Introduction to Artificial Intelligence

Chapter 2: Solving Problems by Searching (2) Uninformed Search

Outline

1. Uninformed Search Strategies
2. Breadth-first Search
3. Uniform-cost Search
4. Depth-first Search
5. Depth-limit Search
6. Iterative Deepening Search
7. Bidirectional Search
8. Summary

Uninformed Search Strategies

- Use only the information available in the problem definition

Uninformed Search Strategies

- An other name: Blind Search

Uninformed search strategies

- Algorithms:
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search
  - Iterative lengthening search
  - Bidirectional search
  - Branch and Bound
  - ...

Review: Tree Search Algorithms

- Tree search can end up repeatedly visiting the same nodes:
  - Arad-Sibiu-Arad-Sibiu-Arad-…

→ *A good search algorithm avoids such paths*

Review: Search Strategies

- A search strategies is defined by picking the **order** of node expansion
- How to evaluate a search strategy?
  - Completeness
  - Time complexity
  - Space complexity
  - Optimality

Measured by *b, d, m*

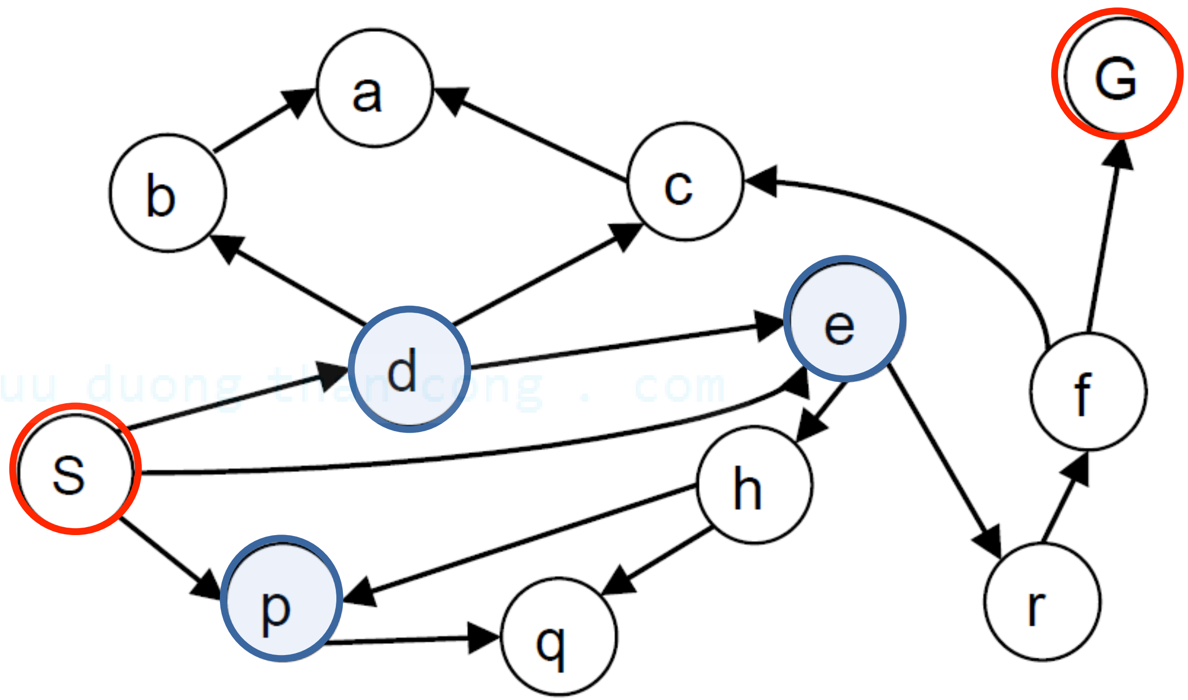
- *b*: maximum number of successors of a node
- *d*: depth of the shallowest goal node
- *m*: maximum length of any path in the state space

Breadth-first Search (BFS)

Breadth-first search

- Expand shallowest unexpanded node
- Implementation: *frontier* is a FIFO queue

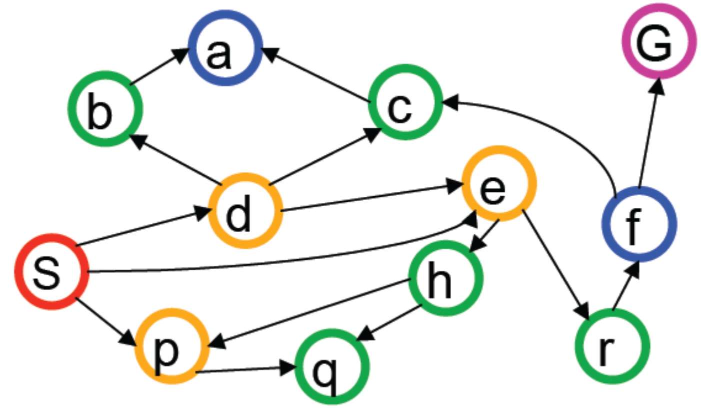
Example state space graph for a tiny search problem

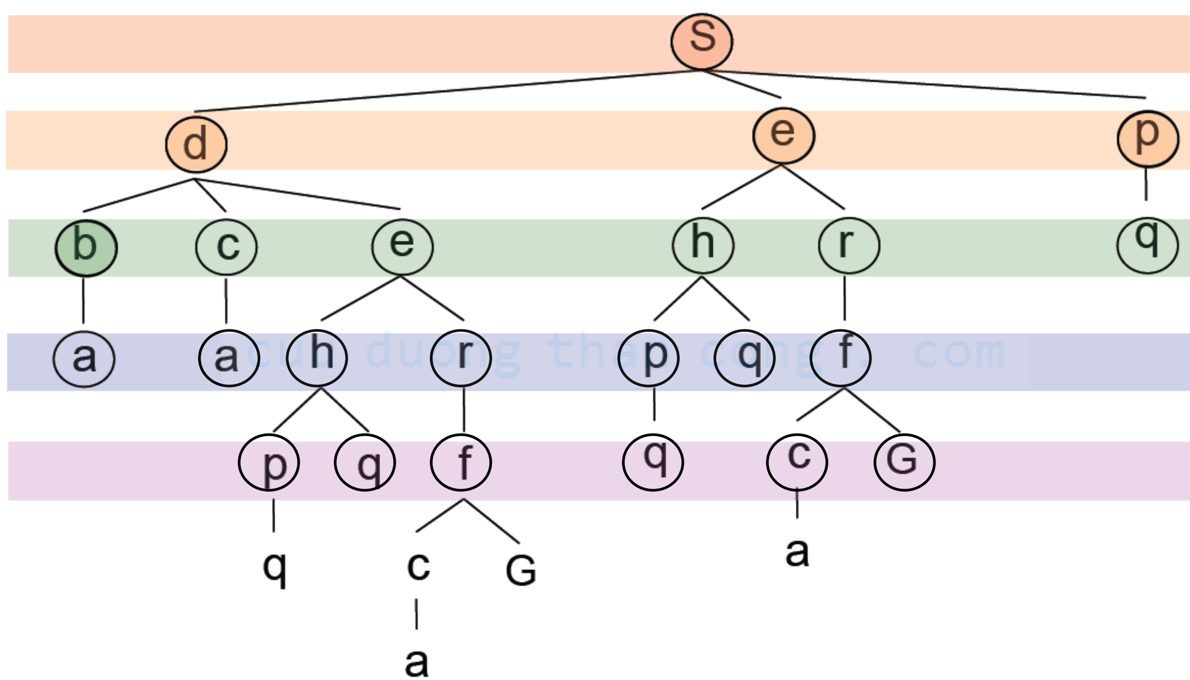


Example from P. Abbeel and D. Klein

Breadth-first search

- ❑ Expansion order: (*S,d,e,p,b,c,e,h,r,q,a,a,h,r,p, q,f,p,q,f,q,c,G*)





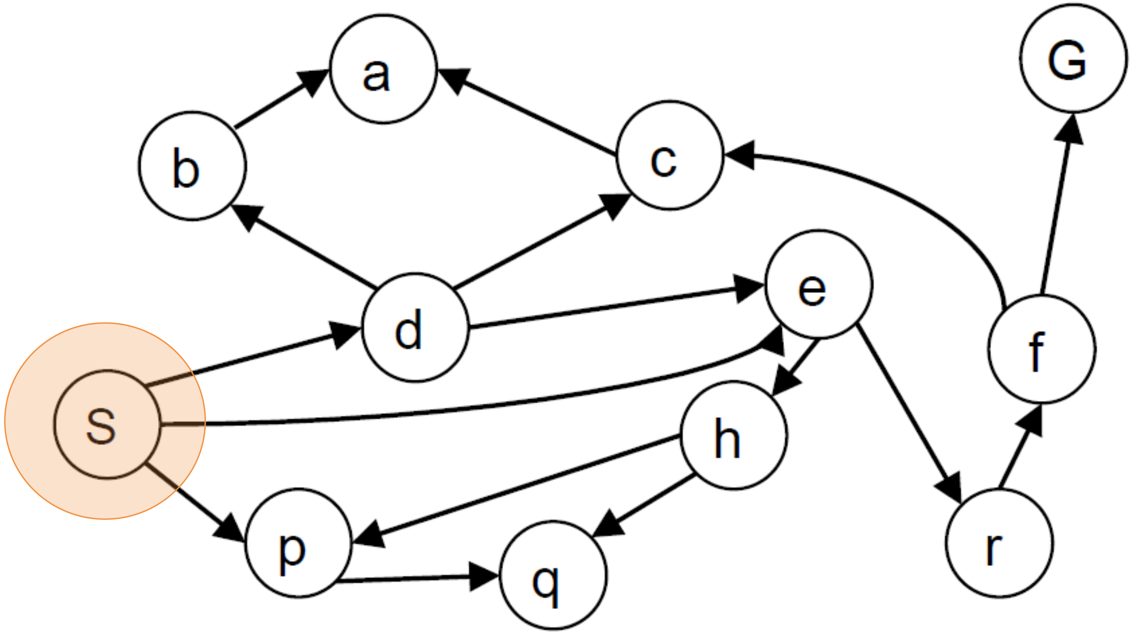
Breadth-first search

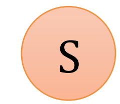
- BFS is an instance of the general graph search algorithm.
  1. The shallowest **<u>unexpanded</u>** node is chosen for expansion
  2. The goal test is applied to each node when it is generated rather than when it is selected for expansion
  3. Discard any new path to a state already in the frontier or explored set

