Search

Chen-Yu Wei

Question

A **farmer** wants to get his **cabbage**, **goat**, and **wolf** across a river. He has a boat that only holds two. He cannot leave the cabbage and goat alone or the goat and wolf alone. How many river crossings does he need?

- 4

- 5

- 6

- 7

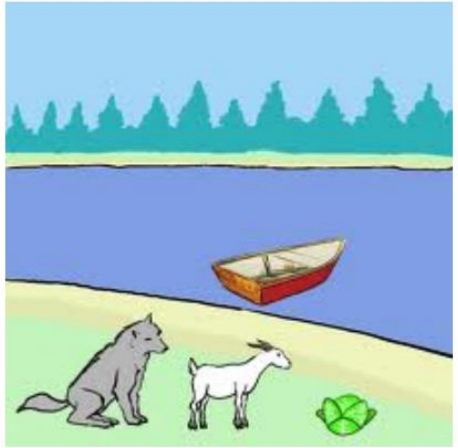
- no solution

https://stanford-cs221.github.io/autumn2023-extra/modules/search/modeling.pdf

Model the Problem

How many different “states”?

How many different “actions”?



Farmer Cabbage Goat Wolf

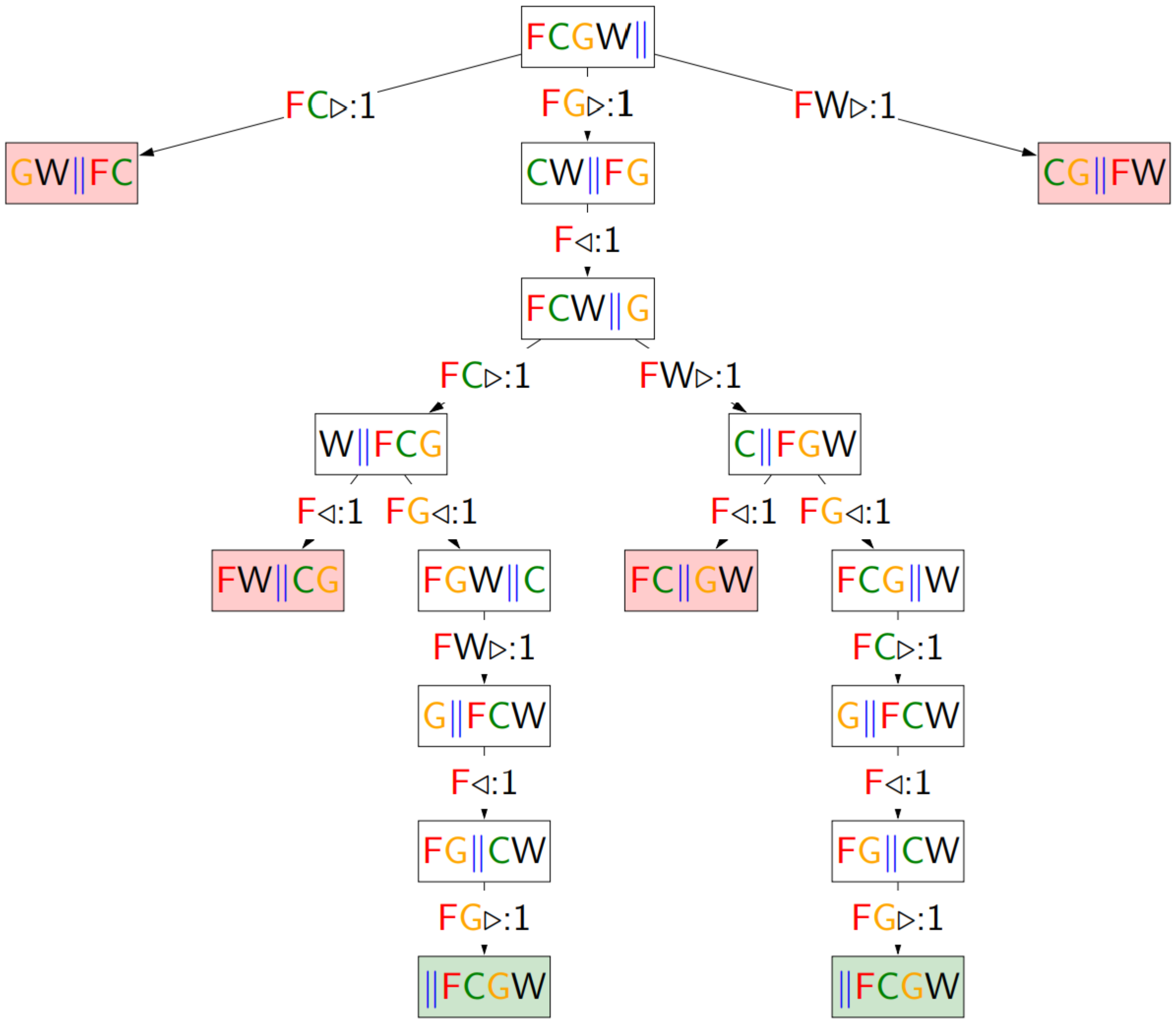
F▷ F◁

FC▷ FC◁

FG▷ FG◁

FW▷ FW◁

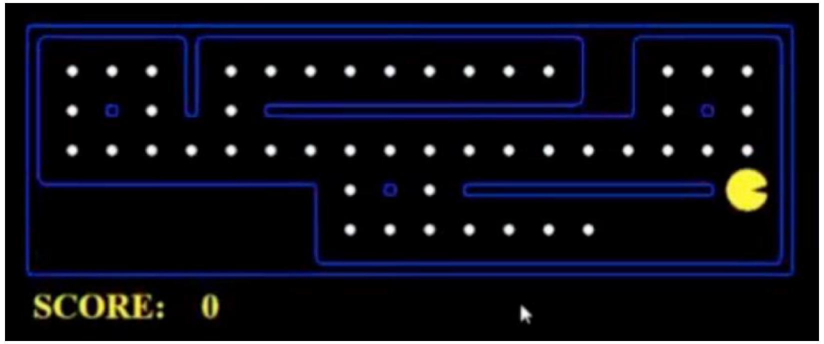
Build a Search Tree



Search Problem

- State space
- Initial state
- Goal test: Given a state, return whether the state is a goal
- Action
- Successor function: Given current state and action, return the new state
- (The cost of an action)

Example: PACMAN



Eat all dots

States: {(x,y), dot booleans}

Actions: NSEW

Successor: update location and possibly a dot boolean

Goal test: dots all false

Go to some destination

States: (x,y) location

Actions: NSEW

Successor: update location only

Goal test: is (x,y)=END

Example: SAINT (Slagle, 1961)

Symbolic Integrator

$$\int \frac{x^4}{(1-x^2)^{5/2}}\,dx \quad = \quad \frac{1}{3}\tan^3(\arcsin x) - \tan(\arcsin x) + \arcsin x$$

States: symbolic expression

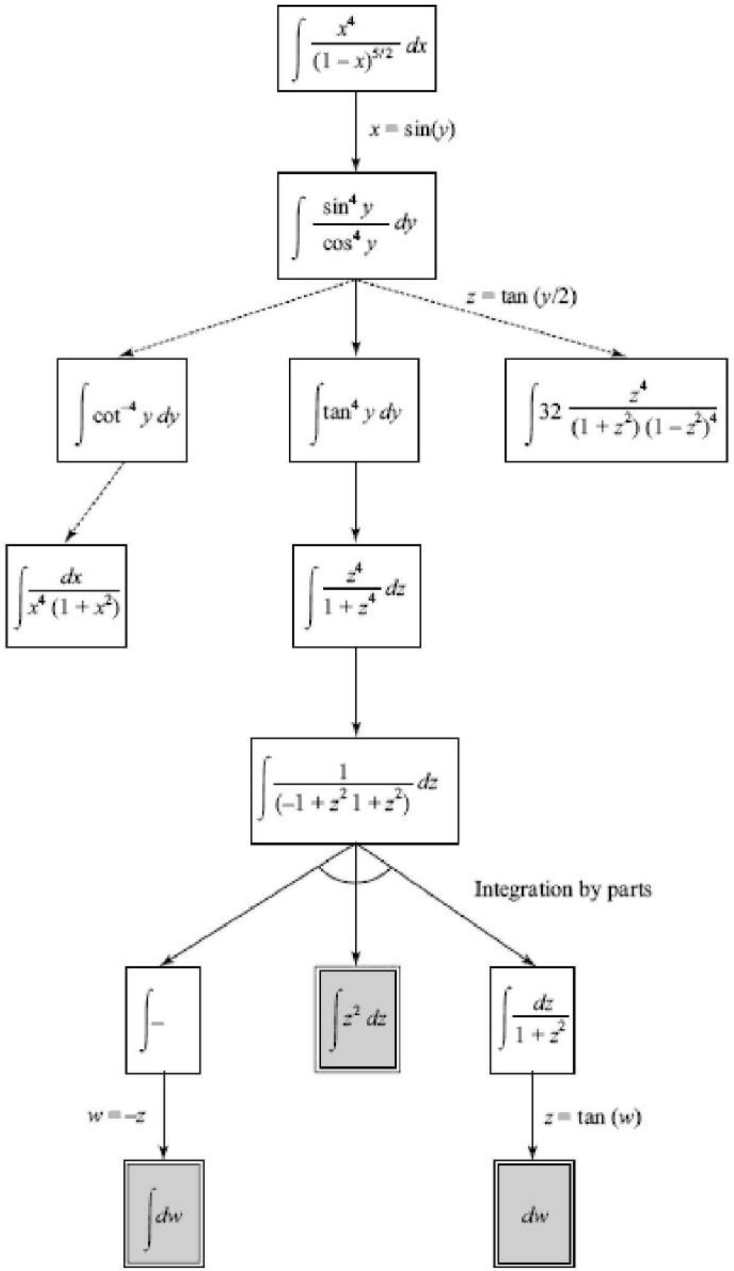
Actions: "common techniques"

Successor: the new expression after applying the technique

Goal test: whether the expression is in "standard form"

"common technique" examples:

- $\int c\, f(x)\mathrm{d}x = c \int f(x)\mathrm{d}x$
- $\int f(\tan x)\mathrm{d}x = \int \frac{f(y)}{1+y^2}\mathrm{d}y$
- If seeing $1 - x^2$, then substitute $x = \sin y$



Example: Machine translation

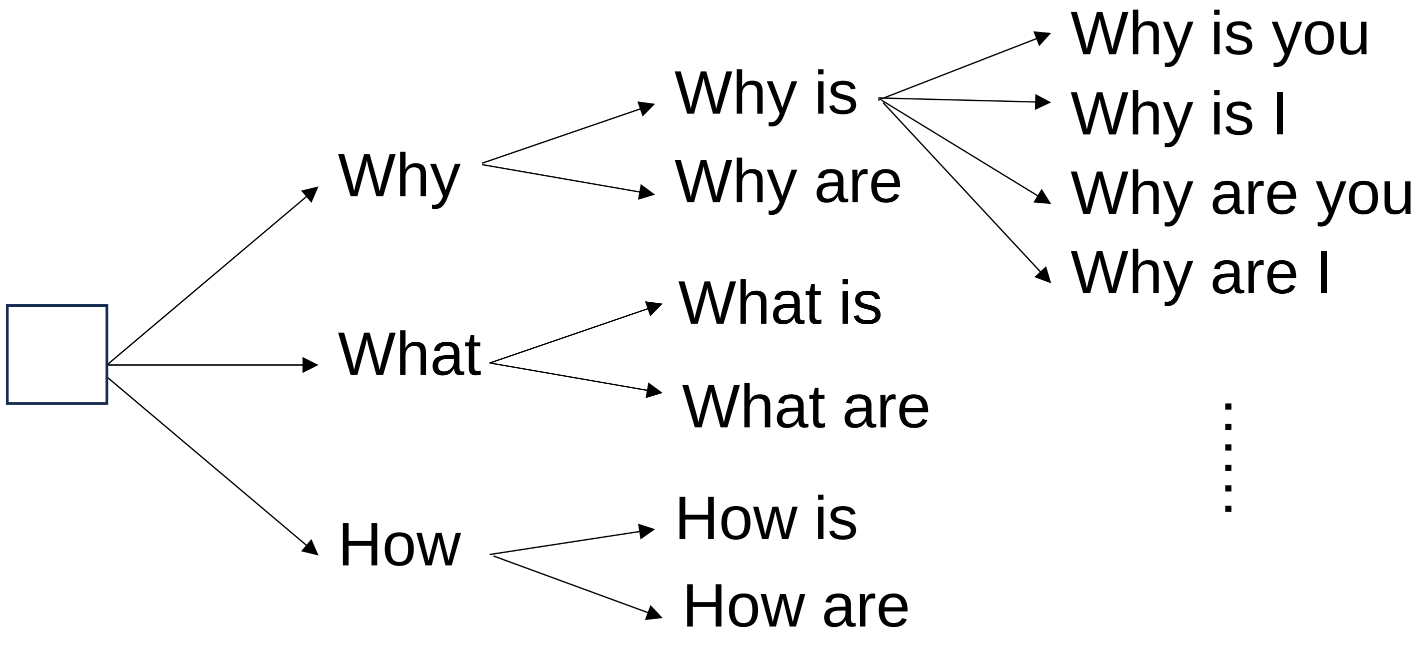
Translate “你好嗎” to English

States: current word sequence

Actions: the next word

Successor: the concatenation of current sequence and next word

Goal test: whether the current sequence means the same as 你好嗎

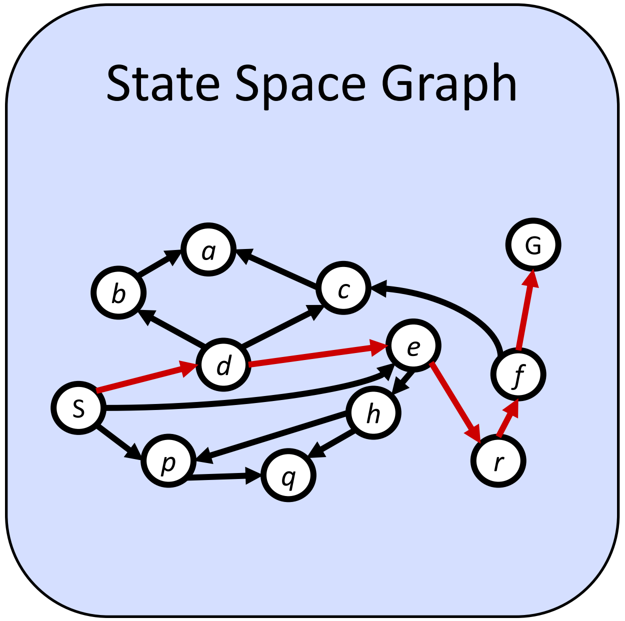


Topics

- BFS
- DFS
- UCS (Dijkstra Algorithm)
- Difference with DSA2:
  - The state space is exponentially large, and it’s unlikely we’ll store the whole state space in memory

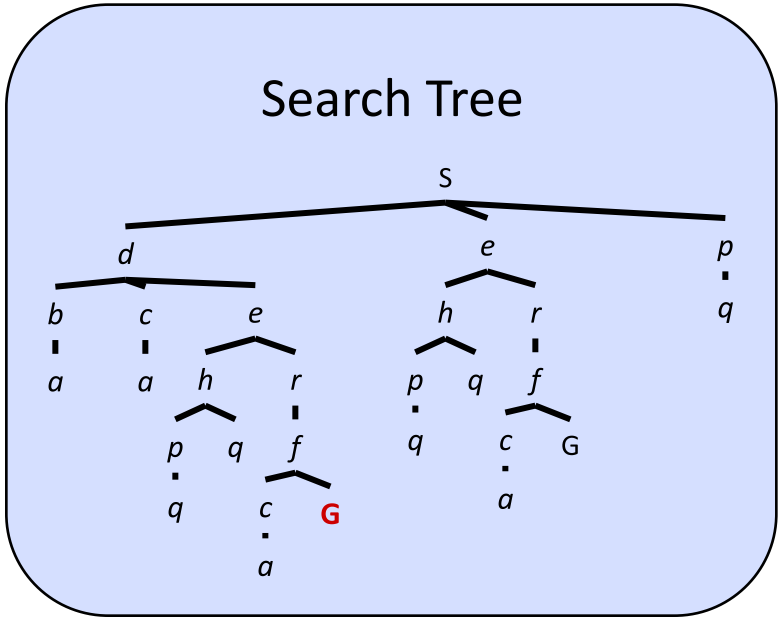
General Framework

State Space and Search Tree



Each NODE in in the search tree is an entire PATH in the state space graph.

We construct both on demand – and we construct as little as possible.



A General Framework

Expanded $\leftarrow$ { }
Frontier $\leftarrow$ { initial_state }
While **Frontier** is not empty:

Choose a node s from **Frontier** ①

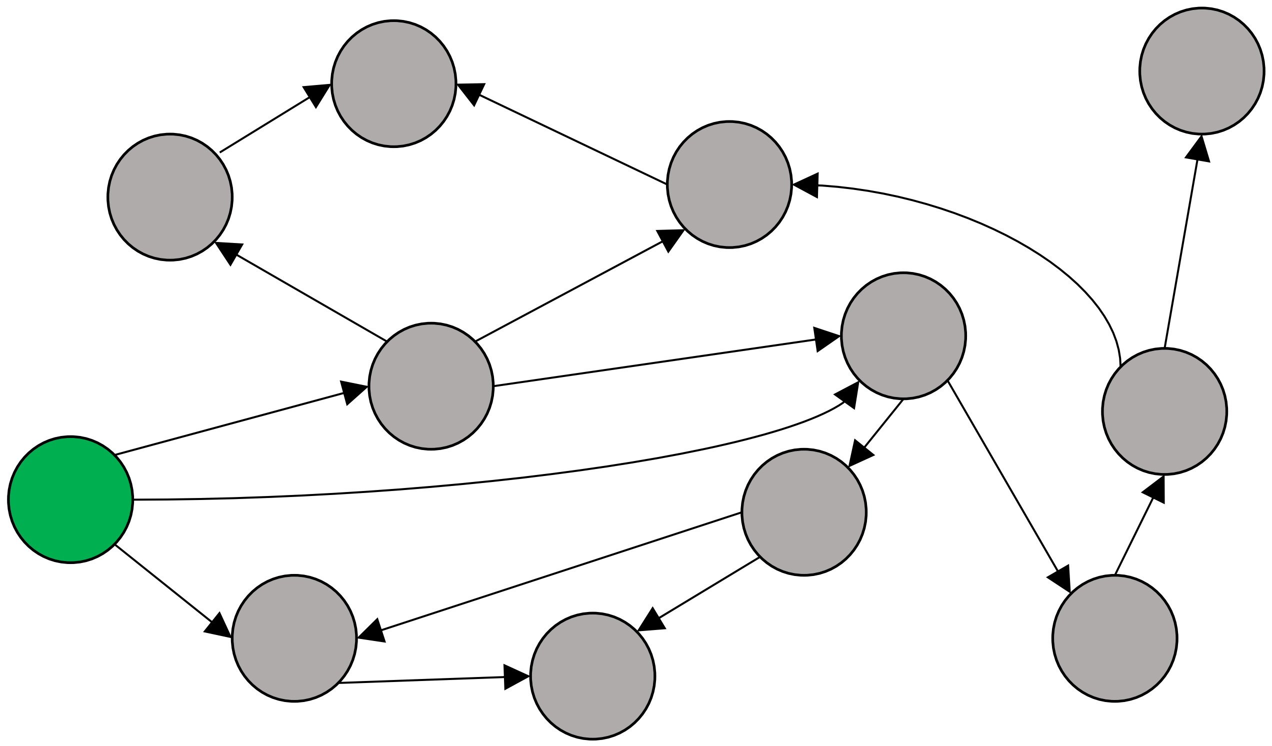
For all action a:
$s' \leftarrow \mathrm{succ}(s, a)$
If s' has not been reached
Put s' in **Frontier** ②

Move s to **Expanded** ③

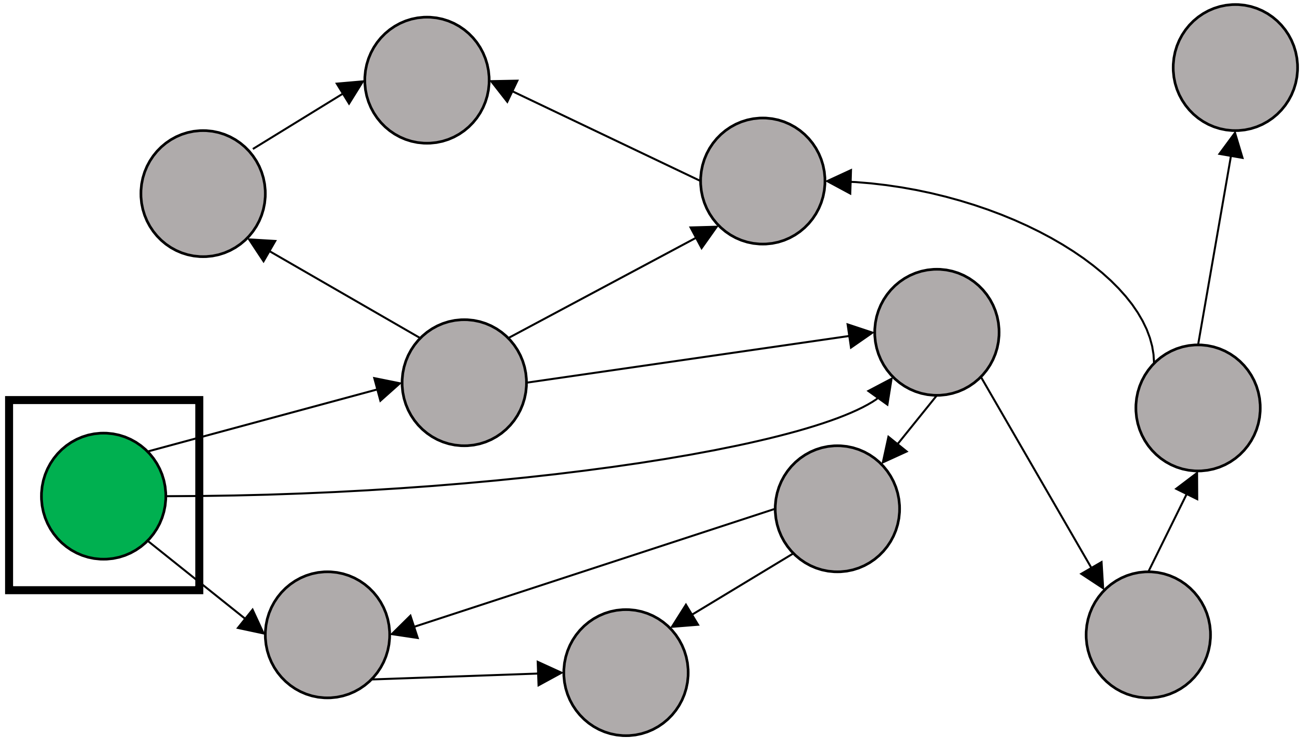
All nodes that are not in **Expanded** or **Frontier**

Nodes are divided into 3 groups: **Expanded**, **Frontier**, and **Unreached.**

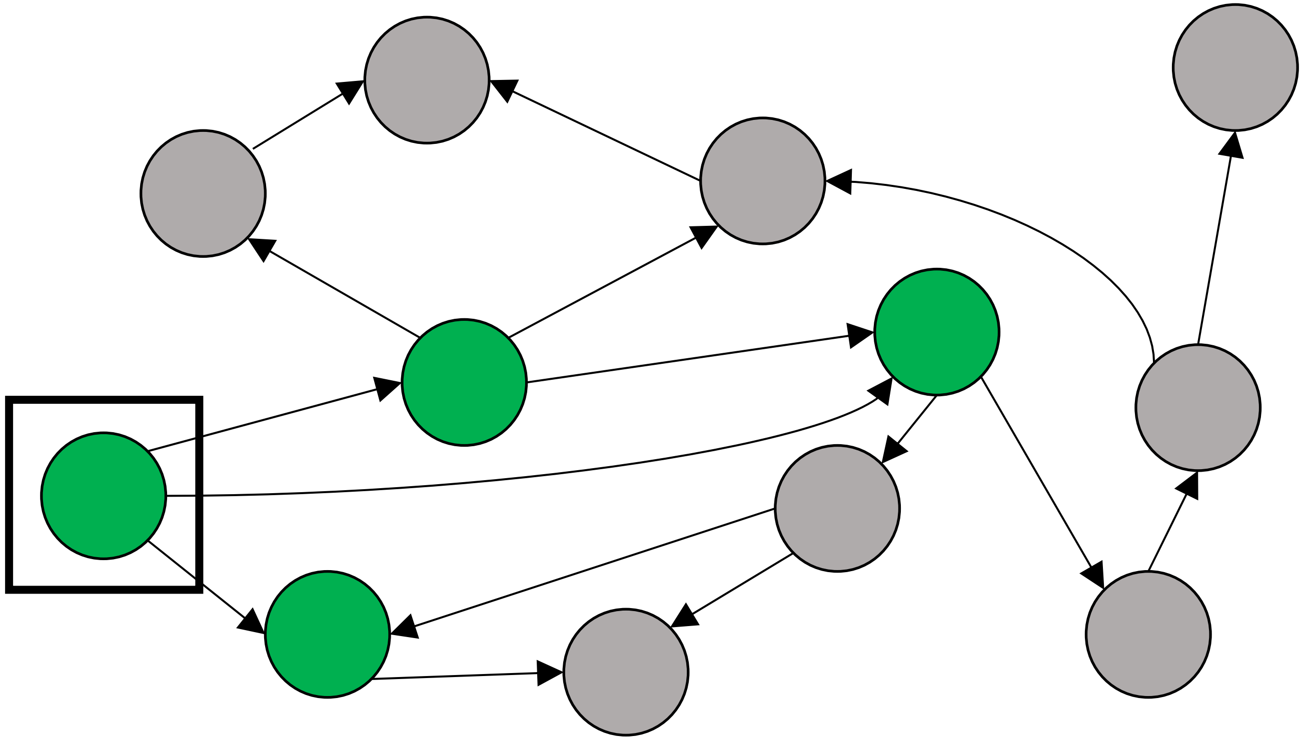
⓪ Place the initial state in **Frontier**



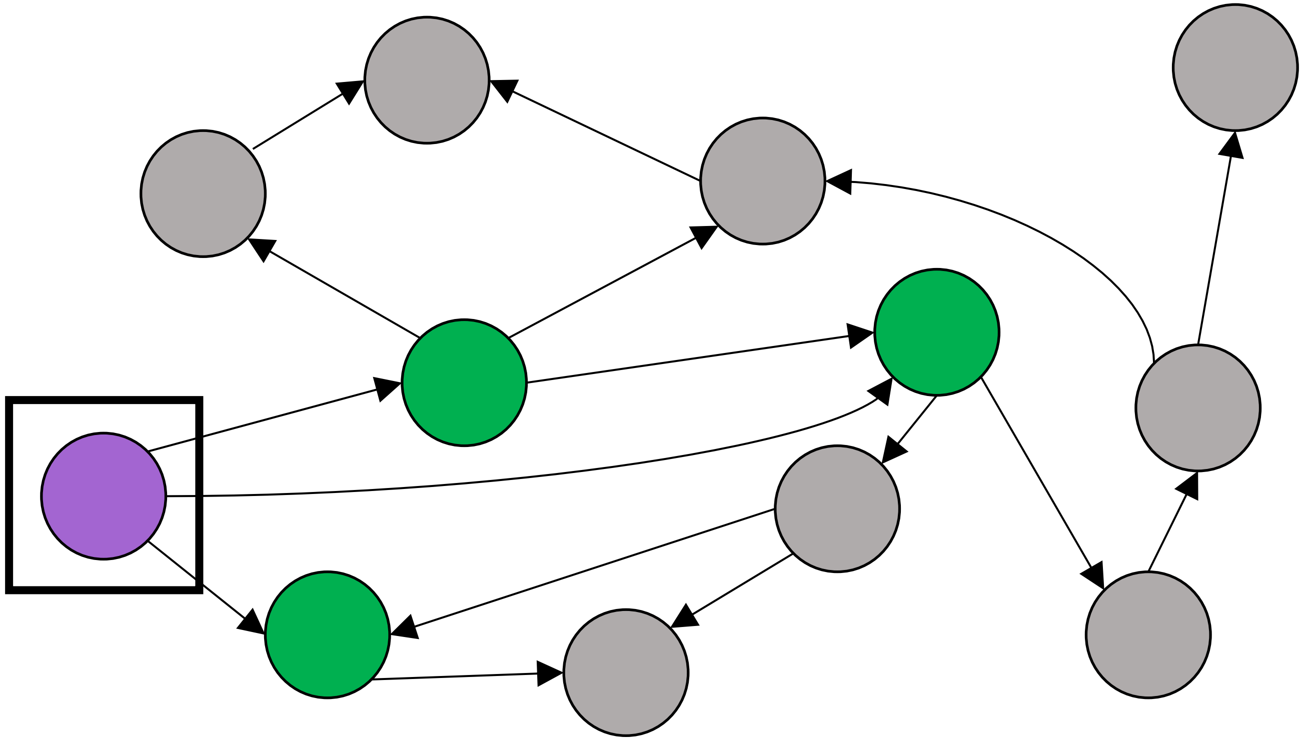
① Choose a node s from **Frontier**



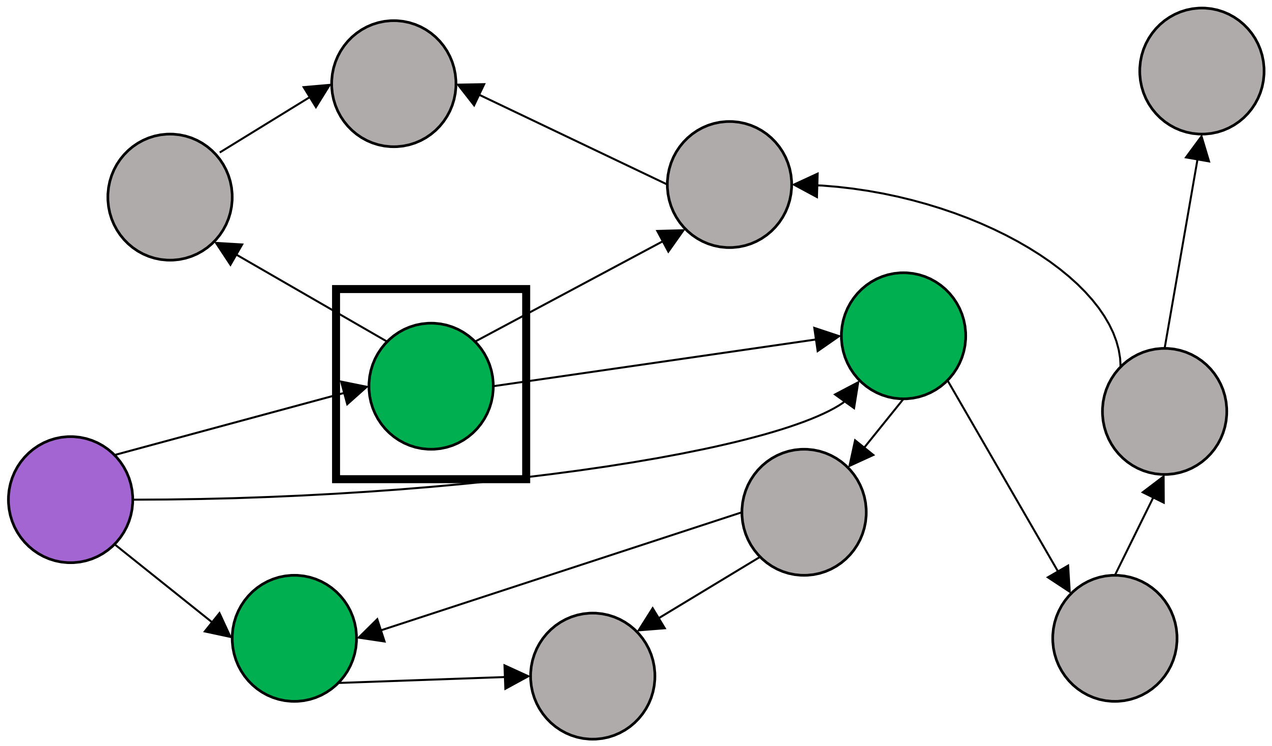
② Move all unreached successors of s to **Frontier**



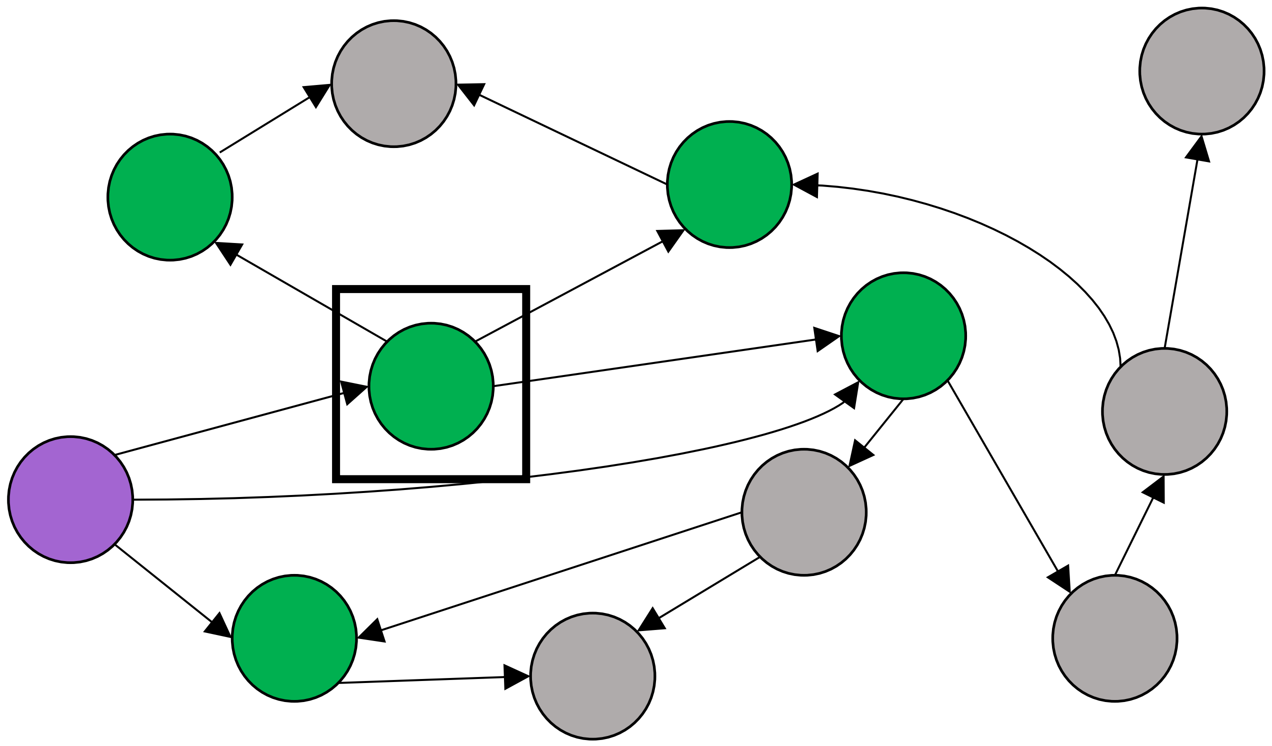
③ Move s to **Expanded**



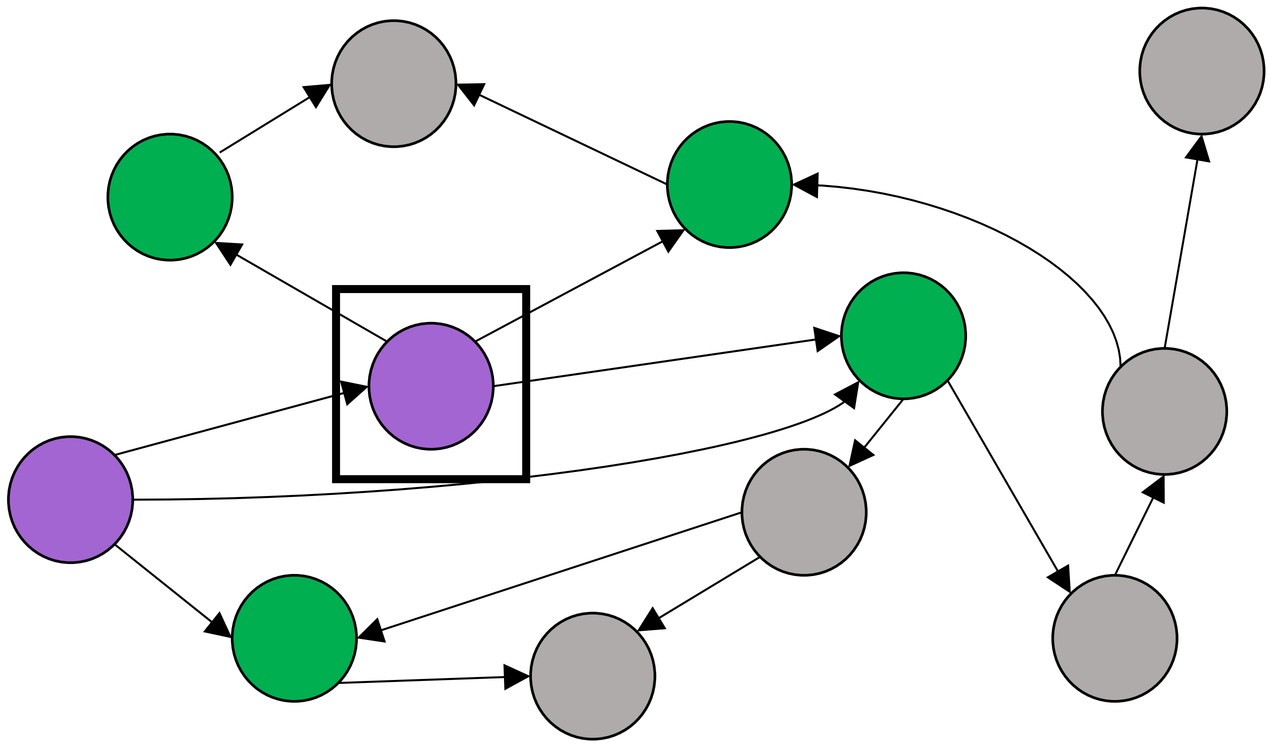
① Choose a node s from **Frontier**



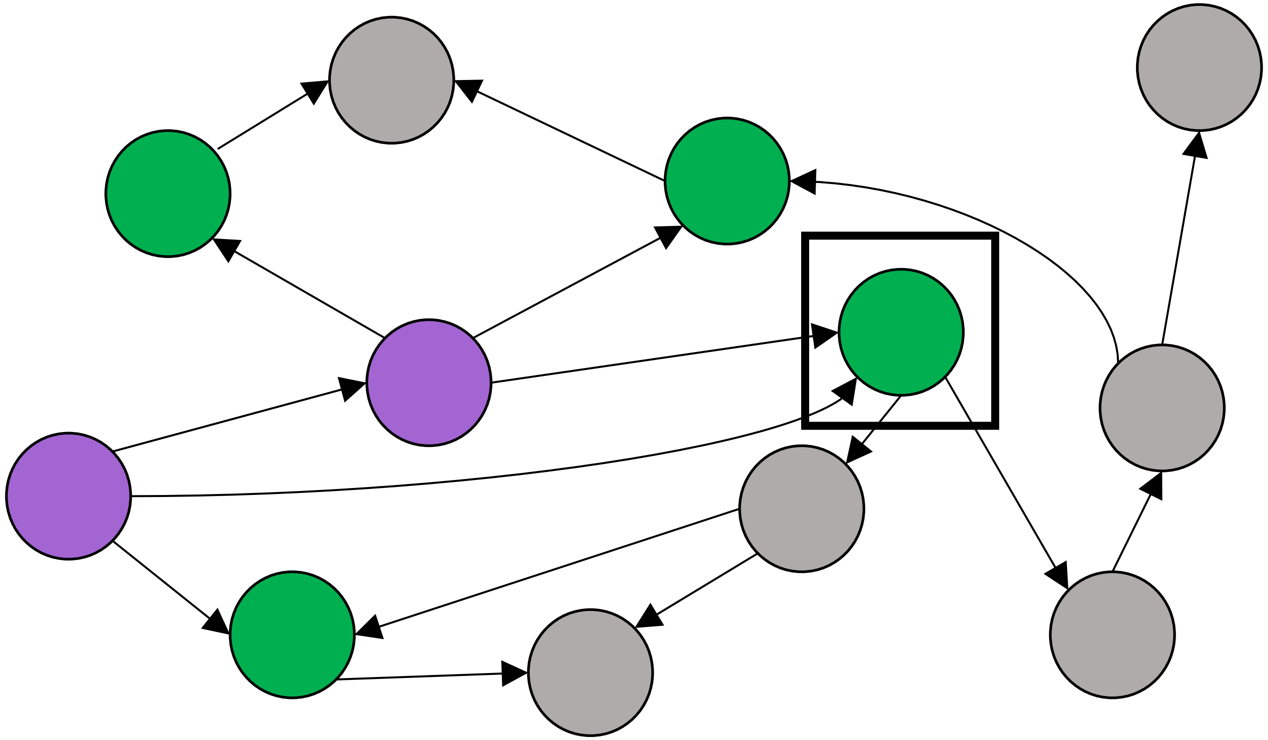
② Move all unreached successors of s to **Frontier**



