Artificial Intelligence

Lec 2: Uninformed Search

Pratik Mazumder

Content taken from various sources including
CS188 Intro to AI at UC Berkeley

Search Problems

- Framework for solving problems

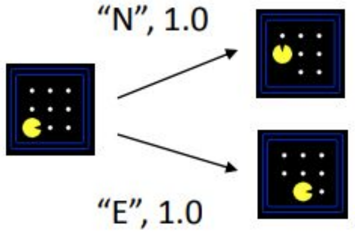
- A search problem consists of:
  - A **state space**: List of all possible states, i.e., all possible configurations of elements/conditions in the world or environment, e.g., passing one pen across the class, situations in a pacman game.

  - A **successor function** (with actions, costs): Action may also change the state.

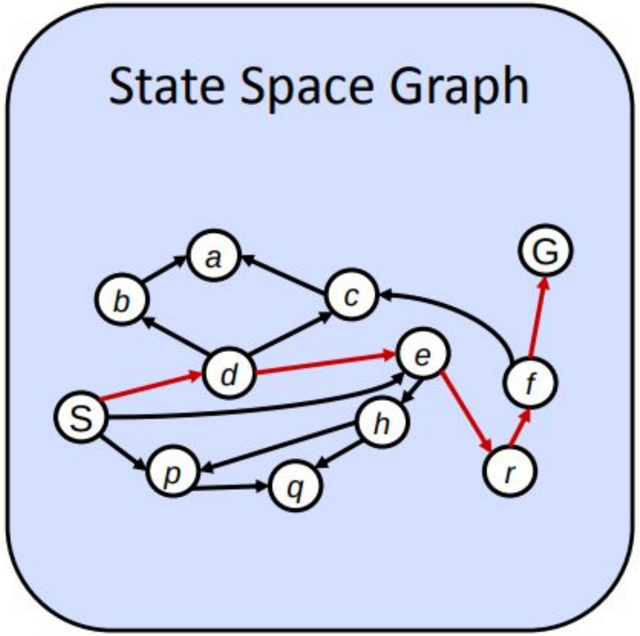
  - A **start state**

  - A **goal test**: There can be multiple goal states

- A solution is a sequence of actions (a plan) which transforms the start state to a goal state

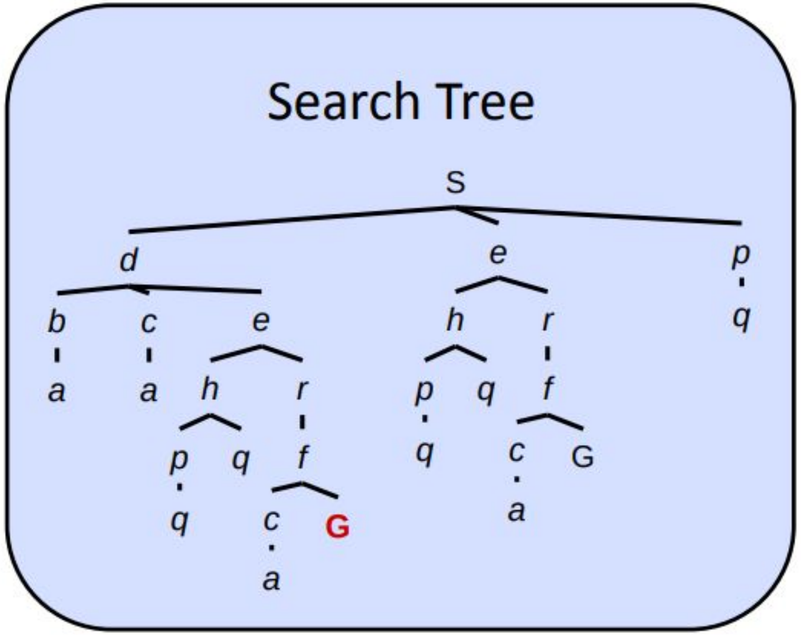


State Space Graphs vs. Search Trees

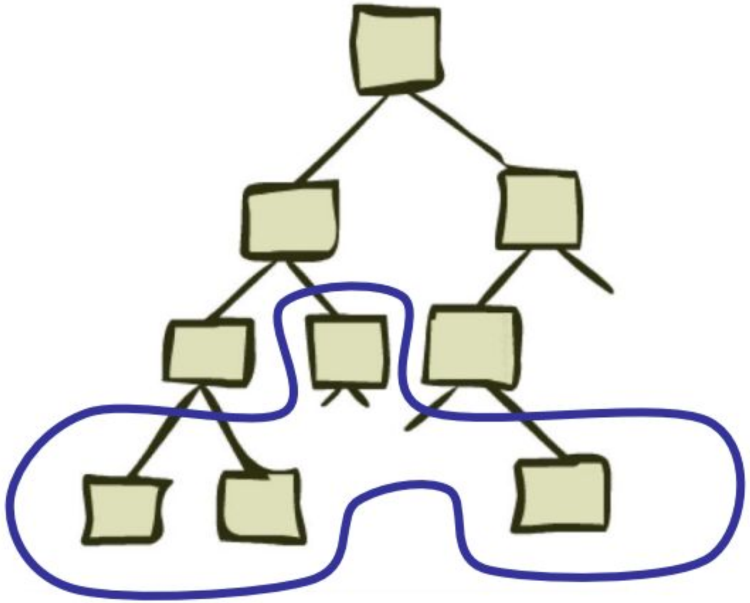


Each NODE in the search tree is an entire PATH in the state space graph.

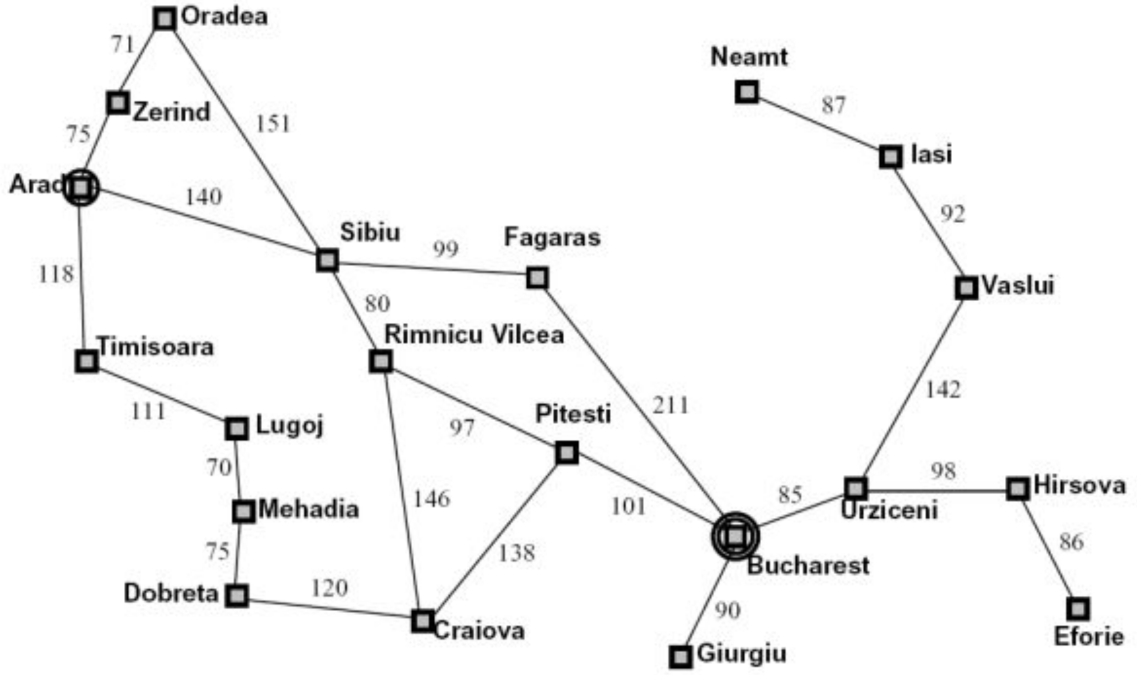
We construct both on demand – and we construct as little as possible.



Tree Search

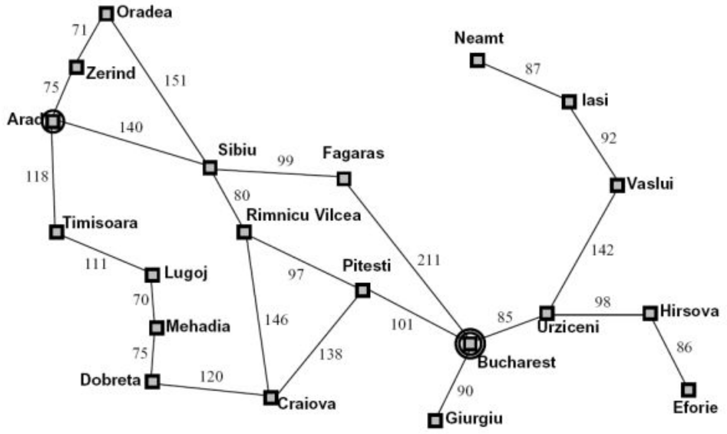


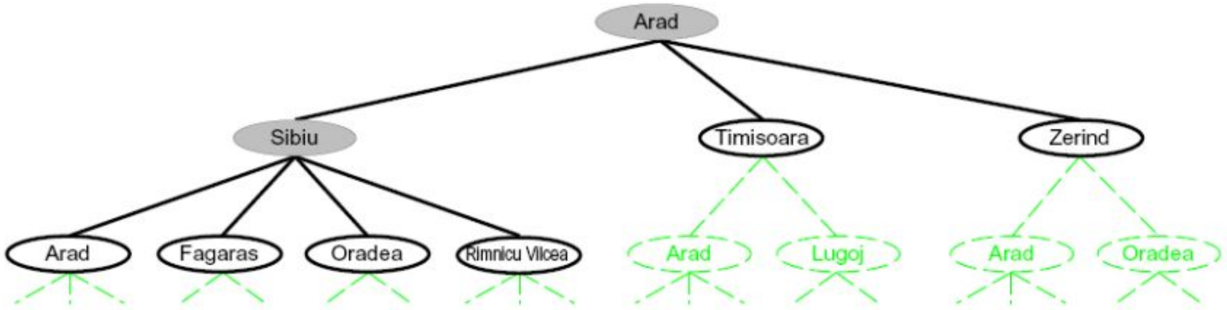
Searching with a Search Tree: Search Problem Example



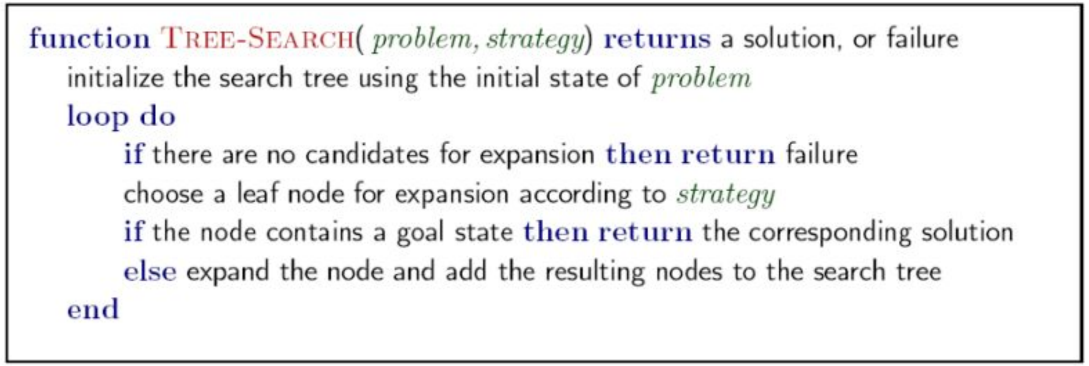
Searching with a Search Tree: Search Problem Example

- Search (for goal) using a Search Tree:
  - Expand out potential plans (tree nodes)
  - Maintain a fringe of partial plans under consideration
  - Try to expand as few tree nodes as possible





