STEVENS
INSTITUTE OF TECHNOLOGY
1870

Artificial Intelligence

Computer Science, CS541 - A

Jonggi Hong

Announcements

- HW #1: Uninformed and Informed Search
  - Has been released! Due 11:59 pm, Tuesday, February 14.

- Slides of Week 1 are updated.

- Questions

1. What is the grading policy for the course is it relative grading?
2. Will the assignments be coding assignments or non-coding assignments or mix of both?
3. Will there be option for take home exam for the final or will it be mandatory in person exam?
4. Last question in the slides the office hours room is mentioned as 251 so is it Gateway South 251?

STEVENS
INSTITUTE OF TECHNOLOGY
1870

Recap

Uninformed search

Search terminology

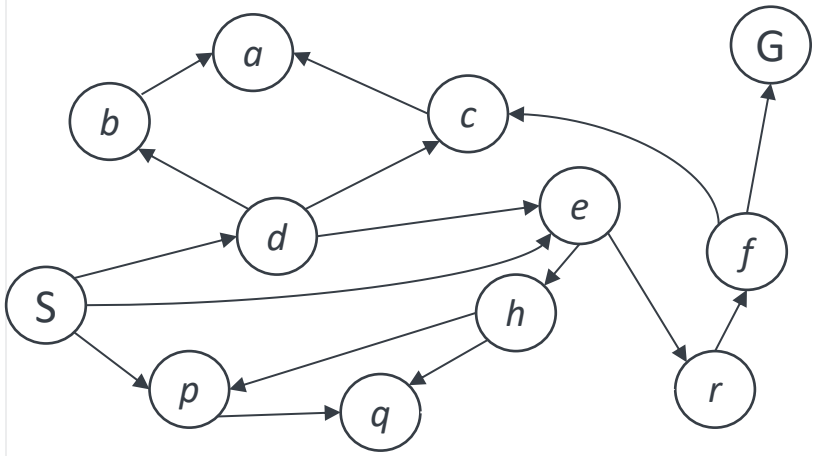
- **Expansion** of nodes
  - As states are explored, the corresponding nodes are expanded by legal operators (*i.e.*, actions).

- The fringe (frontier) is the set of nodes that are newly **generated** and not yet visited.
  - Newly generated nodes are added to the fringe.

- Search strategy
  - determines the selection of the next node to be expanded.
  - can be achieved by ordering the nodes in the fringe.

State Space Graph and Search Tree

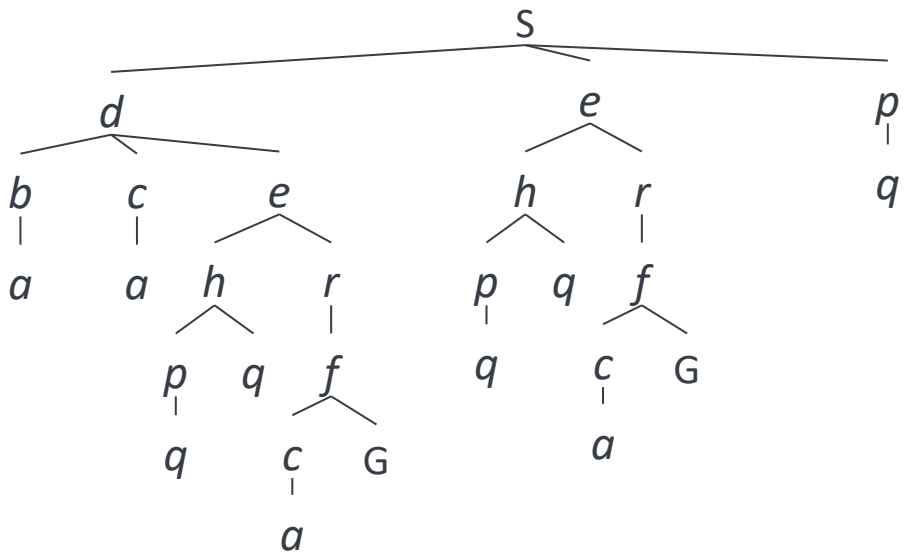
State Space Graph



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.

Search Tree



General tree search

function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier (according to strategy)
    if the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

Main question: which fringe nodes to explore?

This works in general but it can be more efficient when the actions have uniform costs.

General tree search

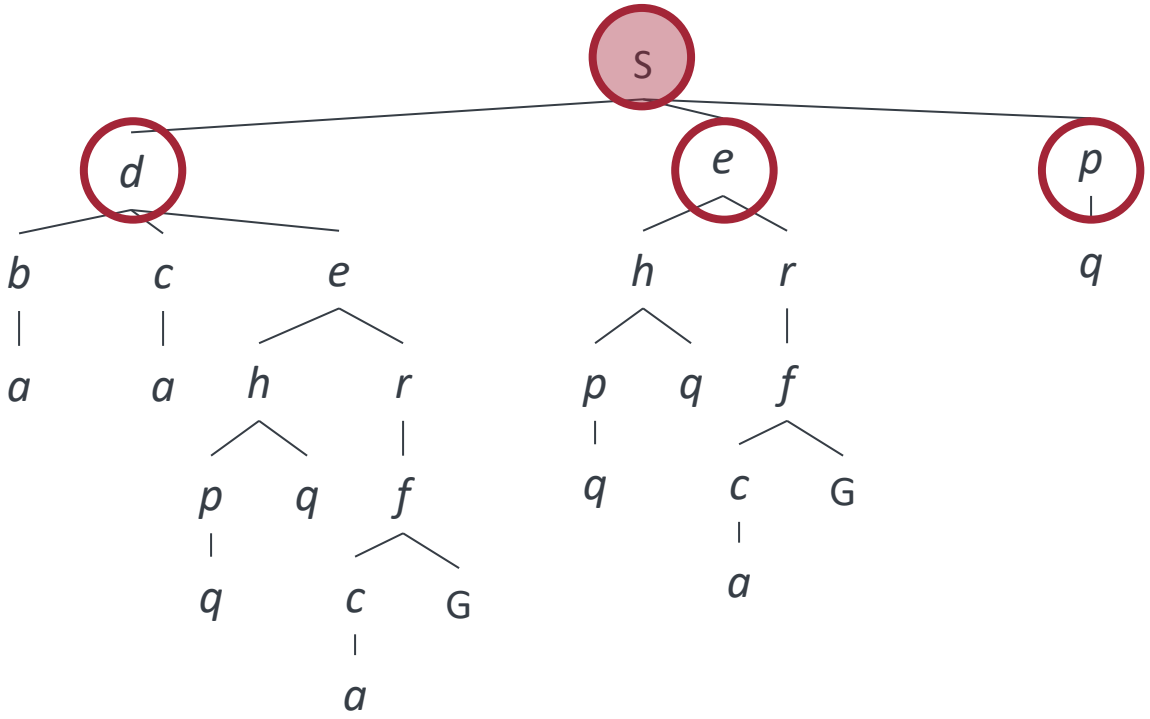
```
function TREE-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier   (according to strategy)
        for succ in successors(node):
                if succ is a goal state then return the corresponding solution
                frontier.insert(<node, succ>)
```

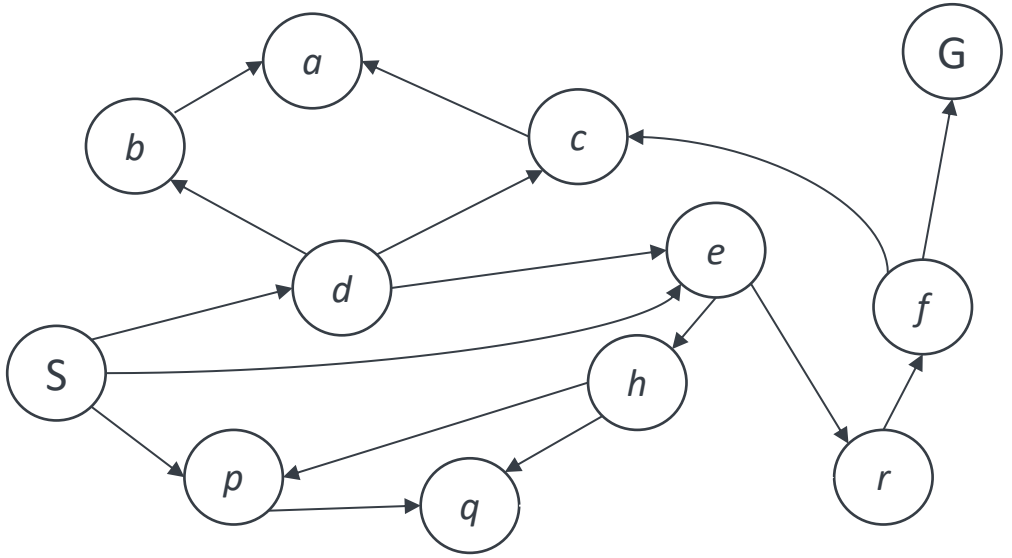
Main question: which fringe nodes to explore?

If the actions have uniform costs, the search can stop earlier.

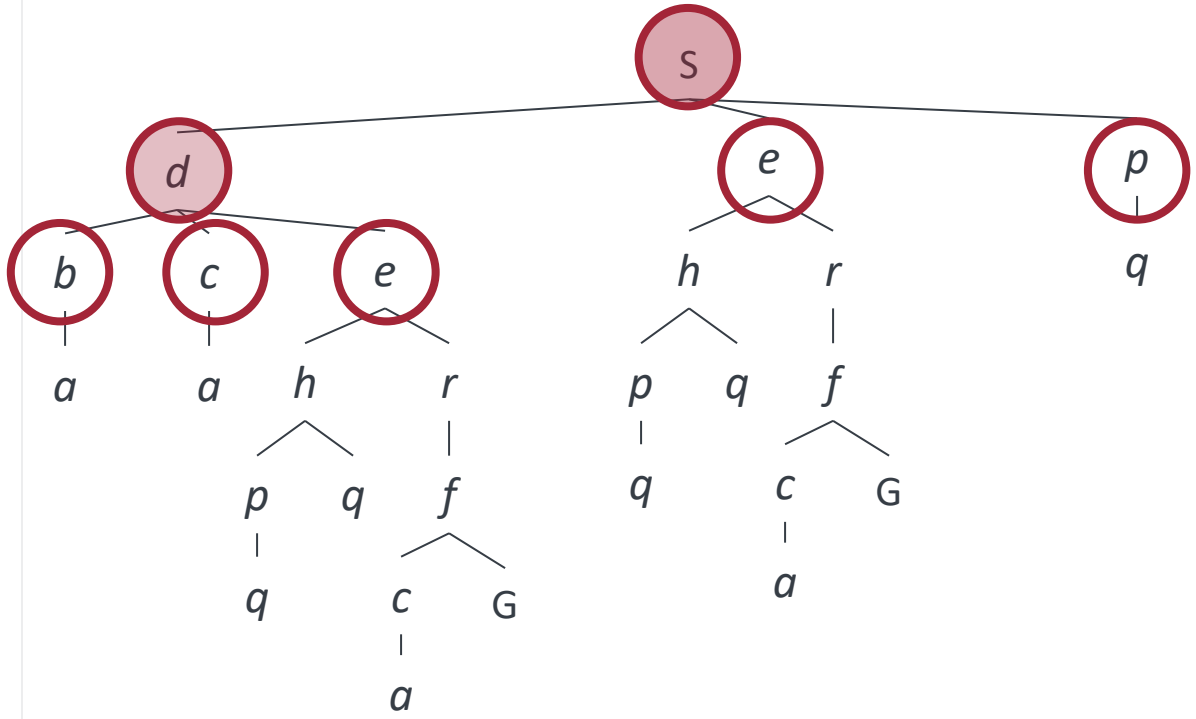
Depth-First Search (DFS)

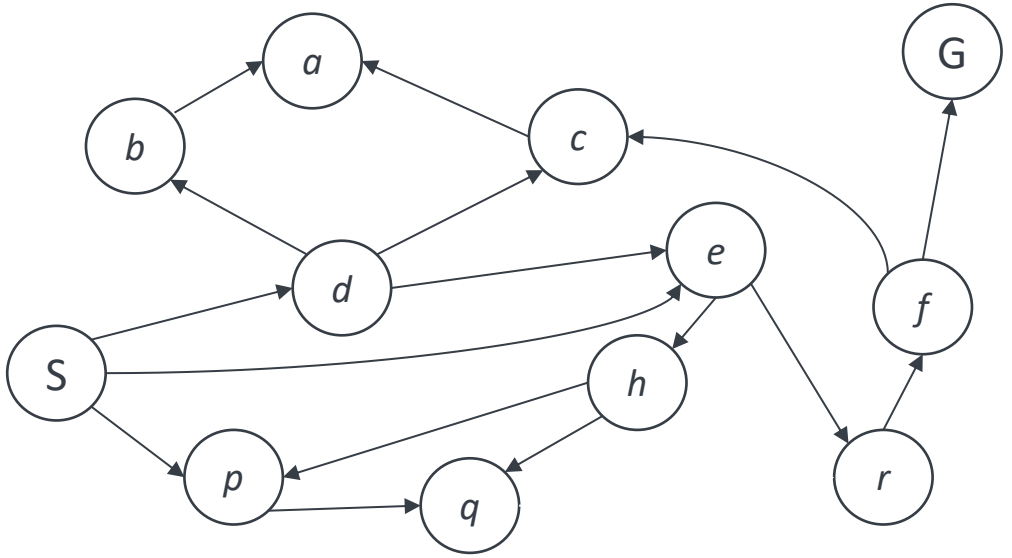
Depth-First Search (DFS)



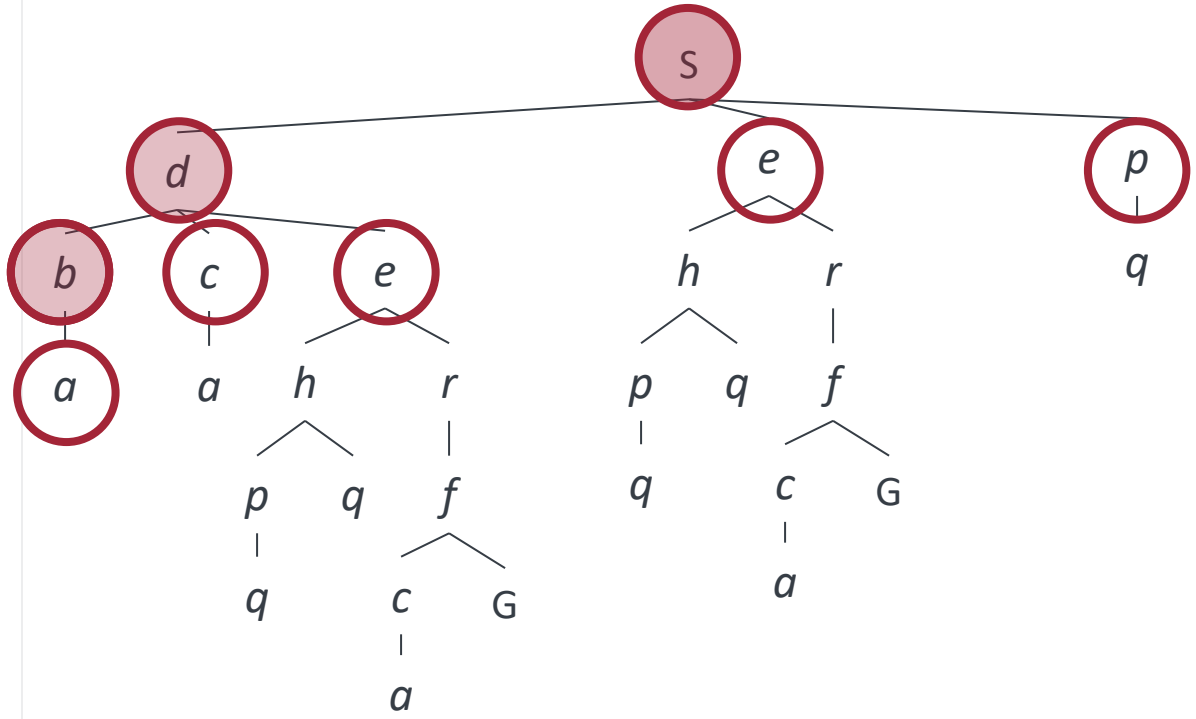


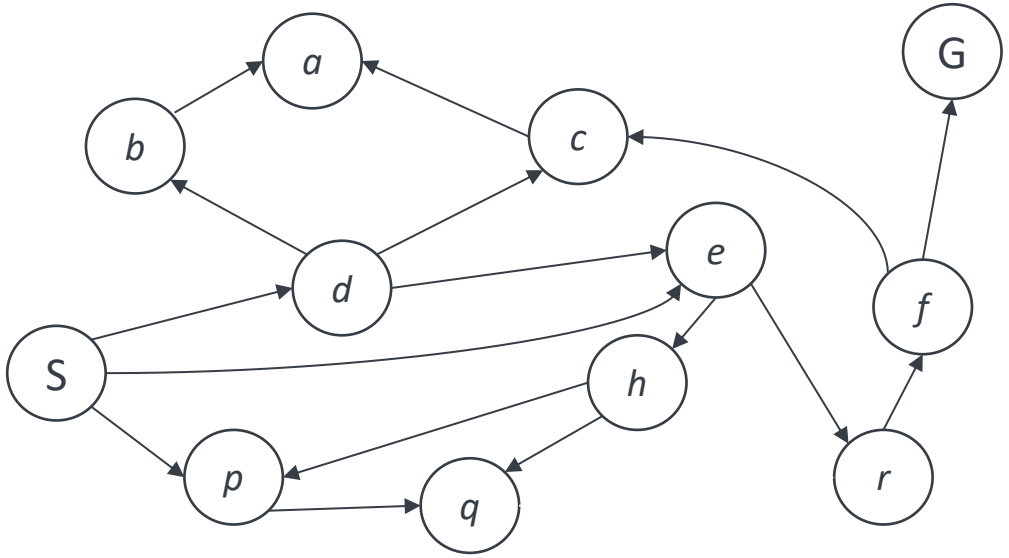
Depth-First Search (DFS)



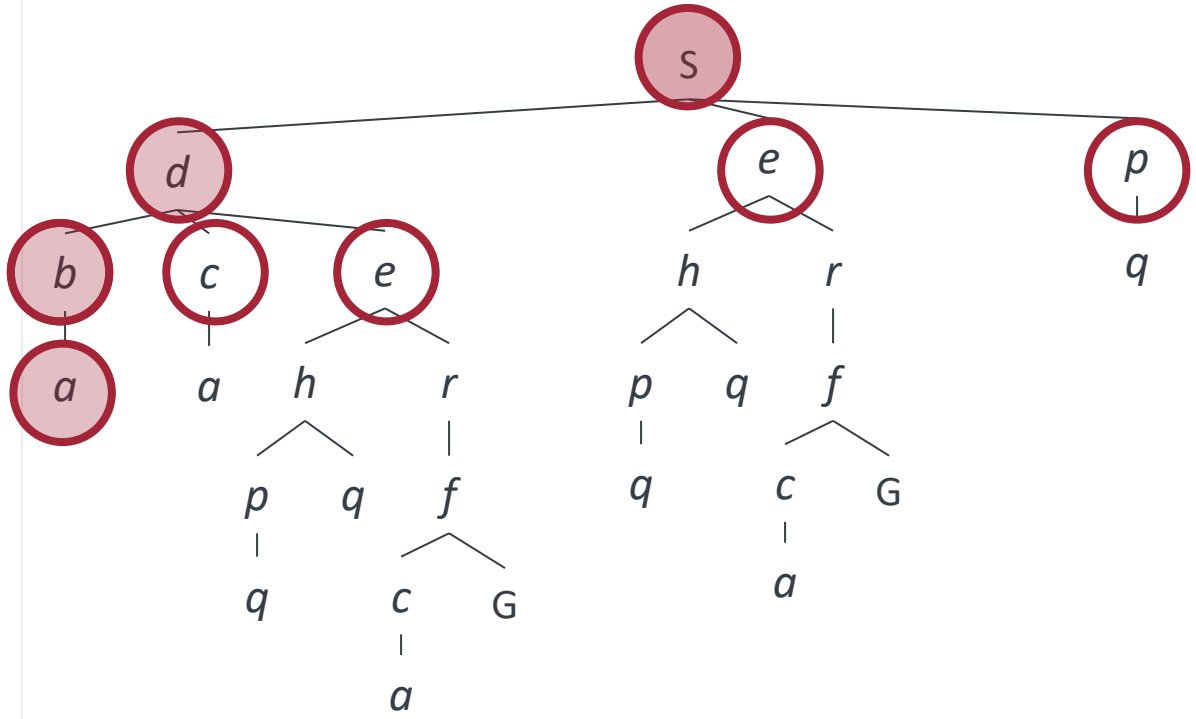


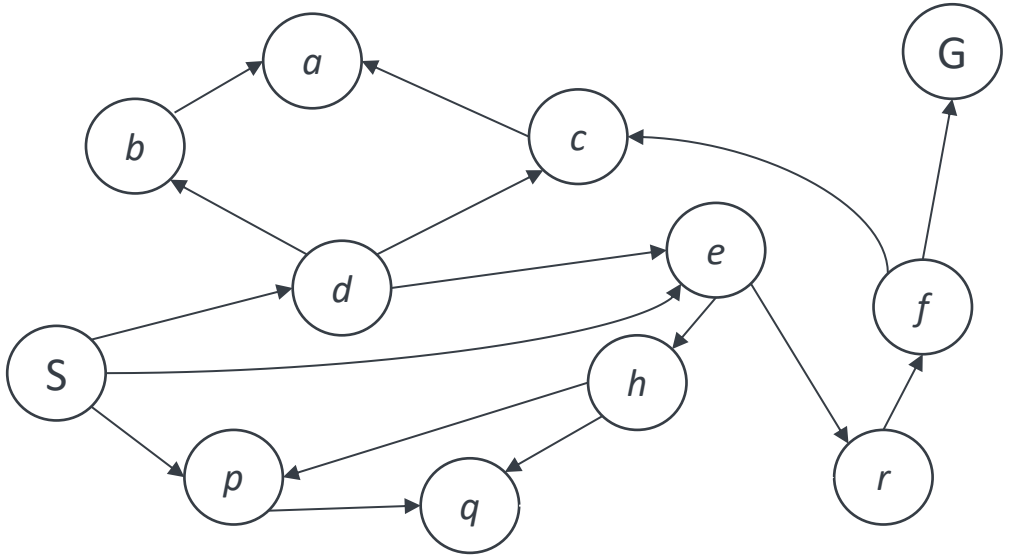
Depth-First Search (DFS)



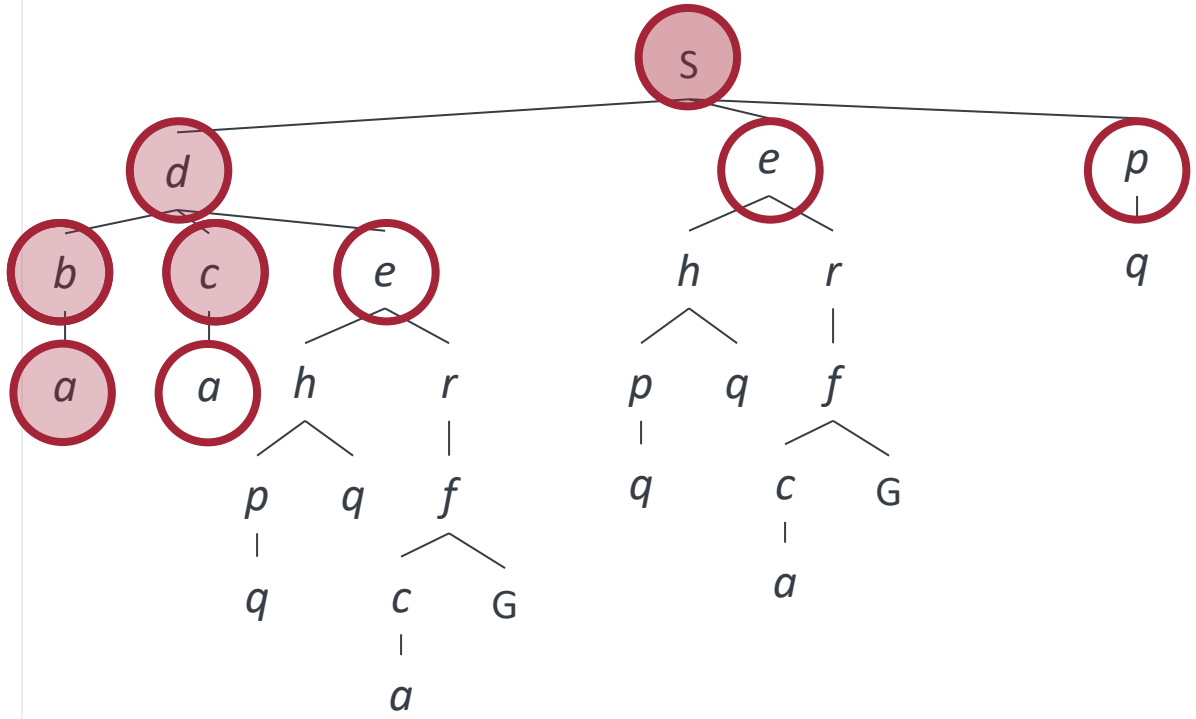


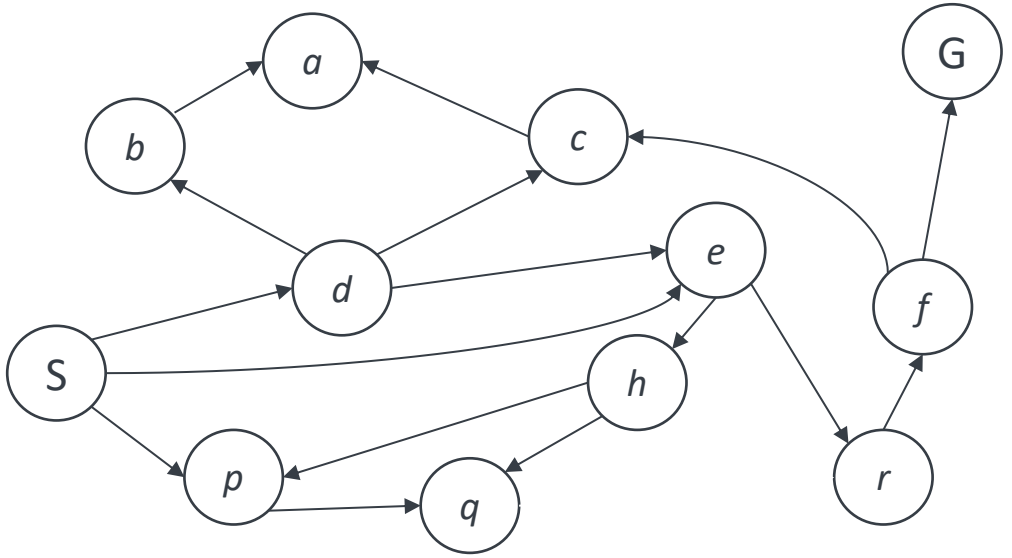
Depth-First Search (DFS)



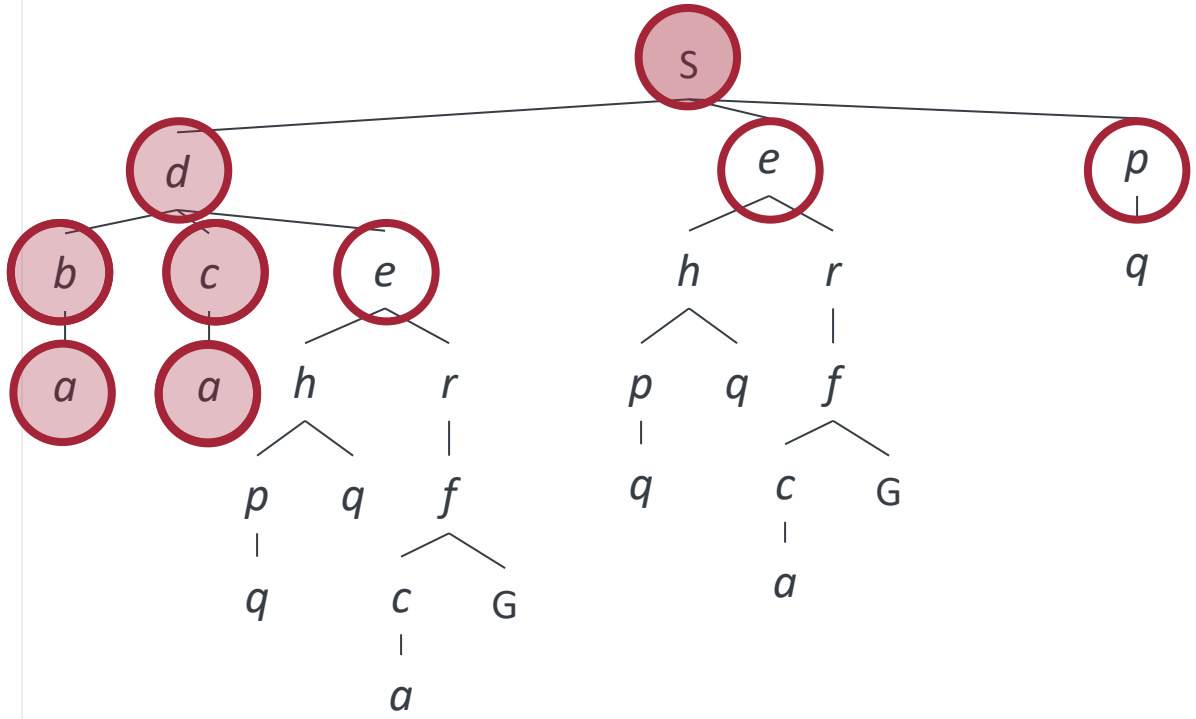


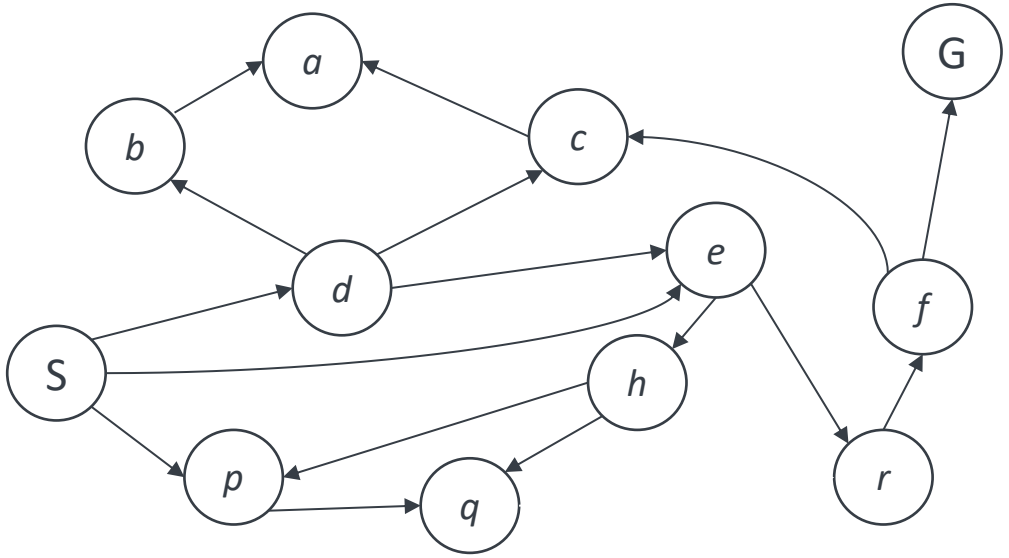
Depth-First Search (DFS)



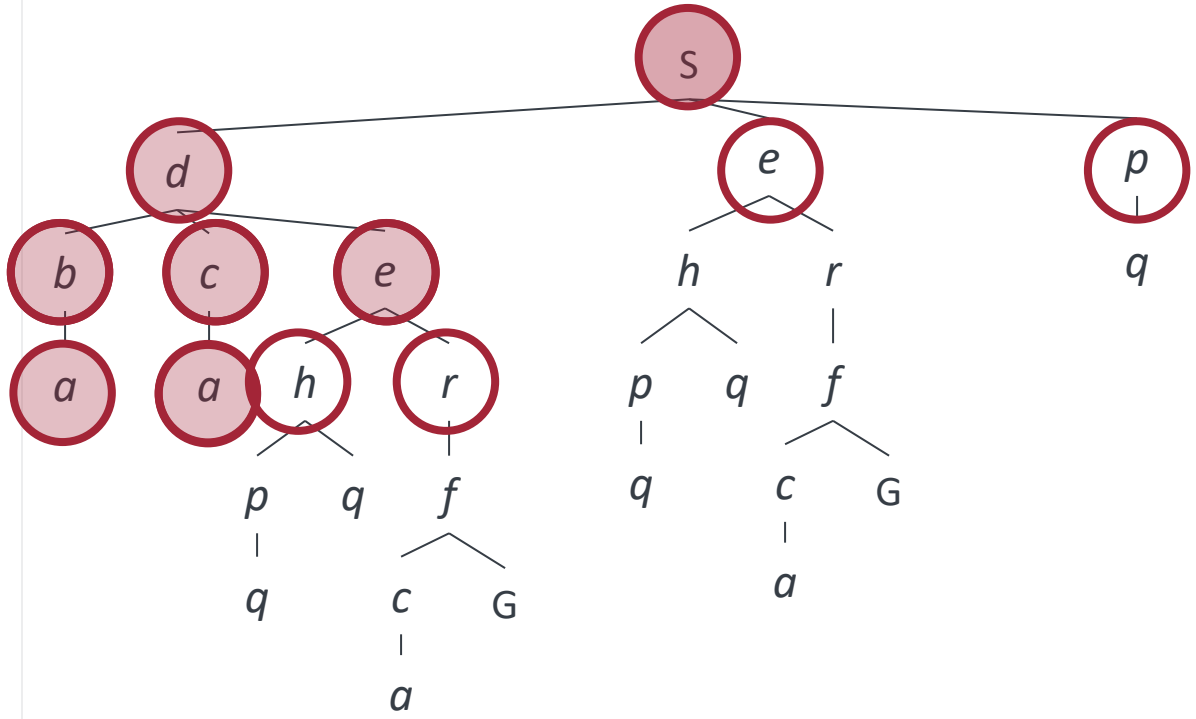


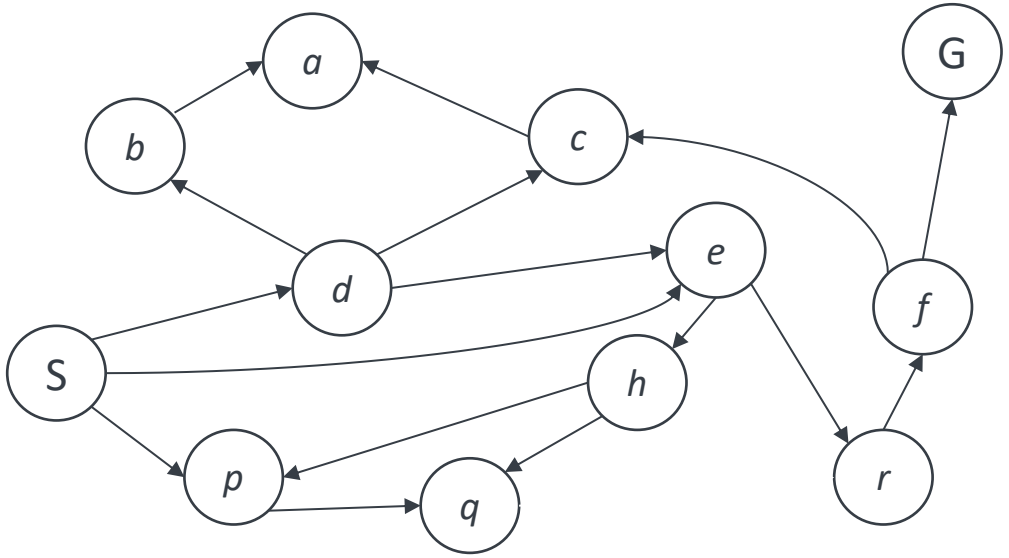
Depth-First Search (DFS)



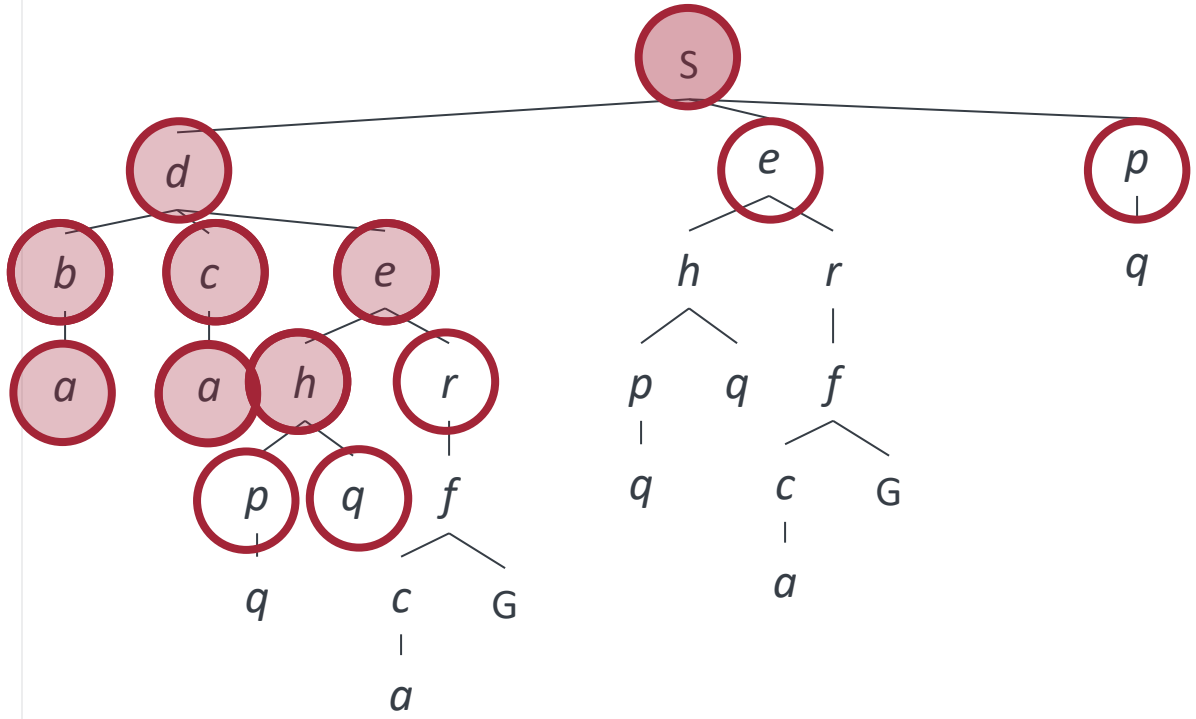


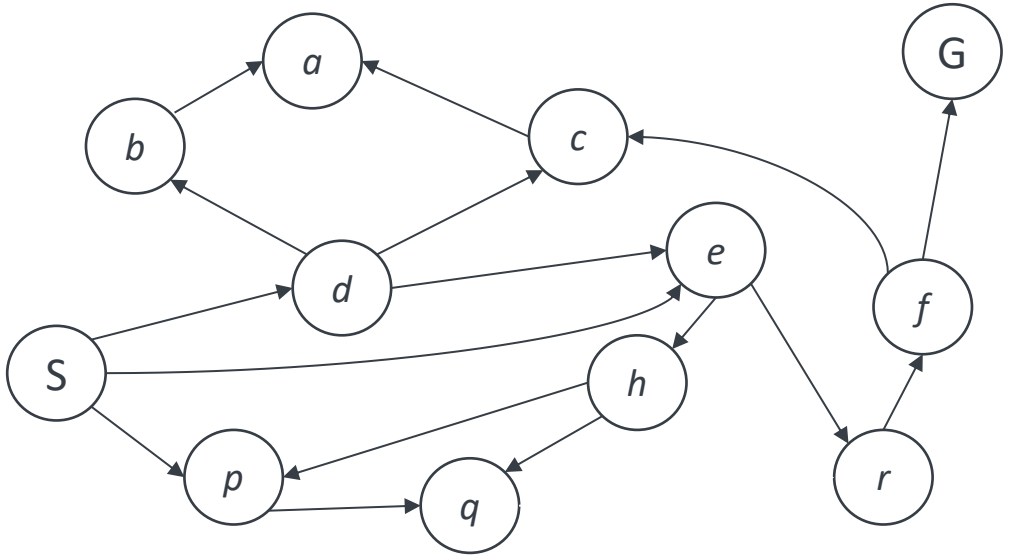
Depth-First Search (DFS)



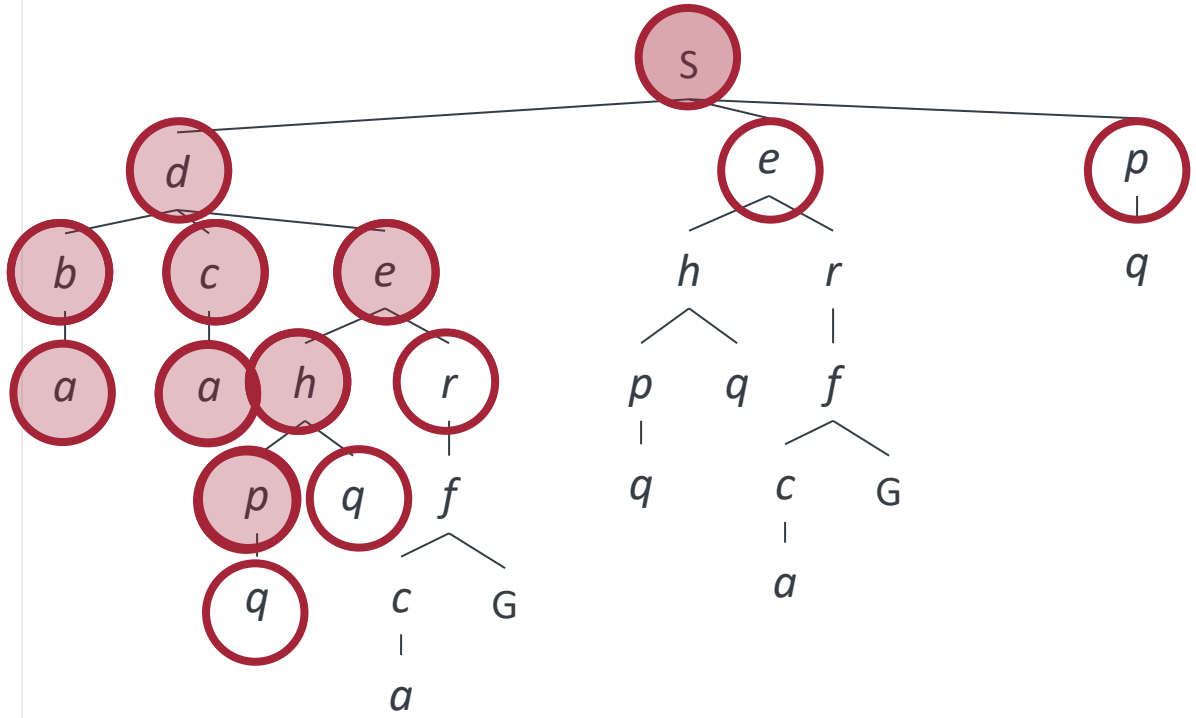


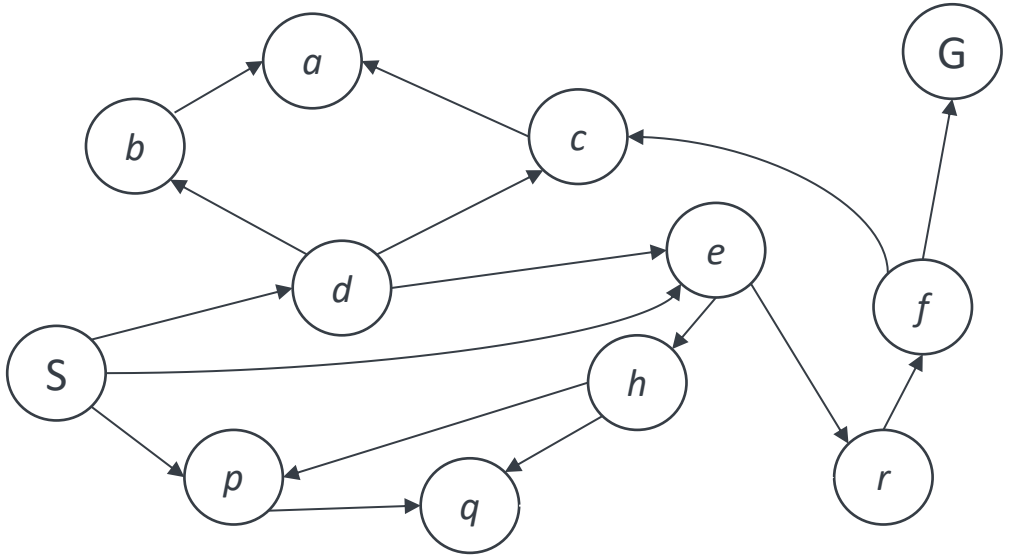
Depth-First Search (DFS)



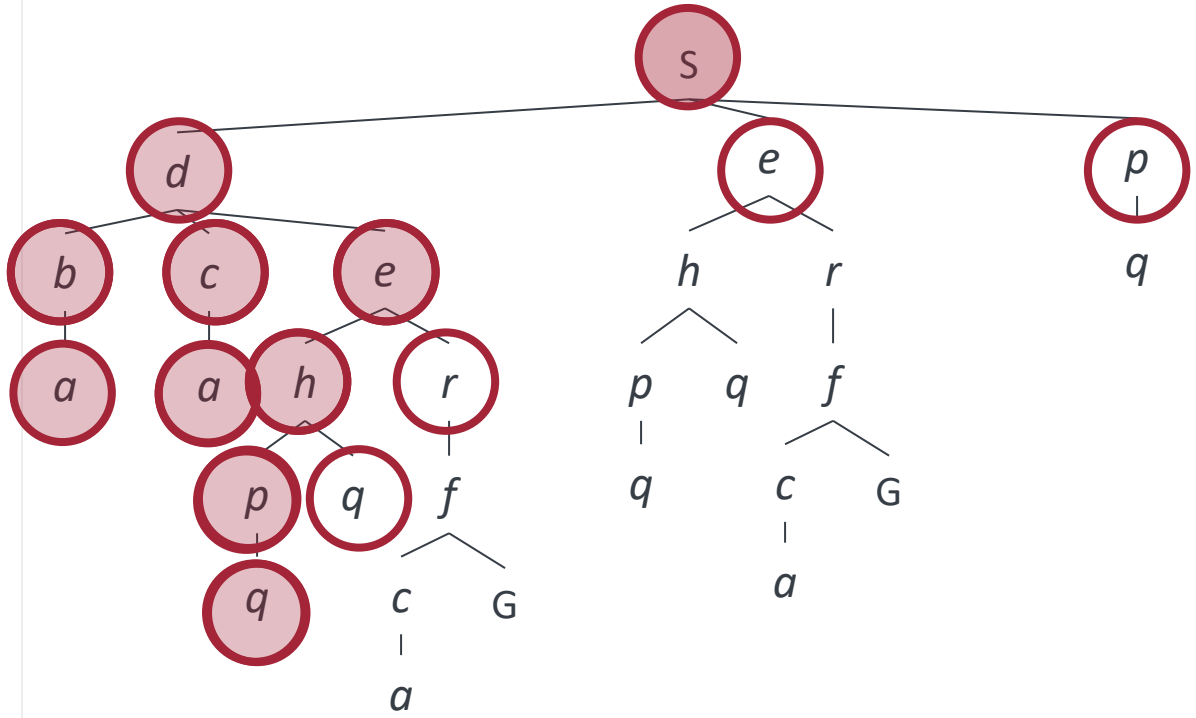


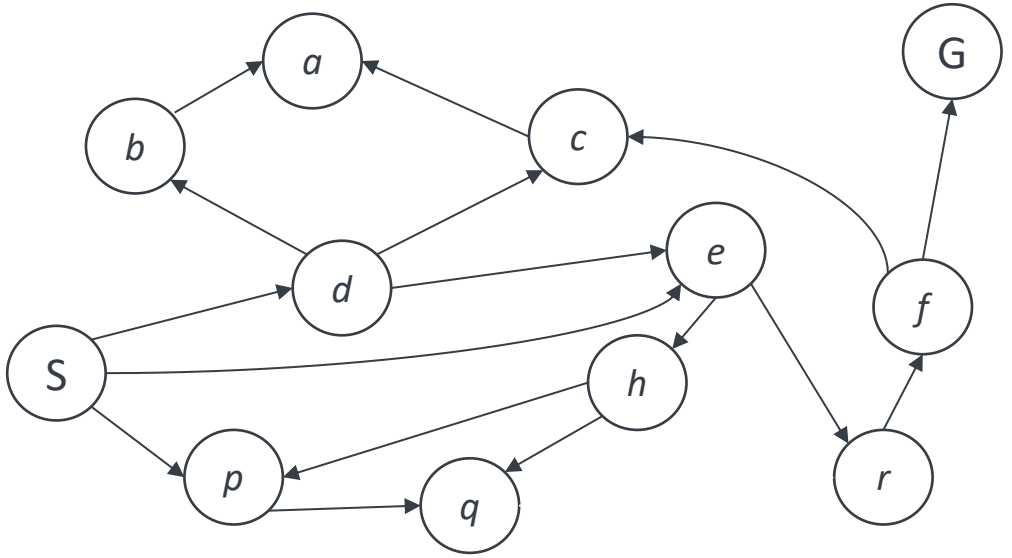
Depth-First Search (DFS)



