Search

- Chapter 3 of R&N 3rd edition is very useful reading.
- Chapter 4 of R&N 3rd edition is worth reading for enrichment.

(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

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?

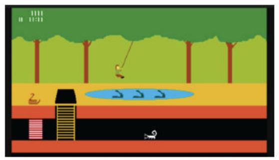
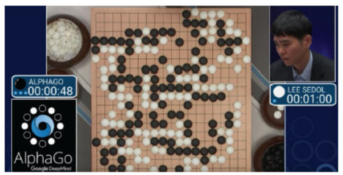
- We must take into account various preferences and constraints to develop a schedule.
- An important technique in developing such a schedule is "hypothetical" reasoning.
- Example: I'm on holiday in B.C.
  - If I fly into Vancouver and drive a car to Whistler, I'll have to drive on the roads at night. 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

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.

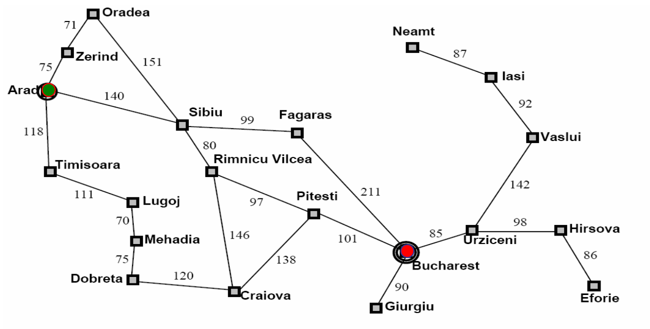
A solution

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.

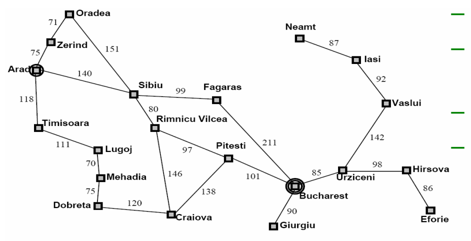
Example 1: Romania Travel

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:
- actions (successor functions):
- initial state:
- desired (or goal) condition:

Example 1: Romania Travel

- state space: the cities where you could be located.

  NB: In our abstraction: we are ignoring the low level details of driving, states where you are on the road between cities, etc.
- actions (successor functions): driving will advance you from one city to the next.
- initial state: in Arad
- desired (or goal) condition: be in a state where you are in Bucharest. (How many states satisfy this condition?)

A solution will be a sequence of cities to travel through to get to Bucharest

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:

–actions (successor functions):

–initial state:

–desired (or goal) condition:

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.

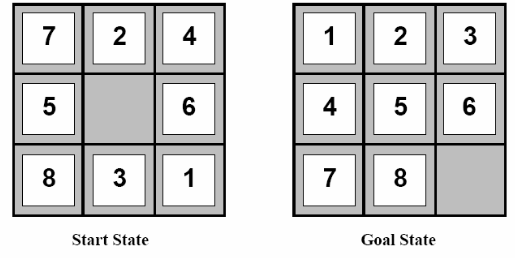
–initial state: Various, e.g., (0,0)

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

Reflections on the Water Jug Problem

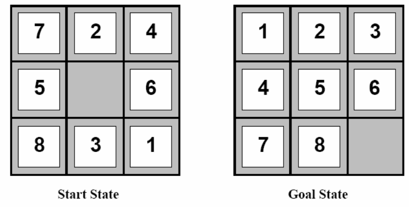
- If we start off with gal3 and gal4 as integers, can only reach integer values.
- Some values, e.g., (1,2) are not reachable from some initial state, e.g., (0,0).
- Some actions are no-ops. They do not change the state, e.g.,
  - (0,0) → Empty-3-Gallon → (0,0)

Example 3: The 8-Puzzle



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

Example 3: The 8-Puzzle



– state space:
– actions (successor functions):
– initial state:
– desired (or goal) condition:

Example 3: The 8-Puzzle

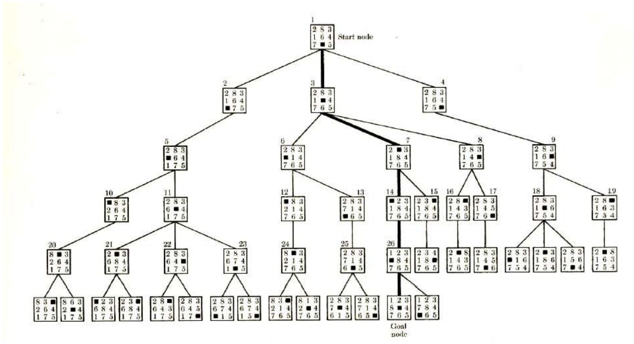
– state space: the different configurations of the tiles. How many different states?
– actions (or successor functions): moving the blank up, down, left, right. *Can every action be performed in every state?*
– initial state: e.g., state shown on previous slide.
– desired (or goal) condition: a state 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

- Although there are 9! different configurations of the tiles (362,880) in fact the state space is divided into two disjoint parts.
- Only when the blank is in the middle are all four actions possible.
- Our goal condition is satisfied by only a single state. But one could easily have a goal condition like:
  – The 8 is in the upper left hand corner.
    - How many different states satisfy this goal?

