UNIVERSITY OF LOUISVILLE
J.B. SPEED SCHOOL OF ENGINEERING

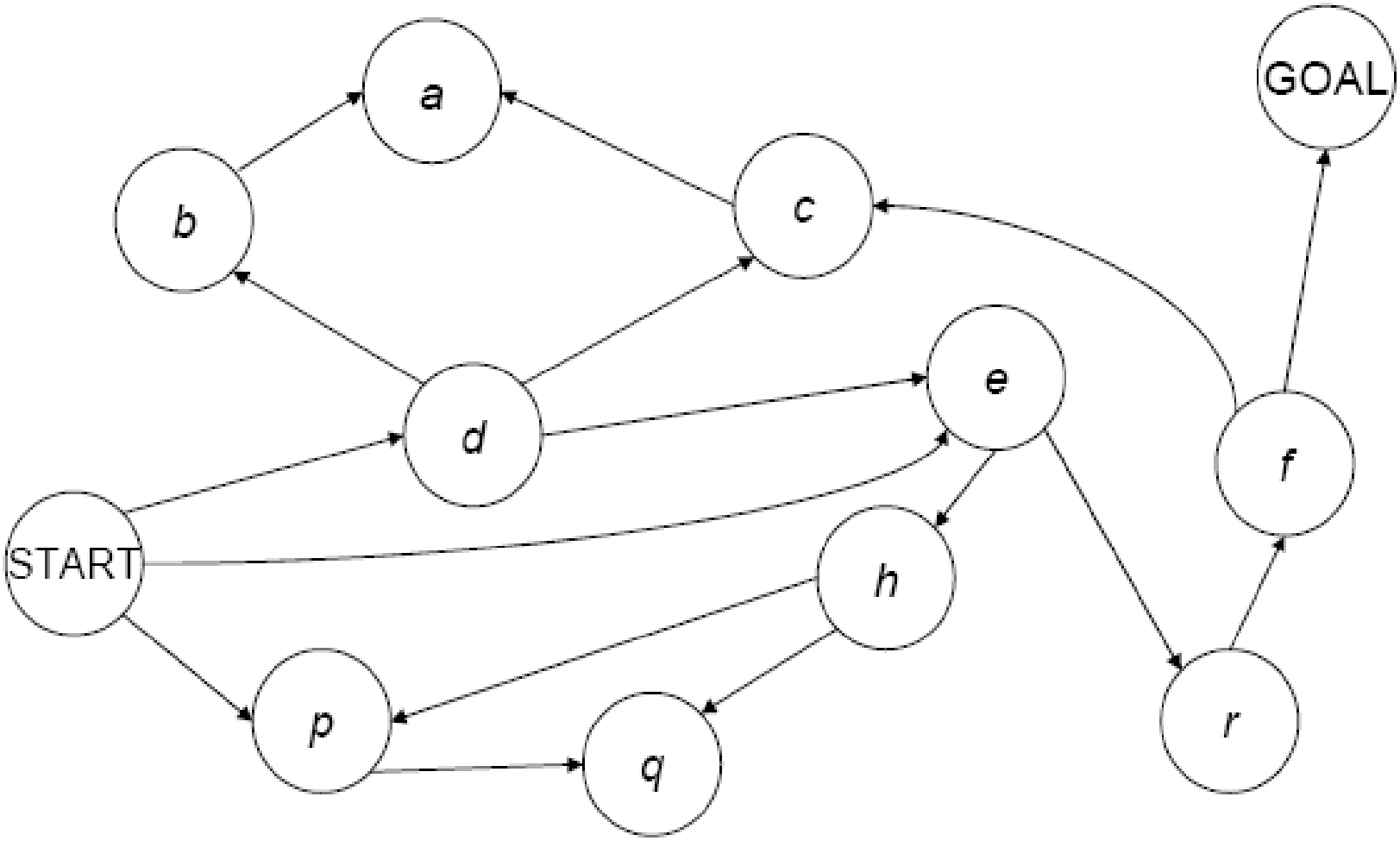
Computer Engineering and Computer Science Department

CECS545-Artificial Intelligence

Search

Dr. Roman Yampolskiy

A search problem



How do we get from S to G? And what's the smallest possible number of transitions?

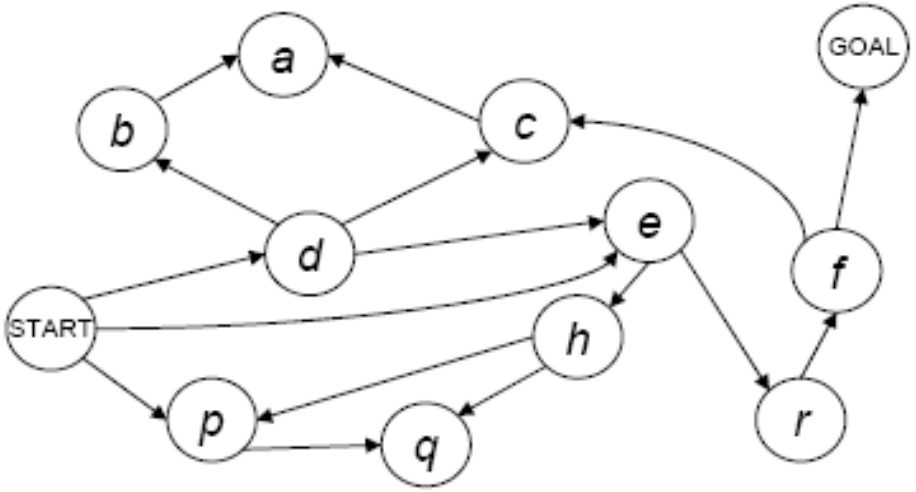
Formalizing a search problem

A search problem has five components:

Q , S , G , **succs** , **cost**

- Q is a finite set of states.
- $S \subseteq Q$ is a non-empty set of start states.
- $G \subseteq Q$ is a non-empty set of goal states.
- **succs** : $Q \rightarrow P(Q)$ is a function which takes a state as input and returns a set of states as output. **succs**(s) means "the set of states you can reach from s in one step".
- **cost** : Q , $Q \rightarrow$ *Positive Number* is a function which takes two states, s and s', as input. It returns the one-step cost of traveling from s to s'. The cost function is only defined when s' is a successor state of s.

Our Search Problem



Q = {START, *a* , *b* , *c* , *d* , *e* , *f* , *h* , *p* , *q* , *r* , GOAL}

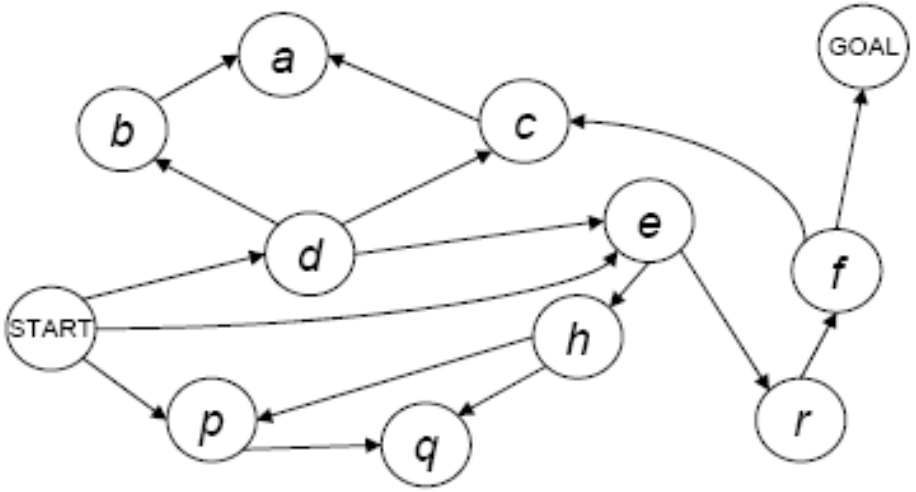
S = { START }

G = { GOAL }

succs(*b*) = { *a* }
succs(*e*) = { *h* , *r* }
succs(*a*) = NULL … etc.

cost(*s*,*s'*) = 1 for all transitions

Our Search Problem



Q = {START, *a* , *b* , *c* , *d* , *e* , *f* , *h* , *p* , *q* , *r* , GOAL}

S = { START }

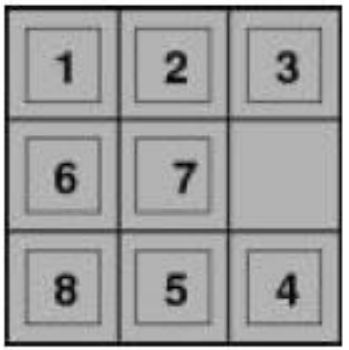
G = { GOAL }

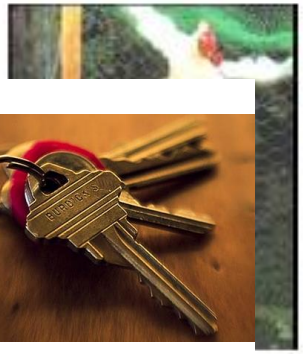
succs(*b*) = { *a* }
succs(*e*) = { *h* , *r* }
succs(*a*) = NULL … etc.

cost(*s*,*s'*) = 1 for all transitions

What problems do you think are like this?

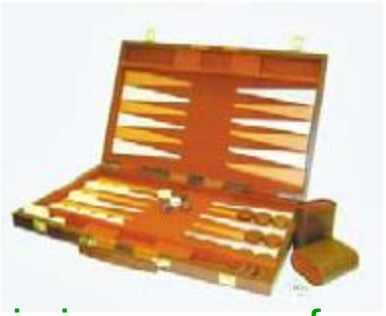
Search Problems





Our definition excludes…

This is a game against an adversary



This is a game of chance

This one has hidden state

This one has an infinite number of states

This one → has all of the problems above

Problem Solving Agents

- Looking to satisfy some **goal**
  - Wants environment to be in particular state
- Have a number of possible actions
  - An action changes environment

- What sequence of actions reaches the goal?
  - Many possible sequences
- Agent must **search** through sequences

Examples of Search Problems

- Chess
  - Each turn, search moves for win
- Route finding
  - Search routes for one that reaches destination
- Theorem proving
  - Search chains of reasoning for proof
- Machine learning
  - Search through concepts for one which achieves target categorisation

Search Terminology

- **States**: “places” the search *can* visit
- **Search space**: the set of possible states
- **Search path**
  - Sequence of states the agent *actually* visits
- **Solution**
  - A state which solves the given problem
    - Either known or has a checkable property
  - May be more than one solution
- **Strategy**
  - How to choose the next state in the path at any given state

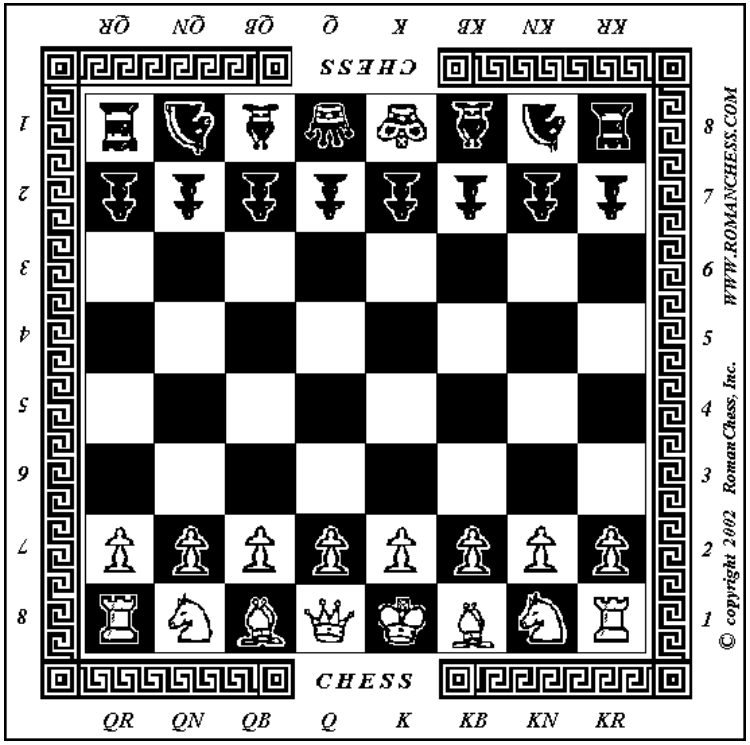
Specifying a Search Problem

1. Initial state
   - Where the search starts
2. Operators
   - Function taking one state to another state
   - How the agent moves around search space
3. Goal test
   - How the agent knows if solution state found

Search strategies apply operators to chosen states

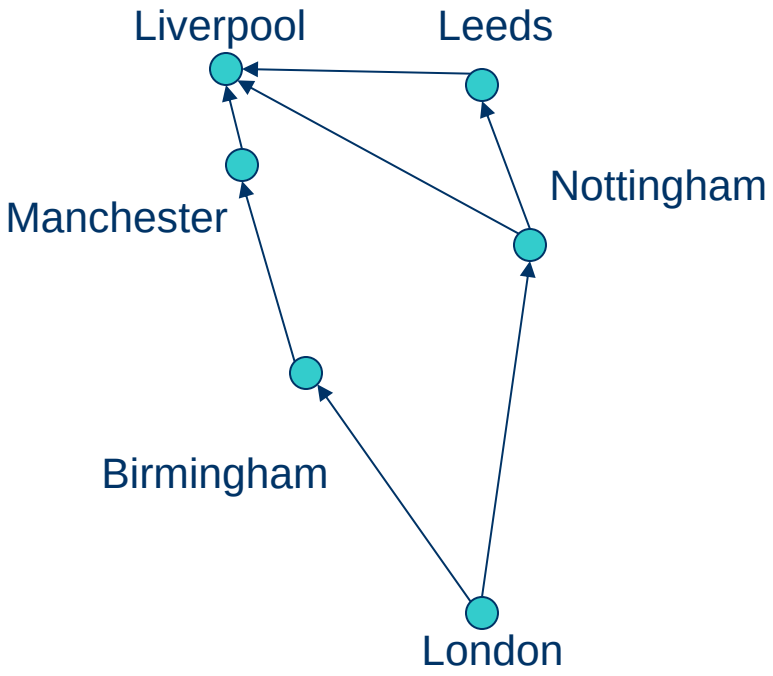
Example: Chess

- Initial state (right)
- Operators
  - Moving pieces
- Goal test
  - Checkmate
    - Can the king move without being taken?



Example: Route Finding

- Initial state
  - City journey starts in
- Operators
  - Driving from city to city
- Goal test
  - Is current location the destination city?



General Search Considerations

1. Artefact or Path?

- Interested in solution only, or path which got there?
- Route finding
  - Known destination, must find the route (**path**)
- Anagram puzzle
  - Doesn't matter how you find the word
  - Only the word itself (**artefact**) is important
- Machine learning
  - Usually only the concept (**artefact**) is important
- Theorem proving
  - The proof is a sequence (**path**) of reasoning steps

General Search Considerations

2. Completeness

- Task may require one, many or all solutions
  - E.g. how many different ways to get from A to B?
- **Complete** search space contains all solutions
  - Exhaustive search explores entire space (assuming finite)
- **Complete** search strategy will find solution if one exists
- **Pruning** rules out certain operators in certain states
  - Space still complete if no solutions pruned
  - Strategy still complete if not all solutions pruned

General Search Considerations

3. Soundness

- A sound search contains only correct solutions
- An unsound search contains incorrect solutions
  - Caused by unsound operators or goal check

- Dangers
  - find solutions to problems with no solutions
    - find a route to an unreachable destination
    - prove a theorem which is actually false
    - (Not a problem if all your problems have solutions)
  - produce incorrect solution to problem

General Search Considerations

4. Time & Space Tradeoffs

- Fast programs can be written
  - But they often use up too much memory
- Memory efficient programs can be written
  - But they are often slow

- Different search strategies have different memory/speed tradeoffs

General Search Considerations

5. Additional Information

- Given initial state, operators and goal test
  - Can you give the agent additional information?
- **Uninformed** search strategies
  - Have no additional information
- **Informed** search strategies
  - Uses problem specific information
  - Heuristic measure (Guess how far from goal)

Graph and Agenda Analogies

- Graph Analogy
  - States are nodes in graph, operators are edges
  - Expanding a node adds edges to new states
  - Strategy chooses which node to expand next

- Agenda Analogy
  - New states are put onto an agenda (a list)
  - Top of the agenda is explored next
    - Apply operators to generate new states
  - Strategy chooses where to put new states on agenda

Example Search Problem

- A genetics professor
  - Wants to name her new baby boy
  - Using only the letters D,N & A
- Search through possible strings (states)
  - D,DN,DNNA,NA,AND,DNAN, etc.
  - 3 operators: add D, N or A onto end of string
  - Initial state is an empty string
- Goal test
  - Look up state in a book of boys' names, e.g. DAN

Uninformed Search Strategies

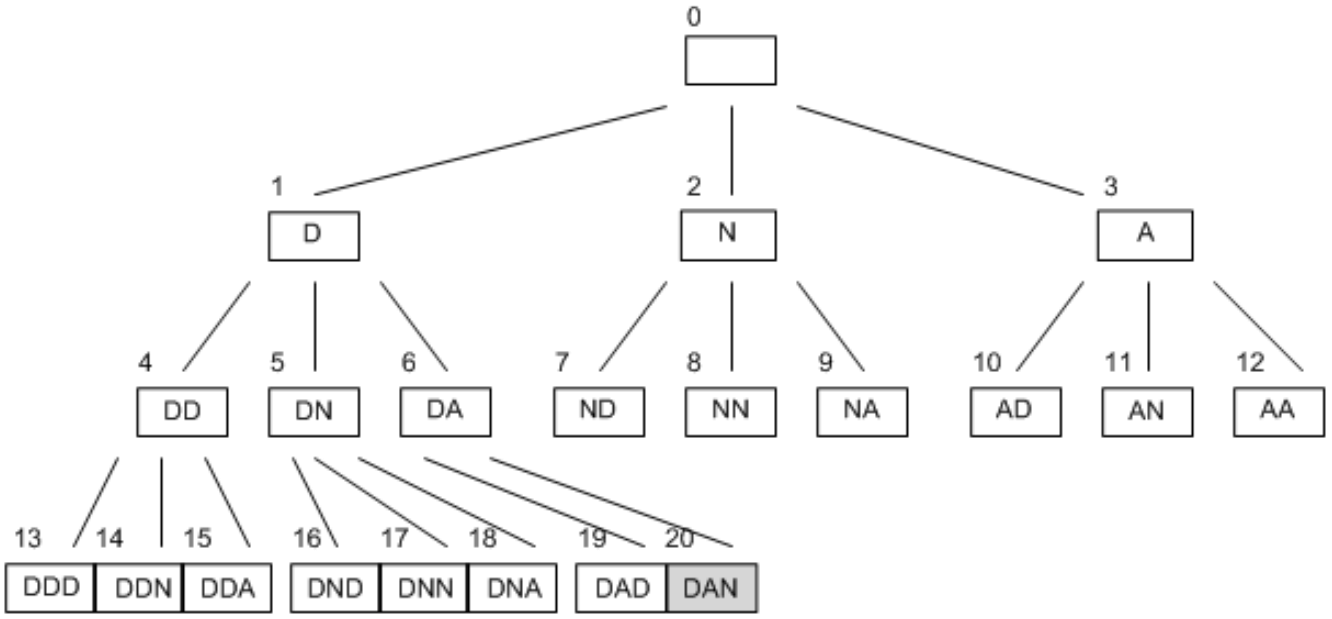
- Breadth-first search
- Depth-first search
- Iterative deepening search
- Bidirectional search
- Uniform-cost search
  - Also known as **blind search**

Breadth-First Search

- Every time a new state is reached
  - New states put on the **bottom** of the agenda
- When state “NA” is reached
  - New states “NAD”, “NAN”, “NAA” added to bottom
  - These get explored later (possibly much later)

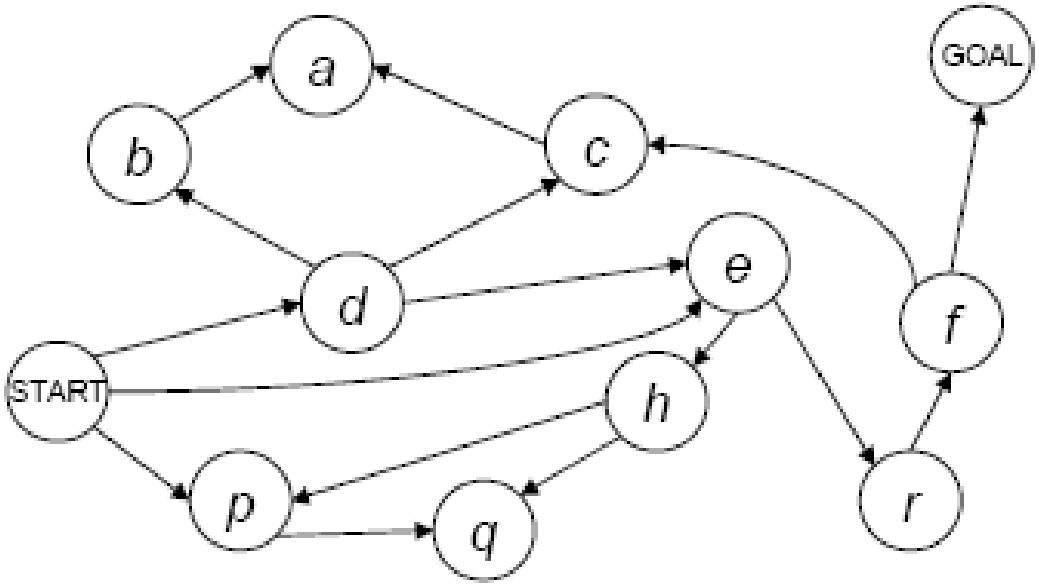
- Graph analogy
  - Each node of depth d is fully expanded before any node of depth d+1 is looked at