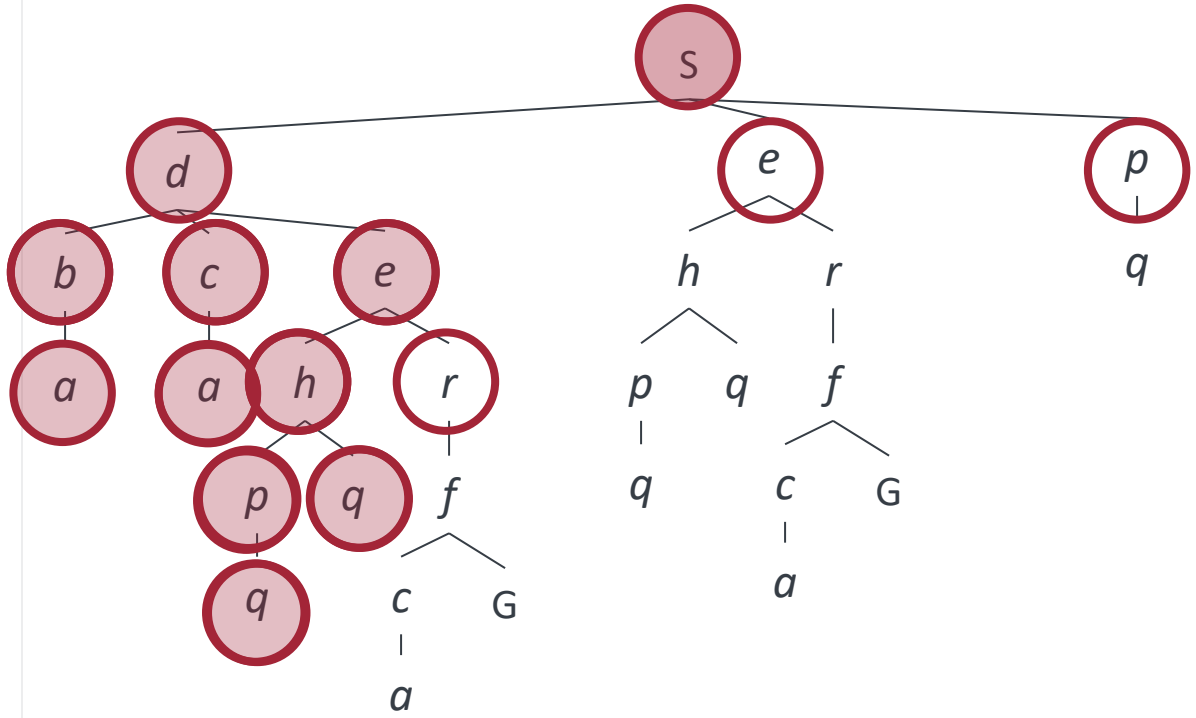


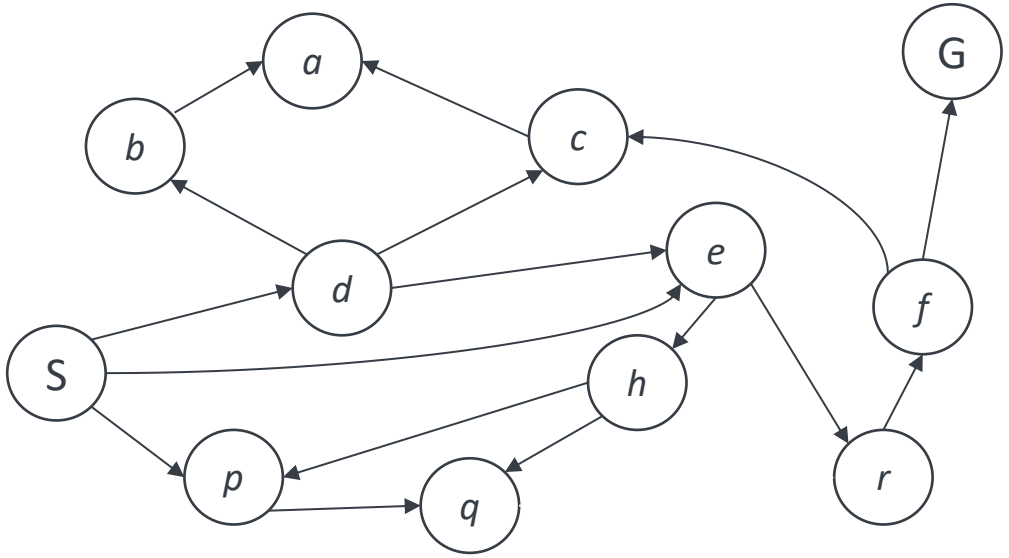
Depth-First Search (DFS)



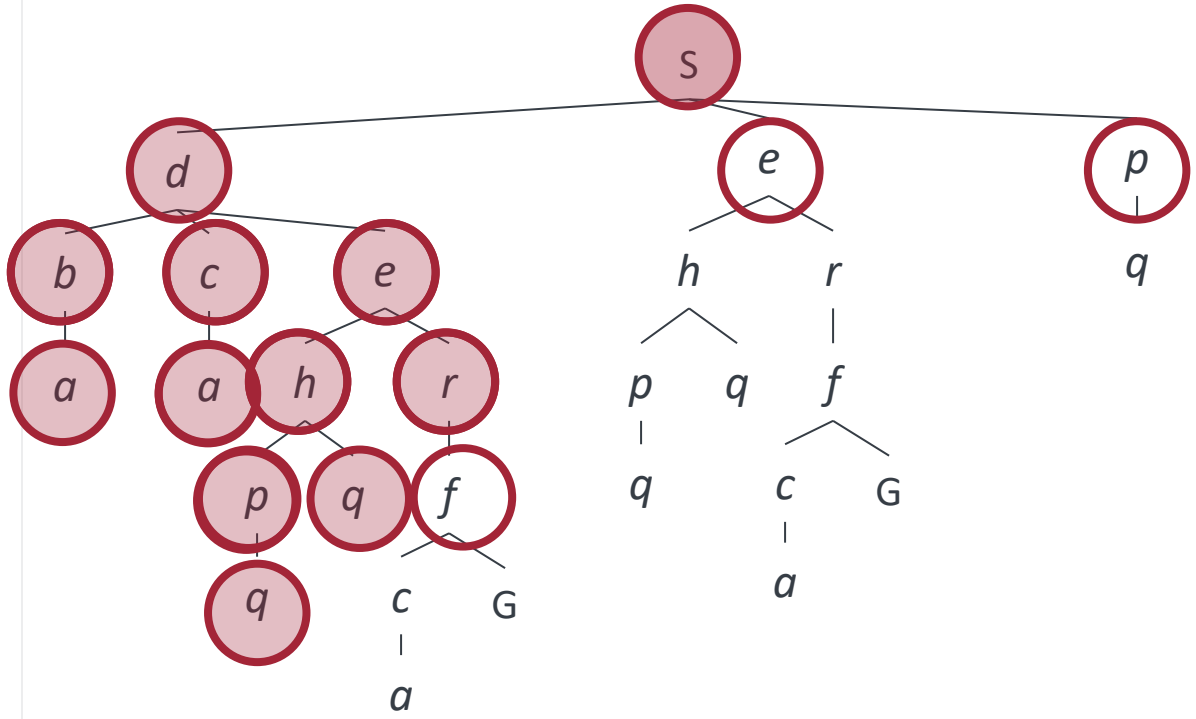


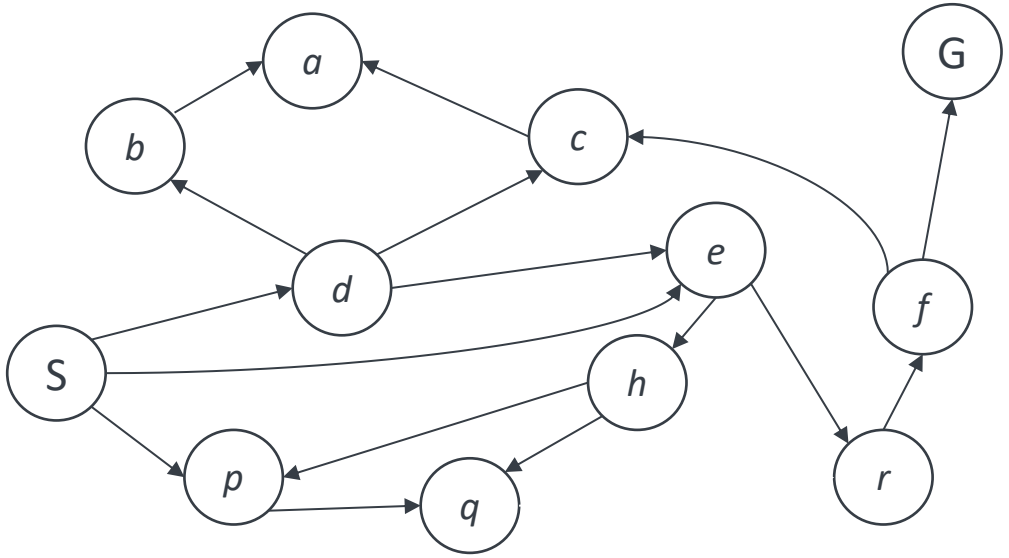
Depth-First Search (DFS)



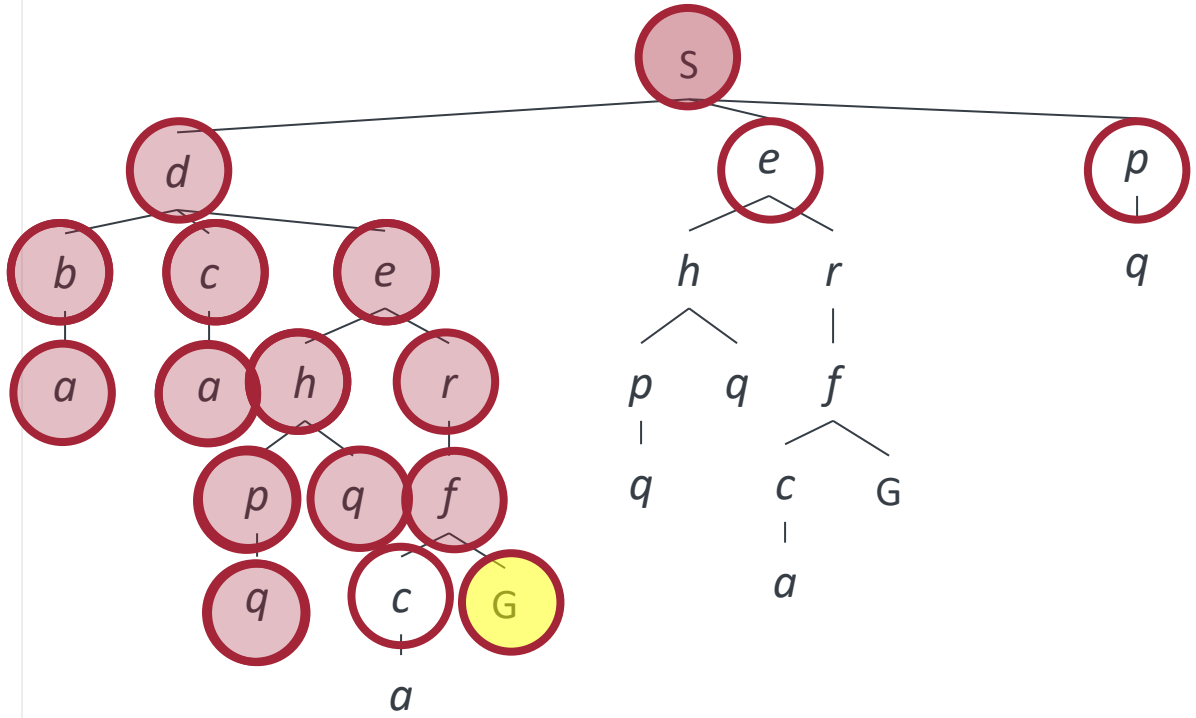


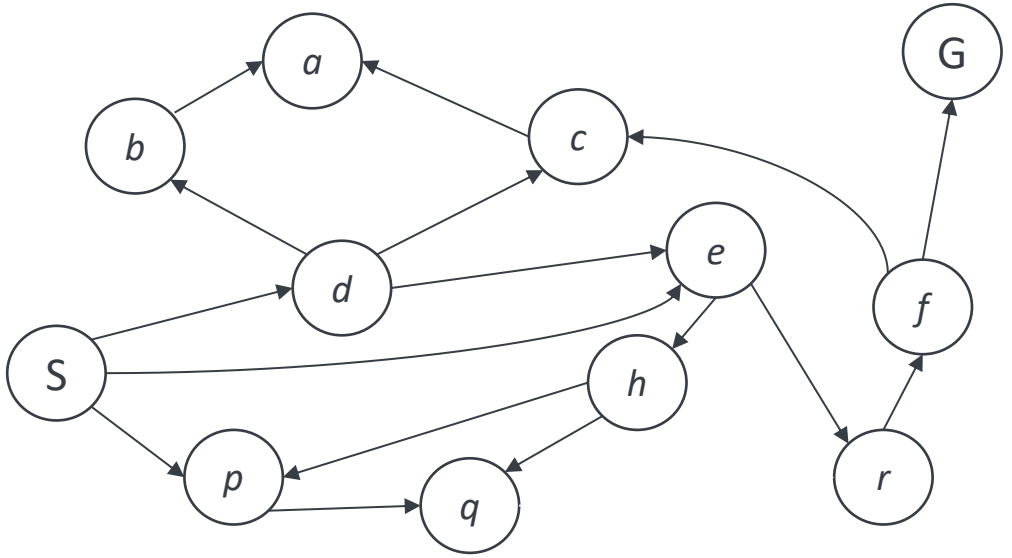
Depth-First Search (DFS)





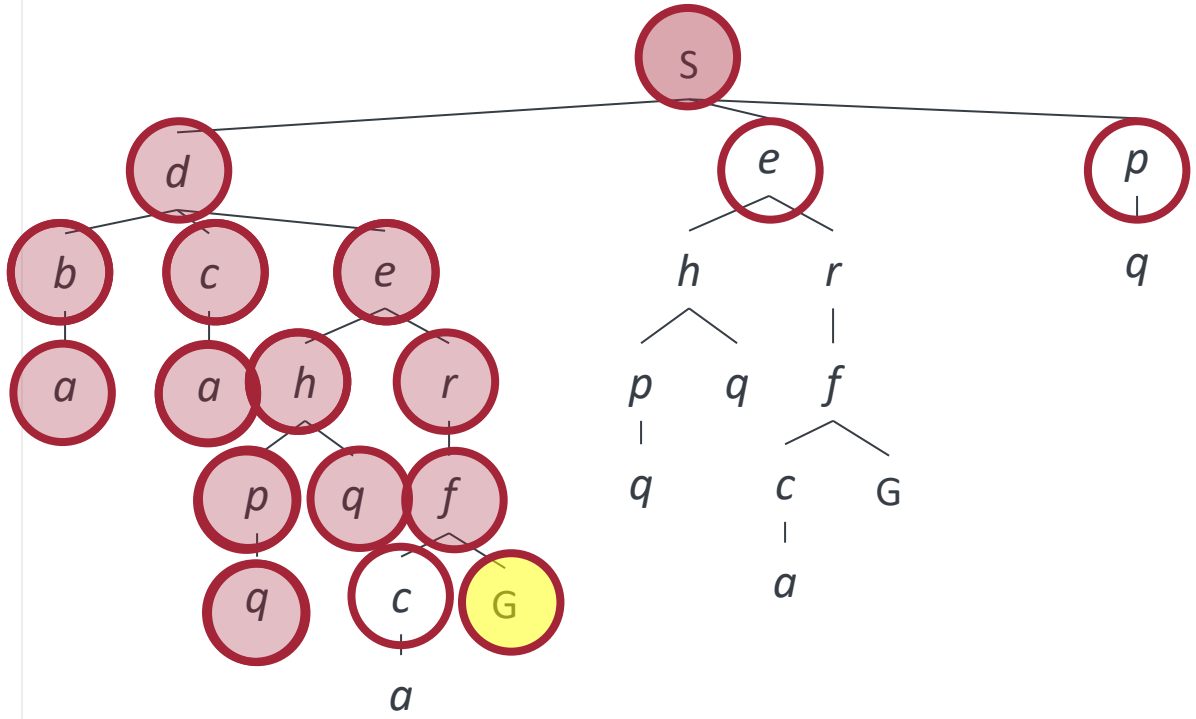
Depth-First Search (DFS)

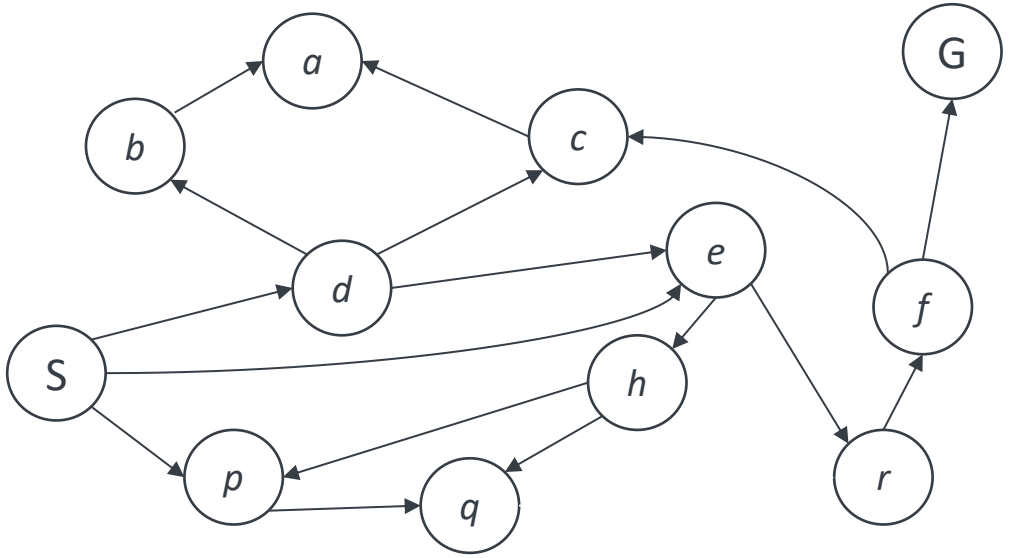




Note that the search ends when the goal node is **generated** (not expanded).

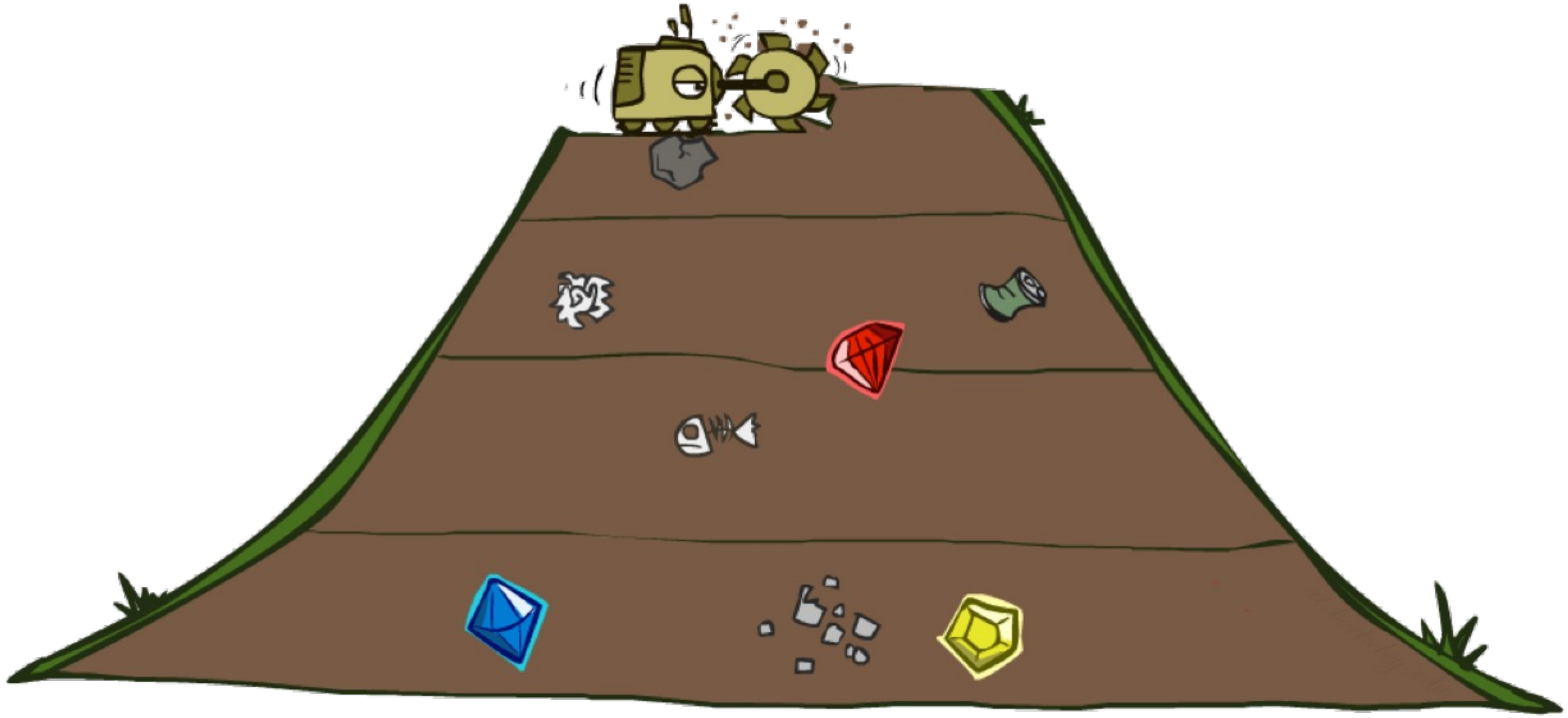
Depth-First Search (DFS)



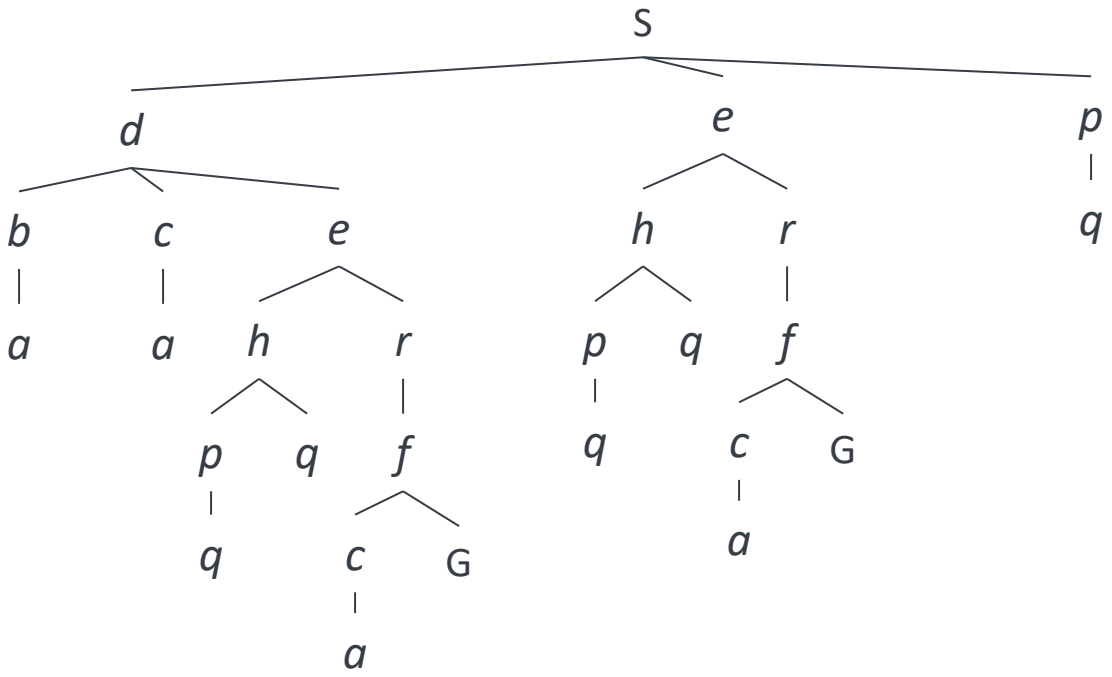


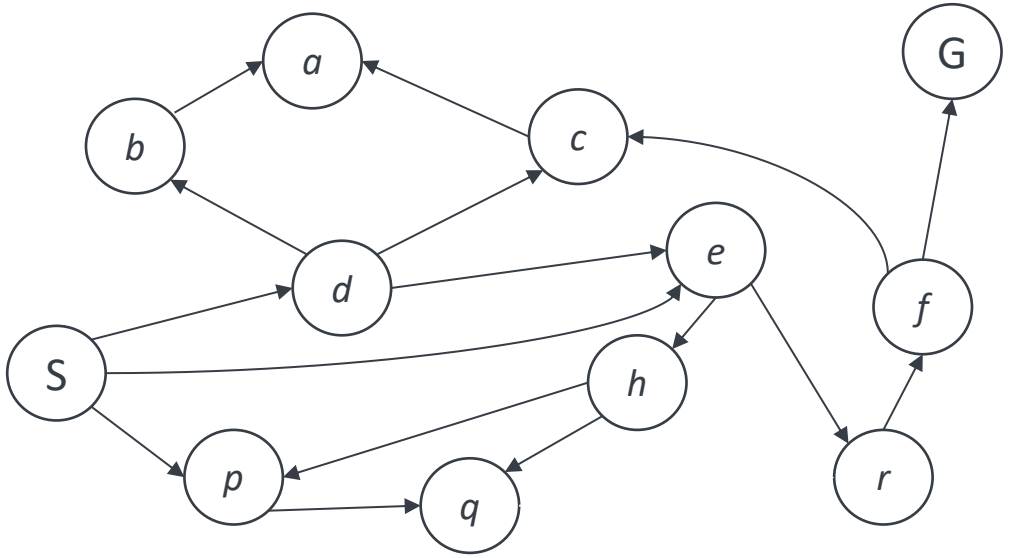
Number of expanded nodes: 13

Breadth-First Search (BFS)

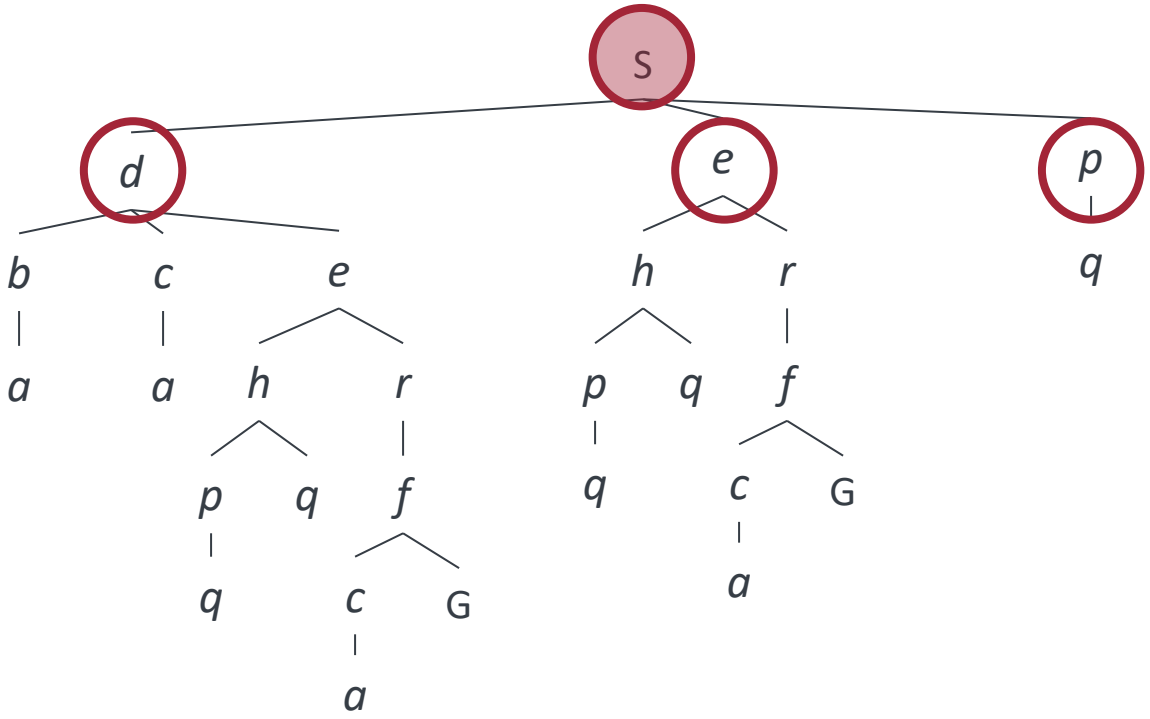


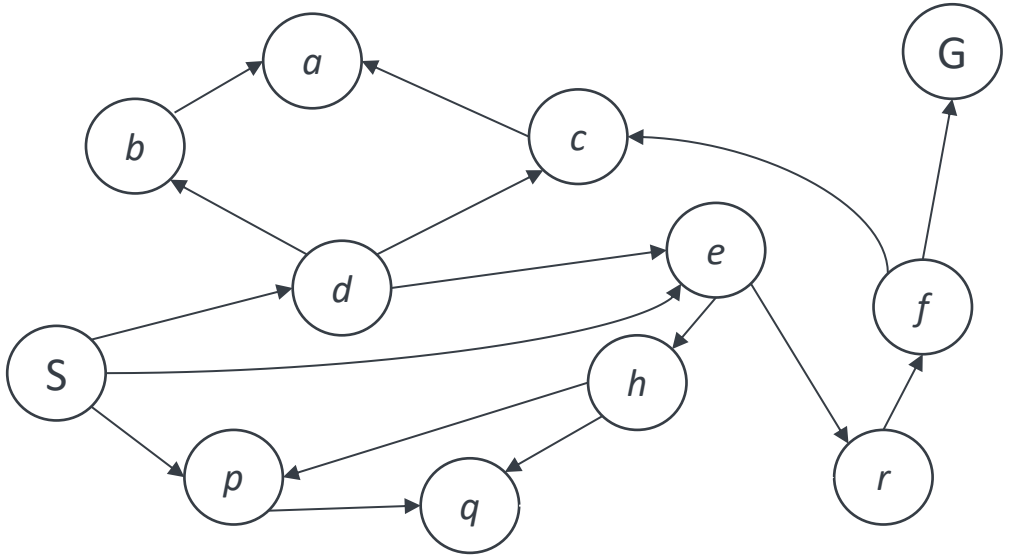
Breadth-First Search (BFS)



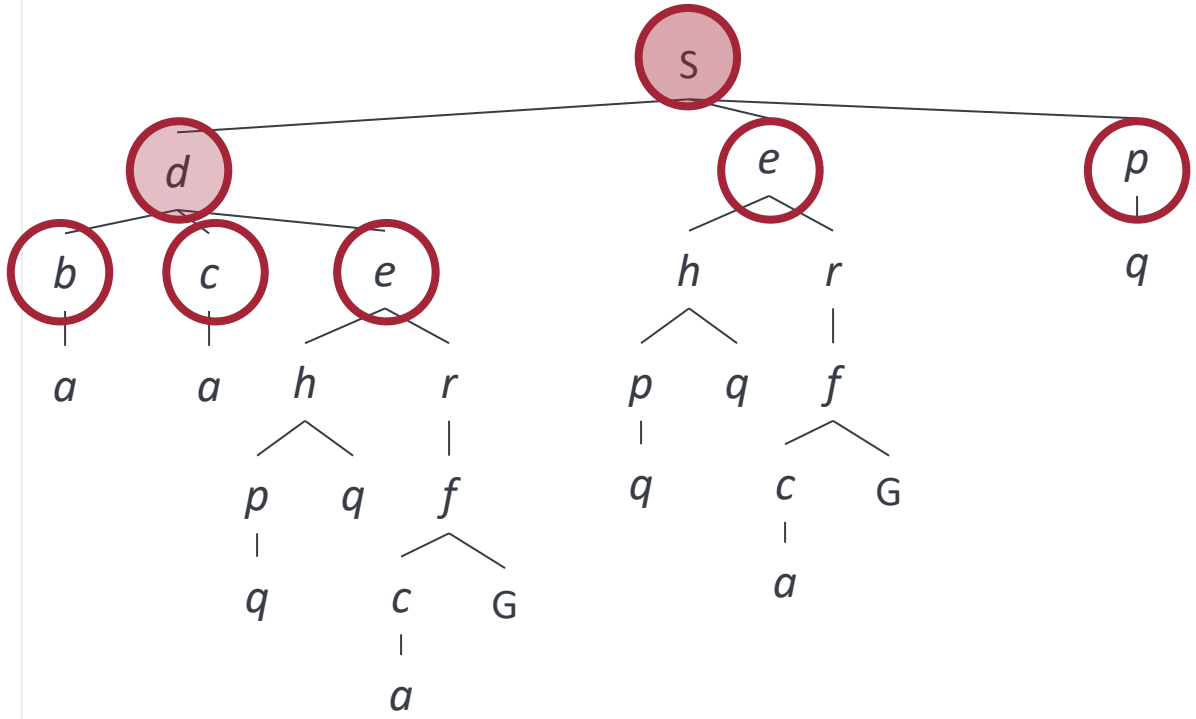


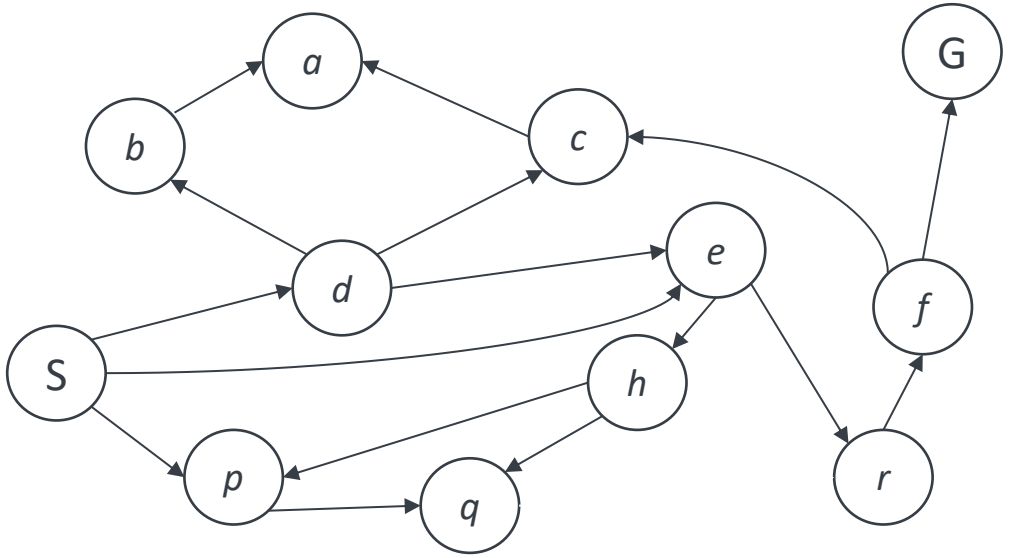
Breadth-First Search (BFS)



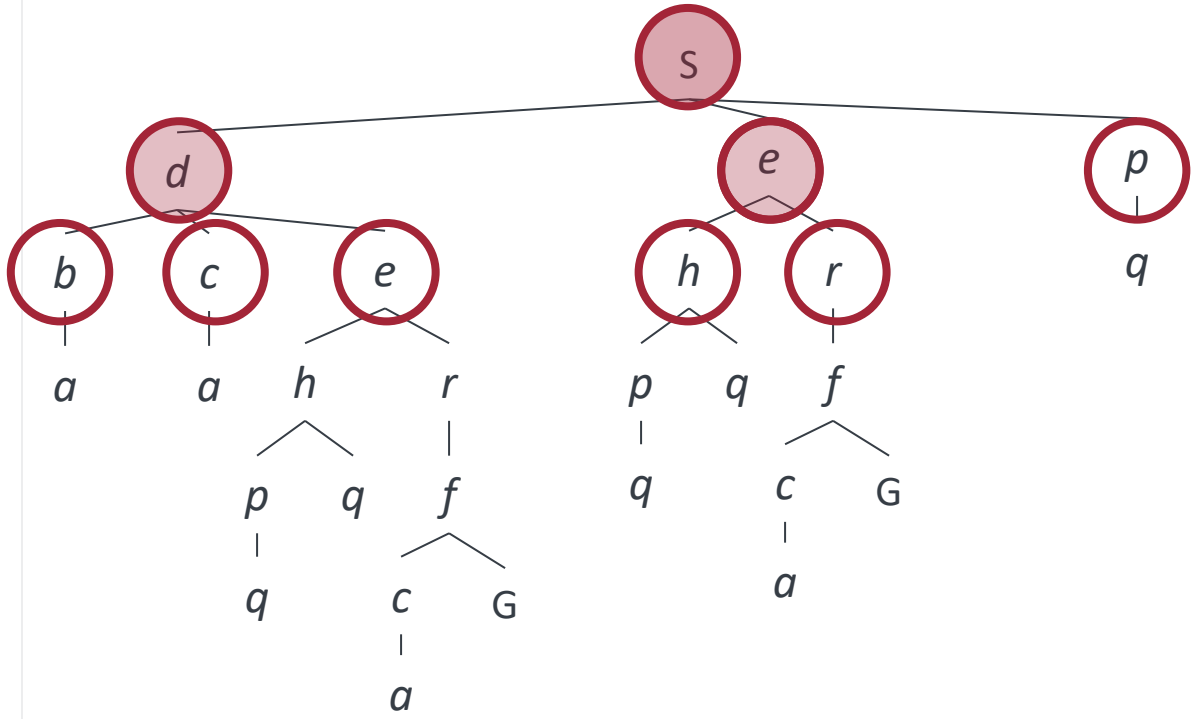


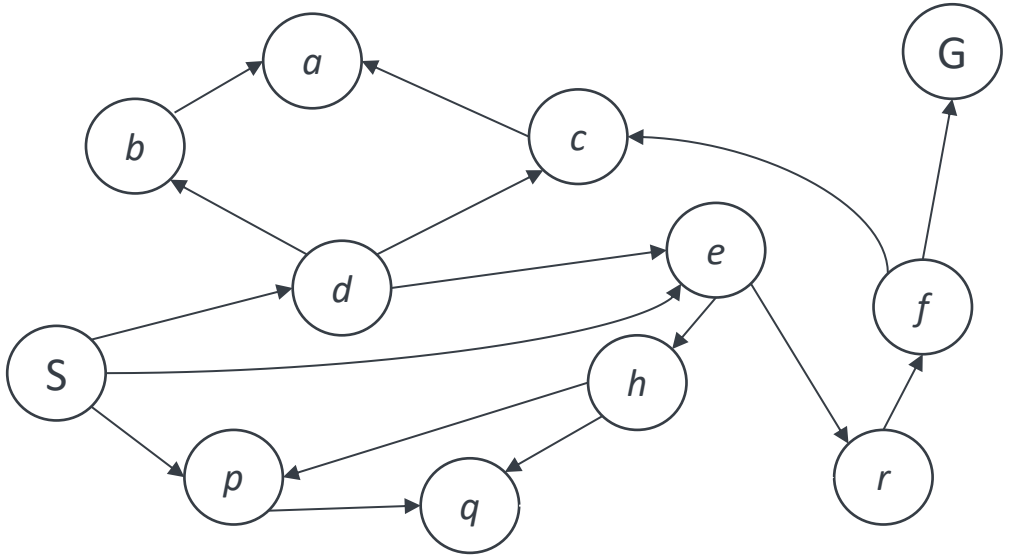
Breadth-First Search (BFS)



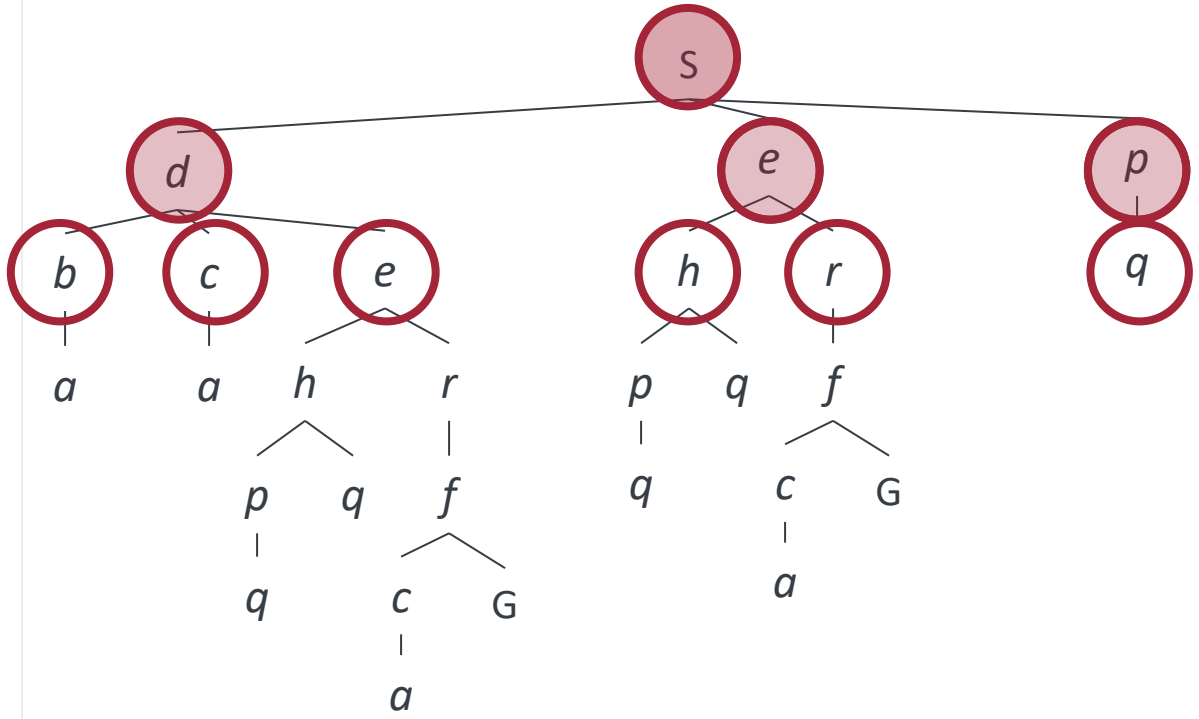


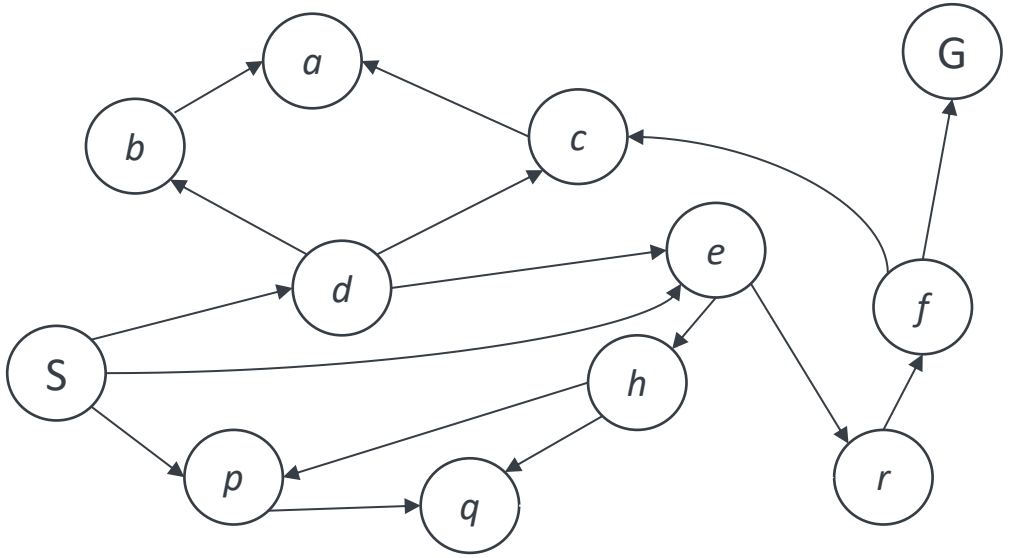
Breadth-First Search (BFS)



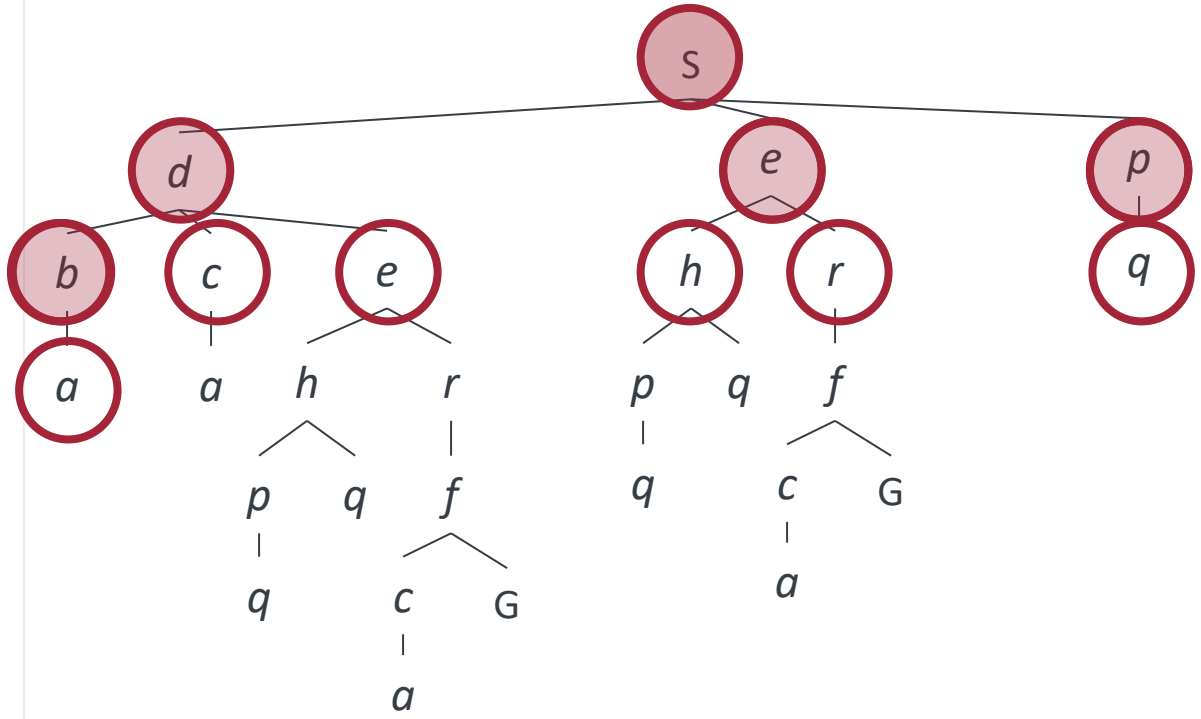


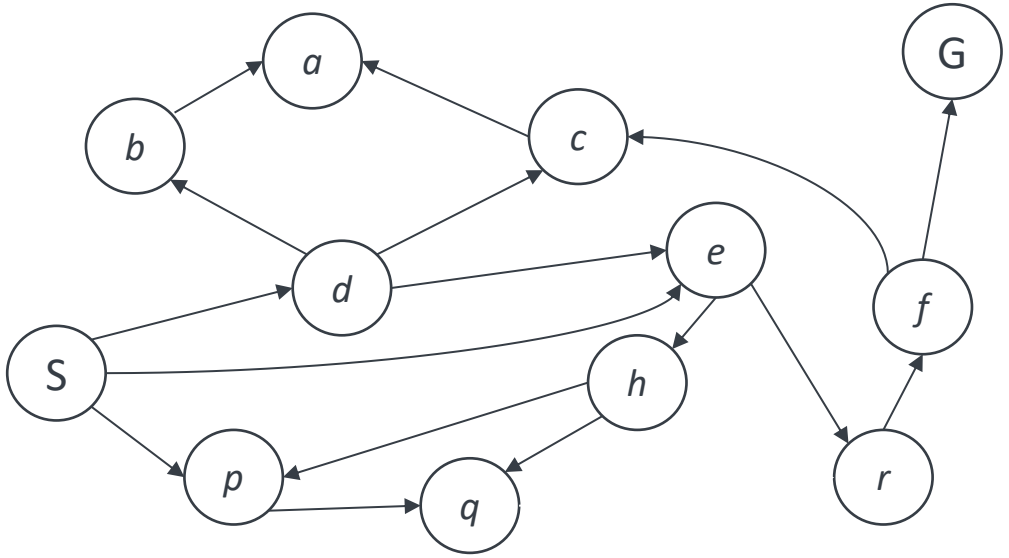
Breadth-First Search (BFS)