Breadth-first search

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier) /* chooses the shallowest node in frontier */
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
           frontier ← INSERT(child, frontier)
```

Breadth-first search

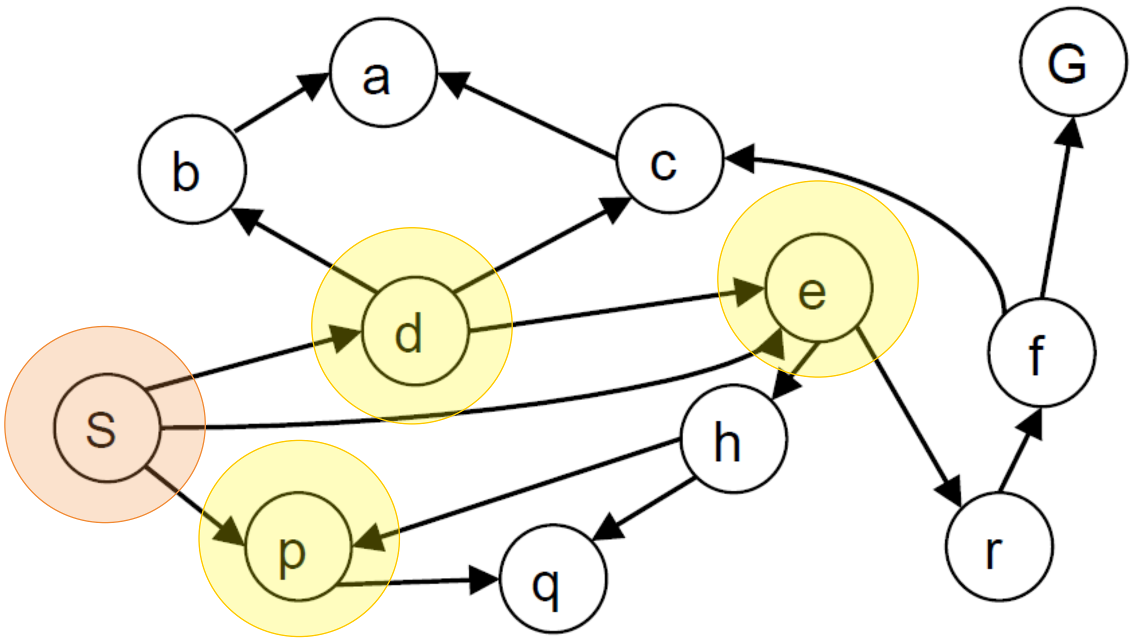




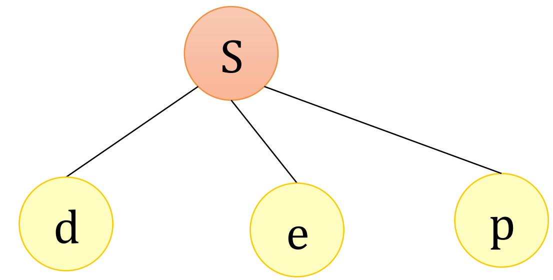
$d = 0$

Search Tree

Breadth-first search

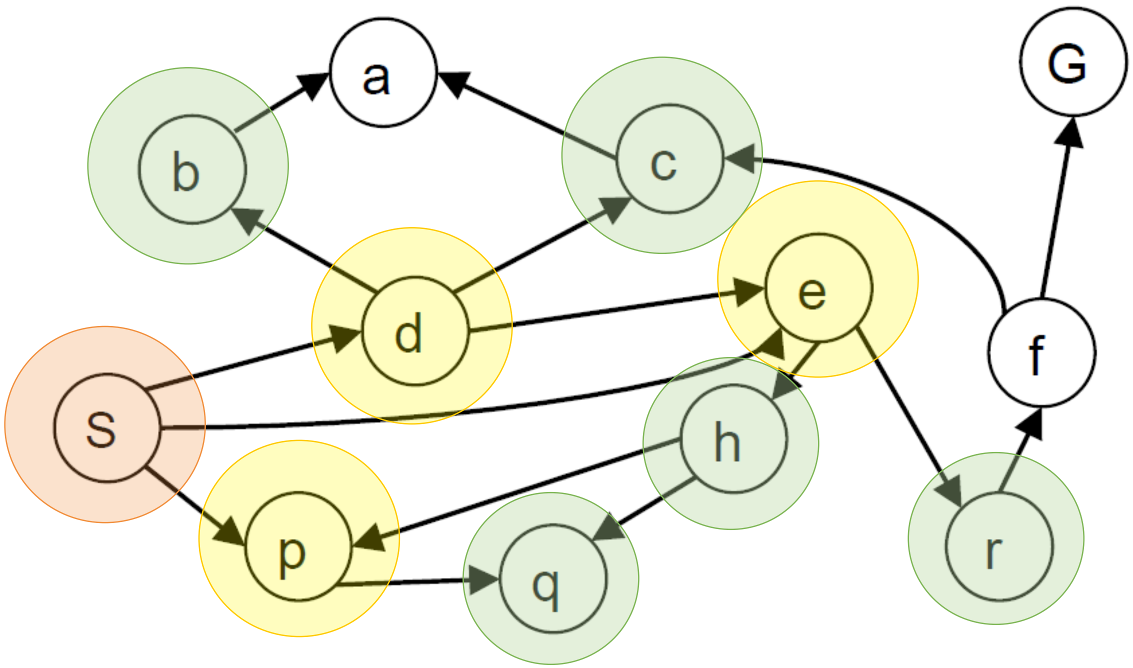


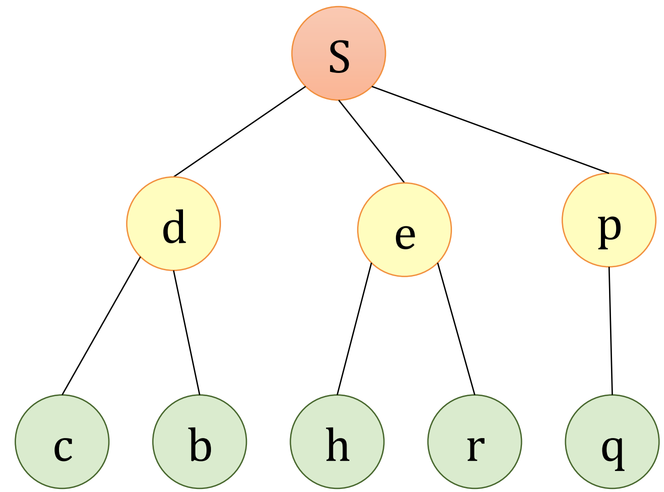
$d = 1$



Search Tree

Breadth-first search

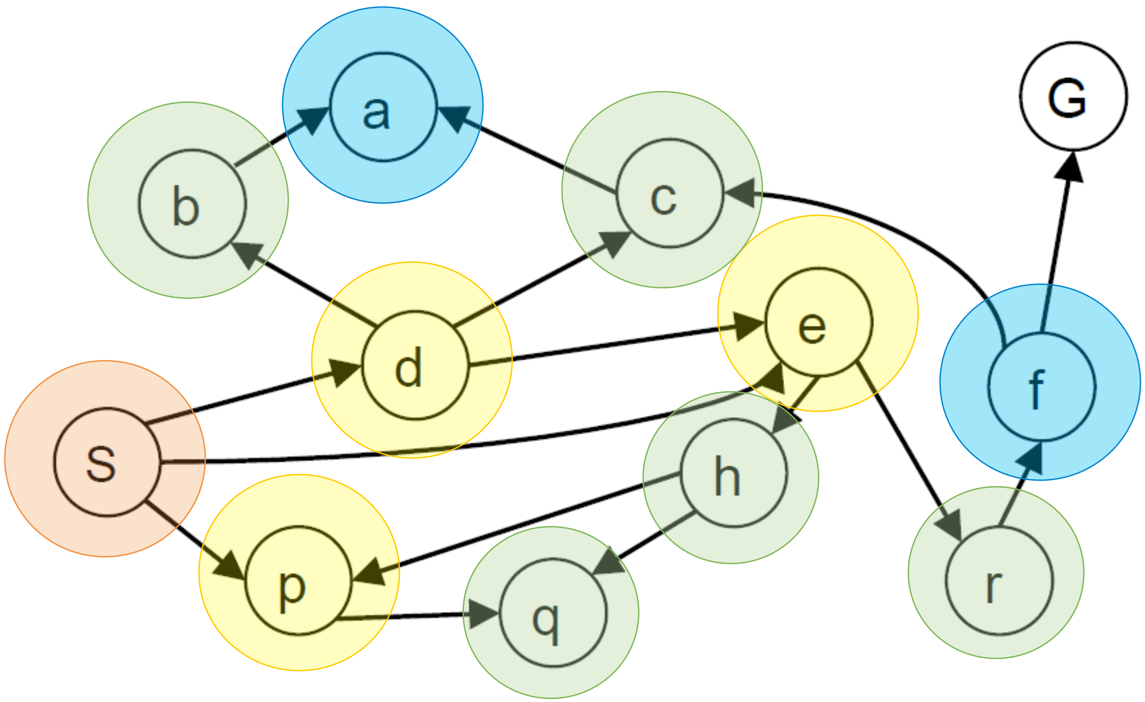


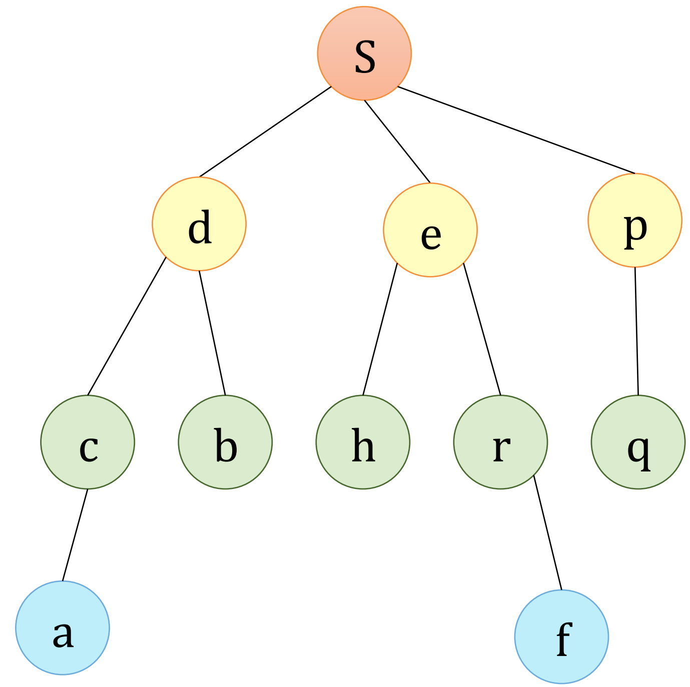


$d = 2$

Search Tree

Breadth-first search

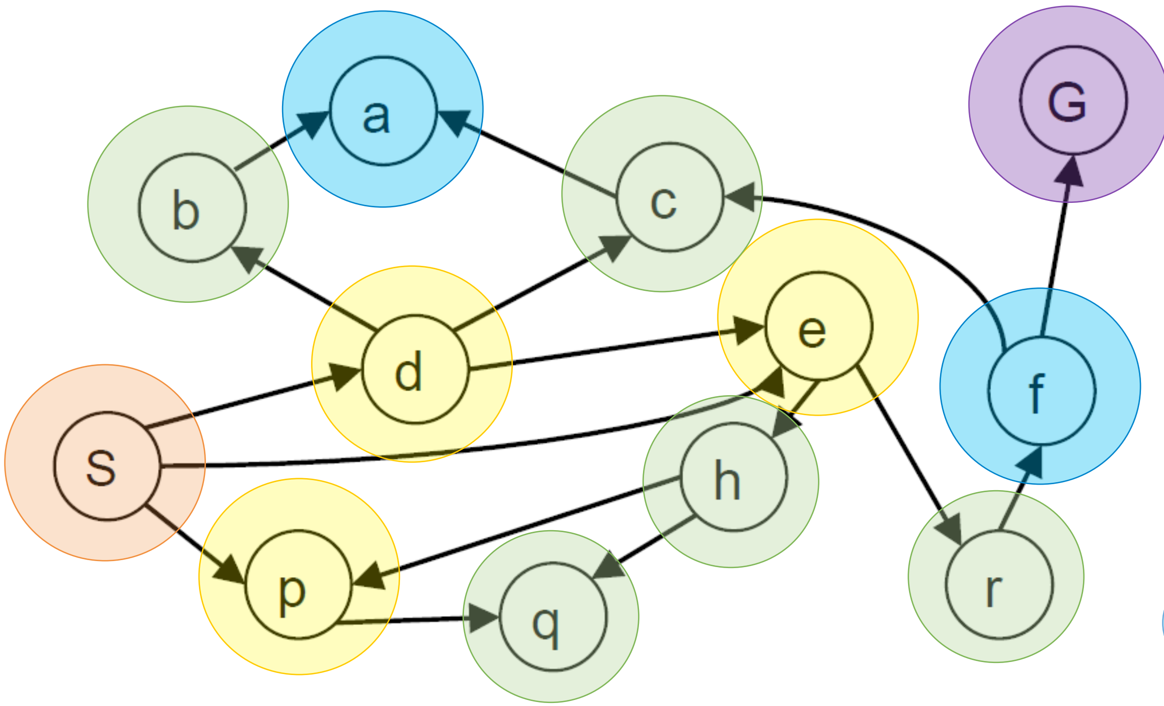




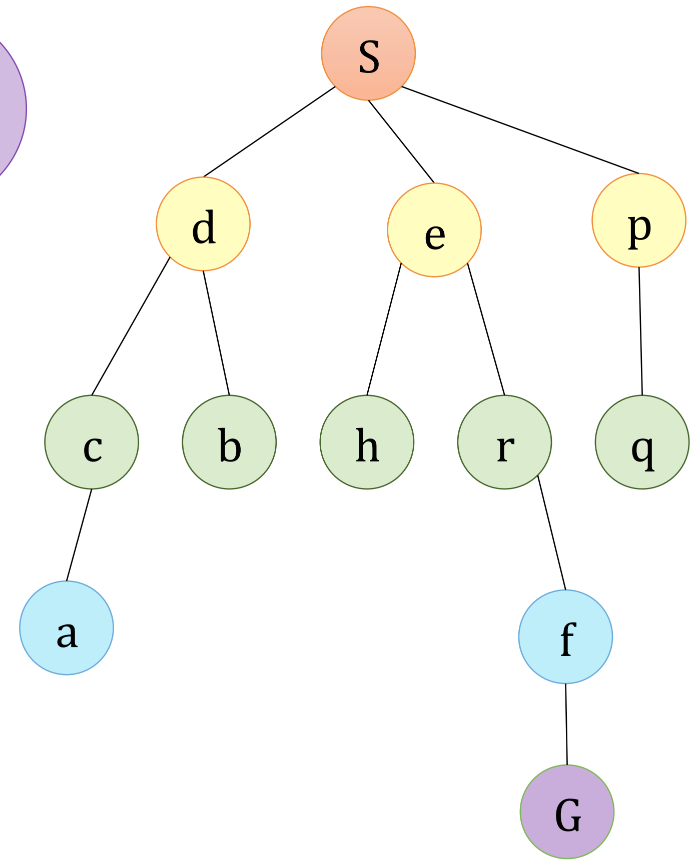
$d = 3$

Search Tree

Breadth-first search



$d = 4$



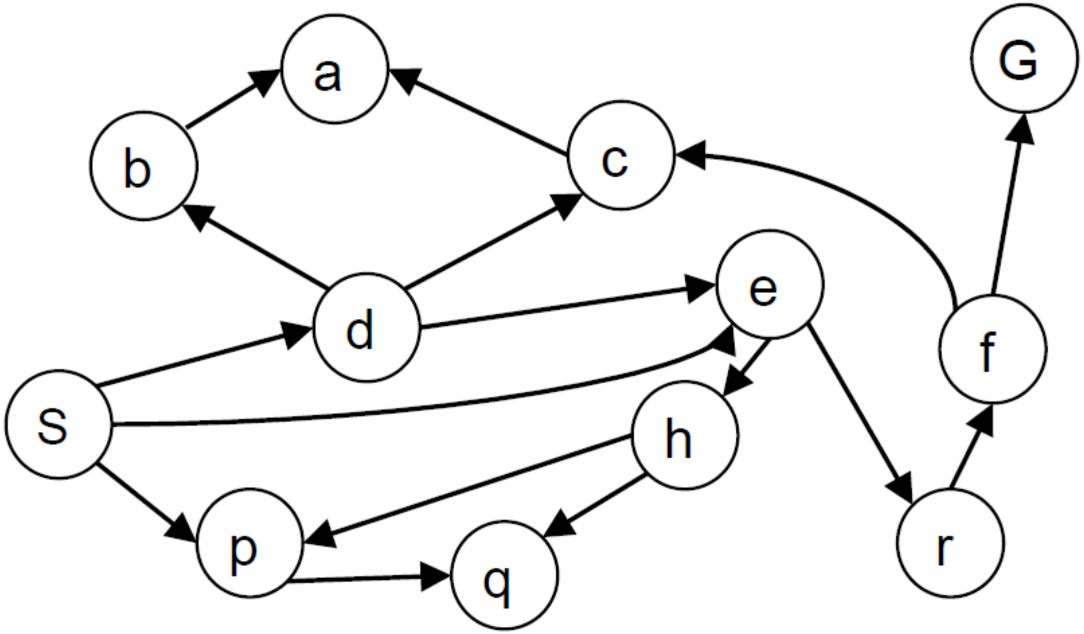
Search Tree

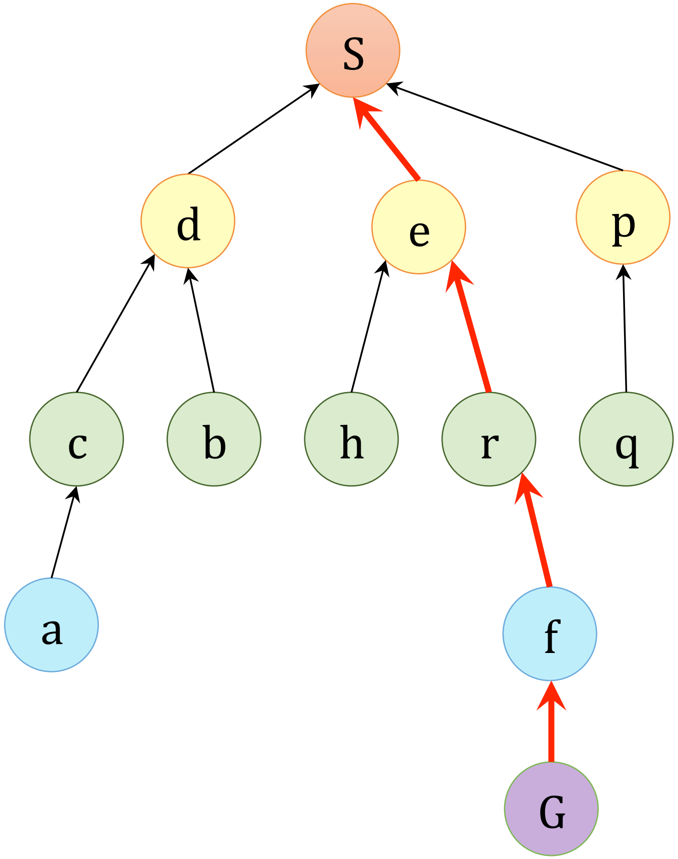
Breadth-first search

- BFS identifies the goal but DOES NOT tell you the path to the goal
- To get the path information, we have to store parent information in the *frontier* (OPEN) and *expanded list* (CLOSE)
  - E.g., OPEN={d,e,p}, CLOSE={S}

  → OPEN={[d,S], [e,S], [p,S]}, CLOSE={[S, Nil]}

Breadth-first search





Search Path: S → e → r → f → G

Search Tree

QUIZ

Draw the search tree for the 8-puzzle problem with *d*=3, given the following initial state and goal state: (do not draw repeated state)

1	8	2
	4	3
7	6	5

initial state

→

1	2	3
4	5	6
7	8	

goal state

Evaluation of BFS

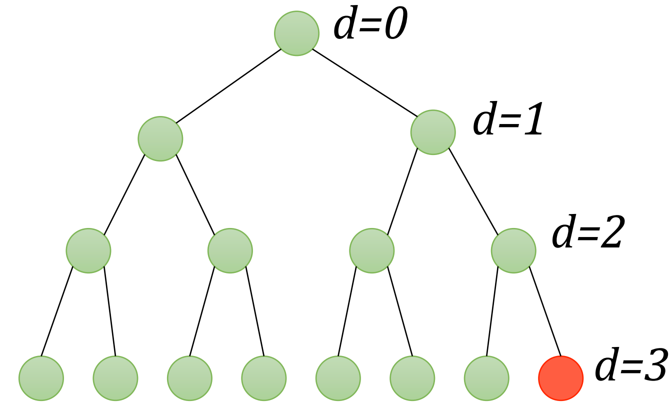
- Completeness
  - Yes
- Optimality
  - Not always
  - When?
- Time complexity:
  - $O(b^d)$
- Space complexity:
  - $O(b^d)$

Main practical drawback

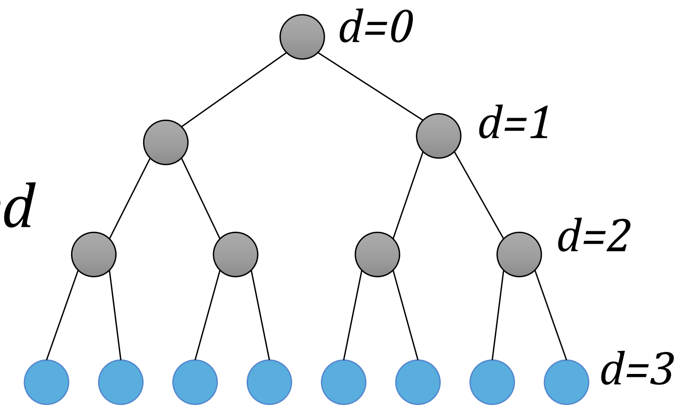
Complexity of BFS

- **Time Complexity:**
  - Worst case: 1 Goal node at the right hand side at depth *d*
  - Number of nodes BFS generates:

$$b + b^2 + \ldots + b^d = \boldsymbol{O}(\boldsymbol{b^d})$$



- **Space complexity:**
  - Worst case: at depth *d*
    - number of nodes in the *expanded set:* $\boldsymbol{O}(\boldsymbol{b^{d-1}})$
    - number of nodes in the *frontier* (queue): $\boldsymbol{O}(\boldsymbol{b^d})$



Complexity of BFS

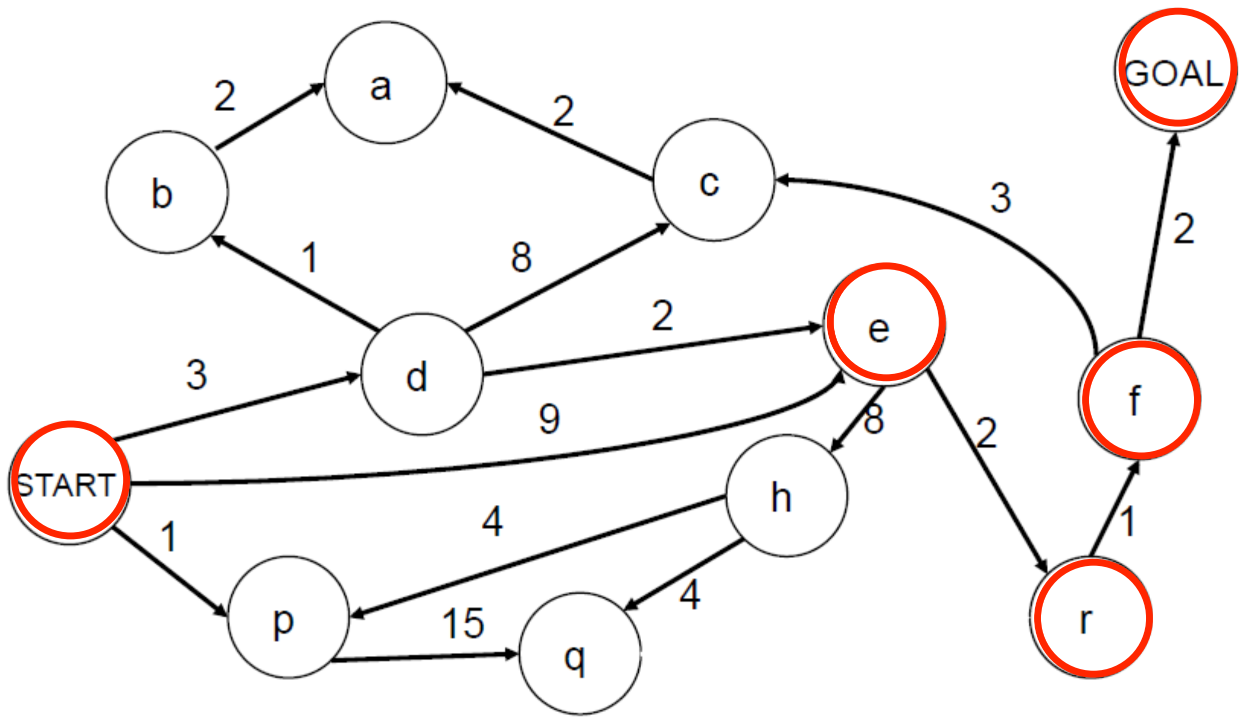
Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Time and memory requirements for BFS. The numbers shown assume branching factor b=10; 1 million nodes/second; 1000 bytes/node.

In general, *exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances.*

Uniform-cost Search (UCS)

Search with varying step costs



- ❑BFS finds the path with the fewest steps, but does not always find the cheapest path

Uniform-cost search

- For each frontier node, save the total cost of the path from the initial state to that node
- Expand the frontier node with the lowest path cost $g(n)$
- Implementation: *frontier* is a priority queue ordered by g

→Equivalent to breadth-first if step costs all equal

→Equivalent to Dijkstra's algorithm in general

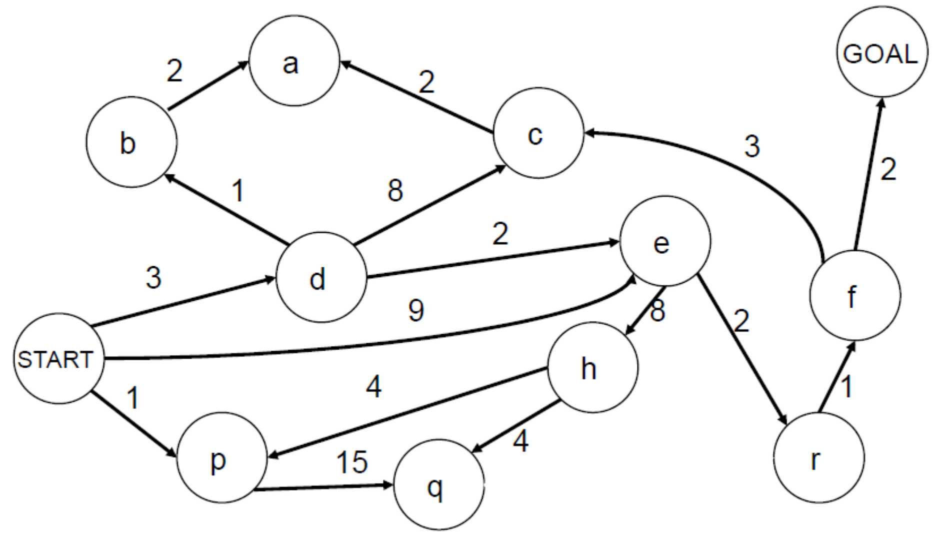
- Significant difference with BFS:
  - Goal test is applied to a node when it is selected for expansion

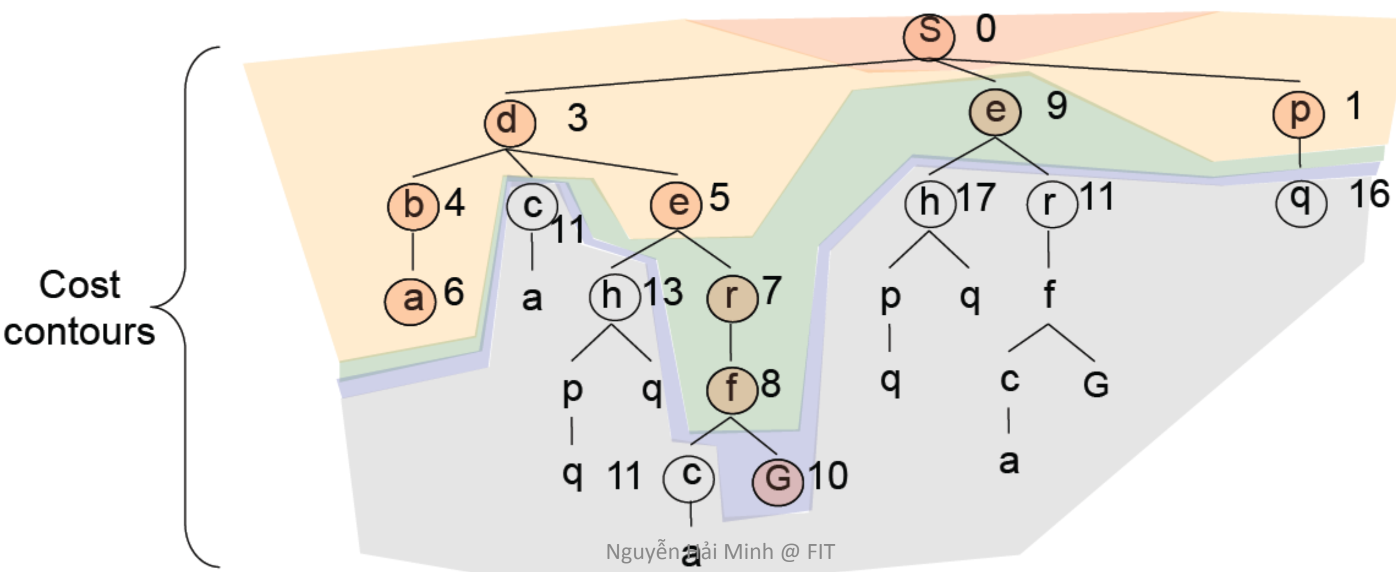
Uniform-cost search

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
      if EMPTY?(frontier) then return failure
      node ← POP(frontier)   /* chooses the lowest-cost node in frontier */
      if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
      add node.STATE to explored
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          if child.STATE is not in explored or frontier then
              frontier ← INSERT(child, frontier)
          else if child.STATE is in frontier with higher PATH-COST then
              replace that frontier node with child
```