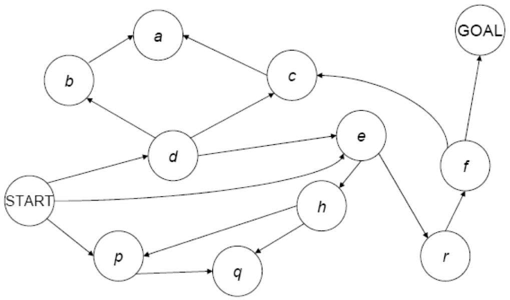
Search Space for 8-Puzzle Problem



More complex situations

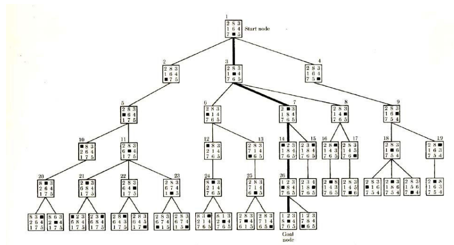
- Sometimes, actions may lead to multiple states, like flipping a coin.
- Other times, we may not be sure of a given state (prize is behind door 1, 2, or 3). In these situations, we might want to consider how likely different states and action outcomes are.
- Later we will see some techniques for reasoning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

Drawing Search: Graphical Representation



It can sometimes be useful to represent a search space 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 Problem (Tree)



Search spaces can be represented by a particular kind of graph called a tree; 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) = {set of states that can be reached from state x 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) = ∞ if a does not yield y from x). Note that different actions might generate the same move of x → 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) yields a set of states that can be reached from x via any single action.

- It may be helpful to annotate states by the action used to obtain them:
  - S(x) = {<y,a>, <z,b>}
    arrive at y via action a, arrive at z via action b.
  - S(x) = {<y,a>, <y,b>}
    arrive at y via action a, also arrive at y via alternative action b.
- It may also be important to reference parents of annotated states (i.e. to store the state that came immediately prior a given action).
- The book's annotation of each state (or node) also includes depth and cost.

A Searching Template

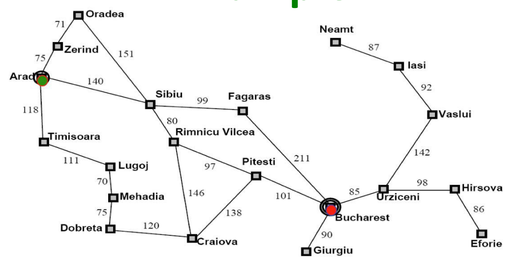
- To explore the state space during a search, we will iteratively apply the successor function to the states we discover.
- Each time, the successor function S(x) will yield a set of states that can be reached from x via any single action.
- As we search, we can annotate states by the action used to obtain them in order to keep a record of paths to a state:
  - S(x) = {<y,a>, <z,b>}
    arrive at y via action a, arrive at z via action b.
  - S(x) = {<y,a>, <z,b>}
    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.

```
TreeSearch(Frontier, Sucessors, 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?)
```

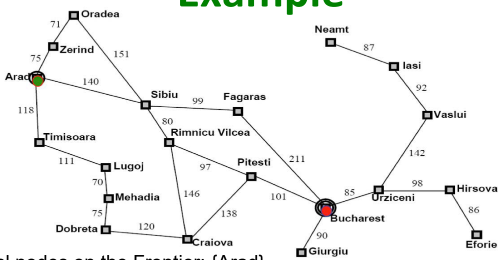

Example



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>**}

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

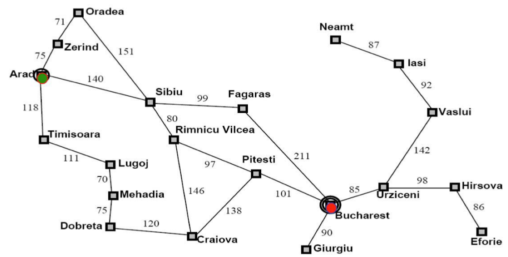
Example



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>**}

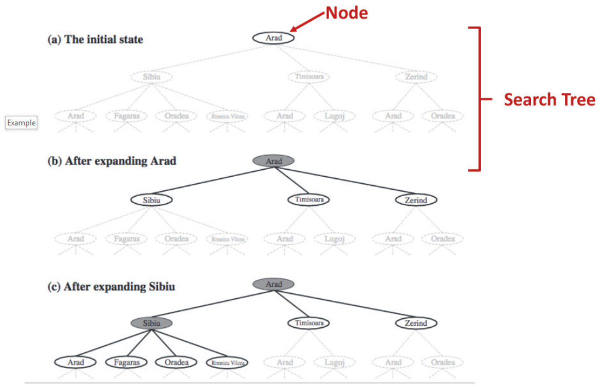
Solution is now on frontier; cost of this solution is **140+80+97+101= 418**

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.

Search Tree Representation



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) than can be expanded or generated?
- Space complexity: what is the maximum number of nodes (paths) that have to be stored in memory?