General Tree Search



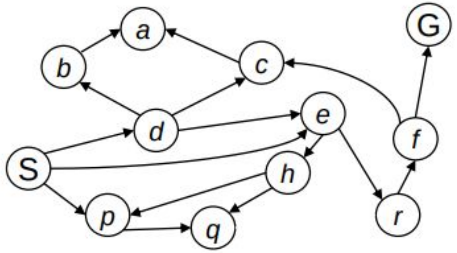
```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

Important ideas:

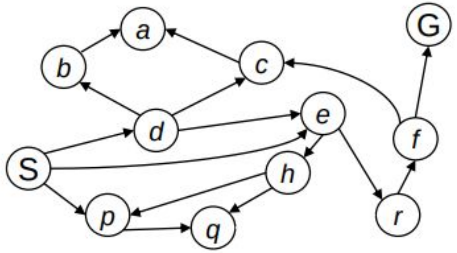
- Fringe (List of partial plans)
- Expansion
- Exploration strategy

Main question: which fringe nodes to explore?

Example: Tree Search



Example: Tree Search

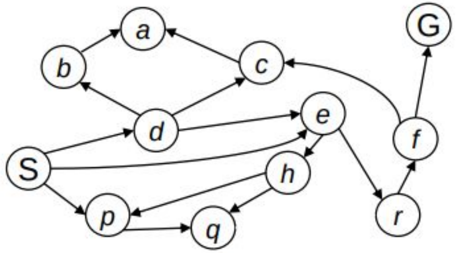


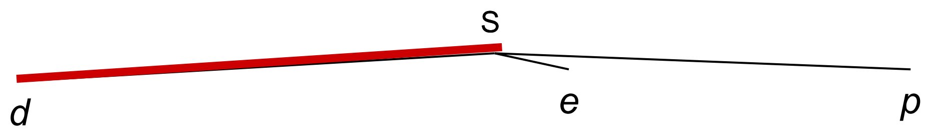
Fringe

s

S

Example: Tree Search

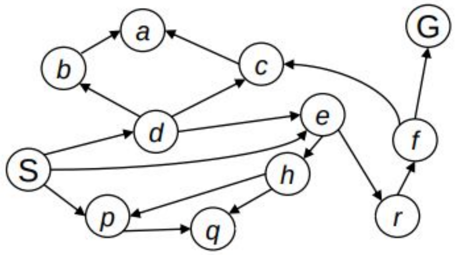




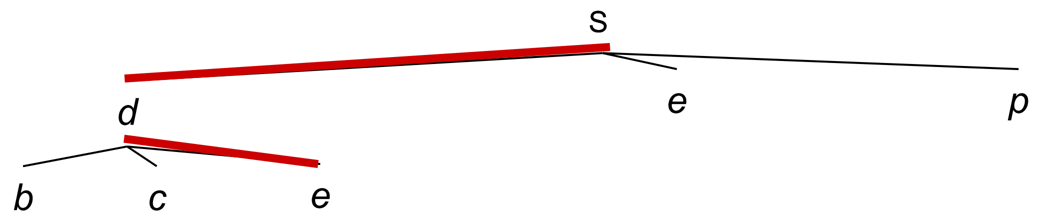
Fringe

~~s~~
s→d
s→e
s→p

Example: Tree Search

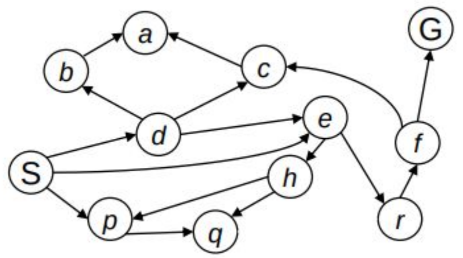


Fringe

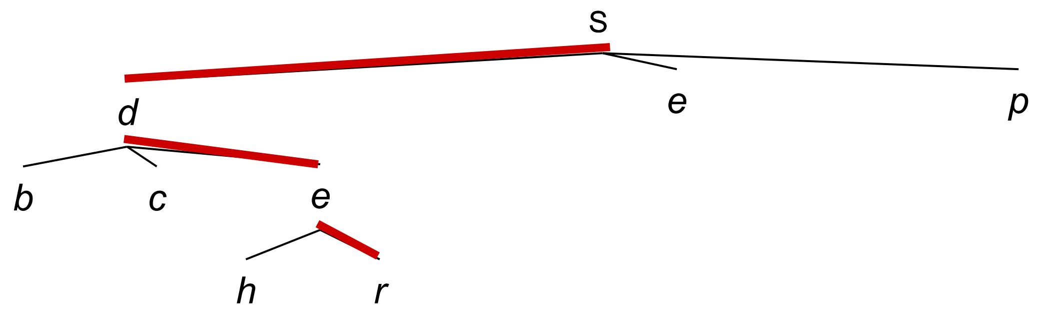


~~s~~
~~s→d~~
s→e
s→p
s→d→b
s→d→c
s→d→e

Example: Tree Search

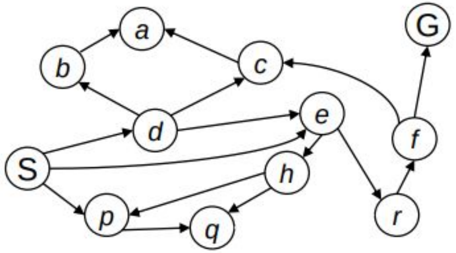


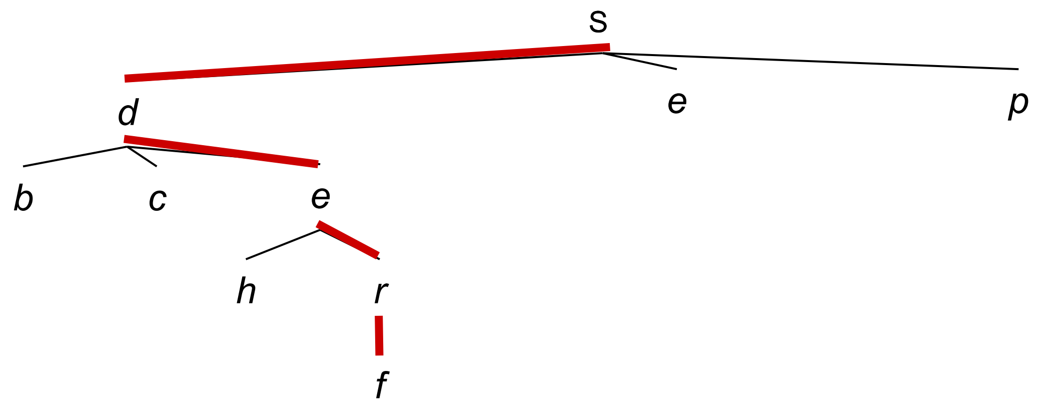
Fringe



~~s~~
~~s→d~~
s→e
s→p
s→d→b
s→d→c
~~s→d→e~~
s → d → e → h
s → d → e → r

Example: Tree Search

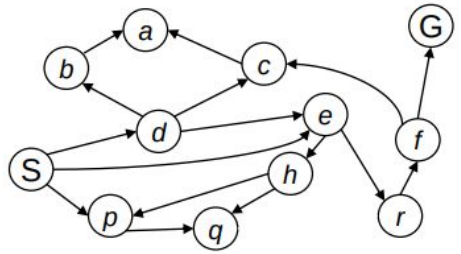


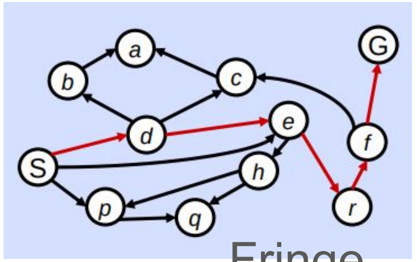


Fringe

~~s~~
~~s→d~~
s→e
s→p
s→d→b
s→d→c
~~s→d→e~~
s → d → e → h
~~s → d → e → r~~
s → d → e → r → f

Example: Tree Search





Fringe

~~s~~

~~s→d~~

s→e

s→p

s→d→b

s→d→c

~~s→d→e~~

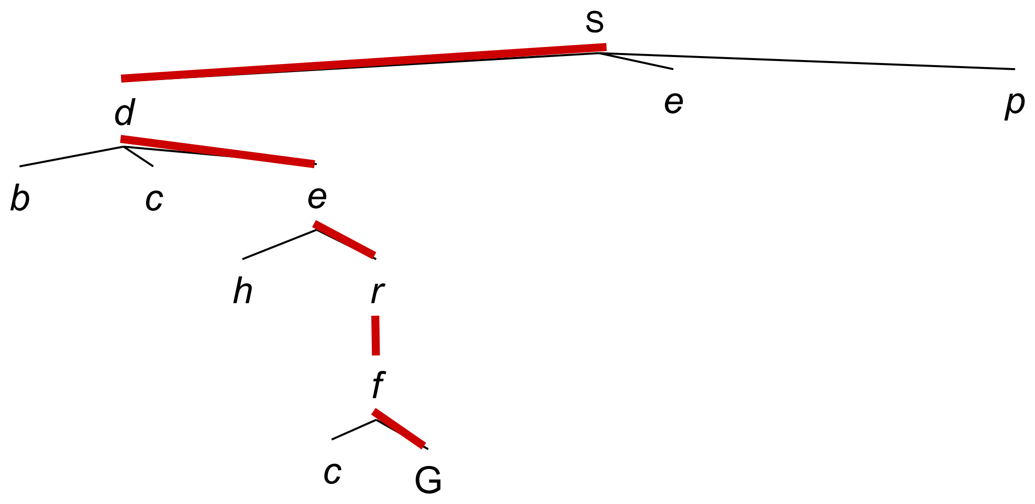
s → d → e → h

~~s → d → e → r~~

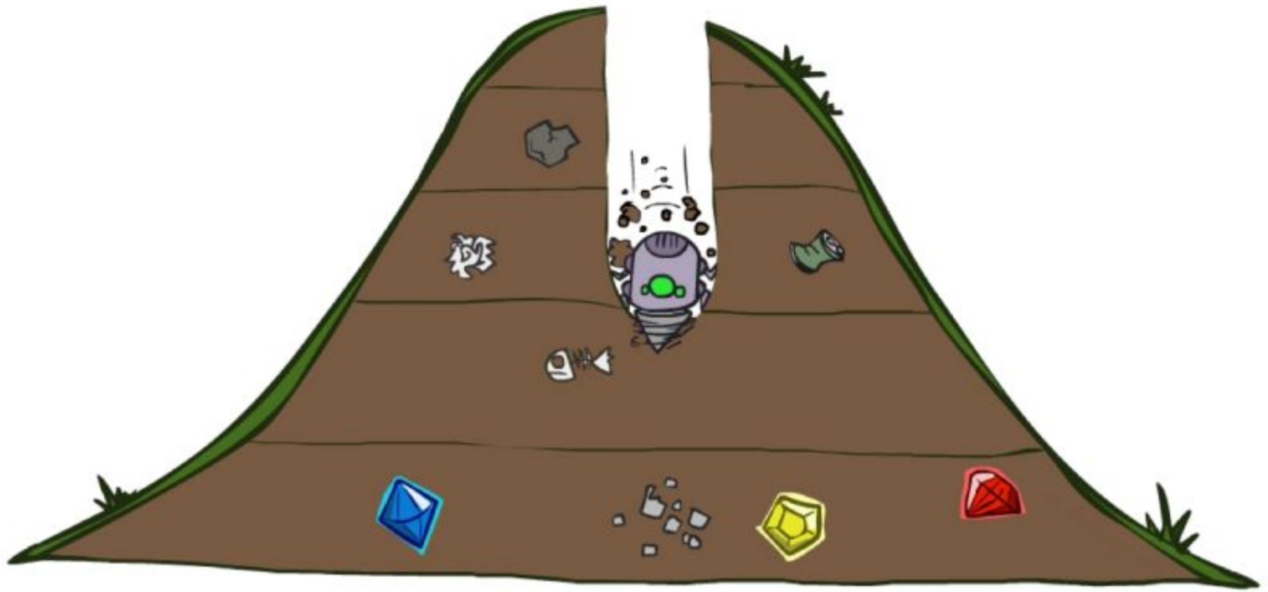
~~s → d → e → r → f~~

s → d → e → r → f → c

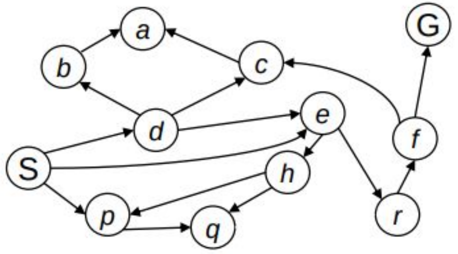
s → d → e → r → f → G



Depth First Search

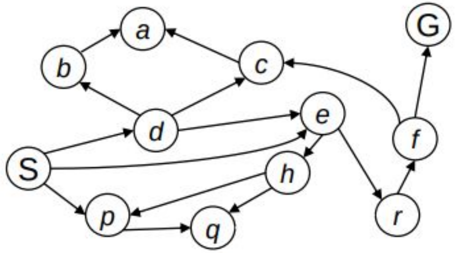


Depth First Search



Strategy: expand a **deepest node first**

Depth First Search

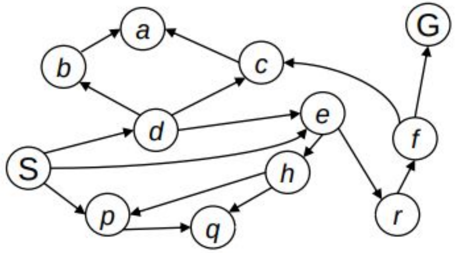


Fringe

s

S

Depth First Search

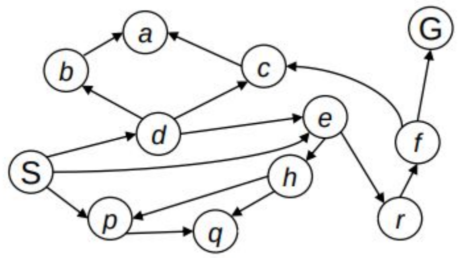


$$\begin{tikzcd} & s \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\ d & e & p \end{tikzcd}$$

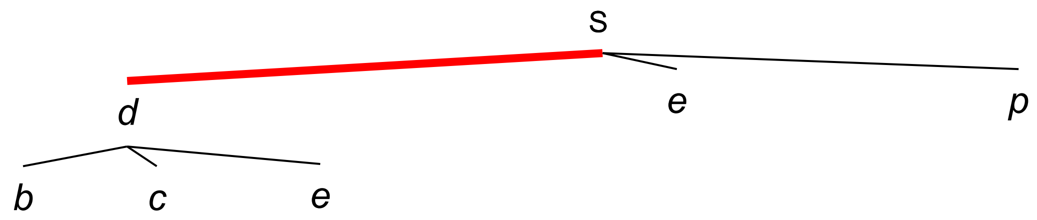
Fringe

~~s~~
s→d
s→e
s→p

Depth First Search

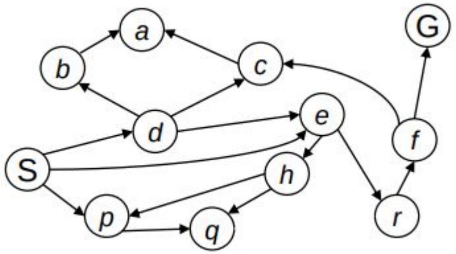


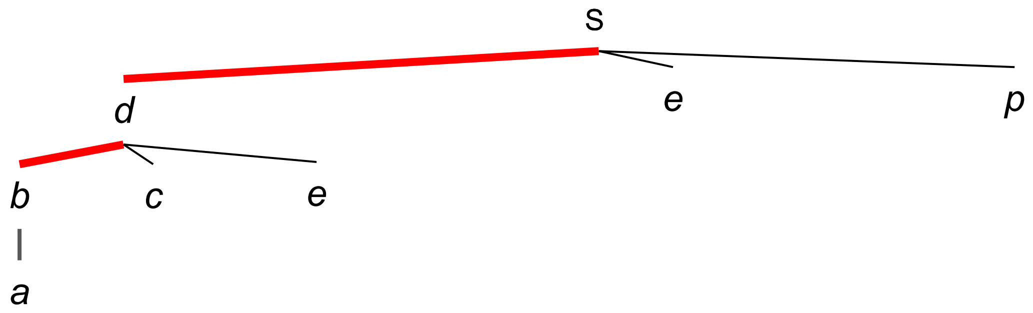
Fringe



~~s~~
~~s→d~~
s→e
s→p
s→d→b
s→d→c
s→d→e

Depth First Search

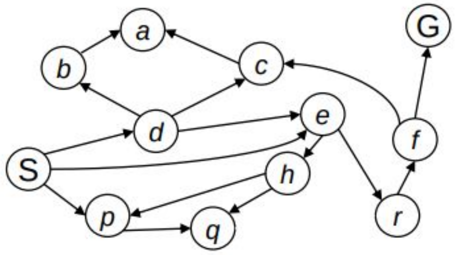




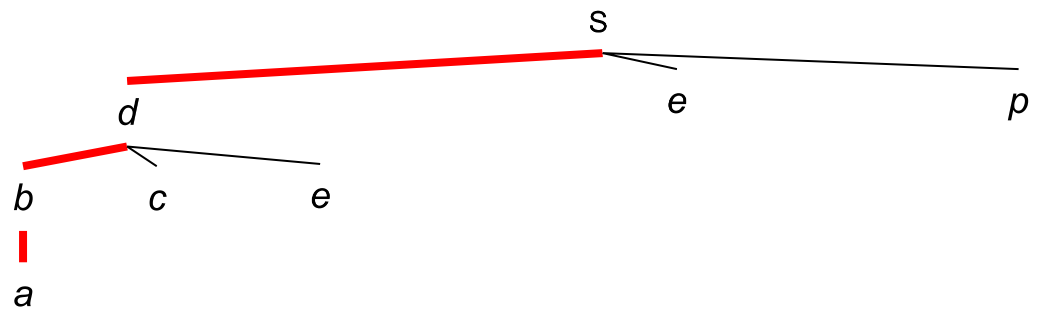
Fringe

~~s~~
~~s→d~~
s→e
s→p
~~s→d→b~~
s→d→c
s→d→e
s→d→b→a

Depth First Search

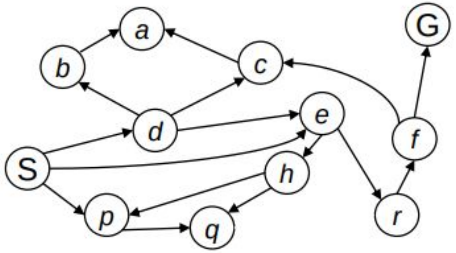


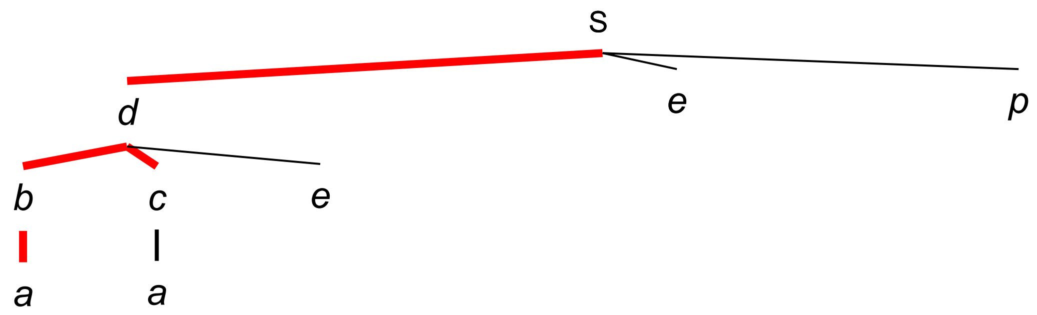
Fringe



~~s~~
~~s→d~~
s→e
s→p
~~s→d→b~~
s→d→c
s→d→e
~~s→d→b→a~~

Depth First Search

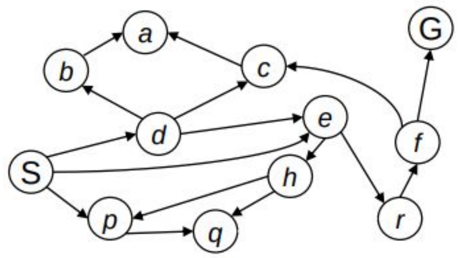


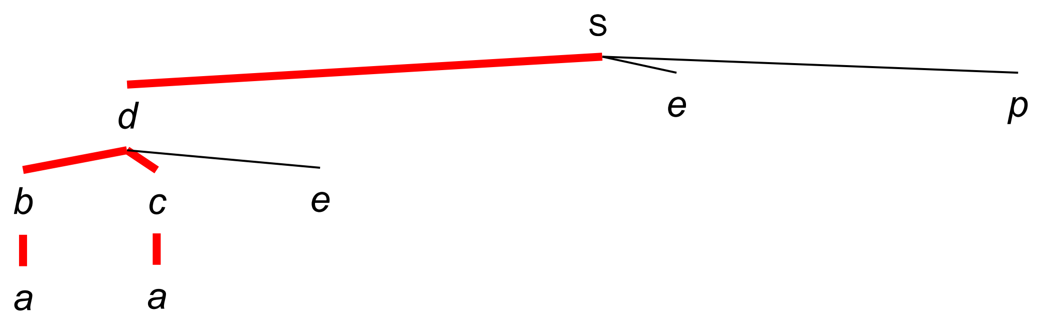


Fringe

~~s~~
~~s→d~~
s→e
s→p
~~s→d→b~~
~~s→d→c~~
s→d→e
~~s→d→b→a~~
s→d→c→a

Depth First Search

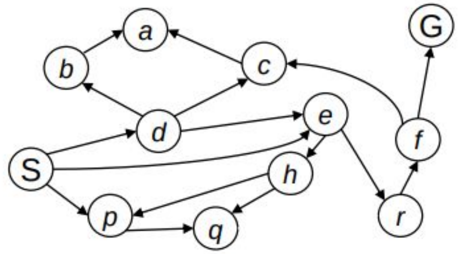


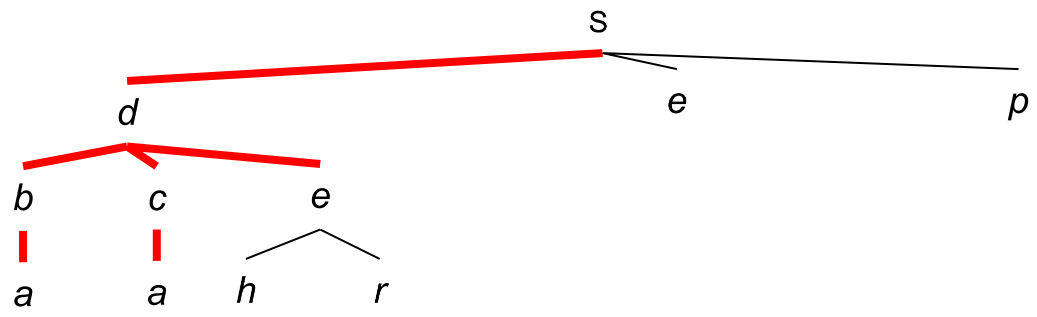


Fringe

~~s~~
~~s→d~~
s→e
s→p
~~s→d→b~~
~~s→d→c~~
s→d→e
~~s→d→b→a~~
~~s→d→c→a~~

Depth First Search

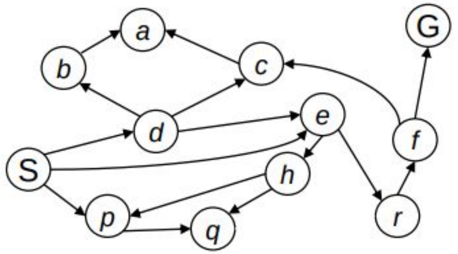


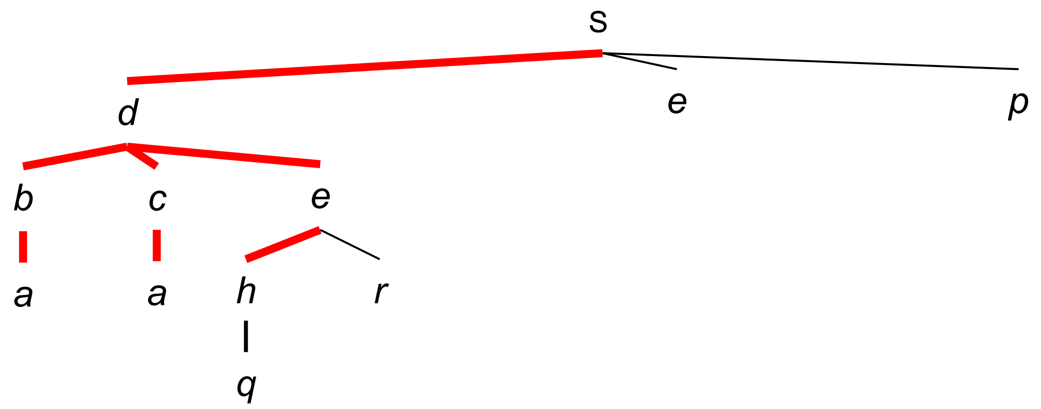