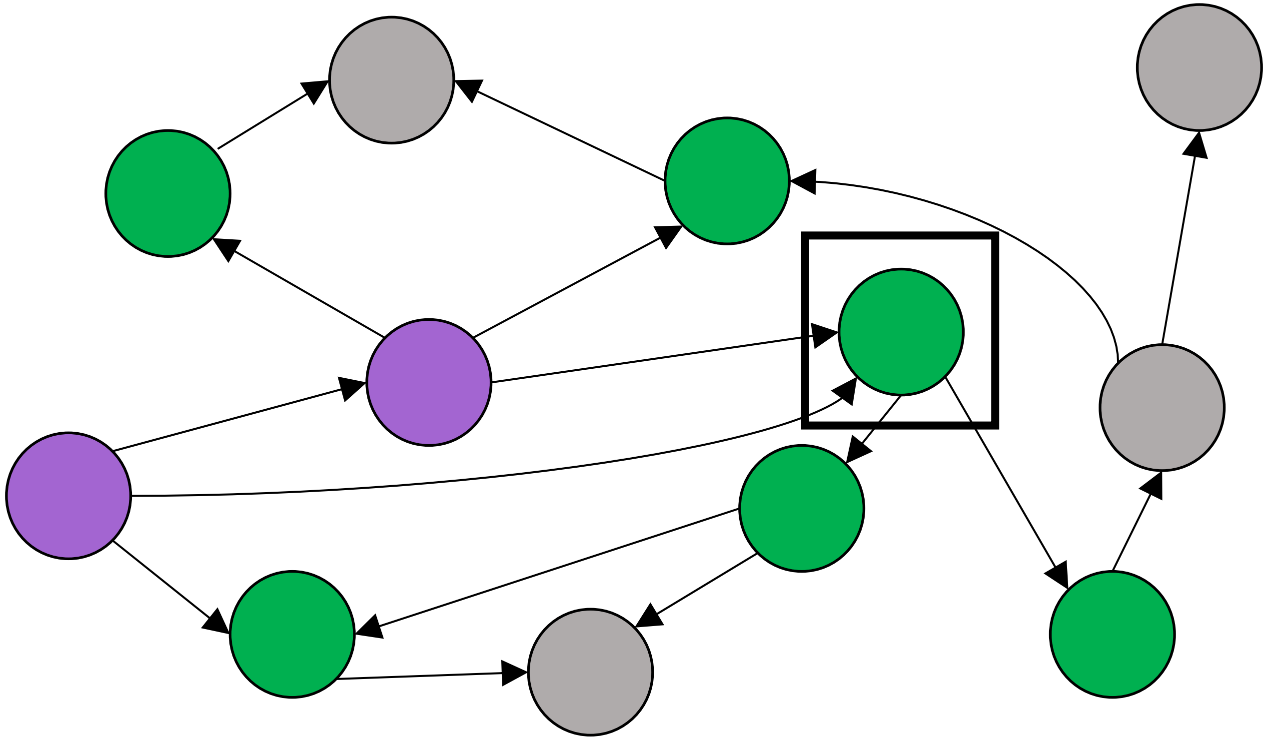
③ Move s to **Expanded**



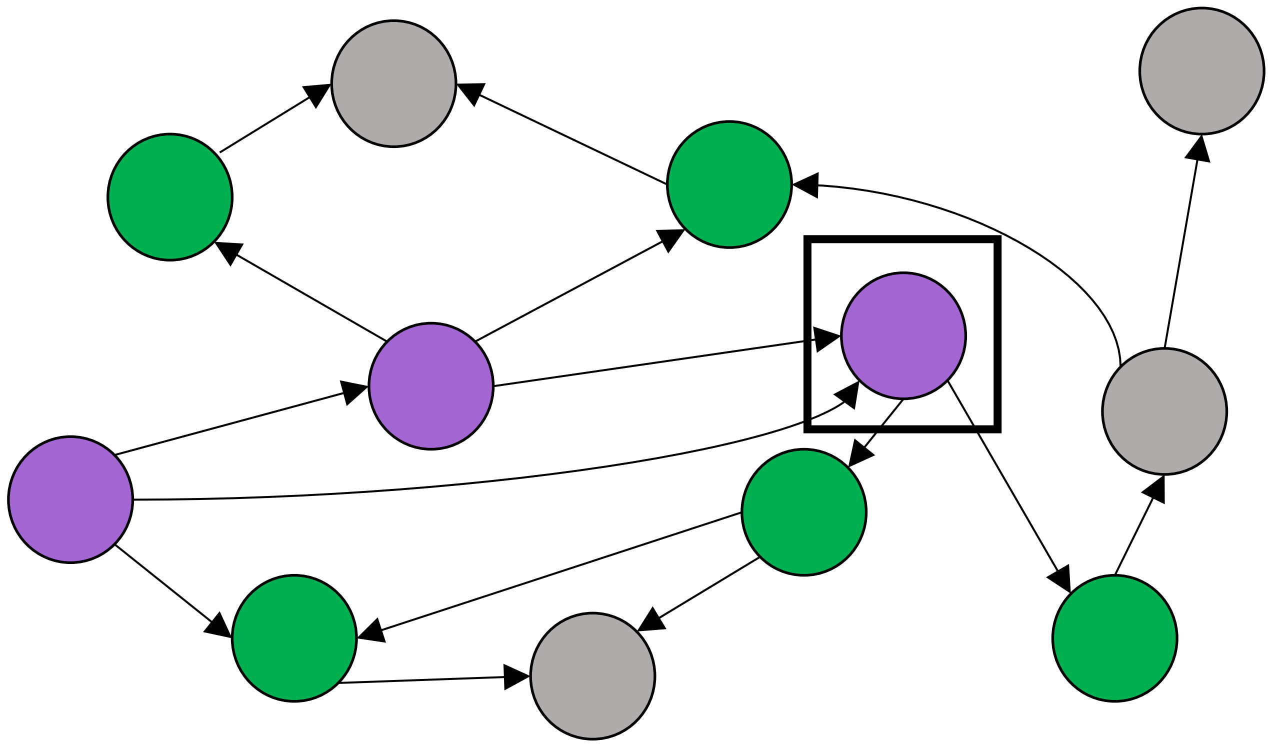
① Choose a node s from **Frontier**



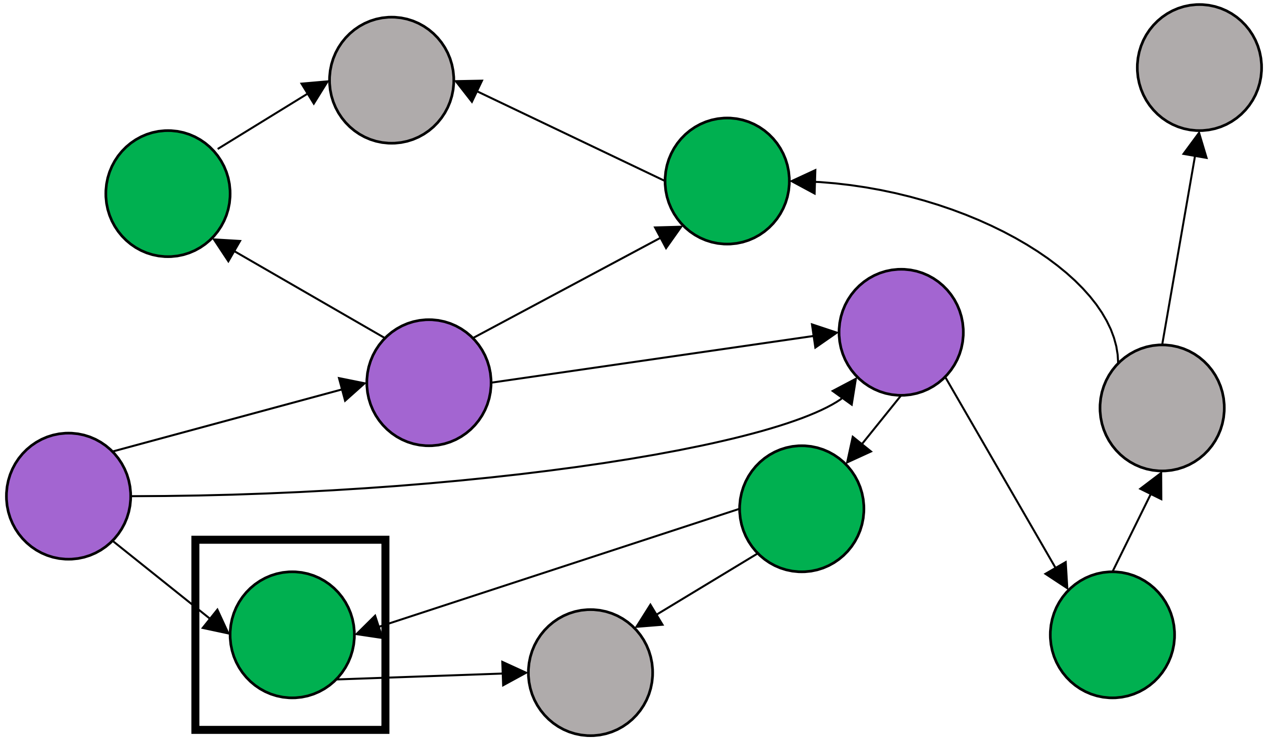
② Move all unreached successors of s to **Frontier**



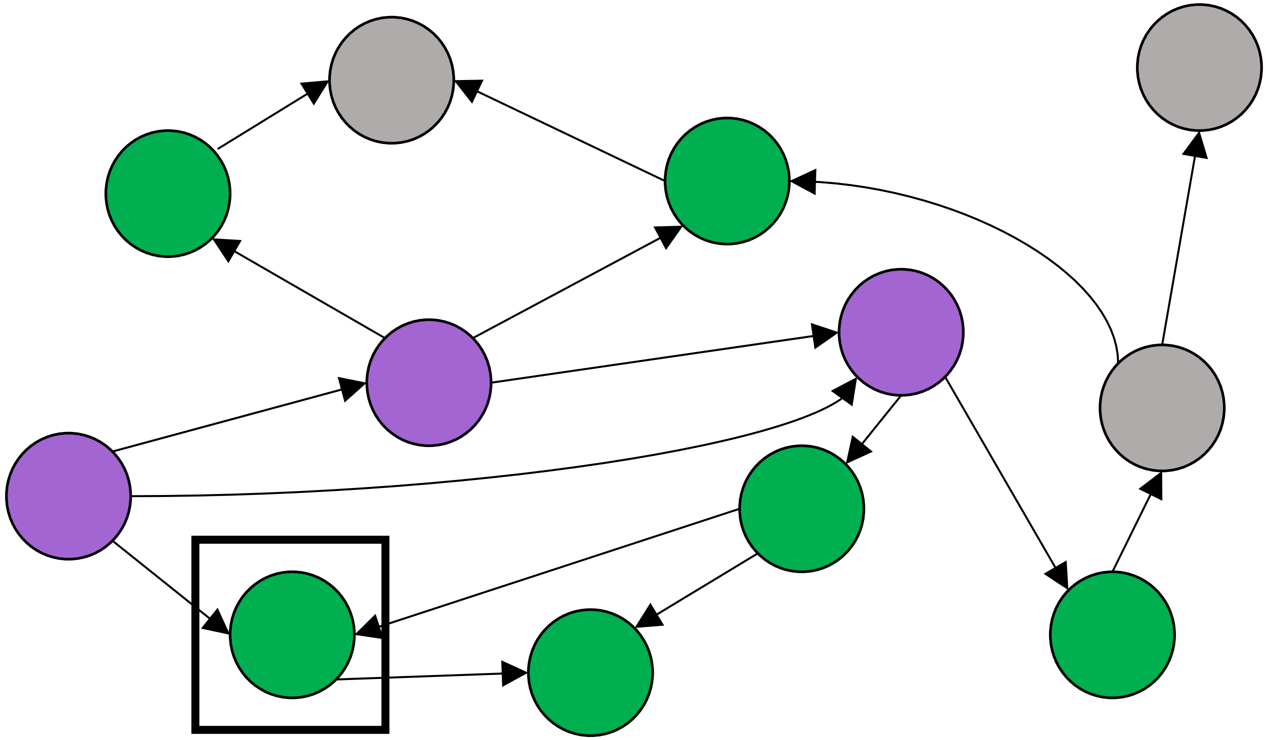
③ Move s to **Expanded**



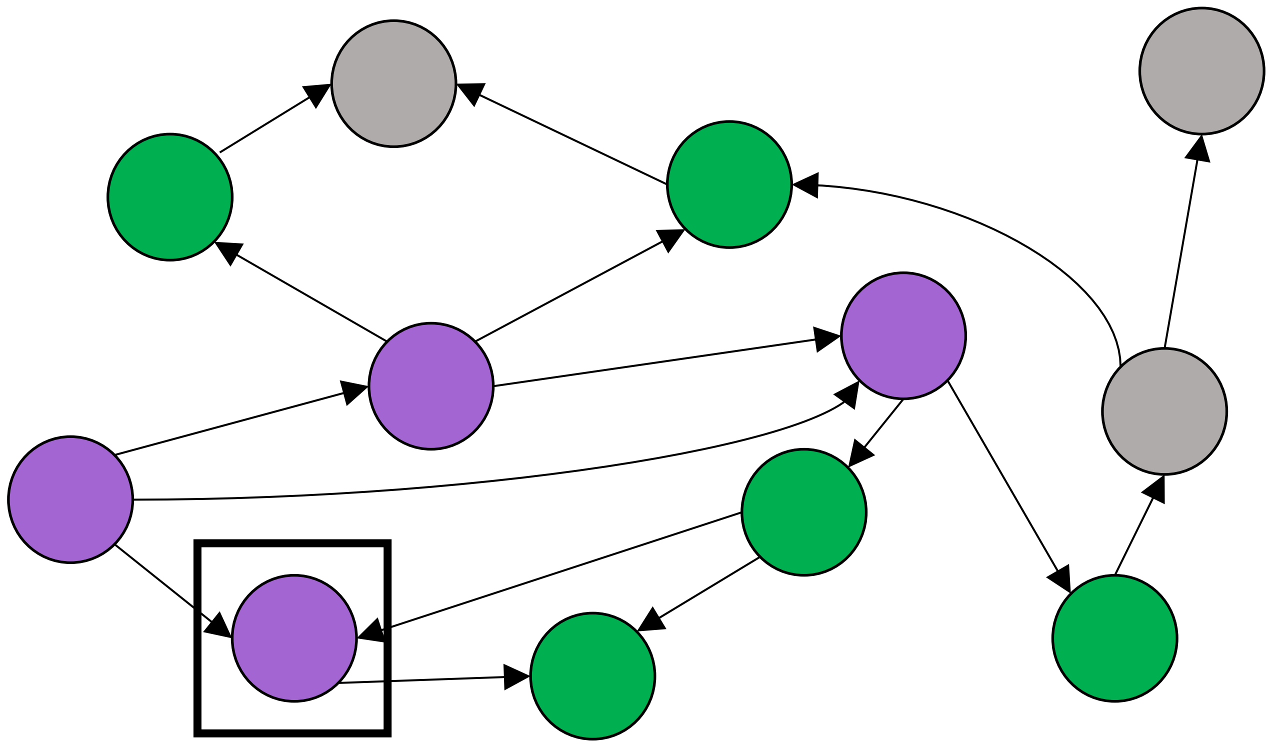
① Choose a node s from **Frontier**



② Move all unreached successors of s to **Frontier**



③ Move s to **Expanded**



Implementation

Choose and then remove it

Expanded $\leftarrow$ { }
Frontier $\leftarrow$ { initial_state }
While **Frontier** is not empty:
    Choose a node s from **Frontier**

    For all action a:
        $s' \leftarrow \mathrm{succ}(s, a)$
        If s' has not been reached:
            Put s' in **Frontier**

    Move s to **Expanded**

Frontier $\leftarrow$ { initial_state }
While **Frontier** is not empty:
    Pop a node s from **Frontier**

    For all action a:
        $s' \leftarrow \mathrm{succ}(s, a)$
        If not **Reached**$[s']$:
            Put s' in **Frontier**
            Reached$[s'] \leftarrow$ True

Termination When Goal is Encountered

Frontier $\leftarrow$ { initial_state }
While **Frontier** is not empty:
    Pop a node s from **Frontier**
    If s is a goal state, terminate

    For all action a:
        $s' \leftarrow \mathrm{succ}(s, a)$
        If not Reached[s']:
            Put s' in **Frontier**
            Reached[s'] $\leftarrow$ True

Late Goal Test

Frontier $\leftarrow$ { initial_state }
While **Frontier** is not empty:
    Pop a node s from **Frontier**

    For all action a:
        $s' \leftarrow \mathrm{succ}(s, a)$
        If not Reached[s']:
            If s' is a goal state, terminate
            Push s' to **Frontier**
            Reached[s'] $\leftarrow$ True

Early Goal Test

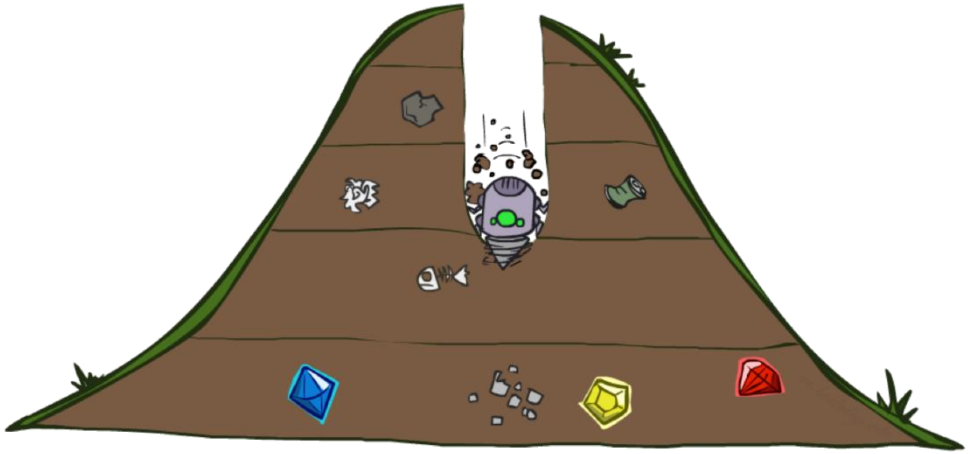
Termination When Goal is Encountered

- Early Goal Test allows quicker termination when a goal is found.
  - Breadth First Search
  - Depth First Search

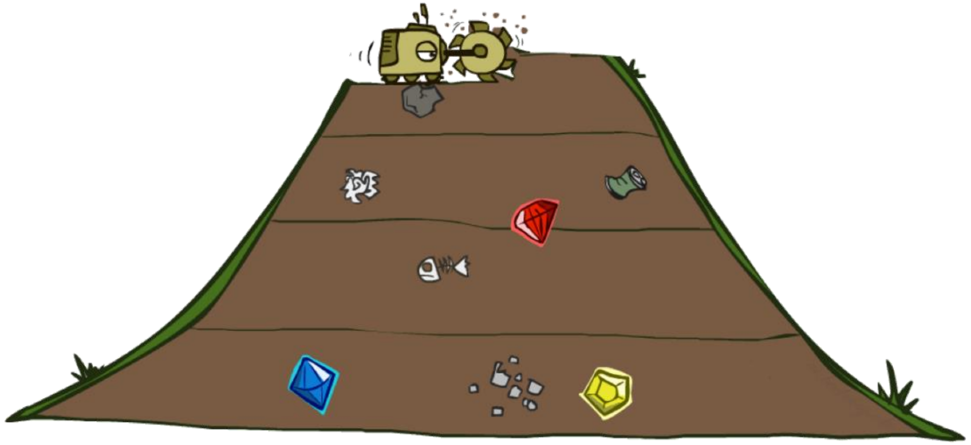
- However, when actions are associated with costs and we want to find a **minimum cost solution** (i.e., cost-sensitive), we may have to use the Late Goal Test.
  - Uniform Cost Search (Dijkstra Algorithm)

Uninformed Search

DFS and BFS



Depth First Search



Breadth First Search

DFS and BFS

Frontier ← { initial_state }
While **Frontier** is not empty:
    Pop a node s from **Frontier**   differ here

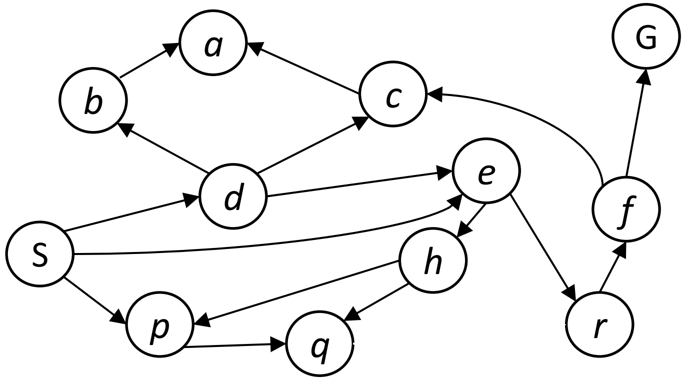
    For all action a:
        $s' \leftarrow \mathrm{succ}(s, a)$
        If not Reached[s']:
            If s' is a goal state, terminate
            Push s' to **Frontier**
            Reached[s'] ← True

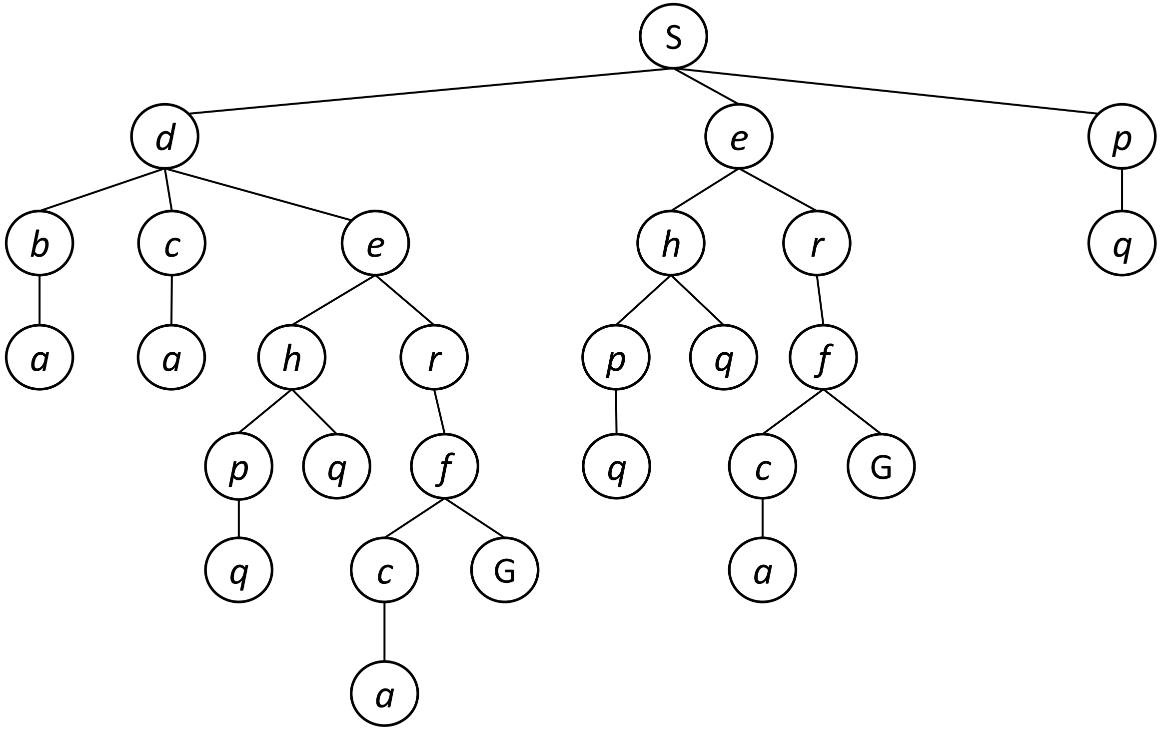
DFS

Frontier ← { initial_state }
While **Frontier** is not empty:
    Pop the **newest** node s from **Frontier**

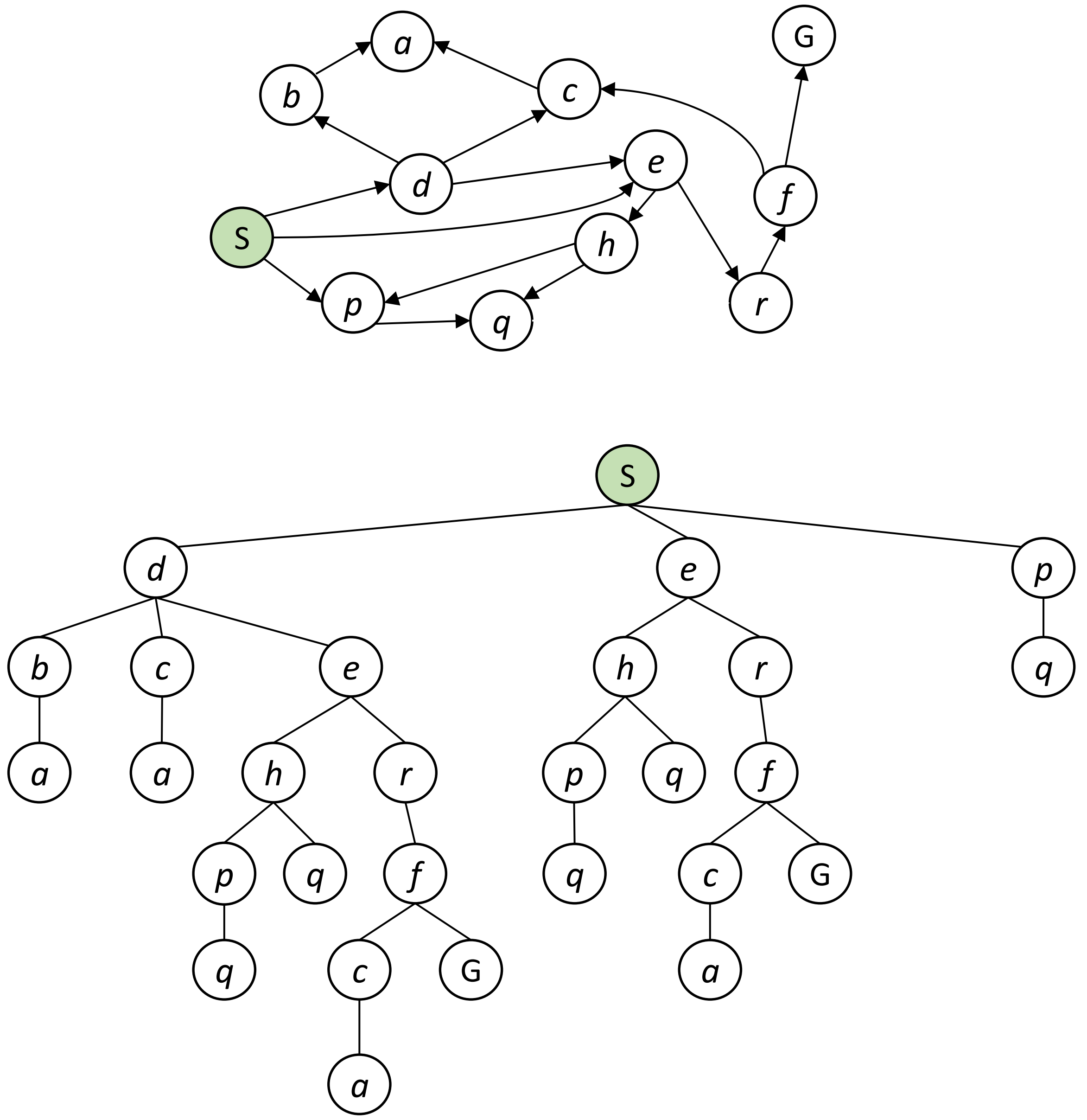
    For all action a:
        $s' \leftarrow \mathrm{succ}(s, a)$
        If not Reached[s']:
            If s' is a goal state, terminate
            Push s' to **Frontier**
            Reached[s'] ← True