Uninformed Search Strategies

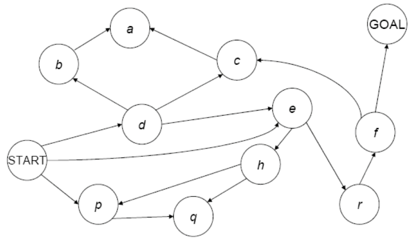
- These are strategies that adopt a fixed rule for selecting the next state to be expanded.
- The rule remains the same for any search problem being solved; it does not change.
- These strategies do not take into account any domain specific information about the particular search problem.
- Uninformed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

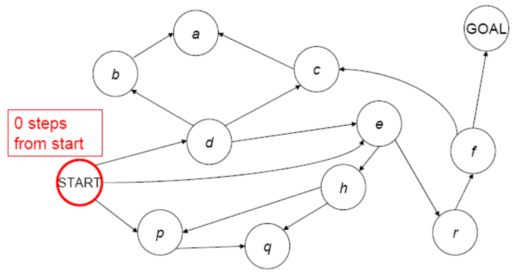
1. Order the elements on the Frontier.
2. Always select the first element.

Breadth First Search

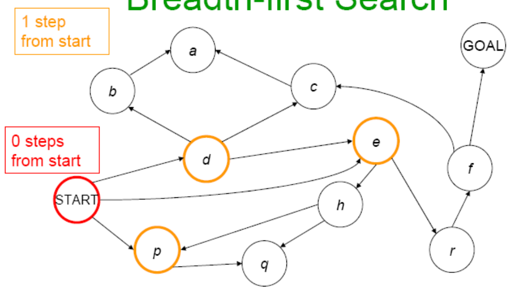


1. Place Start in the Frontier.
2. Expand all nodes reachable from Start in 1 step, but not more than 1; add path to back of Frontier list.
3. Expand all nodes reachable from Start in 2 step, but not more than 2; add path to back of Frontier list.
4. Expand all nodes reachable from Start in 3 step, but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

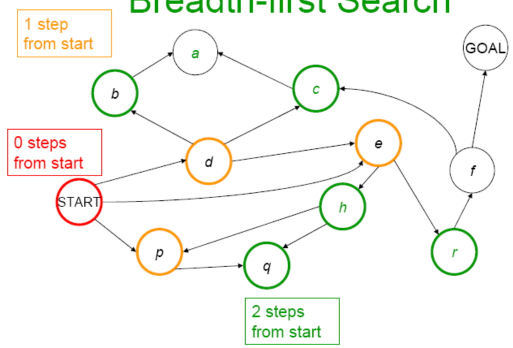
Breadth-first Search



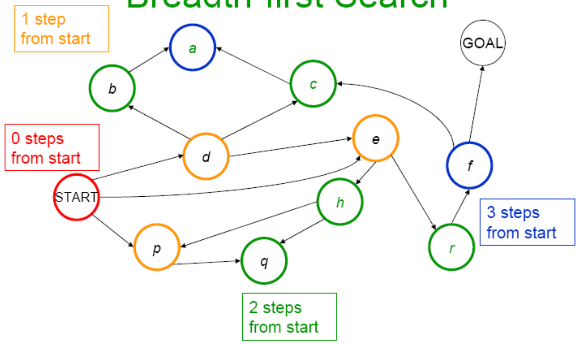
Breadth-first Search



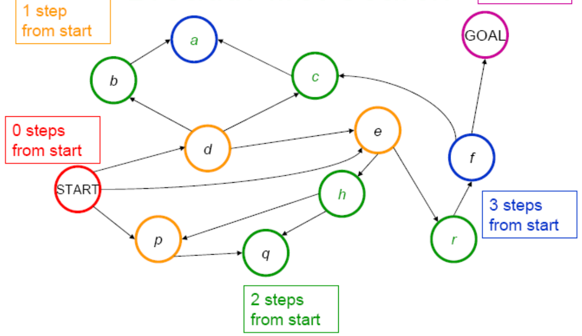
Breadth-first Search



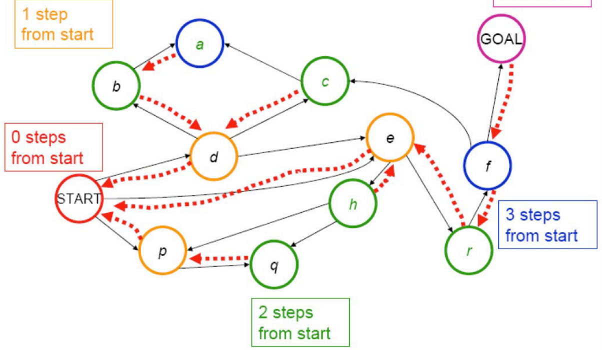
Breadth-first Search



Breadth-first Search



Solved!!



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

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)>}

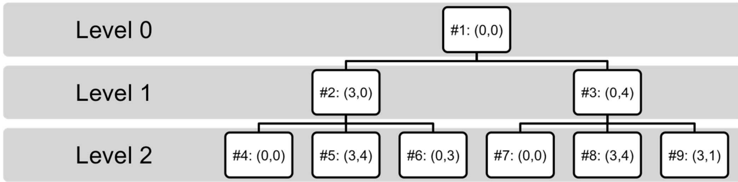
2.Frontier = {<(0,0),(3,0)>, <(0,0),(0,4)>}

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 = length of path - 1

What is the Time Complexity?

Breadth-First Properties

Measuring time and space complexity.

- let b be the maximum number of successors of any node (maximal branching factor).
- let d be the depth of the shortest solution.
  - Root at depth 0 generates a path of length 1
  - So d = length of path - 1

What is the Time Complexity?