Fringe

~~s~~
~~s→d~~
s→e
s→p
~~s→d→b~~
~~s→d→c~~
~~s→d→e~~
~~s→d→b→a~~
~~s→d→c→a~~
s→d→e→h
s→d→e→r

Depth First Search

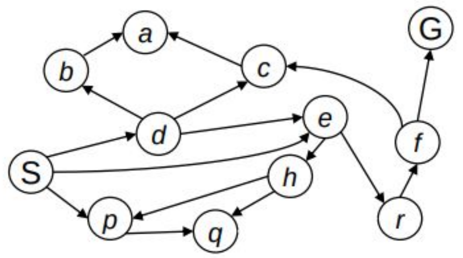


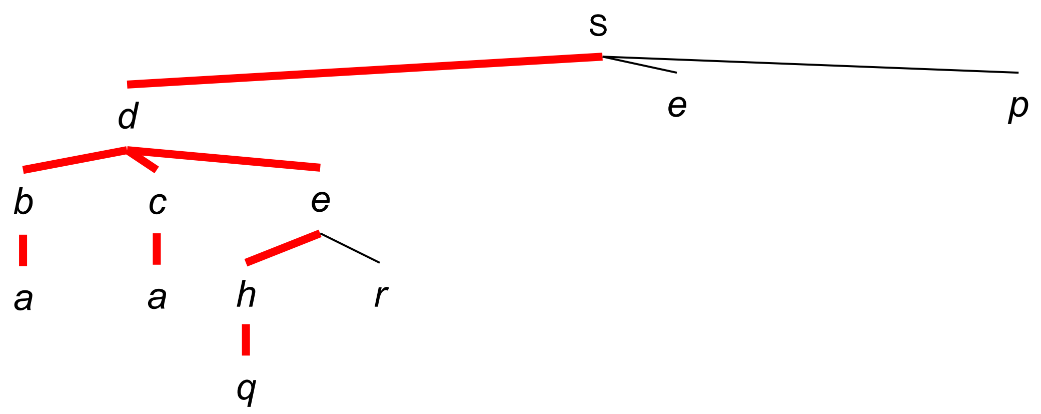


Fringe

~~s~~
~~s→d~~
s→e
s→p
~~s→d→b~~
~~s→d→c~~
~~s→d→e~~
~~s→d→b→a~~
~~s→d→c→a~~
~~s→d→e→h~~
s→d→e→r
s→d→e→h→q

Depth First Search

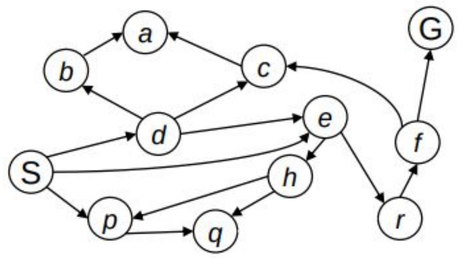




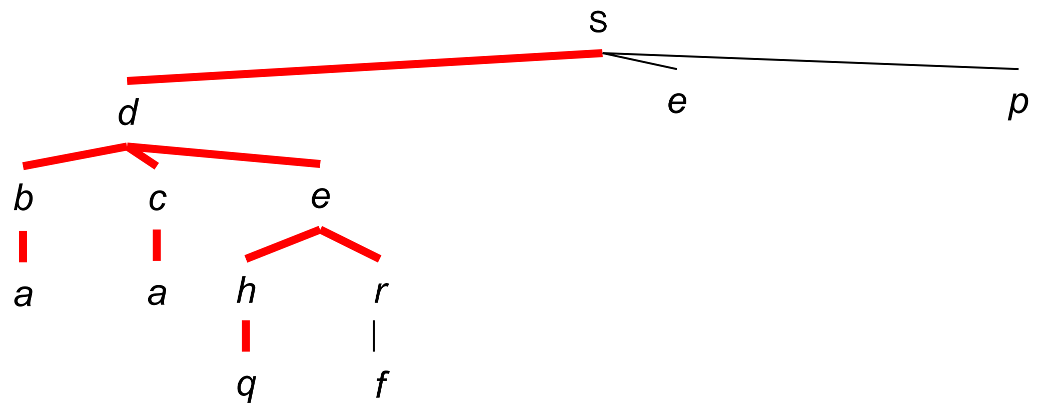
Fringe

~~s~~
~~s→d~~
s→e
s→p
~~s→d→b~~
~~s→d→c~~
~~s→d→e~~
~~s→d→b→a~~
~~s→d→c→a~~
~~s→d→e→h~~
s→d→e→r
~~s→d→e→h→q~~

Depth First Search

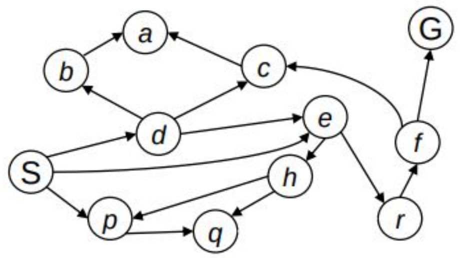


Fringe

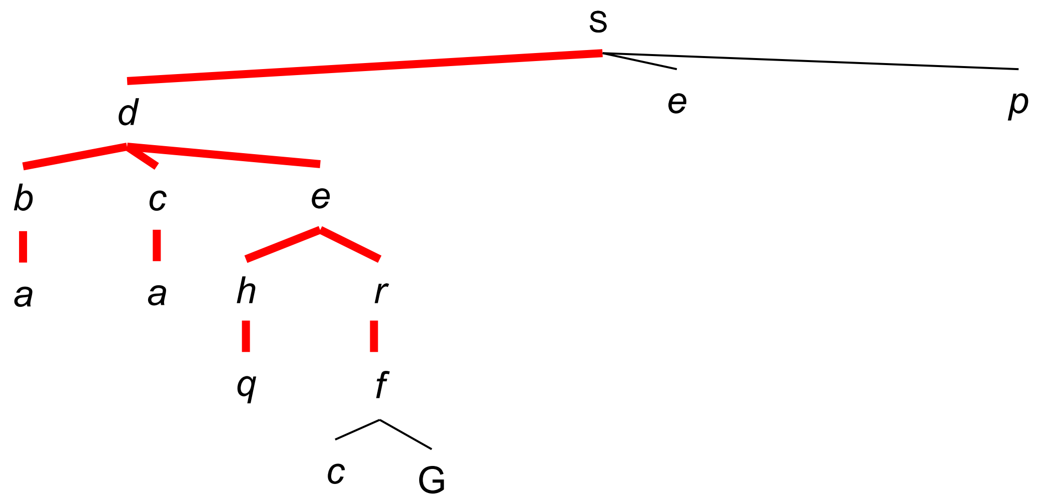


~~s~~
~~s→d~~
s→e
s→p
~~s→d→b~~
~~s→d→c~~
~~s→d→e~~
~~s→d→b→a~~
~~s→d→c→a~~
~~s→d→e→h~~
~~s→d→e→r~~
~~s→d→e→h→q~~
s→d→e→r→f

Depth First Search

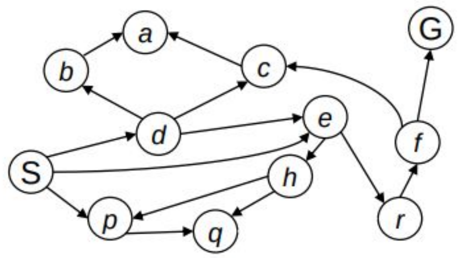


Fringe

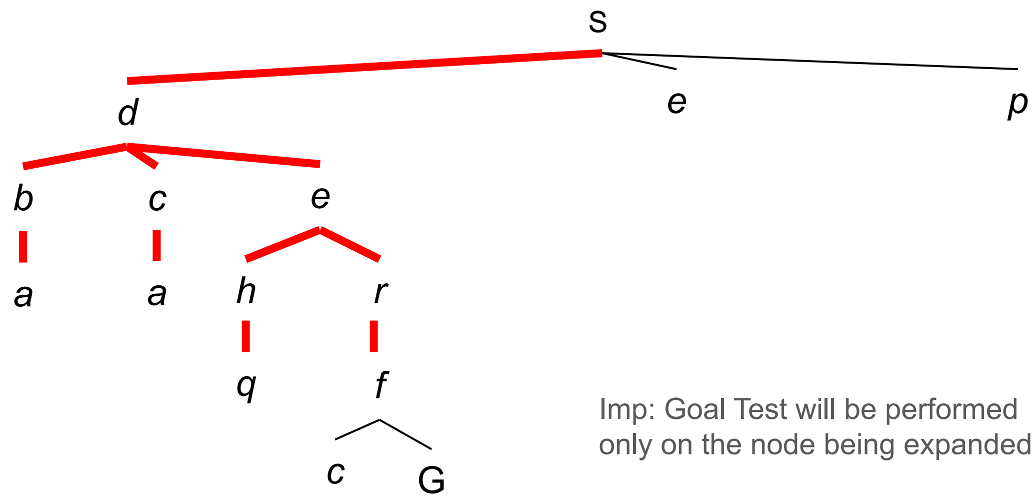


~~s~~
~~s→d~~
s→e
s→p
~~s→d→b~~
~~s→d→c~~
~~s→d→e~~
~~s→d→b→a~~
~~s→d→c→a~~
~~s→d→e→h~~
~~s→d→e→r~~
~~s→d→e→h→q~~
~~s→d→e→r→f~~
s→d→e→r→f→c
s→d→e→r→f→G

Depth First Search



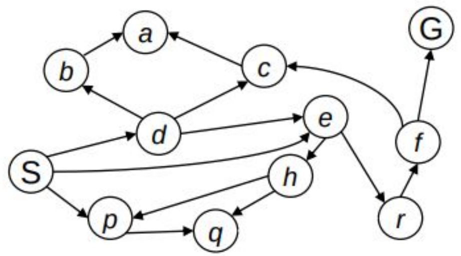
Fringe

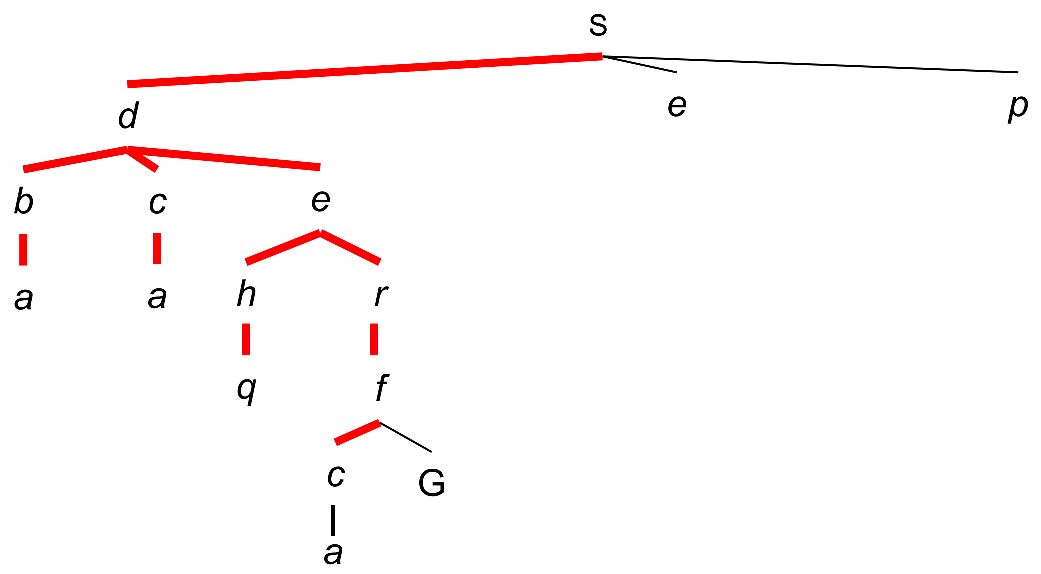


Imp: Goal Test will be performed only on the node being expanded.

~~s~~
~~s→d~~
s→e
s→p
~~s→d→b~~
~~s→d→c~~
~~s→d→e~~
~~s→d→b→a~~
~~s→d→c→a~~
~~s→d→e→h~~
~~s→d→e→r~~
~~s→d→e→h→q~~
~~s→d→e→r→f~~
s→d→e→r→f→c
s→d→e→r→f→G

Depth First Search

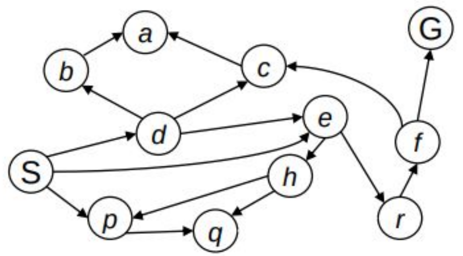


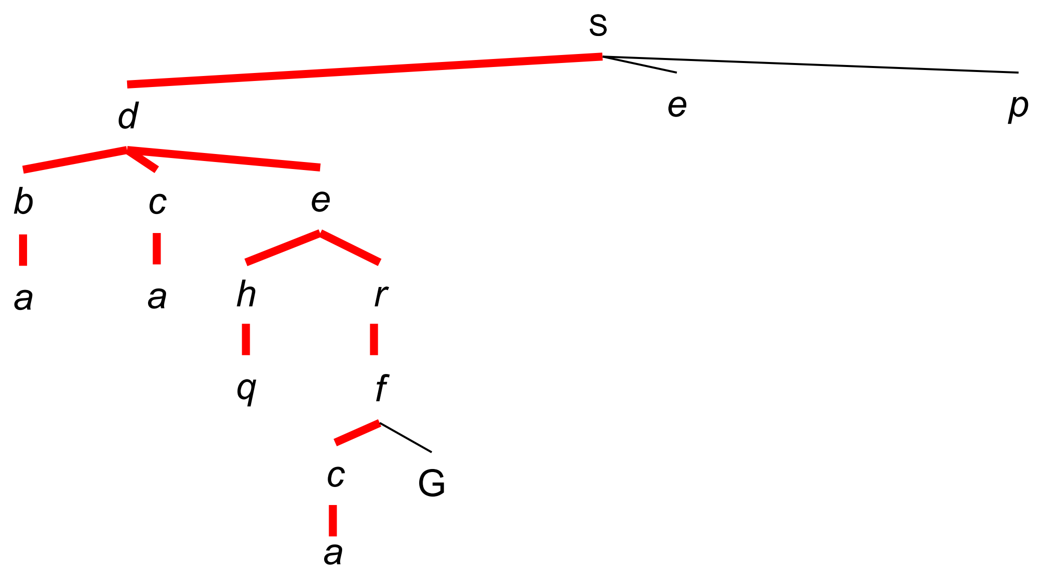


Fringe

~~s~~
~~s→d~~
s→e
s→p
~~s→d→b~~
~~s→d→c~~
~~s→d→e~~
~~s→d→b→a~~
~~s→d→c→a~~
~~s→d→e→h~~
~~s→d→e→r~~
~~s→d→e→h→q~~
~~s→d→e→r→f~~
~~s→d→e→r→f→c~~
s→d→e→r→f→G
s→d→e→r→f→c→a

Depth First Search

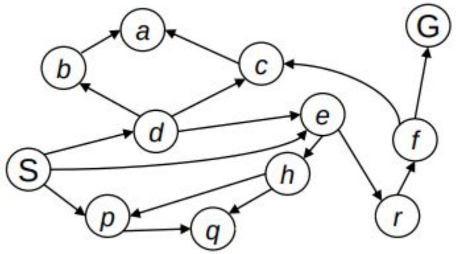


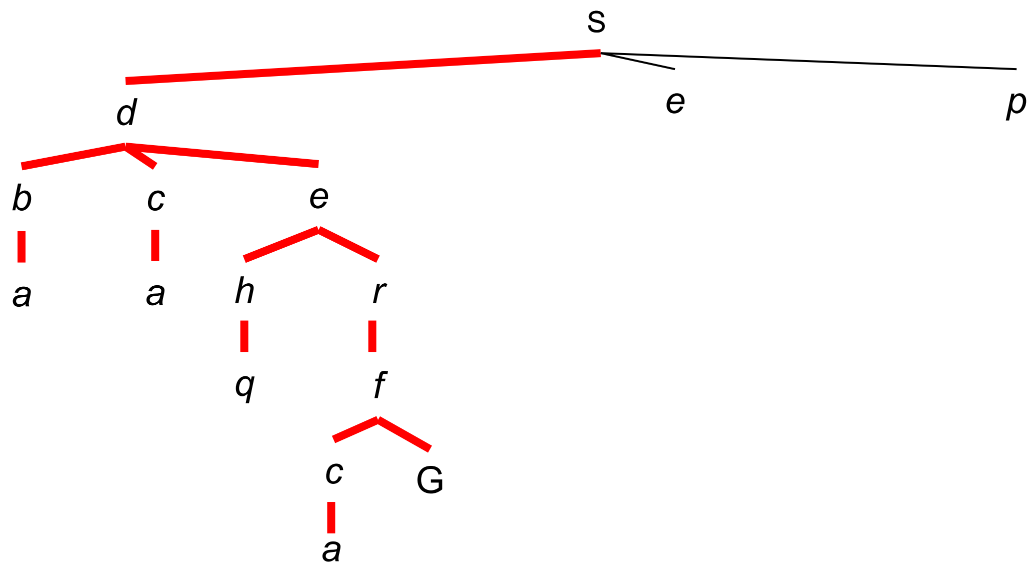


Fringe

~~s~~
~~s→d~~
s→e
s→p
~~s→d→b~~
~~s→d→c~~
~~s→d→e~~
~~s→d→b→a~~
~~s→d→c→a~~
~~s→d→e→h~~
~~s→d→e→r~~
~~s→d→e→h→q~~
~~s→d→e→r→f~~
~~s→d→e→r→f→c~~
s→d→e→r→f→G
~~s→d→e→r→f→c→a~~

Depth First Search

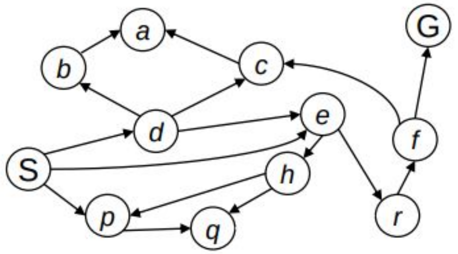




Fringe

~~s~~
~~s→d~~
s→e
s→p
~~s→d→b~~
~~s→d→c~~
~~s→d→e~~
~~s→d→b→a~~
~~s→d→c→a~~
~~s→d→e→h~~
~~s→d→e→r~~
~~s→d→e→h→q~~
~~s→d→e→r→f~~
~~s→d→e→r→f→c~~
s→d→e→r→f→G **solution**
~~s→d→e→r→f→c→a~~

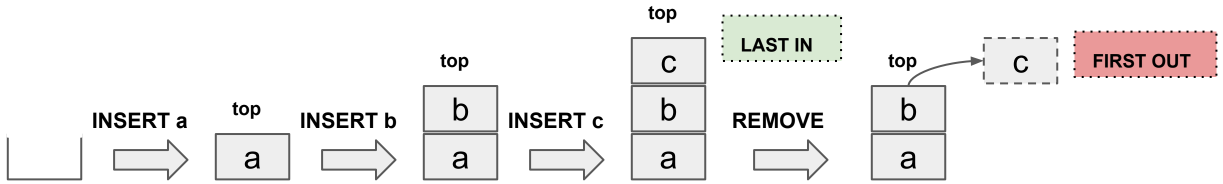
Depth First Search



Strategy: expand a **deepest node first**

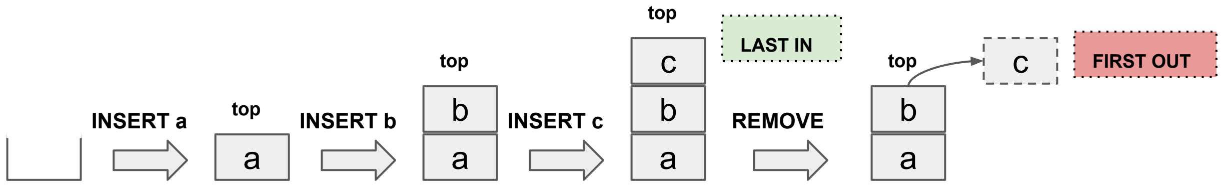
Implementation: List of partial plans/fringe is a LIFO stack

- Stack: Data structure or in simple terms a **type of storage that follows the Last In First Out order**.
  - LIFO - **last entry** in the stack is the **first to be removed** from the stack



Aside: Stack Implementation

- Stack: Data structure or in simple terms a **type of storage that follows the Last In First Out order**.
    - LIFO - **last entry** in the stack is the **first to be removed** from the stack



