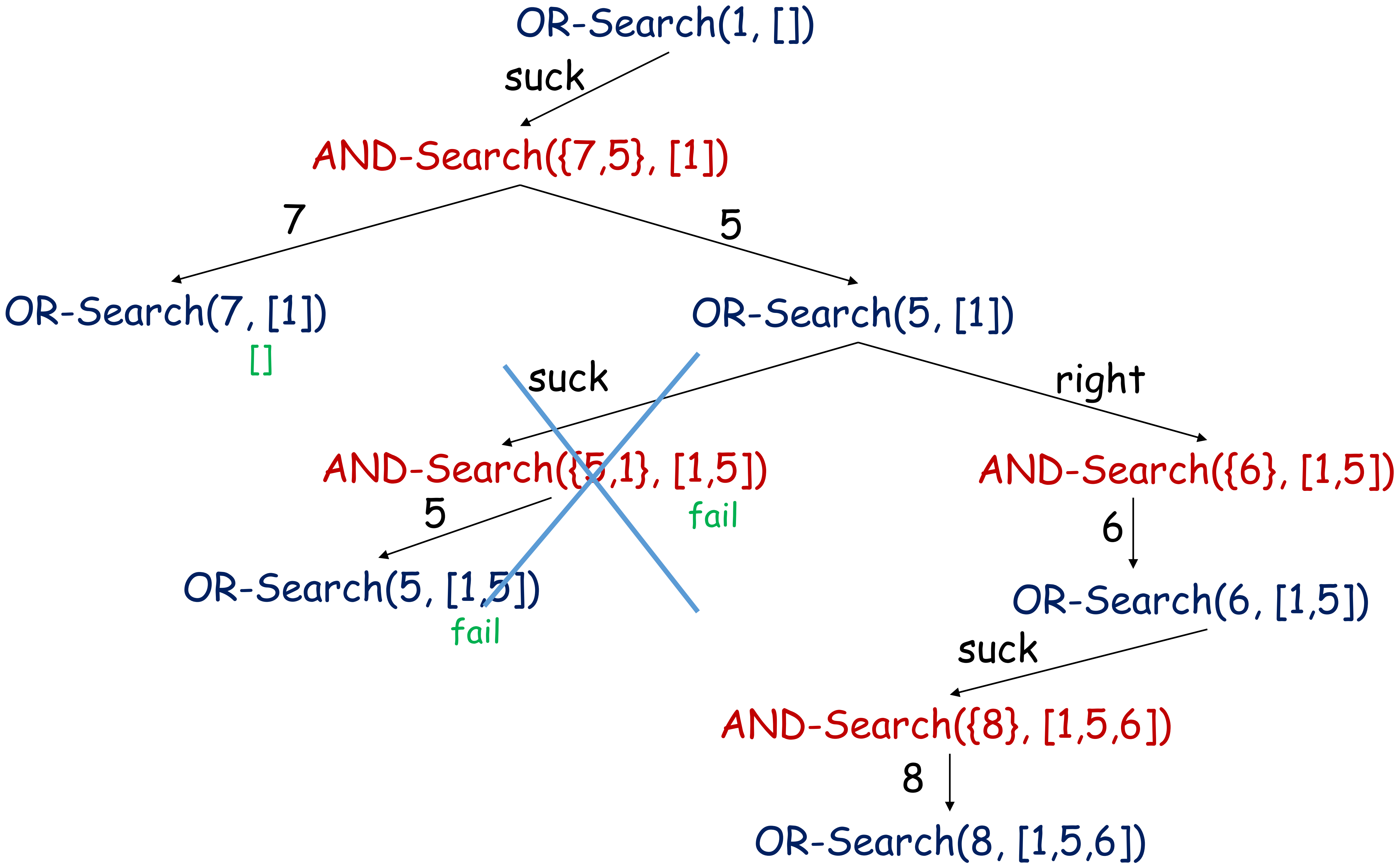

OR-Search(1, [])
suck
AND-Search({7,5}, [1])
7
5
OR-Search(7, [1])
[]
OR-Search(5, [1])
suck
AND-Search({5,1}, [1,5])
fail
5
OR-Search(5, [1,5])
fail


OR-Search(1, [])
suck
AND-Search({7,5}, [1])
7
5
OR-Search(7, [1])
[]
OR-Search(5, [1])
suck
right
AND-Search({5,1}, [1,5])
fail
AND-Search({6}, [1,5])
5
OR-Search(5, [1,5])
fail

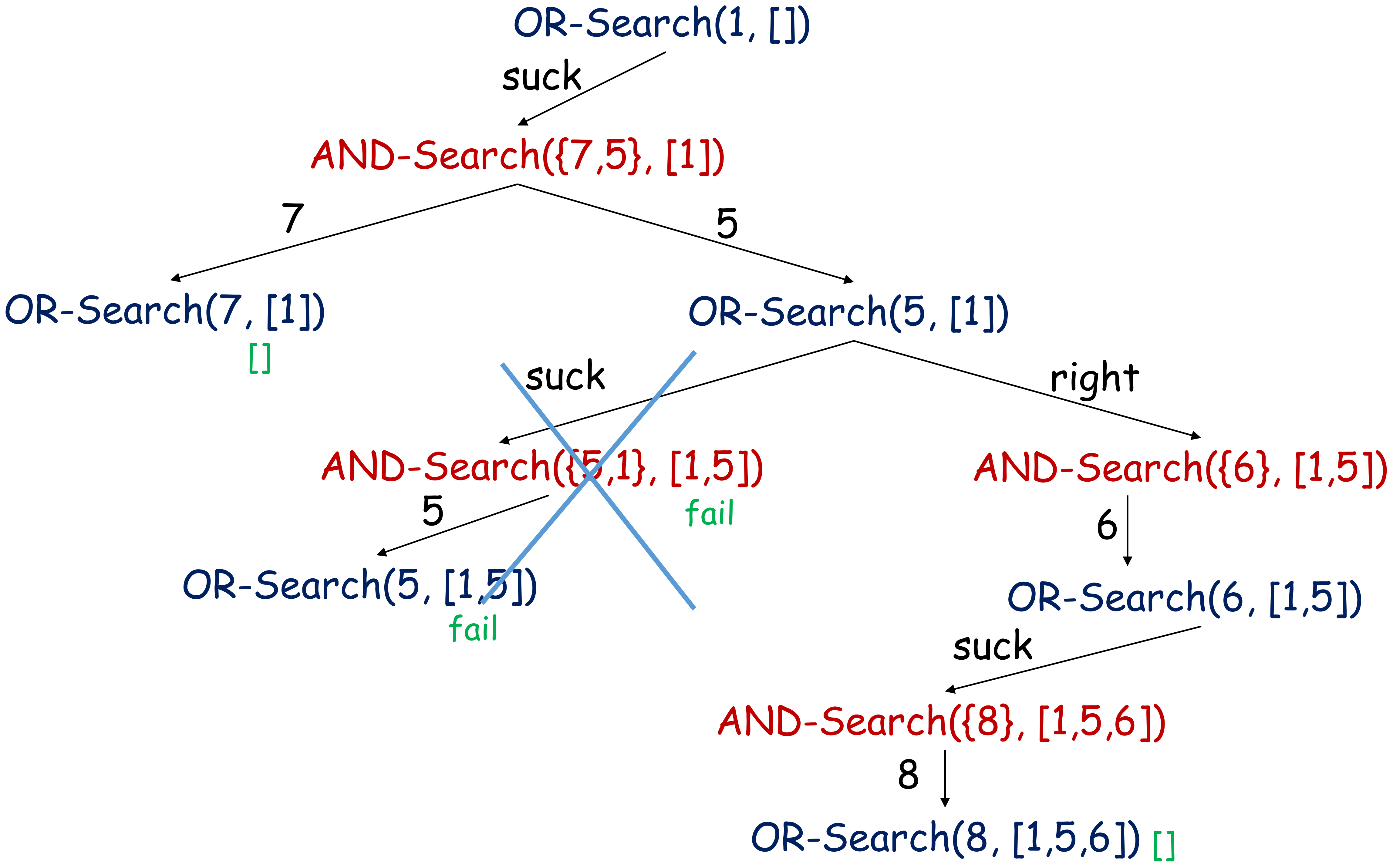

OR-Search(1, [])
suck
AND-Search({7,5}, [1])
7
5
OR-Search(7, [1])
[]
OR-Search(5, [1])
suck
right
AND-Search({5,1}, [1,5])
fail
AND-Search({6}, [1,5])
5
6
OR-Search(5, [1,5])
fail
OR-Search(6, [1,5])

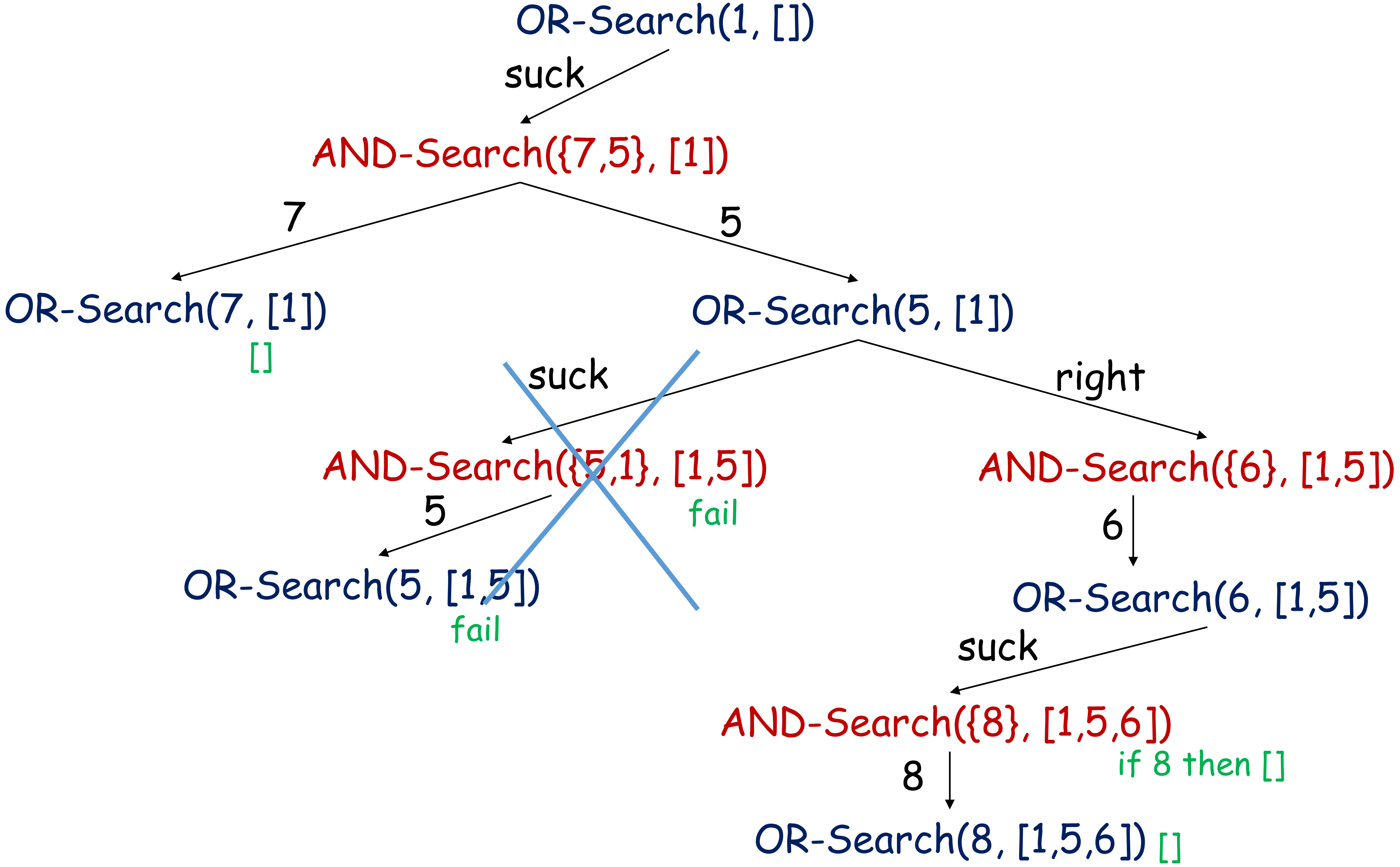


OR-Search(1, [])
suck
AND-Search({7,5}, [1])
7
5
OR-Search(7, [1])
[]
OR-Search(5, [1])
suck
right
AND-Search({5,1}, [1,5])
fail
AND-Search({6}, [1,5])
5
6
OR-Search(5, [1,5])
fail
OR-Search(6, [1,5])
suck
AND-Search({8}, [1,5,6])

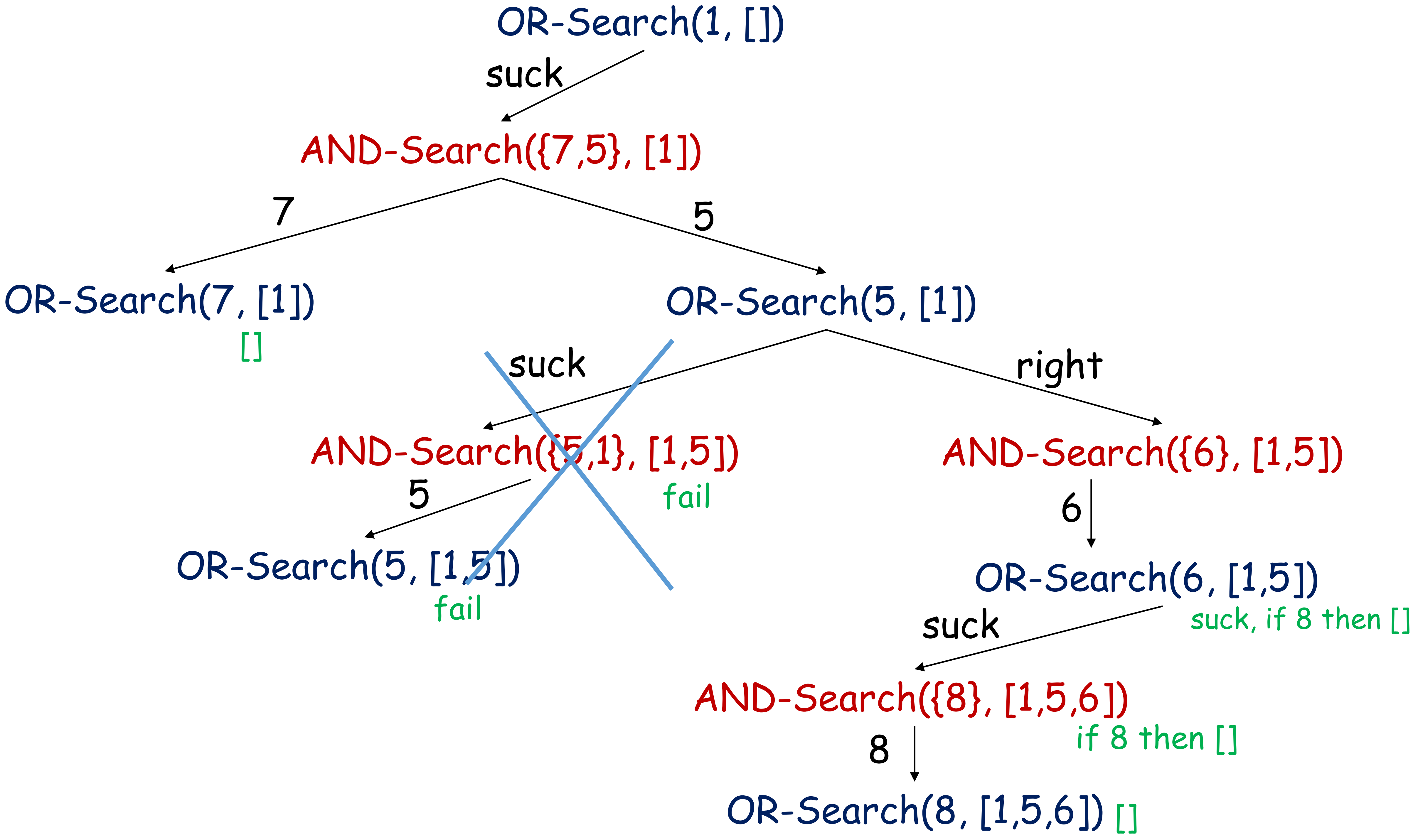


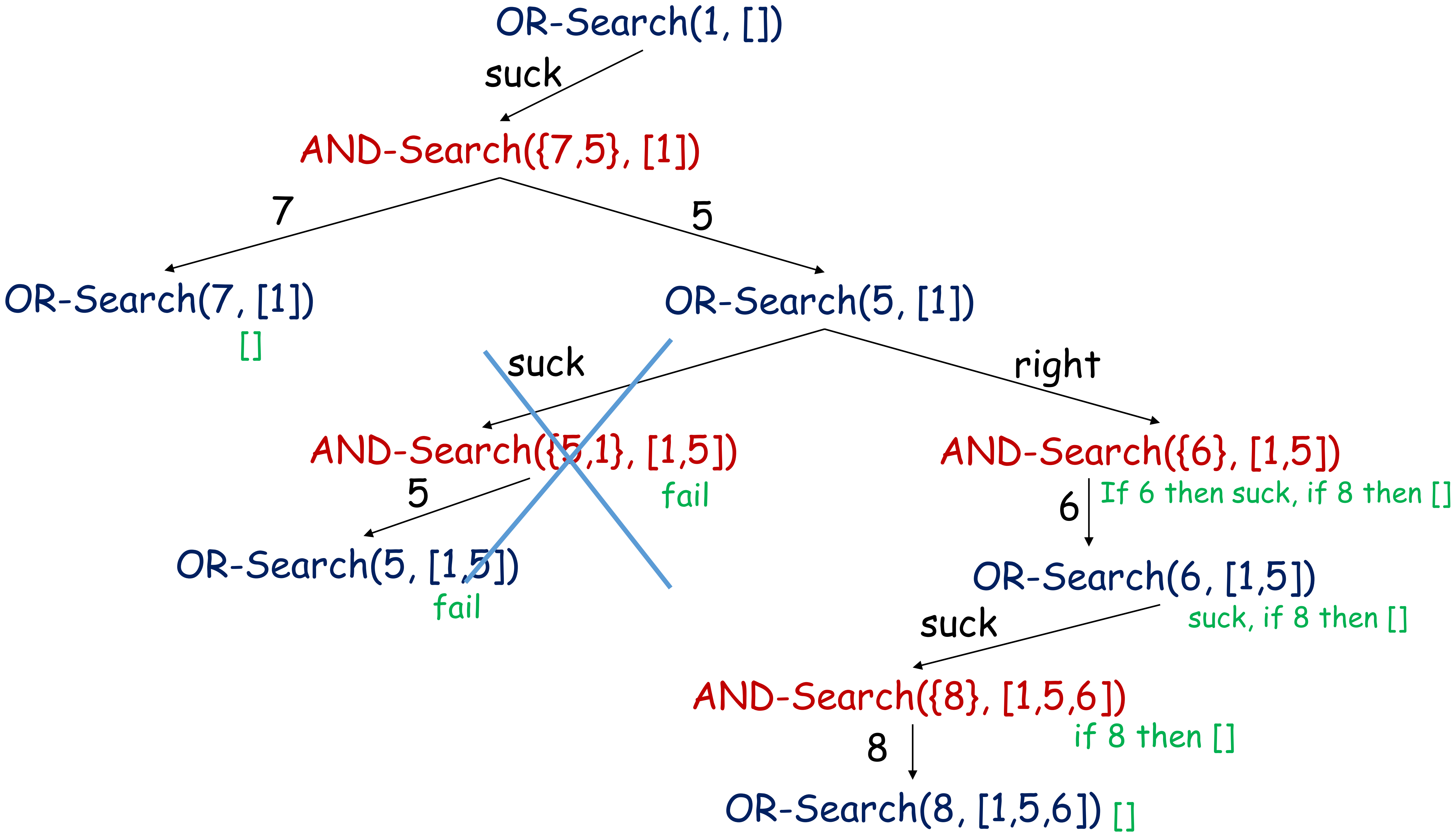
OR-Search(1, [])
suck
AND-Search({7,5}, [1])
7
5
OR-Search(7, [1])
[]
OR-Search(5, [1])
suck
right
AND-Search({5,1}, [1,5])
fail
AND-Search({6}, [1,5])
5
OR-Search(5, [1,5])
fail
6
OR-Search(6, [1,5])
suck
AND-Search({8}, [1,5,6])
8
OR-Search(8, [1,5,6])

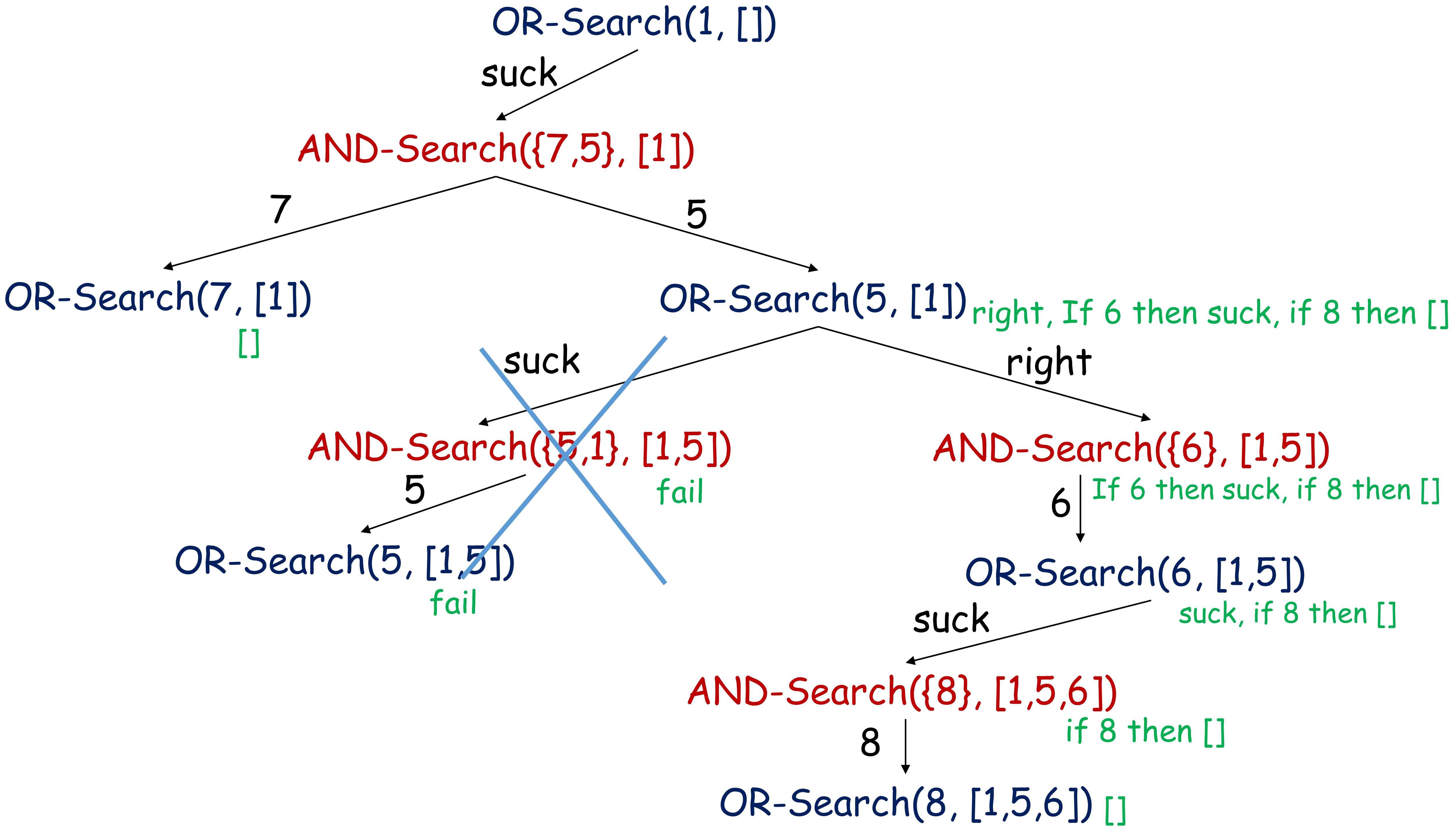

OR-Search(1, [])
suck
AND-Search({7,5}, [1])
7
5
OR-Search(7, [1])
[]
OR-Search(5, [1])
suck
right
AND-Search({5,1}, [1,5])
fail
AND-Search({6}, [1,5])
5
OR-Search(5, [1,5])
fail
6
OR-Search(6, [1,5])
suck
AND-Search({8}, [1,5,6])
8
OR-Search(8, [1,5,6])
[]

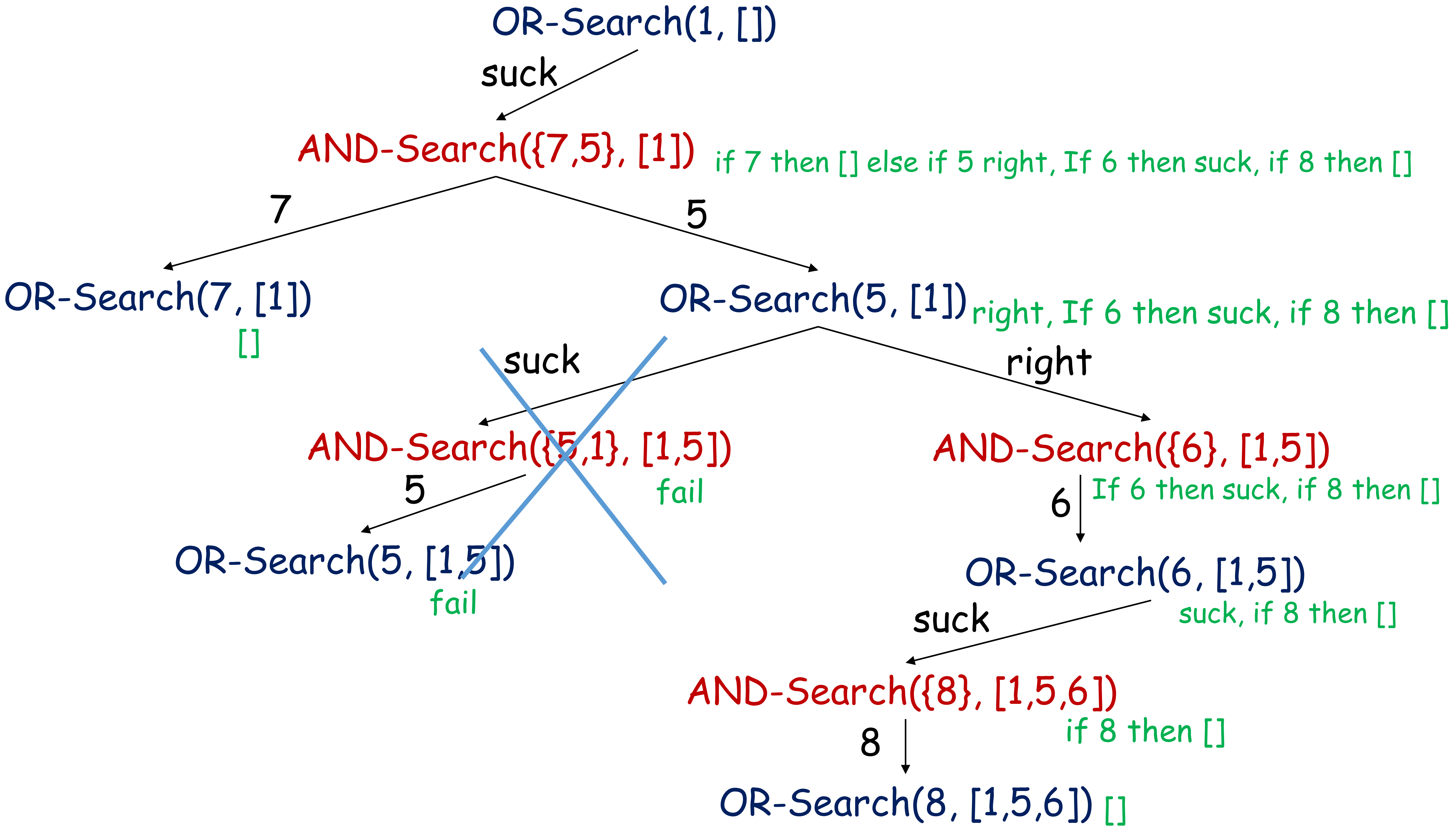


OR-Search(1, [])
suck
AND-Search({7,5}, [1])
7
5
OR-Search(7, [1])
[]
OR-Search(5, [1])
suck
right
AND-Search({5,1}, [1,5])
fail
AND-Search({6}, [1,5])
5
6
OR-Search(5, [1,5])
fail
OR-Search(6, [1,5])
suck
AND-Search({8}, [1,5,6])
if 8 then []
8
OR-Search(8, [1,5,6])
[]

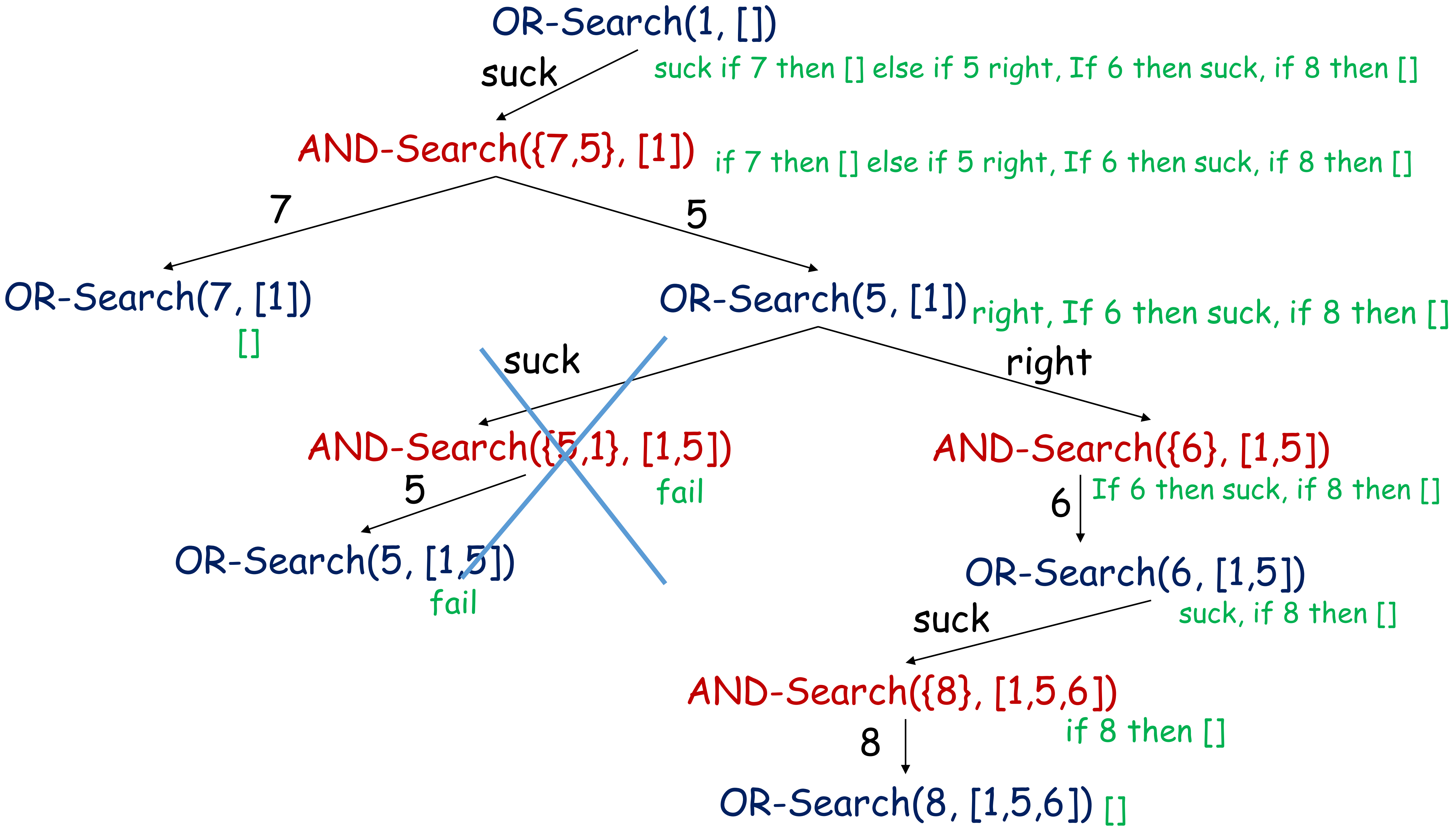

OR-Search(1, [])
suck
AND-Search({7,5}, [1])
7
5
OR-Search(7, [1])
[]
OR-Search(5, [1])
suck
right
AND-Search({5,1}, [1,5])
fail
AND-Search({6}, [1,5])
5
6
OR-Search(5, [1,5])
fail
OR-Search(6, [1,5])
suck
suck, if 8 then []
AND-Search({8}, [1,5,6])
if 8 then []
8
OR-Search(8, [1,5,6])
[]


OR-Search(1, [])
suck
AND-Search({7,5}, [1])
7
5
OR-Search(7, [1])
[]
OR-Search(5, [1])
suck
right
AND-Search({5,1}, [1,5])
fail
5
OR-Search(5, [1,5])
fail
AND-Search({6}, [1,5])
6
If 6 then suck, if 8 then []
OR-Search(6, [1,5])
suck
suck, if 8 then []
AND-Search({8}, [1,5,6])
8
if 8 then []
OR-Search(8, [1,5,6])
[]


OR-Search(1, [])
suck
AND-Search({7,5}, [1])
7
5
OR-Search(7, [1])
[]
OR-Search(5, [1])
right, If 6 then suck, if 8 then []
suck
right
AND-Search({5,1}, [1,5])
fail
AND-Search({6}, [1,5])
5
6
If 6 then suck, if 8 then []
OR-Search(5, [1,5])
fail
OR-Search(6, [1,5])
suck
suck, if 8 then []
AND-Search({8}, [1,5,6])
if 8 then []
8
OR-Search(8, [1,5,6])
[]



OR-Search(1, [])
suck
AND-Search({7,5}, [1])
if 7 then [] else if 5 right, If 6 then suck, if 8 then []
7
5
OR-Search(7, [1])
[]
OR-Search(5, [1])
right, If 6 then suck, if 8 then []
suck
right
AND-Search({5,1}, [1,5])
fail
5
OR-Search(5, [1,5])
fail
AND-Search({6}, [1,5])
6
If 6 then suck, if 8 then []
OR-Search(6, [1,5])
suck, if 8 then []
suck
AND-Search({8}, [1,5,6])
if 8 then []
8
OR-Search(8, [1,5,6])
[]


OR-Search(1, [])
suck
suck if 7 then [] else if 5 right, If 6 then suck, if 8 then []
AND-Search({7,5}, [1])
if 7 then [] else if 5 right, If 6 then suck, if 8 then []
7
5
OR-Search(7, [1])
[]
OR-Search(5, [1])
right, If 6 then suck, if 8 then []
suck
right
AND-Search({5,1}, [1,5])
fail
5
OR-Search(5, [1,5])
fail
AND-Search({6}, [1,5])
6
If 6 then suck, if 8 then []
OR-Search(6, [1,5])
suck
suck, if 8 then []
AND-Search({8}, [1,5,6])
8
if 8 then []
OR-Search(8, [1,5,6])
[]

AND-OR depth first graph search

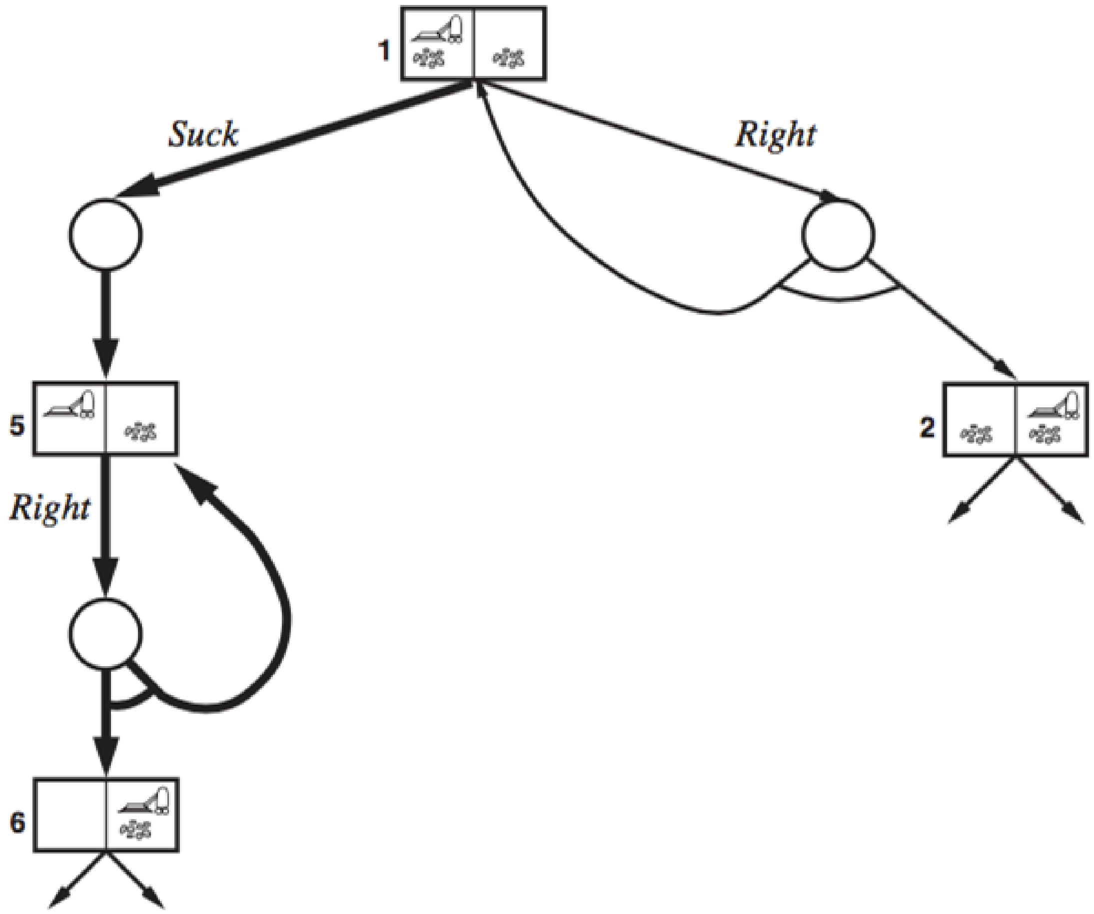
- Cycles arise often in non-deterministic problems
- Algorithm returns with failure when the current state is identical to one of ancestors
  - If there is a non-cyclic path, the earlier consideration of the state is sufficient
- Termination is guaranteed in finite state spaces
  - Every path reaches a goal, a dead-end, or a repeated state

Cycles

- Slippery vacuum world
  - Left and Right actions sometimes fail (leaving the agent in the same location)
  - No acyclic solution
    - Solution?
      - Cyclic plan: keep on trying an action until it works

[Suck, L1: Right, **if** state = 5 **then** L1 **else** Suck]

[Suck, **while** state = 5 **do** Right, Suck]

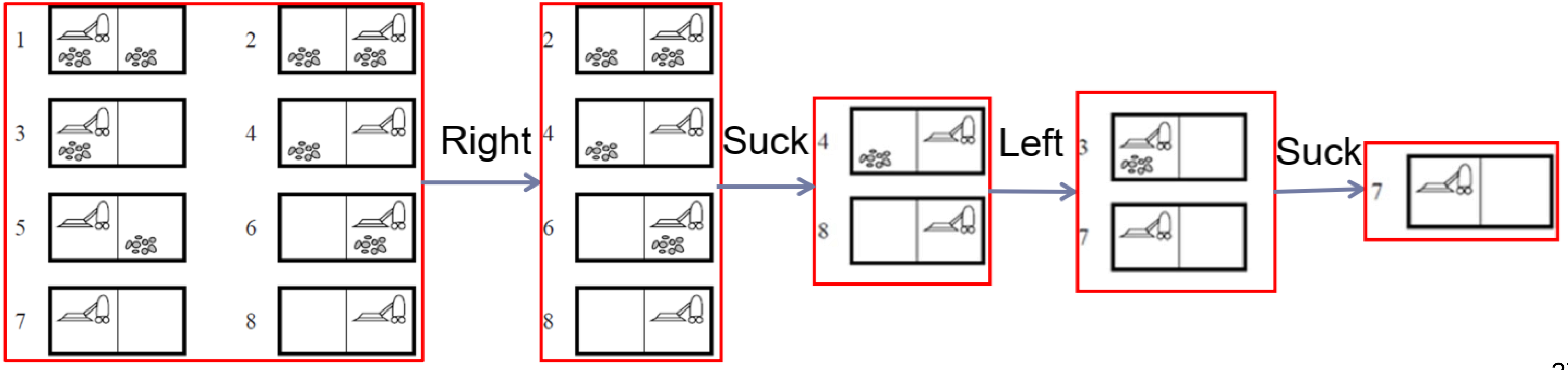


Searching with partial observations

- The agent does not always know its exact state
  - Agent is in one of several possible states and thus an action may lead to one of several possible outcomes
- Belief state
  - Agent's current belief about the possible states, given the sequence of actions and observations up to that point

Searching with unobservable states

- Sensor-less or conformant problem
- Vacuum world example
  - Initial state
    - belief = {1, 2, 3, 4, 5, 6, 7, 8}
  - Action sequence (conformant plan)
    - [Right, Suck, Left, Suck]



Sensor-less problem formulation

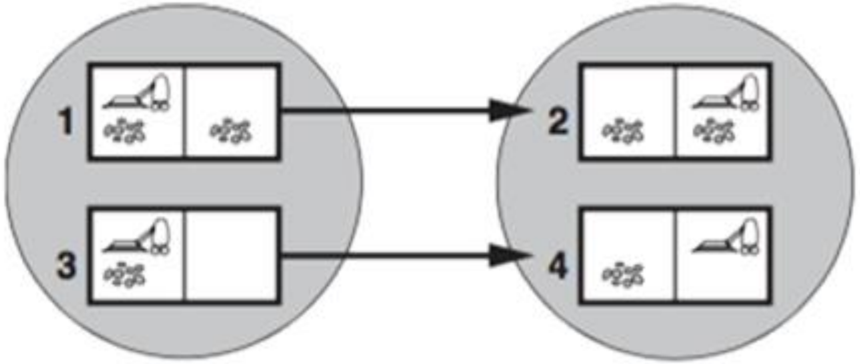
- Belief state space (instead of physical state space)
  - It is fully observable
  - Solution is a sequence of actions (even in non-deterministic environment)
- Physical problem
  - N states, $ACTIONS_p$, $RESULTS_p$, $GOAL\_TEST_p$, $STEP\_COST_p$
- Sensor-less problem
  - Up to 2^N states, ACTIONS, RESULTS, GOAL_TEST, STEP_COST