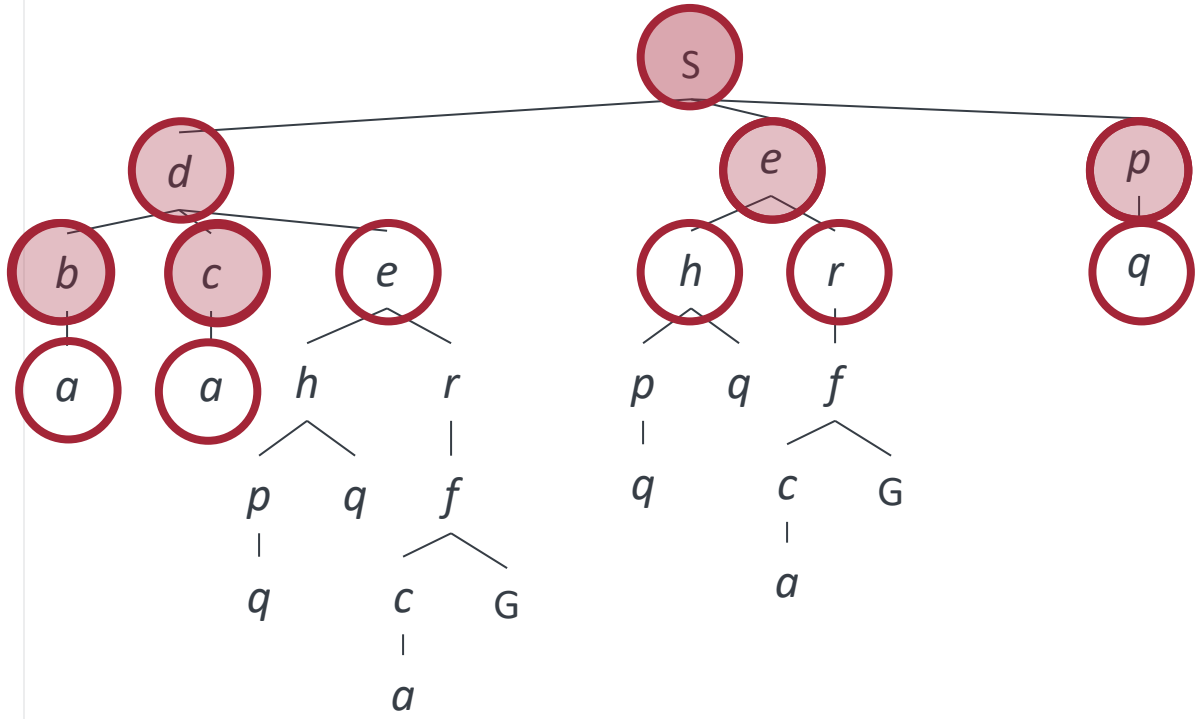


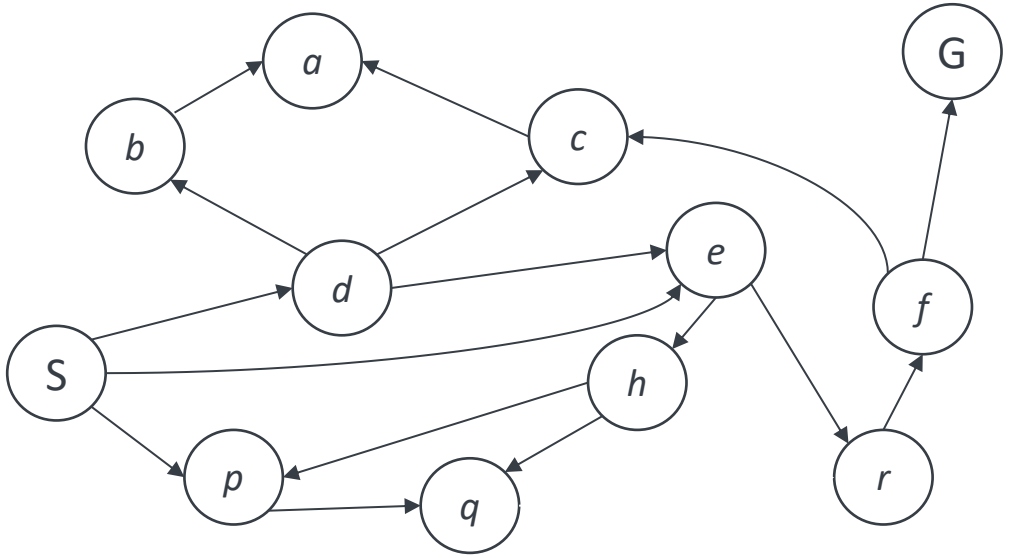
Breadth-First Search (BFS)



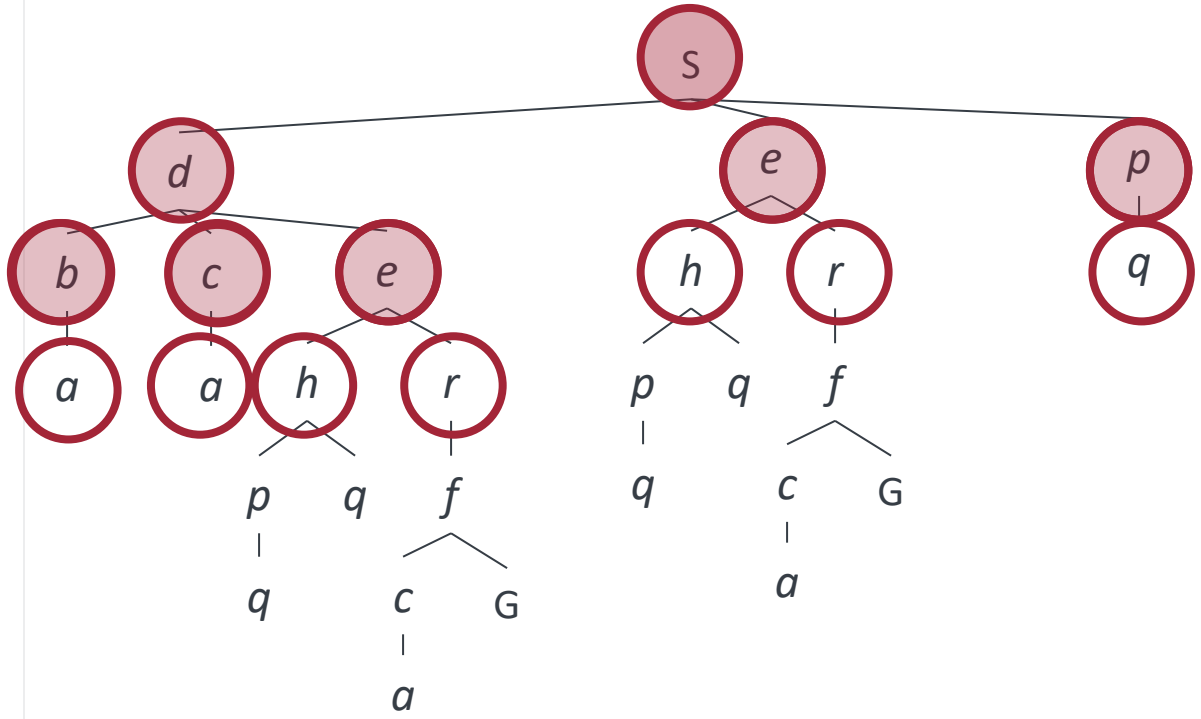


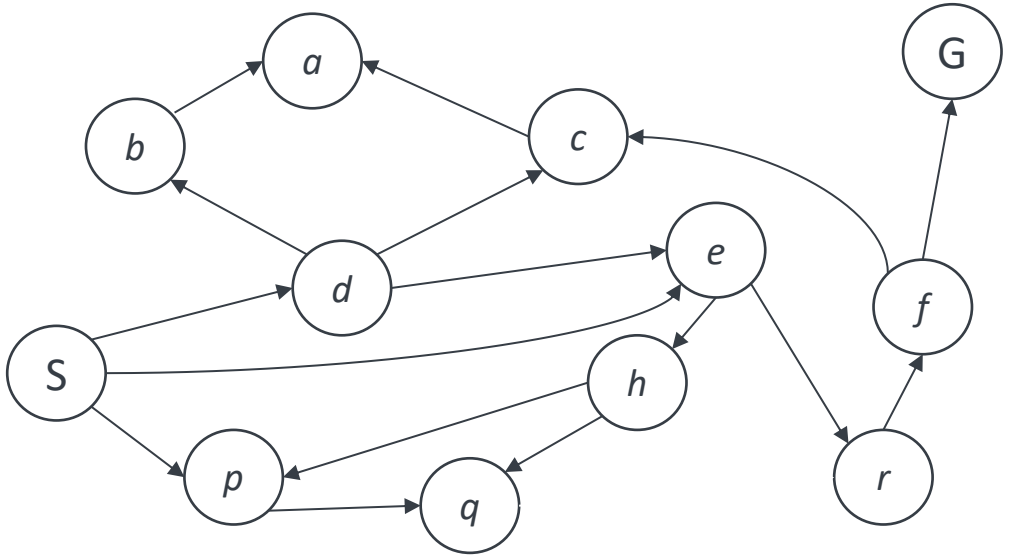
Breadth-First Search (BFS)



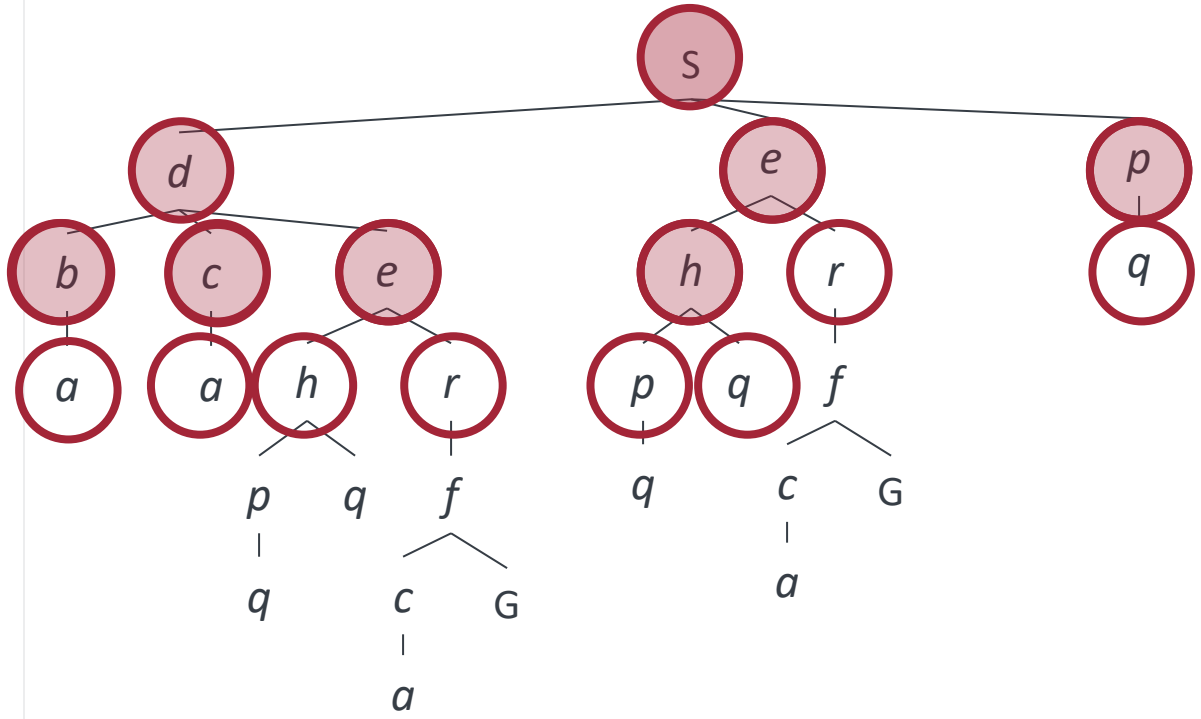


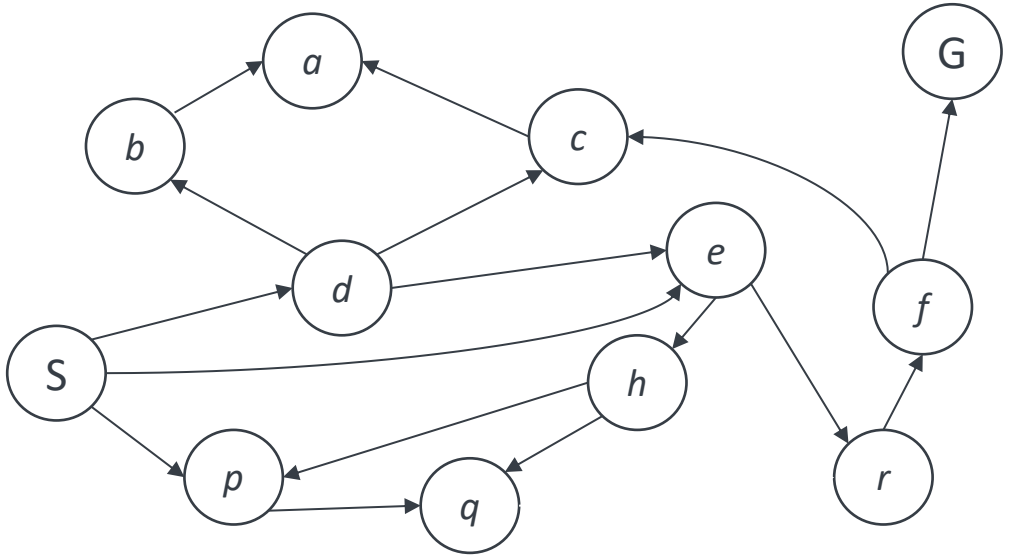
Breadth-First Search (BFS)



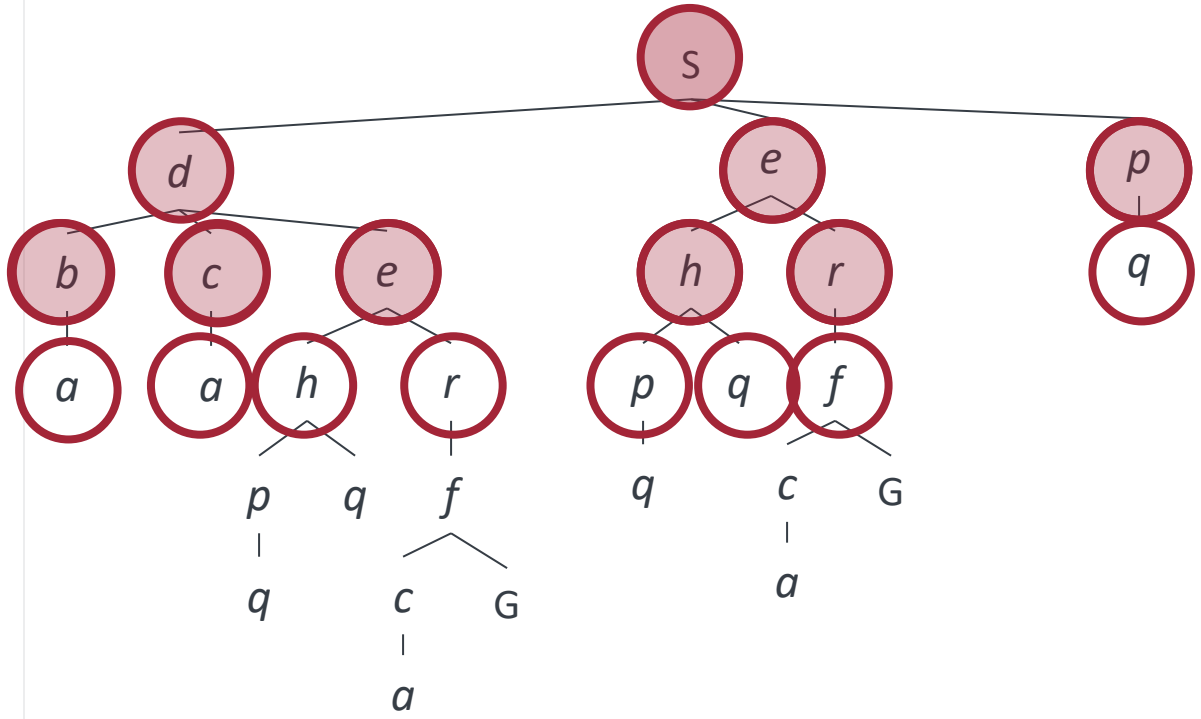


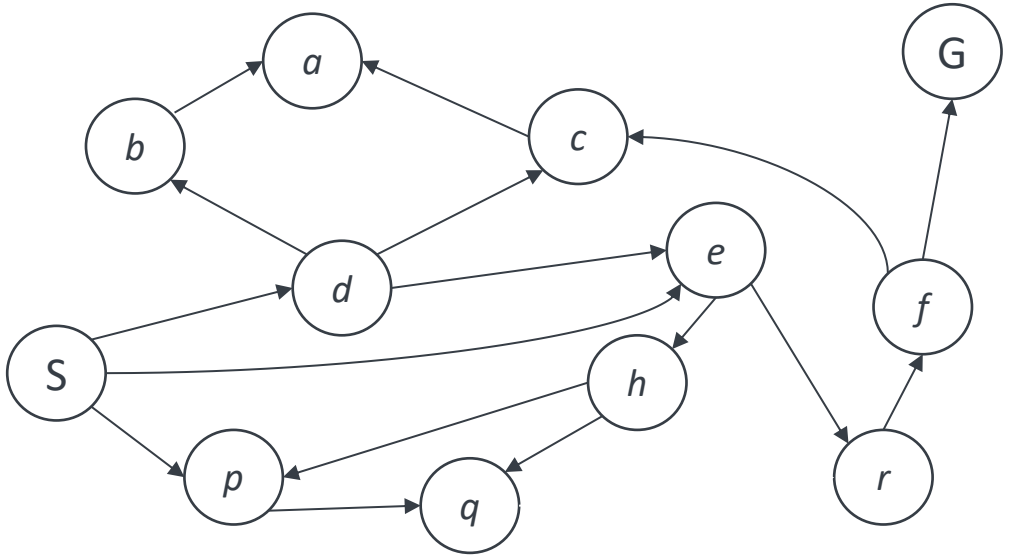
Breadth-First Search (BFS)



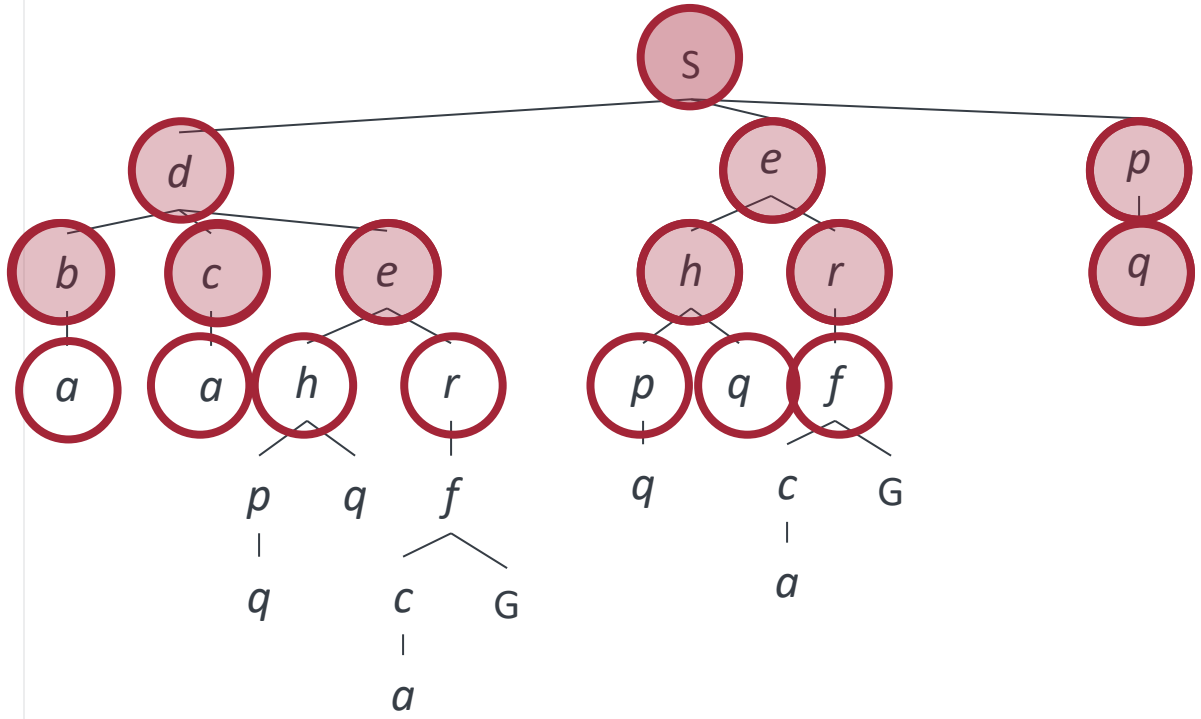


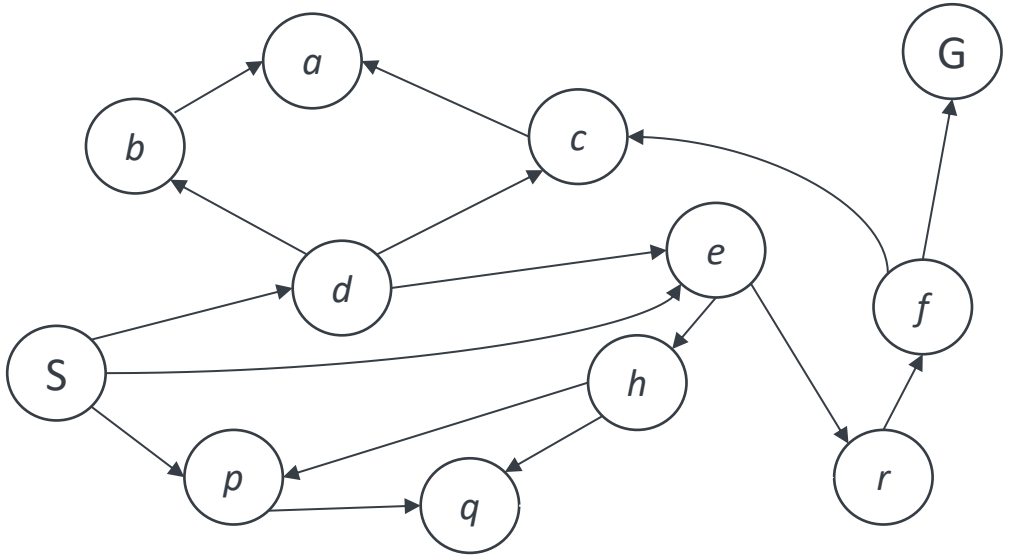
Breadth-First Search (BFS)



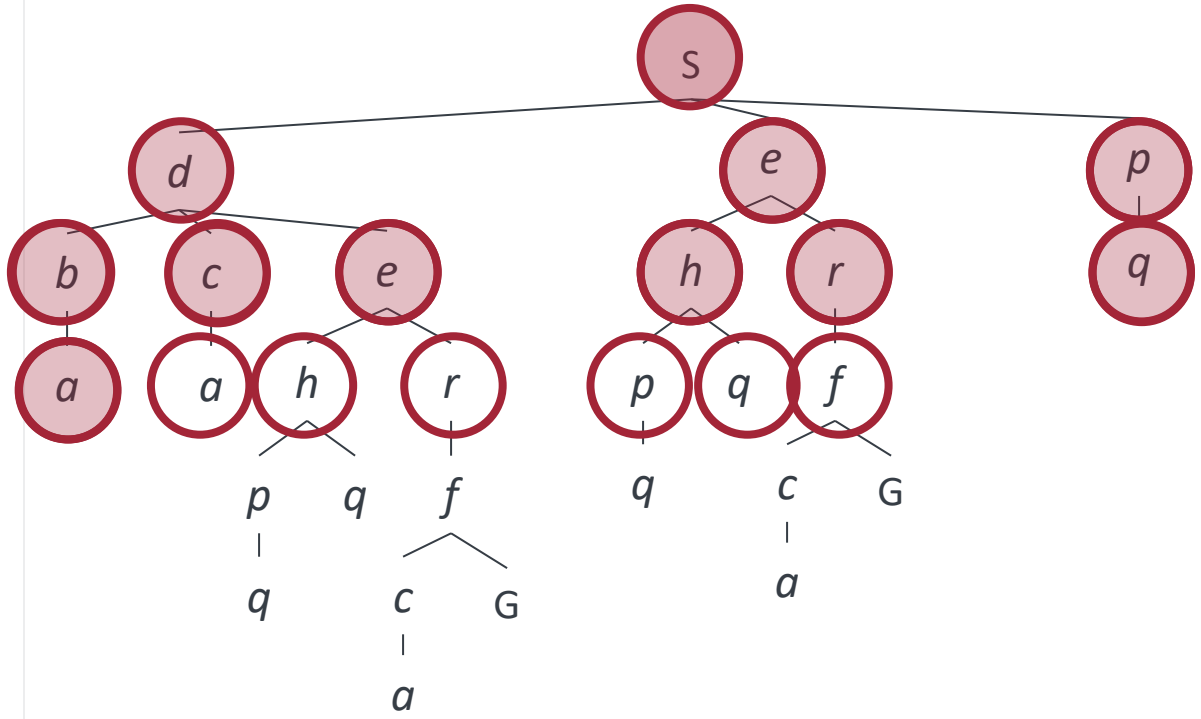


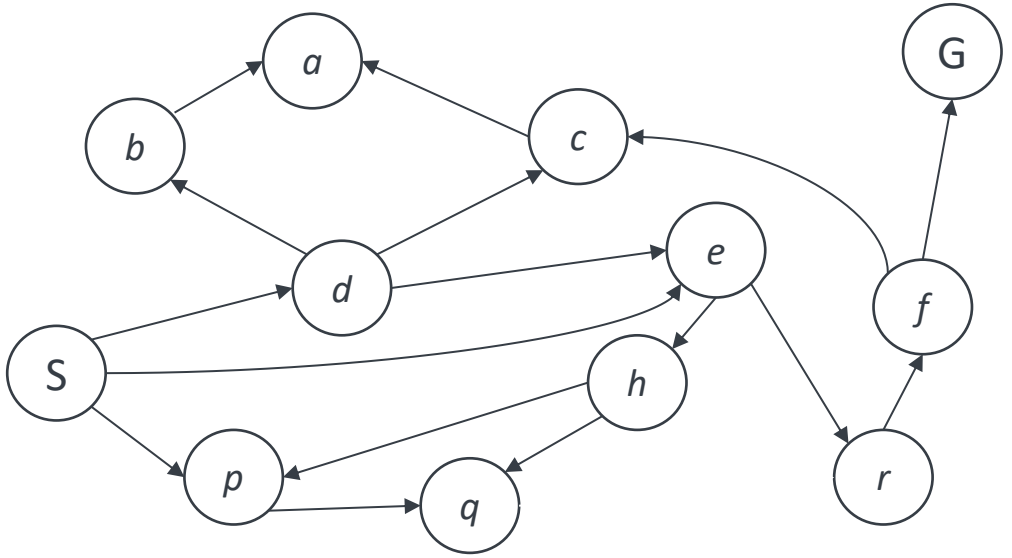
Breadth-First Search (BFS)



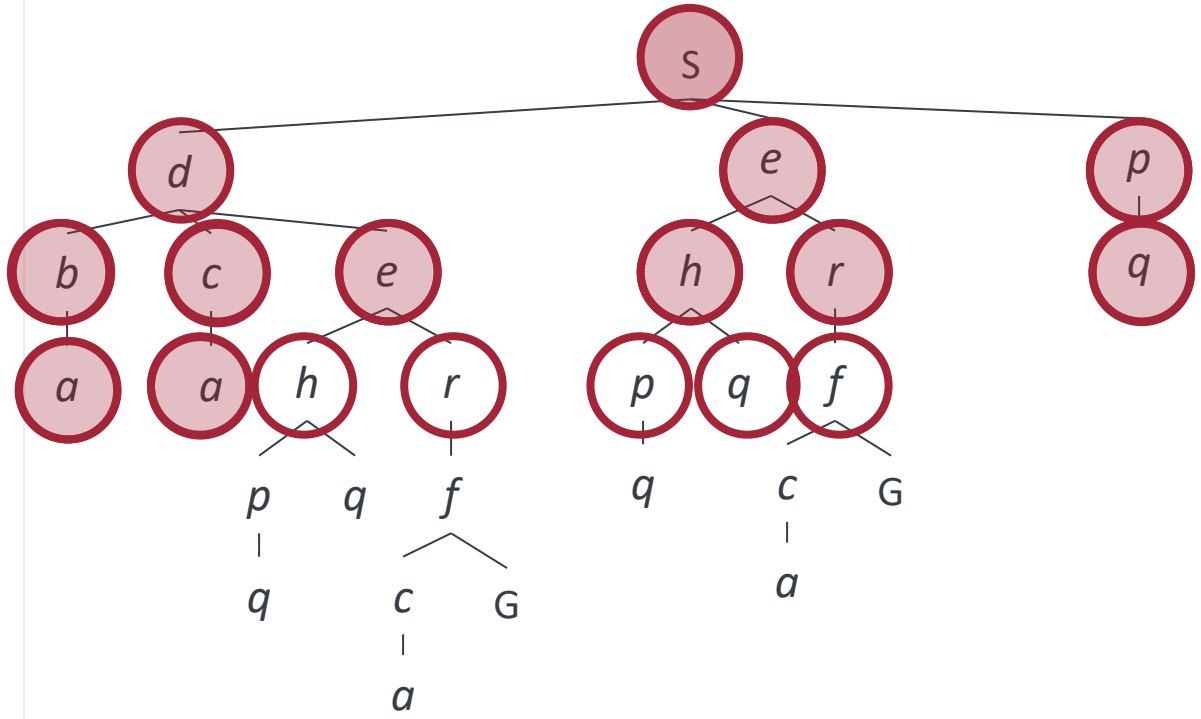


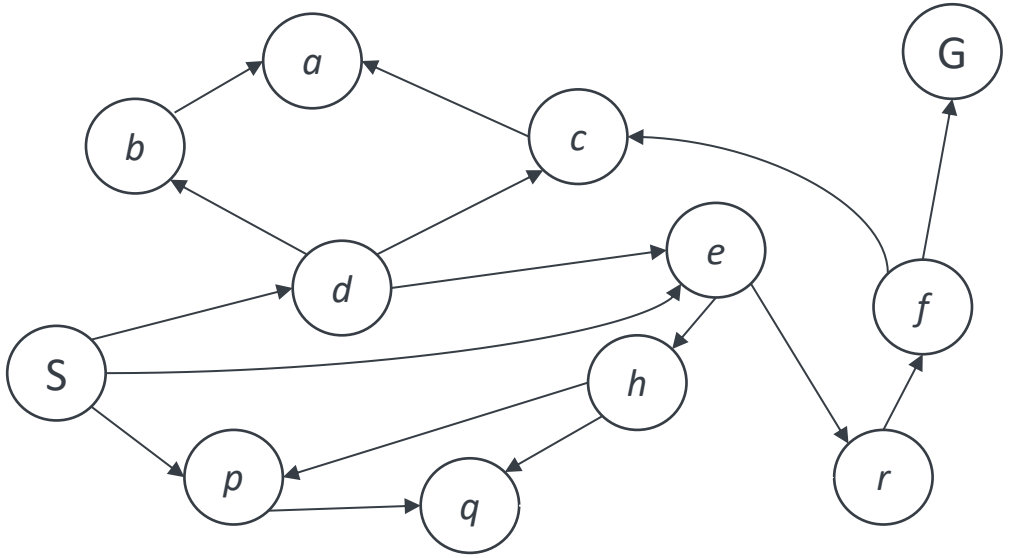
Breadth-First Search (BFS)



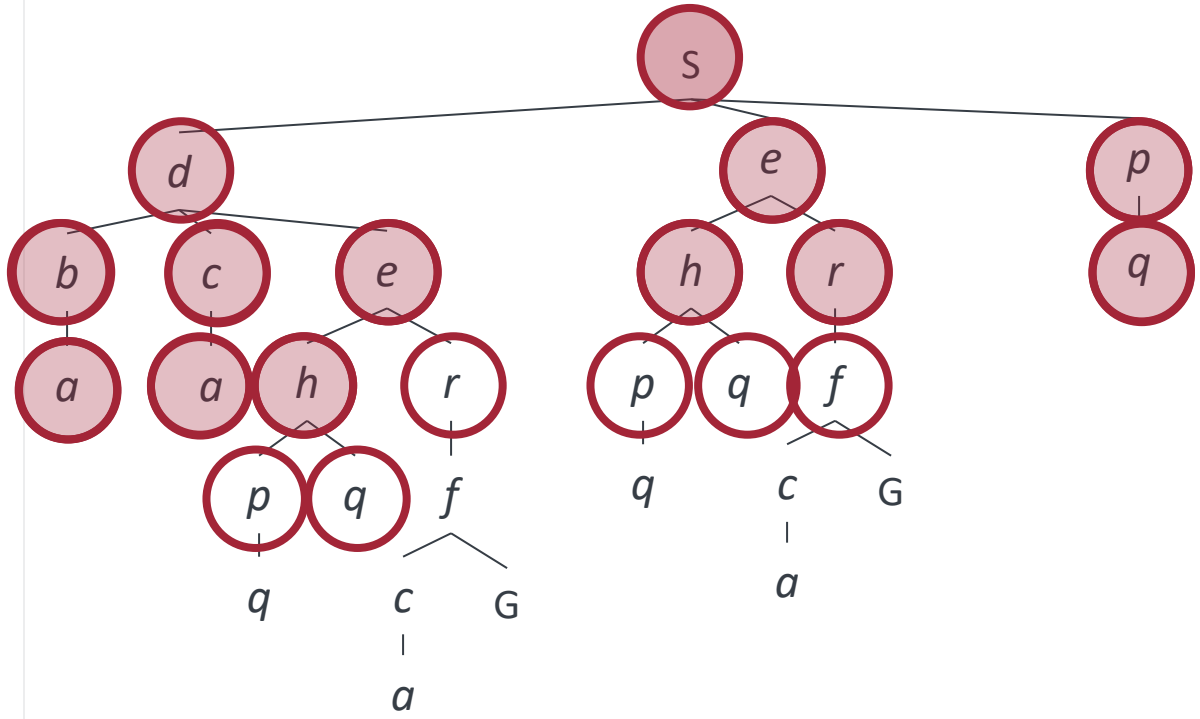


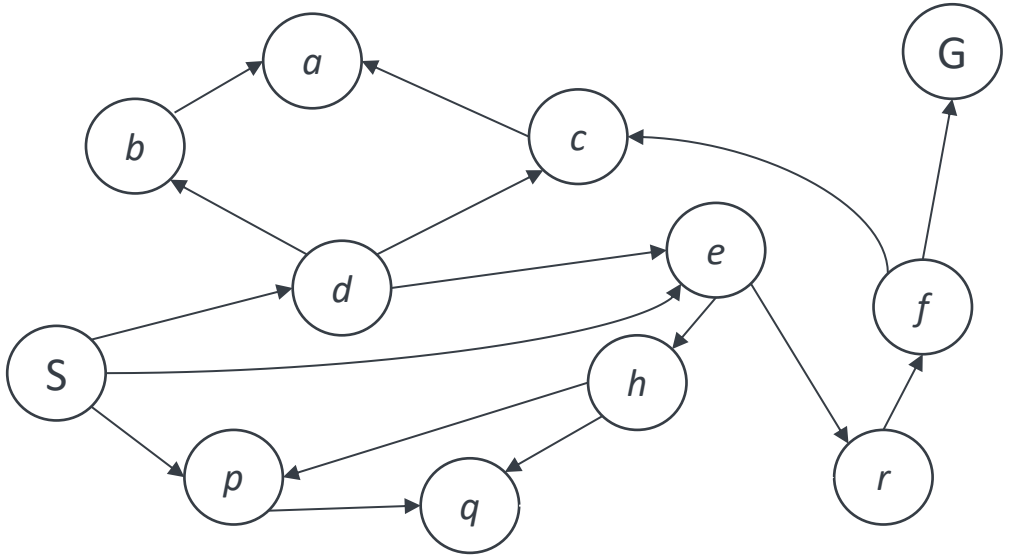
Breadth-First Search (BFS)



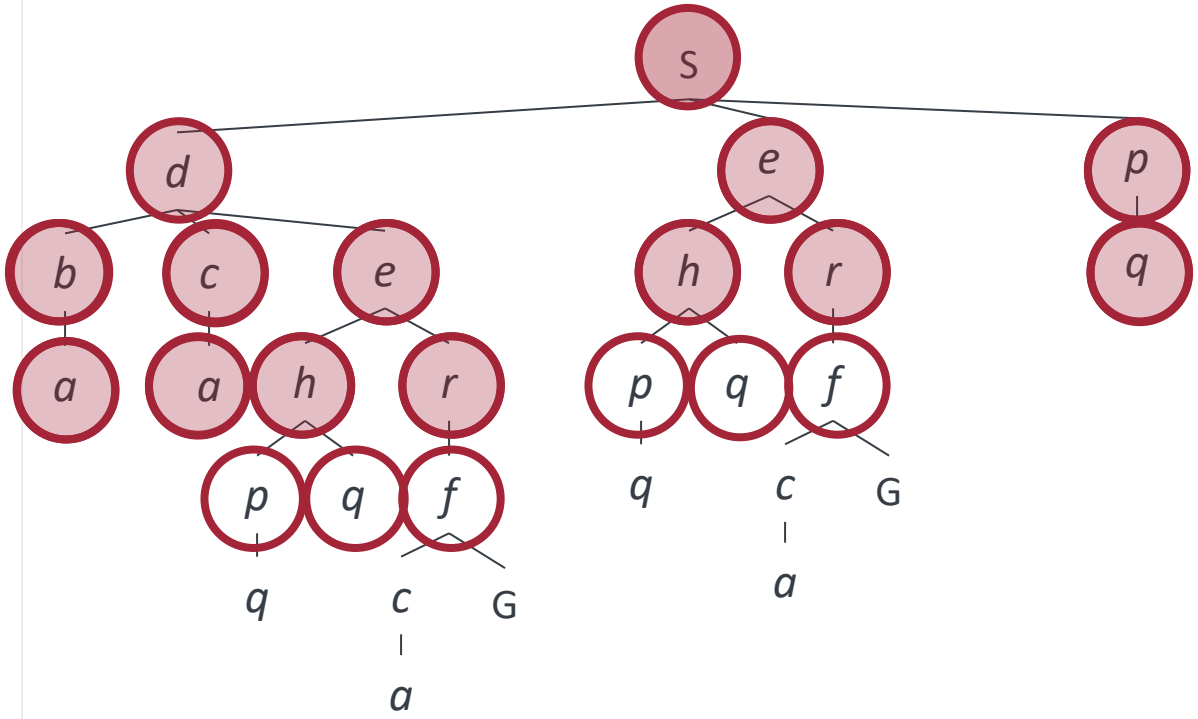


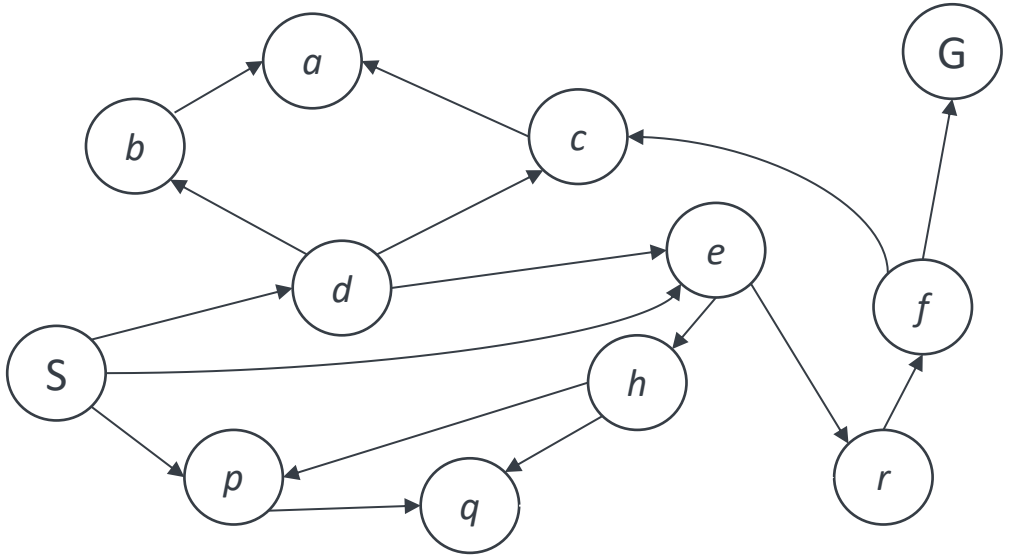
Breadth-First Search (BFS)



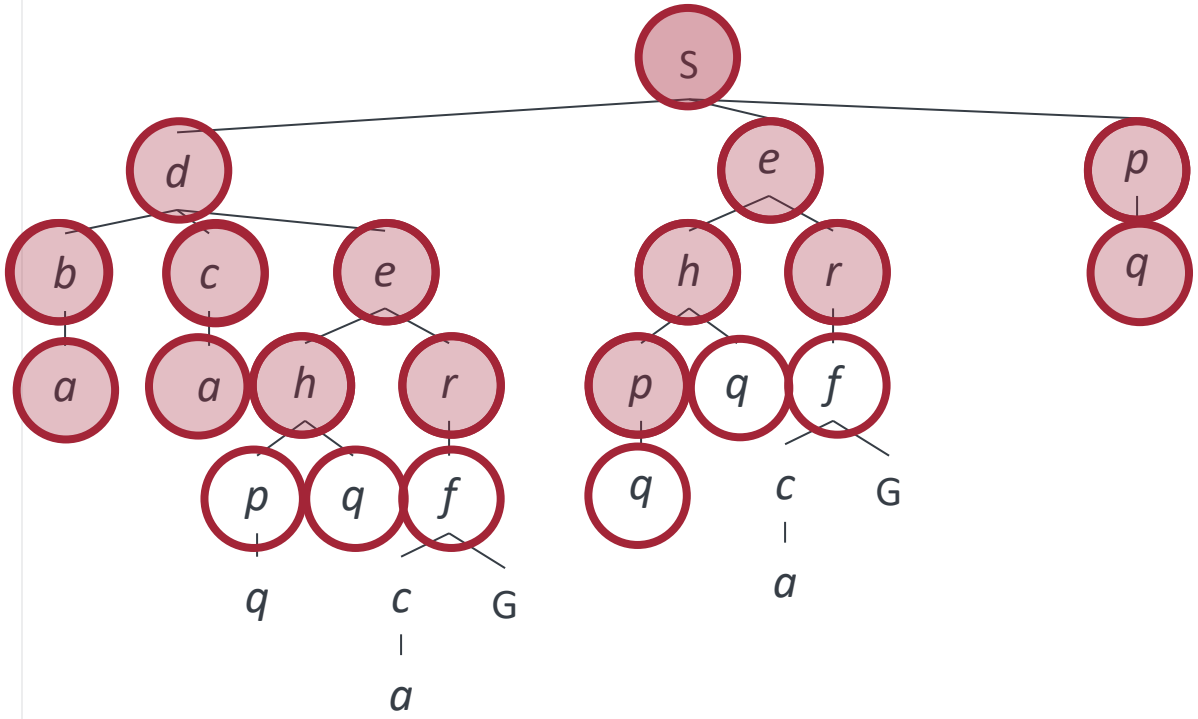


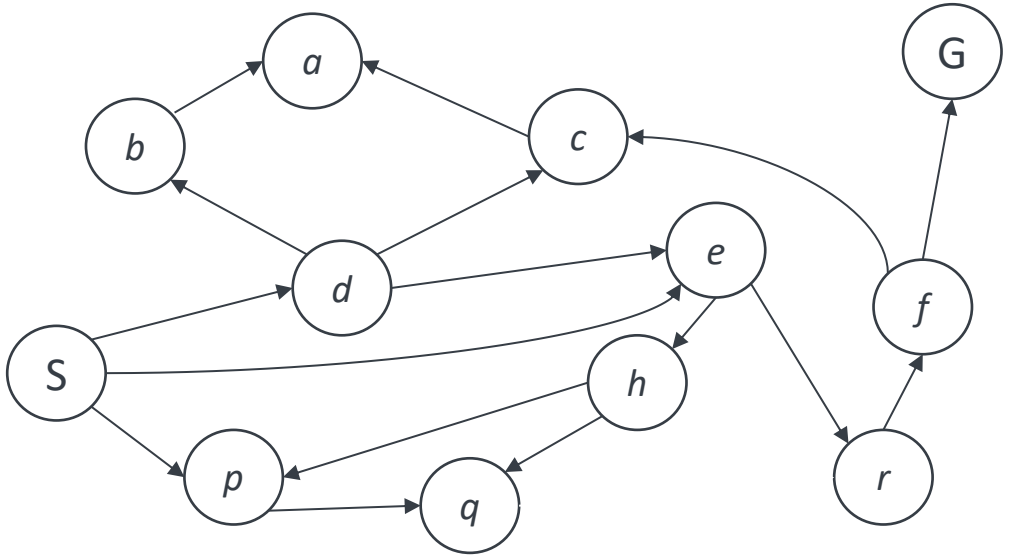
Breadth-First Search (BFS)