```
fringe = FringeUsingStack()

fringe.insert('a')
fringe.insert('b')
fringe.insert('c')
fringe.insert('d')

print(fringe.remove()) # Output?????
```

```
class FringeUsingStack:
    def __init__(self):
        self.lt=[ ]

    def insert(self,i):
        self.lt.append(i)

    def remove(self):
        el = -999
        ln = len(self.lt)
        if(ln>=2):
            el = self.lt[0]
            self.lt = self.lt[1:]
        elif(ln==1):
            el = self.lt[0]
            self.lt = []
        else:
            el=-999

        return el
```

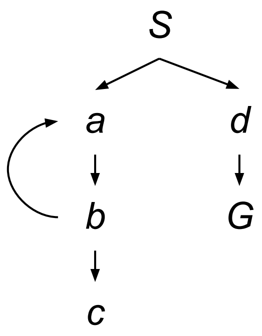
Depth First Search: Properties

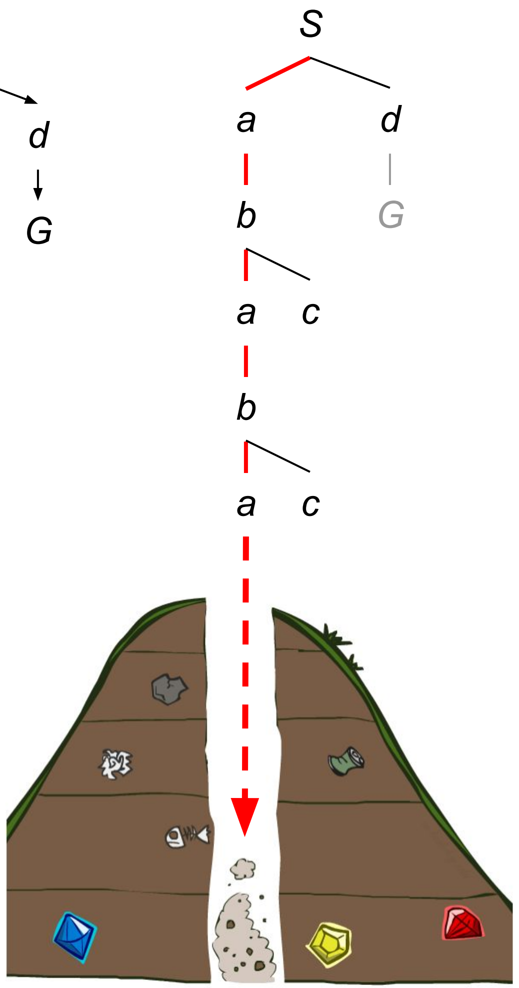
Is it complete? i.e. Is it guaranteed to find a solution if one exists?

- No, not necessarily
    - If no cycles and finite no. of nodes then Yes

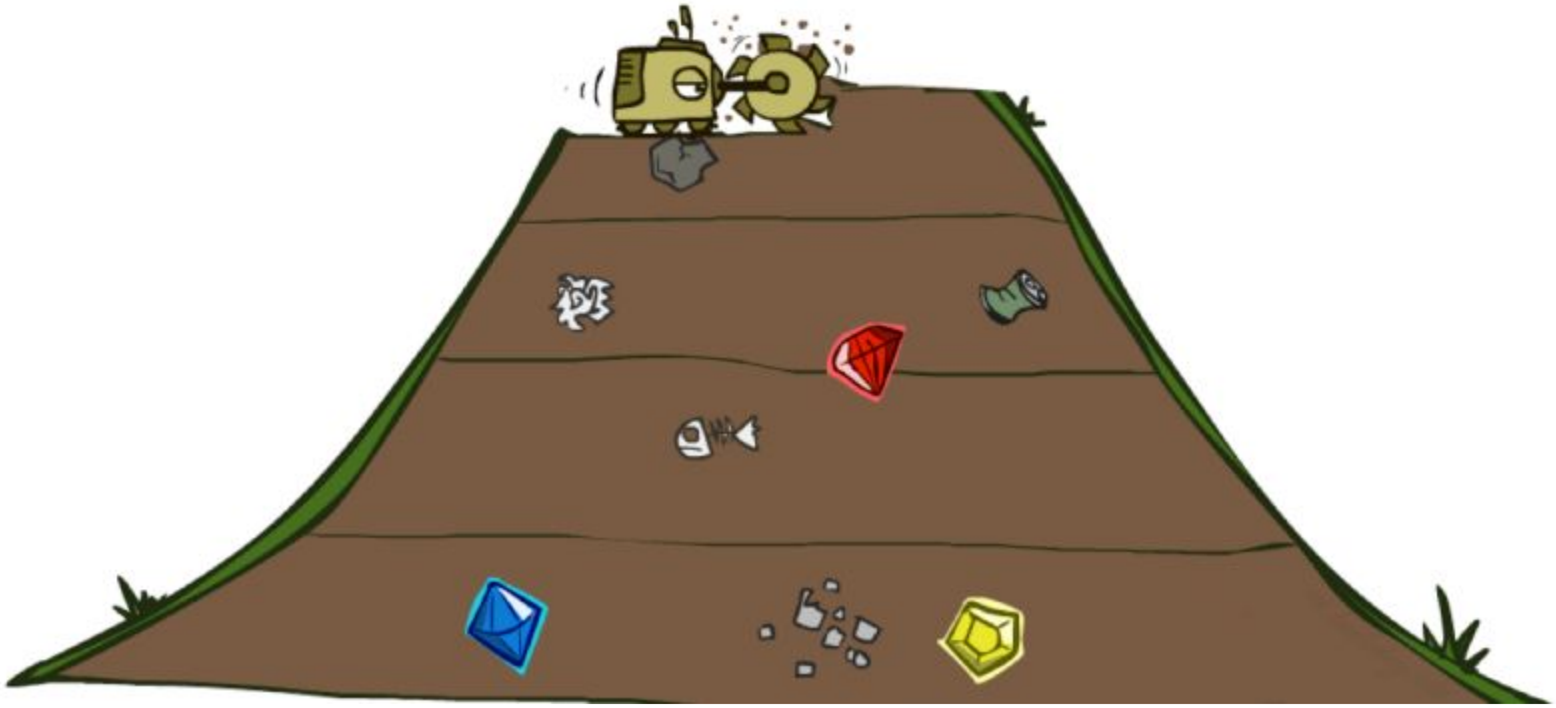
Is it optimal? i.e. Guaranteed to find the least cost path?

- No.
    - Algorithm makes no attempt to reduce number of actions/nodes in the solution

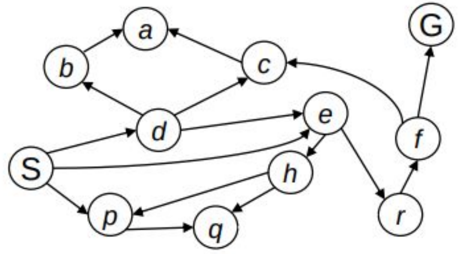




Breadth First Search

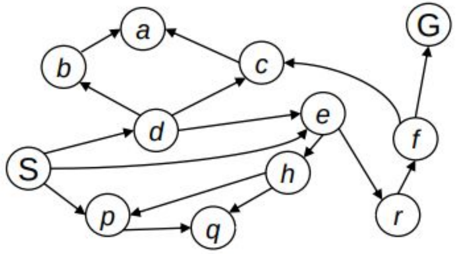


Breadth First Search



Strategy: expand a **shallowest node first**

Breadth First Search

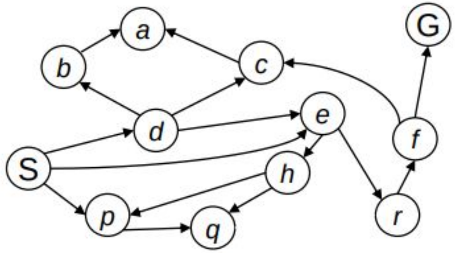


Fringe

S

s

Breadth First Search

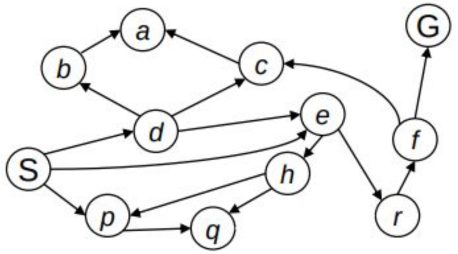


$$\begin{tikzcd}
 & s \arrow[dl, no head] \arrow[d, no head] \arrow[dr, no head] & \\
d & e & p
\end{tikzcd}$$

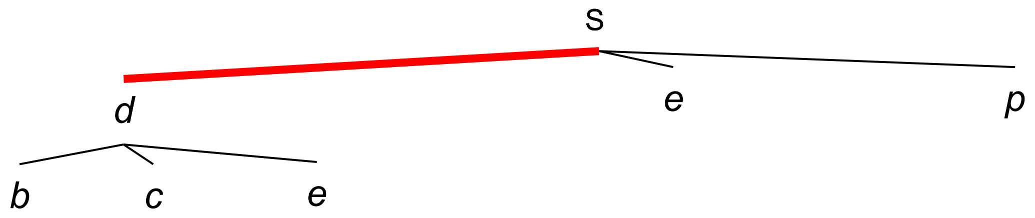
Fringe

~~s~~
s→d
s→e
s→p

Breadth First Search

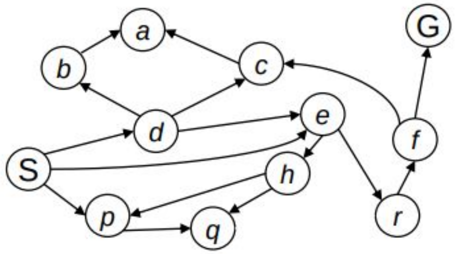


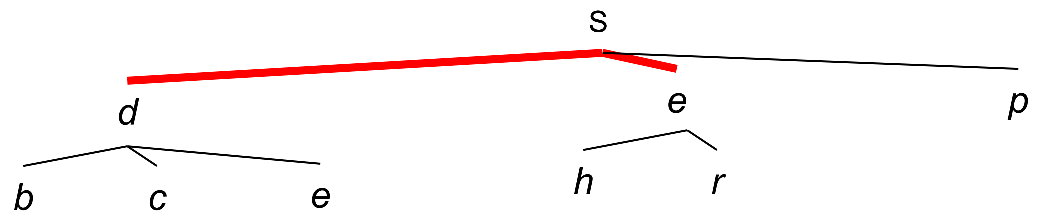
Fringe



~~s~~
~~s→d~~
s→e
s→p
s→d→b
s→d→c
s→d→e

Breadth First Search

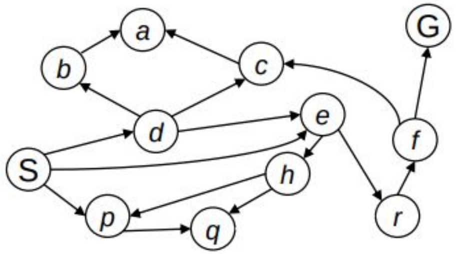




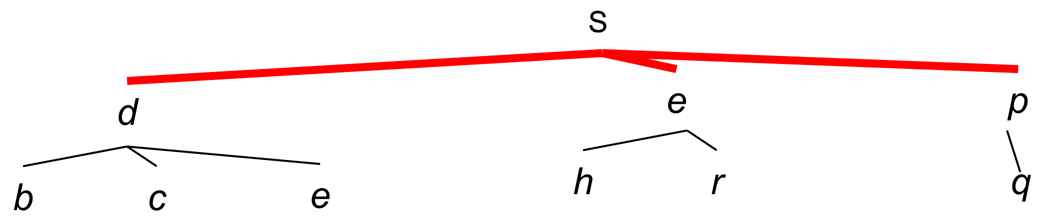
Fringe

~~s~~
~~s→d~~
~~s→e~~
s→p
s→d→b
s→d→c
s→d→e
s→e→h
s→e→r

Breadth First Search

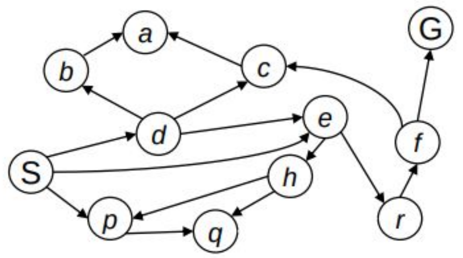


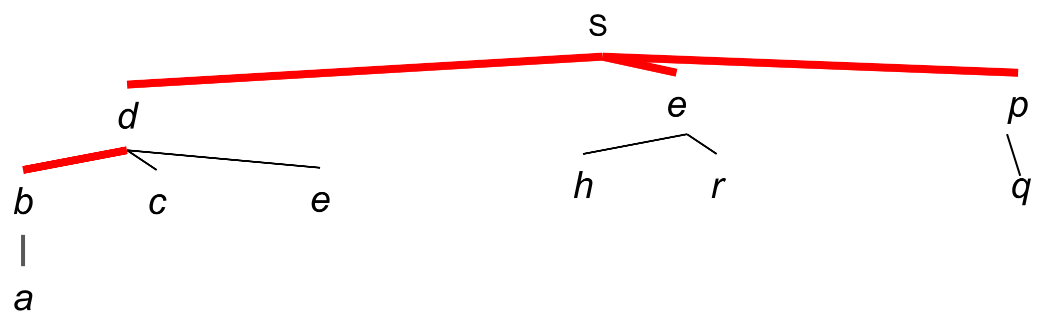
Fringe



~~s~~
~~s→d~~
~~s→e~~
~~s→p~~
s→d→b
s→d→c
s→d→e
s→e→h
s→e→r
s→p→q

Breadth First Search

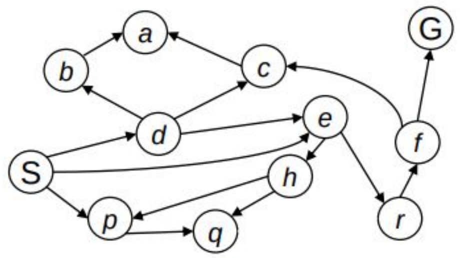




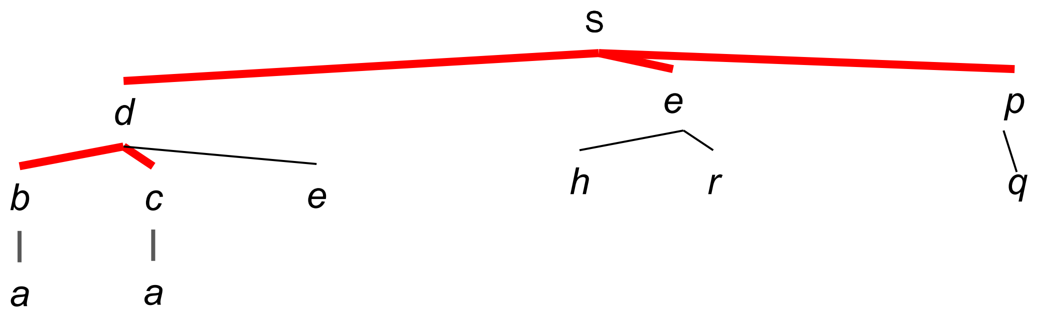
Fringe

~~s~~
~~s→d~~
~~s→e~~
~~s→p~~
~~s→d→b~~
s→d→c
s→d→e
s→e→h
s→e→r
s→p→q
s→d→b→a

Breadth First Search

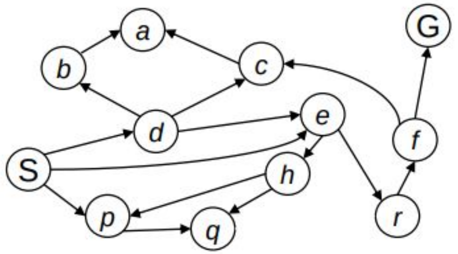


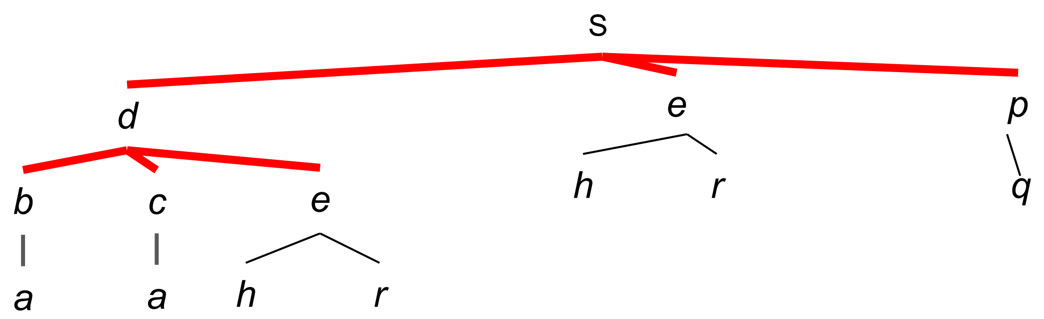
Fringe



~~s~~
~~s→d~~
~~s→e~~
~~s→p~~
~~s→d→b~~
~~s→d→c~~
s→d→e
s→e→h
s→e→r
s→p→q
s→d→b→a
s→d→c→a

Breadth First Search

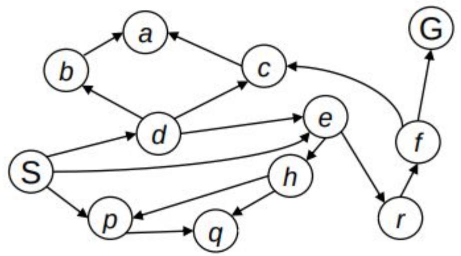


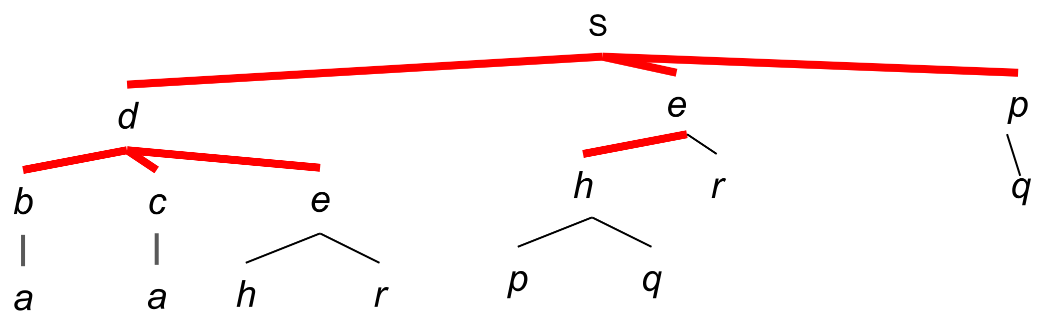


Fringe

~~s~~
~~s→d~~
~~s→e~~
~~s→p~~
~~s→d→b~~
~~s→d→c~~
~~s→d→e~~
s→e→h
s→e→r
s→p→q
s→d→b→a
s→d→c→a
s→d→e→h
s→d→e→r

Breadth First Search

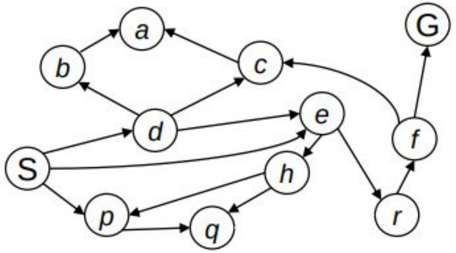


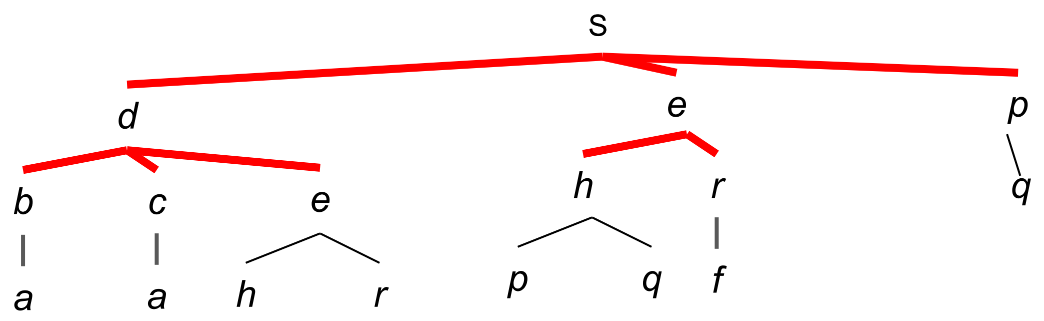


Fringe

~~s~~
~~s→d~~
~~s→e~~
~~s→p~~
~~s→d→b~~
~~s→d→c~~
~~s→d→e~~
~~s→e→h~~
s→e→r
s→p→q
s→d→b→a
