Problem Solving as Search: Deterministic/Single-Agent

Department of Electrical and Electronics Engineering

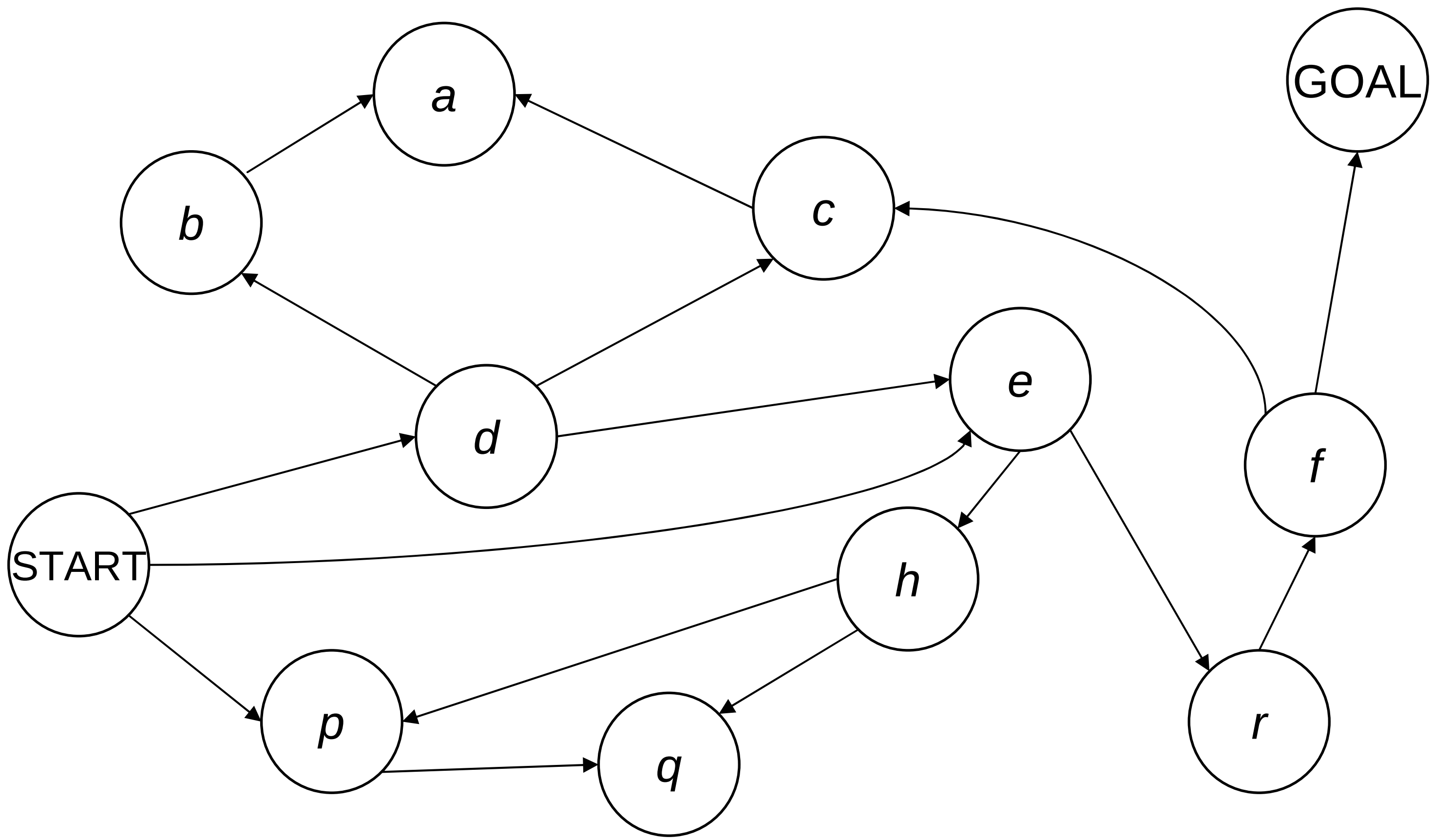
Dr. Afşar Saranlı

Thanks to Professor Andrew W. Moore (Carnegie Mellon University) http://www.cs.cmu.edu/~awm/tutorials
Also: Artificial Intelligence: A Modern Approach, 2nd Ed., Russel & Norvig

Overview

- Deterministic, single-agent, search problems
- Breadth First Search
- Optimality, Completeness, Time and Space complexity
- Search Trees
- Depth First Search
- Iterative Deepening
- Best First “Greedy” Search

A Search Problem



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

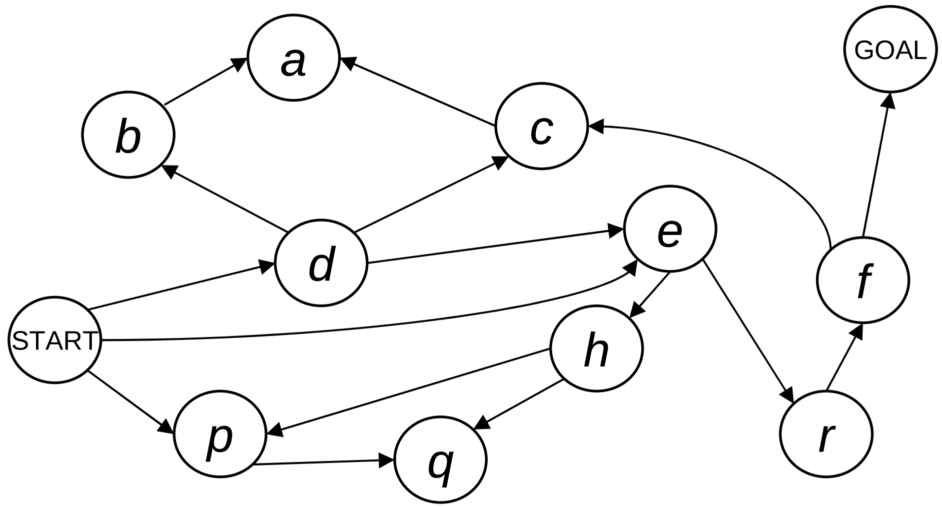
Formalizing a Search Problem

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* → *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}

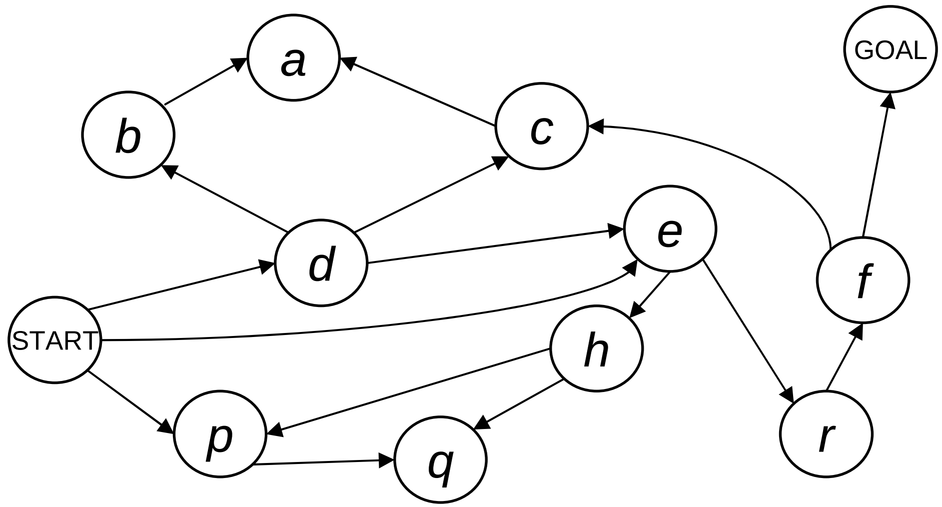
S = { START }

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 }

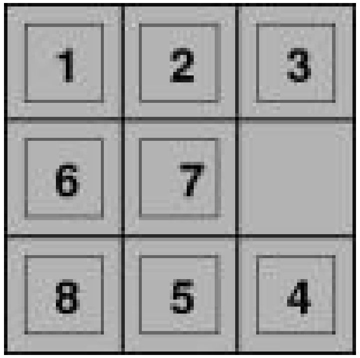
G = { GOAL }

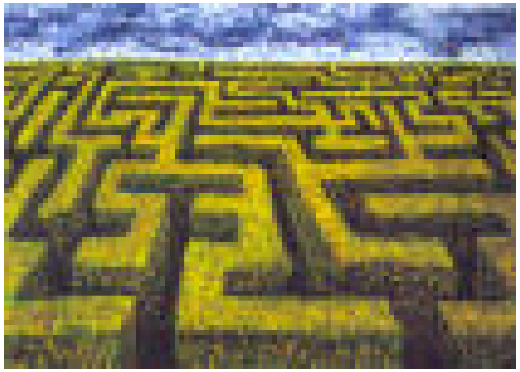
succs(*b*) = { *a* }
succs(*e*) = { *h* , *r* }
succs(*a*) = NULL … etc.

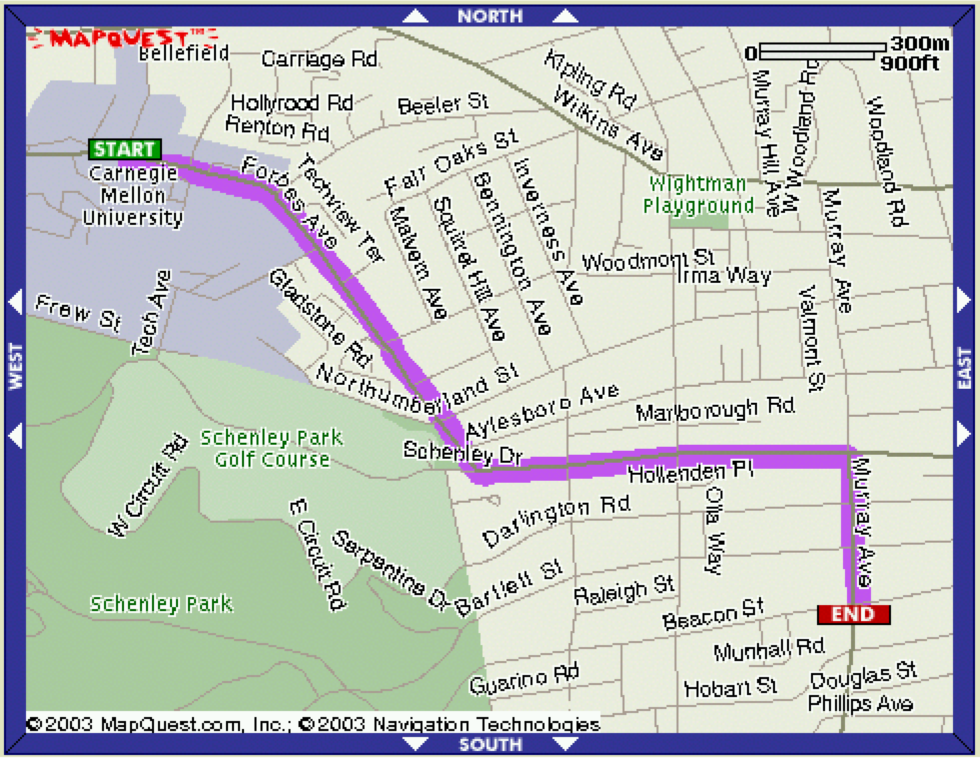
cost(*s*,*s'*) = 1 for all transitions

Why do we care? What problems are like this?

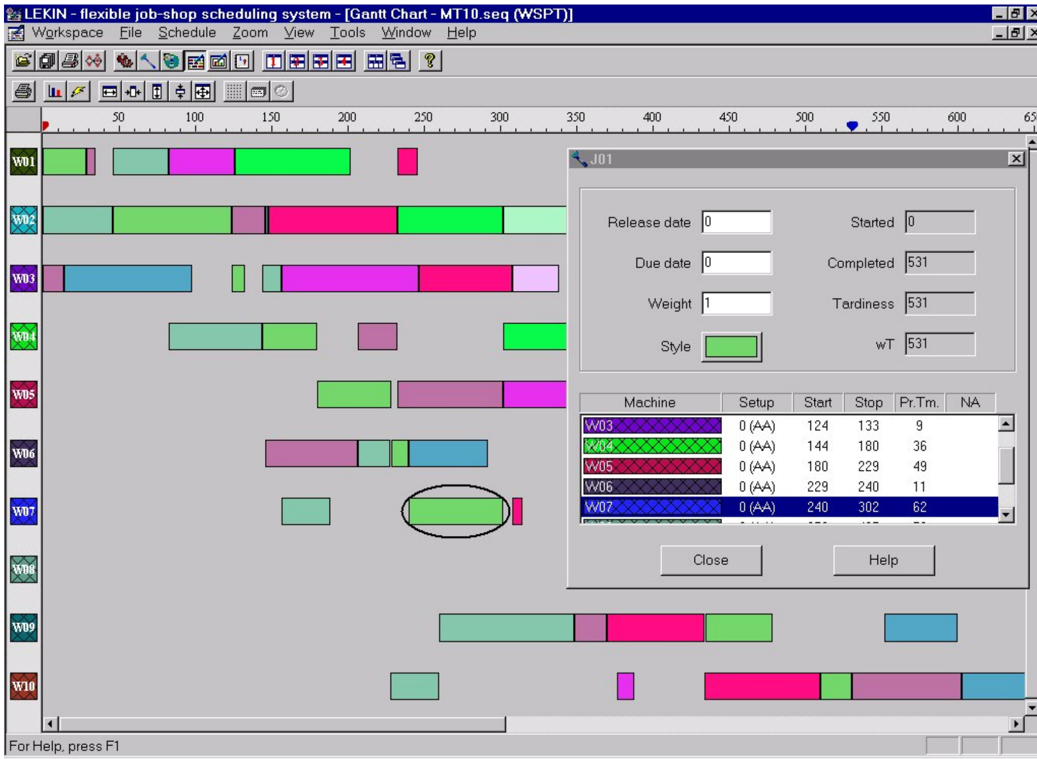
Search Problems





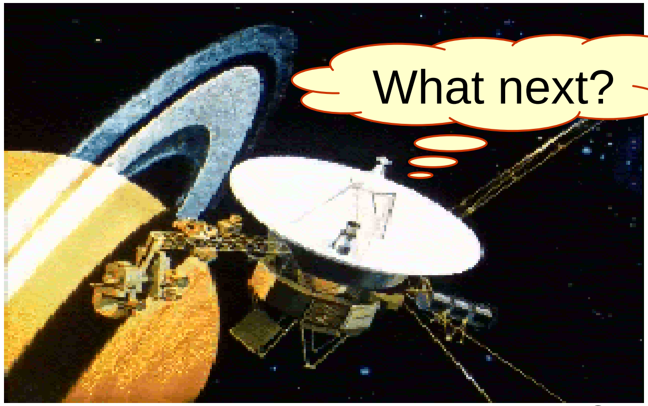


More Search Problems

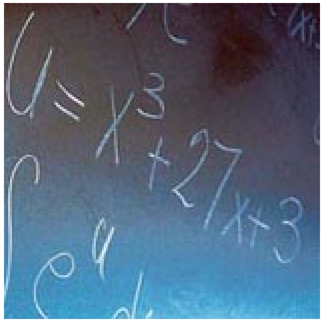


Scheduling

8-Queens



What next?



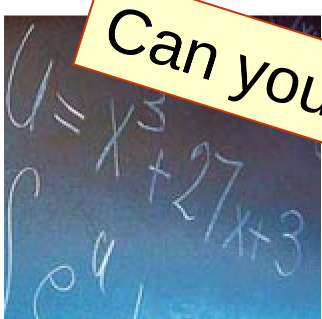
More Search Problems

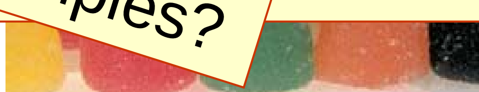
Scheduling

8-Queens

But there are plenty of things which we'd normally call search problems that don't fit our rigid definition…

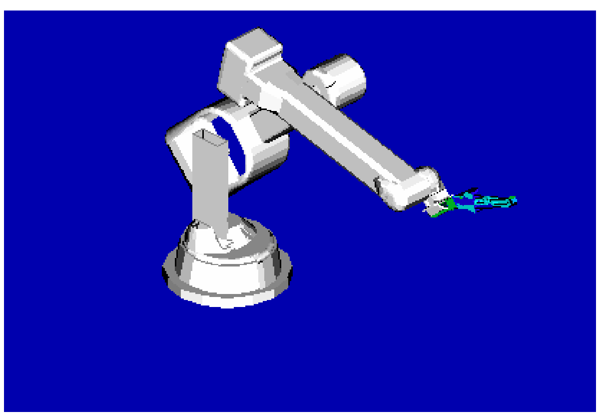
- A search problem has five components:
- *Q* , *S* , *G* , **succs** , **cost**
- *Q* is a finite set of states.
- $S \subseteq Q$ is a non-empty set of start states.
- $G \subseteq Q$ is a non-empty set of goal states.
- **succs** : $Q \rightarrow P(Q)$ is a function which takes a state as input and 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*.





Can you think of examples?