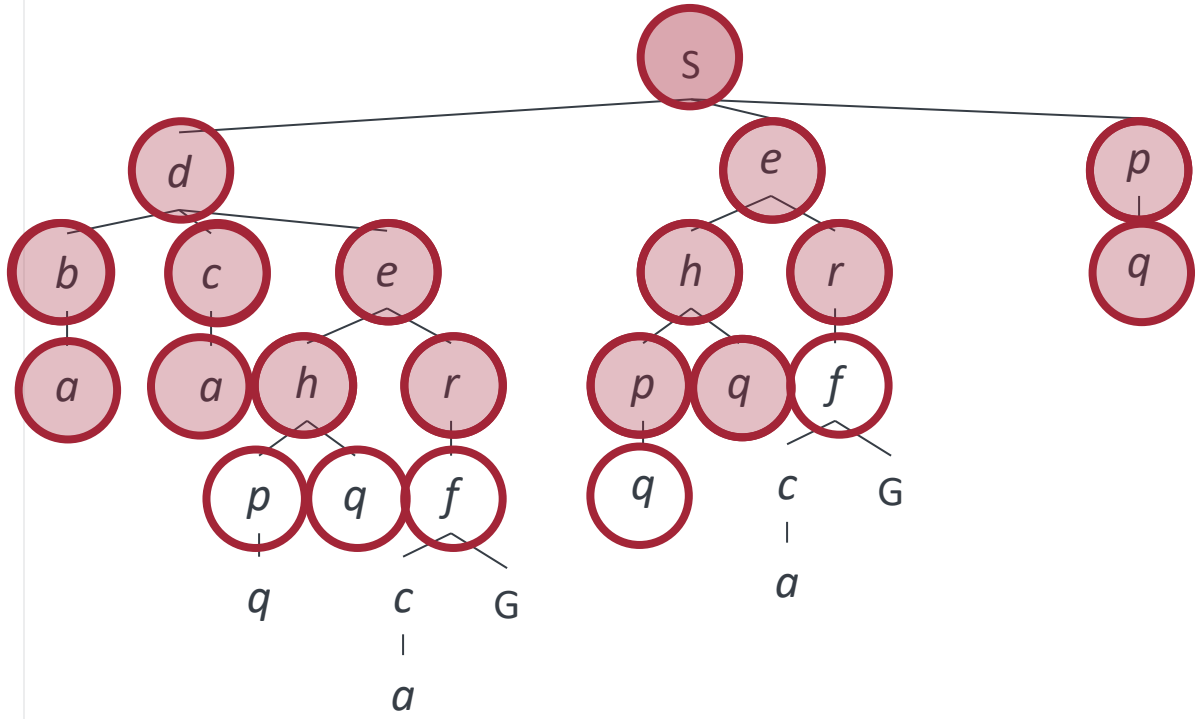


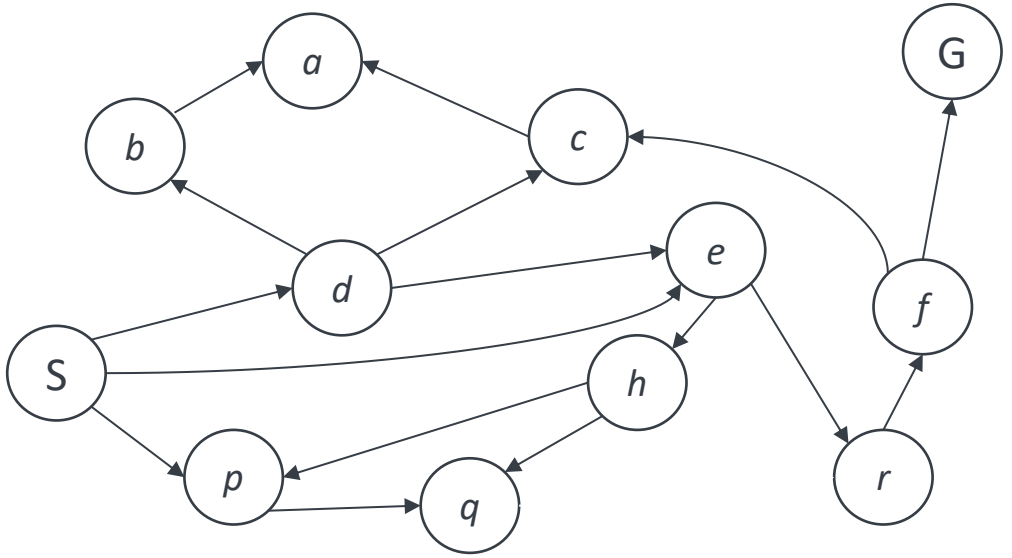
Breadth-First Search (BFS)



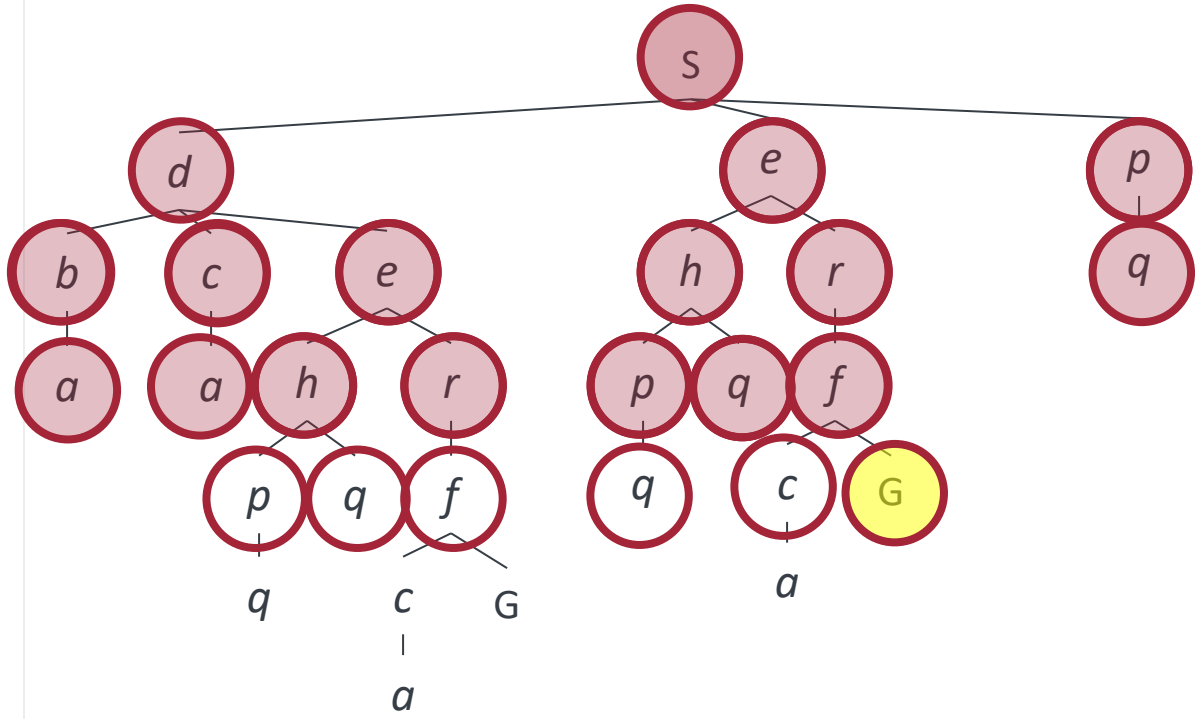


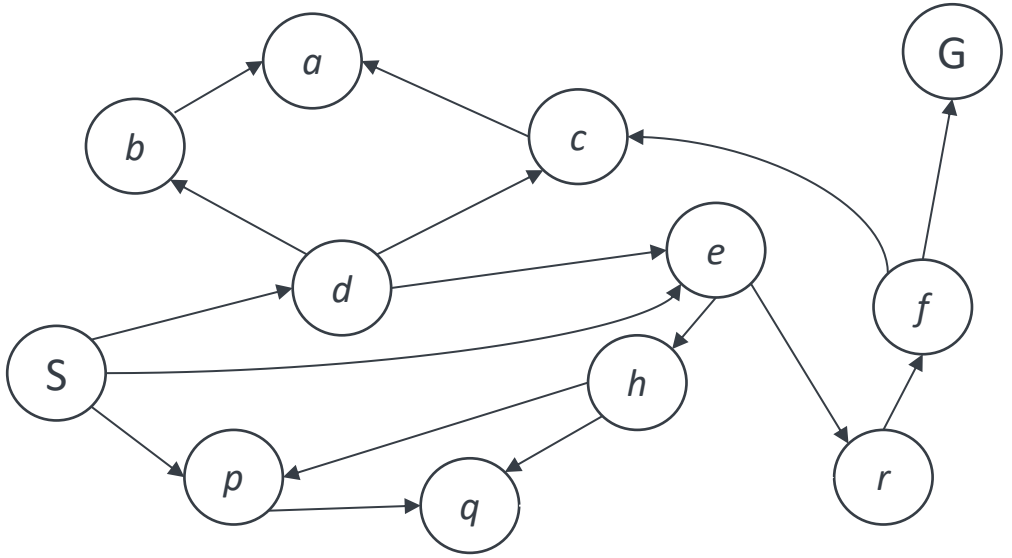
Breadth-First Search (BFS)





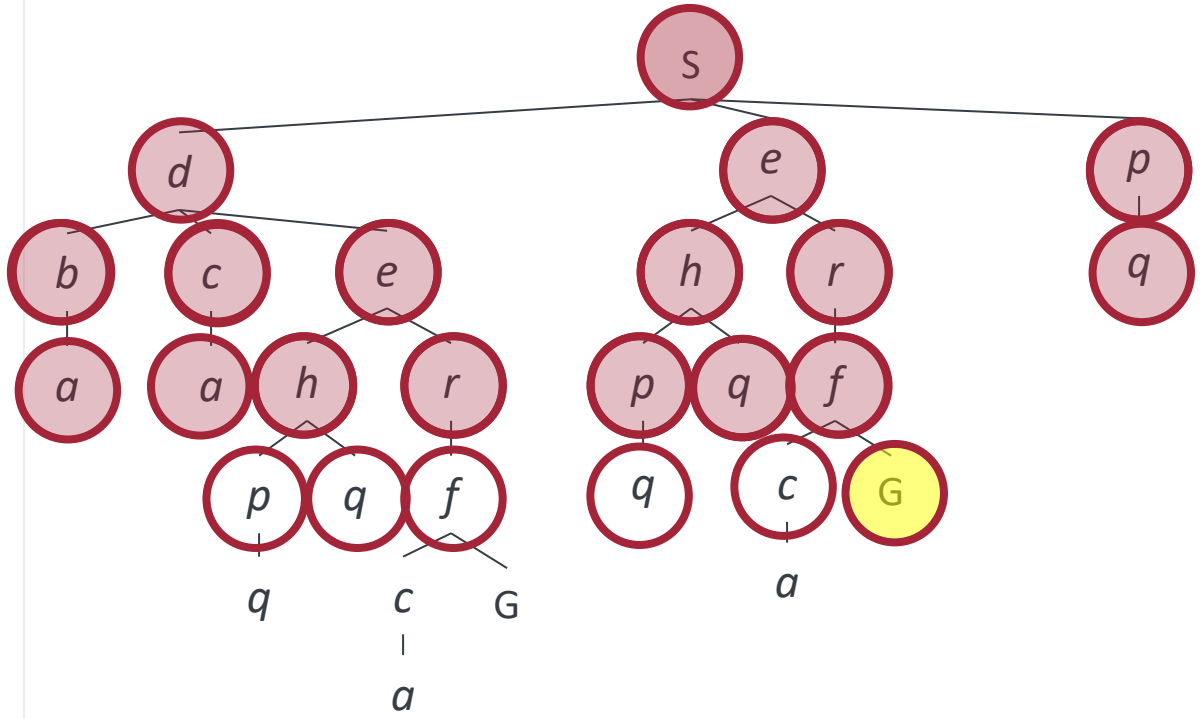
Breadth-First Search (BFS)

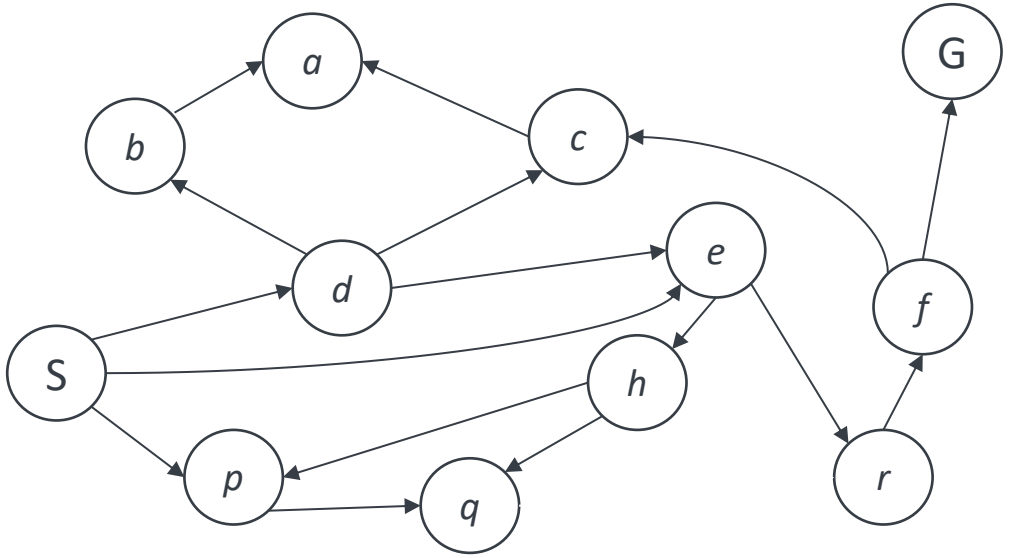




Note that the search ends when the goal node is **generated** (not expanded).

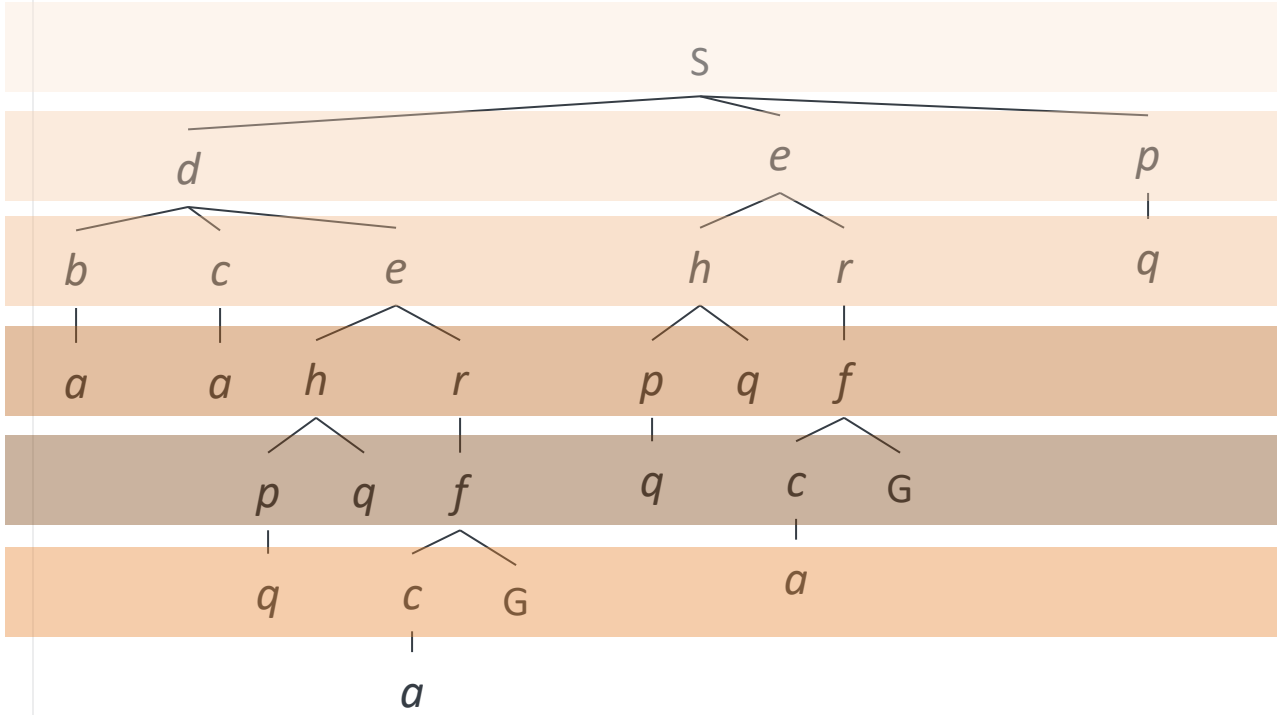
Breadth-First Search (BFS)

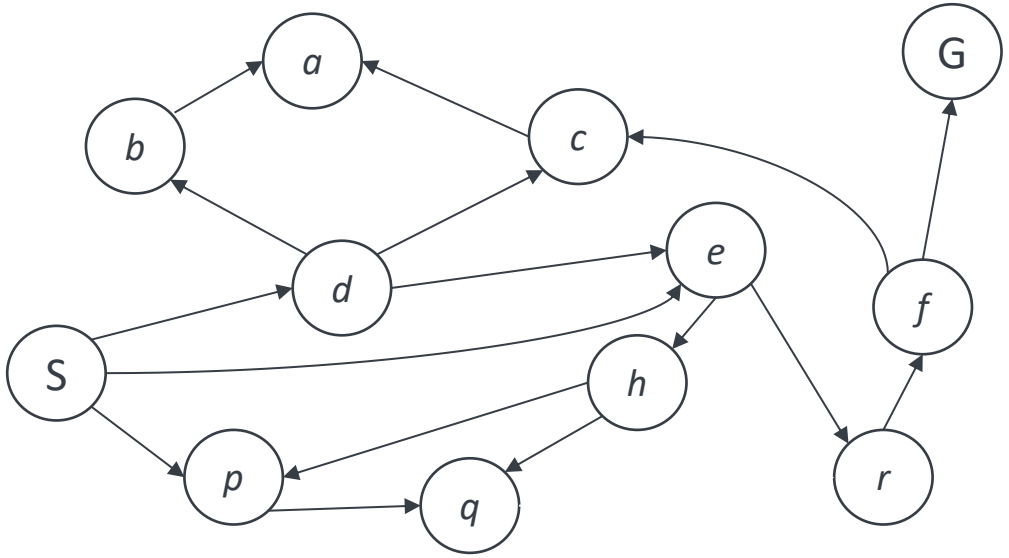




Number of expanded nodes: 17

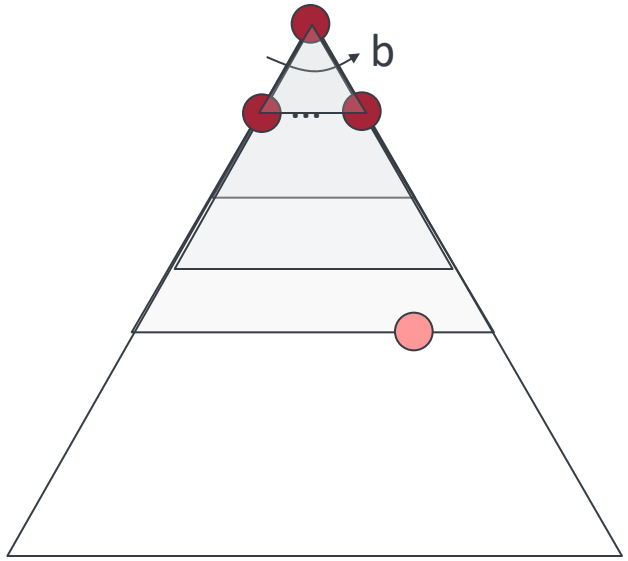
Breadth-First Search (BFS)





Iterative Deepening Search (IDS)

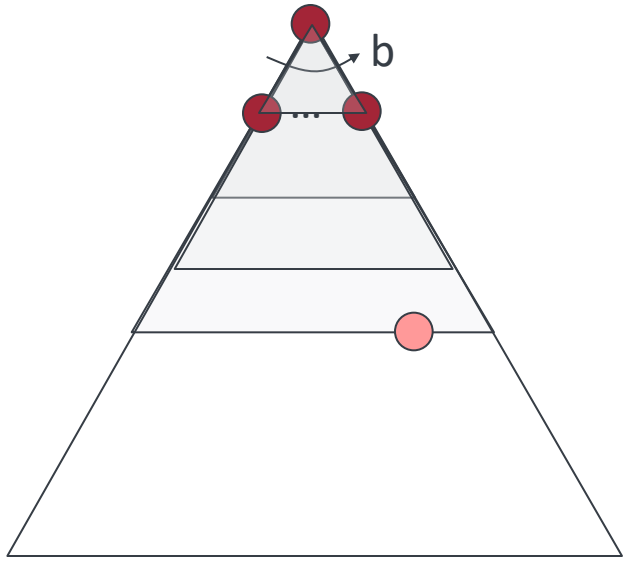
- Idea: get DFS's space advantage with BFS's time / shallow-solution advantages
  - Run a DFS with depth limit 1. If no solution...
  - Run a DFS with depth limit 2. If no solution...
  - Run a DFS with depth limit 3. .....



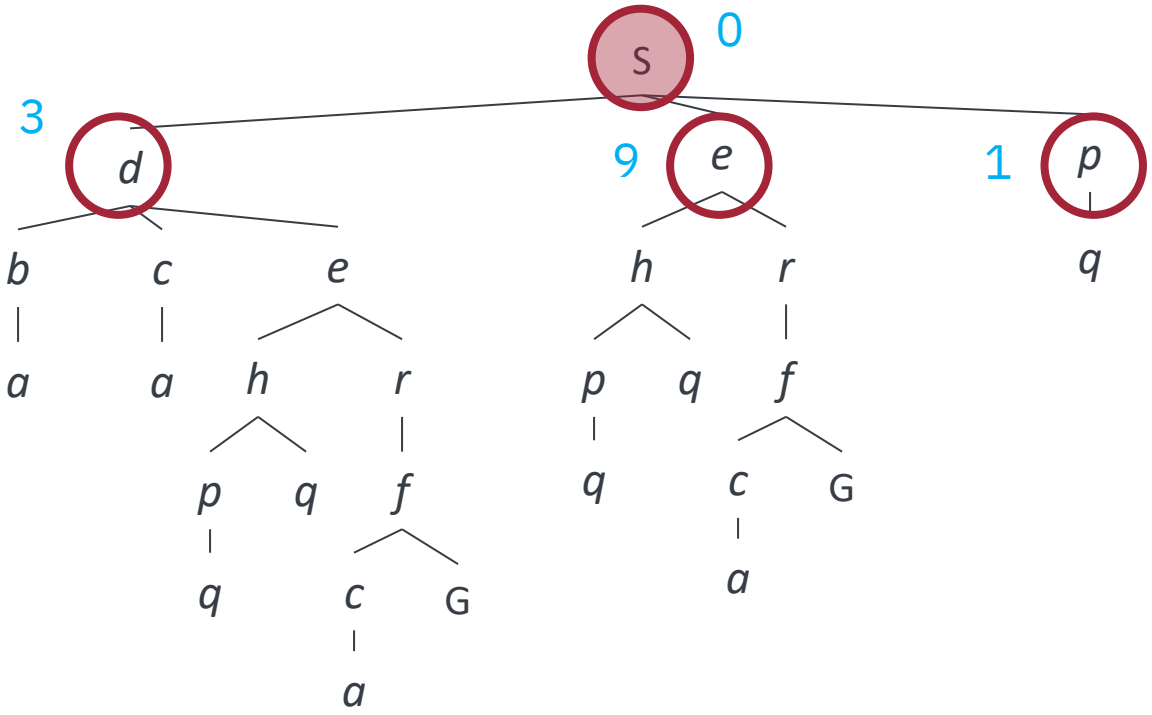
- Isn't that wastefully redundant?
  - Generally most work happens in the lowest level searched, so not so bad!

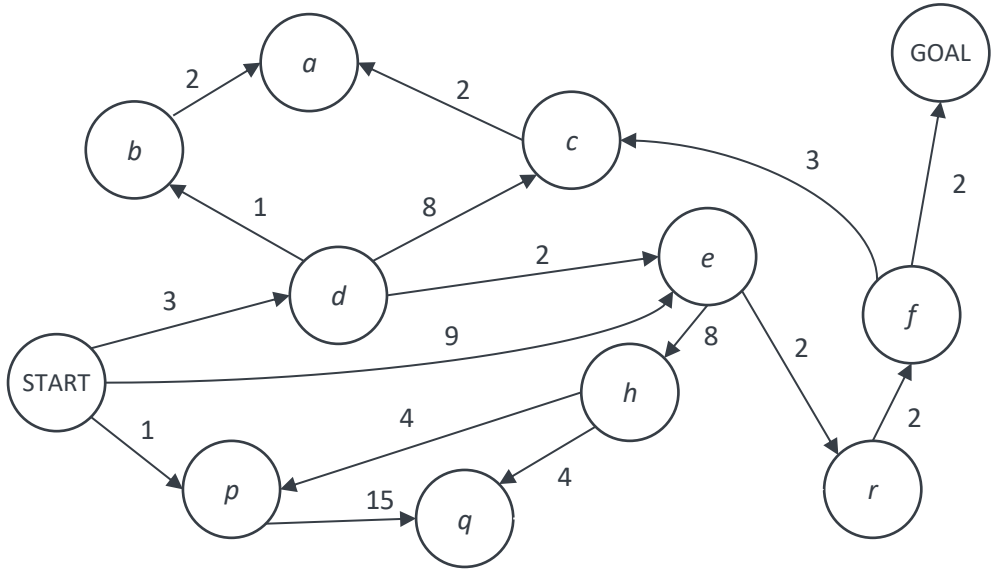
IDS properties

- What nodes IDS expand?
  - Some left prefix of the tree.
  - $mb^0 + (m-1)b^1 + (m-2)b^2 + \ldots + b^{m-1} = O(b^m)$
- How much space does the leaf nodes take?
  - Only has siblings on path to root, so O(bm)
- Is it complete?
  - m could be infinite, so only if we prevent cycles (more later)
- Is it optimal?
  - Only if costs are all 1 (more on costs later)

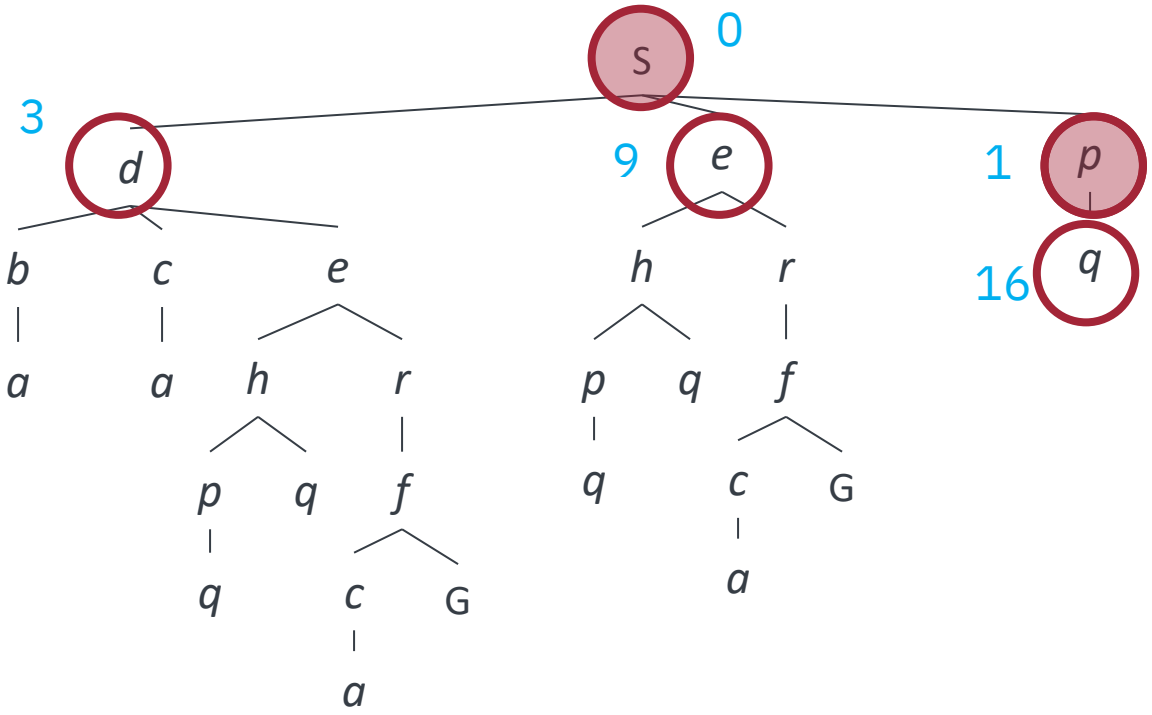


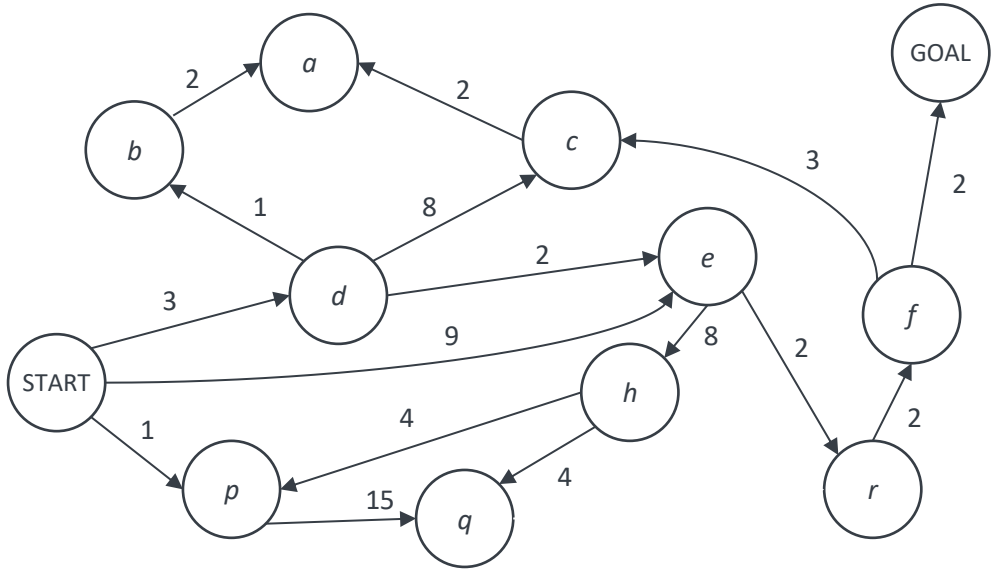
Uniform Cost Search (UCS)



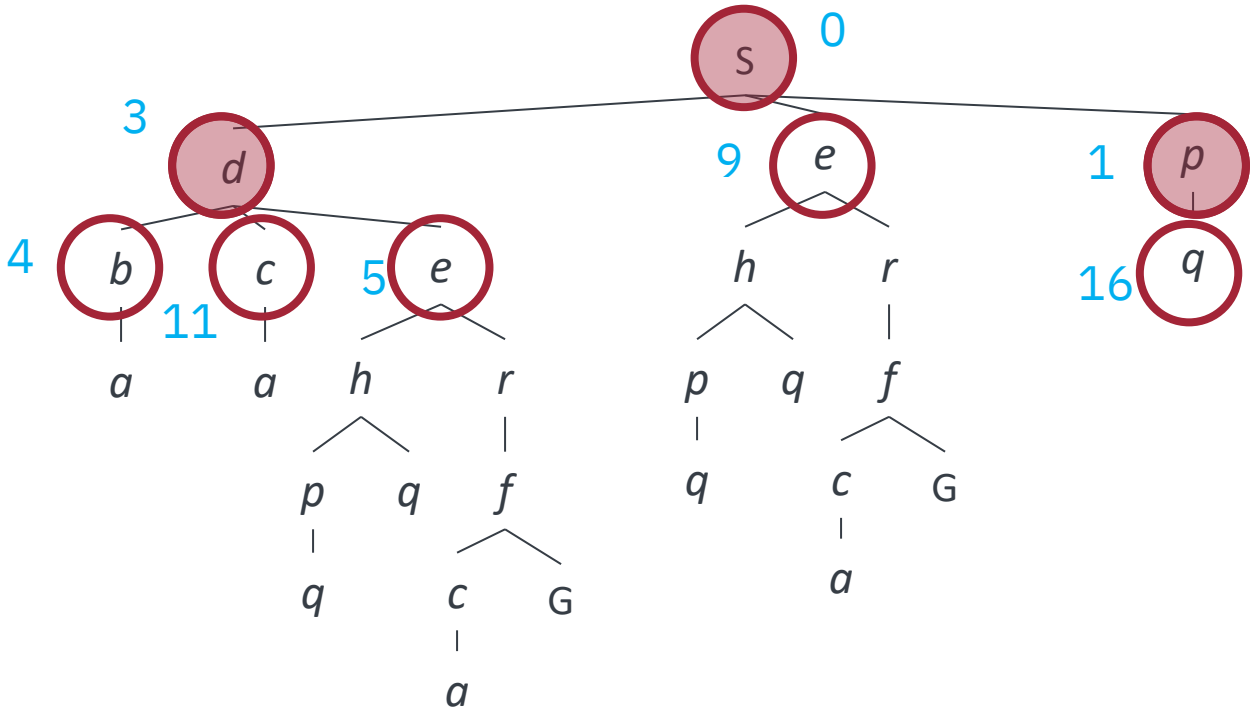


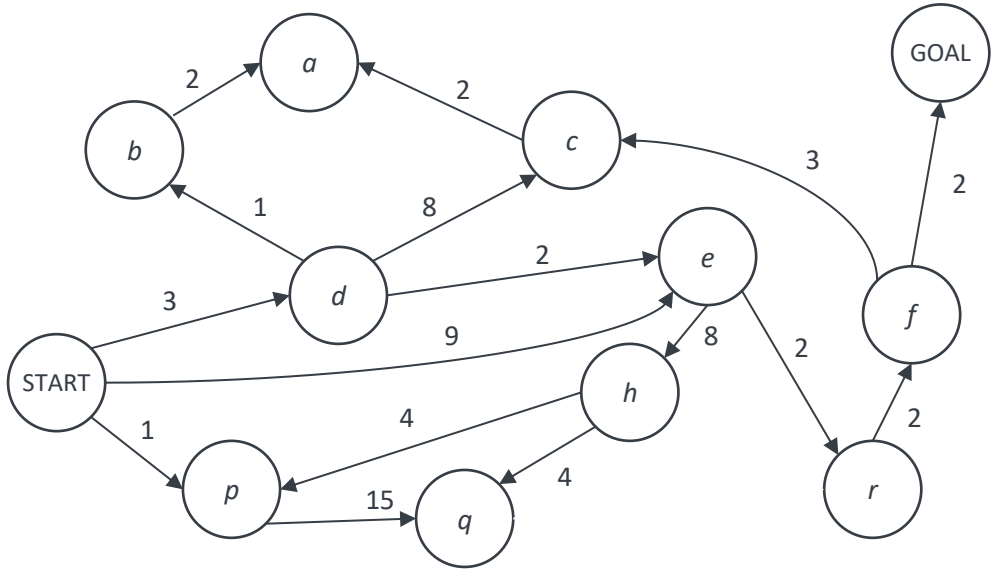
Uniform Cost Search (UCS)



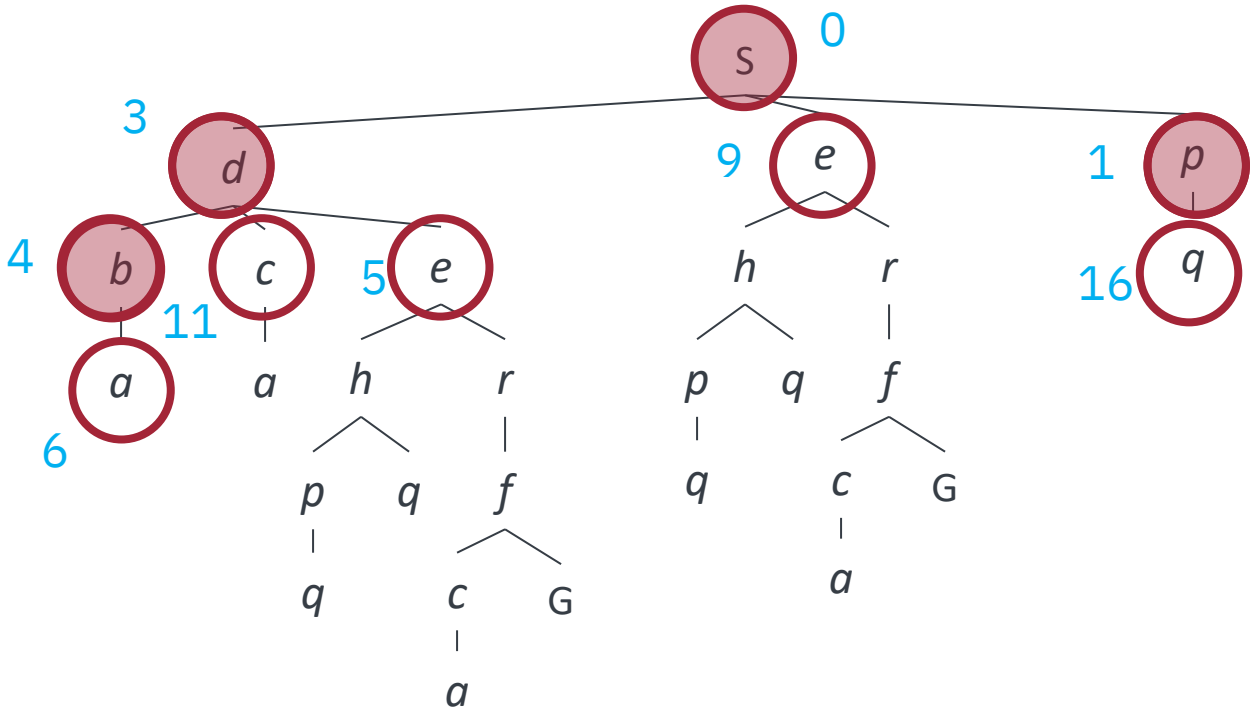


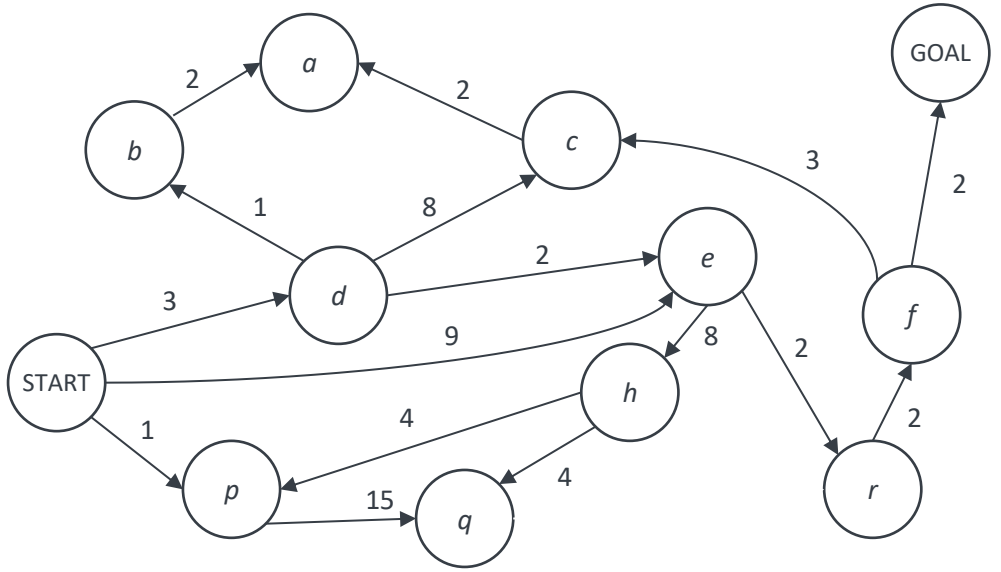
Uniform Cost Search (UCS)



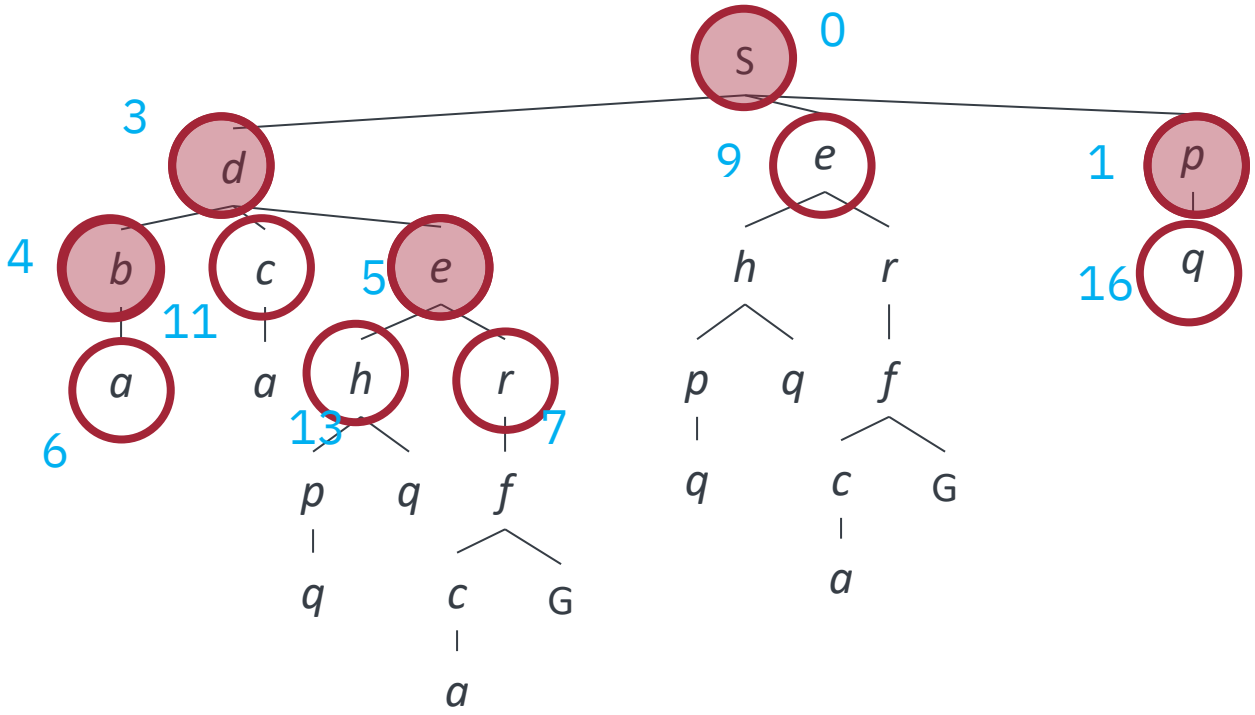


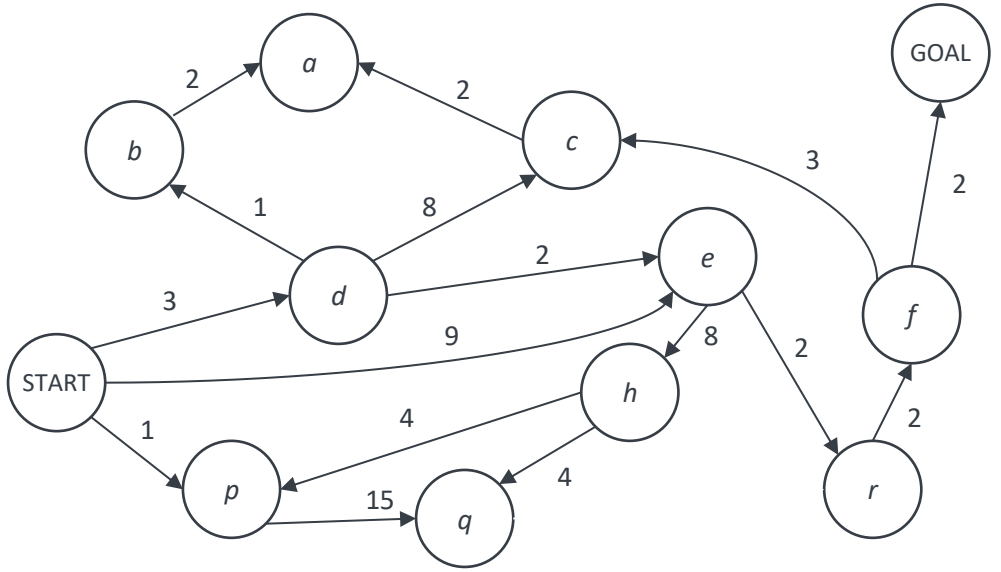
Uniform Cost Search (UCS)



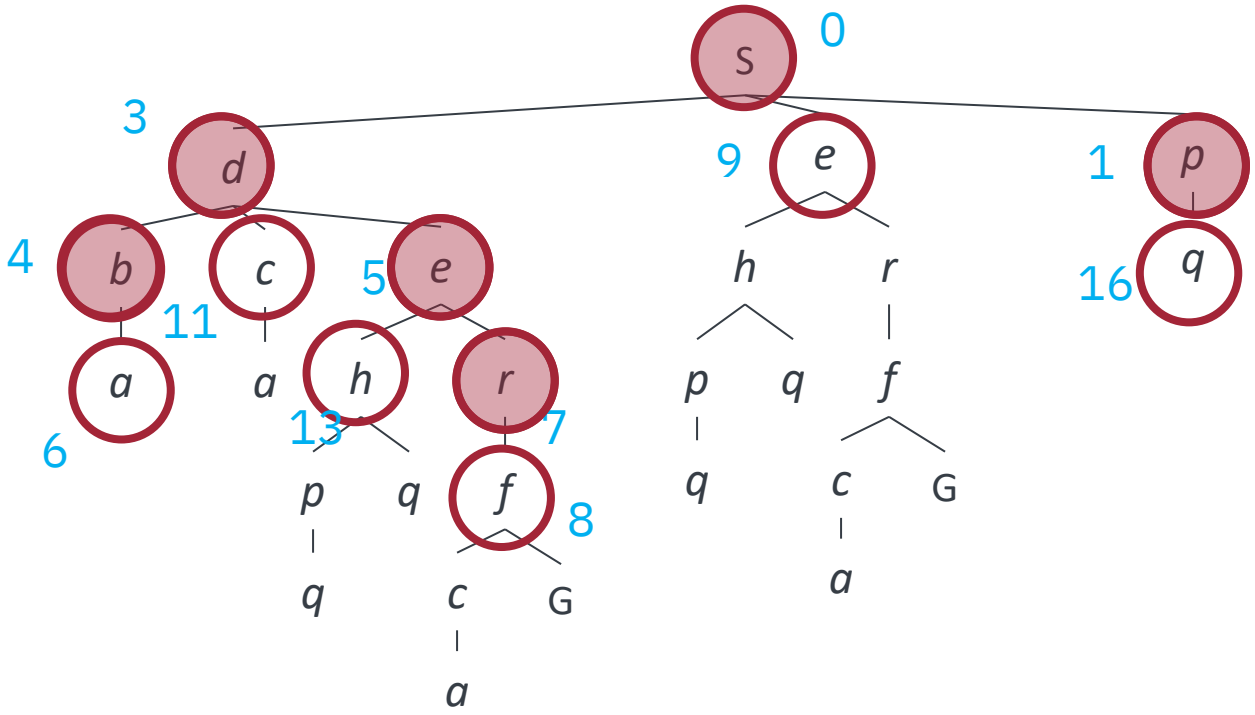


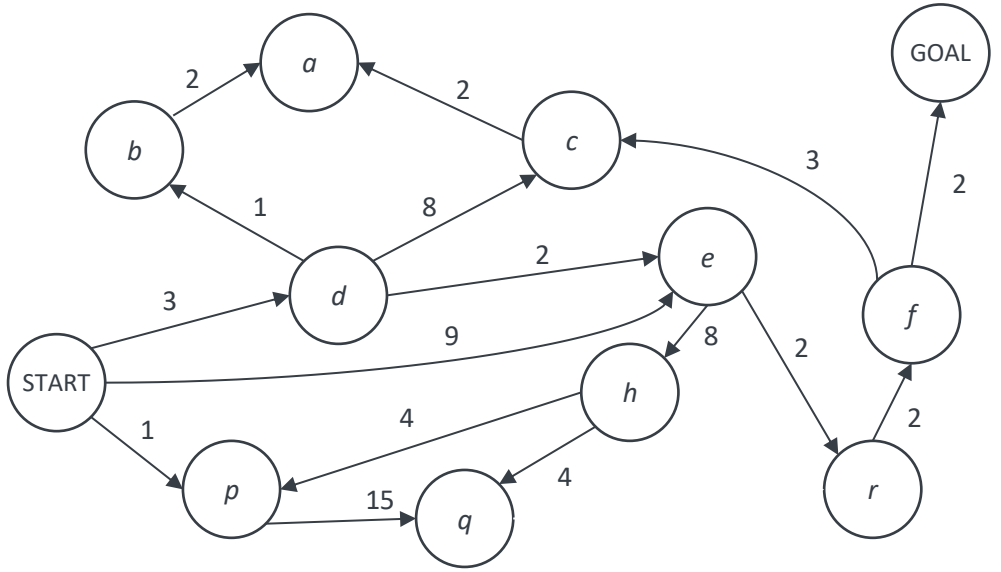
Uniform Cost Search (UCS)