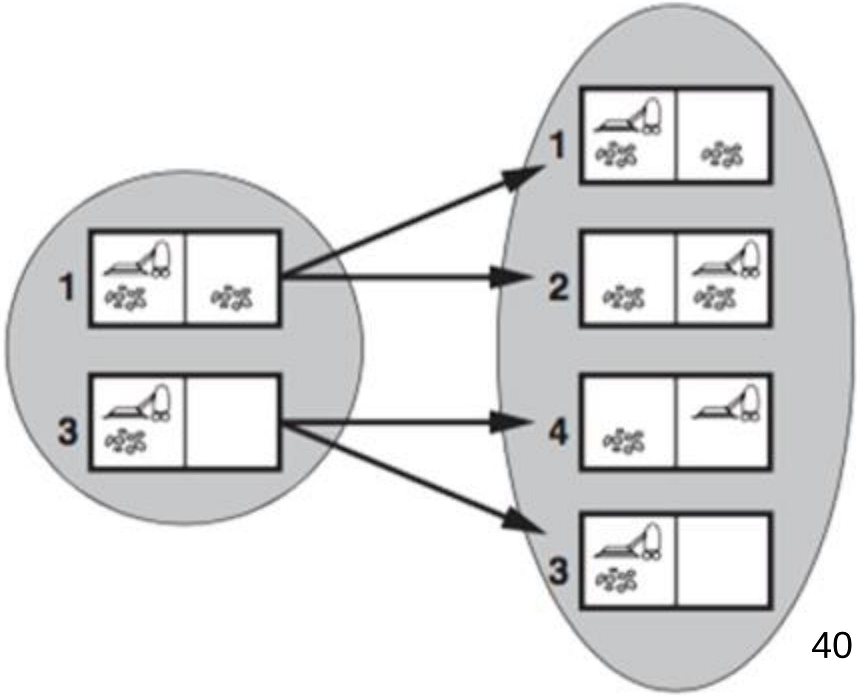
Sensor-less problem formulation

- States
  - Every possible set of physical states, 2^N
- Initial State
  - Usually the set of all physical states
- Actions
  - $ACTIONS(b) = \bigcup_{s \in b} ACTIONS_p(s)$
    - Illegal actions? i.e., $b=\{s_1, s_2\}$, $ACTIONS_p(s_1) \neq ACTIONS_p(s_2)$
    - Illegal actions have no effect on the env. (union of physical actions)
    - Illegal actions are not legal at all (intersection of physical actions)

Sensor-less problem formulation

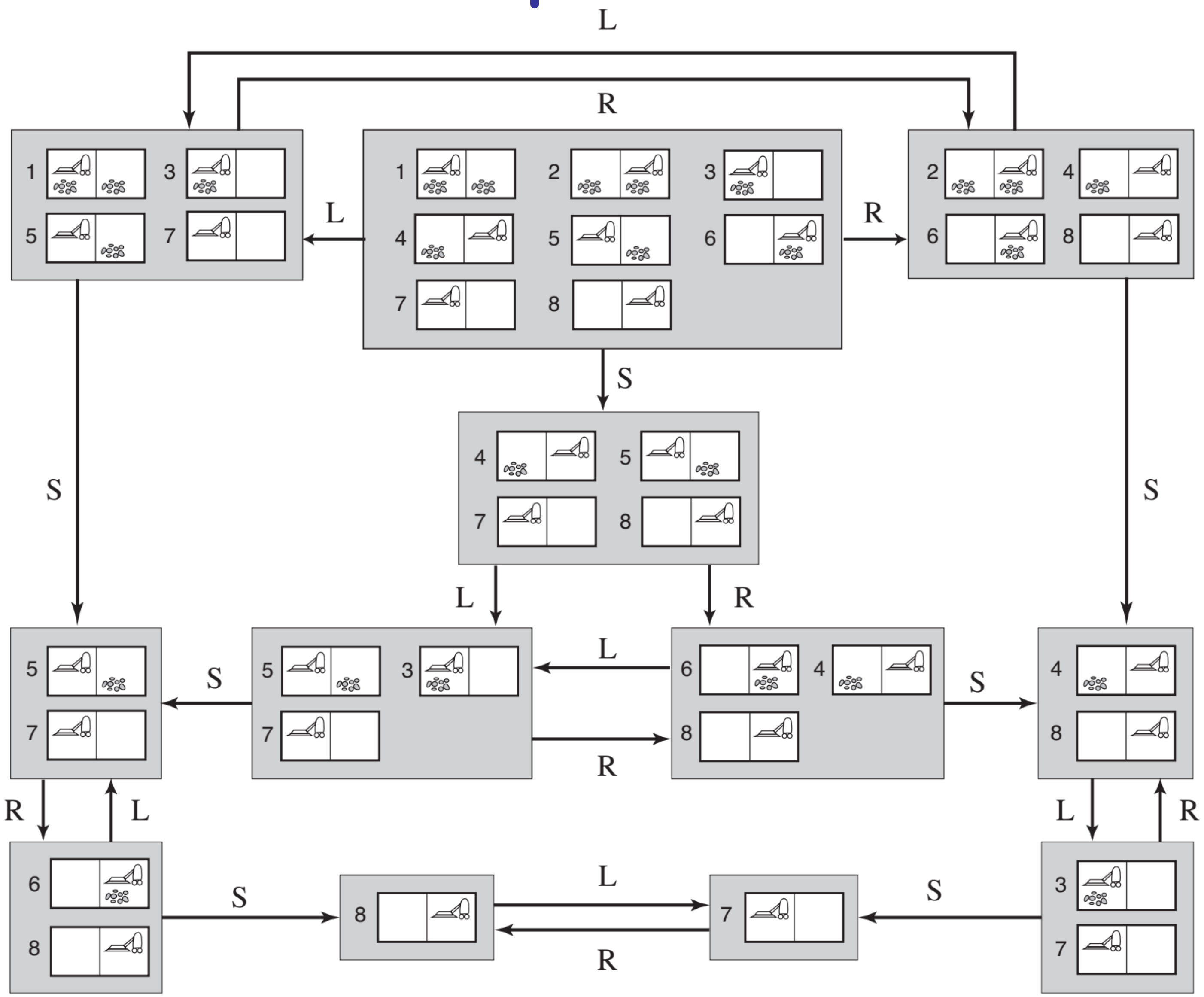
- Transposition model
  - b' = PREDICT(b, a)
    - Deterministic actions
      - $b' = \{s' : s' = RESULTS_p(s, a) \text{ and } s \in b\}$
    - Nondeterministic actions
      - $b' = \cup_{s \in b} RESULTS_p(s, a)$
- Goal test
  - Goal is satisfied when all the physical states in the belief state satisfy GOAL_TEST
- Step cost
  - $STEP\_COST_p$ if the cost of an action is the same in all states





Vacuum world example

- Belief-state space for sensor-less deterministic vacuum world
  - Total number of possible belief states? 2^8
  - Number of reachable belief states? 12



Sensor-less problem: searching

- In general, we can use any standard search algorithm
- Searching in these spaces is not usually feasible (scalability)
  - Problem1: No. of reachable belief states
    - Pruning (subsets or supersets) can reduce this difficulty
    - Branching factor and solution depth in the belief-state space and physical state space are not usually such different
  - Problem2: (main difficulty): No. of physical states in each belief state
    - Using a compact state representation (like formal representation)
    - Incremental belief-state search: Search for solutions by considering physical states incrementally (not whole belief space) to quickly detect failure if we reach an unsolvable physical state

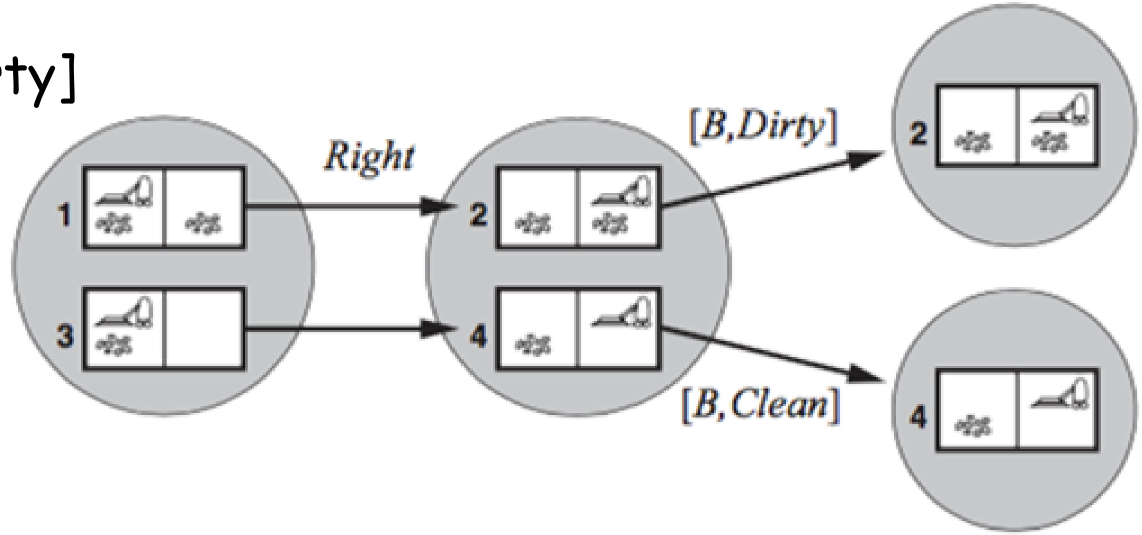
Searching with partial observations

- Similar to sensor-less, after each action the new belief state must be **predicted**
- After each perception the belief state is **updated**
  - E.g., local sensing vacuum world
    - After each perception, the belief state can contain at most two physical states.
- We must plan for different **possible perceptions**

Vacuum world with A position sensor and local dirt sensor

- Deterministic world

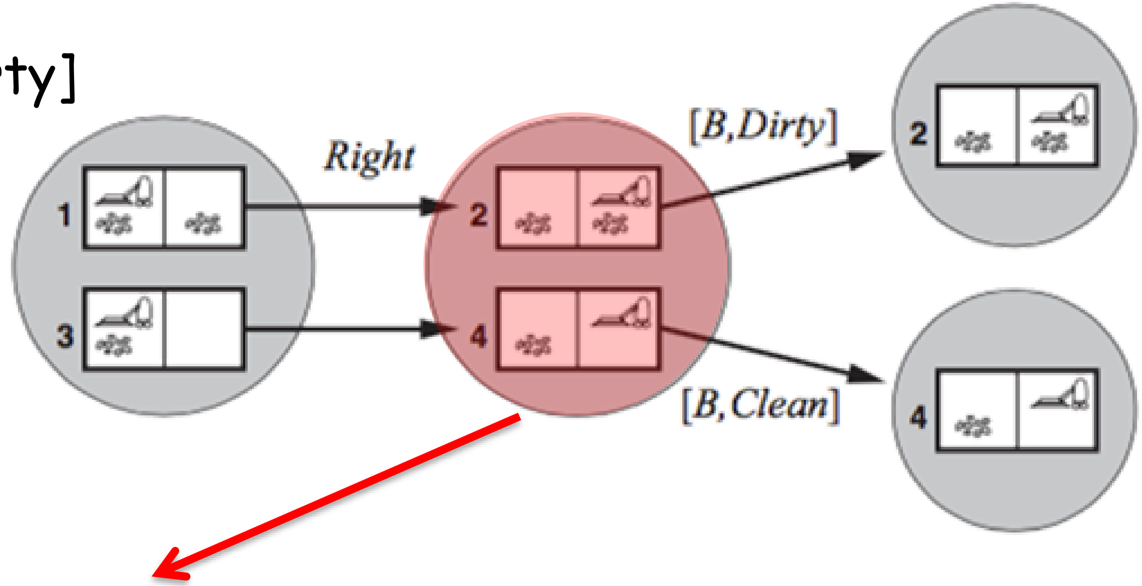
[A, Dirty]



Vacuum world with A position sensor and local dirt sensor

- Deterministic world

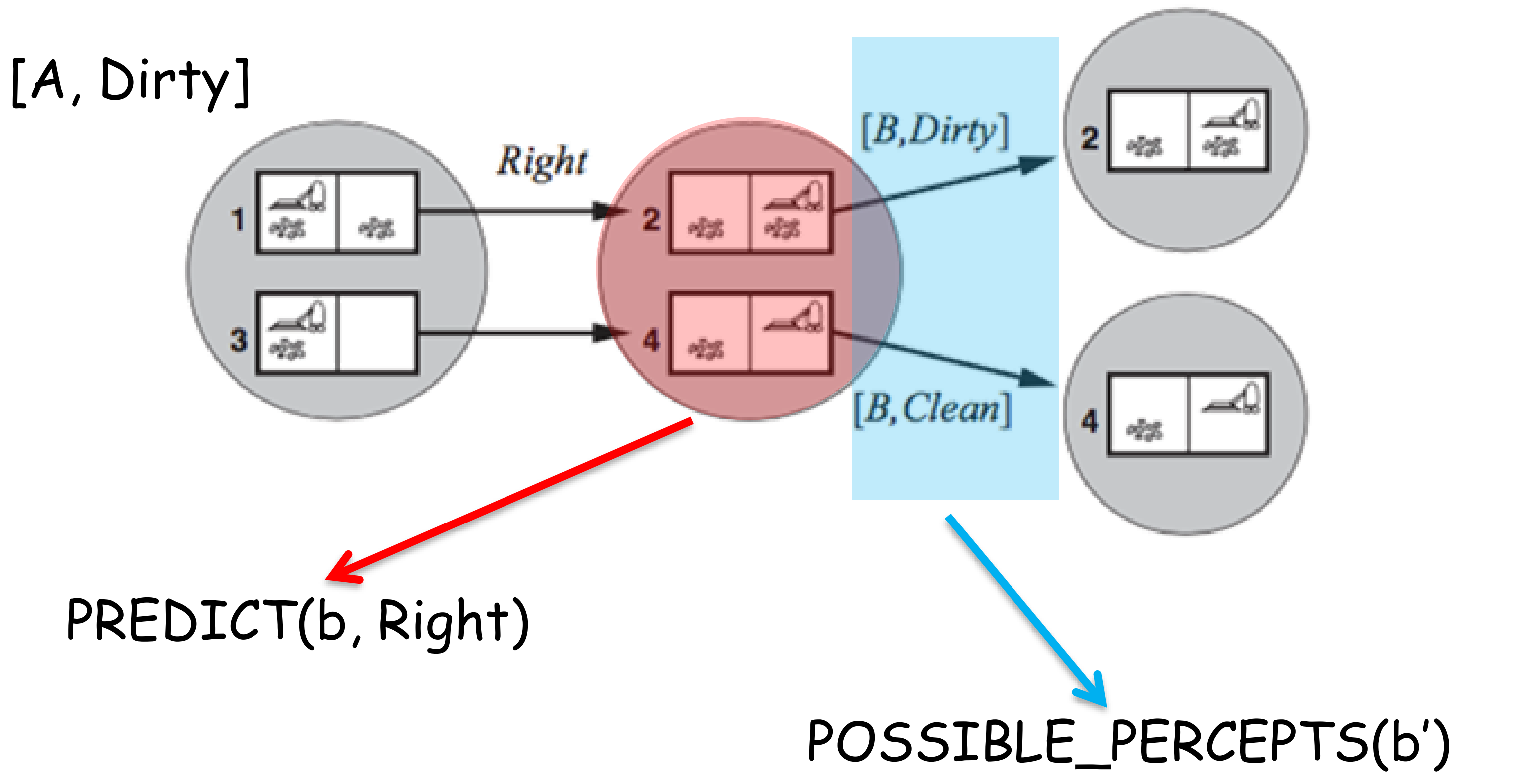
[A, Dirty]



PREDICT(b, Right)

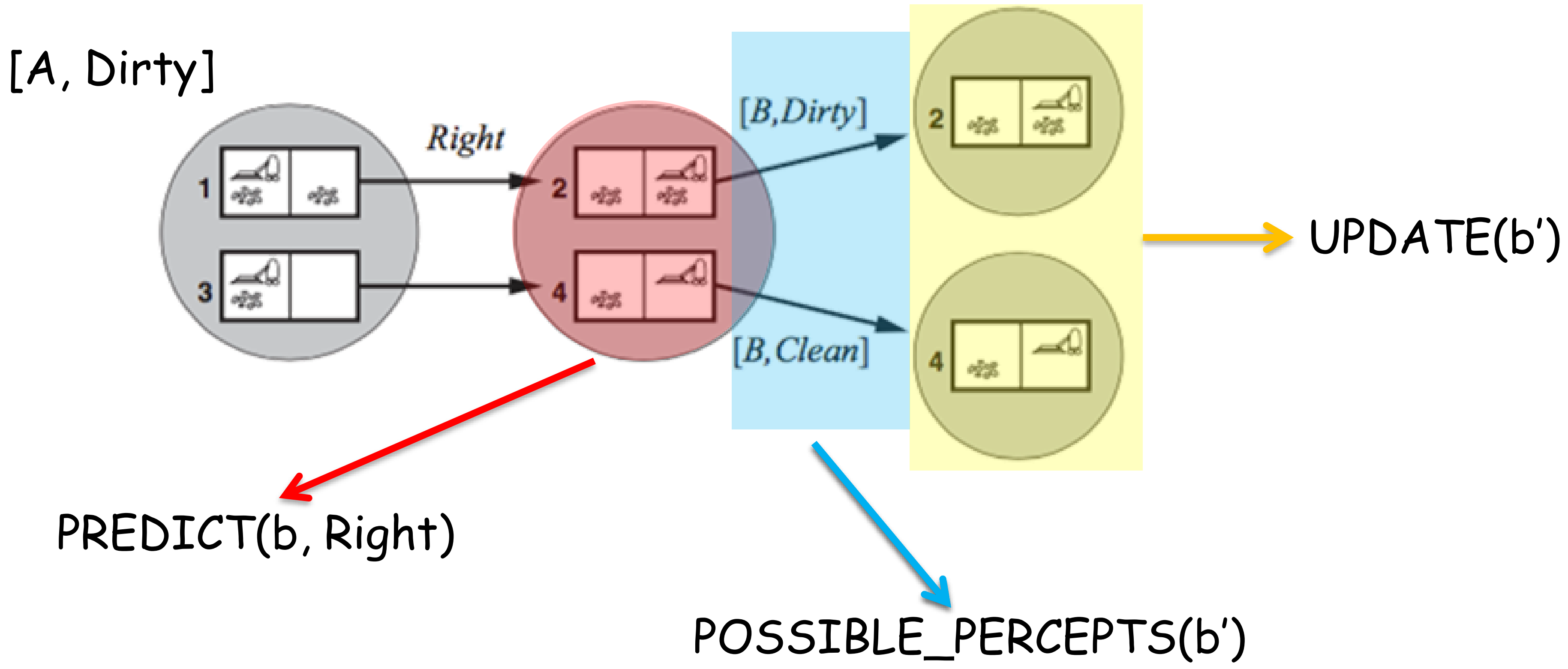
Vacuum world with A position sensor and local dirt sensor

- Deterministic world



Vacuum world with A position sensor and local dirt sensor

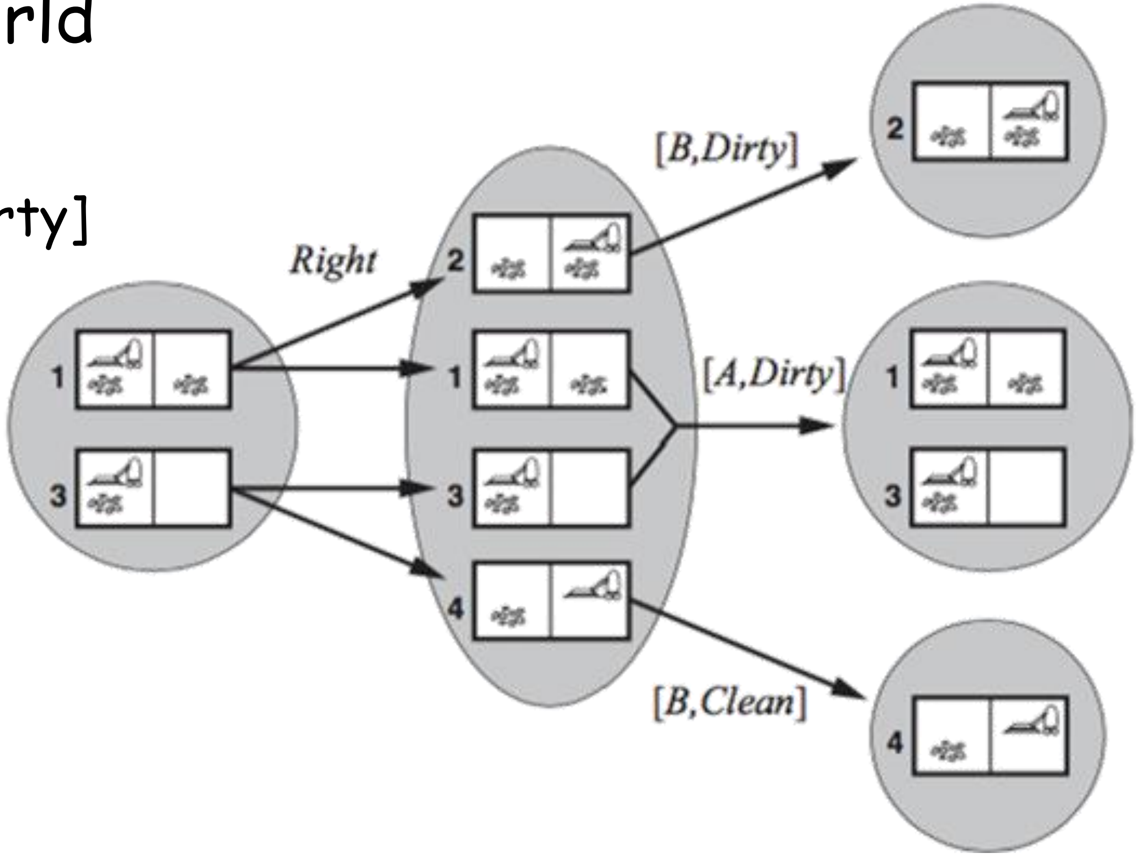
- Deterministic world



Vacuum world with A position sensor and local dirt sensor

- Slippery world

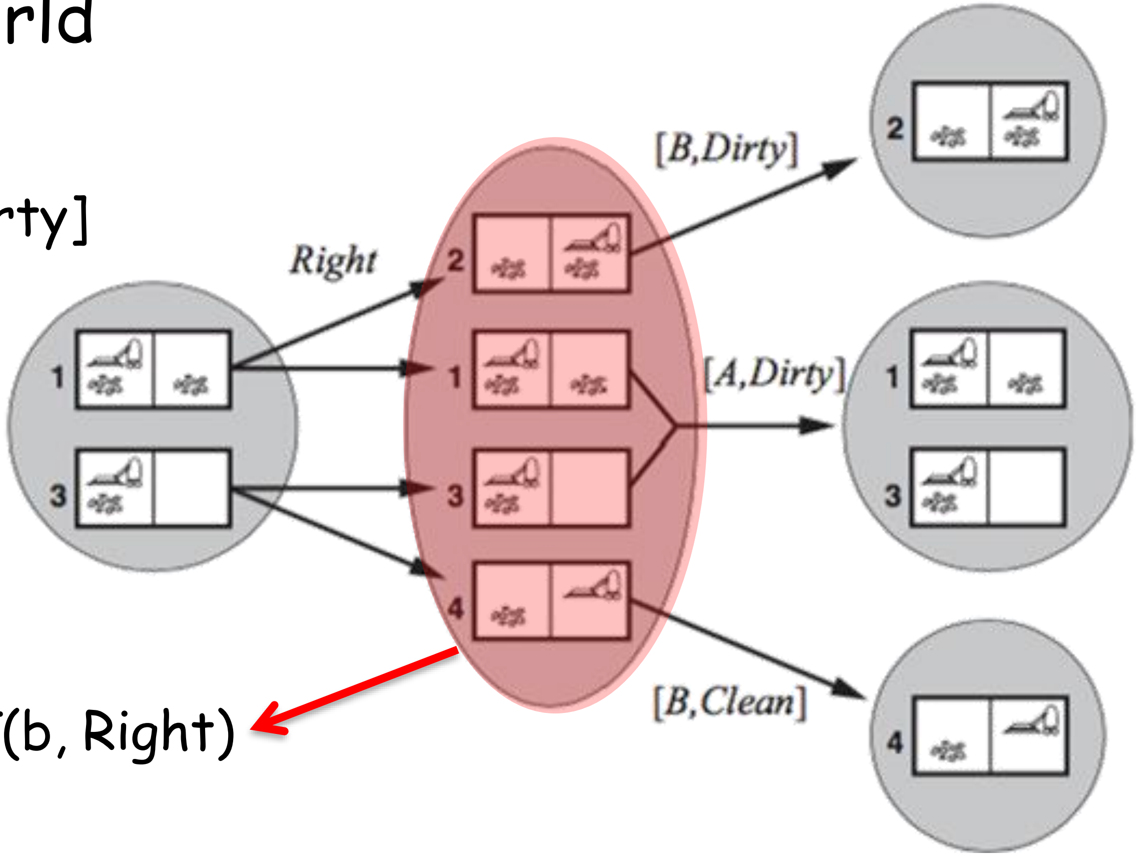
[A, Dirty]



Vacuum world with A position sensor and local dirt sensor

- Slippery world

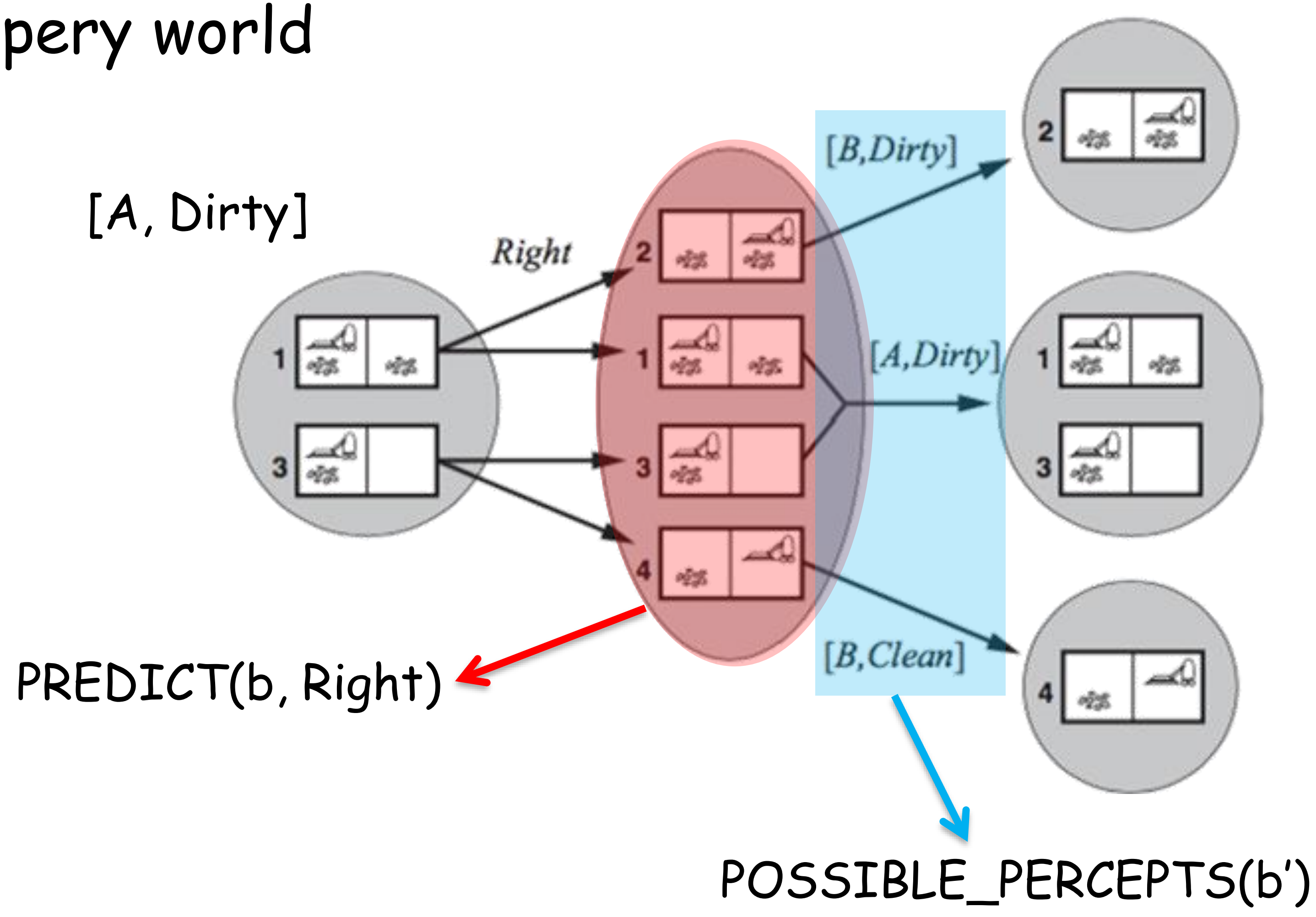
[A, Dirty]



PREDICT(b, Right)

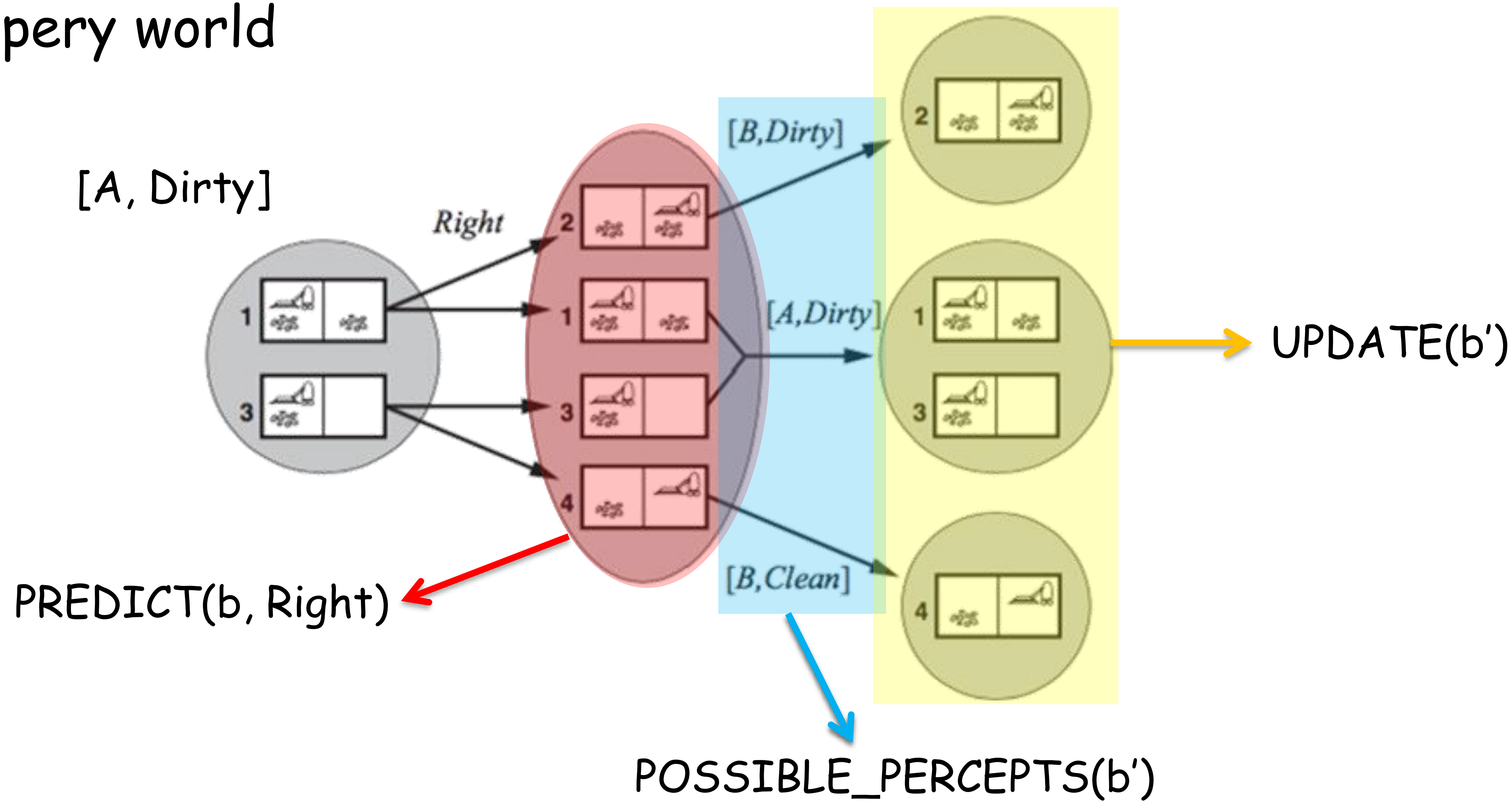
Vacuum world with A position sensor and local dirt sensor

- Slippery world



Vacuum world with A position sensor and local dirt sensor

- Slippery world



Transition model (partially observable env.)

- Prediction stage
  - How does the belief state change after doing an action?

$$b' = \text{PREDICT}(b, a)$$

  - Deterministic actions

$$b' = \{s' : s' = \text{RESULTS}_P(s, a) \text{ and } s \in b\}$$

  - Nondeterministic actions

$$b' = U_{s \in b} \text{RESULTS}_P(s, a)$$

- Possible Perceptions
  - What are the possible perceptions in a belief state?

$$\text{POSSIBLE\_PERCEPTS}(b') = \{o : o = \text{PERCEPT}(s) \text{ and } s \in b'\}$$

Transition model (partially observable env.)

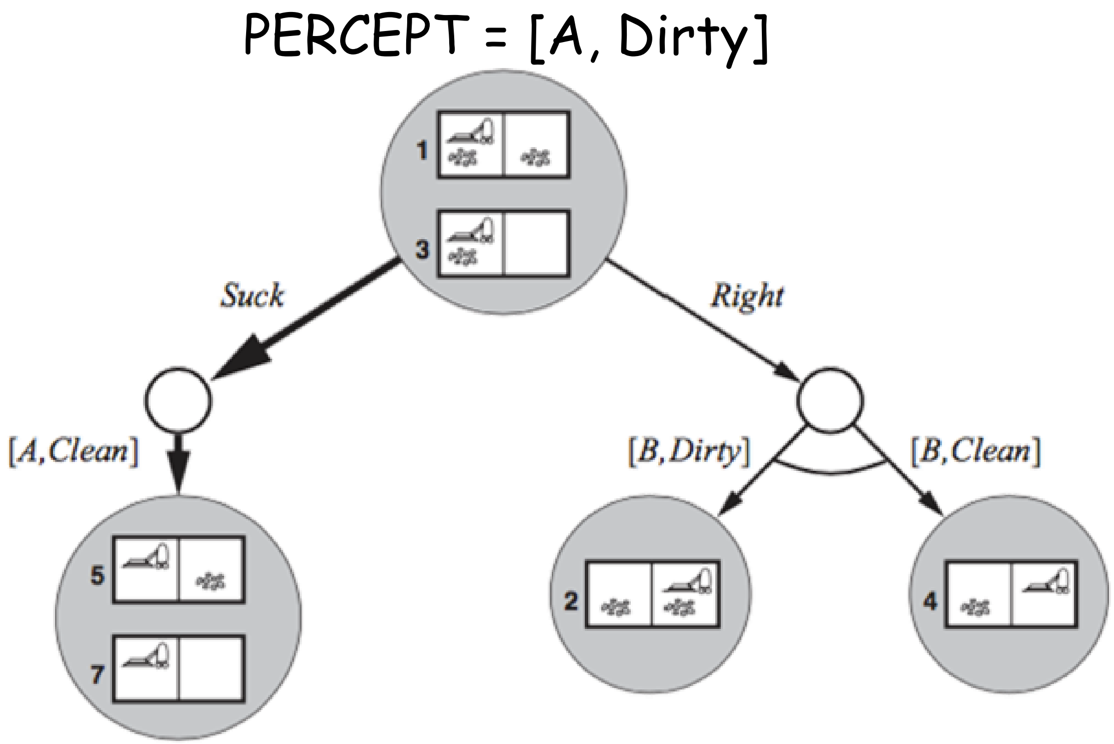
- Update stage
  - How is the belief state updated after a perception?

