

Search

- Chapter 3 of R&N 2nd and 3rd edition is useful reading.
- Also: <https://artint.info/2e/html/ArtInt2e.Ch3.html>
- Chapter 4 of R&N 2nd and 3rd edition is worthwhile too.
- 2nd Edition of R&N is (temporarily) available at: <https://catalog.hathitrust.org/Record/004917484?type%5B%5D=all&lookfor%5B%5D=Artificial%20Intelligence%3A%20A%20Modern%20Approach%20&ft=ft>

(R&N = Russell and Norvig, Artificial Intelligence: a Modern Approach)

Search

Credits: We're often revising and updating slides. Search slides are drawn from or inspired by a multitude of sources including me and ...

Faheim Bacchus,

Sheila McIlraith,

Andrew Moore,

Hojjat Ghaderi,

Craig Boutillier,

Jurgen Strum,

Shaul Markovitch,

Daniel Bauer

Thank you for sharing!!

Search

Successful

- Many other AI problems can be successfully solved by search
- Outperform humans in some areas (e.g. games)

Practical

- Many problems don't have specific algorithms for solving them. Casting as search problems is often the easiest way of solving them.
- Search can also be useful in approximation (e.g., local search in optimization problems).
- Problem specific heuristics provides search with a way of exploiting extra knowledge.

Some critical aspects of “intelligent” behaviour, e.g., planning, can be cast as search.

A Search Problem:

How do we plan our holiday?

- We must take into account various preferences and constraints to develop a schedule.
- An important technique in developing such a schedule is “**hypothetical**” reasoning.
- Example: I’ m on holiday in B.C.
 - If I fly into Vancouver and drive a car to Whistler, I’ ll have to drive on the roads at night. How desirable is this?
 - If I am in Whistler and leave at 6:30am, I can arrive in Kamloops by lunchtime.

A Search Problem:

How do we plan our holiday?

- This kind of hypothetical reasoning involves asking
 - what state will I be in after taking certain actions, or after certain sequences of events?
- From this we can reason about particular sequences of events or actions one should try to bring about to achieve a desirable state.
- Search is a computational method for capturing a particular version of this kind of reasoning.



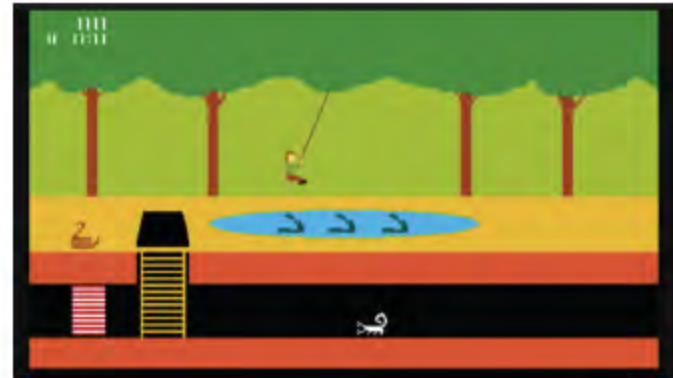
Search Problems



1	2	3
6	7	
8	5	4



More search problems



Limitations of Search

Search only shows how to solve the problem once we have it correctly formulated.

The Formalism

To formulate a problem as a search problem we need the following components:

1. a **state space** over which to search. The state space necessarily involves **abstracting** the real problem.
2. an **initial state** that best represents your current state.
3. a **desired (or goal) condition** you want to achieve.
4. **actions (or successor functions)** that allow move one from state to state. The actions are abstractions of actions you could actually perform.

Optional ingredients:

1. **costs**, which represent the cost of moving from state to state (taking an **action**, advancing to a successor state).
2. **Heuristics**, to help guide the search process.

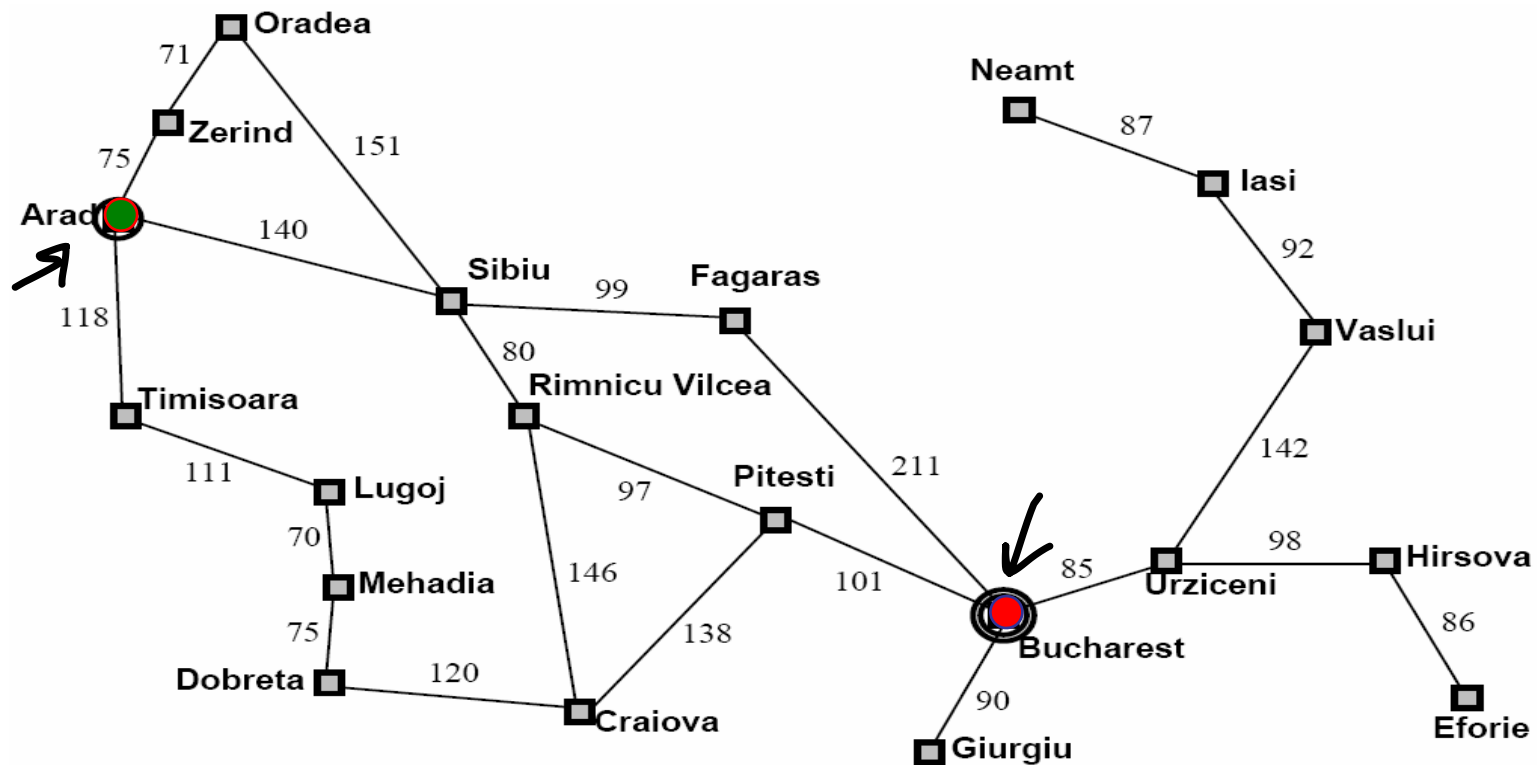
A solution

Once you have a formalized search problem, there are a number of algorithms one can use to solve it.

A **solution** is a **sequence of actions** or moves that can transform your current state into a state where desired (or goal) conditions hold.

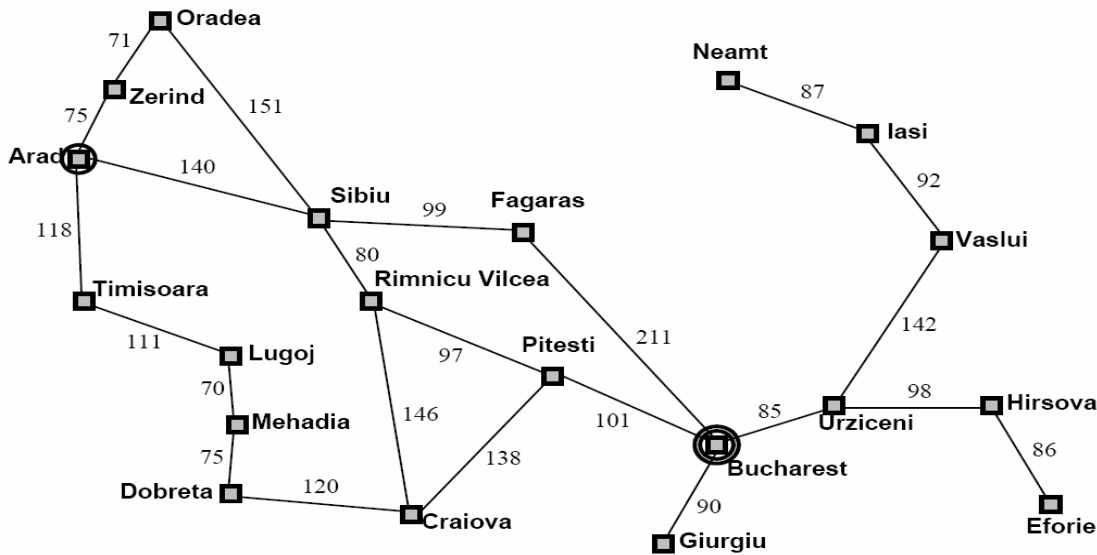
Example 1: Romania Travel

Currently in **Arad**, need to get to **Bucharest** ASAP. Can we formalize this search?



Example 1: Romania Travel

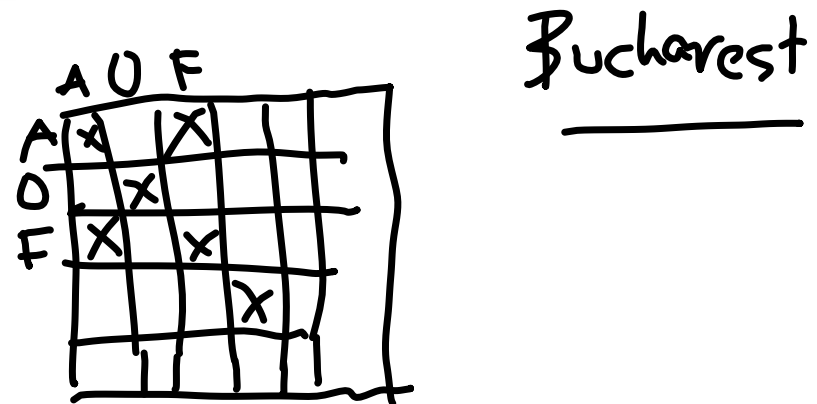
Currently in **Arad**, need to get to **Bucharest** ASAP. What is the state space?



- state space: *Cities on map*
- actions (successor functions): *See below...*
- initial state: Arad
- desired (or goal) condition: Bucharest

What is a solution?

Sequence of Cities, e.g.
A, S, F, B



Here, we can indicate successors using an adjacency matrix. This is not possible if our search space is infinite, however.

Example 2: Water Jugs

We have a 3 gallon (liter) jug and a 4 gallon jug. We can fill either jug to the top from a tap, we can empty either jug, or we can pour one jug into the other (at least until the other jug is full).

–state space: (X, Y) number pairs

–actions (successor functions): various
Fill 3 Gallon (state) e.g.

Fill 4 Gallon (state) \rightarrow State

–initial state:

$(0, 0)$

Empty 4 Gallon
Empty 3 Gallon

–desired (or goal) condition: integer pair

4 to 3 3 to 4

What is a solution?

A sequence of actions + resulting states

Example 2: Water Jugs

We have a 3 gallon (liter) jug and a 4 gallon jug. We can fill either jug to the top from a tap, we can empty either jug, or we can pour one jug into the other (at least until the other jug is full).

–**state space**: pairs of numbers (gal3, gal4) where gal3 is the number of gallons in the 3 gallon jug, and gal4 is the number of gallons in the 4 gallon jug.

–**actions (successor functions)**: Empty-3-Gallon, Empty-4-Gallon, Fill-3-Gallon, Fill-4-Gallon, Pour-3-into-4, Pour 4-into-3.

–**initial state**: Various, e.g., (0,0)

–**desired (or goal) condition**: Various, e.g., (0,2) or (*, 3) where * means we don't care

Reflections on the Water Jug Problem

- Can we reach all states from any given start state?

No, ex: (1,2) is not reachable from (0,0).

Also, if initial state is integer pair, reachable states are ints.

- Will all actions result in a change of state? No.

Empty3Gallon((0,0)) returns (0,0)

Example 3: The 8-Puzzle

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

Rules: Slide a tile into the blank spot. Get numbers in order, with blank spot at bottom right.

Example 3: The 8-Puzzle

7	2	4
5	B	6
8	3	1

Start State

1	2	3
4	5	6
7	8	B

Goal State

- state space:
Configs of 8 ints + "Blank"
- actions (successor functions):
Move "Blank" up, down, l, r
- initial state:
Configuration @ left
- desired (or goal) condition:
Configuration @ right

What is a solution?

Sequence of actions + resulting states.

Example 3: The 8-Puzzle

- **state space**: the different configurations of the tiles. *9!*
How many different states? ≈ 362880
- **actions (or successor functions)**: moving the blank up, down, left, right. *Can every action be performed in every state? No!*
- **initial state**: e.g., state shown on previous slide.
- **desired (or goal) condition**: a state where tiles are in the positions shown on the previous slide.

Solution will be a sequence of moves of the blank that transform the initial state to a goal state.

Reflections on the 8-Puzzle Problem

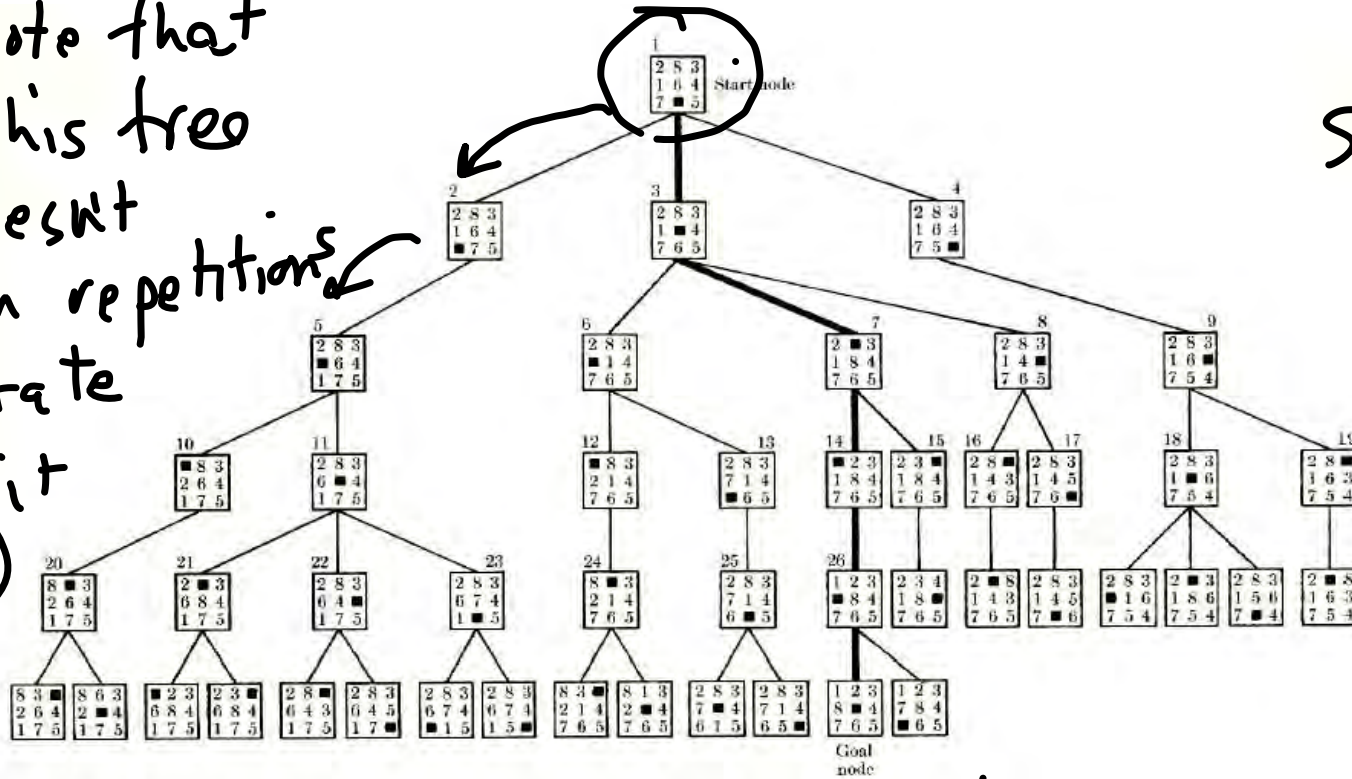
- How many states are there? $9!$
- Note:
 - the state space is divided into two disjoint parts.
 - only when the blank is in the middle are all actions possible.
 - the goal condition is satisfied by only a single state. If the goal were to end with the 8 in the upper left corner, how many states would satisfy the goal? $8!$

Search Space for 8-Puzzle Problem

initial state @ root

Note that this tree doesn't contain repetitions of state (but it could)

Search tree helps us assess complexity of search

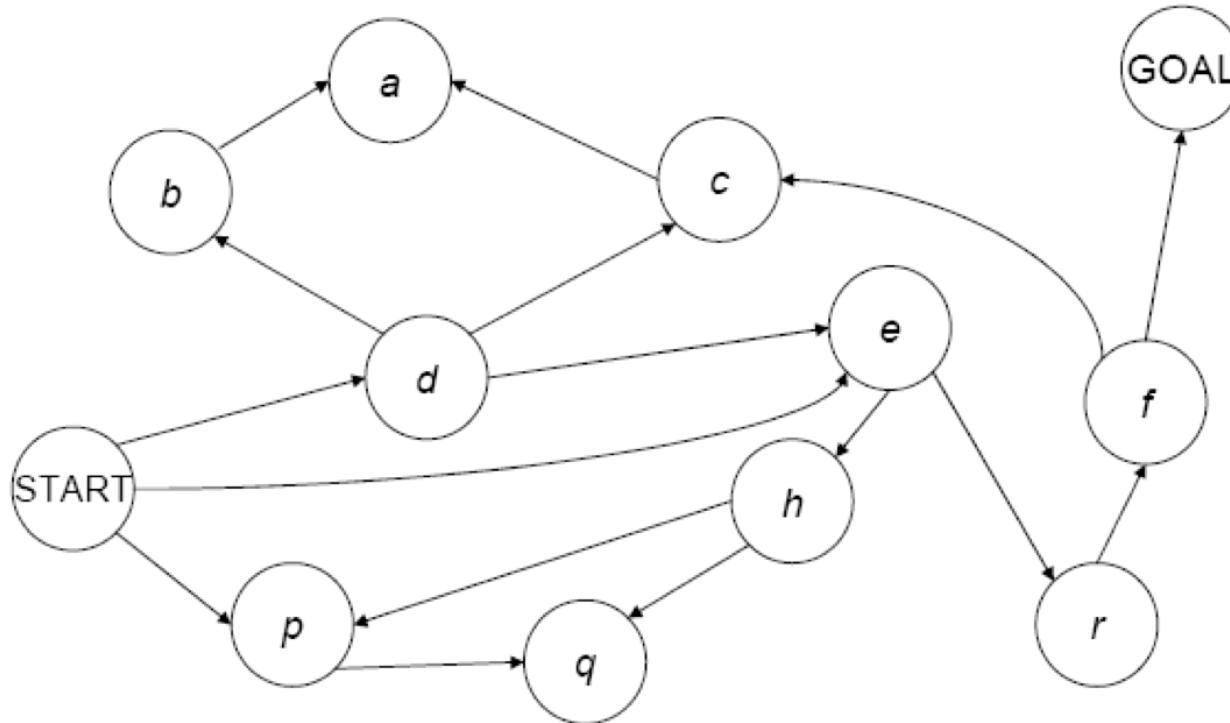


↑ goal state @ leaf

More complex situations

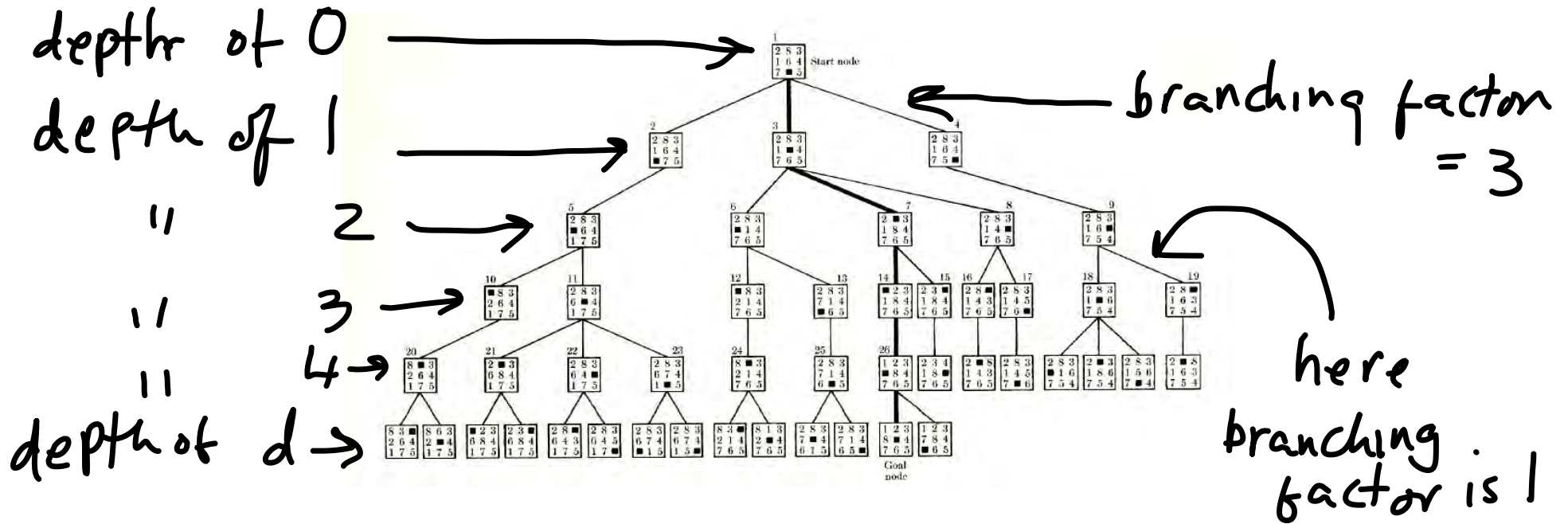
- Sometimes, actions may lead to multiple states, like flipping a coin.
- Other times, we may not be sure of a given state (prize is behind door 1, 2, or 3). In these situations, we might want to consider how **likely** different states and action outcomes are.
- Later we will see some techniques for reasoning under **uncertainty**.
- Some of these will be **probabilistic**, i.e. they will assign probabilities to given outcomes.

Drawing a Search Problem: Graphical Representation



It can sometimes be useful to represent a search problem as a graph containing **Vertices (V)** and **Edges (E)**. Vertices can be used to represent states in the search space and edges to represent transitions resulting from actions (or successor functions). This assumes a finite search space.

Graphical Representation of Search (Tree)



A **search tree** reflects the behaviour of our algorithm as it walks through a search problem. Its attributes include a solution depth (d) and maximum branching factor (b). Note that the same state may appear many times in the tree.

Algorithms for Search

Inputs:

- a specified **initial state** (a specific world state)
- a **successor** function $S(x) = \{\text{set of states that can be reached from state } x \text{ via a single action}\}$.
- a **goal test** a function that can be applied to a state and returns true if the state satisfies the goal condition.
- An **action cost** function $C(x,a,y)$ which determines the cost of moving from state x to state y using action a . ($C(x,a,y) = \infty$ if a does not yield y from x). Note that different actions might generate the same move of $x \rightarrow y$.

Algorithms for Search

Outputs:

- a sequence of states leading from the initial state to a state satisfying the goal test.
- The sequence might be optimal in cost for some algorithms, optimal in length for some algorithms, or it come with no optimality guarantee.

A Searching Template

- To explore the state space during a search, we will iteratively apply the successor function to the states we discover.
- Each time, the successor function $S(x)$ will yield a set of states that can be reached from x via any single action.
- As we search, we can annotate states by the action used to obtain them in order to keep a record of paths to a state:
 - $S(x) = \{ \langle y, a \rangle, \langle z, b \rangle \}$
arrive at y via action a , arrive at z via action b .
 - $S(x) = \{ \langle y, a \rangle, \langle y, b \rangle \}$
arrive at y via action a , also y via alternative action b .
- We can also reference each state's origin as we search (i.e., the preceding state).
- States may also be annotated with the cost of the path traversed in order to arrive at it.

A Searching Template

- We put nodes (or states) we we haven't yet explored or expanded, but want to explore, in a list called the **Frontier (or Open)**.
- Initially, all that is in the Frontier is the **initial state**.
- At each iteration, we pull a node from the Frontier, apply **S(x)**, and insert children back into the Frontier.