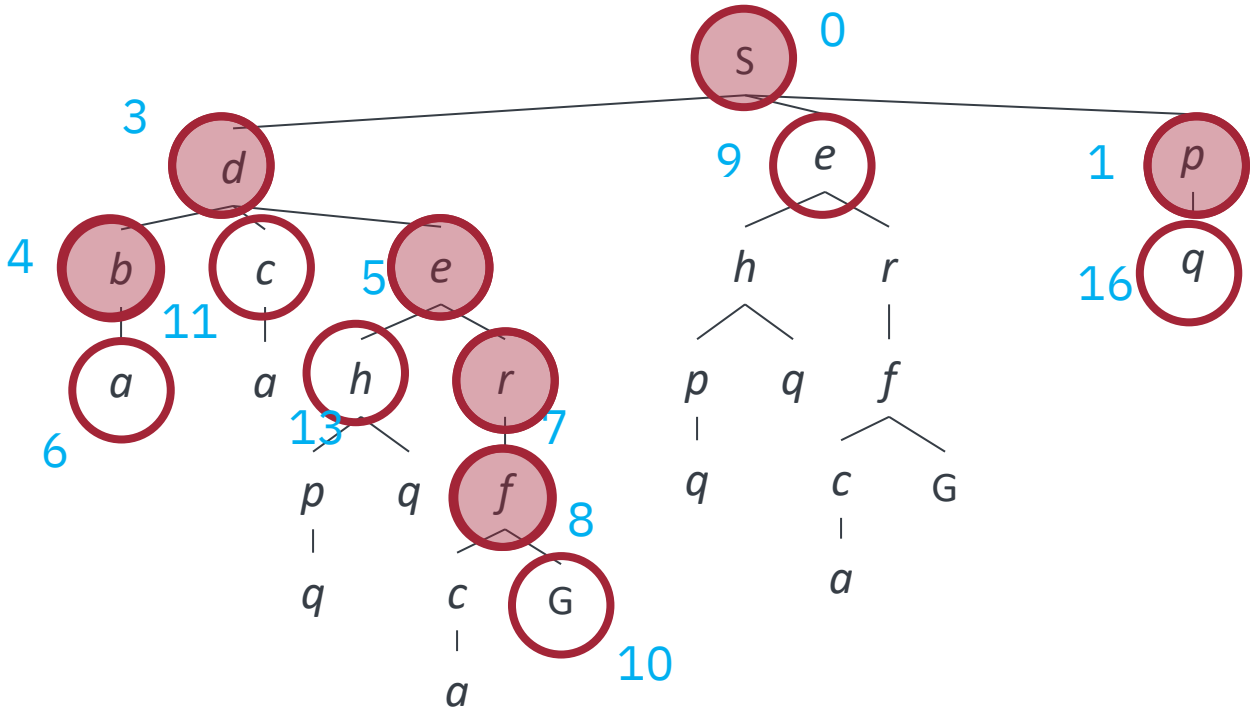


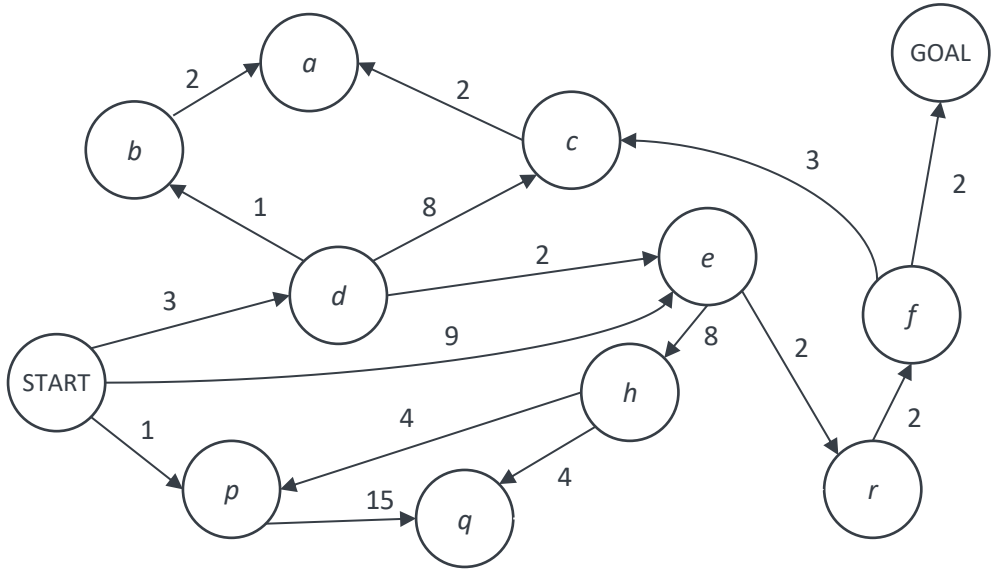
Uniform Cost Search (UCS)



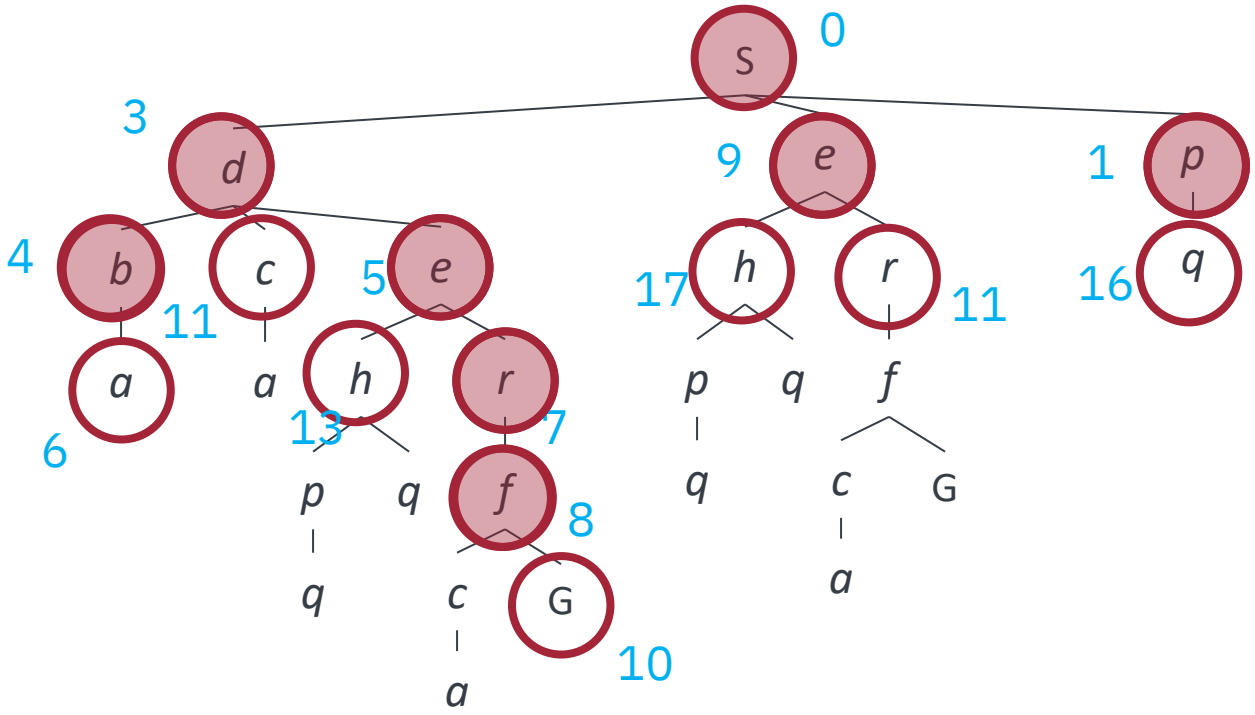


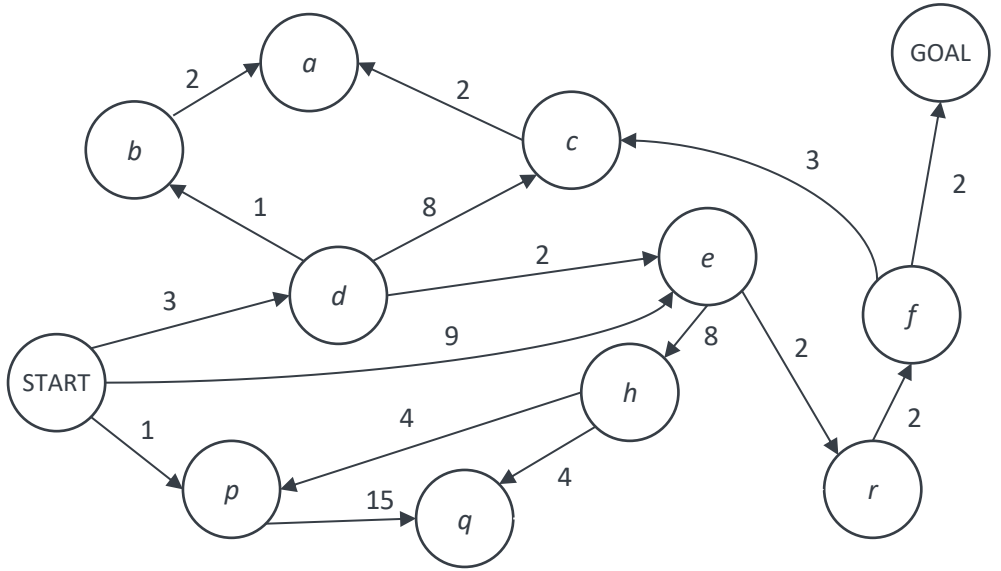
Uniform Cost Search (UCS)



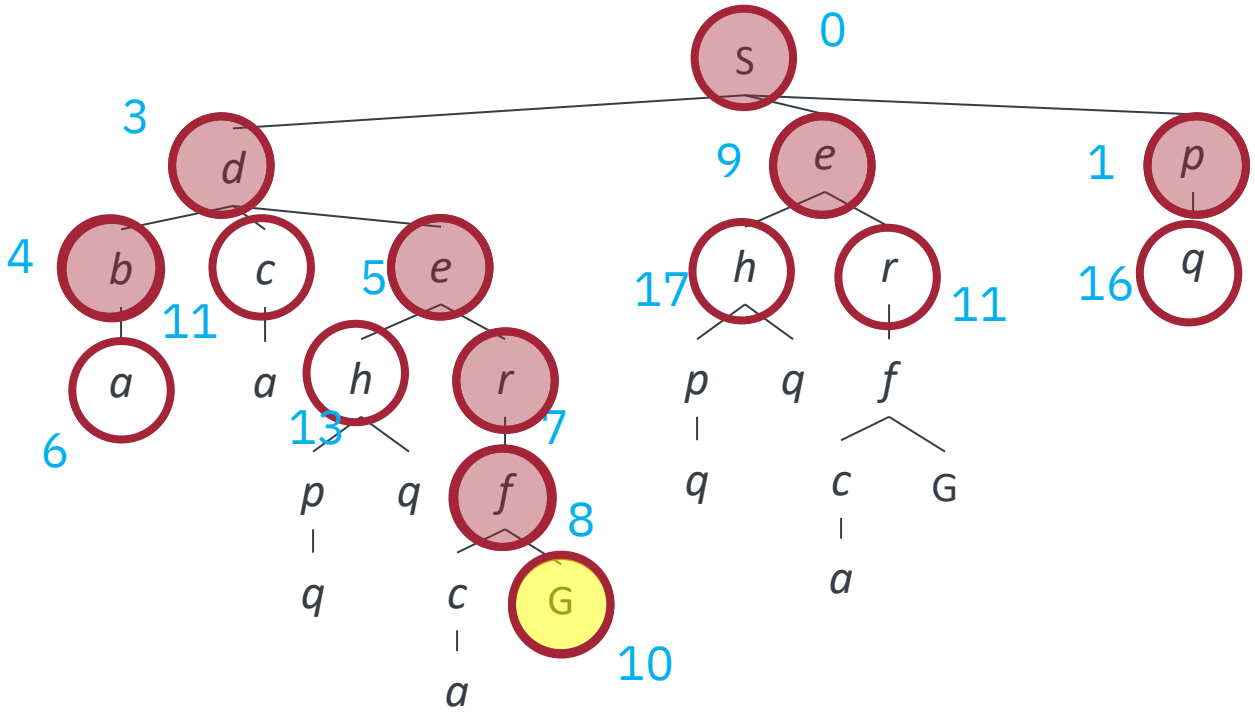


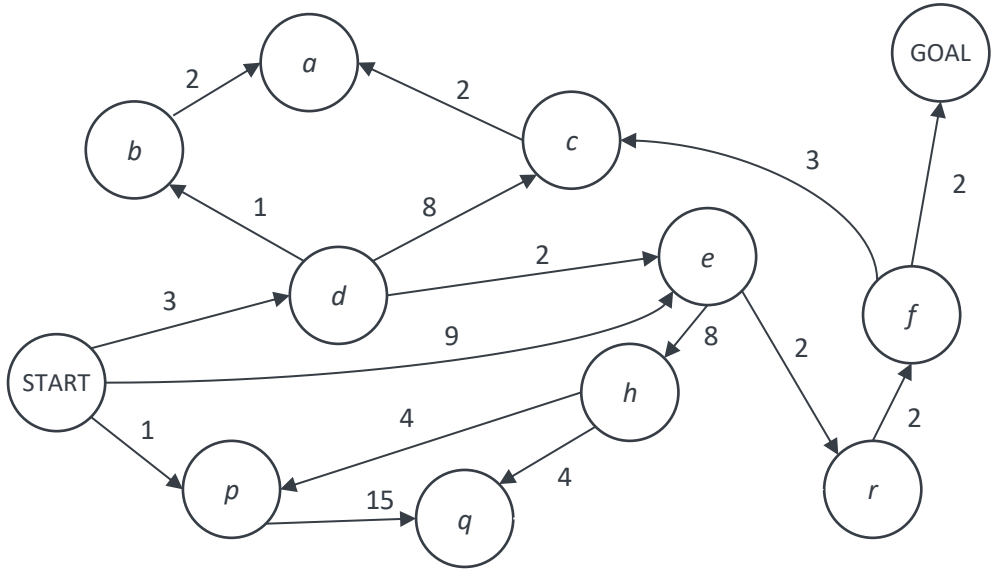
Uniform Cost Search (UCS)



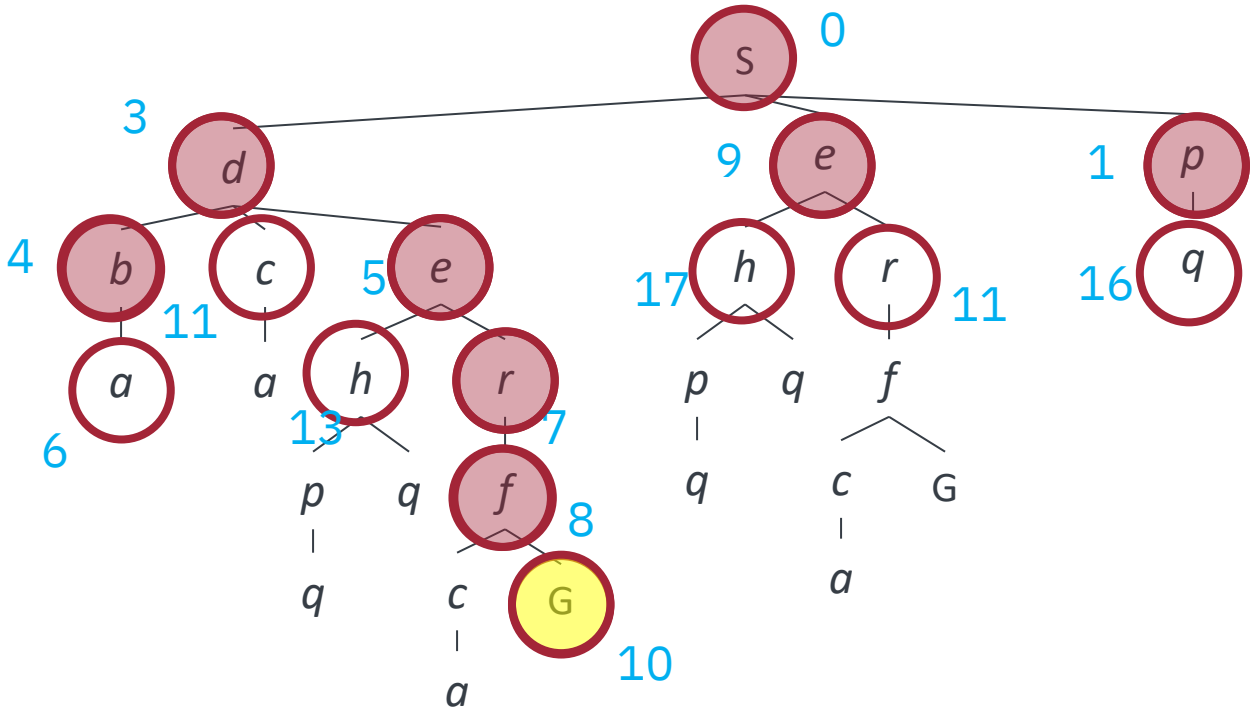


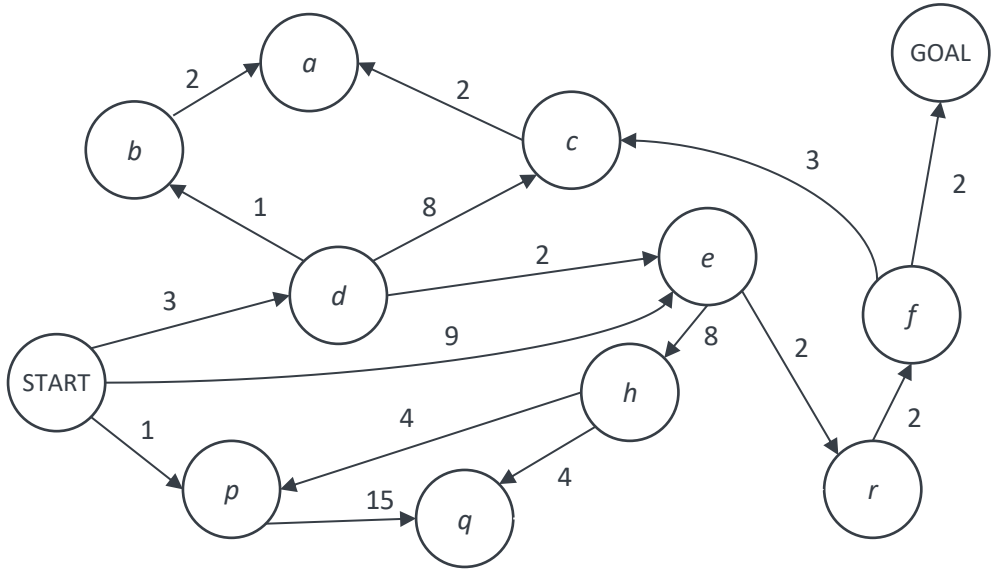
Uniform Cost Search (UCS)





Uniform Cost Search (UCS)

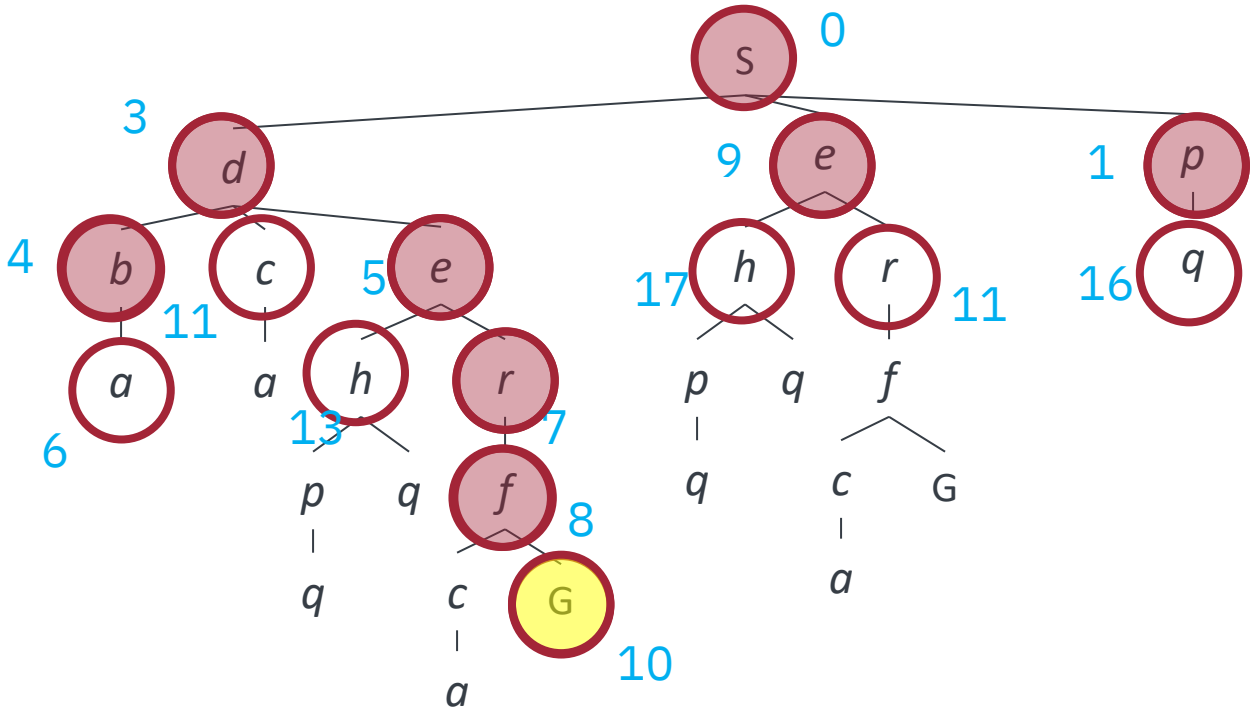


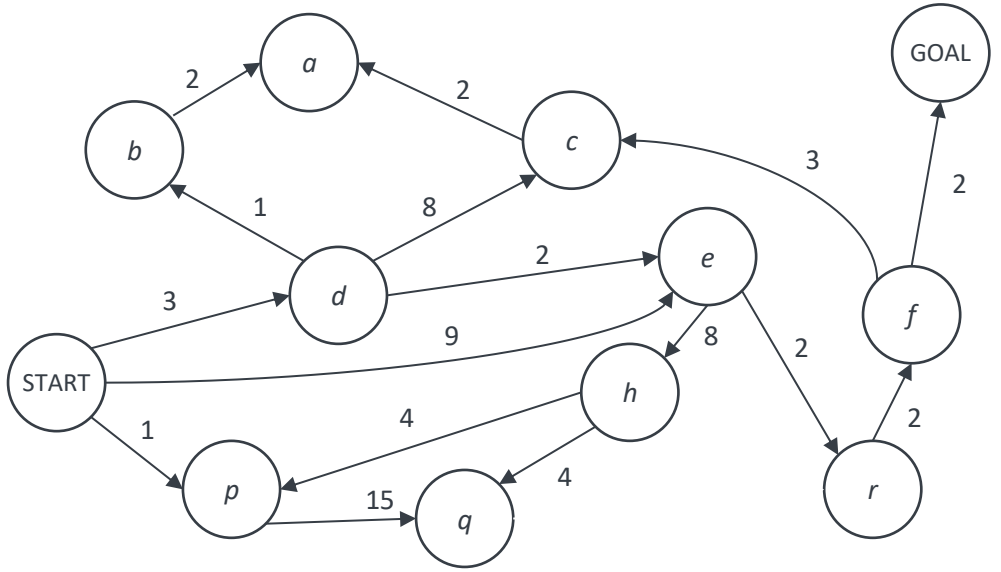


Note that the search ends when the goal node is **expanded** (not generated).

Why?

Uniform Cost Search (UCS)

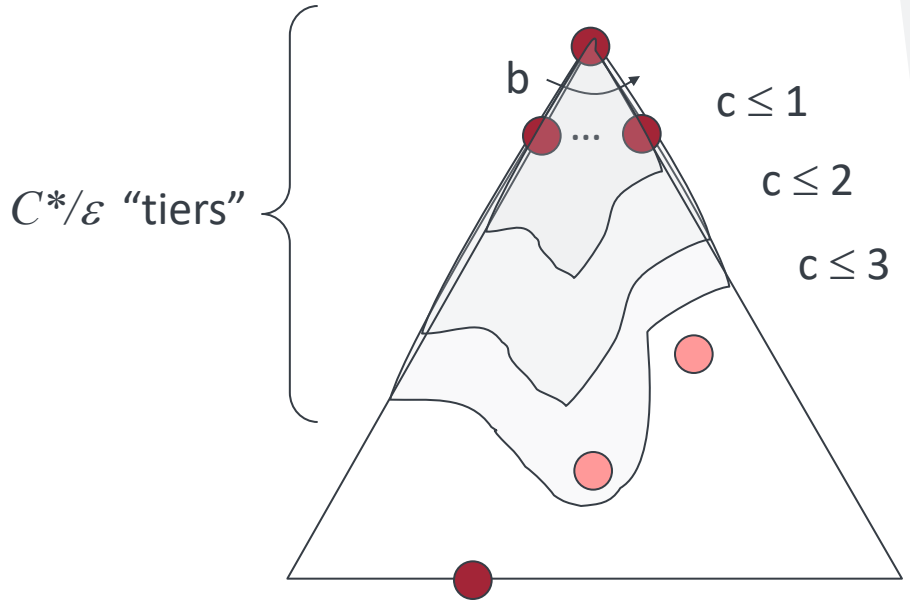




Number of expanded nodes: 9

UCS properties

- What nodes does UCS expand?
  - Processes all nodes with cost less than cheapest solution!
  - If that solution costs C^* and arcs cost at least ε, then the "effective depth" is roughly C^*/ε
  - Takes time $O(b^{C^*/\varepsilon})$ (exponential in effective depth)
- How much space does the fringe take?
  - Has roughly the last tier, so $O(b^{C^*/\varepsilon})$
- Is it complete?
  - Assuming best solution has a finite cost and minimum arc cost is positive, yes!
- Is it optimal?
  - Yes!



UCS: proof of optimality

- Lemma 1
  - Let c(n) be the cost of a generated node n. If n2 is expanded immediately after n1 then c(n1) ≤ c(n2).
- Proof: 2 cases
  - n2 was in the fringe when n1 was expanded:
    - We must have c(n1) ≤ c(n2) otherwise, n2 would have been selected.
  - n2 is a successor of n1:
    - Now c(n1) ≤ c(n2) because the path represented by n2 extends the path represented by n1.

UCS: proof of optimality

- Lemma 2
  - When node n is expanded, every path in the search space with cost strictly less than c(n) has already been expanded.
- Proof
  - Let nk = <Start, s1, …, sk> be any path with cost less than c(n). **We will prove that nk has already been expanded before n was extracted from the fringe.**
  - Define a collection of paths ni each of which is a prefix of the path nk.
    - n0 =<Start>,
    - n1 =<Start, s1>,
    - n2 =<Start, s1, s2>,
    - …,
    - ni = <Start, s1, …, si>,
    - …,
    - nk =<Start, s1, …, sk>.

Note that since each path is a prefix of the subsequent paths c(ni) < c(ni+1) < c(ni+2) … < c(nk).

UCS: proof of optimality

- Proof (cont.)
  - **Let ni be the last node in the sequence that has already been expanded by search when n is extracted from the fringe.**
  - If i=k, then nk has already been expanded and we are done.
  - If i<k, then we show that this is a contradiction.
    - i>0 since the path <Start> is the first node expanded by the search.
    - Since ni is the last node on the sequence to already expanded, ni+1 has not been expanded. ni+1 is a successor of ni, so it must have been added to the fringe. ni+1 should be in the fringe when node n was selected to be expanded, and thus, $c(n_{i+1}) \geq c(n)$.
    - $i+1 \leq k$ because i<k. Therefore, $c(n_{i+1}) \leq c(n_k)$. However, $c(n_k) \leq c(n)$, which contradicts the previous statement, $c(n_{i+1}) \geq c(n)$.

UCS: proof of optimality

- Lemma 3
  - The first time uniform-cost expands a node n terminating at state S, it has found the minimal cost path to S (it might later find other paths to S but none of them can be cheaper).
- Proof
  - All cheaper paths have already been expanded, none of them terminated at S.
  - All paths expanded after n will be at least as expensive, so no cheaper path to S can be found later.

Cycle checking

Cycle checking

If nk is the path <s0, s1, …, sk> and we expand sk to obtain child successor state c, we have <s0, s1, …, sk, c> as the path to c.

We write such paths as <n, c> where n is the prefix and c is the final state in the path.

- Cycle checking
  - Ensure that the state c is not equal to the state reached by any ancestor of c along this path. $c \notin \{s0,s1,\ldots,sk\}$

Cycle checking

```
function Tree-Search(problem) returns a solution, or failure
        Initialize the fringe using the initial state of problem
        loop do
                if the fringe is empty then return failure
                choose a leaf node and remove it from the fringe
                if the node contains a goal state then return the corresponding solution
                for succ in successors(node):
                        if not succ in node:   # Don't put cyclic paths on the fringe.
                                frontier.insert(<node, succ>)
```

Informed (Heuristic) search

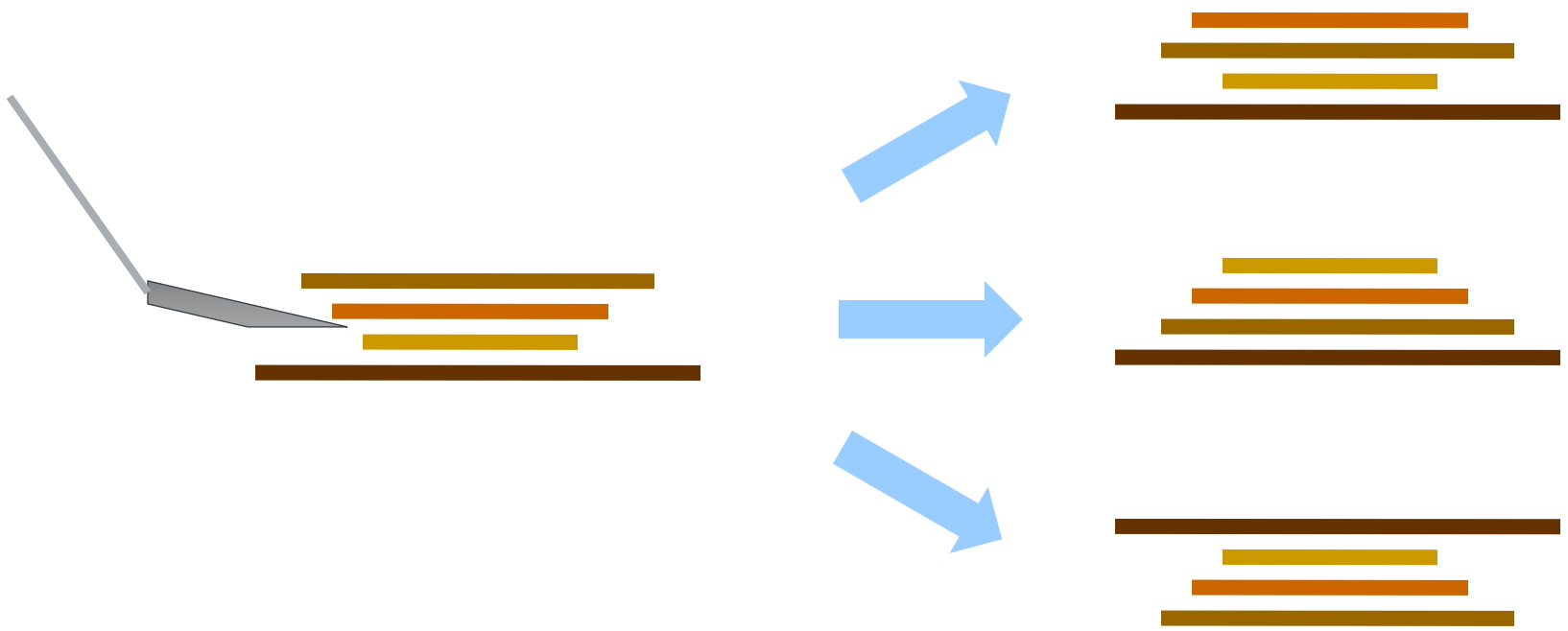
Chapter 3

Informed Search

- Heuristics
- Greedy Search
- A* Search

Graph Search

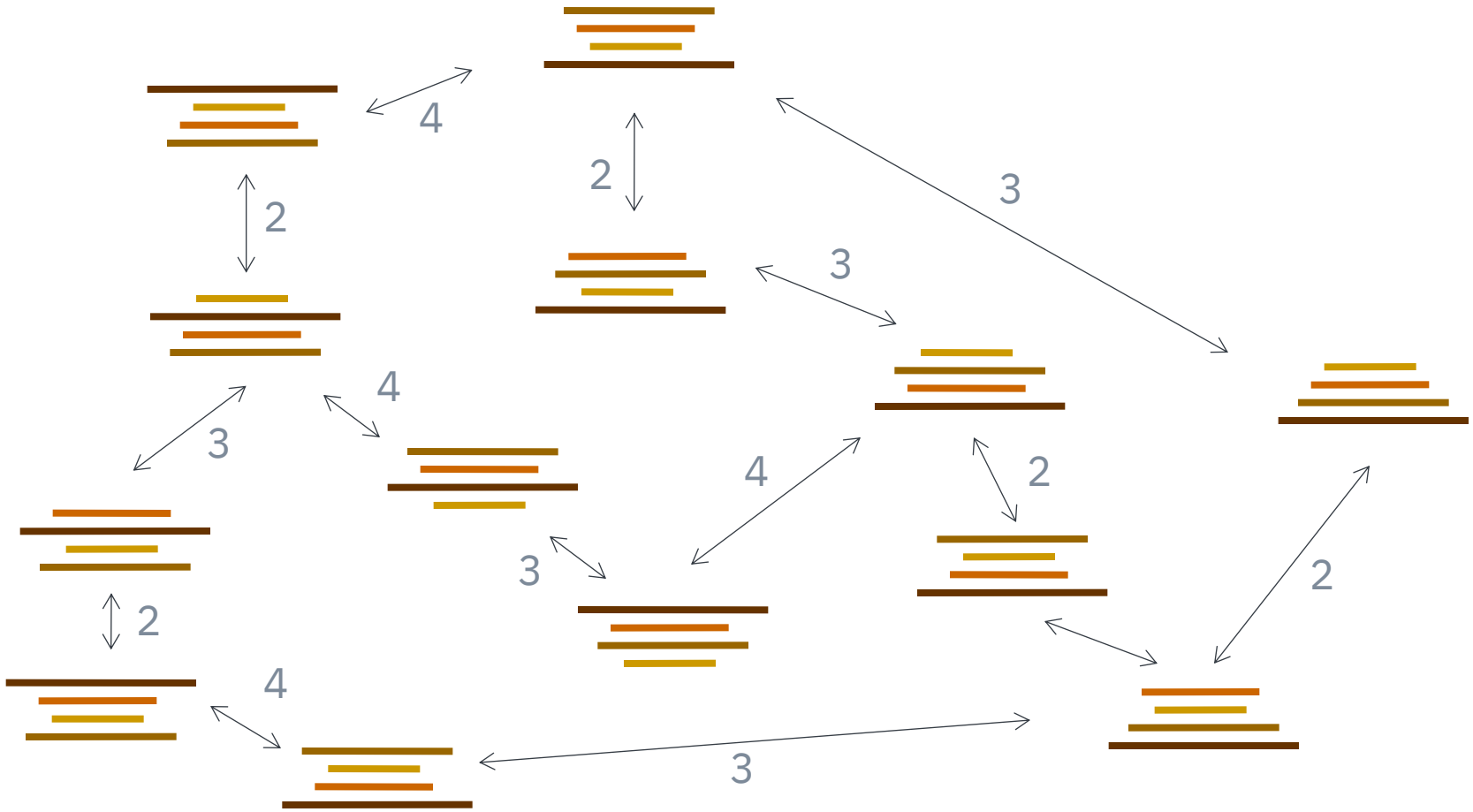
Example: pancake problem



Cost: Number of pancakes flipped

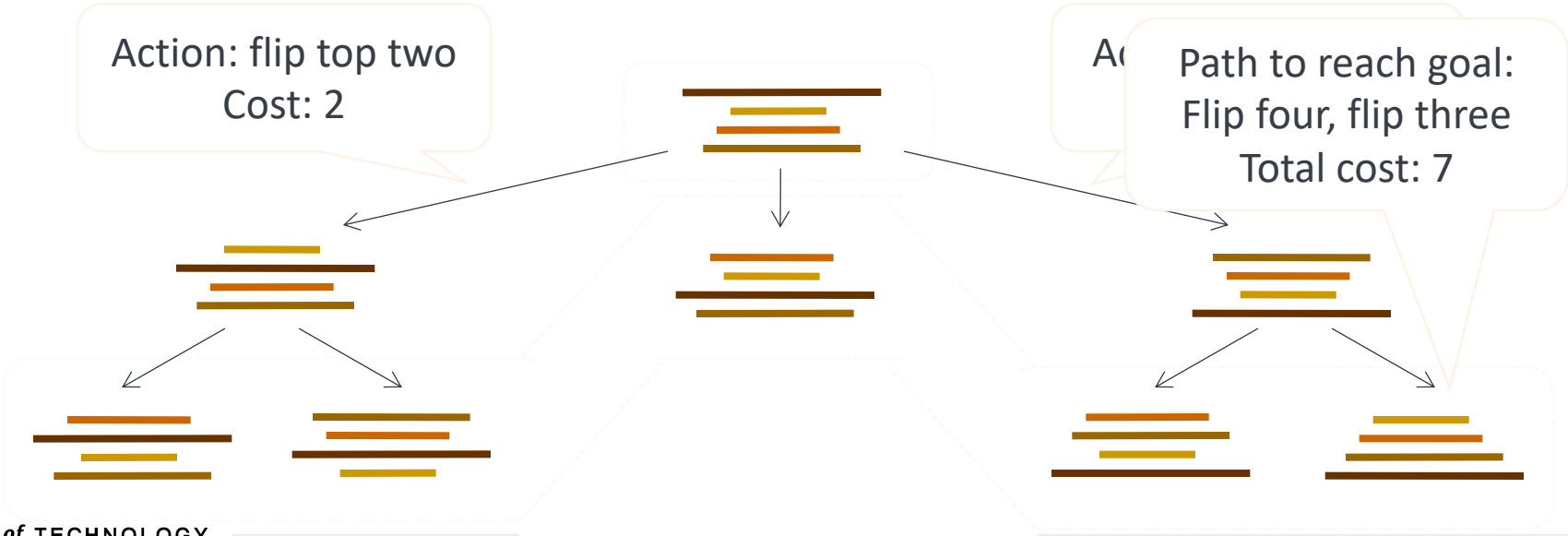
Example: pancake problem

State space graph with costs as weights



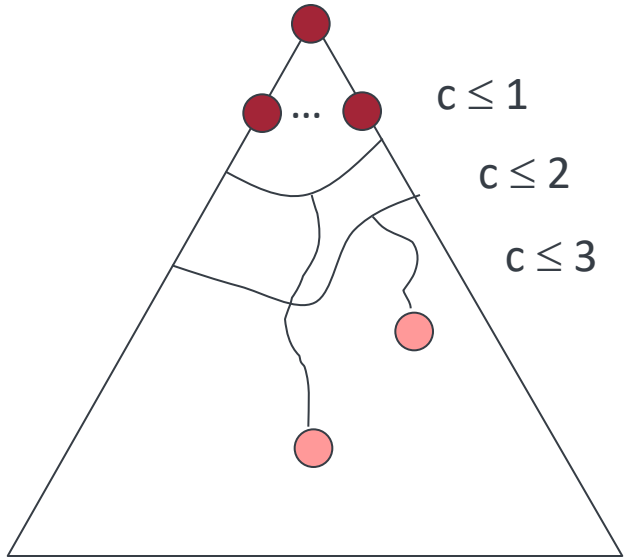
General tree search

function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier



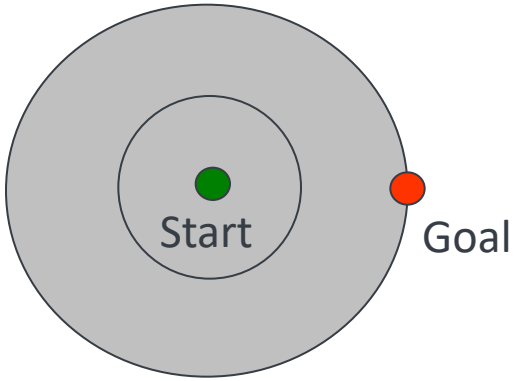
Uninformed search: uniform-cost sarch

- Strategy: expand lowest path cost



- The good: UCS is complete and optimal!

- The bad:
  - Explores options in every “direction”
  - No information about goal location



Informed search

J

What is informed search?

Informed search is a type of search algorithm that uses additional information or heuristics to guide the search process. This information is used to estimate the cost of reaching the goal state from a given state, and the search algorithm uses this estimate to guide its search for a solution. Examples of informed search algorithms include A* and IDA*.

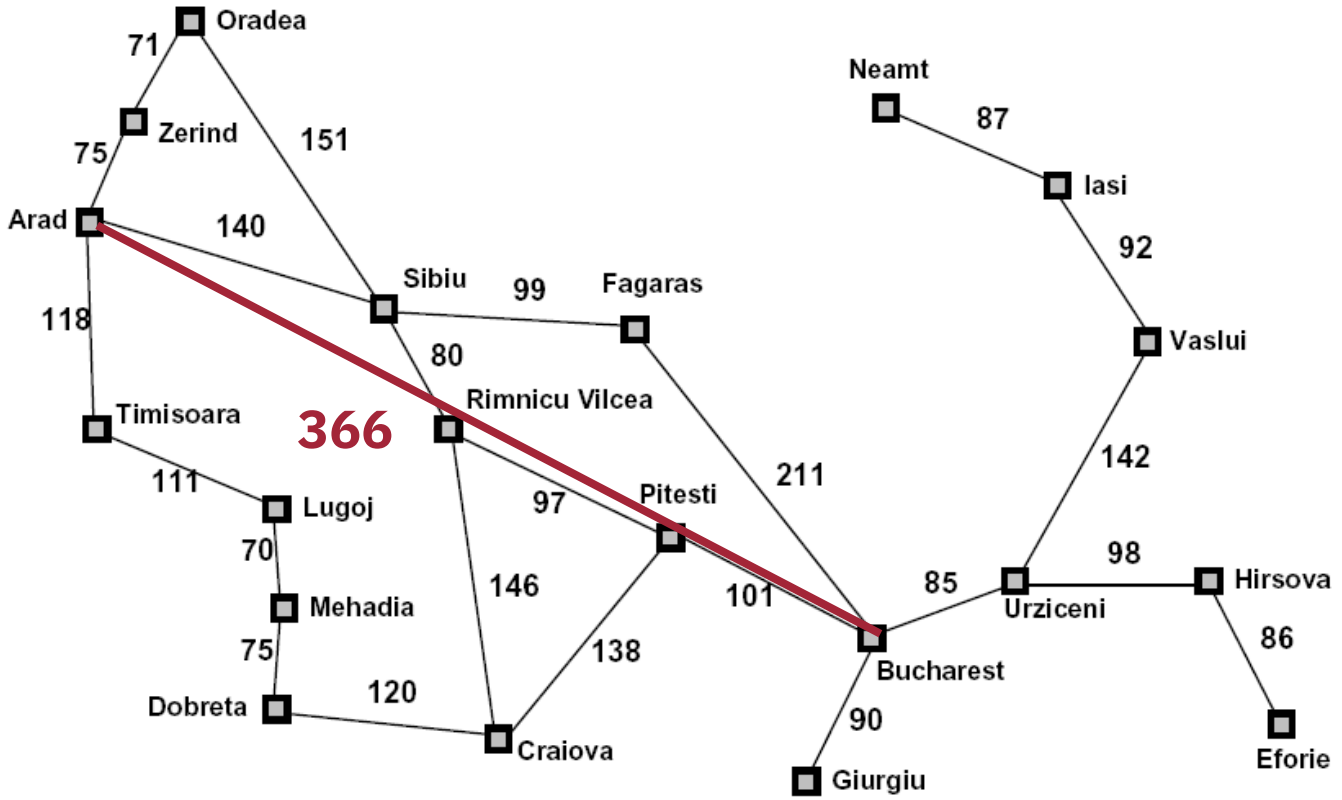
Informed search

- In uninformed search, we don't try to evaluate which of the nodes on the fringe are most promising.
  - We never “look-ahead” to the goal. *E.g.*, in uniform cost search we always expand the cheapest path. We don't consider the cost of getting from the end of the current path to the goal
- Often, we have some other knowledge about the merit of nodes, *e.g.*, going the wrong direction in Romania.