Our Definition Excludes…



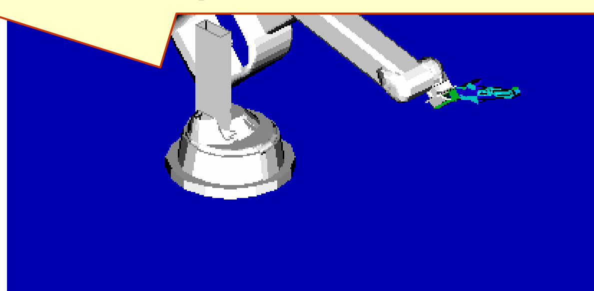
Our Definition Excludes…

Game against adversary

Chance

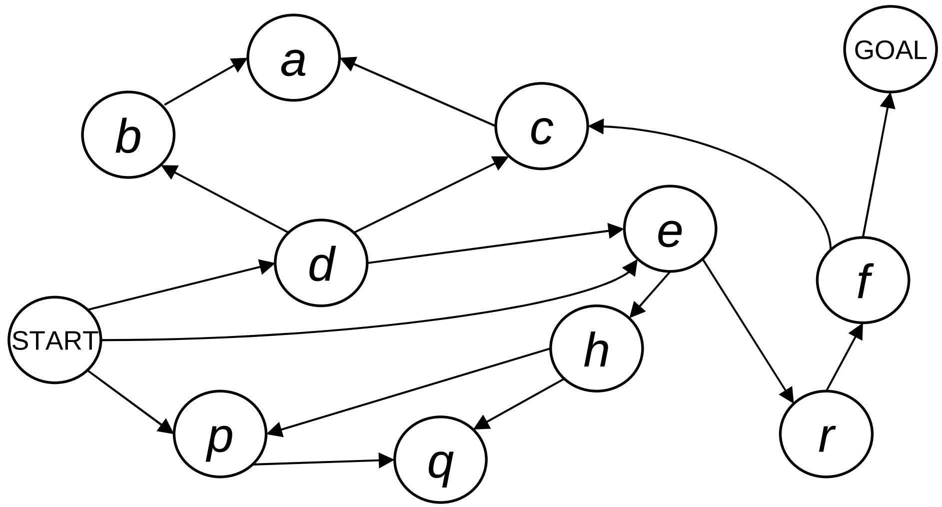
Hidden State

Continuum (infinite number) of states



All of the above, plus distributed team control

Breadth First Search



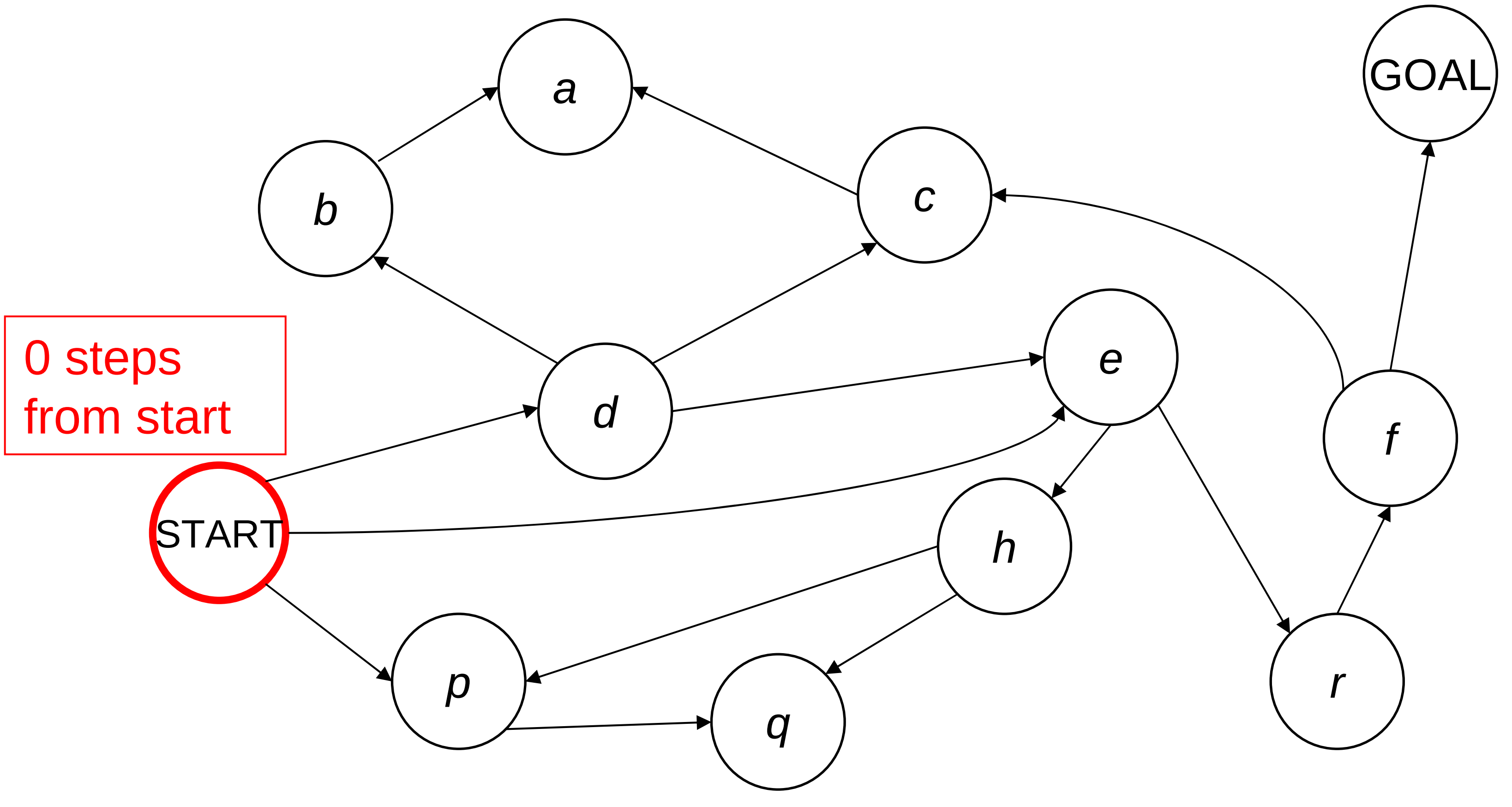
Label all states that are reachable from S in 1 step but aren't reachable in less than 1 step.

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

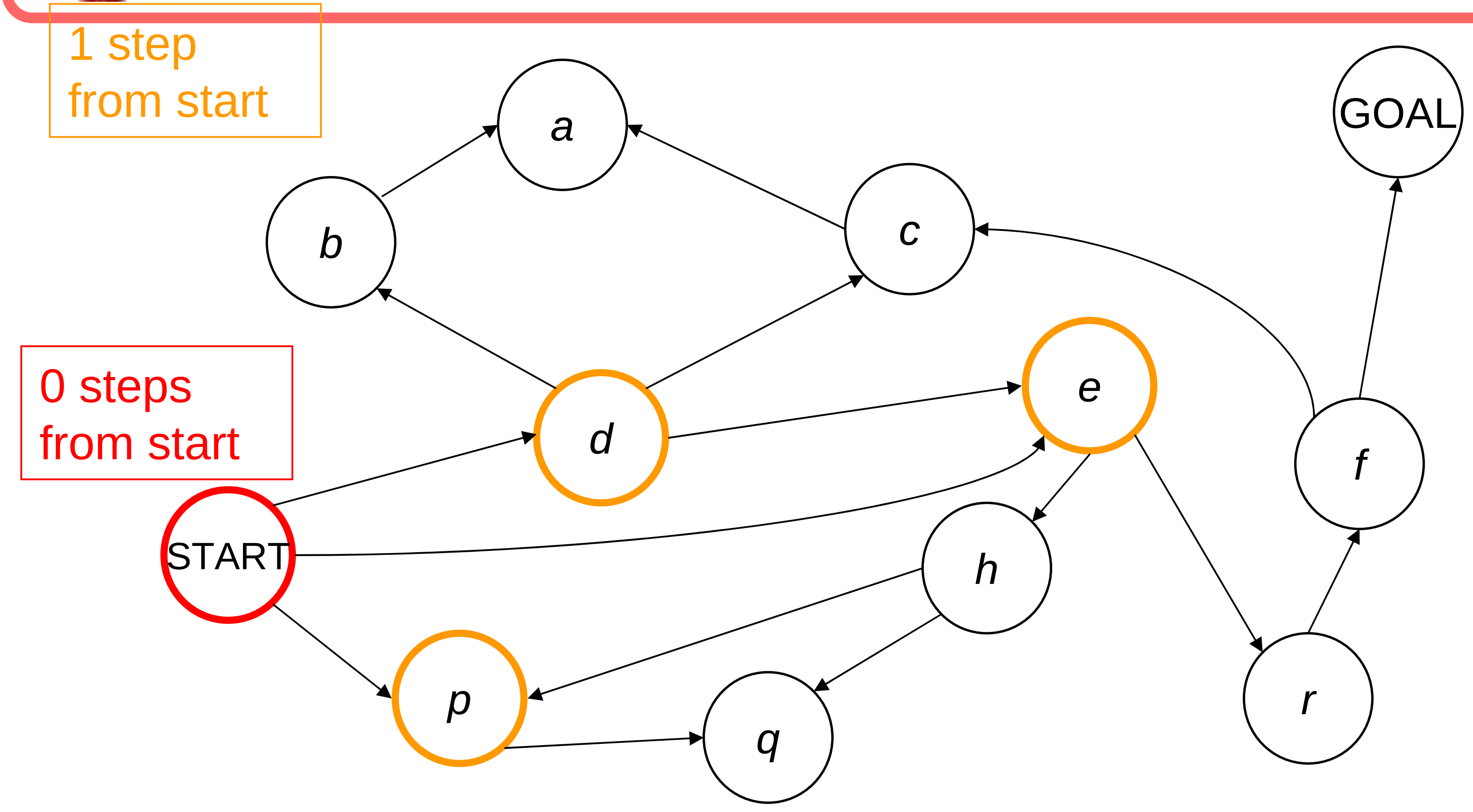
Then label all states that are reachable from S in 3 steps but aren't reachable in less than 3 steps.

Etc… until Goal state reached.