=

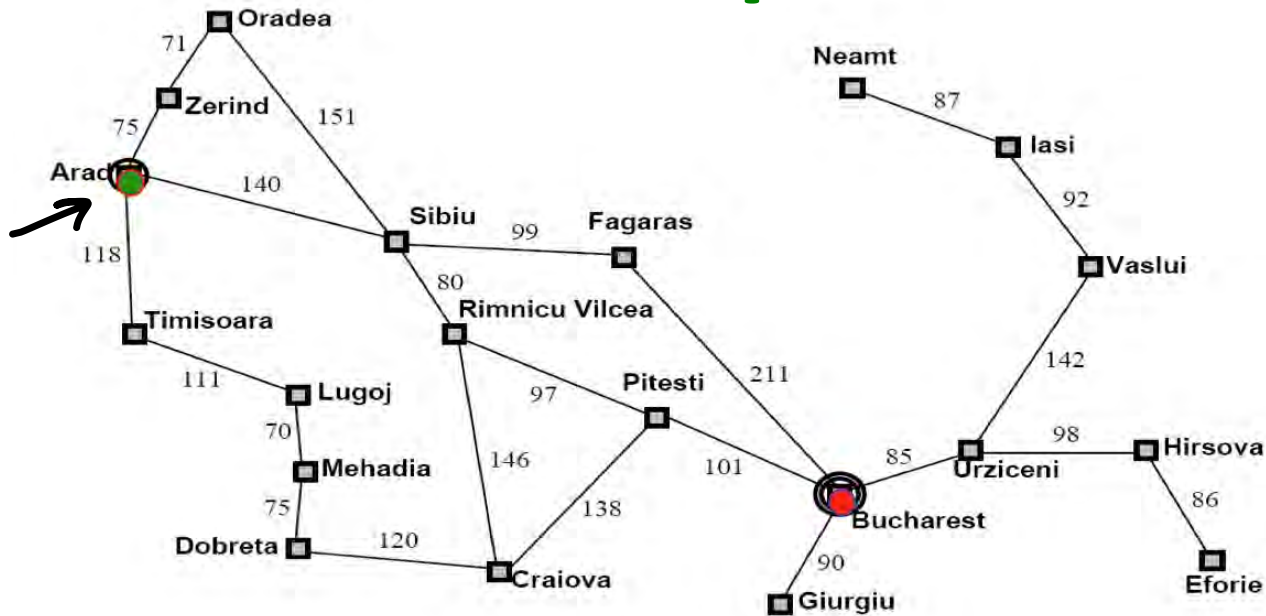
Frontier =

{(initial state)}

```
TreeSearch(Frontier, Successors, Goal? )  
  If Frontier is empty return failure  
  Curr = select state from Frontier  
  If (Goal?(Curr)) return Curr.  
  Frontier' = (Frontier - {Curr}) U Successors(Curr)  
  return TreeSearch(Frontier', Successors, Goal?)
```

goal test
state space

Example

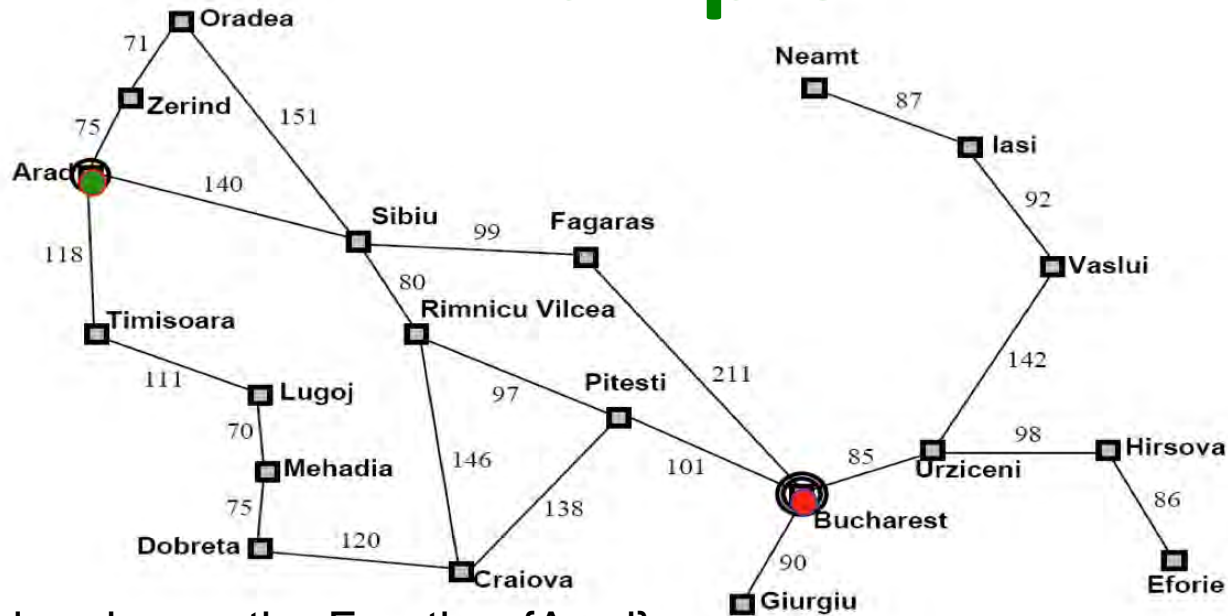


1. Initial nodes on the Frontier: {Arad}.
2. Expand **Arad**: {Z<A>, T<A>, **S<A>**},
3. Expand **Sibiu**: {Z<A>, T<A>, **A<S,A>**, O<S,A>, **F<S,A>**, R<S,A>}
4. Expand **Fagaras**: {Z<A>, T<A>, A<S,A>, O<S,A>, R<S,A>, **S<F,S,A>**, **B<F,S,A>**}

green paths are selected

Solution is now on frontier; cost of this solution is $140+99+211 = 450$ //*

Example



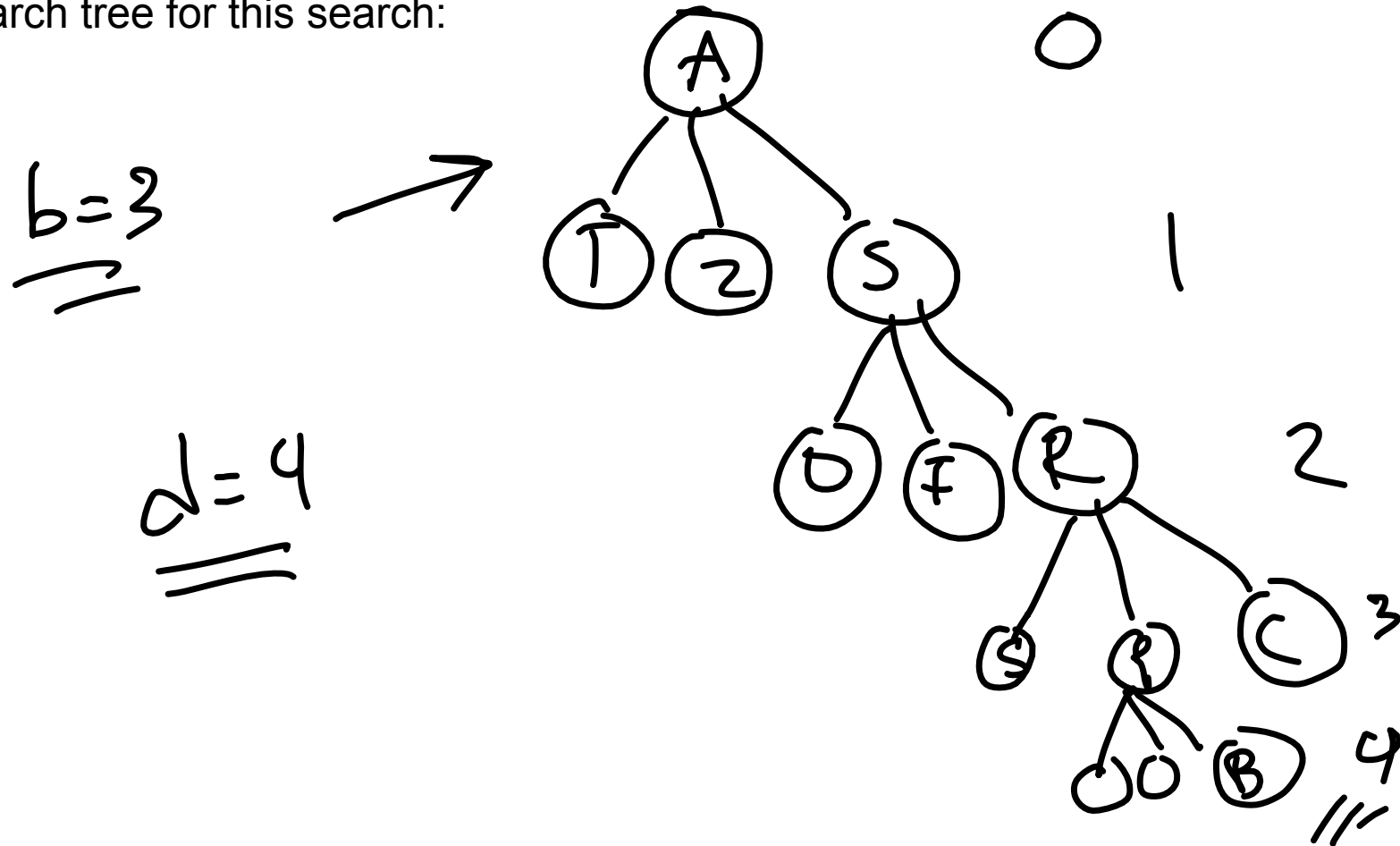
1. Initial nodes on the Frontier: {Arad}.
2. Expand **Arad**: {Z<A>, T<A>, **S<A>**}
3. Expand **Sibiu**: {Z<A>, T<A>, A<S,A>, O<S,A>, F<S,A>, **R<S,A>**}
4. Expand **R.V.**: {Z<A>, T<A>, A<S,A>, O<S,A>, R<S,A>, S<R,S,A>, **P<R,S,A>**, C<R,S,A>}
5. Expand **Pitesti**: {Z<A>, T<A>, A<S,A>, O<S,A>, R<S,A>, S<R,S,A>, P<R,S,A>, C<R,S,A>, R<P,R,S,A>, C<P,R,S,A>, **B<P,R,S,A>**}

← alternate paths selected

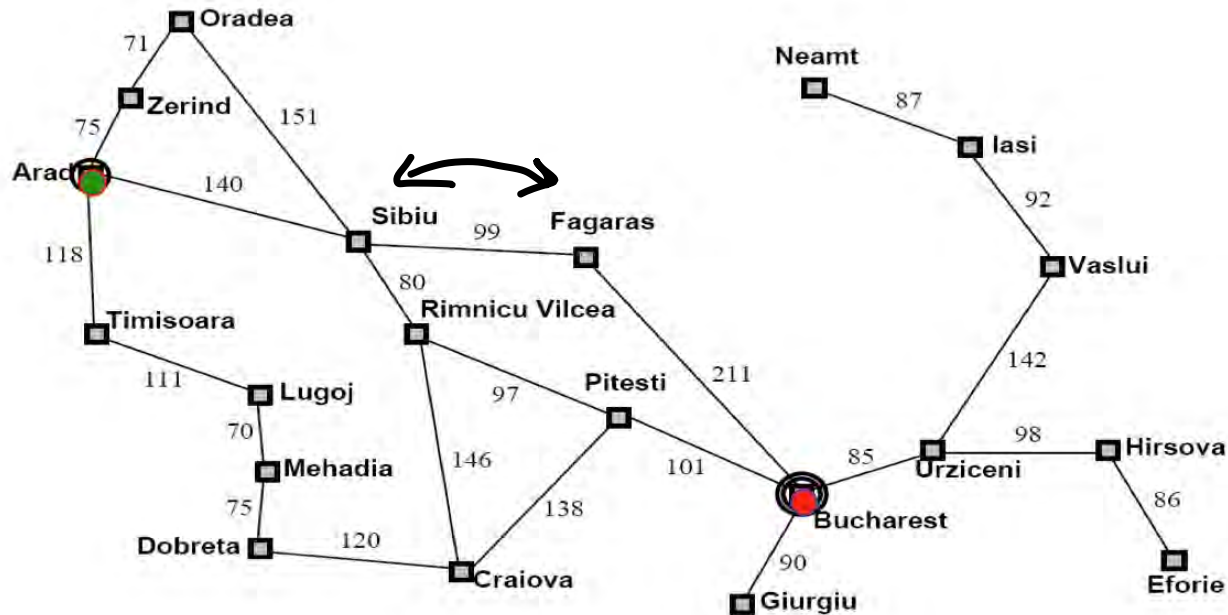
Solution is now on frontier; cost of this solution is **140+80+97+101= 418** ~~118~~

1. Initial nodes on the Frontier: {Arad}.
2. Expand **Arad**: {Z<A>, T<A>, **S<A>**}.
3. Expand **Sibiu**: {Z<A>, T<A>, A<S,A>, O<S,A>, F<S,A>, **R<S,A>**}.
4. Expand **R.V.:** {Z<A>, T<A>, A<S,A>, O<S,A>, R<S,A>, S<R,S,A>, **P<R,S,A>**, C<R,S,A>}.
5. Expand **Pitesti:** {Z<A>, T<A>, A<S,A>, O<S,A>, R<S,A>, S<R,S,A>, P<R,S,A>, C<R,S,A>, R<P,R,S,A>, C<P,R,S,A>, **B<P,R,S,A>**}.

Let's draw the search tree for this search:



Reflections on Example



1. In this problem, the Frontier here contains a set of **paths**, not just **states**.
2. **Cycles** can create problems
3. The **order** states are selected from the Frontier has a **critical** effect on:
 - Whether or not a solution is found
 - The **cost** of the solution that is found.
 - The **time** and **space** required by the search.

Critical Properties of Search

- **Completeness**: will the search always find a solution if a solution exists?
- **Optimality**: will the search always find the least cost solution? (when actions have costs)
- **Time complexity**: what is the maximum number of nodes (paths) that can be expanded or generated?
- **Space complexity**: what is the maximum number of nodes (paths) that have to be stored in memory?

Search tree helps here.

Uninformed Search Strategies

- These are strategies that adopt a fixed rule for selecting the next state to be expanded.
- The rule remains the same for any search problem being solved; it does not change.
- These strategies do not take into account any domain specific information about the particular search problem.
- Uninformed search techniques:
 - Breadth-First, Uniform-Cost, Depth-First, Depth-Limited,
Iterative-Deepening

Selecting Nodes on the Frontier

Selection can be achieved by employing an appropriate ordering of the frontier set, i.e.:

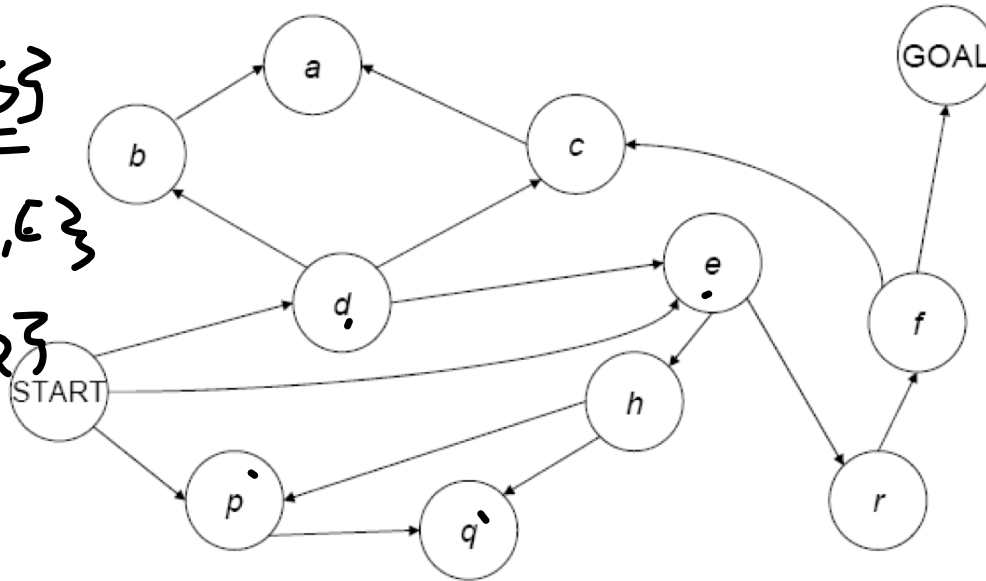
1. Order the elements on the Frontier.

2. Always select the first element.



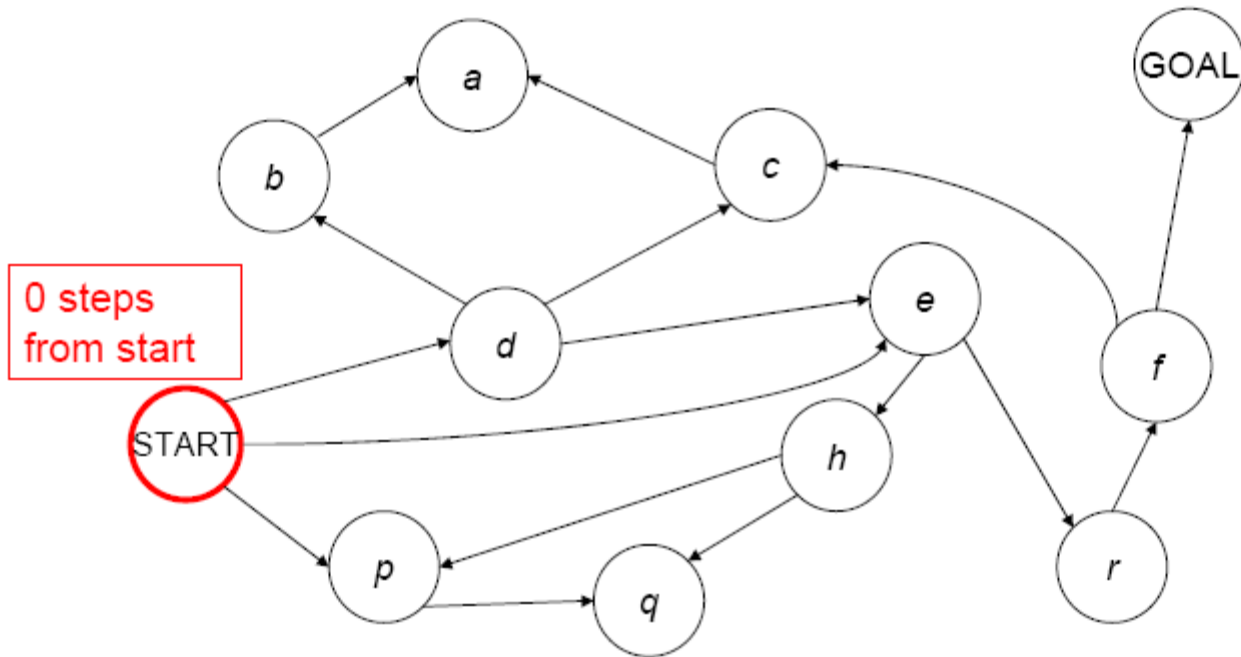
Breadth First Search

- 0. Frontier = {S}
- 1. Frontier = {P, D, E}
- 2. Frontier = {D, E, Q}



1. Place Start in the Frontier.
2. Expand all nodes reachable from Start in 1 step, but not more than 1; add path to back of Frontier list.
3. Expand all nodes reachable from Start in 2 step, but not more than 2; add path to back of Frontier list.
4. Expand all nodes reachable from Start in 3 step, but not more than 3; add to path back of Frontier list.
5. And so on