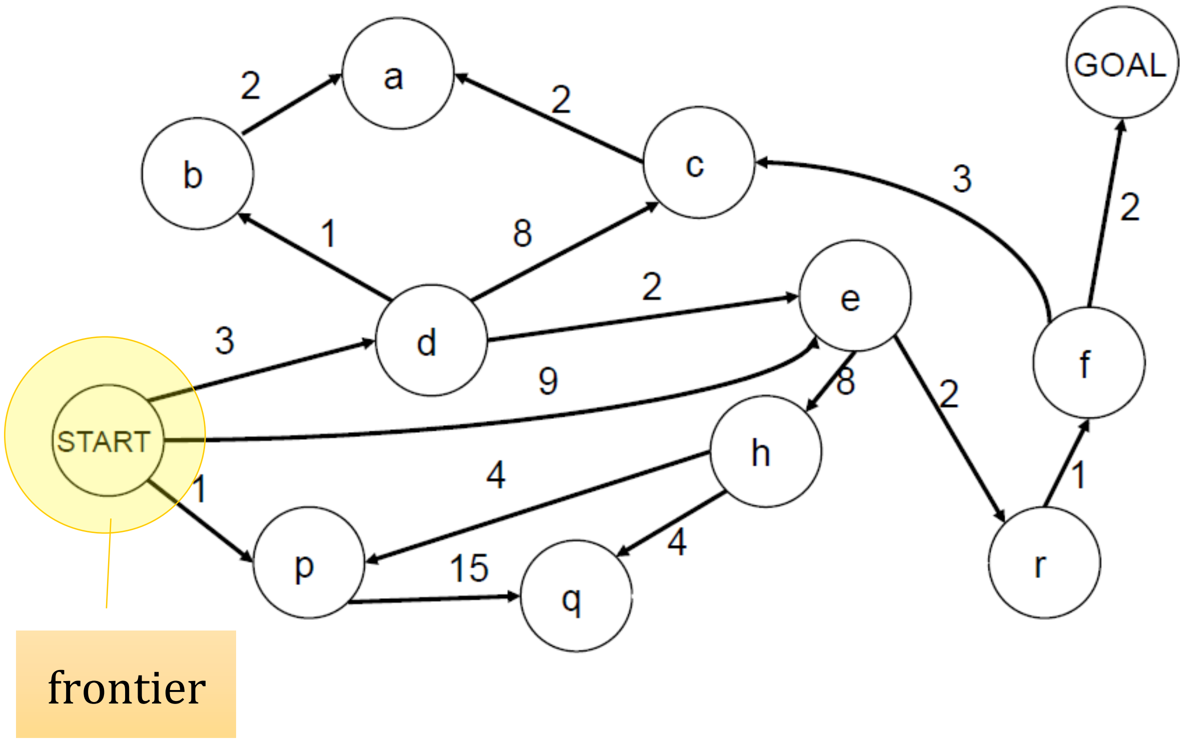
Uniform-cost search example

- ❑Expansion order: (S,p,d,b,e,a,r,f,e,G)