Breadth-First Search



- Branching rate
  - Average number of edges coming from a node (3 above)
- Uniform Search
  - Every node has same number of branches (as above)

Breadth First Search



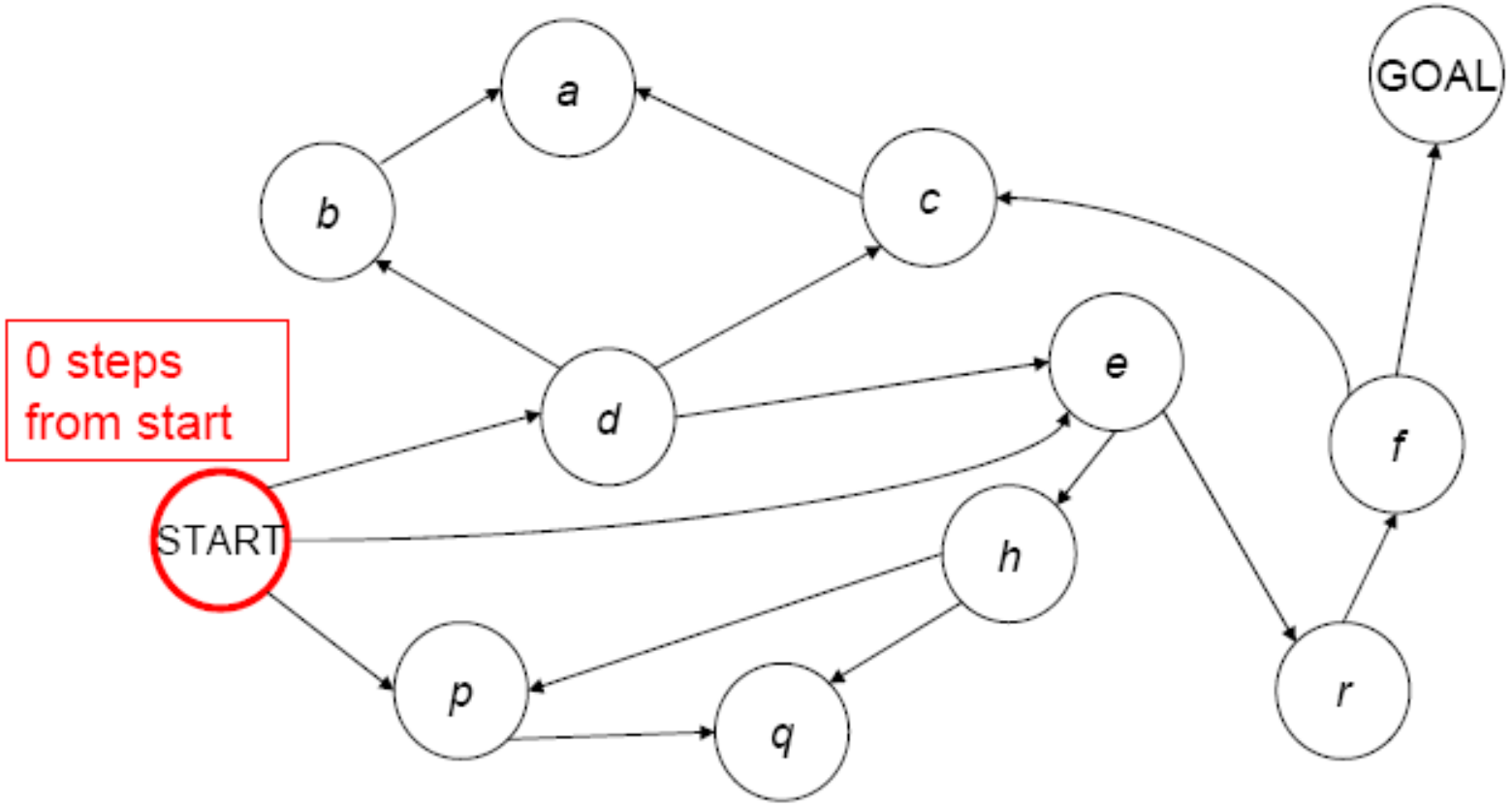
Label all states that are reachable from S in 1 step but aren't reachable in less than 1 step.

Then label all states that are reachable from S in 2 steps but aren't reachable in less than 2 steps.

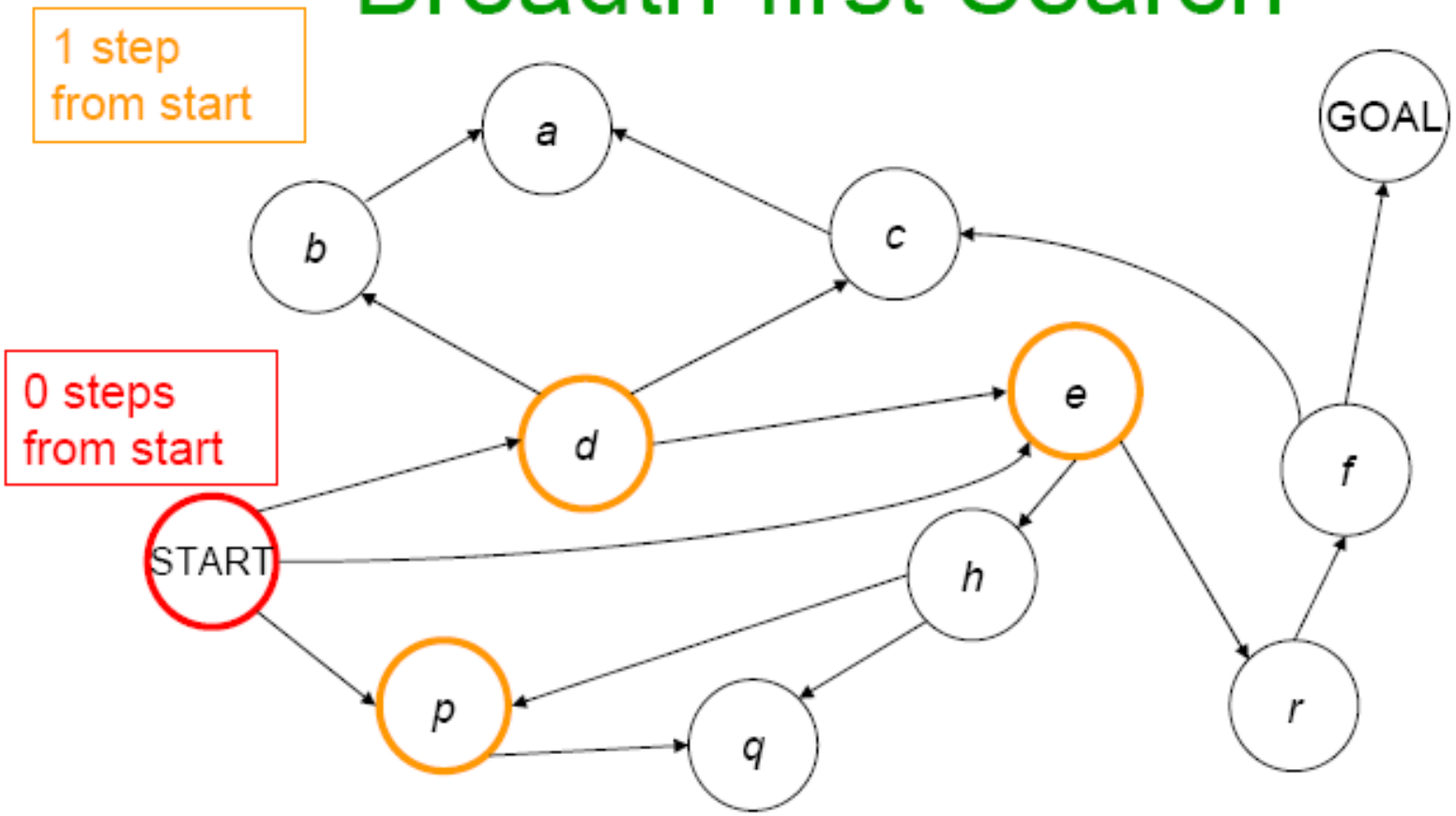
Then label all states that are reachable from S in 3 steps but aren't reachable in less than 3 steps.

Etc… until Goal state reached.