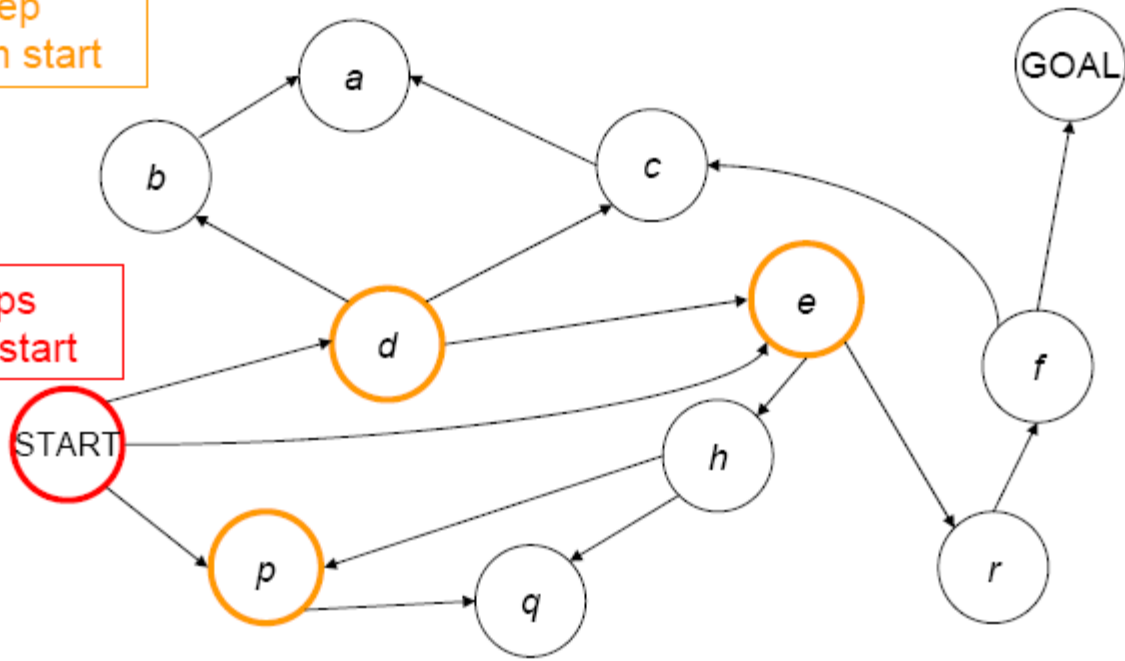
Breadth-first Search



Breadth-first Search

1 step from start

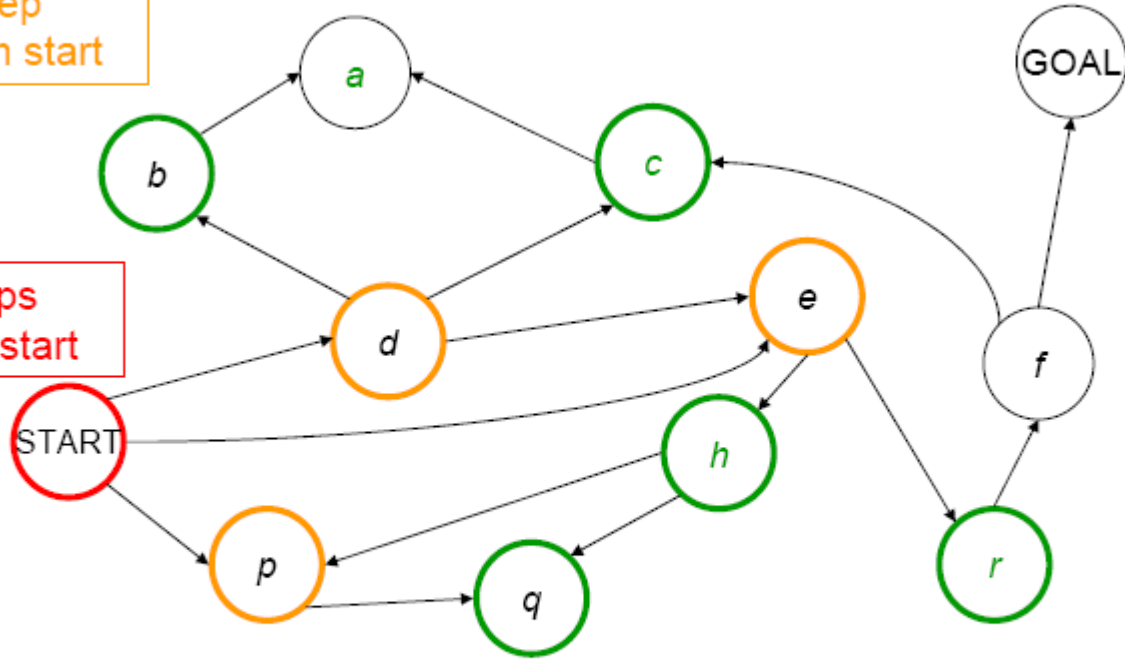
0 steps from start



Breadth-first Search

1 step from start

0 steps from start



2 steps from start

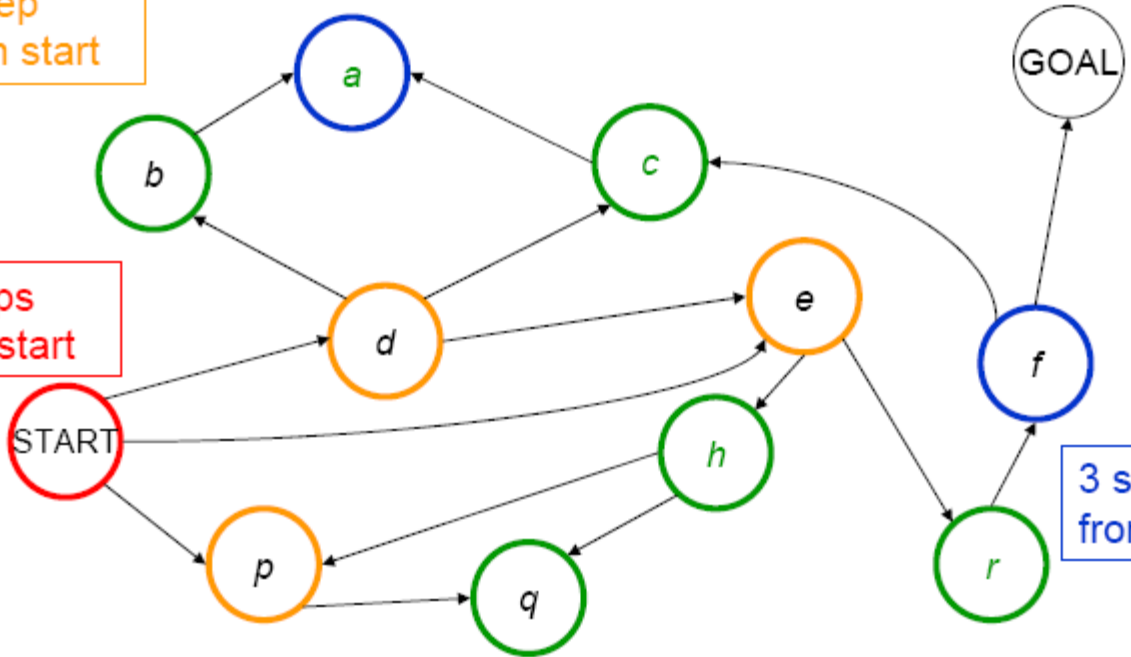
Breadth-first Search

1 step from start

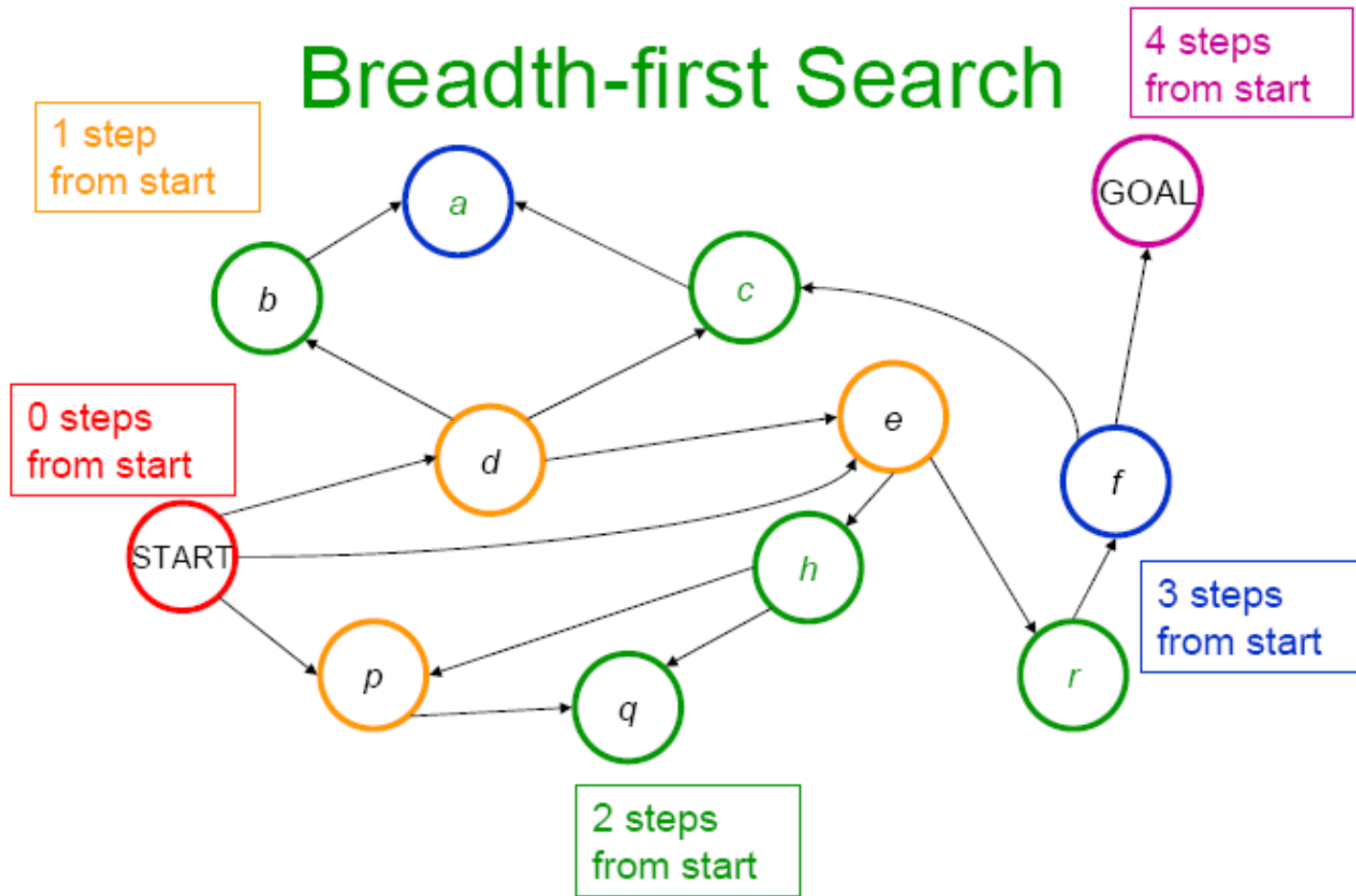
0 steps from start

3 steps from start

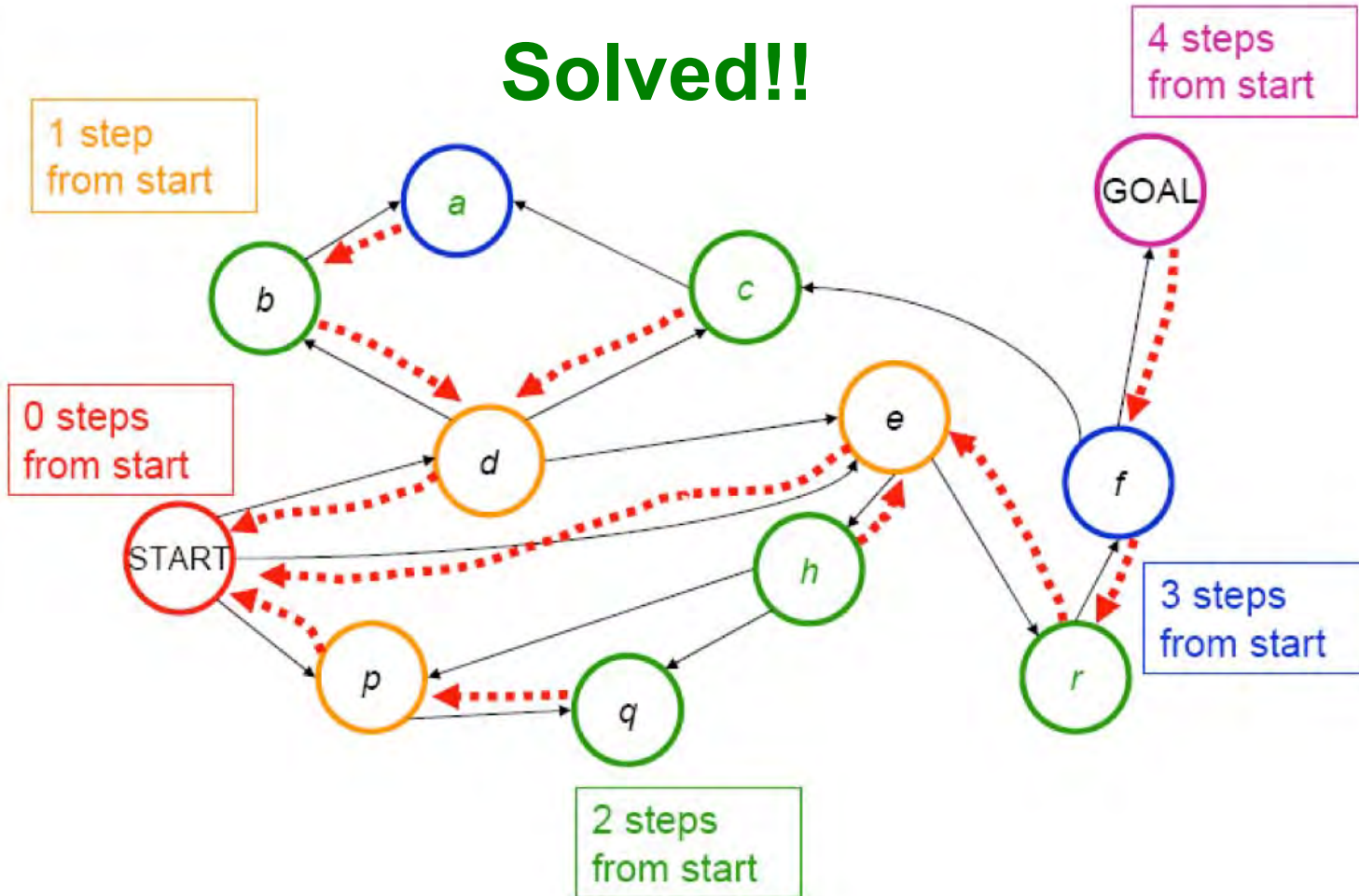
2 steps from start



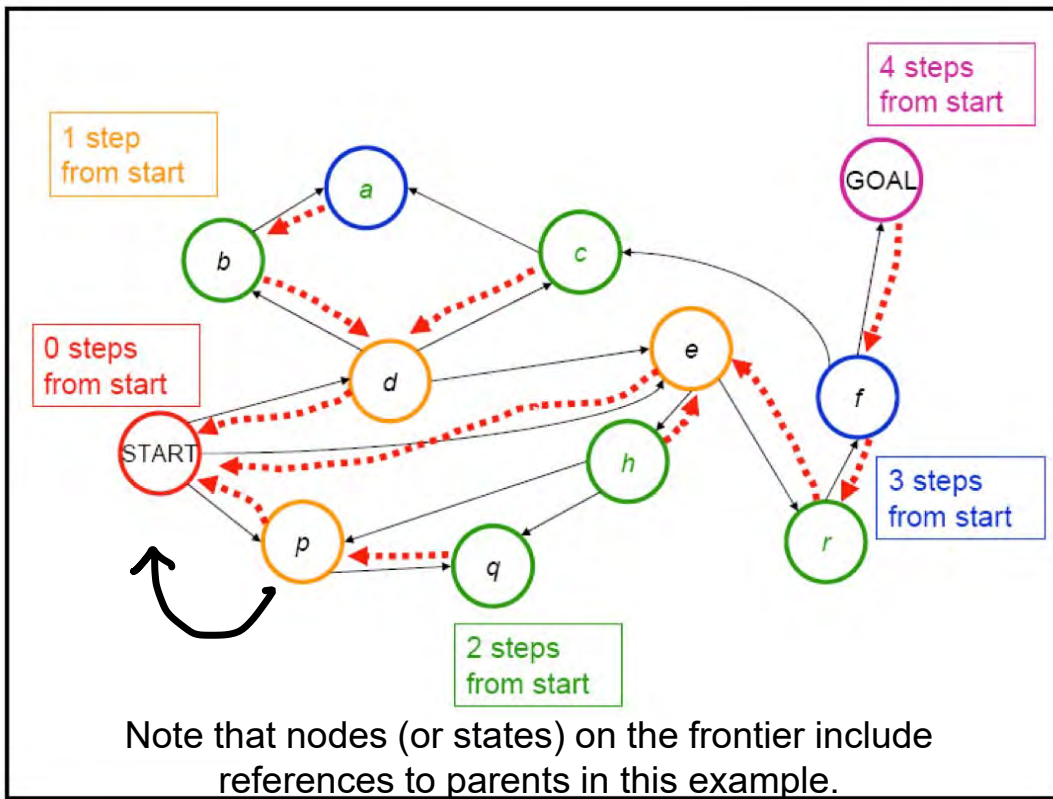
Breadth-first Search



Solved!!



Note that nodes (or states) on the frontier include references to parents in this example.



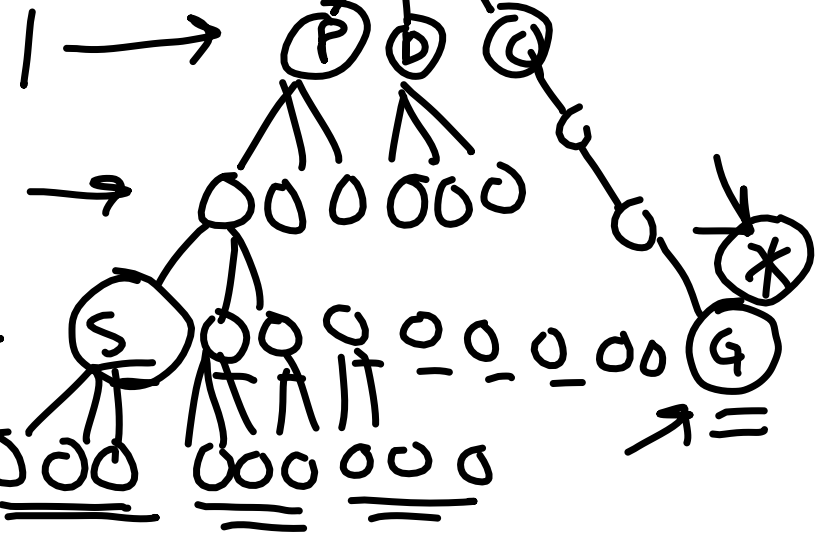
0 Frontier = {S}

1 F = {P, D, E}

2 F = {D, E, G}

3. F = {E, Q, E, C, B} ?

Q → S ←



BFS for the Water Jug Problem

initial state = (0,0), goal state = (*,2), actions (successor functions): Empty-3-Gallon, Empty-4-Gallon, Fill-3-Gallon, Fill-4-Gallon, Pour-3-into-4, Pour 4-into-3.

1. Frontier = {<(0,0)>}

Here, we store complete paths on the frontier.

BFS for the Water Jug Problem

initial state = (0,0), goal state = (*,2), actions (successor functions): Empty-3-Gallon, Empty-4-Gallon, Fill-3-Gallon, Fill-4-Gallon, Pour-3-into-4, Pour 4-into-3.

1. Frontier = {<(0,0)>}

2. Frontier = {<(0,0),(3,0)>, <(0,0),(0,4)>}

Here, we store complete paths on the frontier.

BFS for the Water Jug Problem

initial state = (0,0), goal state = (*,2), actions (successor functions): Empty-3-Gallon, Empty-4-Gallon, Fill-3-Gallon, Fill-4-Gallon, Pour-3-into-4, Pour 4-into-3.

1. Frontier = {<(0,0)>}

2. Frontier = {<(0,0),(3,0)>, <(0,0),(0,4)>}

3. Frontier = {<(0,0),(0,4)>, <(0,0),(3,0),(0,0)>, <(0,0),(3,0),(3,4)>, <(0,0),(3,0),(0,3)>}

Here, we store complete paths on the frontier.

BFS for the Water Jug Problem

initial state = (0,0), goal state = (*,2), actions (successor functions): Empty-3-Gallon, Empty-4-Gallon, Fill-3-Gallon, Fill-4-Gallon, Pour-3-into-4, Pour 4-into-3.

1. Frontier = {<(0,0)>}

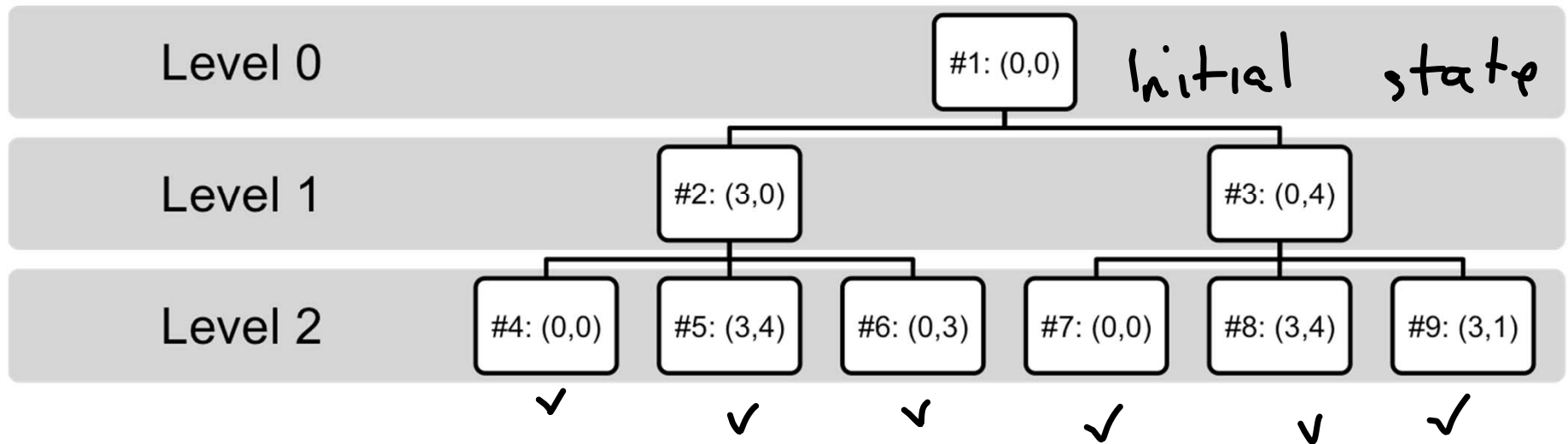
2. Frontier = {<(0,0),(3,0)>, <(0,0),(0,4)>}

3. Frontier = {<(0,0),(0,4)>, <(0,0),(3,0),(0,0)>, <(0,0),(3,0),(3,4)>, <(0,0),(3,0),(0,3)>}

4. Frontier = {<(0,0),(3,0),(0,0)>, <(0,0),(3,0),(3,4)>, <(0,0),(3,0),(0,3)>, <(0,0),(0,4),(0,0)>, <(0,0),(0,4),(3,4)>, <(0,0),(0,4),(3,1)>}

Here, we store complete paths on the frontier.

BFS for the Water Jug Problem



In the tree above we order the states explored; paths to states are represented by the path from the root to that states.

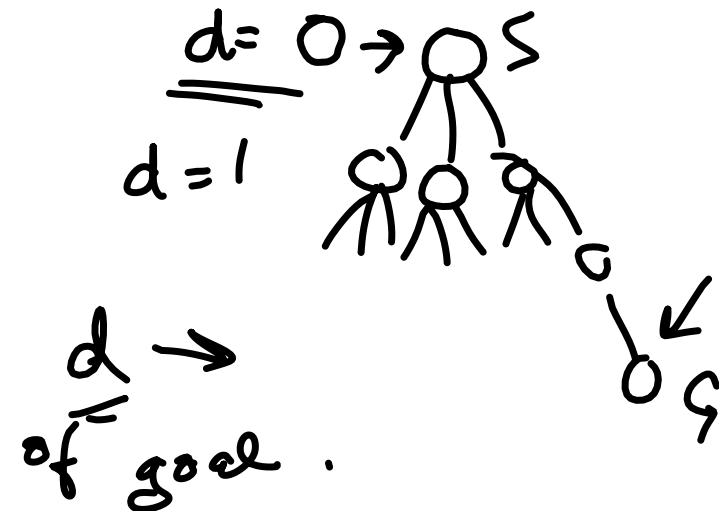
Breadth-First Search explores the search space level by level.

Breadth-First Properties

The tree representation enables us to measure time and space complexity.

- let b be the maximum number of successors of any node (i.e. the maximal **branching factor**).
- let d be the depth of the shortest solution.
 - Root at depth 0 generates a path of length 1
 - So $d = \text{length of path} - 1$

here, b is 3 →



Breadth-First Properties

Completeness?

yes.

we eventually search all paths of length d , and will find a solution if one exists.

Optimality?

yes, shortest path is returned

The first path we discover to a goal will be of length d . There is no shorter solution as we exhausted all paths of length less than d .

Record your answer here: <https://forms.gle/2krMifrptgaEDYyw9>

What is the Time Complexity?

$$b^0 + b^1 + b^2 + \dots$$

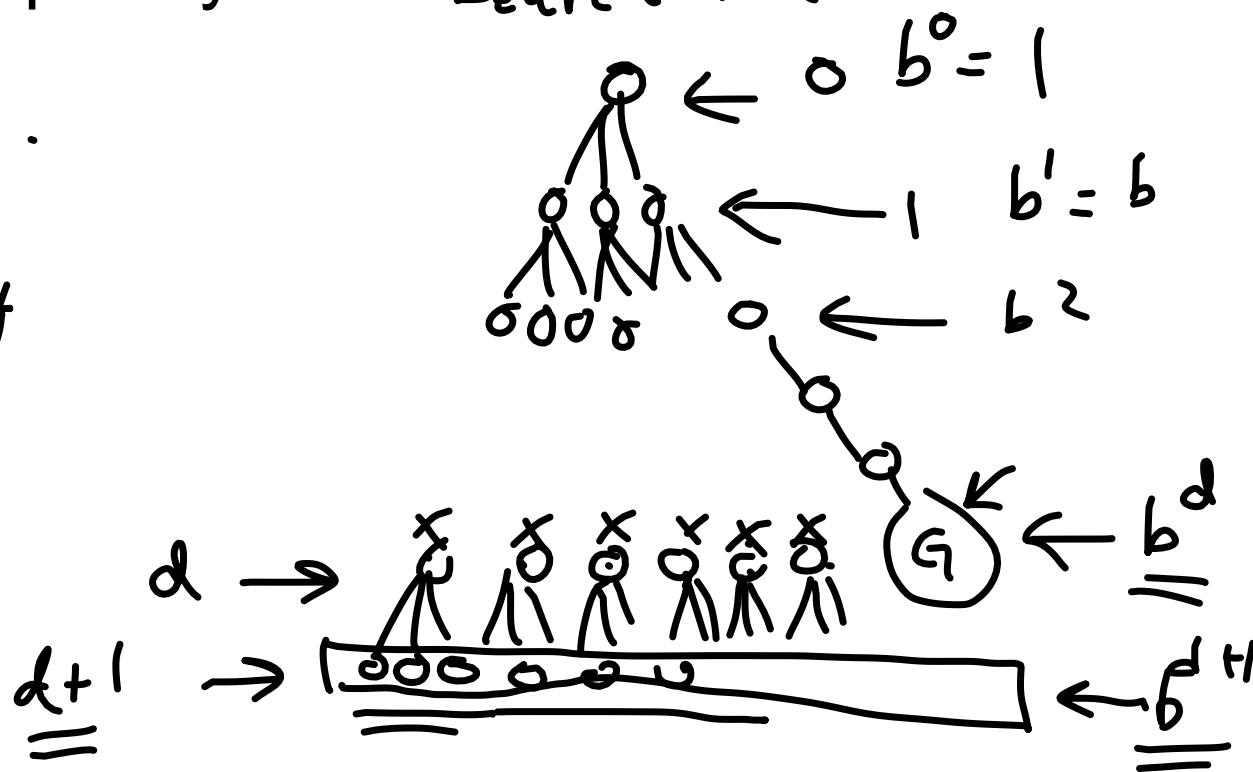
$$\dots + b^d +$$

$$\underline{\underline{b(b^d - 1)}}$$

If a goal node is the last node at level d, we will require the time to generate and store all nodes that are successors of other nodes at level d.

So time complexity is $\underline{\underline{O(b^{d+1})}}$ if using template

Search Tree



What is the Space Complexity?

If a goal node is the last node at level d, all of the successors of the other nodes will be on the Frontier when the goal node is expanded. The number of nodes on the frontier at that time will be:

$$b \cdot (b^d - 1) = O(b^{d+1})$$

Record your answer here: <https://forms.gle/2krMifrptgaEDYyw9>

Breadth-First Properties

Space complexity is a real problem.

- E.g., let $b = 10$, and say 100,000 nodes can be expanded per second and each node requires 100 bytes of storage:

Depth	Nodes	Time	Memory
1	1	0.01 millisec.	100 bytes
6	10^6	10 sec.	100 MB
8	10^8	17 min.	10 GB
9	10^9	3 hrs.	100 GB

- Typically run out of space before we run out of time in most applications.

Depth-First Search

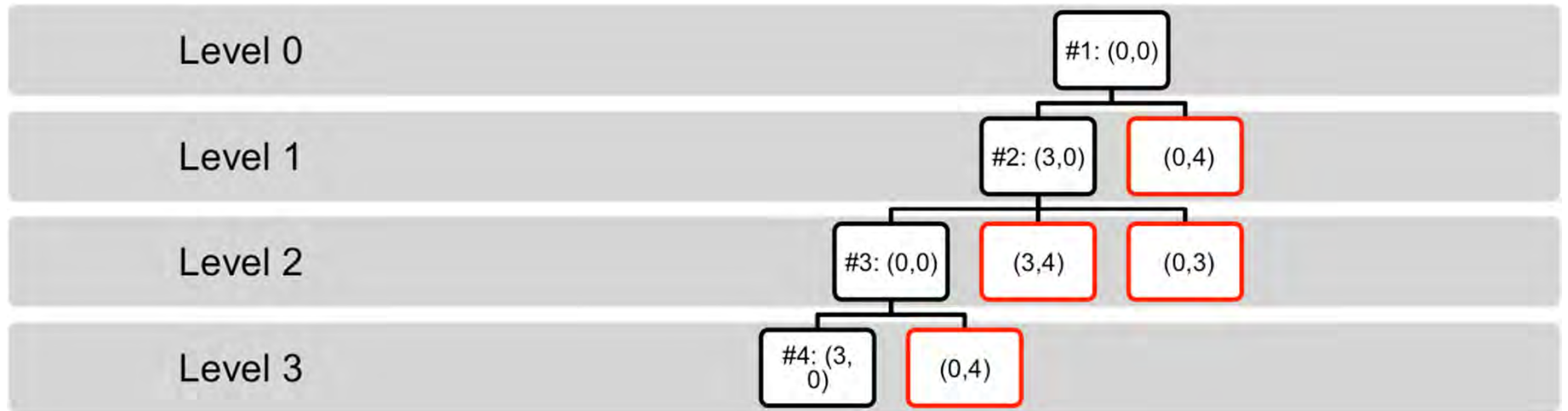
Like BFS, but instead of at the back we place the new paths that extend the current path at the **front** of the Frontier.

Depth-First Search

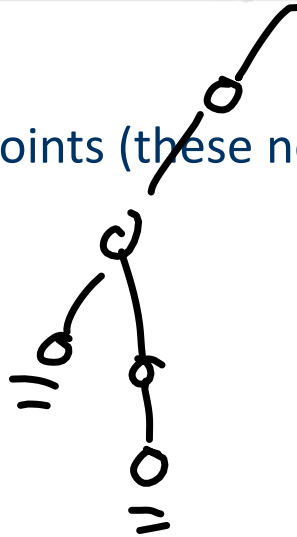
initial state = (0,0), goal state = (*,2), actions (successor functions)
= Empty-3-Gallon, Empty-4-Gallon, Fill-3-Gallon, Fill-4-Gallon, Pour-3-into-4, Pour 4-into-3.