DFS

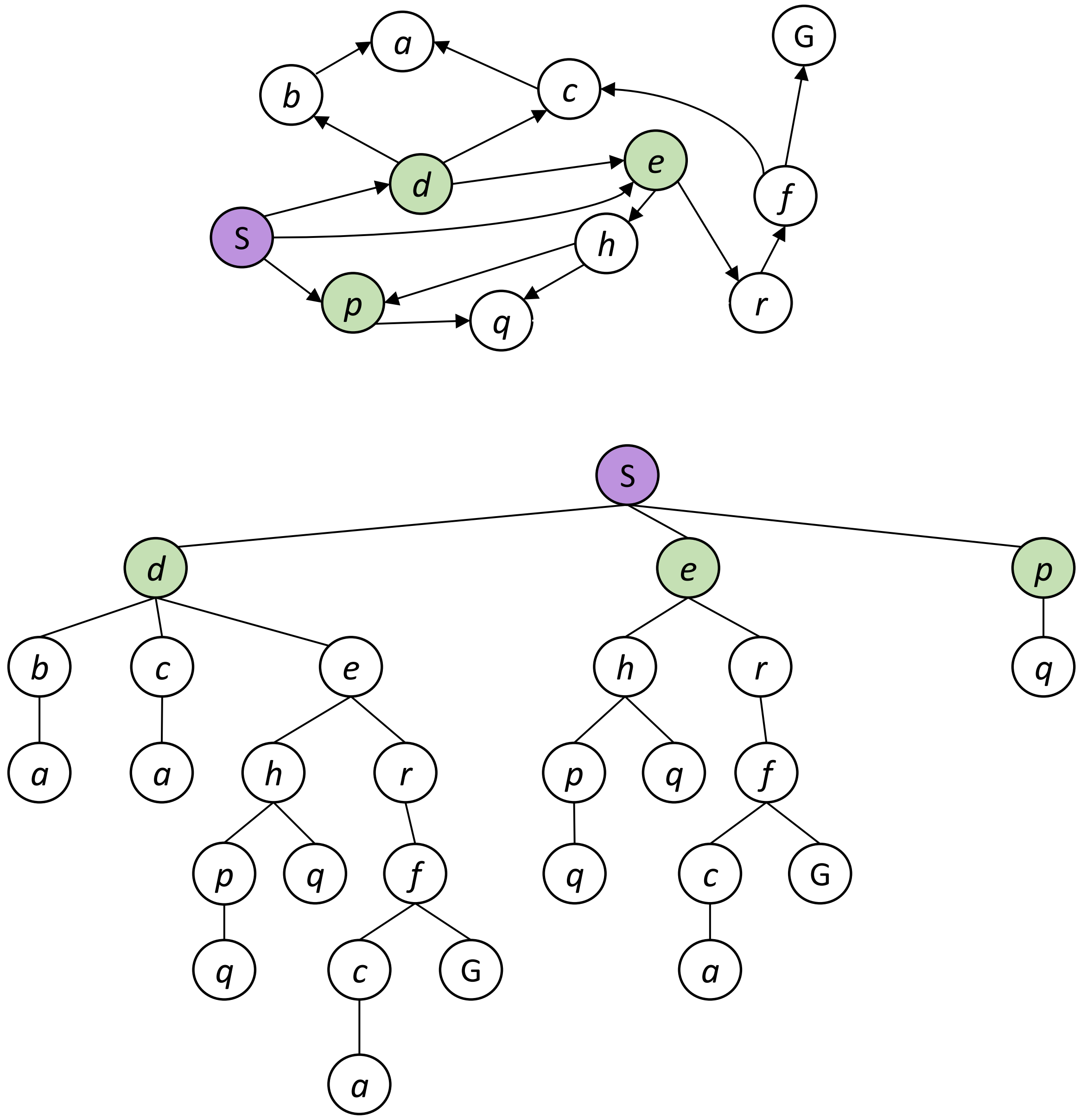




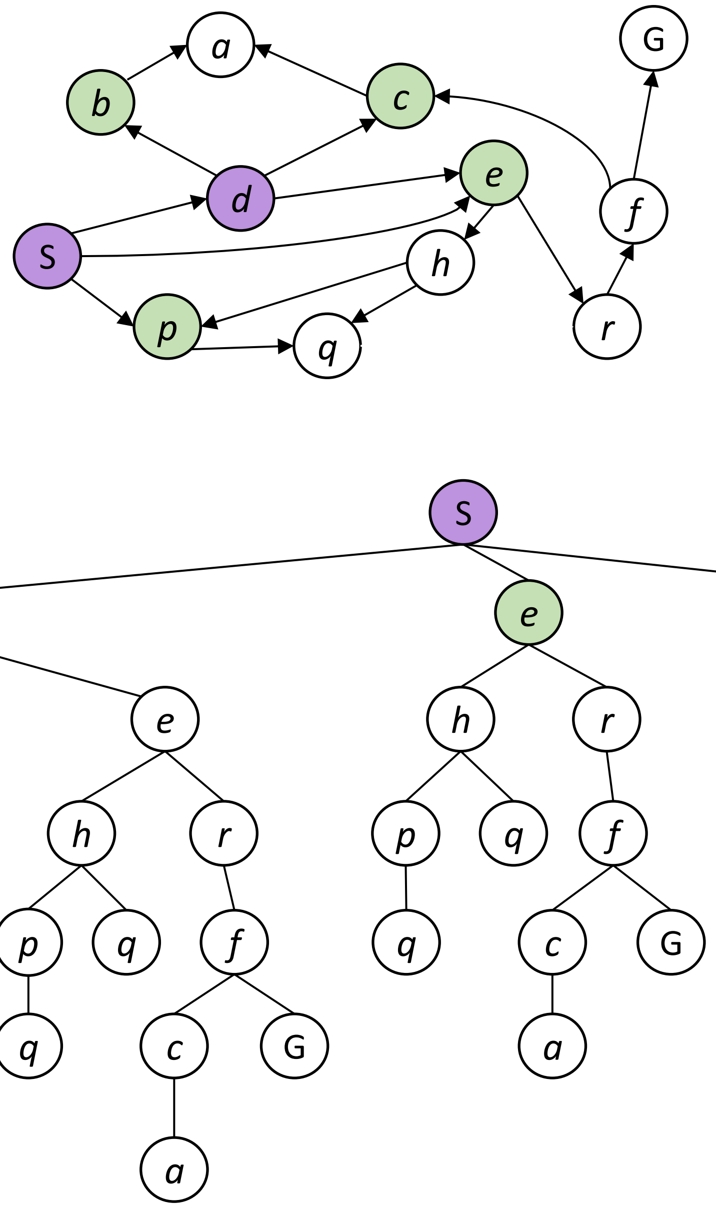
DFS



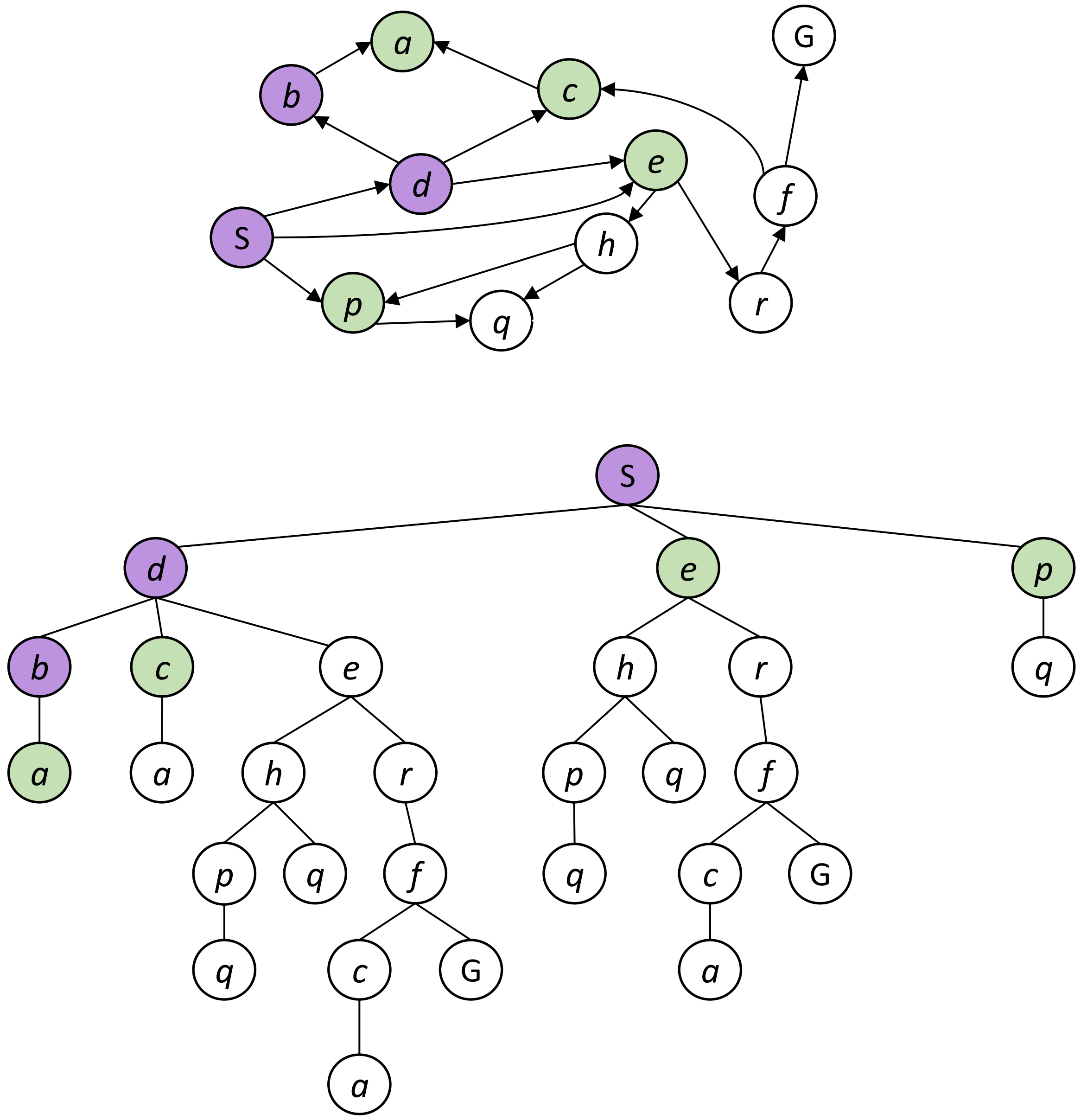
DFS



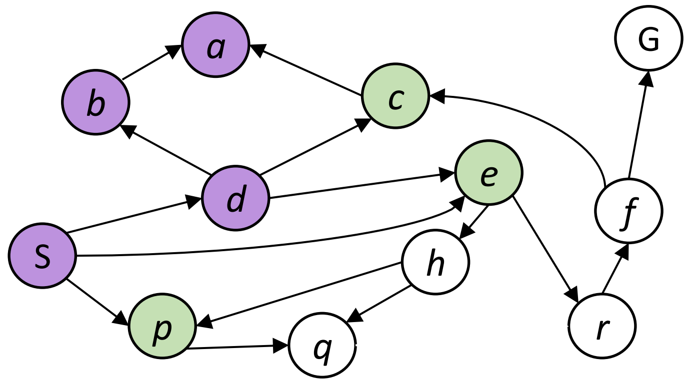
DFS

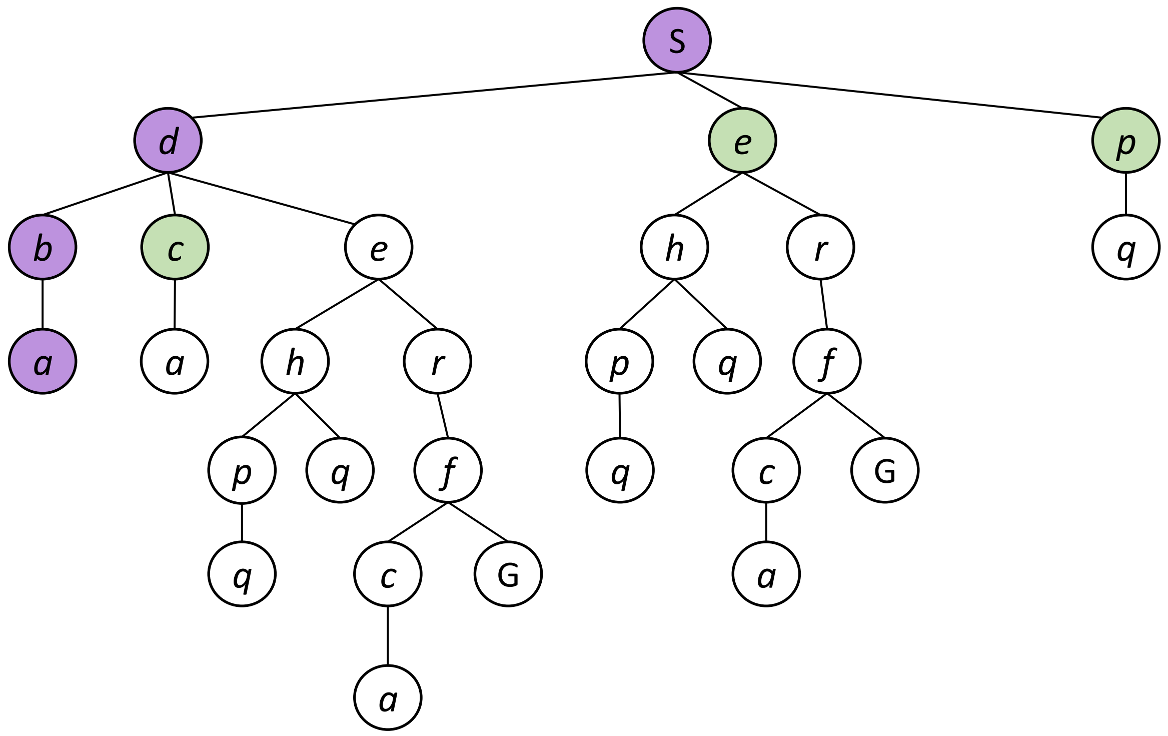


DFS

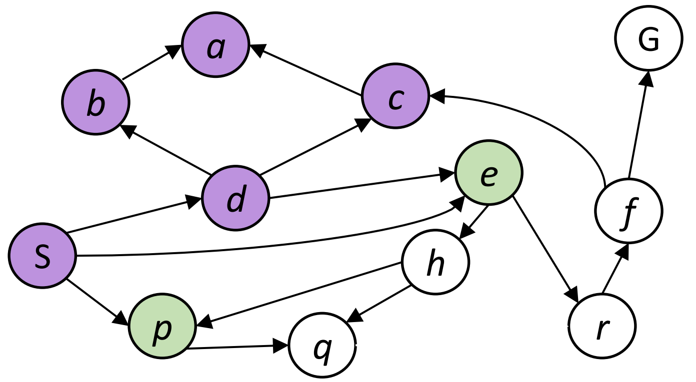


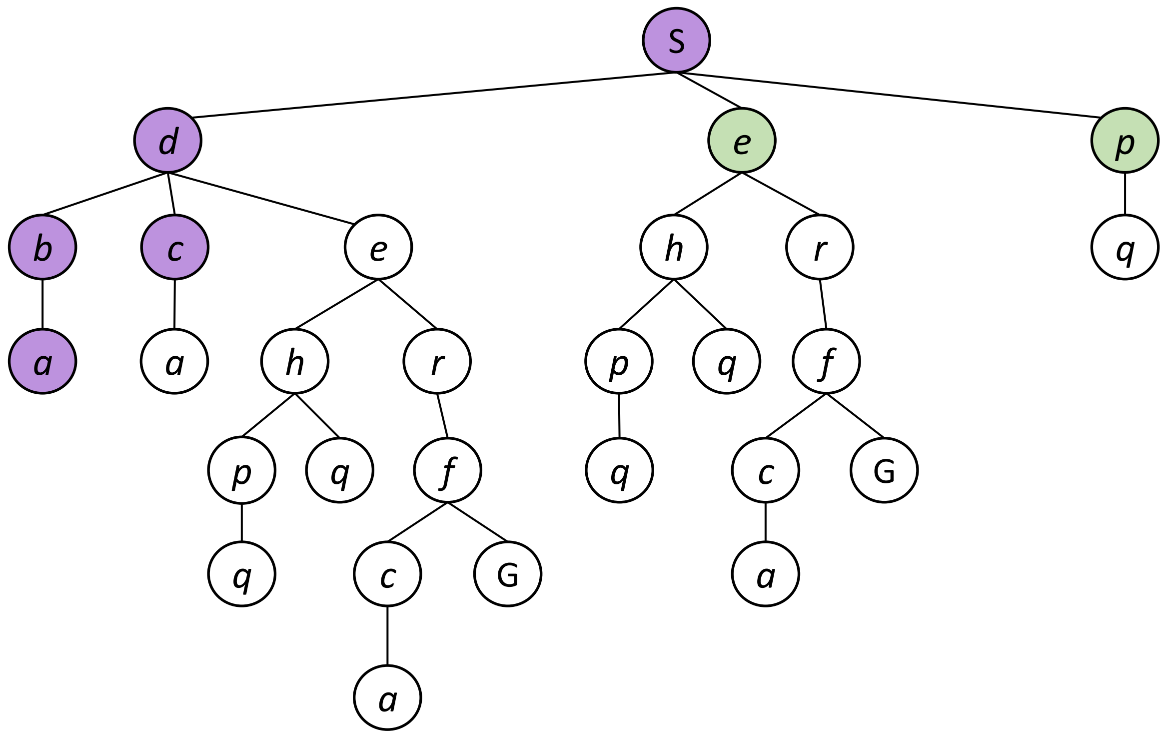
DFS



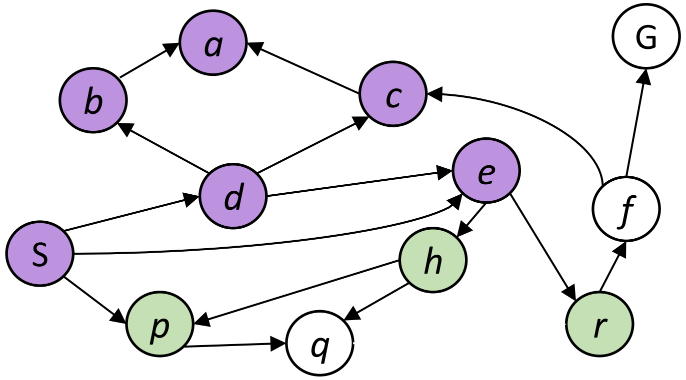


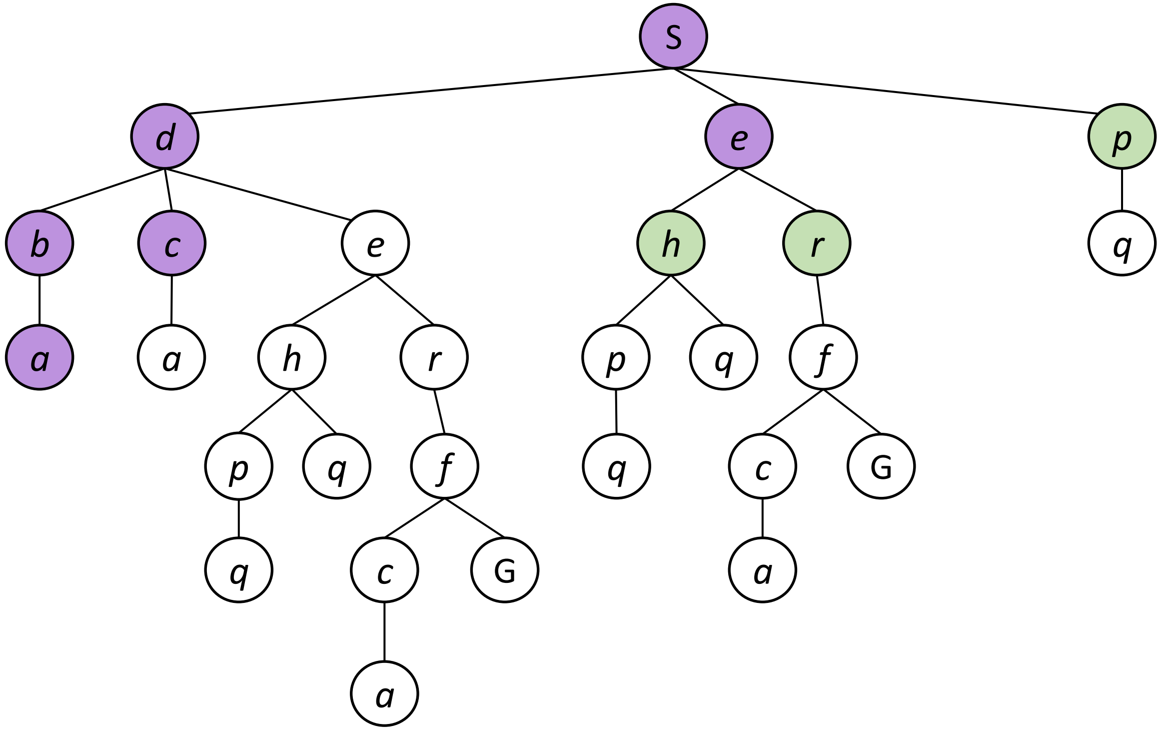
DFS



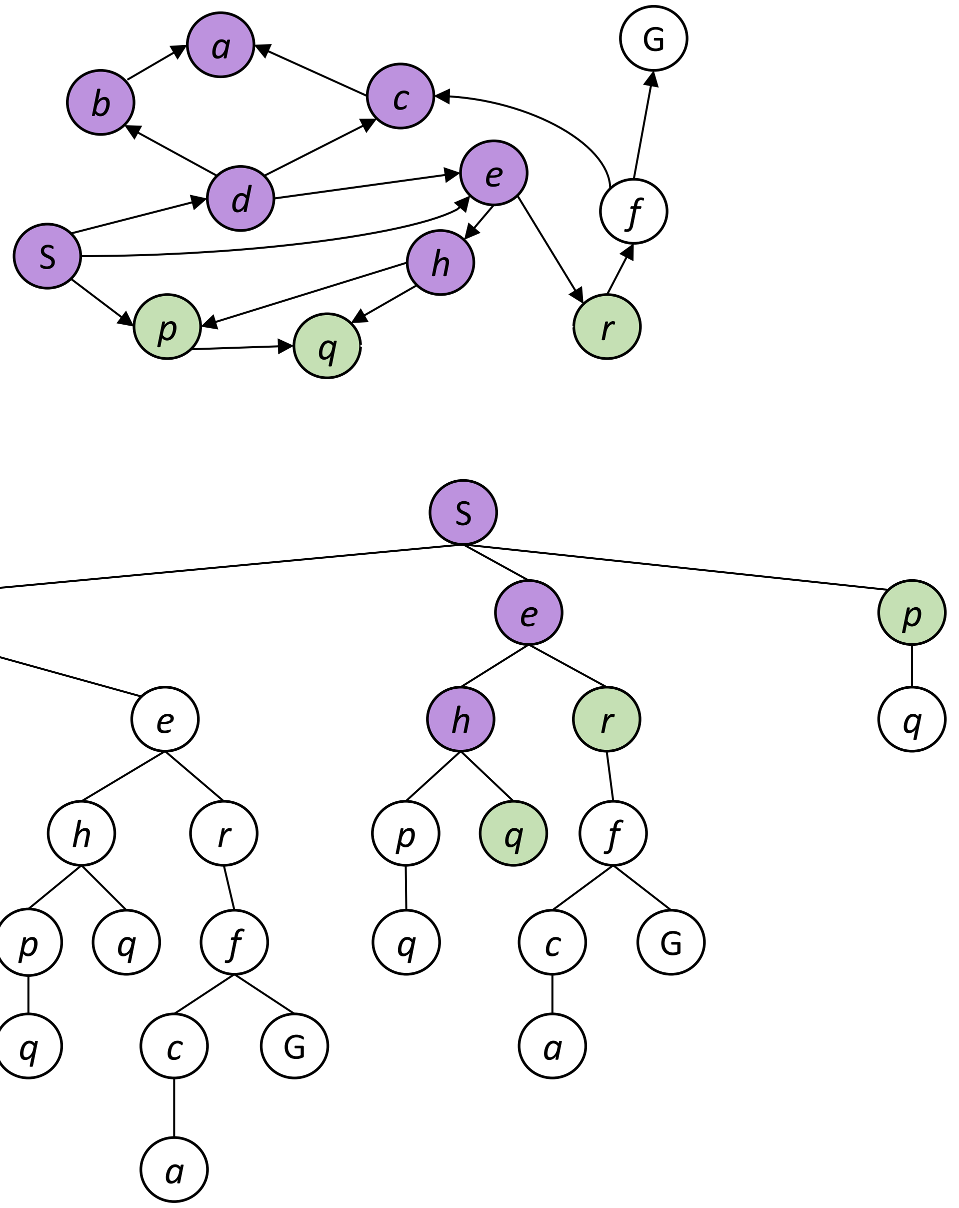


DFS

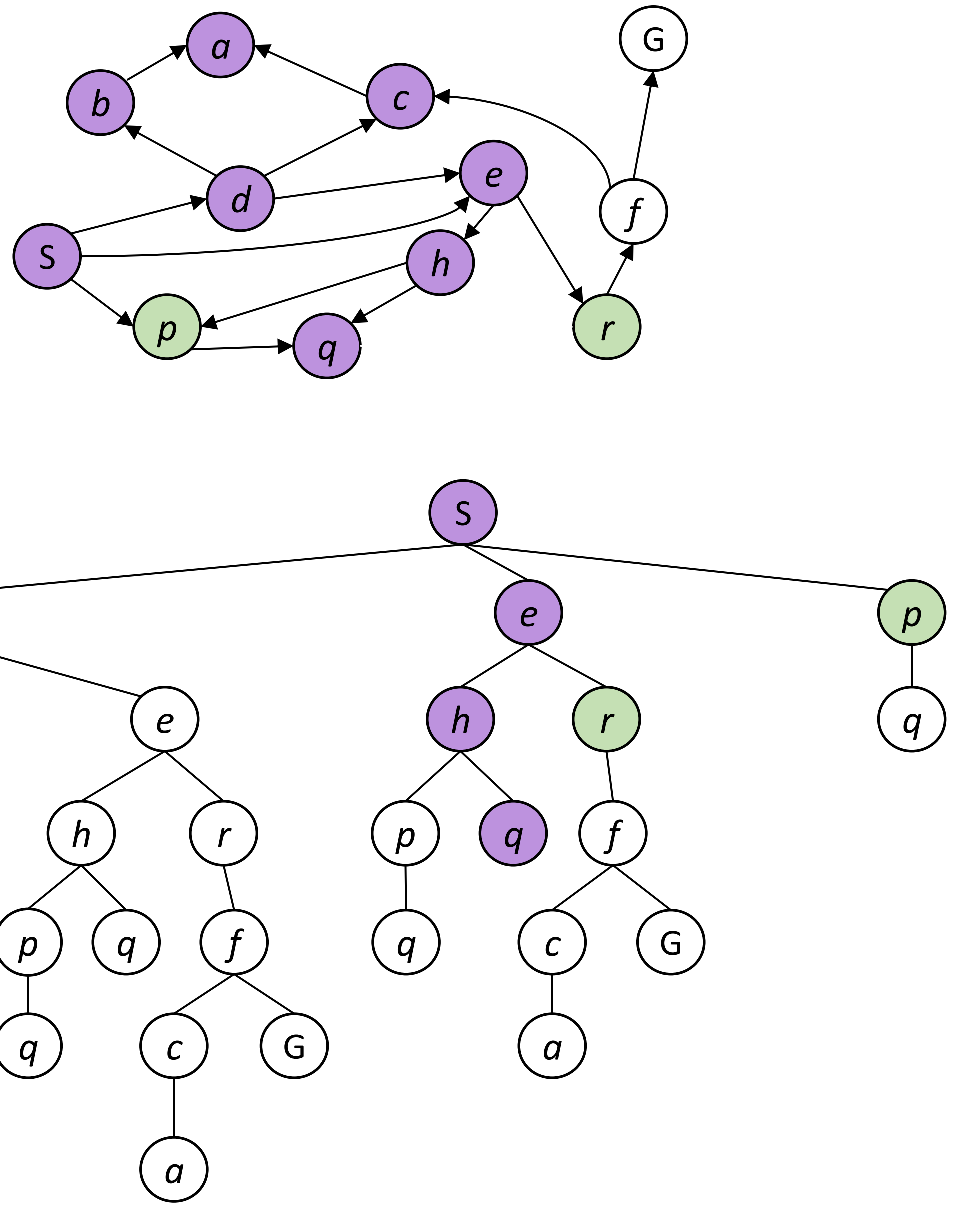




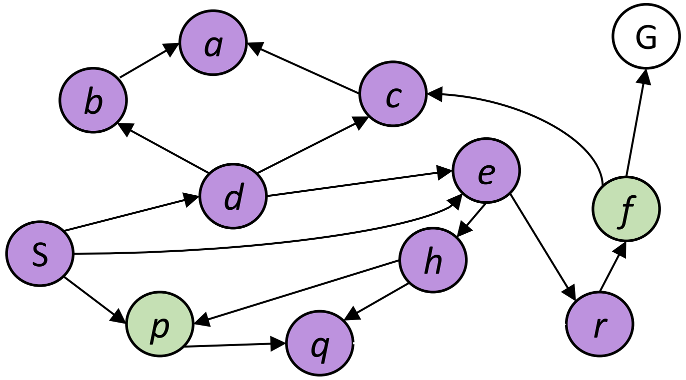
DFS

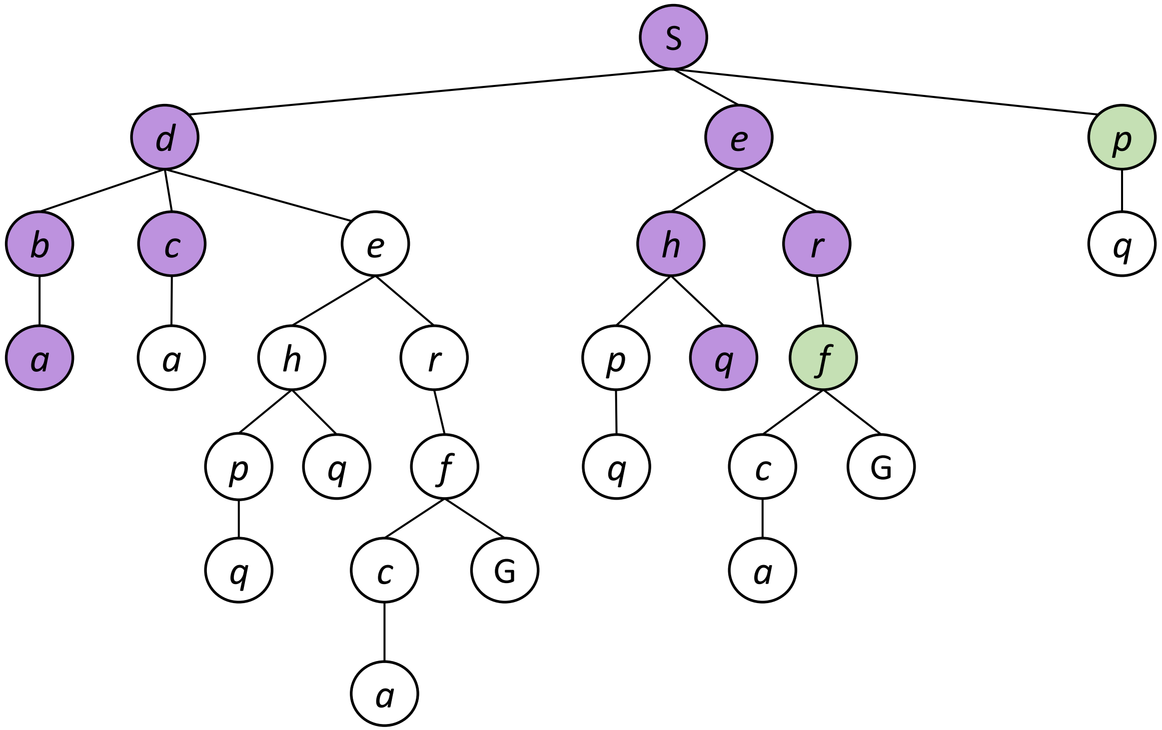


DFS

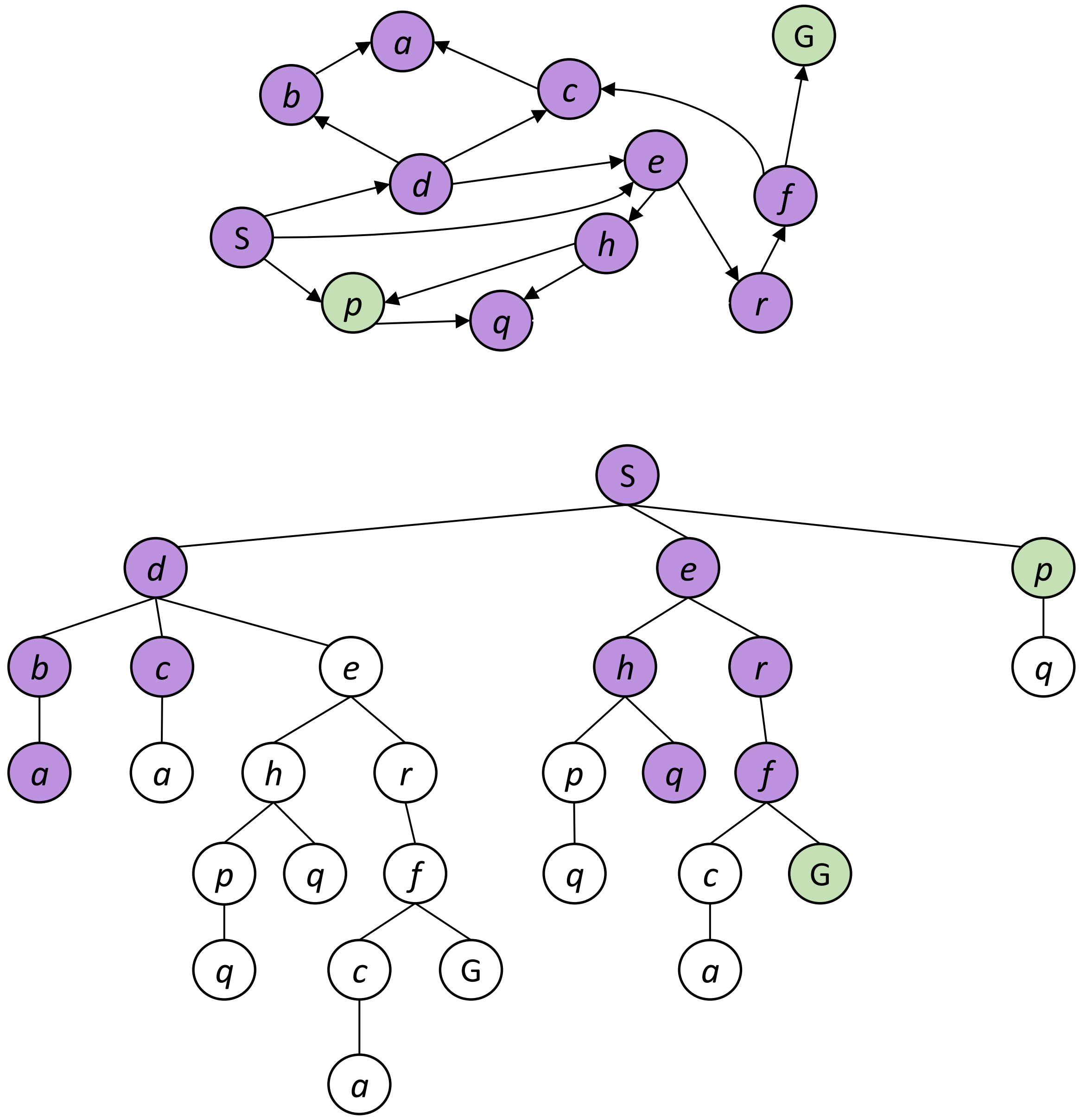


DFS





DFS

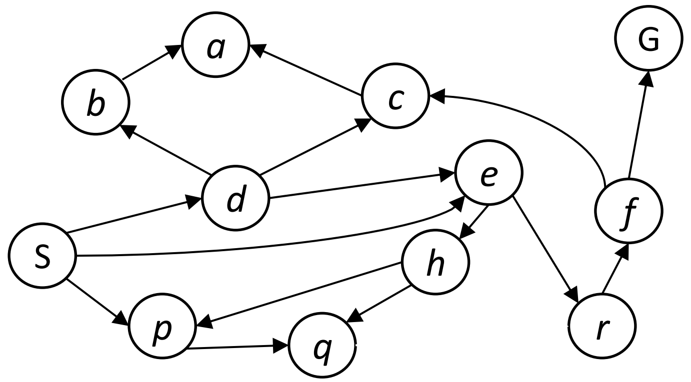


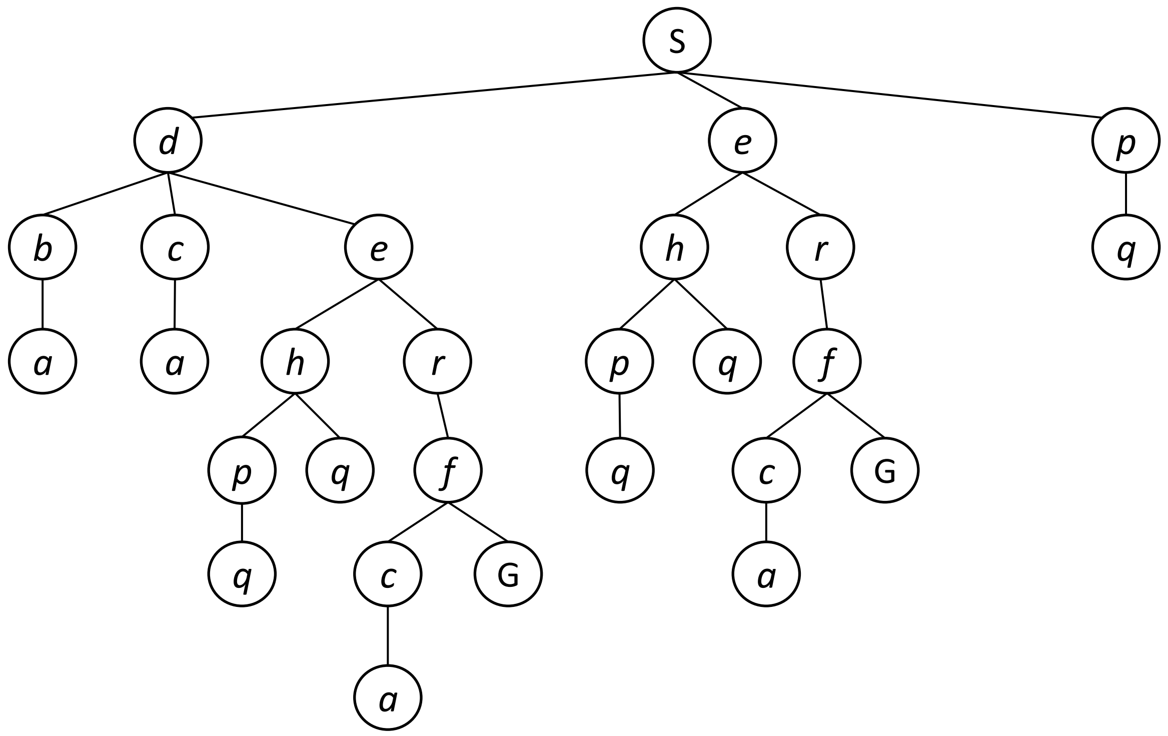
BFS

Frontier ← { initial_state }
While **Frontier** is not empty:
    Pop the **oldest** node s from **Frontier**

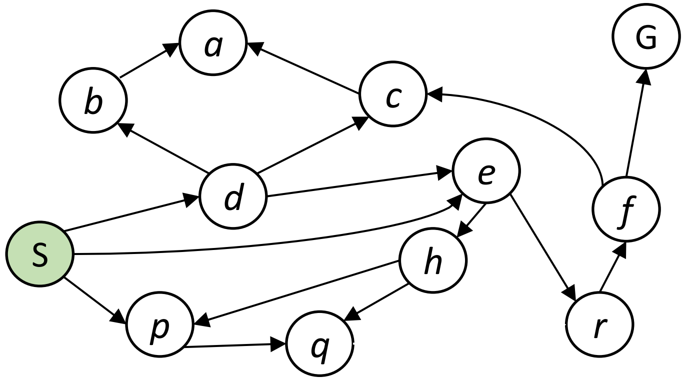
    For all action a:
        $s' \leftarrow \mathrm{succ}(s, a)$
        If not Reached[s']:
            If s' is a goal state, terminate
            Push s' to **Frontier**
            Reached[s'] ← True