s→d→c→a
s→d→e→h
s→d→e→r
s→e→h→p
s→e→h→q

Breadth First Search



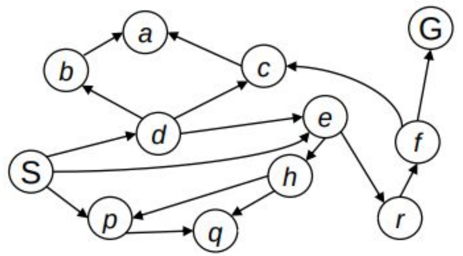


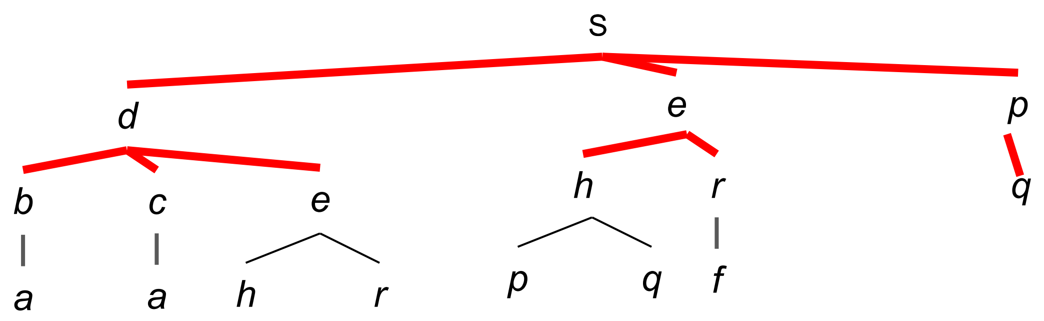
Fringe

~~s~~
~~s→d~~
~~s→e~~
~~s→p~~
~~s→d→b~~
~~s→d→c~~
~~s→d→e~~
~~s→e→h~~
~~s→e→r~~
s→p→q
s→d→b→a
s→d→c→a
s→d→e→h
s→d→e→r
s→e→h→p
s→e→h→q

s→e→r→f

Breadth First Search



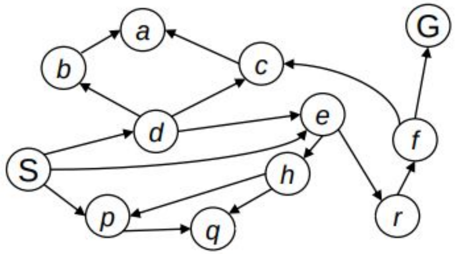


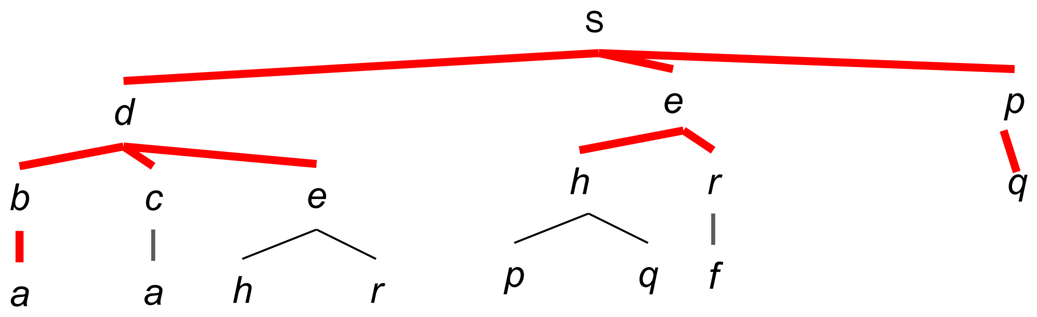
Fringe

~~s~~
~~s→d~~
~~s→e~~
~~s→p~~
~~s→d→b~~
~~s→d→c~~
~~s→d→e~~
~~s→e→h~~
~~s→e→r~~
~~s→p→q~~
s→d→b→a
s→d→c→a
s→d→e→h
s→d→e→r
s→e→h→p
s→e→h→q

s→e→r→f

Breadth First Search



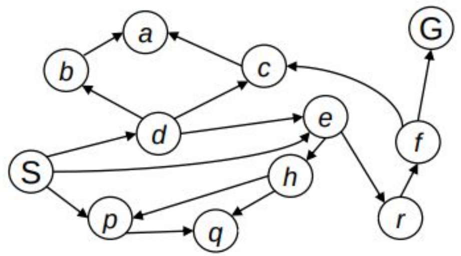


Fringe

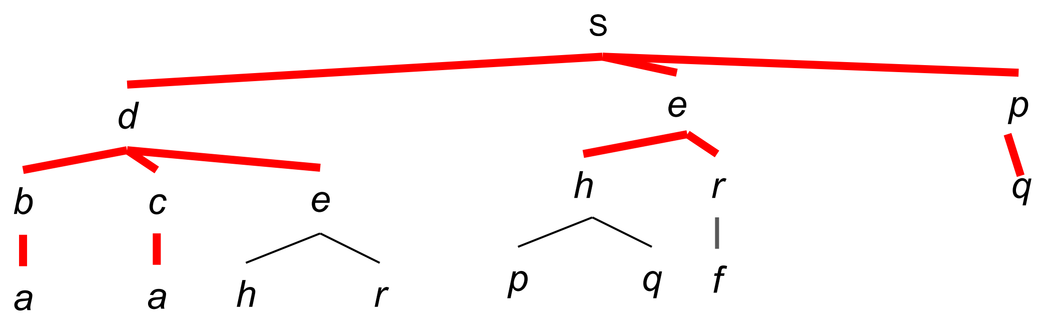
~~s~~
~~s→d~~
~~s→e~~
~~s→p~~
~~s→d→b~~
~~s→d→c~~
~~s→d→e~~
~~s→e→h~~
~~s→e→r~~
~~s→p→q~~
~~s→d→b→a~~
s→d→c→a
s→d→e→h
s→d→e→r
s→e→h→p
s→e→h→q

s→e→r→f

Breadth First Search



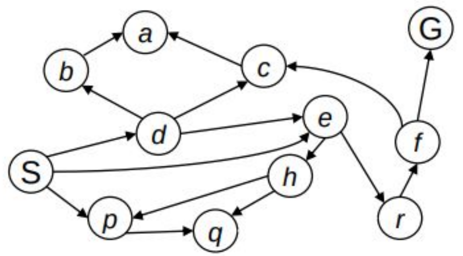
Fringe

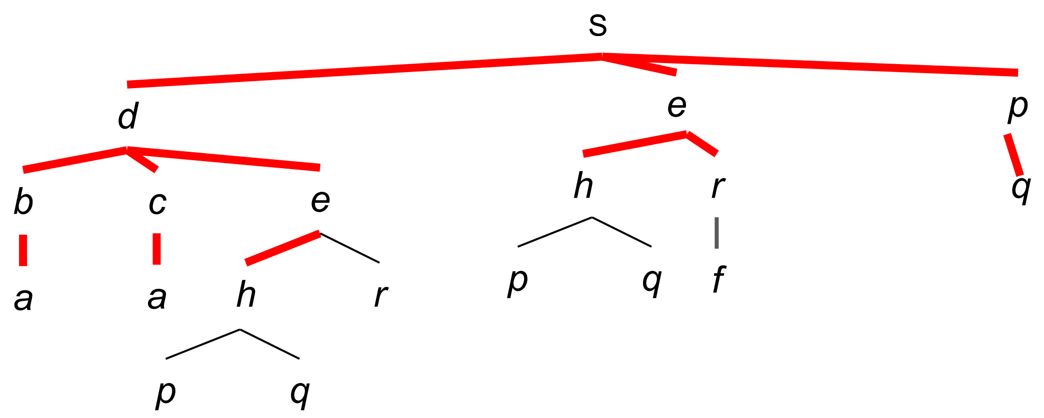


~~s~~
~~s→d~~
~~s→e~~
~~s→p~~
~~s→d→b~~
~~s→d→c~~
~~s→d→e~~
~~s→e→h~~
~~s→e→r~~
~~s→p→q~~
~~s→d→b→a~~
~~s→d→c→a~~
s→d→e→h
s→d→e→r
s→e→h→p
s→e→h→q

s→e→r→f

Breadth First Search

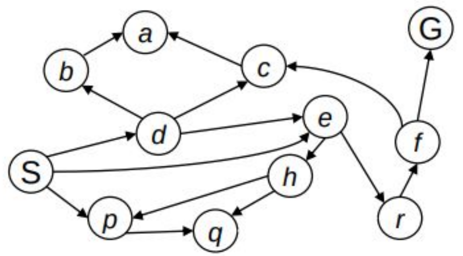




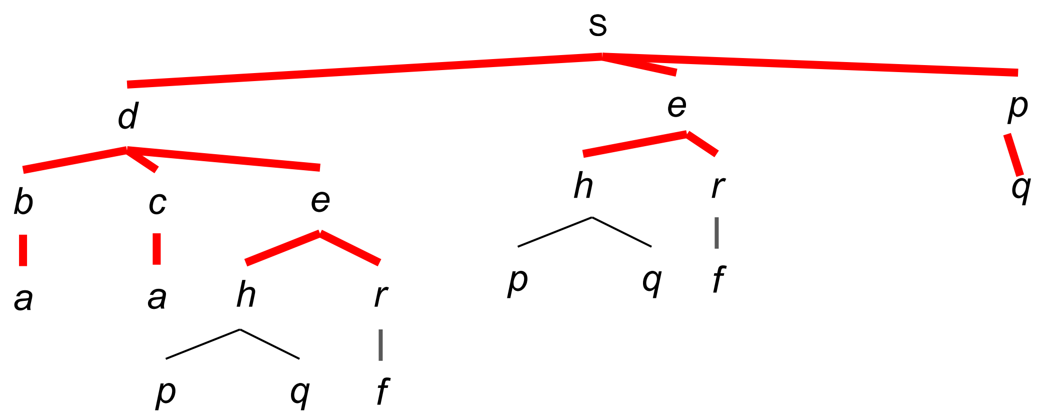
Fringe

~~s~~
~~s→d~~
~~s→e~~
~~s→p~~
~~s→d→b~~
~~s→d→c~~
~~s→d→e~~
~~s→e→h~~
~~s→e→r~~
~~s→p→q~~
~~s→d→b→a~~
~~s→d→c→a~~
~~s→d→e→h~~
s→d→e→r
s→e→h→p
s→e→h→q
s→e→r→f
s→d→e→h→p
s→d→e→h→q

Breadth First Search

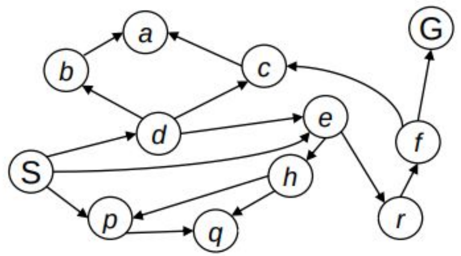


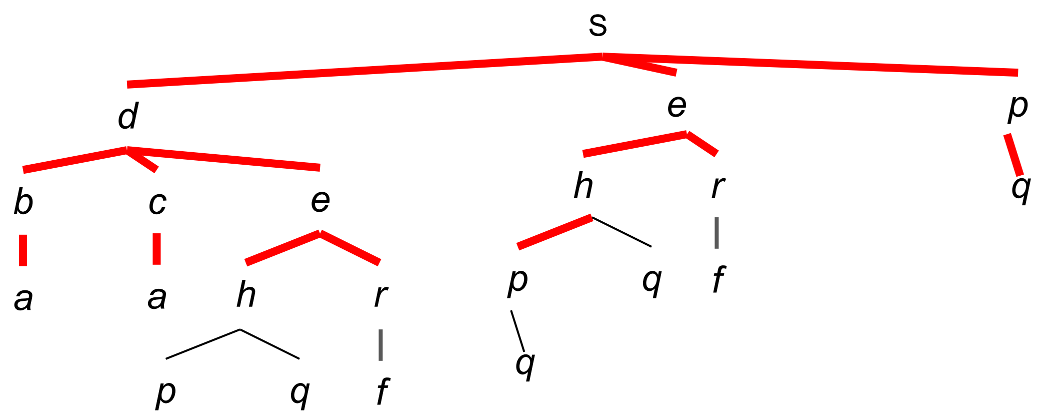
Fringe



~~s~~
~~s→d~~
~~s→e~~
~~s→p~~
~~s→d→b~~
~~s→d→c~~
~~s→d→e~~
~~s→e→h~~
~~s→e→r~~
~~s→p→q~~
~~s→d→b→a~~
~~s→d→c→a~~
~~s→d→e→h~~
~~s→d→e→r~~
s→e→h→p
s→e→h→q
s→e→r→f
s→d→e→h→p
s→d→e→h→q
s→d→e→r→f

Breadth First Search

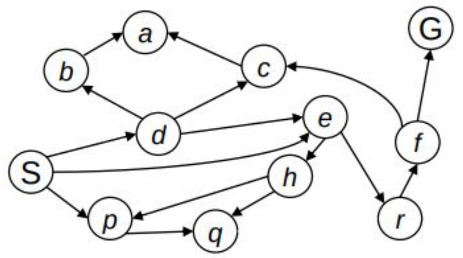


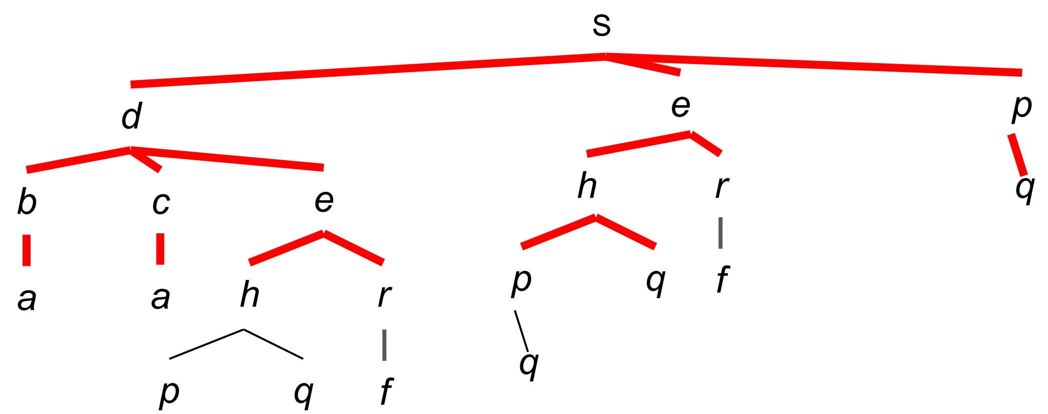


Fringe

~~s~~
~~s→d~~
~~s→e~~
~~s→p~~
~~s→d→b~~
~~s→d→c~~
~~s→d→e~~
~~s→e→h~~
~~s→e→r~~
~~s→p→q~~
~~s→d→b→a~~
~~s→d→c→a~~
~~s→d→e→h~~
~~s→d→e→r~~
~~s→e→h→p~~
s→e→h→q
s→e→r→f
s→d→e→h→p
s→d→e→h→q
s→d→e→r→f
s→e→h→p→q

Breadth First Search

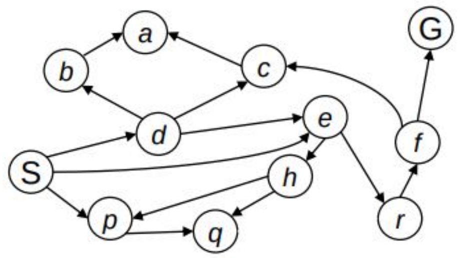


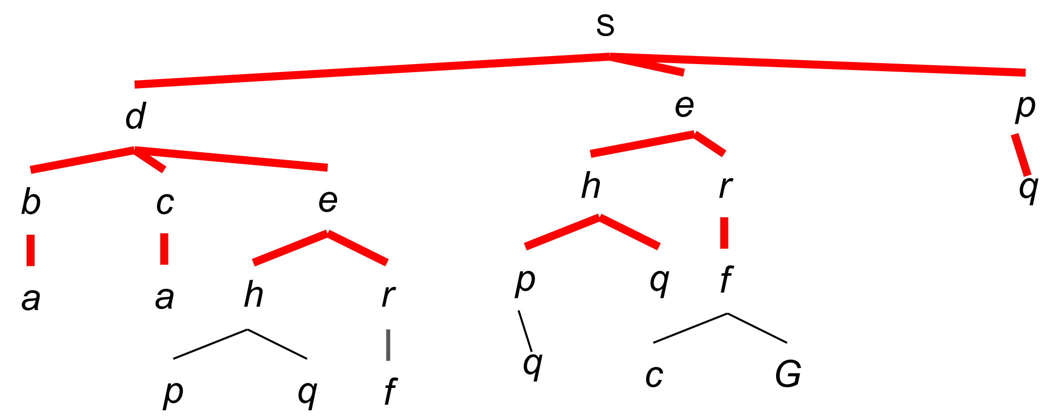


Fringe

~~s~~
~~s→d~~
~~s→e~~
~~s→p~~
~~s→d→b~~
~~s→d→c~~
~~s→d→e~~
~~s→e→h~~
~~s→e→r~~
~~s→p→q~~
~~s→d→b→a~~
~~s→d→c→a~~
~~s→d→e→h~~
~~s→d→e→r~~
~~s→e→h→p~~
~~s→e→h→q~~
s→e→r→f
s→d→e→h→p
s→d→e→h→q
s→d→e→r→f
s→e→h→p→q

Breadth First Search

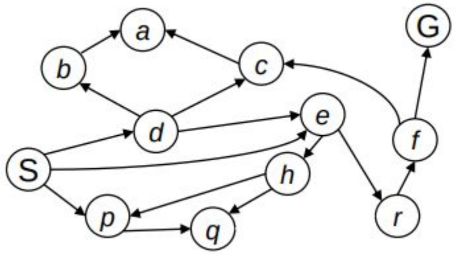




Fringe

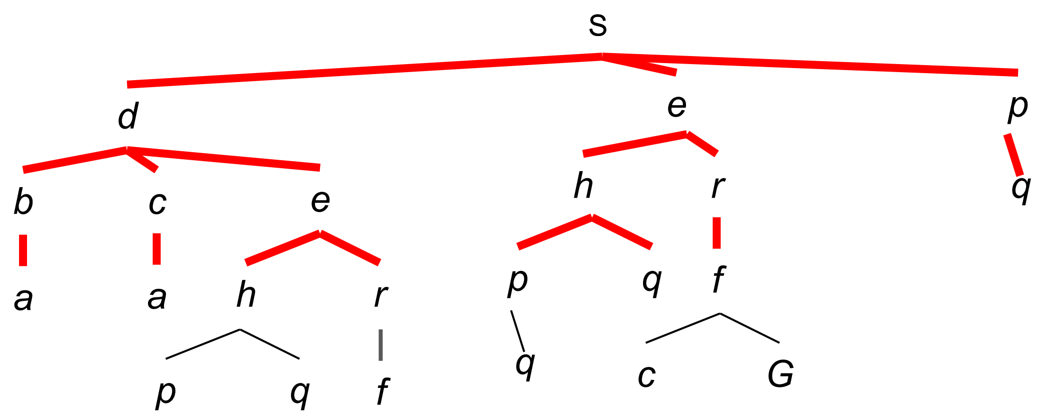
~~s~~
~~s→d~~
~~s→e~~
~~s→p~~
~~s→d→b~~
~~s→d→c~~
~~s→d→e~~
~~s→e→h~~
~~s→e→r~~
~~s→p→q~~
~~s→d→b→a~~
~~s→d→c→a~~
~~s→d→e→h~~
~~s→d→e→r~~
~~s→e→h→p~~
~~s→e→h→q~~
~~s→e→r→f~~
s→d→e→h→p
s→d→e→h→q
s→d→e→r→f
s→e→h→p→q
s→e→r→f→c
s→e→r→f→G

Breadth First Search



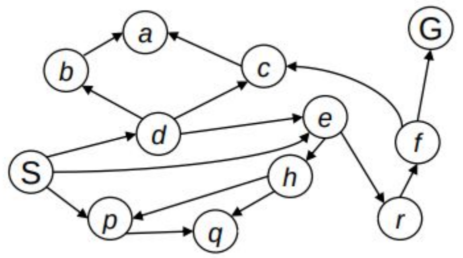
Fringe

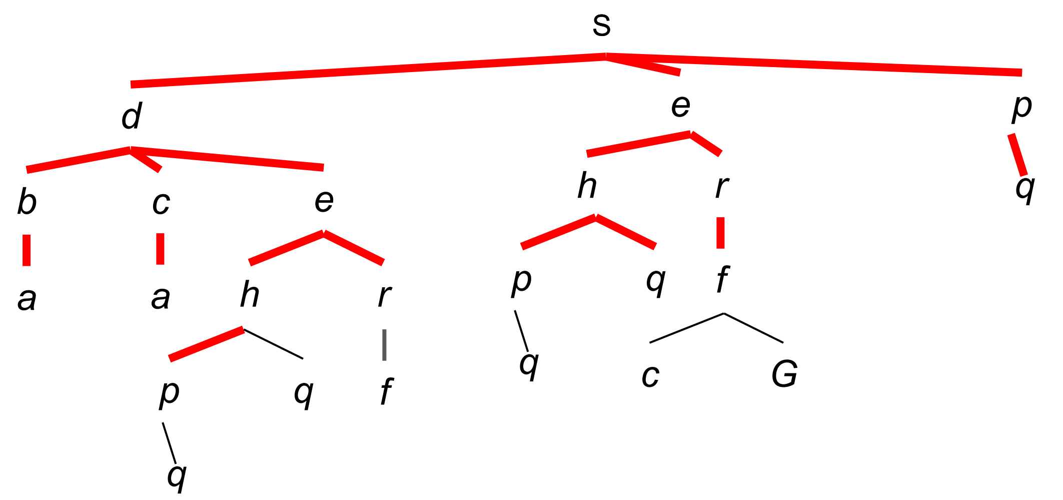
~~s~~
~~s→d~~
~~s→e~~
~~s→p~~
~~s→d→b~~