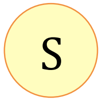




Uniform-cost search example

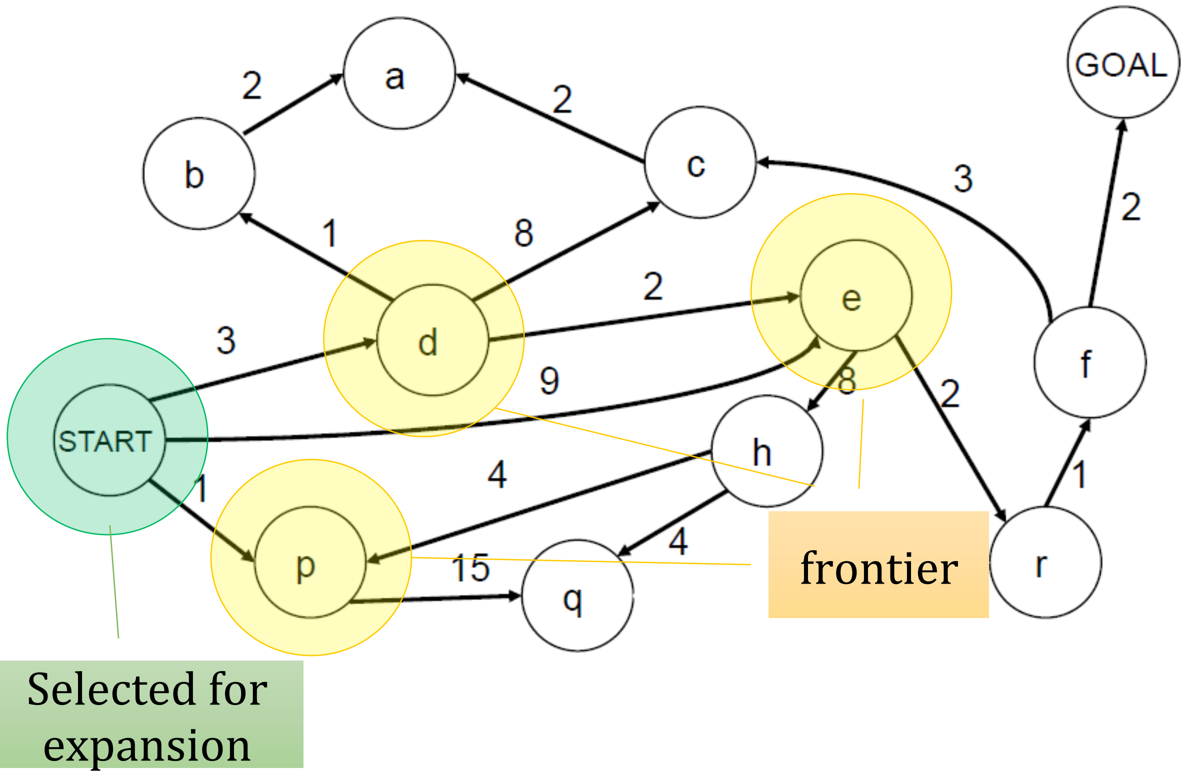


PQ = { (S:0) }

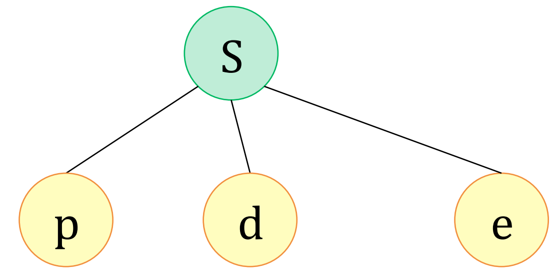


Search Tree

Uniform-cost search example

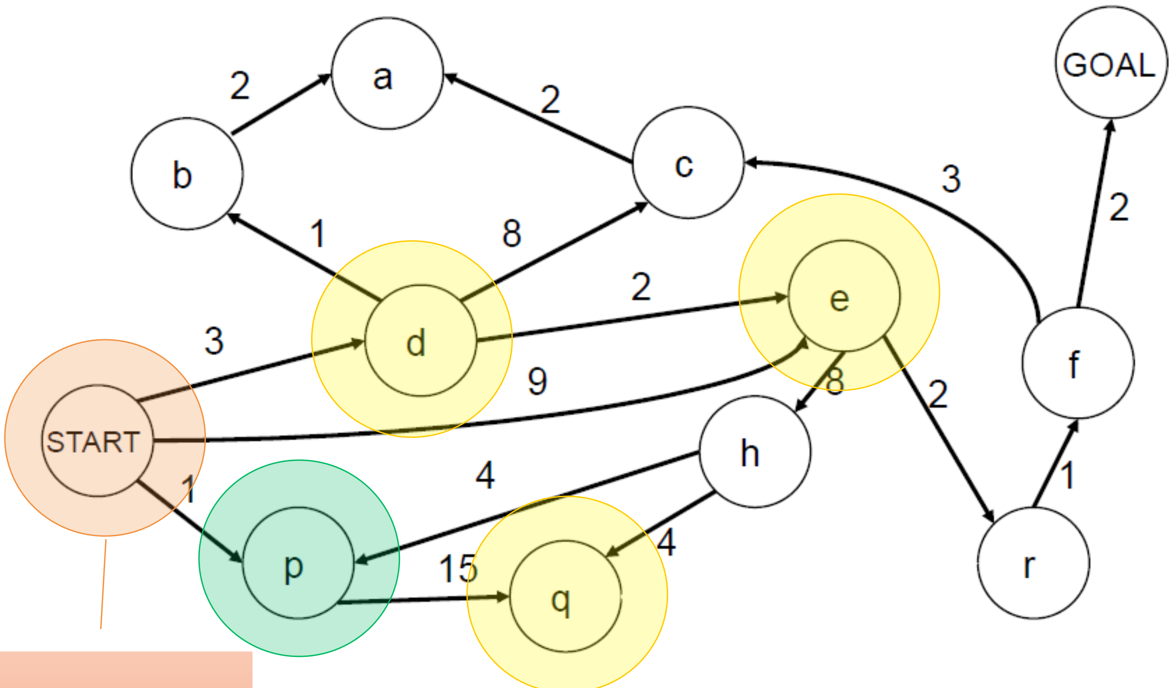


PQ = { (p:1), (d:3), (e:9) }



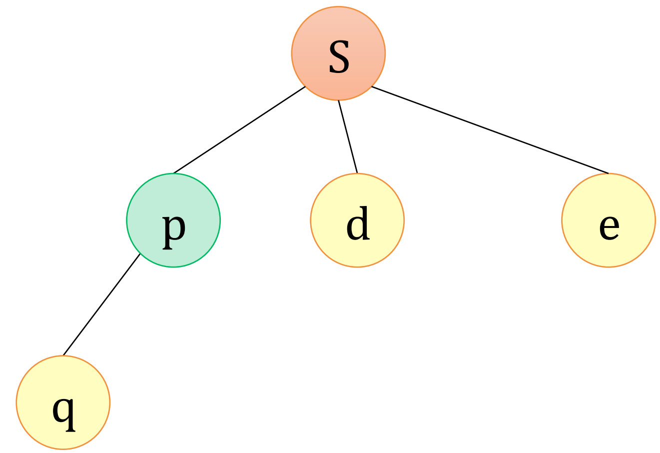
Search Tree

Uniform-cost search example



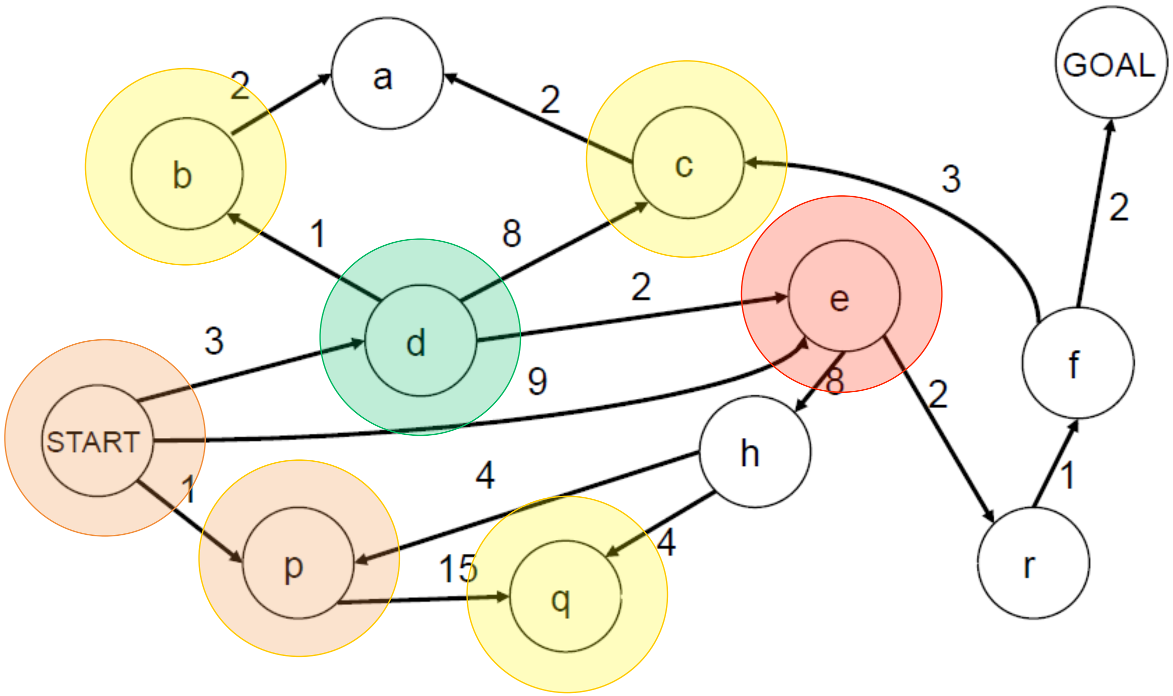
expanded

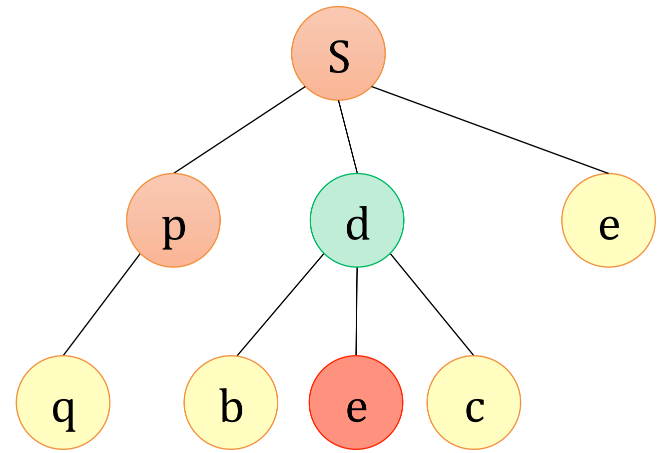
PQ = { (d:3), (e:9), (q:16) }



Search Tree

Uniform-cost search example



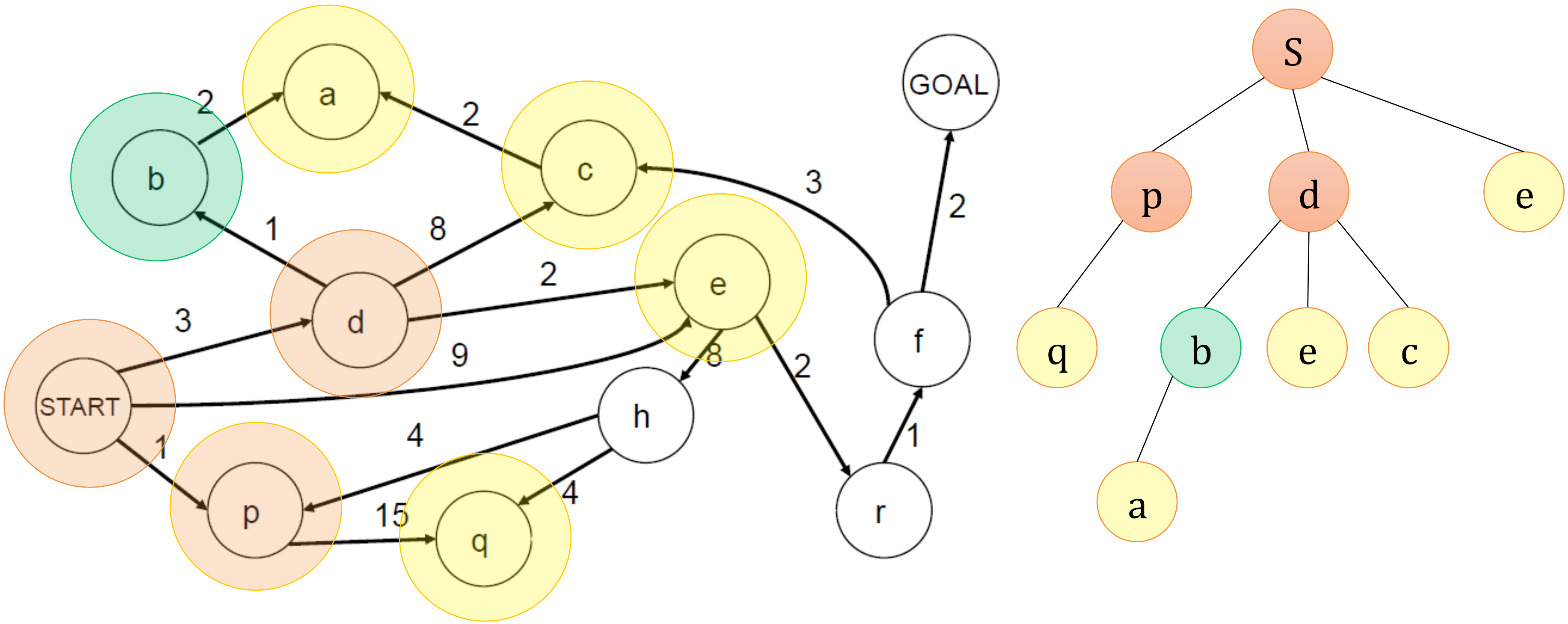


PQ = { (b:4), (e:5), (c:11), (q:16) }

Update path cost of *e*

Search Tree

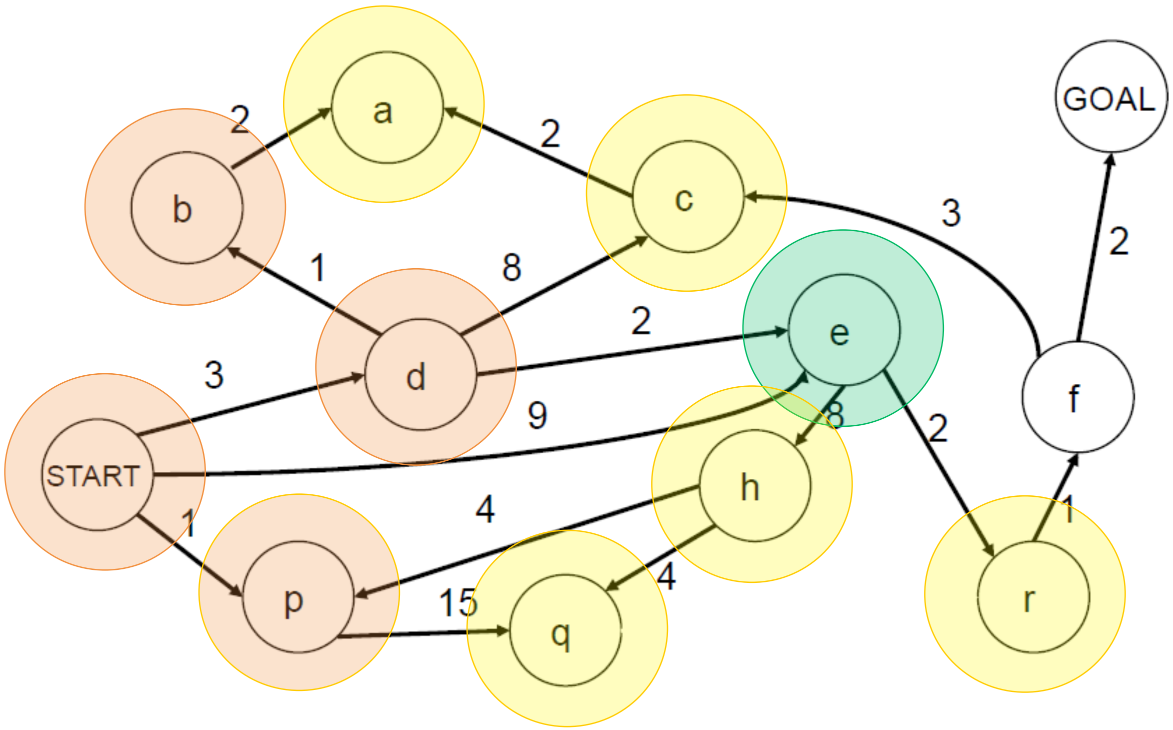
Uniform-cost search example

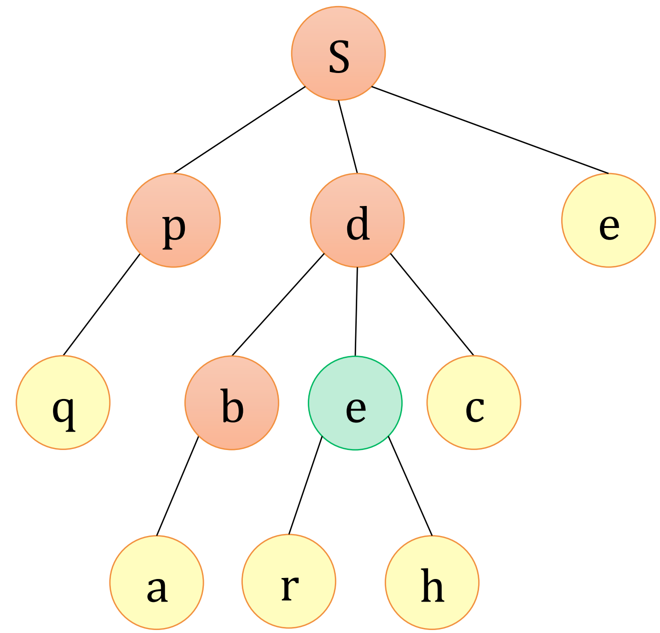


PQ = { (e:5), (a:6), (c:11), (q:16) }

Search Tree

Uniform-cost search example

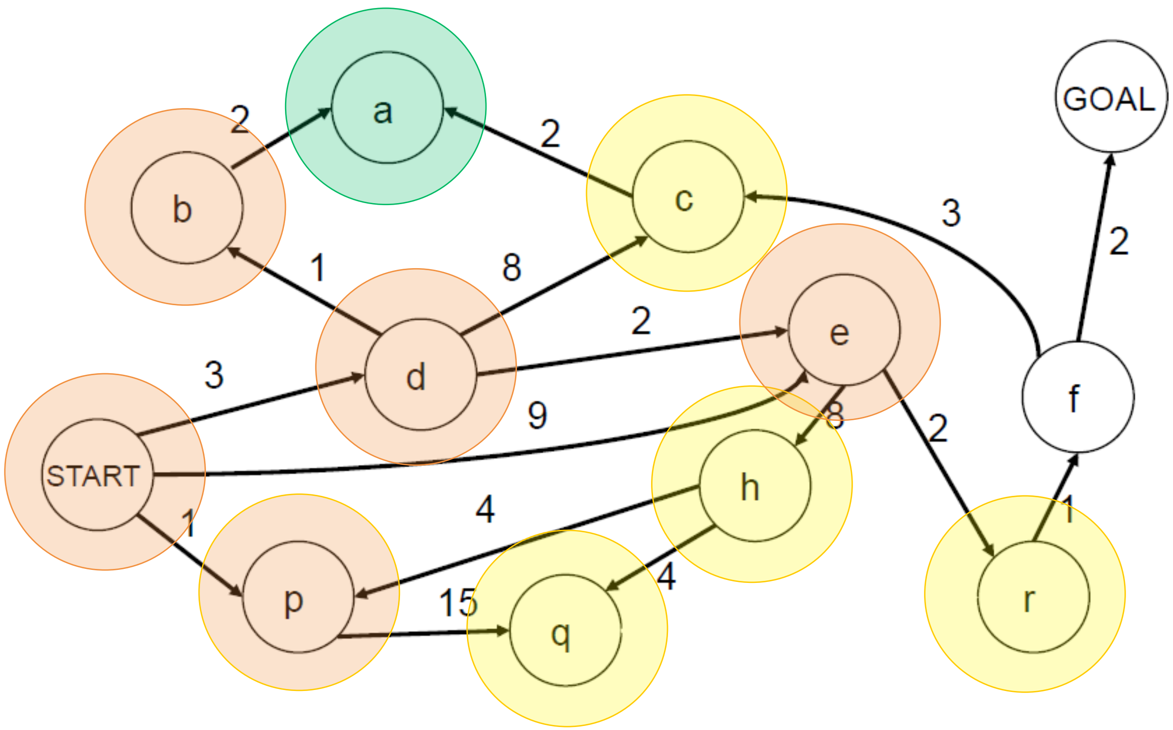


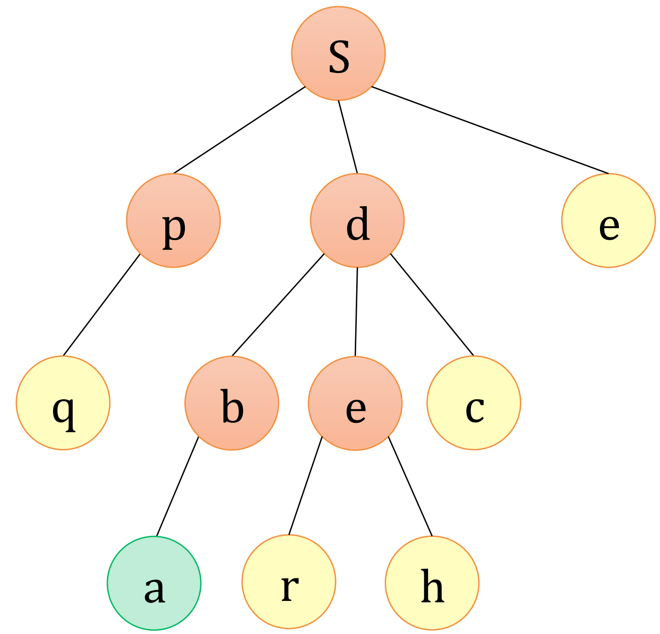


PQ = { (a:6), (r:7), (c:11), (h:13), (q: 16) }

Search Tree

Uniform-cost search example

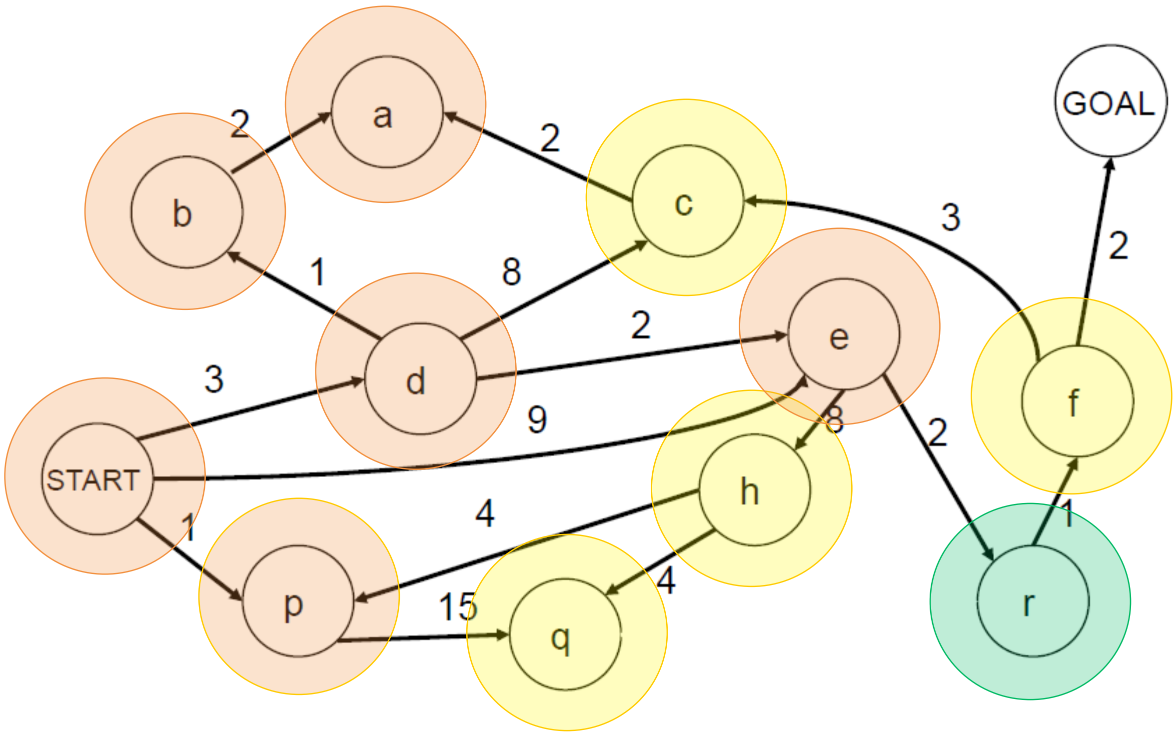


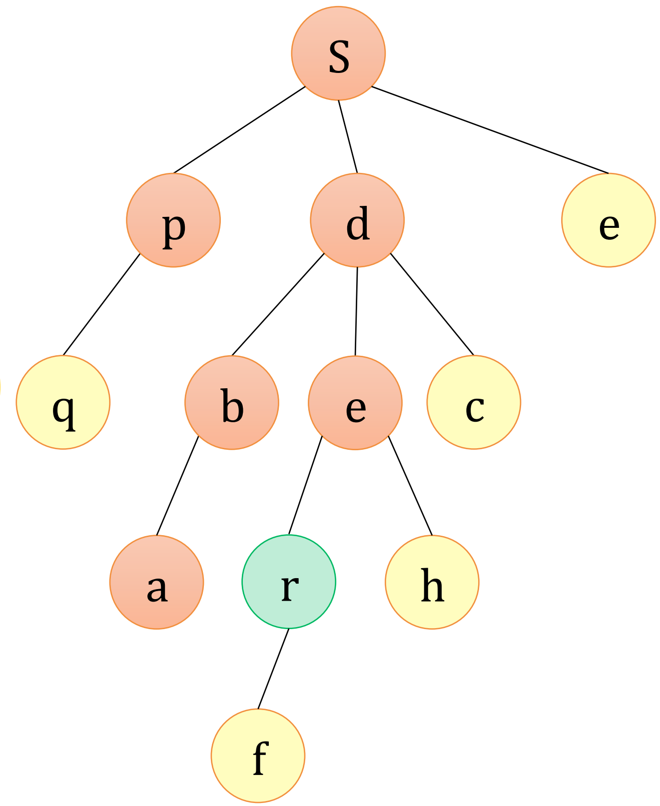


PQ = { (r:7), (c:11), (h:13), (q:16) }

Search Tree

Uniform-cost search example

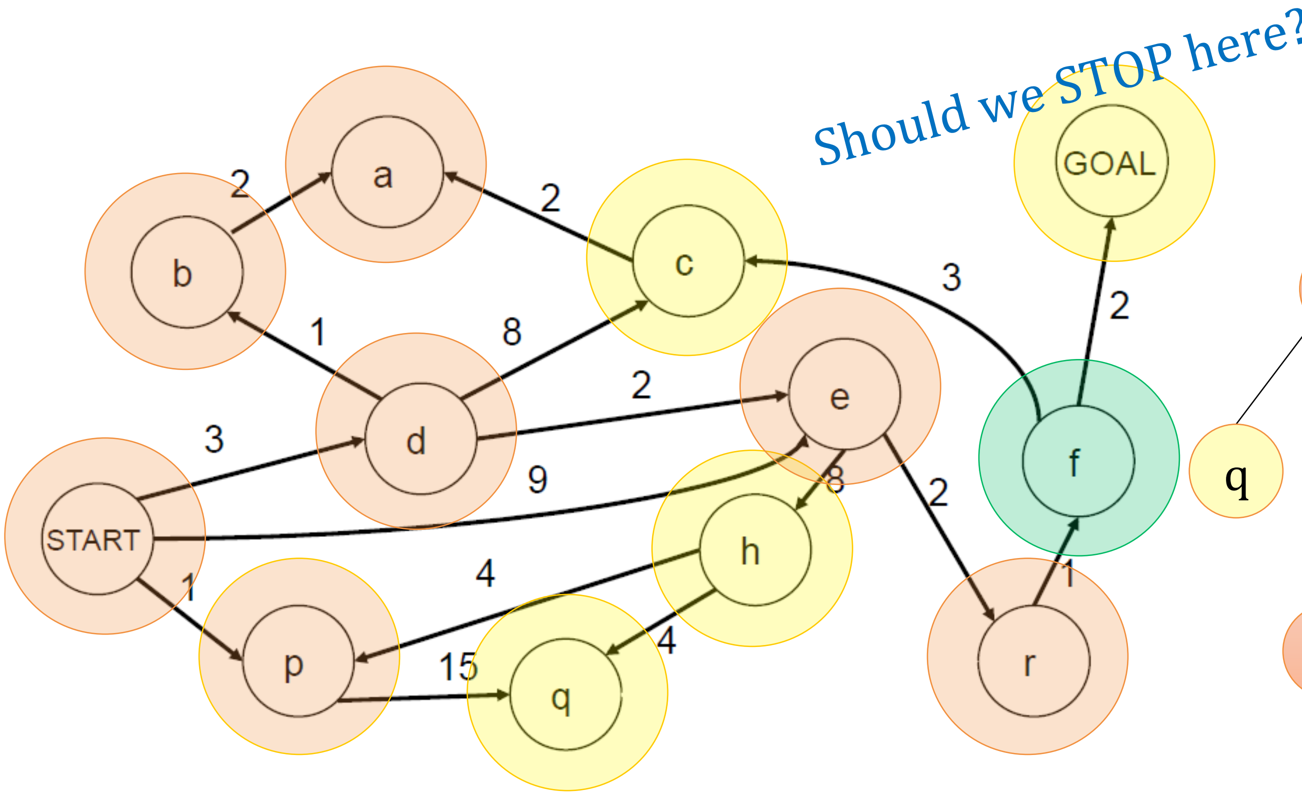




PQ = { (f:8), (c:11), (h:13), (q:16) }

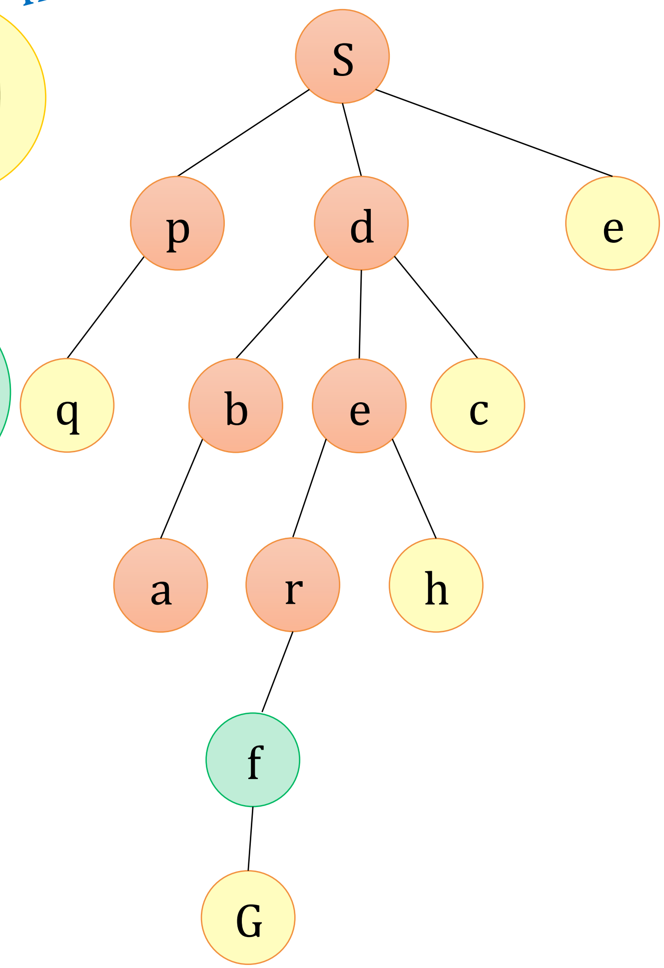
Search Tree

Uniform-cost search example