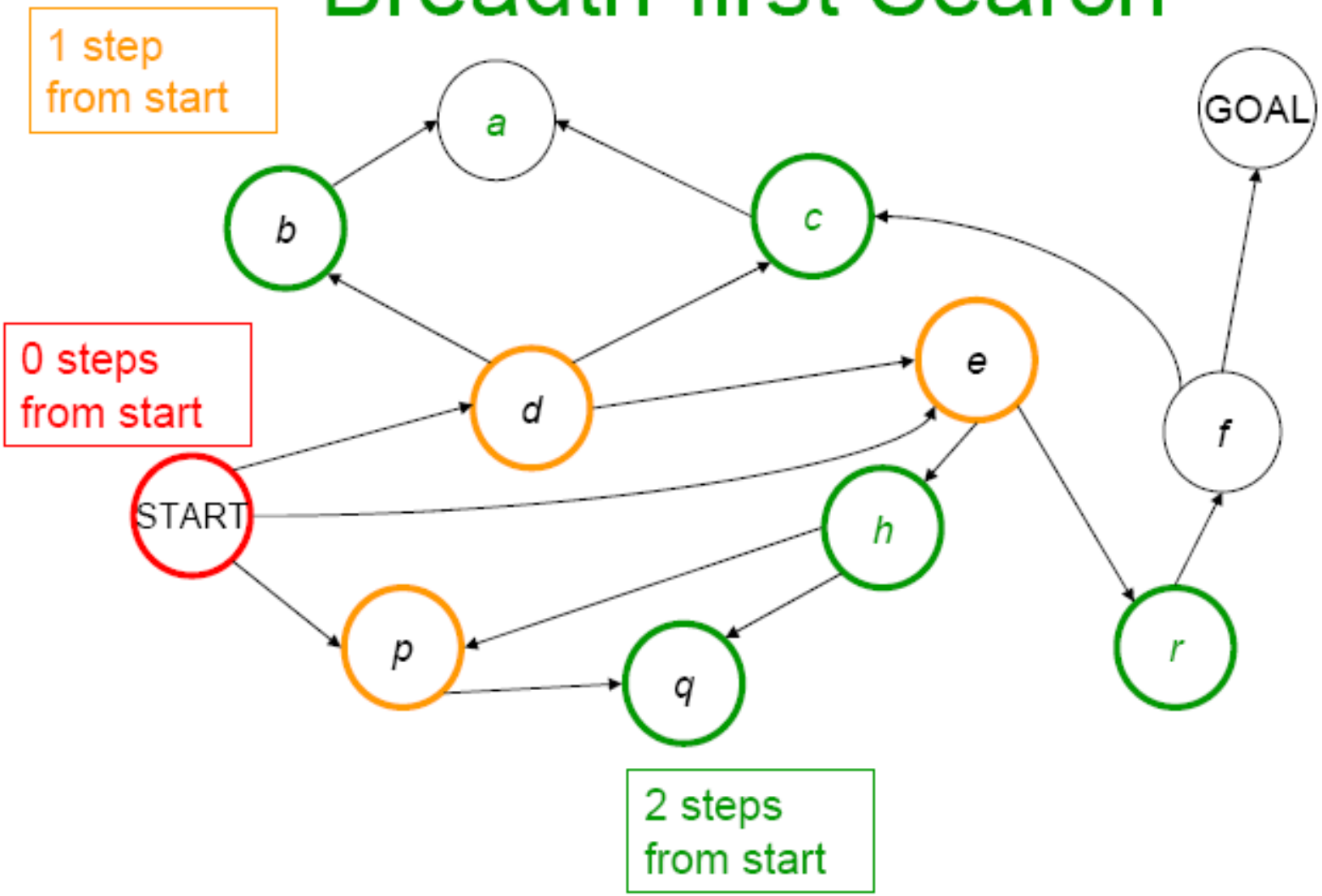
Breadth-first Search



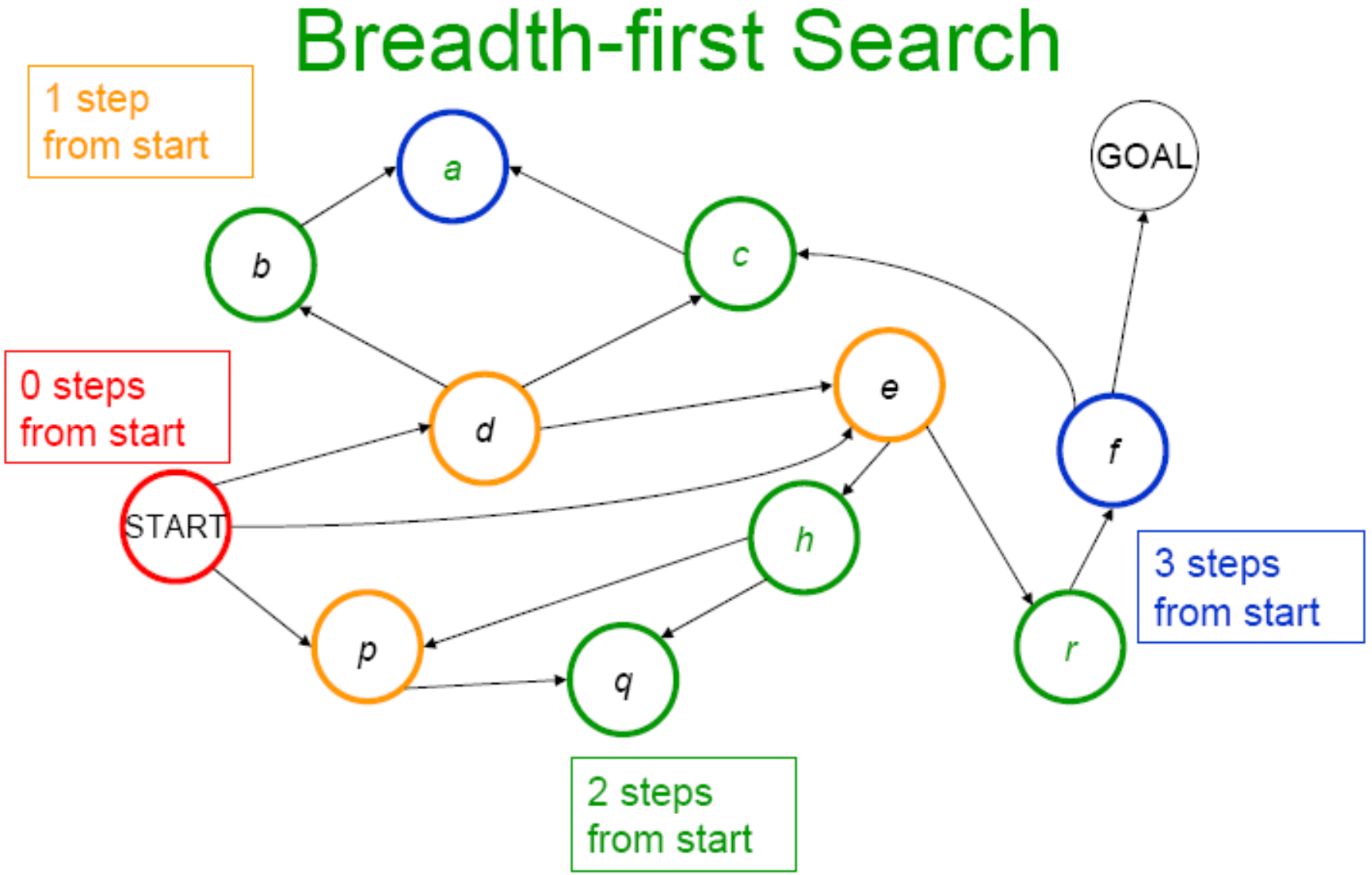
Breadth-first Search



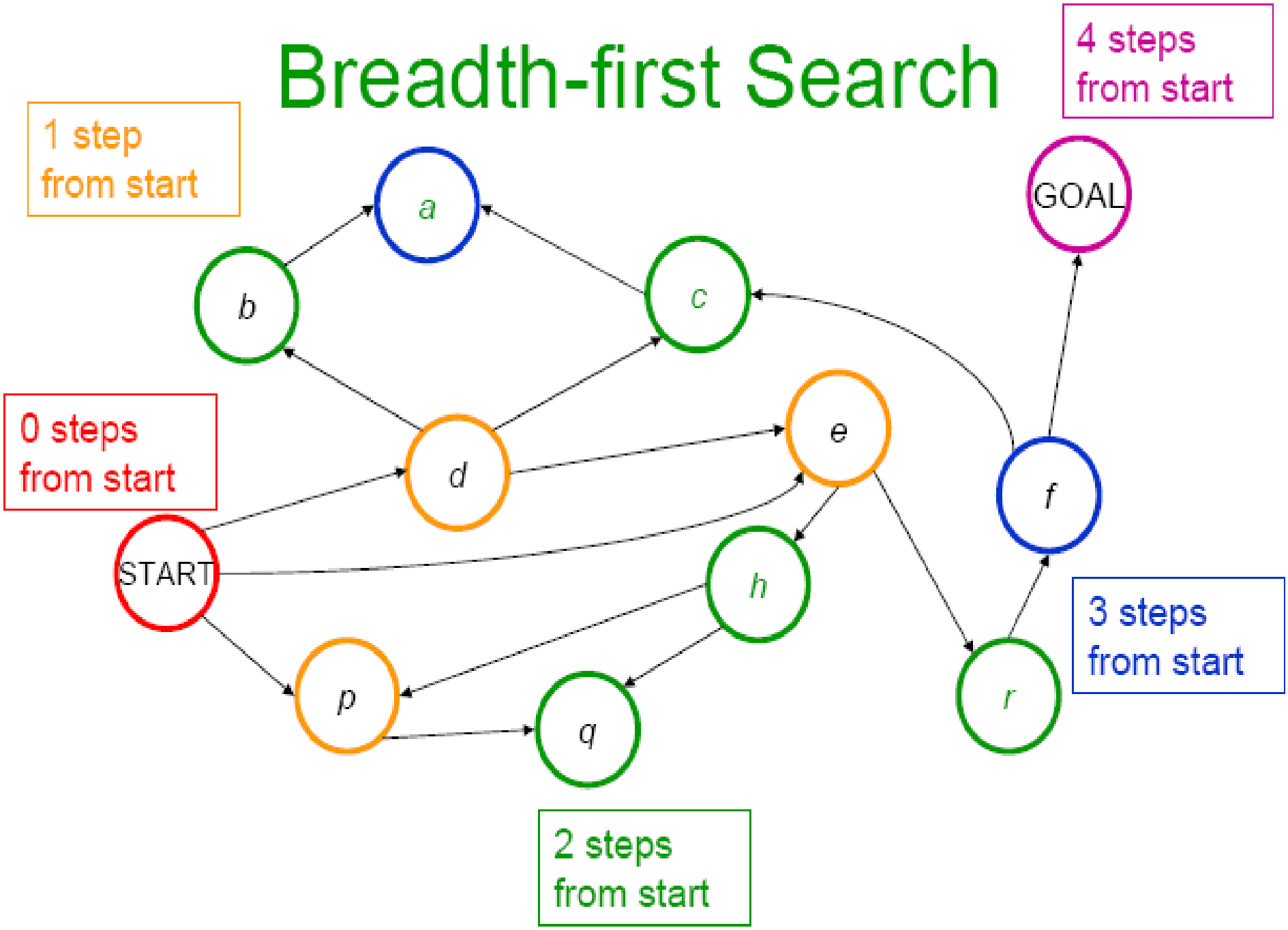
Breadth-first Search



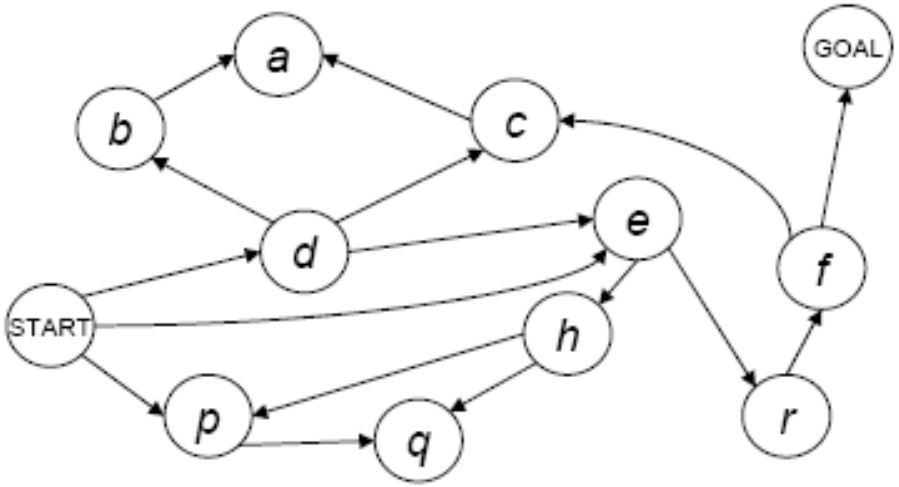
Breadth-first Search



Breadth-first Search



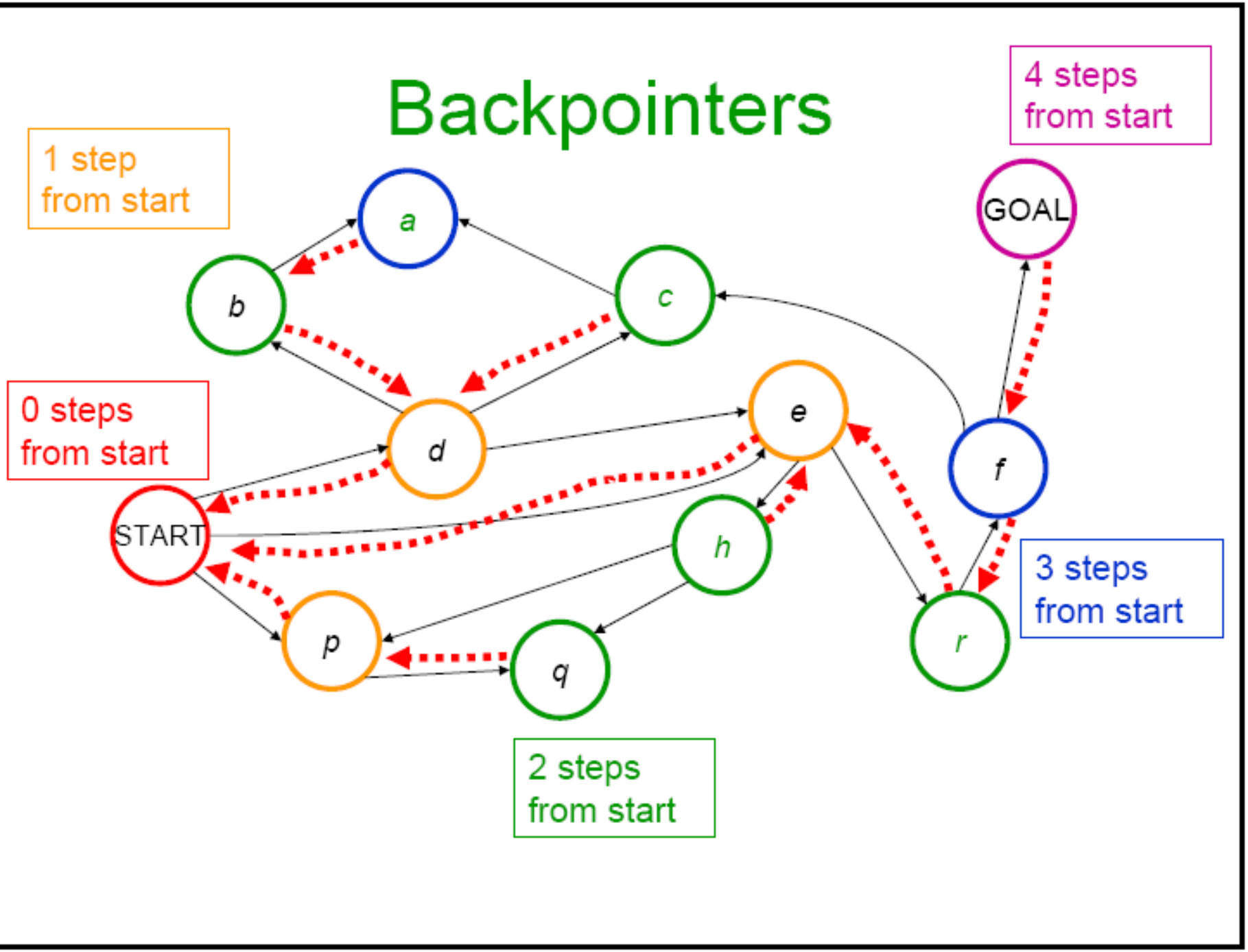
Remember the path!



Also, when you label a state, record the predecessor state. This record is called a *backpointer*. The history of predecessors is used to generate the solution path, once you've found the goal:

"I've got to the goal. I see I was at *f* before this. And I was at *r* before I was at *f*. And I was…

…. so solution path is S → *e* → *r* → *f* → G"


Backpointers
1 step from start
4 steps from start
0 steps from start
3 steps from start
2 steps from start
a
b
c
d
e
f
GOAL
START
h
p
q
r

Starting Breadth First Search

For any state *s* that we've labeled, we'll remember:

- *previous*(*s*) as the previous state on a shortest path from START state to *s*.

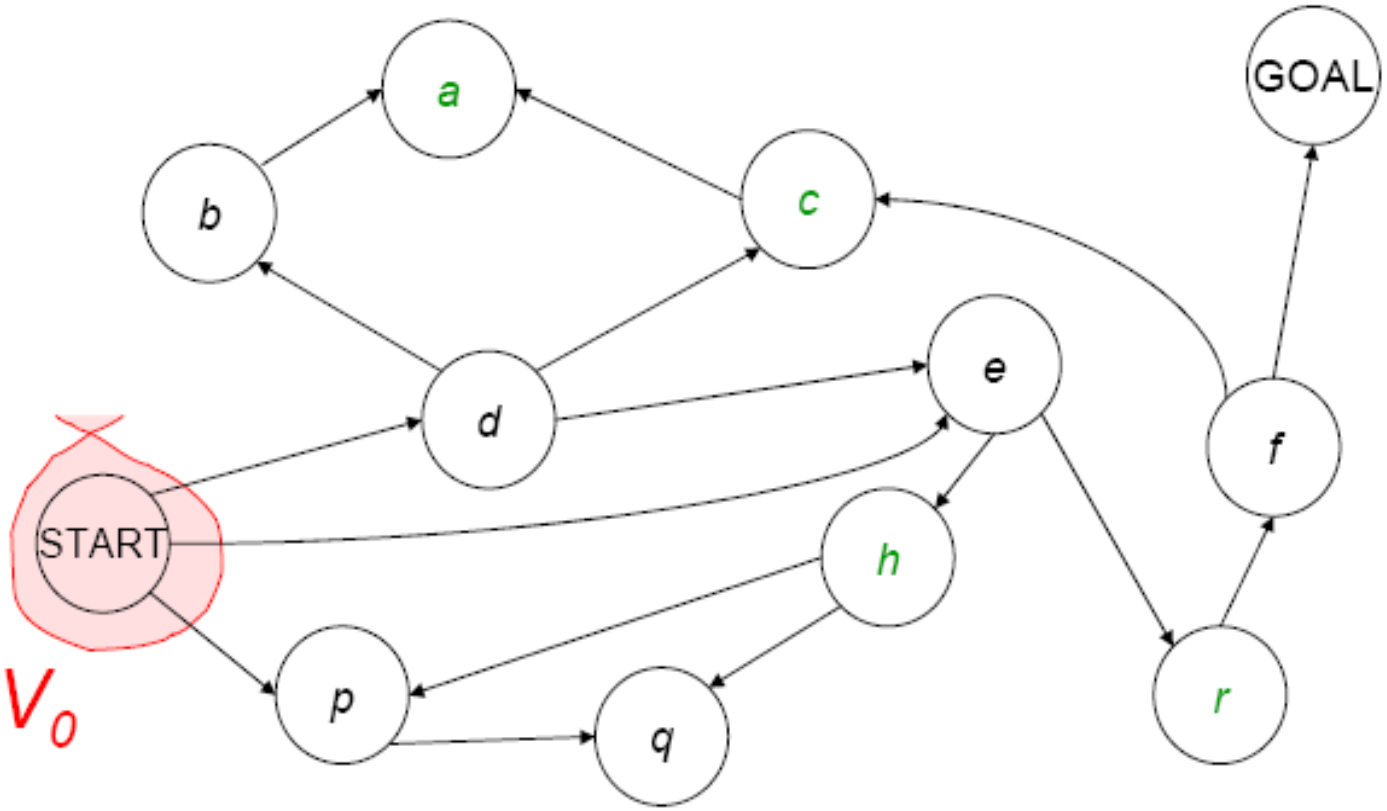
On the *k*th iteration of the algorithm we'll begin with V_k defined as the set of those states for which the shortest path from the start costs exactly *k steps*

Then, during that iteration, we'll compute V_{k+1}, defined as the set of those states for which the shortest path from the start costs exactly *k+1 steps*

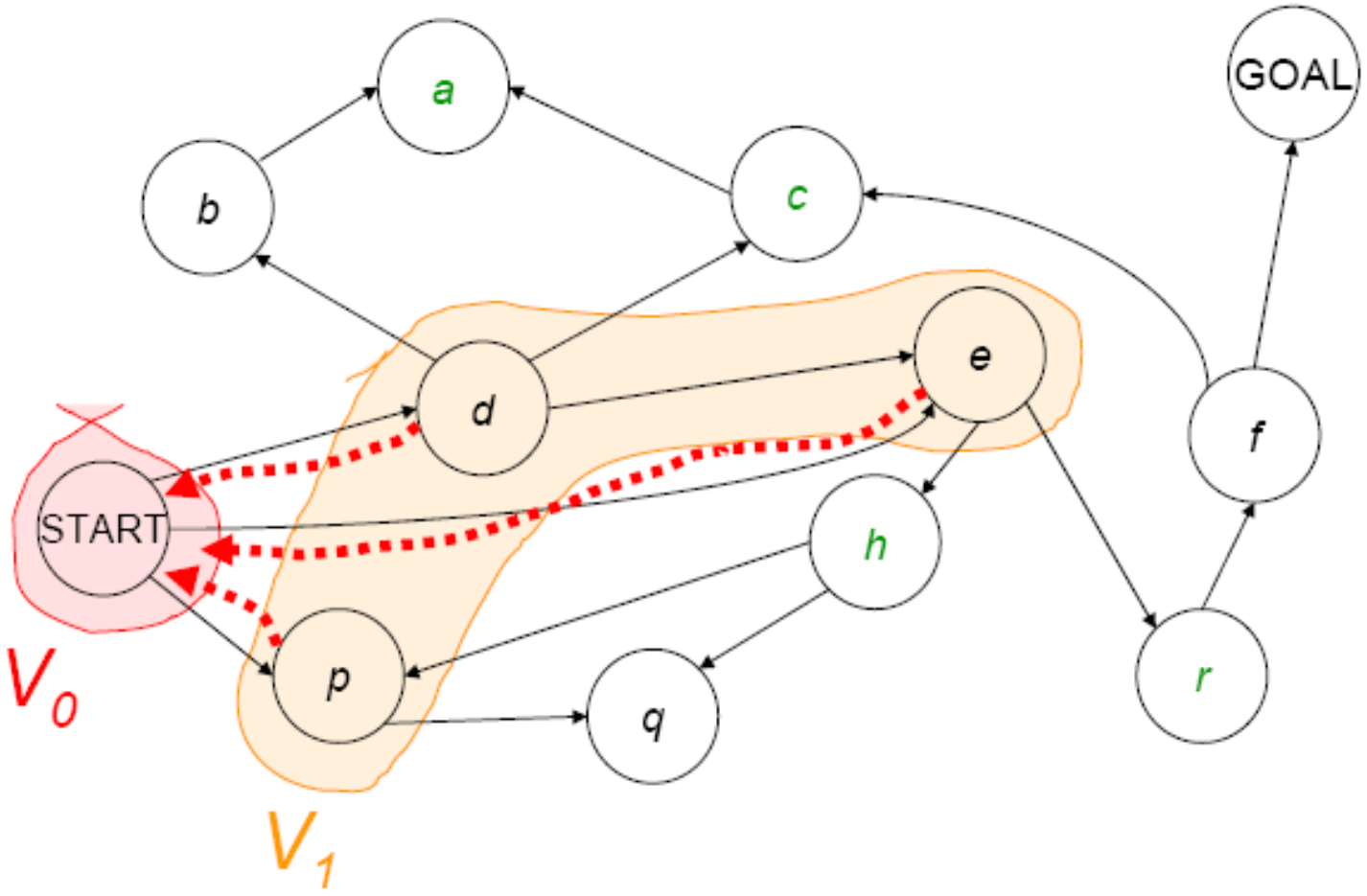
We begin with $k = 0$, V_0 = {START} and we'll define, *previous*(*START*) = *NULL*

Then we'll add in things one step from the START into V_1. And we'll keep going.