~~s→d→c~~
~~s→d→e~~
~~s→e→h~~
~~s→e→r~~
~~s→p→q~~
~~s→d→b→a~~
~~s→d→c→a~~
~~s→d→e→h~~
~~s→d→e→r~~
~~s→e→h→p~~
~~s→e→h→q~~
~~s→e→r→f~~
s→d→e→h→p
s→d→e→h→q
s→d→e→r→f
s→e→h→p→q
s→e→r→f→c
s→e→r→f→G



Imp: Goal Test will be performed only on the node being expanded.

Breadth First Search

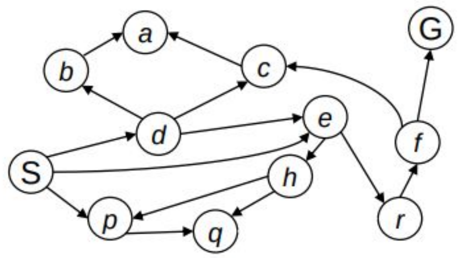


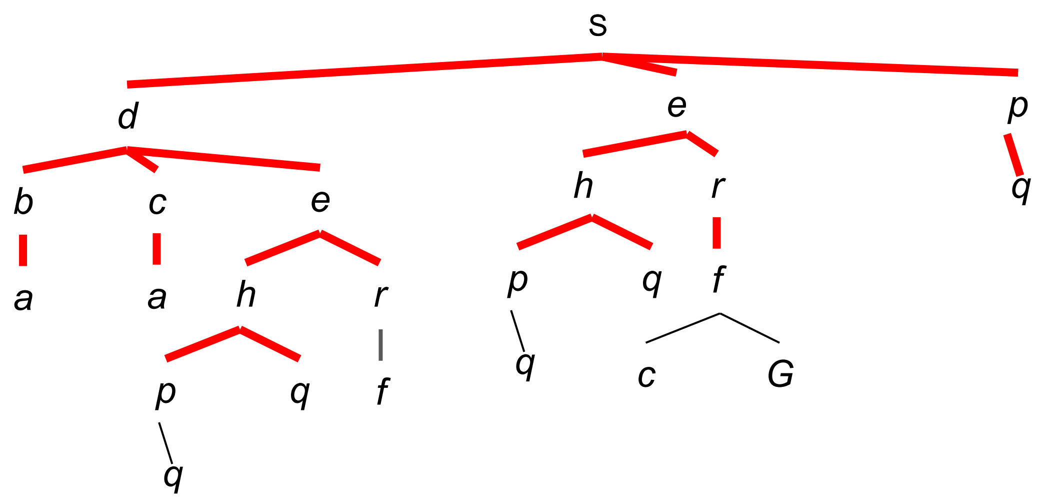


Fringe

~~s~~
~~s→d~~
~~s→e~~
~~s→p~~
~~s→d→b~~
~~s→d→c~~
~~s→d→e~~
~~s→e→h~~
~~s→e→r~~
~~s→p→q~~
~~s→d→b→a~~
~~s→d→c→a~~
~~s→d→e→h~~
~~s→d→e→r~~
~~s→e→h→p~~
~~s→e→h→q~~
~~s→e→r→f~~
~~s→d→e→h→p~~
s→d→e→h→q
s→d→e→r→f
s→e→h→p→q
s→e→r→f→c
s→e→r→f→G

Breadth First Search

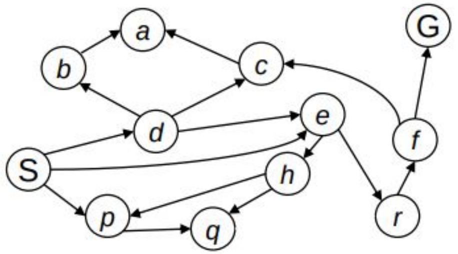


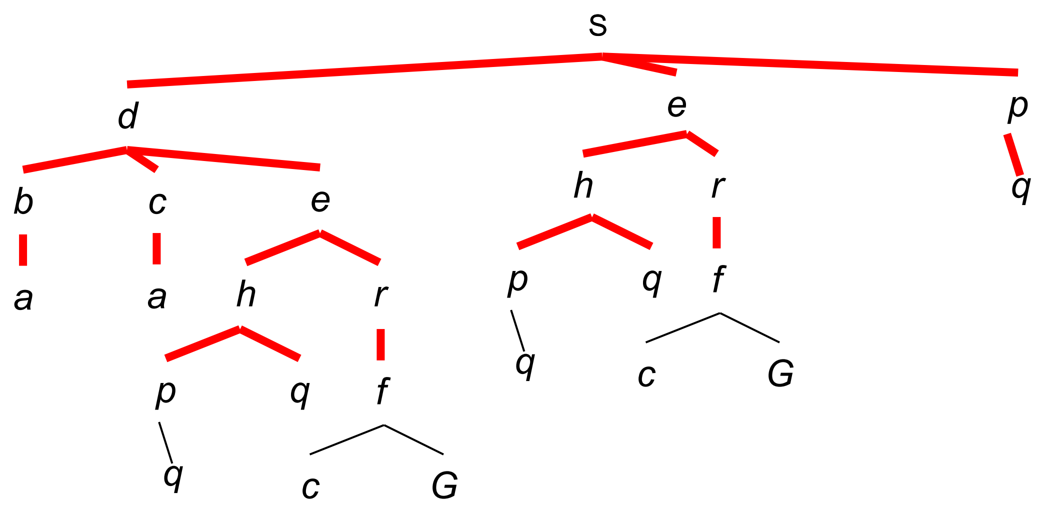


Fringe

~~s~~
~~s→d~~
~~s→e~~
~~s→p~~
~~s→d→b~~
~~s→d→c~~
~~s→d→e~~
~~s→e→h~~
~~s→e→r~~
~~s→p→q~~
~~s→d→b→a~~
~~s→d→c→a~~
~~s→d→e→h~~
~~s→d→e→r~~
~~s→e→h→p~~
~~s→e→h→q~~
~~s→e→r→f~~
~~s→d→e→h→p~~
~~s→d→e→h→q~~
s→d→e→r→f
s→e→h→p→q
s→e→r→f→c
s→e→r→f→G

Breadth First Search

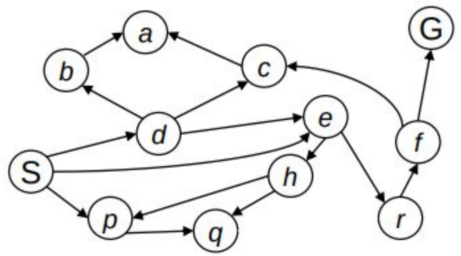


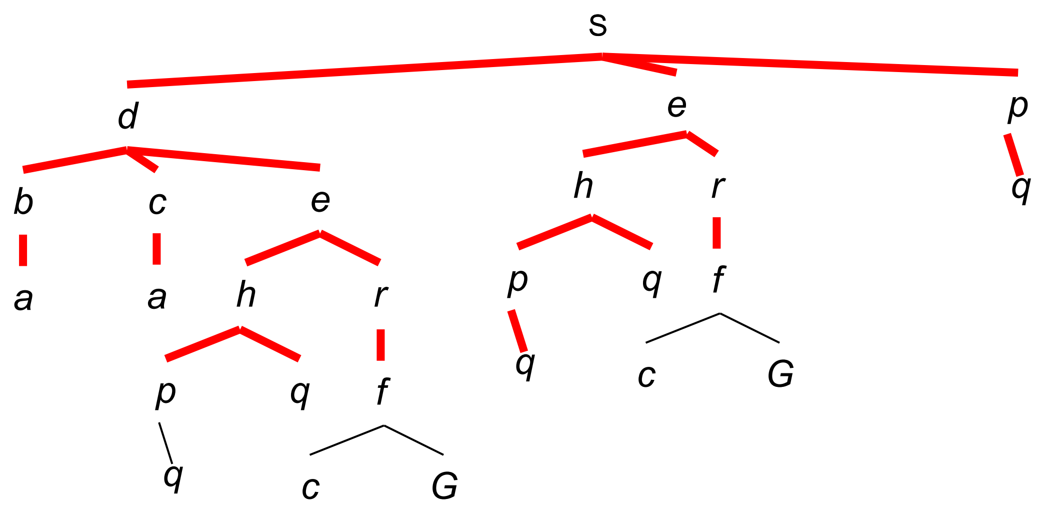


Fringe

~~s~~
~~s→d~~
~~s→e~~
~~s→p~~
~~s→d→b~~
~~s→d→c~~
~~s→d→e~~
~~s→e→h~~
~~s→e→r~~
~~s→p→q~~
~~s→d→b→a~~
~~s→d→c→a~~
~~s→d→e→h~~
~~s→d→e→r~~
~~s→e→h→p~~
~~s→e→h→q~~
~~s→e→r→f~~
~~s→d→e→h→p~~
~~s→d→e→h→q~~
~~s→d→e→r→f~~
s→e→h→p→q
s→e→r→f→c
s→e→r→f→G
s→d→e→r→f→c
s→d→e→r→f→G

Breadth First Search

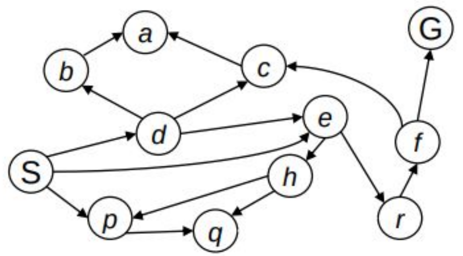


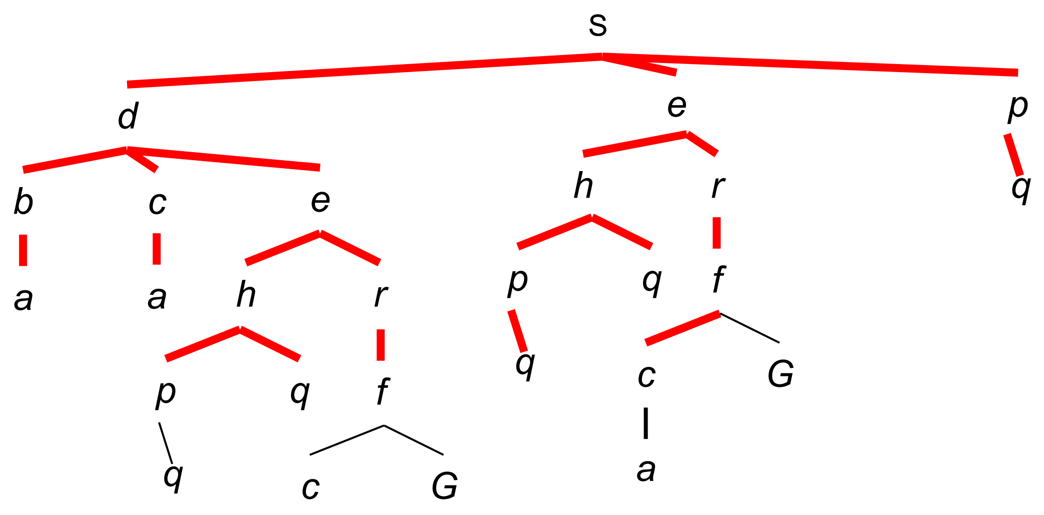


Fringe

~~s~~
~~s→d~~
~~s→e~~
~~s→p~~
~~s→d→b~~
~~s→d→c~~
~~s→d→e~~
~~s→e→h~~
~~s→e→r~~
~~s→p→q~~
~~s→d→b→a~~
~~s→d→c→a~~
~~s→d→e→h~~
~~s→d→e→r~~
~~s→e→h→p~~
~~s→e→h→q~~
~~s→e→r→f~~
~~s→d→e→h→p~~
~~s→d→e→h→q~~
~~s→d→e→r→f~~
~~s→e→h→p→q~~
s→e→r→f→c
s→e→r→f→G
s→d→e→r→f→c
s→d→e→r→f→G

Breadth First Search

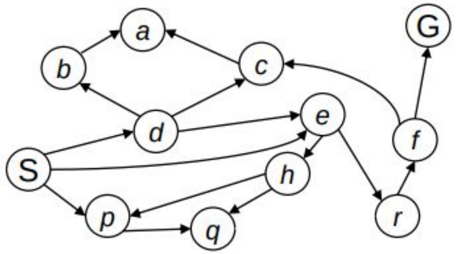


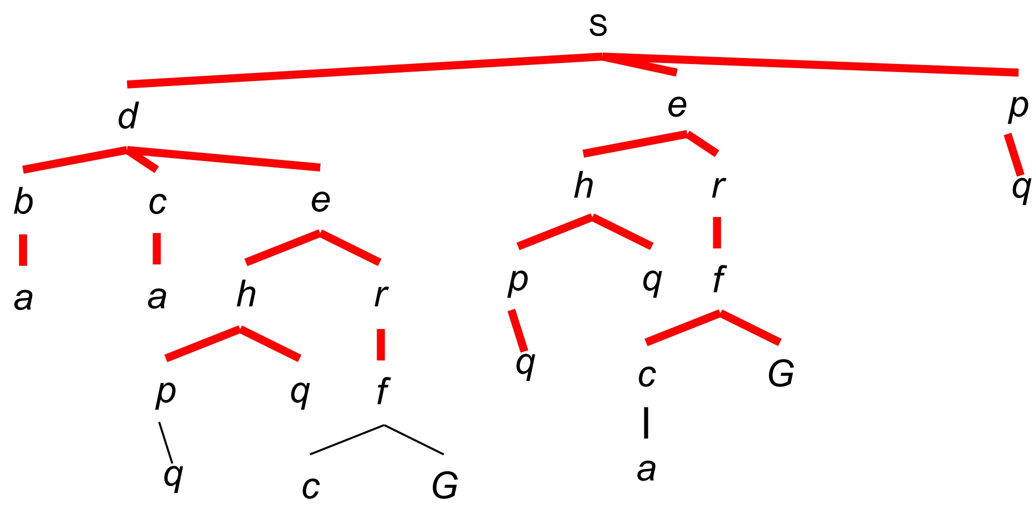


Fringe

~~s~~
~~s→d~~
~~s→e~~
~~s→p~~
~~s→d→b~~
~~s→d→c~~
~~s→d→e~~
~~s→e→h~~
~~s→e→r~~
~~s→p→q~~
~~s→d→b→a~~
~~s→d→c→a~~
~~s→d→e→h~~
~~s→d→e→r~~
~~s→e→h→p~~
~~s→e→h→q~~
~~s→e→r→f~~
~~s→d→e→h→p~~
~~s→d→e→h→q~~
~~s→d→e→r→f~~
~~s→e→h→p→q~~
~~s→e→r→f→c~~
s→e→r→f→G
s→d→e→r→f→c
s→d→e→r→f→G
s→e→r→f→c→a

Breadth First Search

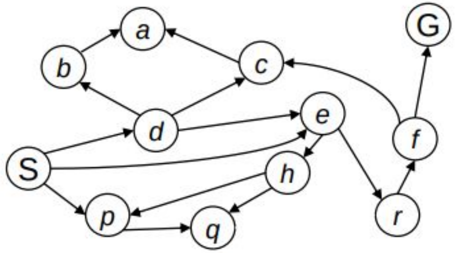


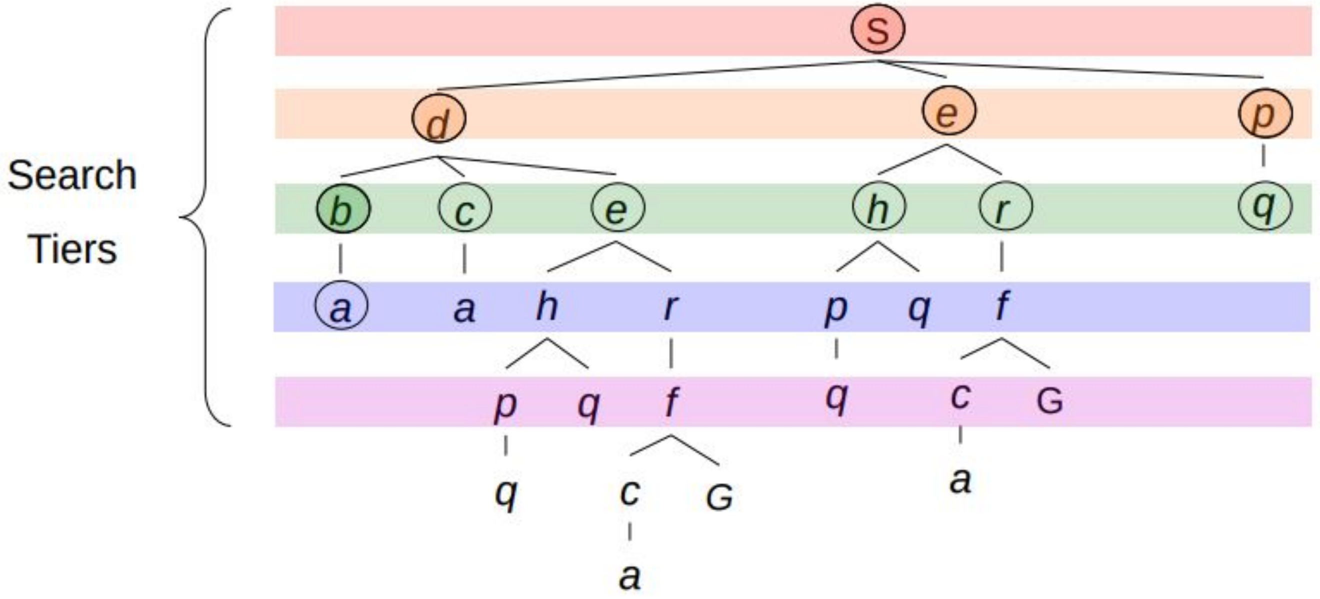


Fringe

~~s~~
~~s→d~~
~~s→e~~
~~s→p~~
~~s→d→b~~
~~s→d→c~~
~~s→d→e~~
~~s→e→h~~
~~s→e→r~~
~~s→p→q~~
~~s→d→b→a~~
~~s→d→c→a~~
~~s→d→e→h~~
~~s→d→e→r~~
~~s→e→h→p~~
~~s→e→h→q~~
~~s→e→r→f~~
~~s→d→e→h→p~~
~~s→d→e→h→q~~
~~s→d→e→r→f~~
~~s→e→h→p→q~~
~~s→e→r→f→c~~
solution s→e→r→f→G
s→d→e→r→f→c
s→d→e→r→f→G
s→e→r→f→c→a

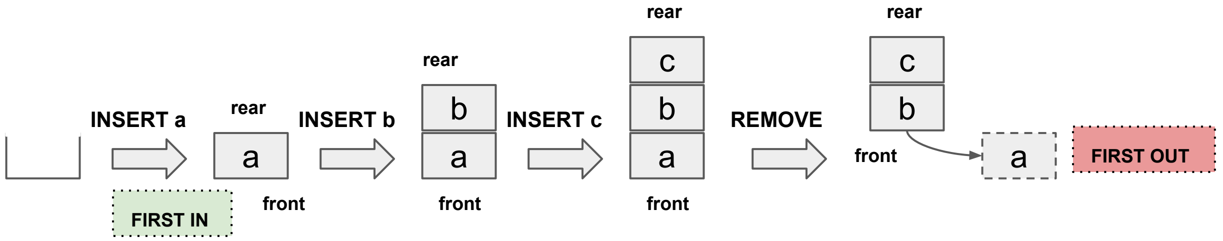
Breadth First Search





Aside: Queue Implementation

- Queue: Data structure or in simple terms a **type of storage that follows the First In First Out order.**
    - FIFO - first entry in the queue is the first to be removed from the queue