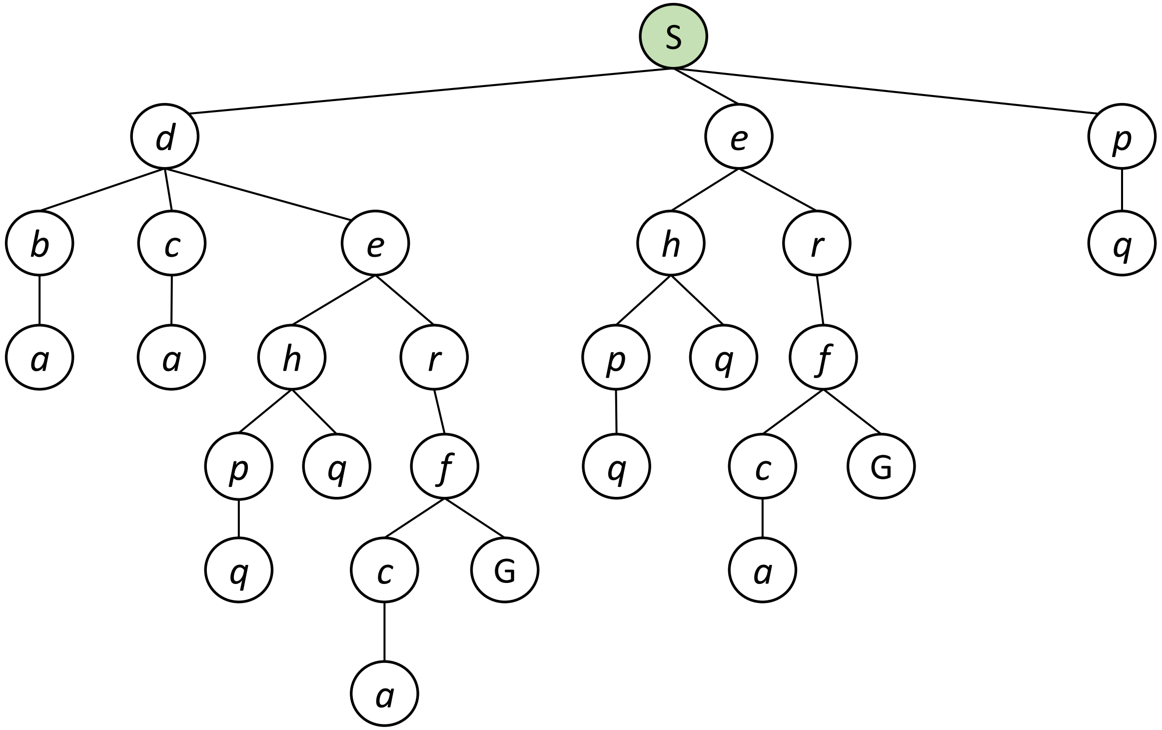
BFS



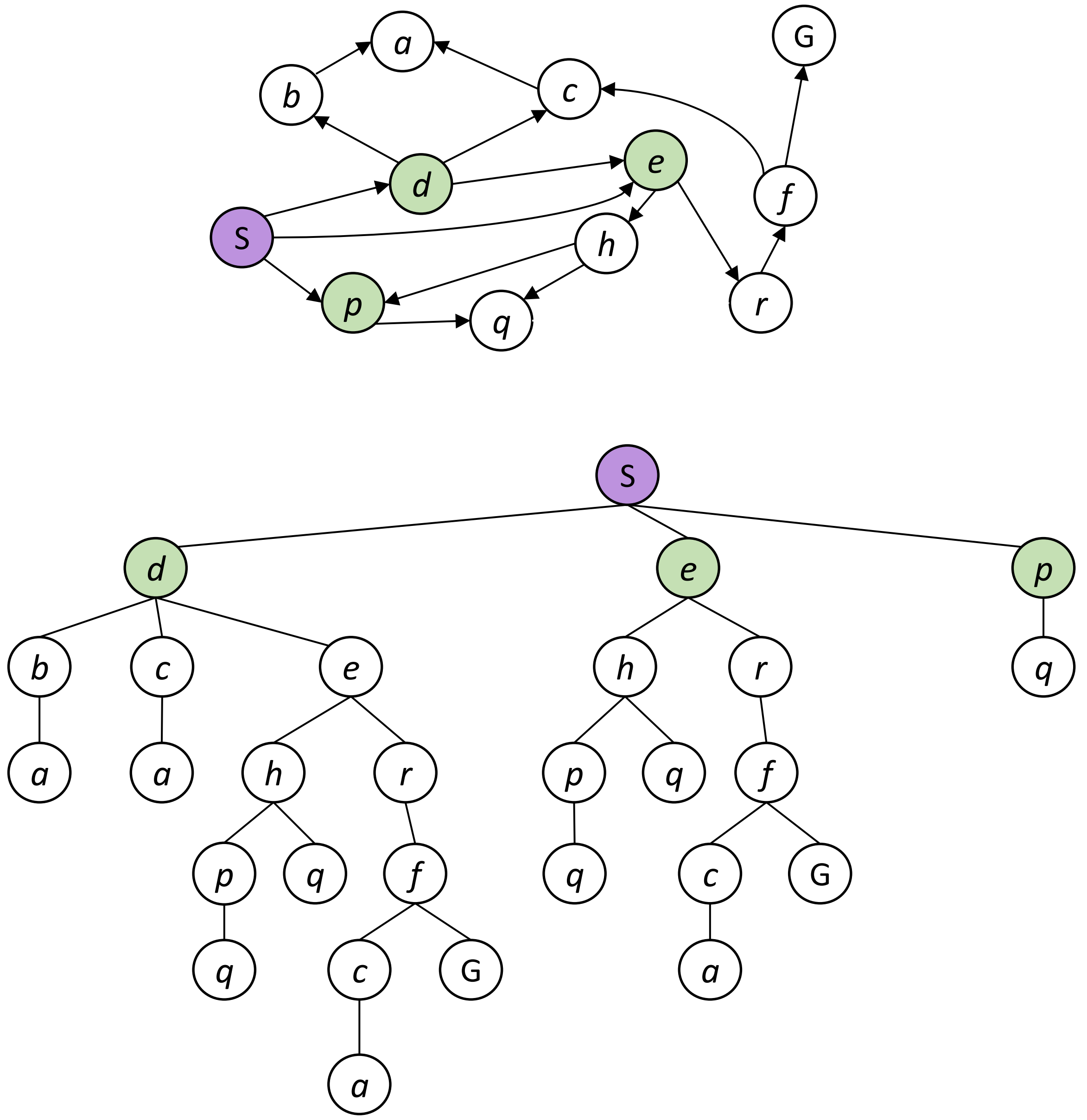


BFS

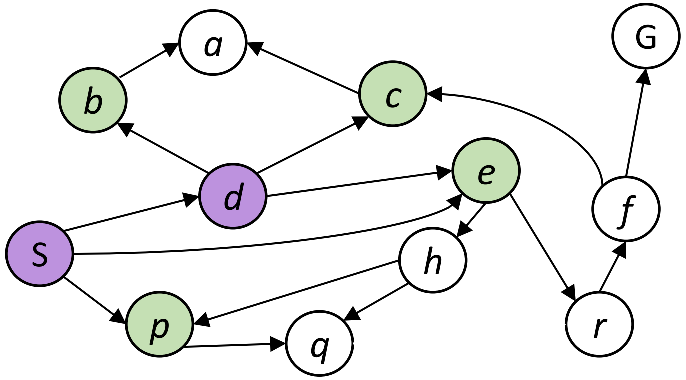


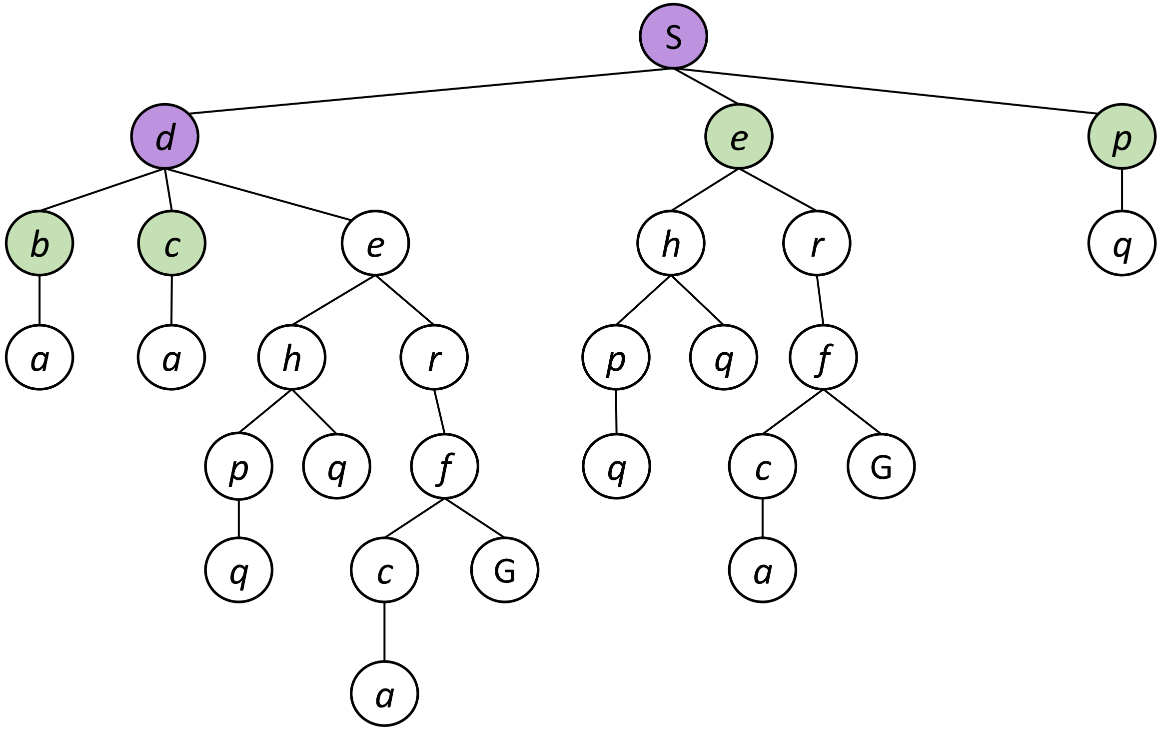


BFS

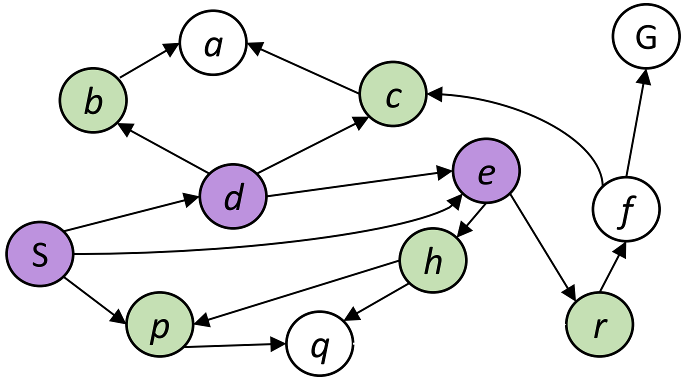


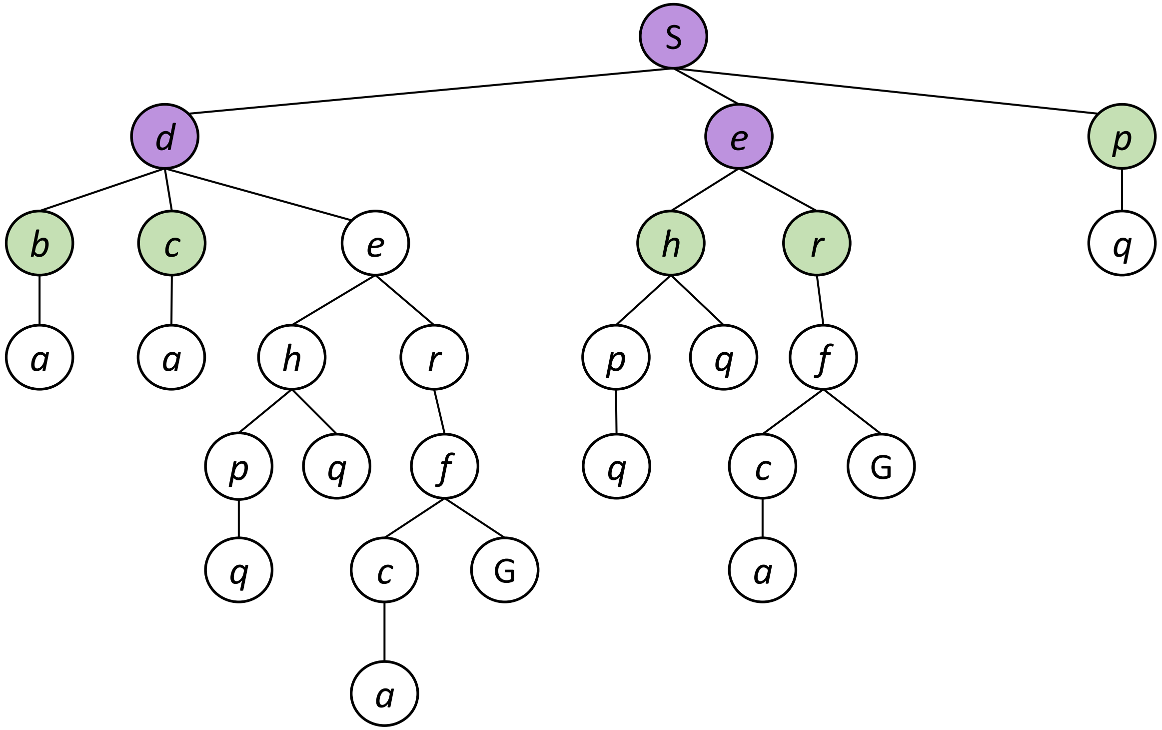
BFS



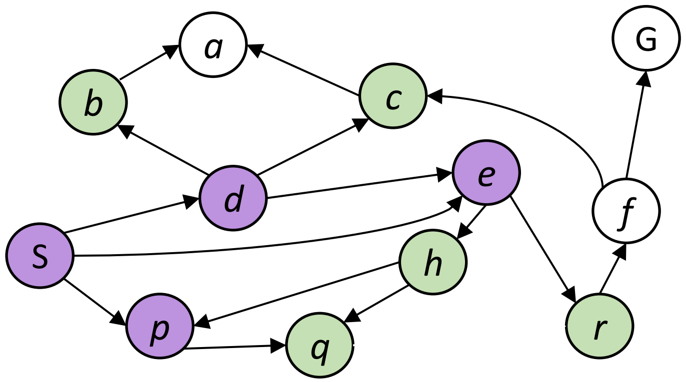


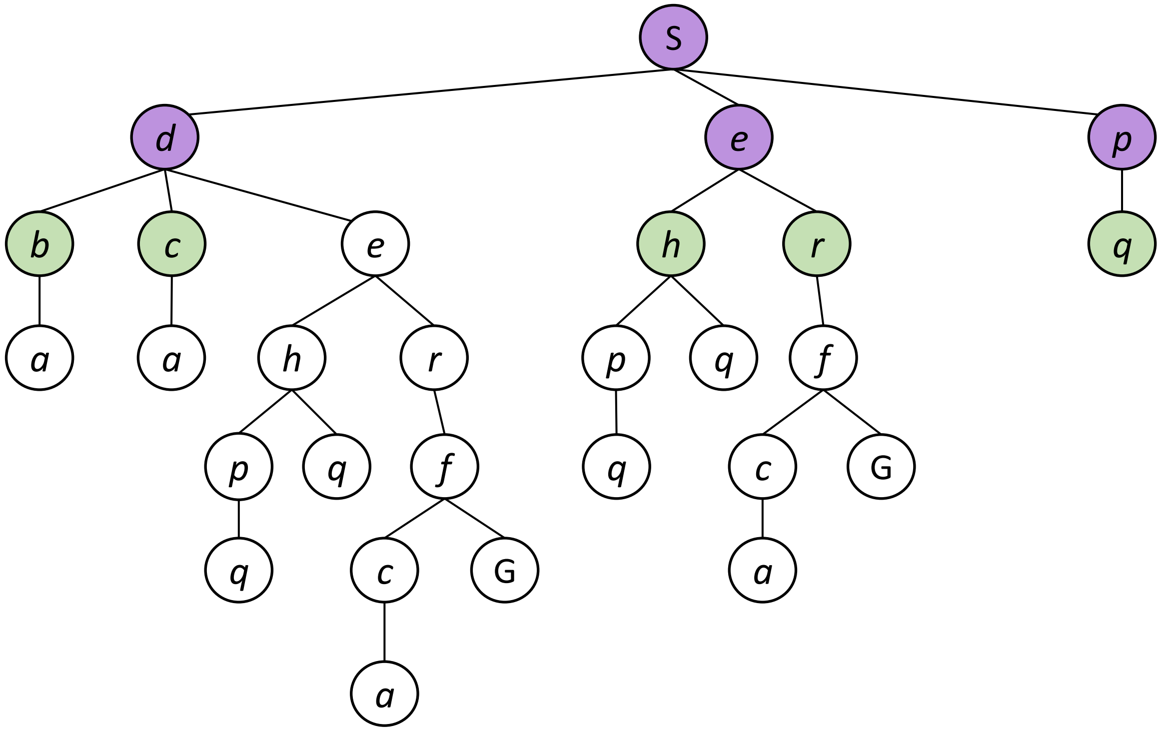
BFS



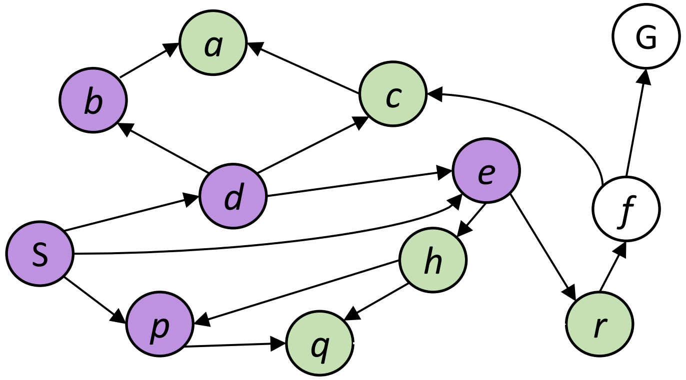


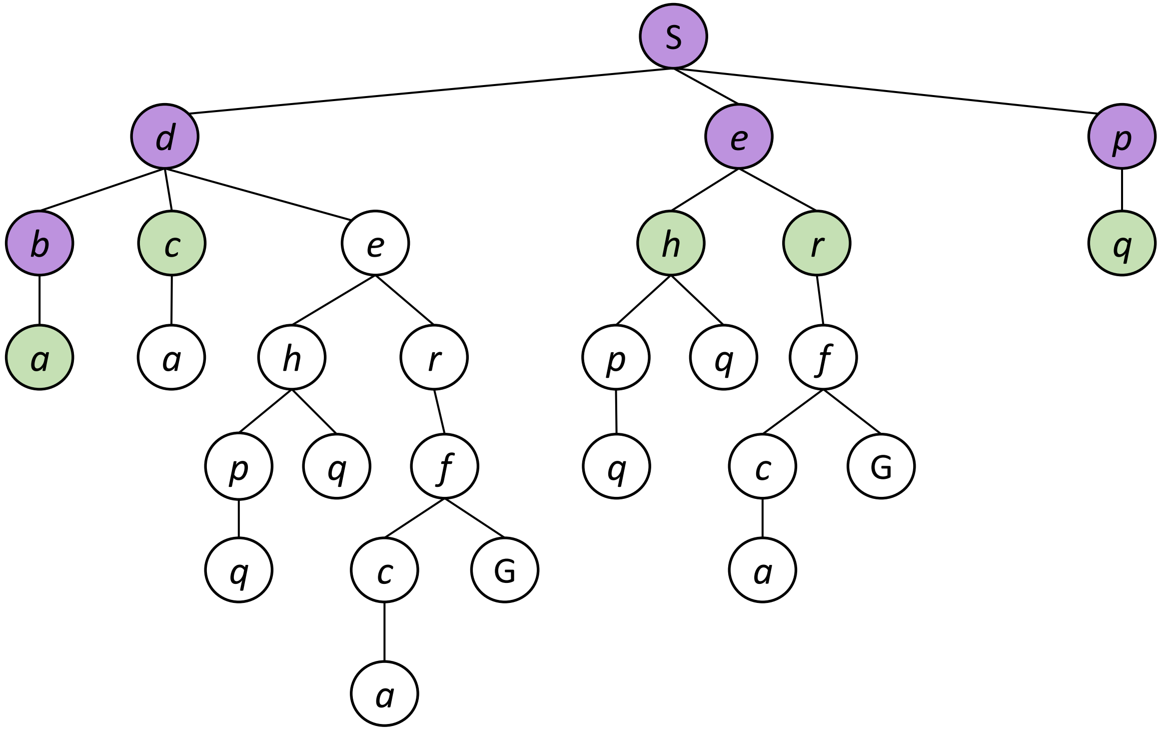
BFS



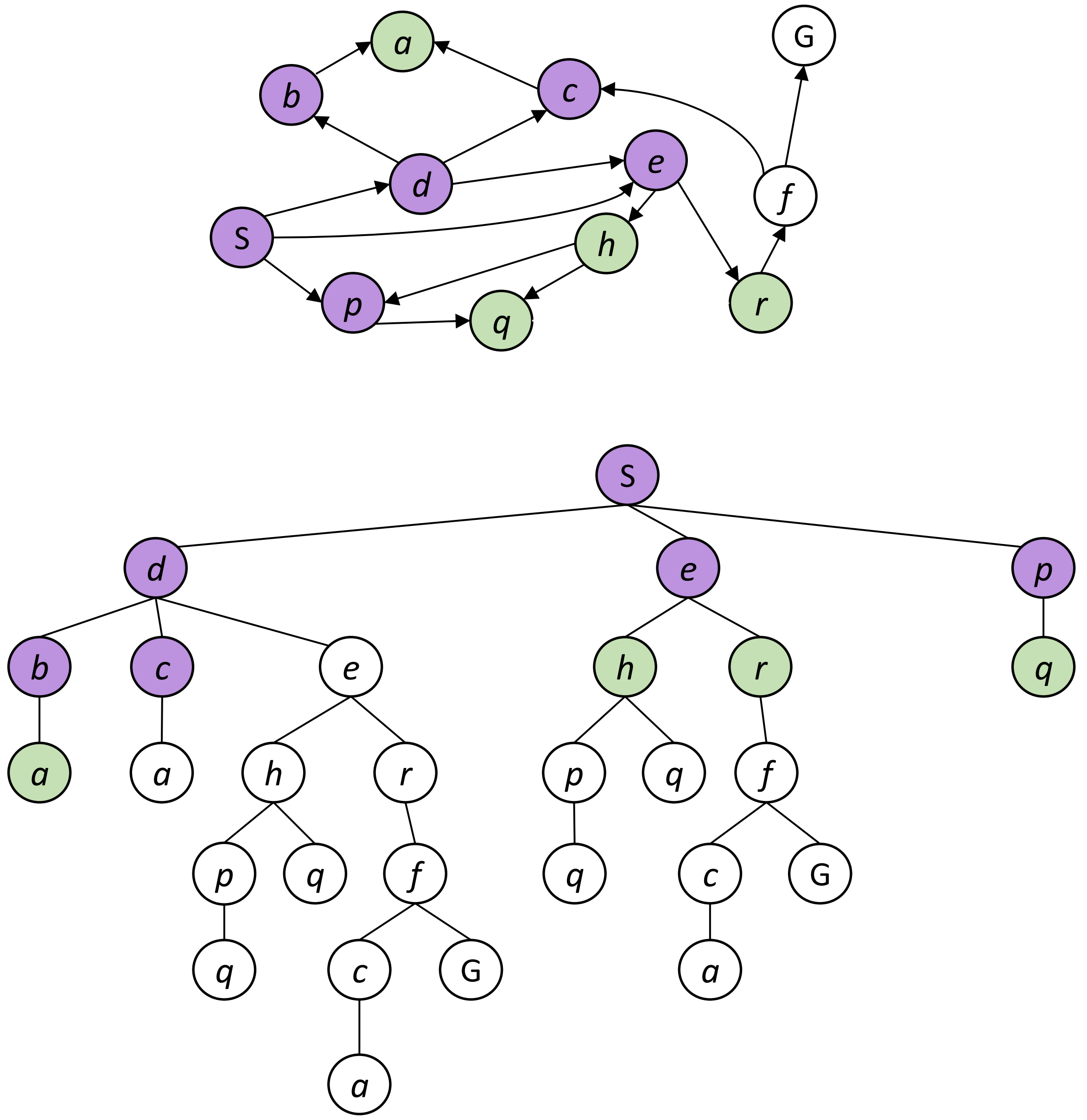


BFS

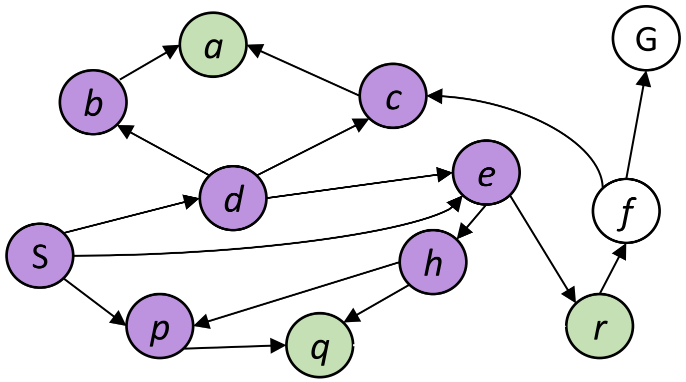


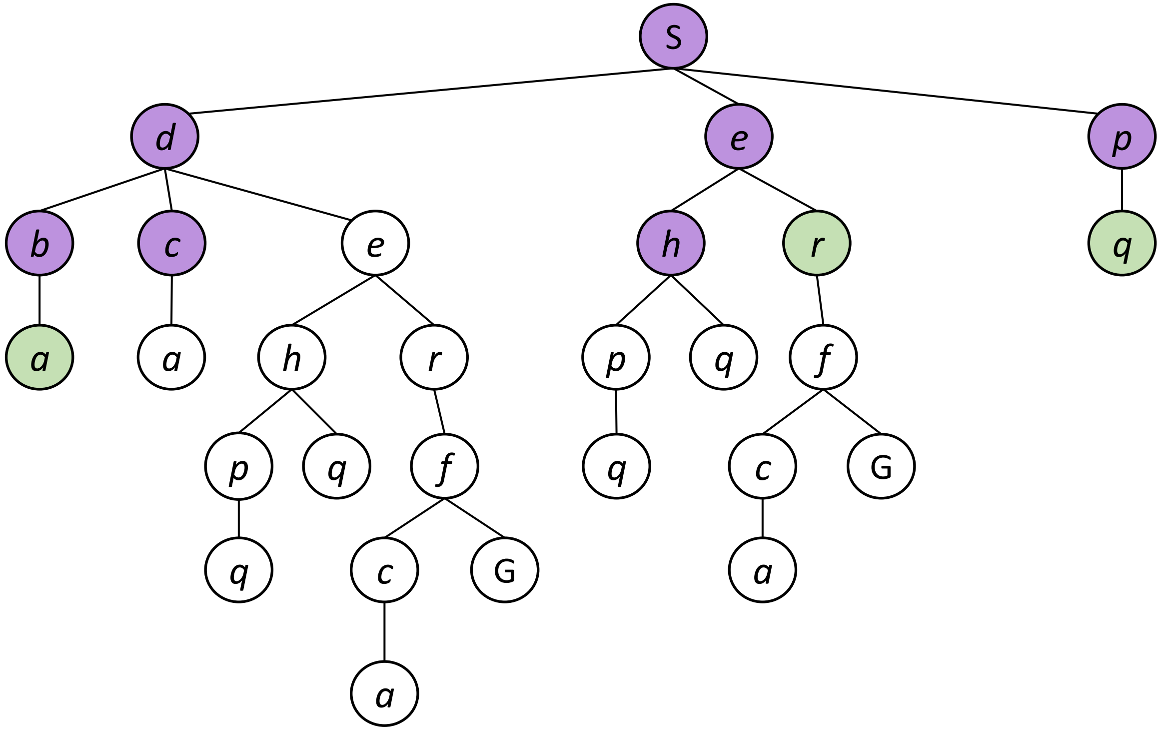


BFS

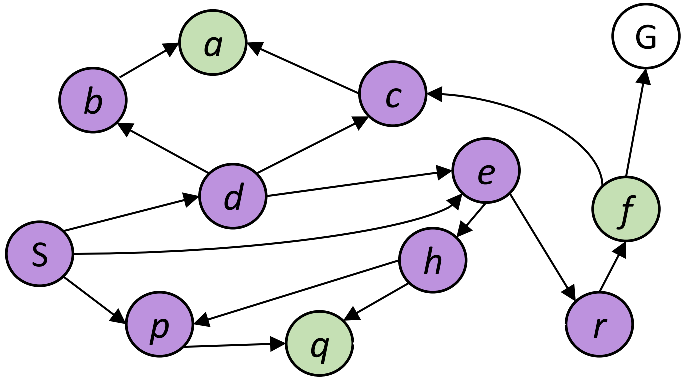


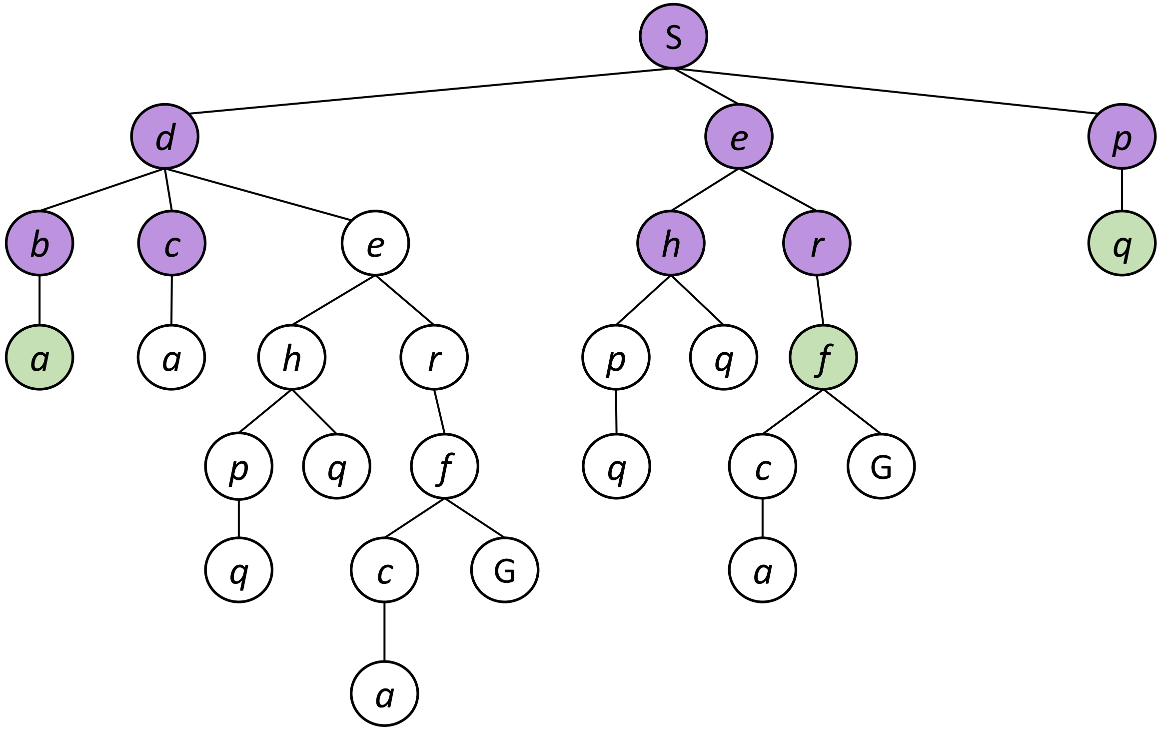
BFS



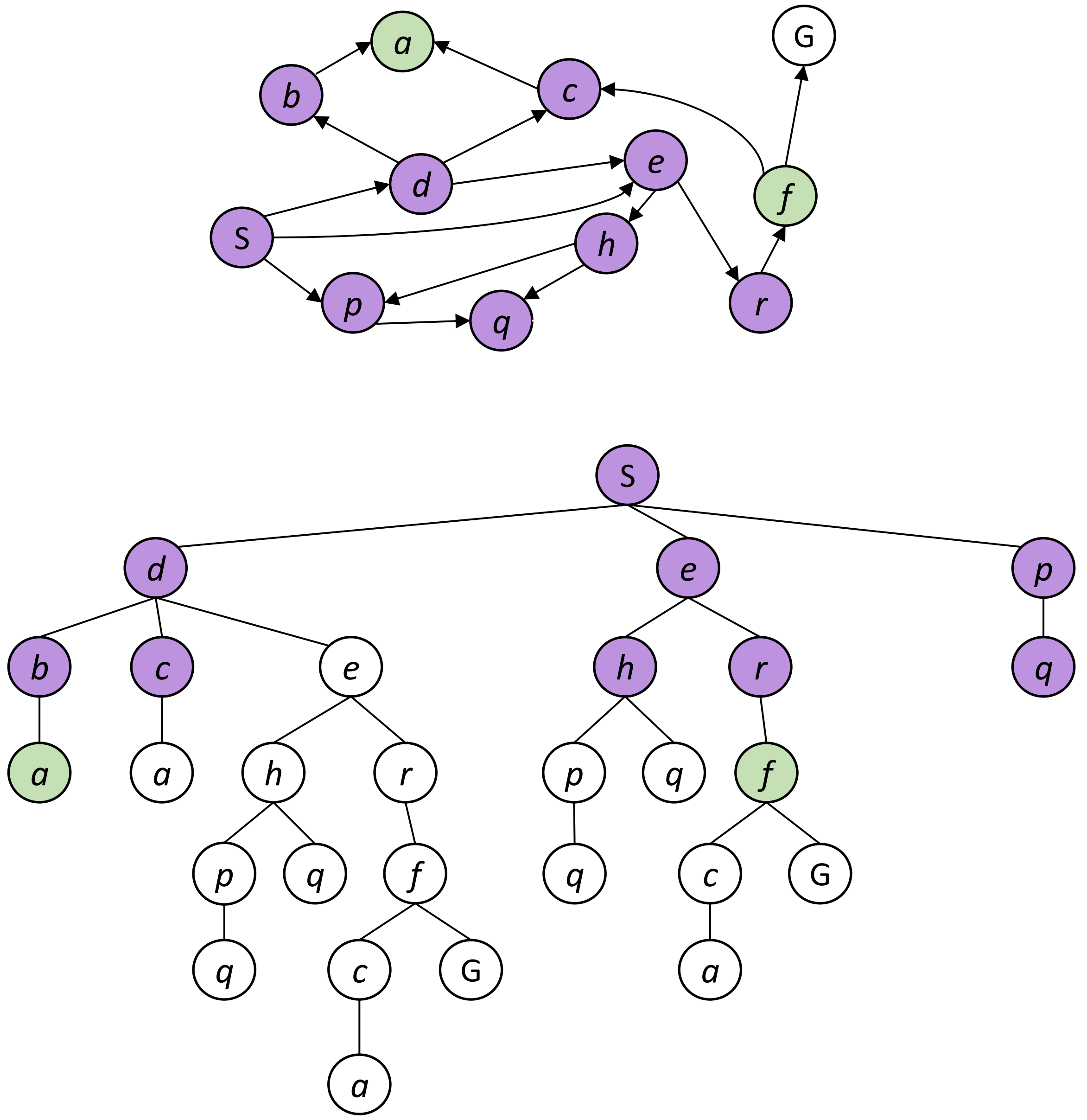


BFS

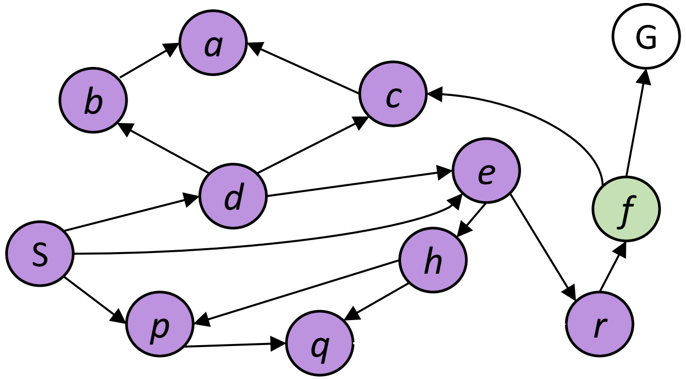


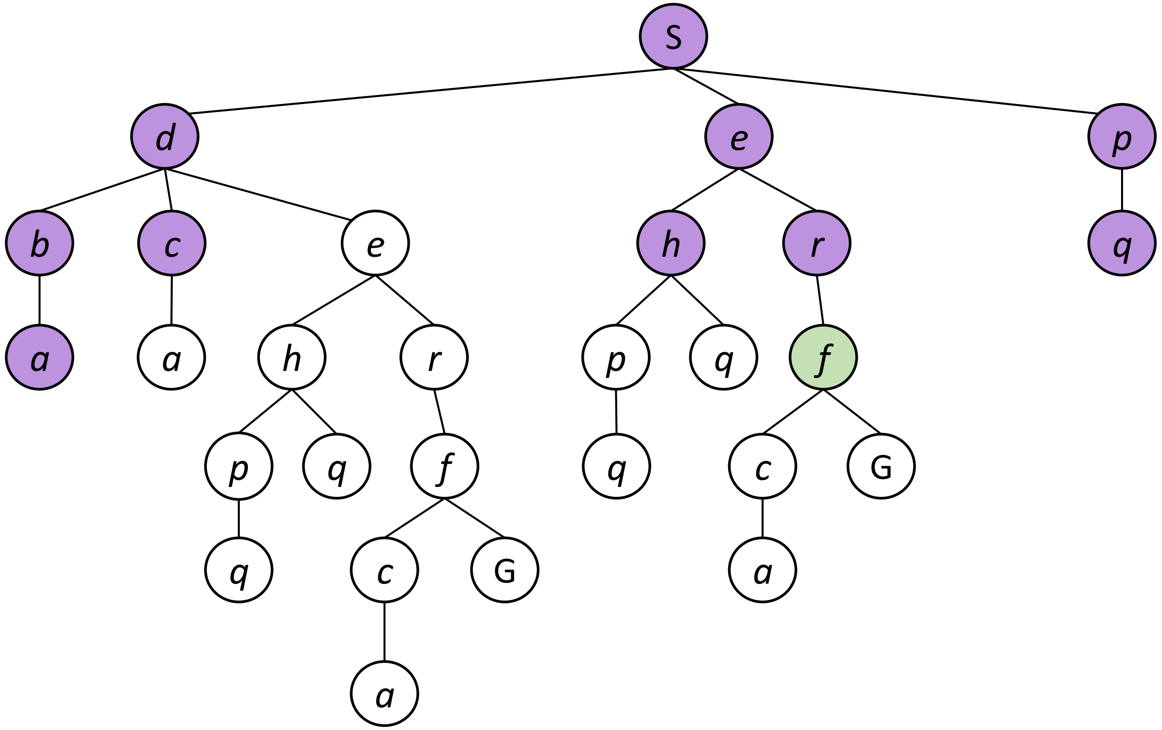


BFS

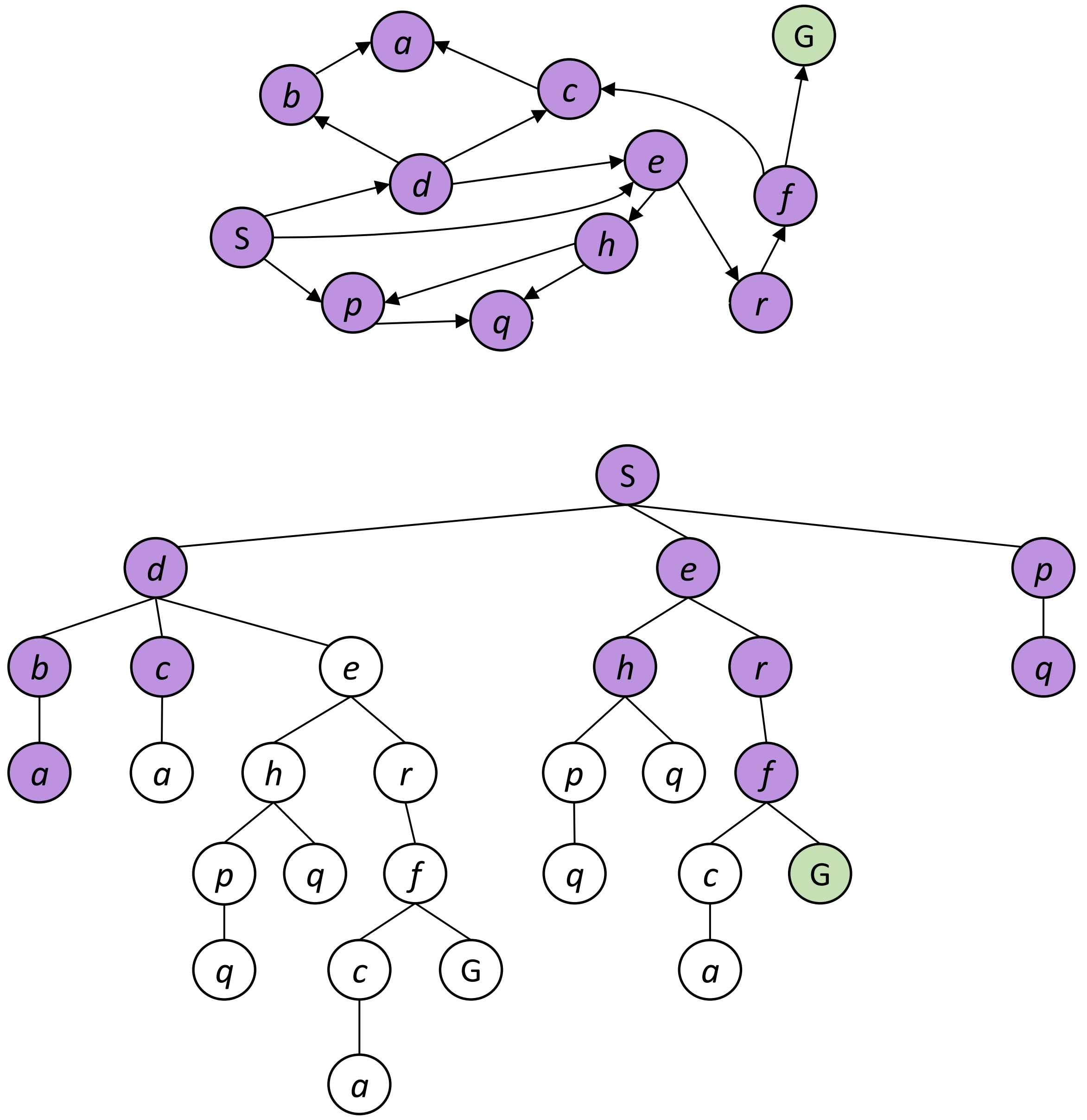


BFS





BFS



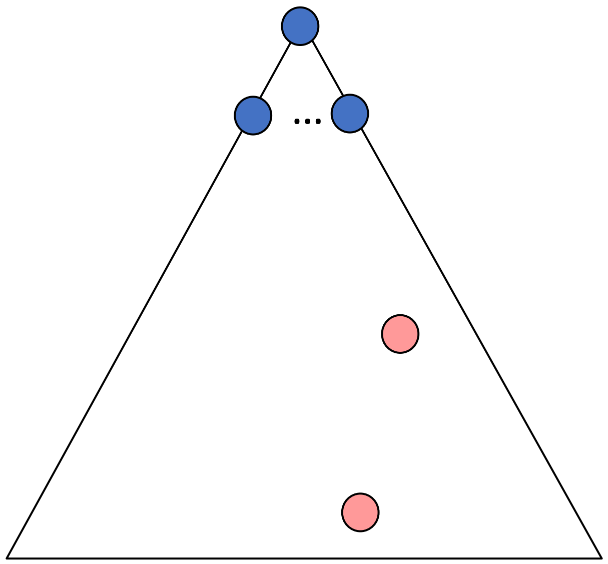
DFS vs. BFS

In what cases is DFS quicker to find the goal?