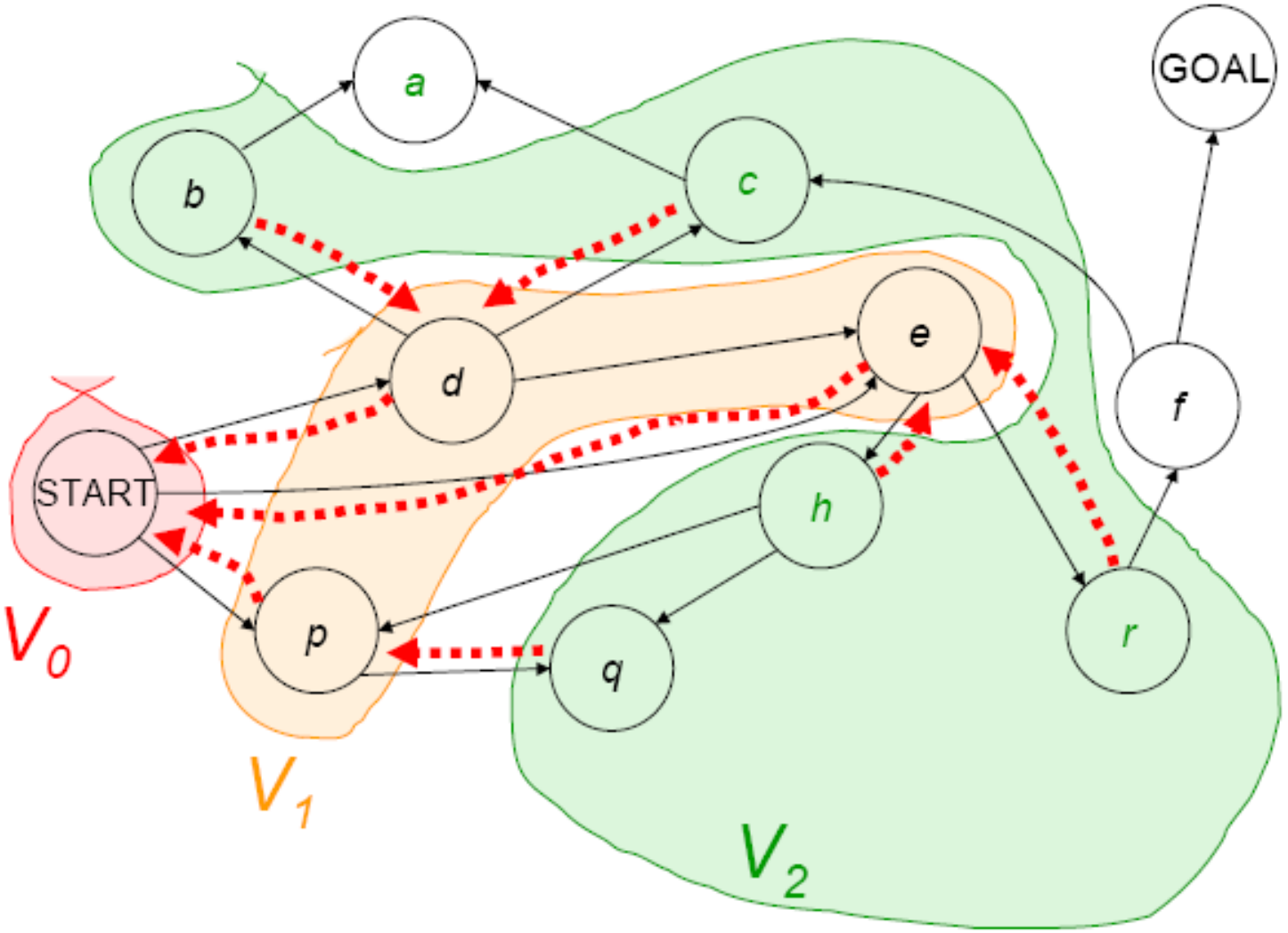
BFS

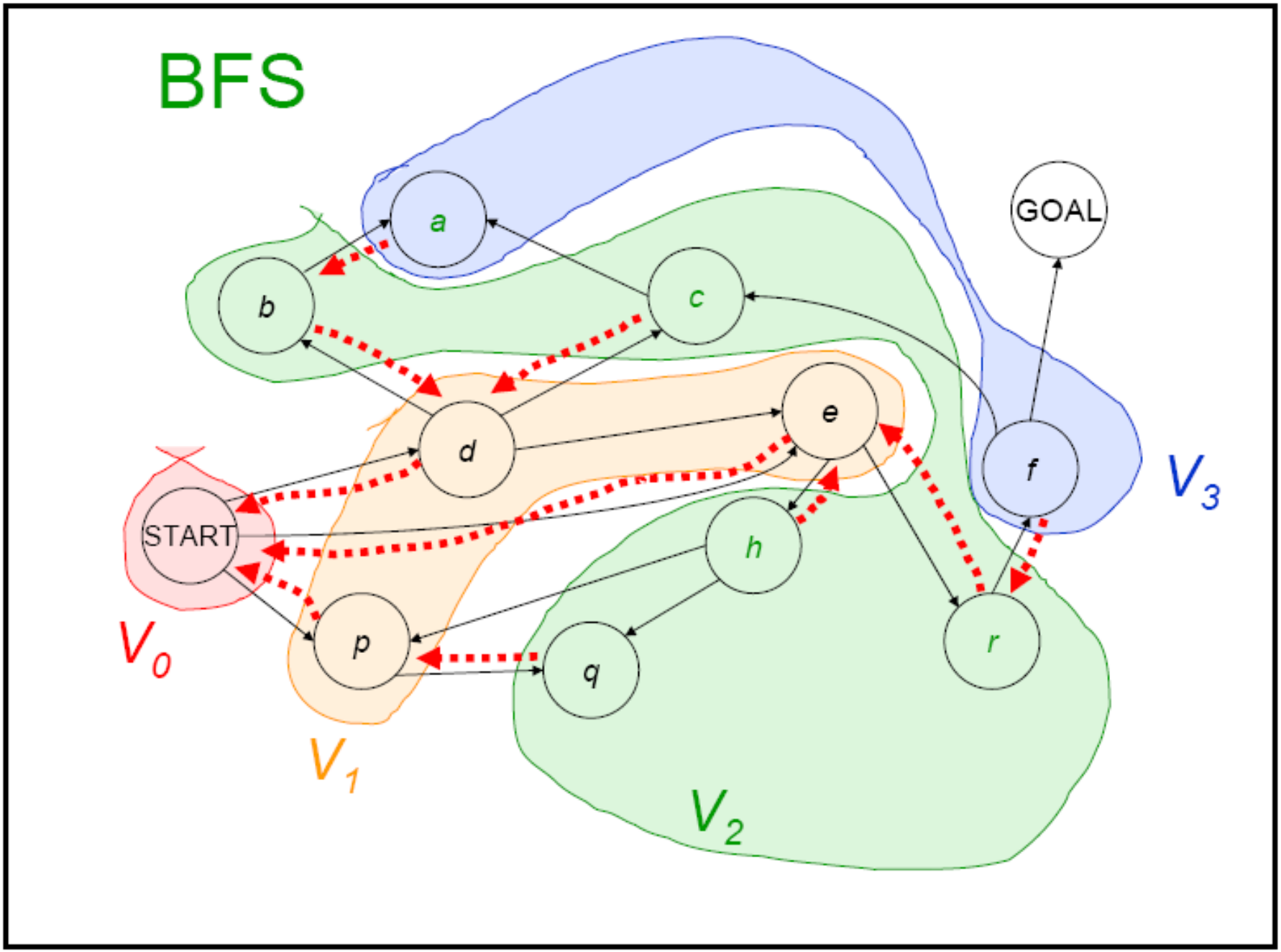


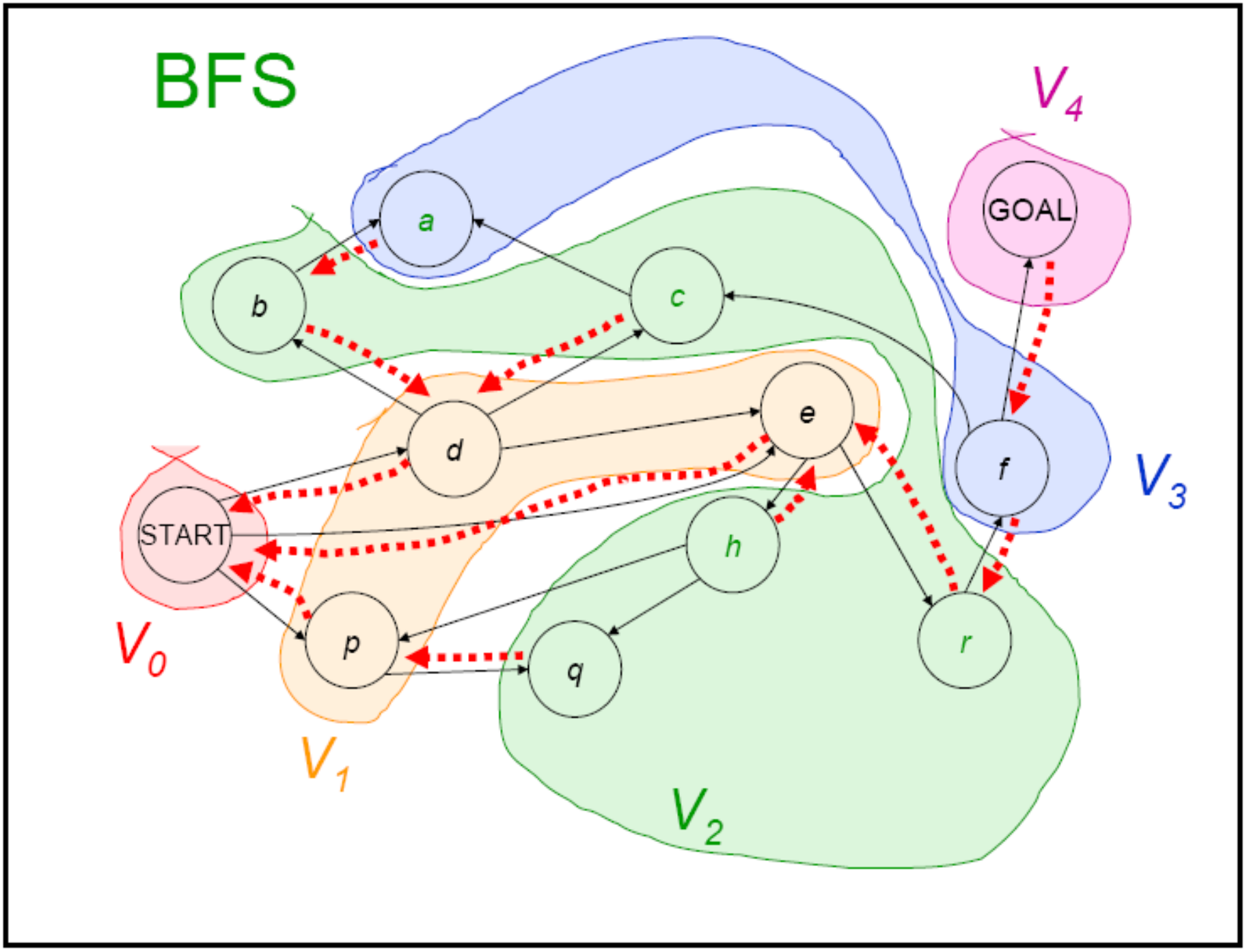
BFS



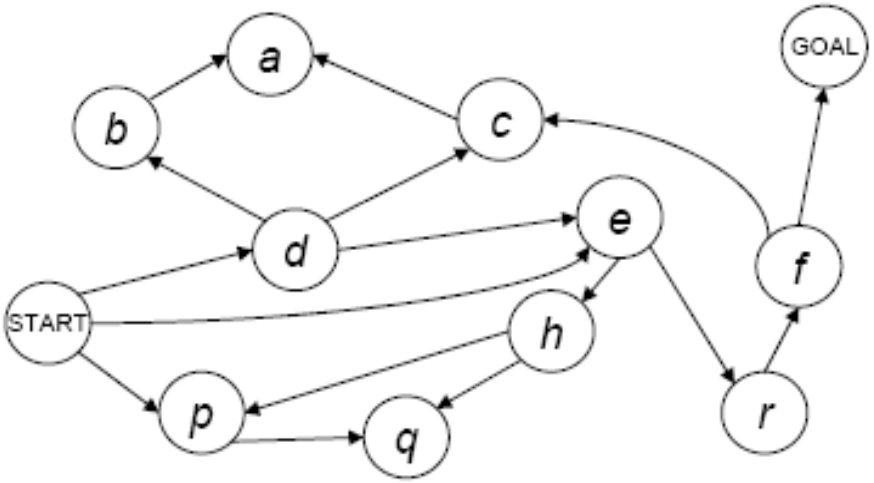
BFS




BFS
a
b
c
GOAL
d
e
f
START
h
p
q
r
V_0
V_1
V_2
V_3


BFS
V_4
GOAL
a
b
c
d
e
f
V_3
START
h
p
q
r
V_0
V_1
V_2

Another way: Work back



Label all states that can reach G in 1 step but can't reach it in less than 1 step.

Label all states that can reach G in 2 steps but can't reach it in less than 2 steps.

Etc. … until start is reached.

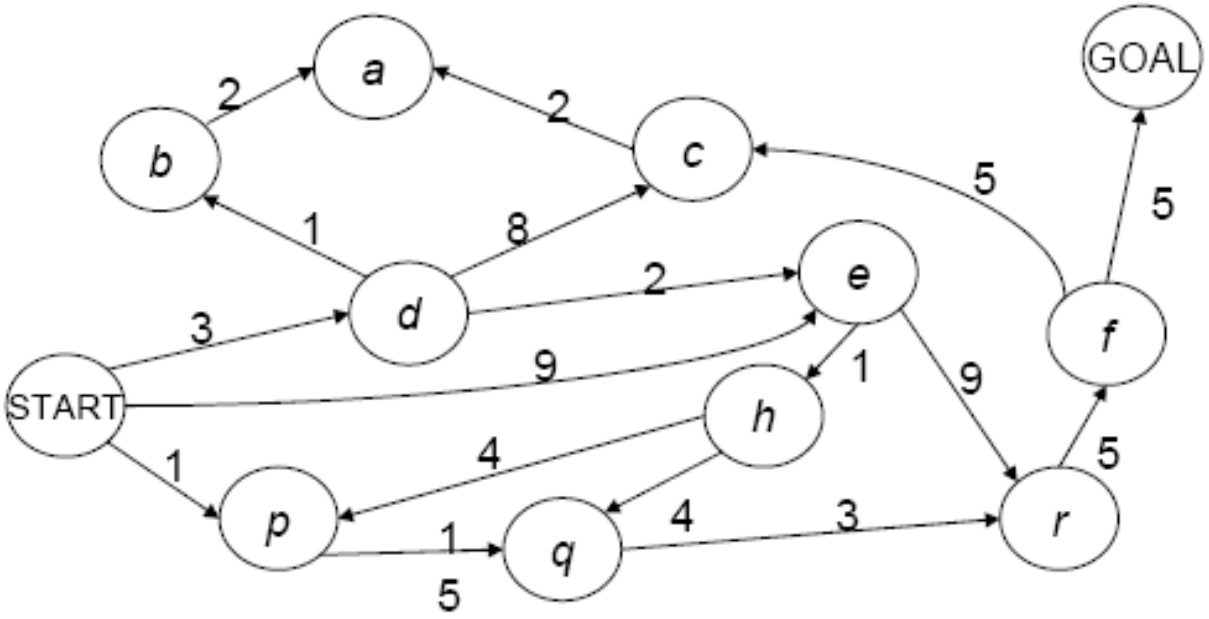
"number of steps to goal" labels determine the shortest path. Don't need extra bookkeeping info.

Breadth First Details

- It is fine for there to be more than one goal state.
- It is fine for there to be more than one start state.
- This algorithm works forwards from the start. Any algorithm which works forwards from the start is said to be *forward chaining*.
- You can also work backwards from the goal.

- Any algorithm which works backwards from the goal is said to be *backward chaining*.
- Backward versus forward. Which is better?

Costs on transitions



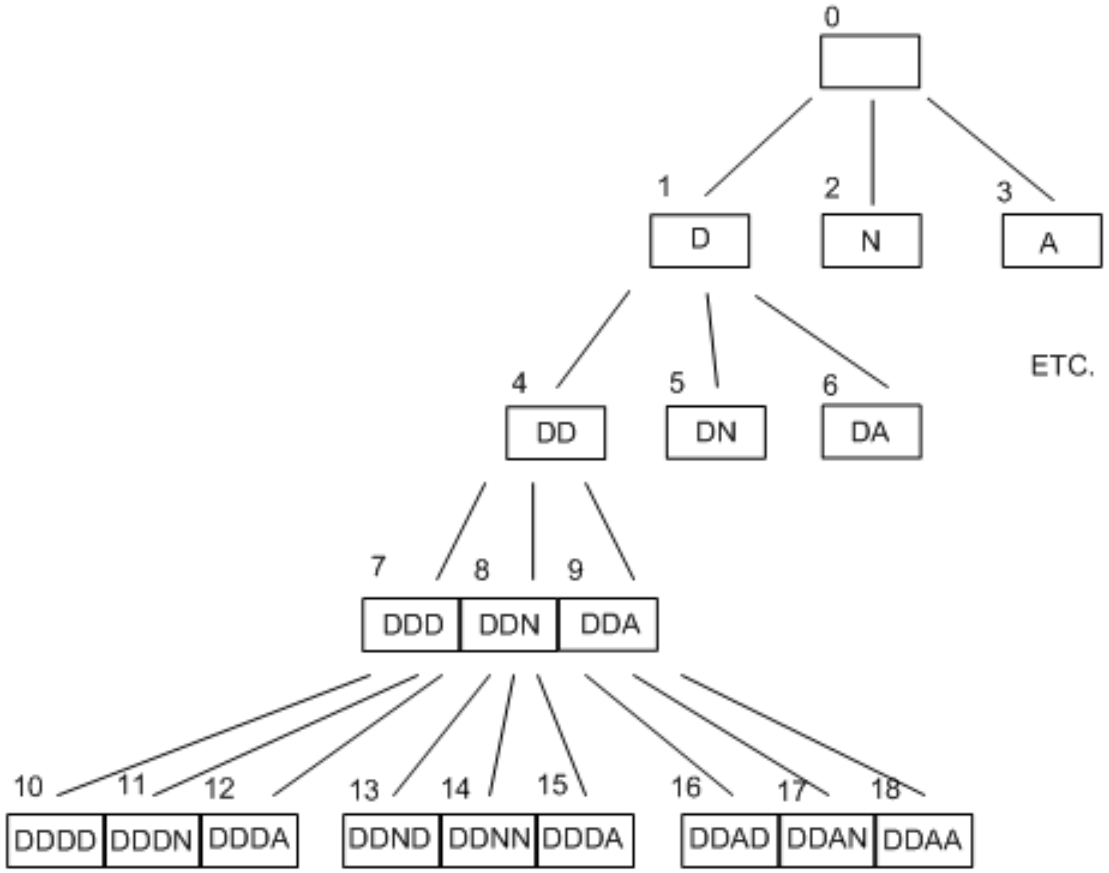
Notice that BFS finds the shortest path in terms of number of transitions. It does not find the least-cost path.

Depth-First Search

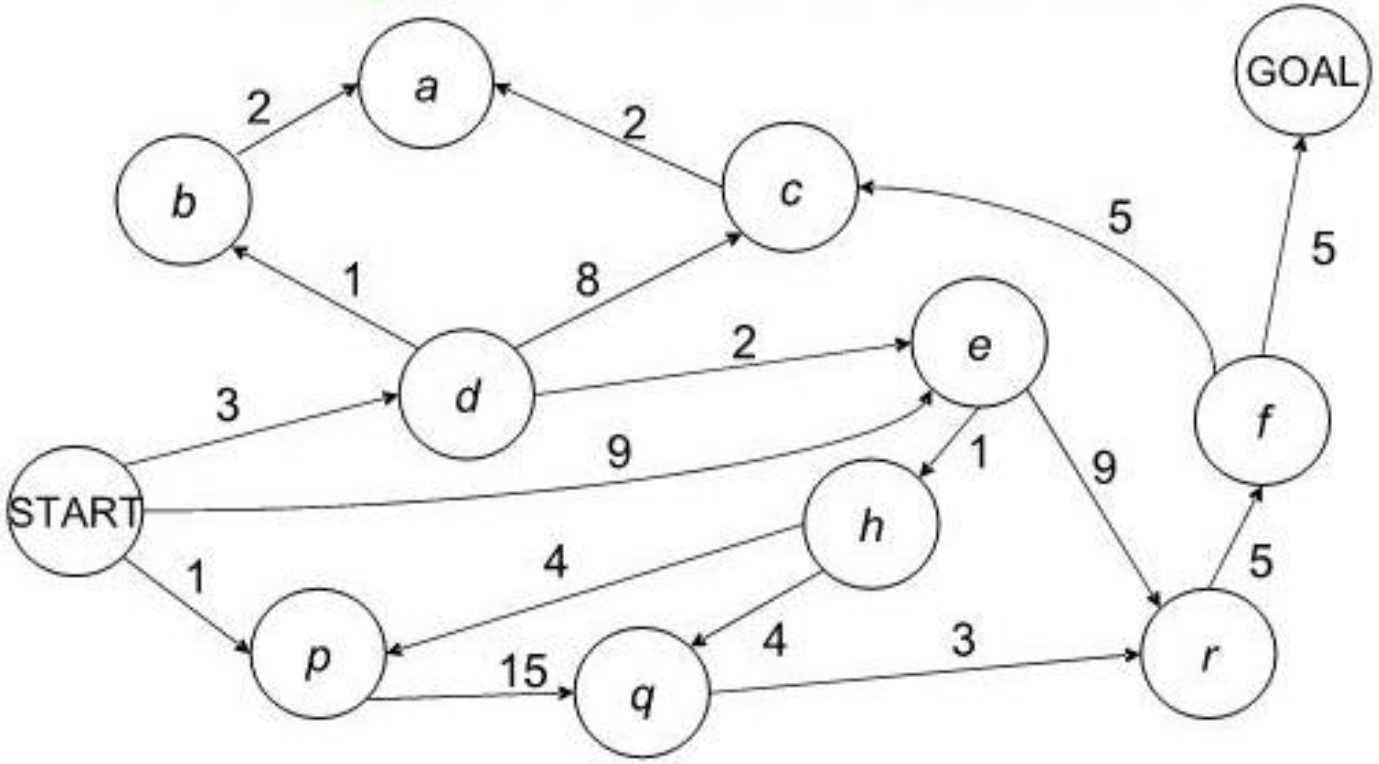
- Same as breadth-first search
  - But new states are put at the **top** of agenda
- Graph analogy
  - Expand deepest and leftmost node next

- But search can go on indefinitely down one path
  - D, DD, DDD, DDDD, DDDDD, …
- One solution to impose a **depth limit** on the search
  - Sometimes the limit is not required
    - Branches end naturally (i.e. cannot be expanded)

Depth-First Search (Depth Limit 4)



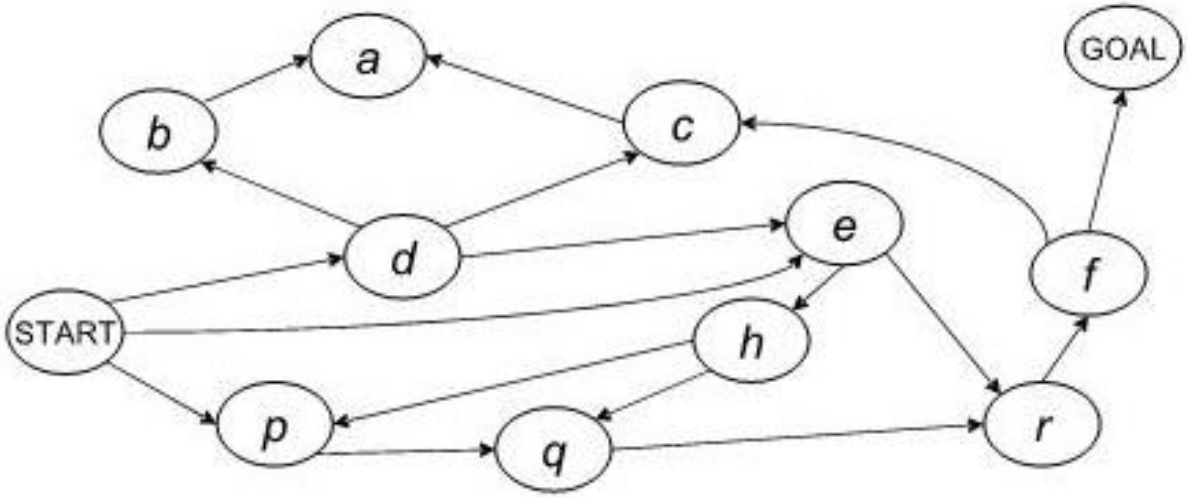
Depth First Search



An alternative to BFS. Always expand from the most-recently-expanded node, if it has any untried successors. Else backup to the previous node on the current path.

DFS in action

START
START *d*
START *d b*
START *d b a*
START *d c*
START *d c a*
START *d e*
START *d e r*
START *d e r f*
START *d e r f c*
START *d e r f c a*
START *d e r f* GOAL



Depth- v. Breadth-First Search

- Suppose branching rate b
- Breadth-first
  - Complete (guaranteed to find solution)
  - Requires a lot of memory
    - At depth d needs to remember up to b^{d-1} states
- Depth-first
  - Not complete because of indefinite paths or depth limit
  - But is memory efficient
    - Only needs to remember up to b*d states

Iterative Deepening Search

- Idea: do repeated depth first searches
  - Increasing the depth limit by one every time
  - DFS to depth 1, DFS to depth 2, etc.
  - Completely re-do the previous search each time
- Most DFS effort is in expanding last line of the tree
  - e.g. to depth five, branching rate of 10
    - DFS: 111,111 states, IDS: 123,456 states
    - Repetition of only 11%
- Combines best of BFS and DFS
  - Complete and memory efficient
  - But slower than either

Iterative Deepening

Iterative deepening is a simple algorithm which uses DFS as a subroutine:

1. Do a DFS which only searches for paths of length 1 or less. (DFS gives up any path of length 2)
2. If "1" failed, do a DFS which only searches paths of length 2 or less.
3. If "2" failed, do a DFS which only searches paths of length 3 or less.

....and so on until success

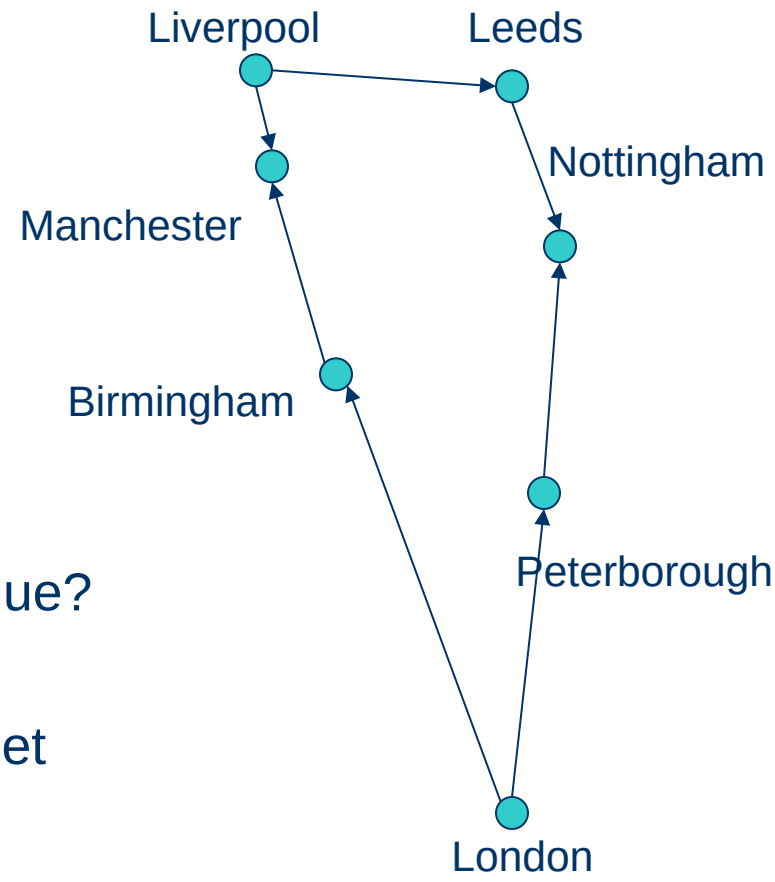
Time to run is:

$$O(b^1 + b^2 + b^3 + b^4 \ldots + b^L) = O(b^L)$$

Bidirectional Search

- If you know the solution state
  - Work forwards and backwards
  - Look to meet in middle
- Only need to go to half depth

- Difficulties
  - Do you really know solution? Unique?
  - Must be able to reverse operators
  - Record all paths to check they meet
    - Memory intensive



Uniform-Cost Search

- Breadth-first search
  - Guaranteed to find the shortest path to a solution
  - Not necessarily the least costly path
- Uniform path cost search
  - Choose to expand node with the least path cost
- Guaranteed to find a solution with least cost
  - If we know that path cost increases with path length
- This method is optimal and complete
  - But can be very slow

Uniform-Cost Search

- A conceptually simple BFS approach when there are costs on transitions
- It uses priority queues

Priority Queue Refresher

A priority queue is a data structure in which you can insert and retrieve (thing, value) pairs with the following operations:

Init-PriQueue(PQ)	initializes the PQ to be empty.
Insert-PriQueue(PQ, thing, value)	inserts *(thing, value)* into the queue.
Pop-least(PQ)	returns the *(thing, value)* pair with the lowest value, and removes it from the queue.

Priority Queue Refresher

For more details, see Knuth or Sedgwick or basically any book with the word "algorithms" prominently appearing in the title.