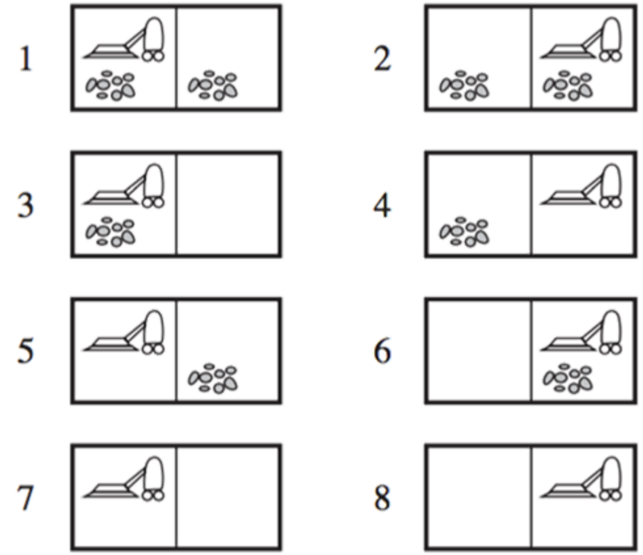
$$b_o = \text{UPDATE}(b', o) = \{s : o = \text{PERCEPT}(s) \text{ and } s \in b'\}$$

$$\text{RESULTS}(b, a) = \{b_o : b_o = \text{UPDATE}(\text{PREDICT}(b, a), o) \text{ and } o \in \text{POSSIBLE-PERCEPTS}(\text{PREDICT}(b, a))\}$$

Local sensing vacuum world

- AND-OR search tree on belief states





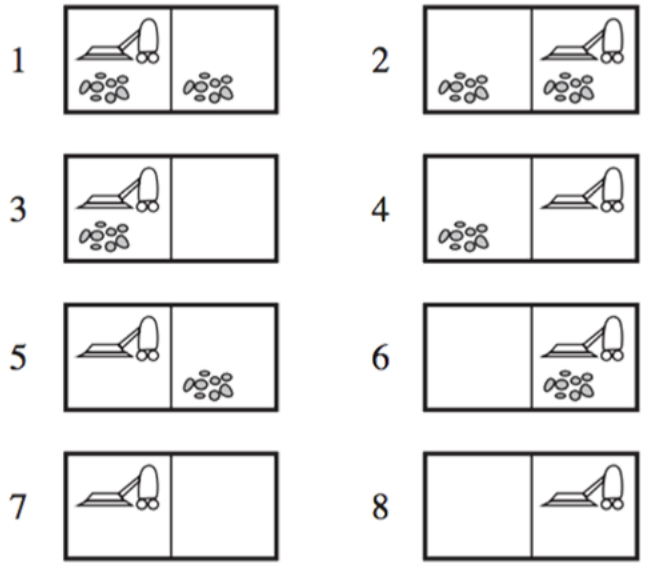
[Suck, Right, **if** Bstate={6} **then** Suck **else** []]

Solving partially observable problems

- AND-OR graph search
- Execute the obtained contingency plan
  - Based on the achieved perception either then-part or else-part of a condition is run
  - Agent's belief state is updated when performing actions and receiving percepts
    - Maintaining the belief state is a core function of any intelligent system

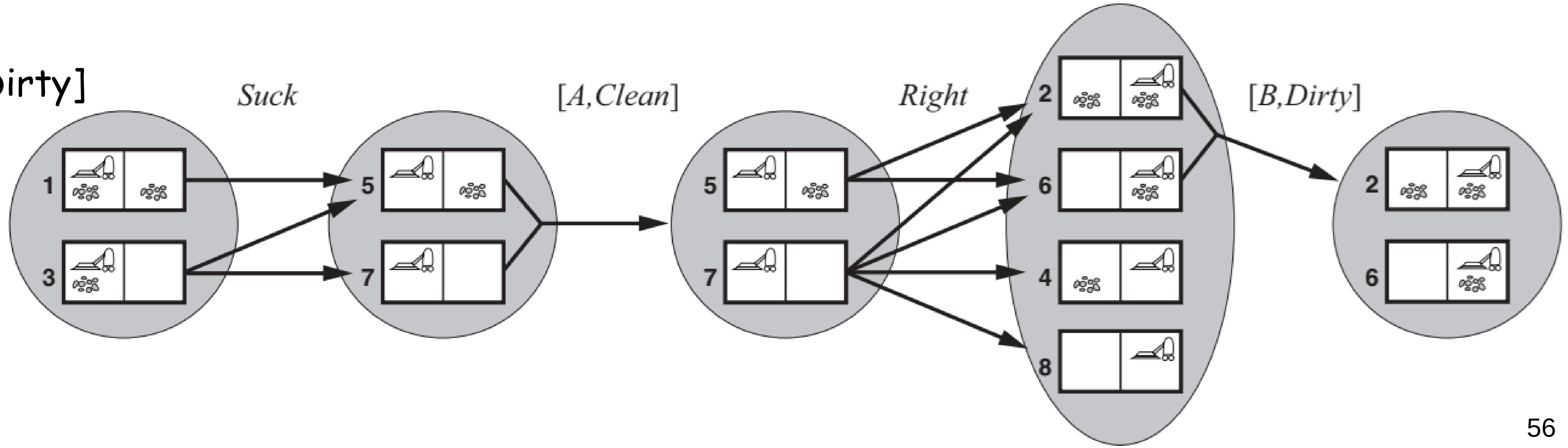
$$b' = \text{UPDATE}(\text{PREDICT}(b, a), o)$$

Vacuum world example



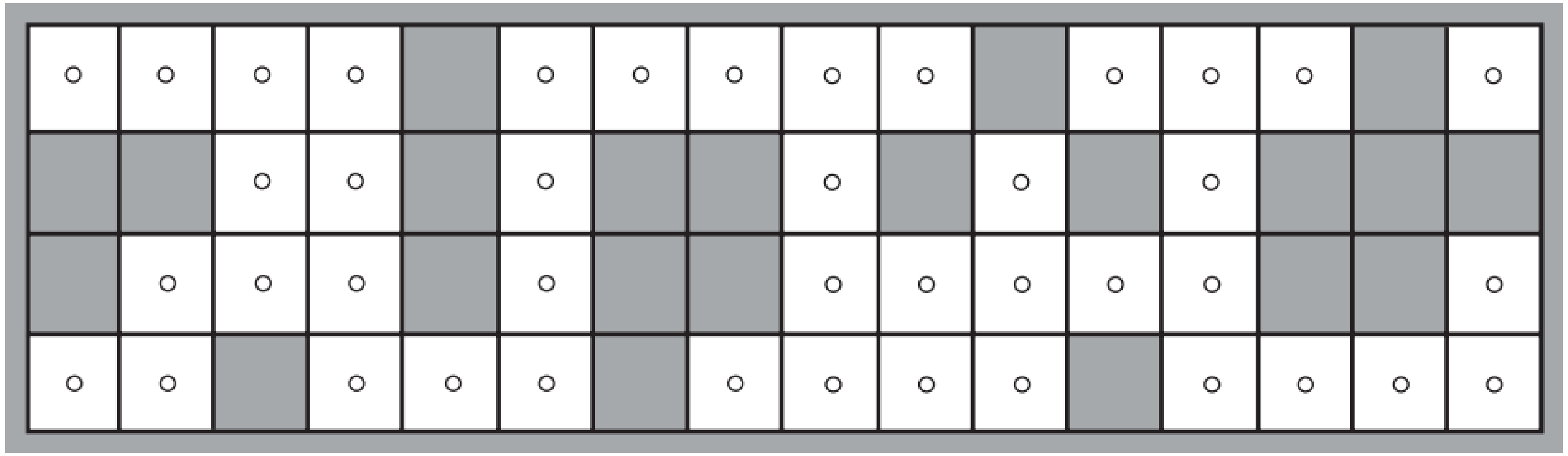
- Local sensing
- Any square may be dirty at any time (unless the agent is now cleaning it)

[A, Dirty]



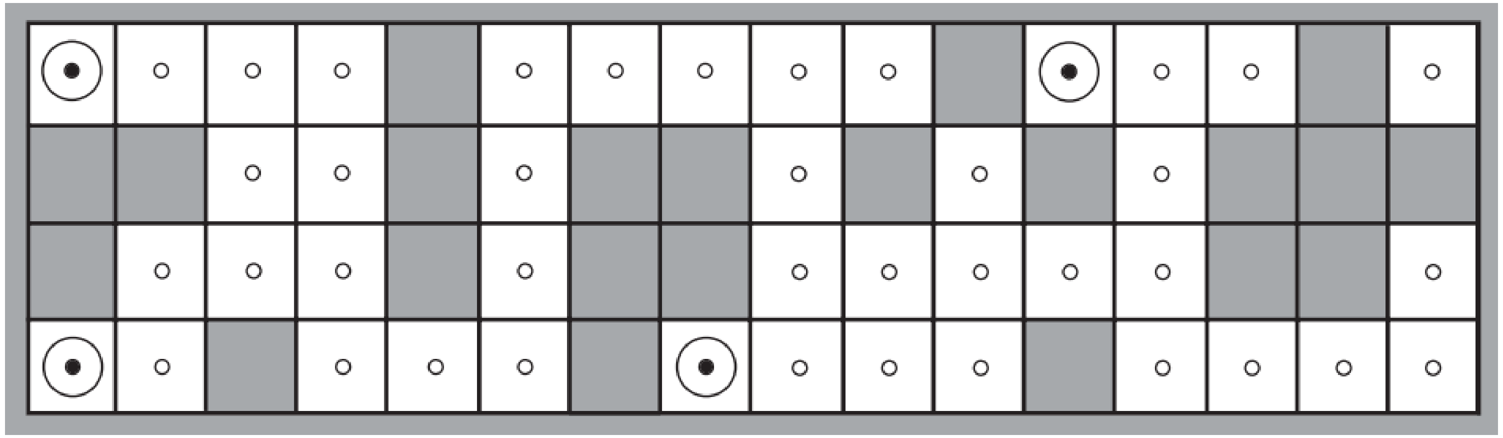
Robot localization example

- Determining current location given a map of the world and a sequence of percepts and actions
  - Perception
    - one sonar sensor in each direction (telling obstacle existence)
      - E.g., percepts=NW means there are obstacles to the north and west
  - Broken navigational system
    - Move action randomly chooses among {Right, Left, Up, Down}

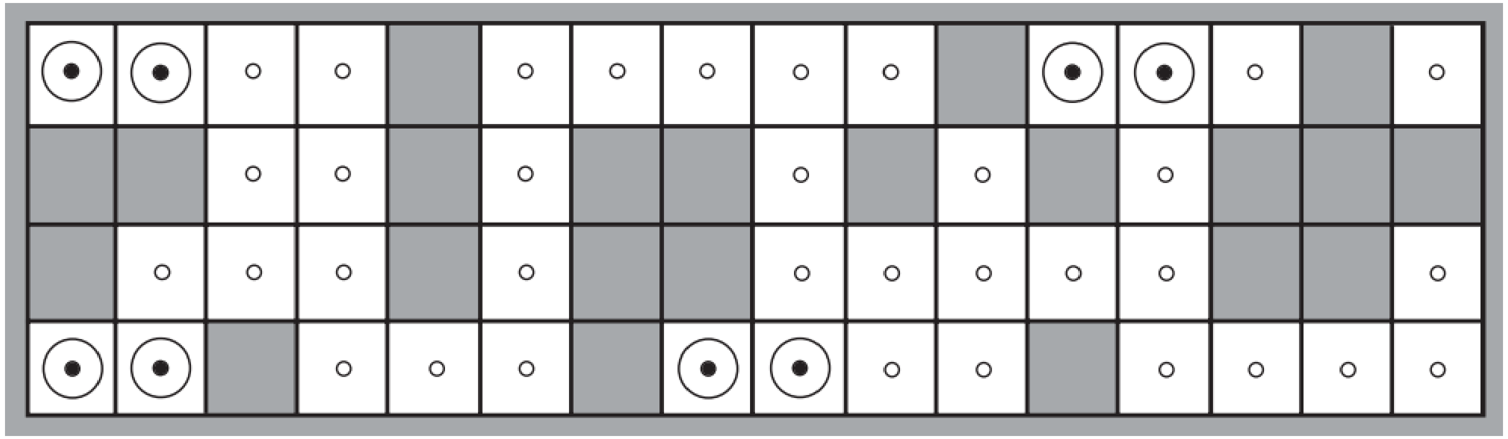


Robot localization example

- b^0 : o squares
- Percept: NSW
- b^1 = UPDATE(b^0, NSW)

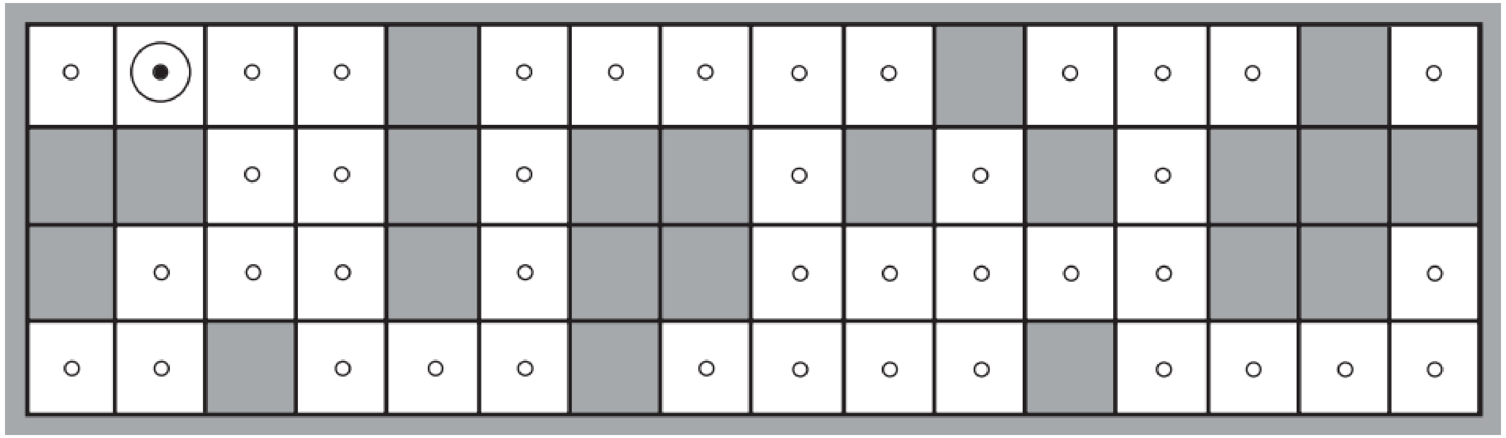


- Execute action a = Move
- b^1_a = PREDICT(b^1, a)



Robot localization example

- Percept: NS
- $b^2 = \text{UPDATE}(b^1_a, NS)$



- This is the only location that could be the result of

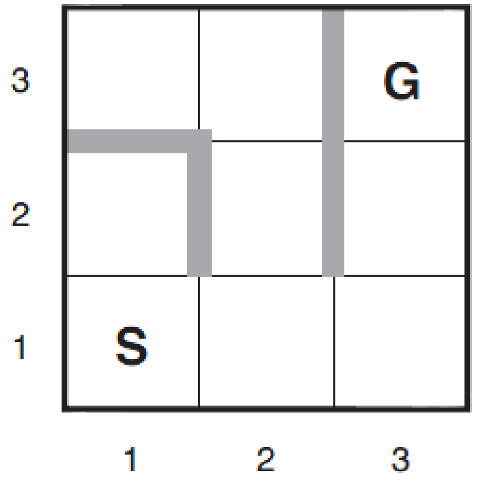
$$\boxed{\text{UPDATE}(\text{PREDICT}(\text{UPDATE}(b, NSW), Move), NS)}$$

Online search

- Off-line Search
  - Solution is found before the agent starts acting in the real world
- On-line search
  - Interleaves search and acting
  - Necessary in unknown environments
  - Useful in dynamic and semi-dynamic environments
  - Saves computational resource in non-deterministic domains (focusing only on the contingencies arising during execution)
    - Tradeoff between finding a guaranteed plan (to not get stuck in an undesirable state during execution) and required time for complete planning ahead
- Examples
  - Robot in a new environment must explore to produce a map
  - Autonomous vehicles

Online search problems

- We assume deterministic & fully observable environment
  - we assume the agent knows
    - ACTIONS(s)
    - c(s, a, s') (can be used after knowing s' as the outcome)
    - GOAL_TEST(s)
- Agent must perform an action to determine its outcome
  - RESULTS(s, a) is found by actually being in s and doing a
  - By filling RESULTS map table, the map of the environment is found
- Agent may access to a heuristic function

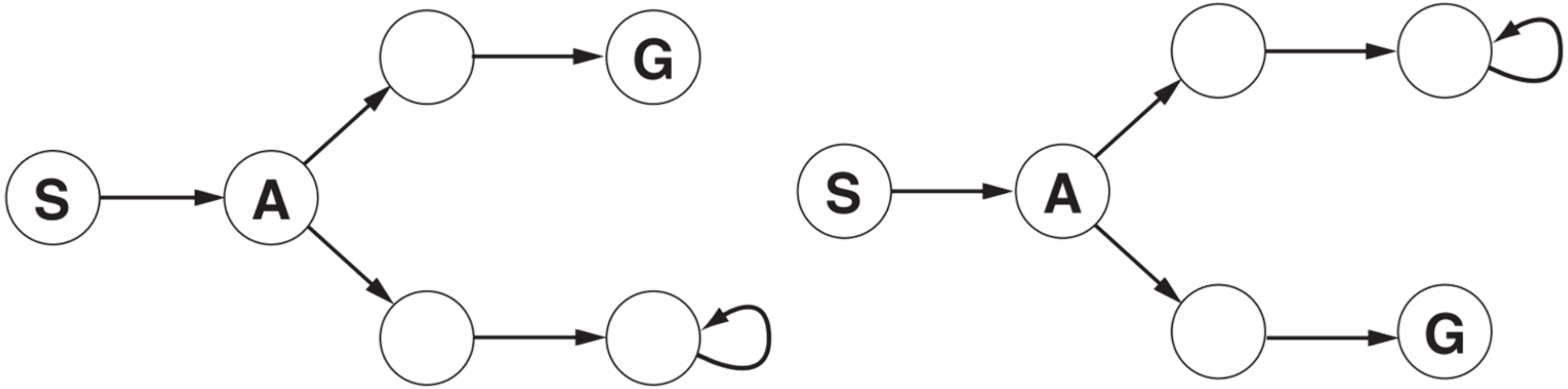


Competitive ratio

- Typically, the agent's objective is to reach a goal state while minimizing cost
  - Online path cost
    - Total cost of the path that the agent actually travels
  - Best cost
    - Cost of the shortest path "if it knew the search space in advance"
- Competitive ratio = Online path cost / Best path cost
  - Smaller values are more desirable
- Competitive ratio may be infinite
  - Dead-end state: no goal state is reachable from it
    - irreversible actions can lead to a dead-end state

Infinite Competitive ratio (Dead-end)

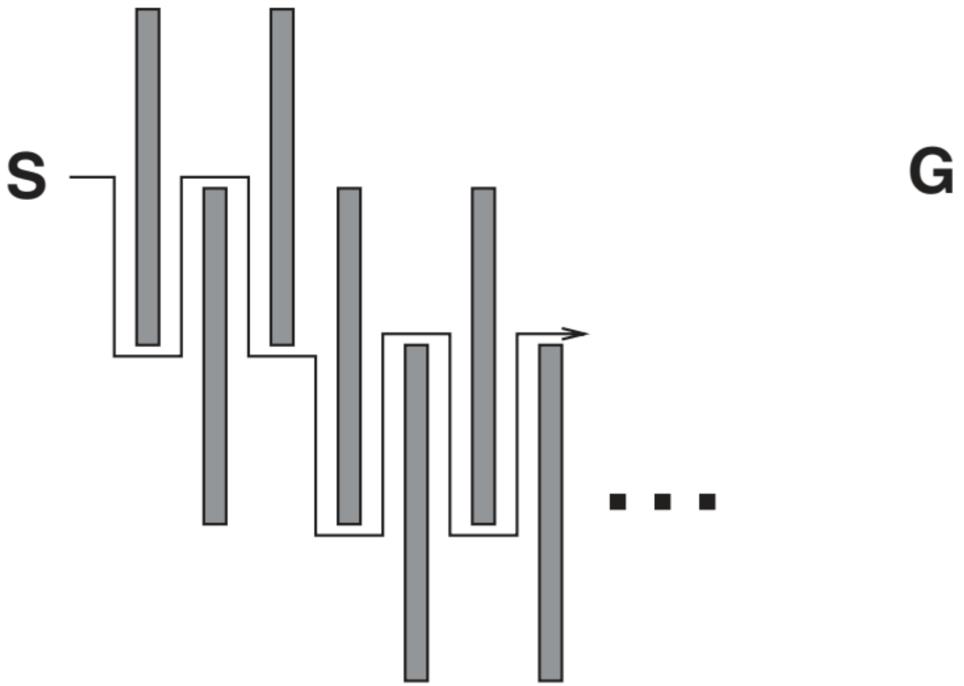
- No algorithm can avoid dead-ends in all state spaces



- Simplifying assumption: Safely explorable state space
  - A goal state is achievable from every reachable state
    - Example: State spaces with reversible actions

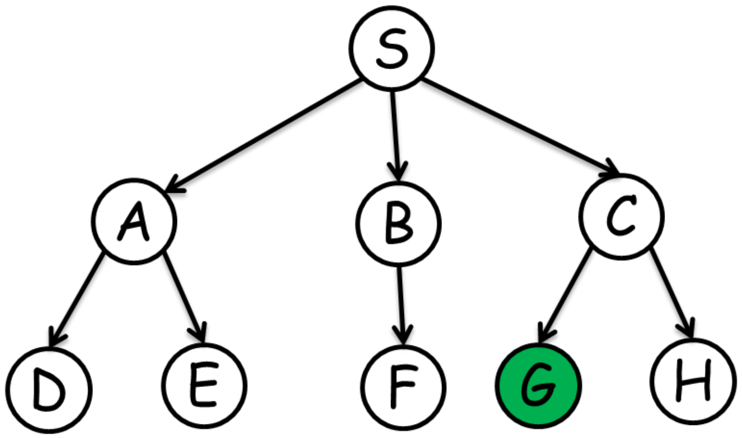
Infinite Competitive ratio (Unbounded cost)

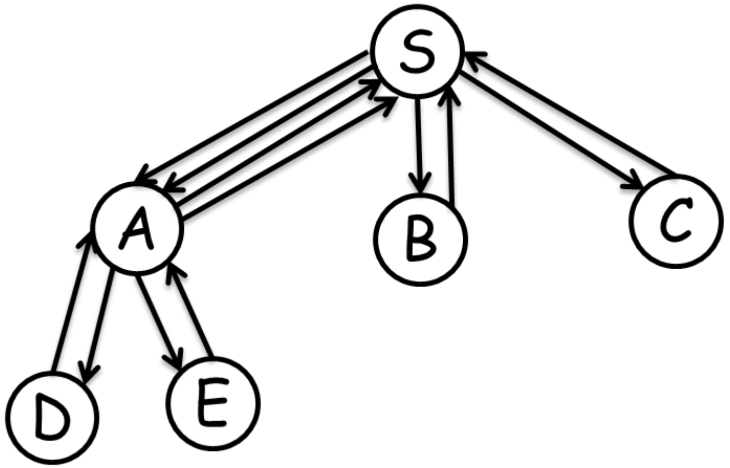
- A two-dimensional environment that can cause an online search agent to follow an arbitrarily inefficient route to the goal.
  - Whichever choice the agent makes, the adversary blocks that route with another long, thin wall, so that the path followed is much longer than the best possible path



Online search agents

- Offline search
  - Node expansion involves simulated rather than real actions
  - Can expand a node in one part of the space and then immediately expand a node in another part of the space
- Online search
  - Can discover successors only for a node that it physically occupies
  - To avoid traveling all the way across the tree to expand the next node, it seems better to expand nodes in a local order

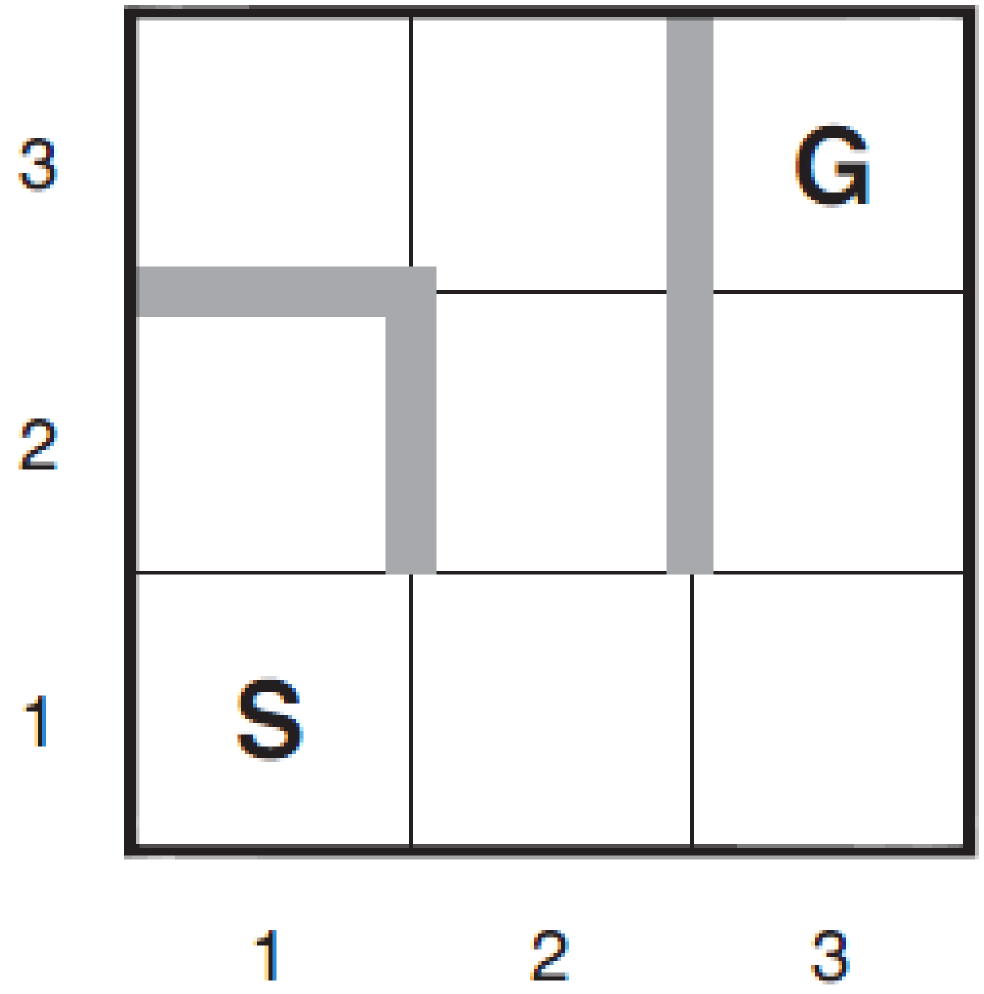


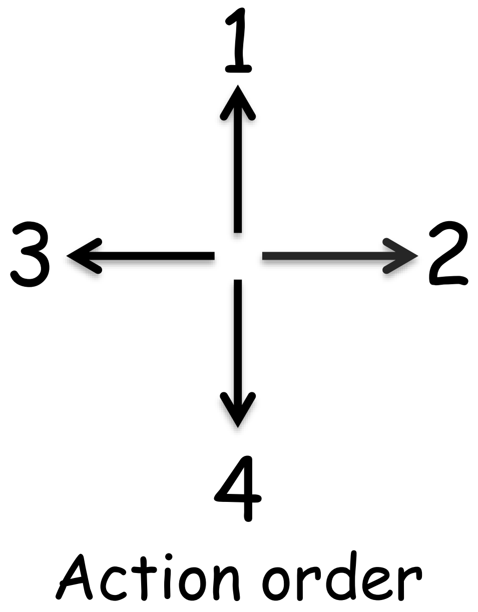


Online search agents

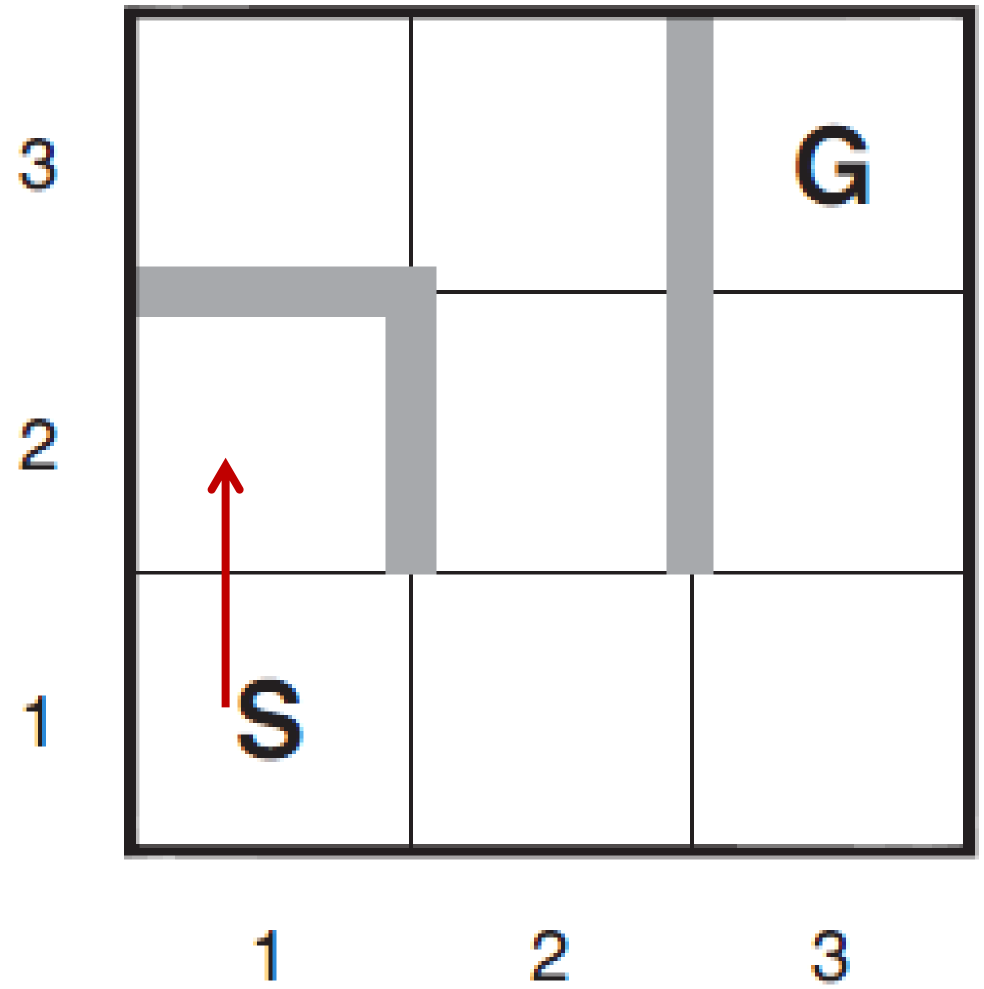
- Online DFS
  - Whenever an action from the current state has not been explored, the agent tries that action.
  - Physical backtrack
    - When the agent has tried all the actions in a state
    - Goes back to the state from which the agent most recently entered the current state
    - Works only for state spaces with reversible actions

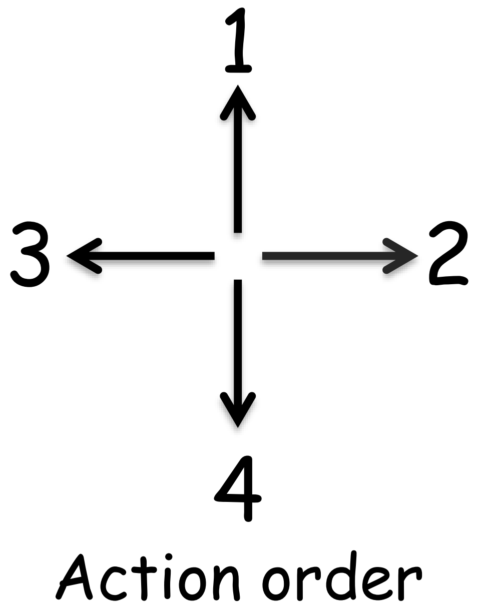
Online DFS (Example)



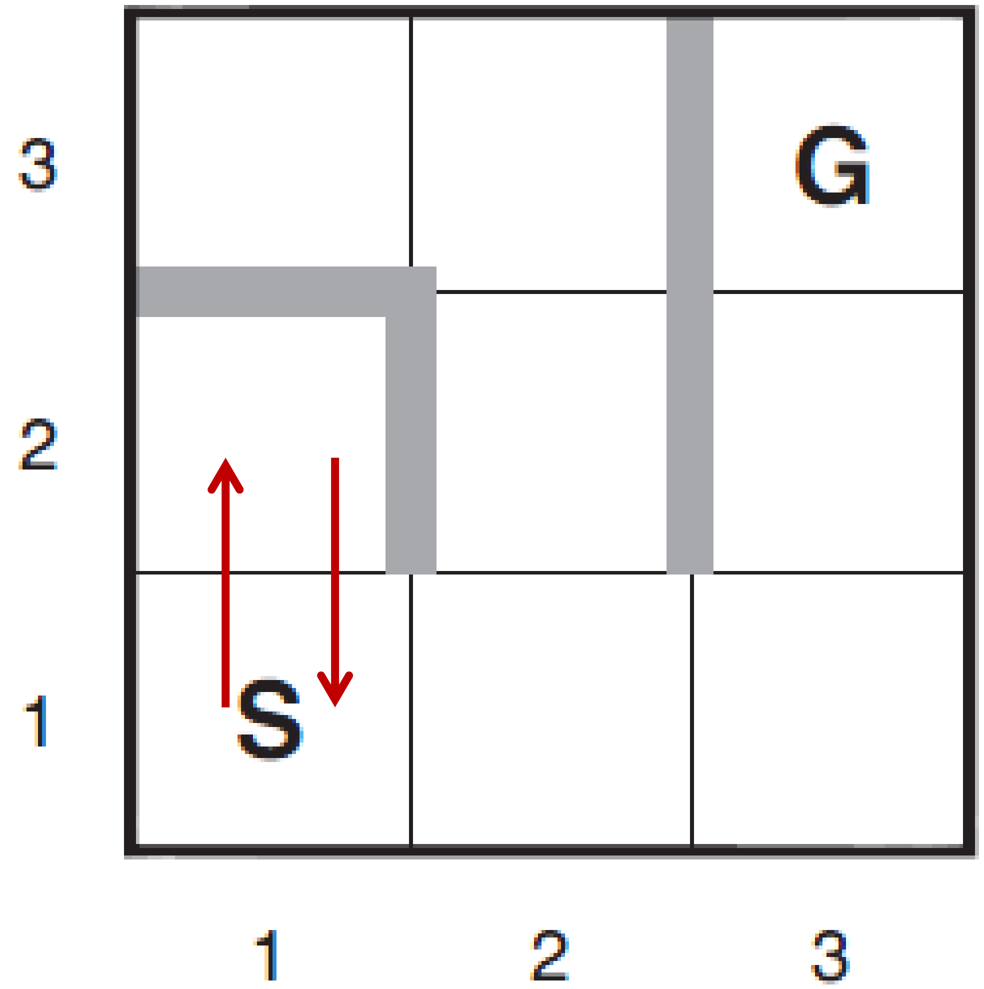


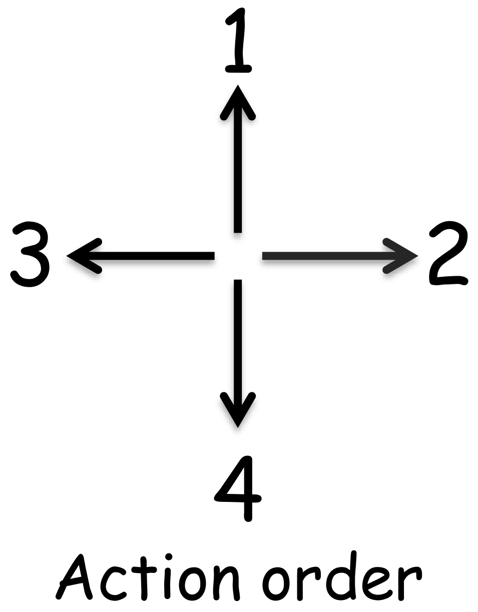
Online DFS (Example)



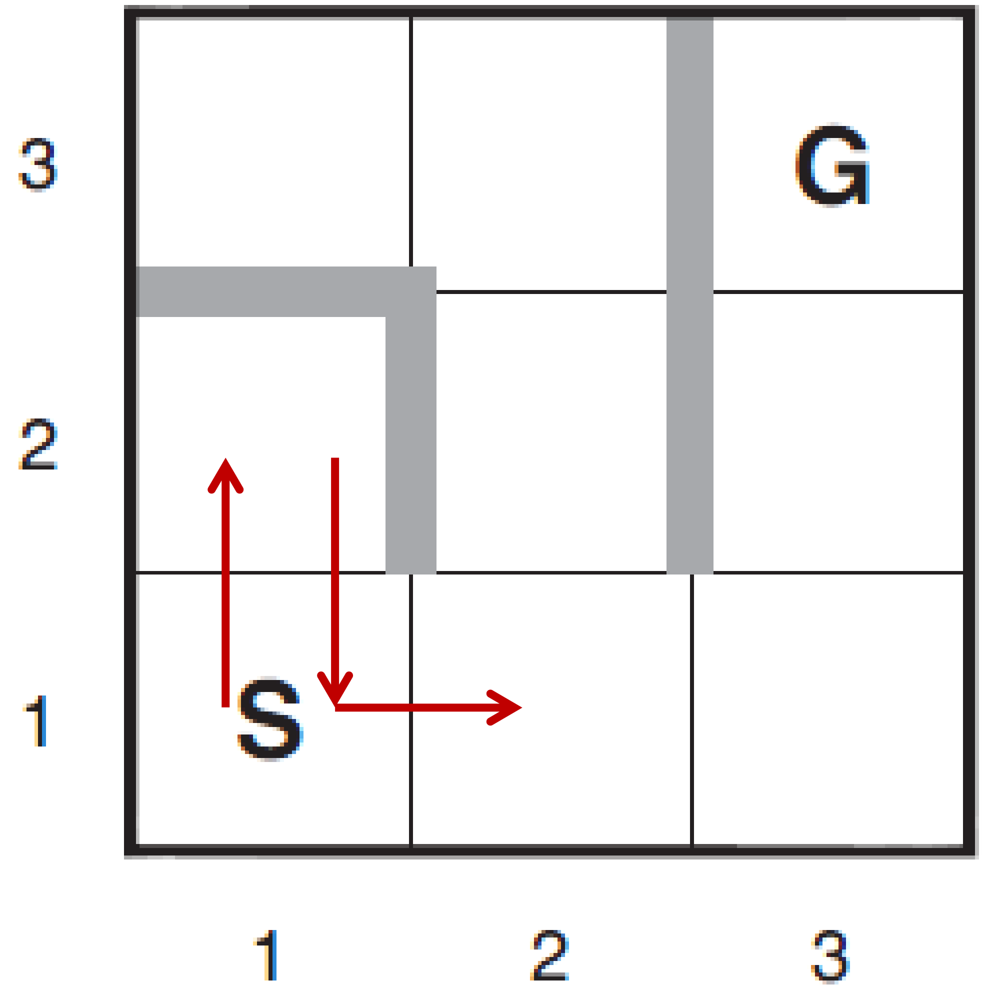


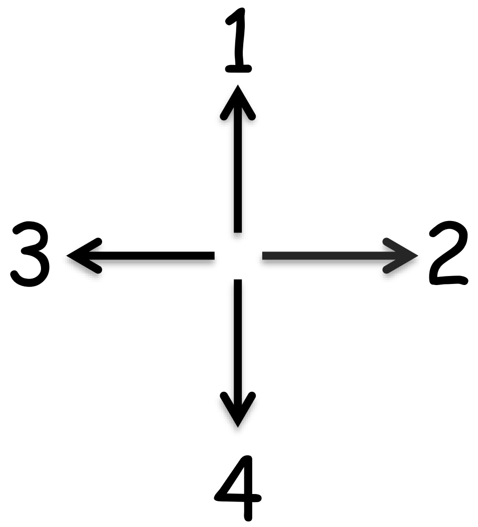
Online DFS (Example)



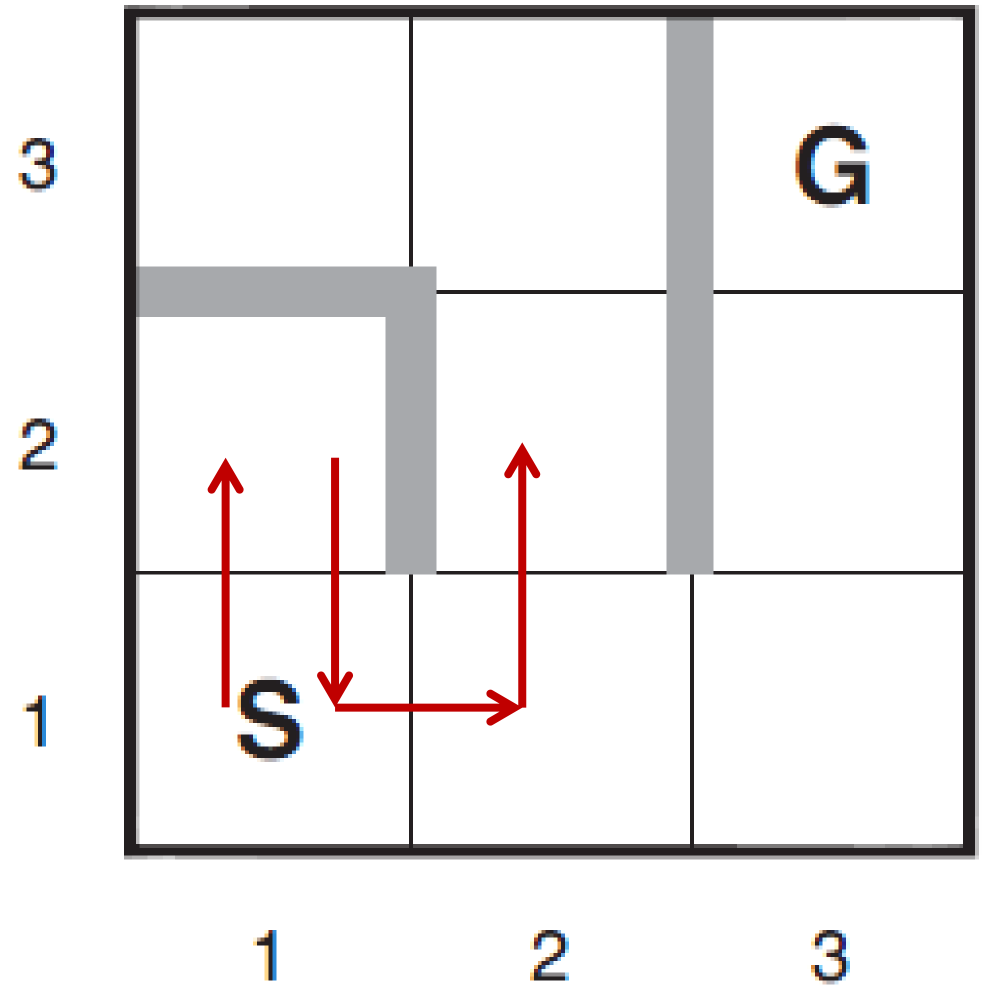


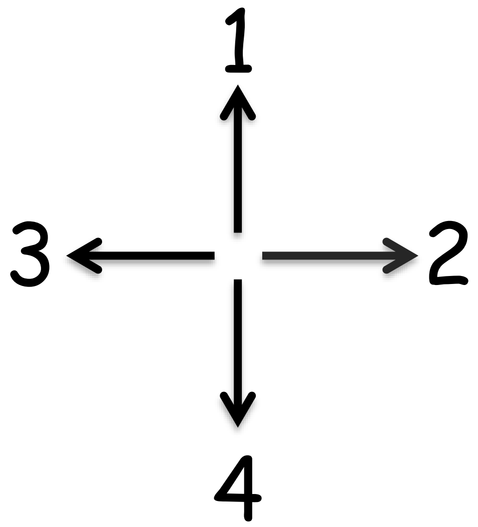
Online DFS (Example)





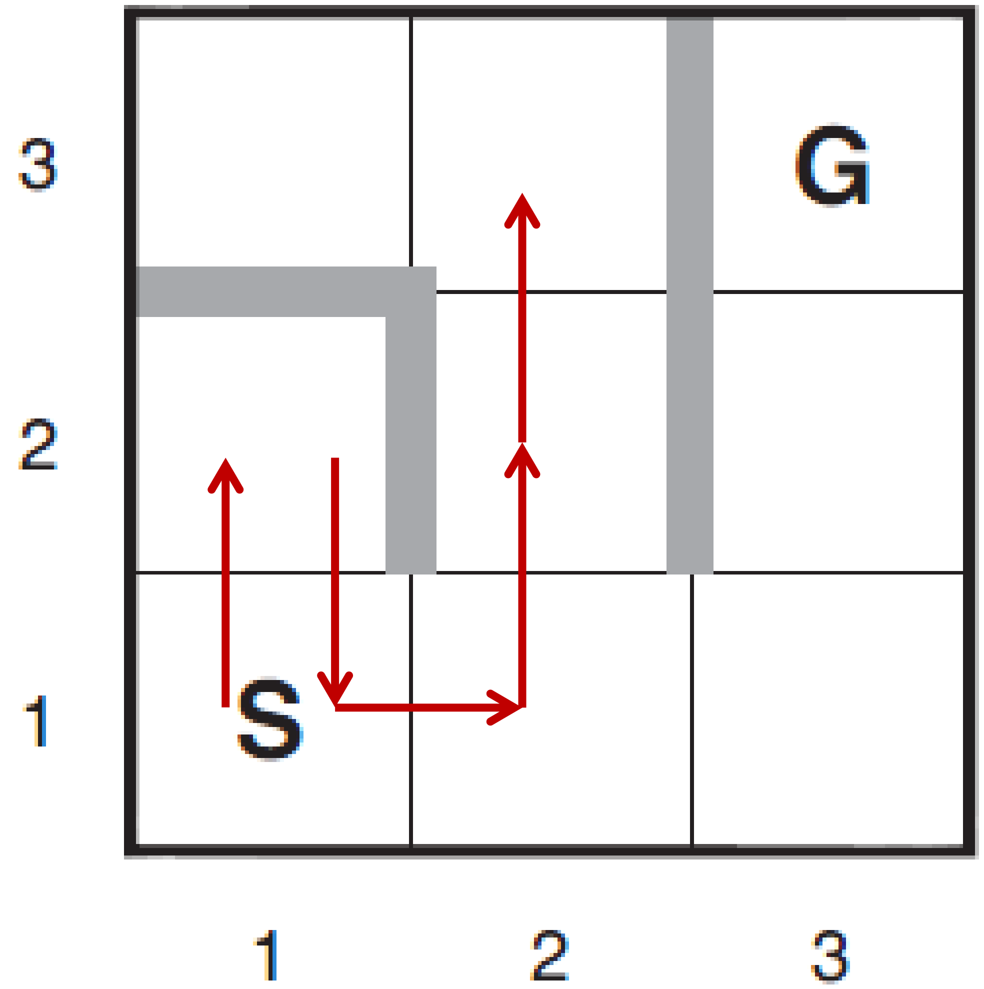
Online DFS (Example)