1. Frontier = {<(0,0)>}
2. Frontier = {<(0,0), (3,0)>, <(0,0), (0,4)>}
3. Frontier = {<(0,0), (3,0), (0,0)>, <(0,0), (3,0), (3,4)>, <(0,0), (3,0), (0,3)>, <(0,4), (0,0)>}
4. Frontier = {<(0,0), (3,0), (0,0), (3,0)>, <(0,0), (3,0), (0,0), (0,4)>, <(0,0), (3,0), (3,4)>, <(0,0), (3,0), (0,3)>, <(0,0), (0,4)>}

Depth-First Search

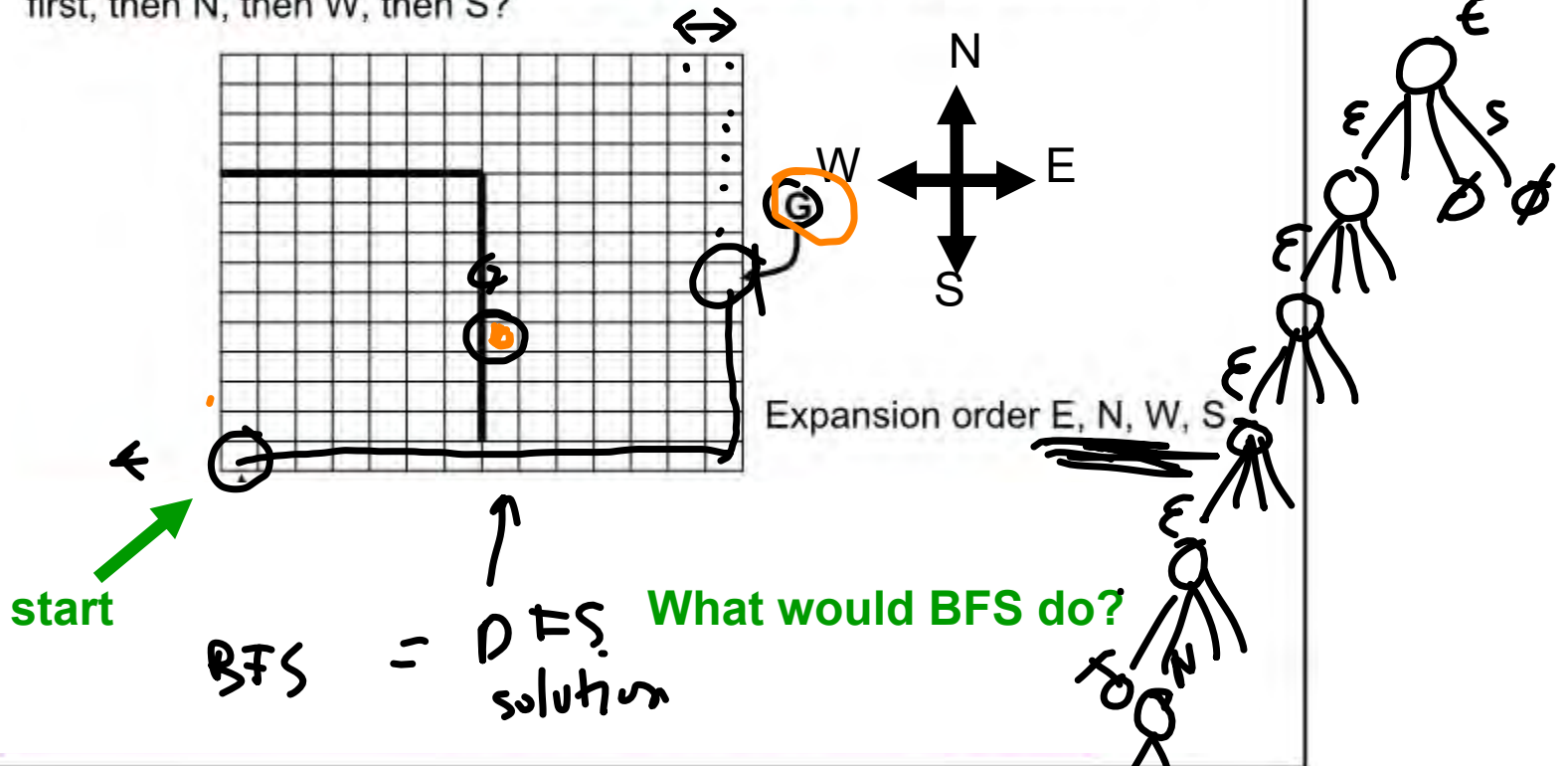


Red nodes are backtrack points (these nodes remain on Frontier).



Maze example

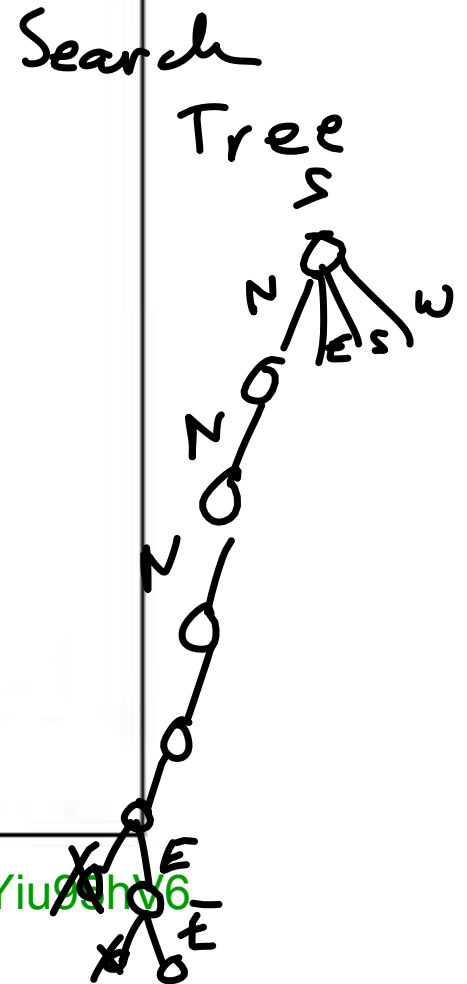
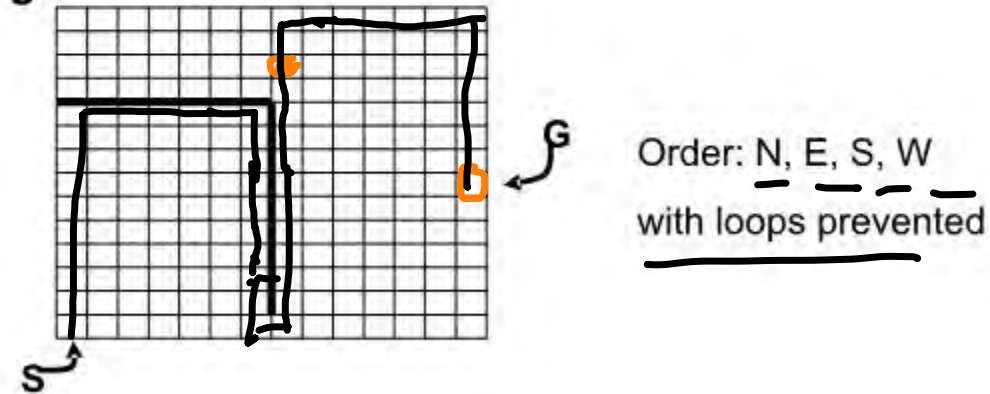
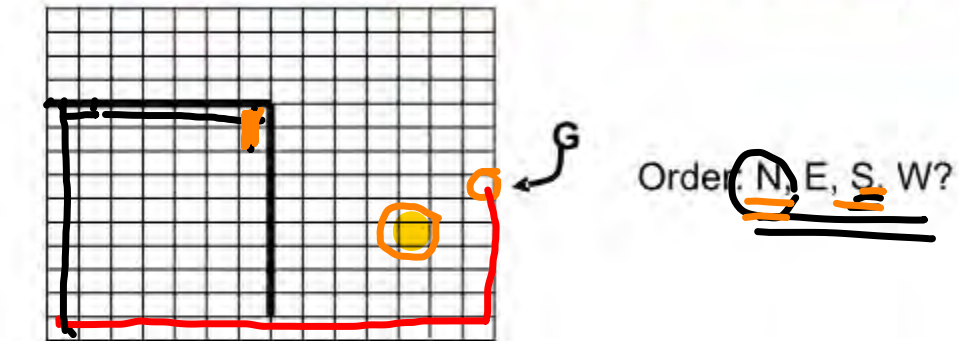
Imagine states are cells in a maze, you can move N, E, S, W. What would **plain DFS** do, assuming it always expanded the E successor first, then N, then W, then S?



Record your answer here: <https://forms.gle/2PFoGVQG9YU95hV6>



Two other DFS examples



Record your answer here: <https://forms.gle/2PFoGVQG9Yiu99hY6>

Depth-First Properties

Complete?

NO, if there are infinite paths
NO, if there are cycles in the graph
– Prune paths with cycles (duplicate states)
YES, if state space is finite.

Optimal? No guarantee!

Record your answer here: <https://forms.gle/2krMifrptgaEDYyw9>

Depth-First Properties

Time Complexity?

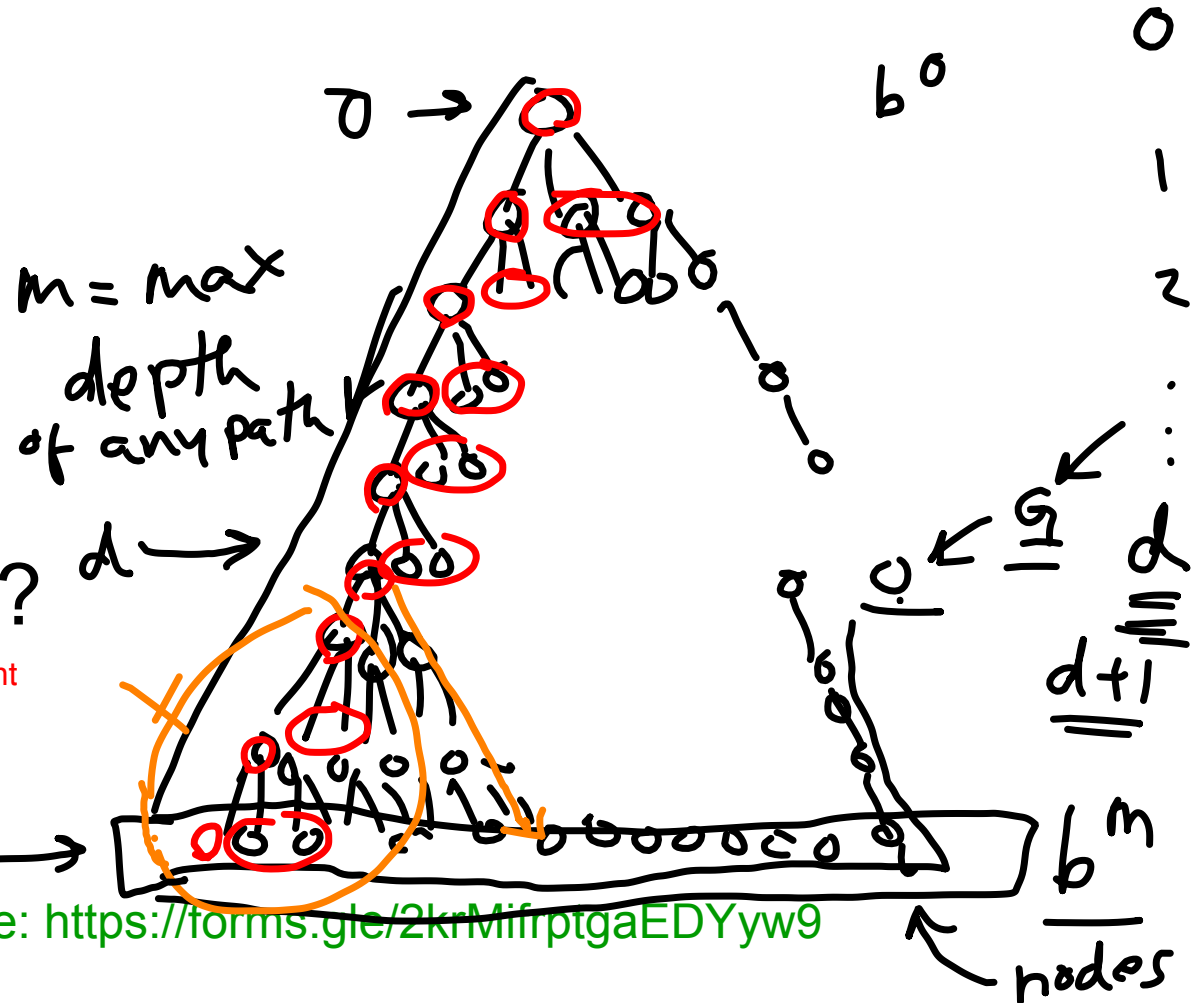
- $O(b^m)$ where m is the length of the longest path in the state space.
- Very bad if m is much larger than d (shortest path to a goal state), but if there are many solution paths it can be much faster than breadth first (by good luck, can bump into a solution quickly).

Space Complexity?

Frontier only contains the deepest node on the current path along with the backtrack points (references to unexplored siblings of states).

$O(b \cdot m)$ linear!

A significant advantage of DFS



Record your answer here: <https://forms.gle/2krMifirptgaEDYyw9>

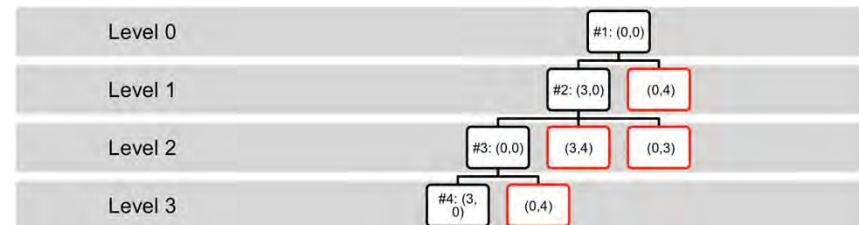
Depth-First Properties

Time Complexity?

- $O(b^m)$ where m is the length of the longest path in the state space.
- Very bad if m is much larger than d (shortest path to a goal state), but if there are many solution paths it can be much faster than breadth first (by good luck, can bump into a solution quickly).

Space Complexity?

- $O(bm)$, linear space!
 - Only explore a single path at a time.
 - Frontier only contains the deepest node on the current path along with the **backtrack** points (references to unexplored siblings of states).
- A significant advantage of DFS

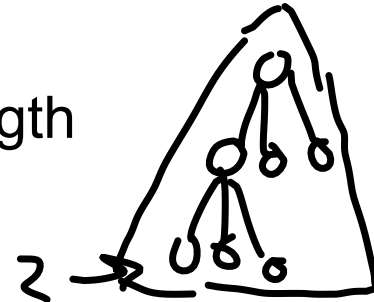


Depth Limited Search

Breadth first has space problems. Depth first can run off down a very long (or infinite) path.

Depth limited search

- Perform depth first search but only to a depth limit d .
 - The ROOT is at DEPTH 0. ROOT is a path of length 1.
- No node representing a path of length more than $d+1$ is placed on the Frontier.
- “Truncate” the search by looking *only* at paths of length $d+1$ or less.



- Now infinite length paths are not a problem.
- But will only find a solution if a solution of DEPTH $\leq d$ exists.

Depth Limited Search

```
DLS (Frontier, Successors, Goal?) /* Call with Frontier = {<START>} */
```

```
  WHILE (Frontier not EMPTY) {  
    n= select first node from Frontier  
    Curr = terminal state of n  
    If(Goal?(Curr)) return n
```

```
    If Depth(n) < D //Don't add successors if Depth(n) = D!!  
      Frontier= (Frontier- {n}) U Successors(Curr)
```

```
  Else
```

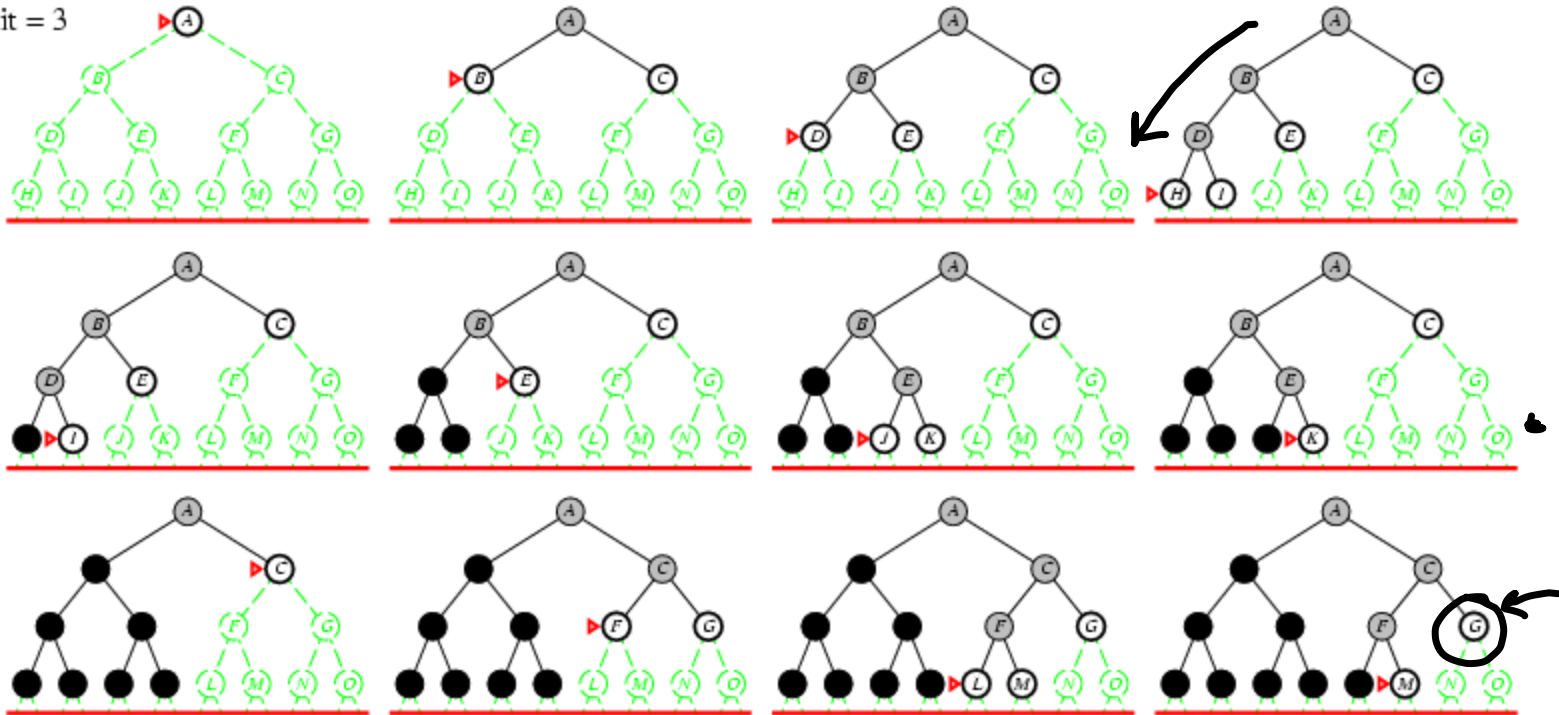
```
    Frontier= Frontier- {n}  
    CutOffOccured = TRUE.
```

```
  }
```

```
  return FAIL
```

Depth Limited Search Example

Limit = 3



Iterative Deepening Search

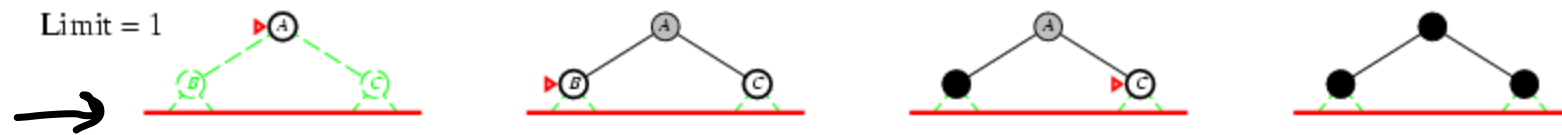
- Solve the problems of depth-first and breadth-first by extending depth limited search.
- Starting at depth limit $L = 0$, we iteratively increase the depth limit, performing a depth limited search for each depth limit.
- Stop if a solution is found, or if the depth limited search failed without cutting off any nodes because of the depth limit.
 - If no nodes were cut off, the search examined all paths in the state space and found no solution \rightarrow no solution exists.

Iterative Deepening Search

Limit = 0

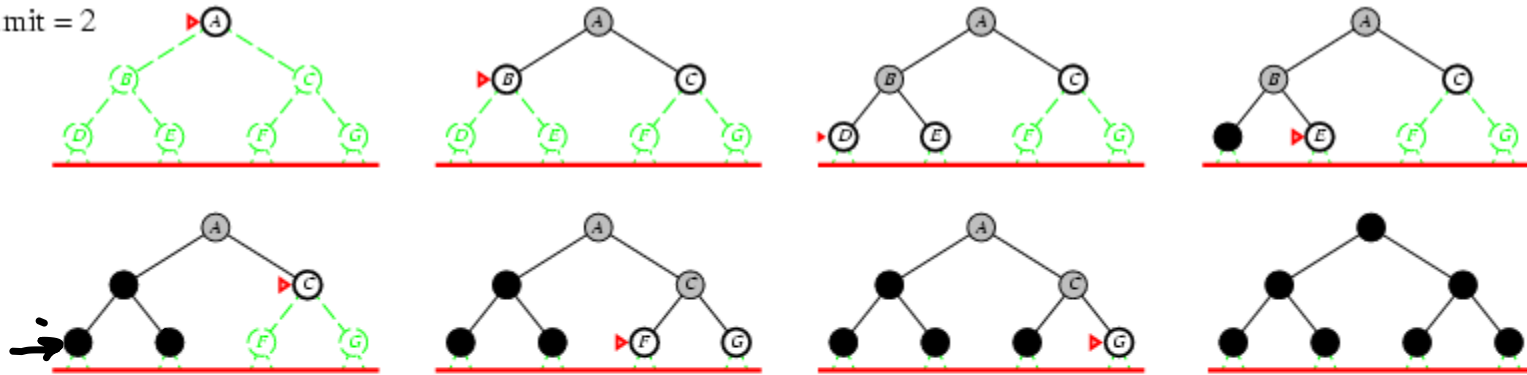


Iterative Deepening Search



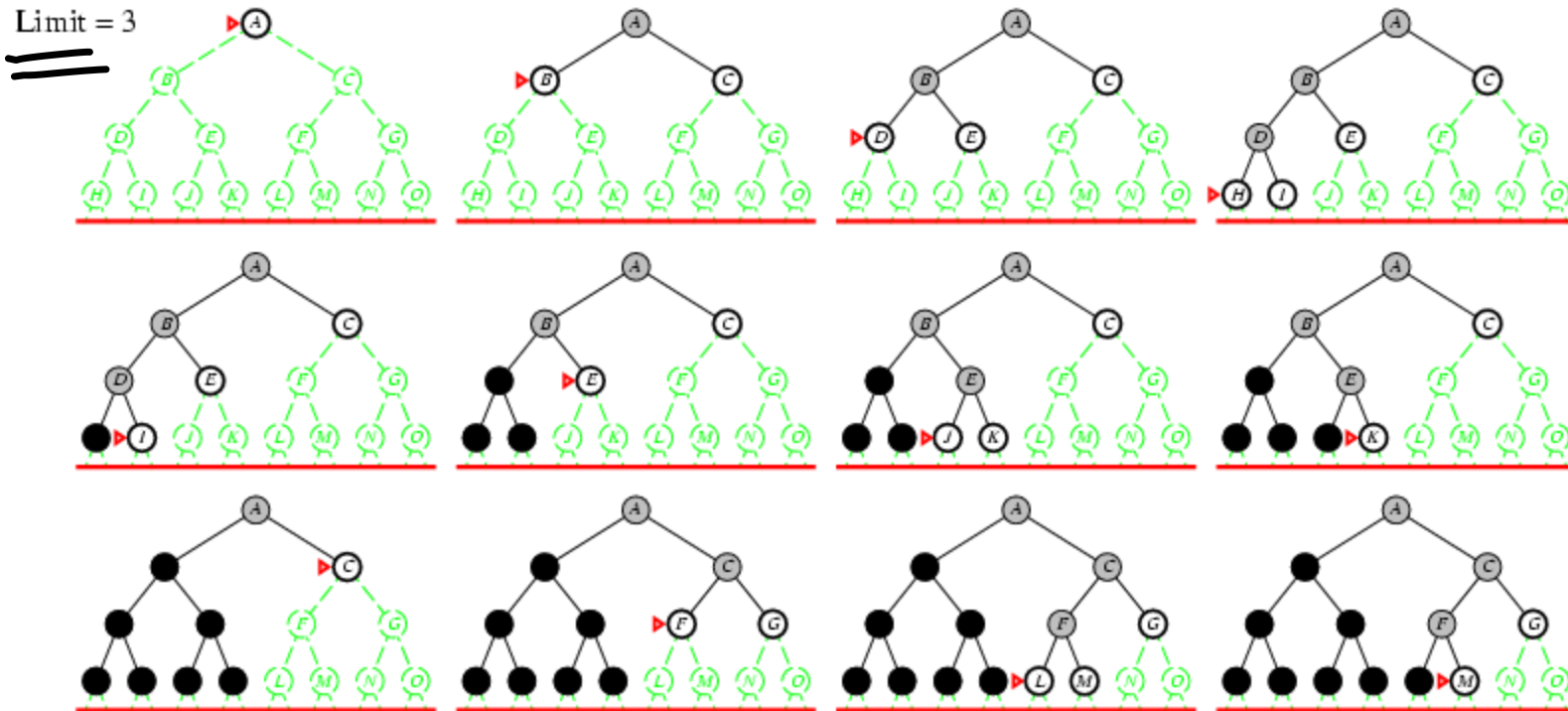
Iterative Deepening Search

Limit = 2

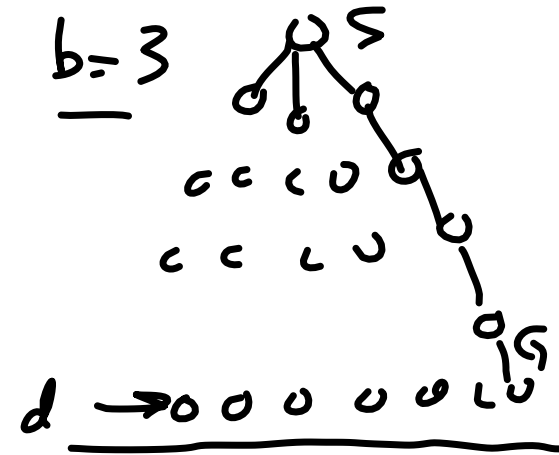


Iterative Deepening Search

Limit = 3



Iterative Deepening Search



Completeness?