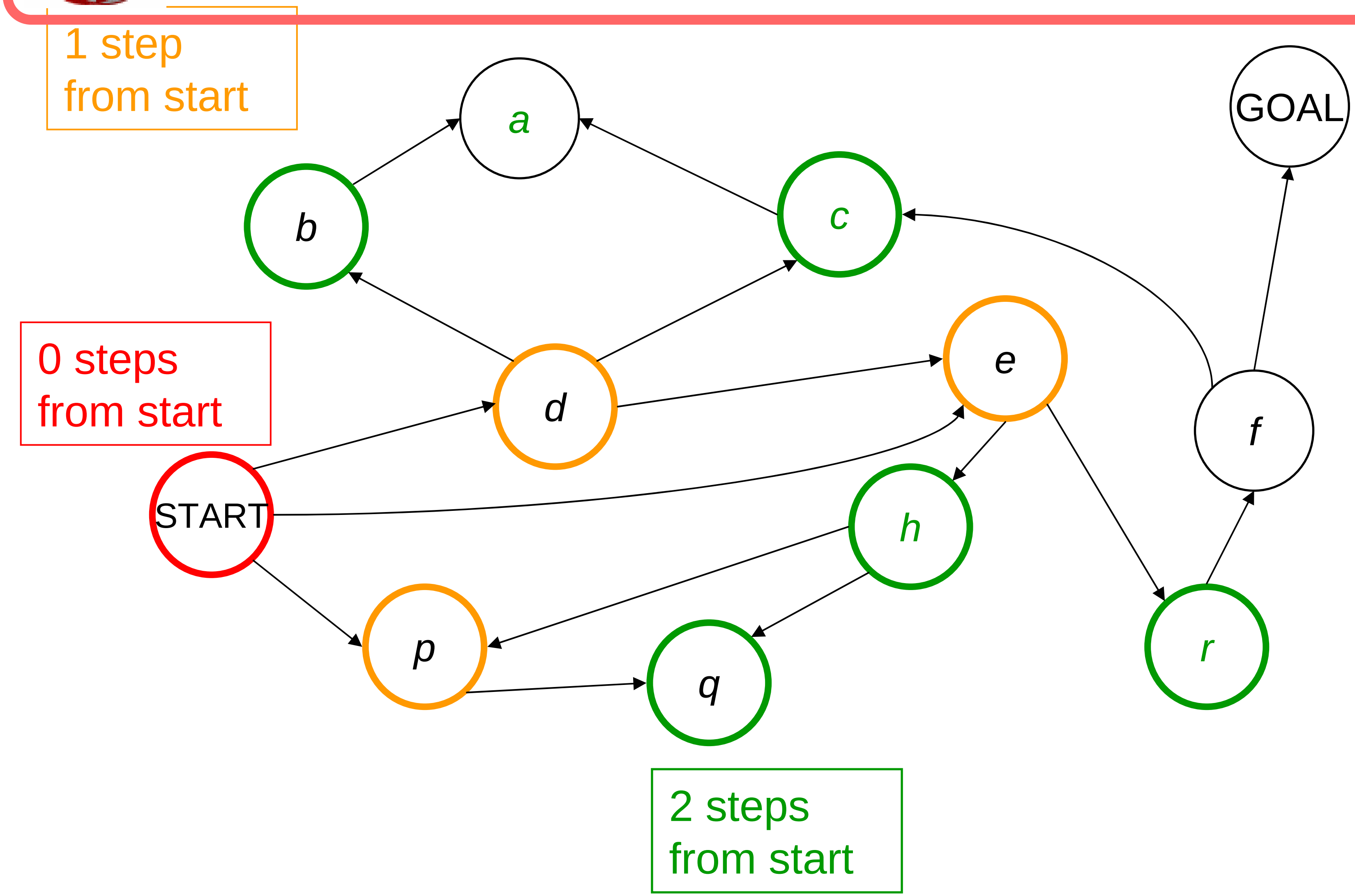
Breadth First Search



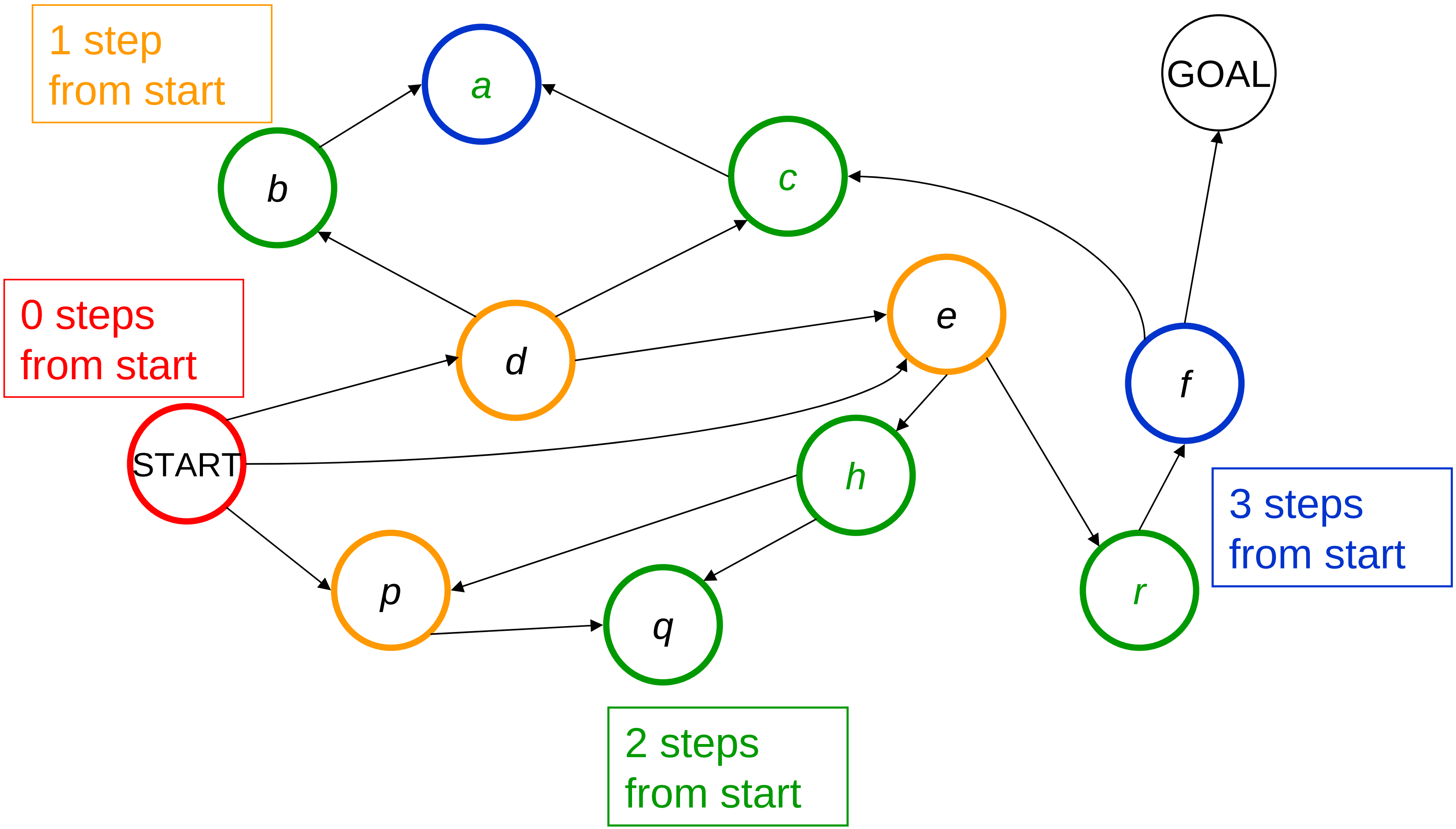
Breadth First Search



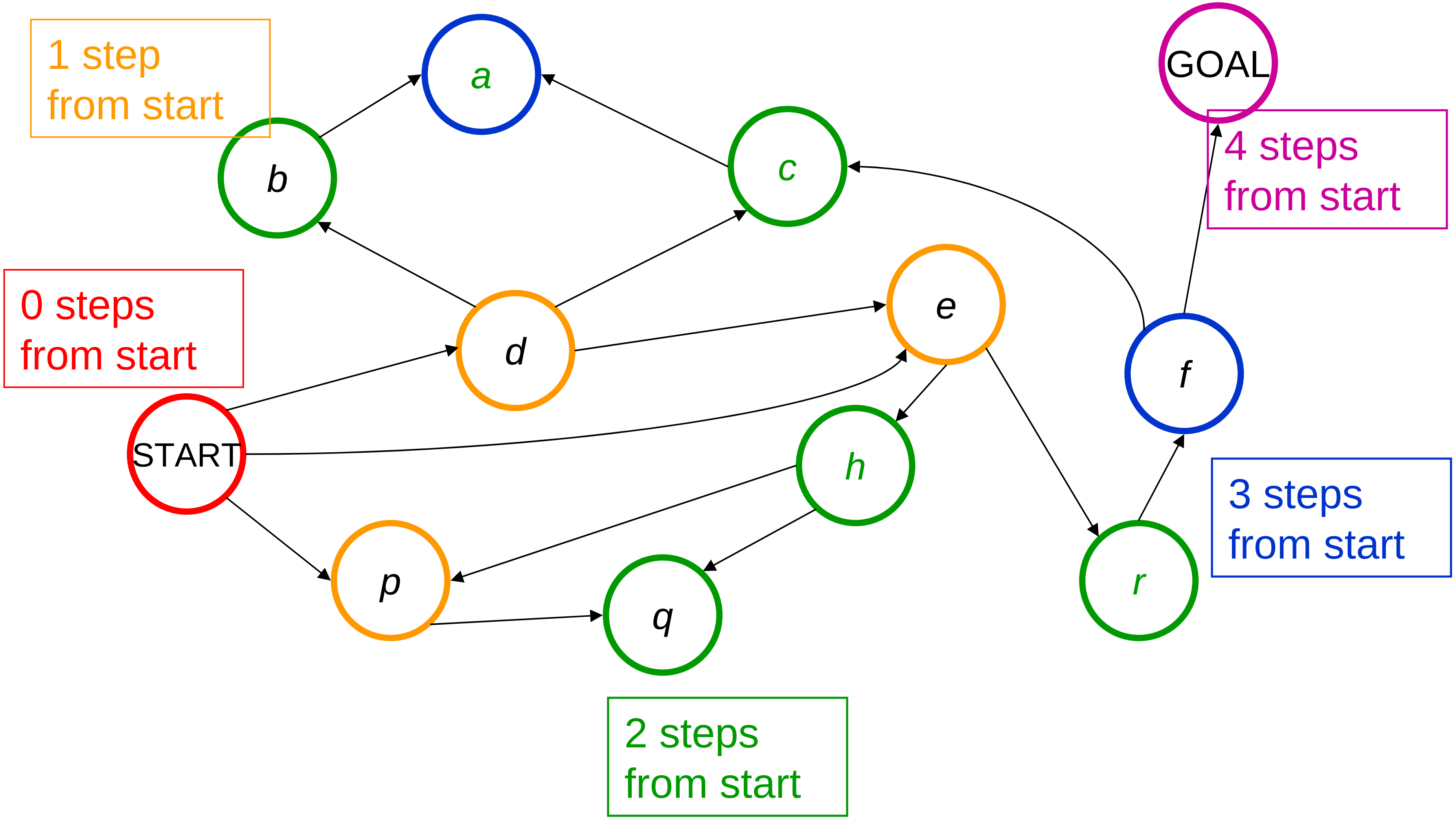
Breadth First Search



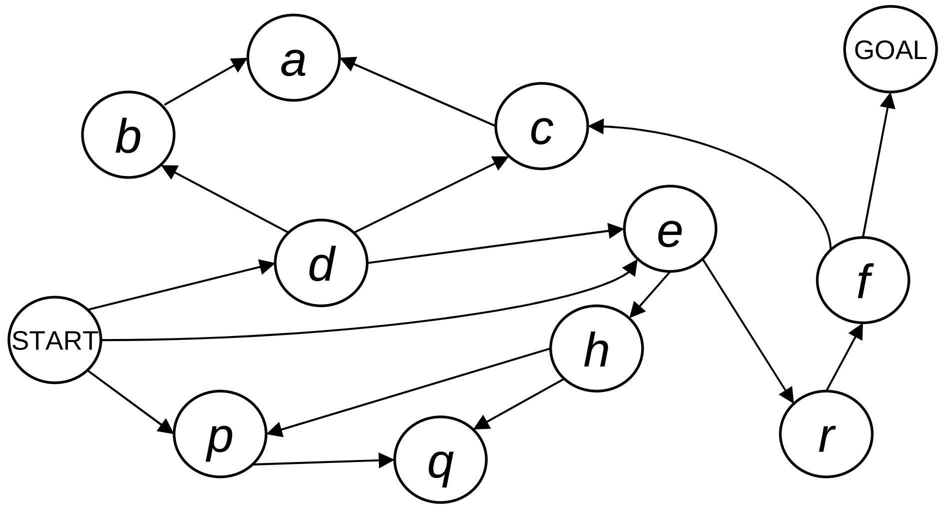
Breadth First Search



Breadth First Search



Remember the Path!

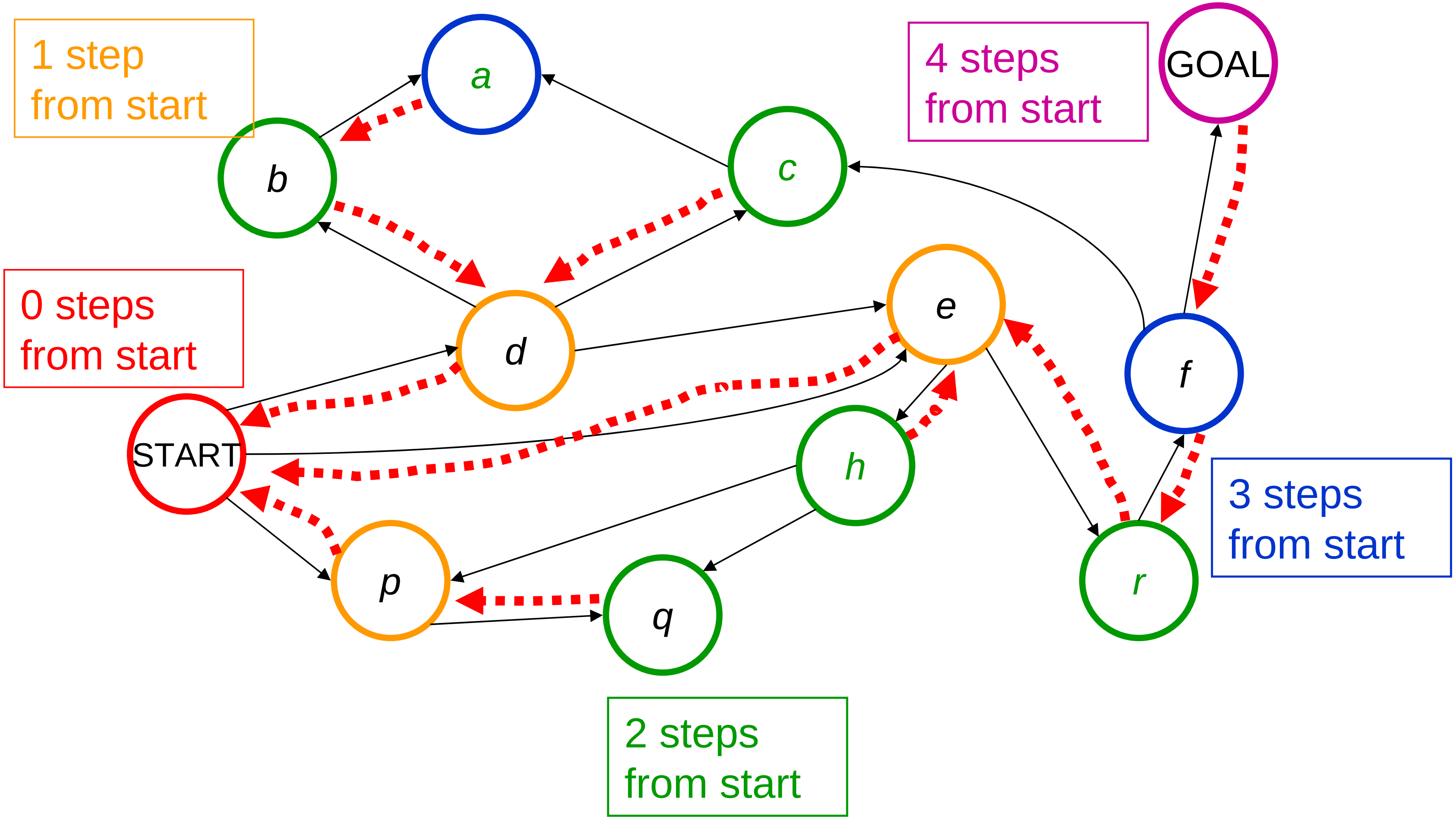


Also, when you label a state, record the predecessor state. This record is called a *backpointer*. The history of predecessors is used to generate the solution path, once you've found the goal:

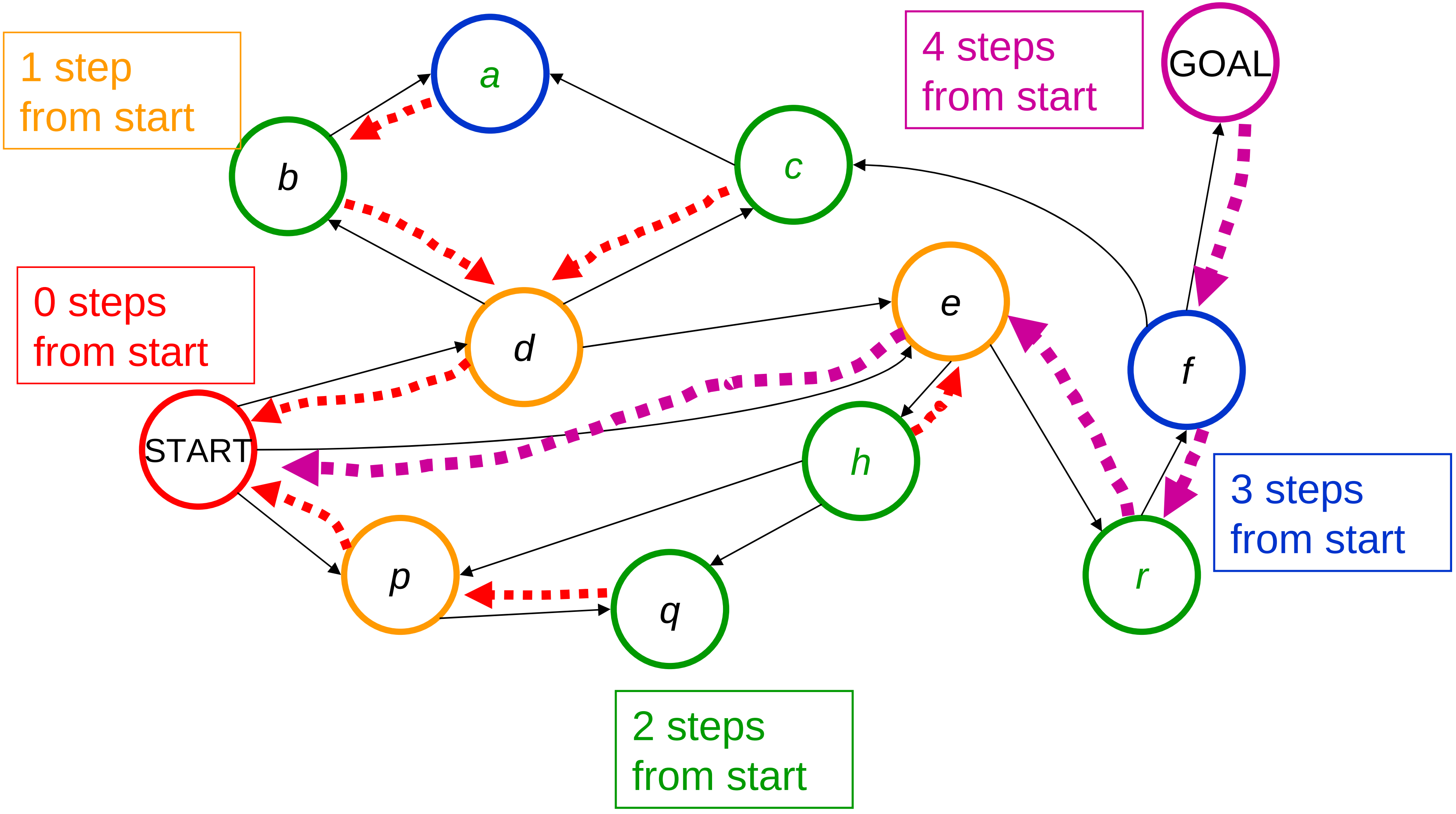
"I've got to the goal. I see I was at *f* before this. And I was at *r* before I was at *f*. And I was…

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

Backpointers…



Backpointers…



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*.

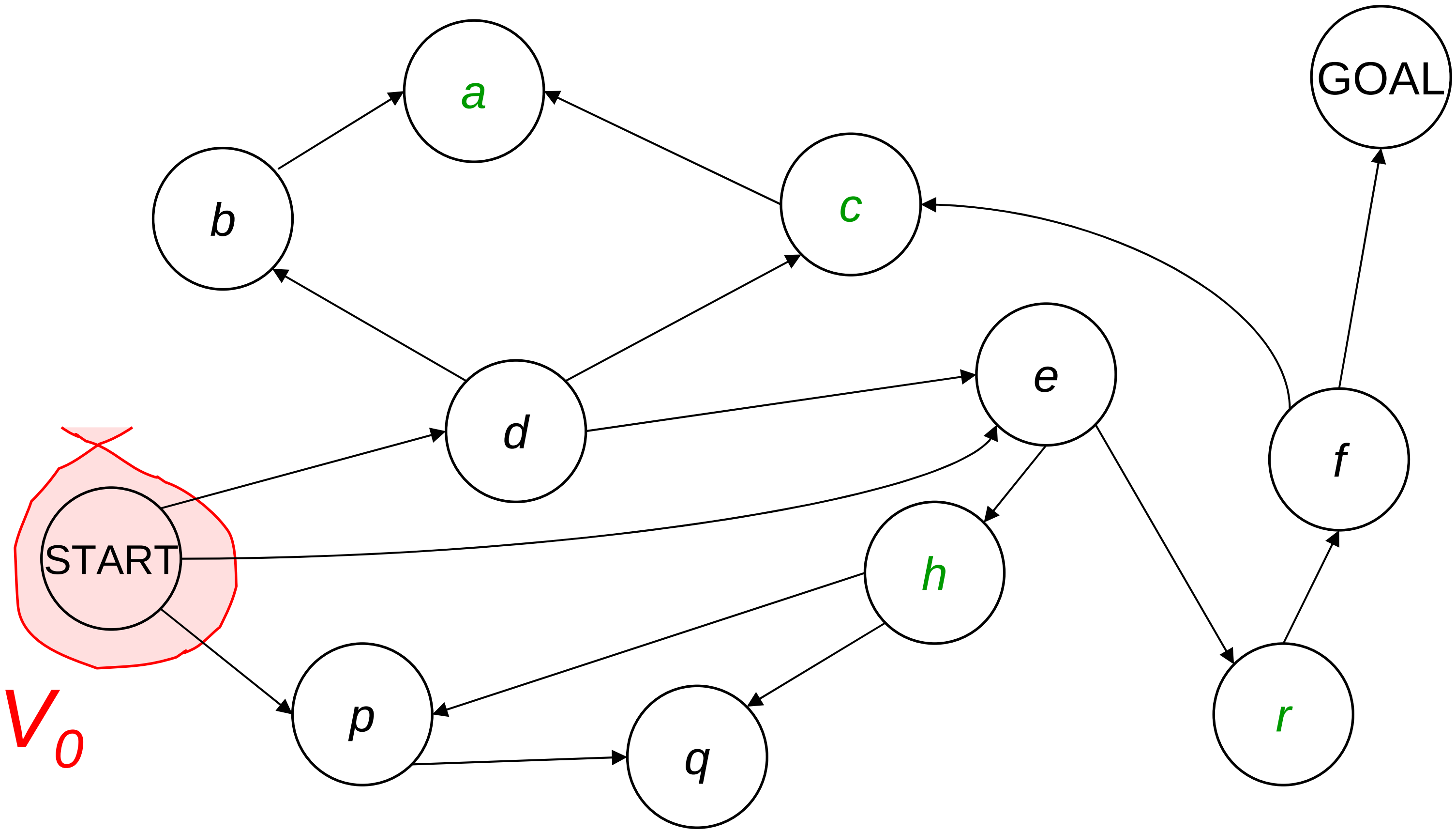
On the *k*th iteration of the algorithm we'll begin with V_k defined as the set of those states for which the shortest path from the start costs exactly *k steps*