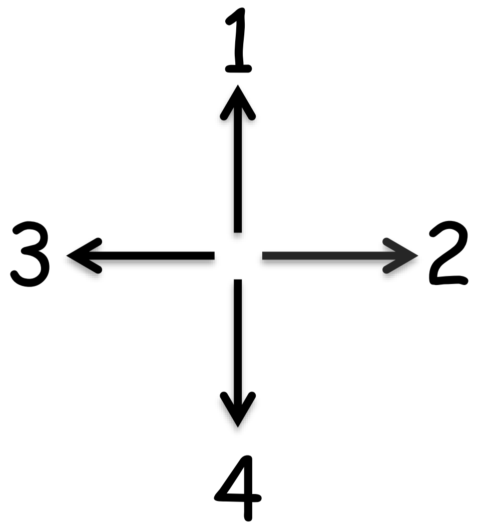




Action order

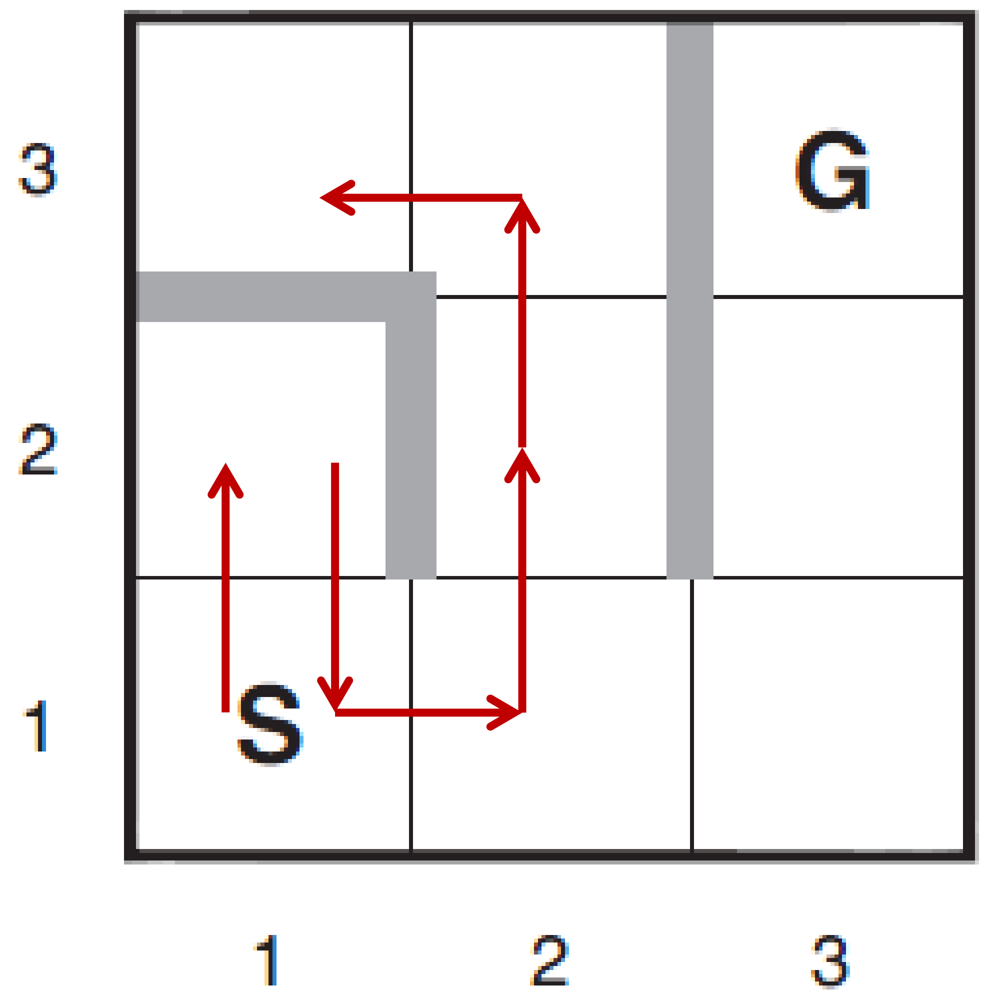
Online DFS (Example)

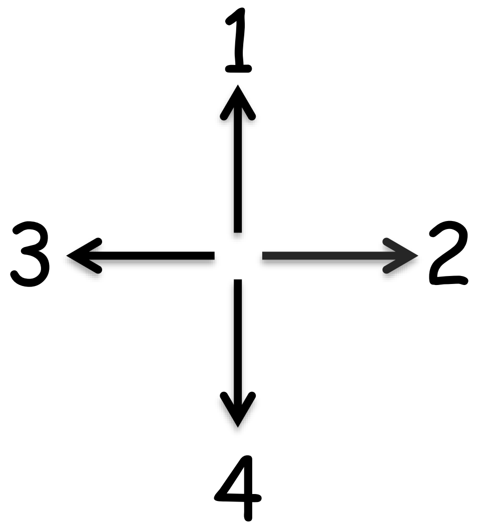




Action order

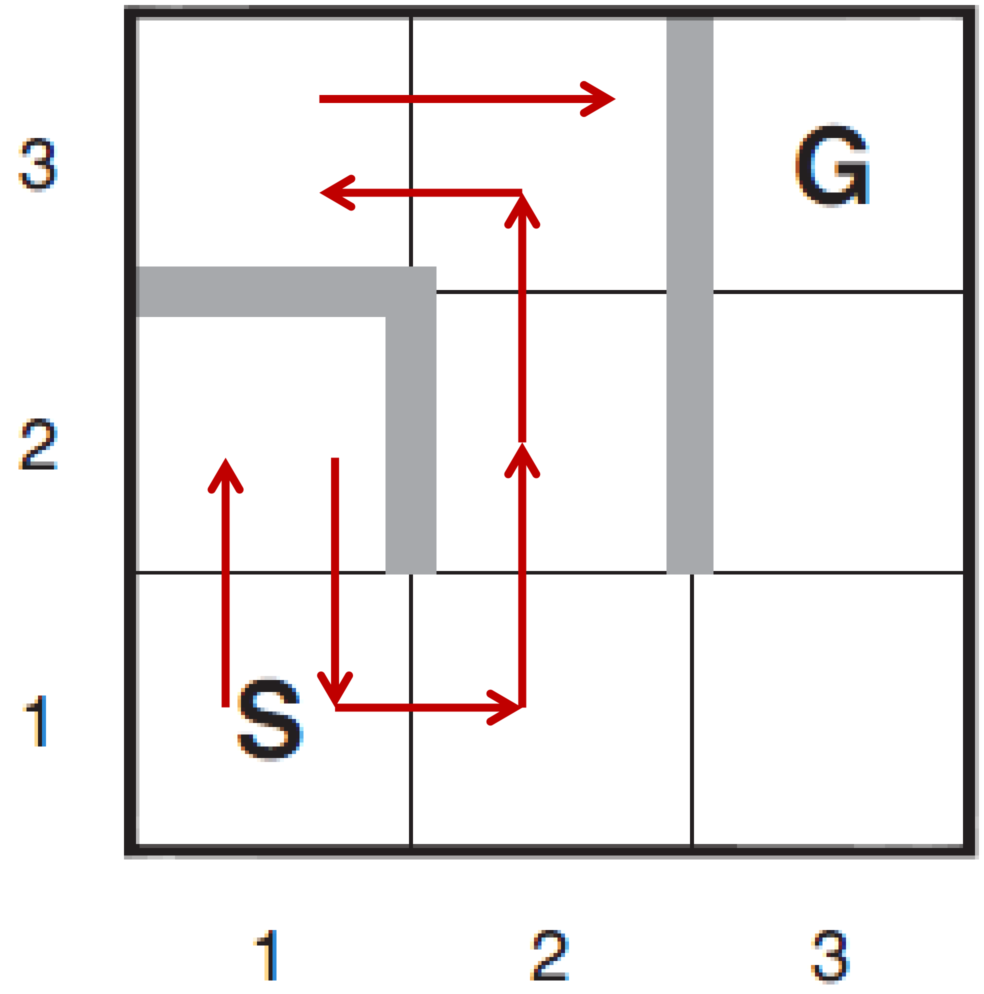
Online DFS (Example)

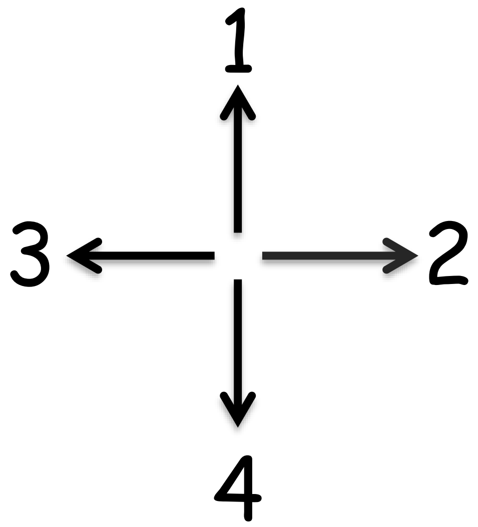




Action order

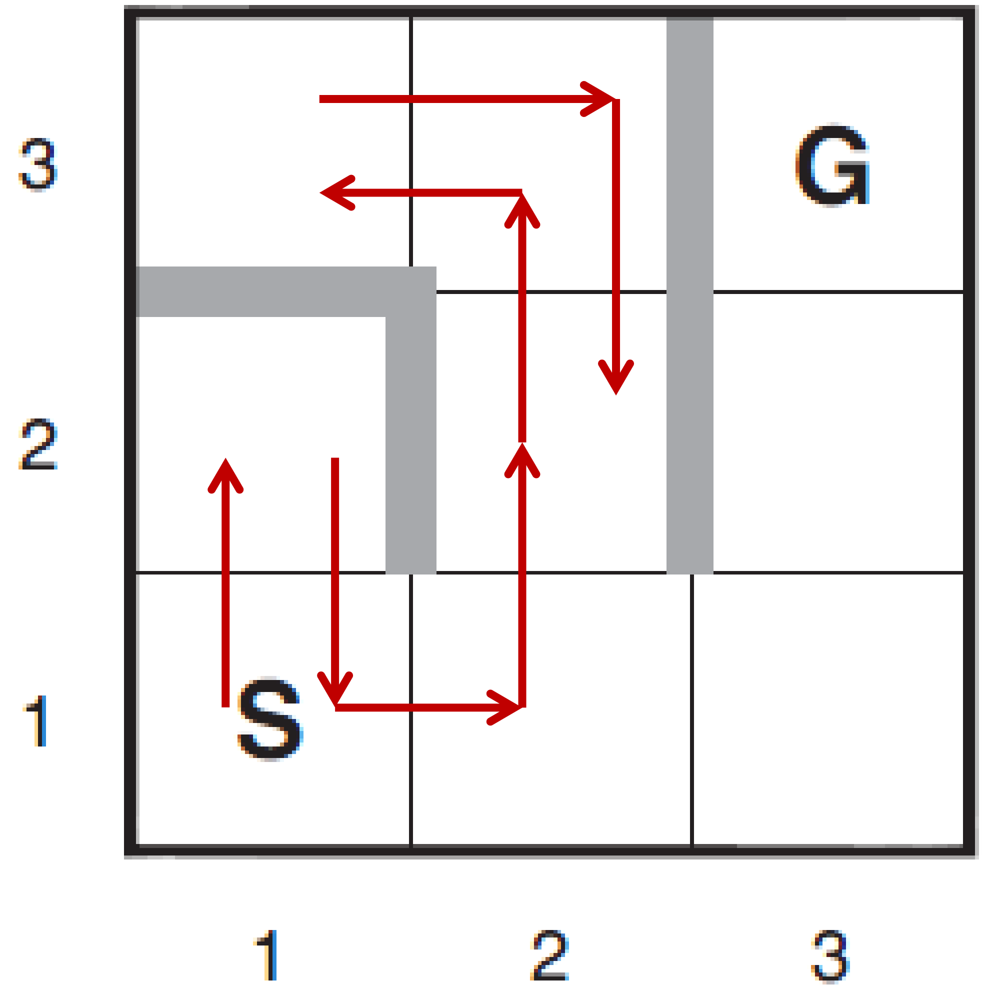
Online DFS (Example)

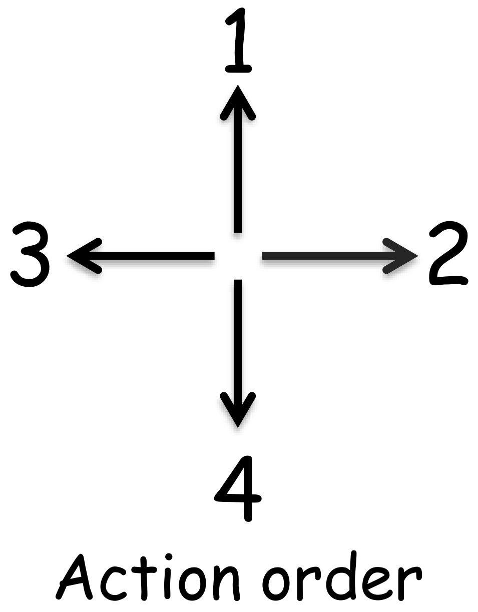




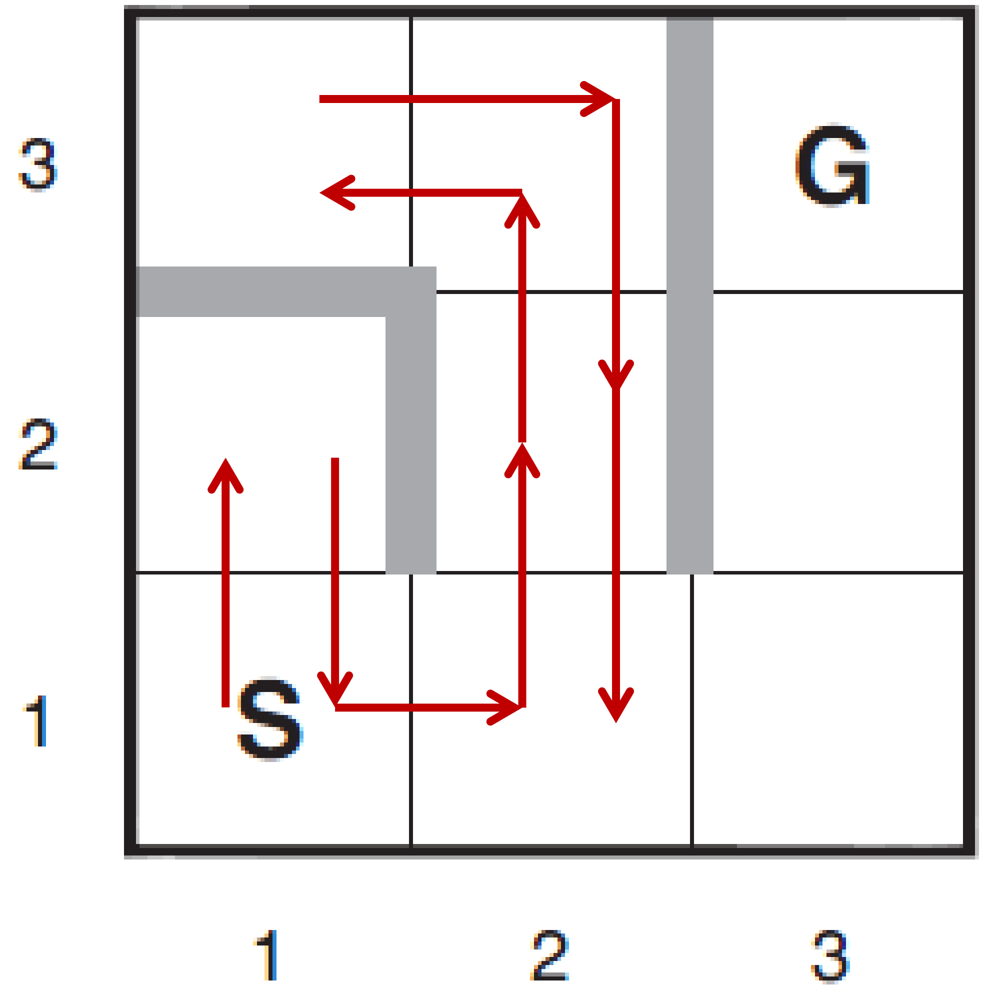
Action order

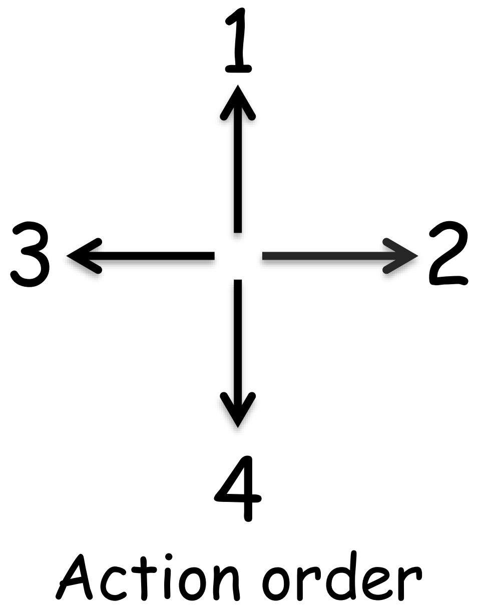
Online DFS (Example)



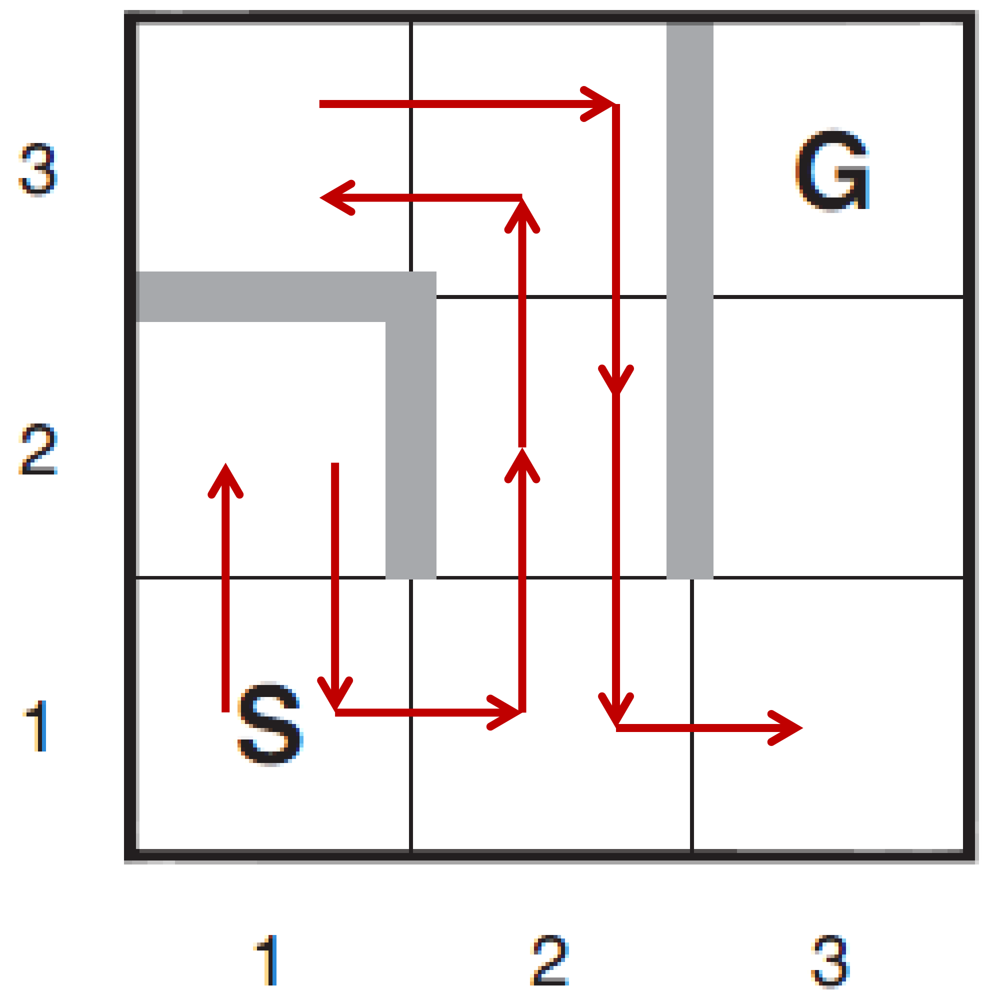


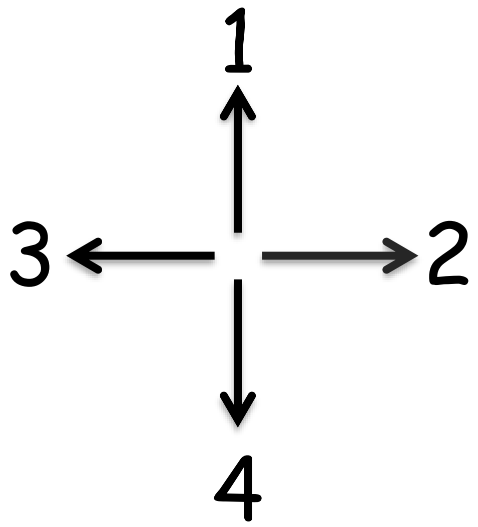
Online DFS (Example)





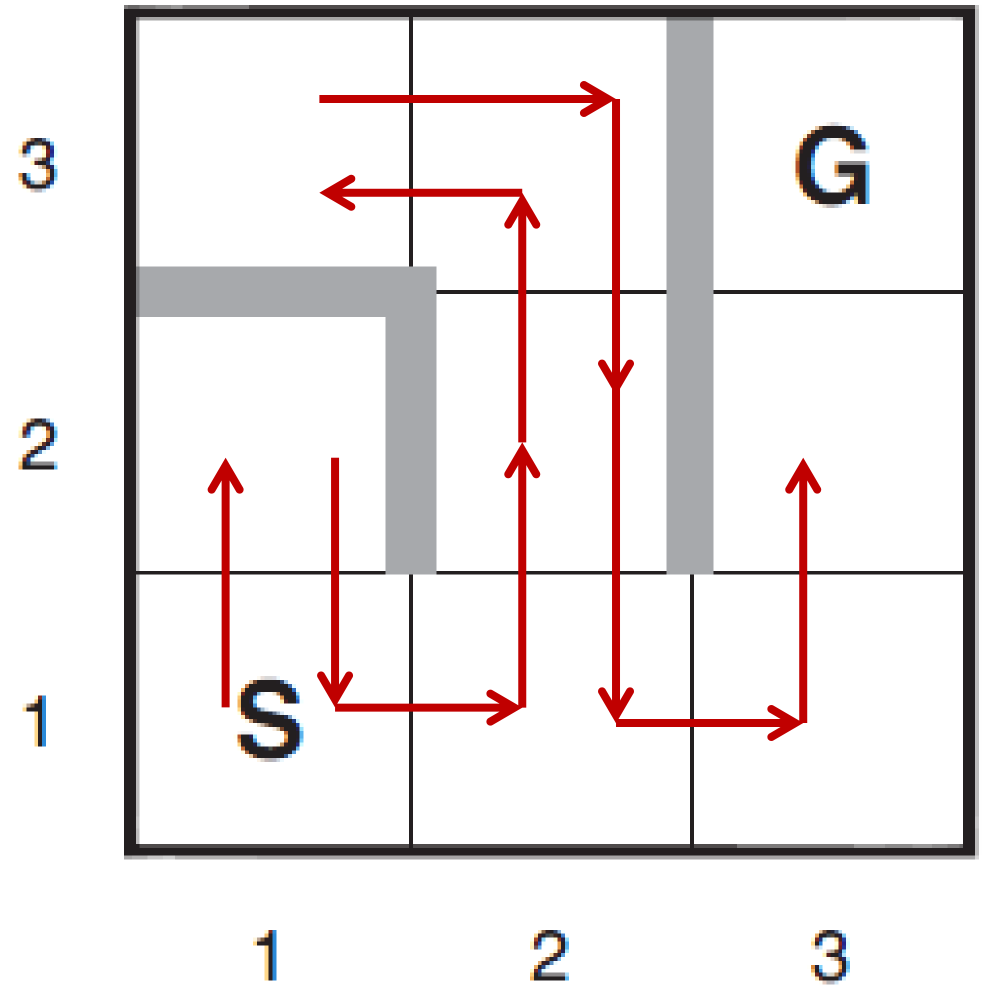
Online DFS (Example)

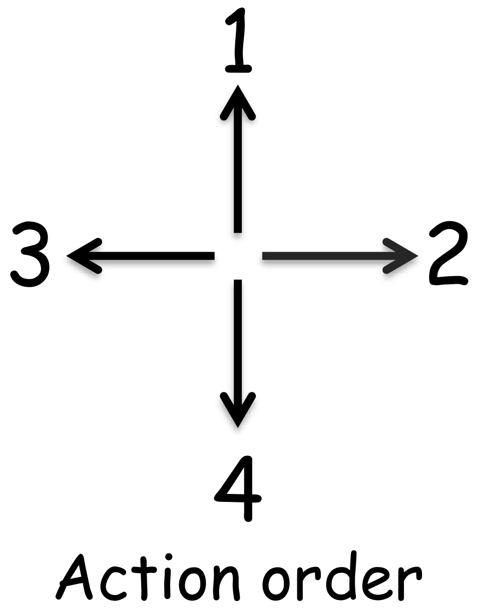




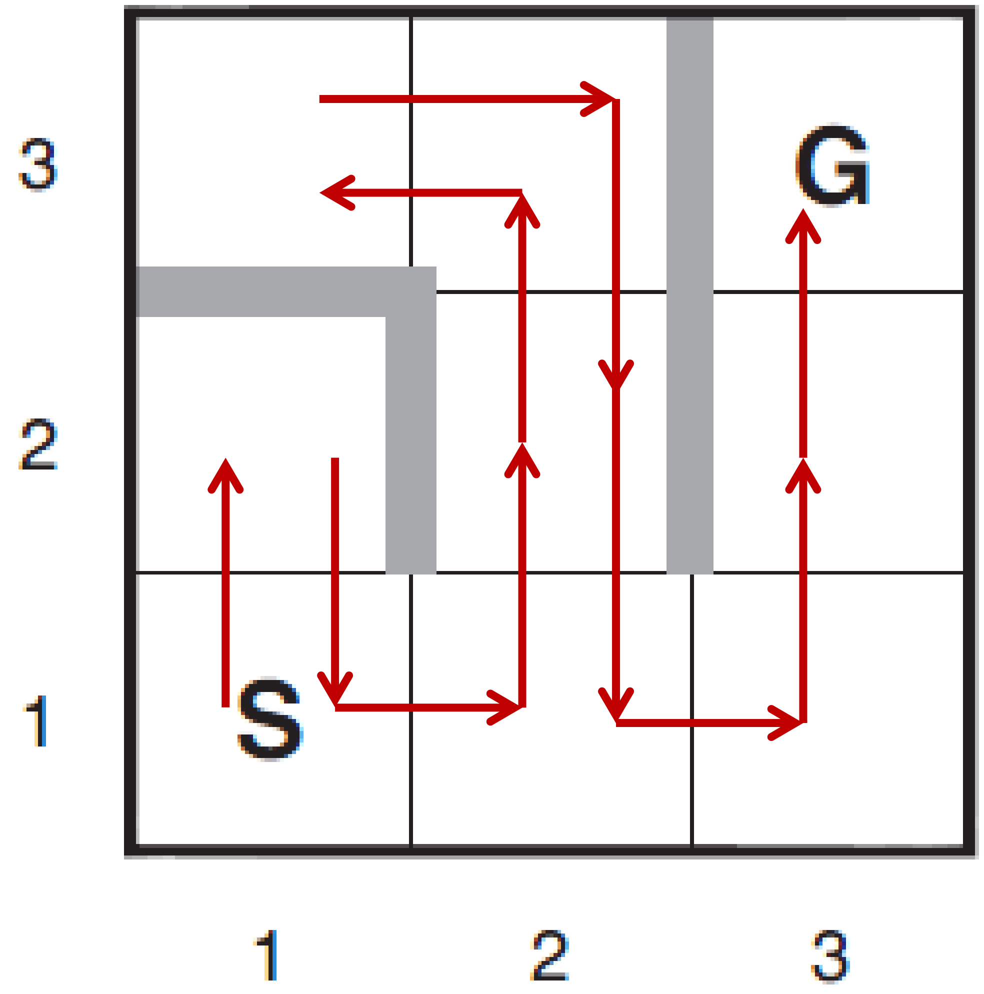
Action order

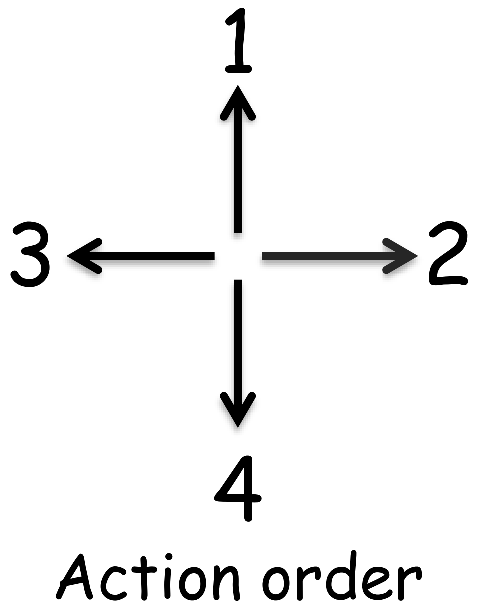
Online DFS (Example)





Online DFS (Example)

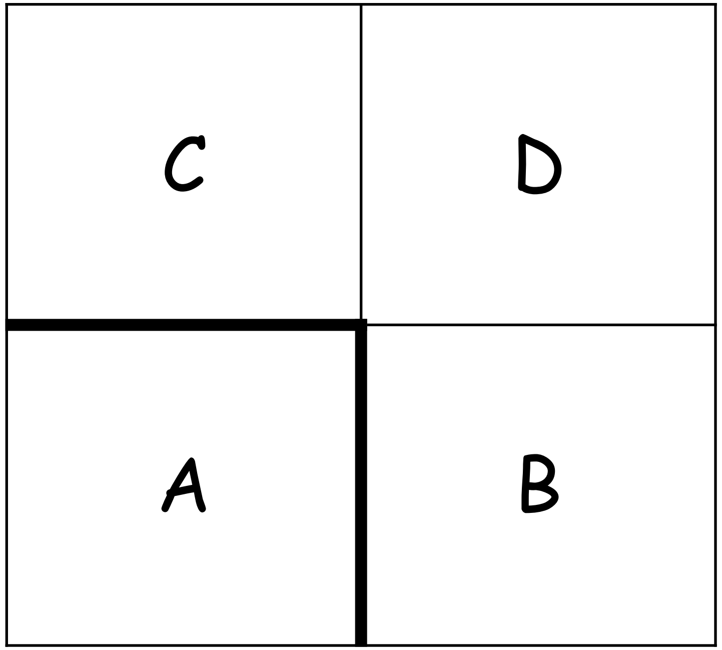




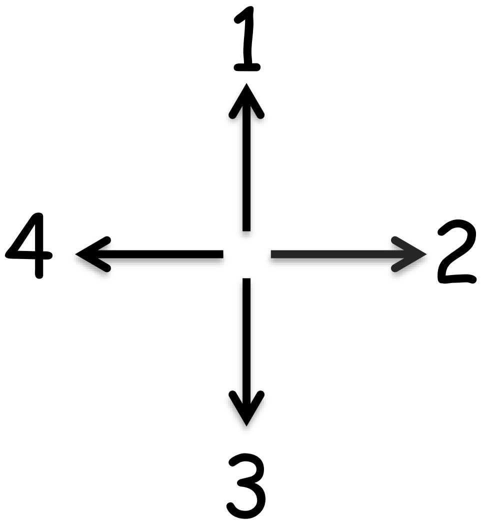
An online search agent that uses depth-first exploration

```
function ONLINE-DFS-AGENT(s') returns an action
    inputs: s', a percept that identifies the current state
    persistent: result, a table indexed by state and action, initially empty
                untried, a table that lists, for each state, the actions not yet tried
                unbacktracked, a table that lists, for each state, the backtracks not yet tried
                s, a, the previous state and action, initially null
    if GOAL-TEST(s') then return stop
    if s' is a new state (not in untried) then untried[s'] ← ACTIONS(s')
    if s is not null then
        result[s, a] ←s'
        add s to the front of unbacktracked[s']
    if untried[s'] is empty then
        if unbacktracked[s'] is empty then return stop
        else a ← an action b such that result[s', b] = POP(unbacktracked[s'])
    else a ← POP(untried[s'])
    s ← s'
    return a
```

Online DFS (Example)

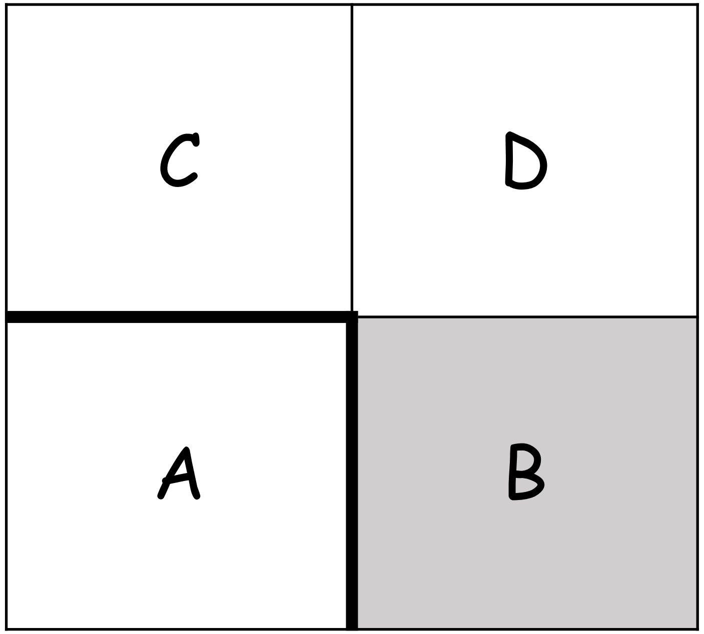


Start: B
Goal: C



Action order

Online DFS (Example)



Step 1

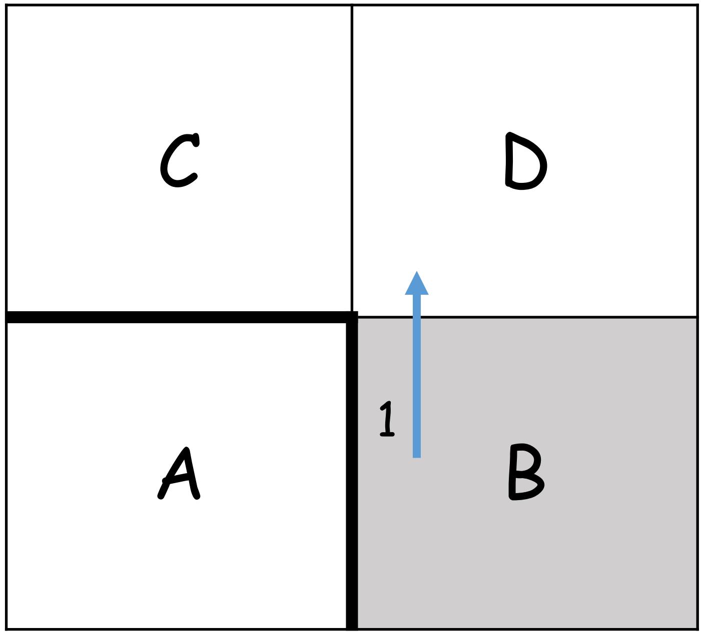
s' = B s = null

untried[B] = {u, r, d, l}

a = u → untried[B] = {r, d, l}

s = B

Online DFS (Example)



Step 1

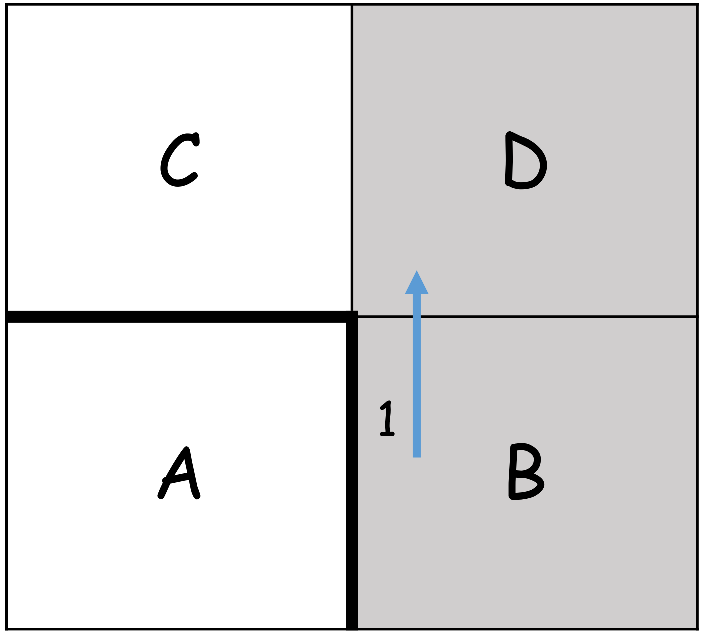
s' = B s = null

untried[B] = {u, r, d, l}

a = u → untried[B] = {r, d, l}

s = B

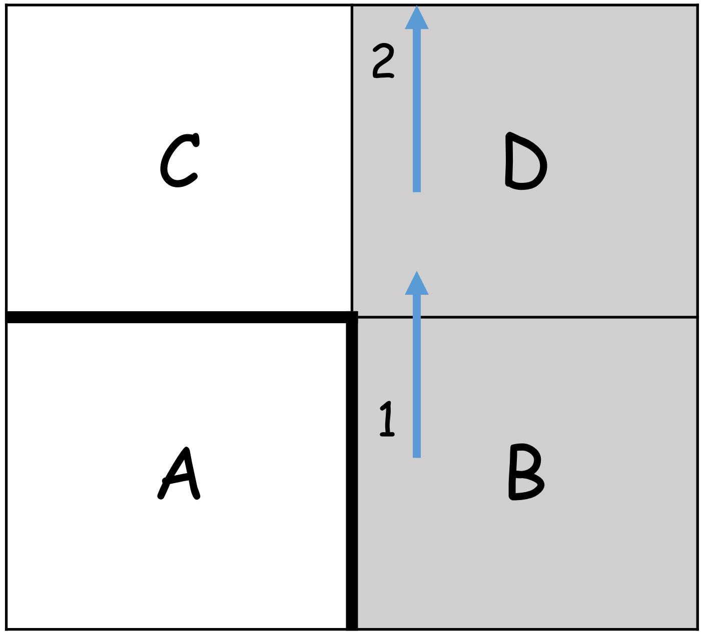
Online DFS (Example)



Step 2

s' = D s = B
untried[D] = {u, r, d, l}
result[B,u] = D
unbacktracked[D] = {B}
a = u → untried[D] = {r, d, l}
s = D

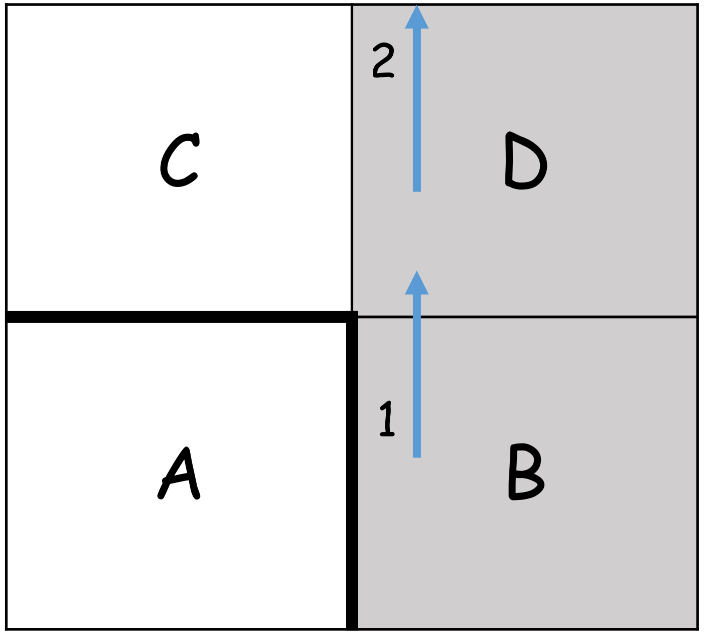
Online DFS (Example)



Step 2

s' = D   s = B
untried[D] = {u, r, d, l}
result[B,u] = D
unbacktracked[D] = {B}
a = u $\rightarrow$ untried[D] = {r, d, l}
s = D

Online DFS (Example)



Step 3

s' = D s = D

untried[D] = {r, d, l}

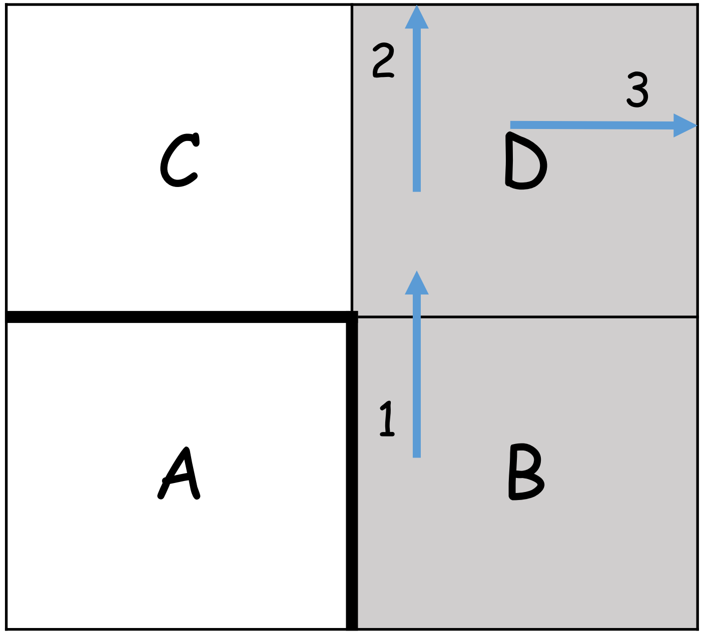
result[D,u] = D

unbacktracked[D] = {B}

a = r → untried[D] = {d, l}

s = D

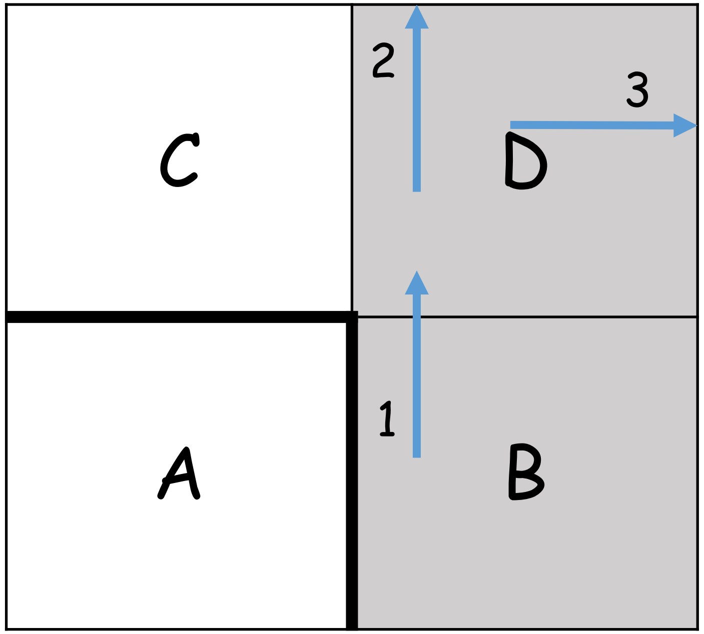
Online DFS (Example)



Step 3

s' = D s = D
untried[D] = {r, d, l}
result[D,u] = D
unbacktracked[D] = {B}
a = r → untried[D] = {d, l}
s = D

Online DFS (Example)