```
fringe = FringeUsingQueue()

fringe.insert('a')
fringe.insert('b')
fringe.insert('c')
fringe.insert('d')

print(fringe.remove()) # Output?????
```

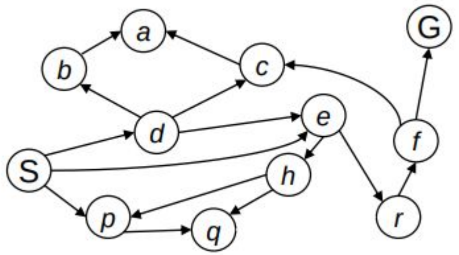
```
class FringeUsingQueue:
    def __init__(self):
        self.lt=[ ]

    def insert(self,i):
        self.lt.append(i)

    def remove(self):
        el = -1
        ln = len(self.lt)
        if(ln>=2):
            el = self.lt[0]
            self.lt = self.lt[1:]
        elif(ln==1):
            el = self.lt[0]
            self.lt = []
        else:
            el=-1

        return el
```

Breadth First Search: Properties



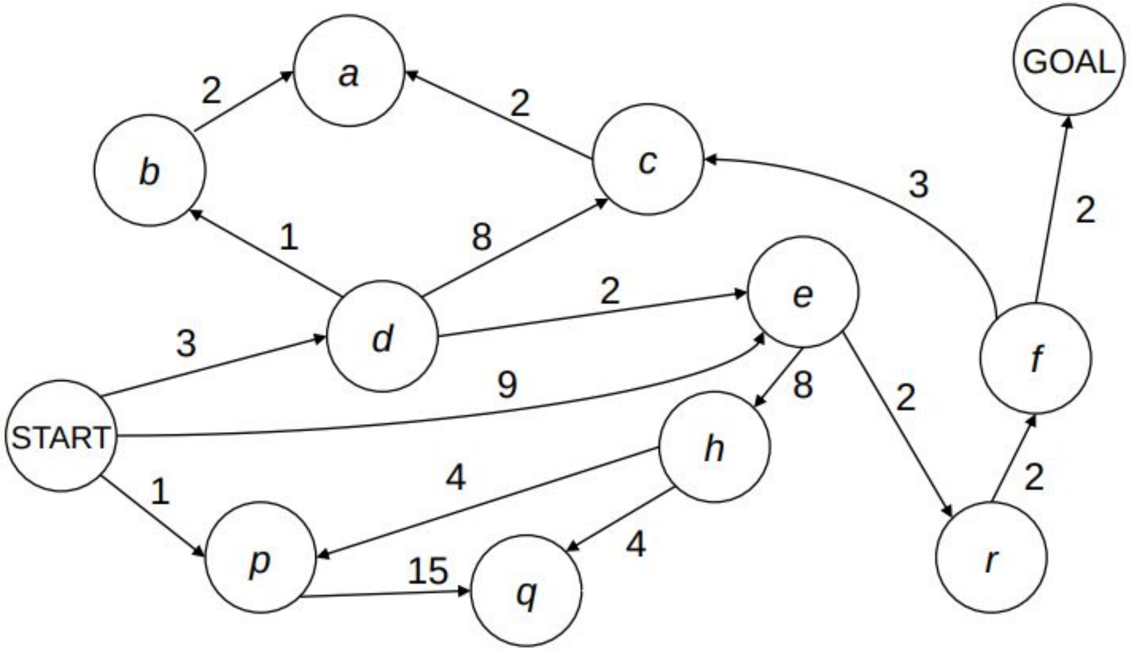
Is it complete? i.e. Is it guaranteed to find a solution if one exists?

- Yes. Will find the shallowest solution

Is it optimal? i.e. Guaranteed to find the least cost path?

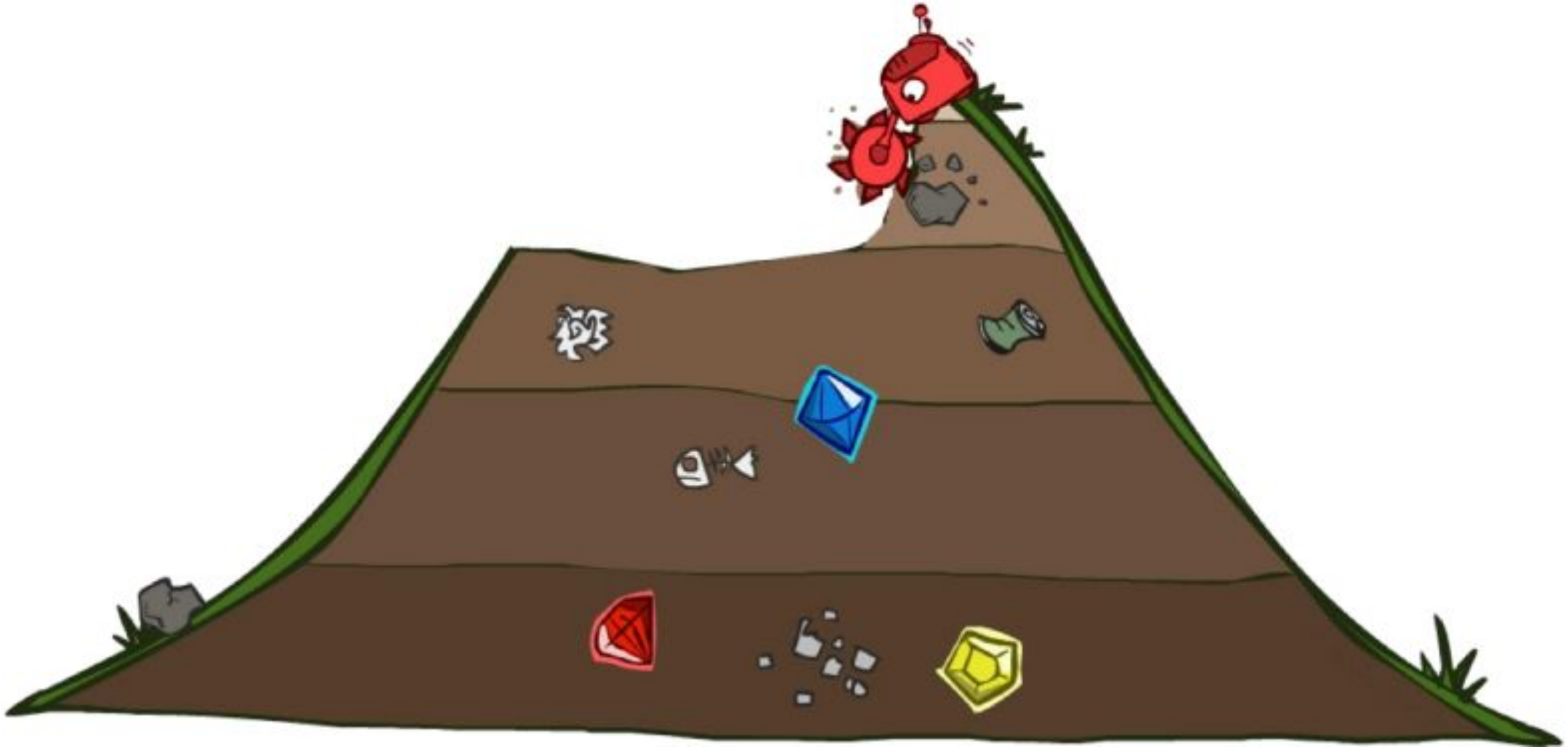
- Yes. Only if costs are all 1. Best in terms of number of actions
    - No attempt to search least cost action first

Cost Sensitive Search

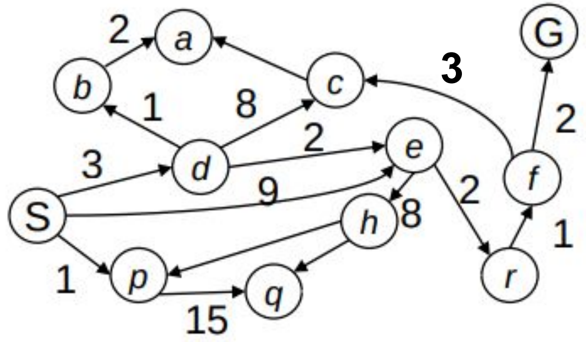


BFS finds the shortest path in terms of number of actions. It does not find the least-cost path. We will now cover a similar algorithm which does find the least-cost path.

Uniform Cost Search

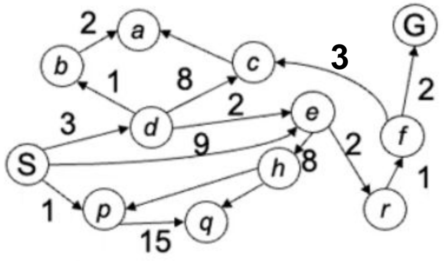


Uniform Cost Search



Strategy: expand a cheapest node first

Uniform Cost Search

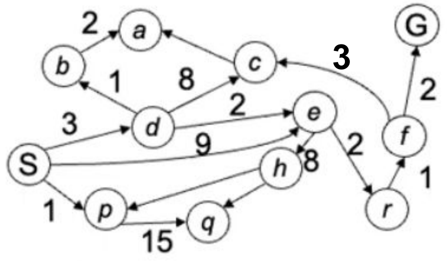


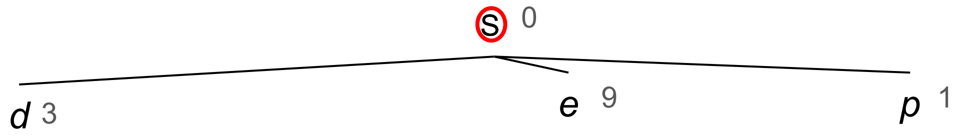
Fringe

s,0

S

Uniform Cost Search

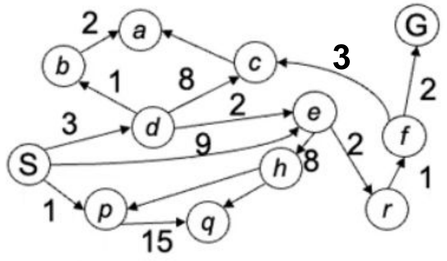




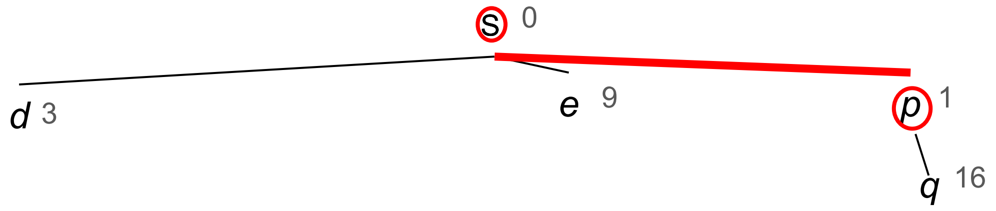
Fringe

~~s,0~~
s→d,3
s→e,9
s→p,1

Uniform Cost Search

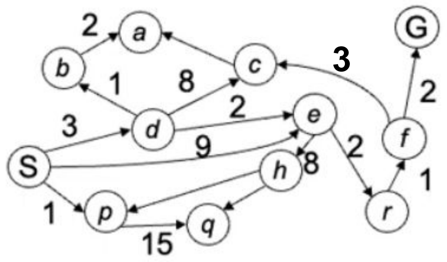


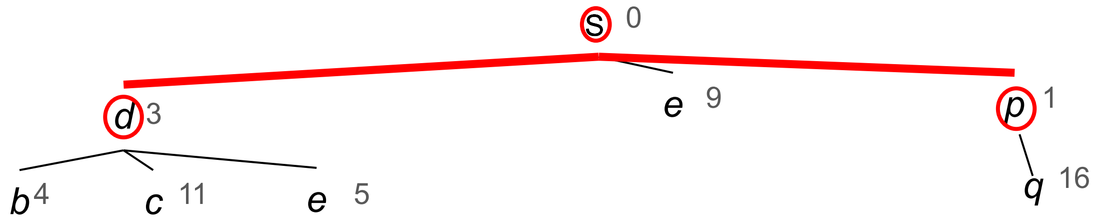
Fringe



~~s,0~~
s→d,3
s→e,9
~~s→p,1~~
s→p→q,16

Uniform Cost Search

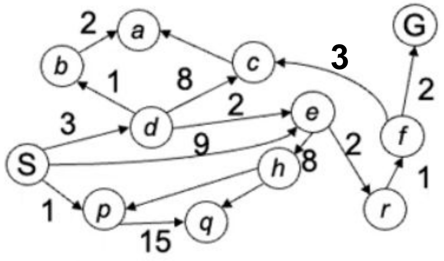


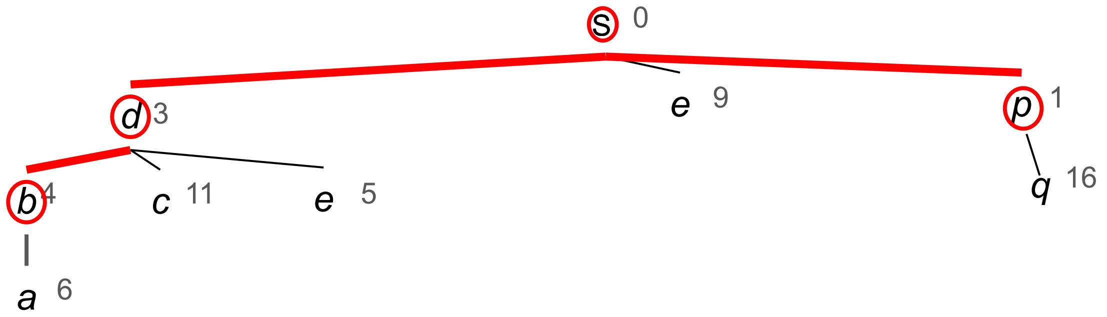


Fringe

~~s,0~~
~~s→d,3~~
s→e,9
~~s→p,1~~
s→p→q,16
s→d→b,4
s→d→c,11
s→d→e,5

Uniform Cost Search

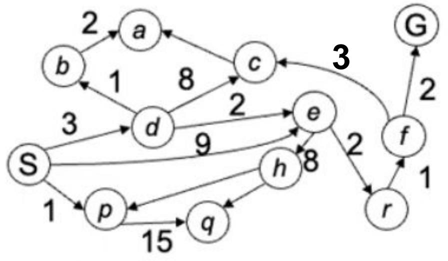


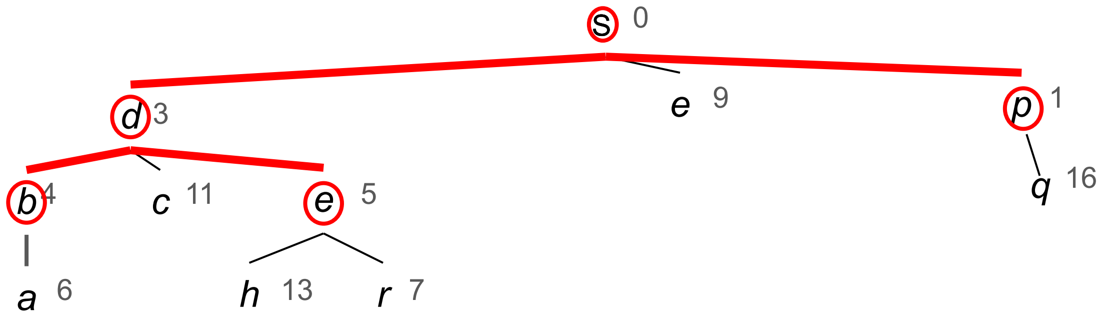


Fringe

~~s,0~~
~~s→d,3~~
s→e,9
~~s→p,1~~
s→p→q,16
~~s→d→b,4~~
s→d→c,11
s→d→e,5
s→d→b→a,6

Uniform Cost Search

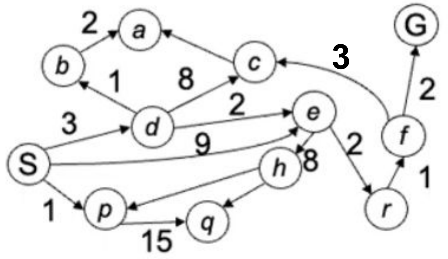




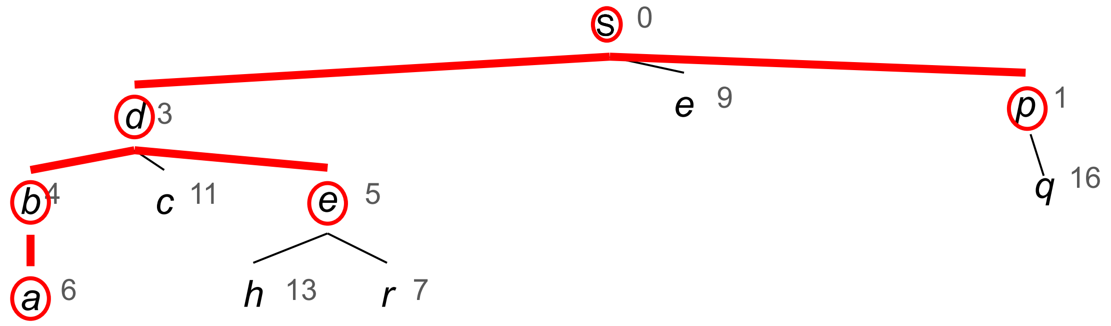
Fringe

~~s,0~~
~~s→d,3~~
s→e,9
~~s→p,1~~
s→p→q,16
~~s→d→b,4~~
s→d→c,11
~~s→d→e,5~~
s→d→b→a,6
s→d→e→h,13
s→d→e→r,7

Uniform Cost Search

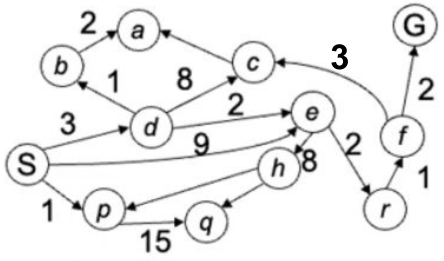


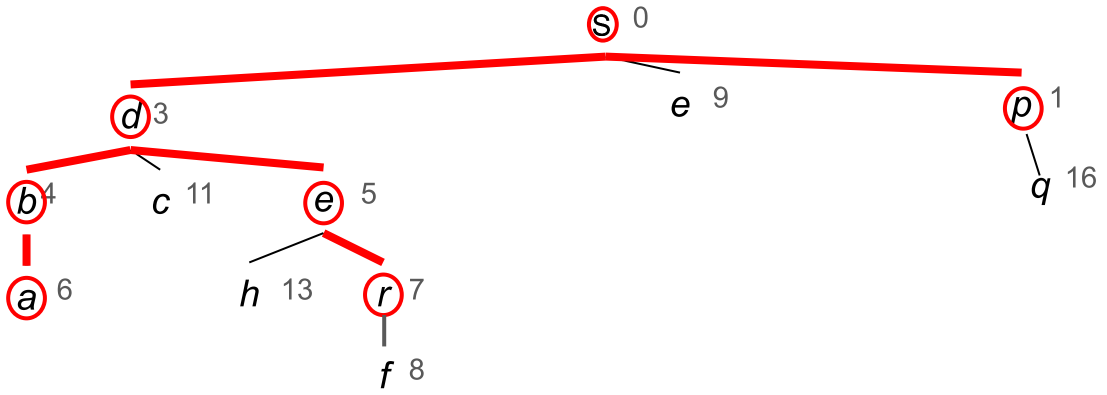
Fringe



~~s,0~~
~~s→d,3~~
s→e,9
~~s→p,1~~
s→p→q,16
~~s→d→b,4~~
s→d→c,11
~~s→d→e,5~~
~~s→d→b→a,6~~
s→d→e→h,13
s→d→e→r,7

Uniform Cost Search

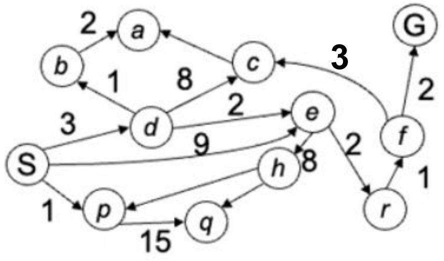


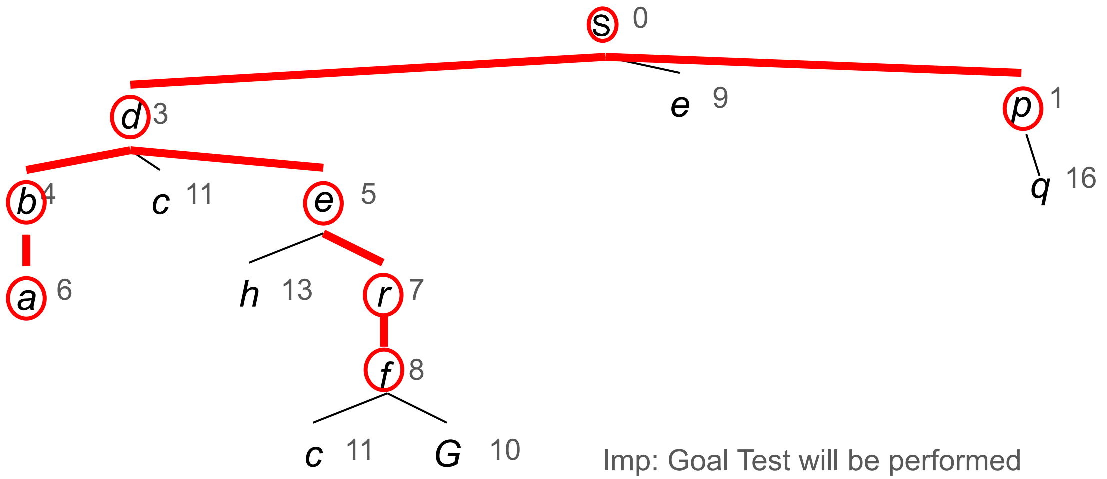


Fringe

~~s,0~~
~~s→d,3~~
s→e,9
~~s→p,1~~
s→p→q,16
~~s→d→b,4~~
s→d→c,11
~~s→d→e,5~~
~~s→d→b→a,6~~
s→d→e→h,13
~~s→d→e→r,7~~