PQ = { (G:10), (c:11), (h:13), (q:16) }

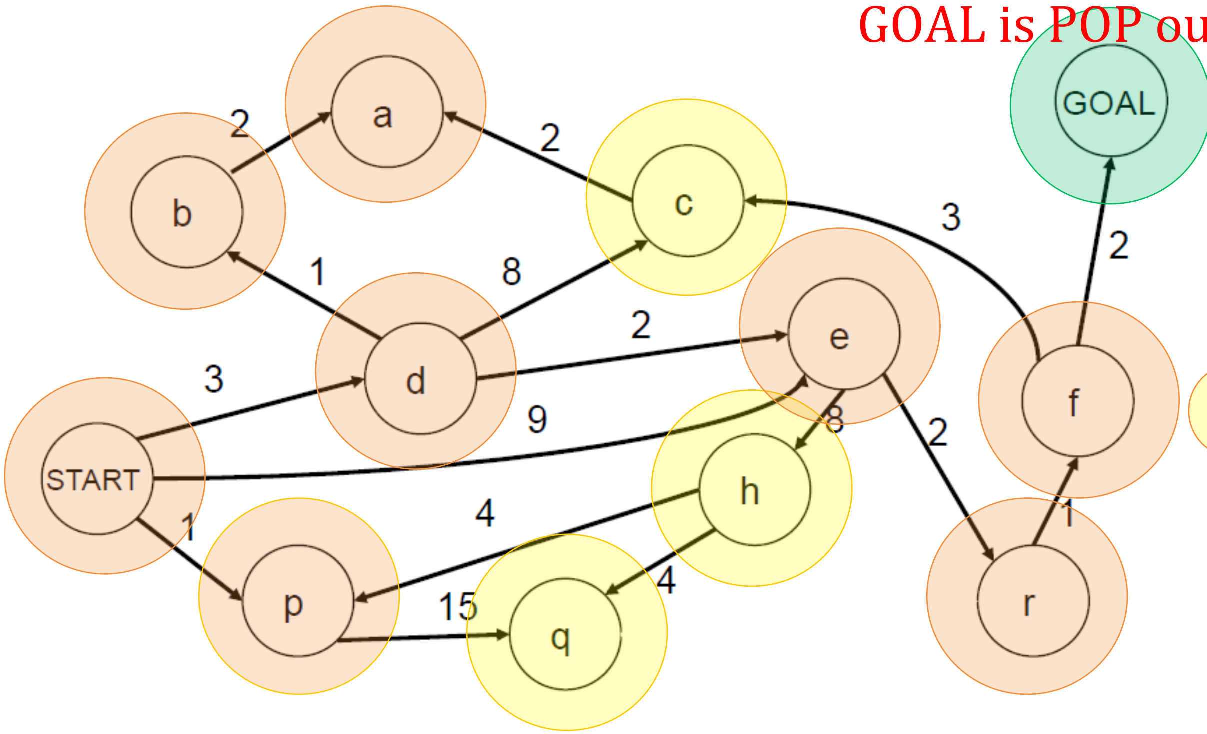
Not update path cost of *c*

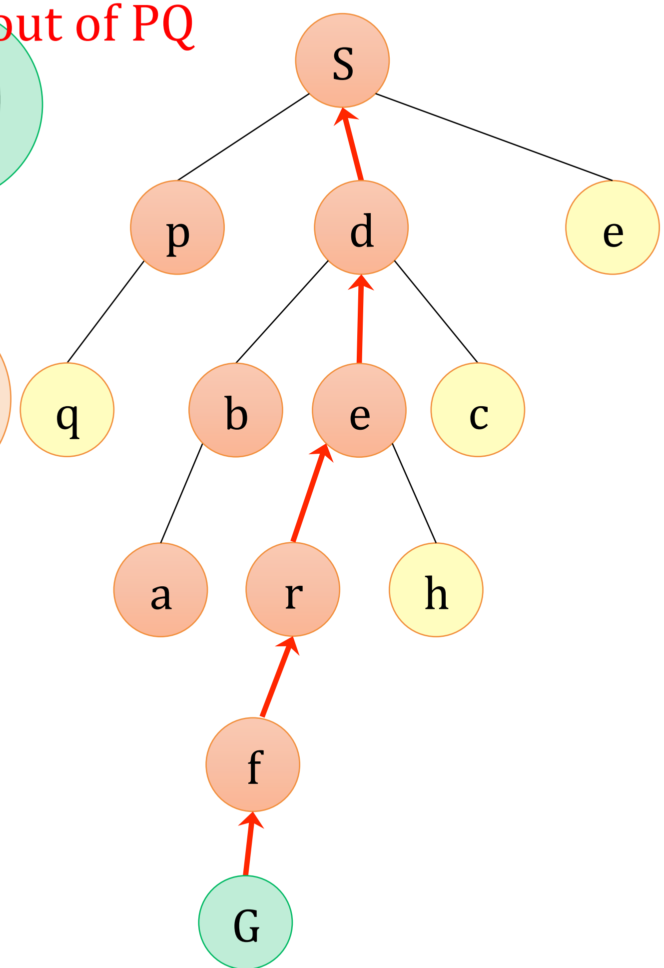


Search Tree

Uniform-cost search example

NO! Only stop when
GOAL is POP out of PQ

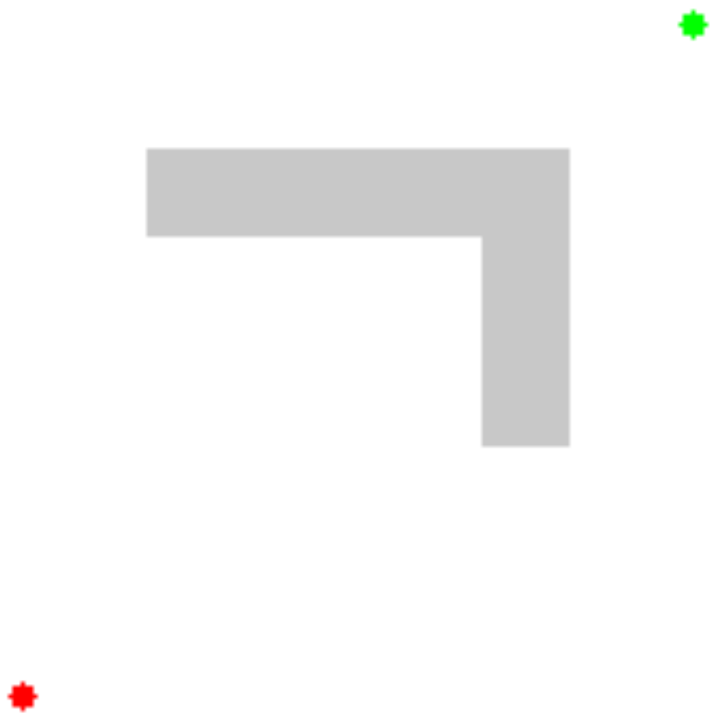




PQ = { (c:11), (h:13), (q:16) }
Goal is taken out of PQ → STOP
Search path: S→d→e→r→f→G, cost = 10

Search Tree

Example of uniform-cost search



Source: Wikipedia

Evaluation of UCS

- **Completeness**
  - Yes, if step cost $\geq \epsilon > 0$
  - Proof:
    - Given that every step costs more than 0, assuming a finite branching factor b, there is a finite number of expansions required before the total path cost is equal to the path cost of the goal state. Hence, we will reach it in a finite number of steps.
- **Optimality**
  - Yes
  - Proof?

Evaluation of UCS

- **Graph separation property**: every path from the initial state to an unexplored state has to pass through a state on the frontier
  - Proved inductively
- Optimality of UCS: proof by contradiction
  - Suppose UCS terminates at goal state n with path cost $g(n) = C$ but there exists another goal state n' with $g(n') < C$
  - Then there must exist a node n'' on the frontier that is on the optimal path to n'
  - But because $g(n'') \leq g(n') < g(n)$, n'' should have been expanded first!

Evaluation of UCS

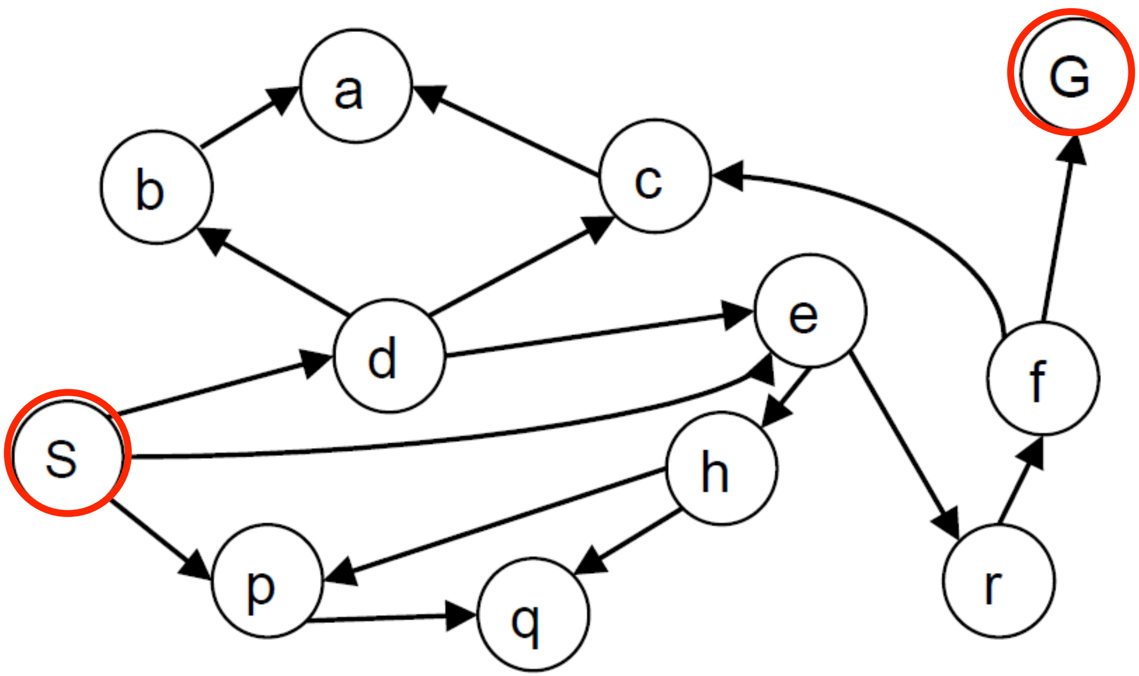