Step 4

s' = D s = D

untried[D] = {d, l}

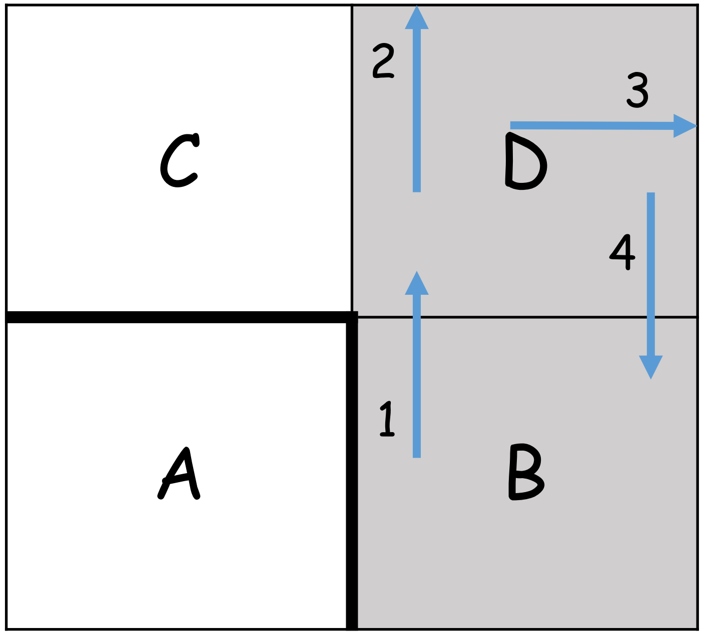
result[D,r] = D

unbacktracked[D] = {B}

a = d → untried[D] = {l}

s = D

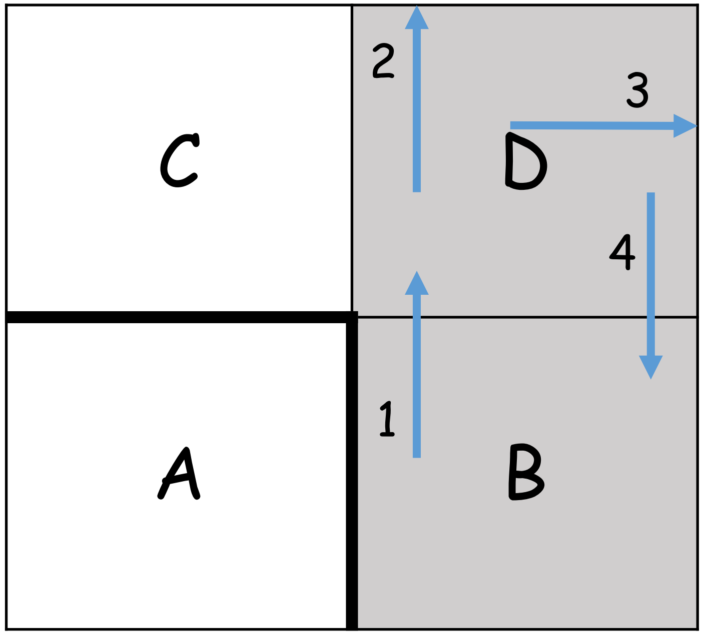
Online DFS (Example)



Step 4

s' = D s = D
untried[D] = {d, l}
result[D,r] = D
unbacktracked[D] = {B}
a = d → untried[D] = {l}
s = D

Online DFS (Example)



Step 5

s' = B s = D

untried[B] = {r, d, l}

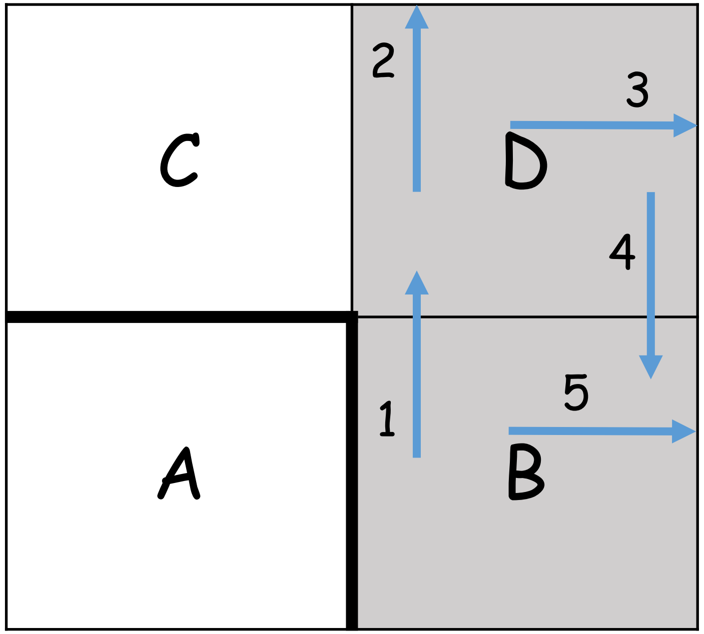
result[D,d] = B

unbacktracked[B] = {D}

a = r → untried[B] = {d, l}

s = B

Online DFS (Example)



Step 5

s' = B   s = D

untried[B] = {r, d, l}

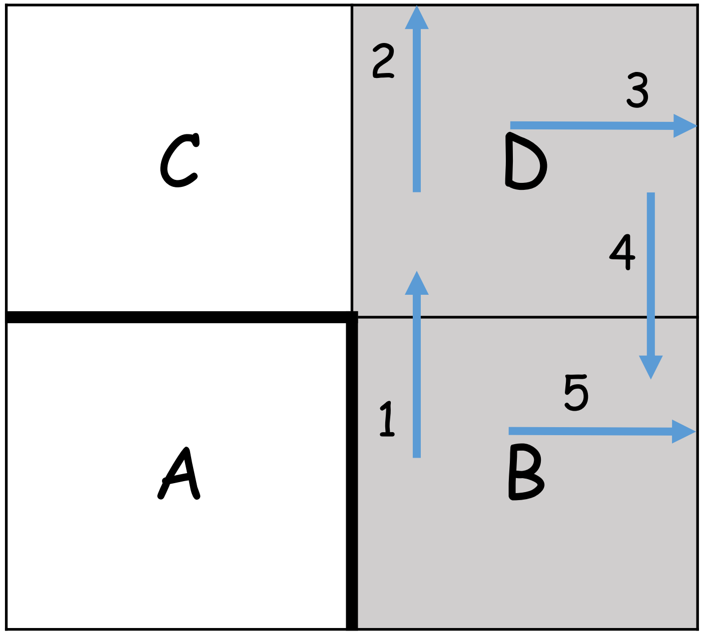
result[D,d] = B

unbacktracked[B] = {D}

a = r $\rightarrow$ untried[B] = {d, l}

s = B

Online DFS (Example)



Step 6

s' = B s = B

untried[B] = {d, l}

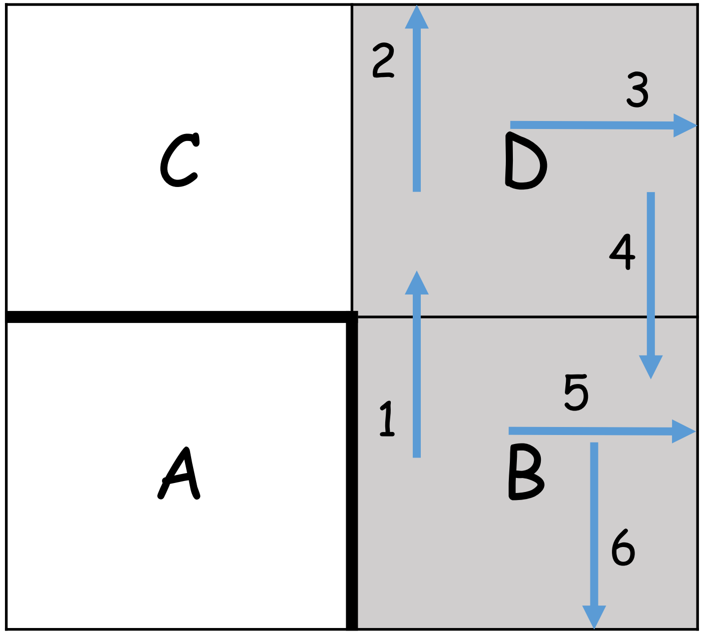
result[B, r] = B

unbacktracked[B] = {D}

a = d → untried[B] = {l}

s = B

Online DFS (Example)



Step 6

s' = B s = B

untried[B] = {d, l}

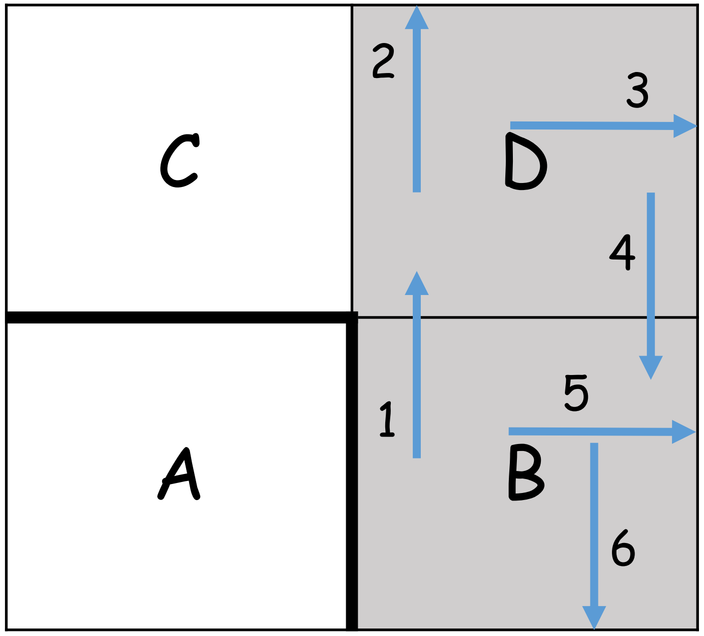
result[B, r] = B

unbacktracked[B] = {D}

a = d → untried[B] = {l}

s = B

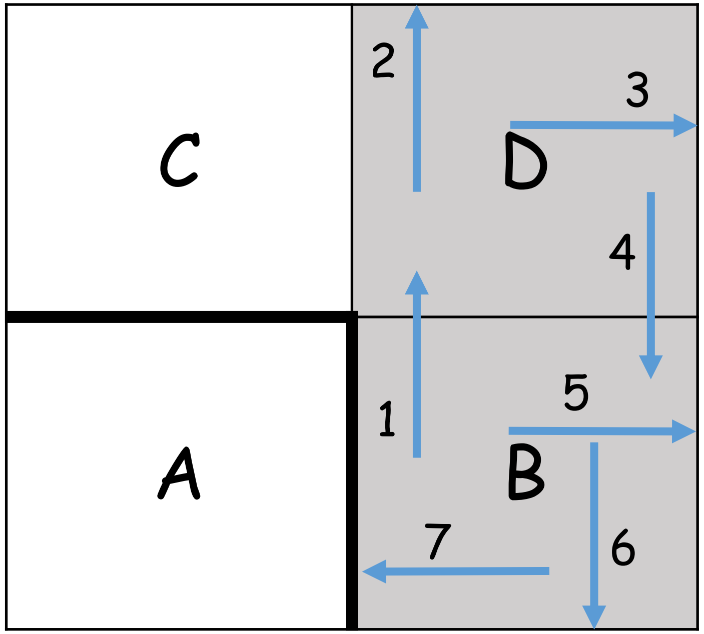
Online DFS (Example)



Step 7

s' = B s = B
untried[B] = {l}
result[B, d] = B
unbacktracked[B] = {D}
a = l → untried[B] = {}
s = B

Online DFS (Example)



Step 7

s' = B   s = B

untried[B] = {l}

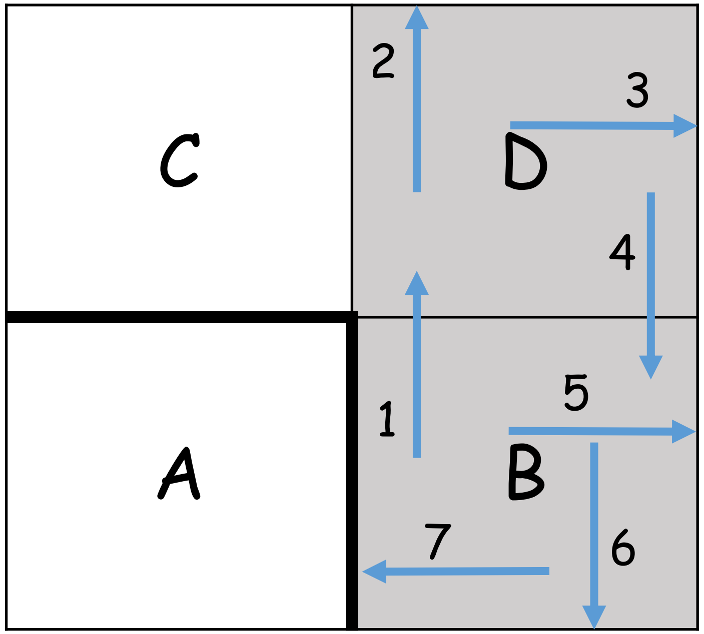
result[B, d] = B

unbacktracked[B] = {D}

a = l → untried[B] = {}

s = B

Online DFS (Example)



Step 8

s' = B   s = B

untried[B] = {}

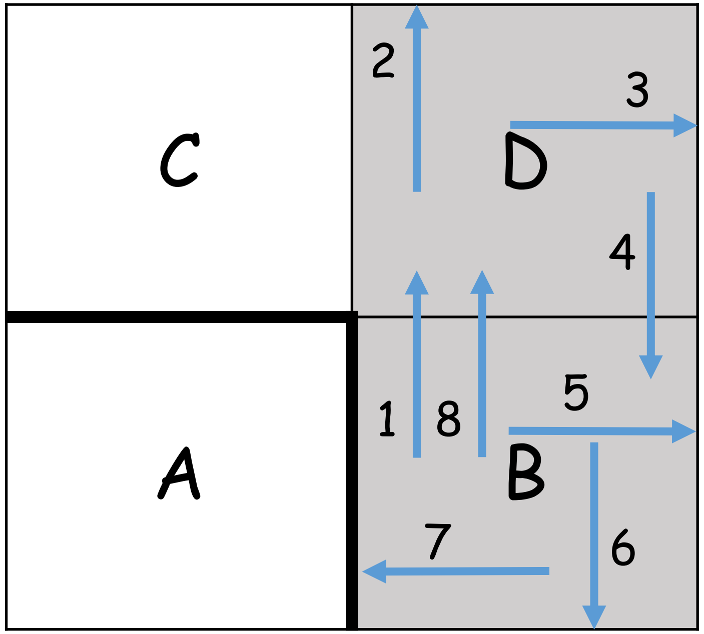
result[B, l] = B

unbacktracked[B] = {D}

a = u   → unbacktracked[B] = {}

s = B

Online DFS (Example)



Step 8

s' = B   s = B

untried[B] = {}

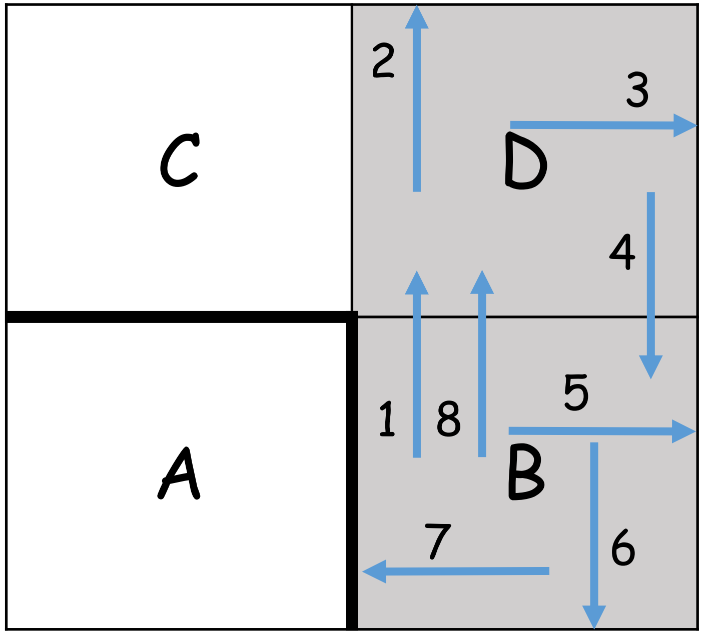
result[B, l] = B

unbacktracked[B] = {D}

a = u   → unbacktracked[B] = {}

s = B

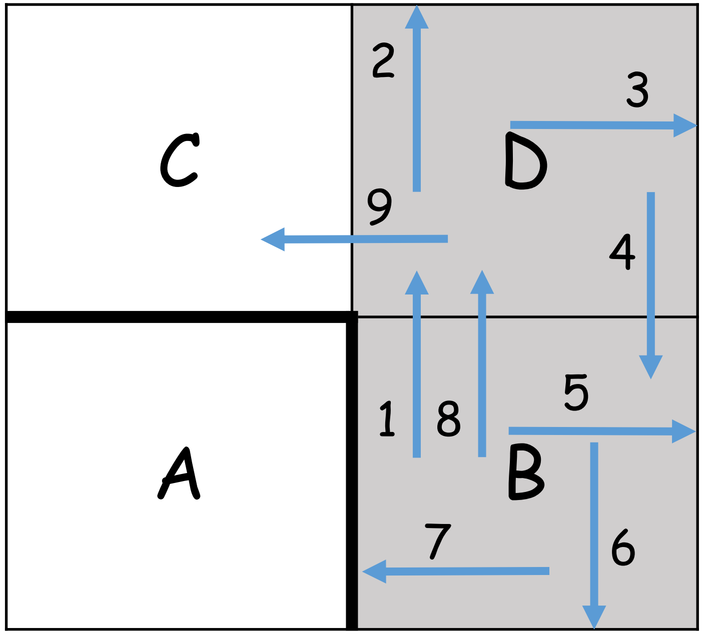
Online DFS (Example)



Step 9

s' = D   s = B
untried[D] = {l}
result[B, u] = D
unbacktracked[D] = {B}
a = l → untried[D] = {}
s = D

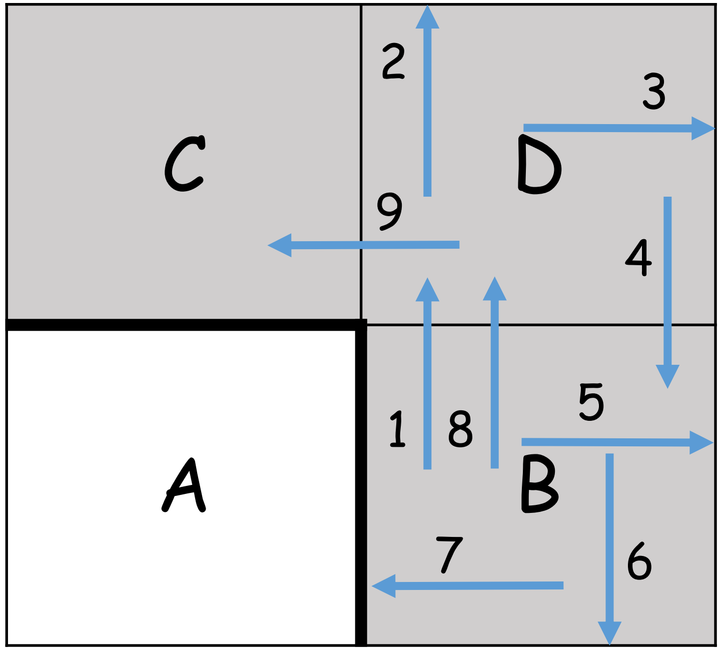
Online DFS (Example)



Step 9

s' = D s = B
untried[D] = {l}
result[B, u] = D
unbacktracked[D] = {B}
a = l → untried[D] = {}
s = D

Online DFS (Example)



Step 10

s' = C s = D

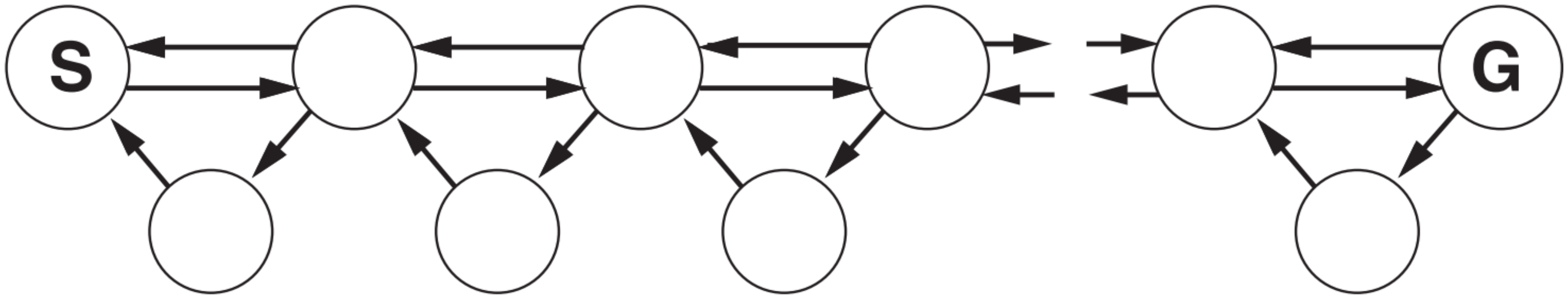
Goal-Test(C) = true → STOP

Online local search

- Hill-climbing
  - Has the property of locality in its node expansions
  - Because it keeps just one current state in memory, hill-climbing search is already an online search algorithm!
  - It leaves the agent sitting at local maxima with nowhere to go
    - Random restarts cannot be used, because the agent cannot transport itself to a new state
- Solution
  - Random walk instead of random restart
    - Randomly selecting one of available actions (preference to untried actions)
  - Adding Memory (Learning Real Time A*): more effective
    - To remember and update the costs of all visited nodes

Random walk

- A random walk simply selects at random one of the available actions from the current state
  - Preference can be given to actions that have not yet been tried
- A random walk will eventually find a goal or complete its exploration, provided that the space is finite.



Learning real-time A* (LRTA*)

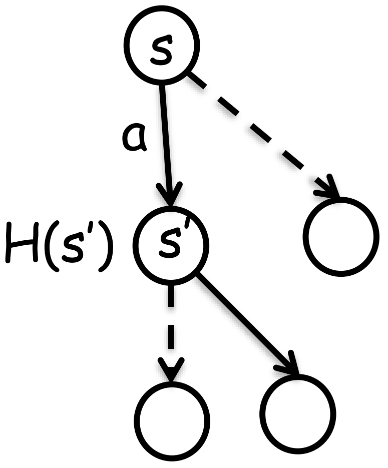
- Augmenting hill climbing with memory rather than randomness turns out to be a more effective approach
  - Store a "current best estimate" H(s) of the cost to reach the goal from each state that has been visited
  - Initially H(s) is a heuristic estimate h(s)
  - H(s) is updated by experience (More accurate estimates are acquired using local updating rules)

$$H(s) \leftarrow \min_{a \in ACTIONS(s)} H(s') + c(s, a, s')$$

  - Assumption: Untried actions in a state s lead to the goal with the least possible cost h(s)
    - Encouraging to explore new (possibly promising) paths

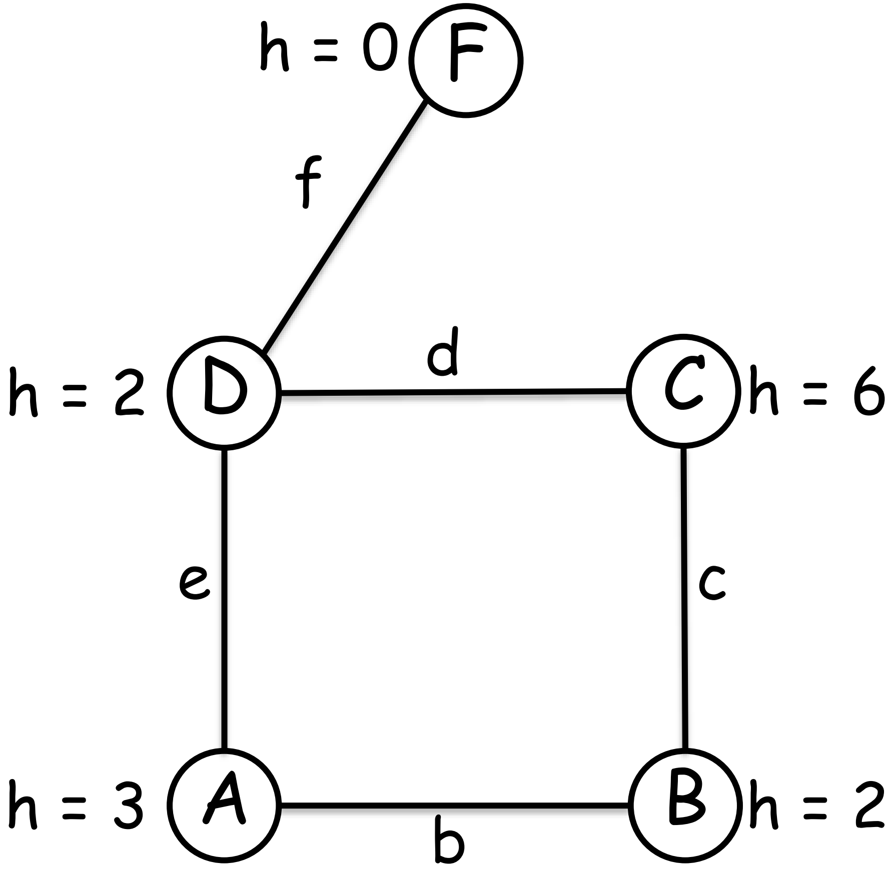
Learning real-time A* (LRTA*)

function LRTA*-AGENT(s') **returns** an action
    inputs: s', a percept that identifies the current state
    persistent: *result*, a table, indexed by state and action, initially empty
        H, a table of cost estimates indexed by state, initially empty
        s, a, the previous state and action, initially null
    if GOAL-TEST(s') **then return** stop
    if s' is a new state (not in H) **then** H[s'] ← h(s')
    if s is not null
        result[s, a] ←s'
        H[s] ← $\min_{b \in ACTIONS(s)} \boldsymbol{LRTA^{*}\_COST(s, b, result[s, b], H)}$
    a ← an action b in ACTIONS(s') that minimizes LRTA*-COST(s', b, *result*[s', b], H)
    s ← s'
    return a

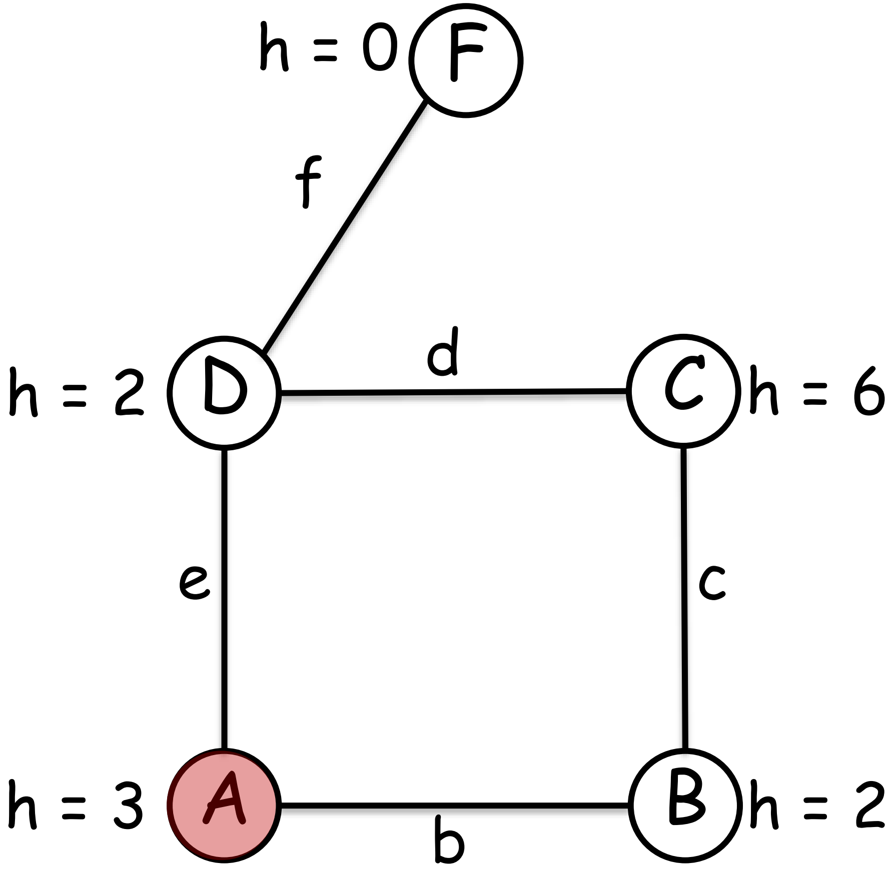


function LRTA*_COST(s, a, s', H) **returns** a cost estimate
    if s' is undefined **then return** h(s)
    else return c(s, a, s') + H[s']

LRTA* (Example)



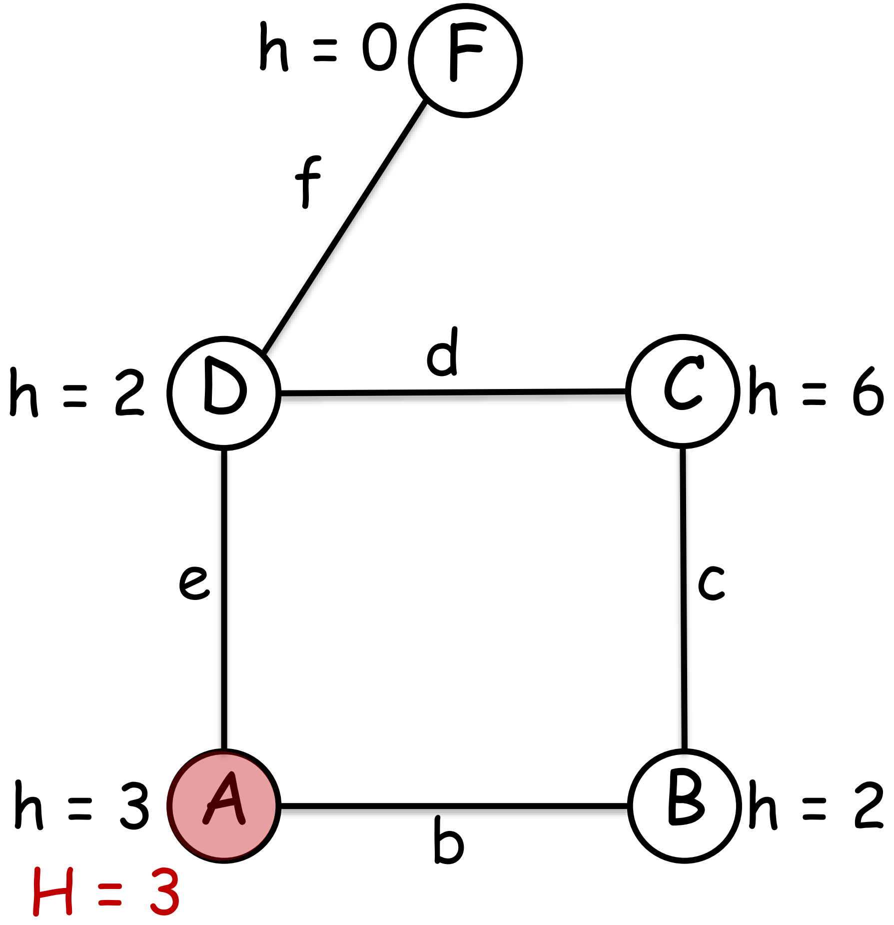
LRTA* (Example)



s' = A

s = null

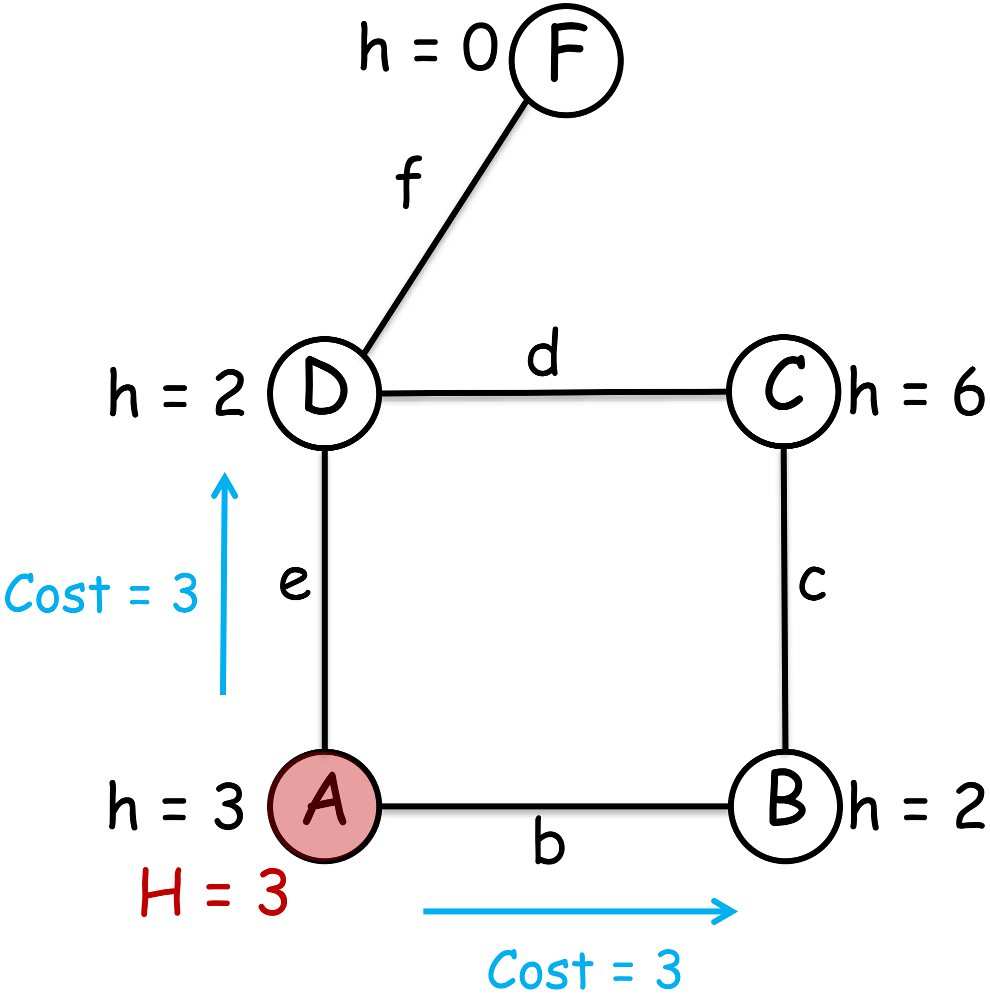
LRTA* (Example)



s' = A   s = null

H(A) = h(A) = 3

LRTA* (Example)



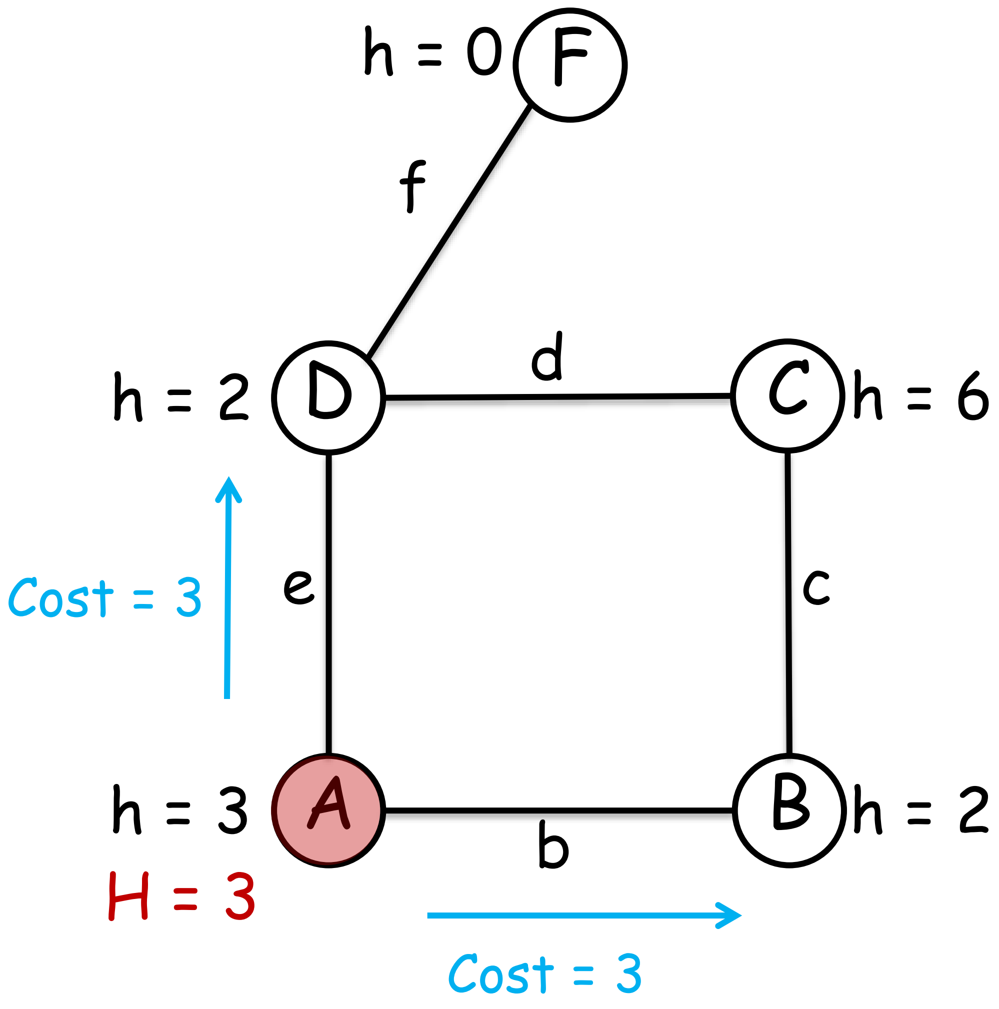
s' = A

s = null

H(A) = h(A) = 3

a = b

LRTA* (Example)



s' = A

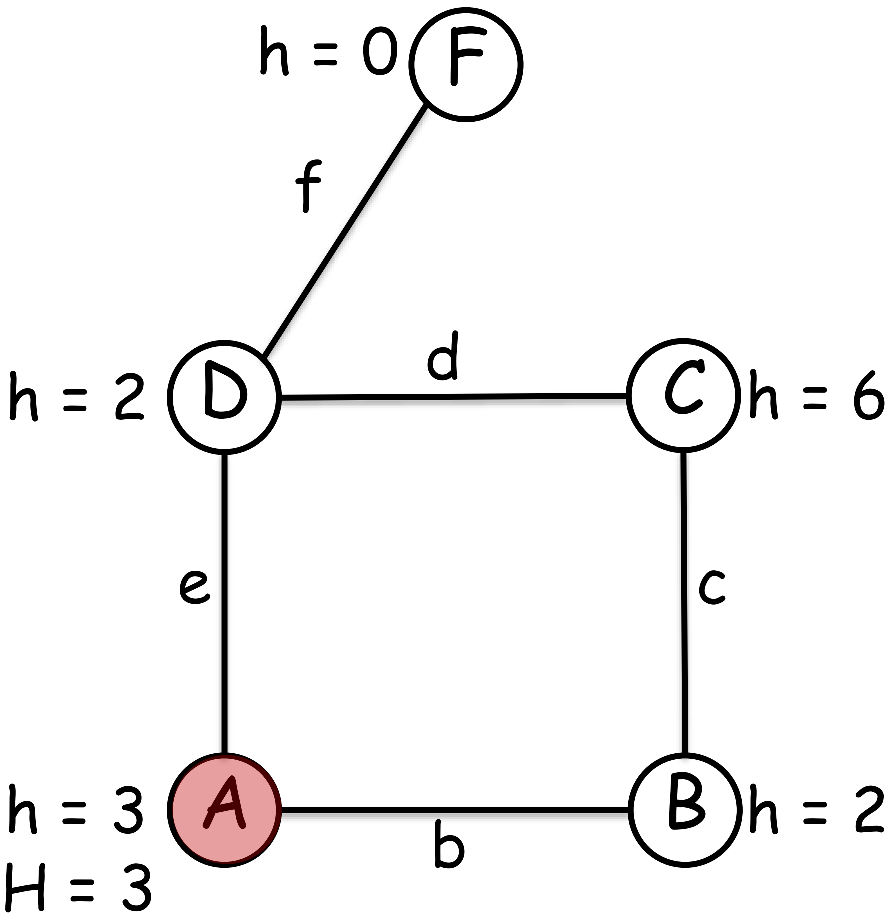
s = null

H(A) = h(A) = 3

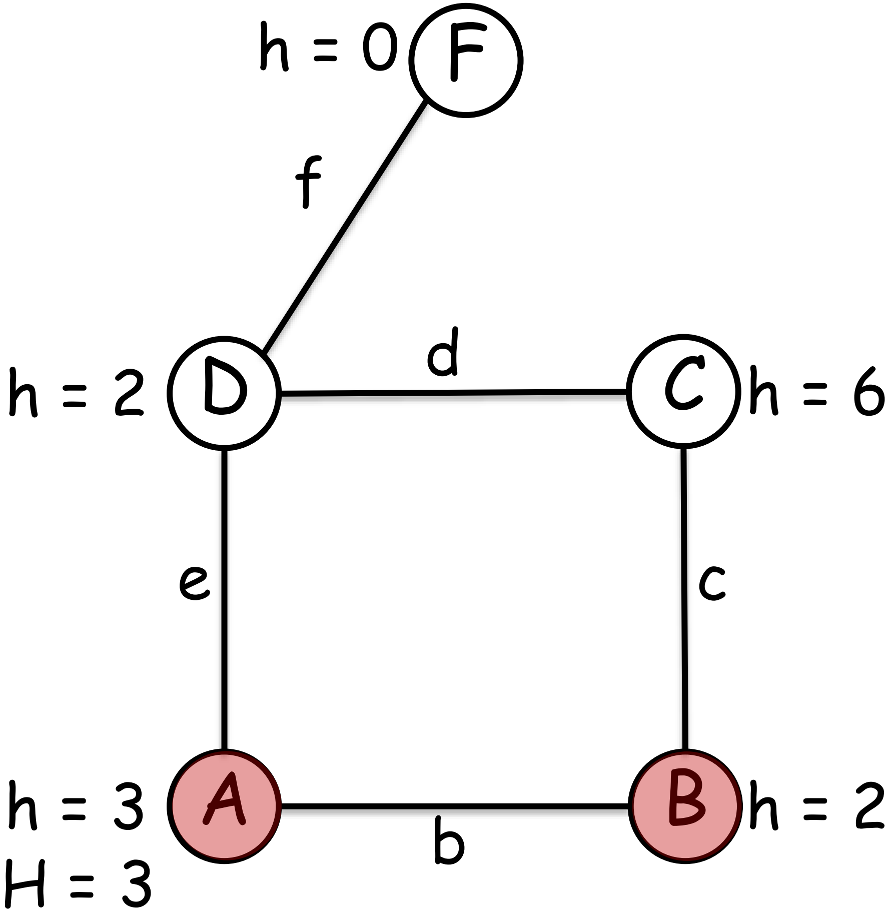
a = b

s = A

LRTA* (Example)