YES, if a minimal depth solution of depth d exists

Optimality?

Yes, assuming you increment depth conservatively

Space Complexity?

$O(d \cdot b)$ still linear!

Record your answer here: <https://forms.gle/2krMifrptgaEDYyw9>

Iterative Deepening Search

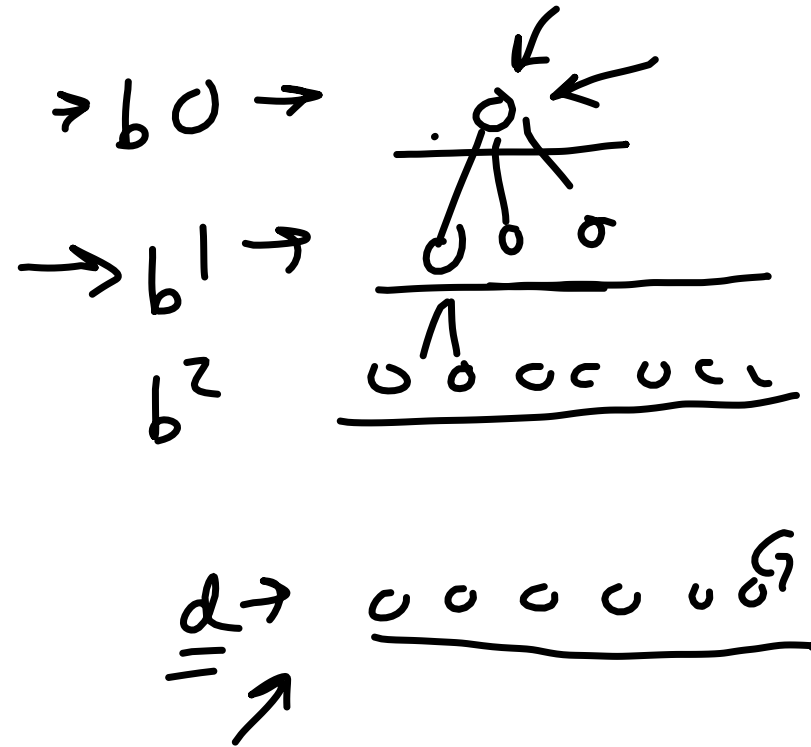
Time Complexity?

$$\underline{\underline{(d+1)b^0}} + db^1 + (d-1)b^2 + \dots + \dots + 1 \times b^d$$

We will see the nodes in layer 0 a total of d+1 times. Nodes in layer 1 will be seen d time, and so on. But the final layer, will will contain all paths of length d, will be seen only once.

result is $\underline{\underline{O(b^d)}}$

$\underline{\underline{BFS}} \quad \underline{\underline{O(b^{d+1})}}$

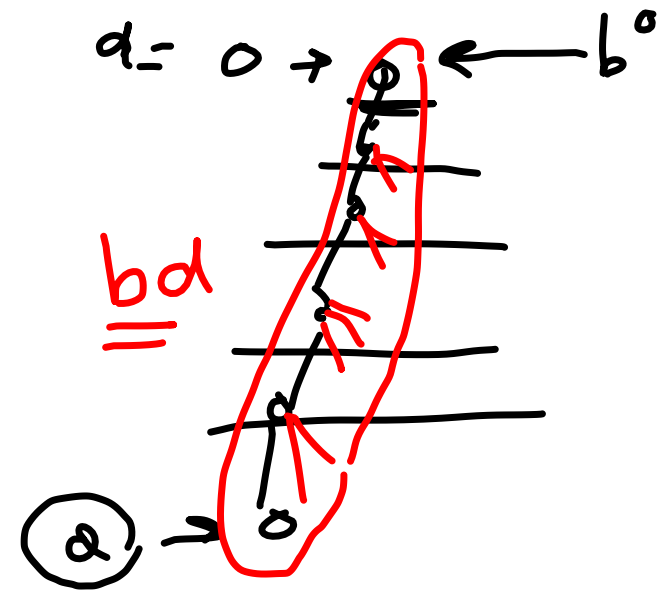


BFS can explore more states than IDS!

- For IDS, the time complexity is
 - $(d+1)b^0 + db^1 + (d-1)b^2 + \dots + b^d = O(b^d)$
- For BFS, the time complexity is
 - $1 + b + b^2 + b^3 + \dots + b^d + b(b^d - 1) = O(b^{d+1})$

E.g. $b=4, d=10$

- For IDS
 - $(11) \cdot 4^0 + 10 \cdot 4^1 + 9 \cdot 4^2 + \dots + \underline{4^{10}} = \underline{1,864,131}$ (states generated)
- For BFS
 - $1 + 4 + 4^2 + \dots + 4^{10} + \underline{4(4^{10} - 1)} = \underline{5,592,401}$ (states generated)
 - In fact IDS can be more efficient than breadth first search: nodes at limit are not expanded. BFS must expand all nodes until it expands a goal node. So at the bottom layer it will add many nodes to Frontier before finding the goal node.



Optimality of Iterative Deepening Search

- IDS finds shortest length (and least cost) solution, if costs are uniform.
- If costs are *not* uniform, we can use a “cost” bound instead.
 - Only expand paths of cost less than the cost bound.
 - Keep track of the minimum cost unexpanded path in each depth first iteration, increase the cost bound to this on the next iteration.
 - This can be more expensive. We will need to run as many iterations of the search as there are distinct path costs.

Path Checking

Recall that paths are commonly stored on the Frontier.

If n_k represents the path $\langle s_0, s_1, \dots, s_k \rangle$ and we expand s_k to obtain child c , we have

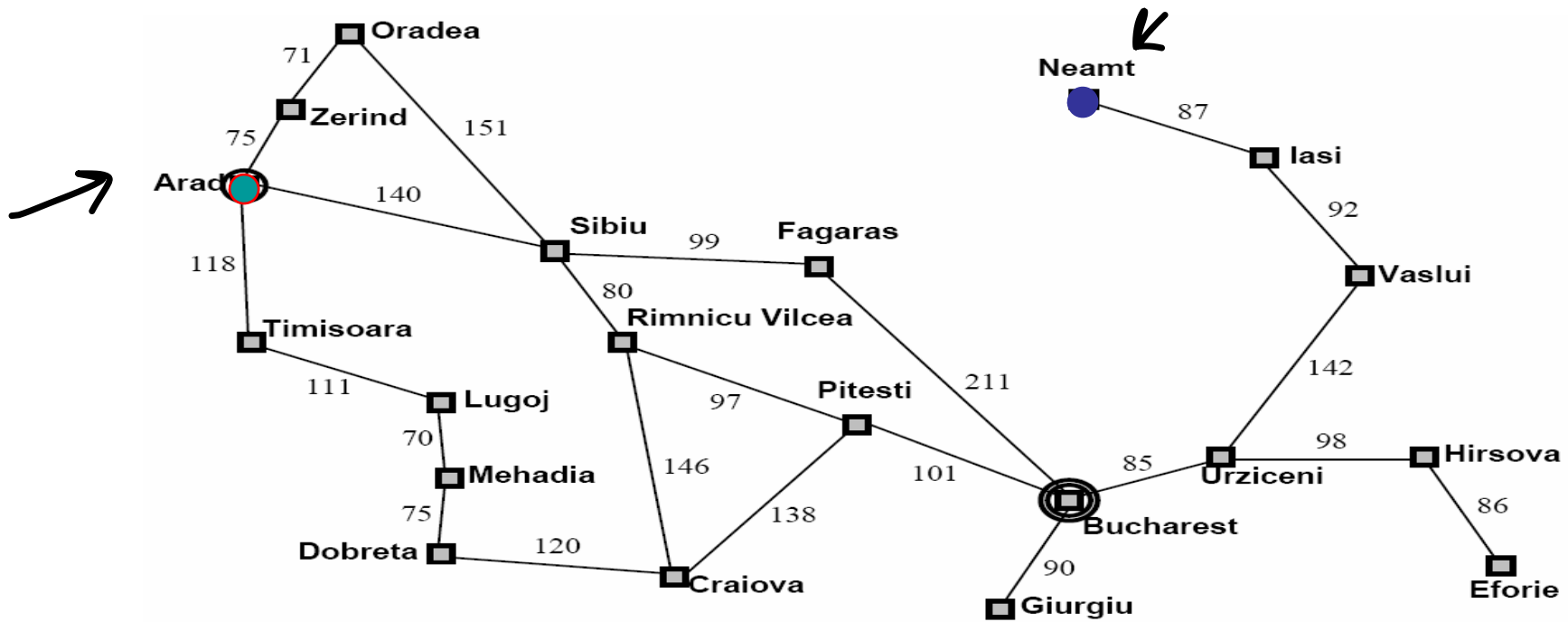
$$\langle s_0, s_1, \dots, s_k, c \rangle$$

as the path to “ c ”.

Path checking:

- Ensure that the state c is not equal to the state reached by any ancestor of c along this path.
- Paths are checked in isolation!

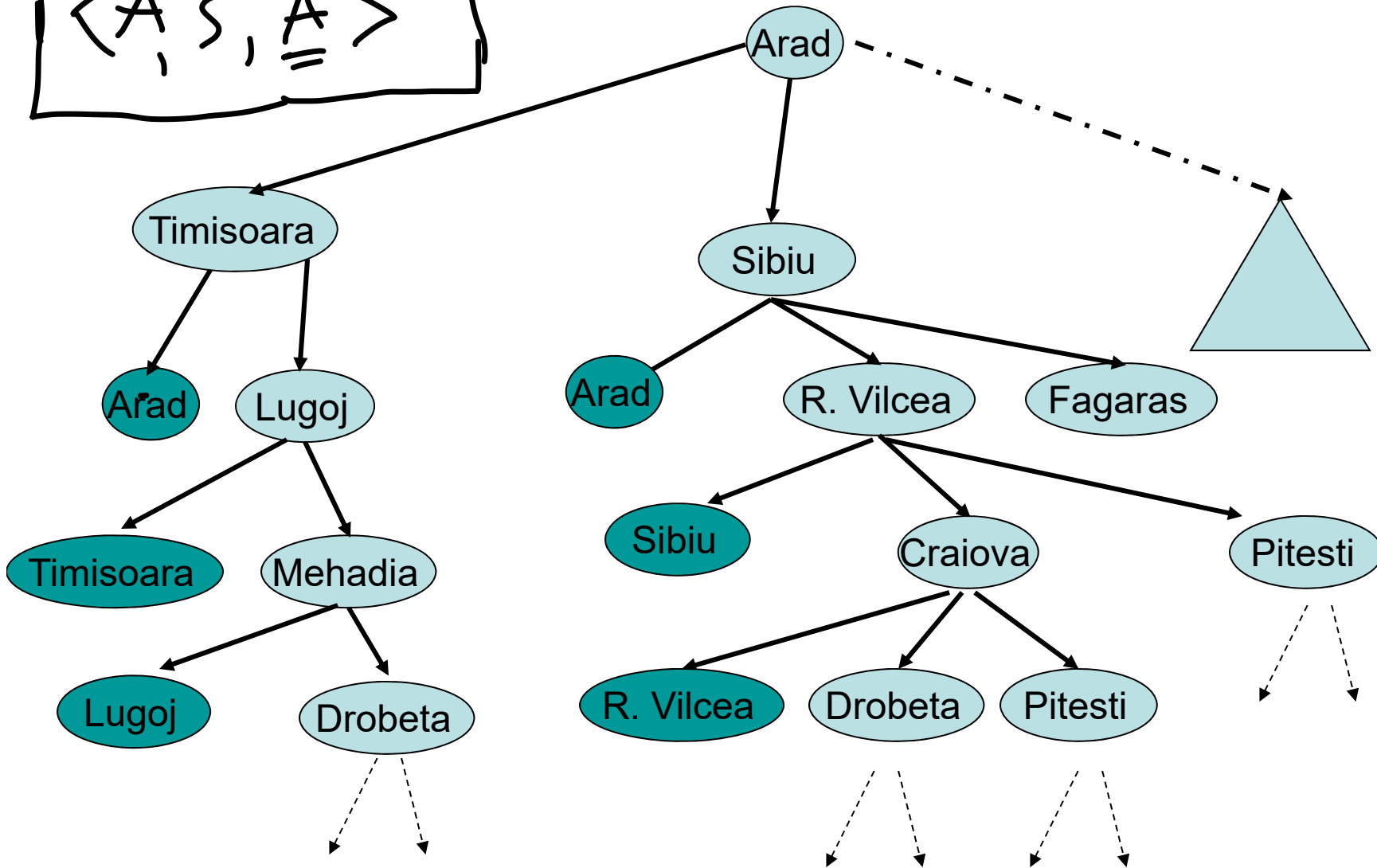
Example: Arad to Neamt



Path:

Path Checking Example

$\langle A, S, \underline{A} \rangle$



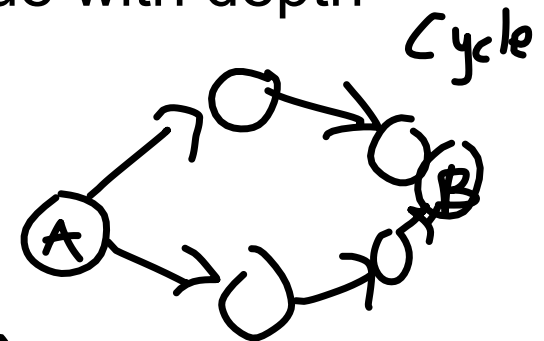
Cycle Checking

typically use
hash table

- Keep track of **all states** previously expanded during the search.
- When we expand n_k to obtain child c
 - Ensure that c is not equal to **any** previously expanded state.
- This is called **cycle checking**, or **multiple path checking**.
- What happens when we utilize this technique with depth-first search?
 - **What happens to space complexity?**

becomes exponential

function
of $b + d$.

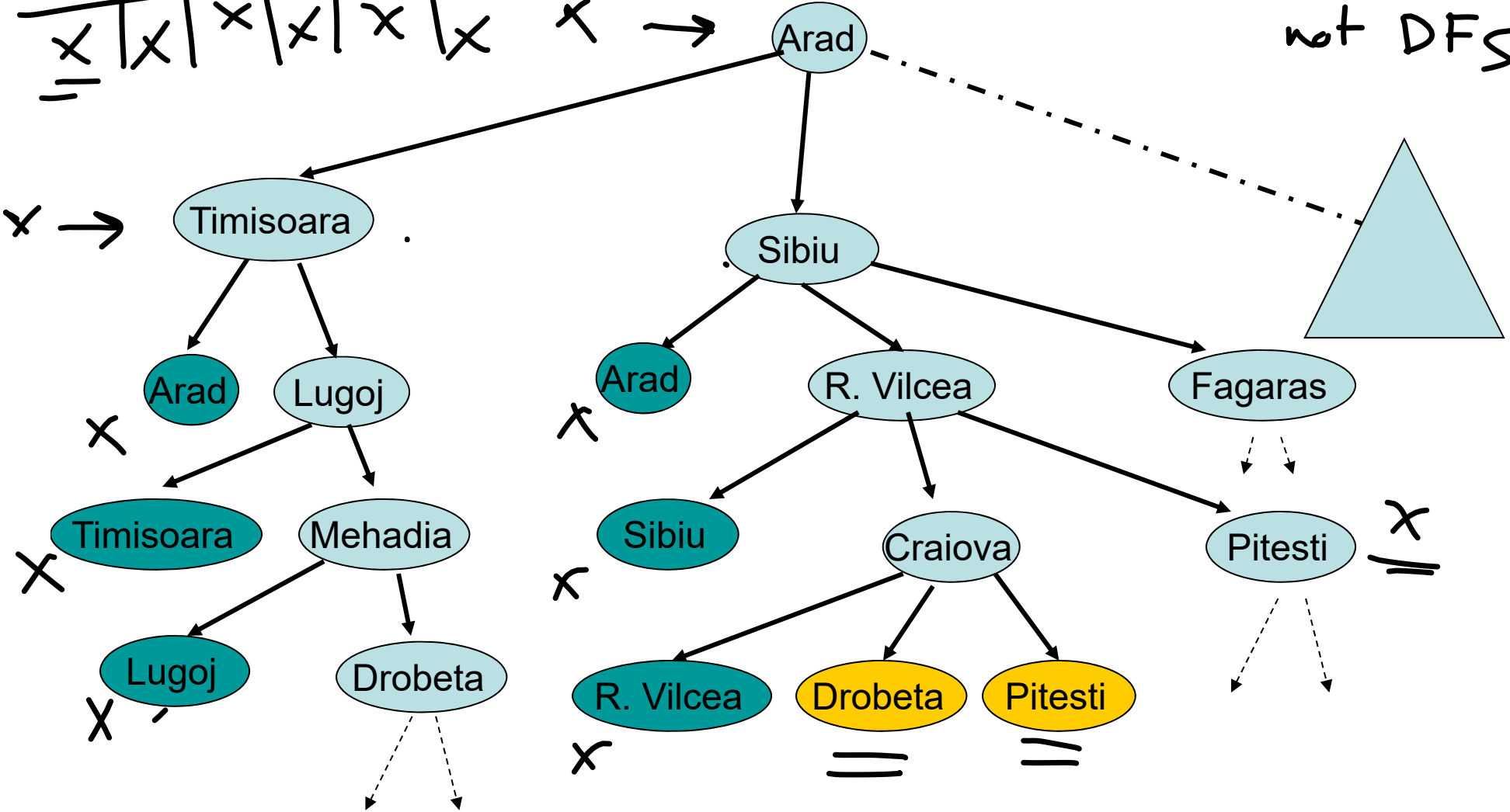


Cycle Checking Example (BFS)

$$O(b^{d+1})$$

not DFS

A	T	S	L	R	F
X	X	X	X	X	X



Cycle Checking

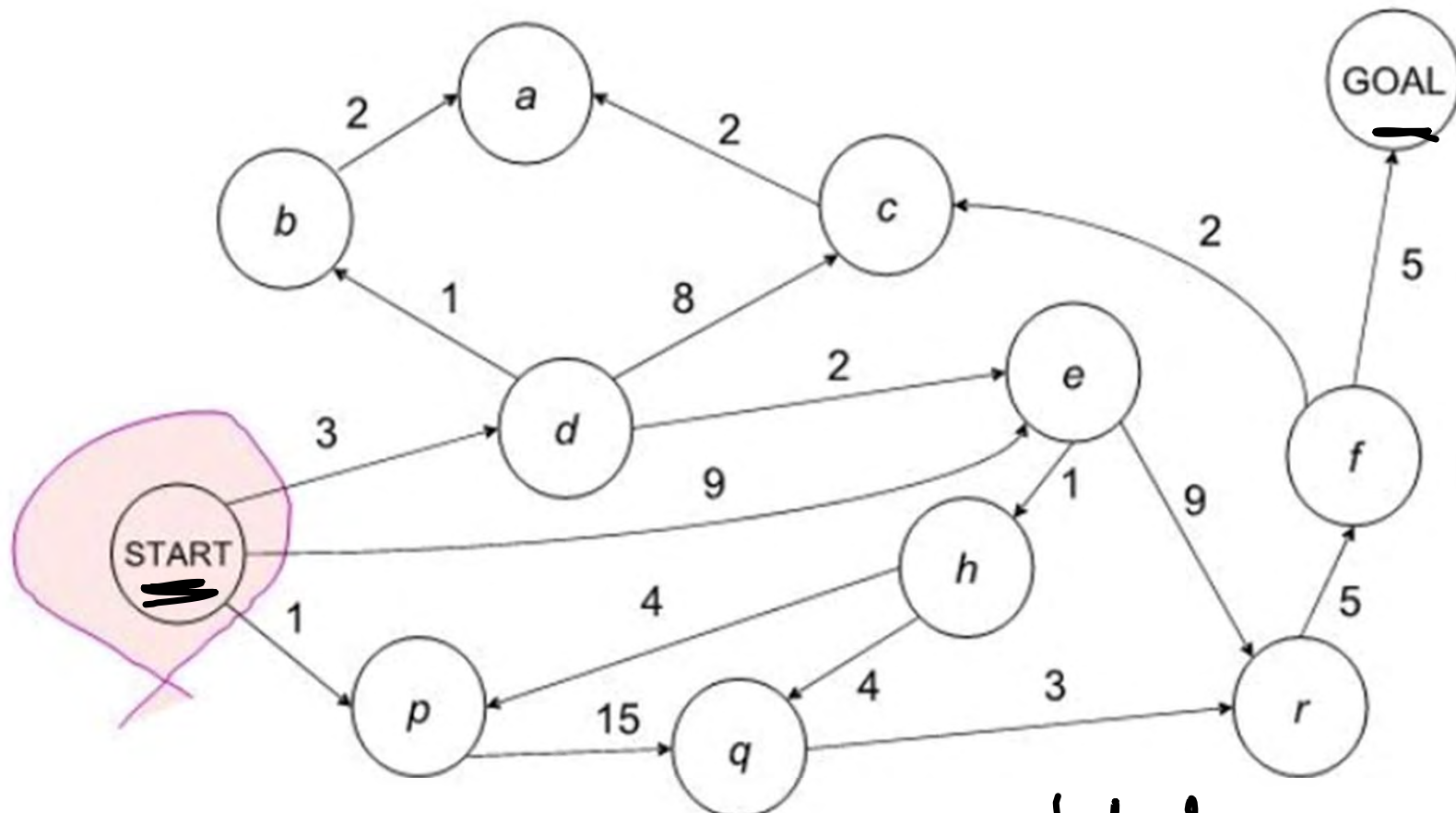
- Higher space complexity (equal to the space complexity of breadth-first search).
- Other issues with cycle checking will come up when we look at heuristic search.