A priority queue is a data structure in which you can insert and retrieve *(thing, value)* pairs with the following operations:

Init-PriQueue(PQ)	initializes the PQ to be empty.
Insert-PriQueue(PQ, thing, value)	inserts *(thing, value)* into the queue.
Pop-least(PQ)	returns the *(thing, value)* pair with the lowest value, and removes it from the queue.

Priority Queues can be implemented in such a way that the cost of the insert and pop operations are

O(log(number of things in priority queue))

Very cheap (though not absolutely, incredibly cheap!)

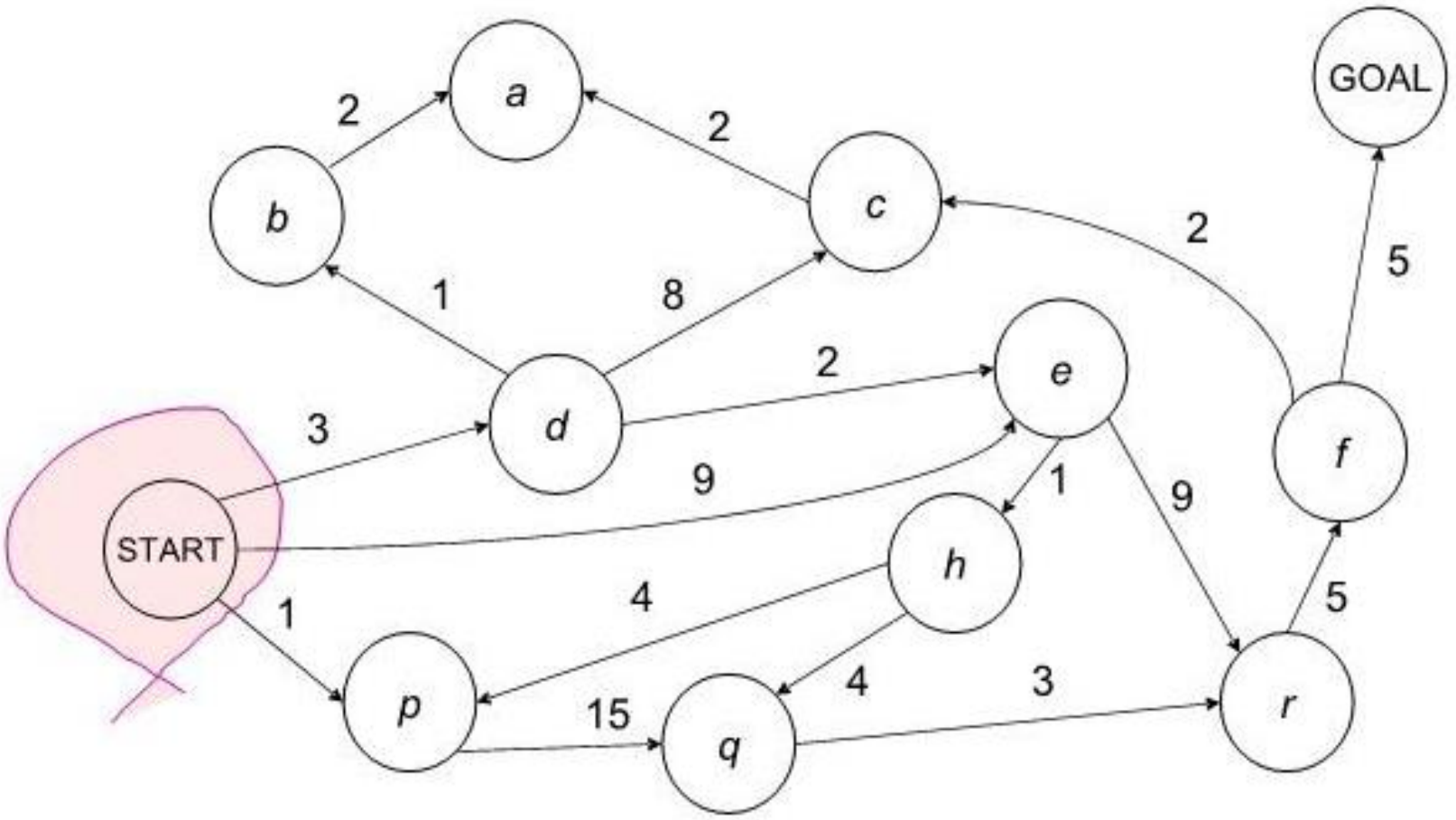
Uniform-Cost Search

- A conceptually simple BFS approach when there are costs on transitions
- It uses a priority queue

PQ = Set of states that have been expanded or are awaiting expansion

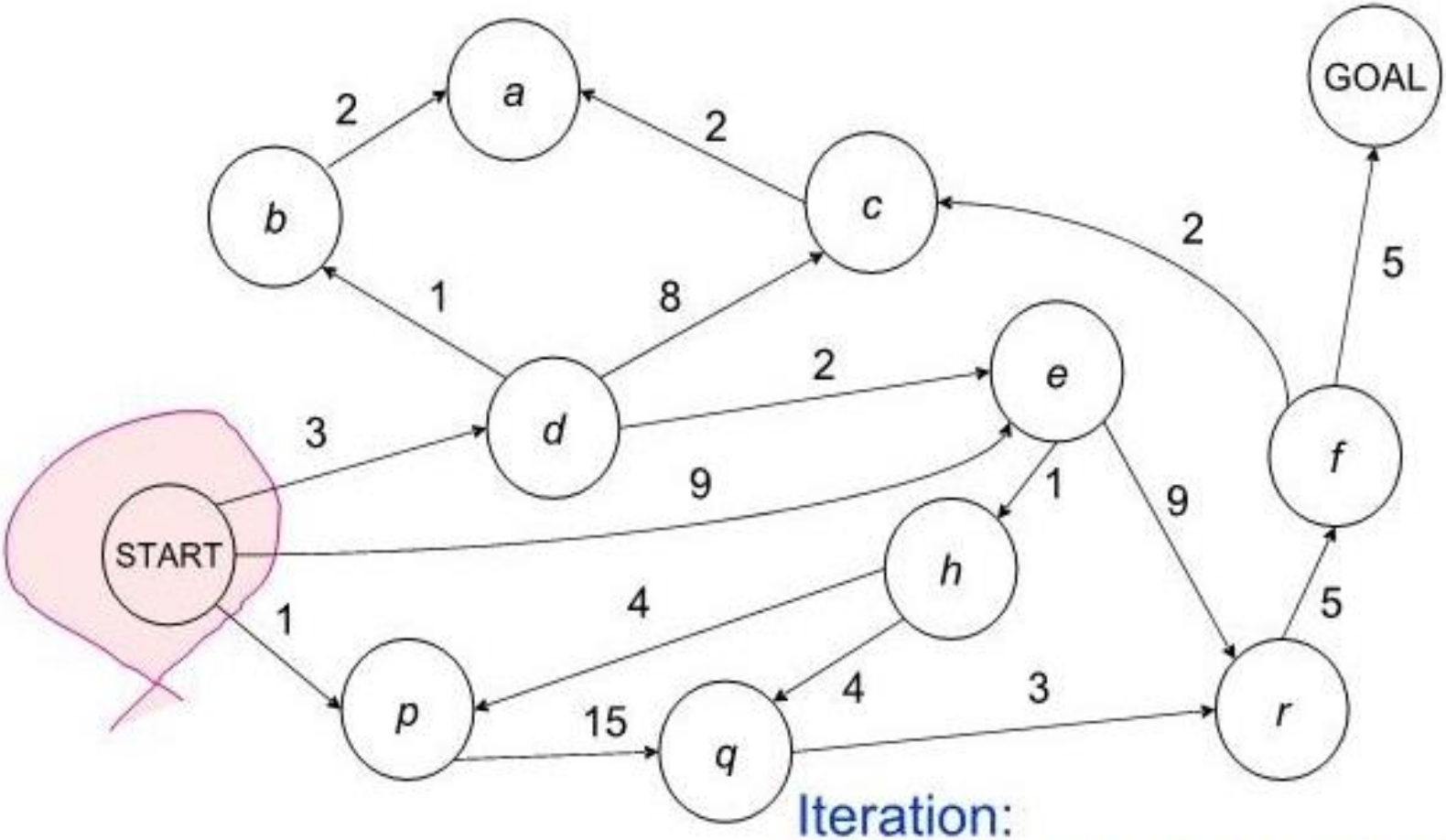
Priority of state s = $g(s)$ = cost of getting to s using path implied by backpointers.

Starting UCS



PQ = { (S,0) }

UCS Iterations

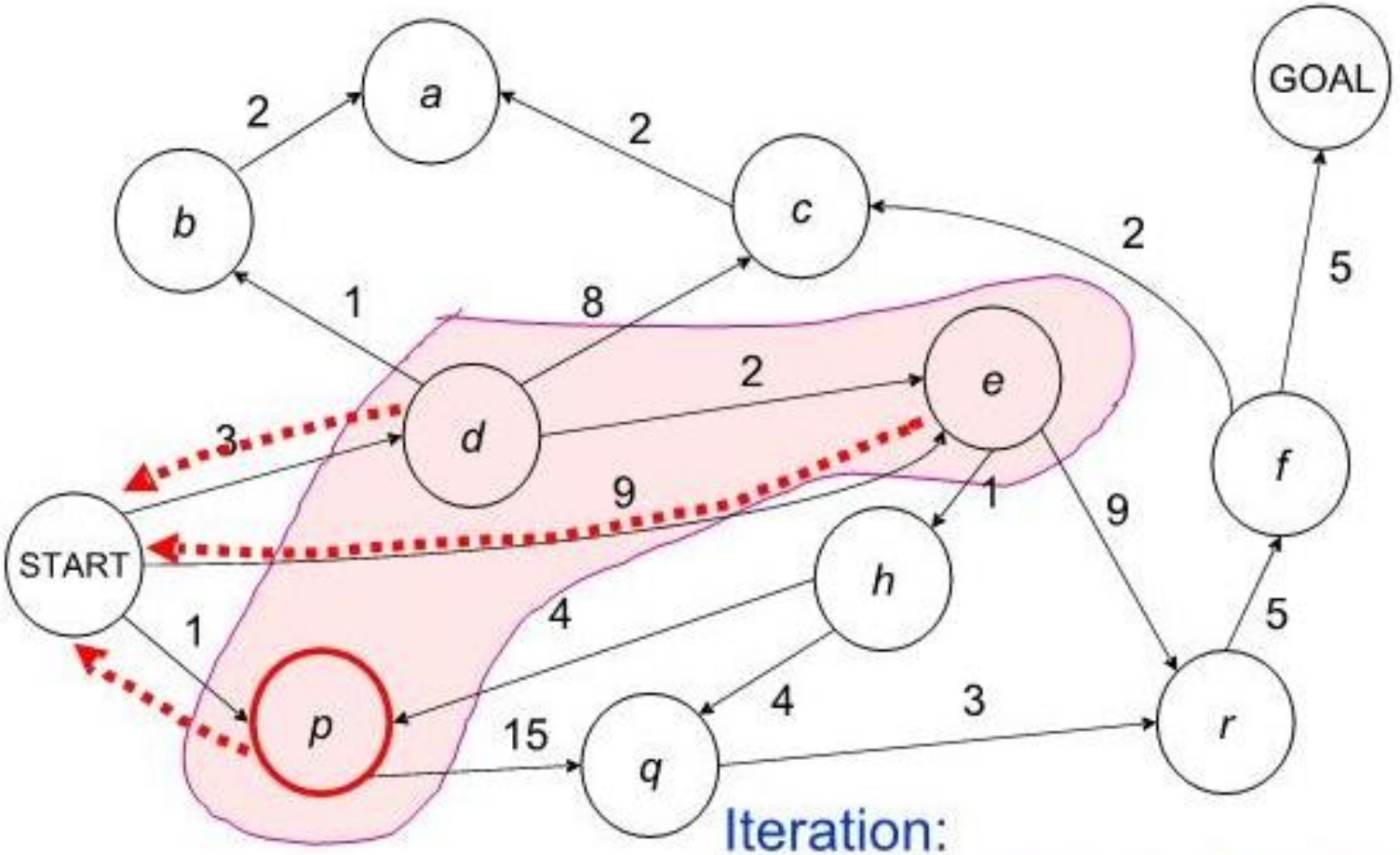


Iteration:

1. Pop least-cost state from PQ
2. Add successors

PQ = { (S,0) }

UCS Iterations

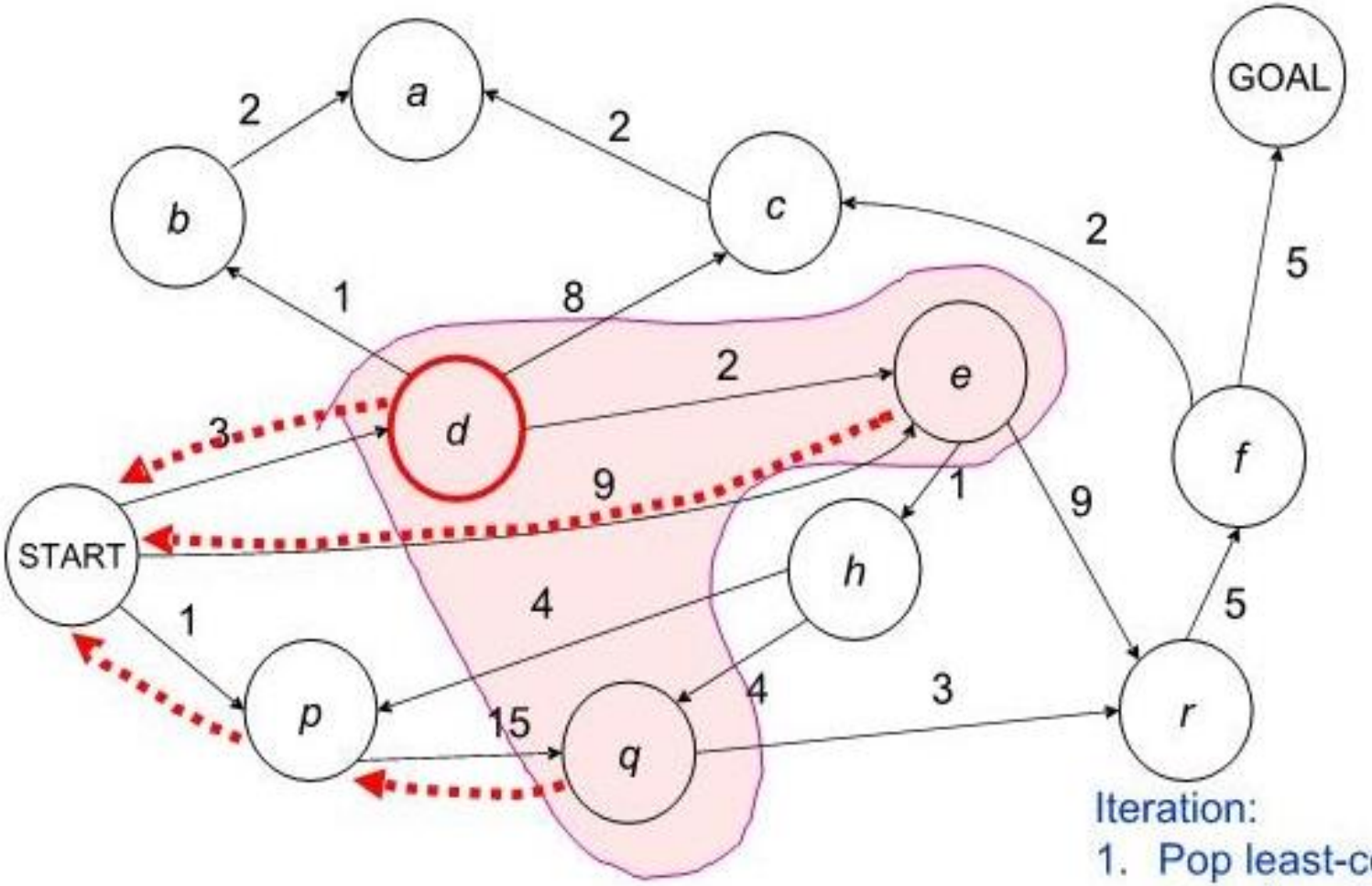


Iteration:

1. Pop least-cost state from PQ
2. Add successors

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

UCS Iterations

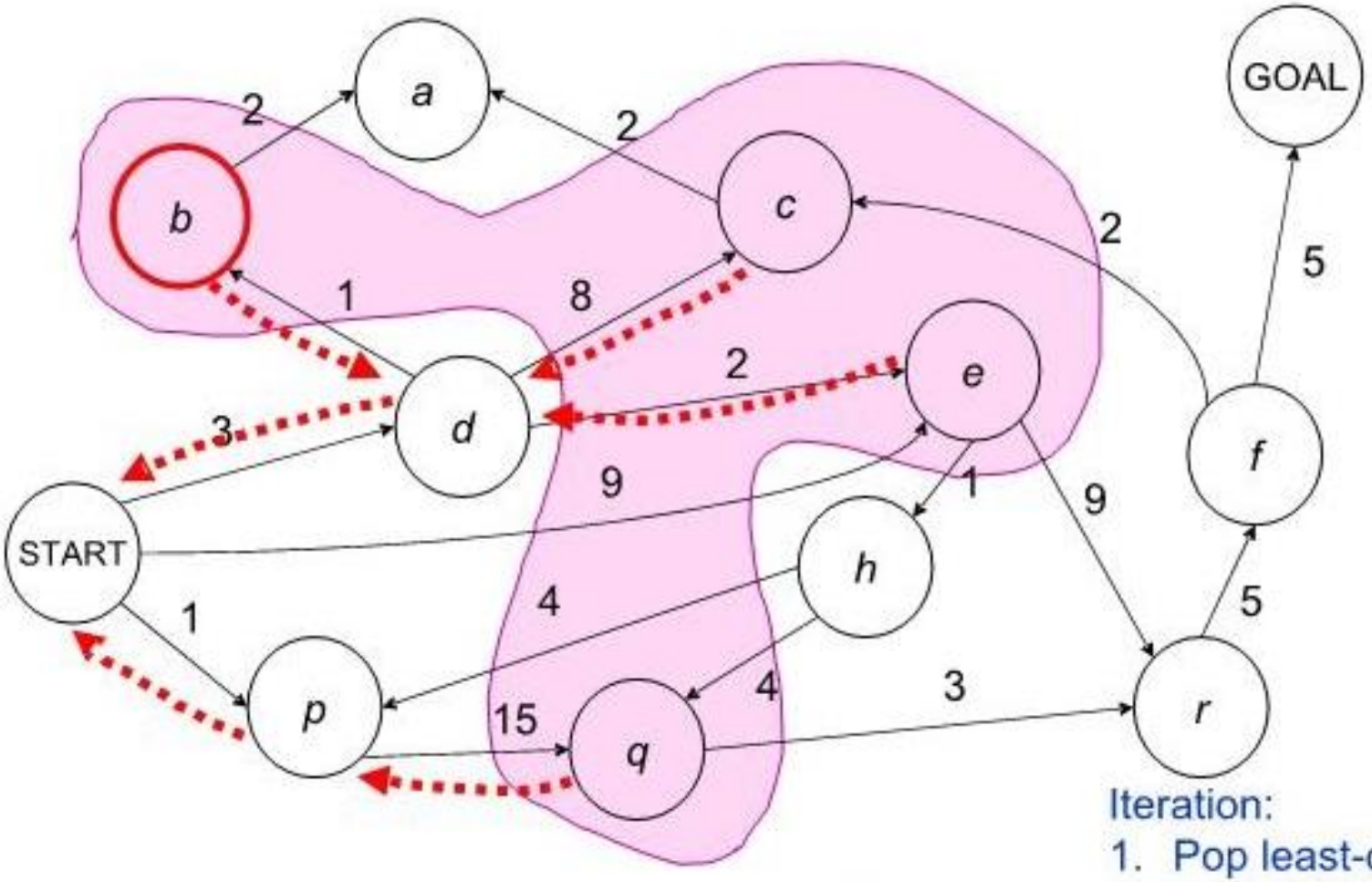


PQ = { (d,3) , (e,9) , (q,16) }

Iteration:

1. Pop least-cost state from PQ
2. Add successors

UCS Iterations

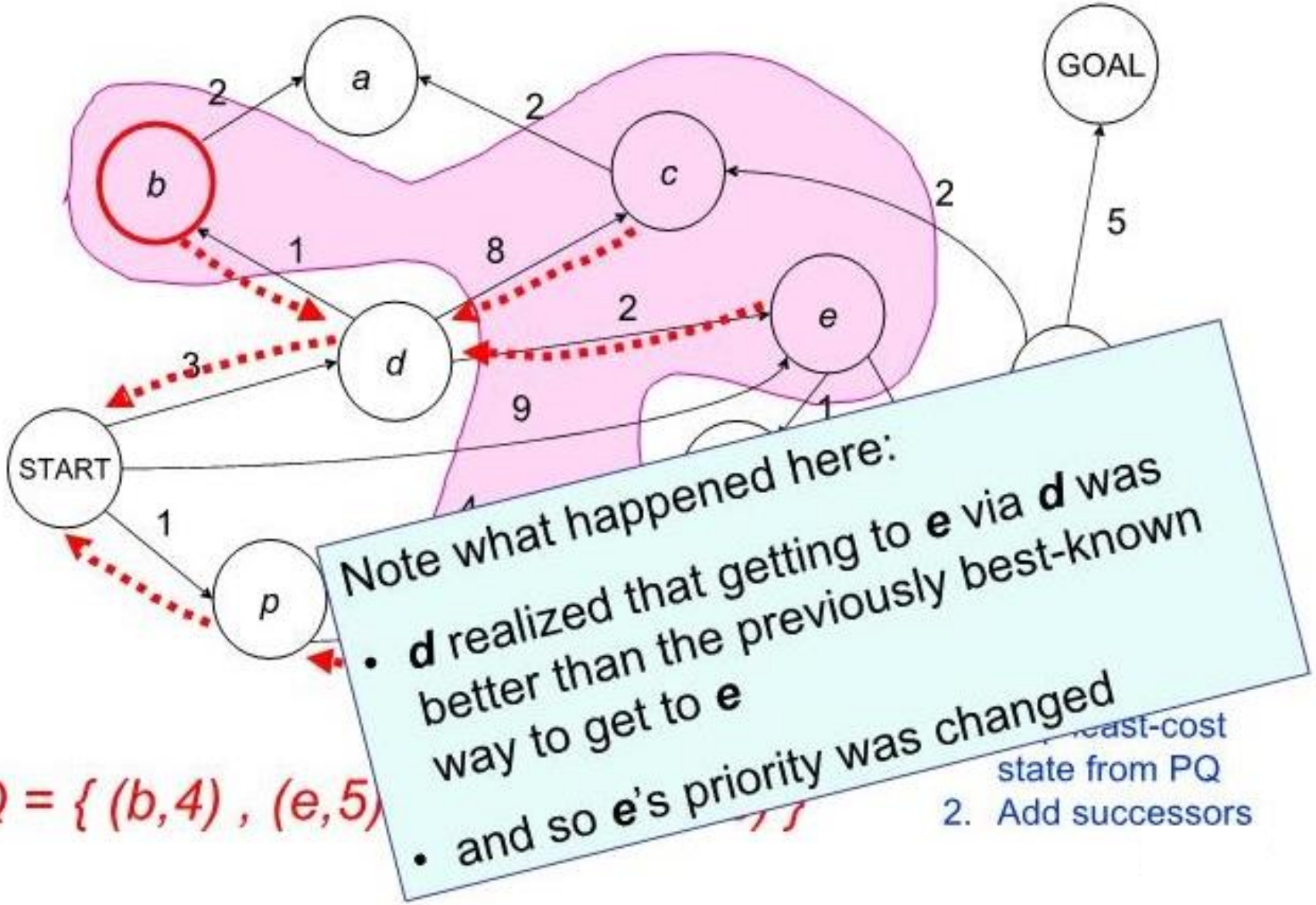


Iteration:

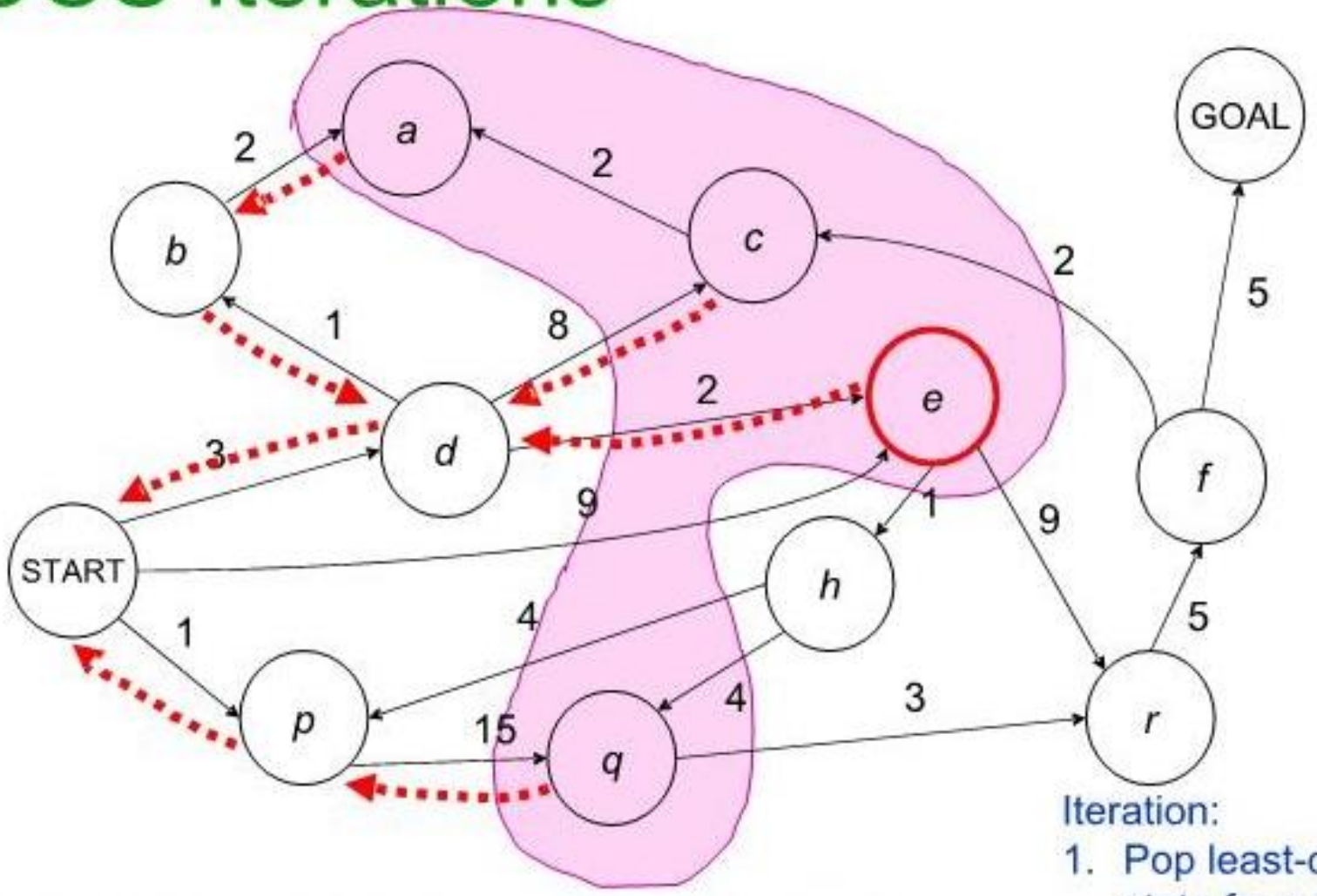
1. Pop least-cost state from PQ
2. Add successors

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

UCS Iterations



UCS Iterations

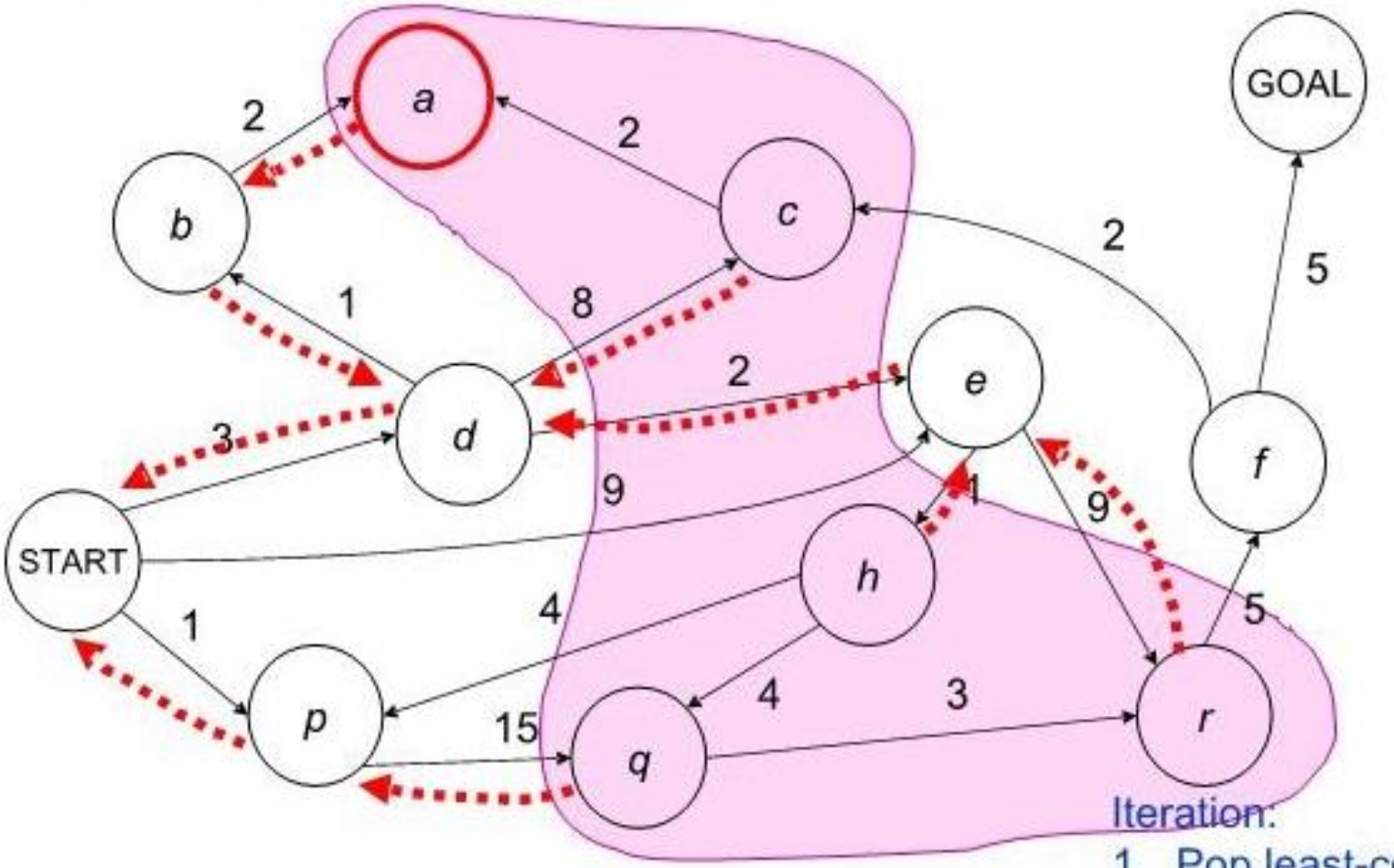


Iteration:

1. Pop least-cost state from PQ
2. Add successors

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

UCS Iterations

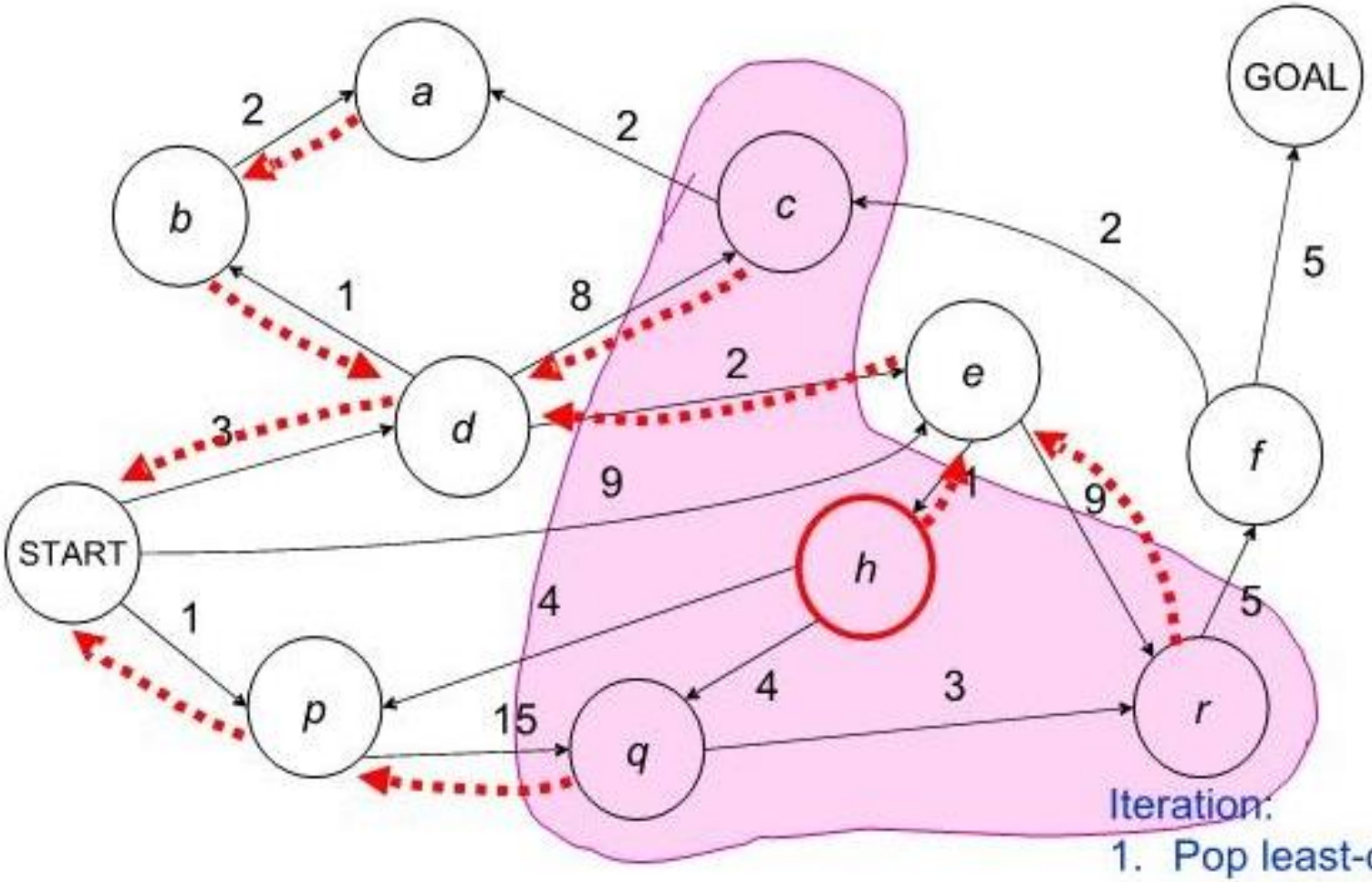


Iteration:

1. Pop least-cost state from PQ
2. Add successors

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

UCS Iterations

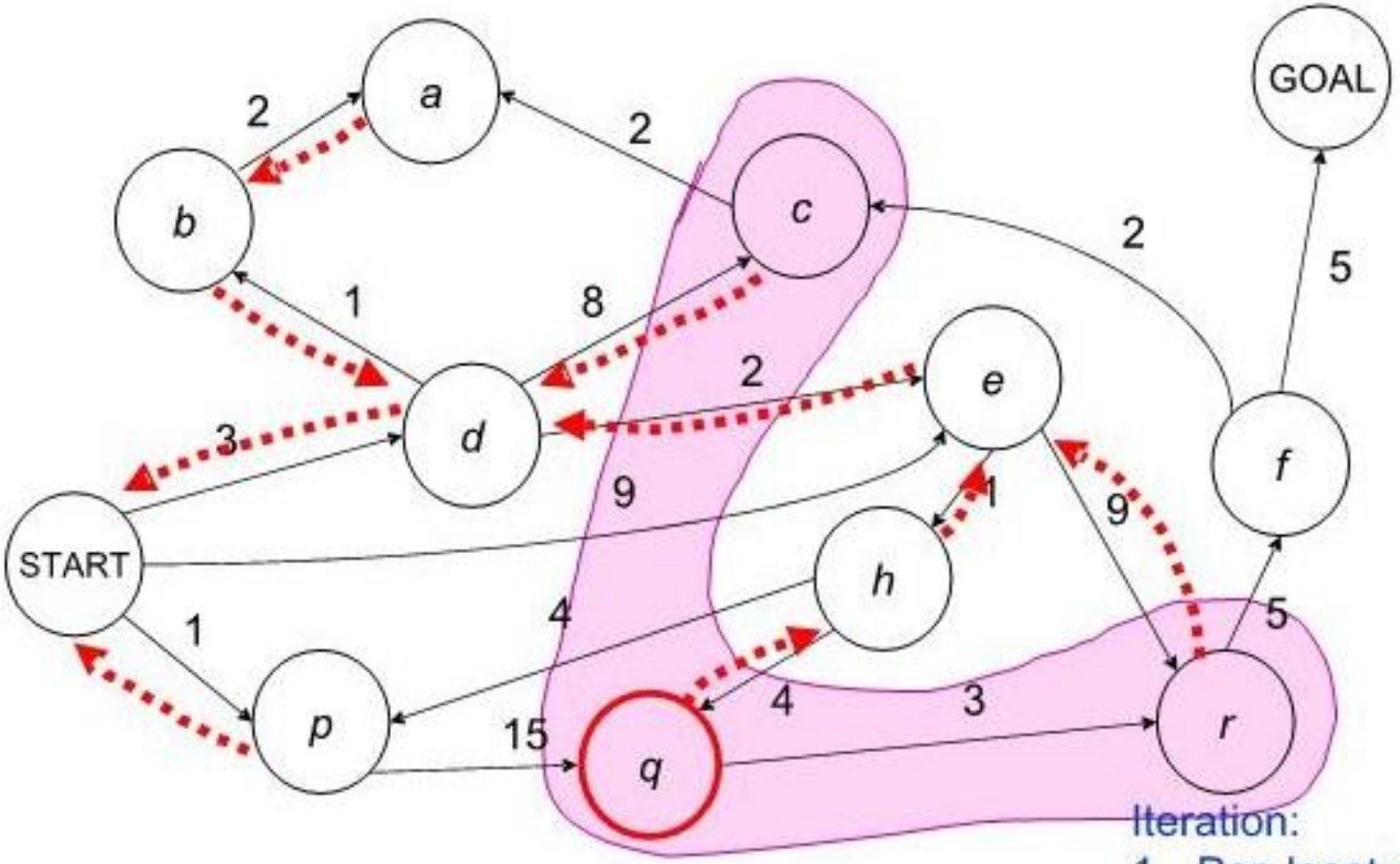


Iteration:

1. Pop least-cost state from PQ
2. Add successors

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

UCS Iterations



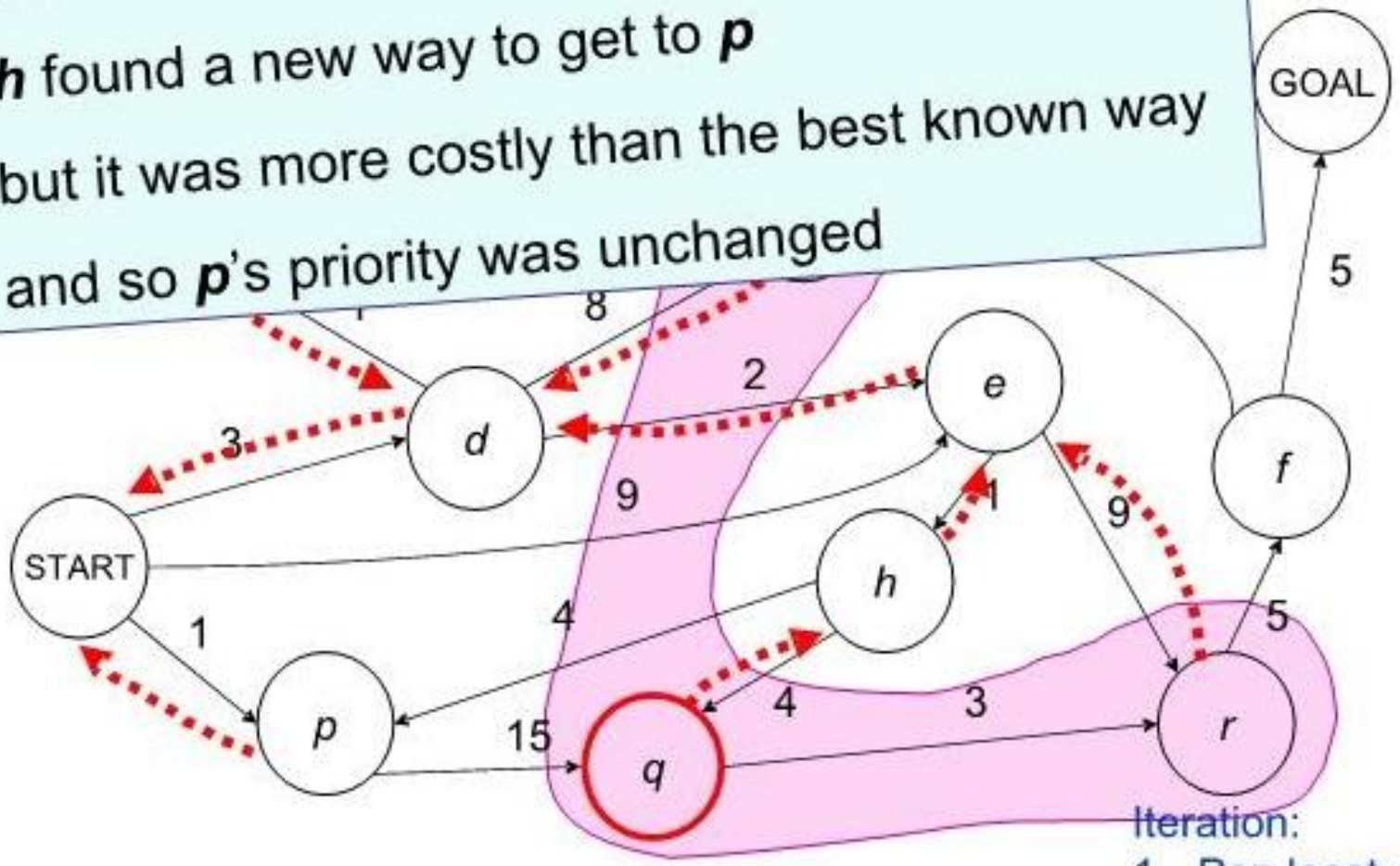
Iteration:

1. Pop least-cost state from PQ
2. Add successors

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

Note what happened here:

- ***h*** found a new way to get to ***p***
- but it was more costly than the best known way
- and so ***p***'s priority was unchanged

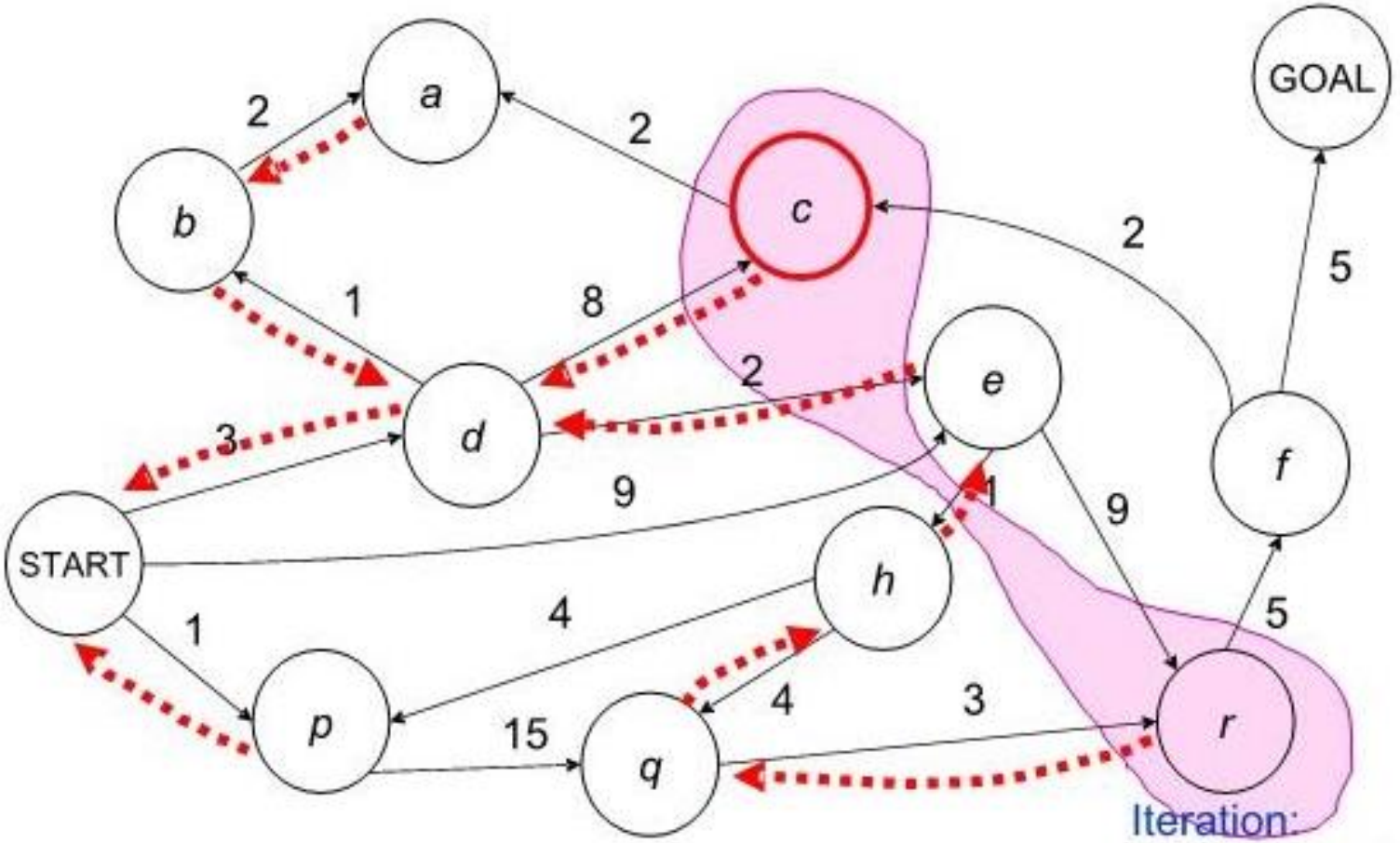


Iteration:

1. Pop least-cost state from PQ
2. Add successors

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

UCS Iterations

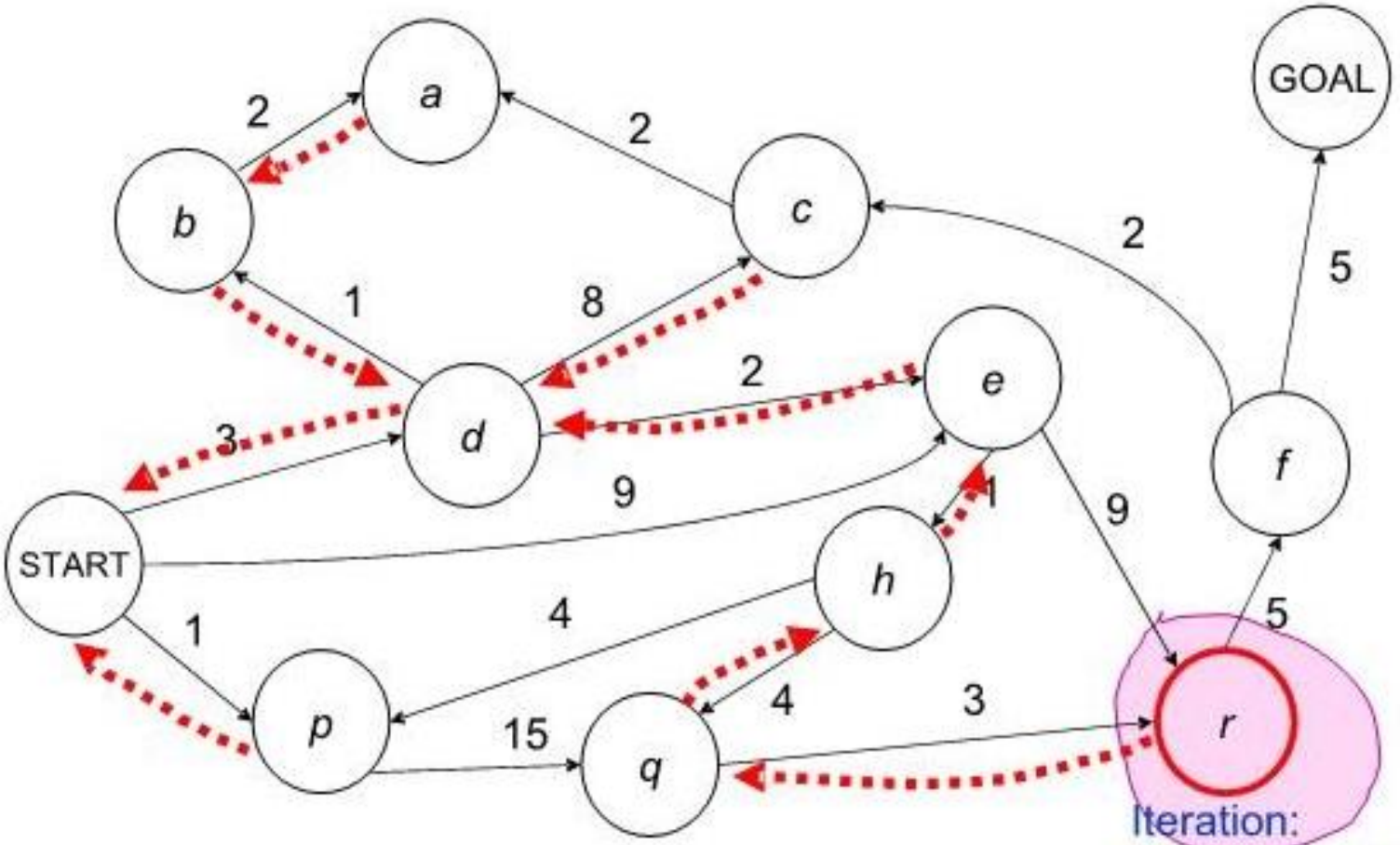


Iteration:

1. Pop least-cost state from PQ
2. Add successors

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

UCS Iterations

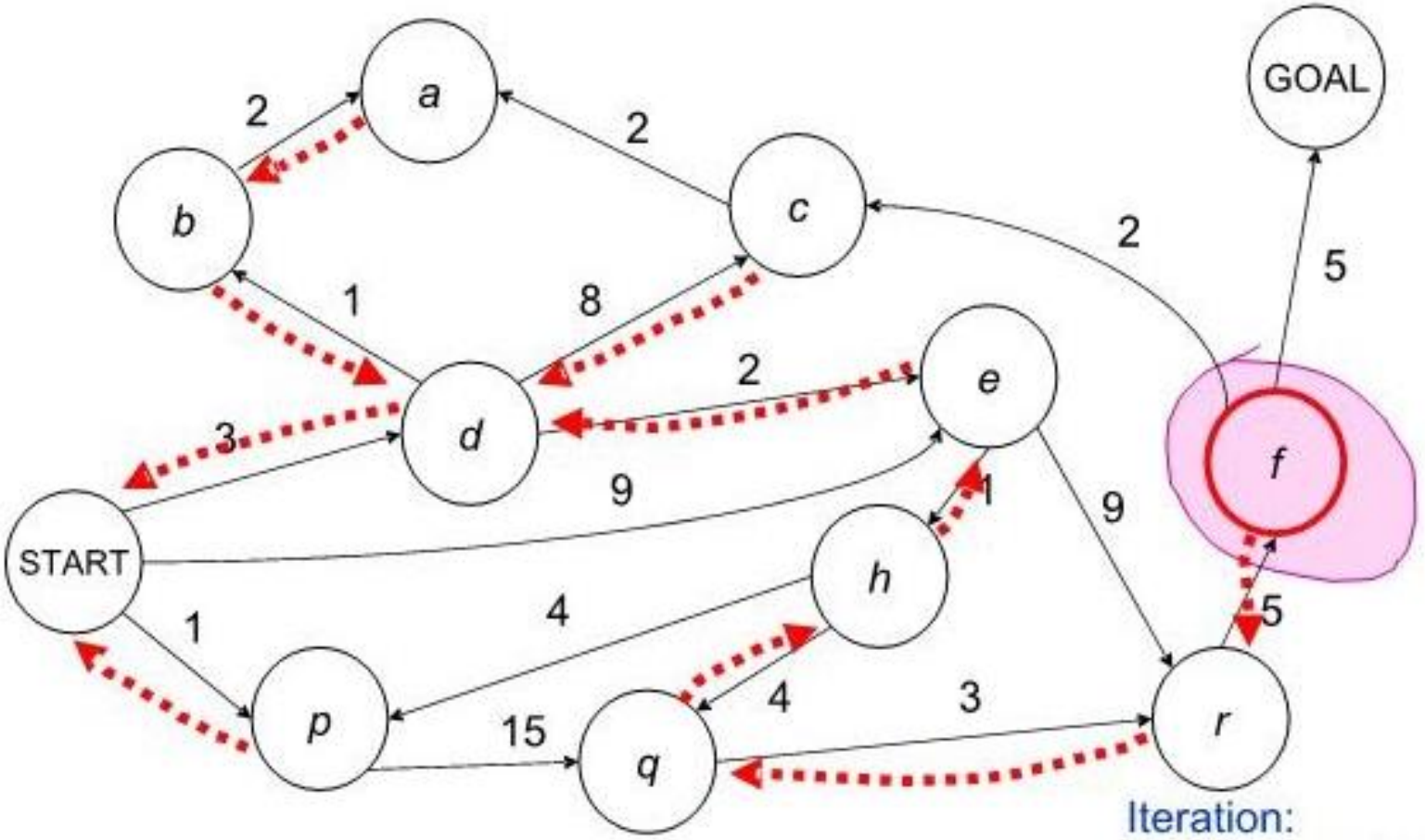


Iteration:

1. Pop least-cost state from PQ
2. Add successors

$PQ = \{ (r, 13) \}$

UCS Iterations

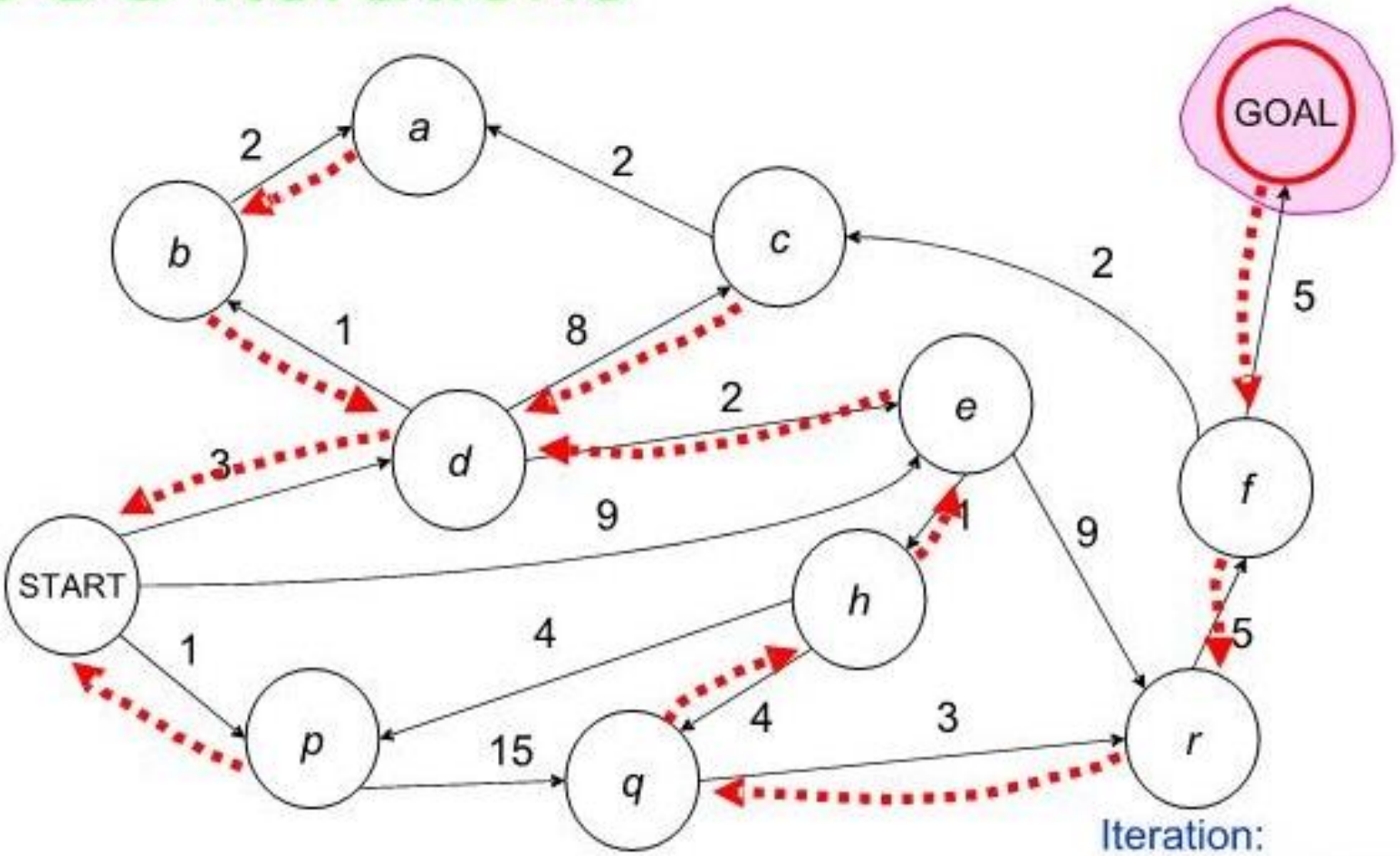


Iteration:

1. Pop least-cost state from PQ
2. Add successors

PQ = { (f,18) }

UCS Iterations

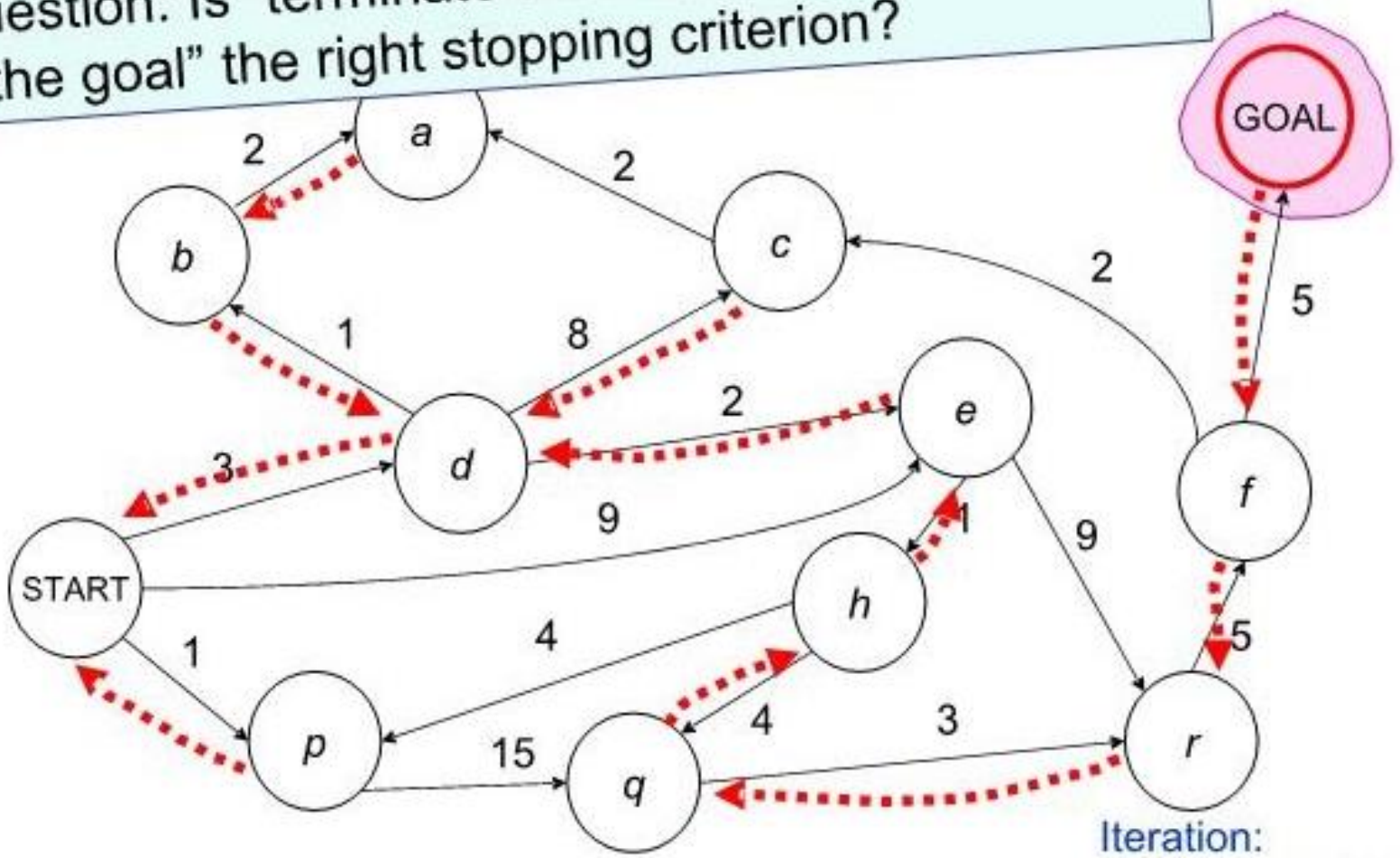


Iteration:

1. Pop least-cost state from PQ
2. Add successors

PQ = { (G,23) }

Question: Is "terminate as soon as you discover the goal" the right stopping criterion?