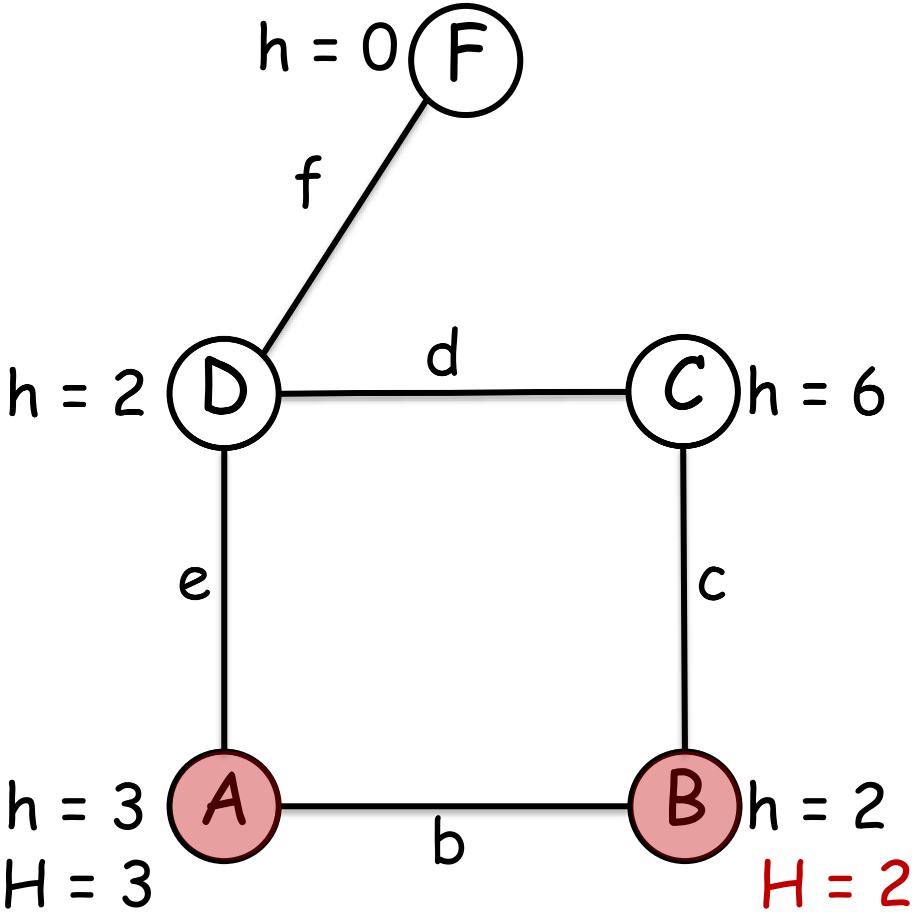
LRTA* (Example)



s' = B

s = A

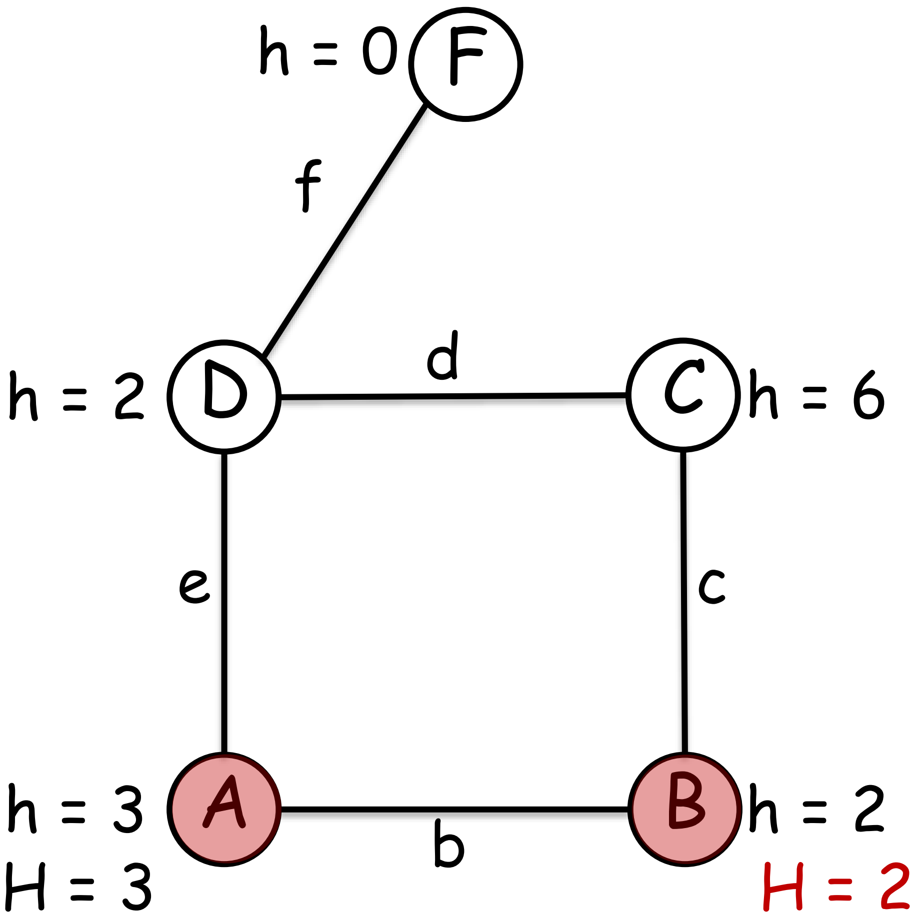
LRTA* (Example)



s' = B   s = A

H(B) = h(B) = 2

LRTA* (Example)



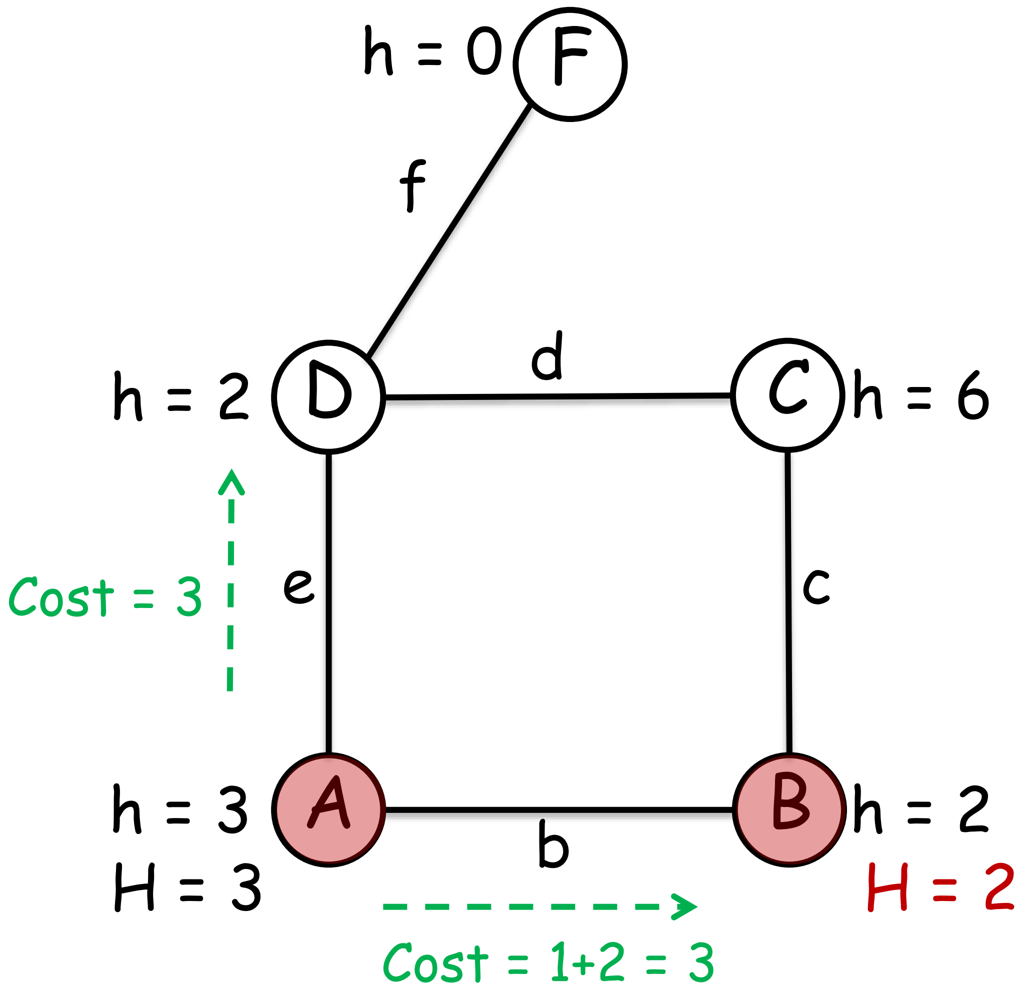
s' = B

s = A

H(B) = h(B) = 2

result(A, b) = B

LRTA* (Example)

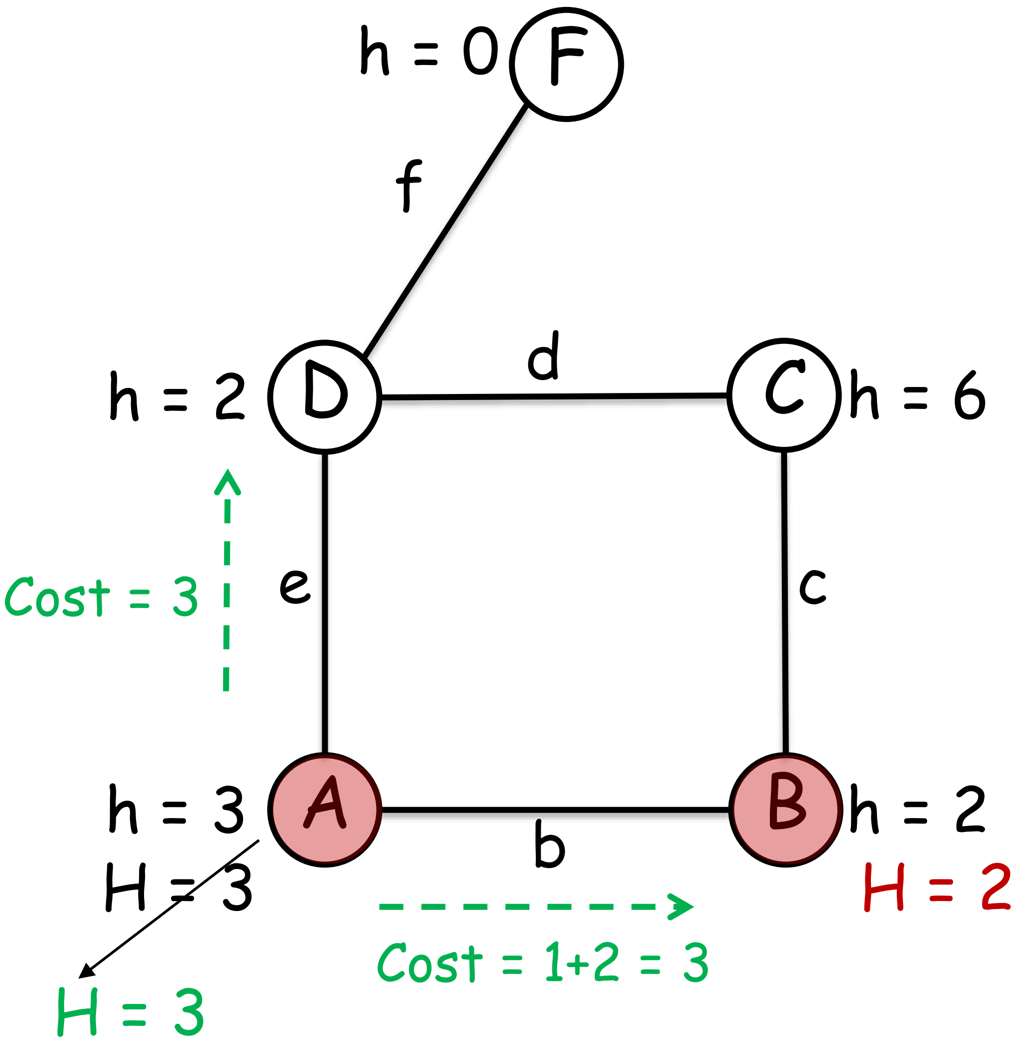


s' = B   s = A

H(B) = h(B) = 2

result(A, b) = B

LRTA* (Example)



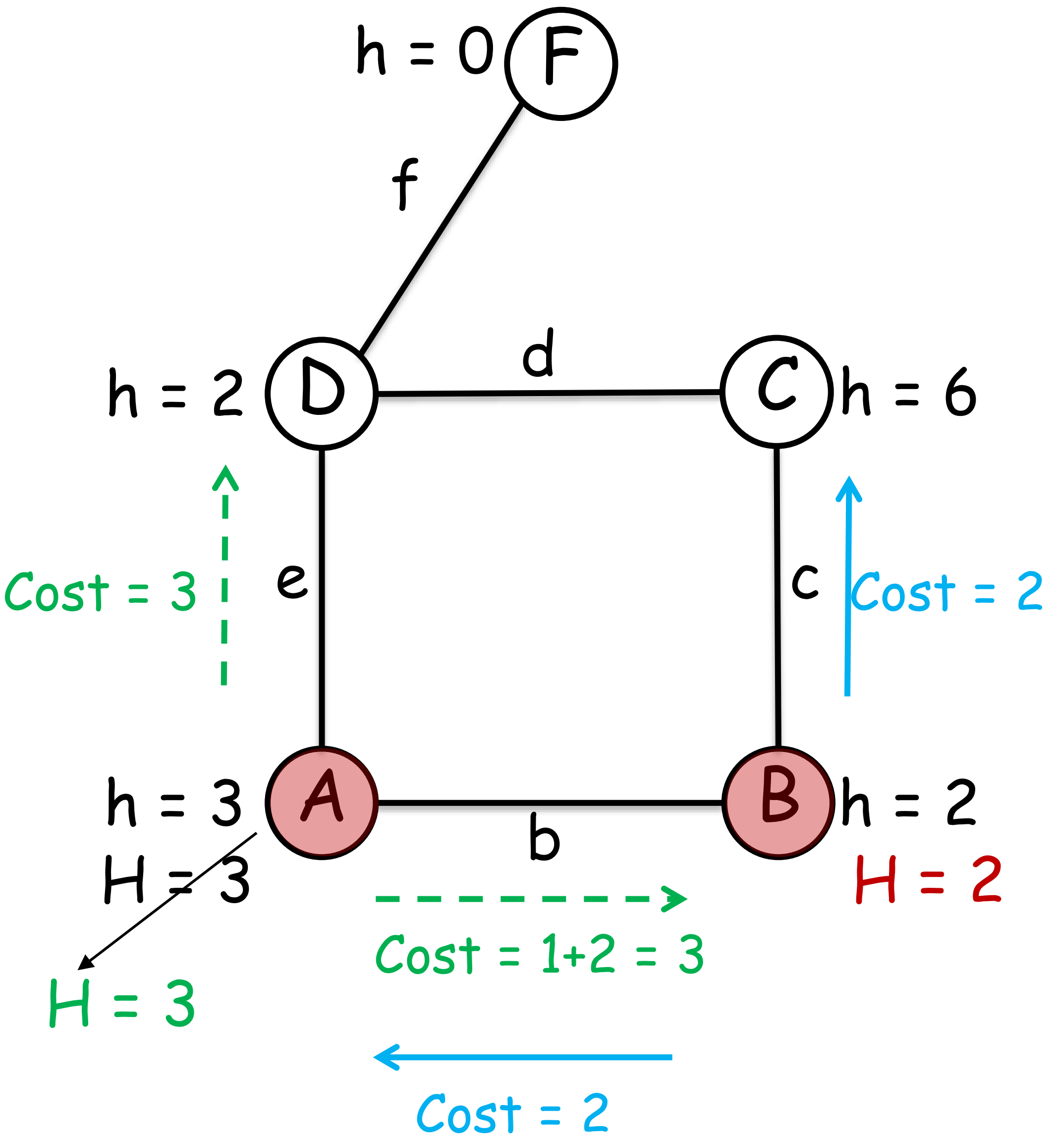
s' = B s = A

H(B) = h(B) = 2

result(A, b) = B

H(A) = 3

LRTA* (Example)



s' = B   s = A

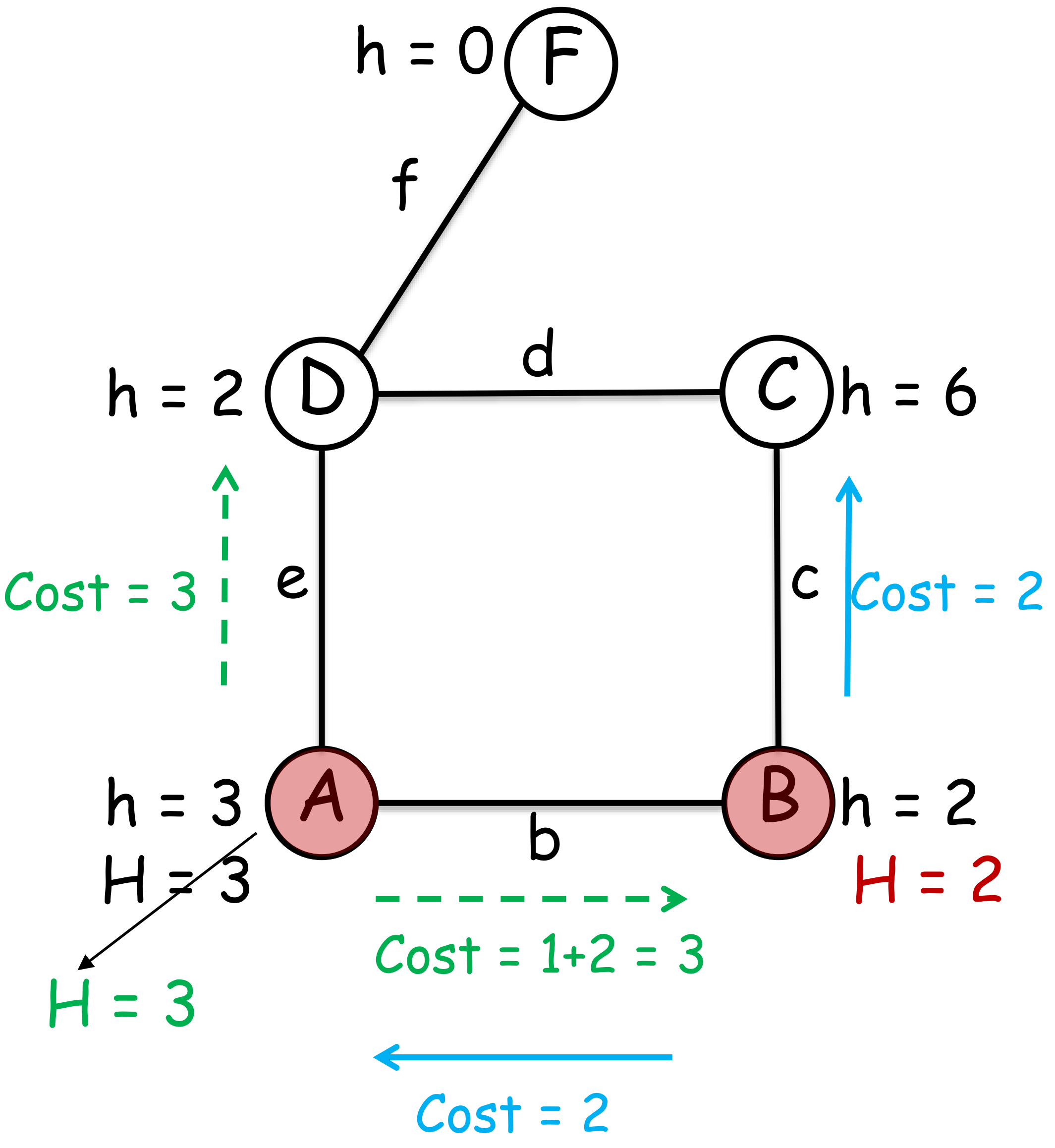
H(B) = h(B) = 2

result(A, b) = B

H(A) = 3

a = c

LRTA* (Example)



$s' = B$ $s = A$

$H(B) = h(B) = 2$

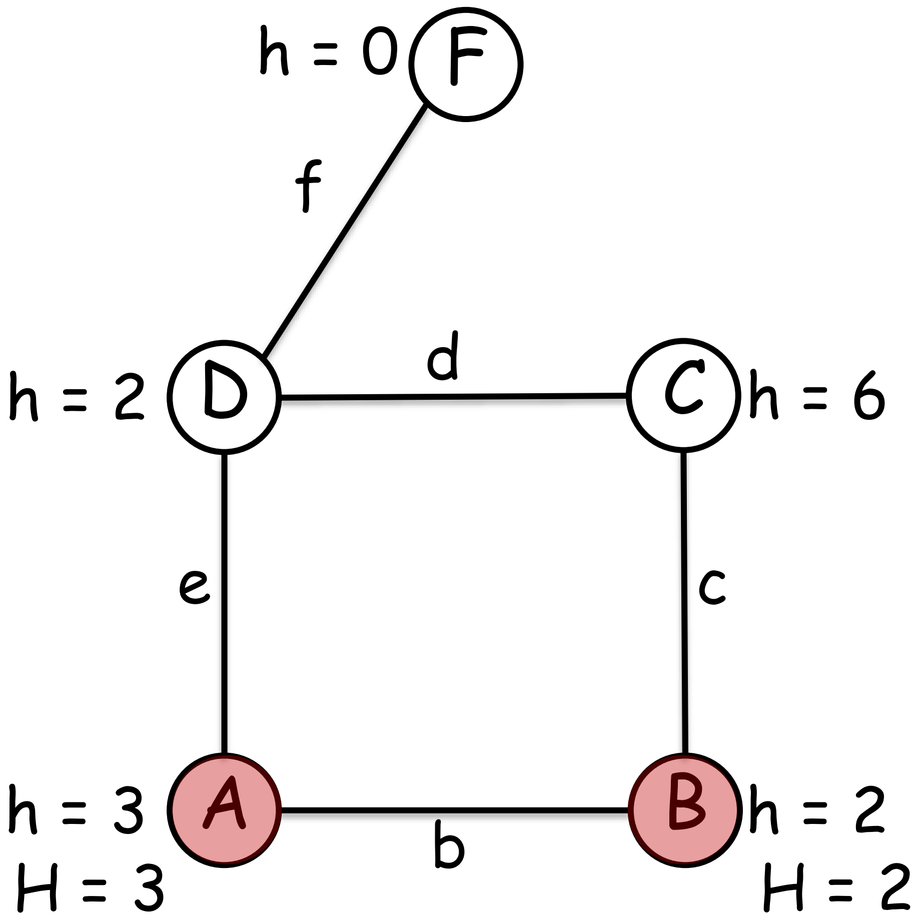
$result(A, b) = B$

$H(A) = 3$

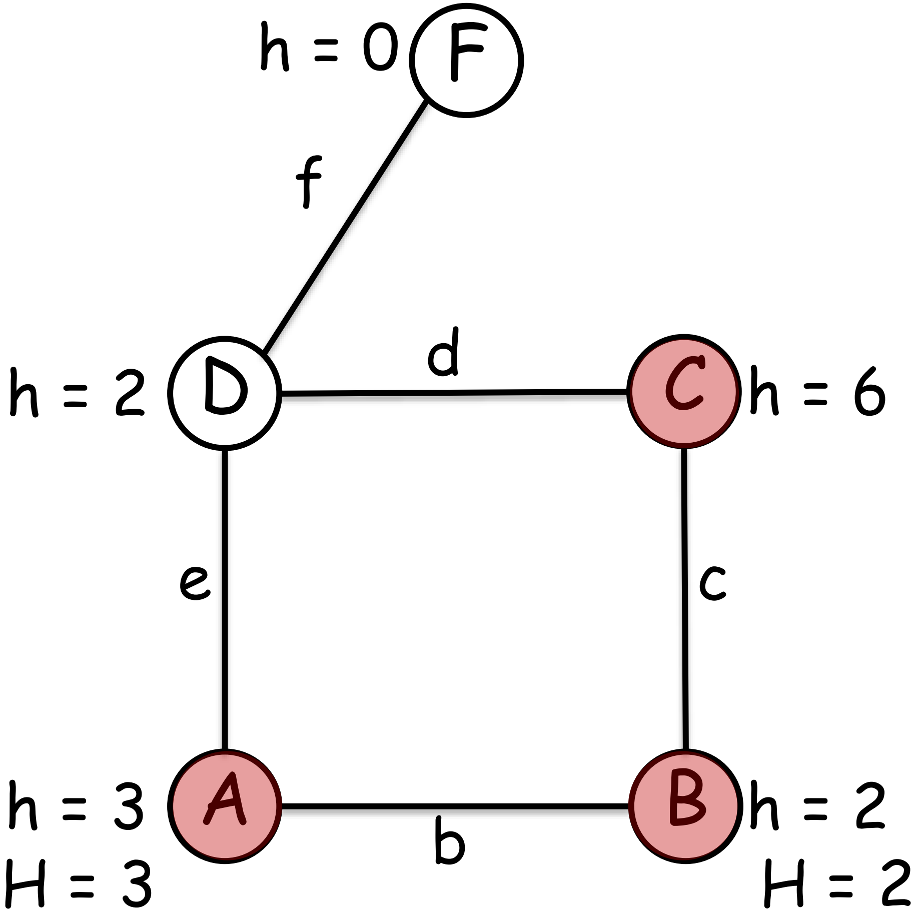
$a = c$

$s = B$

LRTA* (Example)



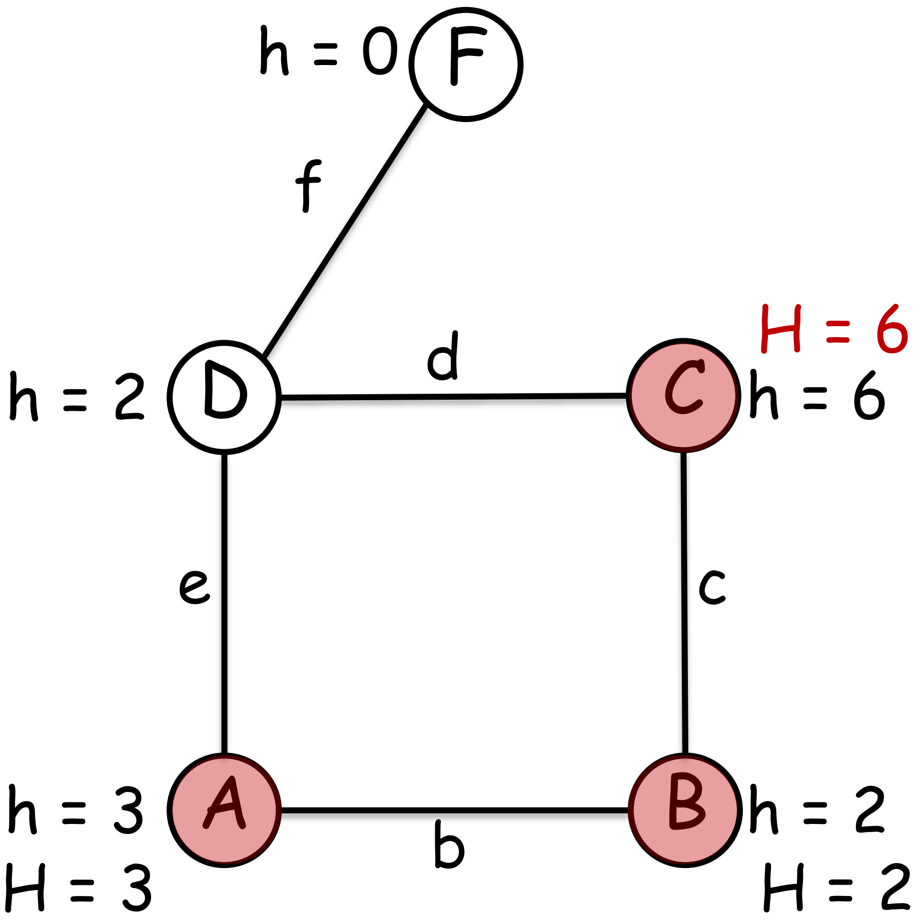
LRTA* (Example)



s' = C

s = B

LRTA* (Example)

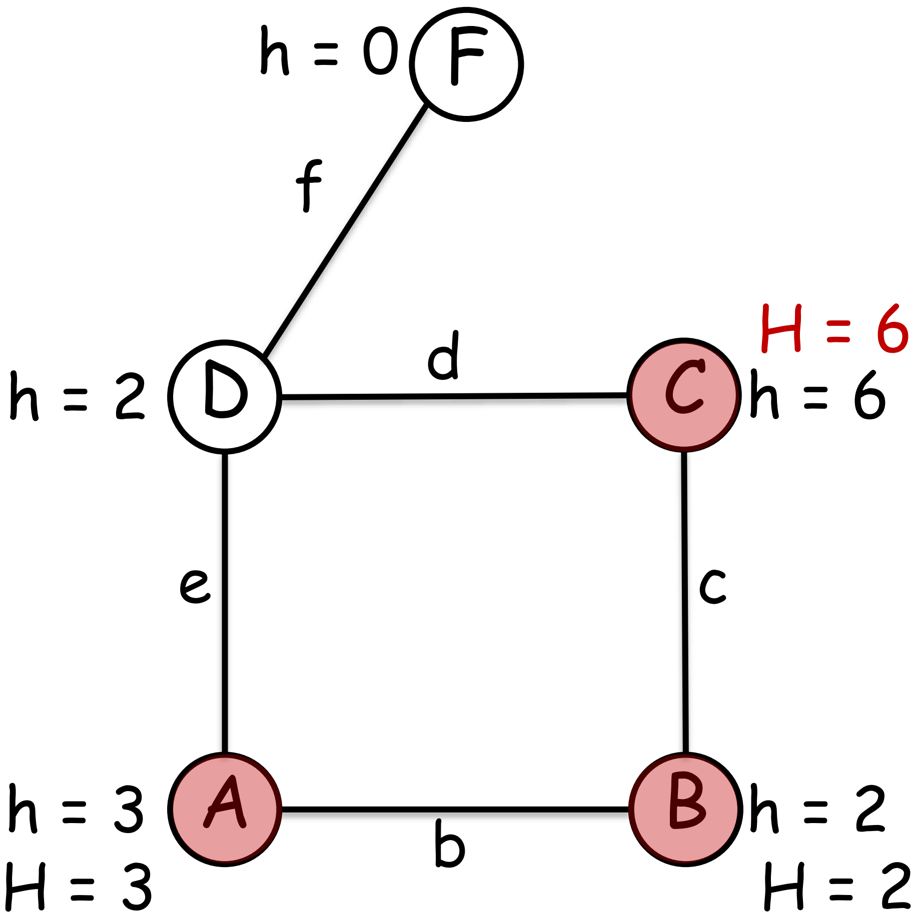


s' = C

s = B

H(C) = h(C) = 6

LRTA* (Example)

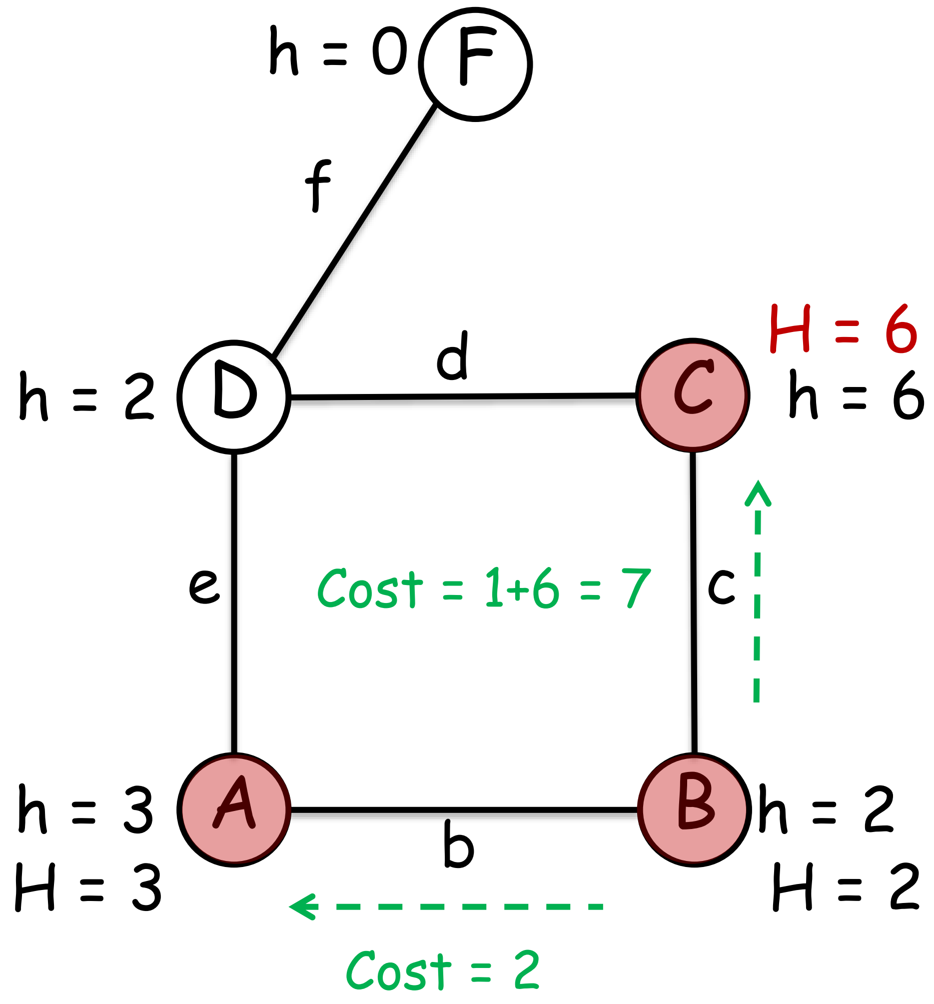


s' = C s = B

H(C) = h(C) = 6

result(B, c) = C

LRTA* (Example)

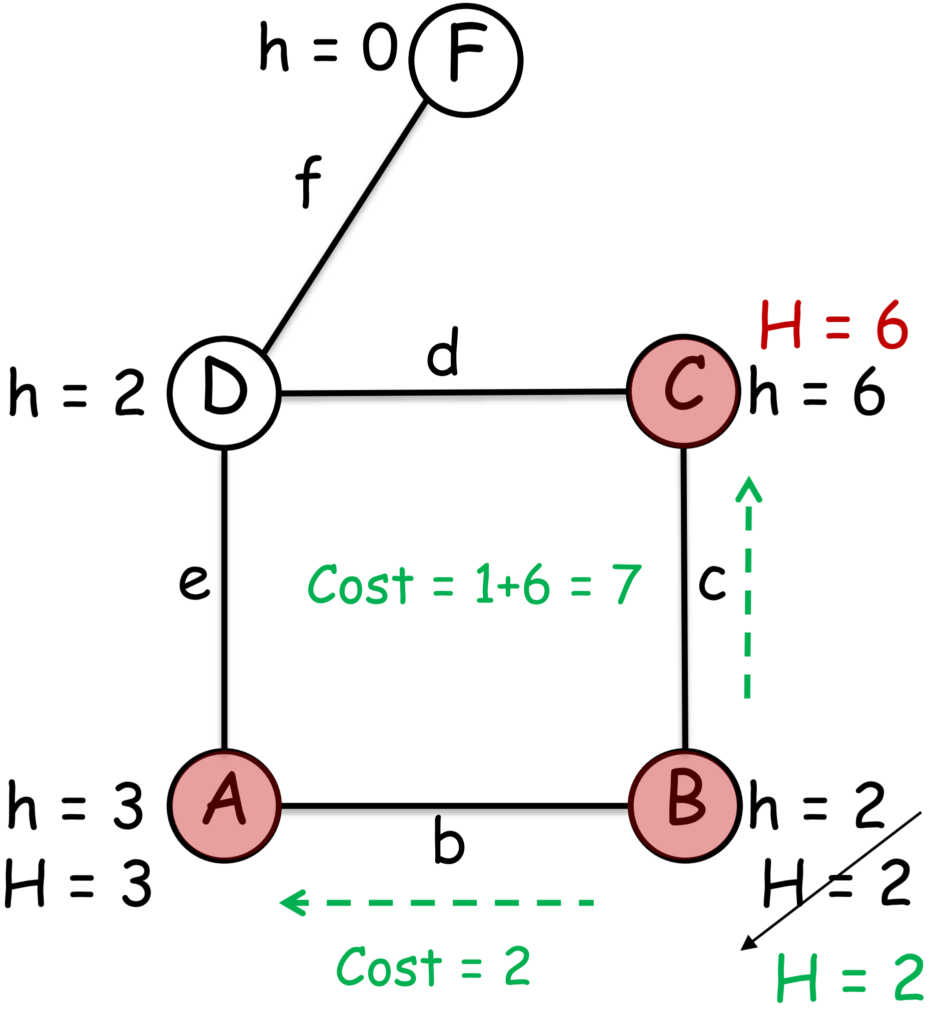


s' = C   s = B

H(C) = h(C) = 6

result(B, c) = C

LRTA* (Example)



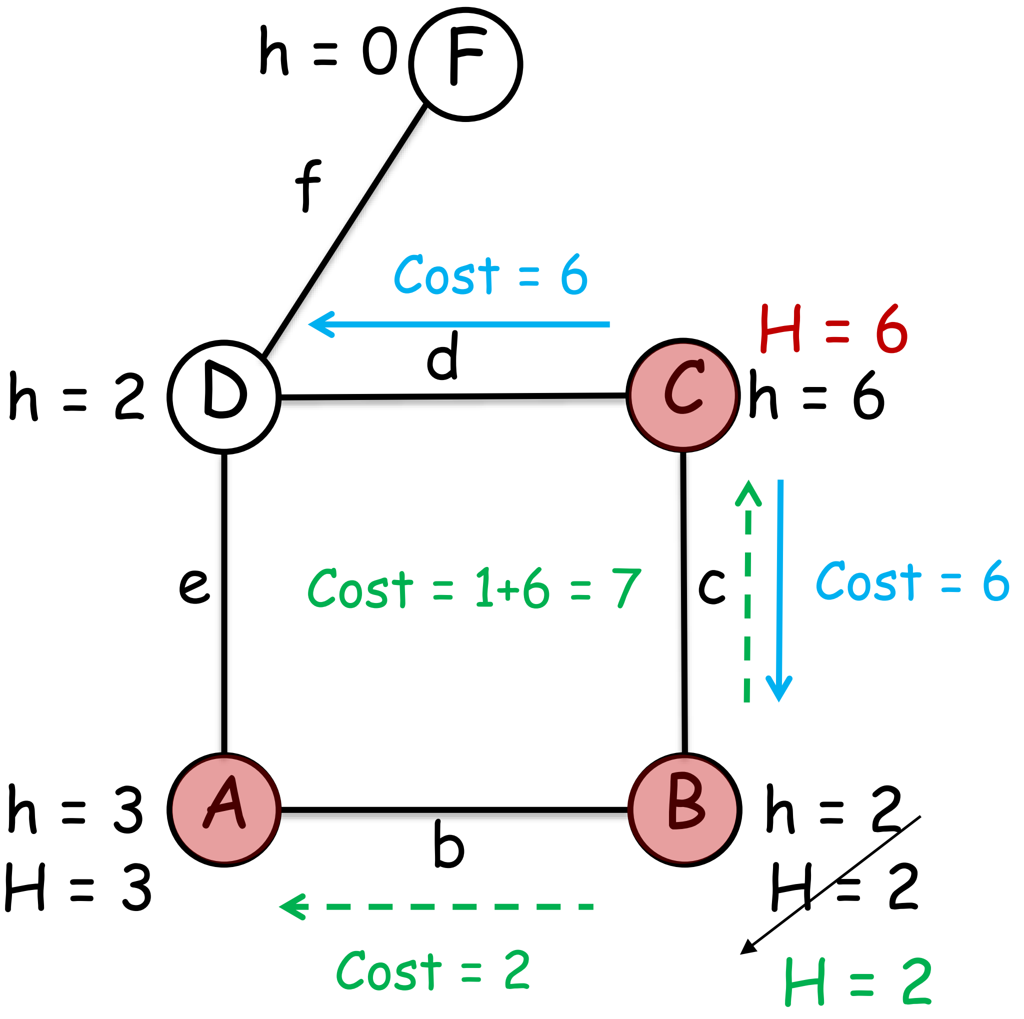
s' = C s = B

H(C) = h(C) = 6

result(B, c) = C

H(B) = 2

LRTA* (Example)



s' = C

s = B

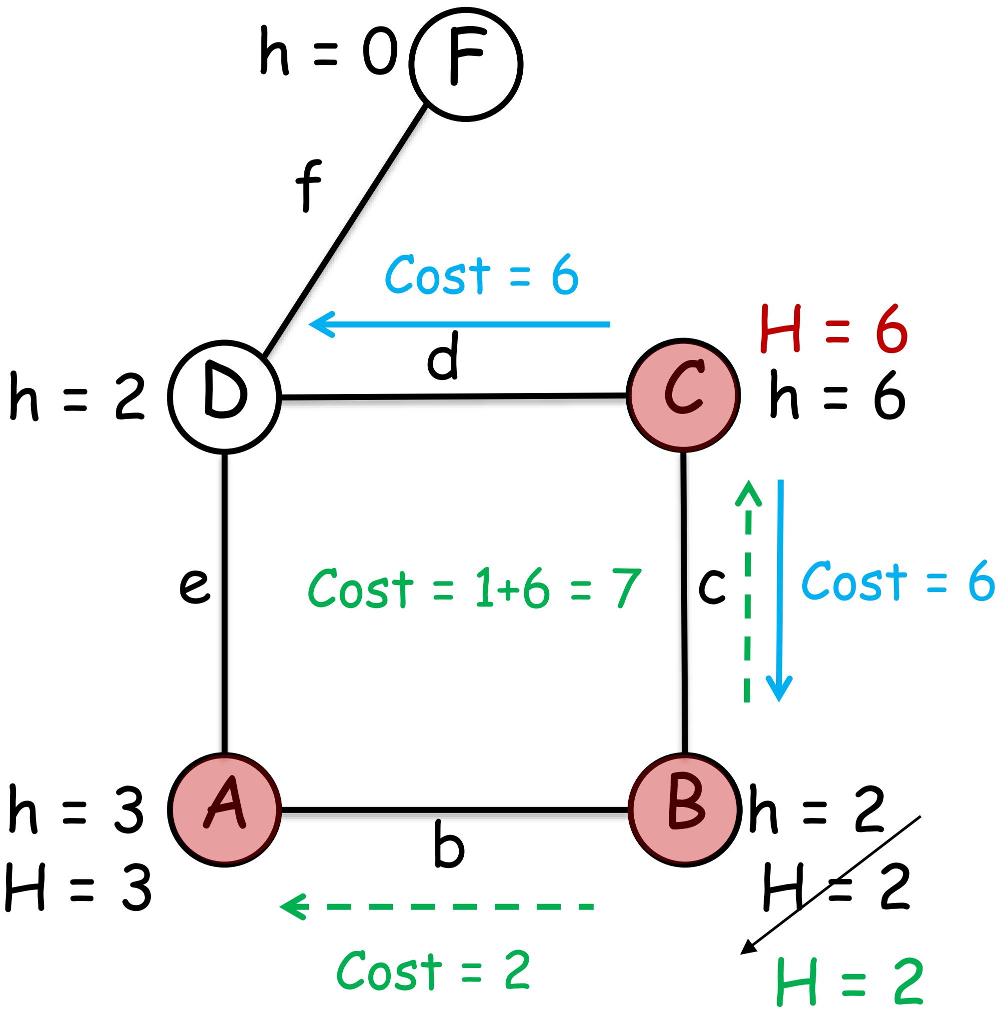
H(C) = h(C) = 6

result(B, c) = C

H(B) = 2

a = c

LRTA* (Example)