- **Time Complexity**
  - Number of nodes with path cost ≤ cost of optimal solution (C^*), $O(b^{1+\lfloor C^* / \epsilon \rfloor})$
  - This can be greater than $O(b^d)$: the search can explore long paths consisting of small steps before exploring shorter paths consisting of larger steps
- **Space Complexity**
  - $O(b^{1+\lfloor C^* / \epsilon \rfloor})$

→ *Compare with BFS when all cost steps are equal?*

Depth-first Search (DFS)

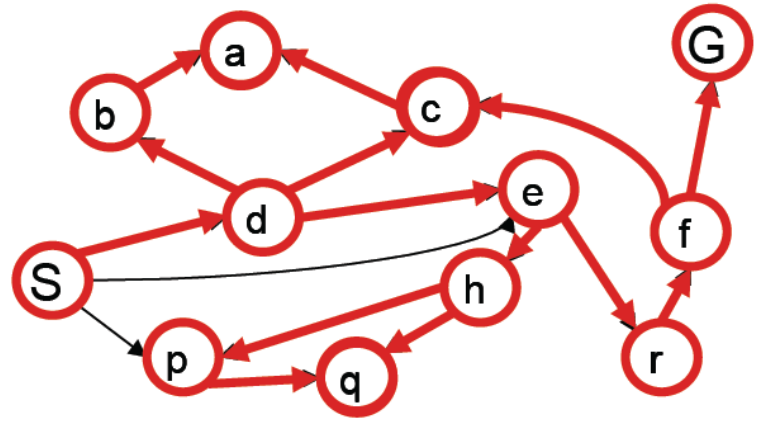
Depth-first search

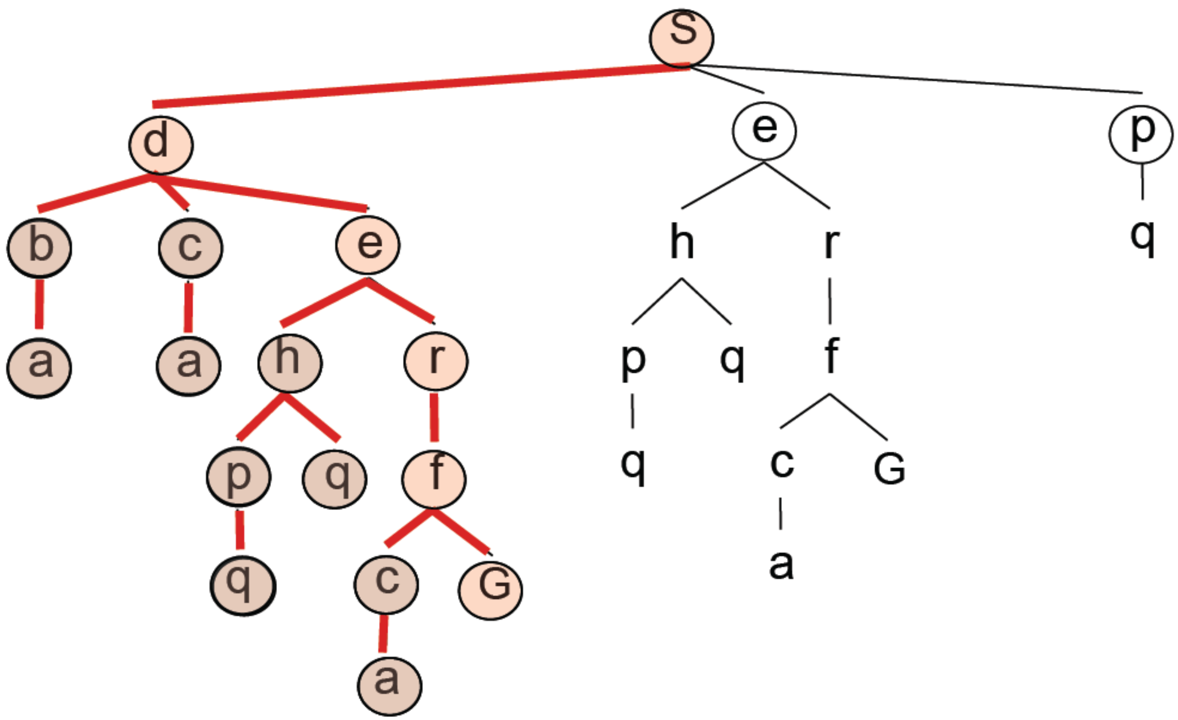
- Expand deepest unexpanded node
  - Repeated state: do not add if that state is on the path from the root to the current node
- Implementation: *frontier* is a LIFO Stack

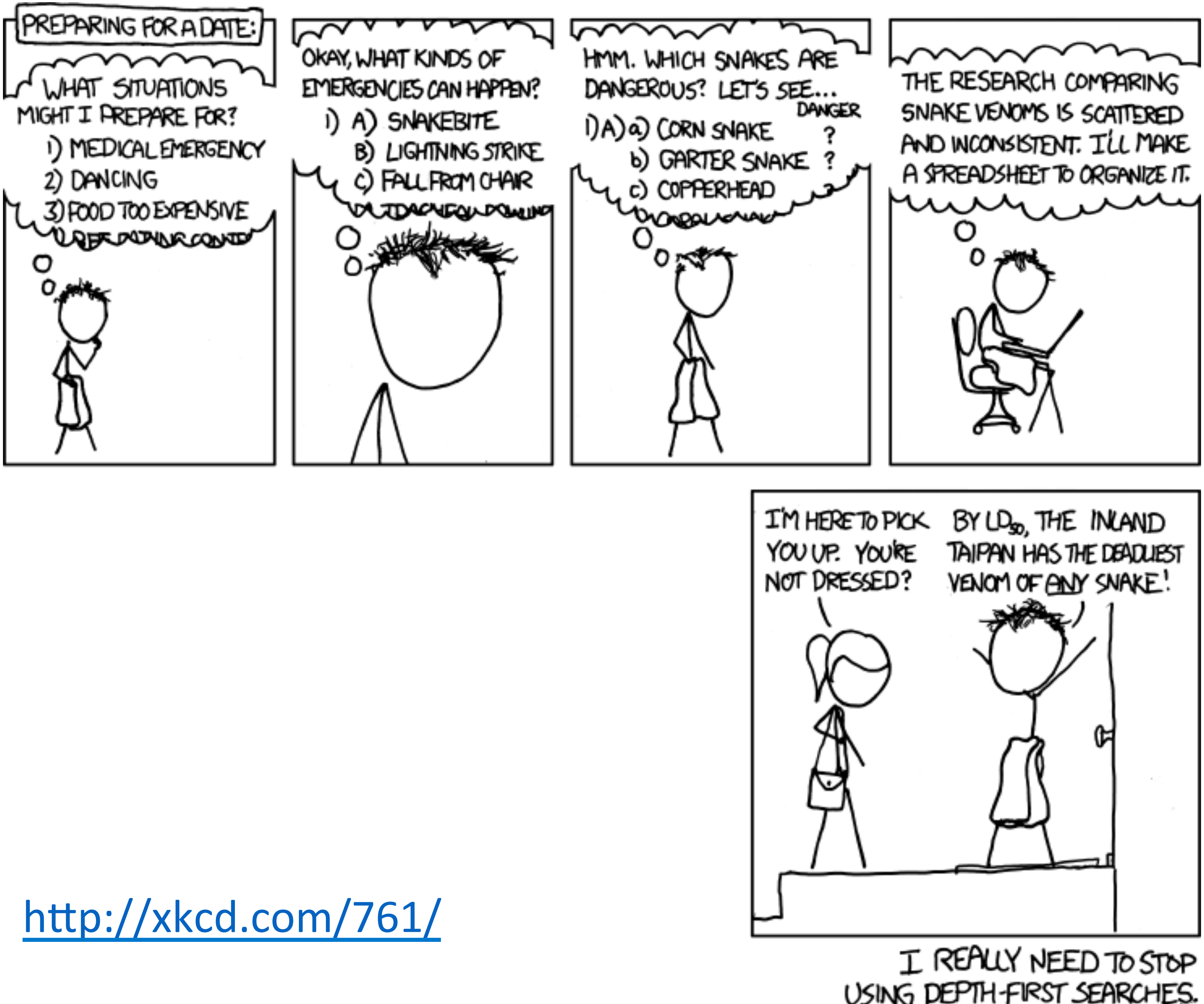


Depth-first search

- Expansion order:
  $(d,b,a,c,a,e,h,p,q,q,r,f,c,a,G)$







http://xkcd.com/761/

Evaluation of DFS

- ❑ Completeness
  - Fails in **infinite-depth spaces**, **spaces with loops**
  - Modify to avoid repeated states along path
    → complete in finite spaces
- ❑ Optimality
  - No – returns the first solution it finds

Evaluation of DFS

- Time Complexity
  - Could be the time to reach a solution at maximum depth m: $O(b^m)$
  - Terrible if m is much larger than d
  - But if there are lots of solutions, may be much faster than BFS
- Space Complexity
  - $O(bm)$, i.e., linear space!