Search heuristics

- A heuristic is:
  - A function that estimates how close a state is to a goal
  - Designed for a particular search problem
  - Examples: Manhattan distance, Euclidean distance for pathing

Example: heuristic function

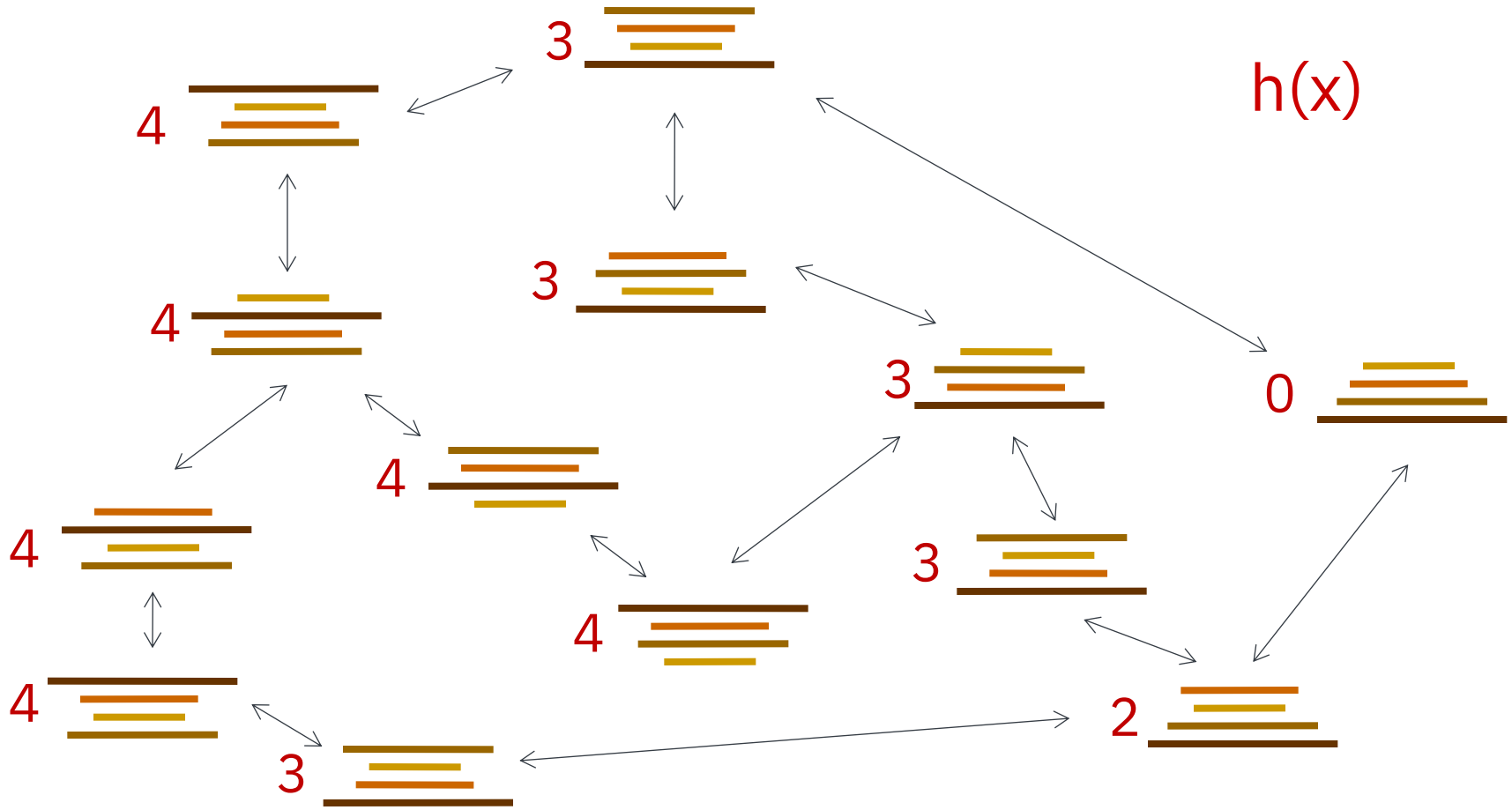


Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

h(x)

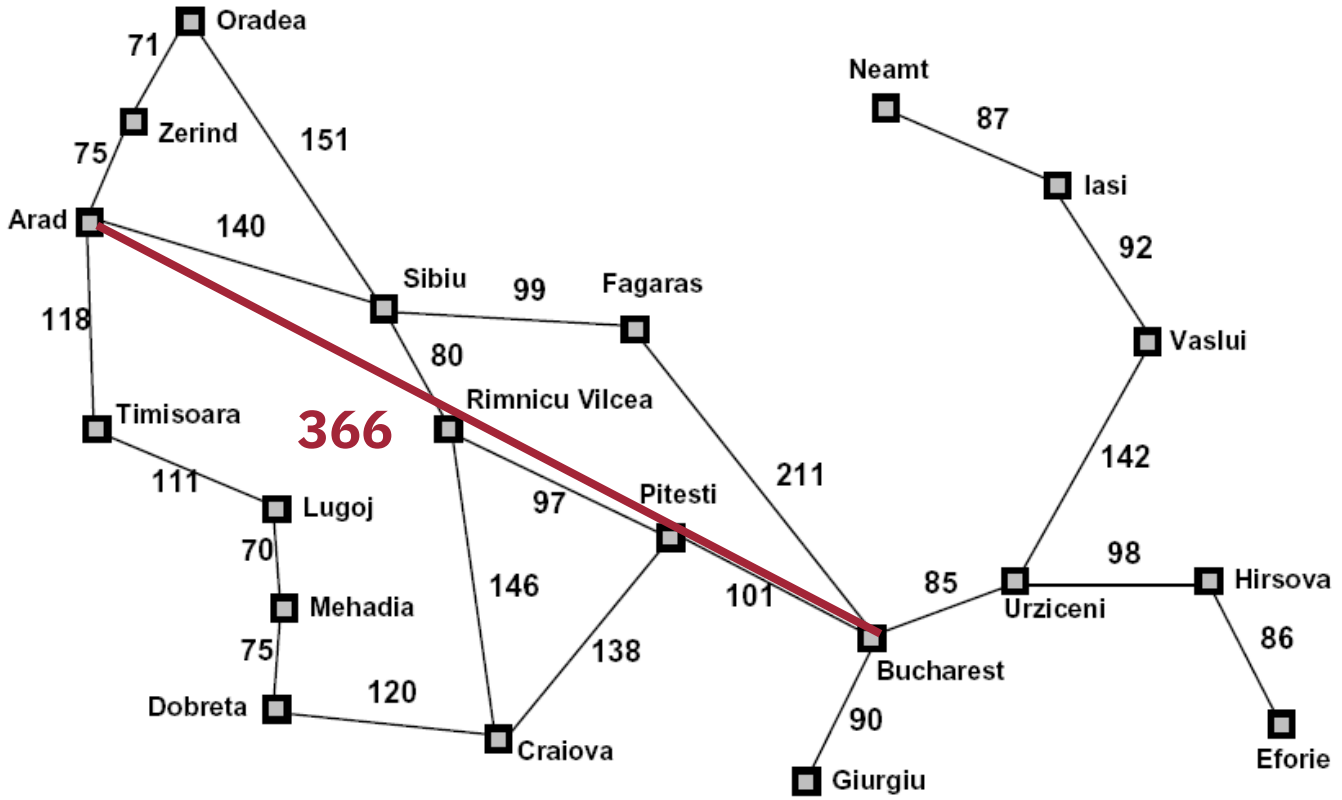
Example: heuristic function

Heuristic: the number of the largest pancake that is still out of place



Greedy search

Example: heuristic function

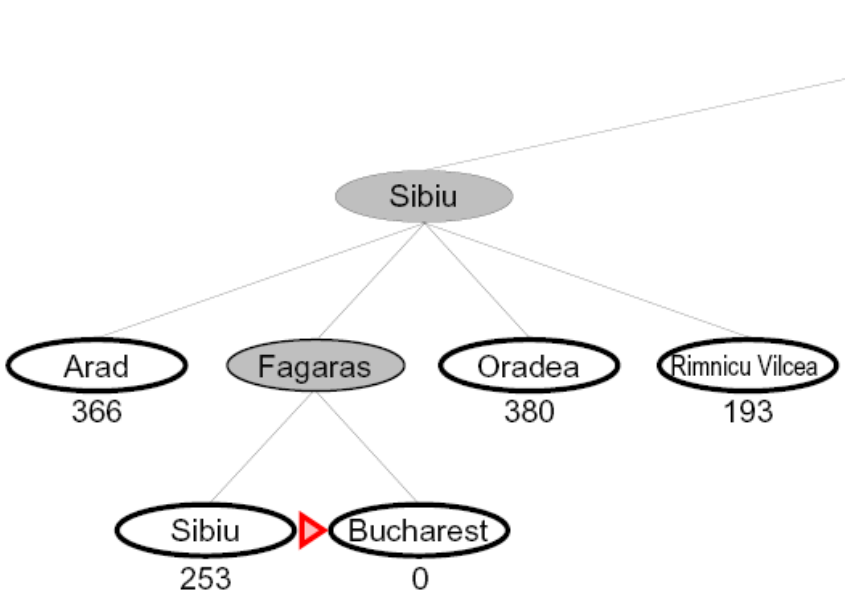


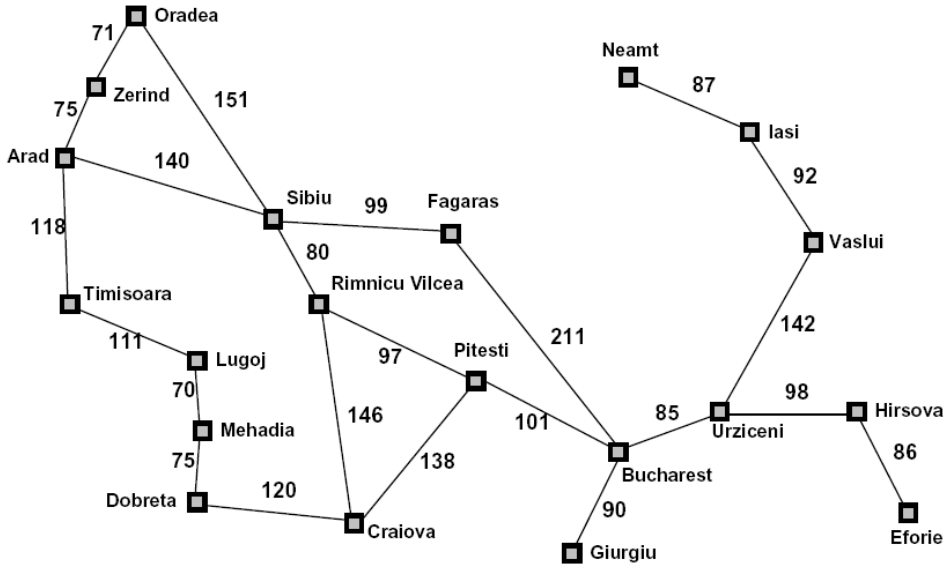
Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

h(x)

Greedy search

- Expand the node that seems closest…



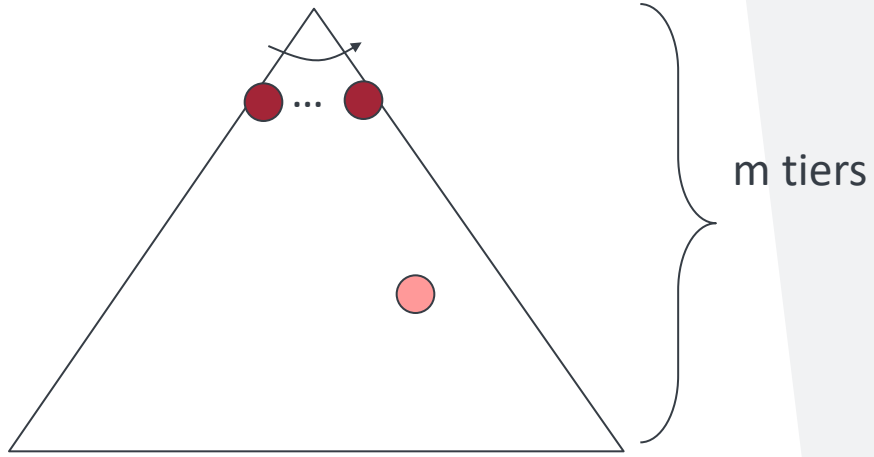


Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

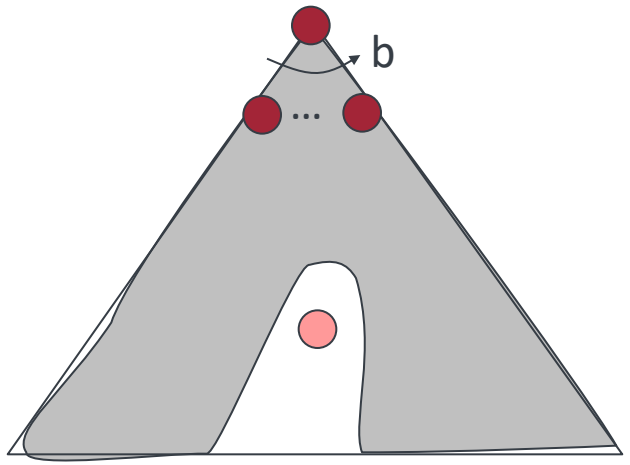
- What can go wrong?

Greedy search

- Strategy: expand a node that you think is closest to a goal state
  - Heuristic: estimate of distance to nearest goal for each state



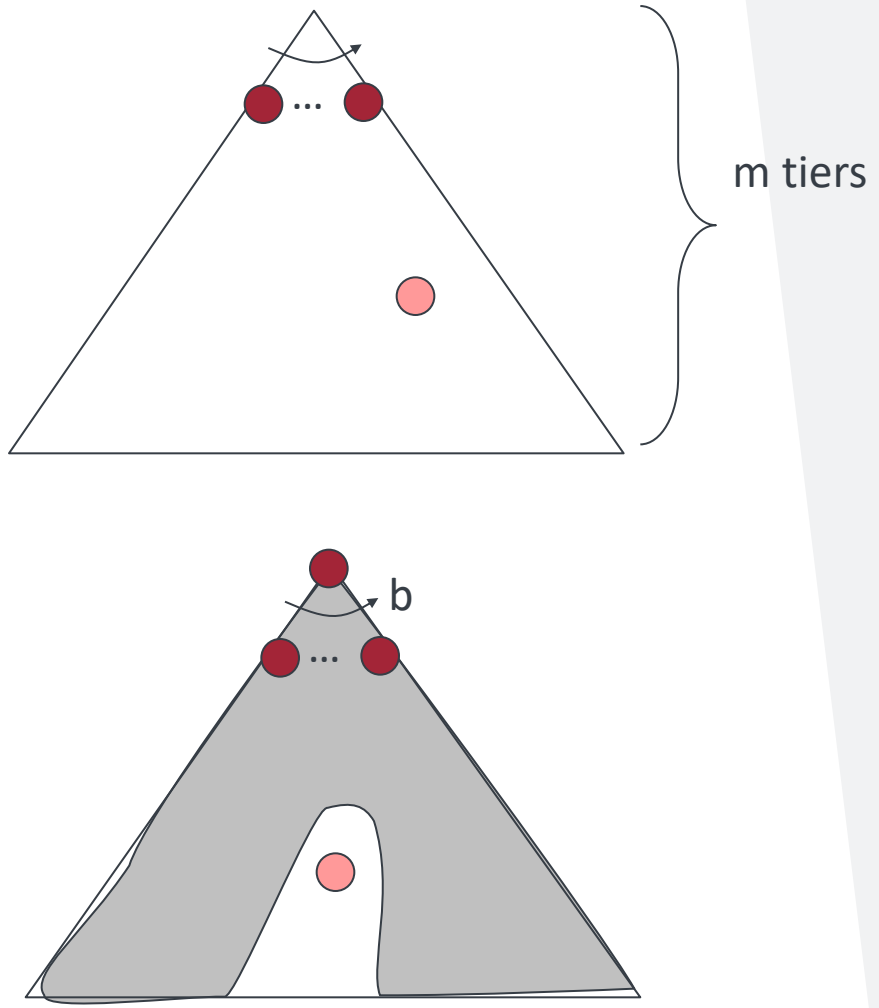
- A common case:
  - Best-first takes you straight to the (wrong) goal



- Worst-case: like a badly-guided DFS
- Complete?
- Optimal?

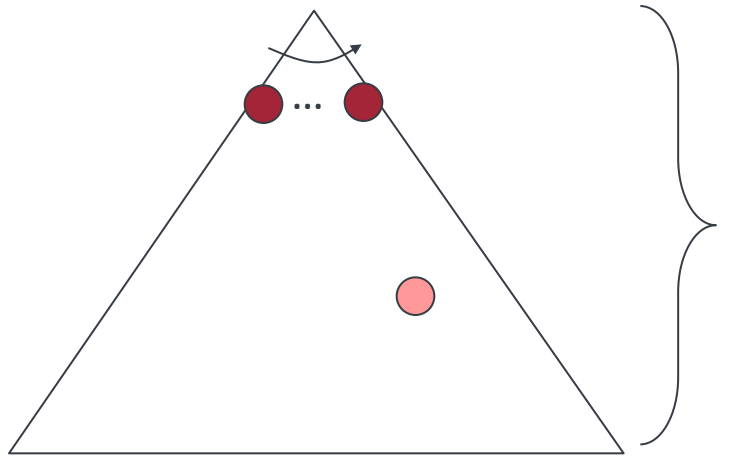
Greedy search

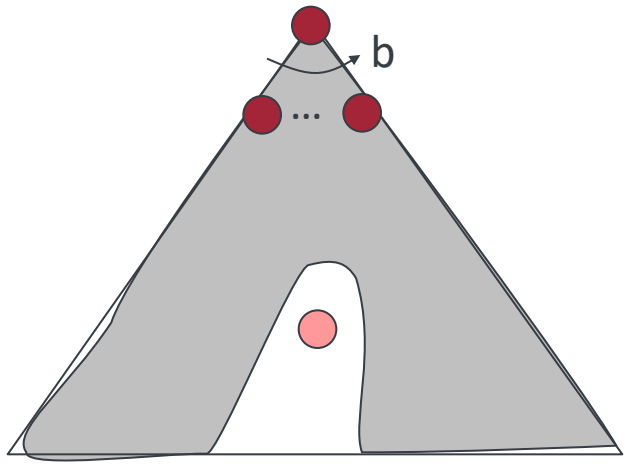
- Time complexity?
- Space complexity?



Greedy search

- Time complexity? $O(b^m)$
- Space complexity? O(bm)
- Depth-first search using heuristics for tie breaker

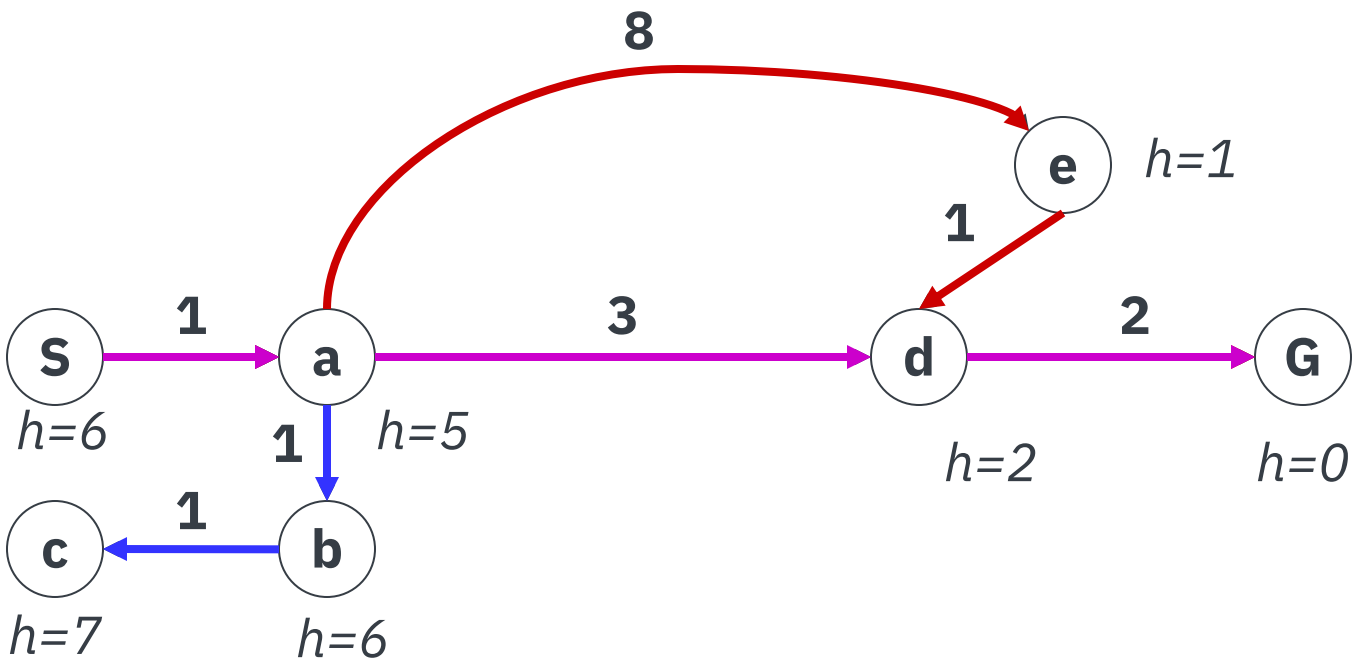


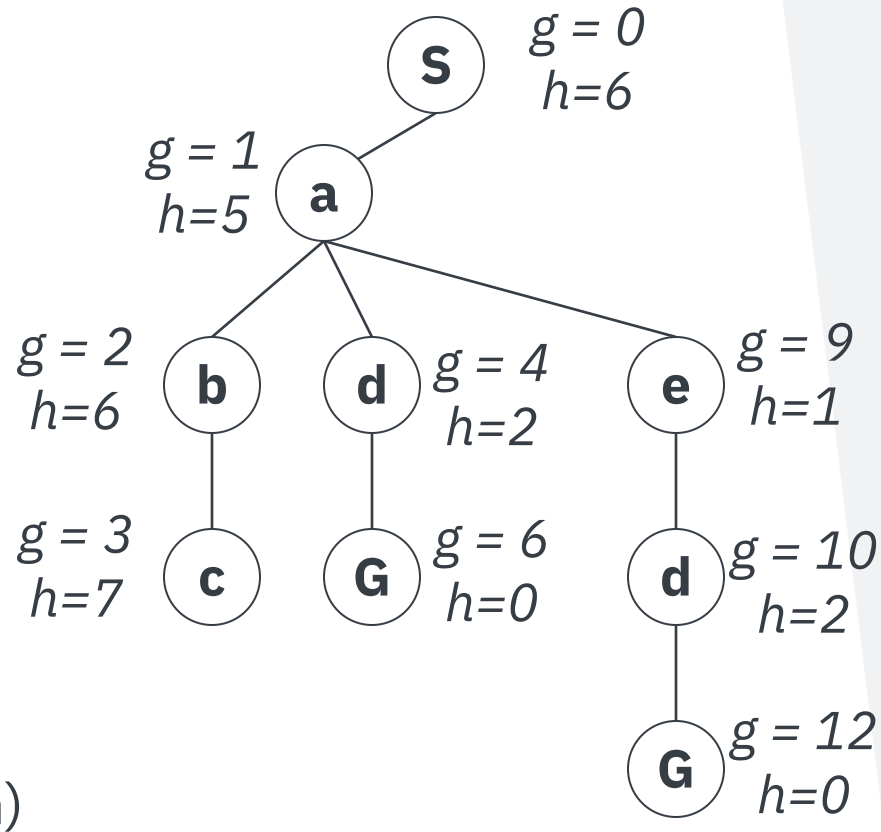


A* Search

Combining UCS and Greedy

- Uniform-cost orders by path cost, or *backward cost* g(n)
- Greedy orders by goal proximity, or *forward cost* h(n)

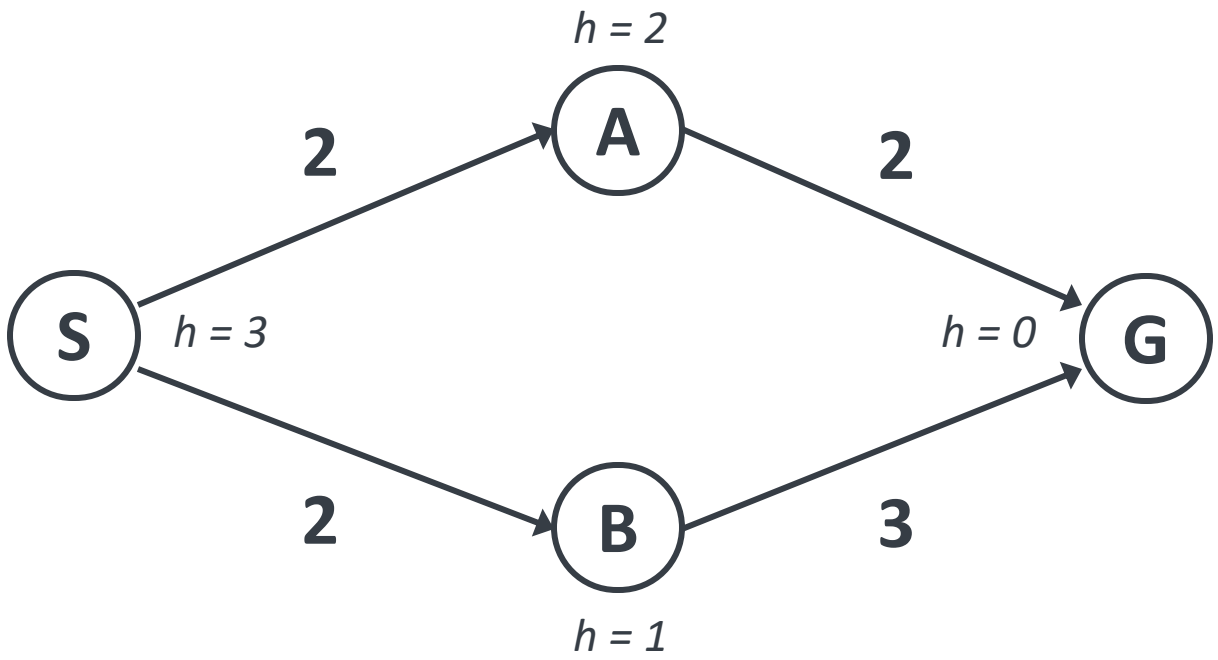




- A* Search orders by the sum: f(n) = g(n) + h(n)

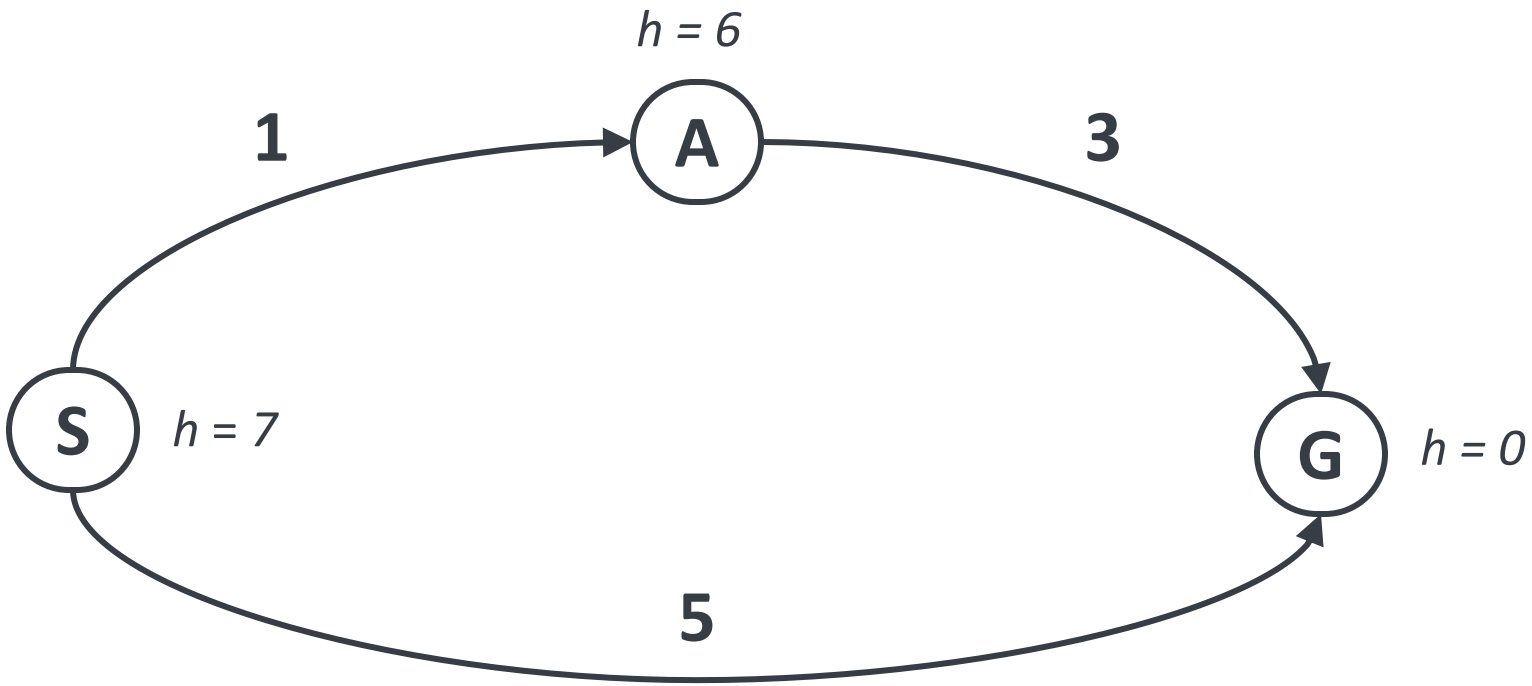
When should A* terminate?

- Should we stop when we **generate** a goal?



- No: only stop when we **expand** a goal

Is A* optimal?



- What went wrong?
- Actual bad goal cost < estimated good goal cost
- We need estimates to be less than actual costs!

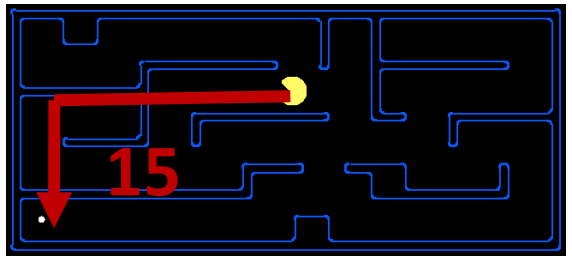
Admissible heuristics

A heuristic h is *admissible* (optimistic) if:

$$0 \leq h(n) \leq h^*(n)$$

where $h^*(n)$ is the true cost to a nearest goal

- Examples:



4

- Coming up with admissible heuristics is most of what's involved in using A* in practice.

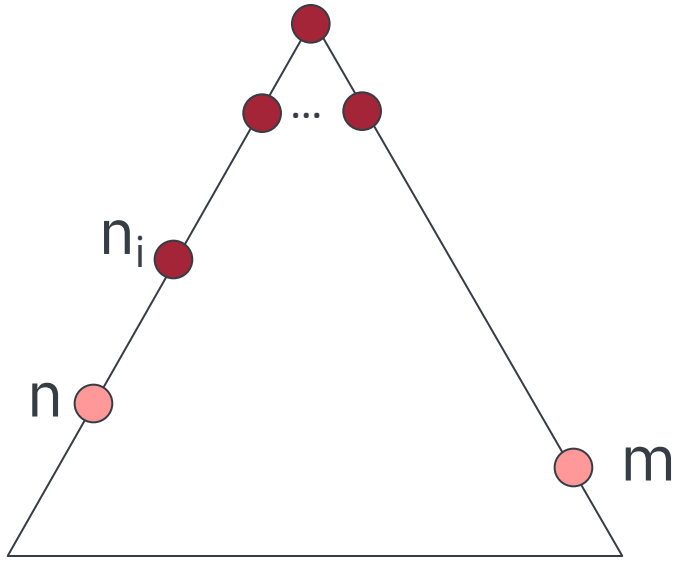
Admissible heuristics

- h(n) ≤ h*(n) means that the search won't miss any promising paths.
  - If it really is cheap to get to a goal via n (*i.e.*, both g(n) and h*(n) are low), then f(n) = g(n) + h(n) will also be low, and the search won't ignore n in favor of more expensive options.
- This can be formalized to show that admissibility implies optimality.

Optimality of A* tree search

Assume:

- n is an optimal goal node
- m is a suboptimal goal node
- h is admissible



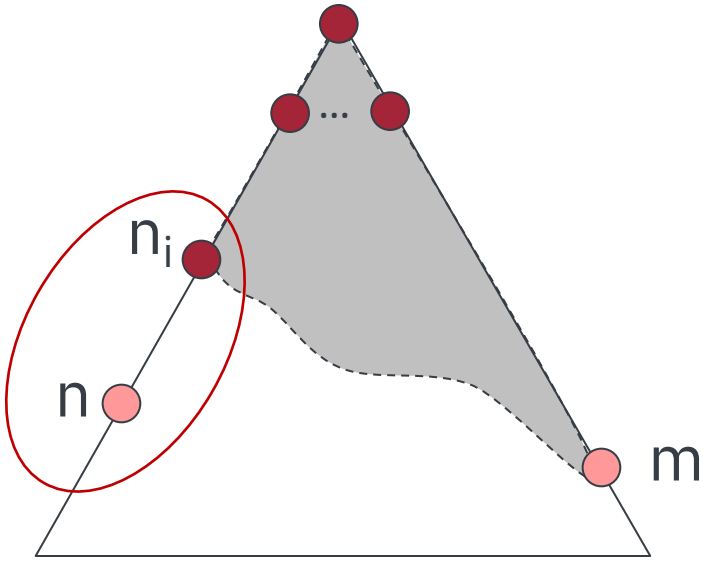
Claim:

- n will exit the fringe before m

Optimality of A* tree search: blocking

Proof:

- Imagine m is on the fringe
- Some ancestor n_i of n is on the fringe, too (maybe n!)
- Claim: *n* will be expanded before m
  1. $f(n_i)$ is less or equal to f(n)



$f(n) = g(n) + h(n)$ Definition of f-cost

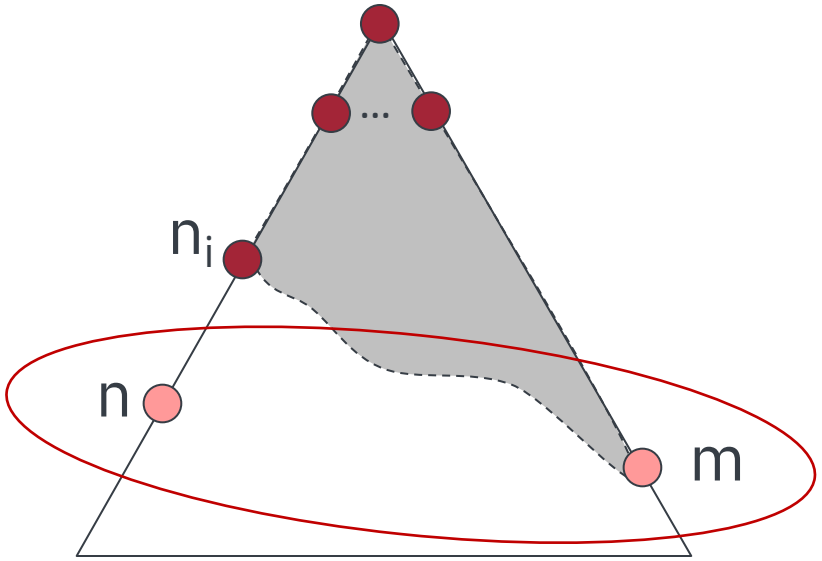
$f(n_i) \leq g(n)$ Admissibility of h

$g(n) = f(n)$ h = 0 at a goal

Optimality of A* tree search: blocking

Proof:

- Imagine m is on the fringe
- Some ancestor n_i of n is on the fringe, too (maybe n!)
- Claim: n_i will be expanded before m
  1. $f(n_i)$ is less or equal to f(n)
  2. f(n) is less than f(m)



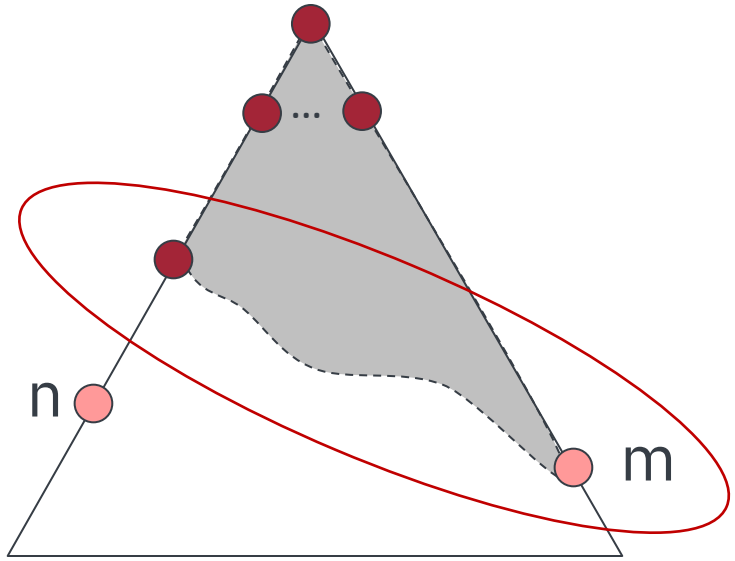
$$g(n) < g(m)$$

$$f(n) < f(m)$$

Optimality of A* tree search: blocking

Proof:

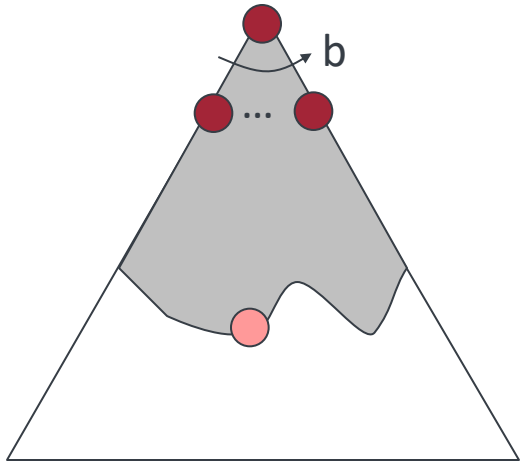
- Imagine m is on the fringe
- Some ancestor n_i of n is on the fringe, too (maybe n!)
- Claim: n_i will be expanded before m
  1. $f(n_i)$ is less or equal to f(n)
  2. f(n) is less than f(m)
  3. n_i expands before m
- All ancestors of n expand before m.
- n expands before m.
- A* search is optimal.



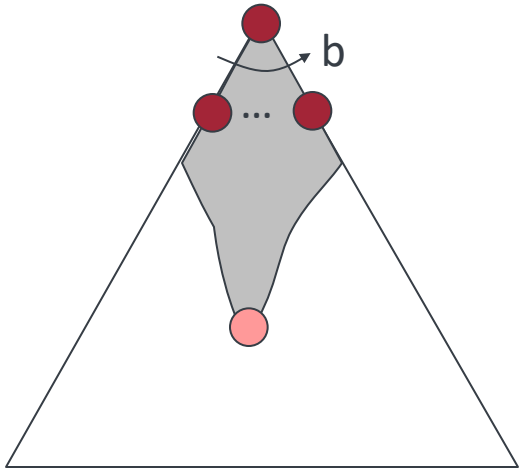
$$f(n_i) \leq f(n) \leq f(m)$$

Properties of A*

Uniform-Cost

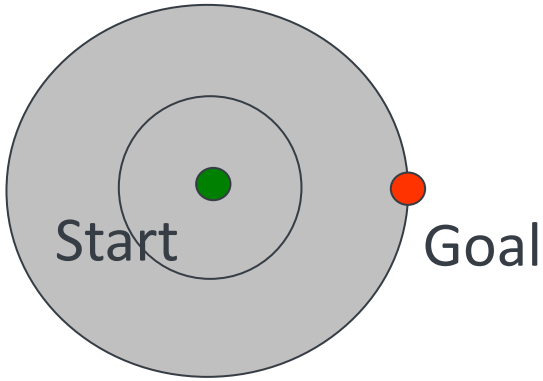


A*

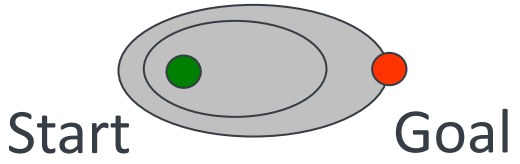


UCS vs. A* contours

- Uniform-cost expands equally in all “directions”



- A* expands mainly toward the goal, but does hedge its bets to ensure optimality



Which search algorithm is not optimal?

- Depth-first search
- Breadth-first search
- Uniform-cost search
- A* search

Total Results: 0

Which search algorithm is not optimal?

Depth-first search

Breadth-first search

Uniform-cost search

A* search

Which search algorithm is not optimal?

Depth-first search

Breadth-first search

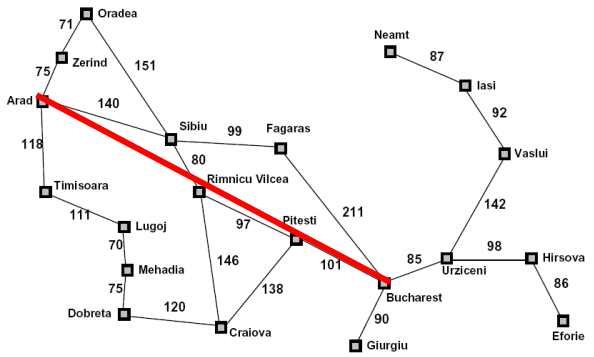
Uniform-cost search

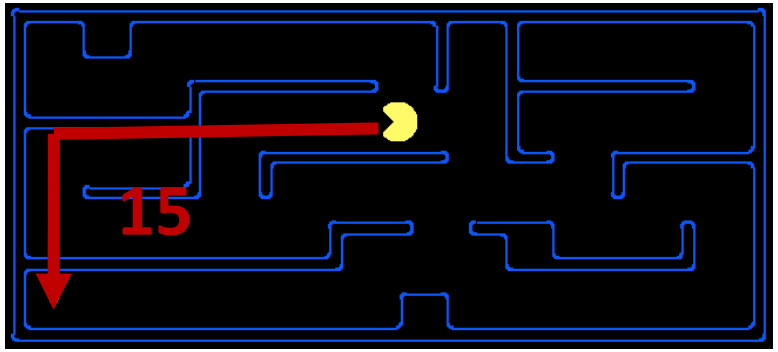
A* search

Creating admissible heuristics

- Most of the work in solving hard search problems optimally is in coming up with admissible heuristics
- Often, admissible heuristics are solutions to *relaxed problems,* where new actions are available

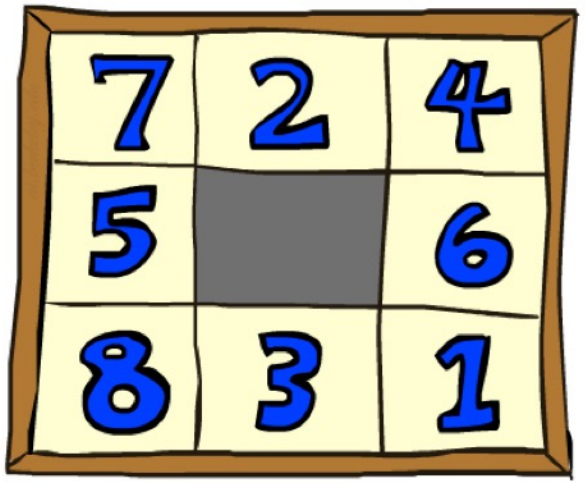
366



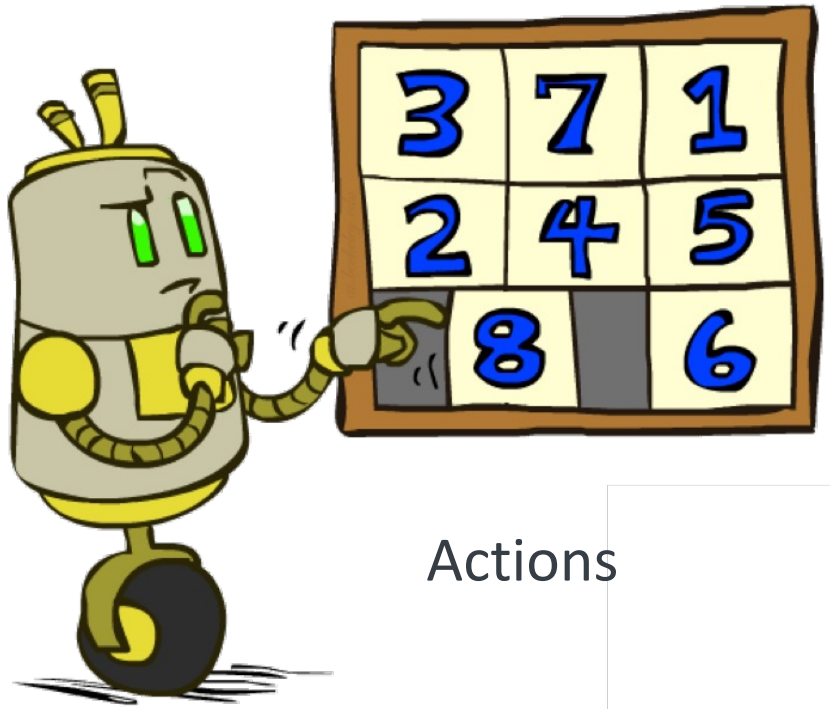


- Inadmissible heuristics are often useful too

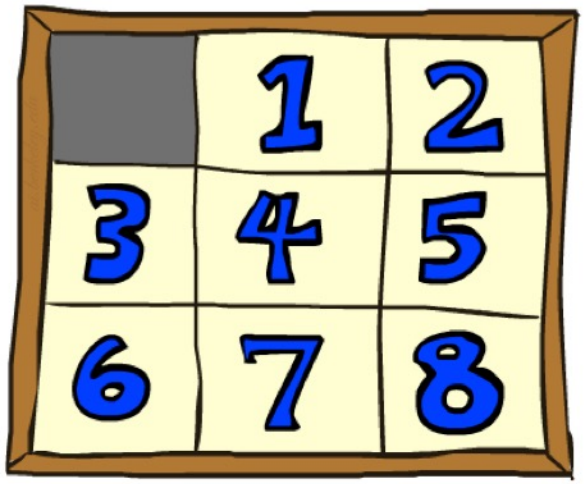
Example: 8 Puzzle



Start State



Actions

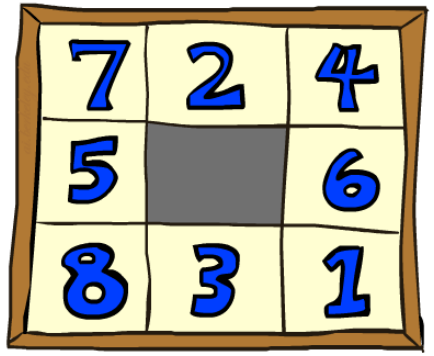


Goal State

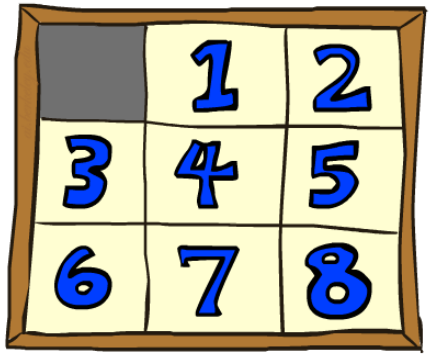
- What are the states?
- How many states?
- What are the actions?
- How many successors from the start state?
- What should the costs be?

8 Puzzle I

- Heuristic: number of tiles misplaced
- Why is it admissible?
- h(start) = 8
- This is a *relaxed-problem* heuristic.



Start State



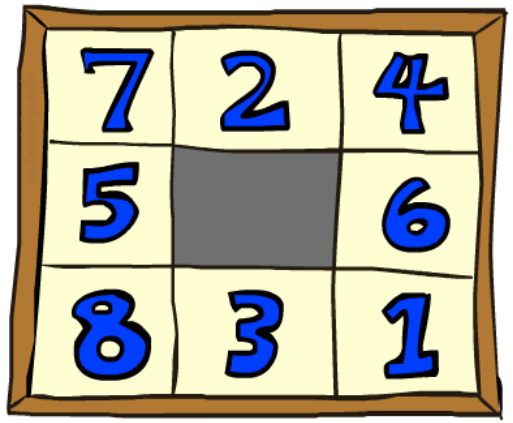
Goal State

	Average nodes expanded when the optimal path has...		
	...4 steps	...8 steps	...12 steps
UCS	112	6,300	3.6×10^6
A* (tiles)	13	39	227

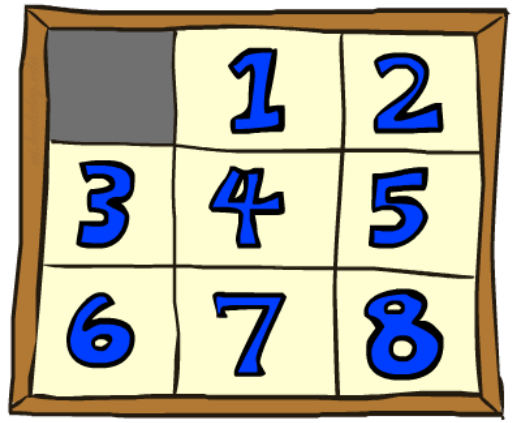
Statistics from Andrew Moore

8 Puzzle II

- What if we had an easier 8-puzzle where any tile could slide any direction at any time, ignoring other tiles?
- Total *Manhattan* distance
- Why is it admissible?
- h(start) = 3 + 1 + 2 + … = 18



Start State



Goal State

	Average nodes expanded when the optimal path has…		
	…4 steps	…8 steps	…12 steps
A* (tiles)	13	39	227
A* (Manhattan)	12	25	73

8 Puzzle III

- How about using the *actual cost* as a heuristic?
  - Would it be admissible?
  - Would we save on nodes expanded?
  - What's wrong with it?

- With A*: a trade-off between quality of estimate and work per node
  - As heuristics get closer to the true cost, you will expand fewer nodes but usually do more work per node to compute the heuristic itself

Trivial heuristics, dominance

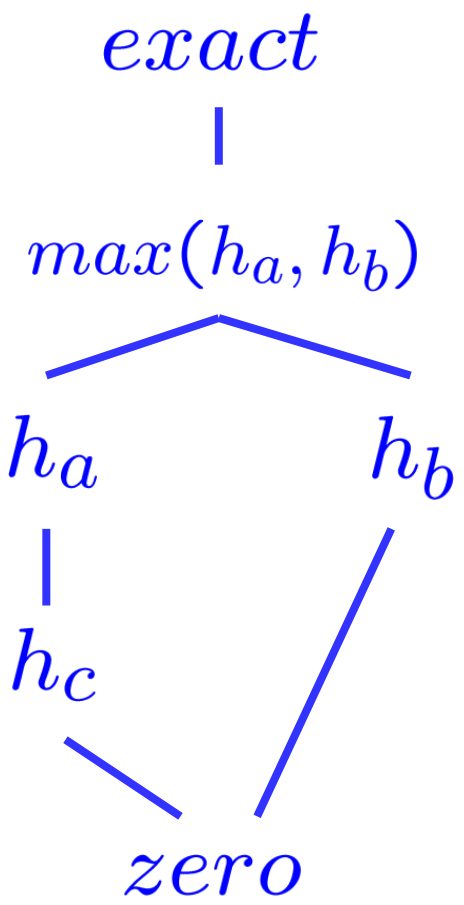
- Dominance: $h_a \geq h_c$ if

$$\forall n : h_a(n) \geq h_c(n)$$

- Heuristics form a semi-lattice:
  - Max of admissible heuristics is admissible

$$h(n) = max(h_a(n), h_b(n))$$

- Trivial heuristics
  - Bottom of lattice is the zero heuristic (what does this give us?)
  - Top of lattice is the exact heuristic



A* applications

- Video games
- Pathing / routing problems
- Resource planning problems
- Robot motion planning
- Language analysis
- Machine translation
- Speech recognition
- …


DFS
BFS
Dijkstra
A*

Memory-bounded heuristic search

- Iterative Deepening A* (IDA*)
  - Basically, depth-first search but using the f to decide which order to consider nodes
  - Use f-limit instead of depth limit
    - New f-limit is the smallest f-value value of any node that exceeded cutoff on previous iteration
  - Additionally keep track of next limit to consider
  - IDA* has same properties as A* but uses less memory

Memory-bounded heuristic search

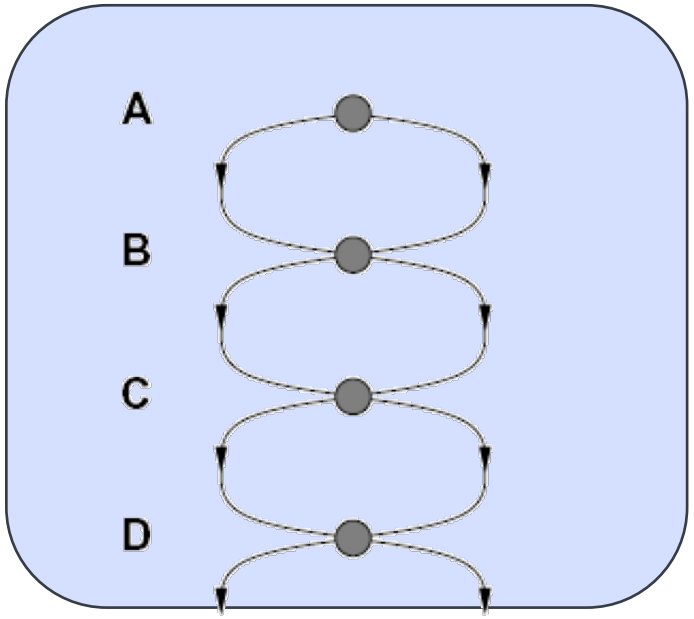
- Simplified Memory-Bounded A* (SMA*)
  - Uses all available memory
  - Proceeds like A* but when it runs out of memory it drops the worst leaf node (one with highest f value)
  - If all leaf nodes have same f-value, drop oldest and expand newest
  - Optimal and complete if depth of shallowest goal node is less than memory size

Graph Search

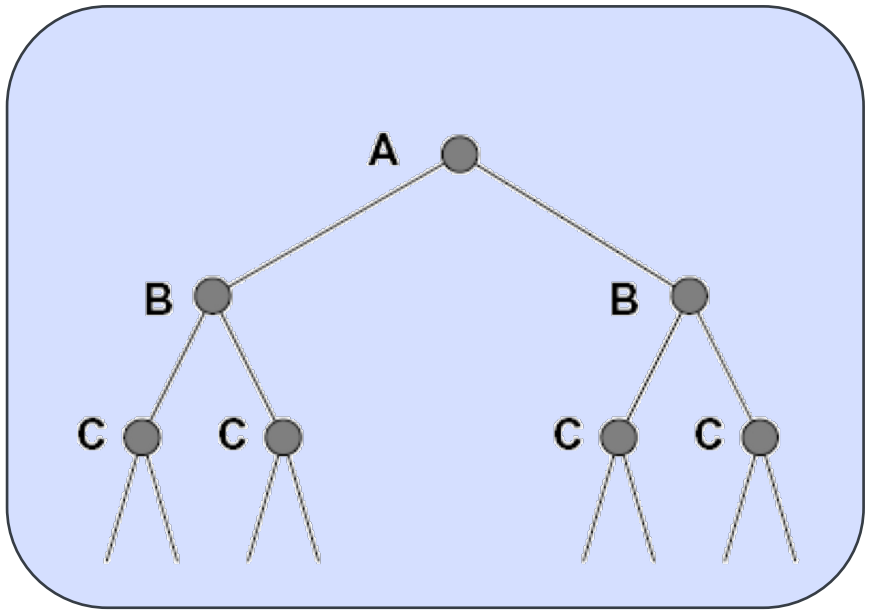
Tree search: extra work!