$1 + b + b^2 + b^3 + \ldots + b^{d-1} + b^d + b(b^d - 1) = O(b^{d+1})$

Breadth-First Properties

Space Complexity?

Breadth-First Properties

Space Complexity?

$O(b^{d+1})$: If goal node is last node at level d, all of the successors of the other nodes will be on the Frontier when the goal node is expanded $b(b^d - 1)$

Optimality?

Breadth-First Properties

Space Complexity?

$O(b^{d+1})$: If goal node is last node at level d, all of the successors of the other nodes will be on the Frontier when the goal node is expanded $b(b^d – 1)$

Optimality?

We will find the shortest length solution. *Is this the least cost solution?*

Completeness?

Breadth-First Properties

Space Complexity?

$O(b^{d+1})$: If goal node is last node at level d, all of the successors of the other nodes will be on the Frontier when the goal node is expanded $b(b^d – 1)$

Optimality?

We will find the shortest length solution. *Is this the least cost solution?*

Completeness?

Eventually we must examine all paths of length d, and thus we will find a solution if one exists.

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

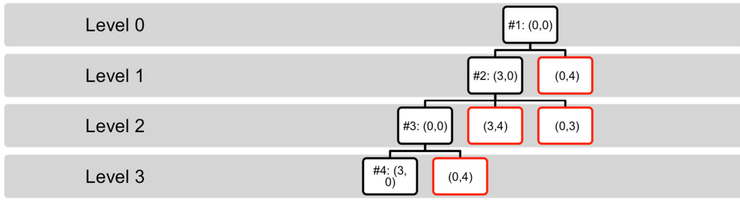
Like BFS, but instead of at the back we place the new paths that extend the current path at the front of the Frontier.

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)>}

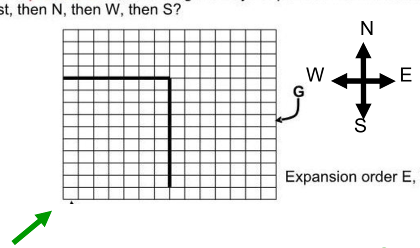
Depth-First Search



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?

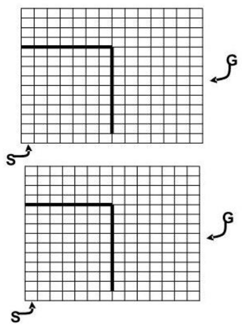


Expansion order E, N, W, S

start

What would BFS do?

Two other DFS examples



Order: N, E, S, W?

Order: N, E, S, W
with loops prevented

Depth-First Properties

Complete?

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?

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

Depth-First Properties

Time Complexity?

Depth-First Properties

Time Complexity?

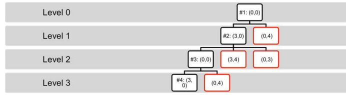
- **$O(b^m)$** where m is the length of the longest path in the state space.
- Very bad if m is much larger than d (shortest path to a goal state), but if there are many solution paths it can be much faster than breadth first (by good luck, can bump into a solution quickly).

Space Complexity?

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 ≤ d exists.