Comparing BFS and DFS

- **Space complexity:**
  - DFS is linear space
  - BFS may store the whole search space.
- **Time complexity:** same, but
  - In the worst-case BFS is always better than DFS
  - Sometime, on the average DFS is better if:
    - many goals, no loops and no infinite paths
- **In general**
  - BFS is better if goal is not deep, if infinite paths, if many loops, if small search space
  - DFS is better if many goals, not many loops,
  - DFS is much better in terms of memory

Depth-limited Search (DLS)

Depth-limited Search (DLS)

DFS with depth limit *l*, i.e., nodes at depth *l* have no successors

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
     cutoff_occurred? ← false
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        result ← RECURSIVE-DLS(child, problem, limit − 1)
        if result = cutoff then cutoff_occurred? ← true
        else if result ≠ failure then return result
     if cutoff_occurred? then return cutoff else return failure
```

- Failure: no solution
- Cutoff: no solution within the depth limit

Depth-limited Search (DLS)

- Standard DFS, but tree is not explored below some depth-limit l
- Solves problem of infinitely deep paths with no solutions
  - But will be incomplete if solution is below depth-limit
- Depth-limit l can be selected based on problem knowledge
  - E.g., diameter of state-space:
    - E.g., max number of steps between 2 cities is 9 (Romania map)
  - But typically not known ahead of time in practice

Evaluation of DLS

- **Completeness:**
  - Maybe NOT if $l < d$
- **Optimality:**
  - NO if $l > d$
- **Time Complexity:**
  - $O(b^l)$
- **Space Complexity:**
  - $O(bl)$

DFS is a special case of DLS when $l = \infty$

Iterative deepening search (IDS)

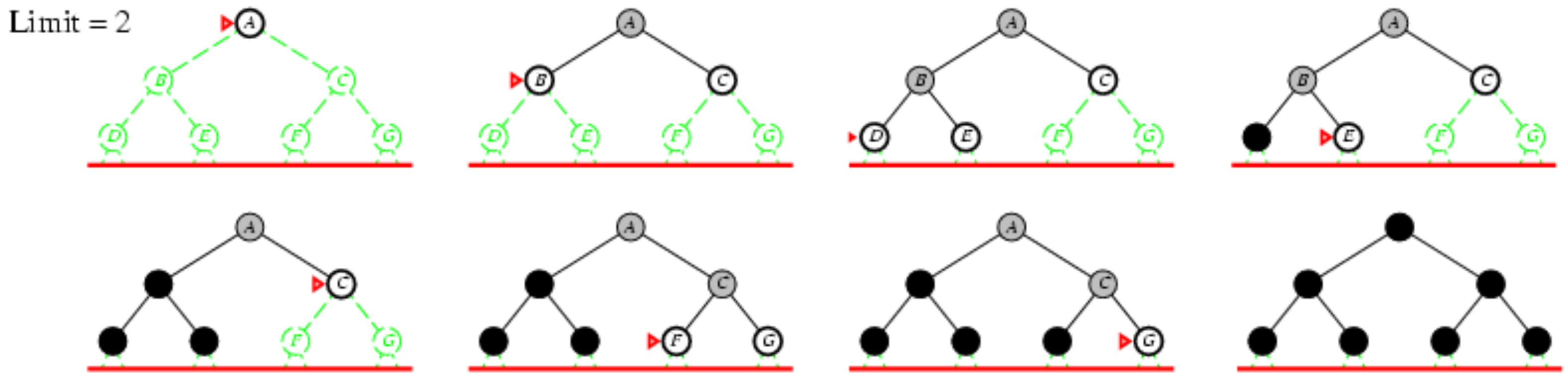
Iterative deepening search

- Use DFS as a subroutine
  1. Check the root
  2. Do a DFS searching for a path of length 1
  3. If there is no path of length 1, do a DFS searching for a path of length 2
  4. If there is no path of length 2, do a DFS searching for a path of length 3…

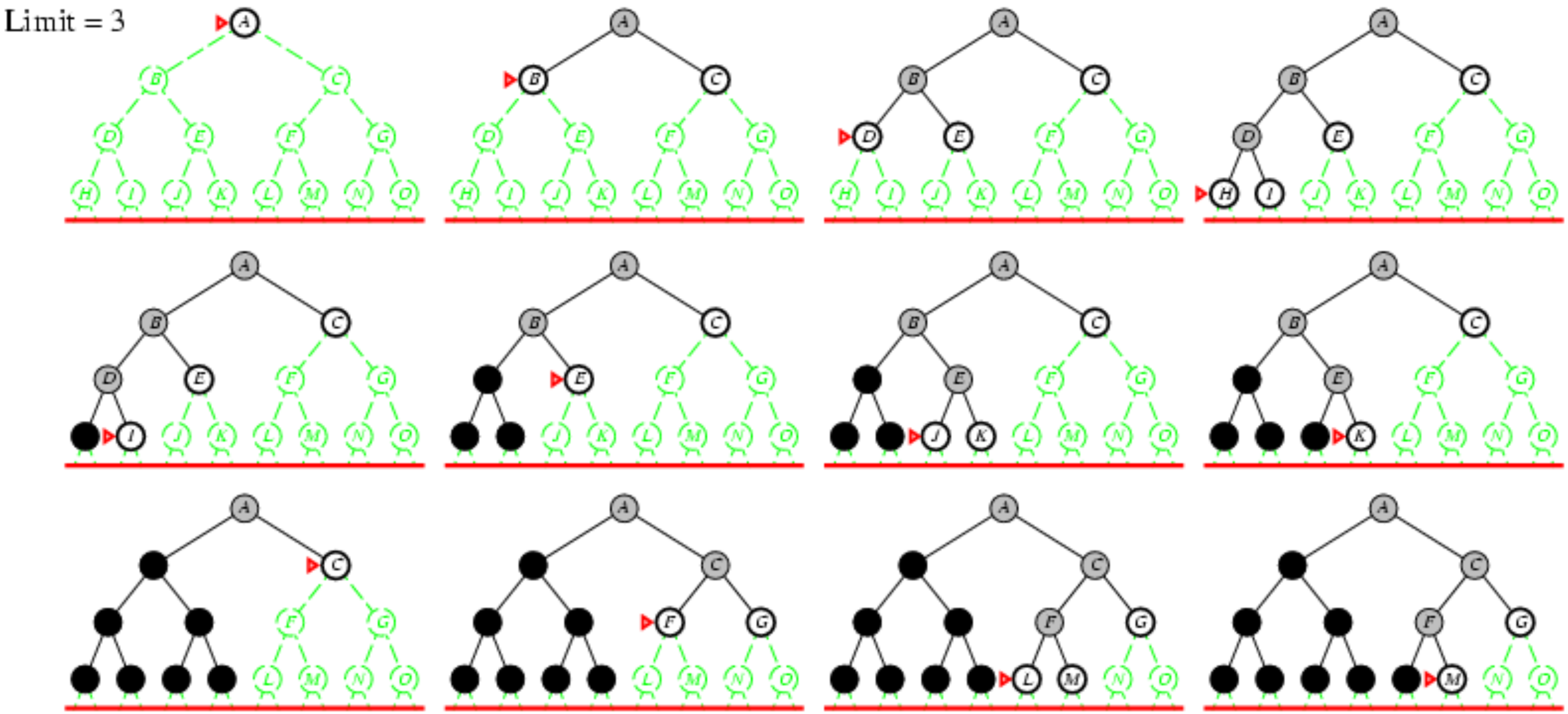
Iterative deepening search l =0

Iterative deepening search l =1

Iterative deepening search l =2



Iterative deepening search *l* =3



Evaluation of IDS

- Completeness
  - Yes
- Optimality
  - Yes, if step cost = 1
- Time Complexity
  - $(d+1)b^0 + d\ b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space Complexity
  - $O(bd)$

QUIZ

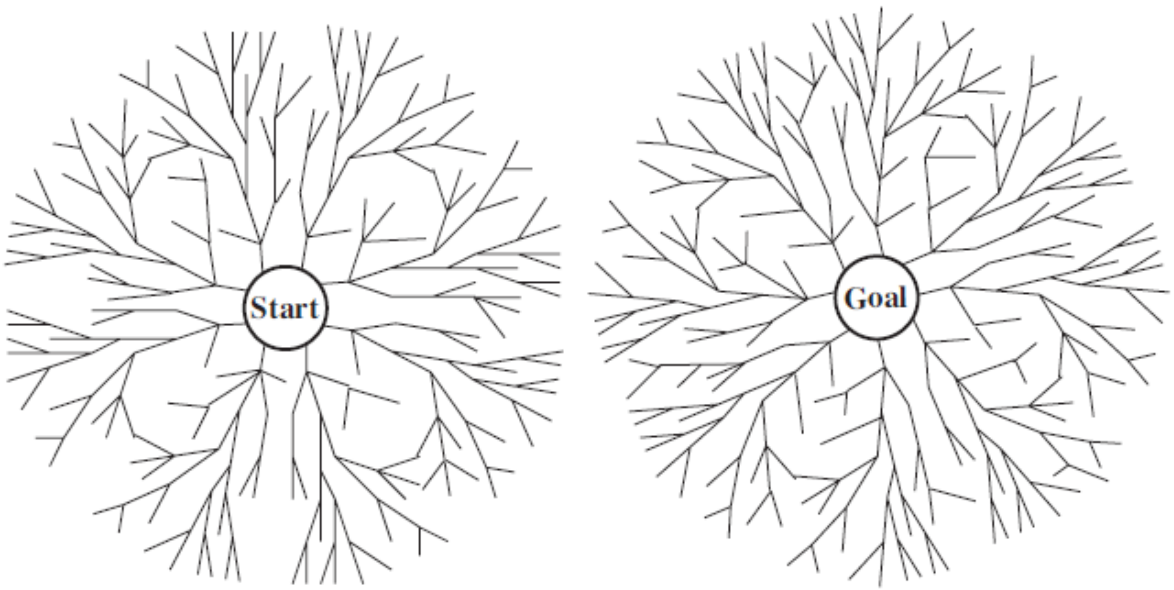
Iterative deepening search may seem wasteful because states are generated multiple times. However, it turns out this is not too costly. Why?

Bidirectional Search

Bidirectional Search

- Two simultaneous searches:
  - From the initial state towards
  - From the goal state backwards

  → Hoping that two searches meet in the middle



Bidirectional Search

- Time & Space Complexity:
  - $O(b^{d/2})$
- Goal test:
  - If the frontiers of two searches intersect?
- It sounds attractive, but what is the **tradeoff**?
  - Space requirement for the frontiers of at least 1 search
  - Not easy to search backwards (requires a method to compute predecessors)
    - In case there are more than 1 goals
    - Especially if the goal is an abstract description (no queen attacks another queen)

Summary

- Comparision between uninformed algorithms:

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Homework #2

- Read chapter **3** in the textbook (3rd edition, page 64-119)
- Answer the questions

Next class

- Chapter 2: Solving Problems by Searching (cont.)
  - Heuristic Search

Group Assignment 1

- Given a graph with nodes and links, we can find the shortest path using Dijkstra's algorithm. It is not hard. We have a polynomial time algorithm to do that.
- In AI we also solving the graph search problems.
- What is the differences between these two graph search strategies? (not AI and AI)
- What is special about AI Search Algorithms? Give a specific example to explain for your ideas.

Evaluation of IDS

- Number of nodes generated in a depth-limited search to depth *d* with branching factor *b*:

$$N_{DLS} = b^0 + b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$$

- Number of nodes generated in an iterative deepening search to depth *d* with branching factor *b*:

$$N_{IDS} = (d+1)b^0 + d\ b\text{^}^1 + (d-1)b\text{^}^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

- For $b = 10$, $d = 5$,
  - $N_{DLS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$
  - $N_{IDS} = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}456$

- Overhead = (123,456 - 111,111)/111,111 = 11%

Breadth-first search

Frontier (QUEUE)	Expanded	Select (POP)	Child	Goal Test
{}	{}		*S*	F
{*S*}	{}	*S*	*d*	F
{d}	{S}		*e*	F
{d,e}	{S}		*p*	F
{d,e,p}	{S}	*d*	*b*	F
{e,p,b}	{S,d}		*c*	F
{e,p,bc}	{S,d}		*e*	x
{e,p,b,c}	{S,d}	*e*	*h*	F
{p,b,c}	{S,d,e}		*r*	F