- Failure to detect repeated states can cause exponentially more work.

State Graph

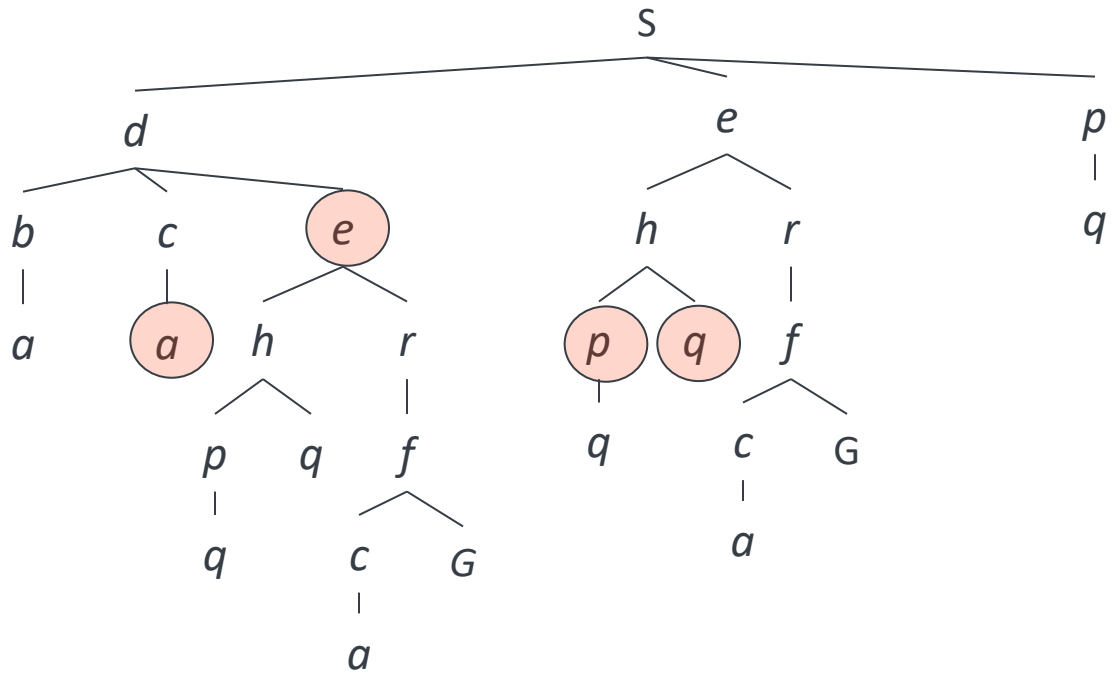


Search Tree



Graph search

- In BFS, for example, we shouldn't bother expanding the circled nodes (why?)

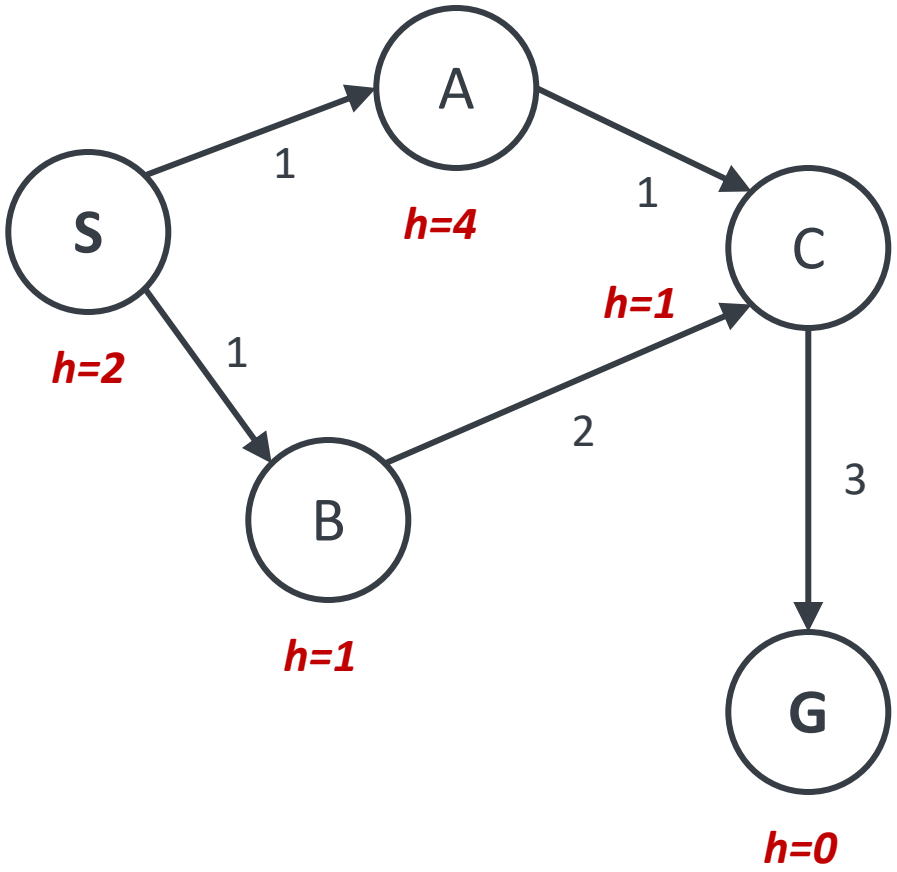


Graph search

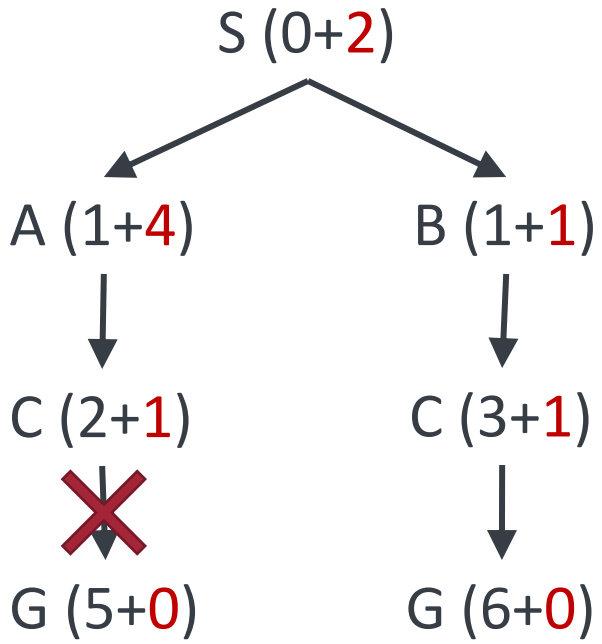
- Idea: never expand a state twice
- How to implement:
  - Tree search + set of expanded states (“closed set”)
  - Expand the search tree node-by-node, but…
  - Before expanding a node, check to make sure its state has never been expanded before
  - If not new, skip it, if new add to closed set
- Important: store the closed set as a set, not a list
- Can graph search wreck completeness? Why/why not?
- How about optimality?

A* graph search gone wrong?

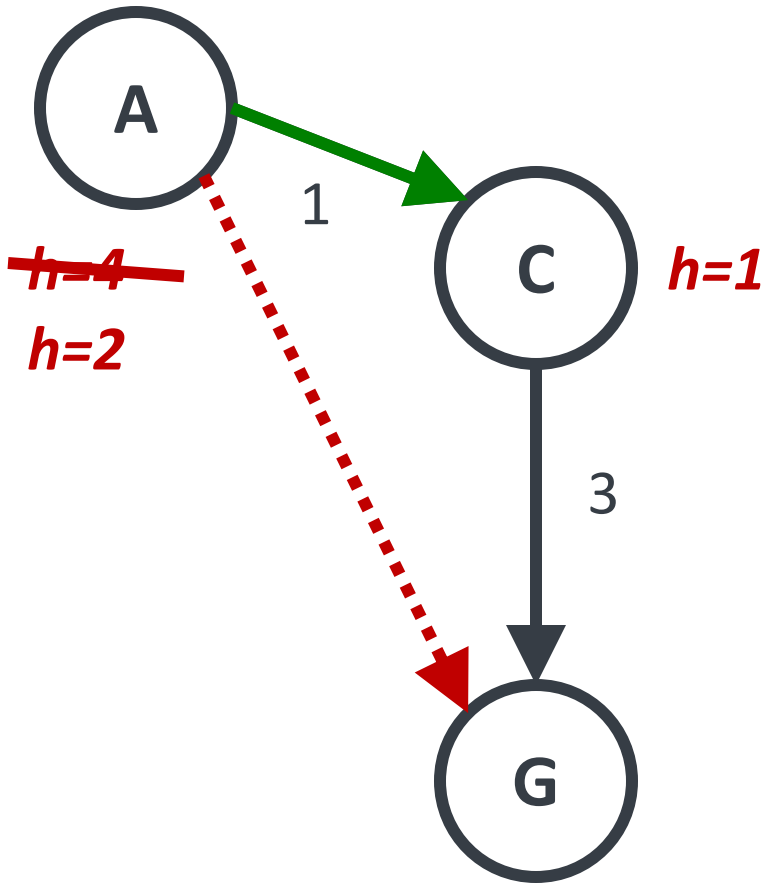
State space graph



Search tree



Consistency of heuristics



- Main idea: estimated heuristic costs ≤ actual costs
  - Admissibility: heuristic cost ≤ actual cost to goal

    h(A) ≤ actual cost from A to G
  - Consistency: heuristic "arc" cost ≤ actual cost for each arc

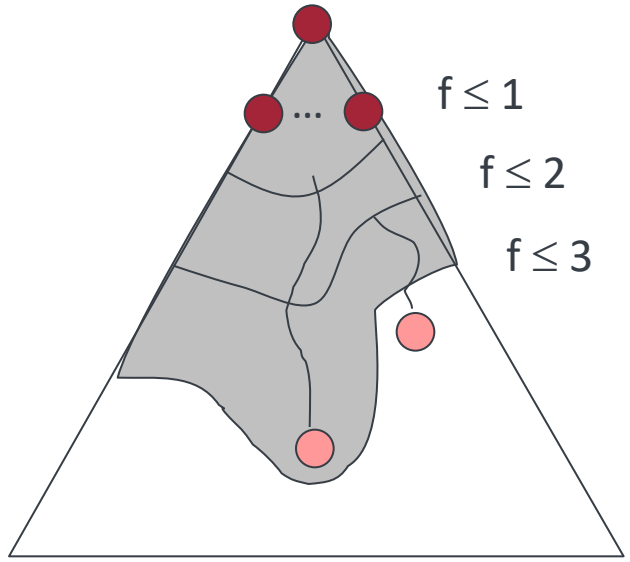
    h(A) – h(C) ≤ cost(A to C)

- Consequences of consistency:
  - The f value along a path never decreases. Proof?

    h(A) ≤ cost(A to C) + h(C)
  - A* graph search is optimal.

Optimality of A* Graph Search

- Sketch: consider what A* does with a consistent heuristic:
  - Fact 1: In tree search, A* expands nodes in increasing total f value (f-contours)
  - Fact 2: For every state s, nodes that reach s optimally are expanded before nodes that reach s suboptimally
  - Result: A* graph search is optimal.



Consistent heuristics are admissible

- If $h(n) \leq c(n,a,n2) + h(n2)$, $h(n) \leq h^*(n)$
- Proof
  - **If** no path exists from n to a goal then $h^*(n) = \infty$ and $h(n) \leq h^*(n)$
  - **Else** let <sn,sn+1,…,g> be an **optimal** path from n to a goal state g.

    (Note the cost of this path is $h^*(n)$, and each subpath ni =<sn+i,…,g> has cost equal to $h^*(ni)$.)
- Prove $h(n) \leq h^*(n)$ by induction on the length of this optimal path.
  - **Base Case: n = g**

  By our conditions on h, h(n) and h(n)* are equal to zero so $h(n) \leq h(n)^*$
  - **Induction Hypothesis: $h(n1) \leq h^*(n1)$**

  $h(n) \leq c(n,a1,n1) + h(n1) \leq c(n,a1,n1) + h^*(n1) = h^*(n)$

Optimality

- Tree search:
  - A* is optimal if heuristic is admissible
  - UCS is a special case (h = 0)
- Graph search:
  - A* optimal if heuristic is consistent
  - UCS optimal (h = 0 is consistent)
- Consistency implies admissibility.
- In general, most natural admissible heuristics tend to be consistent, especially if from relaxed problems.

A*: Summary

- A* uses both backward costs and (estimates of) forward costs
- A* is optimal with admissible / consistent heuristics
- Heuristic design is key: often use relaxed problems

Tree Search Pseudo-Code

```
function TREE-SEARCH(problem, fringe) return a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        for child-node in EXPAND(STATE[node], problem) do
            fringe ← INSERT(child-node, fringe)
        end
    end
```

Graph Search Pseudo-Code

```
function GRAPH-SEARCH(problem, fringe) return a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            for child-node in EXPAND(STATE[node], problem) do
                fringe ← INSERT(child-node, fringe)
            end
    end
```

STEVENS
INSTITUTE OF TECHNOLOGY
1870

15 min. break

STEVENS
INSTITUTE OF TECHNOLOGY
1870

Constraint Satisfaction

Chapter 6

Backtracking

- Forward checking
- Constraint propagation
- Ordering

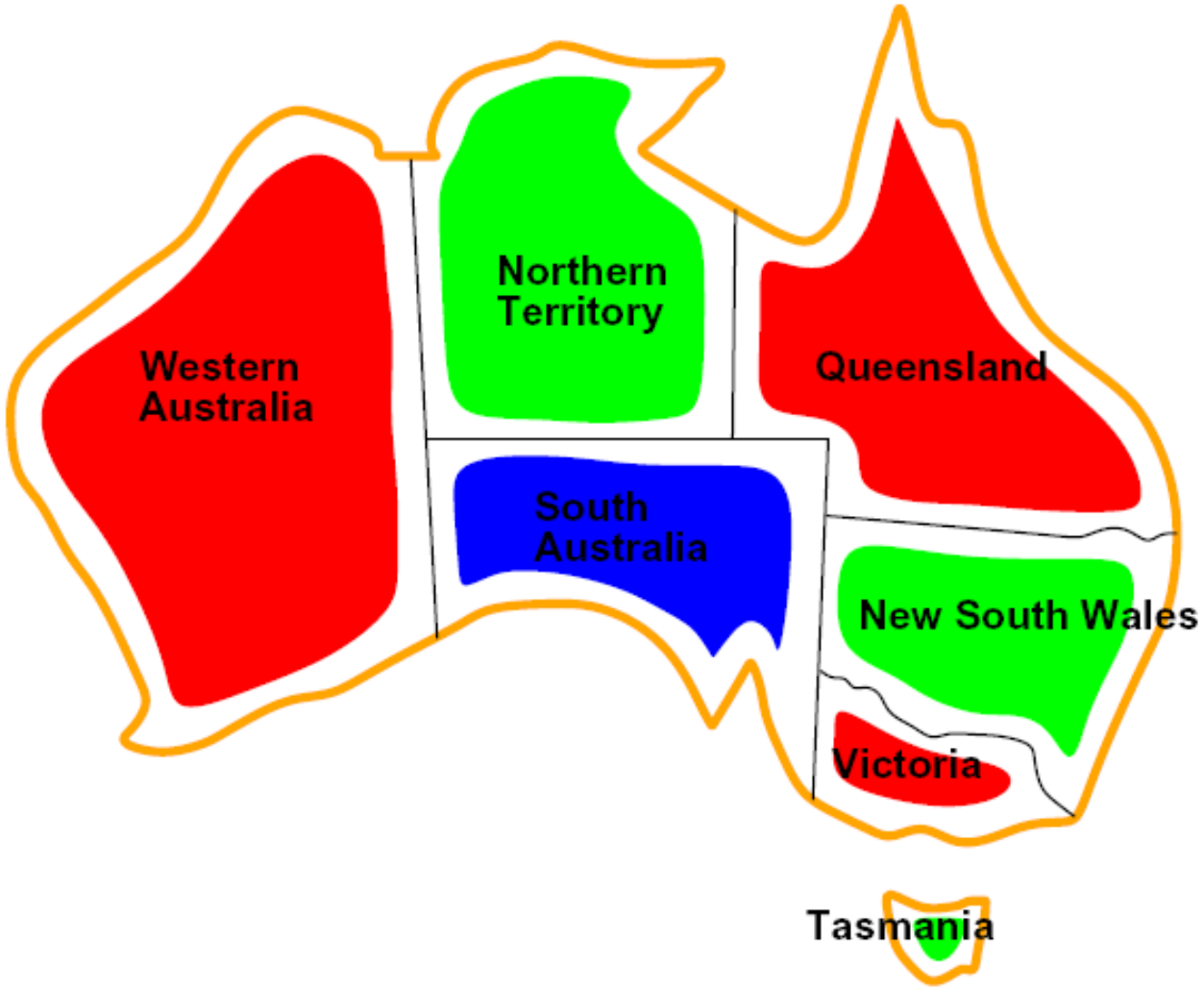
Tree-structured CSP

Constraint satisfaction problems

- Constraint satisfaction problems (CSPs):
  - A special subset of search problems
  - State is defined by variables X_i with values from a domain D (sometimes D depends on i)
  - Goal test is a set of constraints specifying allowable combinations of values for subsets of variables

- Simple example of a *formal representation language*
- Allows useful general-purpose algorithms with more power than standard search algorithms

CSP examples

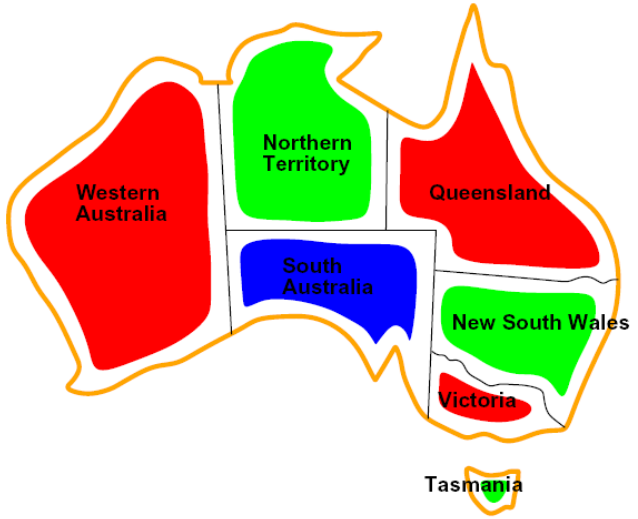


Example: map coloring

- Variables: WA, NT, Q, NSW, V, SA, T
- Domains: $D = \{red, green, blue\}$
- Constraints: adjacent regions must have different colors

  Implicit: $WA \neq NT$

  Explicit: $(WA, NT) \in \{(red, green), (red, blue), \ldots\}$

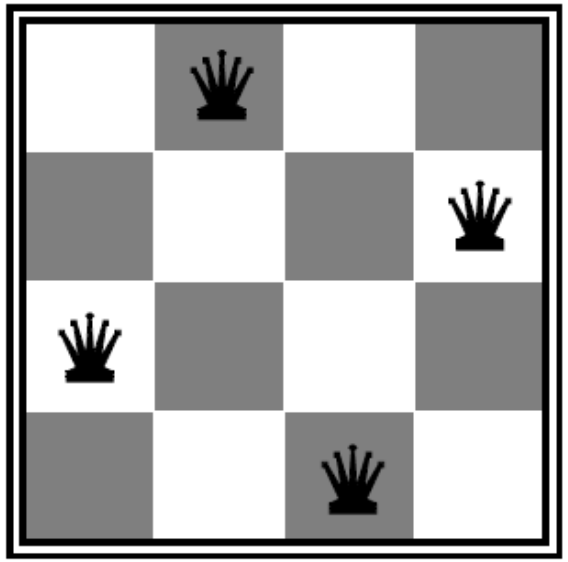


- Solutions are assignments satisfying all constraints, e.g.:

  {WA=red, NT=green, Q=red, NSW=green, V=red, SA=blue, T=green}

Example: N-Queens

- Formulation 1:
  - Variables: X_{ij}
  - Domains: $\{0, 1\}$
  - Constraints

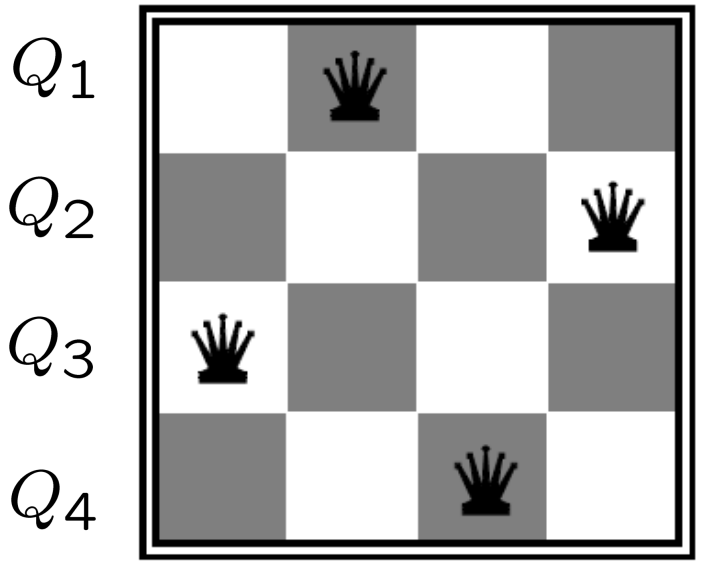


$$\forall i, j, k \quad (X_{ij}, X_{ik}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \quad (X_{ij}, X_{kj}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \quad (X_{ij}, X_{i+k,j+k}) \in \{(0,0), (0,1), (1,0)\}$$
$$\forall i, j, k \quad (X_{ij}, X_{i+k,j-k}) \in \{(0,0), (0,1), (1,0)\}$$

$$\sum_{i,j} X_{ij} = N$$

Example: N-Queens

- Formulation 2:
  - Variables: Q_k
  - Domains: $\{1, 2, 3, \ldots N\}$
  - Constraints:

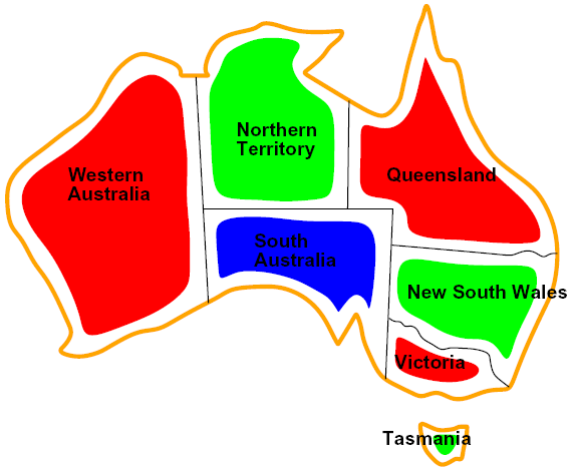


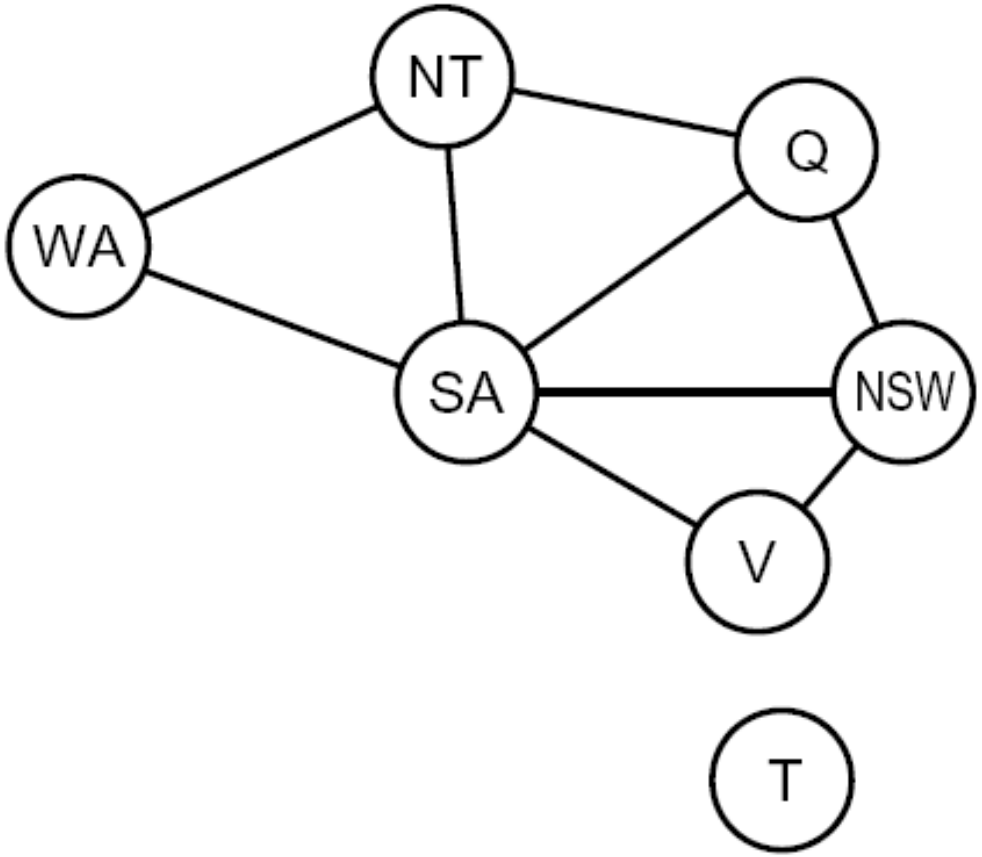
Implicit: $\forall i, j$ non-threatening(Q_i, Q_j)

Explicit: $(Q_1, Q_2) \in \{(1,3), (1,4), \ldots\}$

$\ldots$

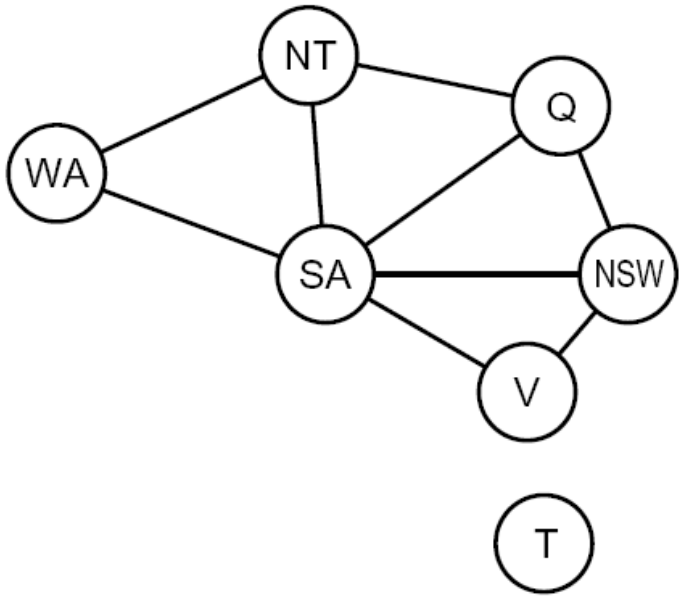
Constraint graphs





Constraint graphs

- Binary CSP: each constraint relates (at most) two variables
- Binary constraint graph: nodes are variables, arcs show constraints
- General-purpose CSP algorithms use the graph structure to speed up search (*e.g.*, Tasmania is an independent subproblem).



Example: Cryptarithmetic

- Variables:

  $F\ T\ U\ W\ R\ O\ X_1\ X_2\ X_3$

- Domains:

  $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

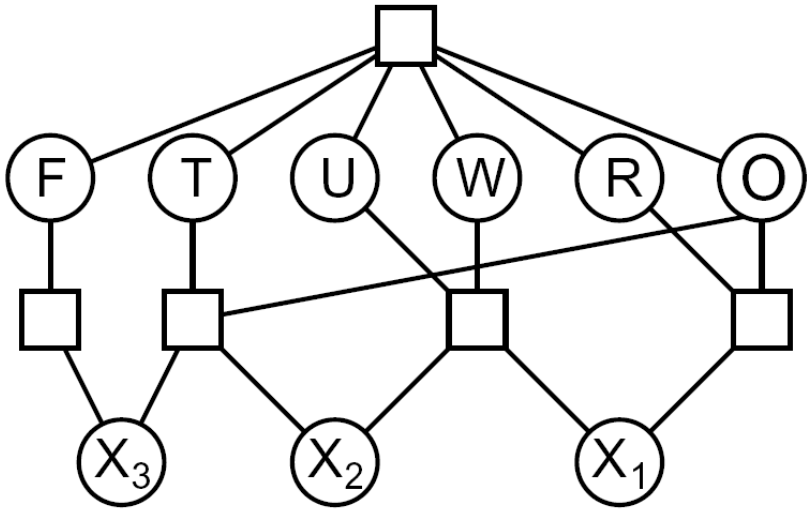
- Constraints:

  alldiff(F, T, U, W, R, O)

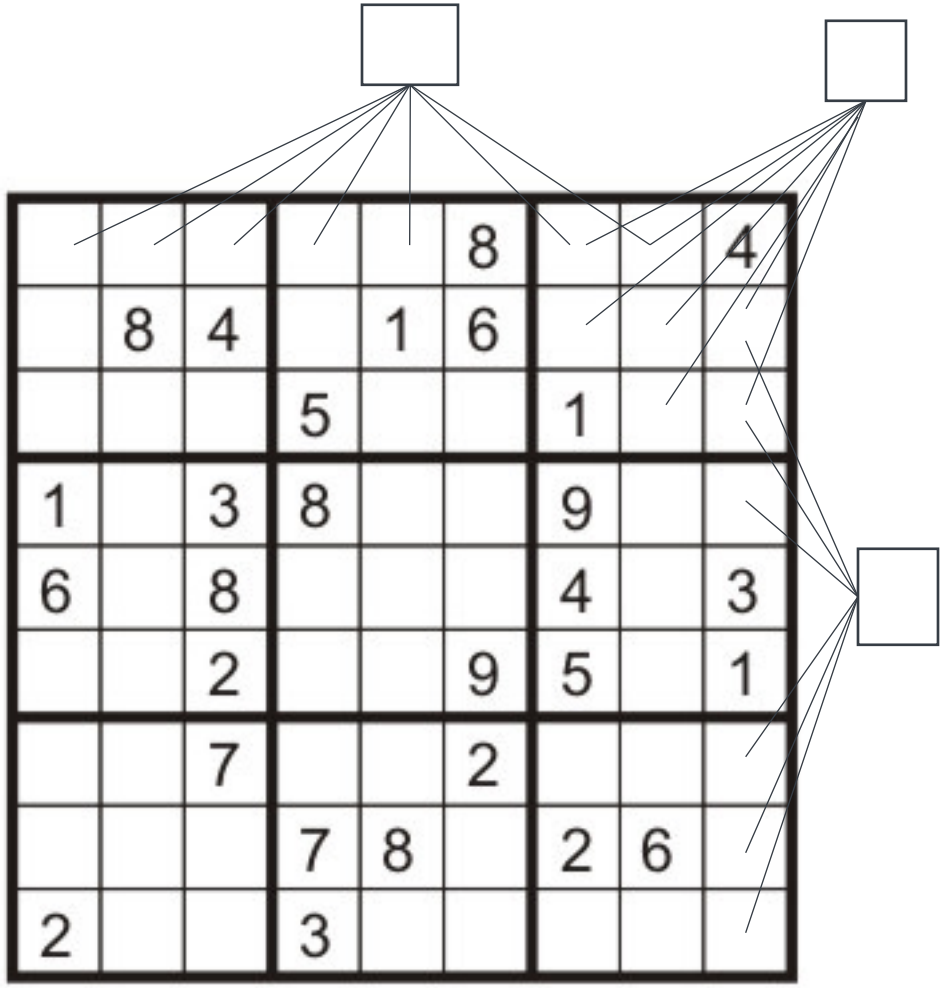
  $O + O = R + 10 \cdot X_1$

  $\cdots$

$$\begin{array}{r} \text{T W O} \\ + \text{T W O} \\ \hline \text{F O U R} \end{array}$$



Example: Sudoku



- Variables:
  - Each (open) square
- Domains:
  - {1,2,…,9}
- Constraints:

  9-way alldiff for each column

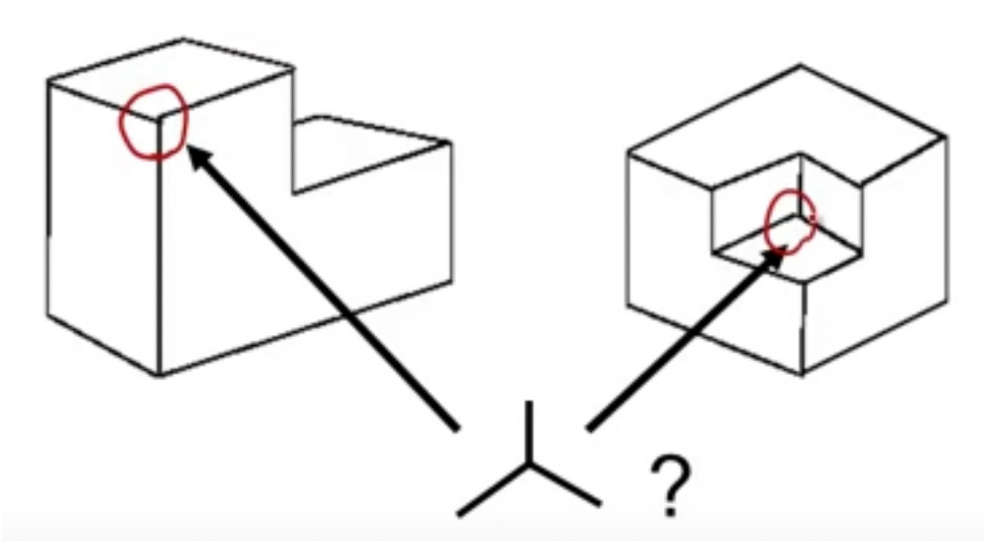
  9-way alldiff for each row

  9-way alldiff for each region

  (or can have a bunch of pairwise inequality constraints)

Example: The Waltz Algorithm

- The Waltz algorithm is for interpreting line drawings of solid polyhedra as 3D objects
- An early example of an AI computation posed as a CSP



- Approach:
  - Each intersection is a variable
  - Adjacent intersections impose constraints on each other
  - Solutions are physically realizable 3D interpretations

Varieties of CSPs

- Discrete Variables
  - Finite domains
    - Size d means $O(d^n)$ complete assignments
    - *E.g.*, Boolean CSPs, including Boolean satisfiability (NP-complete)
  - Infinite domains (integers, strings, etc.)
    - *E.g.*, job scheduling, variables are start/end times for each job
    - Linear constraints solvable, nonlinear undecidable

- Continuous variables
  - *E.g.*, start/end times for Hubble Telescope observations
  - Linear constraints solvable in polynomial time by Linear Programming methods.

Varieties of Constraints

- Varieties of Constraints
  - Unary constraints involve a single variable (equivalent to reducing domains), *e.g.*:

    $$SA \neq green$$

  - Binary constraints involve pairs of variables, e.g.:

    $$SA \neq WA$$

  - Higher-order constraints involve 3 or more variables:
    e.g., cryptarithmetic column constraints

- Preferences (soft constraints):
  - *E.g.*, red is better than green
  - Often representable by a cost for each variable assignment
  - Gives constrained optimization problems
  - (We'll ignore these now)

Real-World CSPs

- Assignment problems: e.g., who teaches what class
- Timetabling problems: e.g., which class is offered when and where?
- Hardware configuration
- Transportation scheduling
- Factory scheduling
- Circuit layout
- Fault diagnosis
- … lots more!

- Many real-world problems involve real-valued variables…

Solving CSPs

Standard search formulation

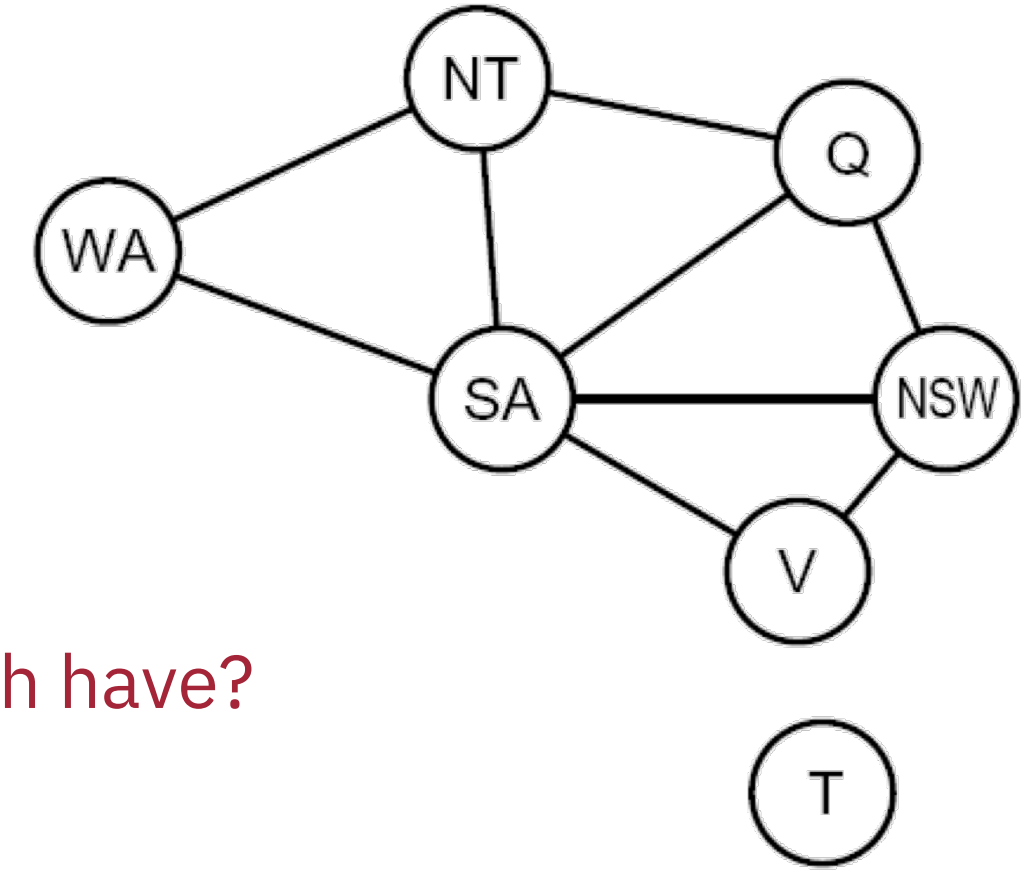
- Standard search formulation of CSPs

- States defined by the values assigned so far (partial assignments)
  - Initial state: the empty assignment, {}
  - Successor function: assign a value to an unassigned variable
  - Goal test: the current assignment is complete and satisfies all constraints

- We'll start with the straightforward, naïve approach, then improve it.

Search methods

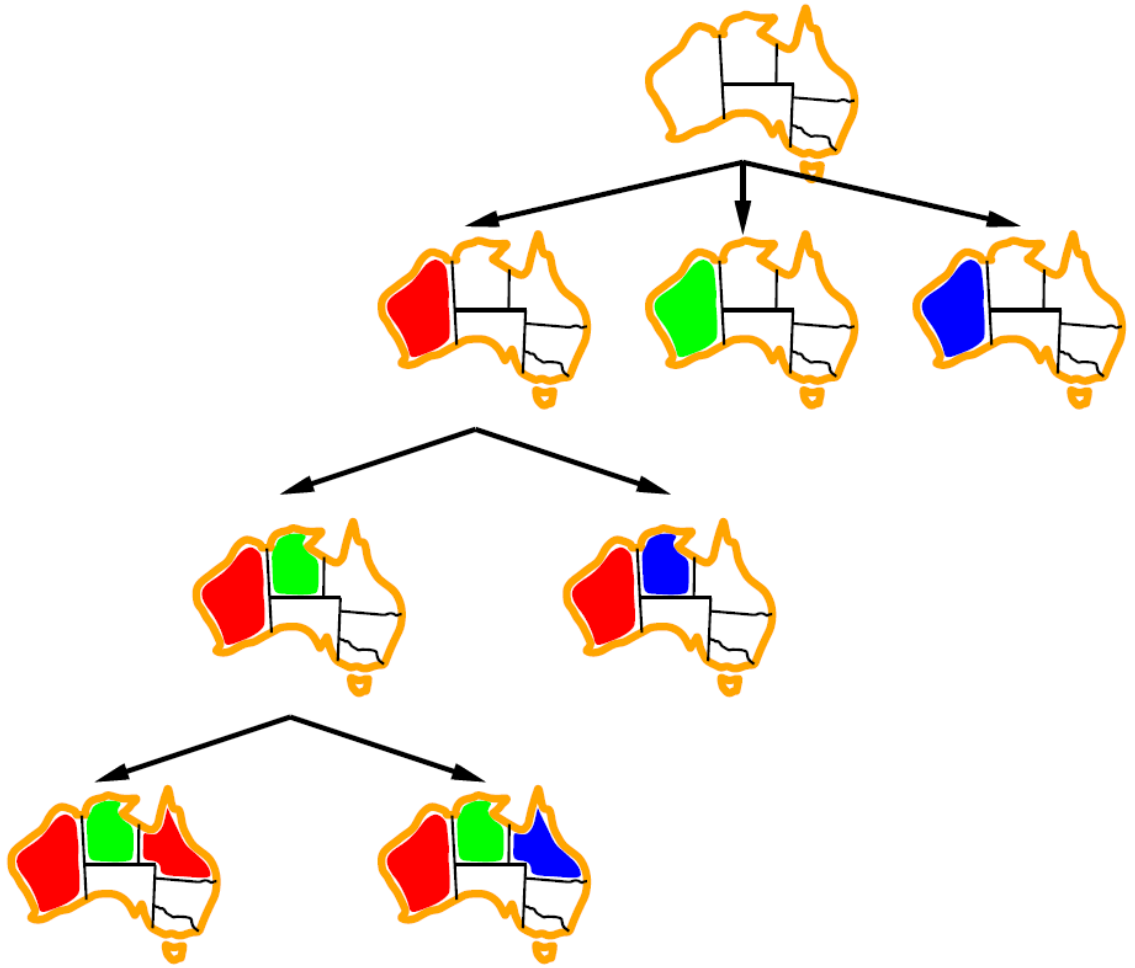
- What would BFS do?
- What would DFS do?
- What problems does naïve search have?



Backtracking Search

- Backtracking search is the basic uninformed algorithm for solving CSPs.
- Idea 1: One variable at a time
  - Variable assignments are commutative, so fix ordering
  - *I.e.*, [WA = red then NT = green] same as [NT = green then WA = red]
  - Only need to consider assignments to a single variable at each step
- Idea 2: Check constraints as you go
  - I.e. consider only values which do not conflict previous assignments
  - Might have to do some computation to check the constraints
  - "Incremental goal test"
- Depth-first search with these two improvements is called *backtracking search* (not the best name).
- Can solve n-queens for n ≈ 25.