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*

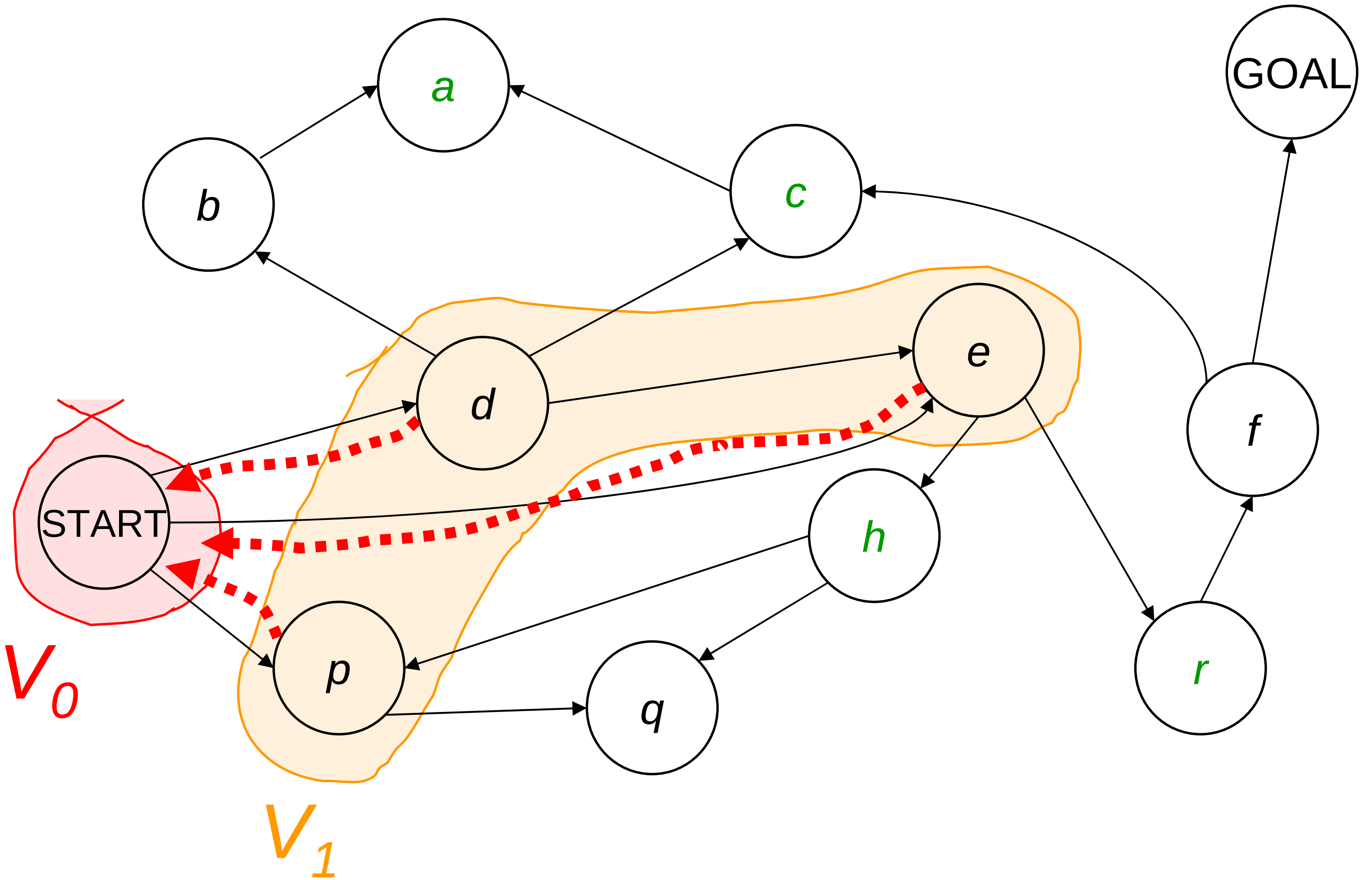
We begin with $k = 0$, V_0 = {START} and we'll define, *previous*(*START*) = *NULL*

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

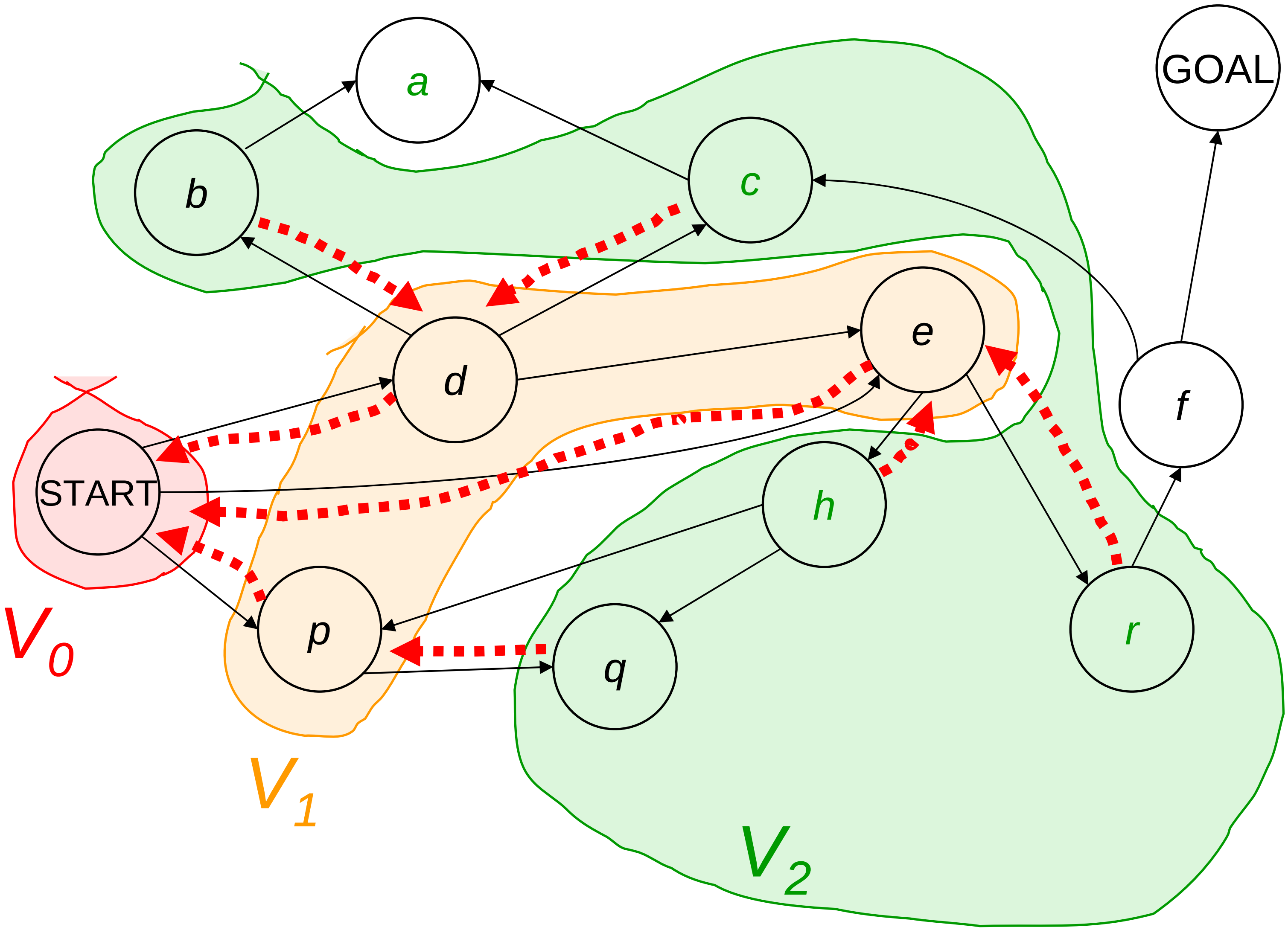
Breadth First Search



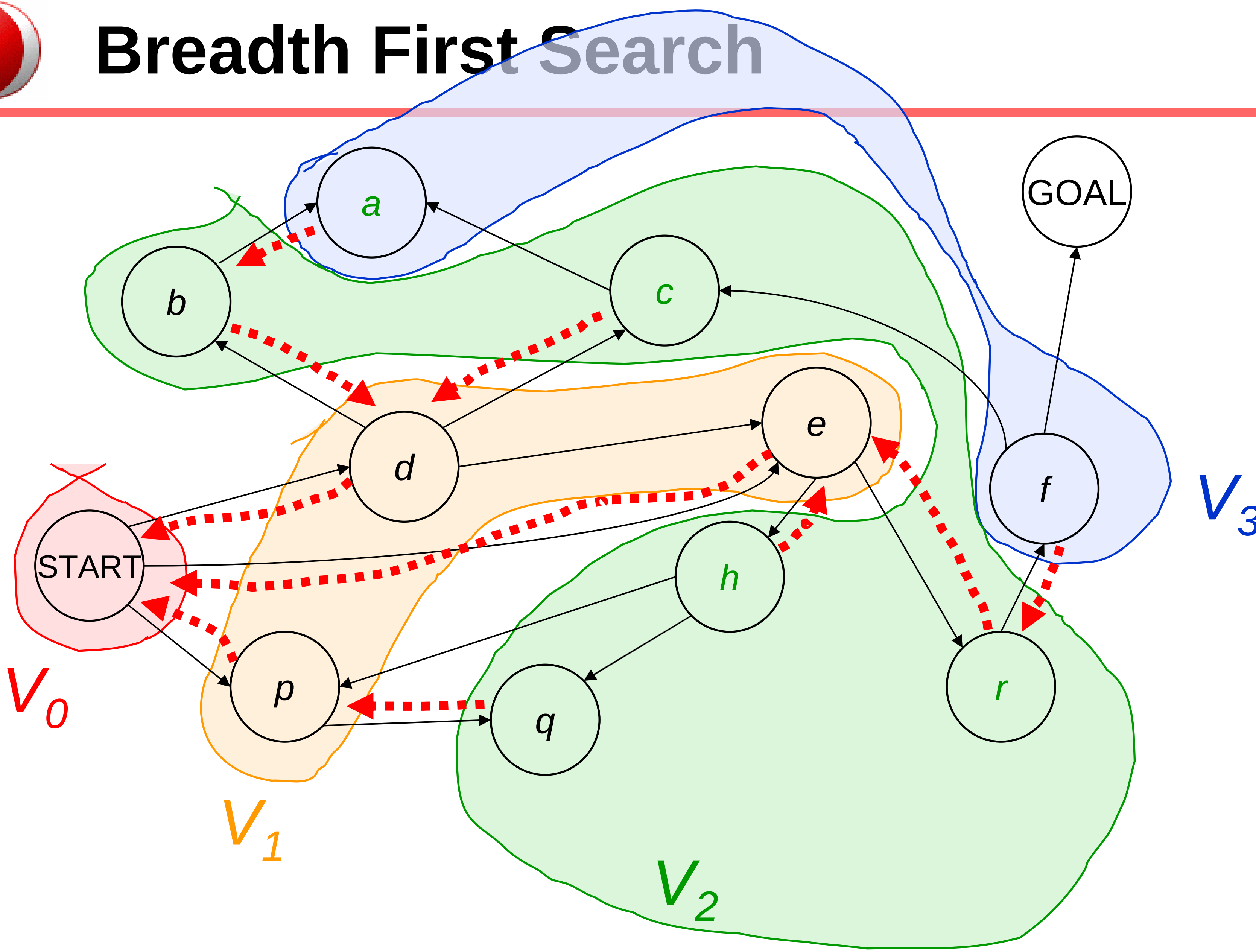
Breadth First Search



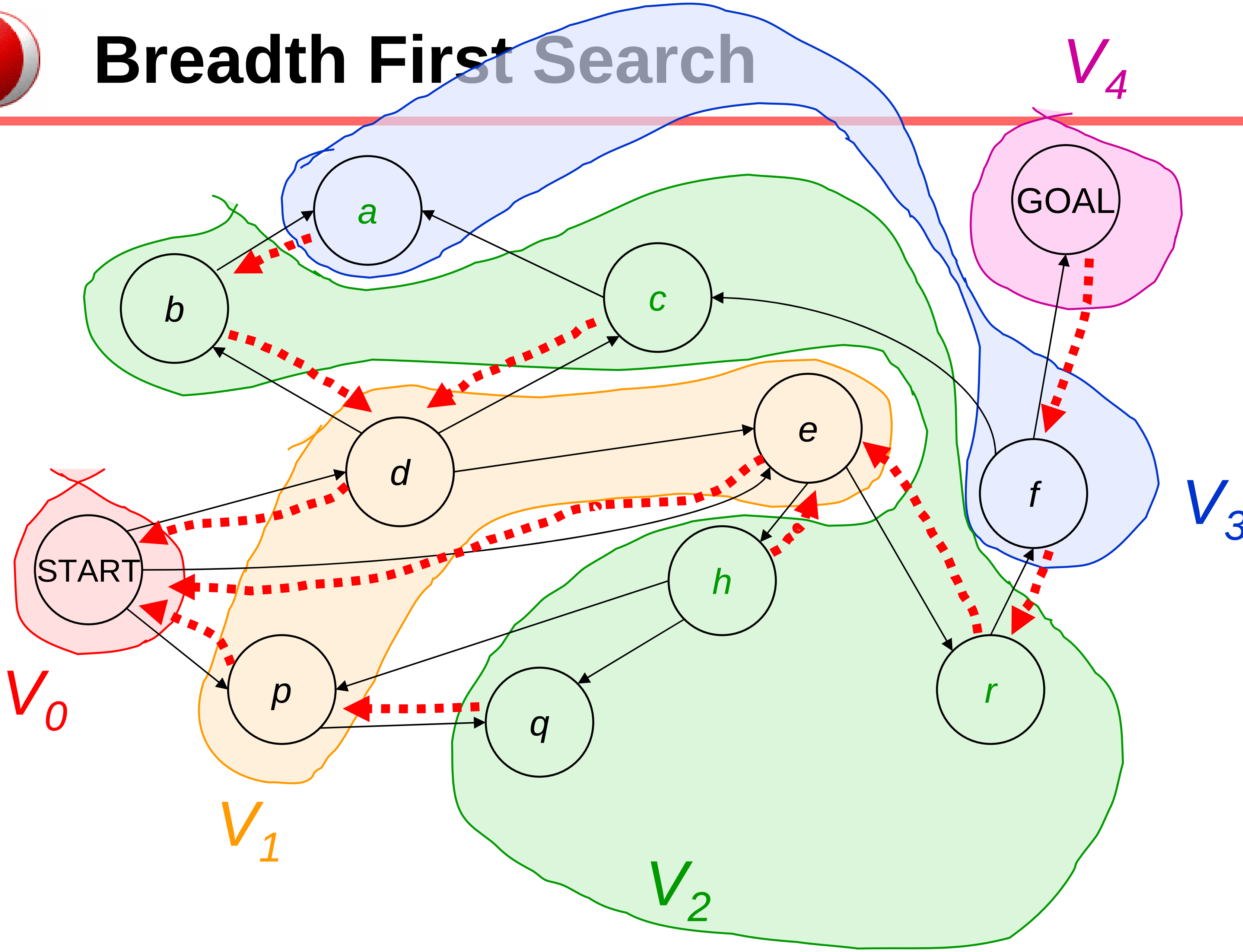
Breadth First Search



Breadth First Search



Breadth First Search



Breadth First Search

V_0 := S (the set of start states)

previous(*START*) := *NIL*

$k := 0$

while (no goal state is in V_k and V_k is not empty) **do**

    V_{k+1} := empty set

    For each state s in V_k

        For each state s' in **succs**(s)

            If s' has not already been labeled

                Set *previous*(s') := s

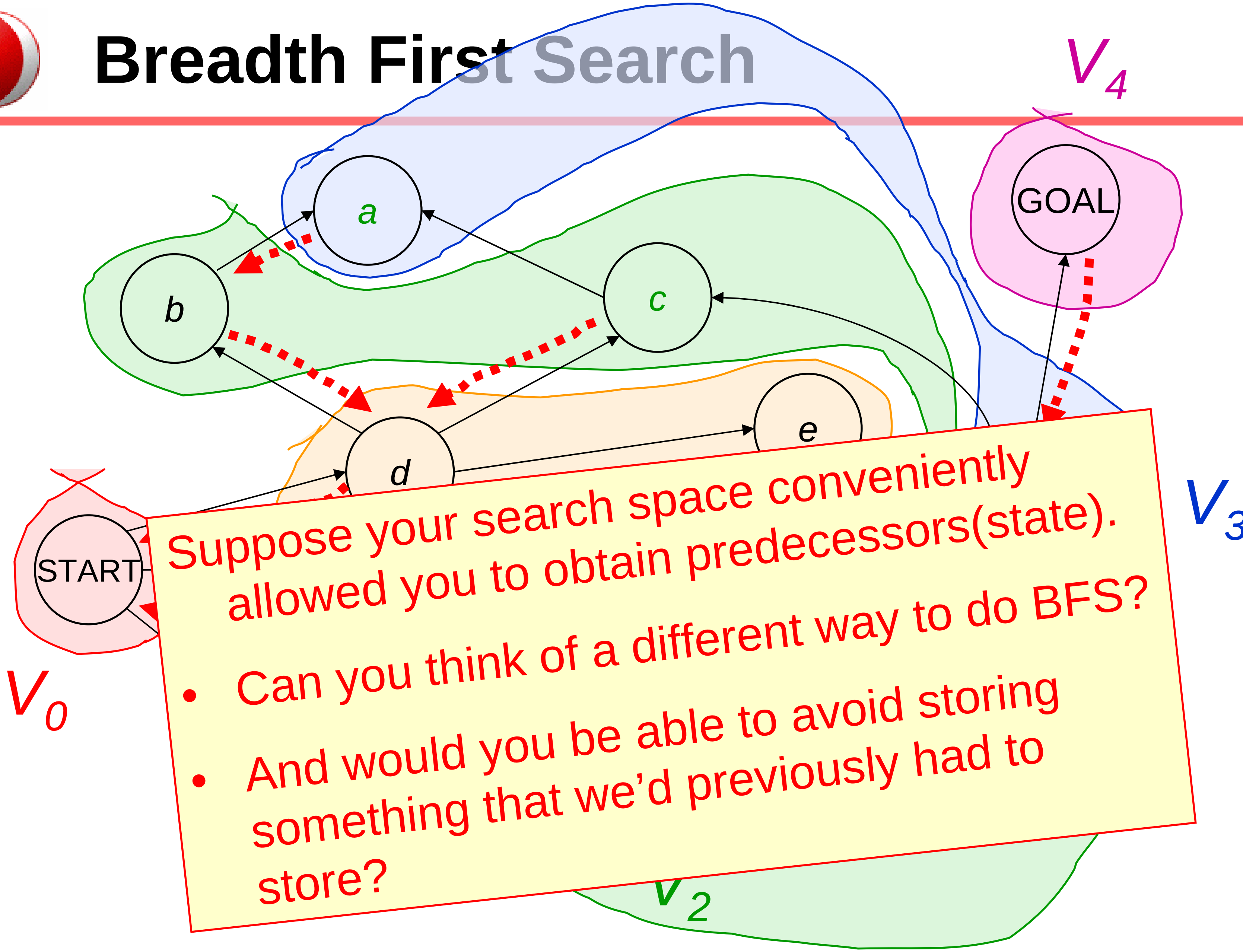
                Add s' into V_{k+1}

    $k := k+1$

If V_k is empty signal FAILURE

Else build the solution path thus: Let S_i be the ith state in the shortest path. Define S_k = GOAL, and forall $i <= k$, define $S_{i-1} = previous(S_i)$.