s→d→e→r→f,8

Uniform Cost Search



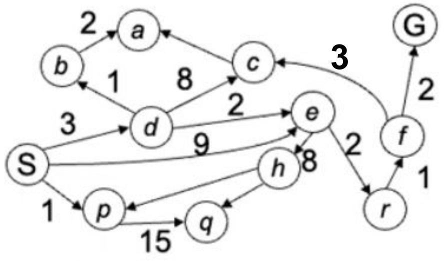


Imp: Goal Test will be performed only on the node being expanded.

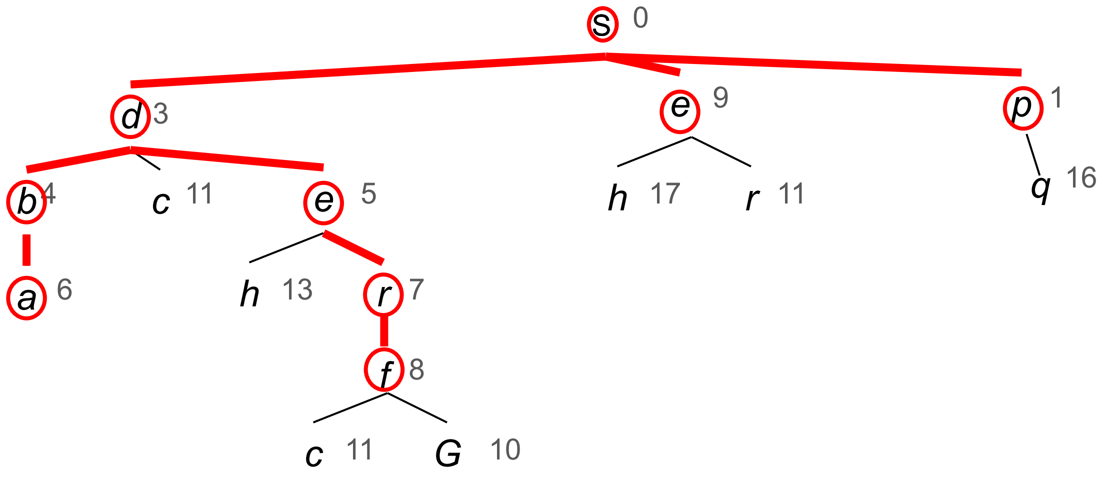
Fringe

~~s,0~~
~~s→d,3~~
s→e,9
~~s→p,1~~
s→p→q,16
~~s→d→b,4~~
s→d→c,11
~~s→d→e,5~~
~~s→d→b→a,6~~
s→d→e→h,13
~~s→d→e→r,7~~
~~s→d→e→r→f,8~~
s→d→e→r→f→c,11
s→d→e→r→f→G,10

Uniform Cost Search

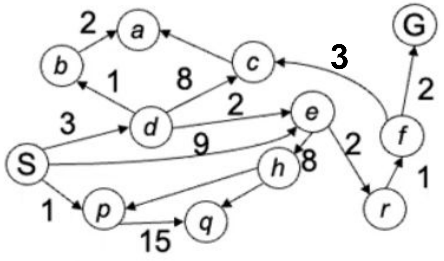


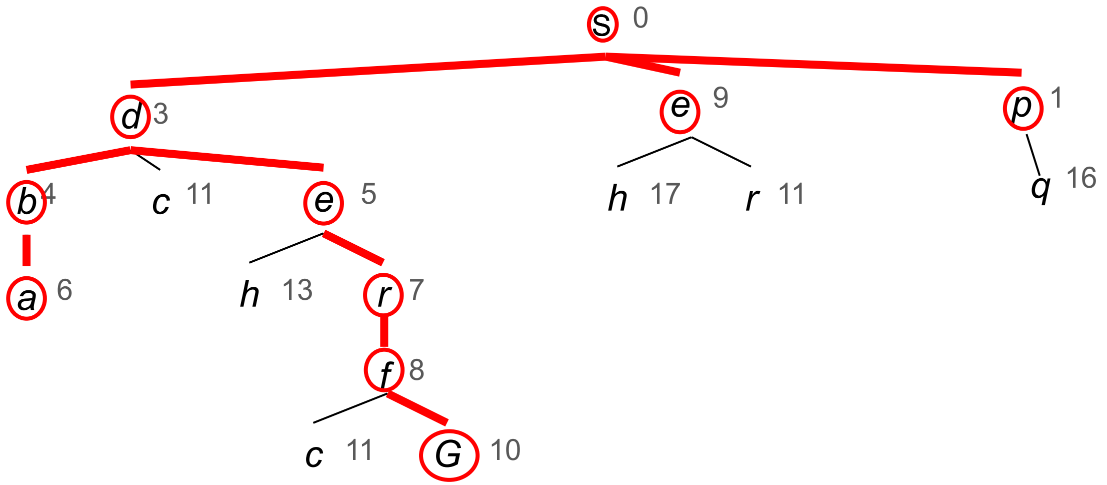
Fringe



~~s,0~~
~~s→d,3~~
~~s→e,9~~
~~s→p,1~~
s→p→q,16
~~s→d→b,4~~
s→d→c,11
~~s→d→e,5~~
~~s→d→b→a,6~~
s→d→e→h,13
~~s→d→e→r,7~~
~~s→d→e→r→f,8~~
s→d→e→r→f→c,11
s→d→e→r→f→G,10
s→e→h,17
s→e→r,11

Uniform Cost Search





Fringe

~~s,0~~

~~s→d,3~~

~~s→e,9~~

~~s→p,1~~

s→p→q,16

~~s→d→b,4~~

s→d→c,11

~~s→d→e,5~~

~~s→d→b→a,6~~

s→d→e→h,13

~~s→d→e→r,7~~

~~s→d→e→r→f,8~~

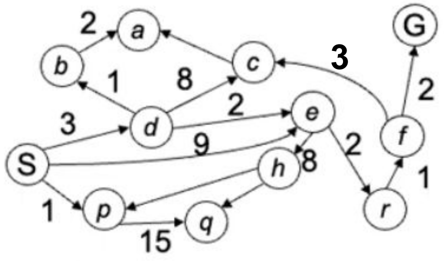
s→d→e→r→f→c,11

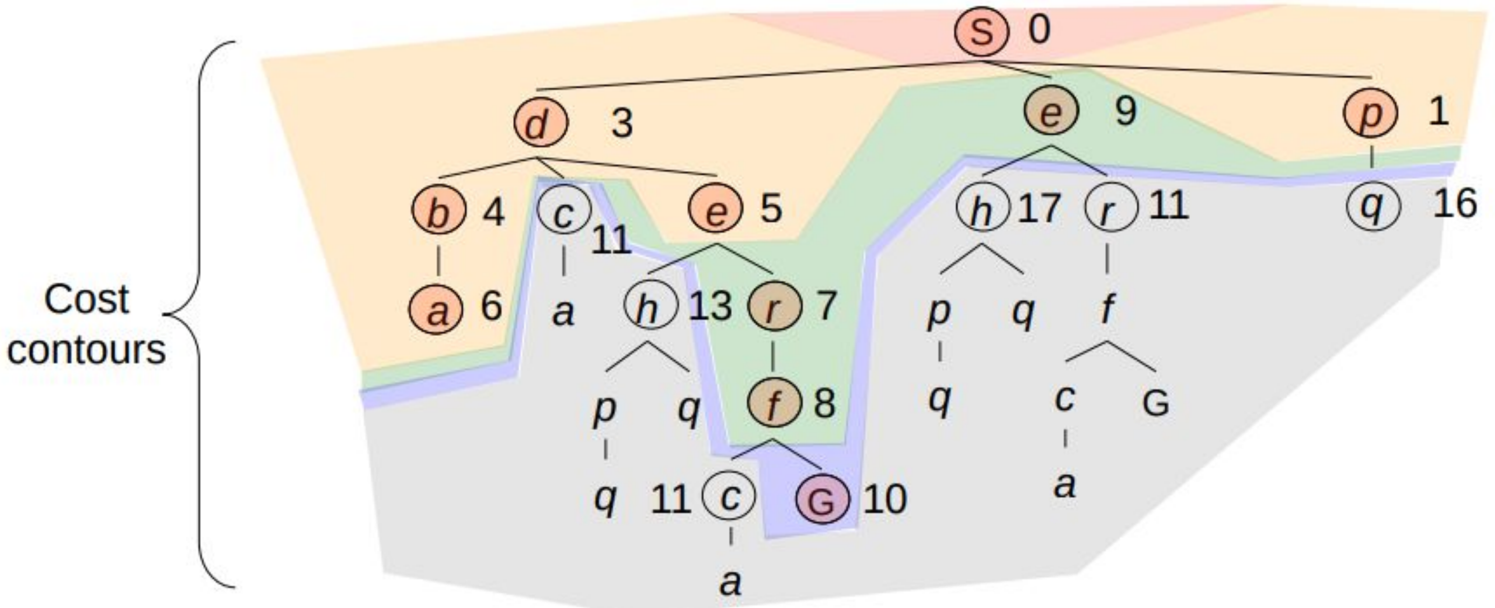
solution s→d→e→r→f→G,10

s→e→h,17

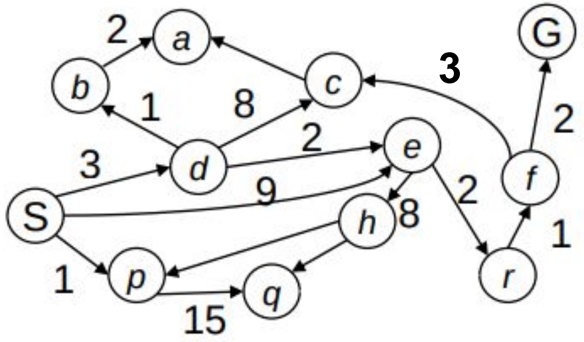
s→e→r,11

Uniform Cost Search





Uniform Cost Search



Strategy: expand a cheapest node first

Implementation: List of partial plans/fringe is a priority queue (priority: cumulative cost)

- Priority Queue: Data structure or in simple terms a type of storage from which you can get items in terms of priority irrespective of the order of insertion

Aside: Priority Queue Implementation

- Priority Queue: Data structure or in simple terms a type of storage from which you can get items in terms of priority irrespective of the order of insertion

```
from queue import PriorityQueue

class FringeUsingPriorityQueue:
    def __init__(self):
        self.lt=PriorityQueue()

    def insert(self,i,cost):
        self.lt.put((cost,i))

    def delete(self):
        if(self.lt.empty()):
            el = -1
        else:
            el = self.lt.get()

        return el
```

```
fringe = FringeUsingPriorityQueue()

fringe.insert((‘a’,10))
fringe.insert((‘b’,15))
fringe.insert((‘c’,5))
fringe.insert((‘d’,11))

print(fringe.remove()) # Output?????
```

For Background, you can look into Heap data structure and Priority Queue using Heap

Uniform Cost Search: Properties

Is it complete? i.e. Is it guaranteed to find a solution if one exists?

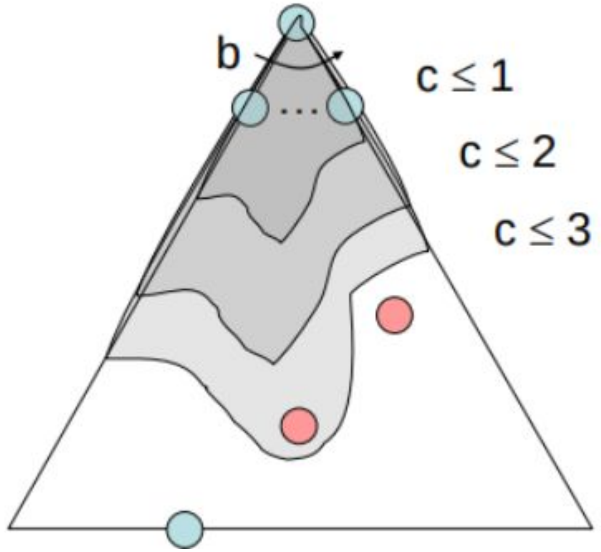
- Yes, assuming best solution has a finite cost and minimum arc cost is positive.

Is it optimal? i.e. Guaranteed to find the least cost path?

- Yes, since it is similar to BFS but traversing cost-wise instead of level-wise

The bad aspect:

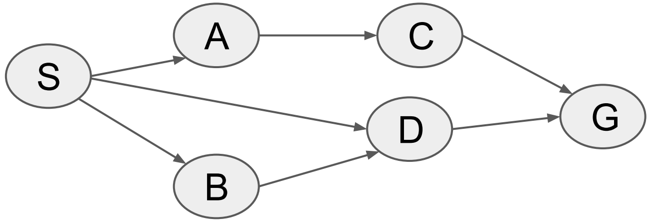
- Explores options in every "direction"
- No information about goal location



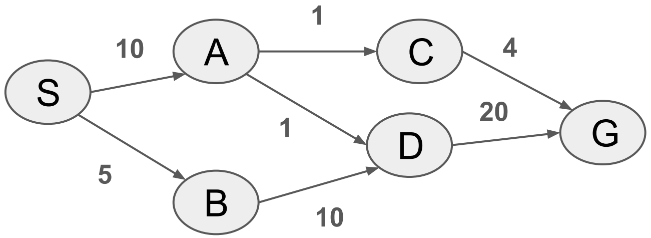
Practice

Which chooses the shorter plan? BFS, DFS, UCS

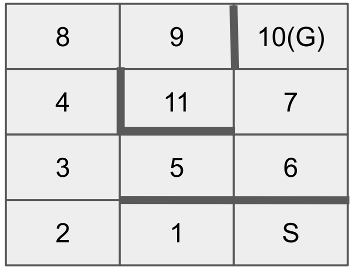
Ques1



Ques2



Ques3


8
9
10(G)
4
11
7
3
5
6
2
1
S