Backtracking Example



Backtracking Search

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment given CONSTRAINTS[csp] then
            add {var = value} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
            remove {var = value} from assignment
    return failure
```

- Backtracking = DFS + variable-ordering + fail-on-violation

Can we improve backtracking?

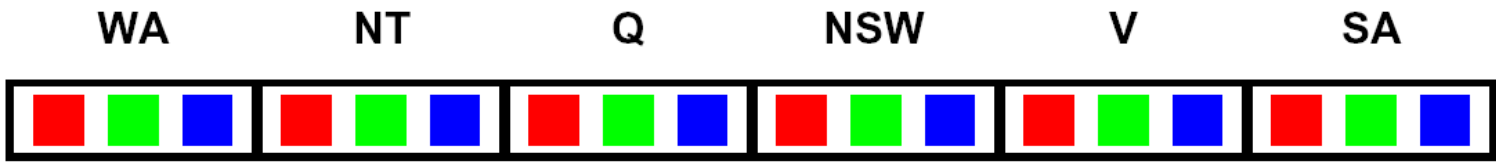
Improving backtracking

- General-purpose ideas give huge gains in speed.
- **Ordering**:
  - Which variable should be assigned next?
  - In what order should its values be tried?
- **Filtering**: Can we detect inevitable failure early?
- **Structure**: Can we exploit the problem structure?

Filtering

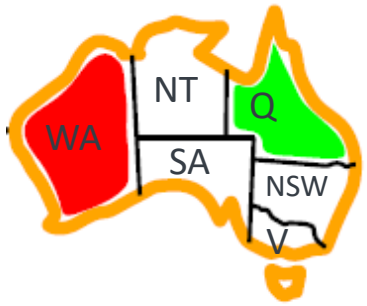
Filtering: forward checking

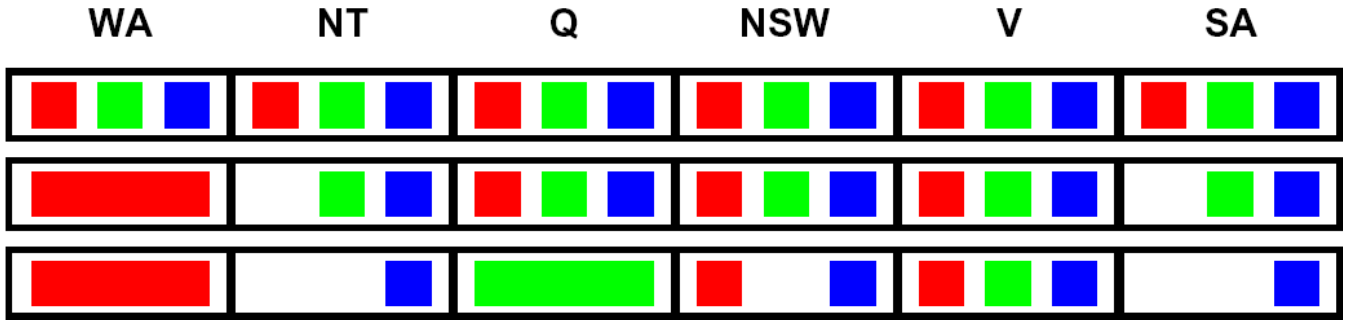
- Filtering: Keep track of domains for unassigned variables and cross off bad options
- Forward checking: Cross off values that violate a constraint when added to the existing assignment



Filtering: constraint propagation

- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:

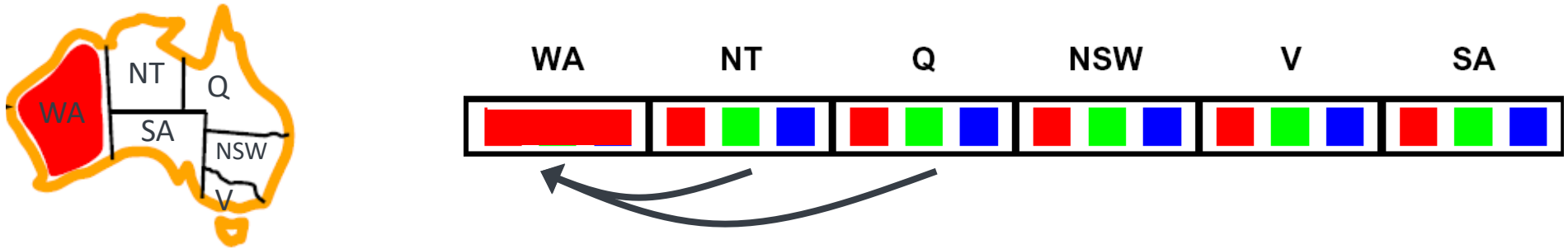




- NT and SA cannot both be blue!
- Why didn’t we detect this yet?
- *Constraint propagation:* reason from constraint to constraint

Consistency of a single arc

- An arc X → Y is **consistent** iff for *every* x in the tail there is *some* y in the head which could be assigned without violating a constraint

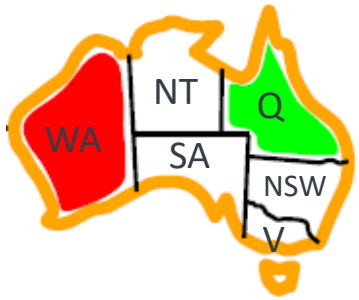


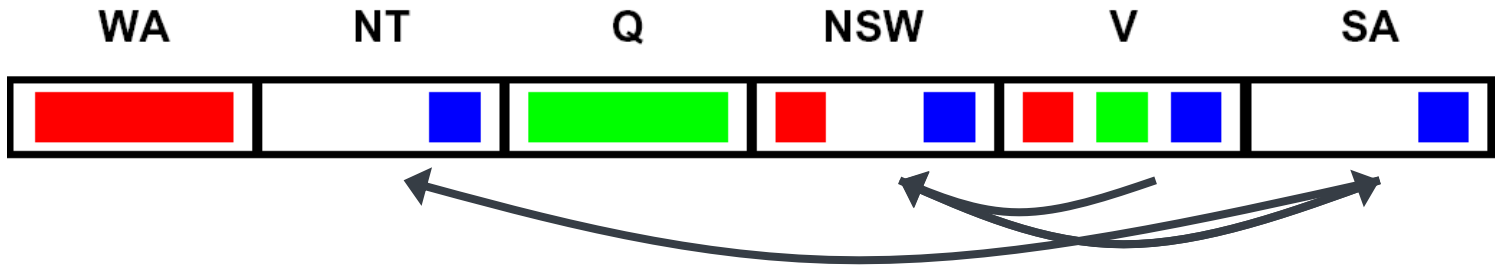
- Forward checking: enforcing consistency of arcs pointing to each new assignment

Remember: Delete from the tail!

Arc consistency of an Entire CSP

- A simple form of propagation makes sure all arcs are consistent:





- Important: If X loses a value, neighbors of X need to be rechecked!
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment
- What's the downside of enforcing arc consistency?

Remember: Delete from the tail!

Enforcing arc consistency in a CSP

```
function AC-3( csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables {X_1, X_2, ..., X_n}
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) ← REMOVE-FIRST(queue)
    if REMOVE-INCONSISTENT-VALUES(X_i, X_j) then
      for each X_k in NEIGHBORS[X_i] do
        add (X_k, X_i) to queue

---------------------------------------------------------------------

function REMOVE-INCONSISTENT-VALUES( X_i, X_j) returns true iff succeeds
  removed ← false
  for each x in DOMAIN[X_i] do
    if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint X_i ↔ X_j
      then delete x from DOMAIN[X_i]; removed ← true
  return removed
```

- Runtime: $O(n^2d^3)$, can be reduced to $O(n^2d^2)$
- … but detecting all possible future problems is NP-hard – why?

Constraint propagation detects all future failures.

True

False

Total Results: 0

Constraint propagation detects all future failures.

True

False

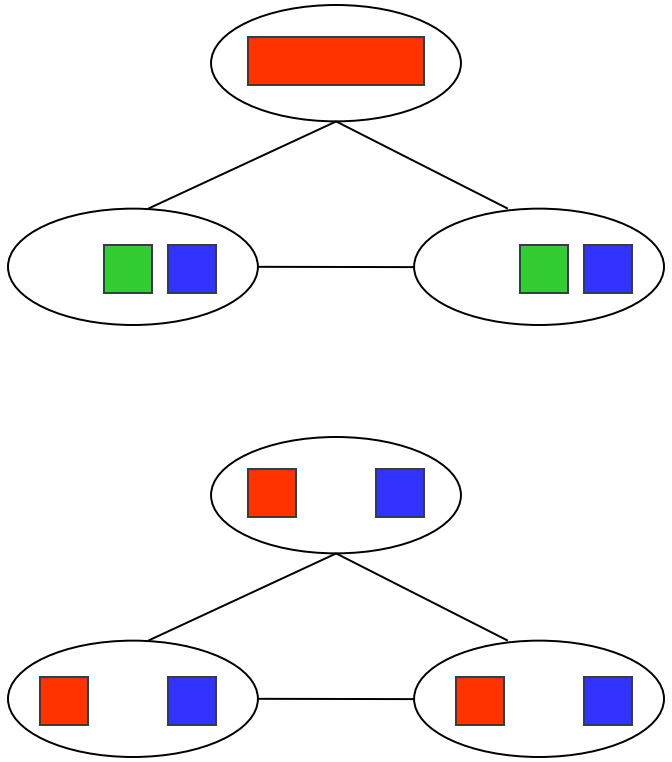
Constraint propagation detects all future failures.

True

False

Limitations of arc consistency

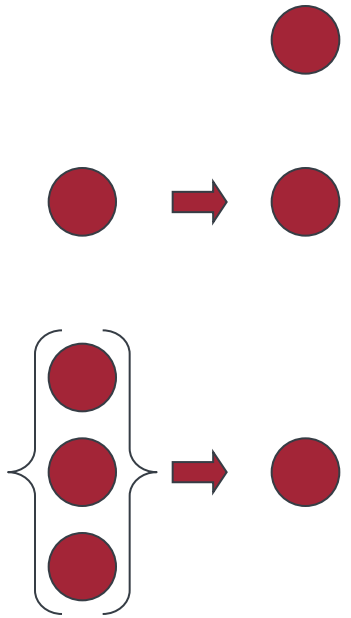
- After enforcing arc consistency:
  - Can have one solution left
  - Can have multiple solutions left
  - Can have no solutions left (and not know it)



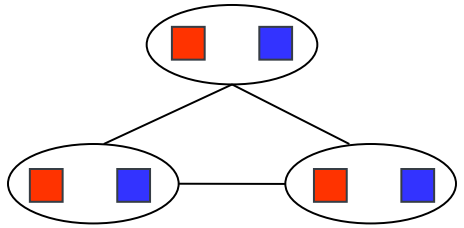
What went wrong here?

K-consistency

- Increasing degrees of consistency
  - 1-Consistency (node consistency): Each single node's domain has a value which meets that node's unary constraints
  - 2-Consistency (arc consistency): For each pair of nodes, any consistent assignment to one can be extended to the other
  - K-Consistency: For each k nodes, any consistent assignment to k-1 can be extended to the k^{th} node.



- Higher k more expensive to compute
- (You need to know the k=2 case: arc consistency)



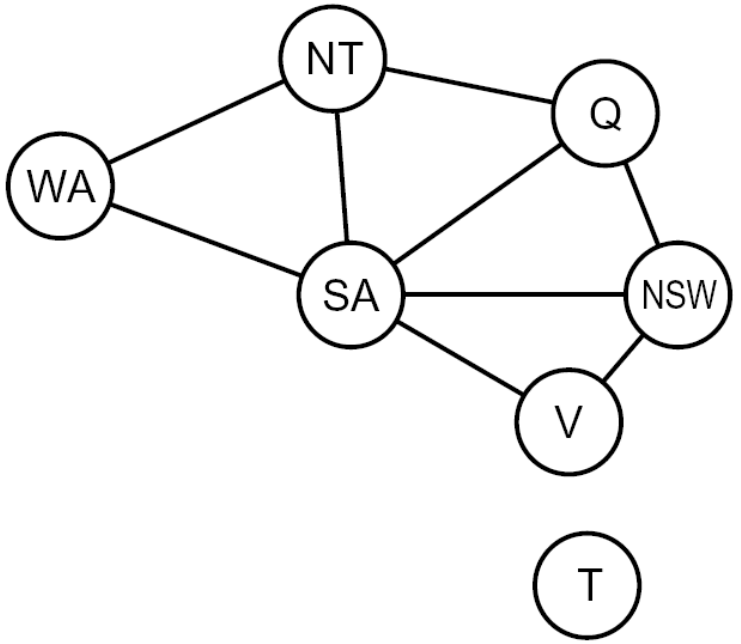
Strong K-consistency

- Strong k-consistency: also k-1, k-2, … 1 consistent
- Claim: strong n-consistency means we can solve without backtracking!
- Why?
  - Choose any assignment to any variable
  - Choose a new variable
  - By 2-consistency, there is a choice consistent with the first
  - Choose a new variable
  - By 3-consistency, there is a choice consistent with the first 2
  - …
- Lots of middle ground between arc consistency and n-consistency! (*e.g.*, k=3, called path consistency)

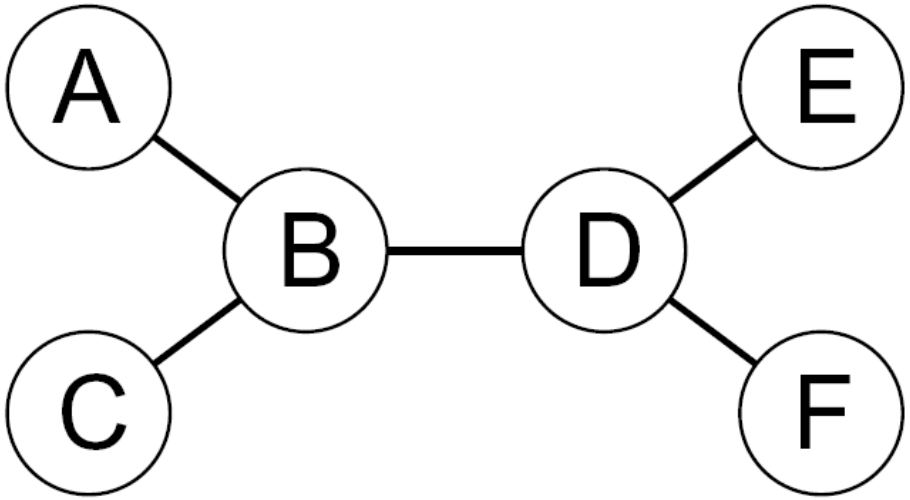
Structure

Problem structure

- Extreme case: independent subproblems
  - Example: Tasmania and mainland do not interact
- Independent subproblems are identifiable as connected components of constraint graph
- Suppose a graph of n variables can be broken into subproblems of only c variables:
  - Worst-case solution cost is $O((n/c)(d^c))$, linear in n
  - E.g., n = 80, d = 2, c =20
  - 2^{80} = 4 billion years at 10 million nodes/sec
  - $(4)(2^{20})$ = 0.4 seconds at 10 million nodes/sec



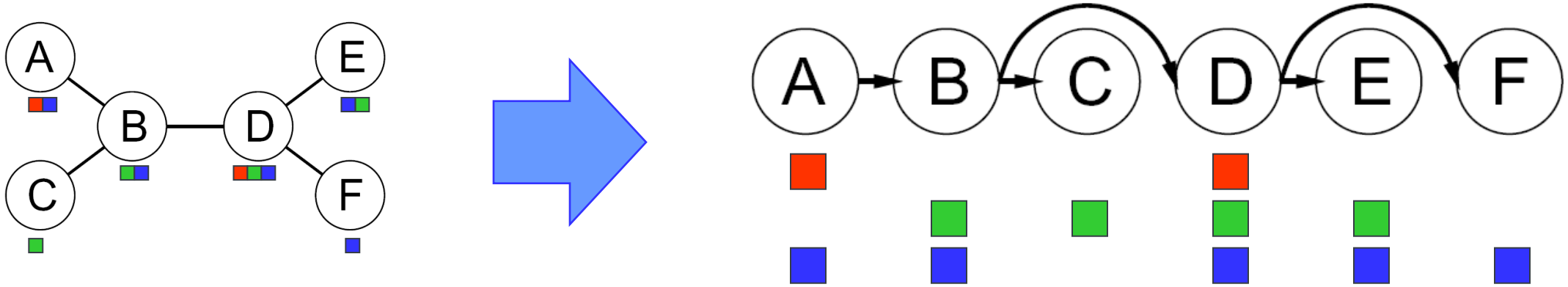
Tree-structured CSPs



- if the constraint graph has no loops, the CSP can be solved in $O(n\ d^2)$ time
  - Compare to general CSPs, where worst-case time is $O(d^n)$

Tree-structured CSPs

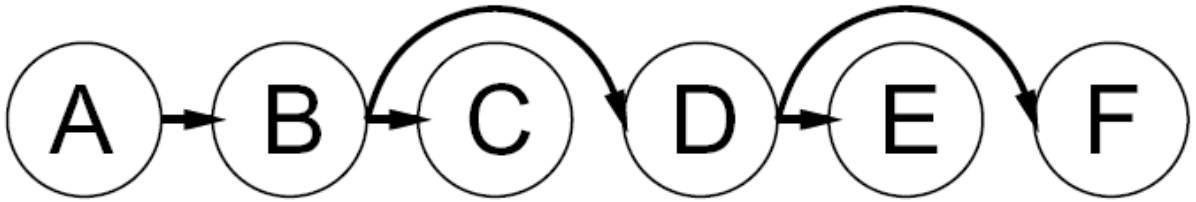
- Algorithm for tree-structured CSPs:
  - Order: Choose a root variable, order variables so that parents precede children



  - Remove backward: For i = n : 2, apply RemoveInconsistent(Parent(X_i),X_i)
  - Assign forward: For i = 1 : n, assign X_i consistently with Parent(X_i)
- Runtime: $O(nd^2)$ (why?)

Tree-Structured CSPs

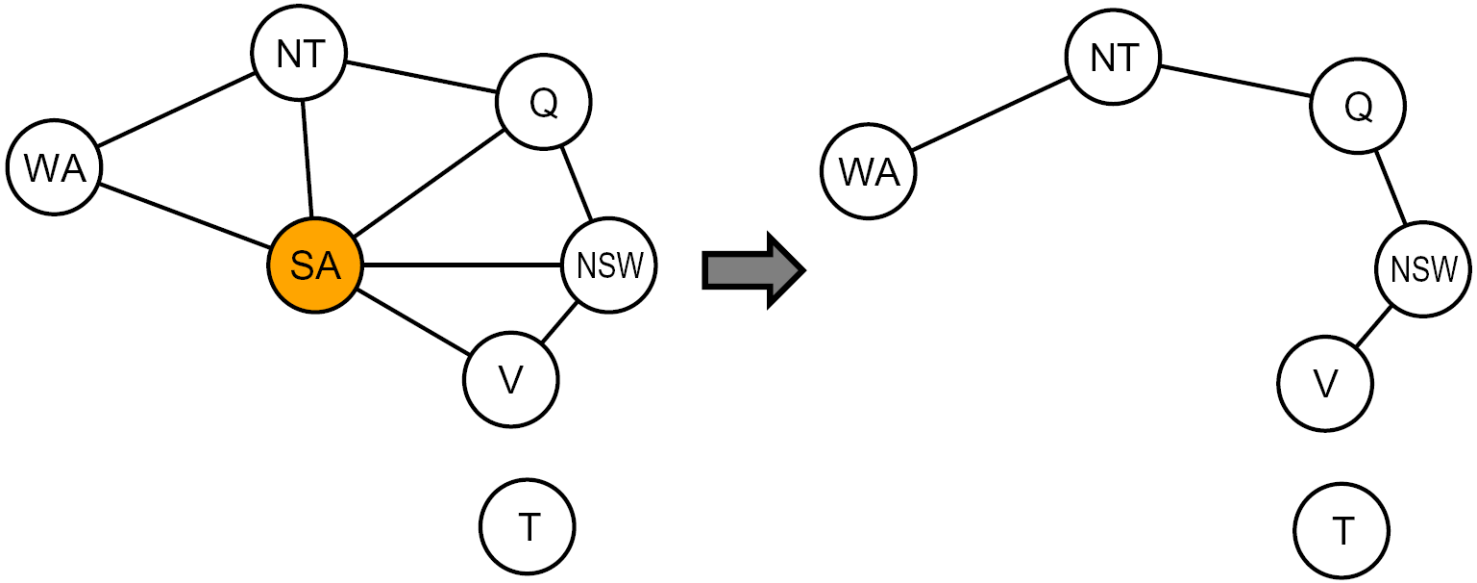
- Claim 1: After backward pass, all root-to-leaf arcs are consistent
- Proof: Each X→Y was made consistent at one point and Y's domain could not have been reduced thereafter (because Y's children were processed before Y)



- Claim 2: If root-to-leaf arcs are consistent, forward assignment will not backtrack
- Proof: Induction on position
- Why doesn't this algorithm work with cycles in the constraint graph?

Improving structure

Nearly tree-structured CSPs



- Conditioning: instantiate a variable, prune its neighbors' domains
- Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree
- Cutset size c gives runtime O((d^c) (n-c) d^2), very fast for small c

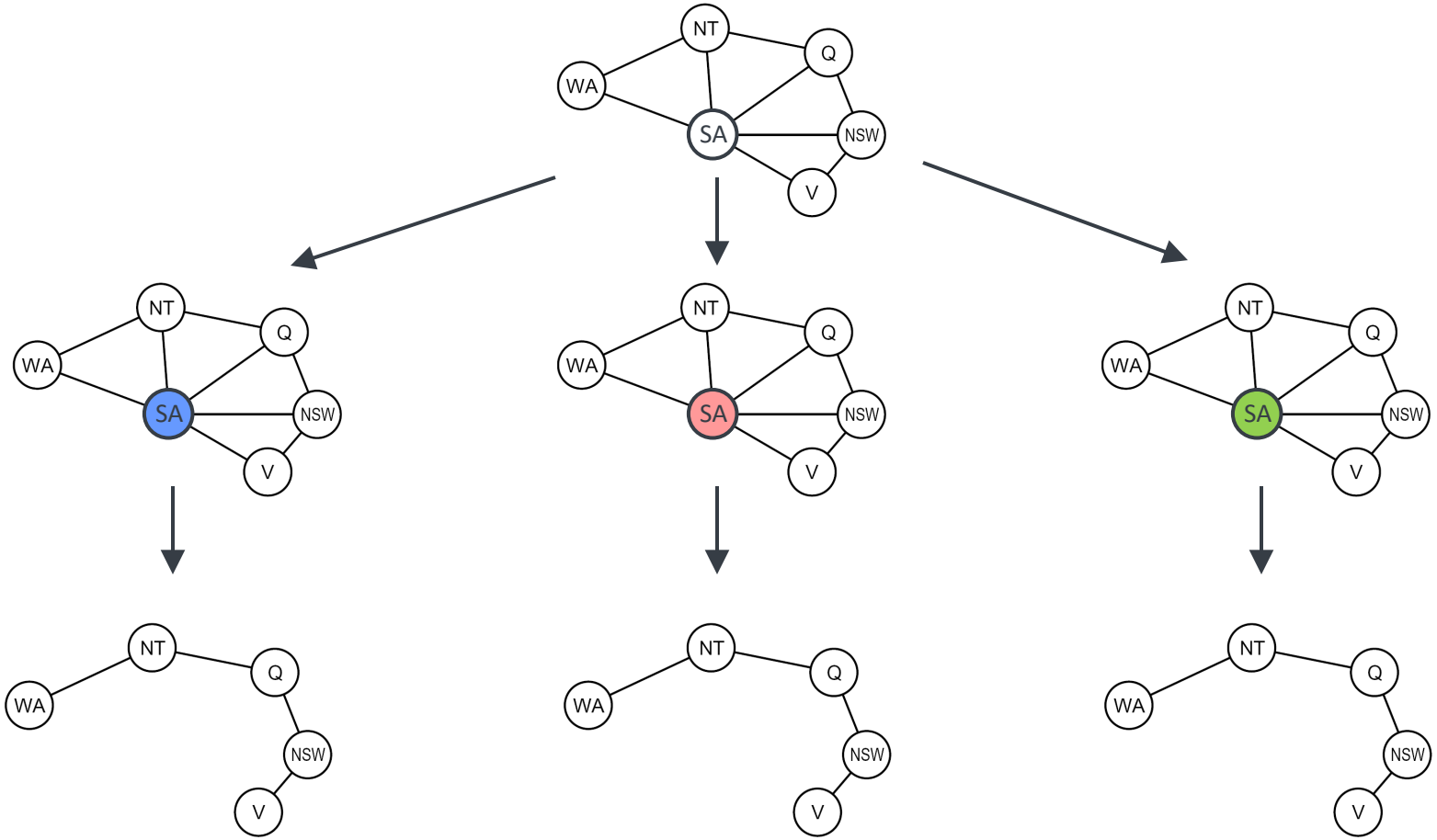
Cutset conditioning

Choose a cutset

Instantiate the cutset (all possible ways)

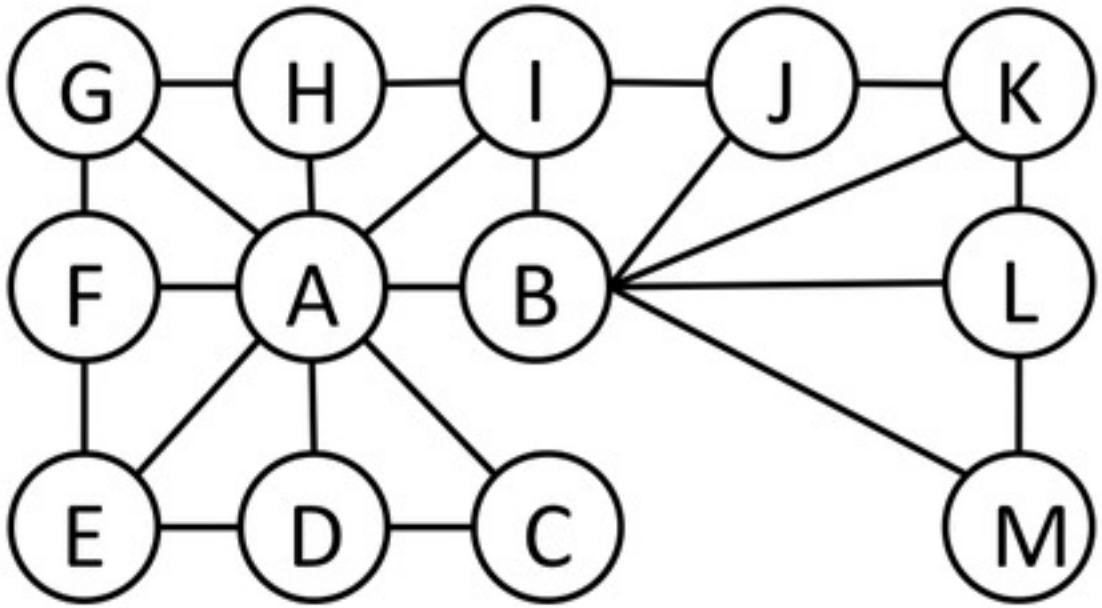
Compute residual CSP for each assignment

Solve the residual CSPs (tree structured)



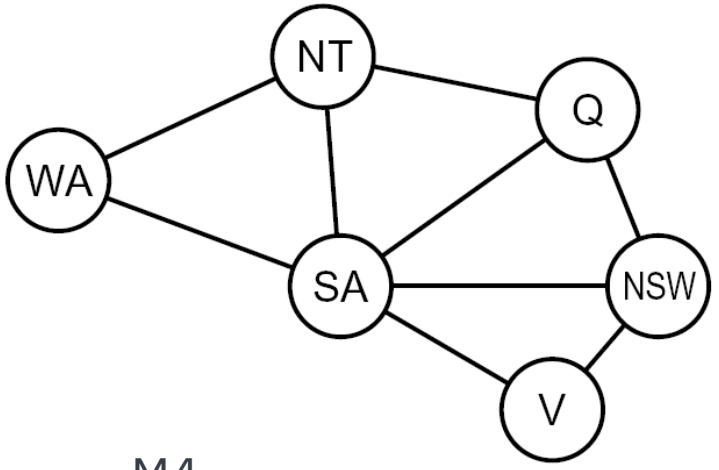
Cutset quiz

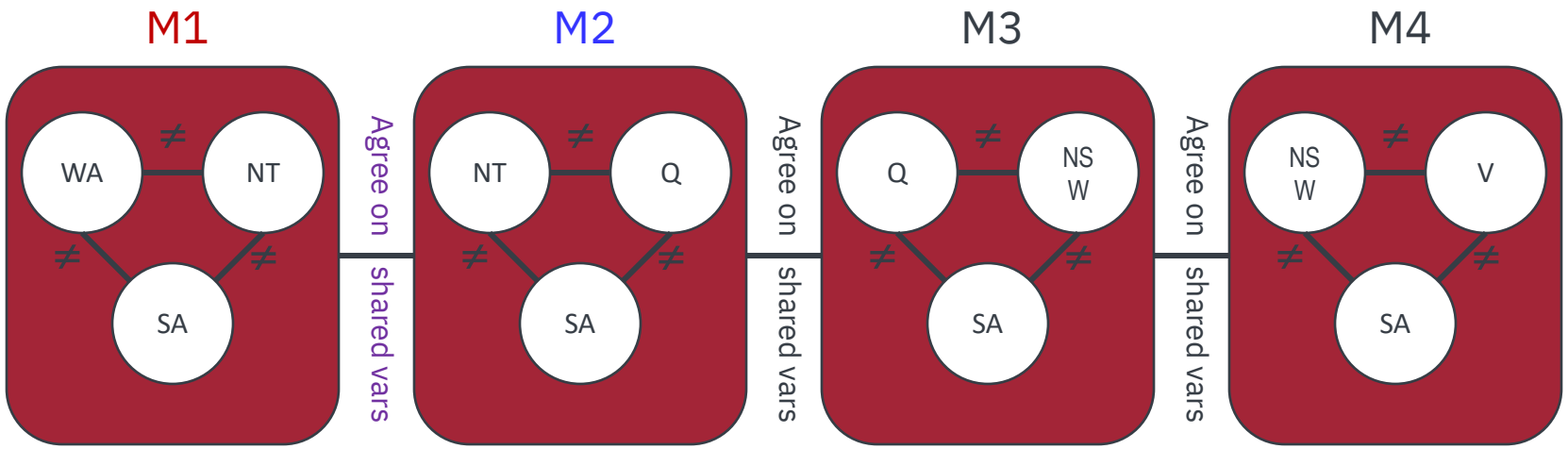
- Find the smallest cutset for the graph below.



Tree decomposition

- Idea: create a tree-structured graph of mega-variables
- Each mega-variable encodes part of the original CSP
- Subproblems overlap to ensure consistent solutions





{(WA=r,SA=g,NT=b),
(WA=b,SA=r,NT=g),
...}

{(NT=r,SA=g,Q=b),
(NT=b,SA=g,Q=r),
...}

Agree: (M1,M2) ∈
{((WA=g,SA=g,NT=g), (NT=g,SA=g,Q=g)), ...}

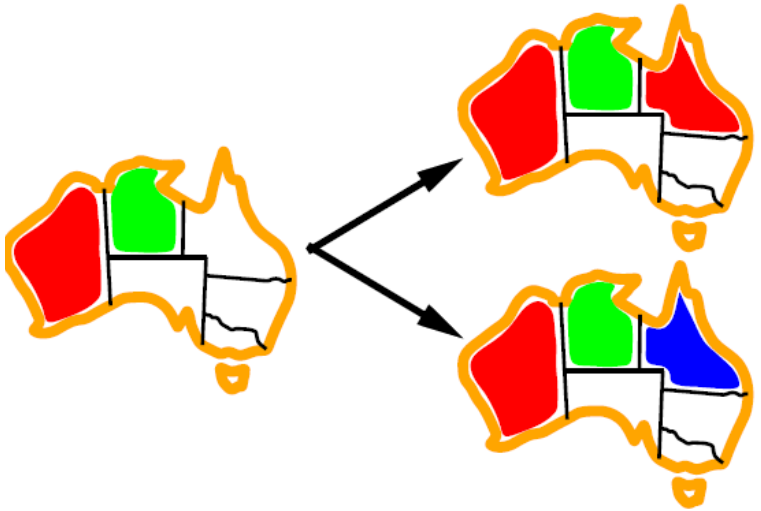
Ordering: minimum remaining values

- Variable ordering: Minimum Remaining Values (MRV):
  - Choose the variable with the fewest legal left values in its domain

- Why min rather than max?
- Also called “most constrained variable”
- “Fail-fast” ordering

Ordering: least constraining value

- Value ordering: Least Constraining Value (LCV)
  - Given a choice of variable, choose the *least constraining value*
  - *I.e.*, the one that rules out the fewest values in the remaining variables
  - Note that it may take some computation to determine this! (*E.g.*, rerunning filtering)



- Why least rather than most?
- Combining these ordering ideas makes 1000 queens feasible

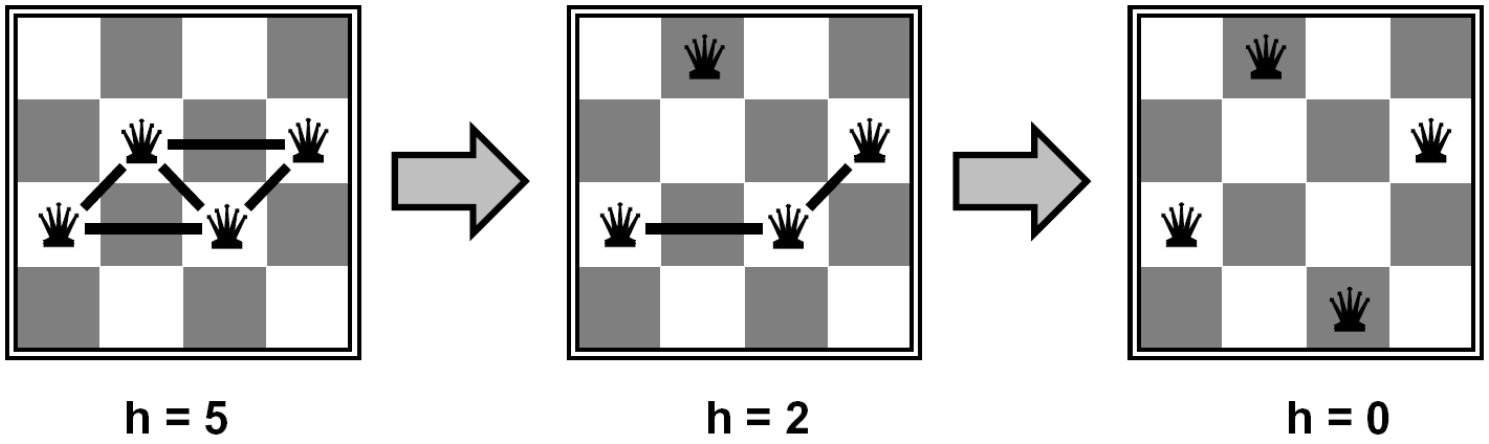
Iterative Improvement

Iterative algorithms for CSPs

- Local search methods typically work with “complete” states, i.e., all variables assigned
- To apply to CSPs:
  - Take an assignment with unsatisfied constraints
  - Operators *reassign* variable values
  - No fringe! Live on the edge.

- Algorithm: While not solved,
  - Variable selection: randomly select any conflicted variable
  - Value selection: min-conflicts heuristic:
    - Choose a value that violates the fewest constraints
    - *I.e.*, hill climb with h(n) = total number of violated constraints

Example: 4-Queens

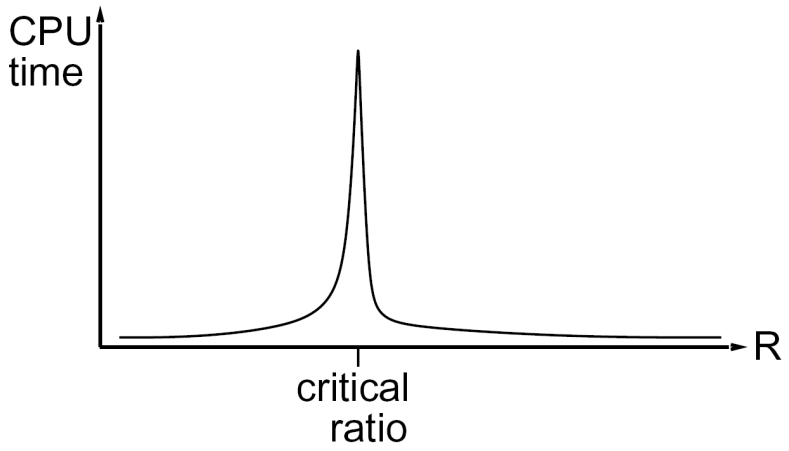


- States: 4 queens in 4 columns ($4^4 = 256$ states)
- Operators: move queen in column
- Goal test: no attacks
- Evaluation: c(n) = number of attacks

Performance of min-conflicts

- Given random initial state, can solve n-queens in almost constant time for arbitrary n with high probability (*e.g.*, n = 10,000,000)!
- The same appears to be true for any randomly-generated CSP *except* in a narrow range of the ratio

$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$



Summary: CSPs

- CSPs are a special kind of search problem:
  - States are partial assignments
  - Goal test defined by constraints
- Basic solution: backtracking search
- Speed-ups:
  - Ordering
  - Filtering
  - Structure
- Iterative min-conflicts is often effective in practice

STEVENS
INSTITUTE OF TECHNOLOGY
1870

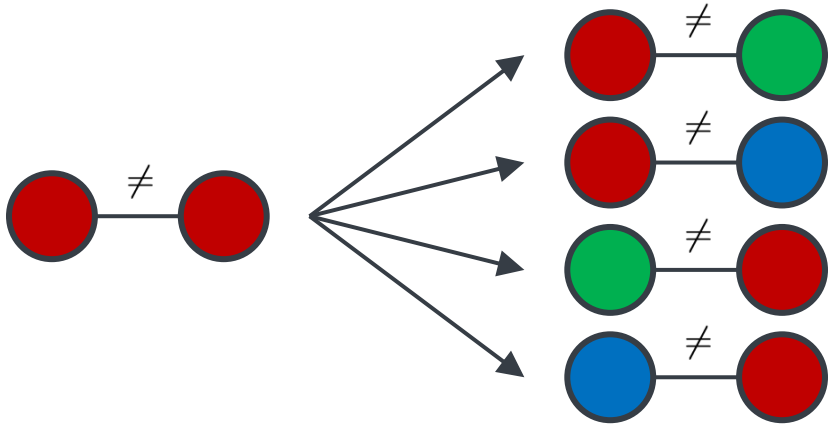
Local Search

Chapter 4

Hill climbing
Simulated annealing
Genetic algorithms

Local search

- Tree search keeps unexplored alternatives on the fringe (ensures completeness)
- Local search: improve a single option until you can't make it better (no fringe!)
- New successor function: local changes

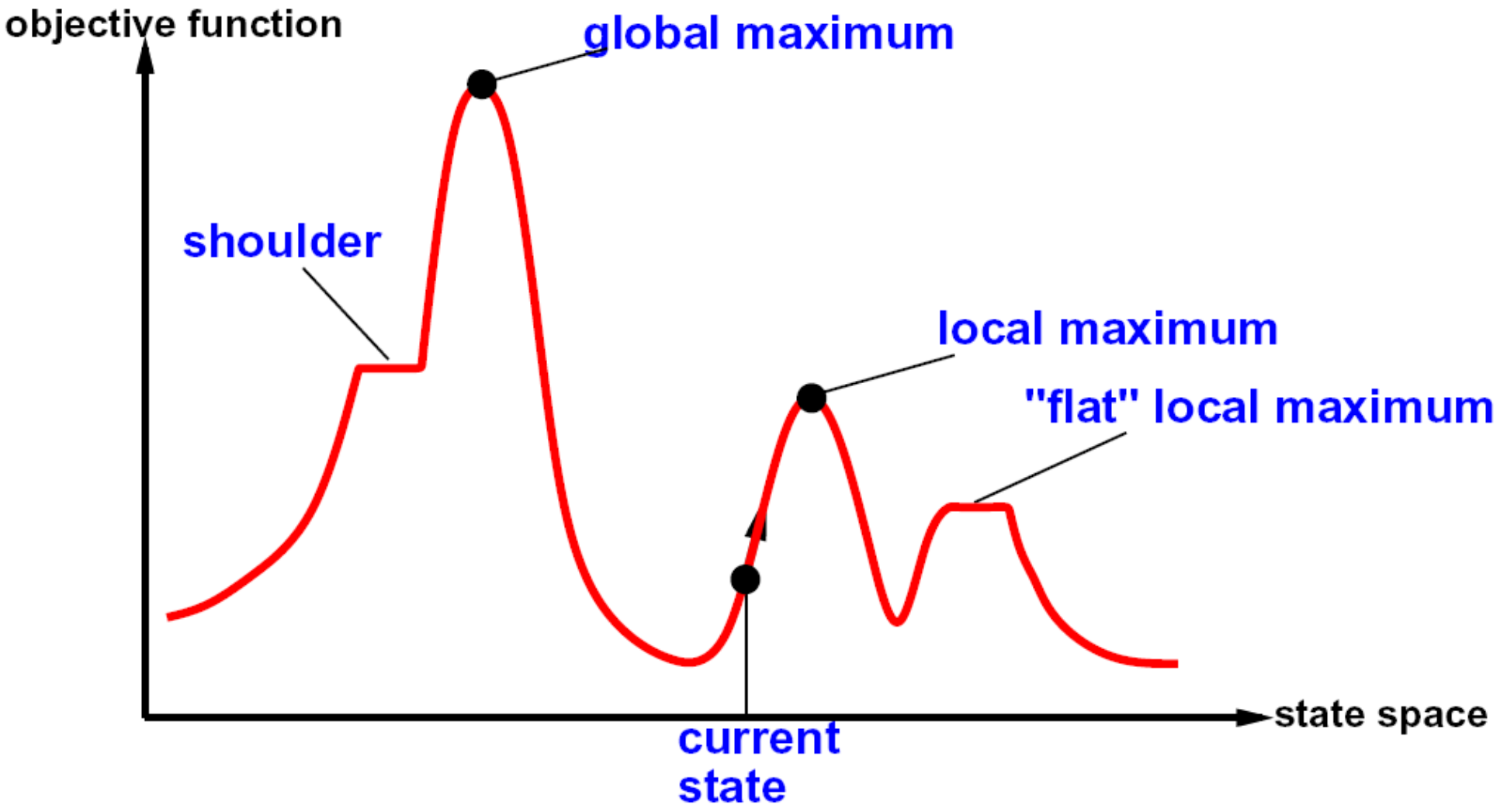


- Generally much faster and more memory efficient (but incomplete and suboptimal)

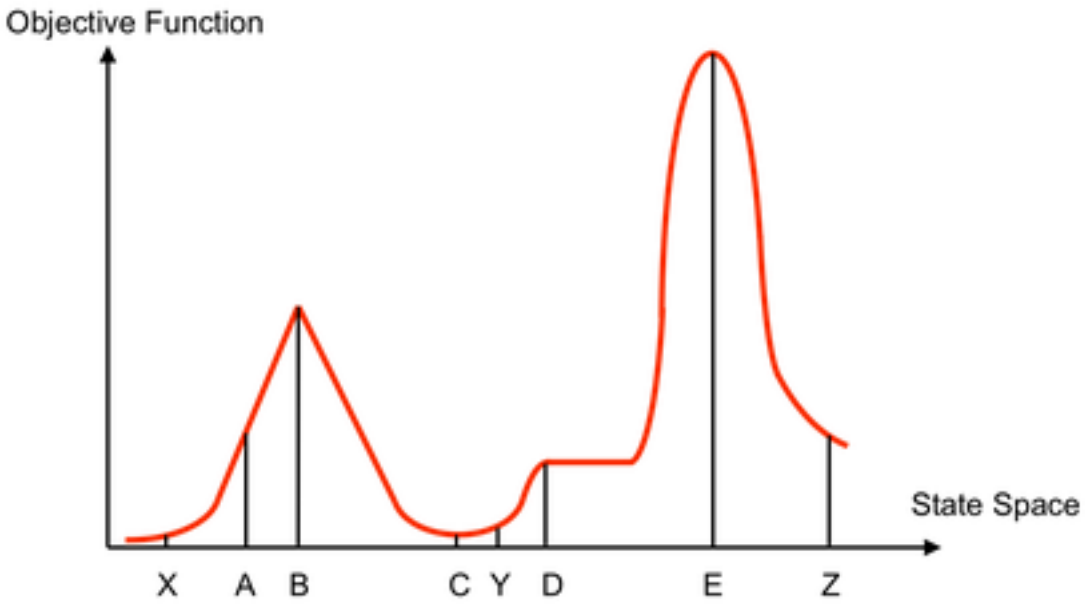
Hill climbing

- Simple, general idea:
  - Start wherever
  - Repeat: move to the best neighboring state
  - If no neighbors better than current, quit
- What's bad about this approach?
  - Complete?
  - Optimal?
- What's good about it?

Hill climbing diagram



Hill climbing quiz



Starting from X, where do you end up ?

Starting from Y, where do you end up ?

Starting from Z, where do you end up ?

Simulated annealing

- Idea: Escape local maxima by allowing downhill moves
  - But make them rarer as time goes on

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"
  local variables: current, a node
                   next, a node
                   T, a "temperature" controlling prob. of downward steps

  current ← MAKE-NODE(INITIAL-STATE[problem])
  for t ← 1 to ∞ do
      T ← schedule[t]
      if T = 0 then return current
      next ← a randomly selected successor of current
      ΔE ← VALUE[next] – VALUE[current]
      if ΔE > 0 then current ← next
      else current ← next only with probability e^(Δ E/T)
```

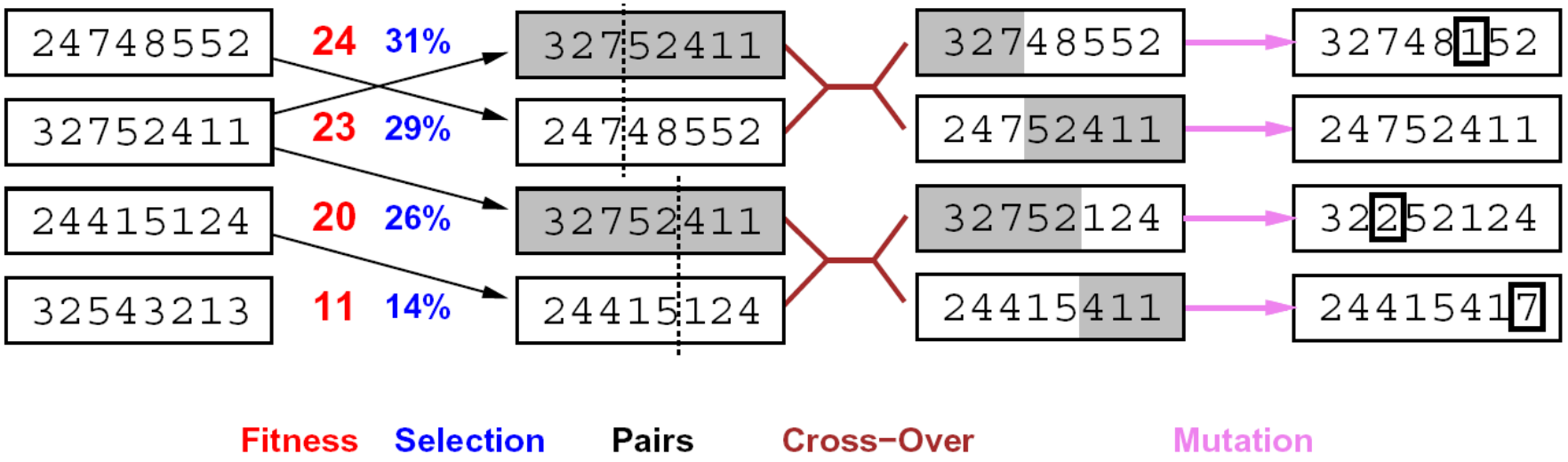
Simulated annealing

- Theoretical guarantee:
  - Stationary distribution: $p(x) \propto e^{\frac{E(x)}{kT}}$
  - If T decreased slowly enough, will converge to optimal state!

- Is this an interesting guarantee?

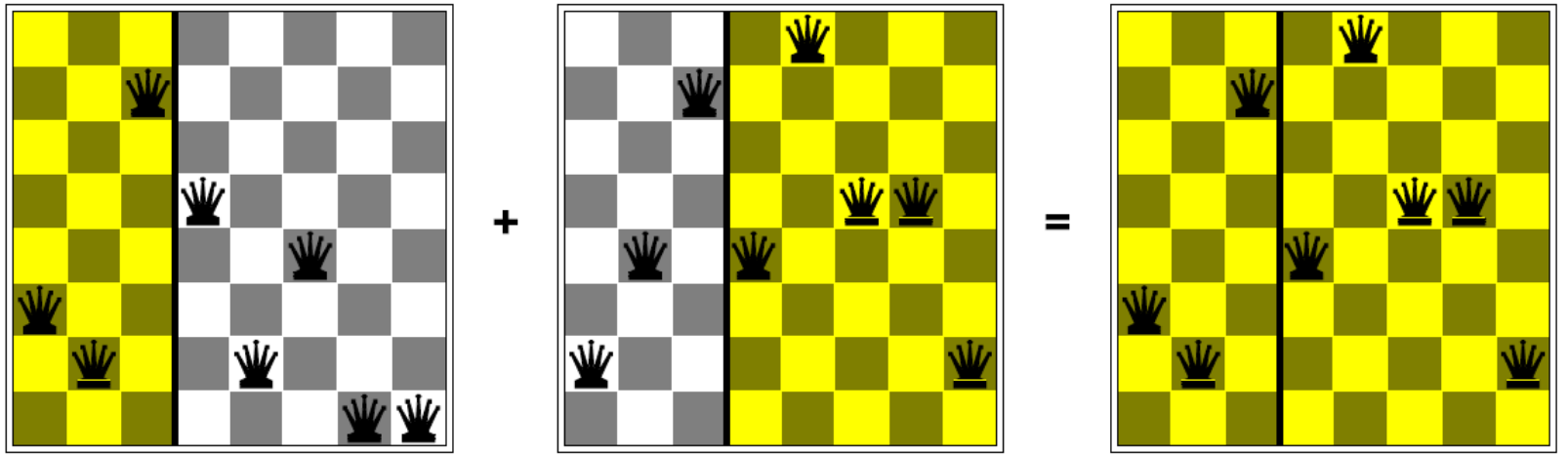
- Sounds like magic, but reality is reality:
  - The more downhill steps you need to escape a local optimum, the less likely you are to ever make them all in a row
  - People think hard about *ridge operators* which let you jump around the space in better ways

Genetic algorithms



- Genetic algorithms use a natural selection metaphor
  - Keep best N hypotheses at each step (selection) based on a fitness function
  - Also have pairwise crossover operators, with optional mutation to give variety
- Possibly the most misunderstood, misapplied (and even maligned) technique around

Example: N-Queens



- Why does crossover make sense here?
- When wouldn’t it make sense?
- What would mutation be?
- What would a good fitness function be?

STEVENS
INSTITUTE OF TECHNOLOGY
1870

Questions?

Chapter 1, 2, 3

Acknowledgement

Fahiem Bacchus, University of Toronto
Dan Klein, UC Berkeley
Kate Larson, University of Waterloo

Attendance

- We use QR codes to save time.
  - Submit a google form through the QR code.

- How to scan a QR code?
  - Android
    - Built-in camera (Google lens)
    - https://play.google.com/store/apps/details?id=com.camvision.qrcode.barcode.reader&hl=en_US&gl=US&pli=1
  - iOS
    - Built-in camera
    - https://apps.apple.com/us/app/qr-code-reader/id1200318119