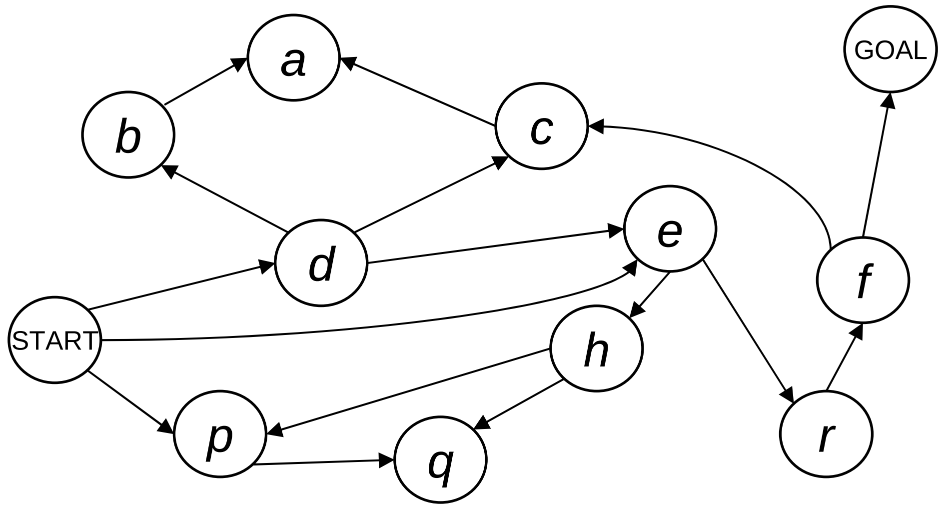
Breadth First Search



Suppose your search space conveniently allowed you to obtain predecessors(state).

- Can you think of a different way to do BFS?
- And would you be able to avoid storing something that we'd previously had to store?

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.

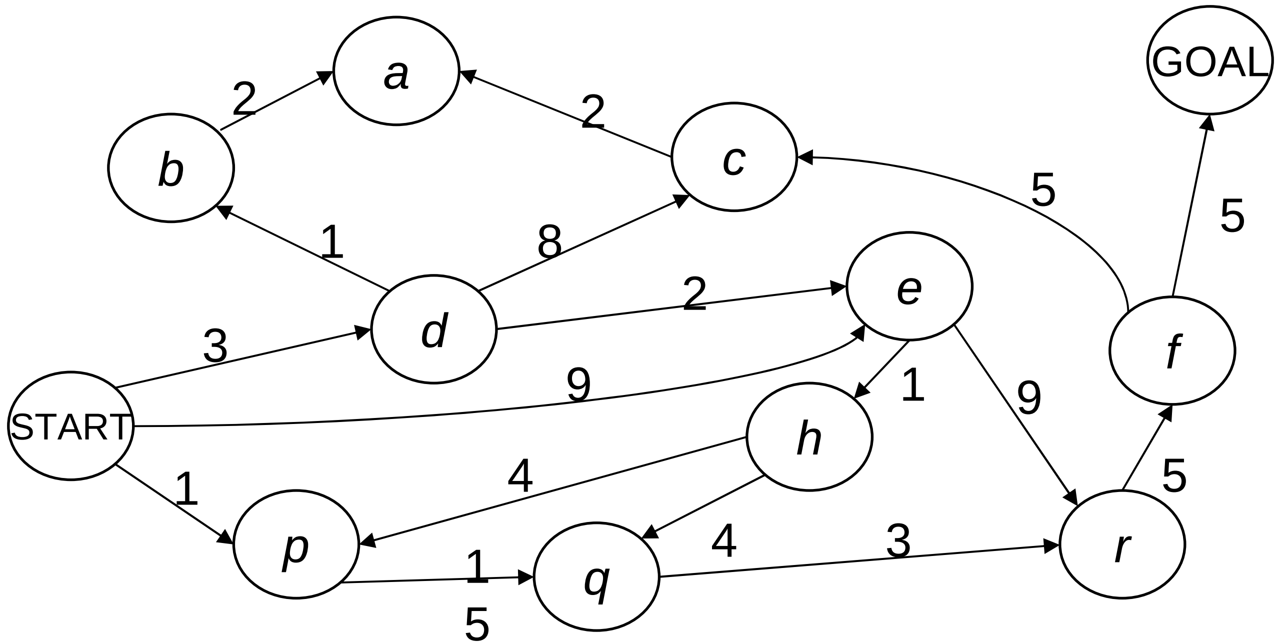
Etc. … until start is reached.

"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. This algorithm is very similar to Dijkstra's algorithm.
- 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.

We will quickly review an algorithm which does find the least-cost path. On the *k*th iteration, for any state *S*, write *g*(*s*) as the least-cost path to *S* in *k* or fewer steps.

Least Cost Breadth First

V_k = the set of states which can be reached in exactly k steps, and for which the least-cost k-step path is less costly than any path of length less than k. In other words, V_k = the set of states whose values changed on the previous iteration.

V_0 := S (the set of start states)

previous(*START*) := *NIL*

g(*START*) = *0*

k := *0*

while (V_k is not empty) **do**

    V_{k+1} := empty set

    For each state *s* in V_k

        For each state *s'* in **succs**(*s*)

            If *s'* has not already been labeled

            OR if *g*(*s*) + *Cost*(*s*,*s'*) < *g*(*s'*)

                Set *previous*(*s'*) := *s*

                Set *g*(*s'*) := *g*(*s*) + *Cost*(*s*,*s'*)

                Add *s'* into V_{k+1}

    k := *k+1*

If GOAL not labeled, exit signaling FAILURE

Else build the solution path thus: Let S_k be the *k*th state in the shortest path. Define S_k = GOAL, and forall $i \le k$, define S_{i-1} = *previous*(S_i).

Uniform Cost Search

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

Priority Queues

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

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

Very cheap (though not absolutely, incredibly cheap!)

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

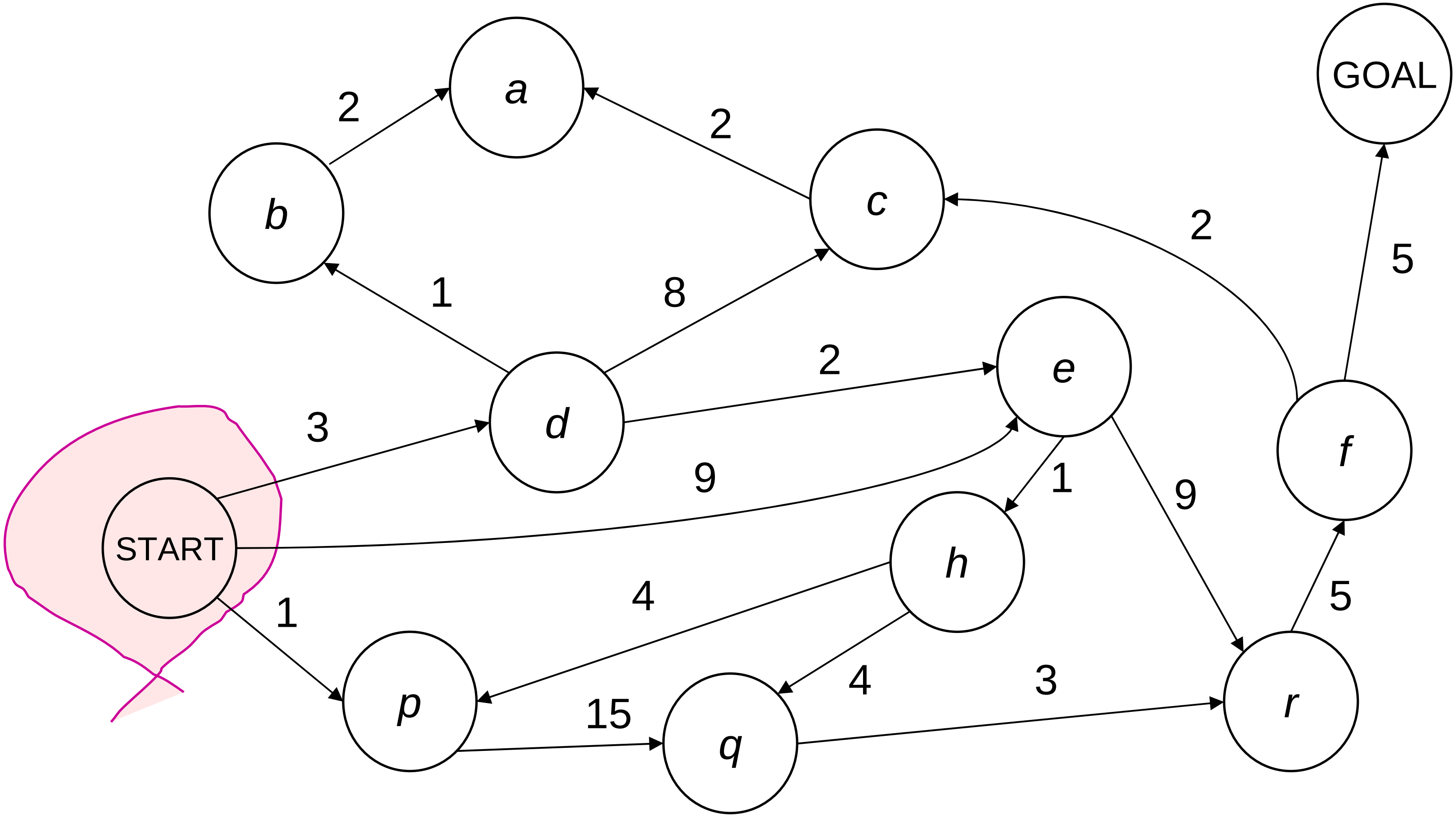
Uniform Cost Search

- 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

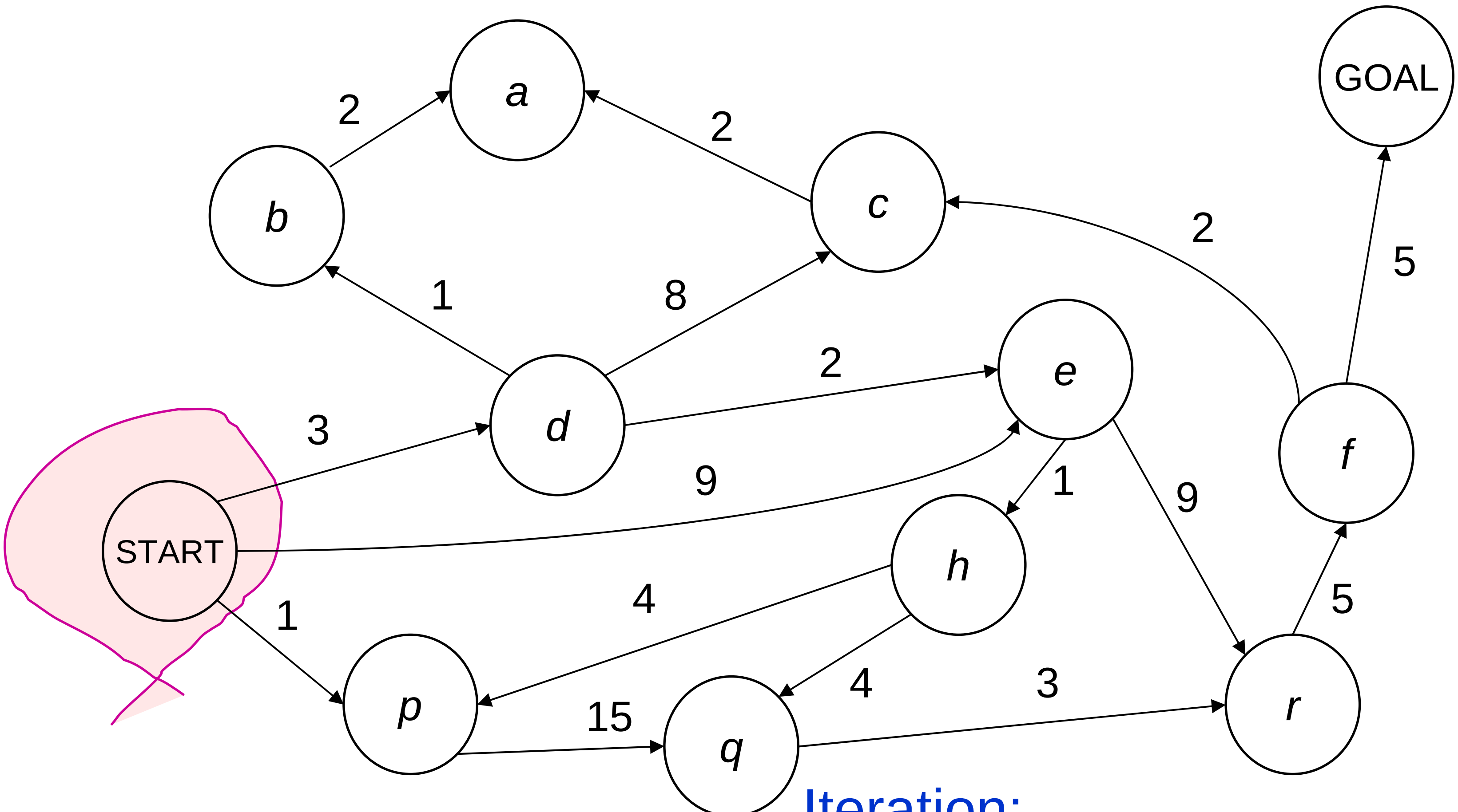
Priority of state *s* = *g(s)* = cost of getting to *s* using path implied by backpointers.