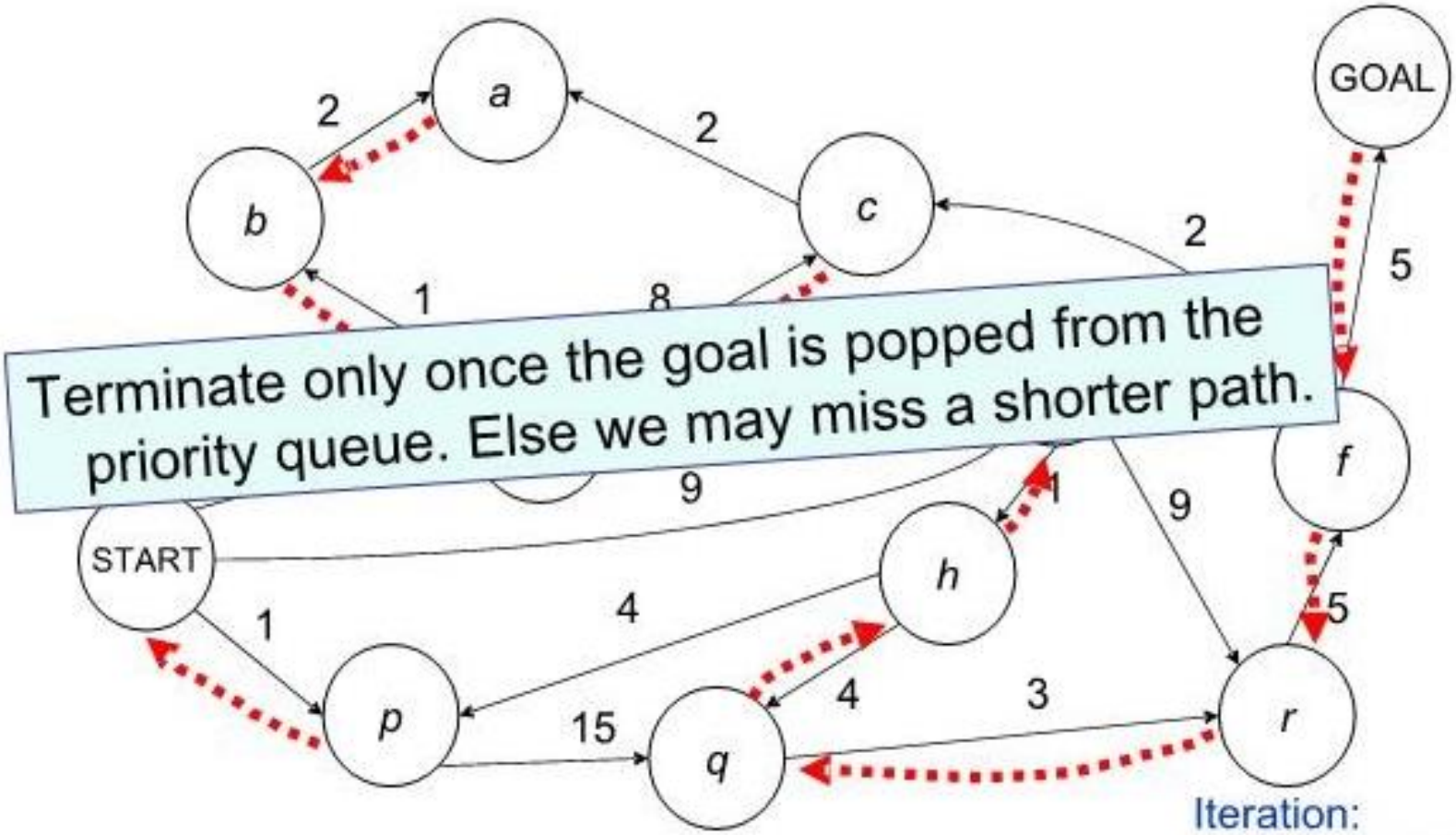


Iteration:

1. Pop least-cost state from PQ
2. Add successors

PQ = { (G,23) }

UCS terminates



Terminate only once the goal is popped from the priority queue. Else we may miss a shorter path.

PQ = { }

Iteration:

1. Pop least-cost state from PQ
2. Add successors

Informed Search Strategies

- Greedy search
- A* search
- IDA* search
- Hill climbing
- Simulated annealing

- Also known as **heuristic search**
  - require heuristic function

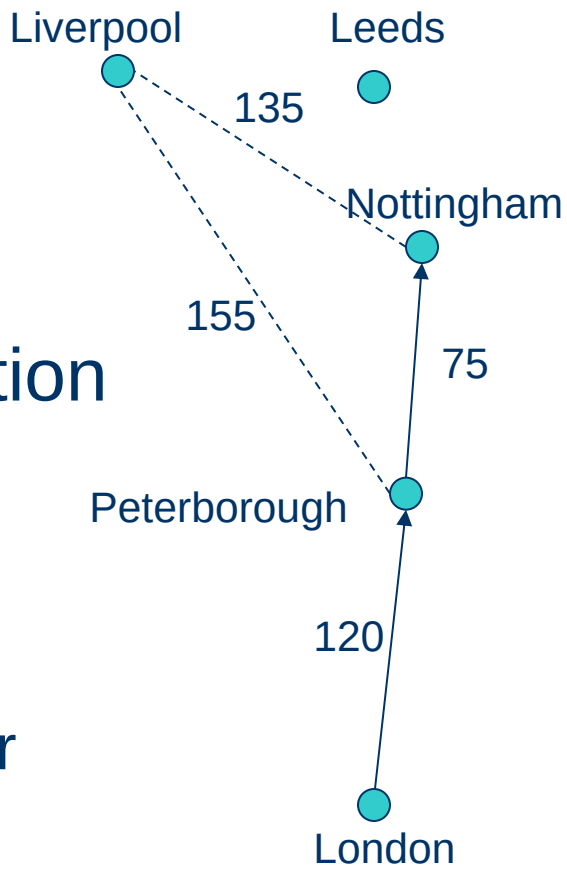
Best-First Search

- Evaluation function f gives cost for each state
  - Choose state with smallest f(state) (‘the best’)
  - Agenda: f decides where new states are put
  - Graph: f decides which node to expand next

- Many different strategies depending on f
  - For uniform-cost search f = path cost
  - Informed search strategies defines f based on heuristic function

Heuristic Functions

- Estimate of path cost h
  - From state to nearest solution
  - h(state) >= 0
  - h(solution) = 0
- Strategies can use this information
- Example: straight line distance
  - As the crow flies in route finding
- Where does h come from?
  - maths, introspection, inspection or programs



Heuristic Search

- In the searches we have covered so far, we never determine which states look most promising for expansion at a given time point. We never “look-ahead” to the goal.
- Often, however, we have some knowledge about the merit of states.

Heuristic functions

- We can encode each notion of the “merit” of a state into a heuristic function, h(s).

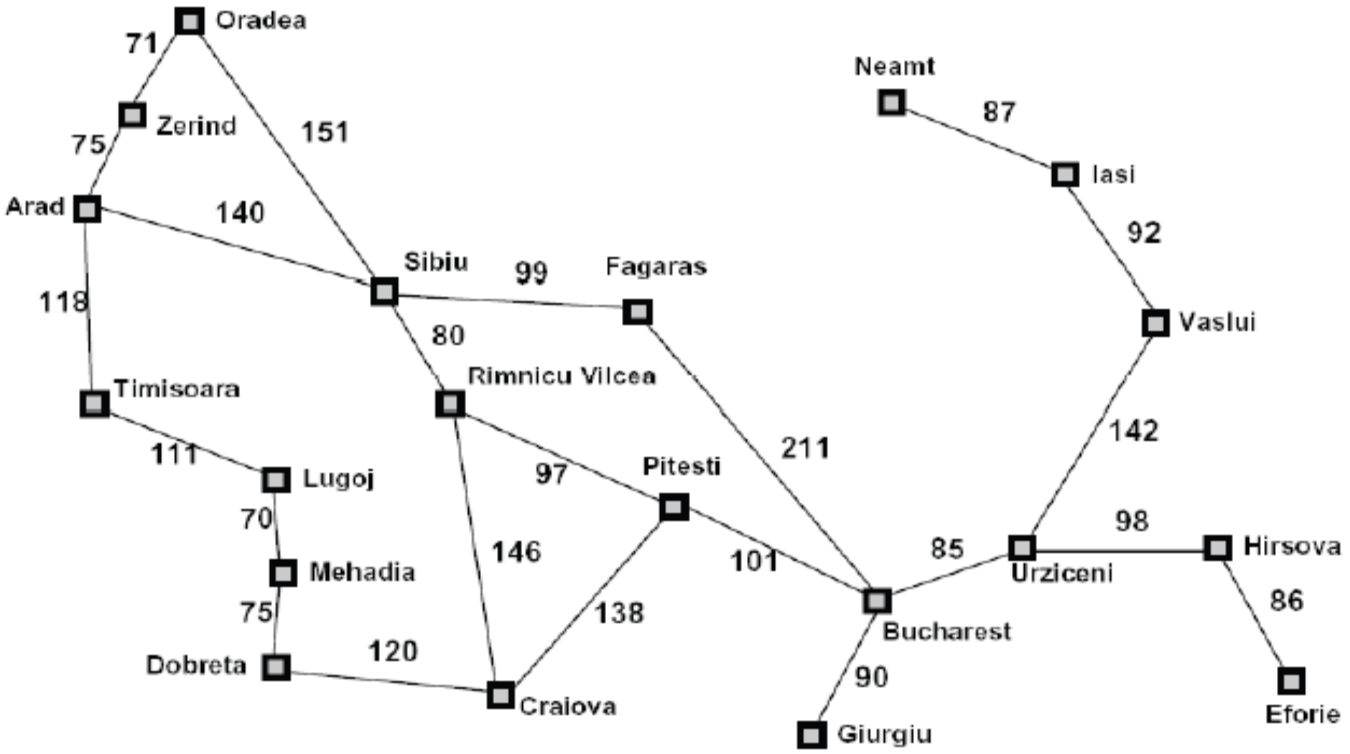
A heuristic function maps a state onto an estimate of the cost to the goal from that state.

Can you think of examples of heuristics?

For planning a path through a maze?

For solving a Rubick’s cube?

Euclidean distance as h(s)



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

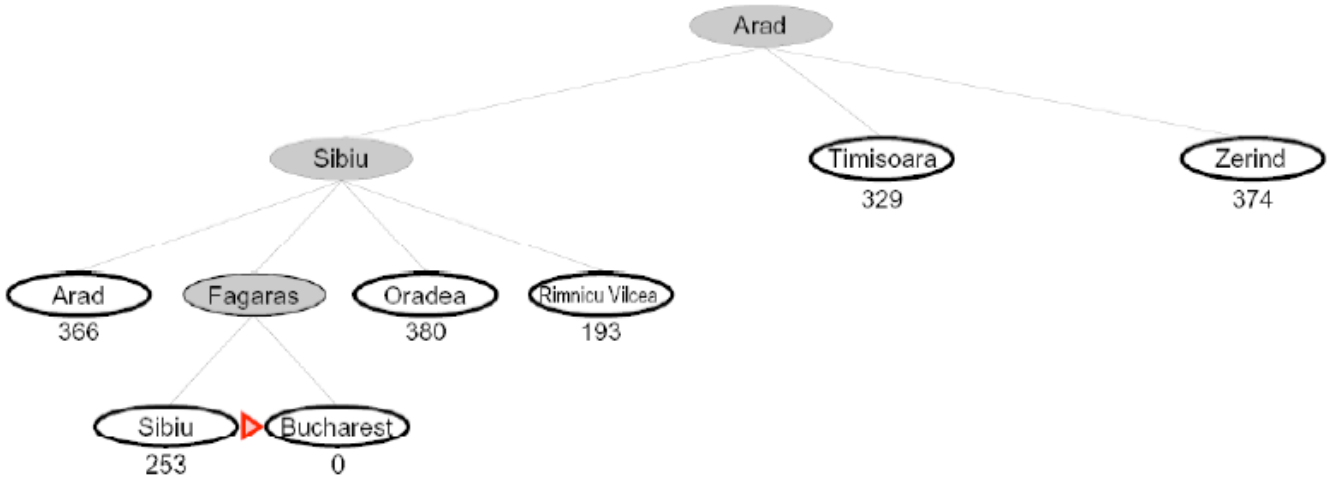
Say we want to plan a path from Arad to Bucharest, and we know the straight line distance from each city to our goal. This lets us plan our trip by picking cities at each time point that minimize the distance to our goal (or maximize our heuristic).

Greedy Search

- Always take the biggest bite
- f(state) = h(state)
  - Choose smallest estimated cost to solution
- Ignores the path cost

- Blind alley effect: early estimates very misleading
  - One solution: delay the use of greedy search
- Not guaranteed to find optimal solution
  - Remember we are *estimating* the path cost to solution

Greedy best first search

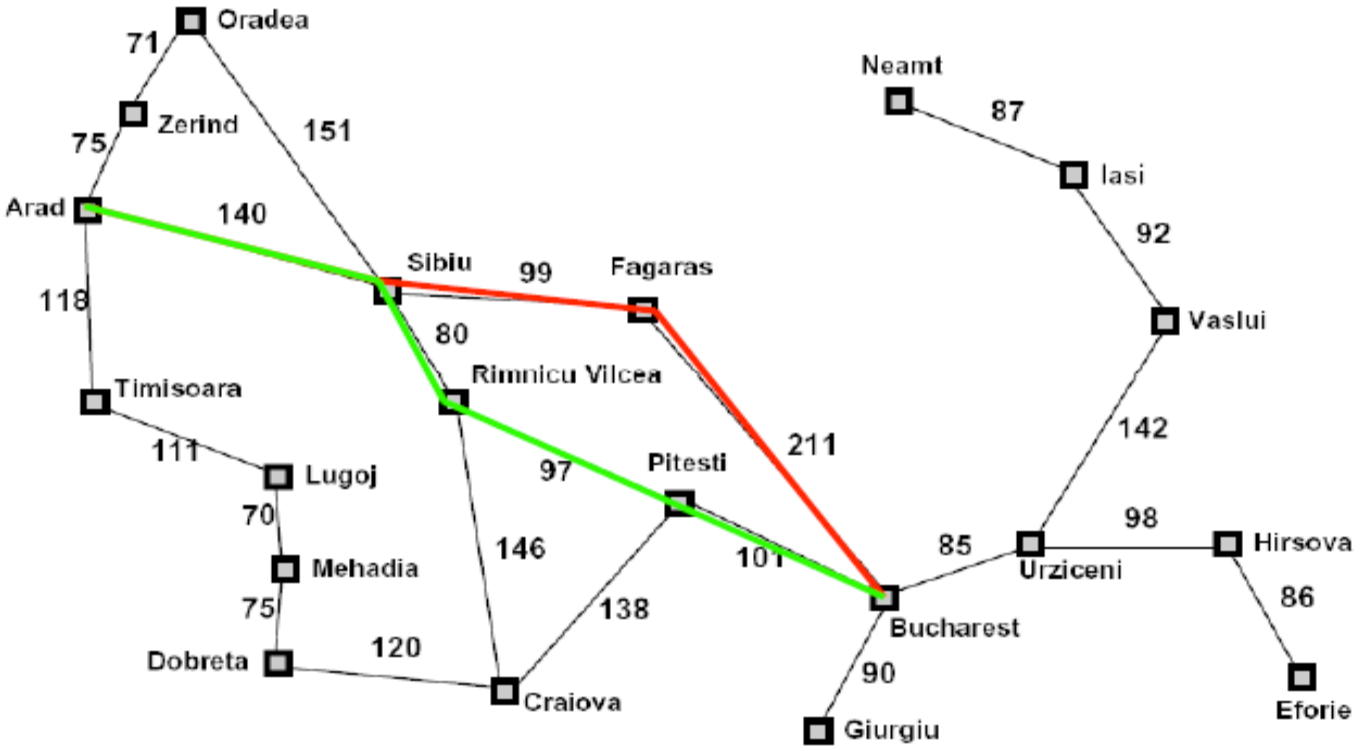


Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

This search strategy is called greedy best first search.

How can it possibly go wrong??

Greedy best first search



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

In red is the path we selected. In green is the shortest path between Arad and Bucharest. What happened?

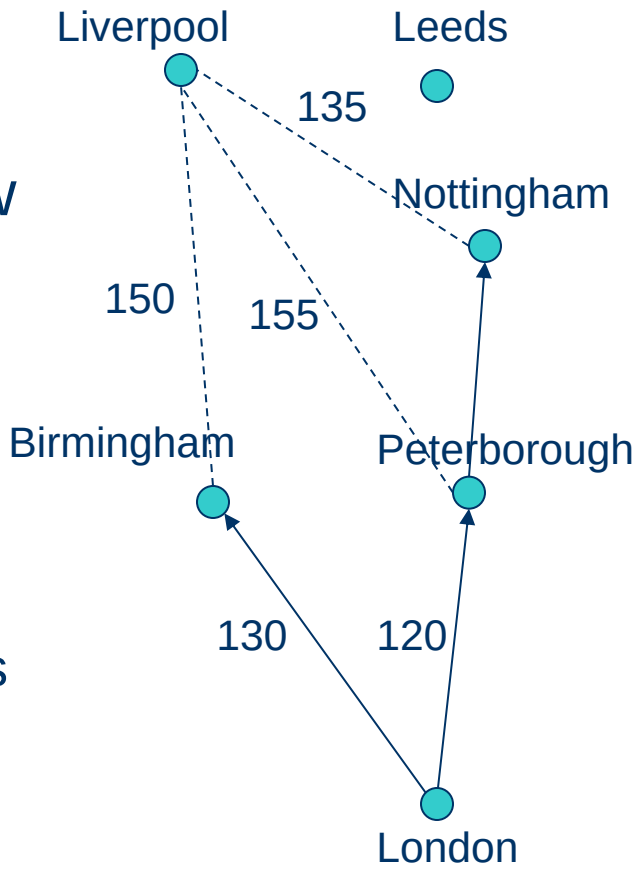
A* Search

- Path cost is g and heuristic function is h
  - f(state) = g(state) + h(state)
  - Choose smallest overall path cost (known + estimate)

- Combines uniform-cost and greedy search
- Can prove that A* is complete and optimal
  - But only if h is admissable,
    i.e. underestimates the true path cost from state to solution
  - See Russell and Norvig for proof

A* Example: Route Finding

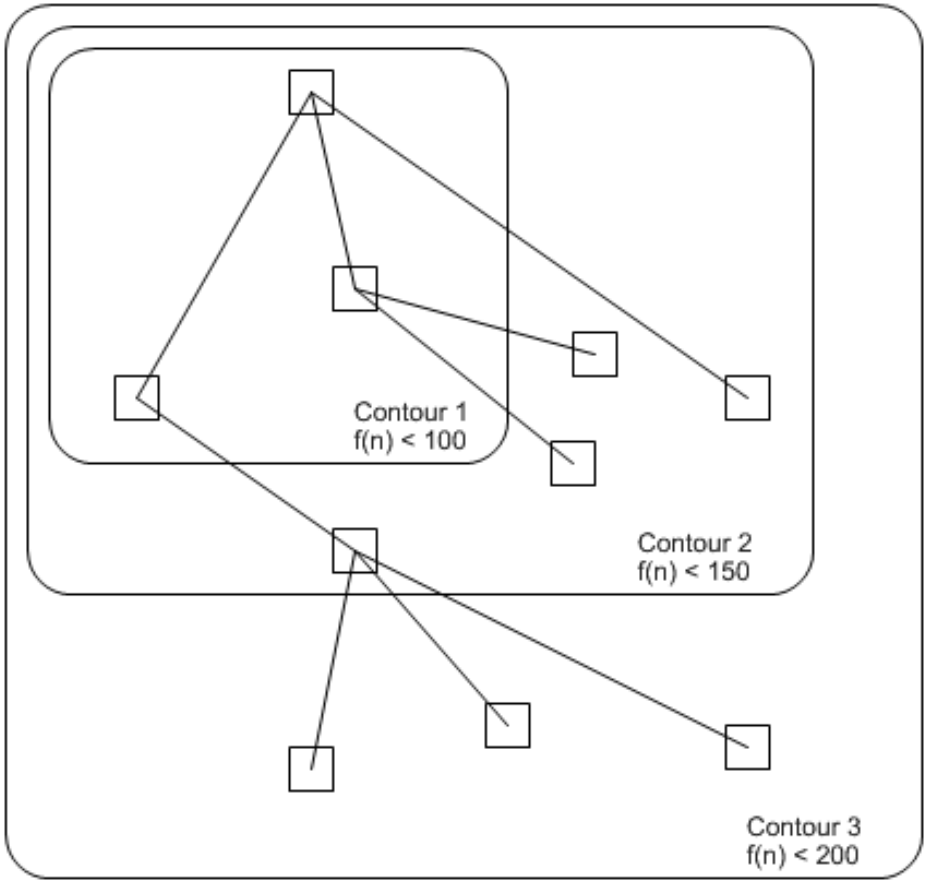
- First states to try:
  - Birmingham, Peterborough
- f(n) = distance from London + crow flies distance from state
  - i.e., solid + dotted line distances
  - f(Peterborough) = 120 + 155 = 275
  - f(Birmingham) = 130 + 150 = 280
- Hence expand Peterborough
  - But must go through Leeds from Notts
  - So later Birmingham is better



IDA* Search

- Problem with A* search
  - You have to record all the nodes
  - In case you have to back up from a dead-end
- A* searches often run out of memory, not time
- Use the same iterative deepening trick as IDS
  - But iterate over f(state) rather than depth
  - Define contours: f < 100, f < 200, f < 300 etc.
- Complete & optimal as A*, but less memory

IDA* Search: Contours



- Find all nodes
  - Where f(n) < 100
  - Ignore f(n) >= 100
- Find all nodes
  - Where f(n) < 200
  - Ignore f(n) >= 200
- And so on…

Search Strategies

Uninformed

- Breadth-first search
- Depth-first search
- Iterative deepening
- Bidirectional search
- Uniform-cost search

Informed

- Greedy search
- A* search
- IDA* search

The End!