Uniform-Cost Search

- Keeps **Frontier** ordered by **increasing cost of the path** (*know a good data structure for this?*)
- Always expand the **least cost path**.
- Identical to Breadth First Search if each action has the same cost

$$c(x, y) \geq 0$$

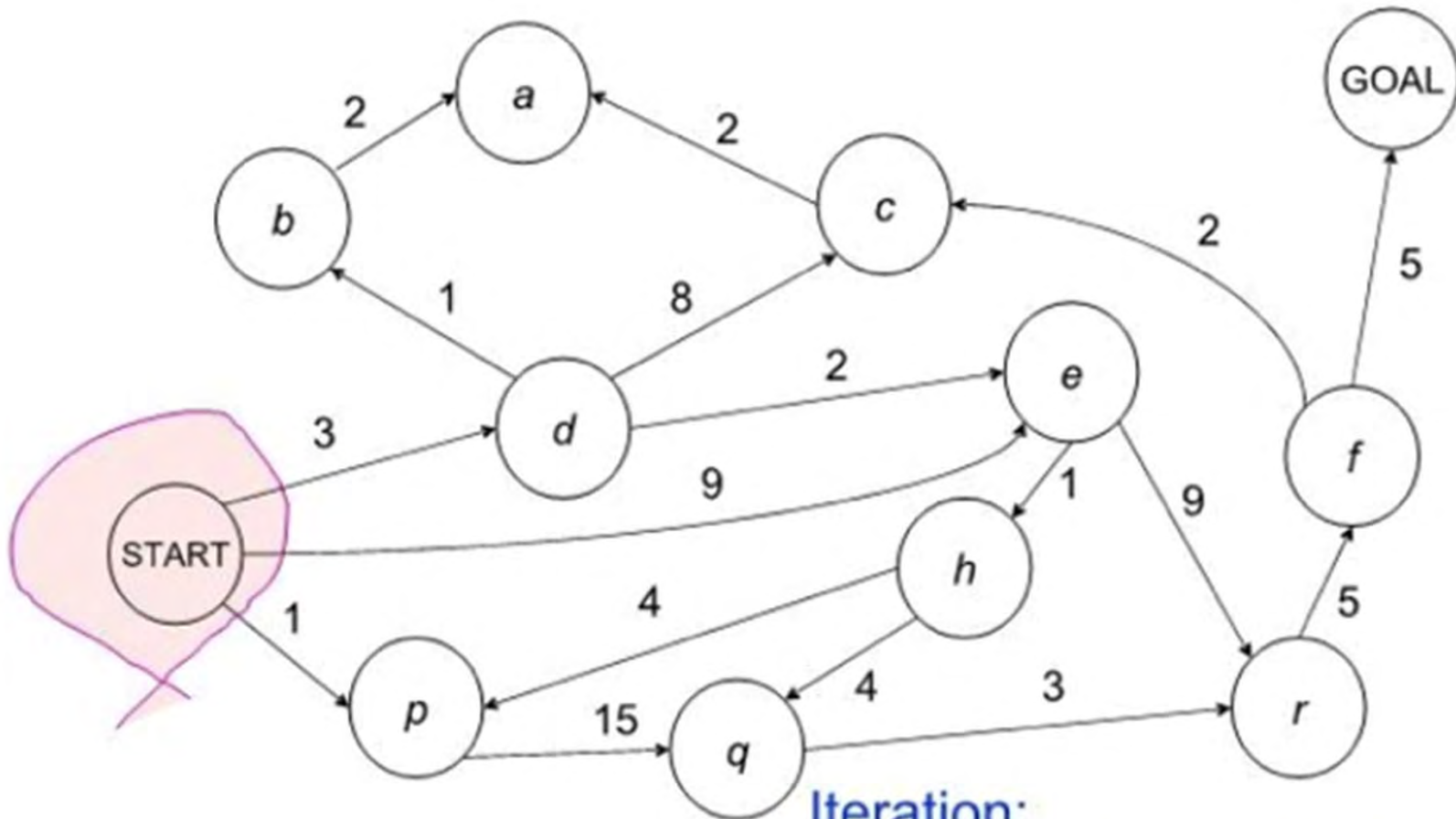
Starting UCS



Frontier = {(START,0)}

Cost accumulated

UCS Iterations

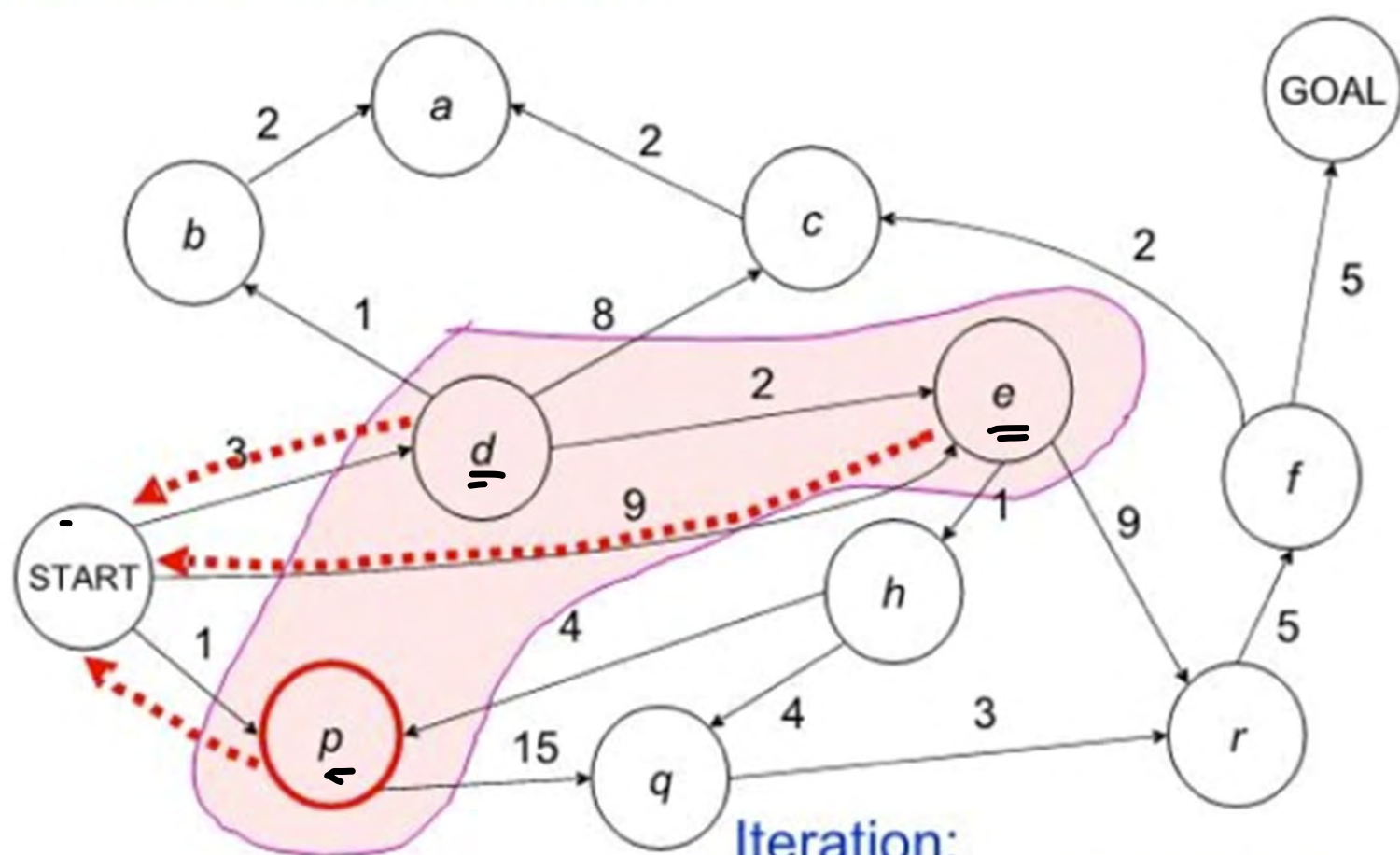


Frontier = {(START,0)}

Iteration:

1. Pop least-cost state ✖
2. Add successors

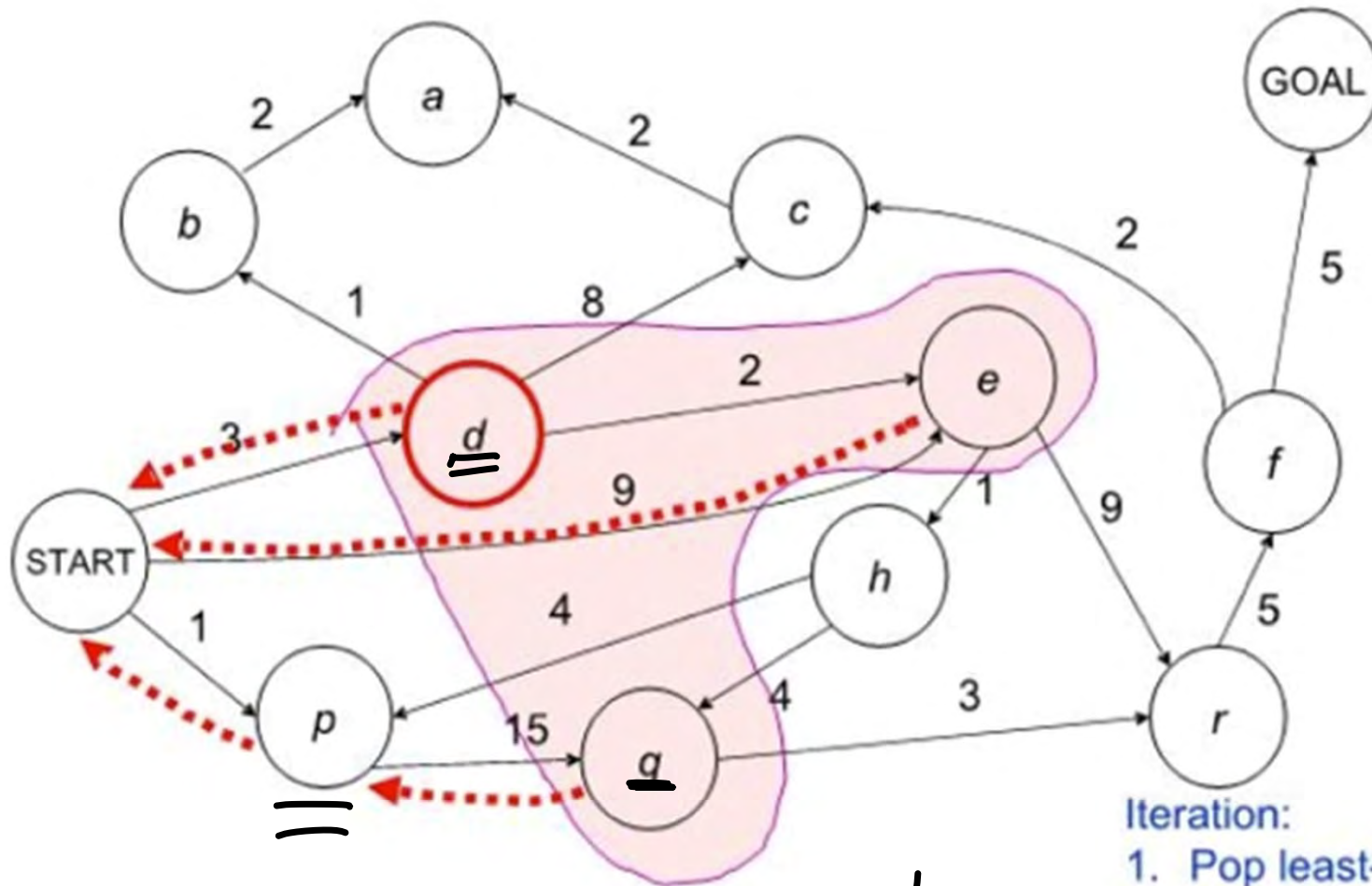
UCS Iterations



Frontier = $\{(p,1), (d,3), (e,9)\}$

- Iteration:
1. Pop least-cost state
 2. Add successors

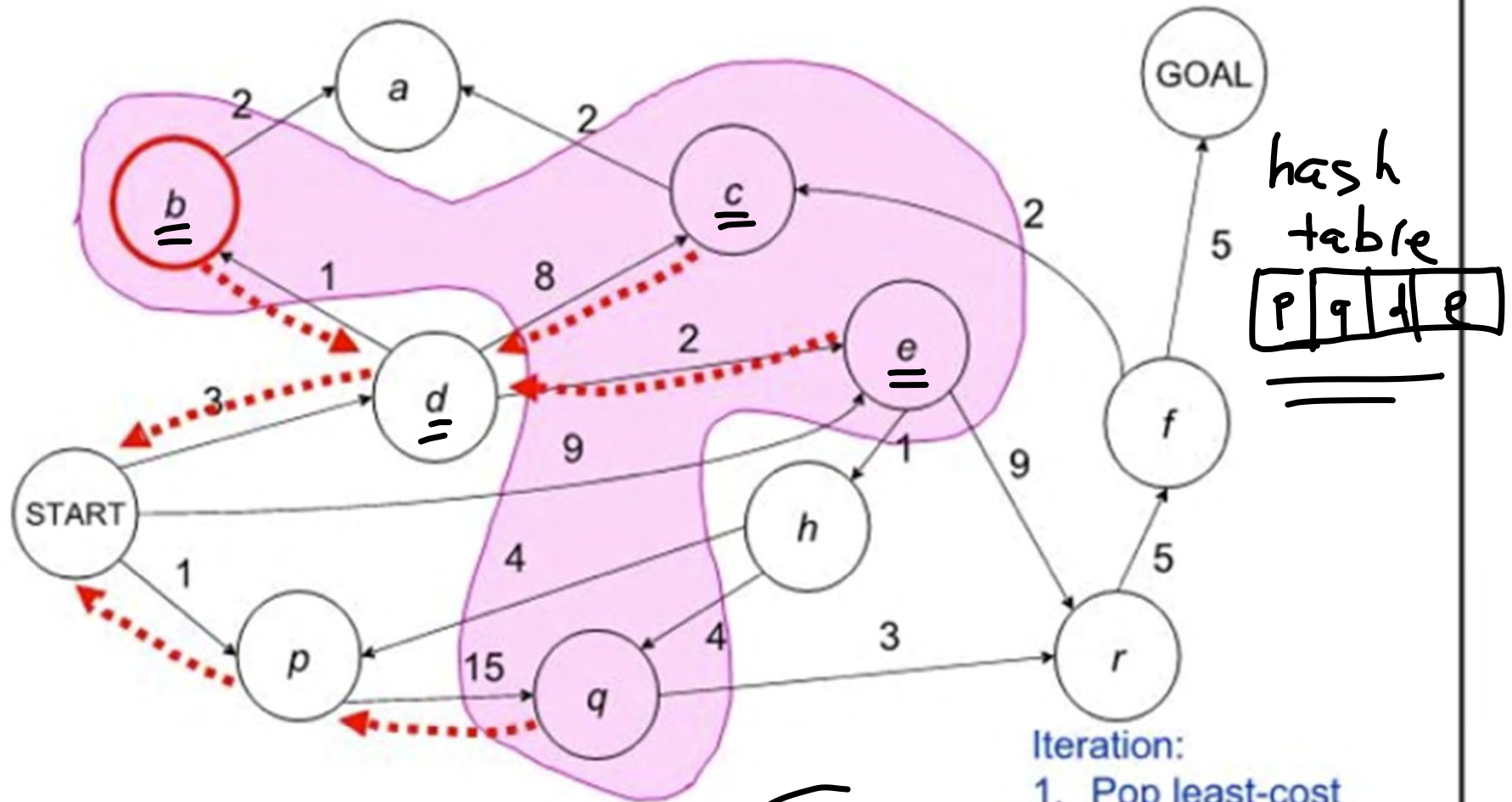
UCS Iterations



Frontier = $\{(\underline{d}, 3), (\underline{e}, 9), (\underline{q}, 16)\}$

cost
1+15

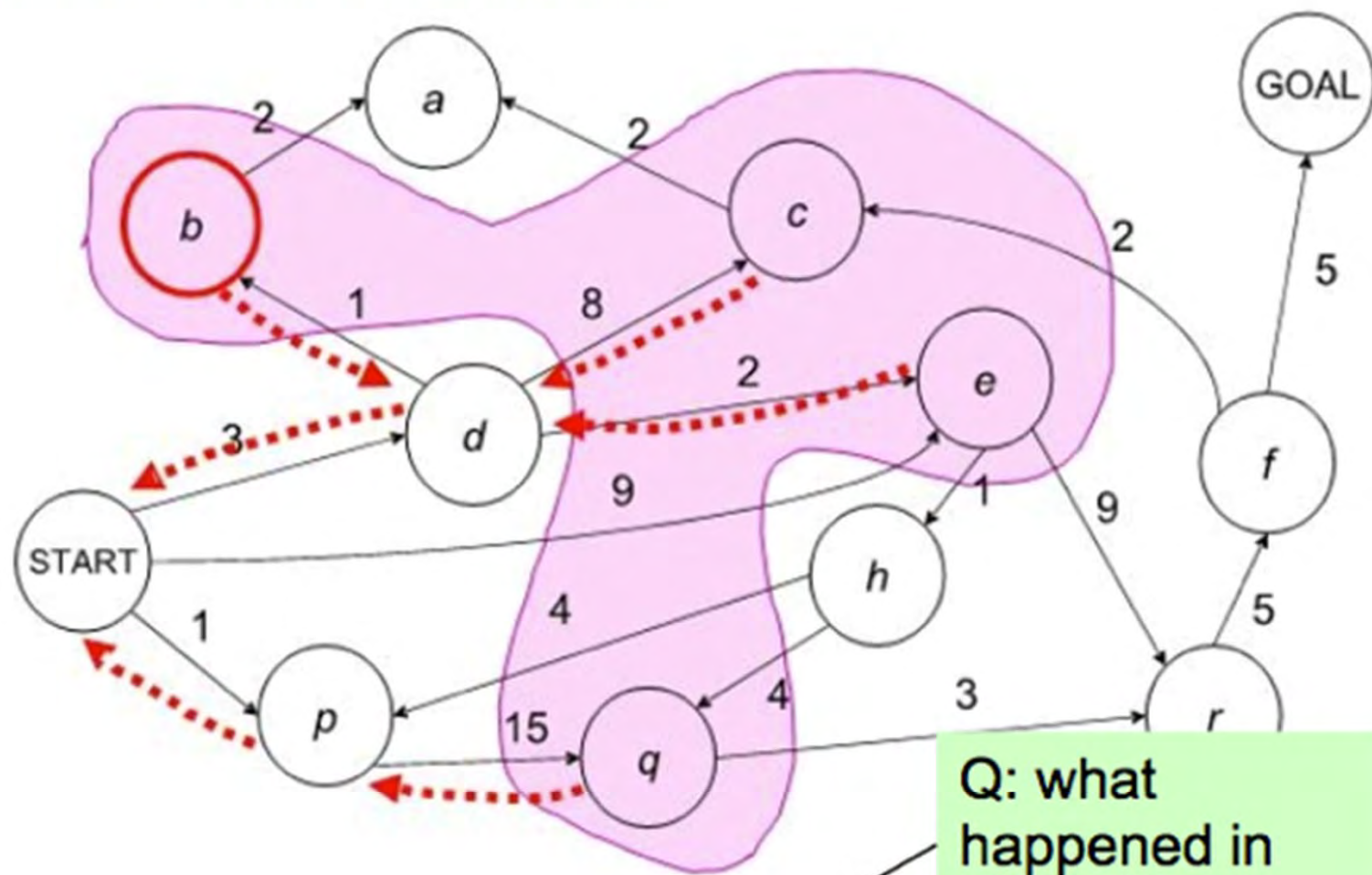
UCS Iterations



Frontier = $\{(b,4), (e,5), (c,11), (q,16)\}$

- Iteration:
1. Pop least-cost state
 2. Add successors

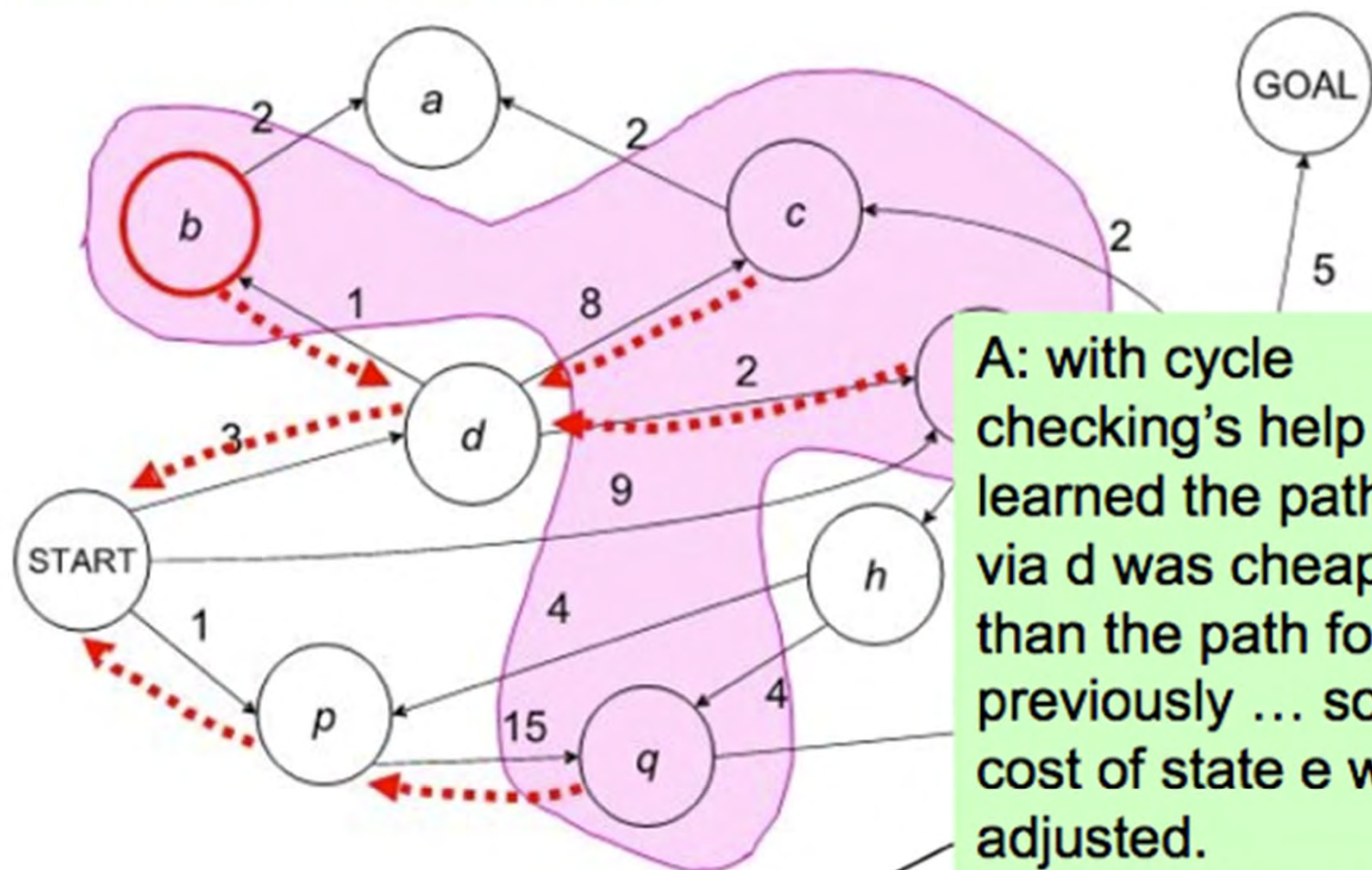
UCS Iterations



Frontier = $\{(b,4), (e,5), (c,11), (q,16)\}$

2. Add successors

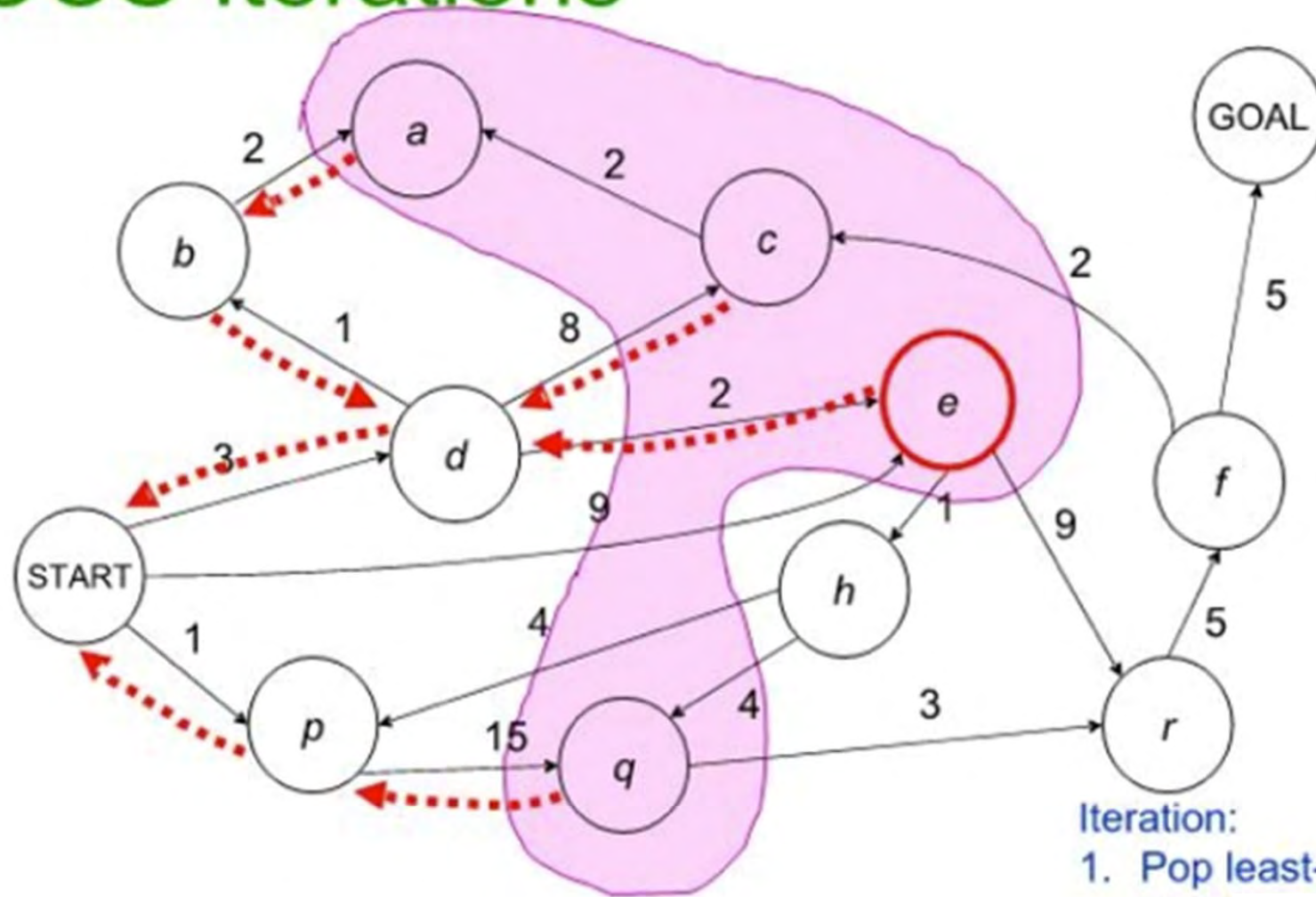
UCS Iterations



Frontier = $\{(b,4), (e,5), (c,11), (q,16)\}$

state
2. Add successors

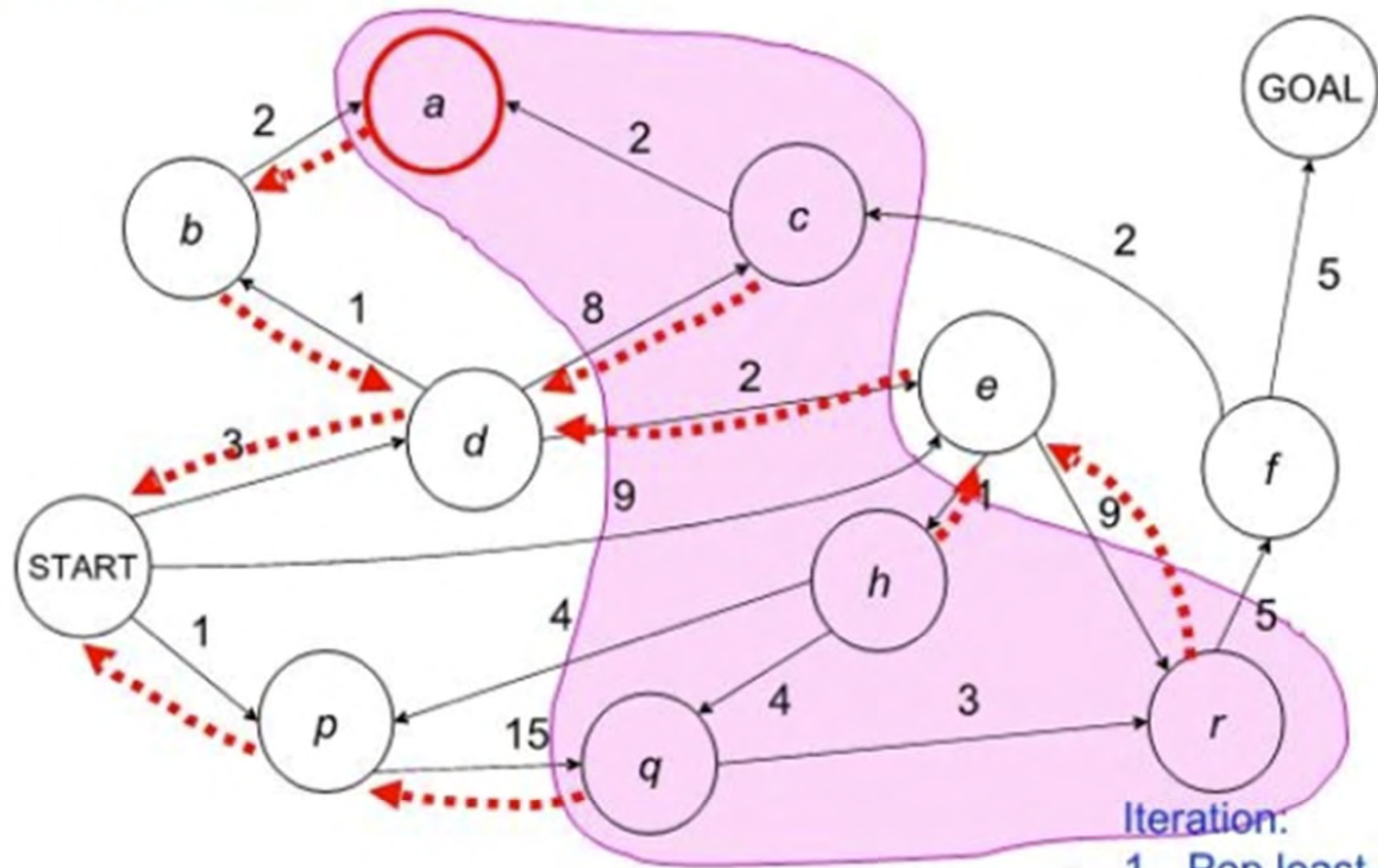
UCS Iterations



- Iteration:
1. Pop least-cost state
 2. Add successors

Frontier = $\{(e,5), (a,6), (c,11), (q,16)\}$

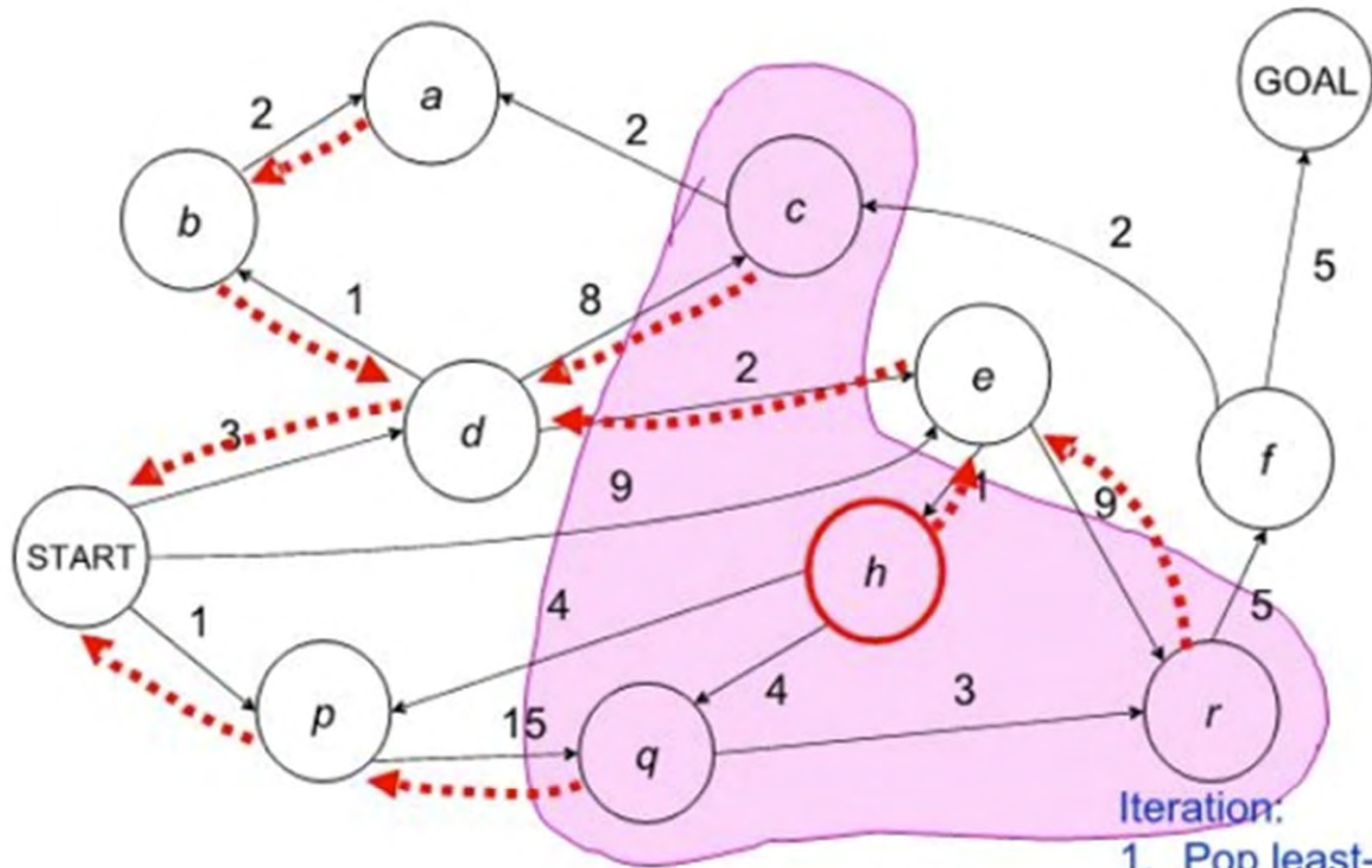
UCS Iterations



Frontier = $\{(a,6), (h,6), (c,11), (r,14), (q,16)\}$

- Iteration:
1. Pop least-cost state
 2. Add successors

UCS Iterations



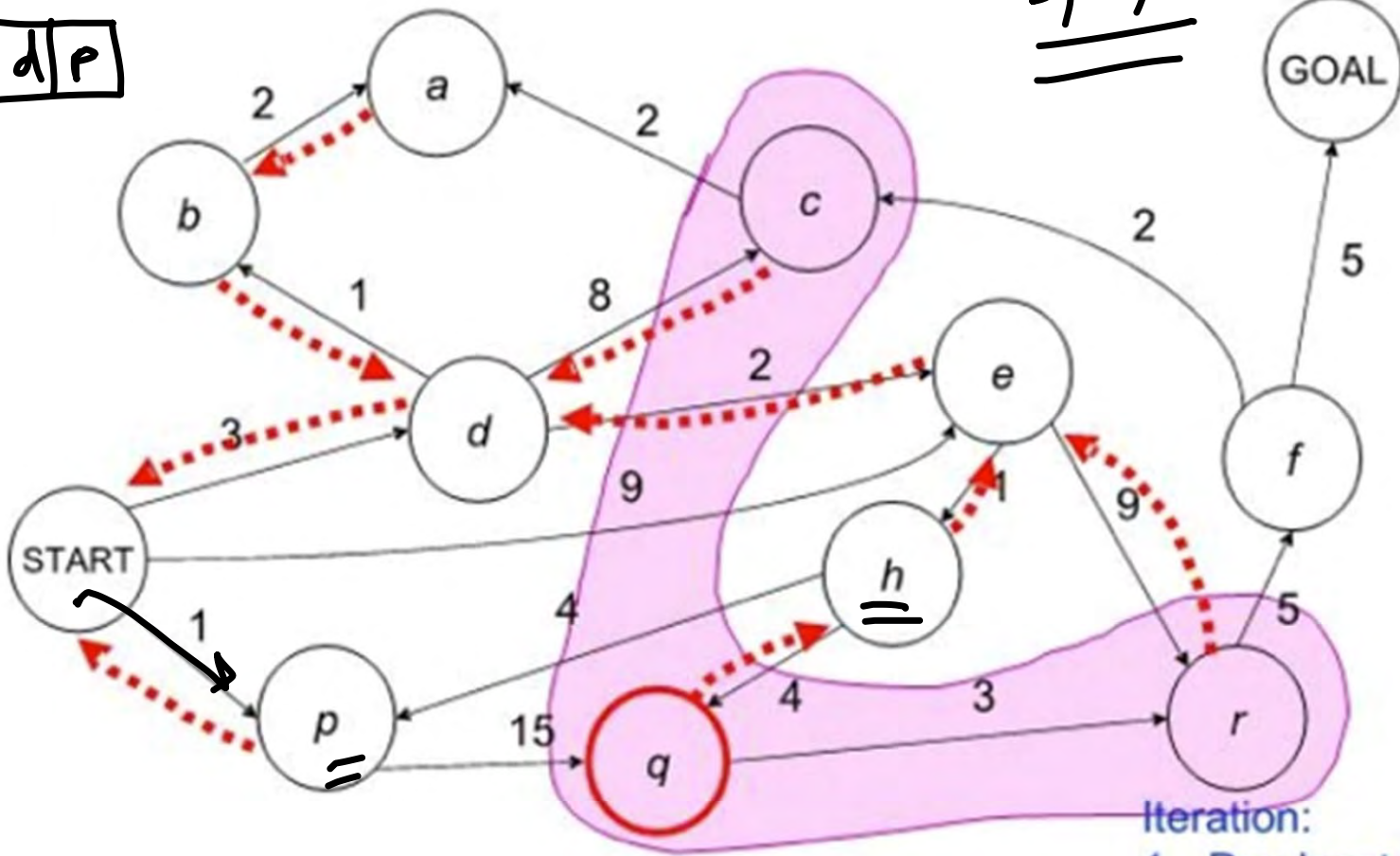
Frontier = $\{(h,6), (c,11), (r,14), (q,16)\}$

- Iteration:
1. Pop least-cost state
 2. Add successors

UCS Iterations

s | a | b | d | p

s, e, h, p



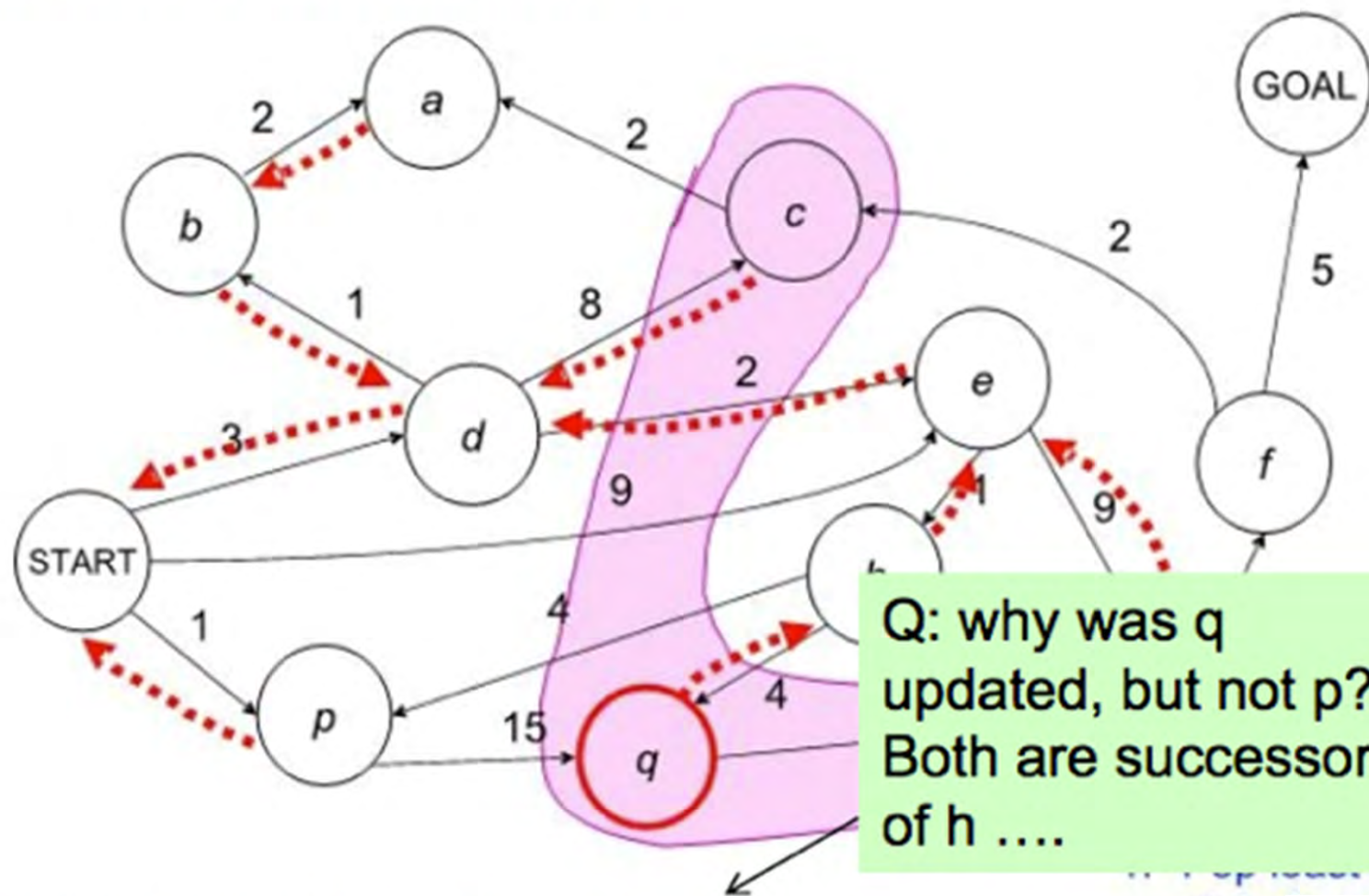
(h, 6)
(p, 10)