Starting UCS



PQ = { (S,0) }

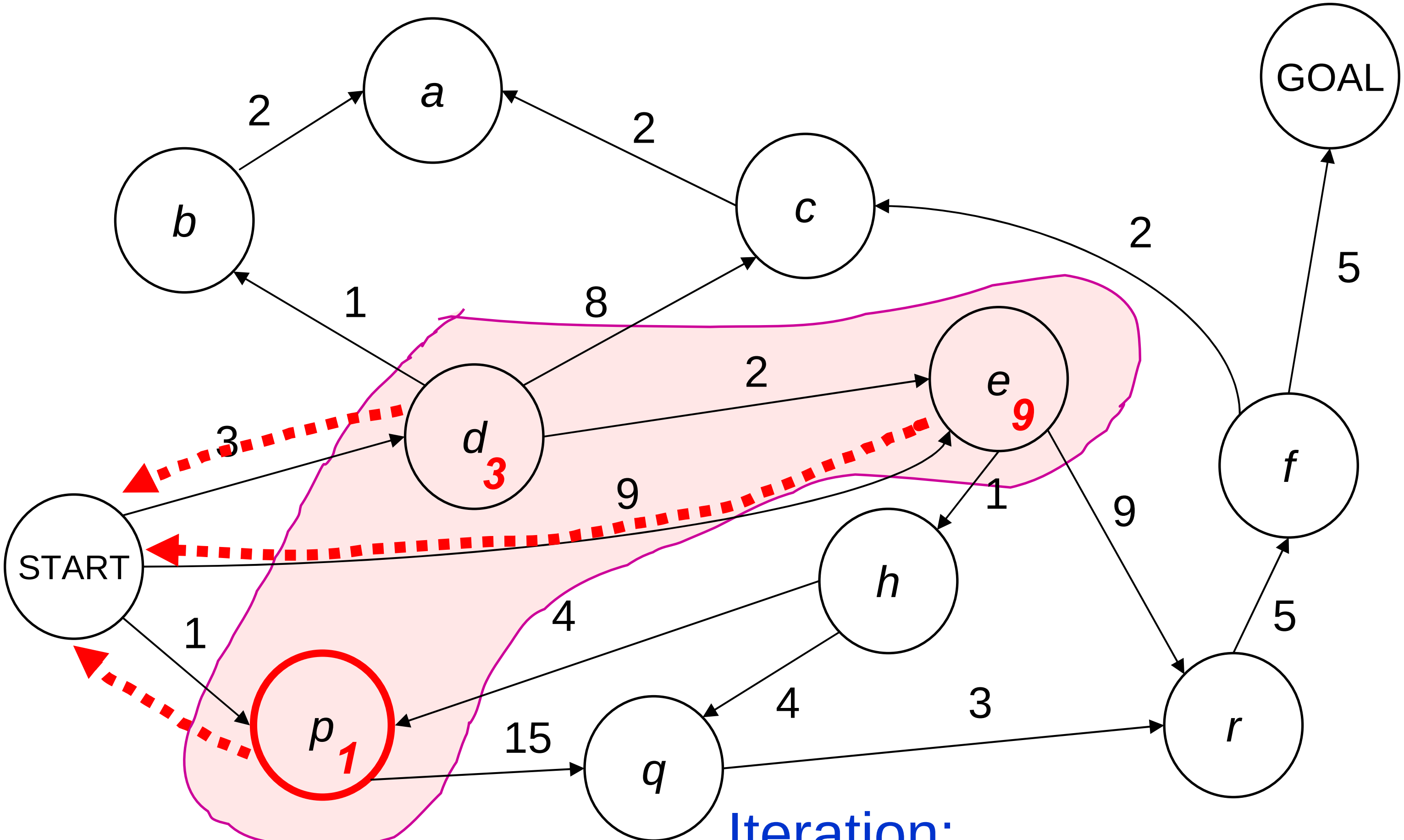
UCS Iterations



PQ = { (S,0) }

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

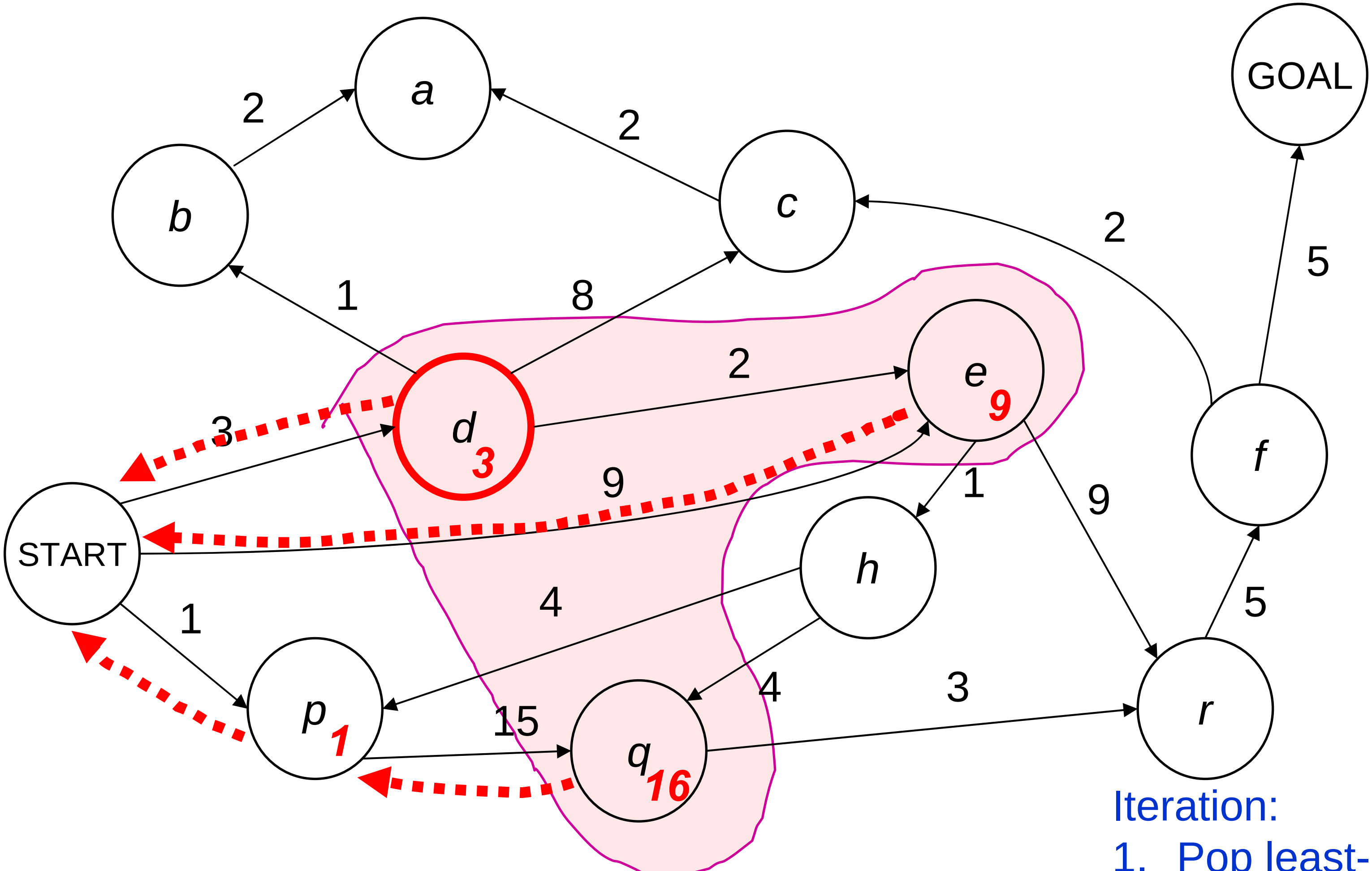
UCS Iterations



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

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

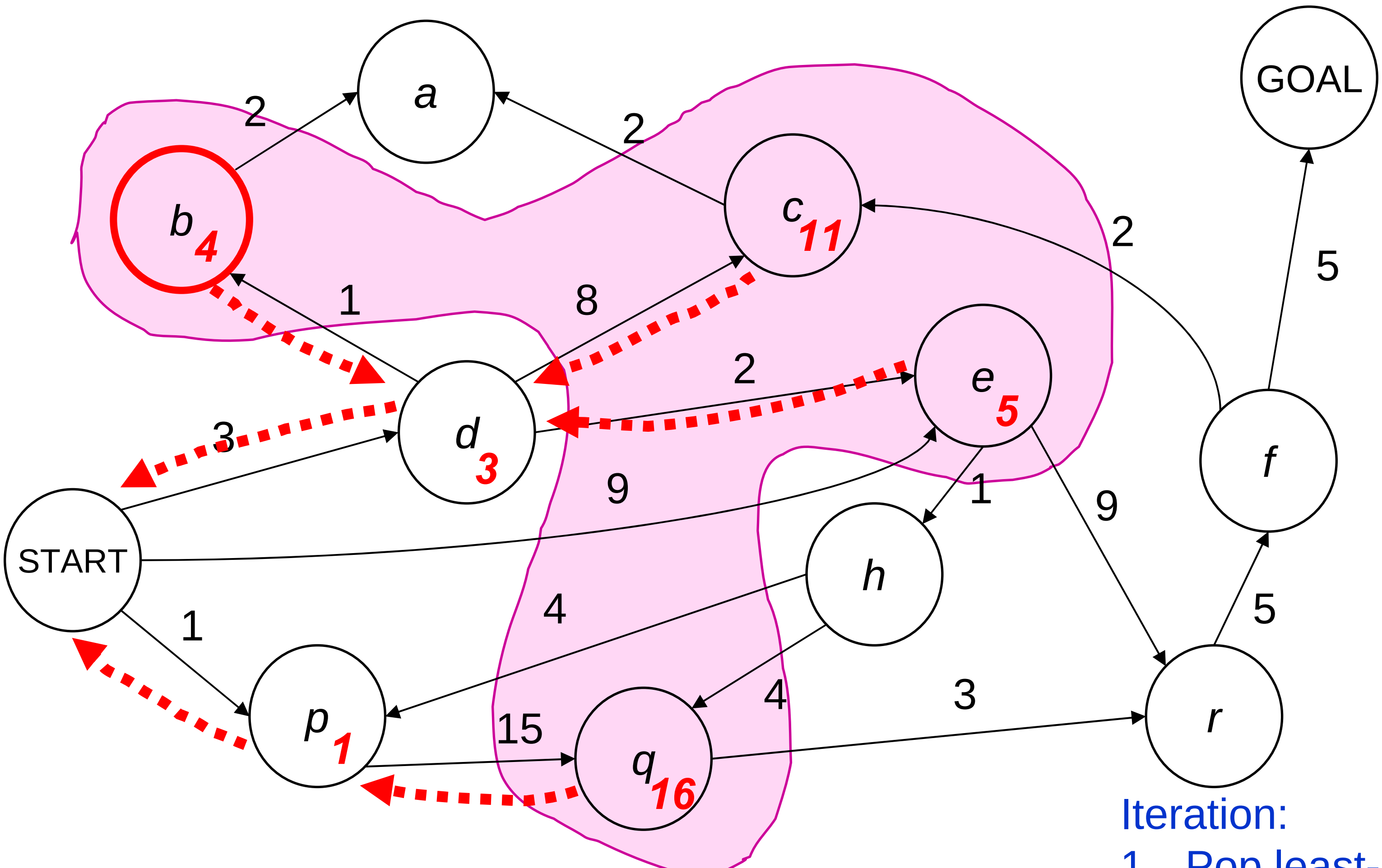
UCS Iterations



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

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

UCS Iterations

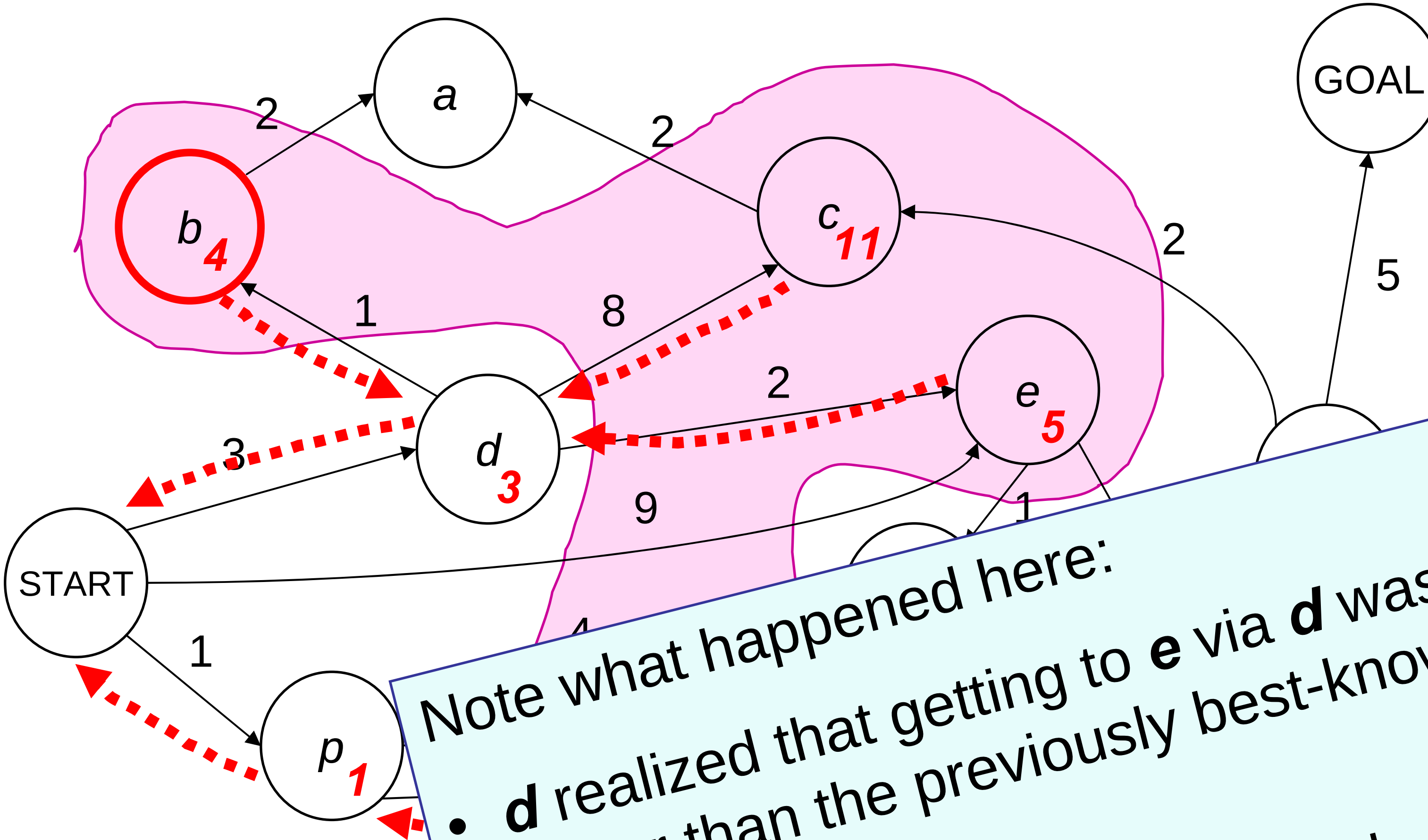


PQ = { (b,4) , (e,5) , (c,11) , (q,16) }

Iteration:

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

UCS Iterations



Note what happened here:

- **d** realized that getting to **e** via **d** was better than the previously best-known way to get to **e**
- and so **e**'s priority was changed

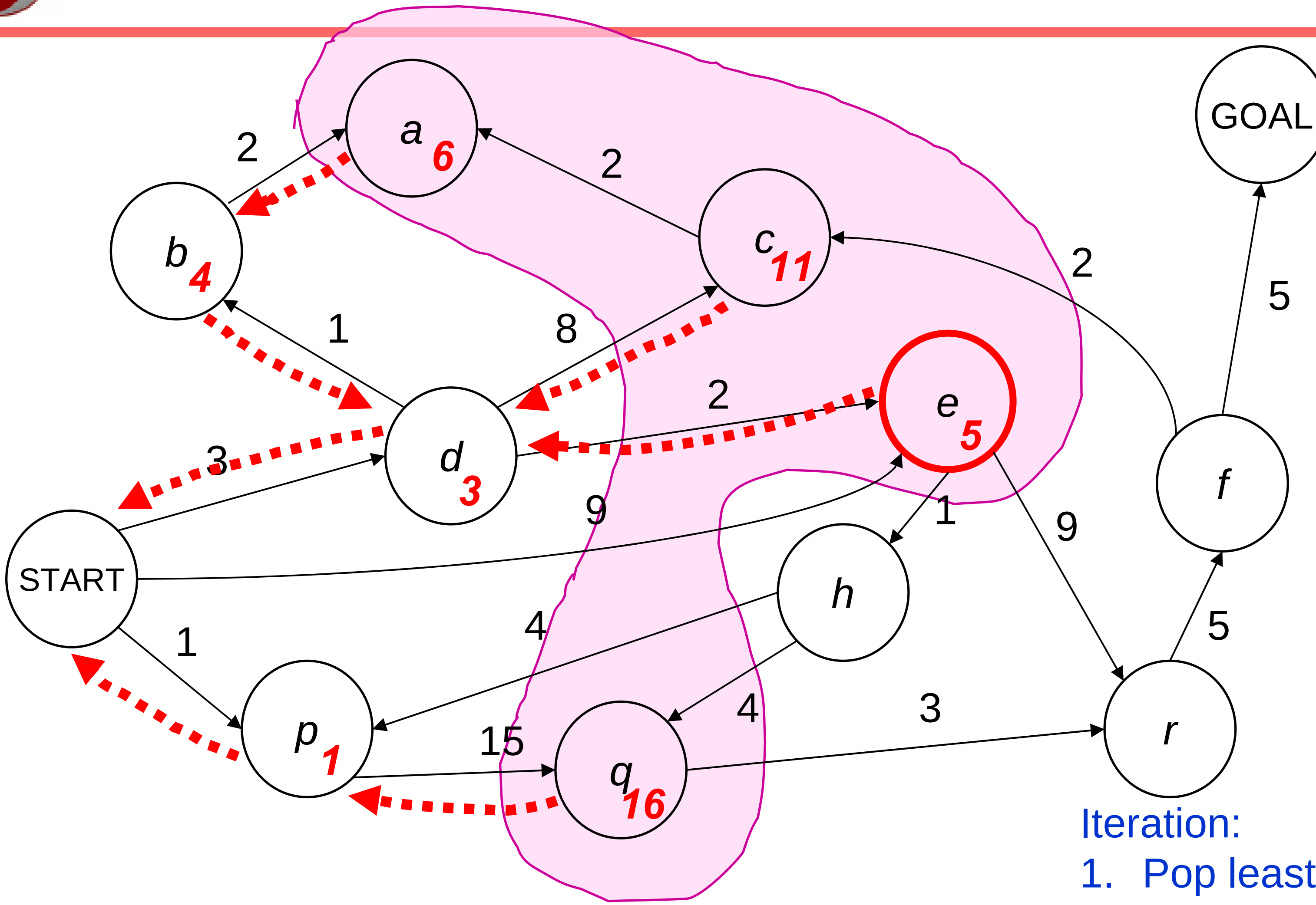
PQ = { (b,4) , (e,5)