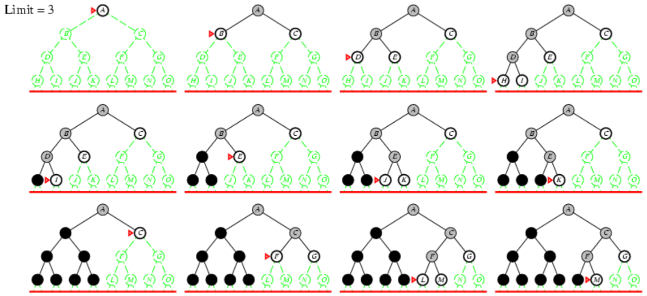
Depth Limited Search

```
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



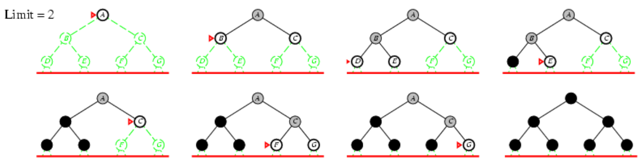
Iterative Deepening Search

- 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 → no solution exists.

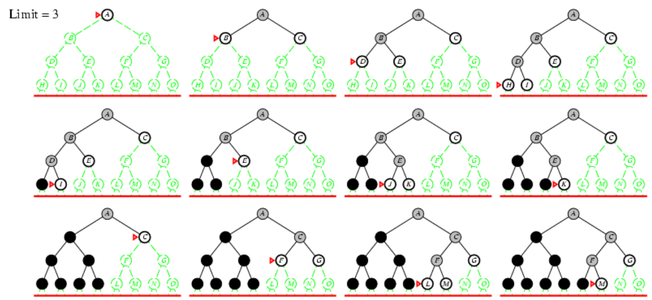
Iterative Deepening Search

Iterative Deepening Search

Iterative Deepening Search



Iterative Deepening Search



Iterative Deepening Search

Completeness?

- YES, if a minimal depth solution of depth d exists.
  - What happens when the depth limit L=d?
  - What happens when the depth limit L<d?

Time Complexity?

Iterative Deepening Search

Time Complexity?

- $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- E.g. b=4, d=10
  - $(11)*4^0 + 10*4^1 + 9*4^2 + \ldots + 4^{10} = 1{,}864{,}131$
  - $4^{10} = 1{,}048{,}576$
  - Most nodes lie on bottom layer.

BFS can explore more states than IDS!

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

E.g. b=4, d=10

- For IDS
  – $(11)*4^0 + 10*4^1 + 9*4^2 + \ldots + 4^{10} =$ 1,864,131 (states generated)
- For BFS
  – $1 + 4 + 4^2 + \ldots + 4^{10} + 4(4^{10} - 1) =$ 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 a the bottom layer it will add many nodes to Frontier before finding the goal node.

Iterative Deepening Search Properties

Space Complexity?

– O(bd) … still linear!

Optimal?

– Will find shortest length solution which is optimal 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. Need as many iterations of the search as there are distinct path costs.

Path Checking

Recall that paths are commonly stored on the Frontier.

If nk represents the path $<s_0,s_1,\ldots,s_k>$ and we expand s_k to obtain child c, we have

$<s_0,s_1,\ldots,s_k,c>$