Frontier = {(q, 10), (c, 11), (r, 14)}

? why isn't p here

- Iteration:
1. Pop least-cost state
 2. Add successors

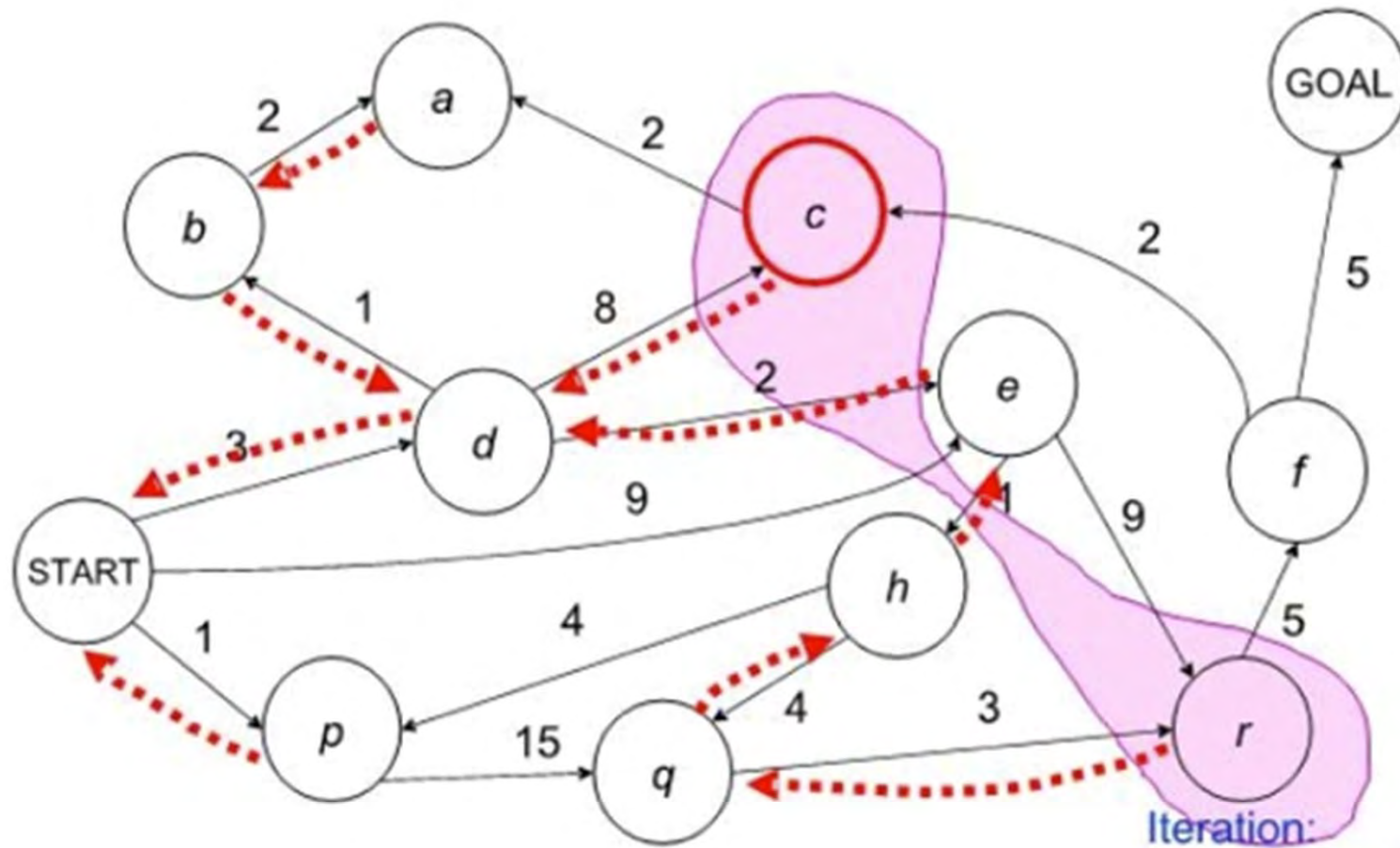
UCS Iterations



Frontier = $\{(q,10), (c,11), (r,14)\}$

1. Remove lowest cost state
2. Add successors

UCS Iterations

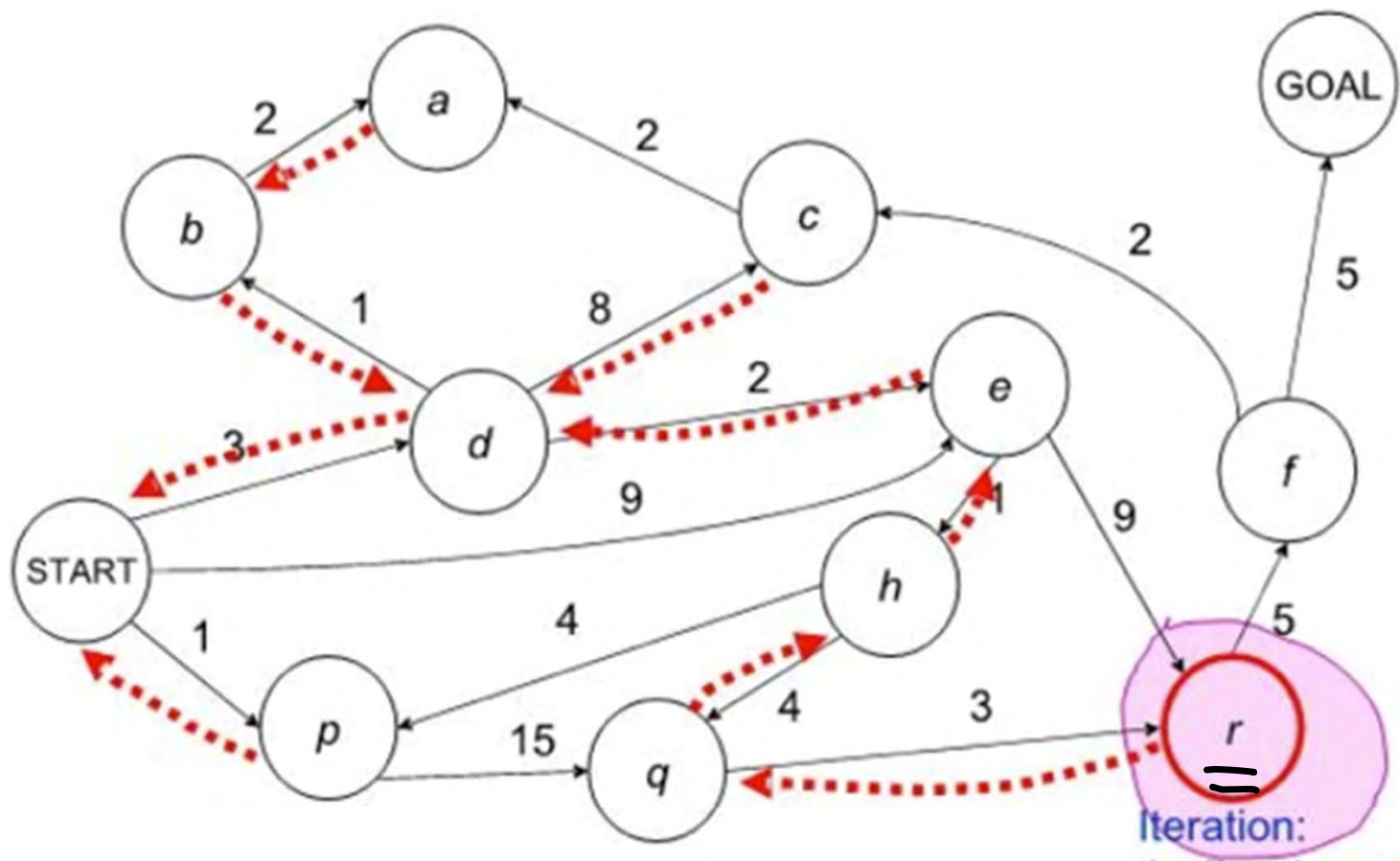


- Iteration:
1. Pop least-cost state
 2. Add successors

Frontier = $\{(c, 11), (r, 13)\}$

||

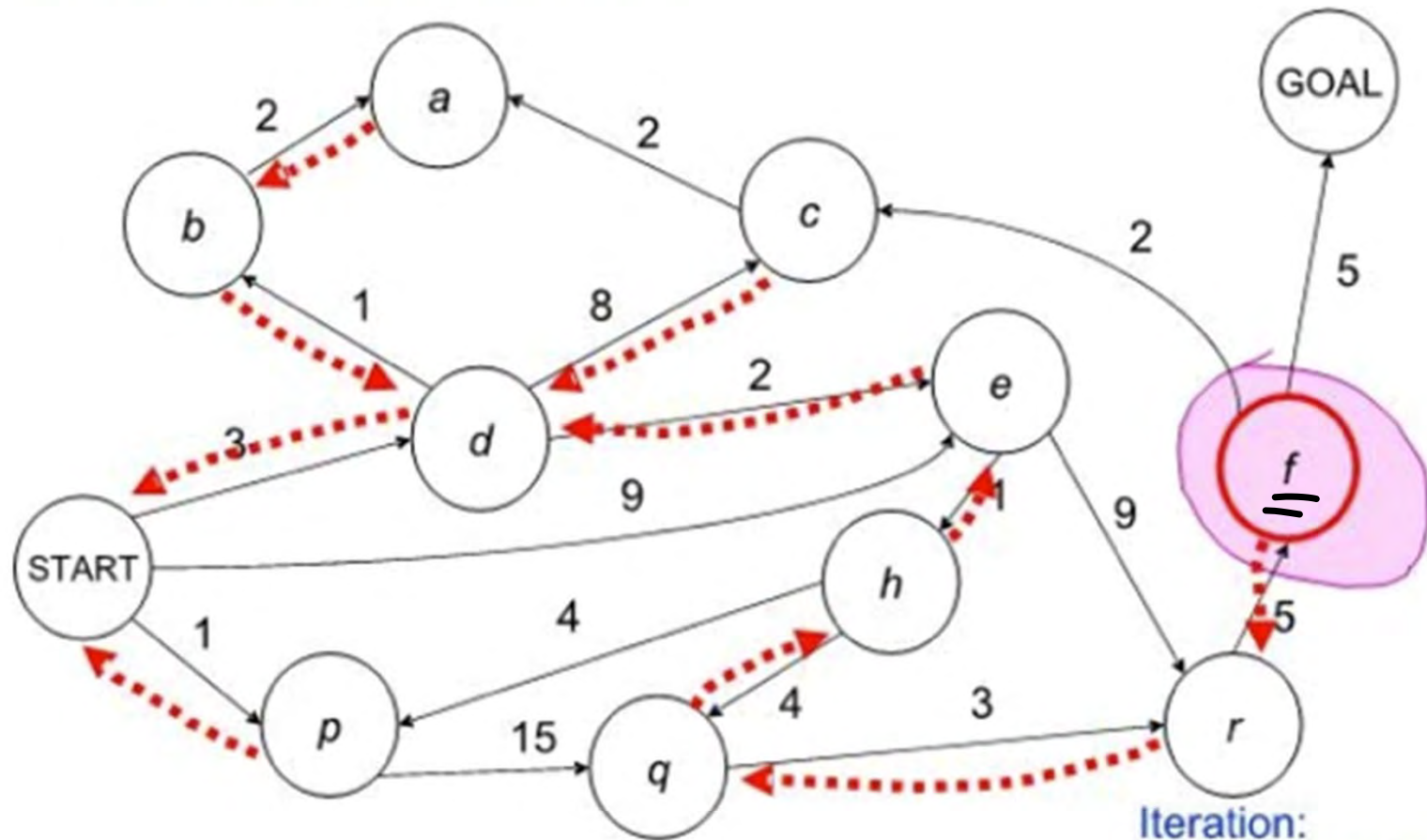
UCS Iterations



Frontier = {(r, 13)}

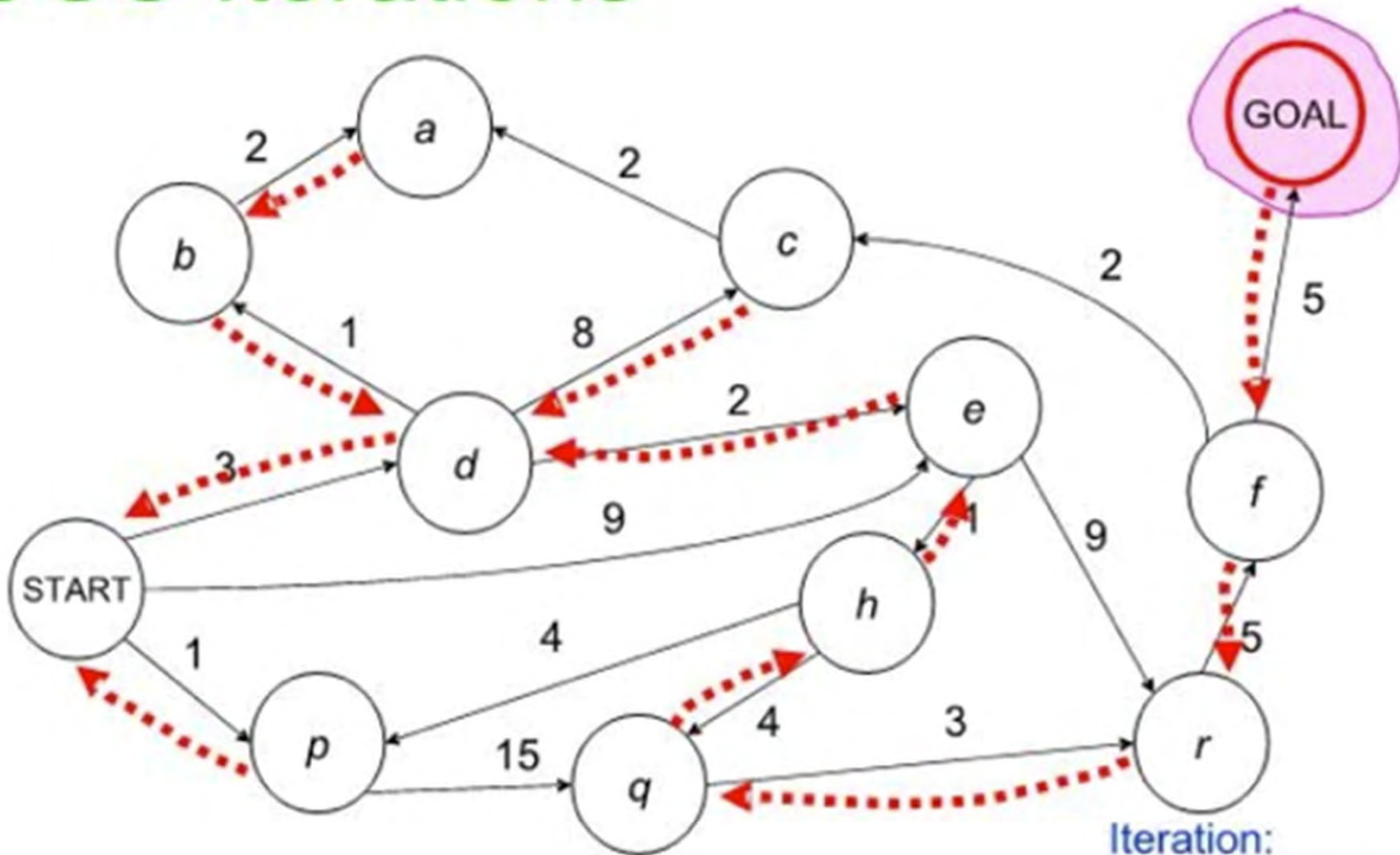
- Iteration:
1. Pop least-cost state
 2. Add successors

UCS Iterations



Frontier = {(f, 18)}

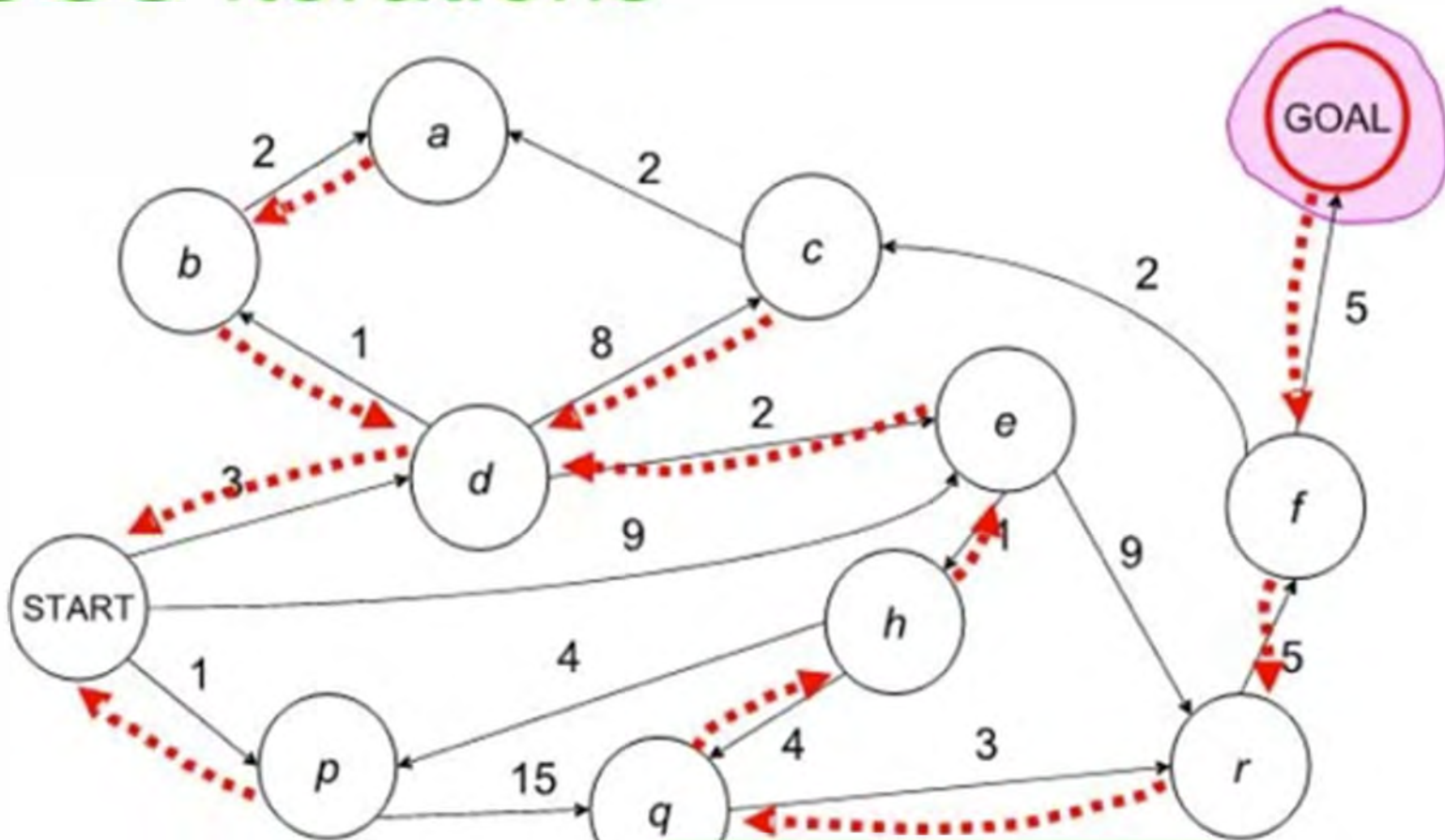
UCS Iterations



- Iteration:
1. Pop least-cost state
 2. Add successors

Frontier = {(GOAL,23)}

UCS Iterations

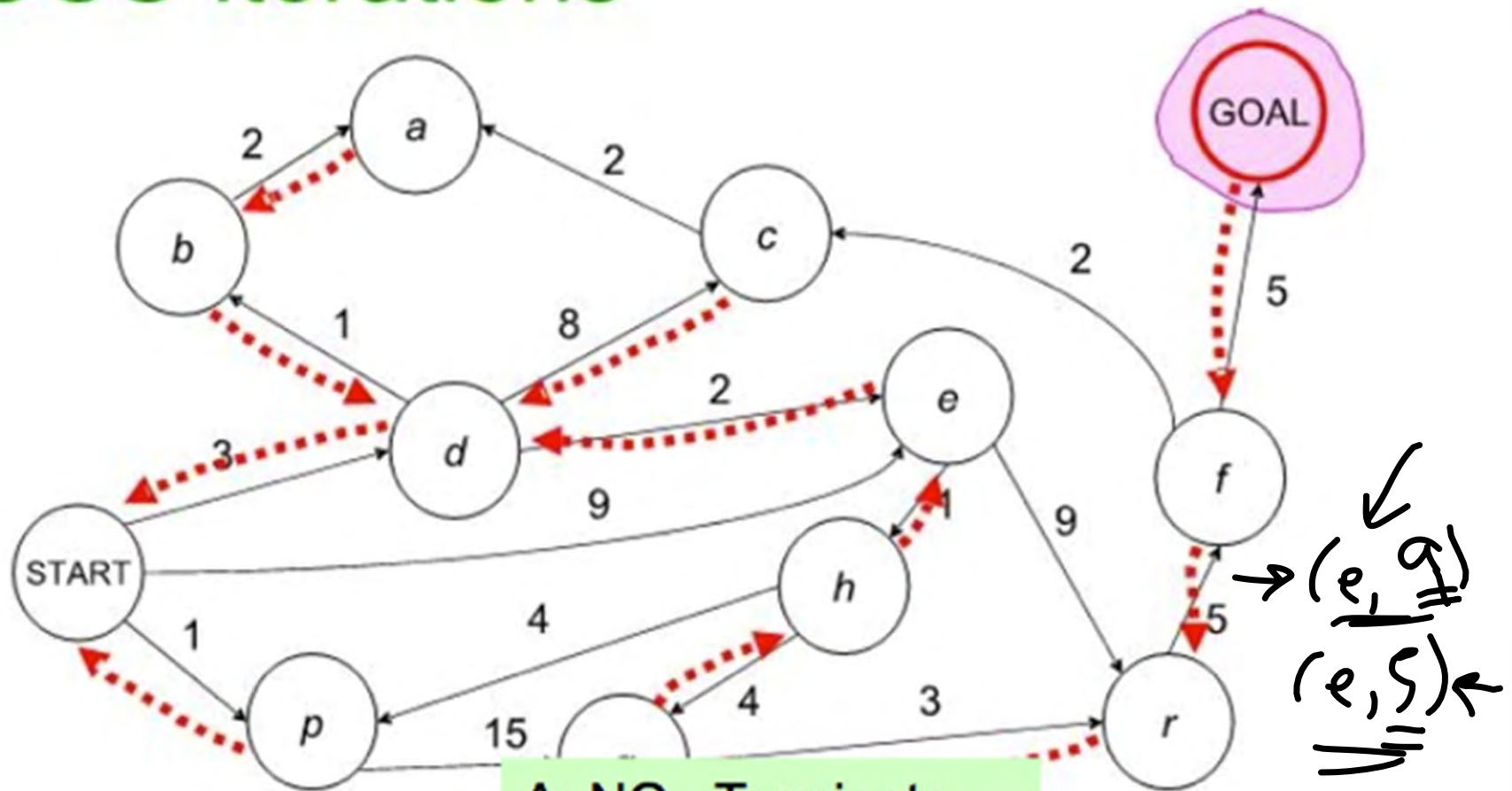


Q: are we done??

1. Pop least-cost state
2. Add successors

Frontier = {(GOAL,23)}

UCS Iterations



A: NO. Terminate only when goal is removed from Frontier.

- Iteration:
1. Pop least-cost state
 2. Add successors

Frontier = {}

Is cycle checking required to guarantee an optimal solution? * NO!!

Uniform-Cost Properties

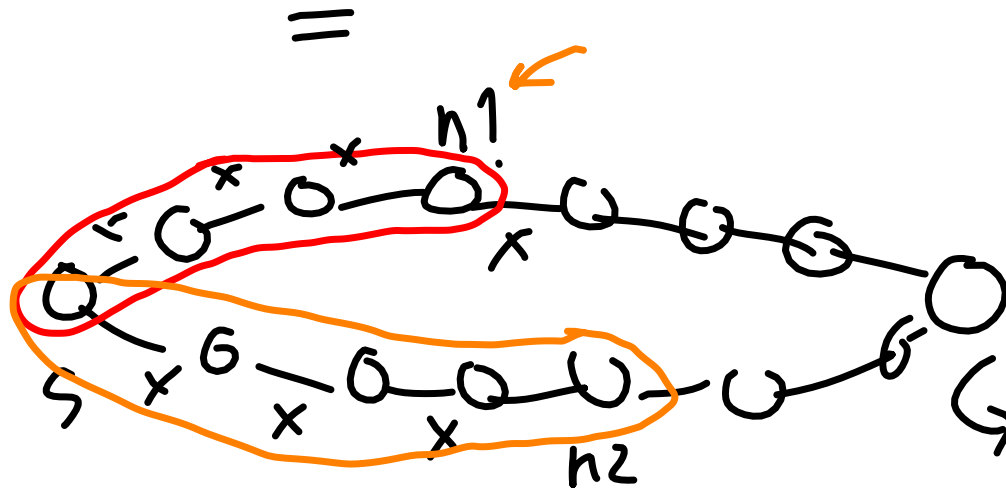
Optimality?

- **YES.** Let's prove this. Note that the arguments we see here will be used again when we examine heuristic search. We will move thru the proof as follows:
 1. Paths are expanded in order of increasing cost.
 2. Once you see a path with a cost X , you've expanded all paths with cost less than X .
 3. The first path you expand to a node (or state) is the cheapest path to that node (or state), be it to the goal or otherwise.

Uniform-Cost Search. Proof of Optimality

Given: each transition has cost $\geq \epsilon > 0$.

Lemma 1: Let $c(n)$ be the cost of node n on Frontier (cost of the path to n represented by $c(n)$). If n_2 is expanded IMMEDIATELY after n_1 then $c(n_1) \leq c(n_2)$.



Uniform-Cost Search. Proof of Optimality

Given: each transition has cost $\geq \epsilon > 0$.

Lemma 1: Let $c(n)$ be the cost of node n on Frontier (cost of the path to n represented by $c(n)$). If n_2 is expanded IMMEDIATELY after n_1 then $c(n_1) \leq c(n_2)$.

Proof of Lemma 1: there are 2 cases:

n_2 was on Frontier when n_1 was expanded:

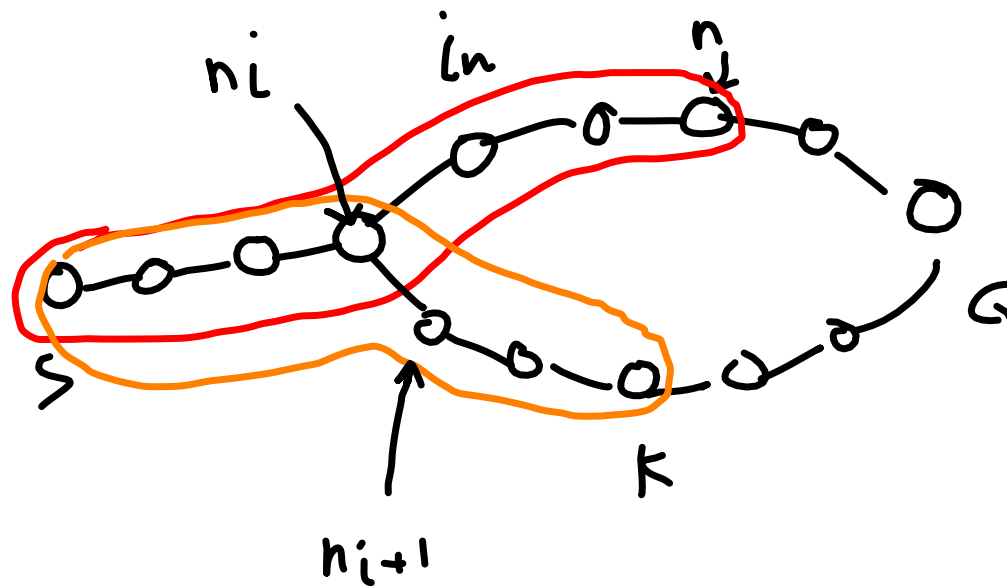
We must have $c(n_1) \leq c(n_2)$ otherwise n_2 would have been selected for expansion rather than n_1

n_2 was added to Frontier when n_1 was expanded:

Now $c(n_1) < c(n_2)$ since the path represented by n_2 extends the path represented by n_1 and thus costs at least ϵ more.

Uniform-Cost Search. 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.




$$c(h_{i+1}) < c(K) < c(n)$$

Uniform-Cost Search. 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:

- Assume we've just expanded n .
- Let $n_0 = \langle \text{Start} \rangle$
- Let $n_k = \langle \text{Start}, n_0, n_1, \dots, n_k \rangle$ be a path with cost less than $c(n)$, i.e. $\underline{c(n_k)} < \underline{c(n)}$.
- Let $\underline{n_i}$ be *the last node on this path expanded by our search*: $\langle \text{Start}, n_0, n_1, n_{i-1}, \underline{n_i}, n_{i+1}, \dots, n_k \rangle$
- So, $\underline{n_{i+1}}$ must still be on the frontier. Also $c(n_{i+1}) < c(n)$ since the cost of the entire path to n_k is $< c(n)$.
- But then uniform-cost would have expanded n_{i+1} not n . 
- So every node on this path must already be expanded as it is a lower cost path, i.e., this path has already been expanded.

Uniform-Cost Search. 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).

Uniform-Cost Search. 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.

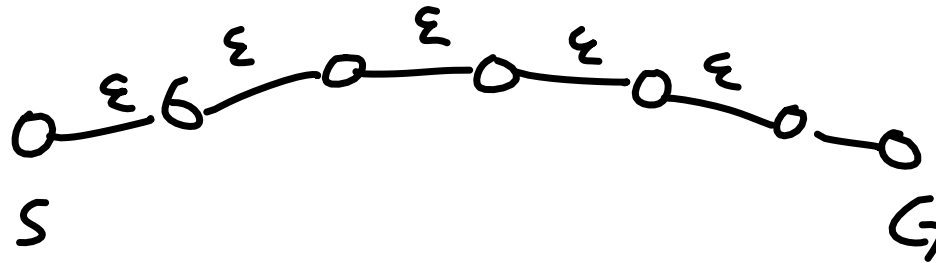
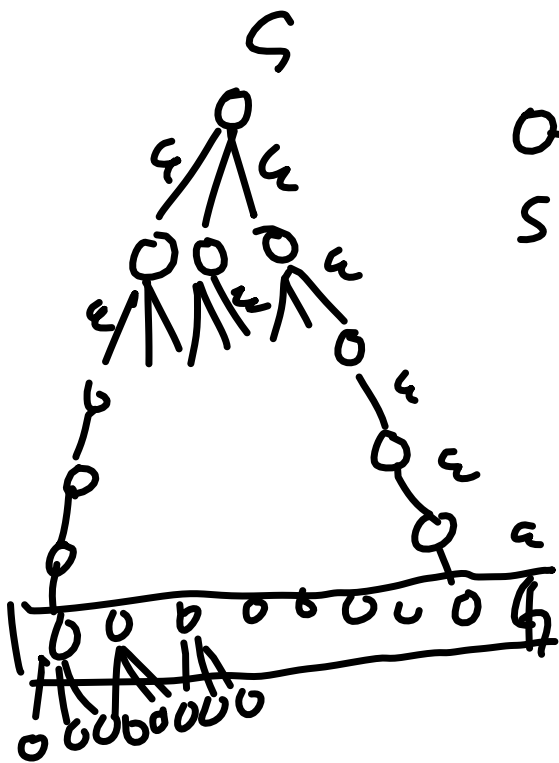
So, when a path to a goal state is expanded the path must be optimal (lowest cost).

Uniform-Cost Properties

- Completeness?
 - **YES.** Given positive, nonzero transition costs, the previous argument used for breadth first search holds: the cost of the path represented by each node n chosen to be expanded must be non-decreasing.

Uniform-Cost Properties

Time and Space Complexity? Assume each transition cost is $\geq \epsilon > 0$.



C^*/ϵ

$C^*(g)$

C^*

$\approx d$

Uniform-Cost Properties

Time and Space Complexity? Assume each transition cost is $\geq \epsilon > 0$.

↙ a lot like BFS.

- $O(b^{C^*/\epsilon + 1})$ where C^* is the cost of the optimal solution and ϵ the minimal cost of transitions.
- Paths with cost lower than C^* can be as long as C^*/ϵ (why not longer?)
- There may be many paths with cost $\leq C^*$; Uniform Cost Search must explore them all.
- We may have $b^{C^*/\epsilon}$ paths to explore and expand before finding the optimal cost path.

Consider a search where the start state is number 1 and the successor function for any state n returns two states: the numbers 2n and 2n + 1.

1. Draw the search tree that illustrates the state space for a search from the state 1 to the state 15

2. Say the initial state is 1 and the goal state is 11. List the order in which nodes will be visited for breadth-first search and iterative deepening search.

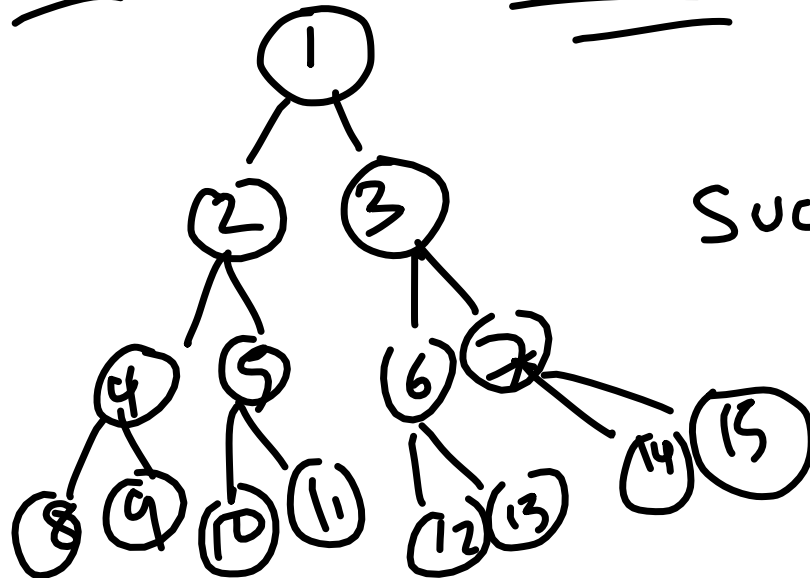
3. Can you design a bidirectional search for this problem, i.e. a search that would run simultaneously forwards (from the initial state) and backwards (from the goal state)? If so, describe in detail how this would work.

4. What'd be the time and space complexity for your bidirectional search algorithm?

1/x

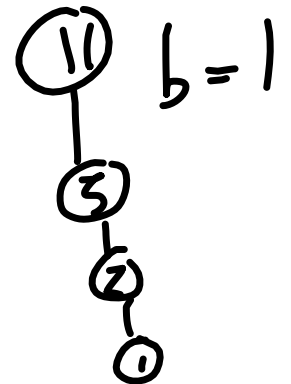
Answer here if you like <https://forms.gle/Uj1ai5omLd3xcuGB7>

Succ(n) → (2n, 2n+1)



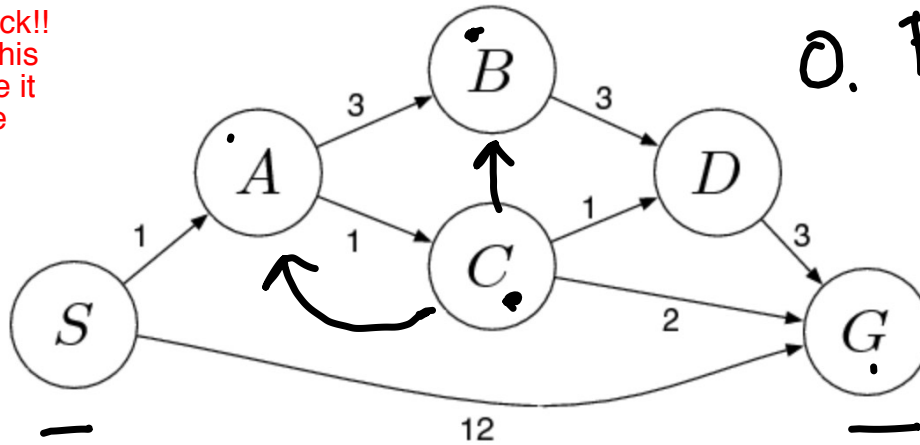
$Succ^{-1}(n) = \lfloor \frac{n}{2} \rfloor$

States: Integers
Start: 11
goal: 1



1. Illustrate the frontier at each iteration of Uniform Cost Search for a search from the start (S) to the goal (G) for the following problem:

Hello and Welcome back!!
 We will start by doing this problem together. Give it a try and we'll compare notes at 6:10.



S A B C
 X X X X
 4

0. $F = \{ \underline{S, 0} \}$

1. $\{ \underline{SA, 1}, SG, 12 \}$

2. $\{ \underline{SAC, 2}, SAB, 4, SG, 12 \}$

3. $\{ \underline{SACD, 3}$

$SACG, 4$ $SAB, 4$ $SG, 12 \}$

4. $\{ \underline{SACG, 4}, SAB, 4, \underline{SACDG, 6}, SG, 12 \}$