$s' = C$ $\quad s = B$

$H(C) = h(C) = 6$

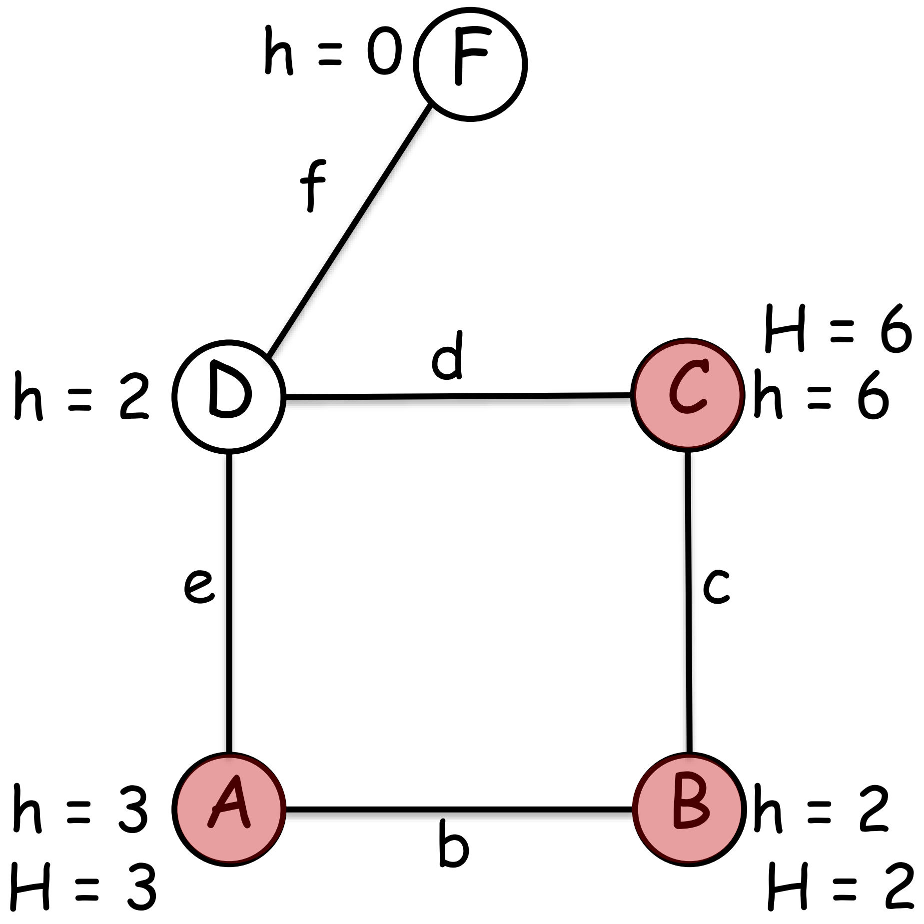
$\text{result}(B, c) = C$

$H(B) = 2$

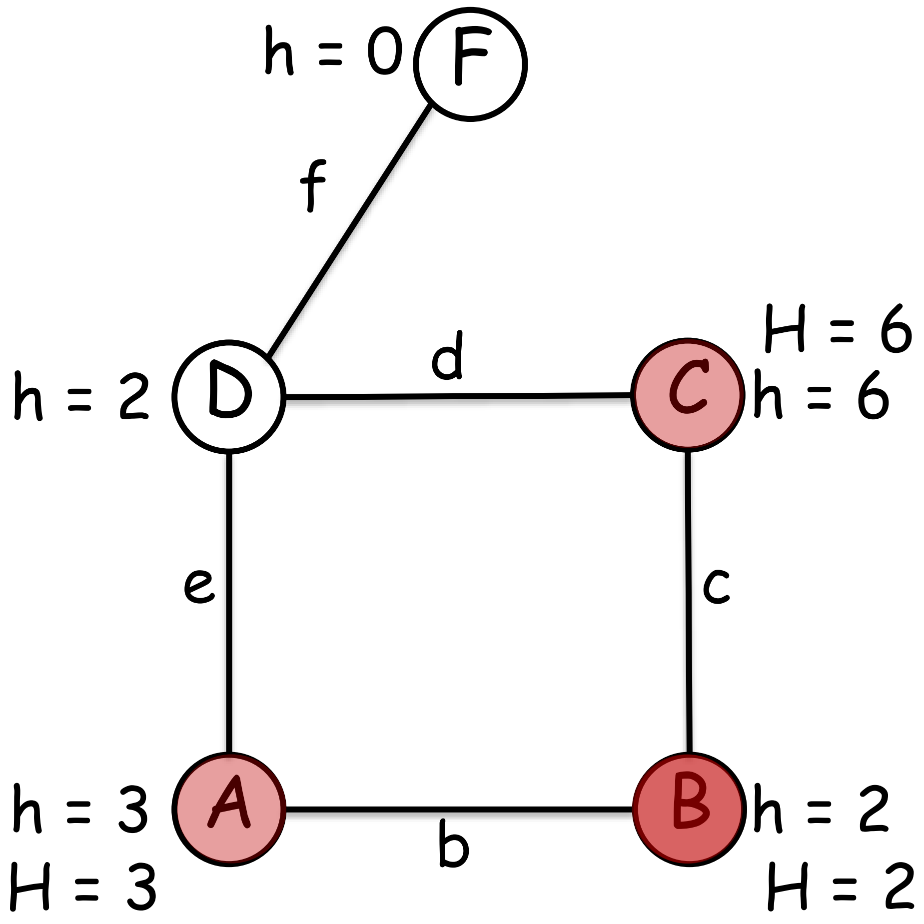
$a = c$

$s = C$

LRTA* (Example)



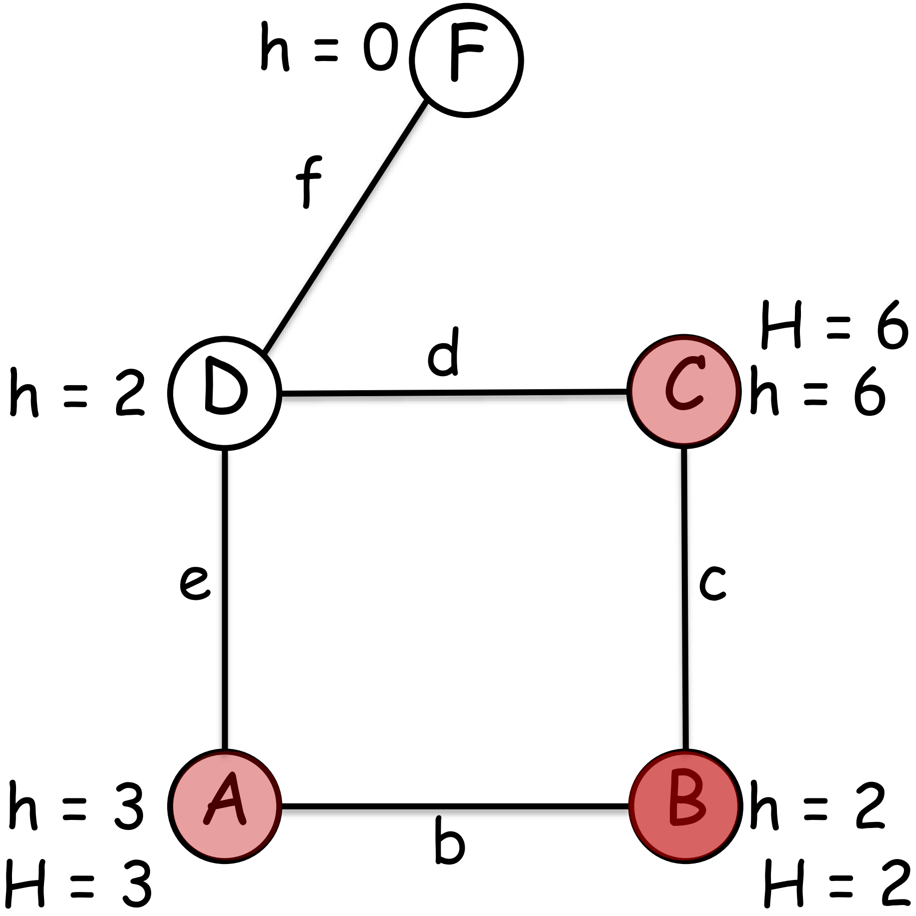
LRTA* (Example)



s' = B

s = C

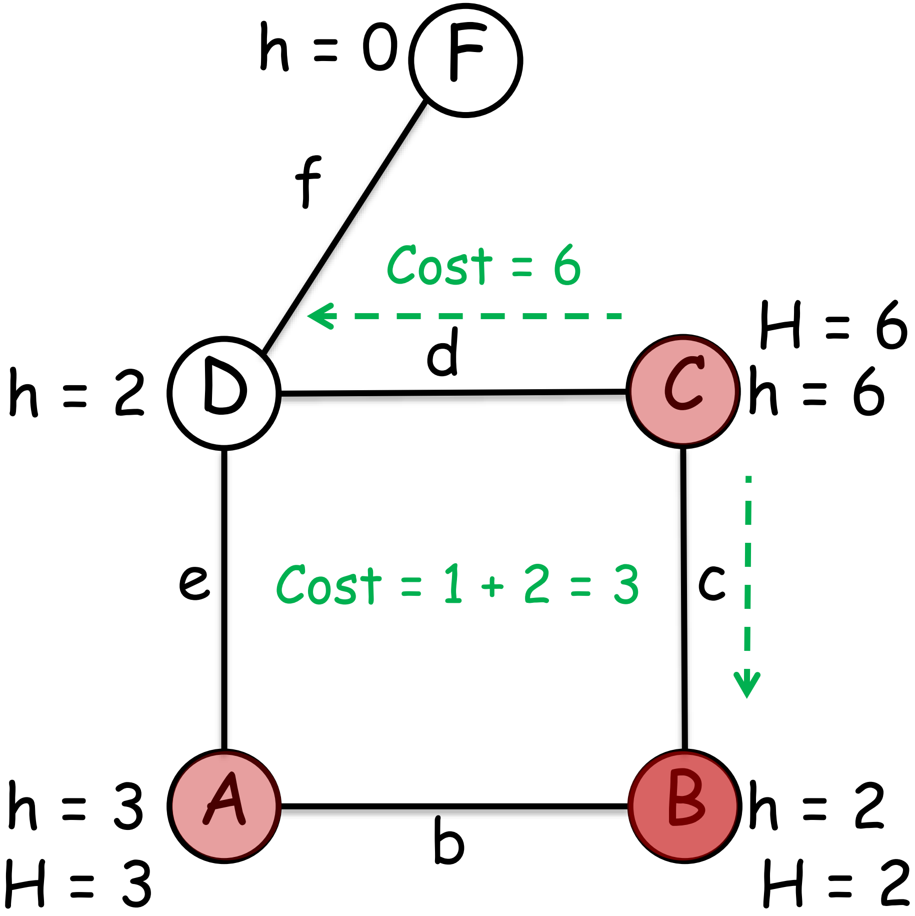
LRTA* (Example)



s' = B   s = C

result(C, c) = B

LRTA* (Example)

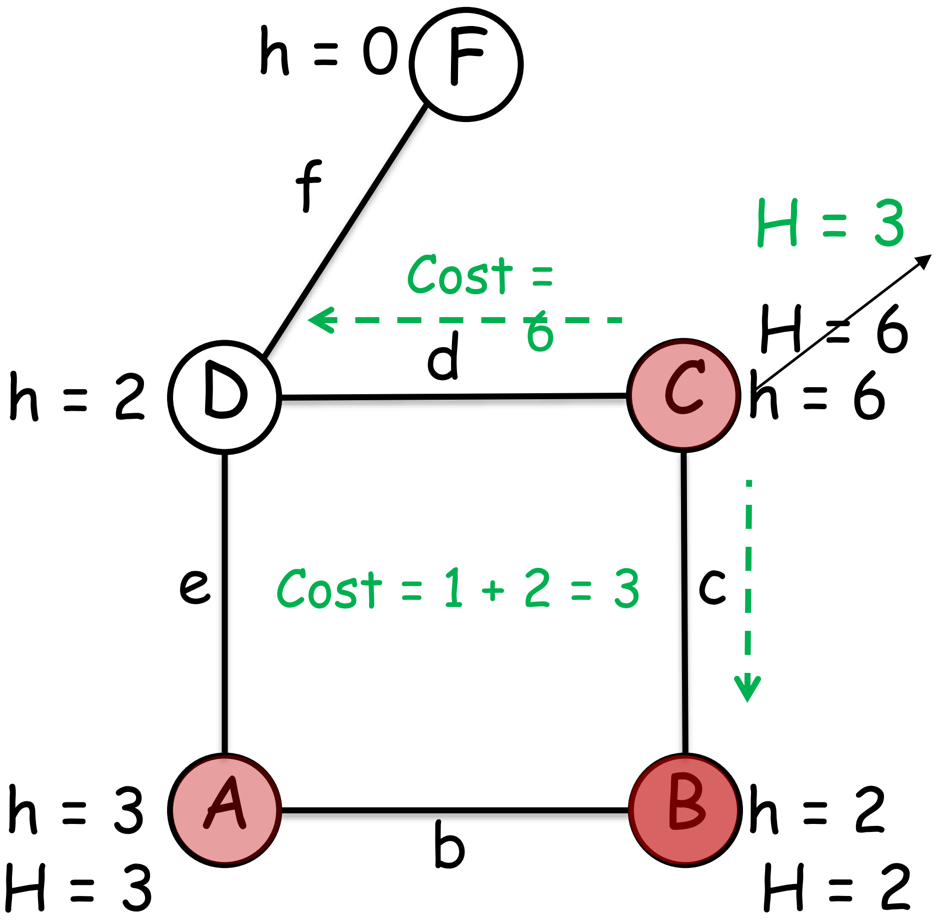


s' = B

s = C

result(C, c) = B

LRTA* (Example)



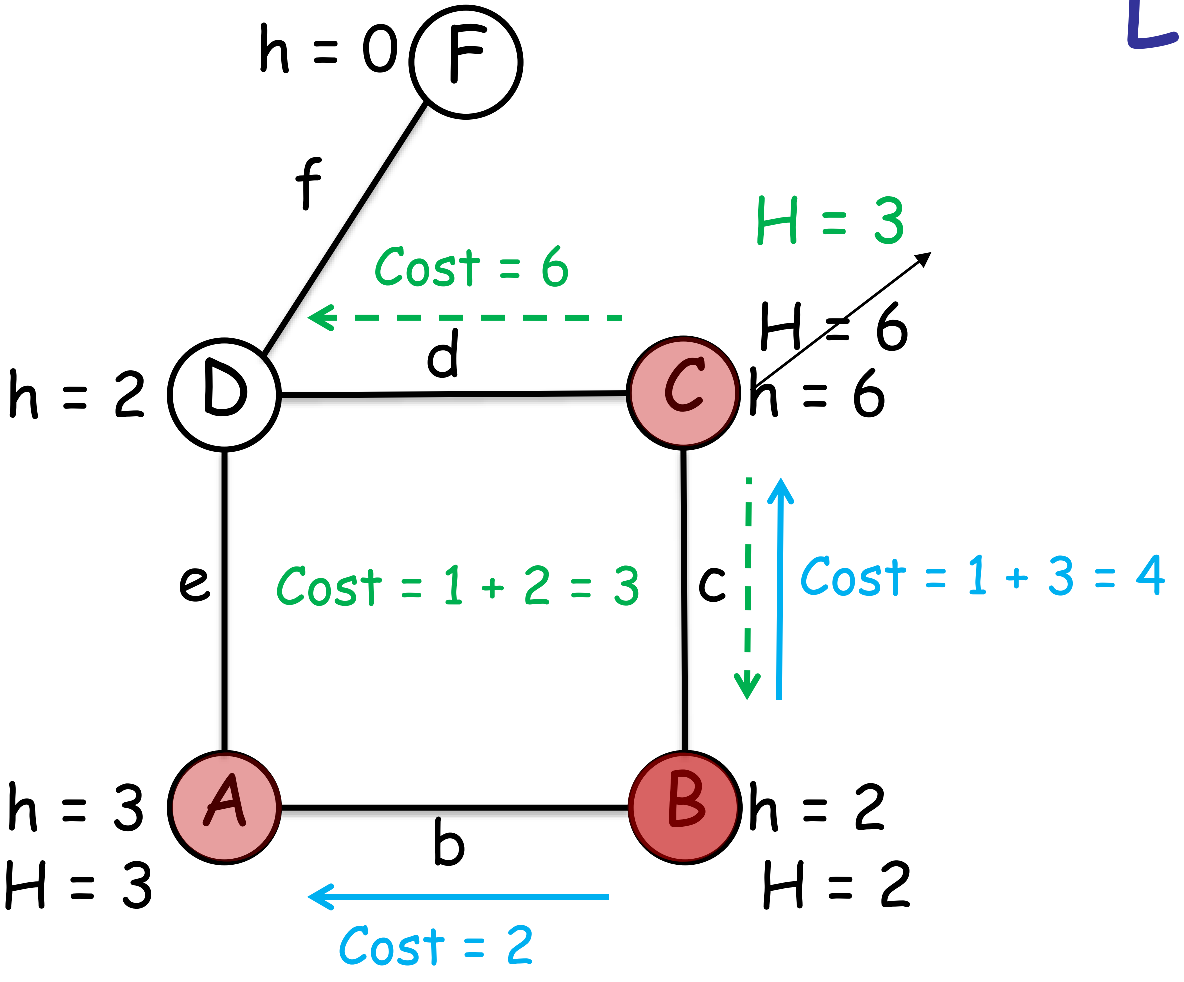
s' = B

s = C

result(C, c) = B

H(C) = 3

LRTA* (Example)



s' = B

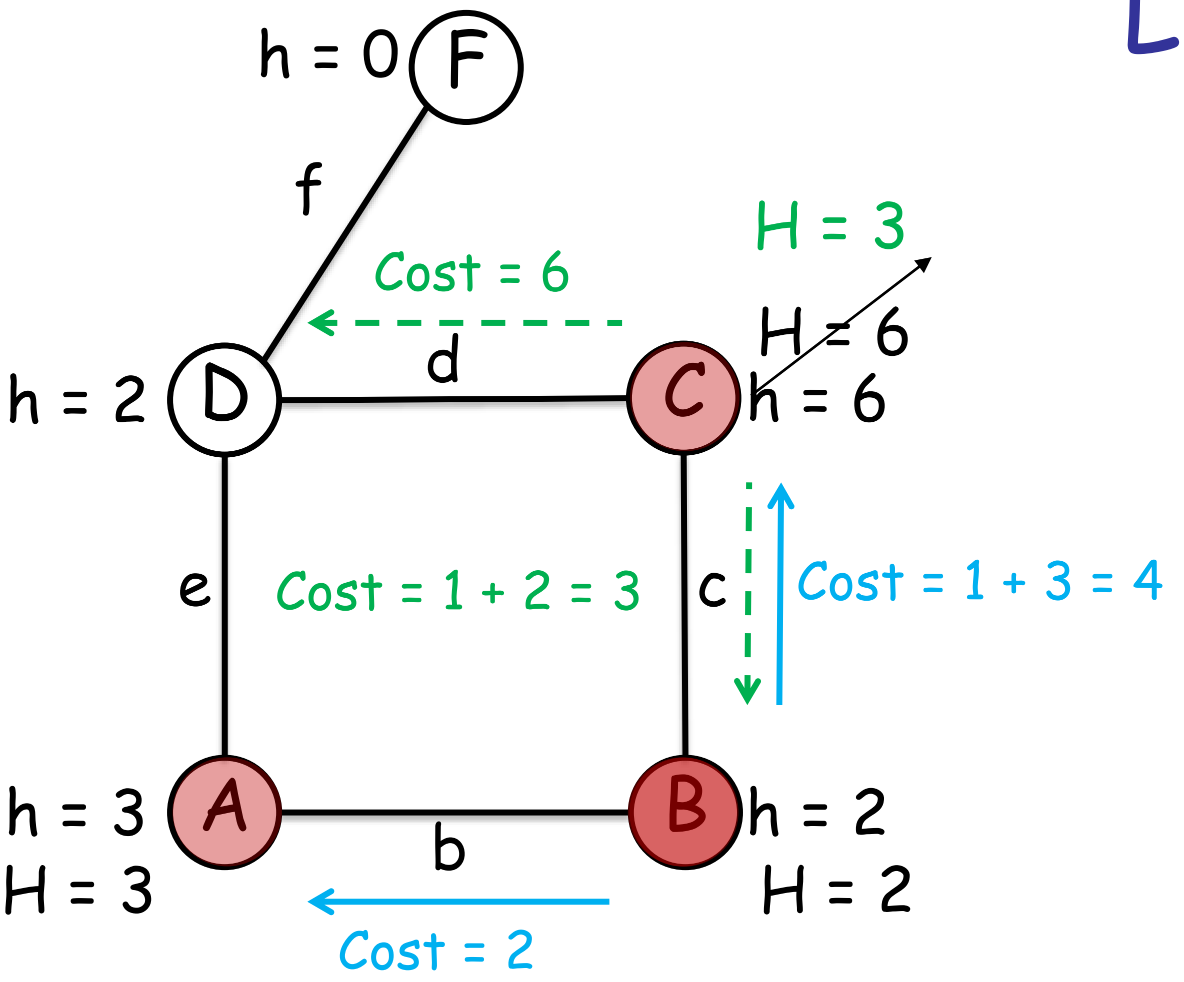
s = C

result(C, c) = B

H(C) = 3

a = b

LRTA* (Example)



s' = B

s = C

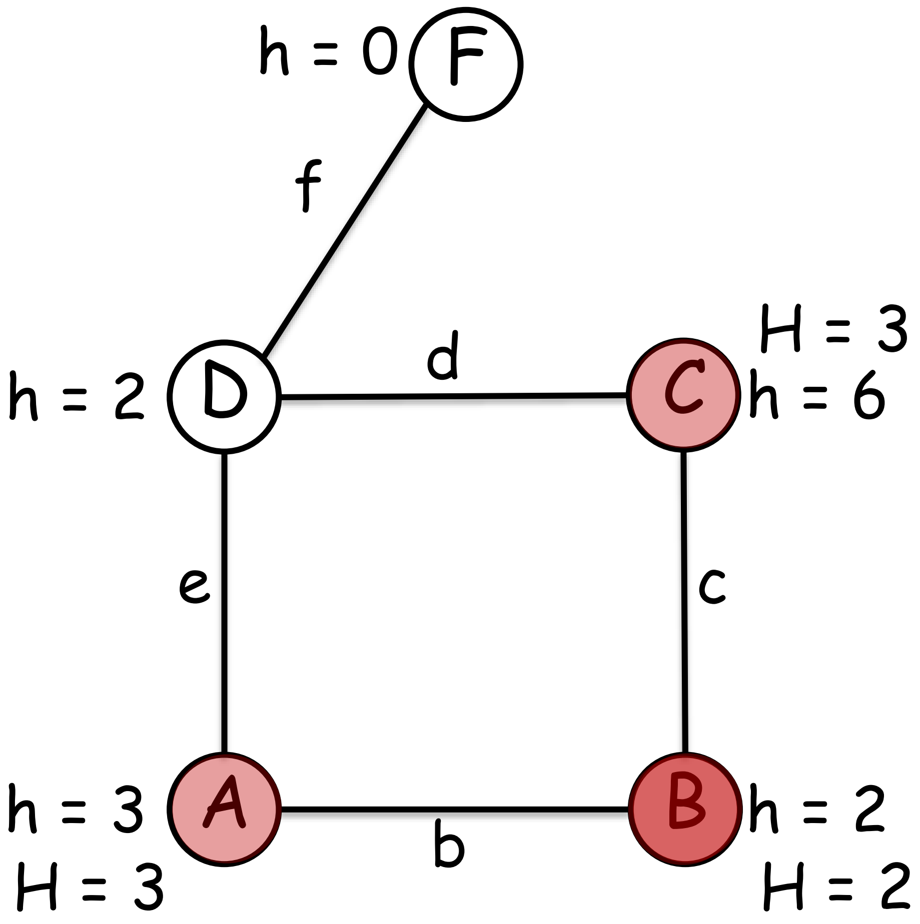
result(C, c) = B

H(C) = 3

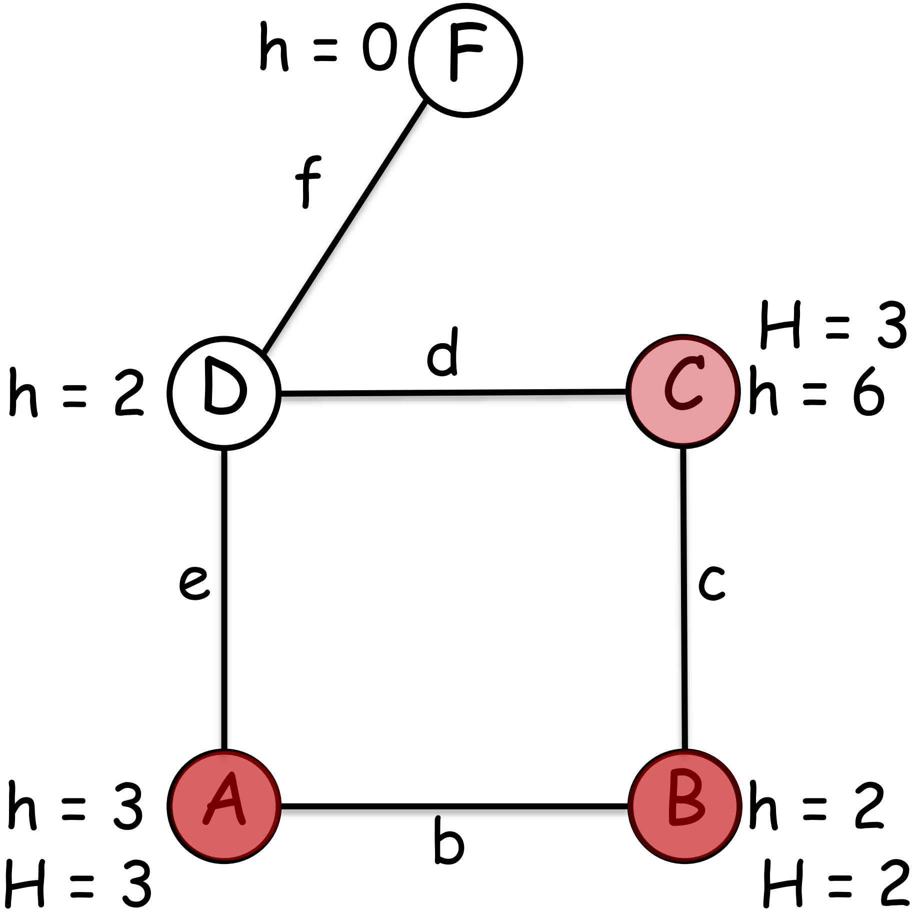
a = b

s = B

LRTA* (Example)

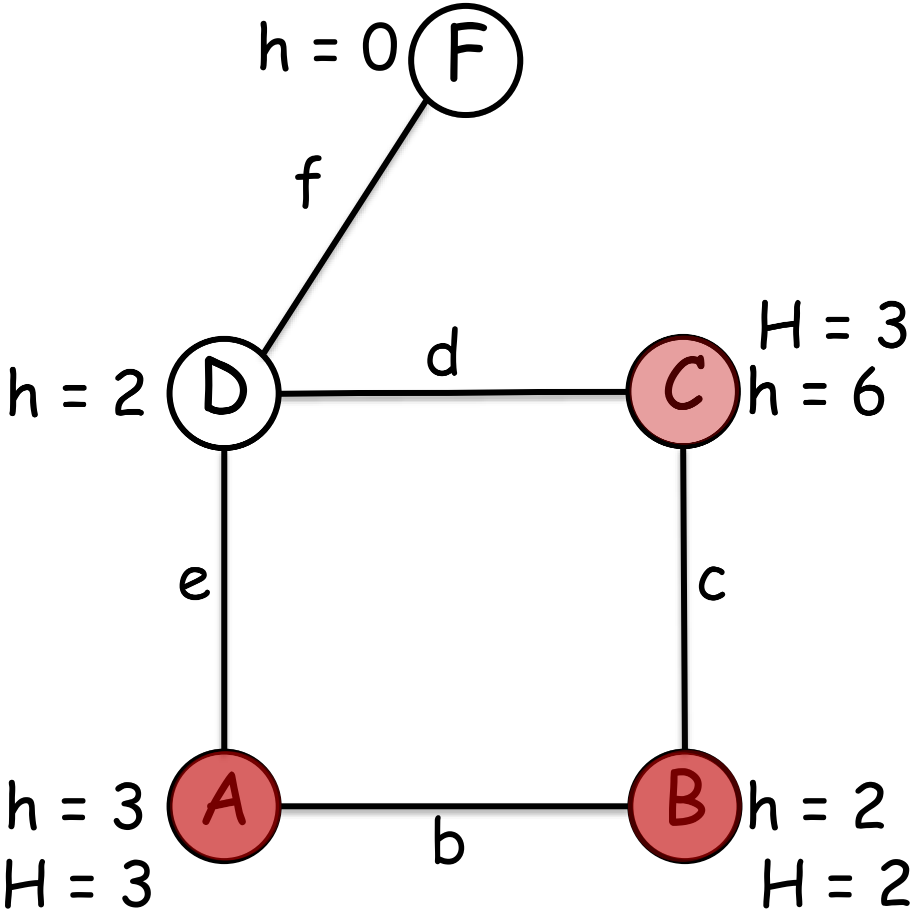


LRTA* (Example)



s' = A   s = B

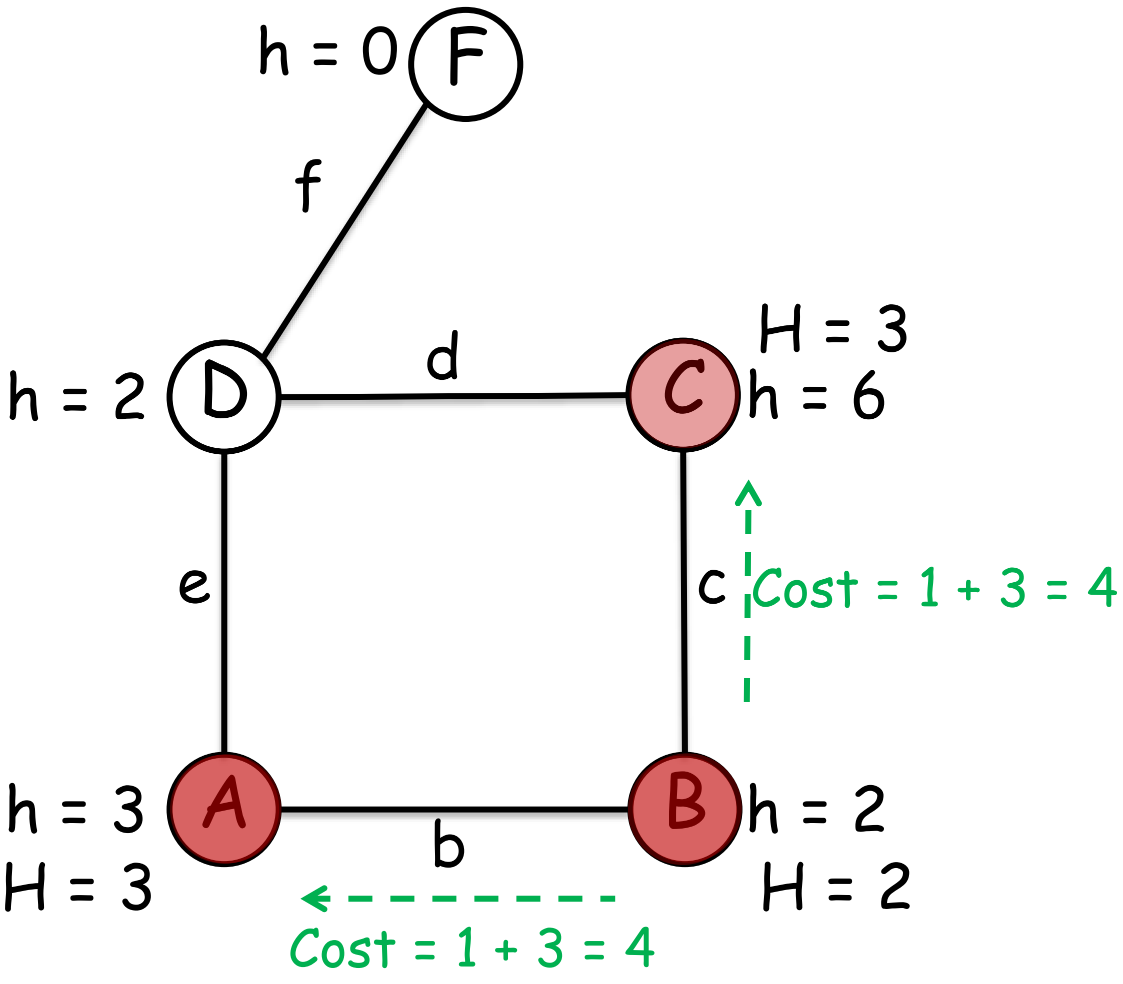
LRTA* (Example)



s' = A s = B

result(B, b) = A

LRTA* (Example)

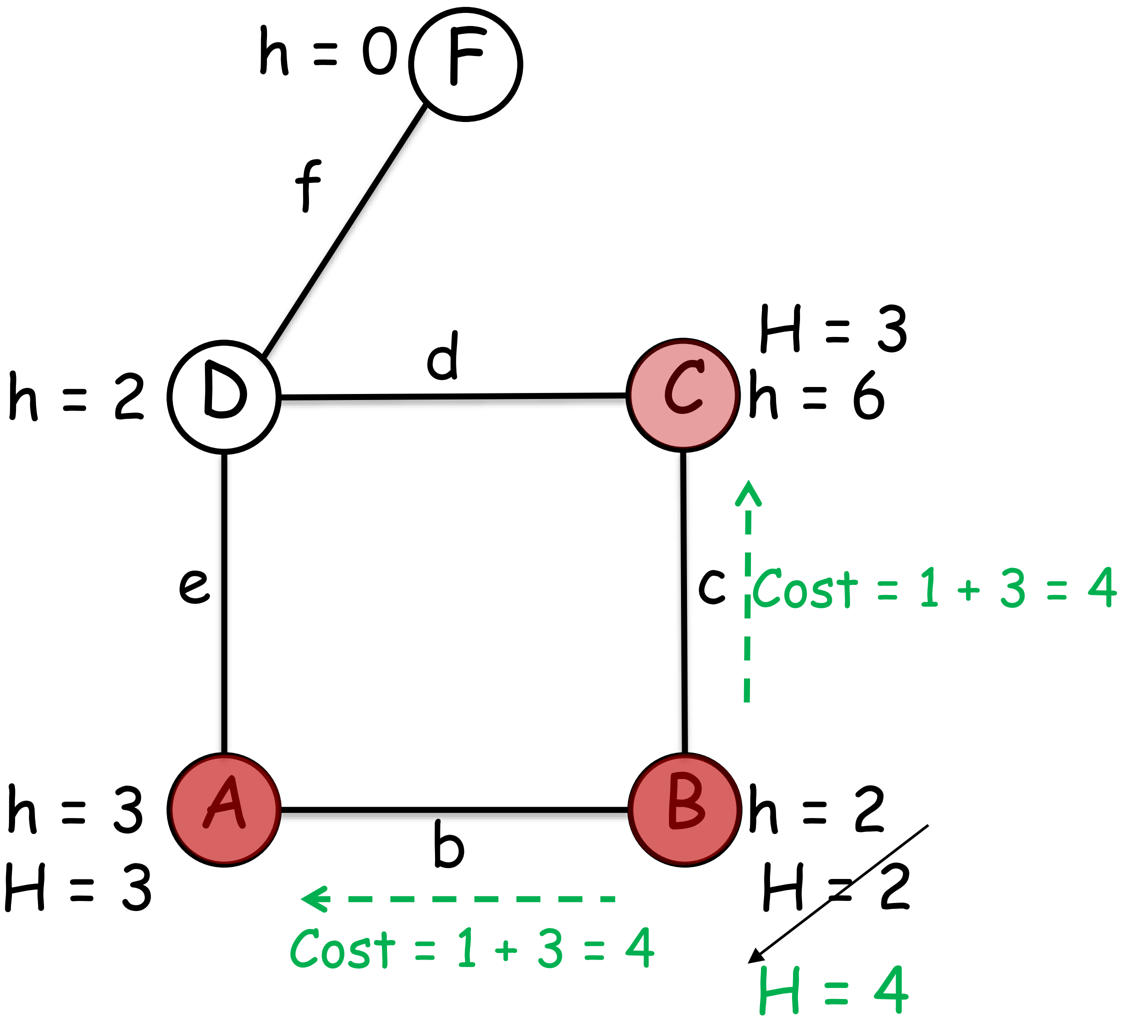


s' = A

s = B

result(B, b) = A

LRTA* (Example)

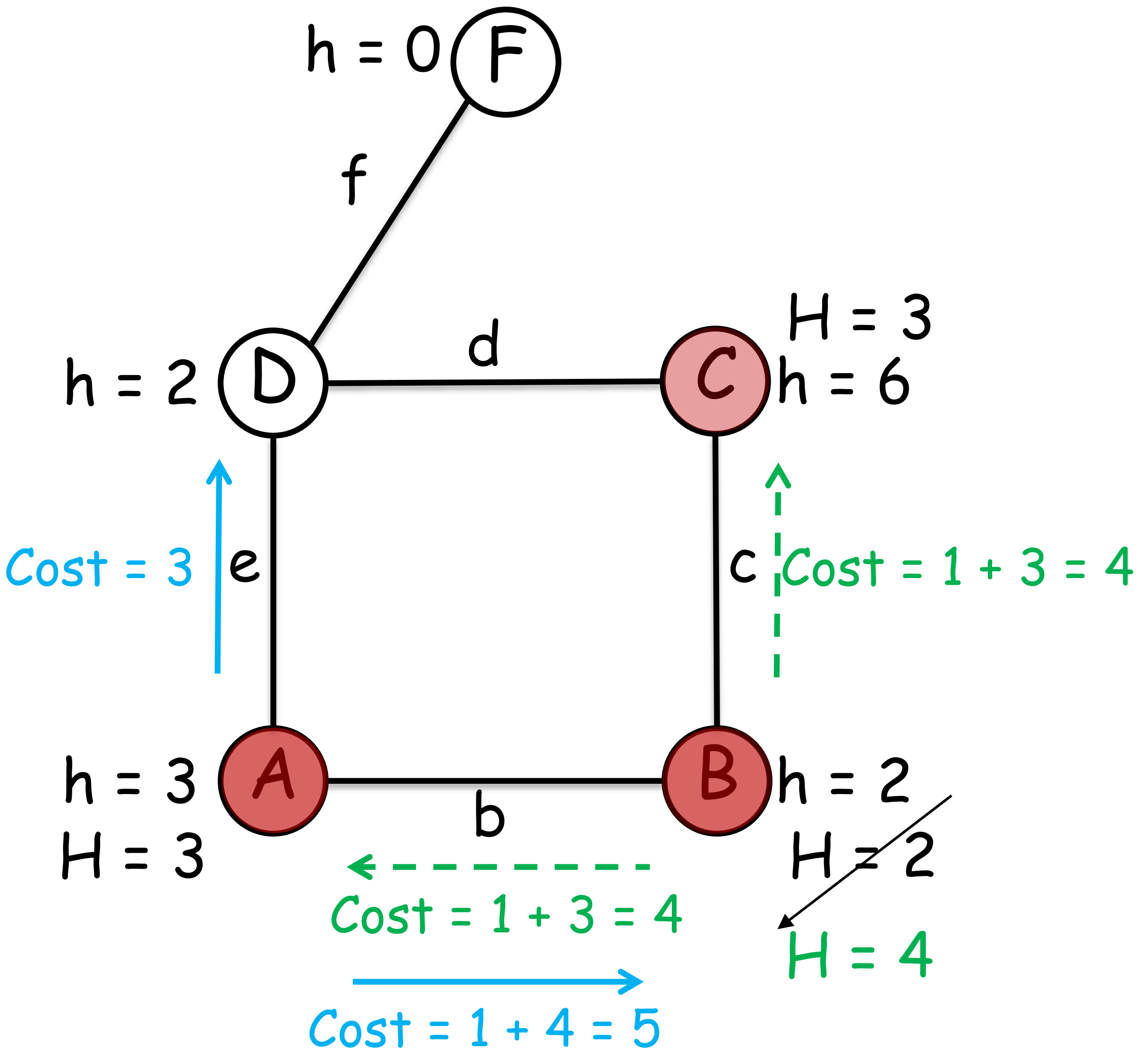


s' = A   s = B

result(B, b) = A

H(B) = 4

LRTA* (Example)