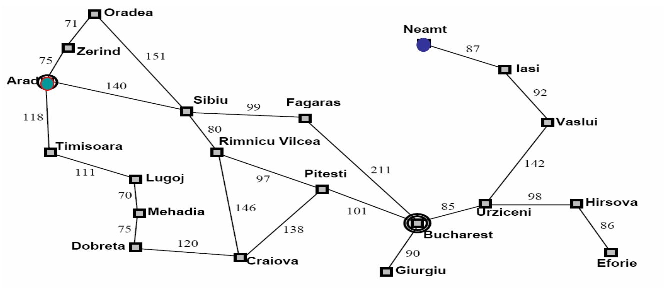
as the path to "c".

Path checking:

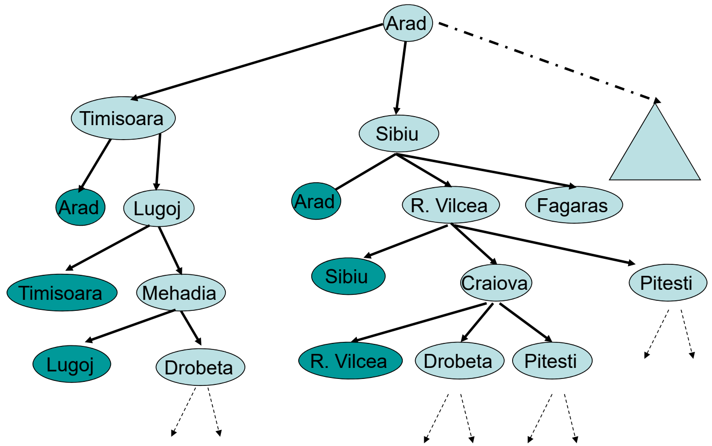
–Ensure that the state c is not equal to the state reached by any ancestor of c along this path.

–Paths are checked in isolation!

Example: Arad to Neamt



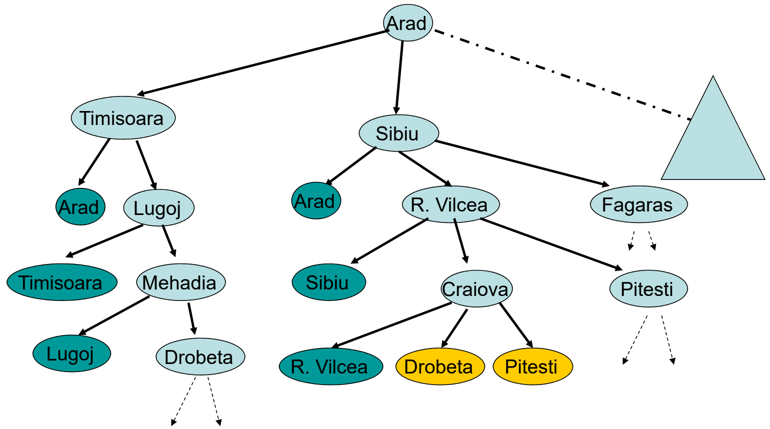
Path Checking Example



Cycle Checking

- 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?**

Cycle Checking Example (BFS)



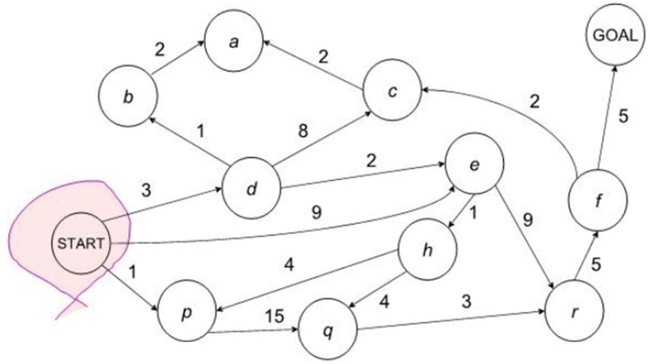
Cycle Checking

- 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

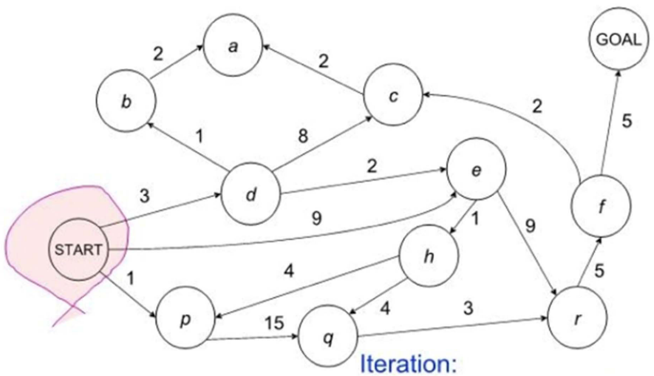
- Keeps Frontier ordered by increasing cost of the path *(know a good data structure for this?)*
- Always expand the least cost path.
- Identical to Breadth First Search if each action has the same cost

Starting UCS



Frontier = {(START,0)}

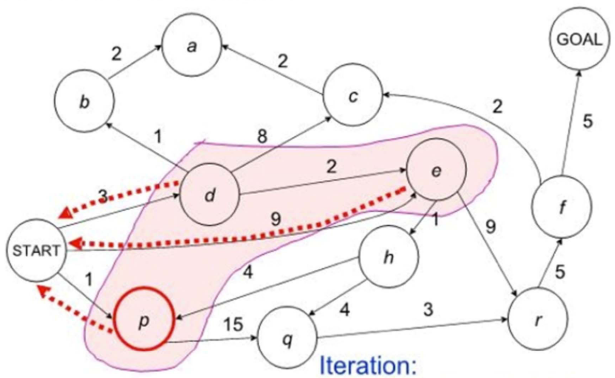
UCS Iterations



Frontier = {(START,0)}

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

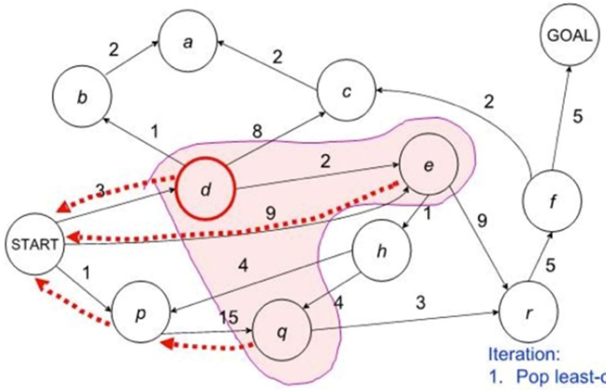
UCS Iterations



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

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

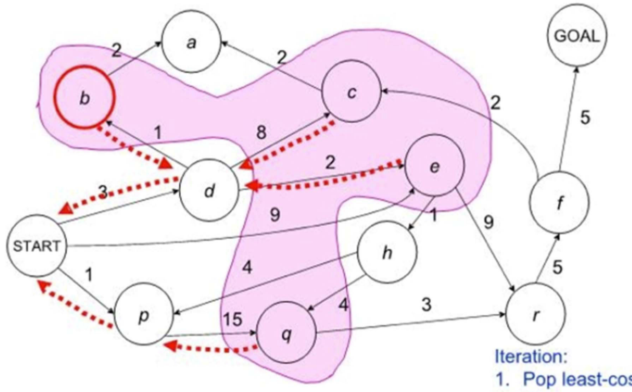
UCS Iterations



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

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

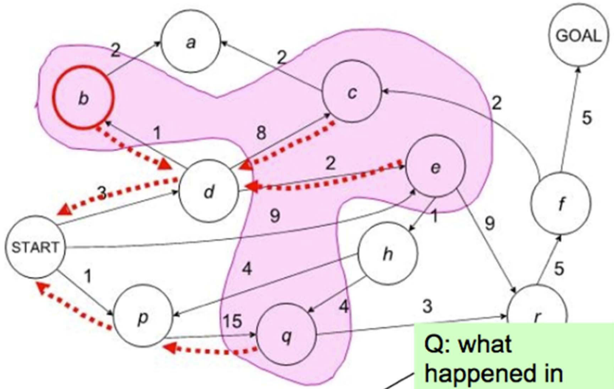
UCS Iterations



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

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

UCS Iterations

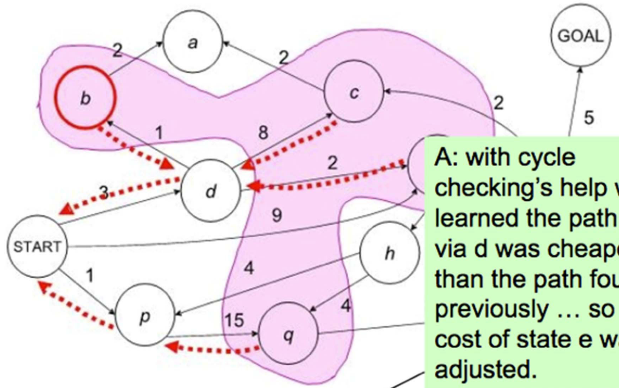


Q: what happened in here????

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

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

UCS Iterations

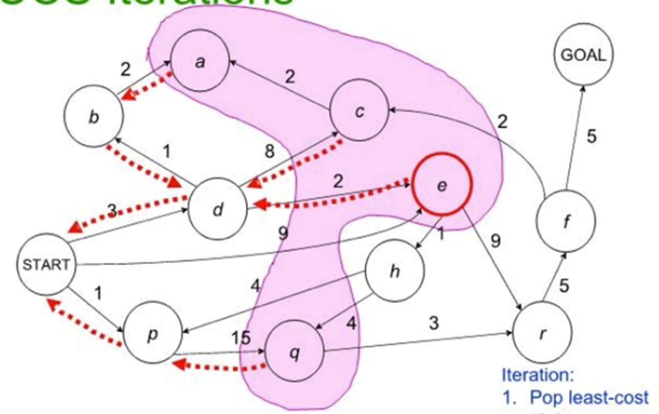