In what cases is BFS quicker to find the goal?

DFS vs. BFS

Does DFS / BFS find the goal with the smallest depth?

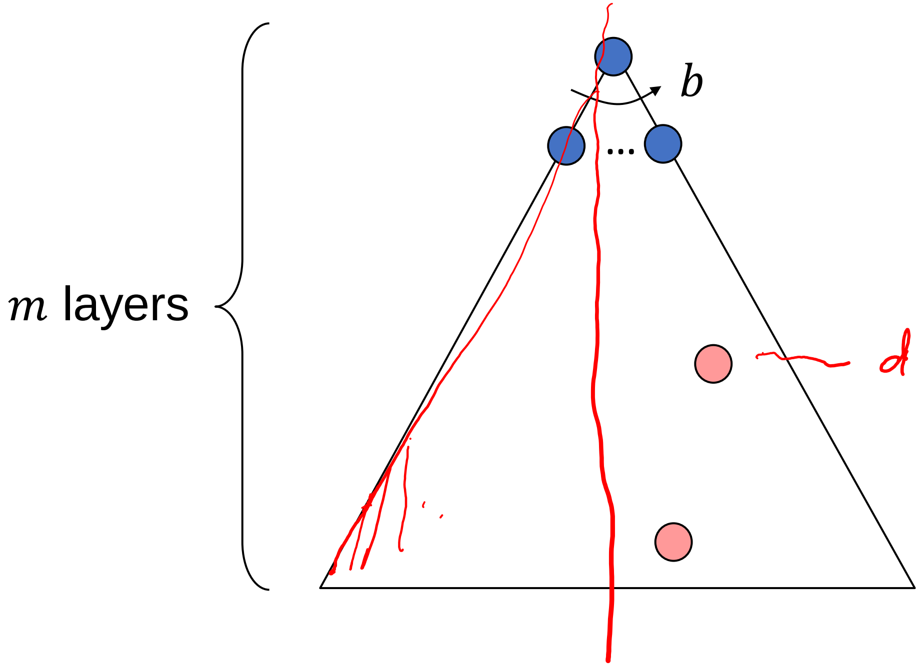


DFS vs. BFS

Suppose there exists a goal at layer $\leq d$.
What is the time complexity for DFS / BFS to find a goal?

BFS: $O(\underbrace{b}_{\text{expand layer 1}} + b^2 + \cdots + b^{d-1}) = O(b^d)$

DFS: $O(b^m)$



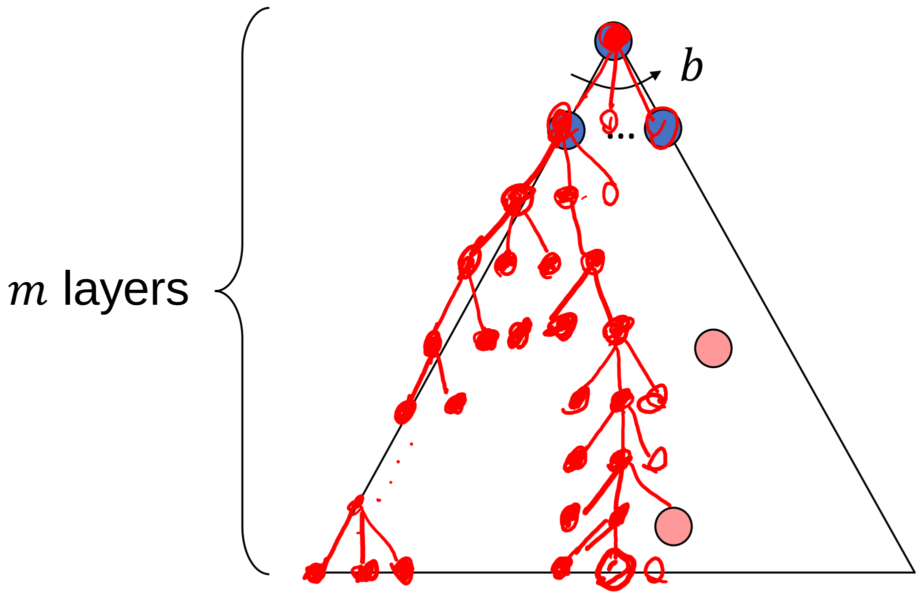
b: branching factor
m: maximum depth
Goals at various depths

DFS vs. BFS

What's the maximum possible size of **Frontier** in DFS / BFS?

BFS: $O(b^d)$

DFS: $O(bm)$



b: branching factor
m: maximum depth
Goals at various depths

DFS vs. BFS

$m > d$

	Time	Frontier Size
DFS	b^m	bm
BFS	b^d	b^d

So DFS can be more memory-efficient than BFS?

Yes … but not with our current implementation

DFS

Frontier ← { initial_state }
While **Frontier** is not empty:
    Pop the newest node s from **Frontier**

    For all action a:
        $s' \leftarrow \mathrm{succ}(s, a)$
        If not **Reached**$[s']$:
            If s' is a goal state, terminate
            Push s' to **Frontier**
            Reached$[s'] \leftarrow$ True

DFS

Frontier ← { initial_state }
While **Frontier** is not empty:

Pop the newest node s from **Frontier**

For all action a:

$s' \leftarrow \mathrm{succ}(s, a)$

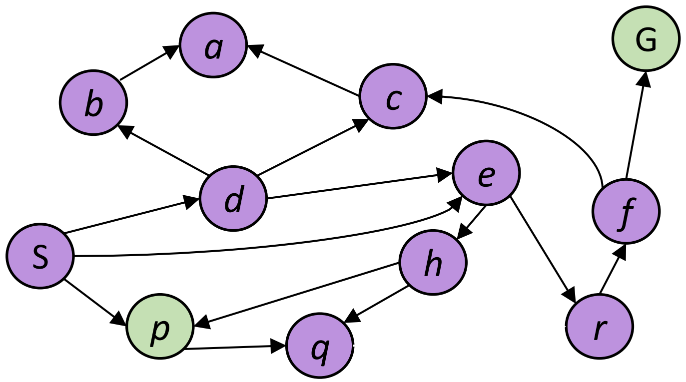
If s' is a goal state, terminate

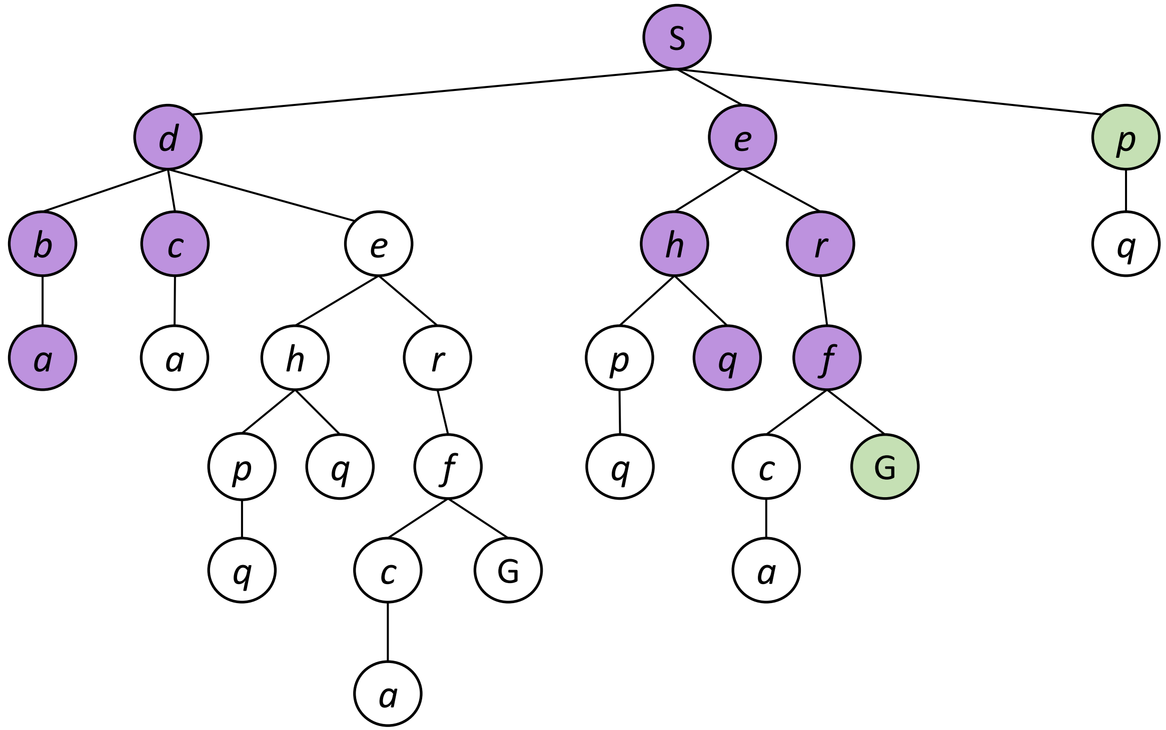
Push s' to **Frontier**

Because we omit the check, the algorithm may end up search in same sub-trees multiple times.

A Memory Efficient Version of DFS for Acyclic Graphs

Previous DFS Example

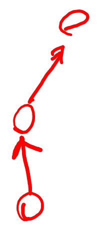




DFS

Frontier ← { initial_state }
While **Frontier** is not empty:
    Pop the **newest** node s from **Frontier**

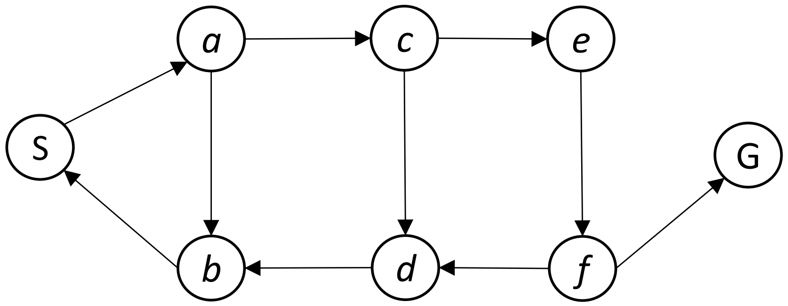
    For all action a:
        $s' \leftarrow \mathrm{succ}(s, a)$
        If s' is not an ancestor of s:
            If s' is a goal state, terminate
            Push s' to **Frontier**



A Memory Efficient Version of DFS for Cyclic Graphs

(Memory Efficient) DFS

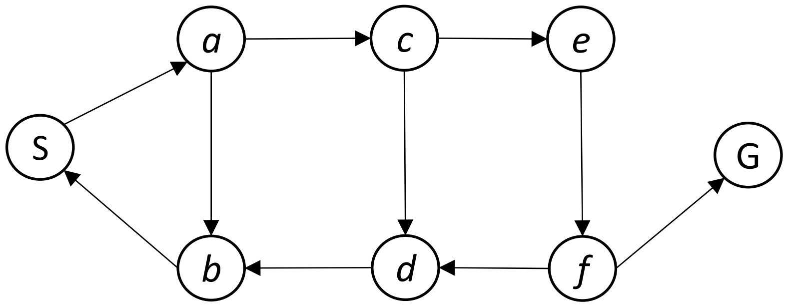
handling cycles



s

(Memory Efficient) DFS

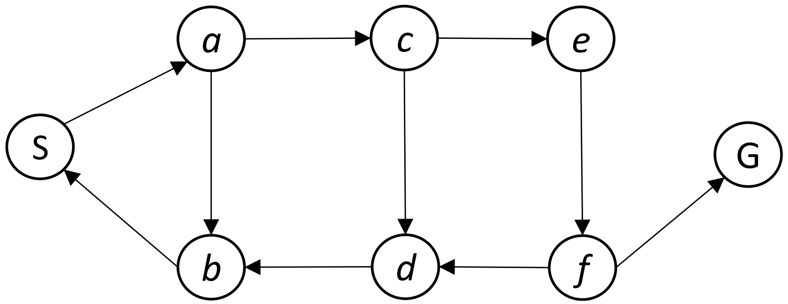
handling cycles

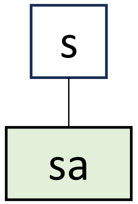


S

(Memory Efficient) DFS

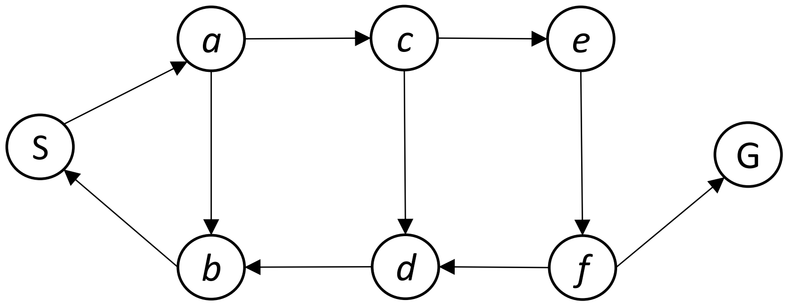
handling cycles

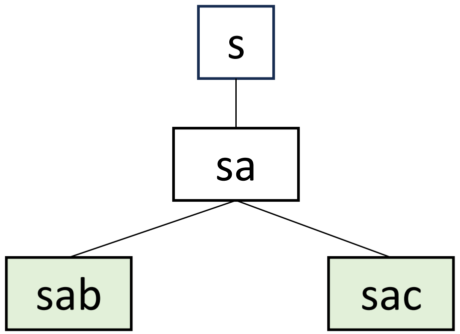




(Memory Efficient) DFS

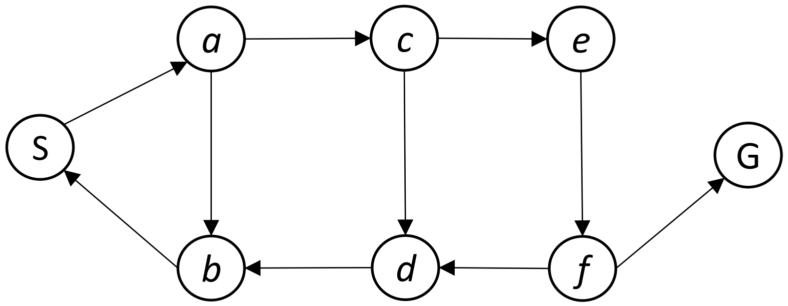
handling cycles

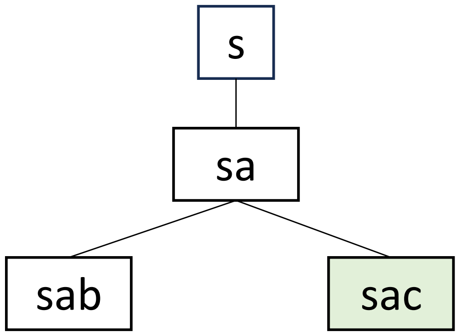




(Memory Efficient) DFS

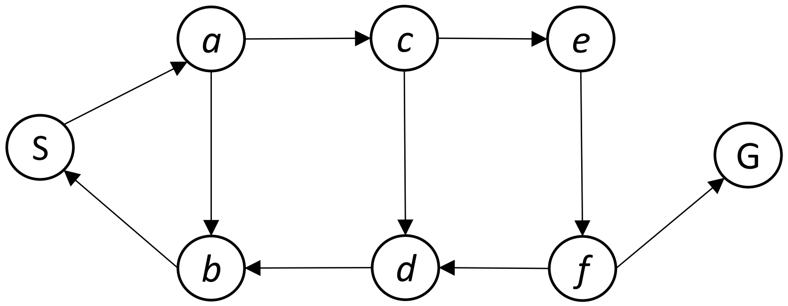
handling cycles

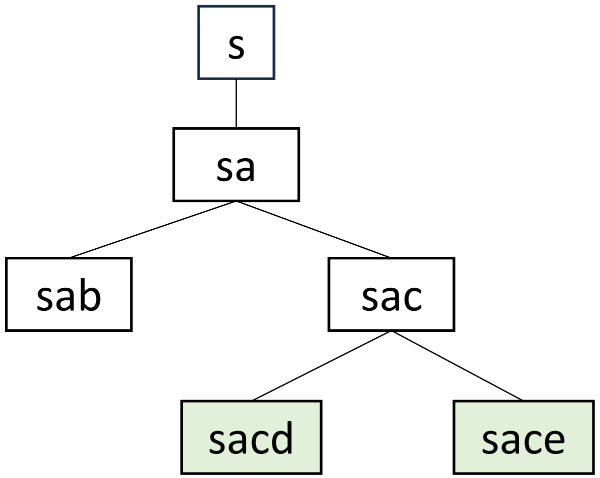




(Memory Efficient) DFS

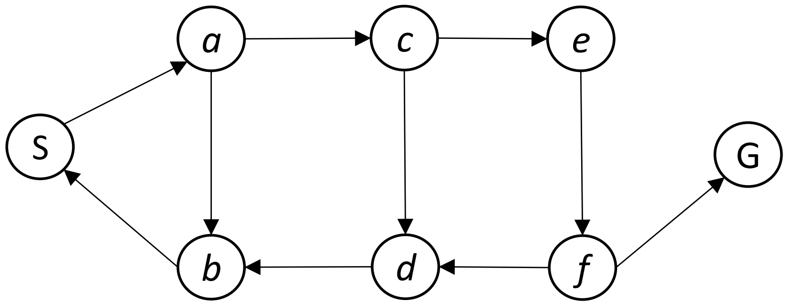
handling cycles

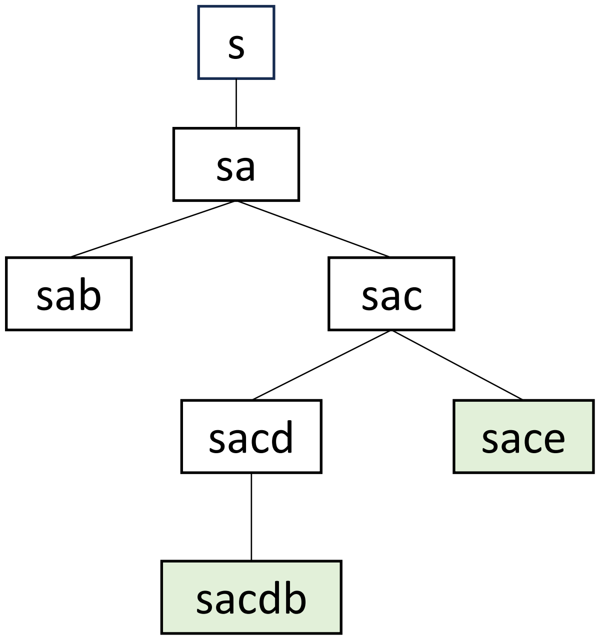




(Memory Efficient) DFS

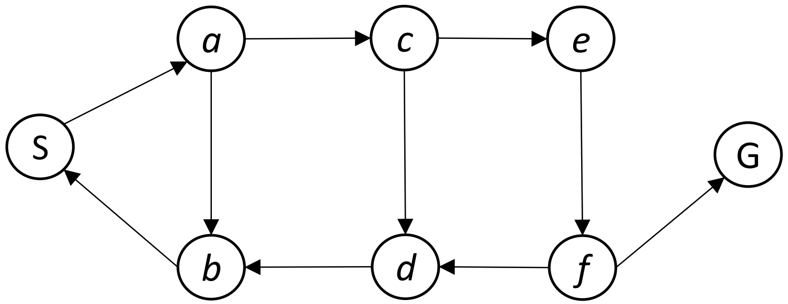
handling cycles

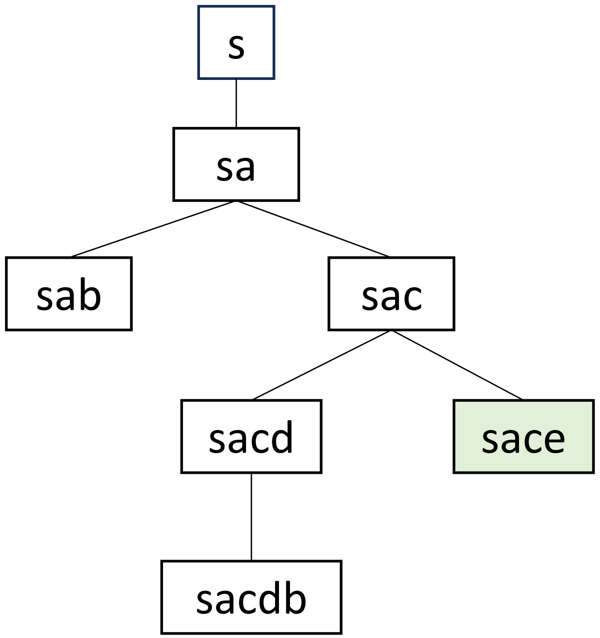




(Memory Efficient) DFS

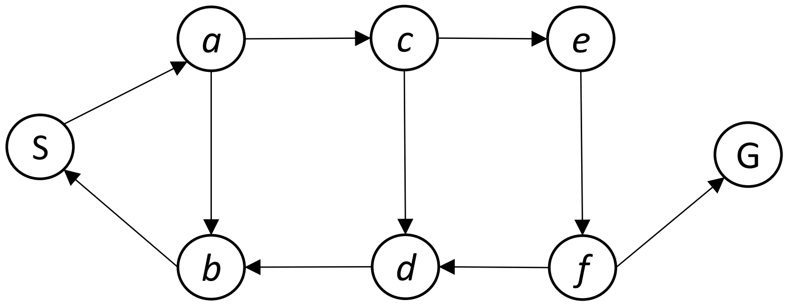
handling cycles

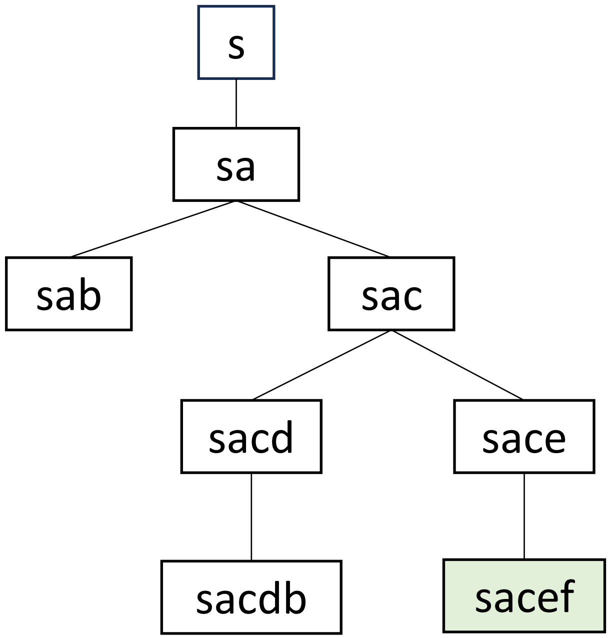




(Memory Efficient) DFS

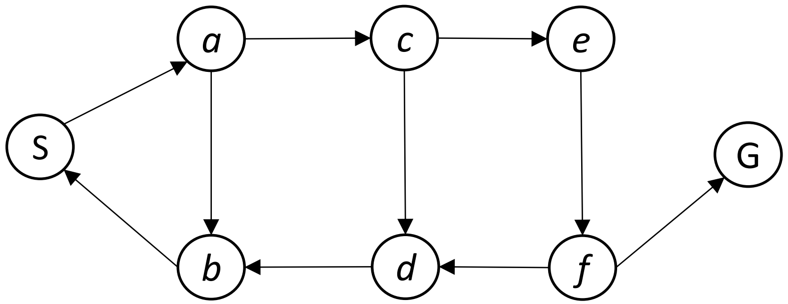
handling cycles

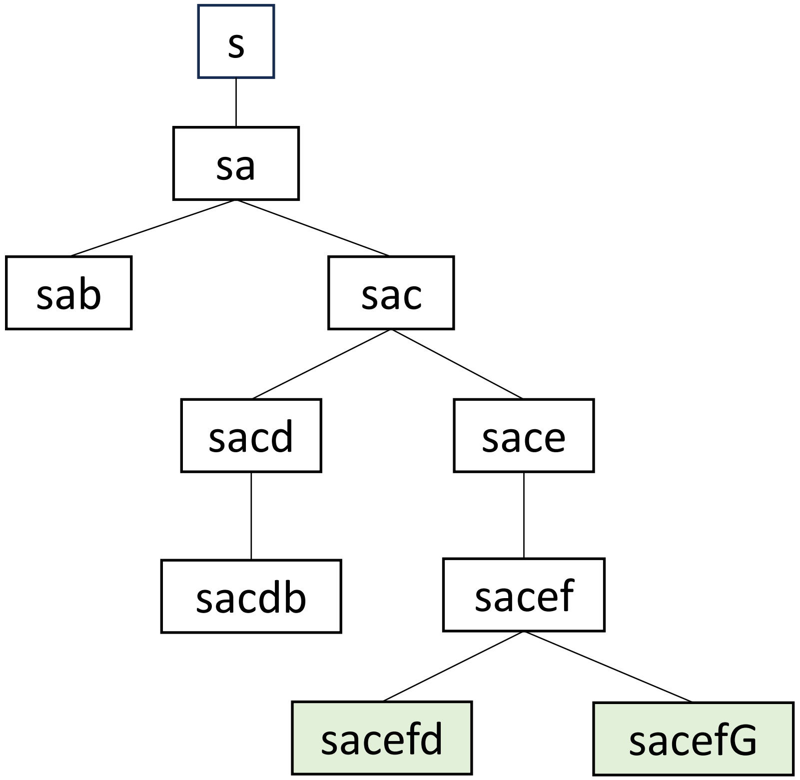


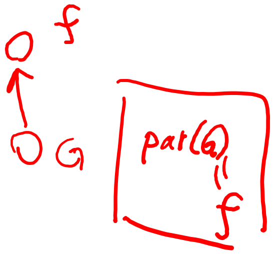


(Memory Efficient) DFS

handling cycles







DFS vs. BFS

	Time	Space
DFS (memory-efficient version)	$m \cdot b^m$	bm^2
BFS	b^d	b^d
IDS	$O\left(\sum_{i=1}^{d} i \cdot b^i\right) \leq O\left(d \cdot \sum_{i=1}^{d} b^i\right)$	bd^2

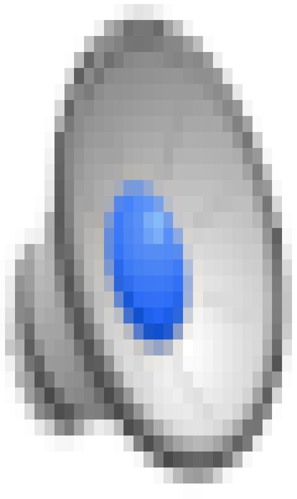
$\leq O\left(d \cdot b^d\right)$

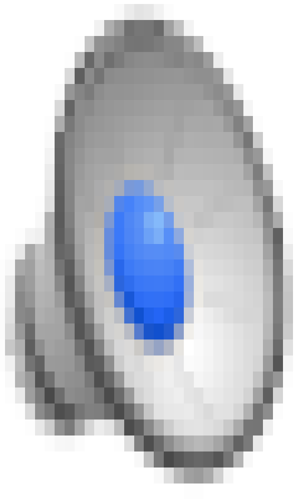
Iterative Deepening Search (IDS)

- Idea: get DFS's space advantage with BFS's time advantage
  - Run a DFS with depth limit 1. If no solution… b^1
  - Run a DFS with depth limit 2. If no solution… b^2
  - Run a DFS with depth limit 3. ….. $\vdots$

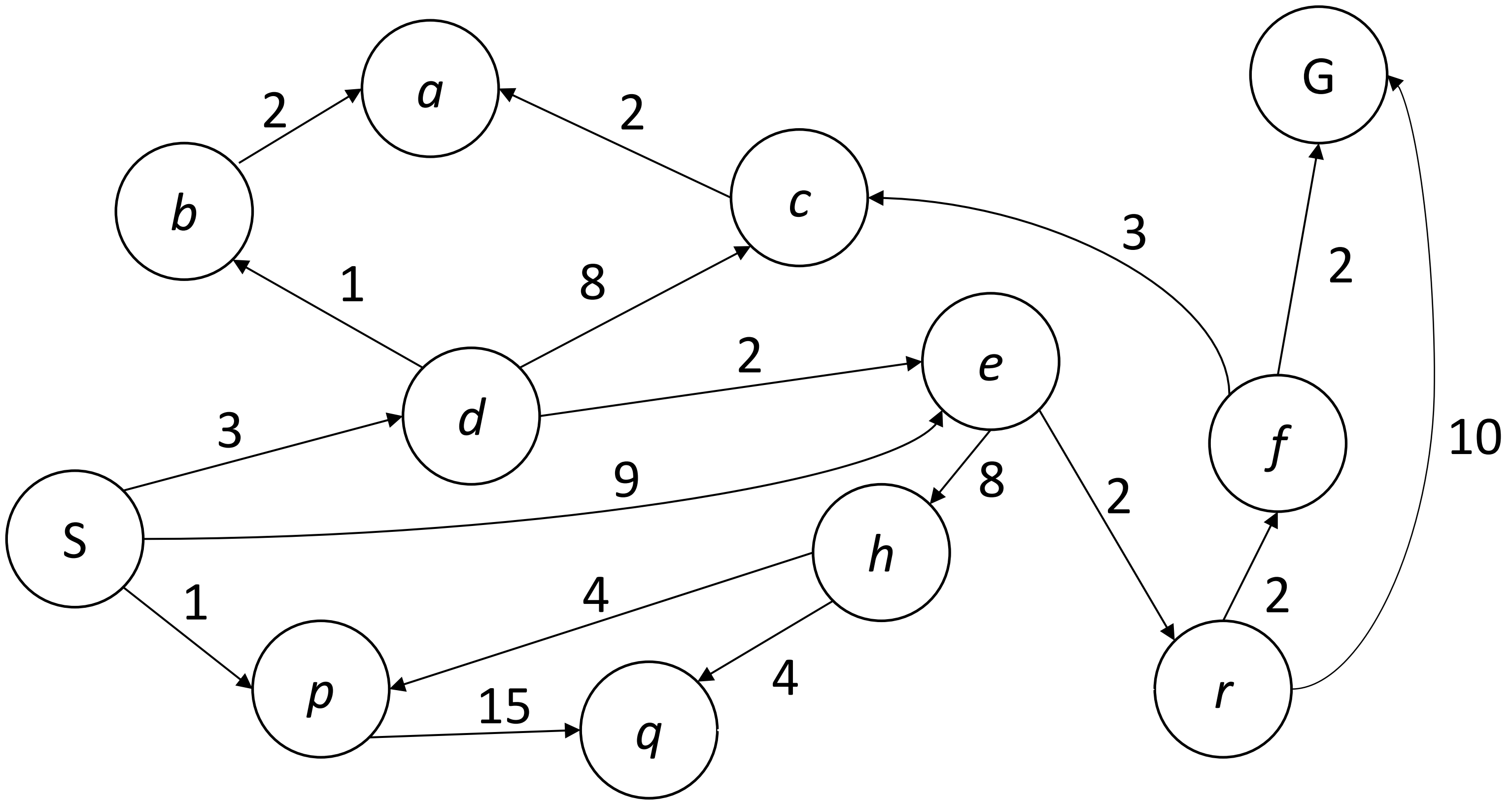
- Isn't that wastefully redundant? b^d
  - limit d.
  - Generally most work happens in the last level
  - Branching factor 10, solution 5 deep:
    - BFS: 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110
    - IDS: 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450

Which One is DFS/BFS?