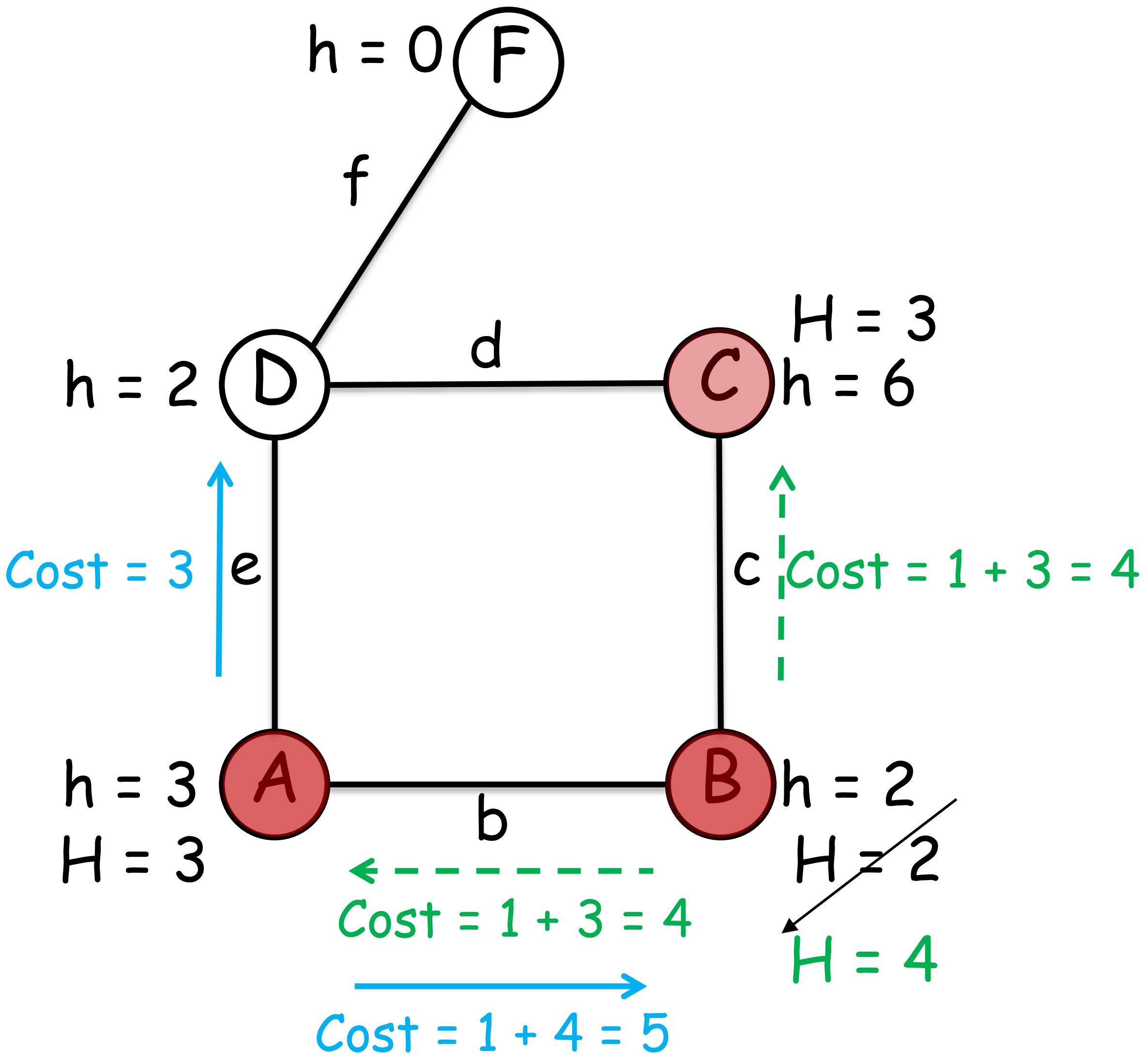
s' = A   s = B

result(B, b) = A

H(B) = 4

a = e

LRTA* (Example)



s' = A   s = B

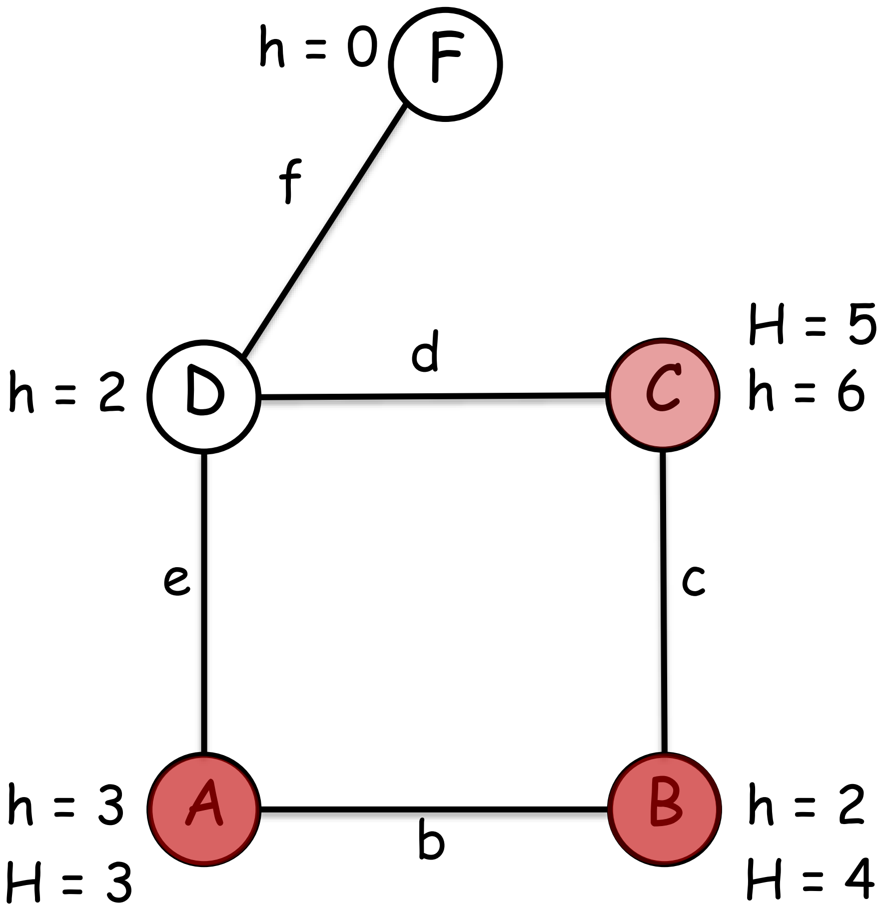
result(B, b) = A

H(B) = 4

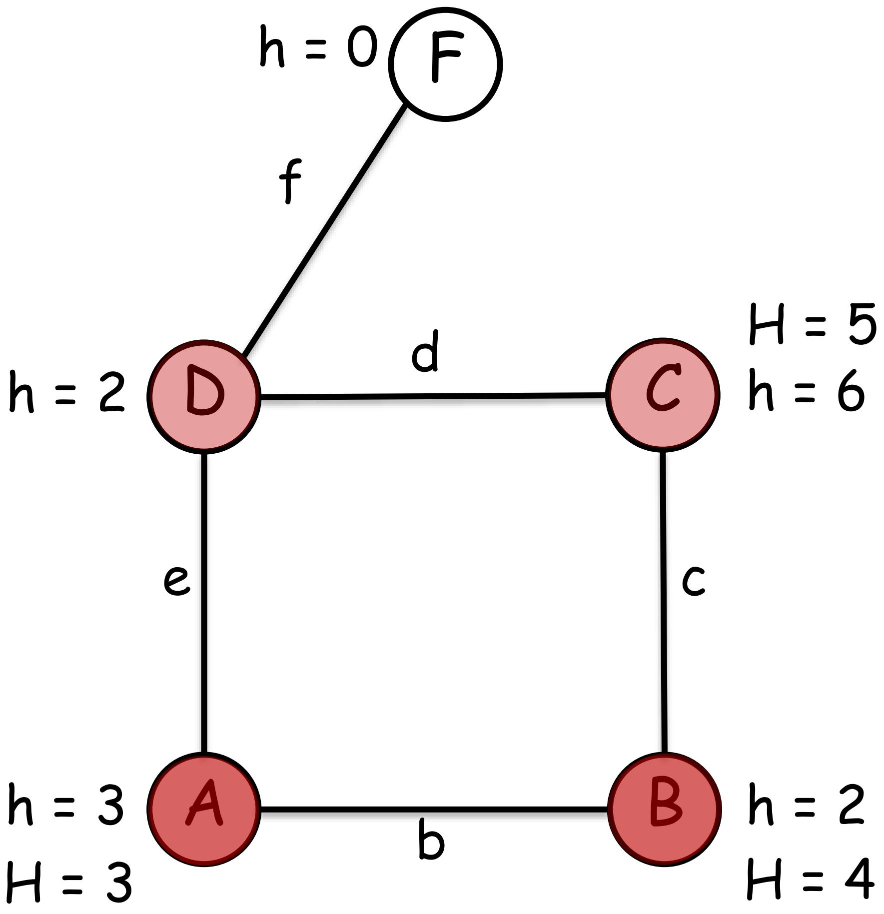
a = e

s = A

LRTA* (Example)



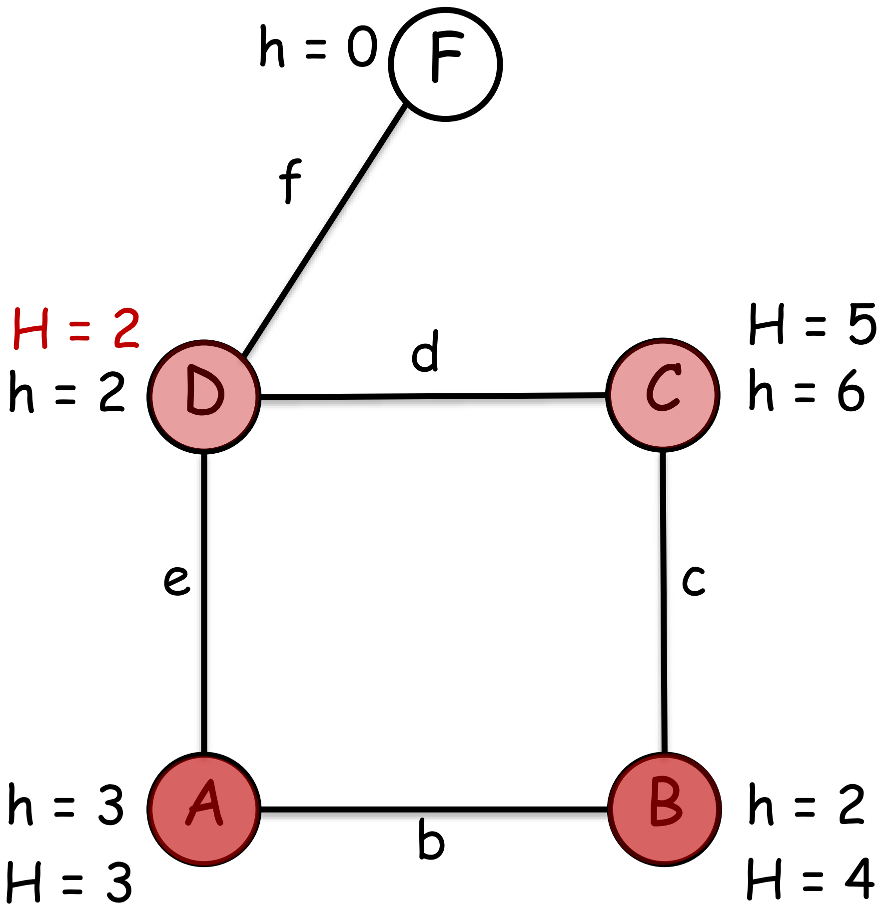
LRTA* (Example)



s' = D

s = A

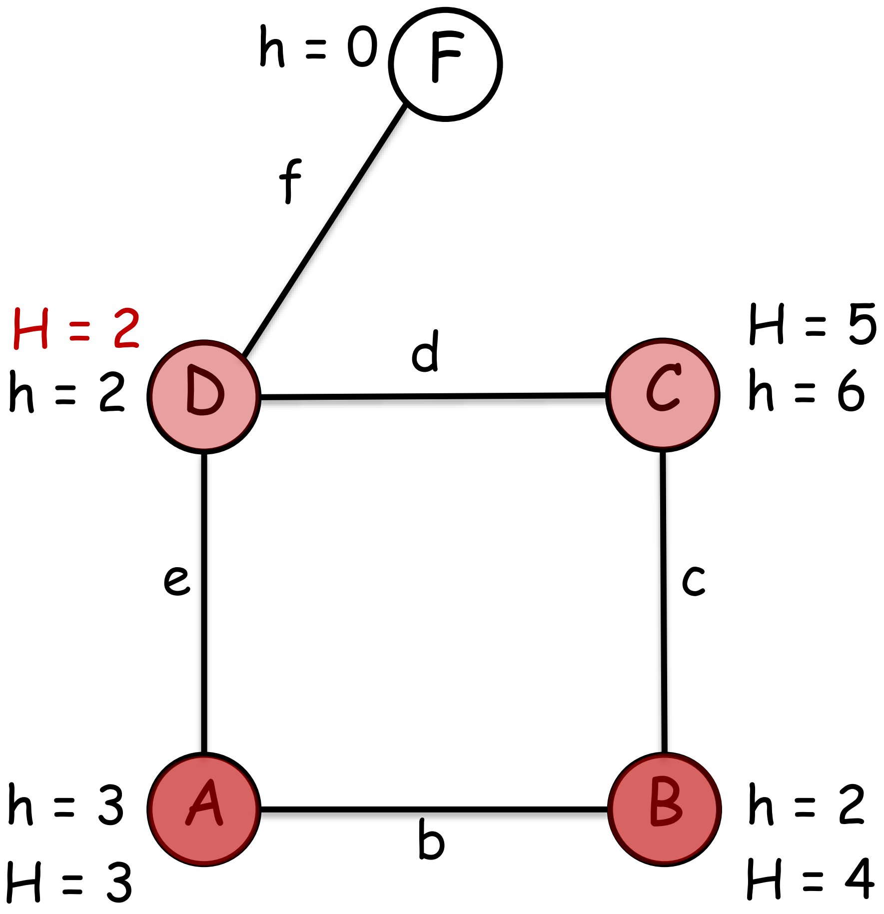
LRTA* (Example)



s' = D s = A

H(D) = h(D) = 2

LRTA* (Example)

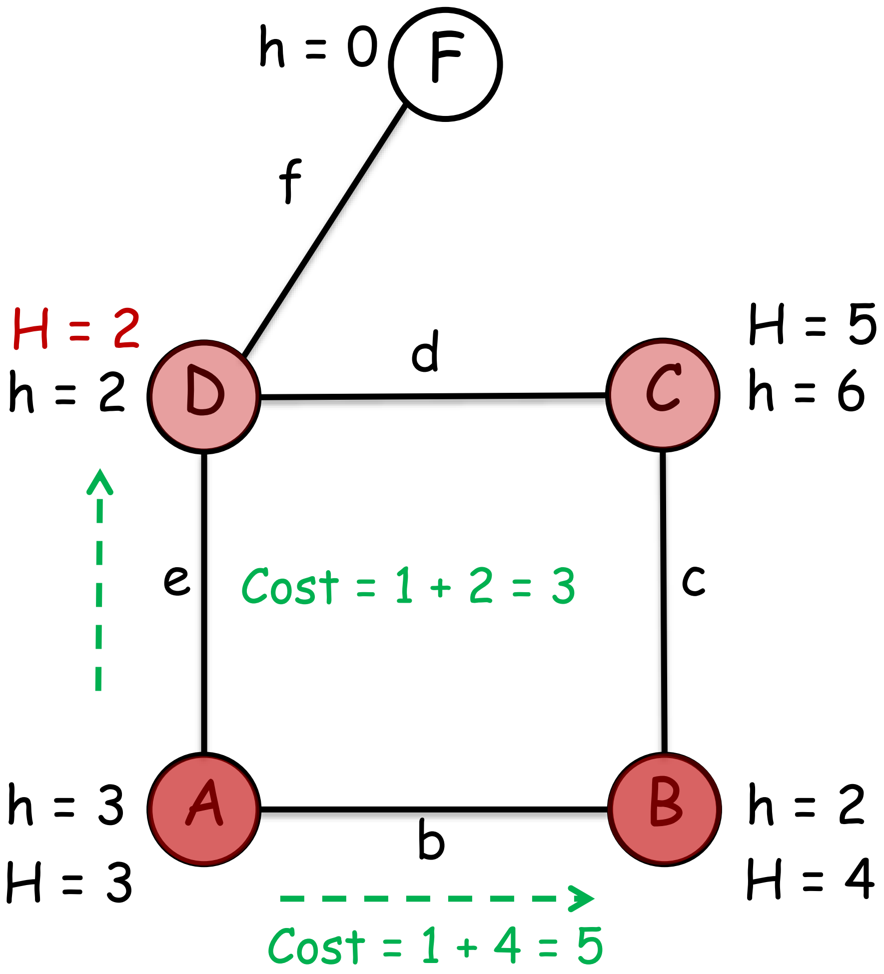


$s' = D$ $s = A$

$H(D) = h(D) = 2$

$result(A, e) = D$

LRTA* (Example)



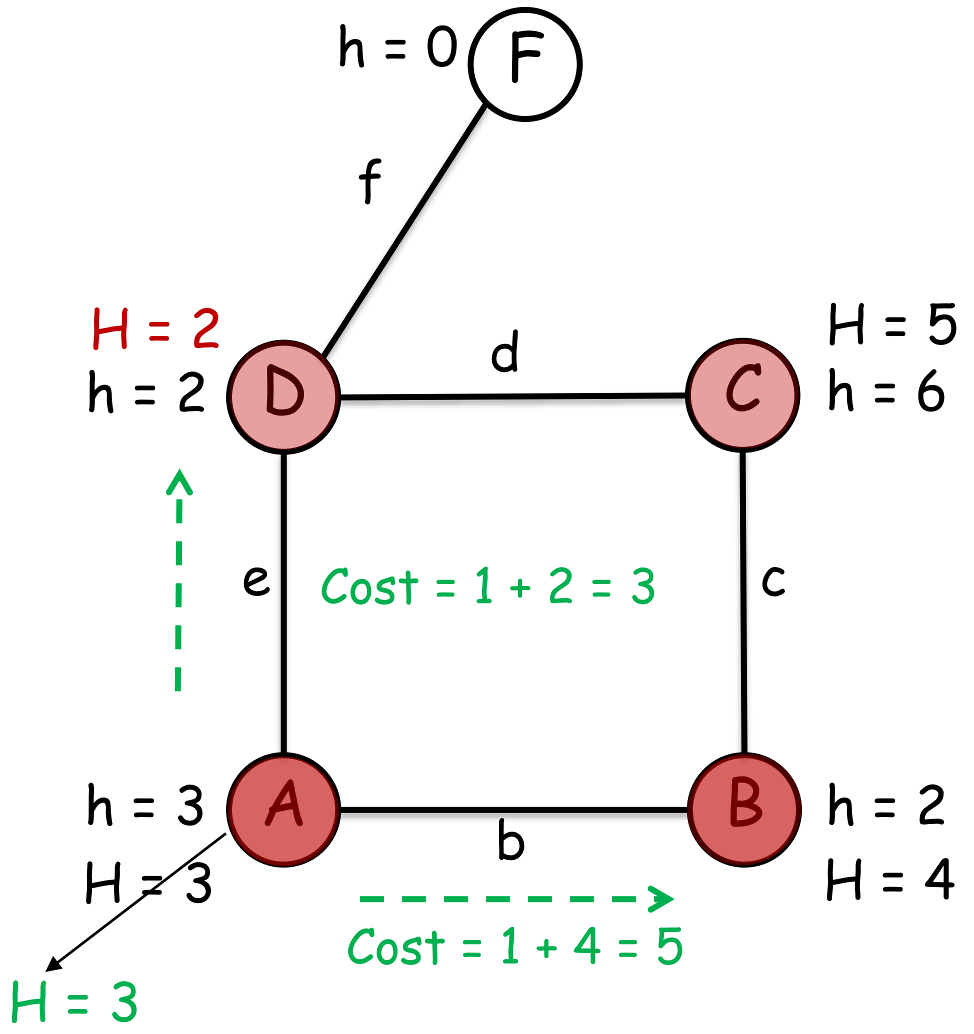
s' = D

s = A

H(D) = h(D) = 2

result(A, e) = D

LRTA* (Example)



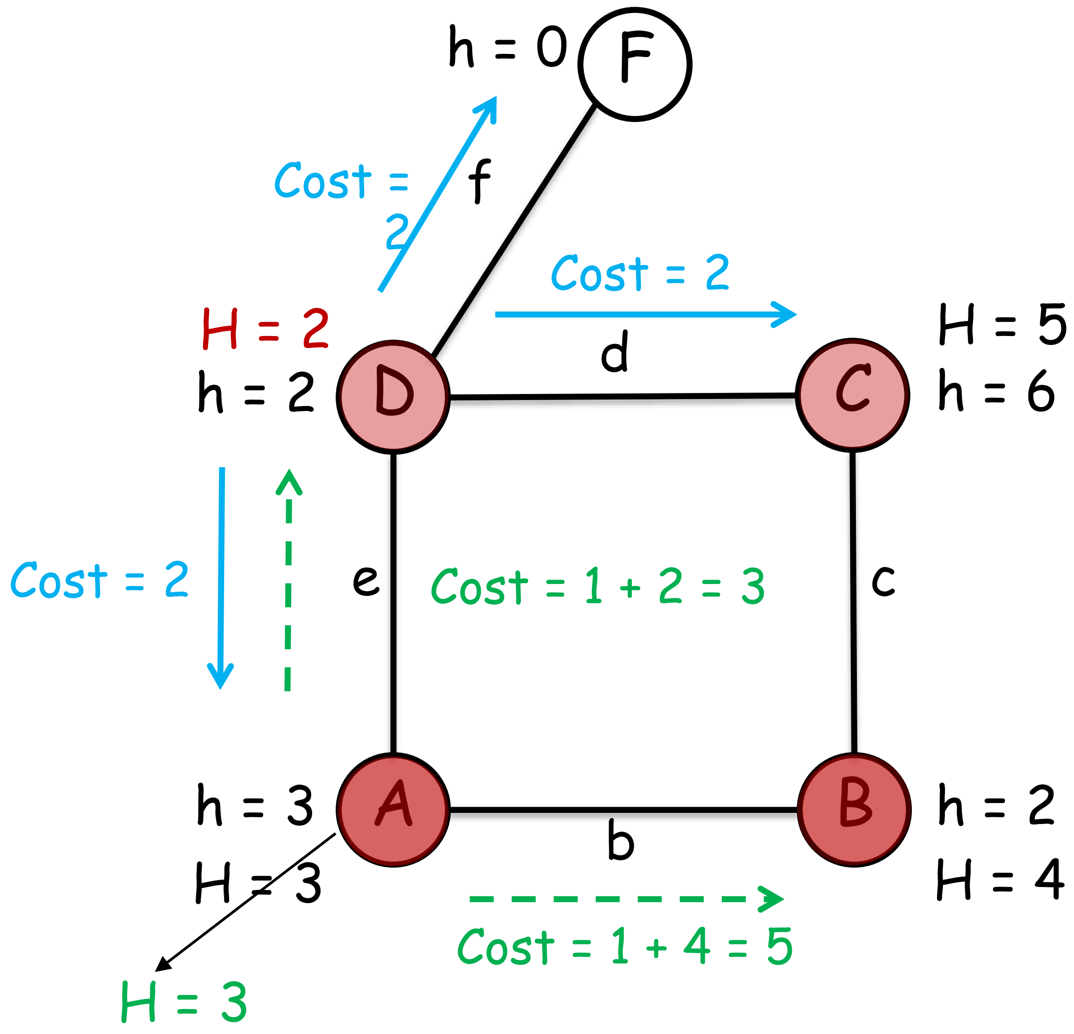
s' = D s = A

H(D) = h(D) = 2

result(A, e) = D

H(A) = 3

LRTA* (Example)



s' = D   s = A

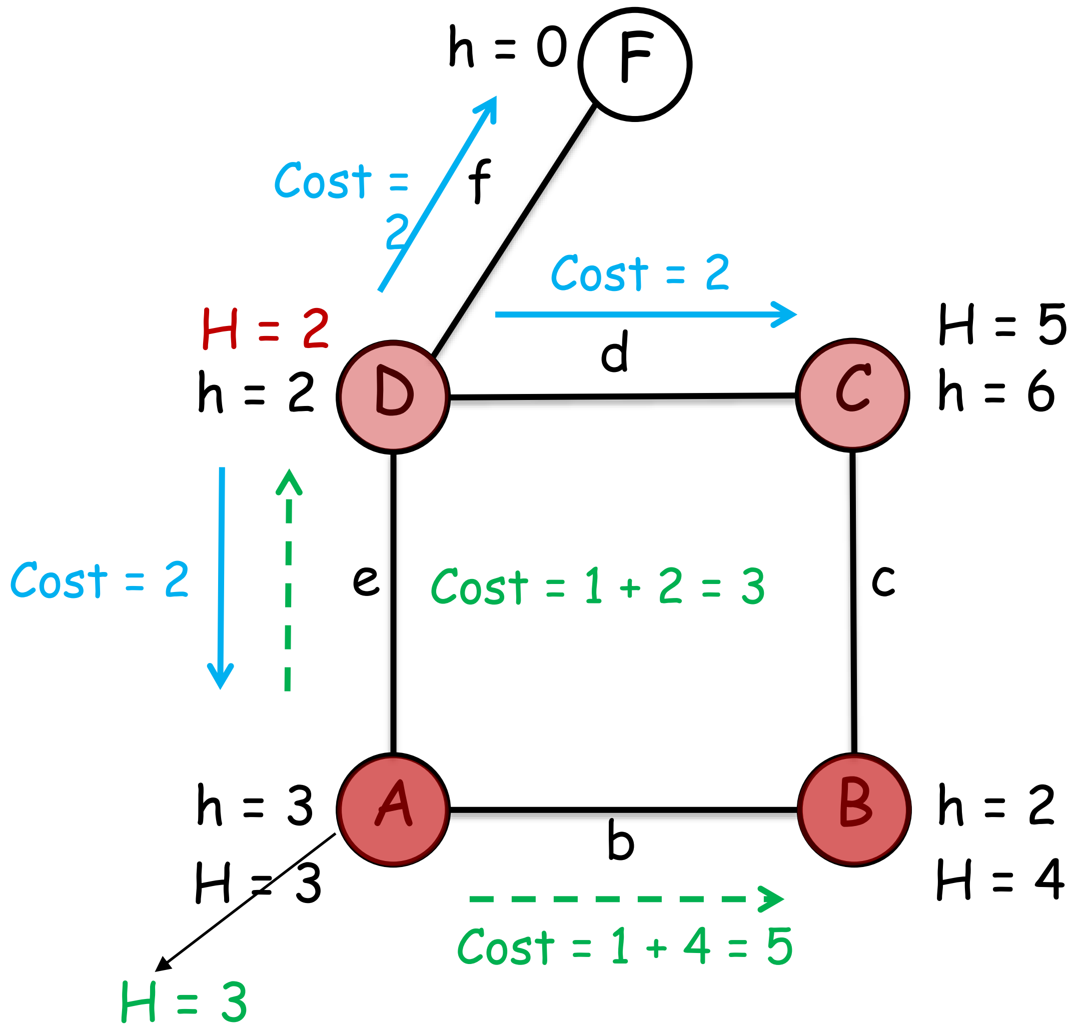
H(D) = h(D) = 2

result(A, e) = D

H(A) = 3

a = f

LRTA* (Example)



s' = D s = A

H(D) = h(D) = 2

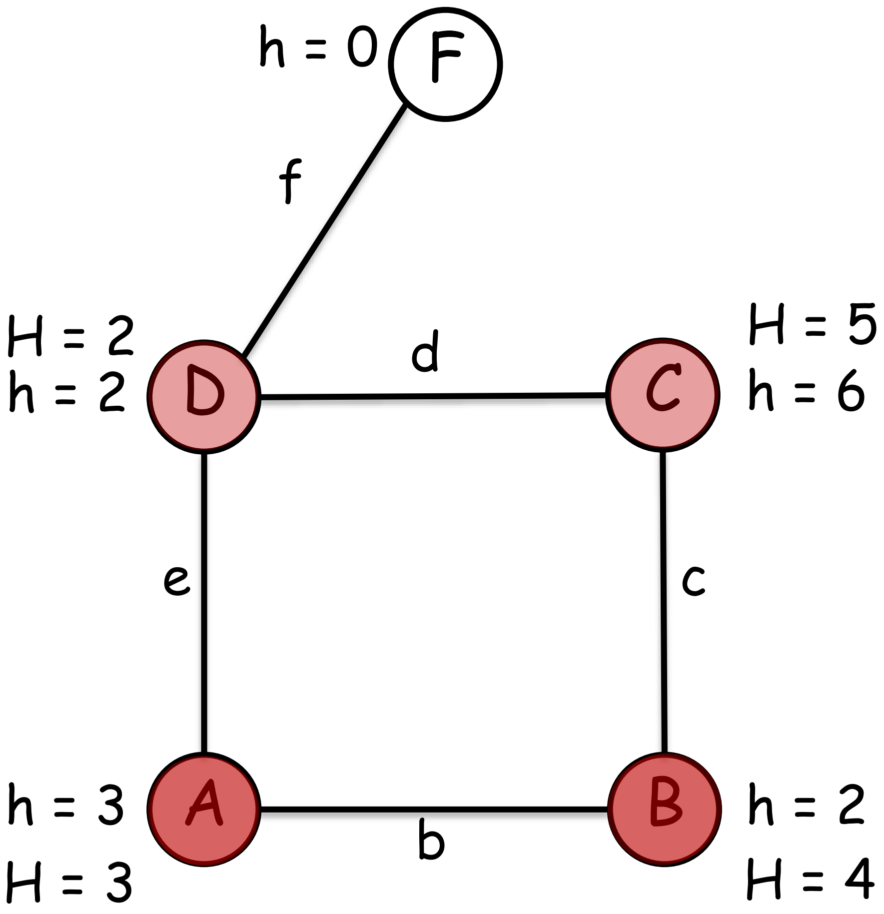
result(A, e) = D

H(A) = 3

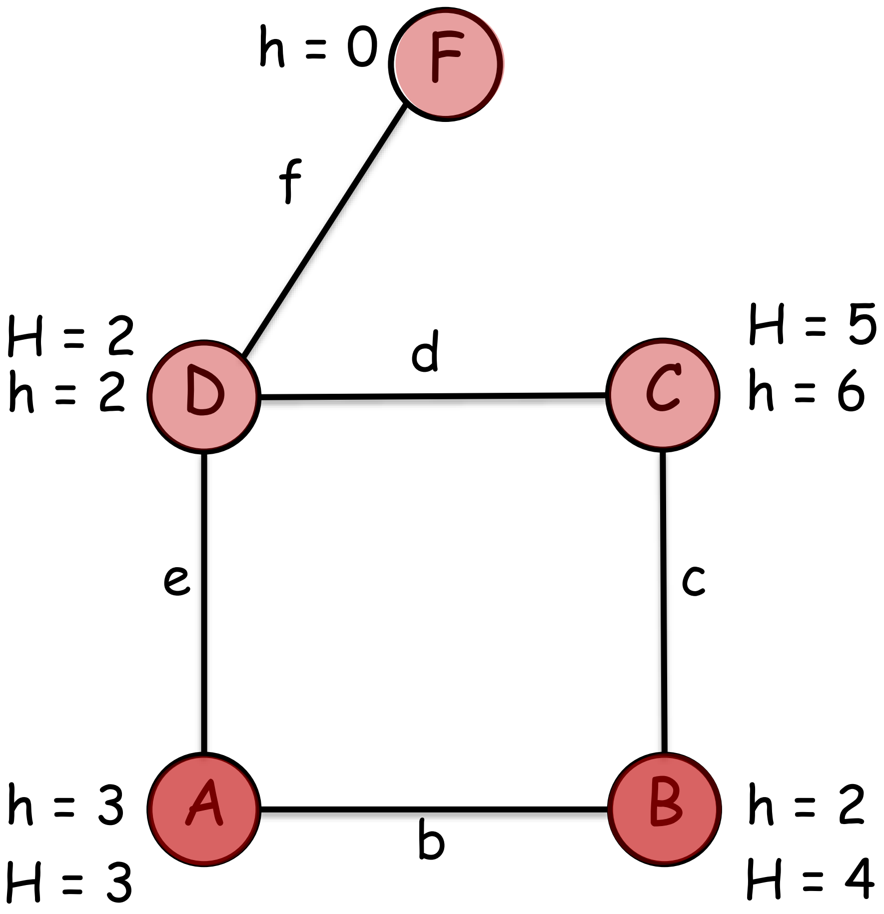
a = f

s = D

LRTA* (Example)



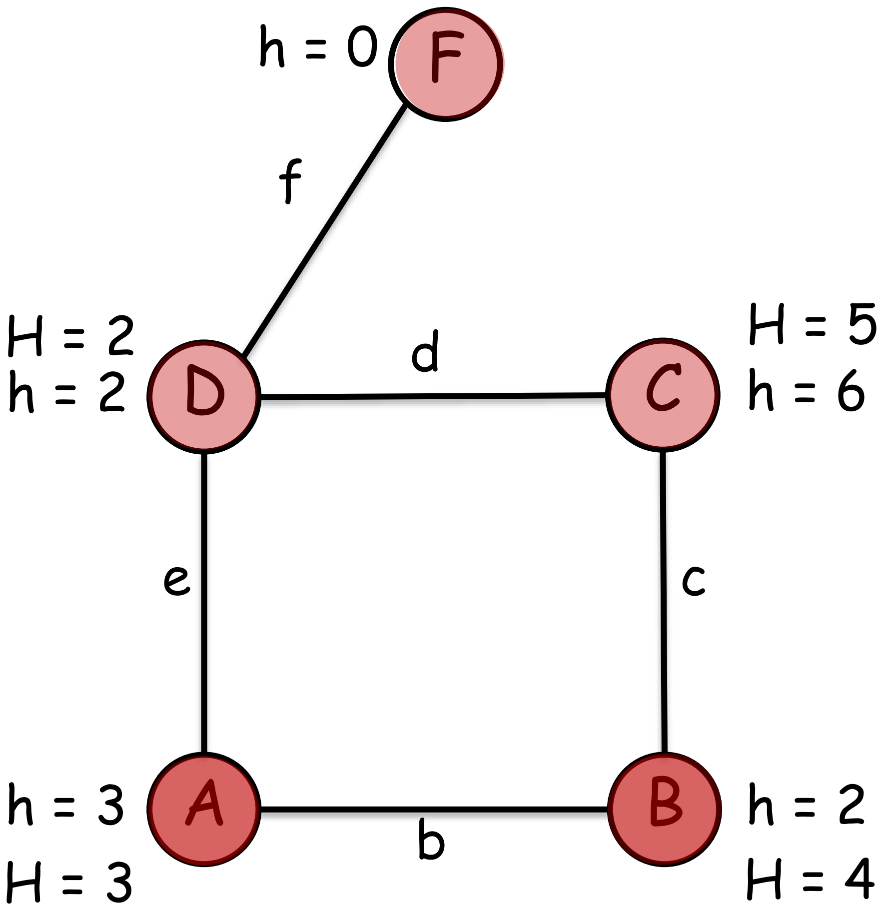
LRTA* (Example)



s' = F

s = D

LRTA* (Example)

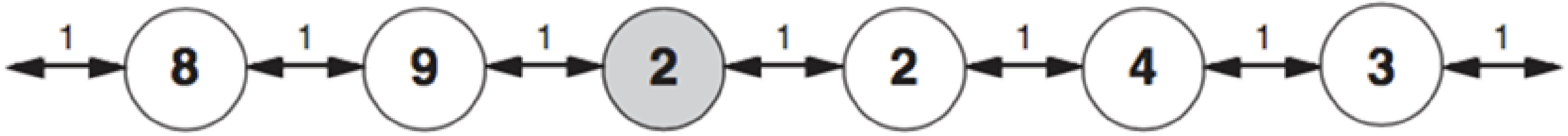


s' = F s = D

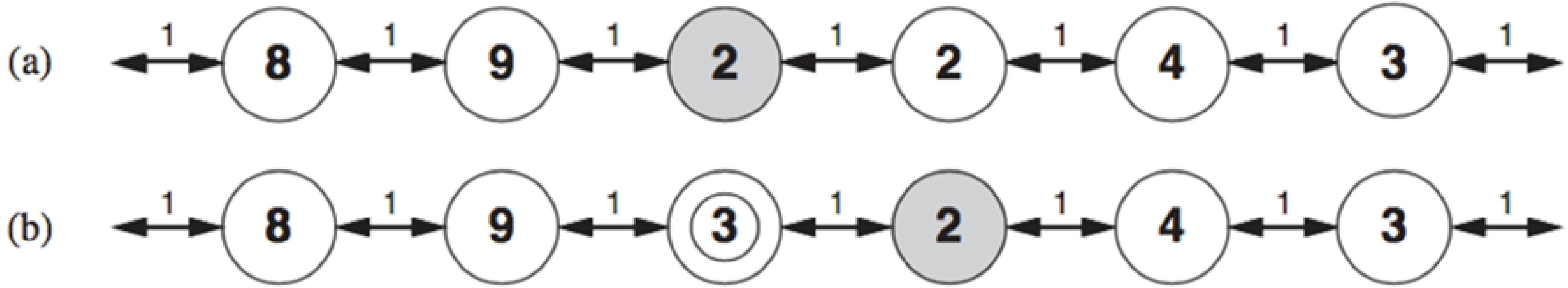
F is Goal return STOP

LRTA* (Example)

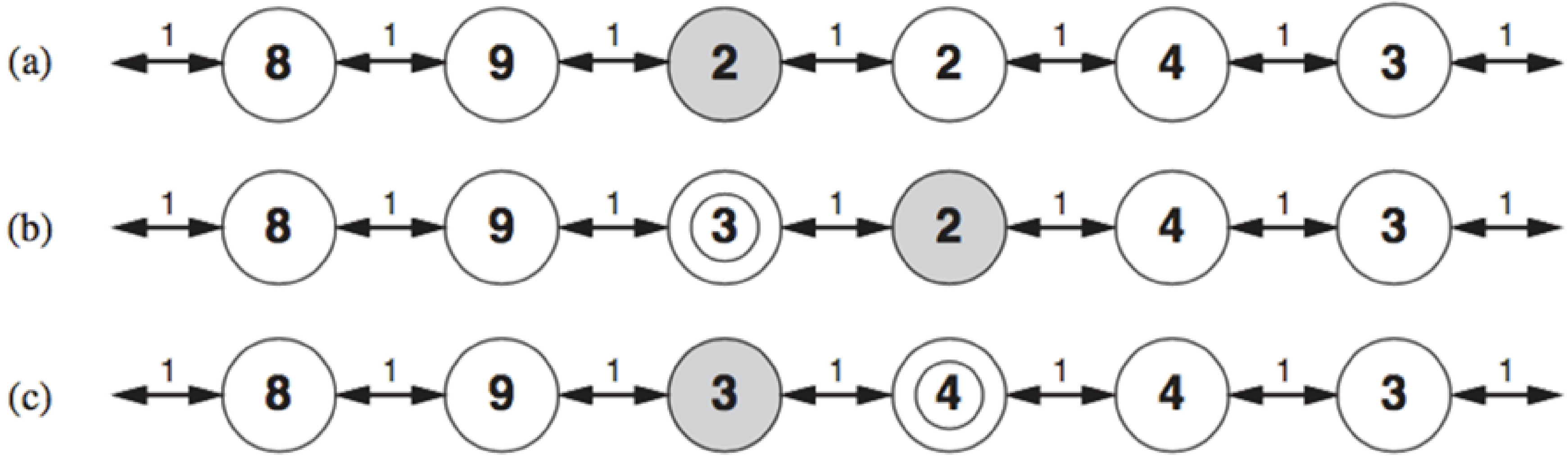
(a)



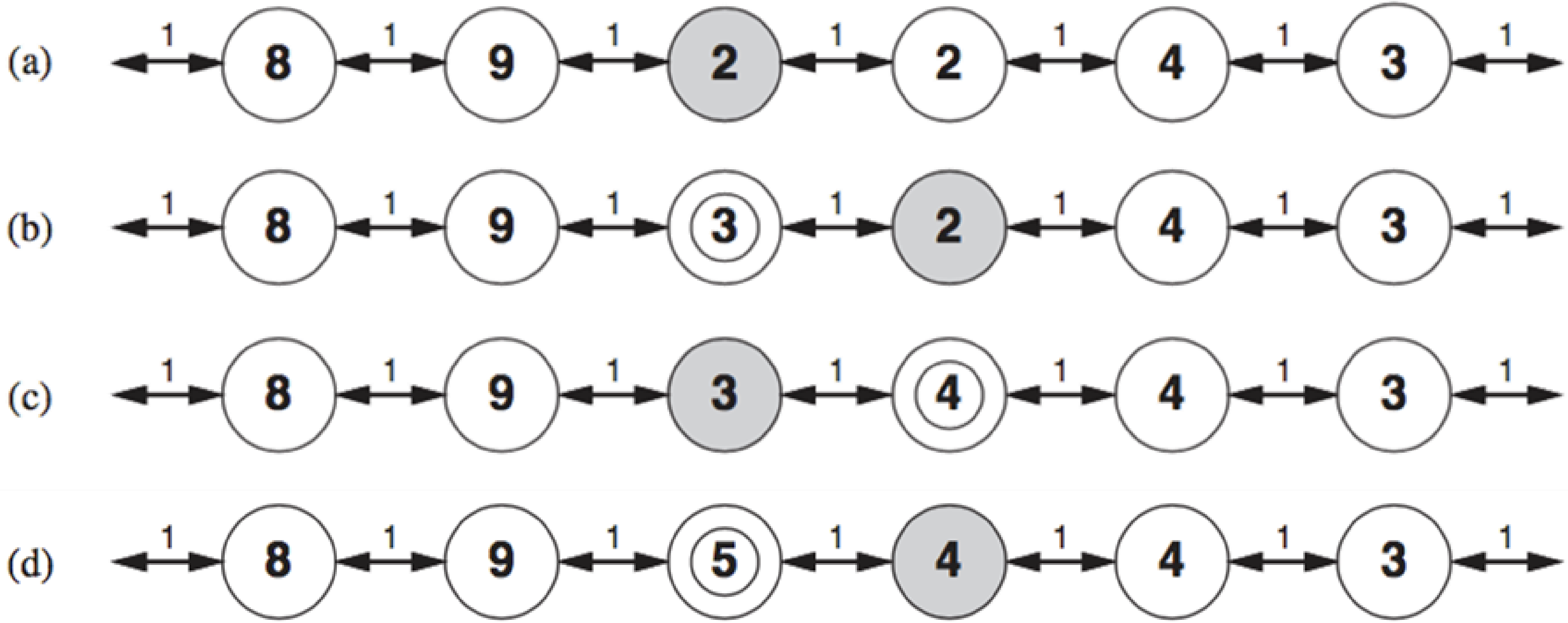
LRTA* (Example)



LRTA* (Example)



LRTA* (Example)



LRTA* (Example)