A: with cycle checking's help we learned the path to e via d was cheaper than the path found previously … so the cost of state e was adjusted.

state
2. Add successors

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

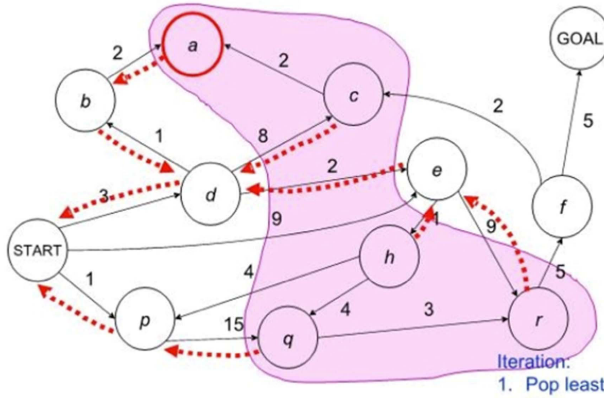
UCS Iterations



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

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

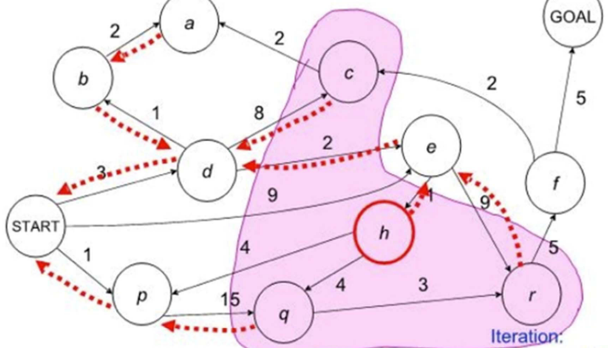
UCS Iterations



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

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

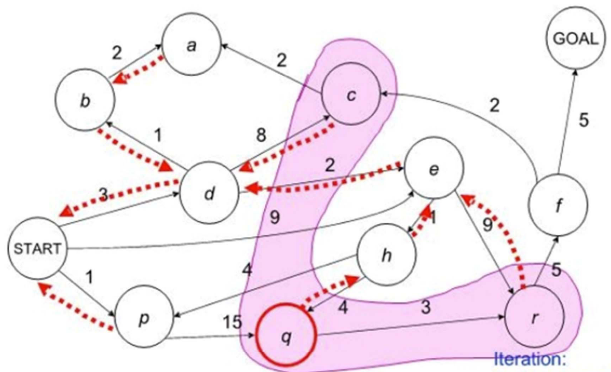
UCS Iterations



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

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

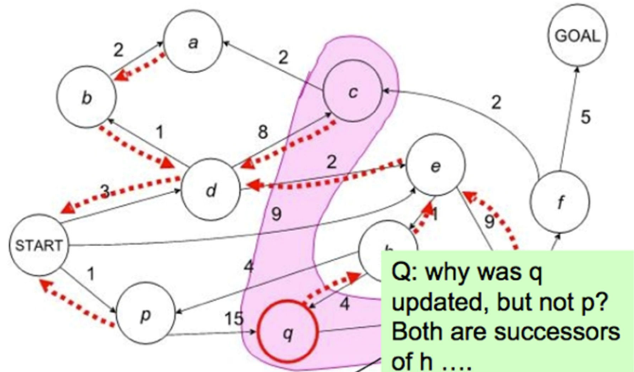
UCS Iterations



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

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

UCS Iterations

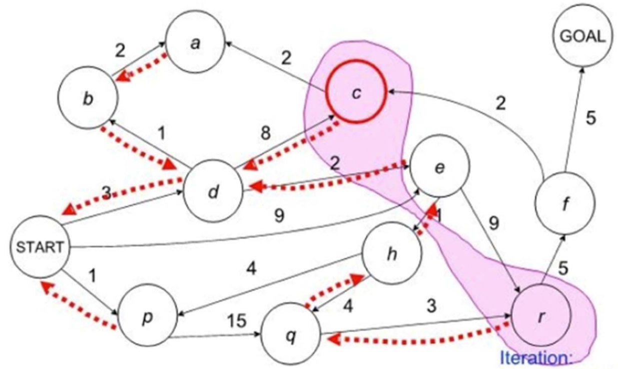


Q: why was q updated, but not p? Both are successors of h ….

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

state
2. Add successors

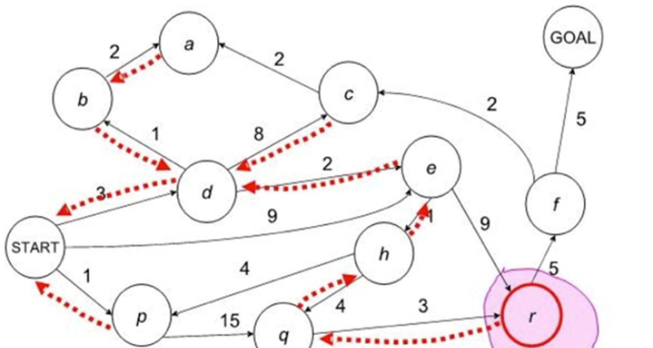
UCS Iterations



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

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

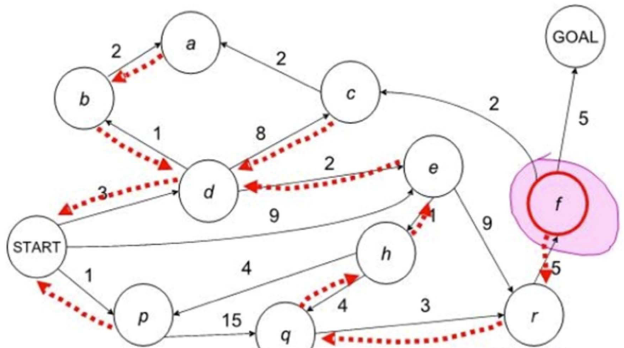
UCS Iterations



Frontier = {(r,13)}

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

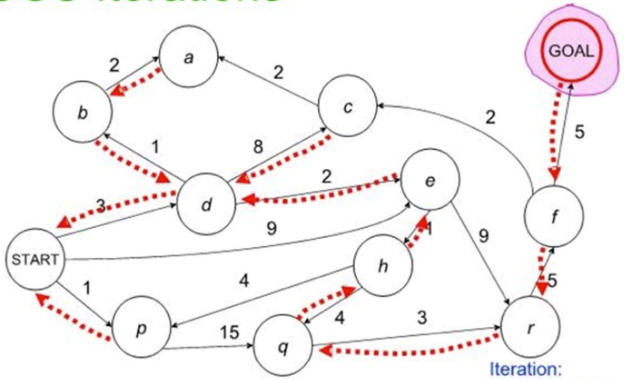
UCS Iterations



Frontier = {(f,18)}

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

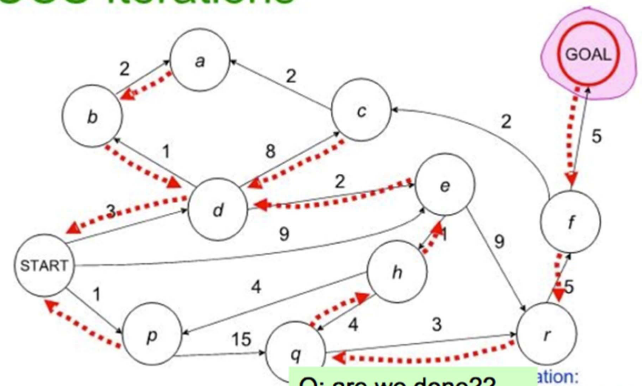
UCS Iterations



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

Frontier = {(GOAL,23)}

UCS Iterations

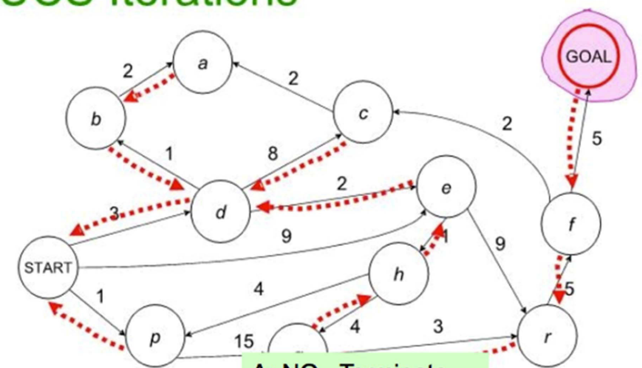


Q: are we done??

ation:
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 = {}

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.

Is cycle checking required to guarantee an optimal solution?

Uniform-Cost Search. Proof of Optimality

Given: each transition has cost ≥ ε > 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 n2 is expanded IMMEDIATELY after n1 then c(n1) ≤ c(n2).

When n1 was expanded the Frontier could have looked one of two ways. What are these?

Uniform-Cost Search. Proof of Optimality

Given: each transition has cost ≥ ε > 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 n2 is expanded IMMEDIATELY after n1 then c(n1) ≤ c(n2).

Proof of Lemma 1: there are 2 cases:

n2 was on Frontier when n1 was expanded:

We must have c(n1) ≤ c(n2) otherwise n2 would have been selected for expansion rather than n1

n2 was added to Frontier when n1 was expanded:

Now c(n1) < c(n2) since the path represented by n2 extends the path represented by n1 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.

Proof:

- Assume we've just expanded n.
- Let n0 = <Start>
- Let nk = <Start, n0, n1, …, nk> be a path with cost less than c(n), i.e. c(nk) < c(n).
- Let **ni** be *the last node on this path expanded by our search:* <Start, n0, n1, ni-1, **ni**, ni+1, …, nk>
- So, **ni+1** must still be on the frontier. Also c(ni+1) < c(n) since the cost of the entire path to nk is < c(n).
- But then uniform-cost would have expanded ni+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).

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?

Assuming each transition cost is $\geq \varepsilon > 0$.

- $O(b^{C^*/\varepsilon + 1})$ 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 $b^{C^*/\varepsilon}$ paths to explore and expand before finding the optimal cost path.