Cost-Sensitive Search Problem

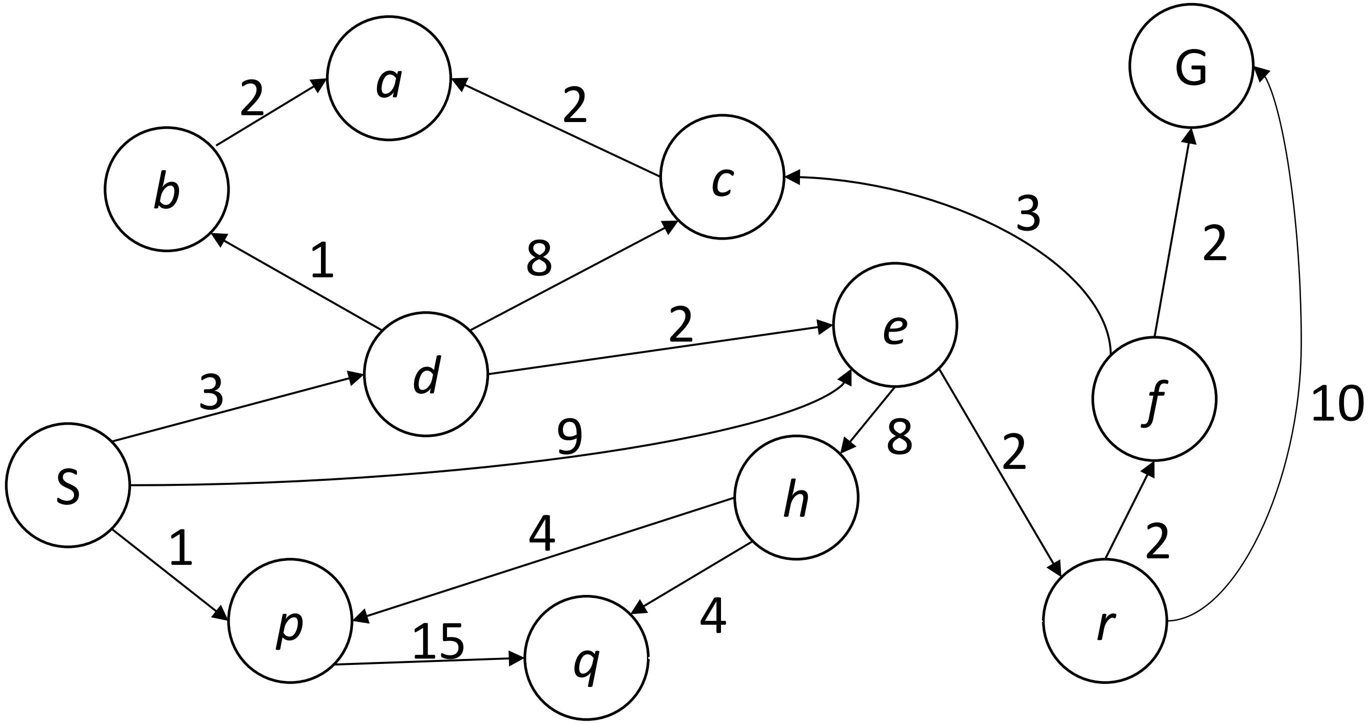


Uniform Cost Search (Dijkstra)

Frontier ← { initial_state }
While **Frontier** is not empty:
    Pop a node s from **Frontier** ← Choose the one with smallest $g(s)$
    If s is a goal state, then terminate

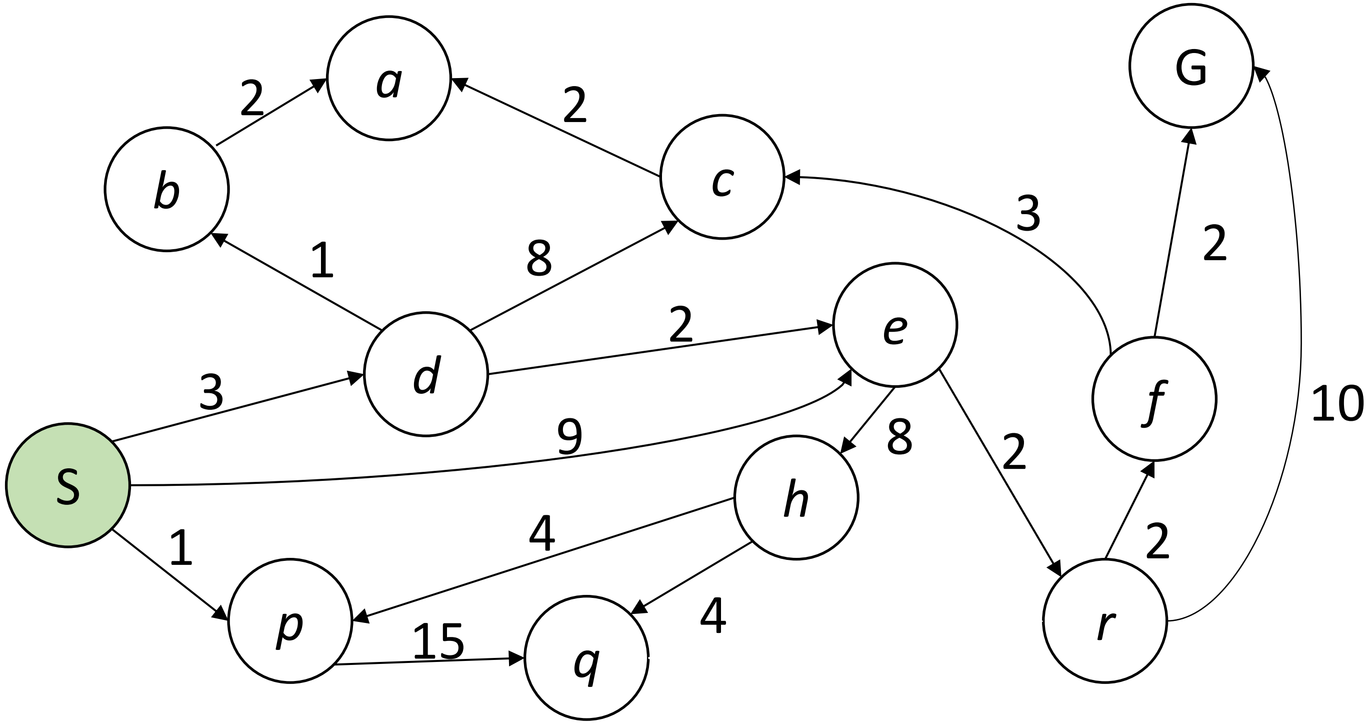
    For all action a:
        $s' \leftarrow \mathrm{succ}(s, a)$
        If not **Reached**[s']:
            Put s' in **Frontier**
            Reached[s'] ← True
        $g(s') \leftarrow \min\{\, g(s'), g(s) + \mathrm{cost}(s, a) \,\}$

UCS



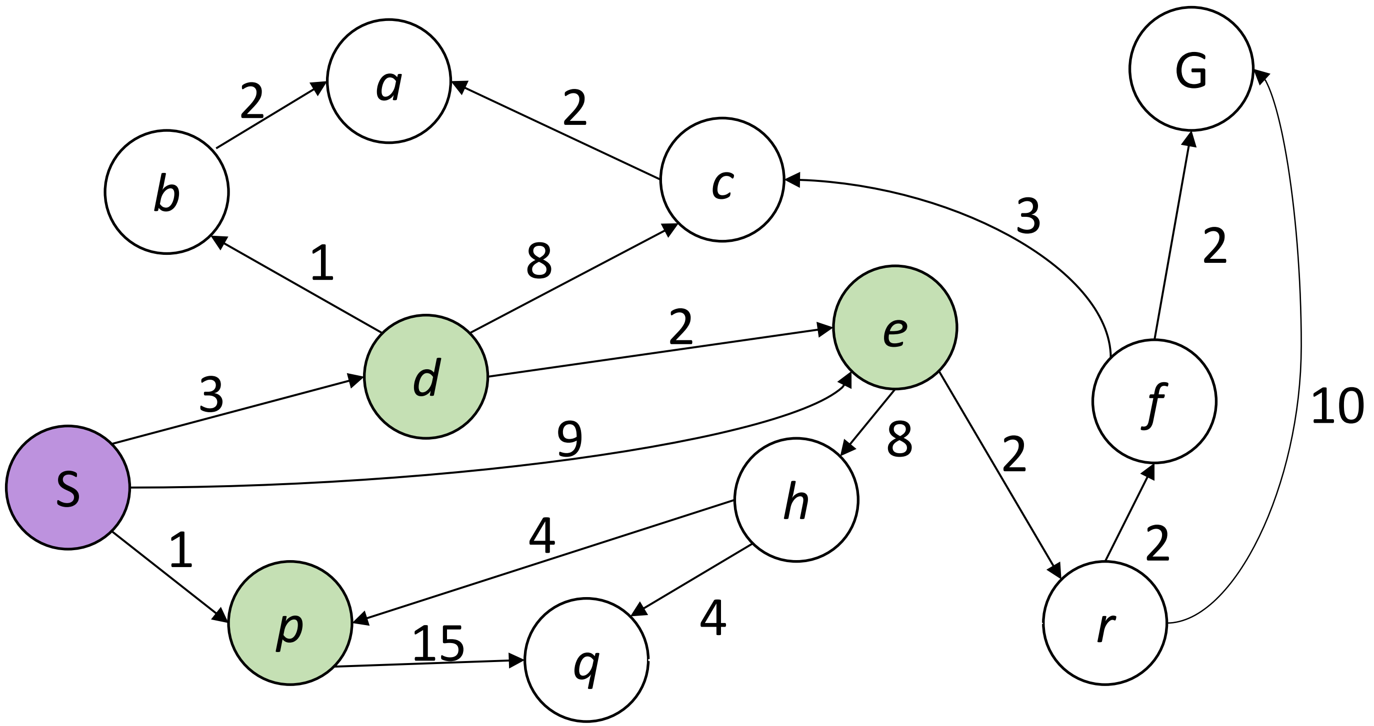
x	$g(x)$
S	
a	
b	
c	
d	
e	
f	
h	
p	
q	
r	
G	

UCS



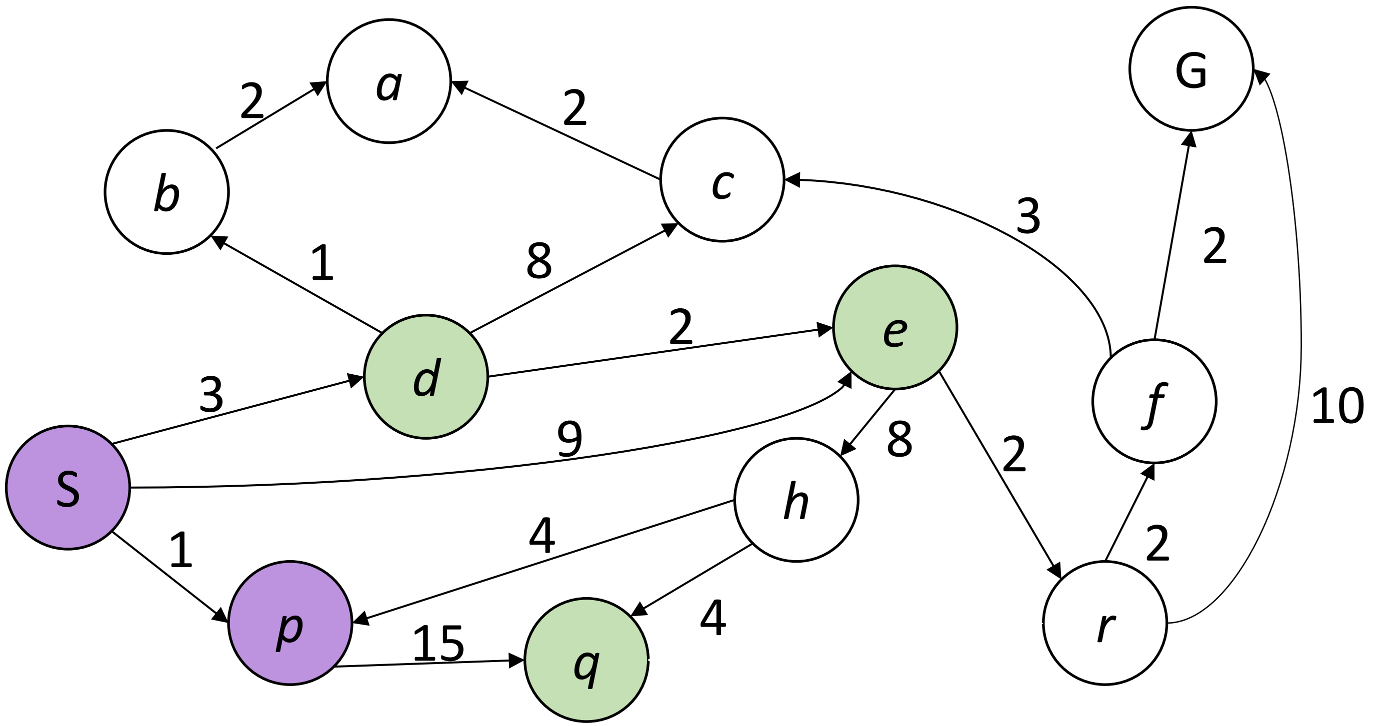
x	$g(x)$
S	0
a	∞
b	∞
c	∞
d	∞
e	∞
f	∞
h	∞
p	∞
q	∞
r	∞
G	∞

UCS



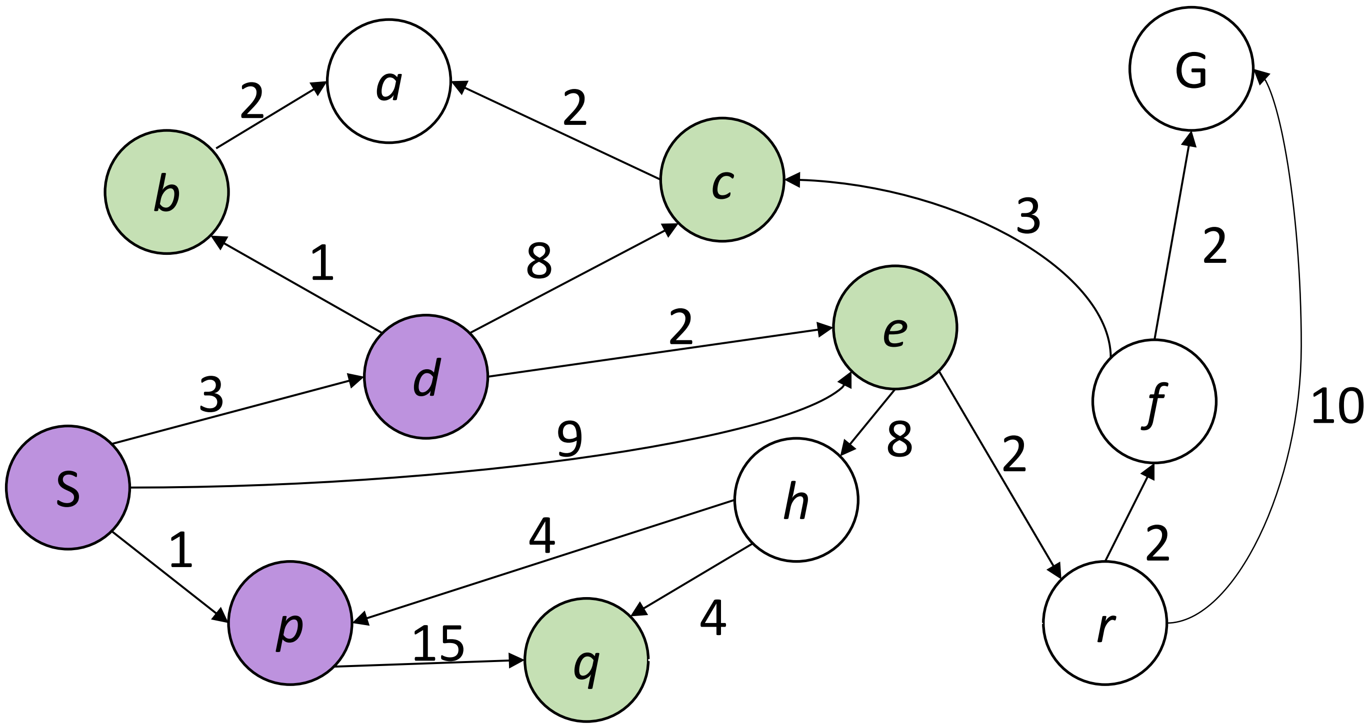
x	$g(x)$
S	0
a	∞
b	∞
c	∞
d	3
e	9
f	∞
h	∞
p	1
q	∞
r	∞
G	∞

UCS



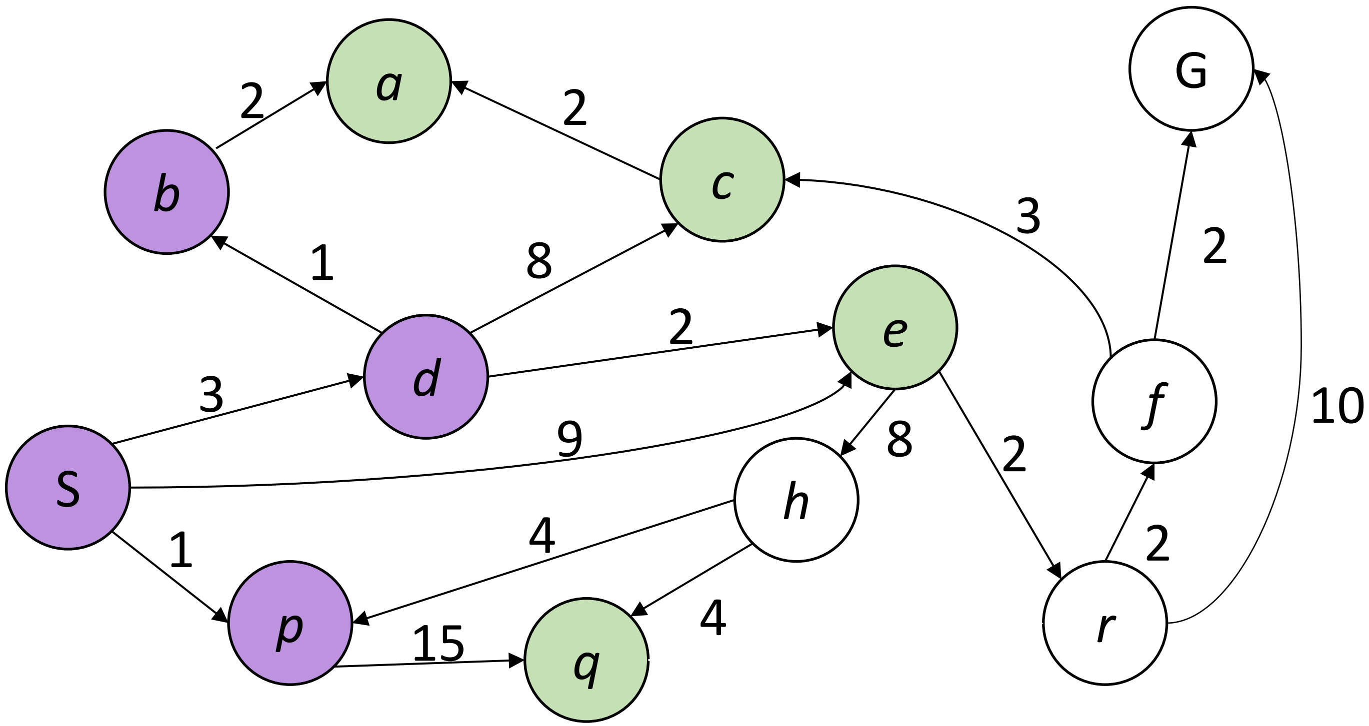
x	$g(x)$
S	0
a	∞
b	∞
c	∞
d	3
e	9
f	∞
h	∞
p	1
q	16
r	∞
G	∞

UCS



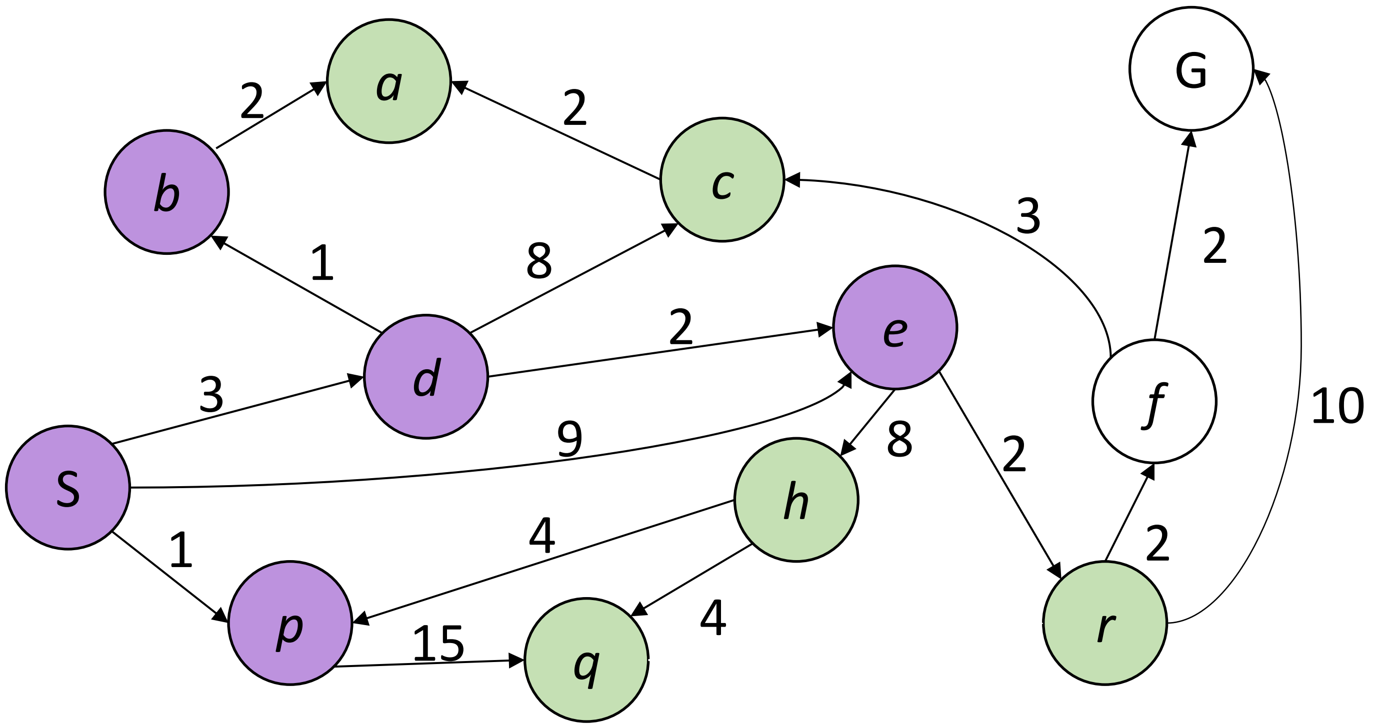
x	$g(x)$
S	0
a	∞
b	4
c	11
d	3
e	5
f	∞
h	∞
p	1
q	16
r	∞
G	∞

UCS



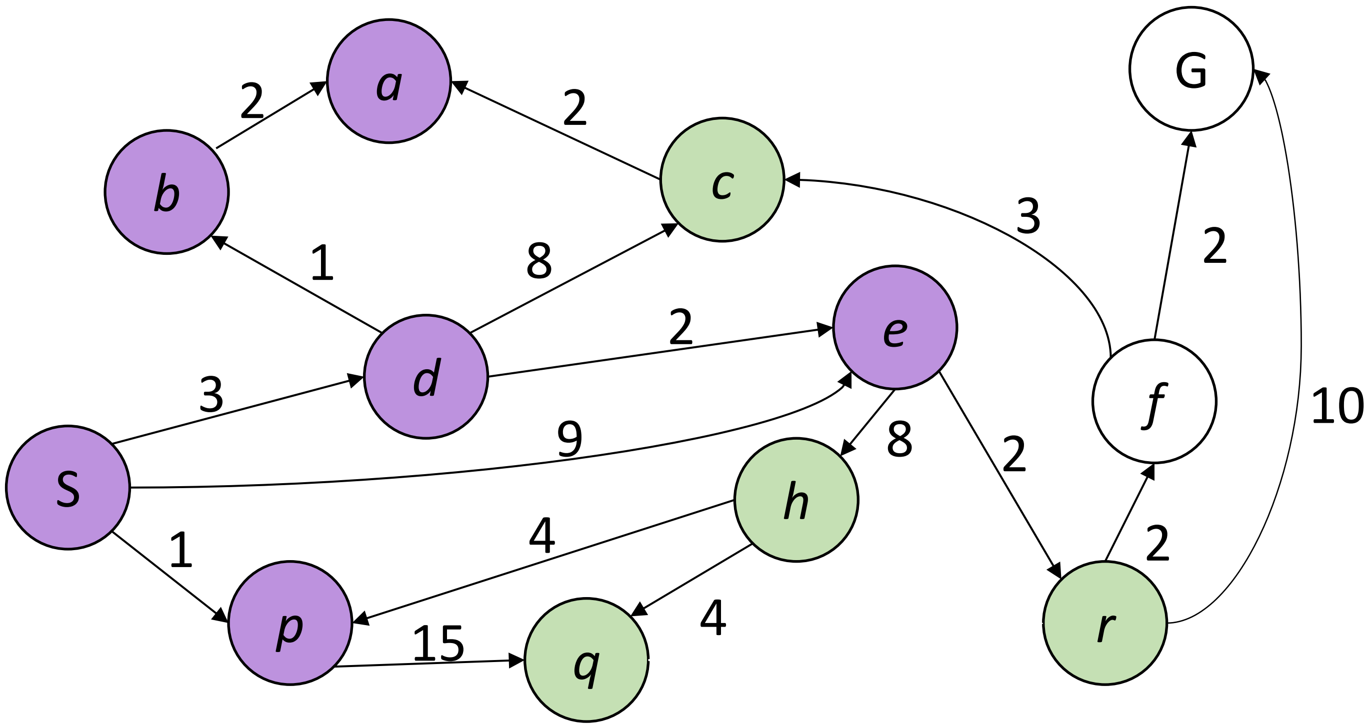
x	$g(x)$
S	0
a	6
b	4
c	11
d	3
e	5
f	∞
h	∞
p	1
q	16
r	∞
G	∞

UCS



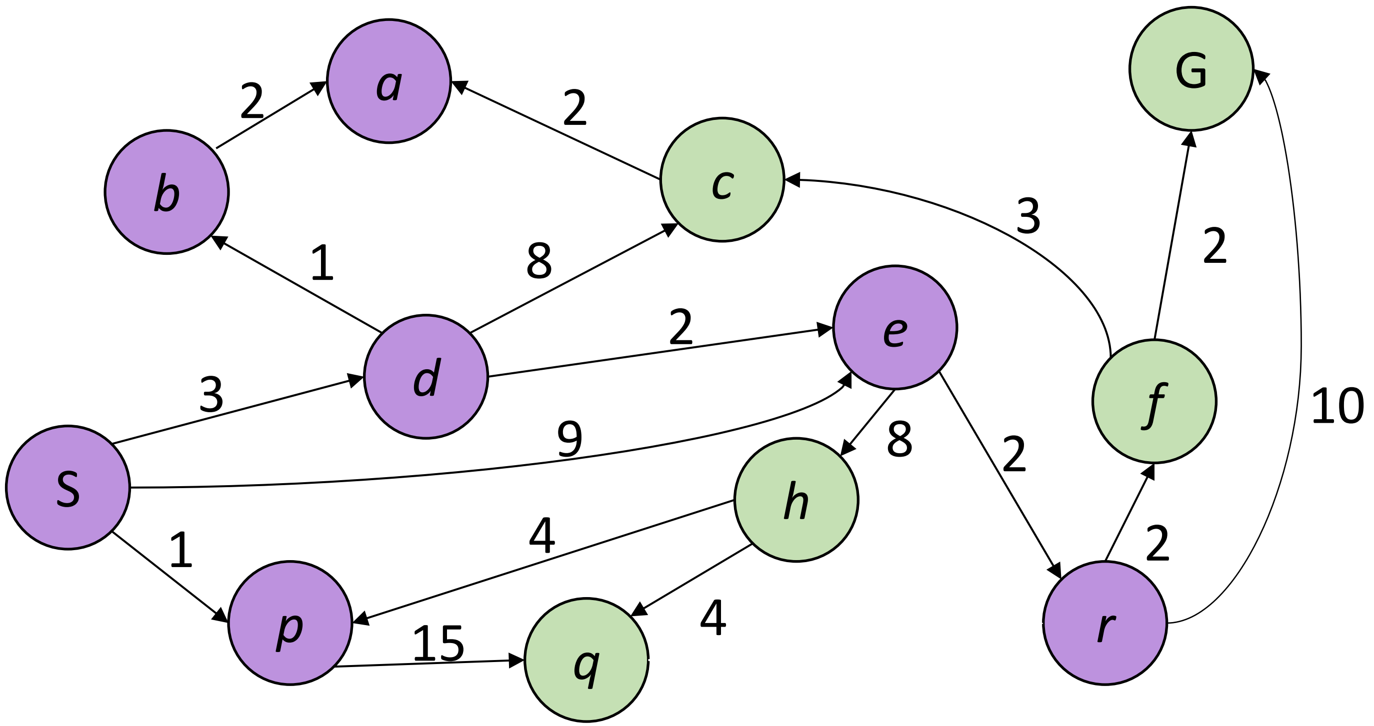
x	$g(x)$
S	0
a	6
b	4
c	11
d	3
e	5
f	∞
h	13
p	1
q	16
r	7
G	∞

UCS



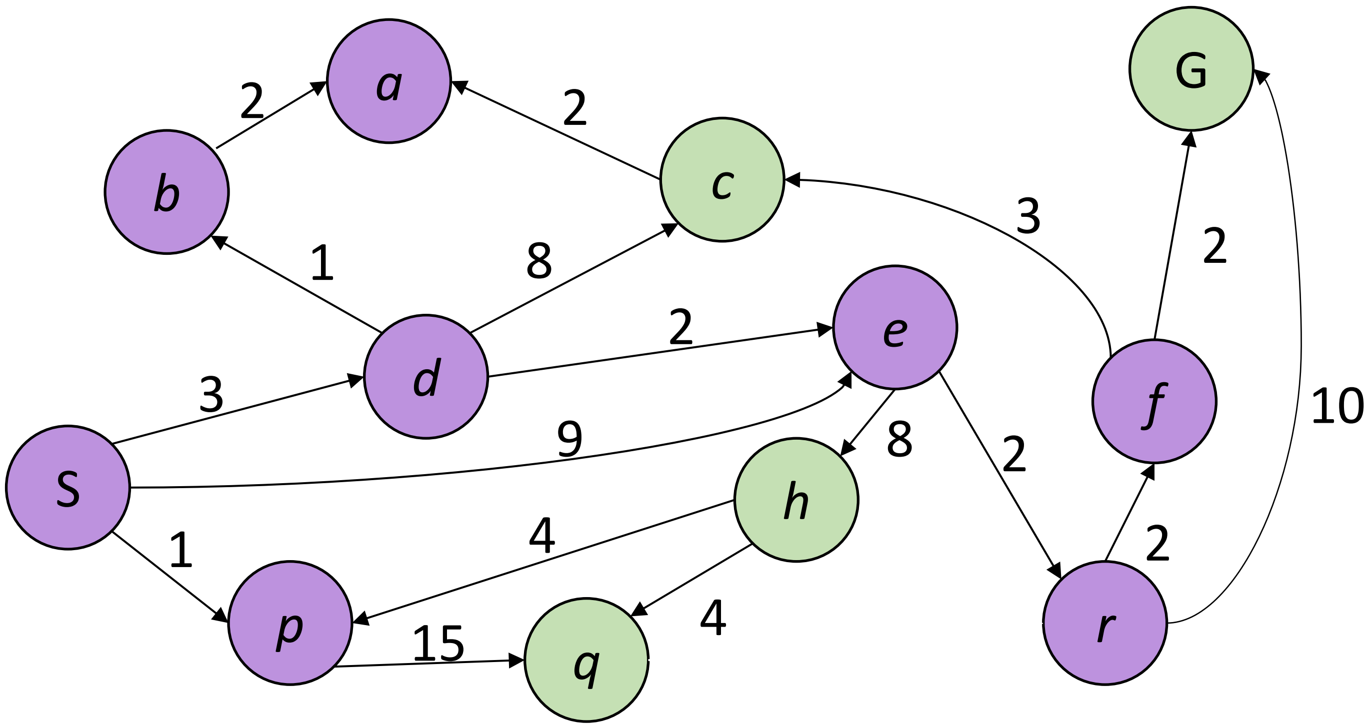
x	$g(x)$
S	0
a	6
b	4
c	11
d	3
e	5
f	∞
h	13
p	1
q	16
r	7
G	∞

UCS



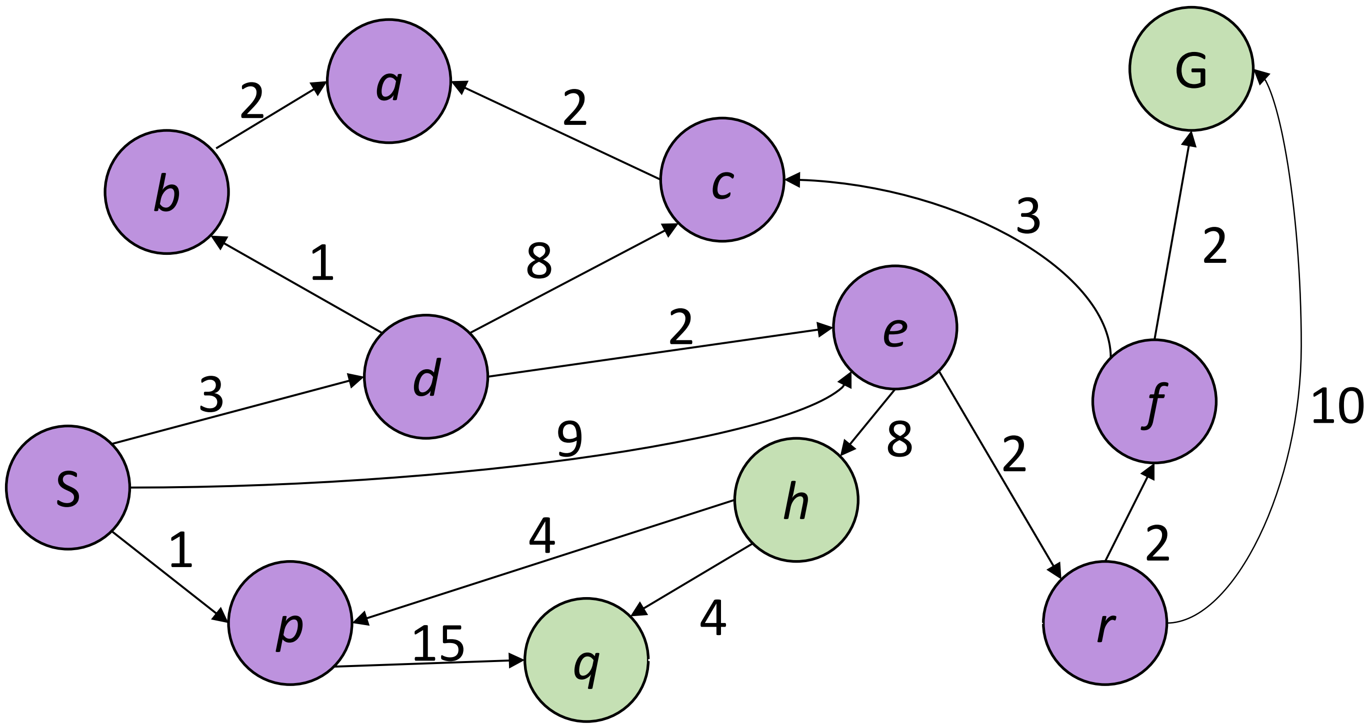
x	$g(x)$
S	0
a	6
b	4
c	11
d	3
e	5
f	9
h	13
p	1
q	16
r	7
G	17

UCS



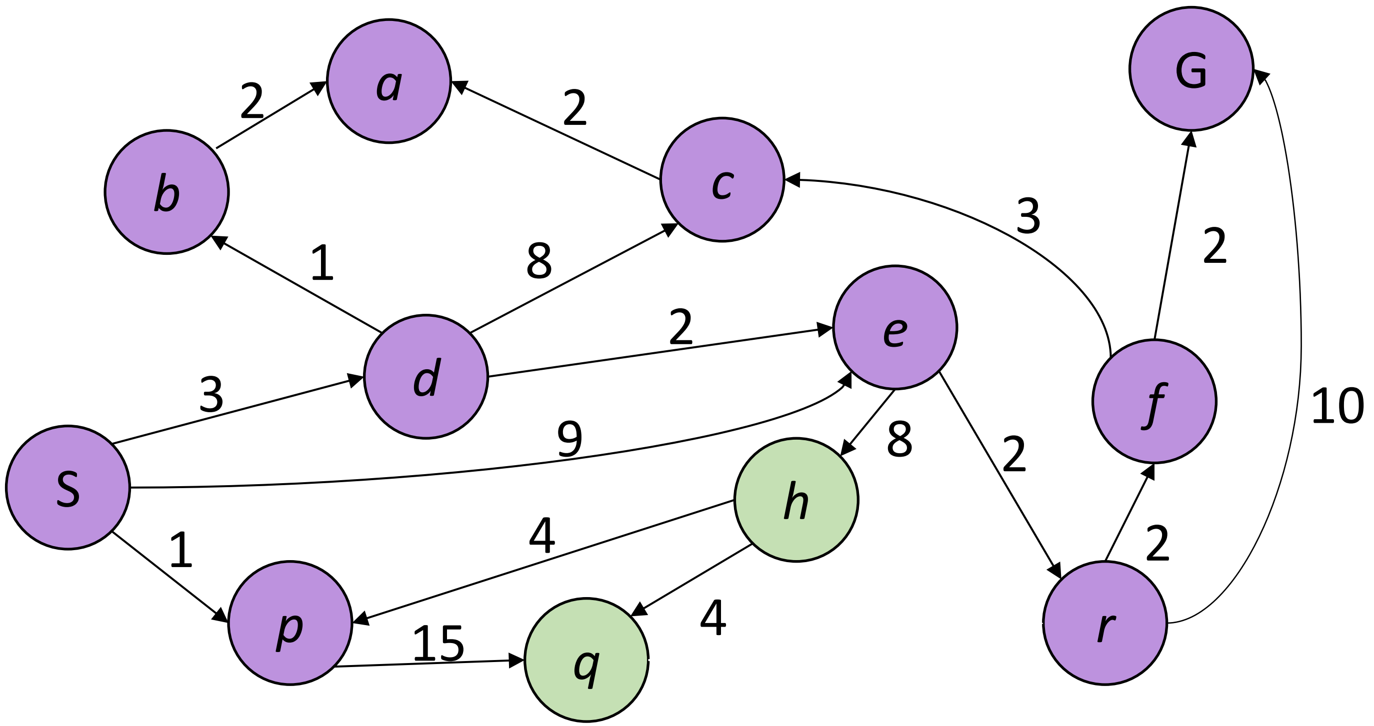
x	$g(x)$
S	0
a	6
b	4
c	11
d	3
e	5
f	9
h	13
p	1
q	16
r	7
G	11

UCS



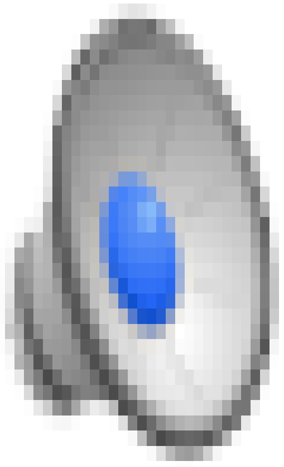
x	$g(x)$
S	0
a	6
b	4
c	11
d	3
e	5
f	9
h	13
p	1
q	16
r	7
G	11

UCS

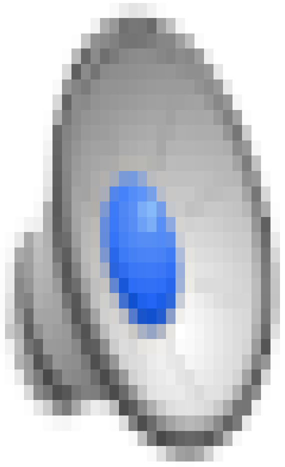


x	$g(x)$
S	0
a	6
b	4
c	11
d	3
e	5
f	9
h	13
p	1
q	16
r	7
G	11

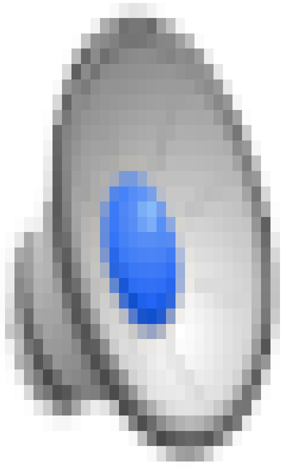
DFS/BFS/UCS? (Deep/light blue → high/low cost)



DFS/BFS/UCS? (Deep/light blue → high/low cost)

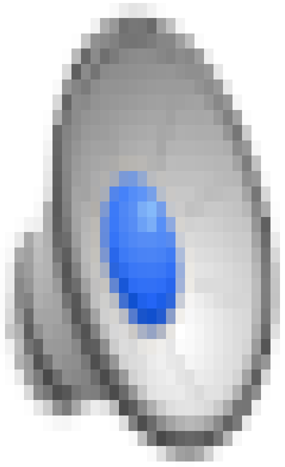


DFS/BFS/UCS? (Deep/light blue → high/low cost)

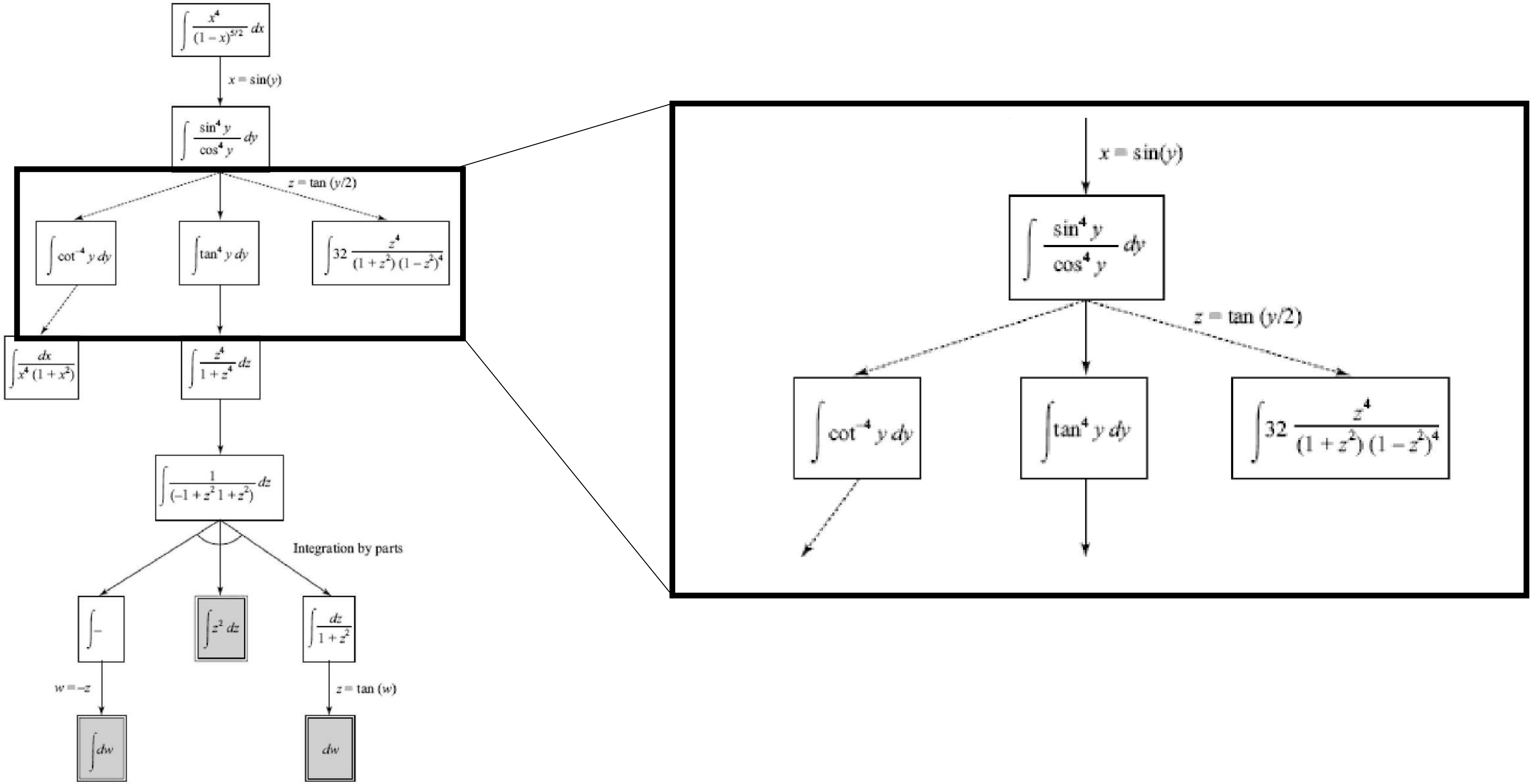


Informed Search

Inefficiency of the Search Algorithms We See So Far



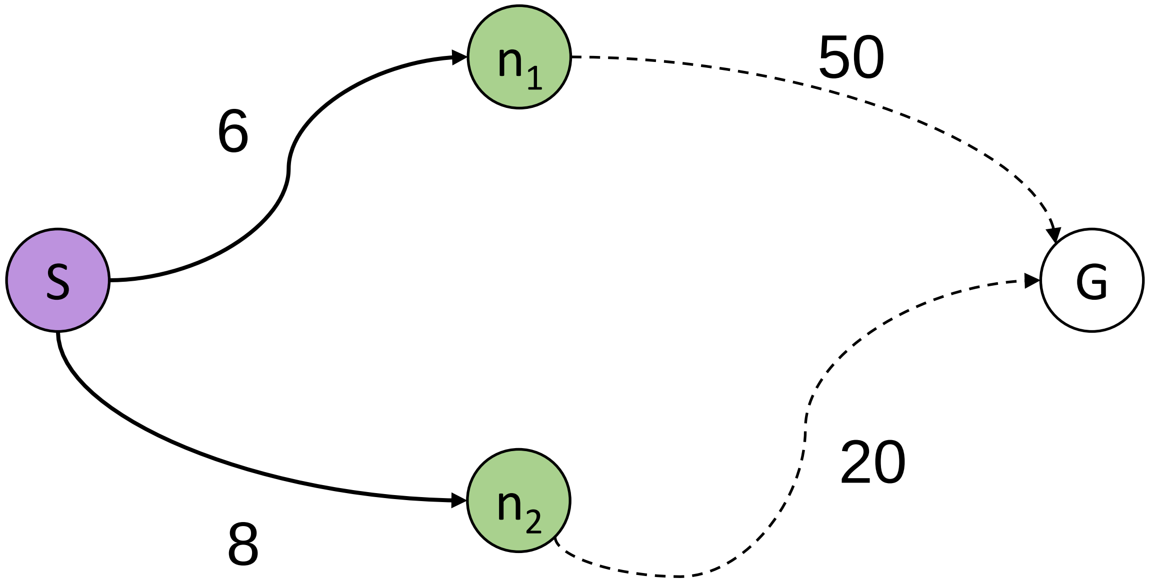
Inefficiency of the Search Algorithms We See So Far



Heuristic Function

Suppose we have some **“guess”** for the distance from every node to the goal.

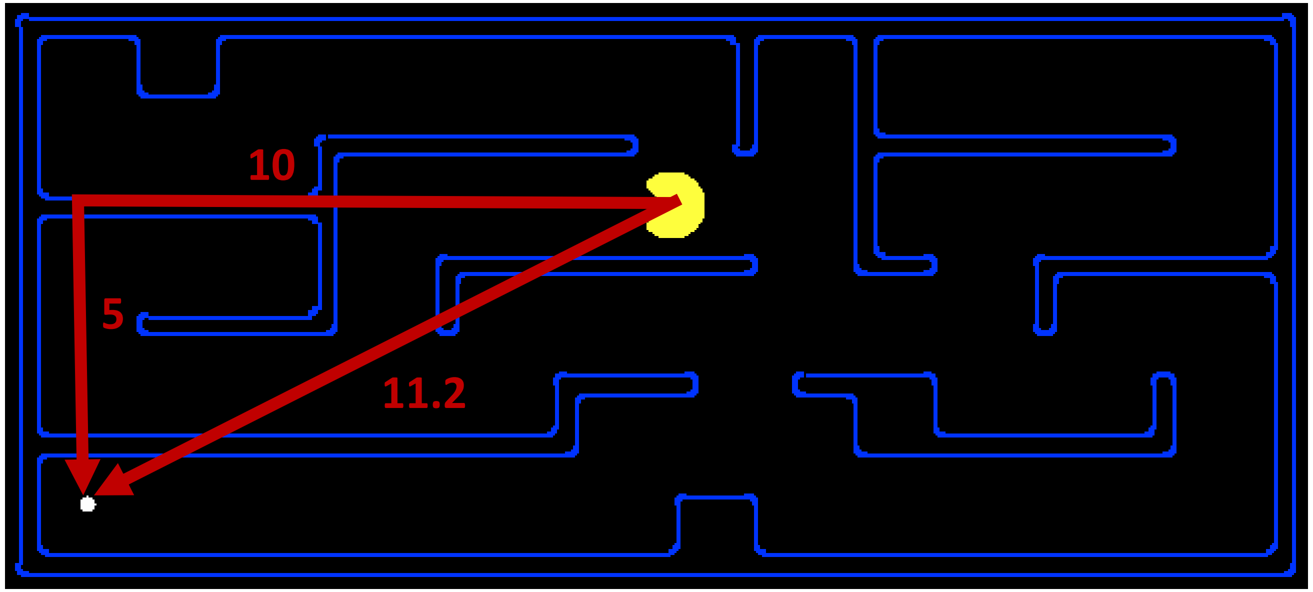
Can we leverage it to accelerate the search?



Heuristic Function

Having a heuristic function that accurately predict the distance might be impossible.

However, some function that **correlates** with the true distance may be easy to find.



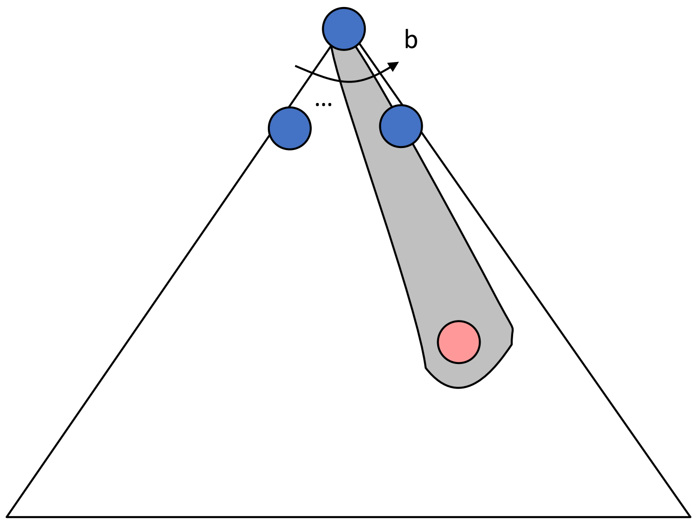
Greedy Best-First Search

Suppose we have a heuristic function $h(s)$.

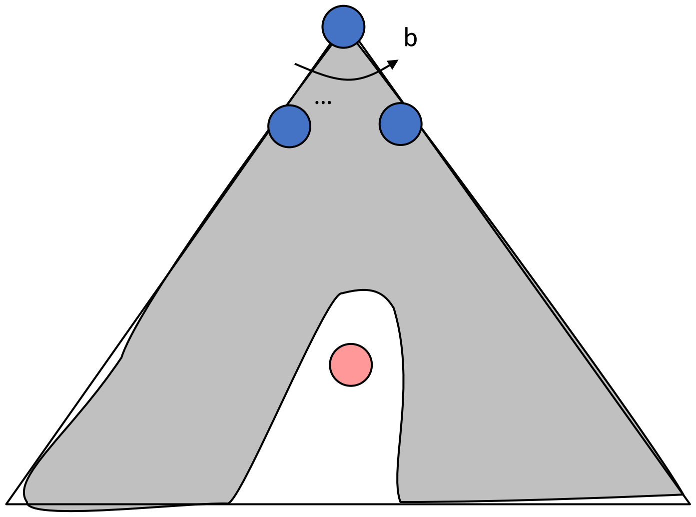
Frontier $\leftarrow$ { initial_state }
While **Frontier** is not empty:
    Pop a node s from **Frontier** $\leftarrow$ Choose the one with smallest $h(s)$
    For all action a:
        $s' \leftarrow \mathrm{succ}(s, a)$
        If not Reached[s']:
            If s' is a goal state, then terminate
            Push s' to **Frontier**
            Reached[s'] $\leftarrow$ True

Greedy Best-First Search

- If the heuristic is good
  - Take us directly to the goal
  - Like a nicely-guided DFS



- In the worst case
  - Take us to the wrong way
  - Like a badly-guided DFS



A* Search: Combining Greedy and UCS

Suppose we have a heuristic function $h(s)$.

Backward cost **Forward cost**

Frontier ← { initial_state }

While **Frontier** is not empty:

Pop a node s from **Frontier** ← Choose the one with smallest $g(s) + h(s)$

If s is a goal state, then terminate

UCS is a special case with $h(s) = 0$

For all action a:

$s' \leftarrow \mathrm{succ}(s, a)$

If not Reached[s']:

Push s' to **Frontier**

Reached[s'] ← True

$g(s') \leftarrow \min \{ g(s'), g(s) + \mathrm{cost}(s, a) \}$

A* Search

Evaluation functions:

- Greedy (Greedy Best-First) Search: $h(s)$
- Uniform Cost Search: $g(s)$
- A* Search: $g(s) + h(s)$
- Weighted A* Search: $g(s) + w \cdot h(s)$ for some $w \in (0, \infty)$

cost(s → s') = cost(s, a) where s' = succ(s, a)

A* Search

A* Search = Uniform Cost Search with modified cost

$$\widetilde{\text{cost}}(s,a) = \text{cost}(s,a) + h(s') - h(s)$$

$\tilde{c} = c + h(s') - h(s) < c$

good h: $h(s') < h(s)$

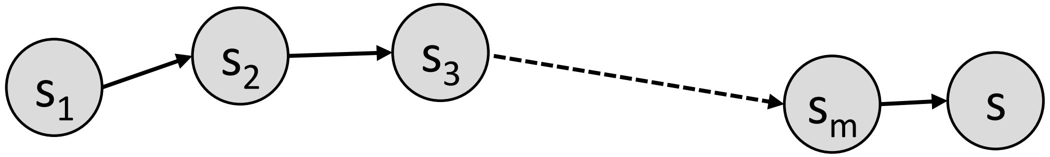
Proof

Let $\tilde{g}(s)$ be the values of UCS with the modified loss

$$\tilde{g}(s) = \widetilde{\text{cost}}(s_1 \to s_2) + \widetilde{\text{cost}}(s_2 \to s_3) + \cdots + \widetilde{\text{cost}}(s_m \to s)$$

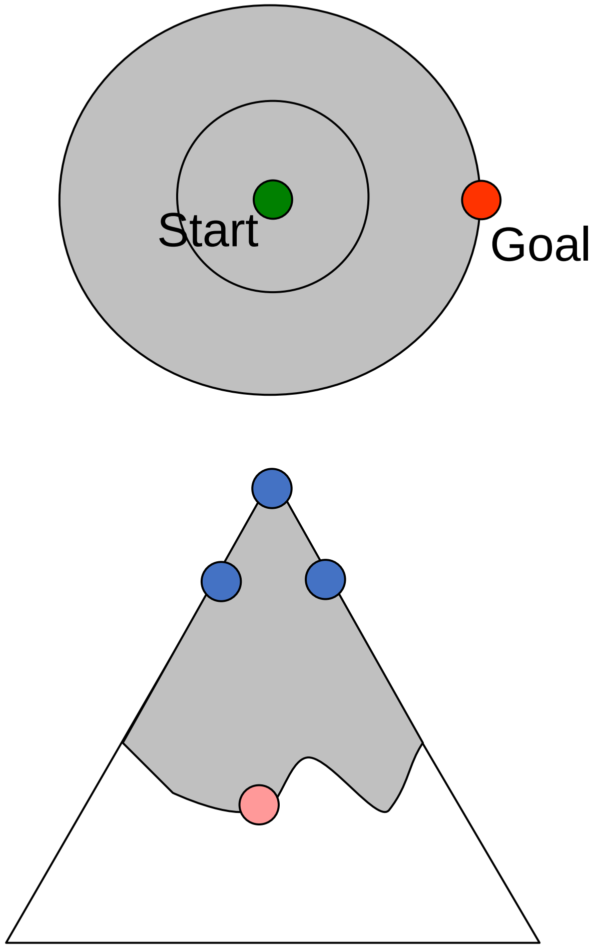
$$= [\text{cost}(s_1 \to s_2) + \cancel{h(s_2)} - h(s_1)] + [\text{cost}(s_2 \to s_3) + \cancel{h(s_3)} - \cancel{h(s_2)}] + \cdots$$
$$+ [\text{cost}(s_m \to s) + h(s) - \cancel{h(s_m)}]$$

$$= g(s) + h(s) - h(s_1)$$

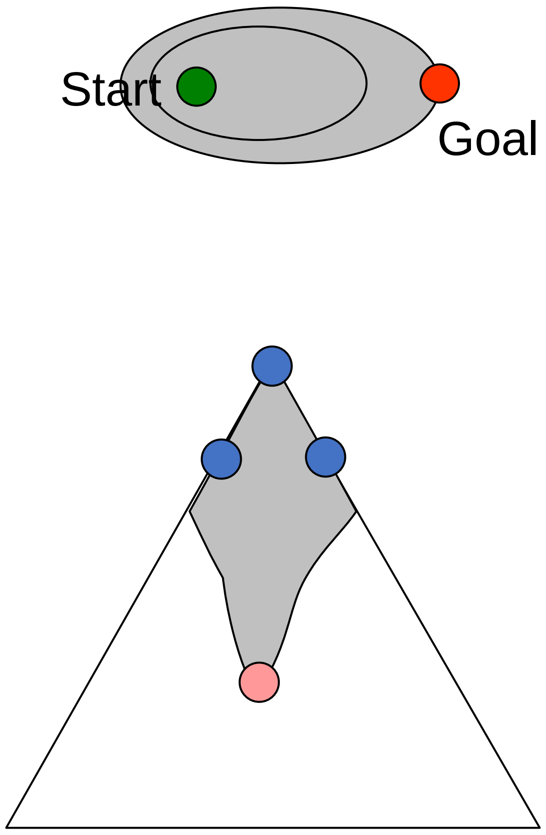


initial state

UCS vs. A*

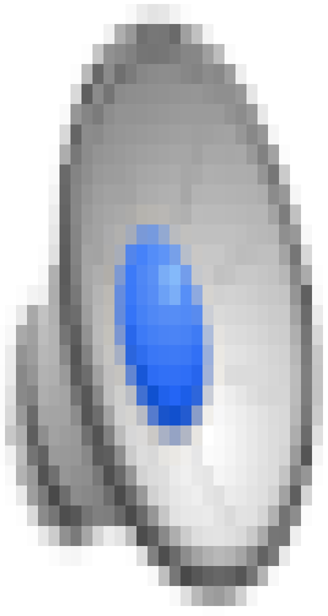


UCS

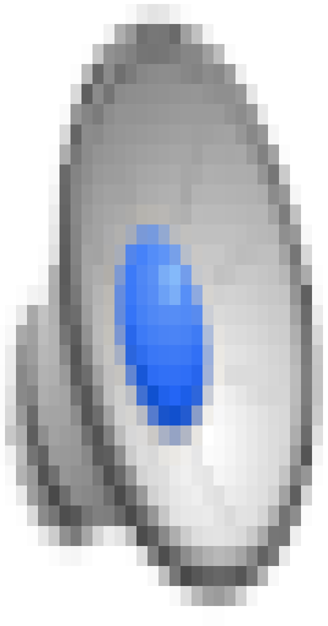


A*

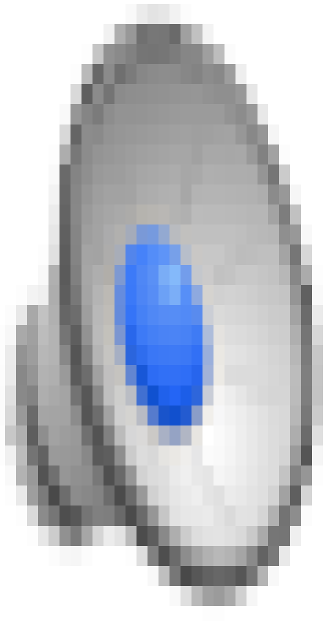
UCS / Greedy Best-First / A* ?



UCS / Greedy Best-First / A* ?



UCS / Greedy Best-First / A* ?



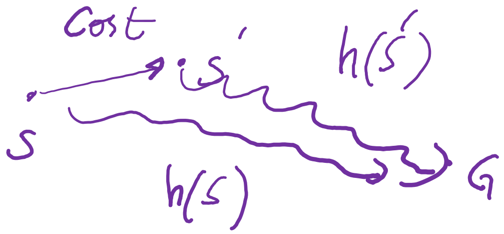
Comparison

Greedy

UCS

A*

The Optimality of A*



Definitions

A heuristic function h is called **consistent** if for all s and s'

$$h(s) \leq h(s') + \text{cost}(s \to s') \qquad \text{(triangle inequality)}$$

and $h(G) = 0$ for any goal state G.

The Optimality of A*

Theorem

If the heuristic function is **consistent**, then A* returns minimum-cost solution.

Proof A* is equivalent to UCS with $\widetilde{\mathrm{cost}}(s \to s') := \underbrace{\mathrm{cost}(s \to s') + h(s') - h(s)}_{\text{Consistency}} \geq 0$

Total modified cost of any path $s_1 \to s_2 \to \cdots \to s_m \to G$ is

$$\widetilde{\mathrm{cost}}(s_1 \to s_2) + \widetilde{\mathrm{cost}}(s_2 \to s_3) + \cdots + \widetilde{\mathrm{cost}}(s_m \to G)$$

$$= [\mathrm{cost}(s_1 \to s_2) + h(s_2) - h(s_1)] + [\mathrm{cost}(s_2 \to s_3) + h(s_3) - h(s_2)] + \cdots$$
$$+ [\mathrm{cost}(s_m \to G) + h(G) - h(s_m)]$$

$$= \mathrm{cost}(s_1 \to s_2) + \mathrm{cost}(s_2 \to s_3) + \cdots + \mathrm{cost}(s_m \to G) - h(s_1) + h(G)$$

Since $\widetilde{\mathrm{cost}}(s \to s') \geq 0$ by the consistency of h, A*'s optimality follows UCS's optimality under non-negative cost.

The Optimality of A*

Definitions

A heuristic function h is called **admissible** if for all s

$$0 \leq h(s) \leq h^{\star}(s)$$

where $h^{\star}(s)$ is the true minimum distance from s to goal.

Theorems

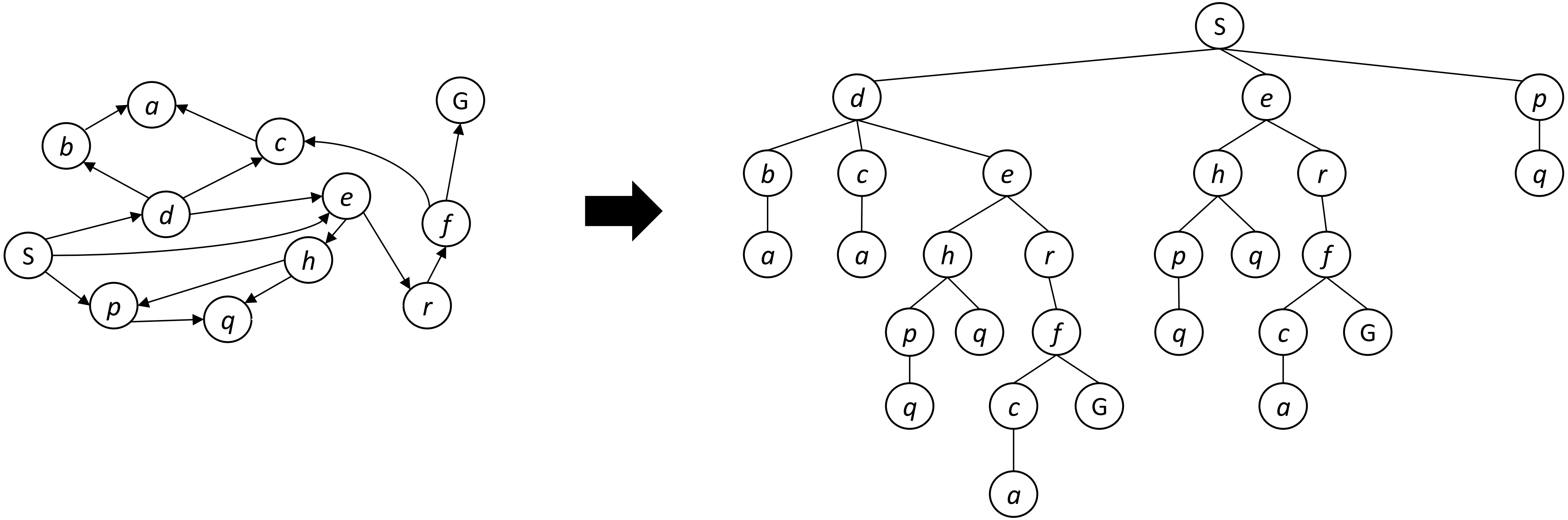
If h is consistent, then it is also admissible.

If the heuristic function is **admissible** and **the graph is a tree,** then A* returns minimum-cost solution.

The Optimality of A*

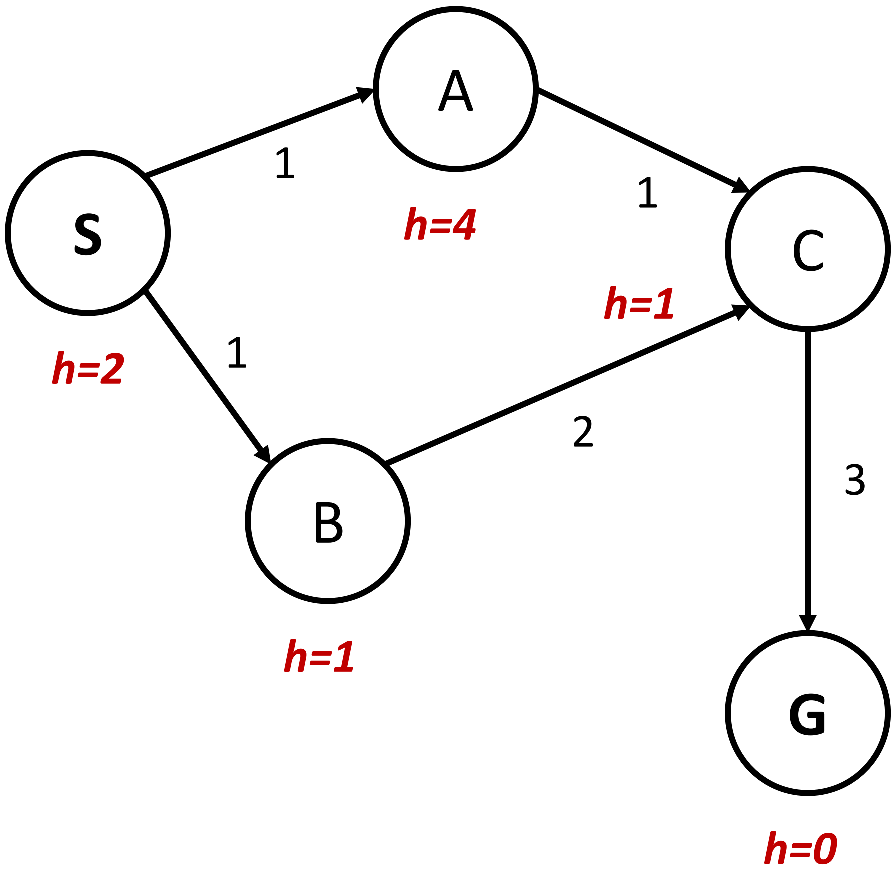
For general graphs, we can **treat it like a tree** by omitting the condition **“if not Reached[s’]”** in the graph search algorithm (like what we did for memory-efficient DFS).

This allows us to apply the theorem in the previous page.

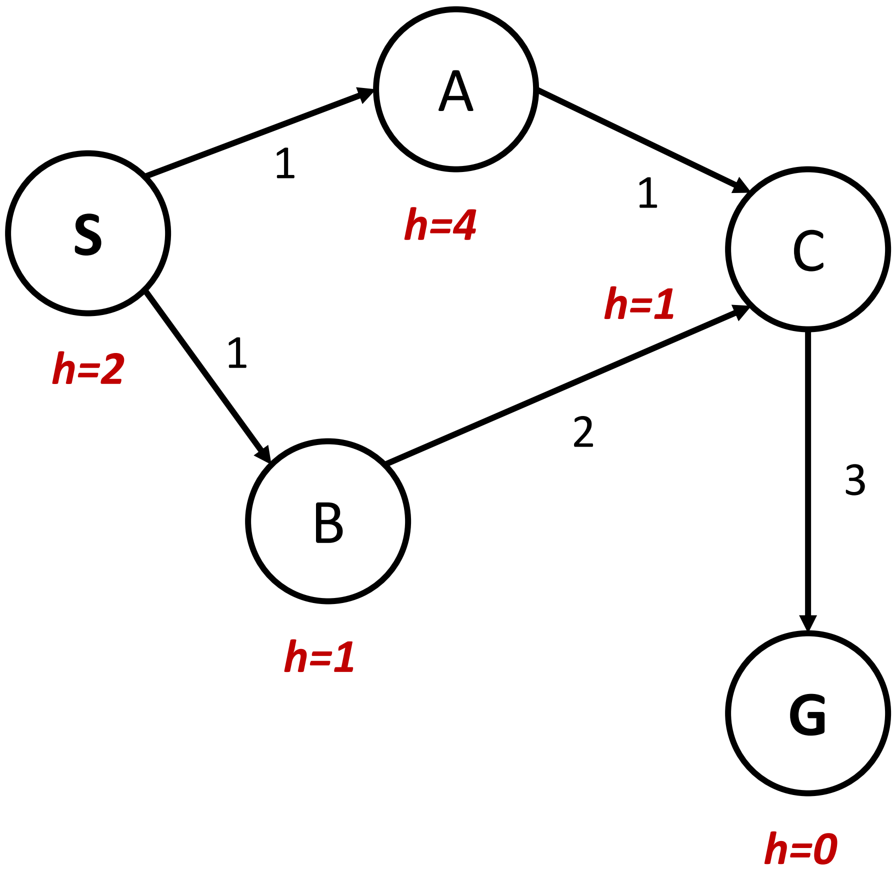


Admissible but not Consistent

$\cancel{h(A) \le Cost(A \to C) + h(C)}$



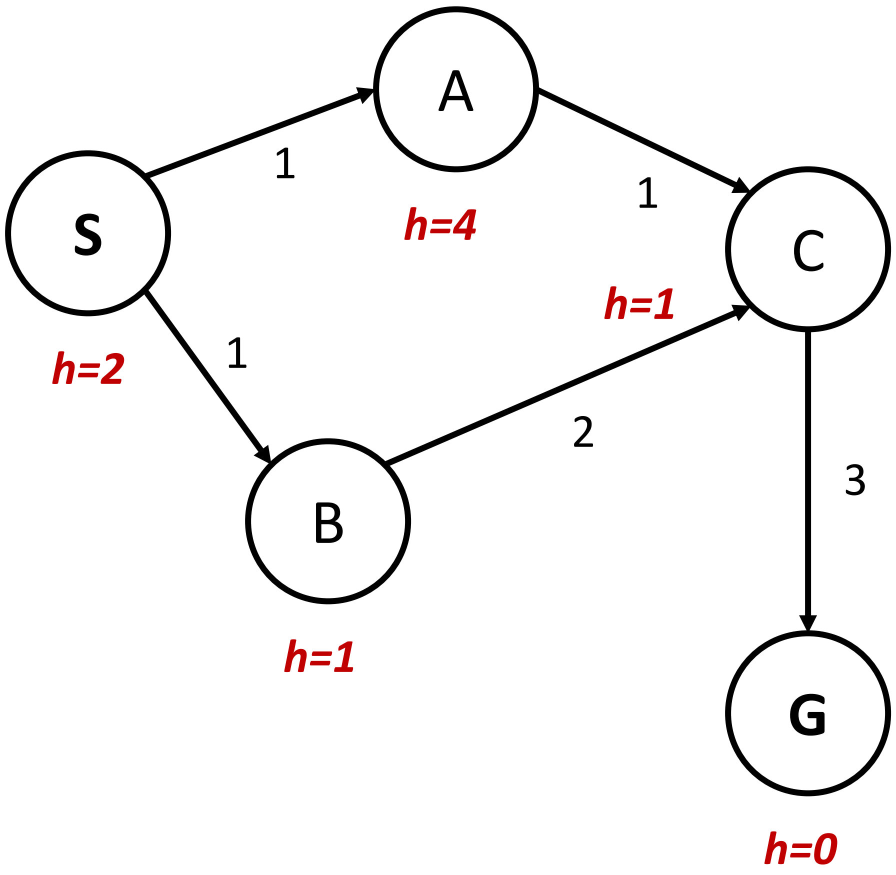
Admissible but not Consistent

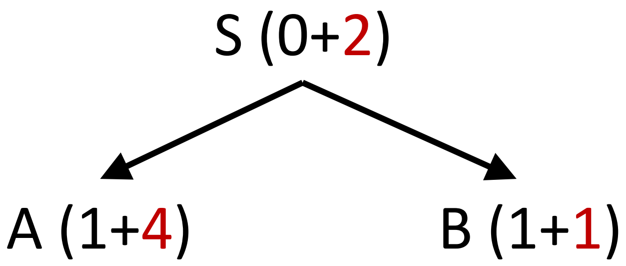


S (0+2)

Expanded = { }

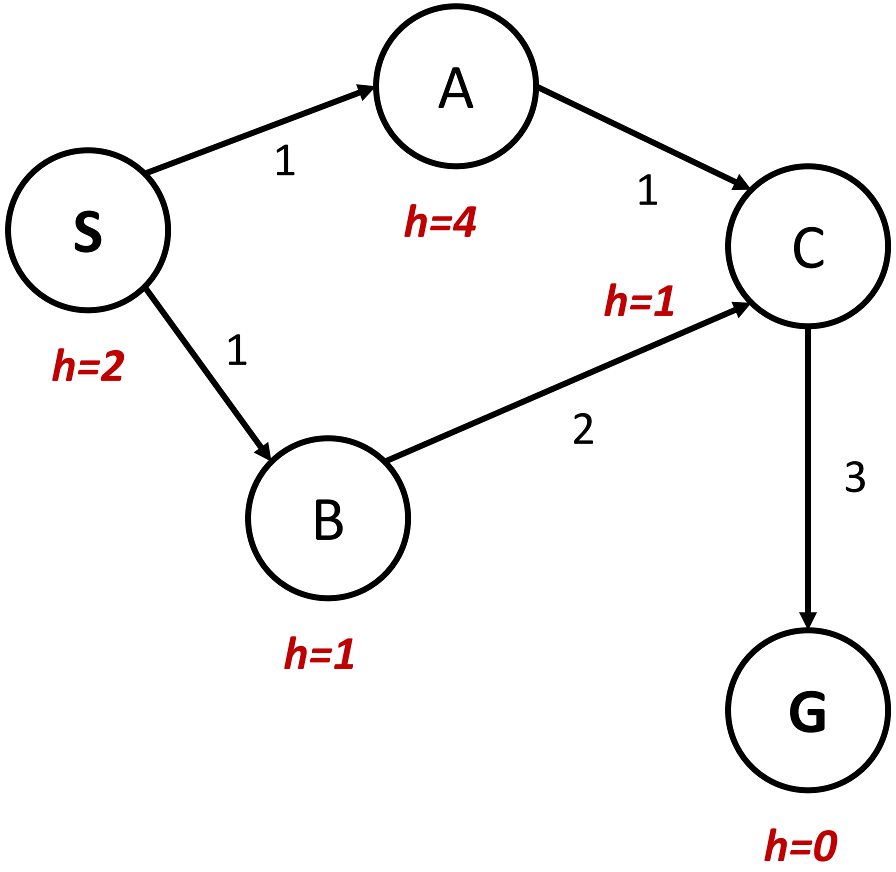
Admissible but not Consistent

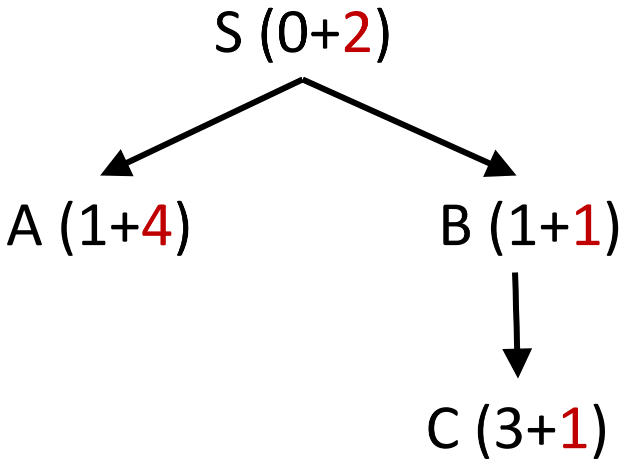




Expanded = { S }

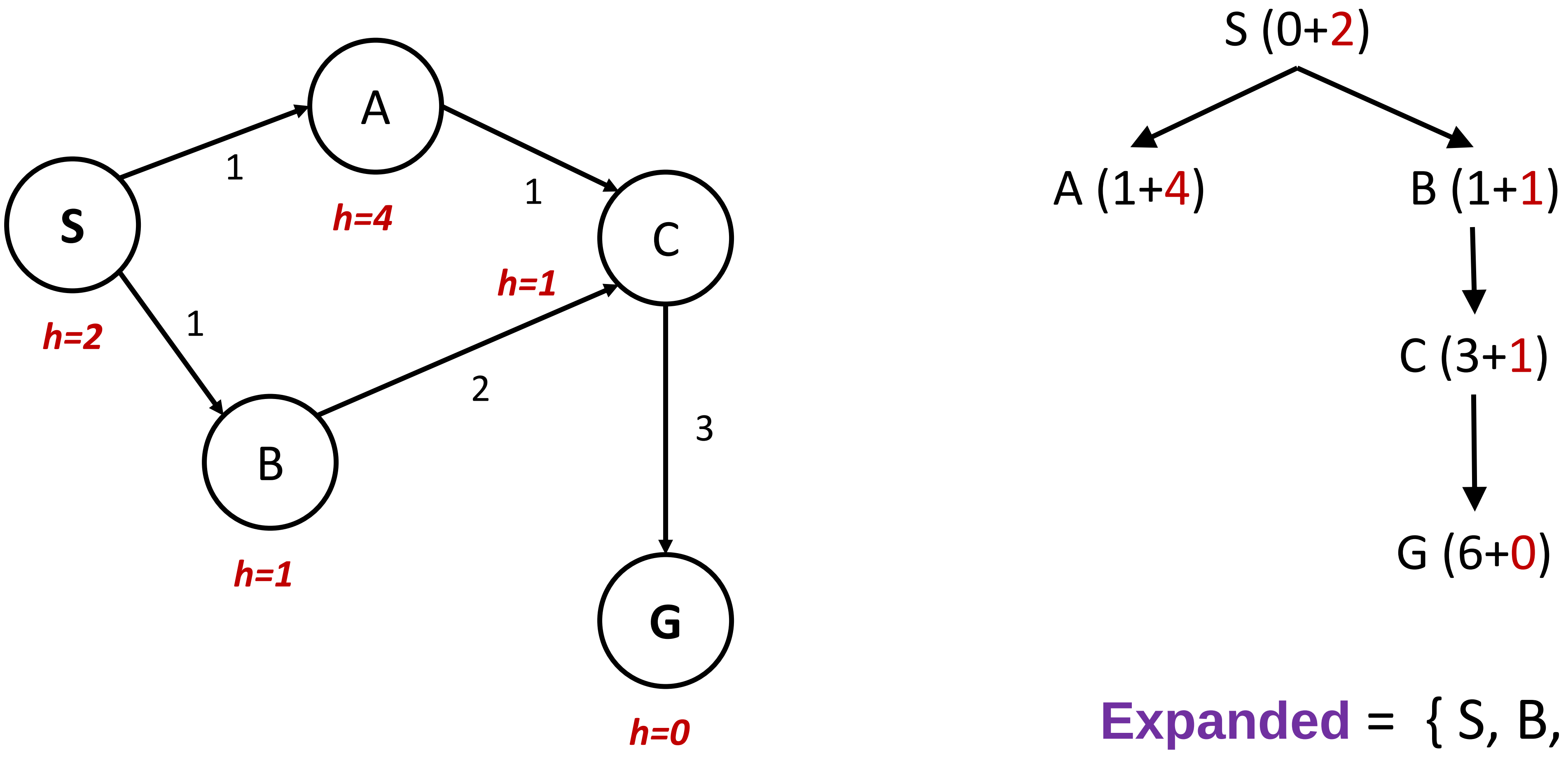
Admissible but not Consistent





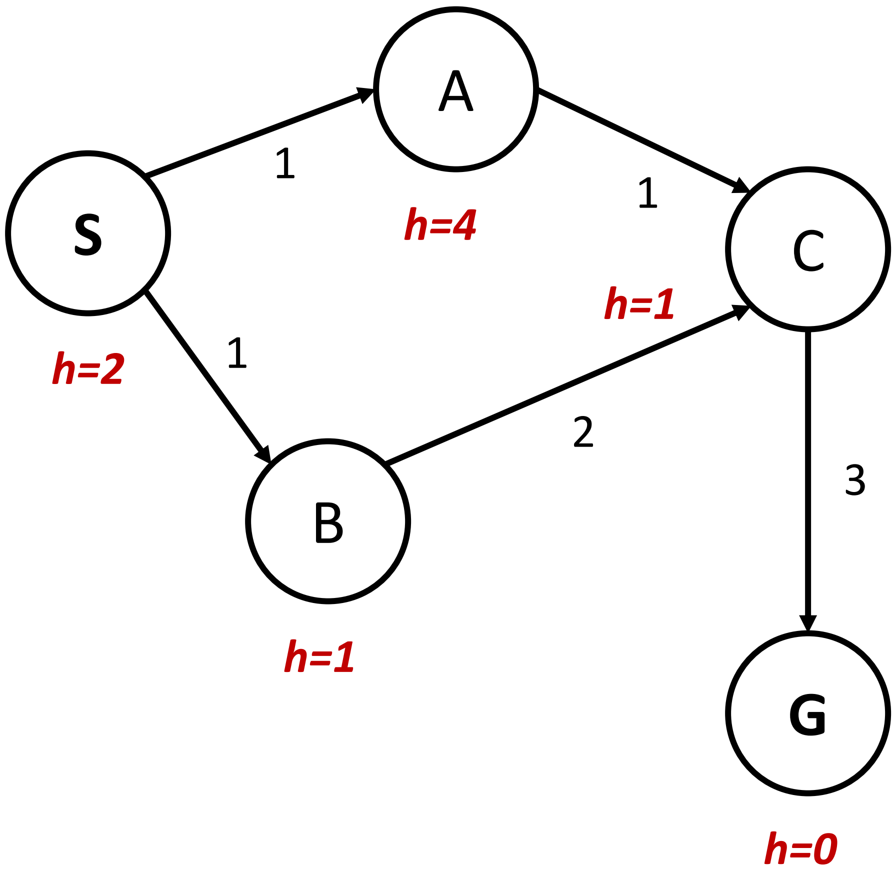
Expanded = { S, B }

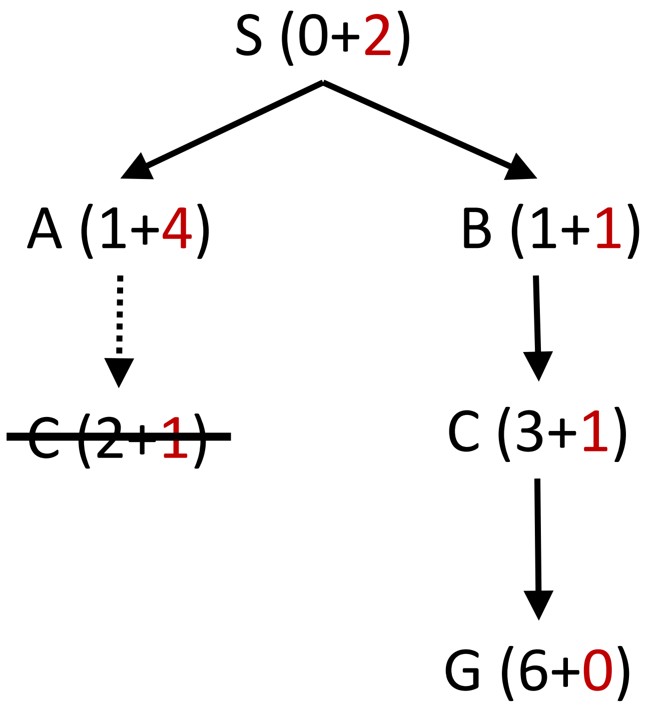
Admissible but not Consistent



Expanded = { S, B, C }

Admissible but not Consistent



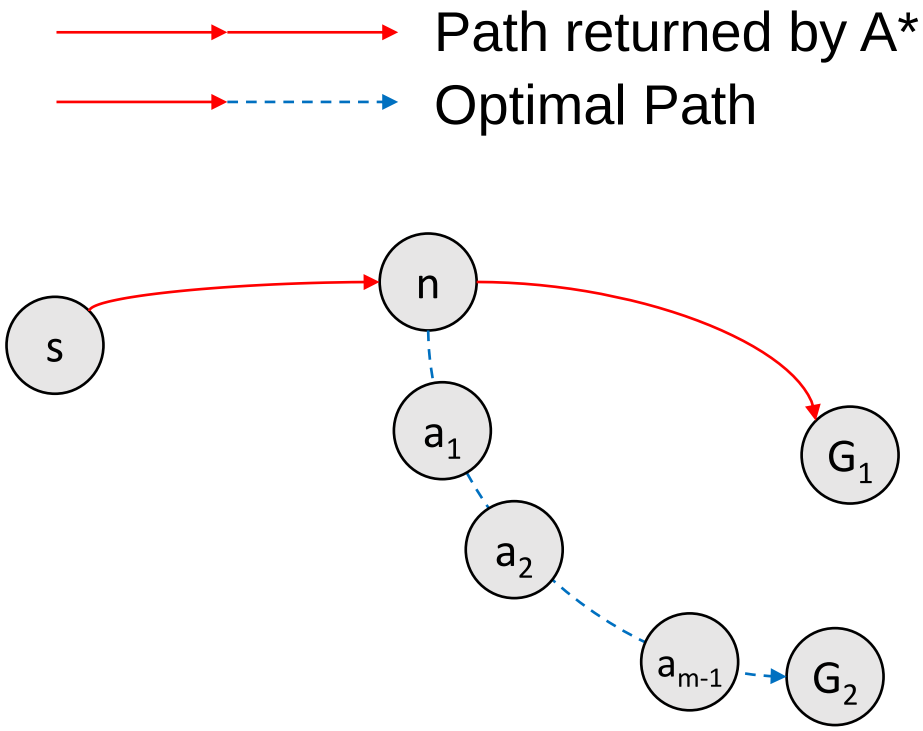


Expanded = { S, B, C }

The Optimality of A*

If **admissible** + **tree**, then A* returns minimum-cost solution.

Proof by contradiction



Assume $\text{dist}(s, G_2) < \text{dist}(s, G_1)$

By the tree structure, every node can only be reached by its unique predecessor

- n expanded earlier than G_1
- a_1 expanded earlier than G_1

$$\because f(a_1) = \boxed{g(a_1) + h(a_1)} \le g(a_1) + h^\star(a_1)$$
$$= \text{dist}(s, G_2) < \text{dist}(s, G_1) \le g(G_1) + \underbrace{h(G_1)}_{=0} = f(G_1)$$

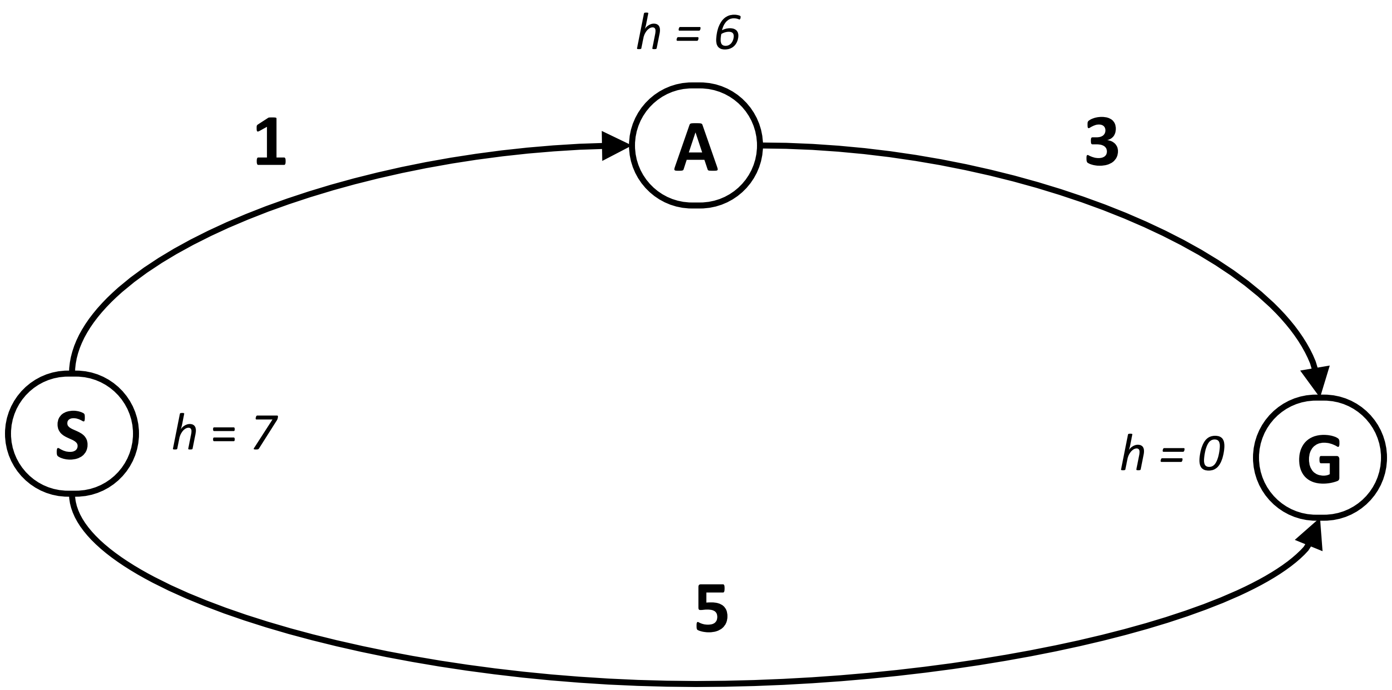
- a_2 expanded earlier than G_1

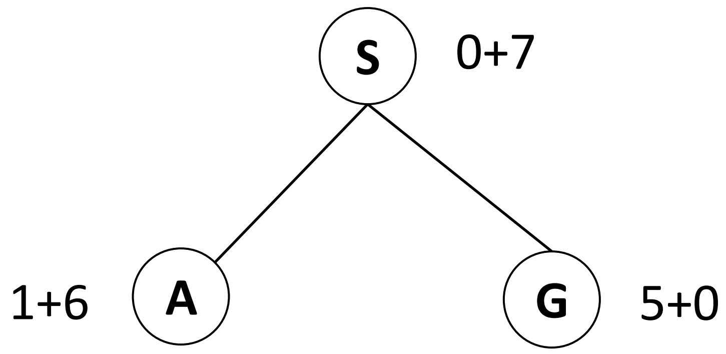
$$\because f(a_2) = g(a_2) + h(a_2) \le g(a_2) + h^\star(a_2)$$
$$= \text{dist}(s, G_2) < f(G_1)$$

$$\vdots$$

- a_{m-1} expanded earlier than G_1
- G_2 expanded earlier than G_1

Why “Admissibility”? (with an example)





We hope that **the nodes on the shortest path** are expanded before **G** is expanded.

$g(A) + h(A) \leq g(A) + h^*(A)$ $\qquad$ $g(G)$

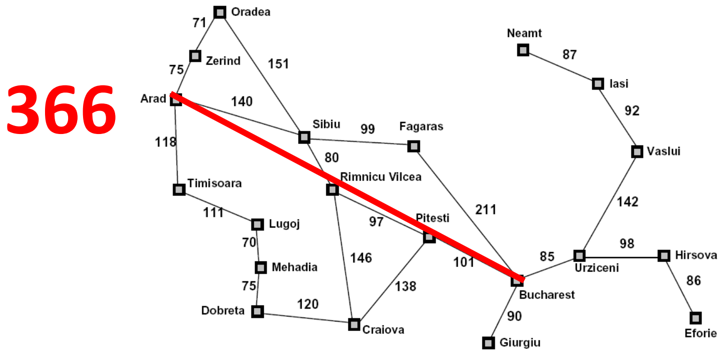
Shortest path length from S to G $\qquad$ Some path length from S to G

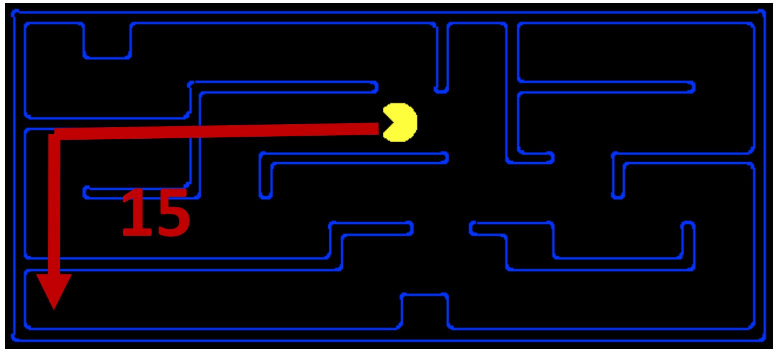
Admissibility ensures $g(A) + h(A) \leq g(G)$

Creating Admissible/Consistent Heuristics

- Most of the work in solving hard search problems is in coming up with admissible/consistent heuristics.

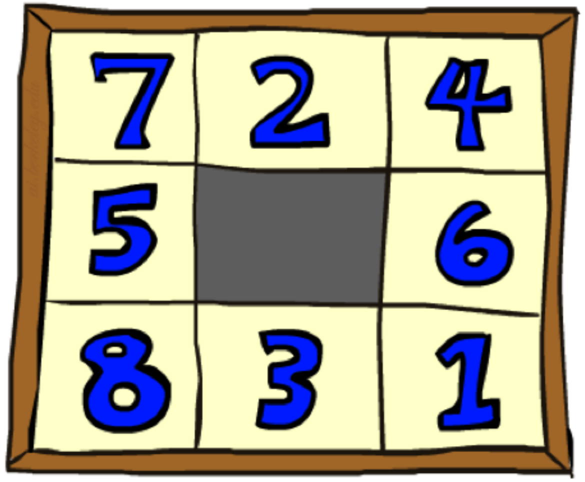
- Often, admissible/consistent heuristics are solutions to *relaxed problems*, where new actions are available.



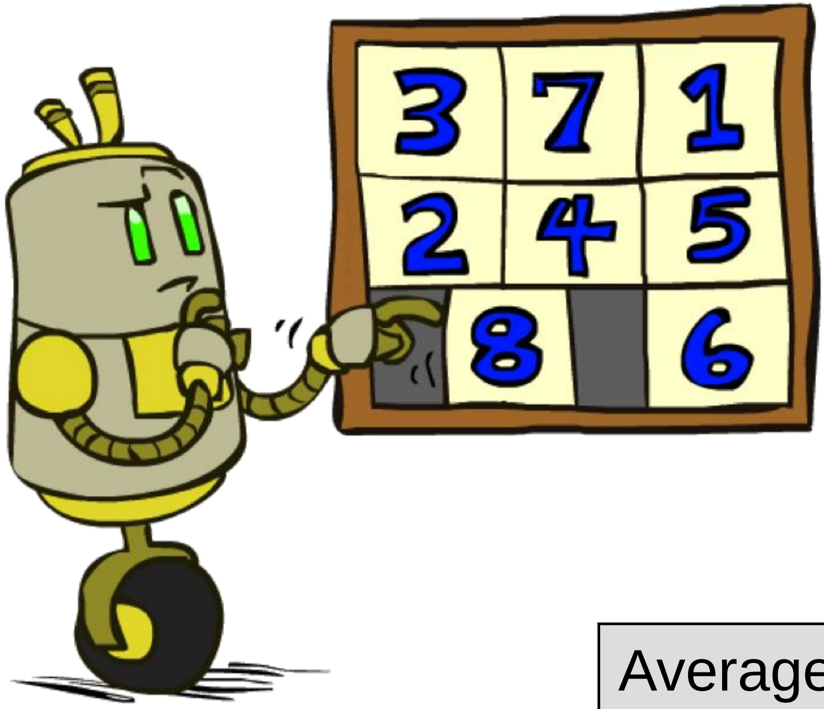


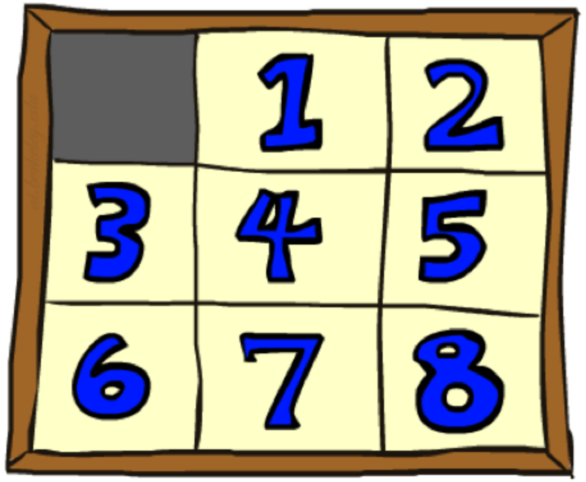
- Inadmissible heuristics are often useful too.

Example: 8 Puzzles



Start State





Goal State

	Average nodes expanded when the optimal path has…		
	…4 steps	…8 steps	…12 steps
UCS	112	6,300	3.6×10^6
#wrong tile	13	39	227
Manhattan	12	25	73

Homework 1

Xuhui Kang, Haolin Liu

Deadline: 11:59PM, September 16

Homework 1

1. Choice Questions (10 points)
    a. 14 questions.
    b. Choice questions are 10 points in total and distributed evenly

2. Program Questions (25 points)

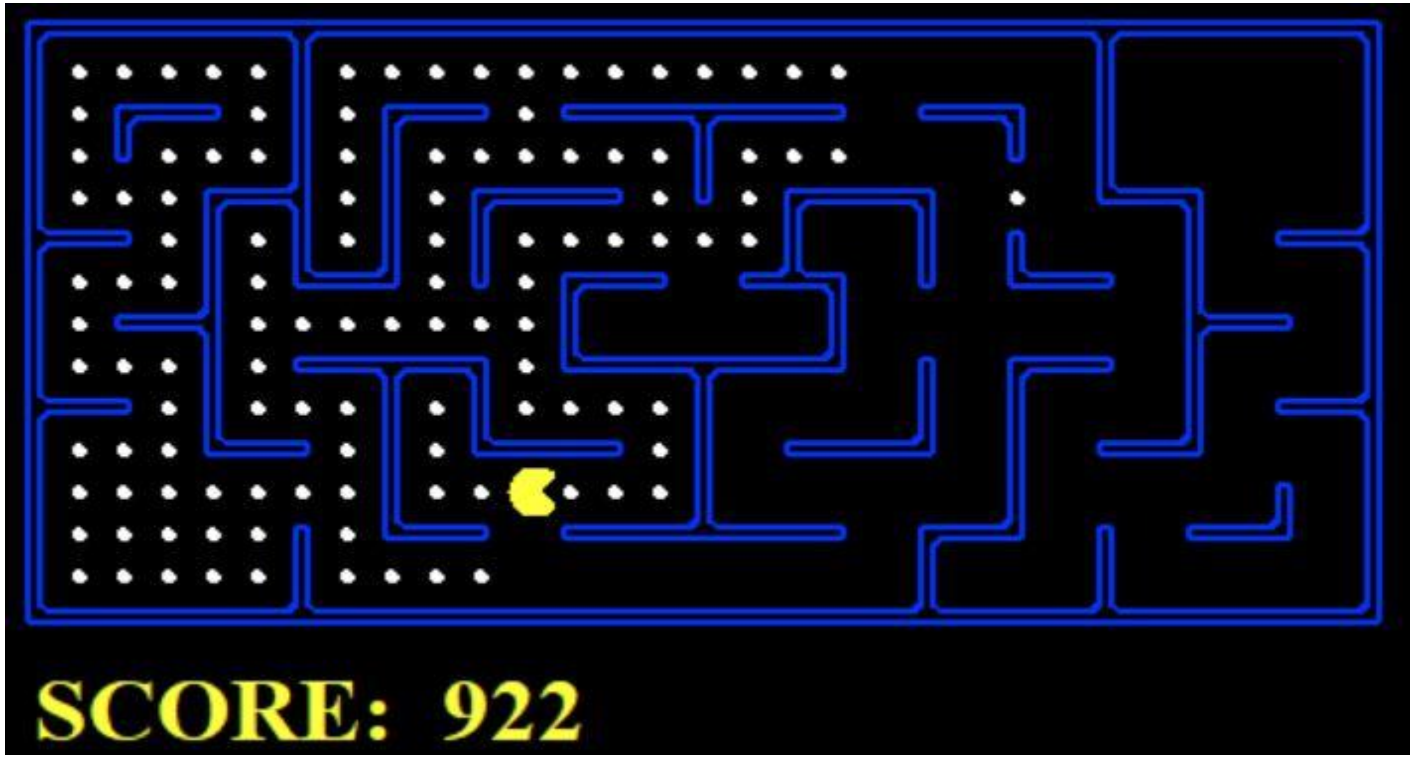
Homework 1: Choice Questions

- Each question may be either single-choice or multiple-choice. Read carefully and select your answers accordingly.
- Grading:
  - Full Credit: If all correct options are selected.
  - Partial Credit: If only some correct options are selected, with no wrong options chosen.
  - No Credit: If any incorrect options are selected.
- Submission: Please answer directly on Gradescope. No need to submit a separate PDF.
- Scores and correct answers will not be released immediately after submission.

Homework 1: Coding

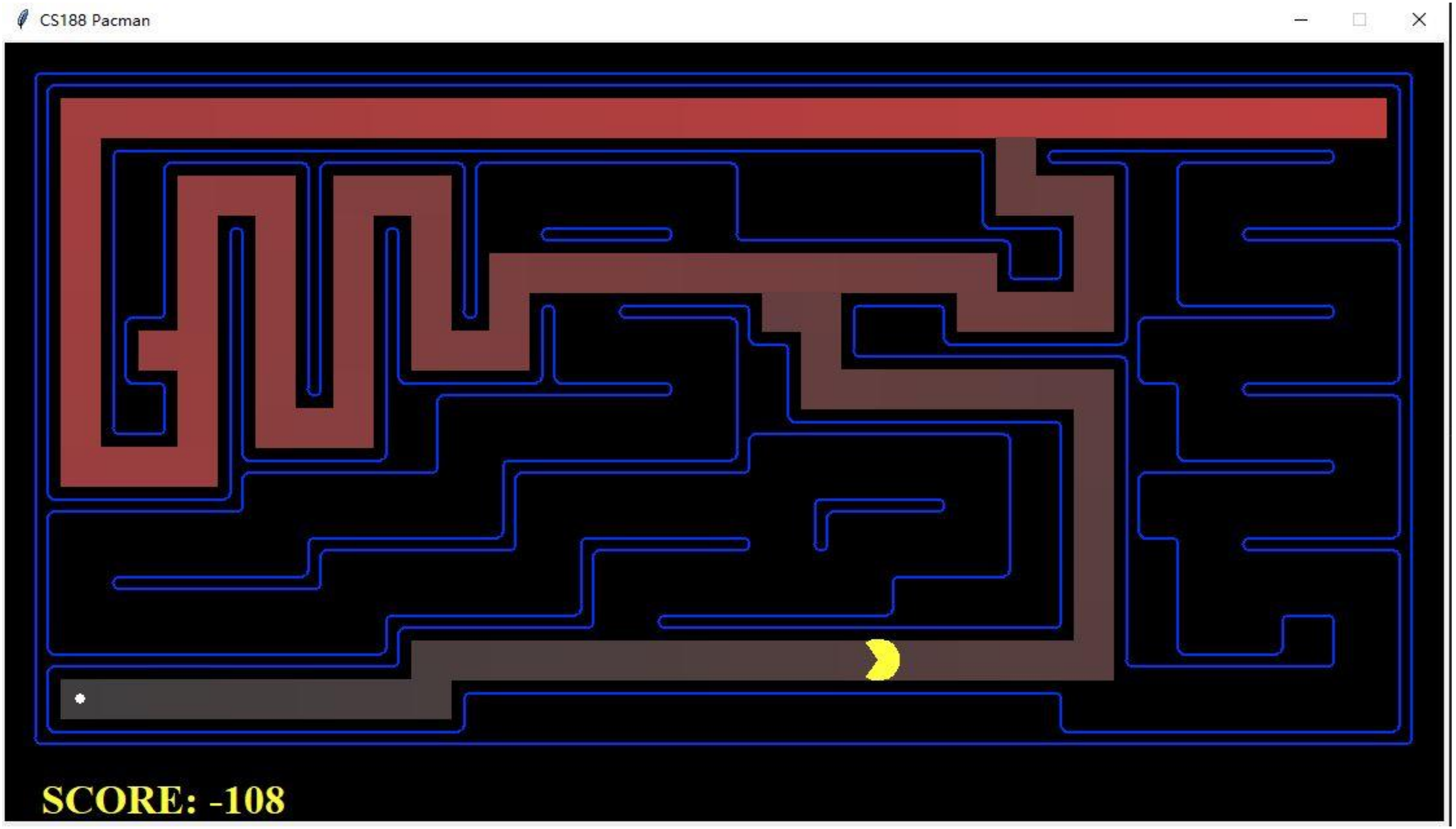
In this problem, you are going to help Pacman find paths in a maze world by focusing on implementing search algorithms.

Autograder is given both offline and online in GradeScope. Your grade in gradescope is the final grade.



Homework 1: Coding

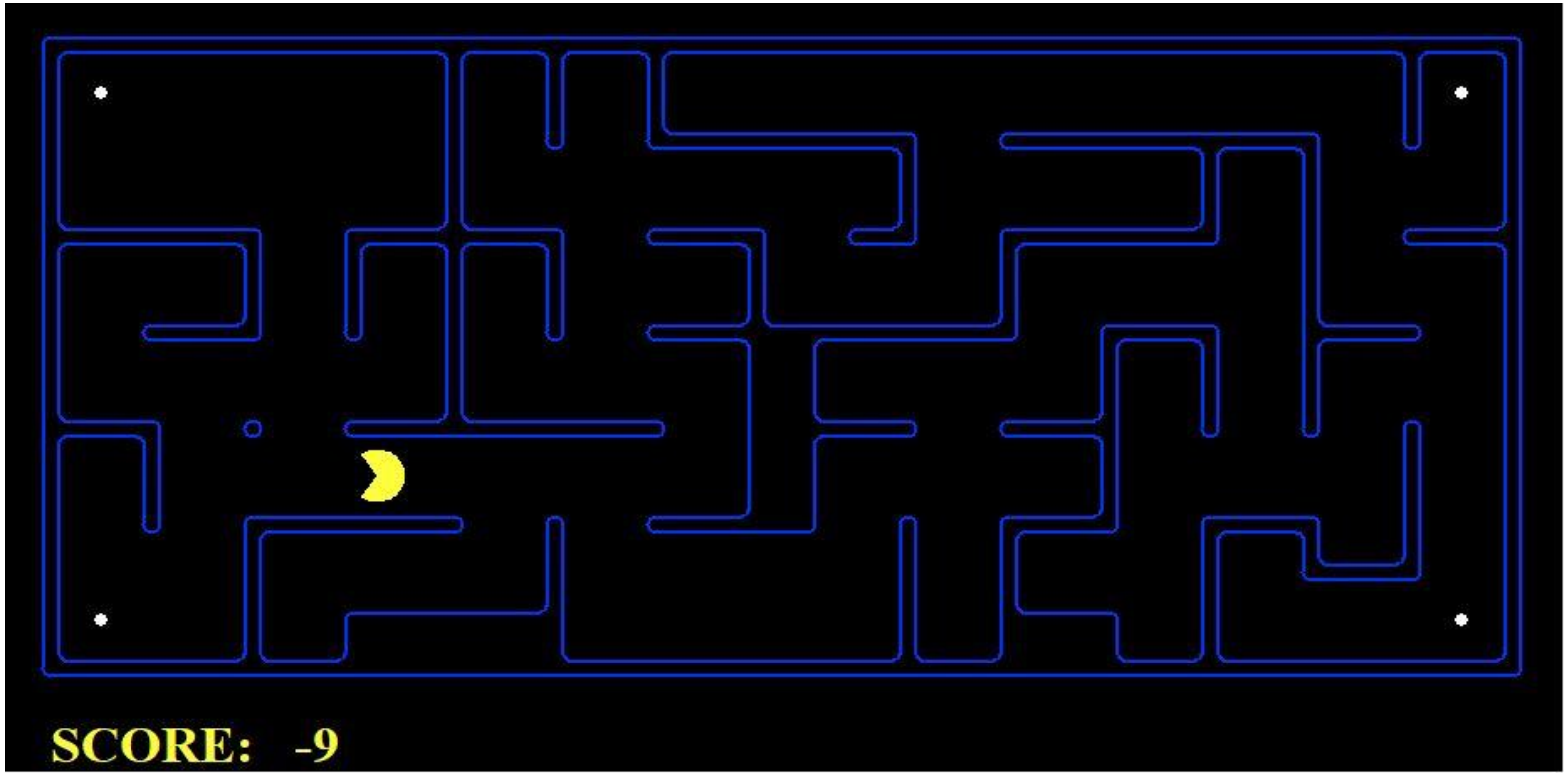
Question 1 -- 4: Implement Depth First Search, Breadth First Search, Uniform Cost Search, and A* algorithm. Your goal is to reach a target area.



Homework 1: Coding

Question 5 : The goal of this question is to visit all four corners rather than reaching a destination state.

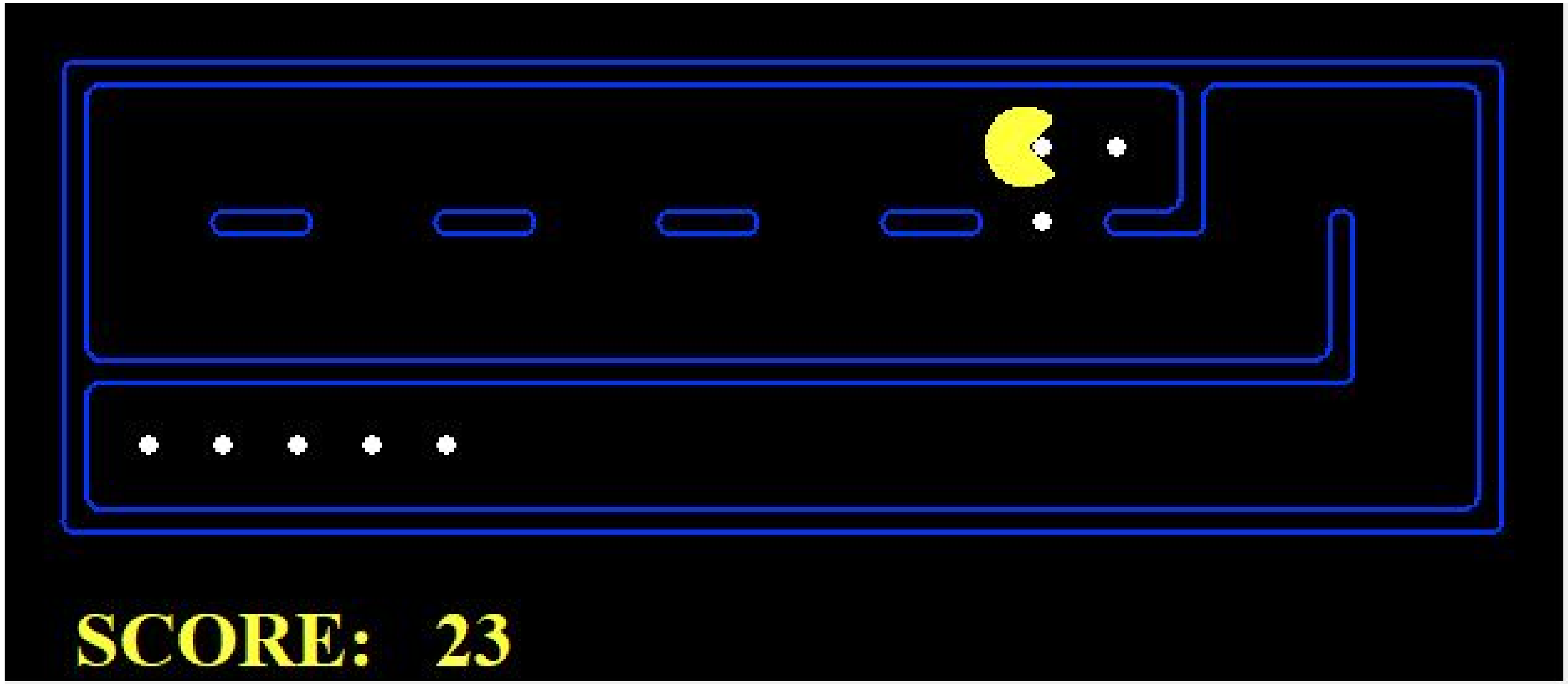
Question 6: In the corner search problem, you need to implement an admissible heuristic function for A* algorithm.



You may need to consider more complex state space, which not only contain possible coordinate of the Pac-Man, but tracking the visitation of corners as well.

Homework 1: Coding

Question 7 : The goal of this question is to find a way to eat all of the pellets in the maze. The position of the pellets is known to the pacman.



Homework 1: Coding

Question 8 : The goal of this question is to eat the **closest** dot (pellet) by finding the path to it.