) }

least-cost state from PQ

2. Add successors

UCS Iterations

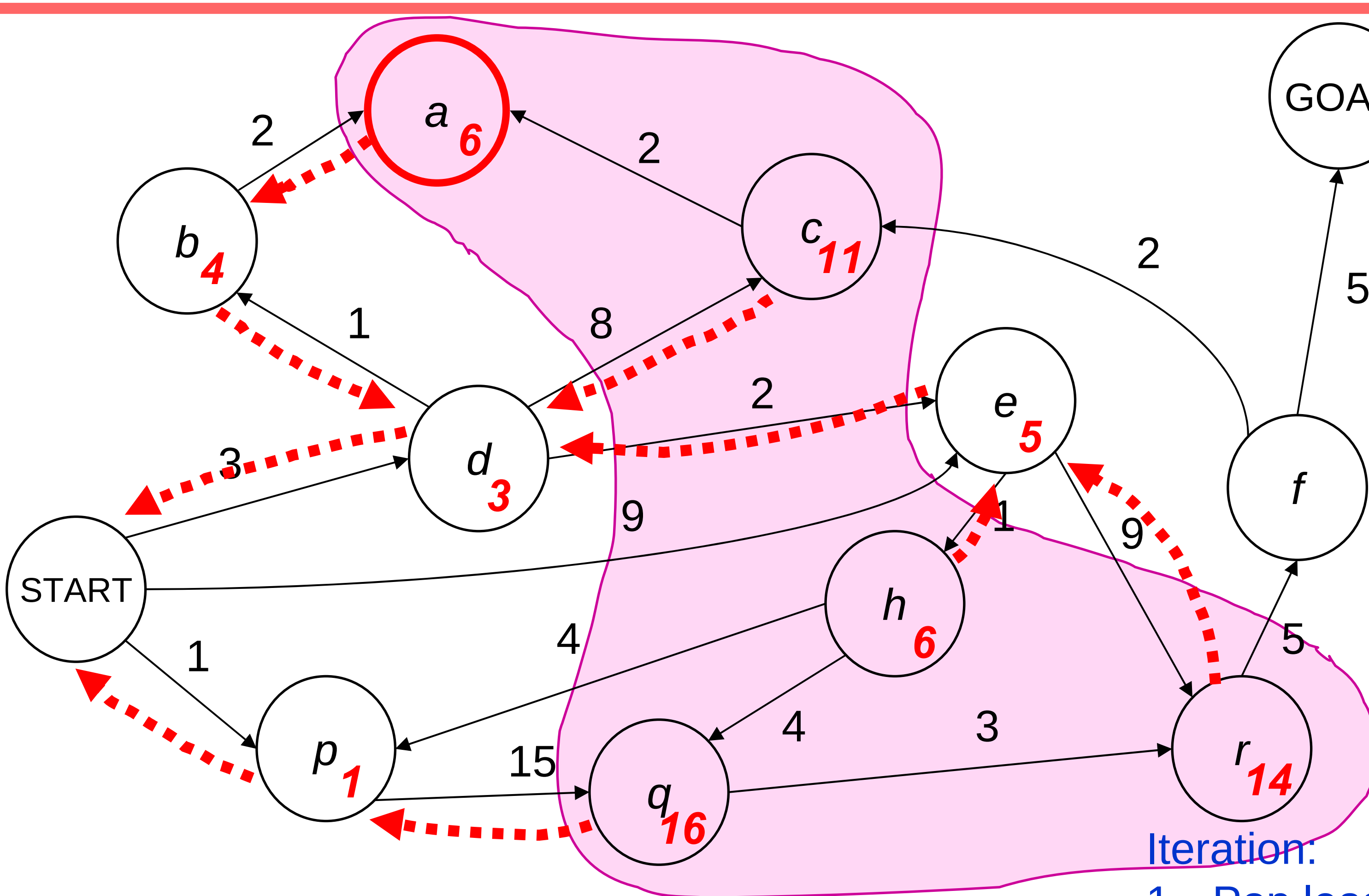


$PQ = \{ (e,5) , (a,6) , (c,11) , (q,16) \}$

Iteration:

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

UCS Iterations

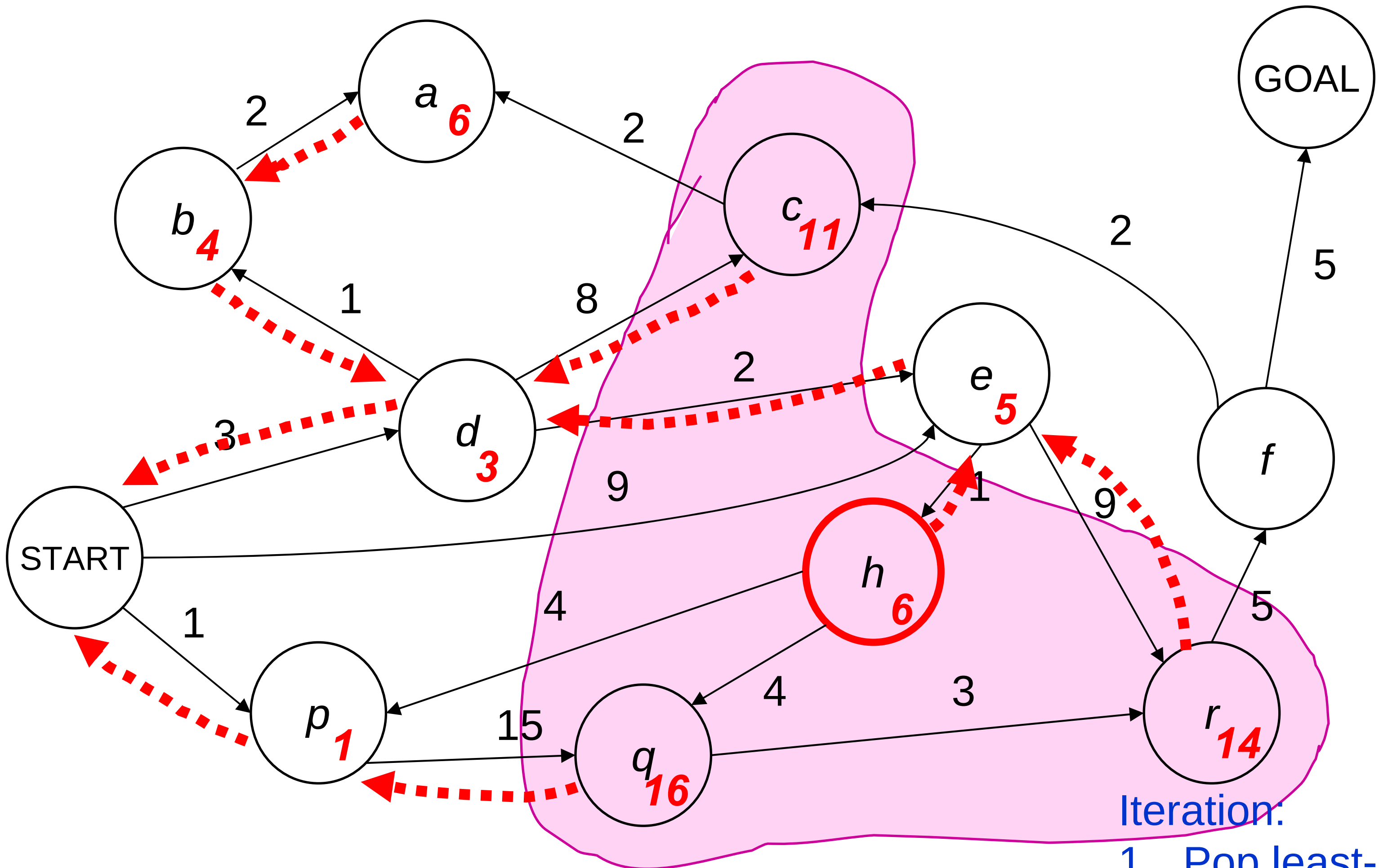


PQ = { (a,6),(h,6),(c,11),(r,14),(q,16) }

Iteration:

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

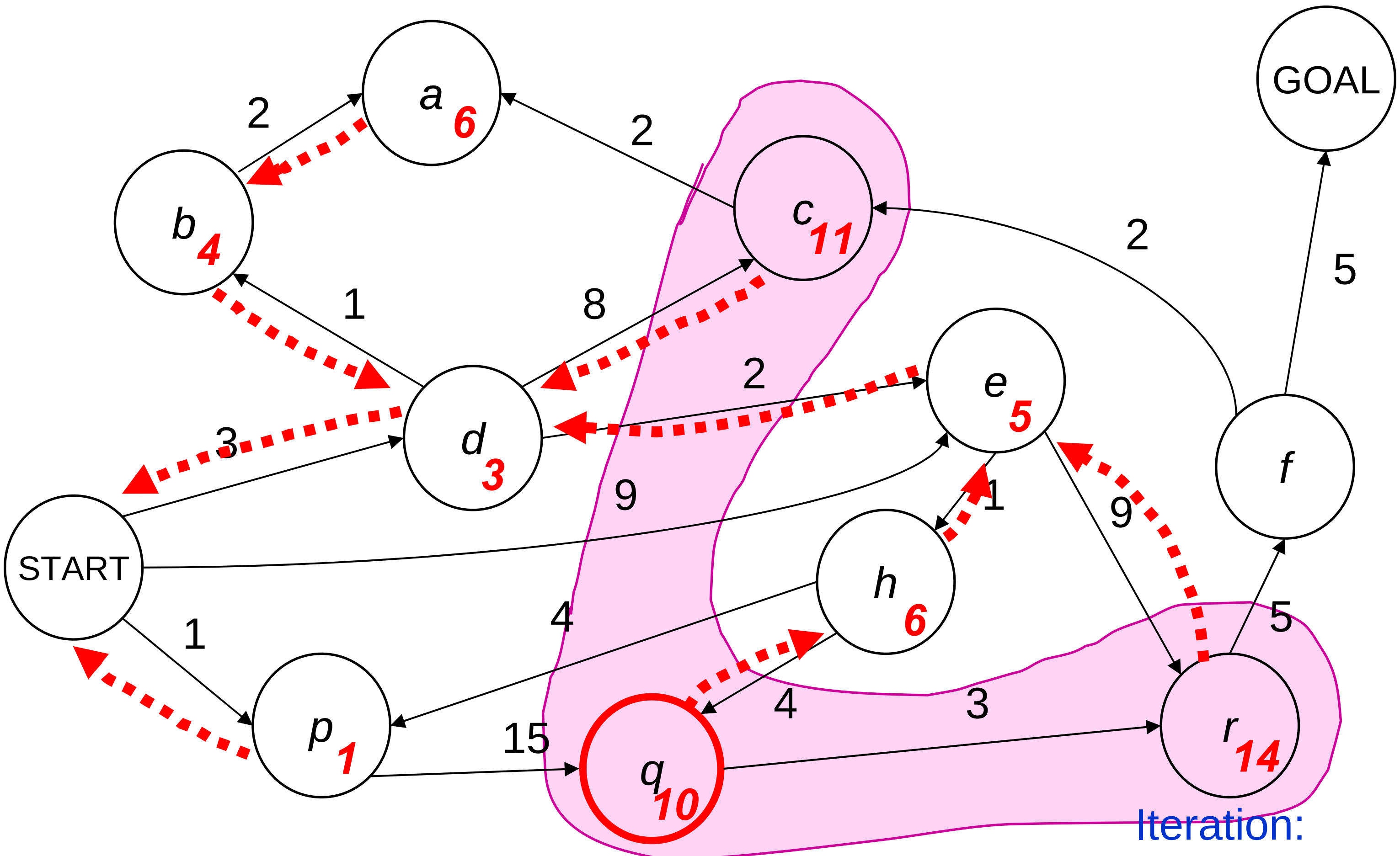
UCS Iterations



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

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

UCS Iterations



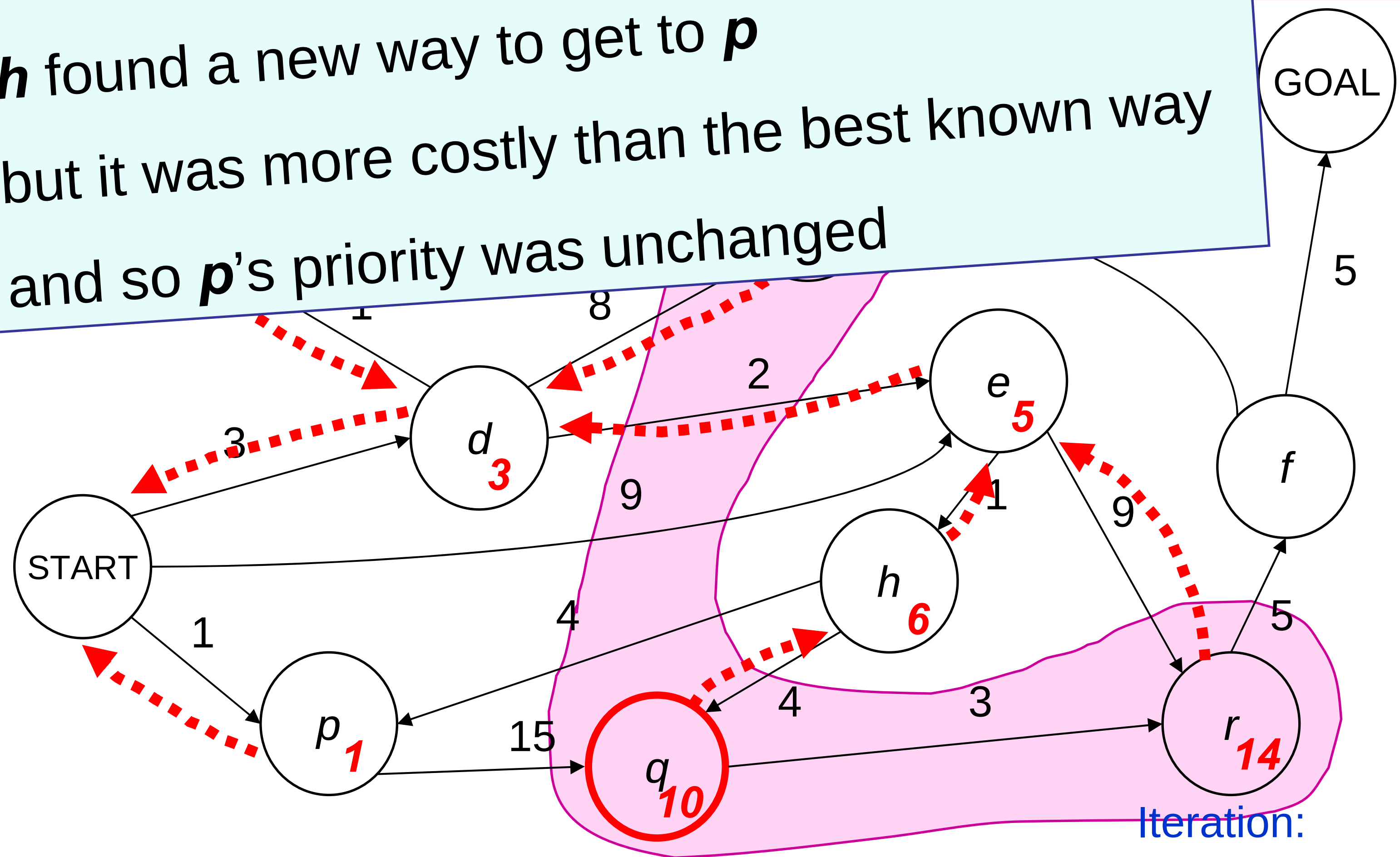
PQ = { (q,10), (c,11),(r,14) }

Iteration:

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

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

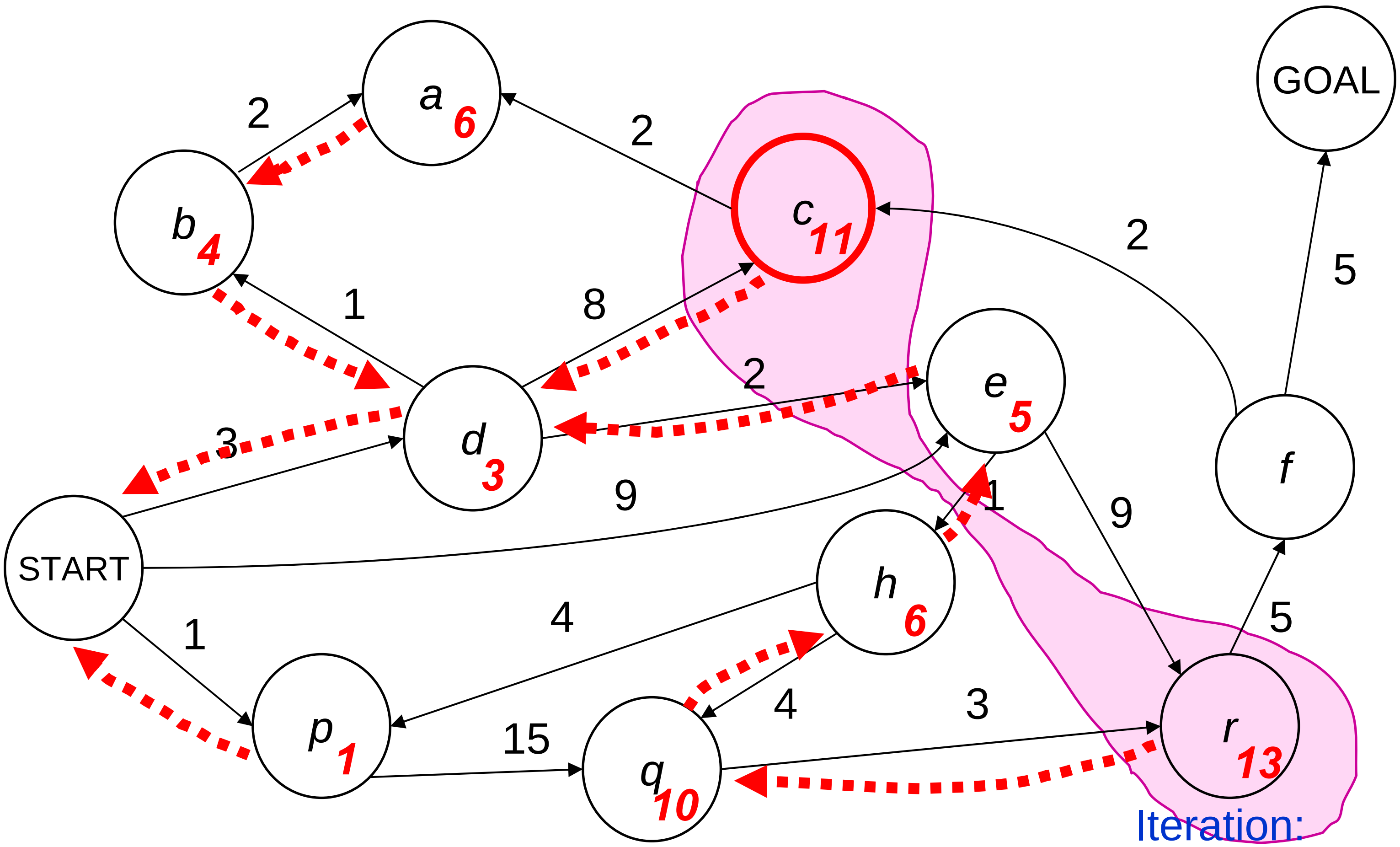


PQ = { (q,10), (c,11),(r,14) }

Iteration:

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

UCS Iterations

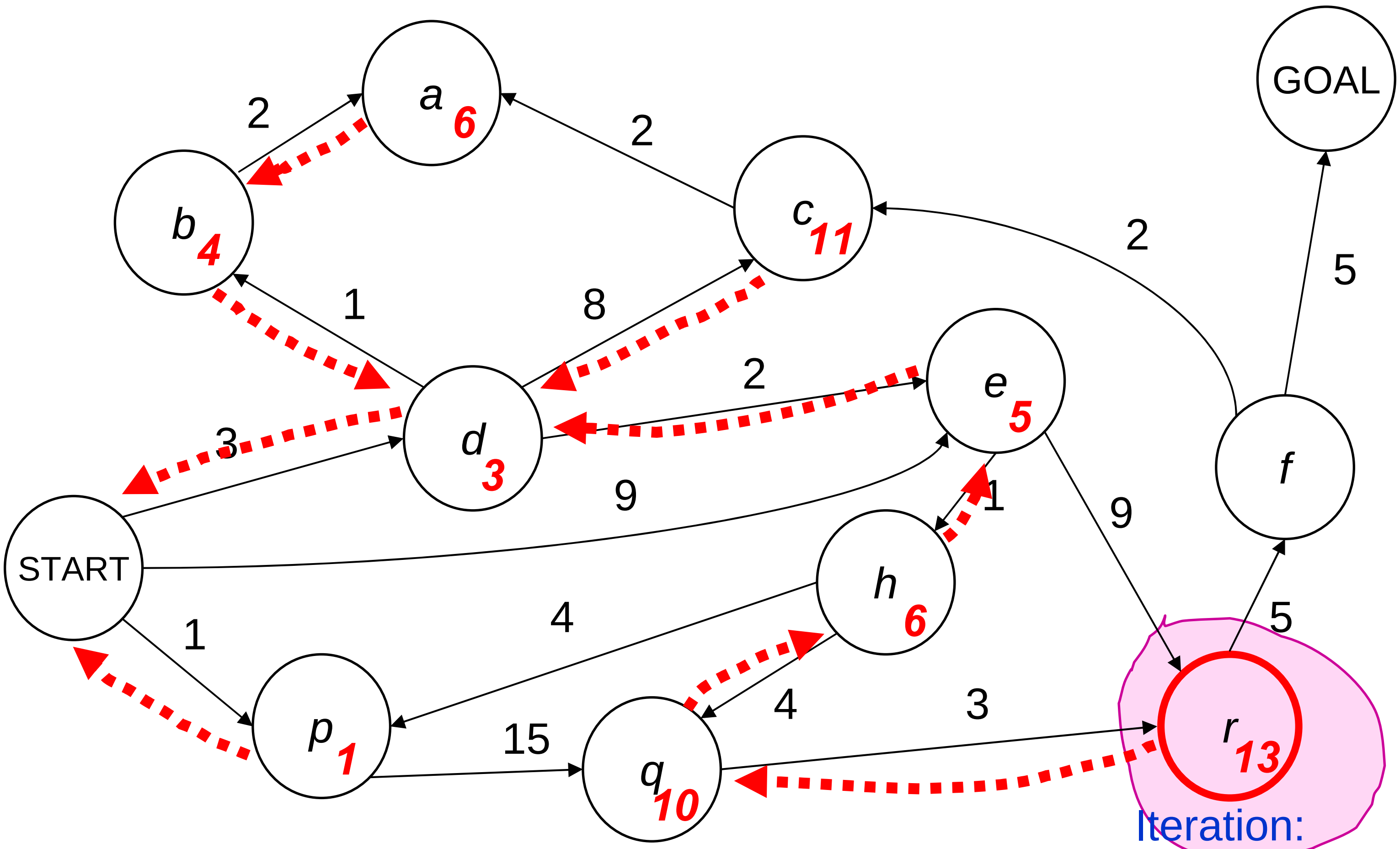


$PQ = \{ (c,11),(r,13) \}$

Iteration:

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

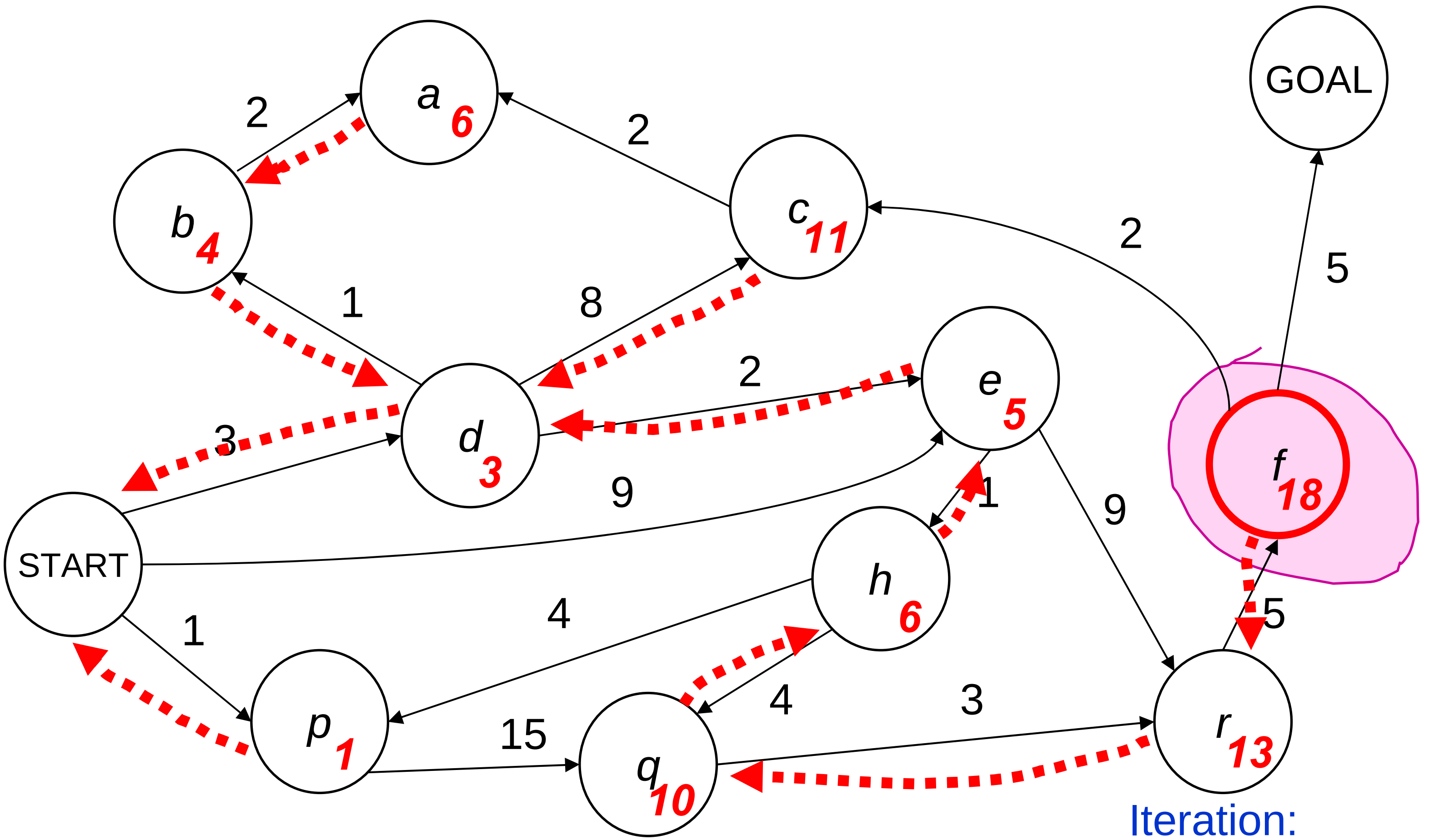
UCS Iterations



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

PQ = { (r,13) }

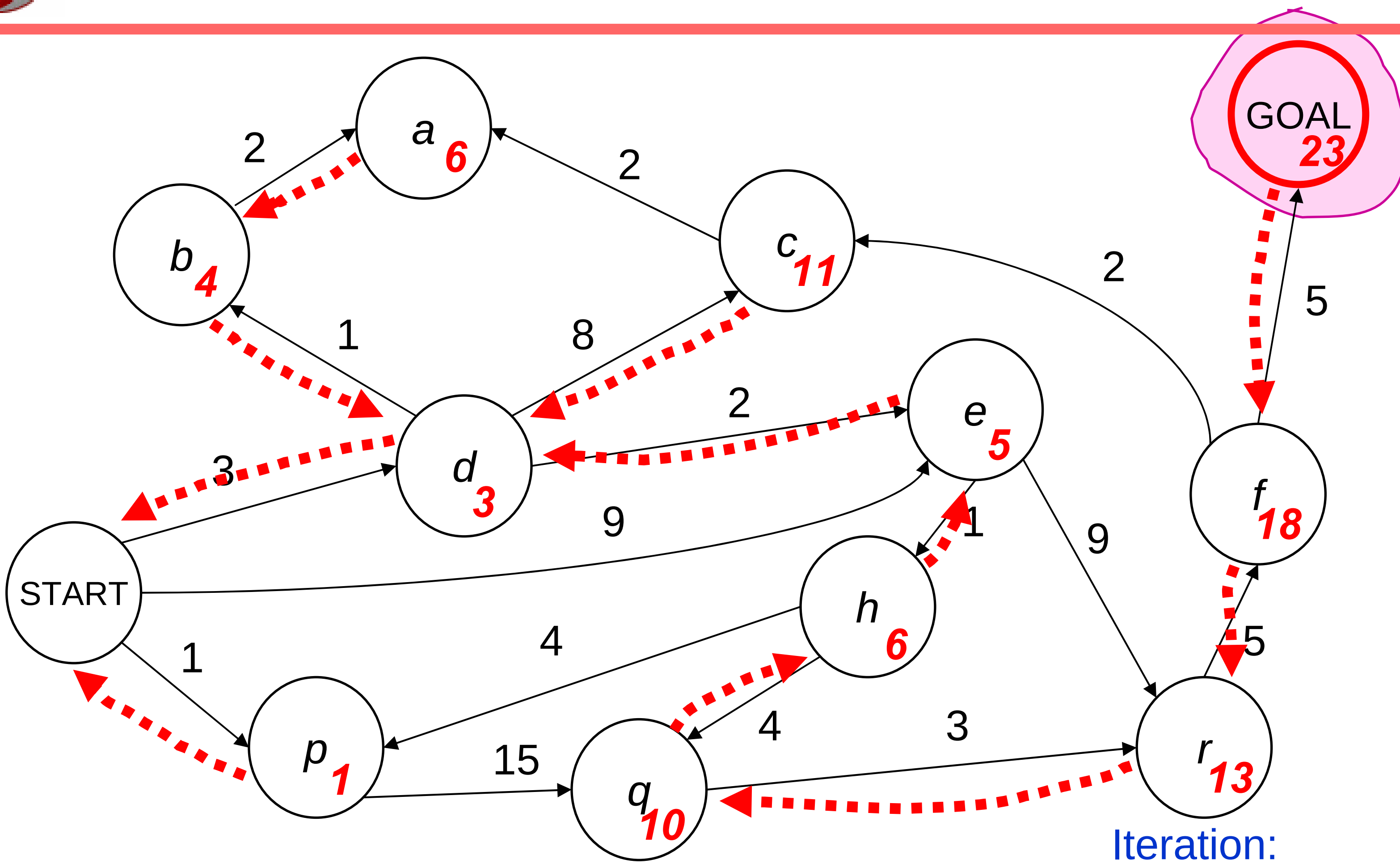
UCS Iterations



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

PQ = { (f,18) }

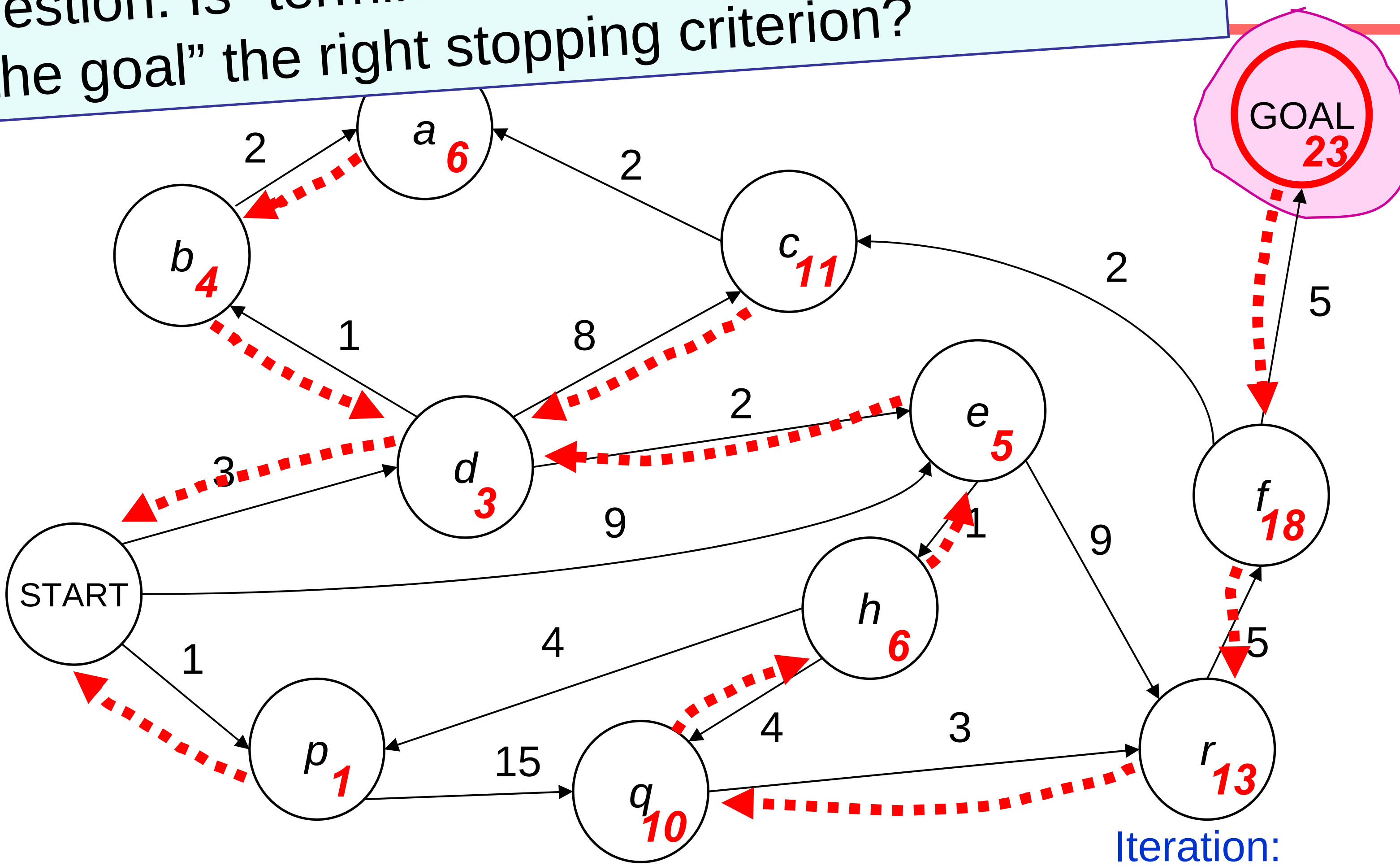
UCS Iterations



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?

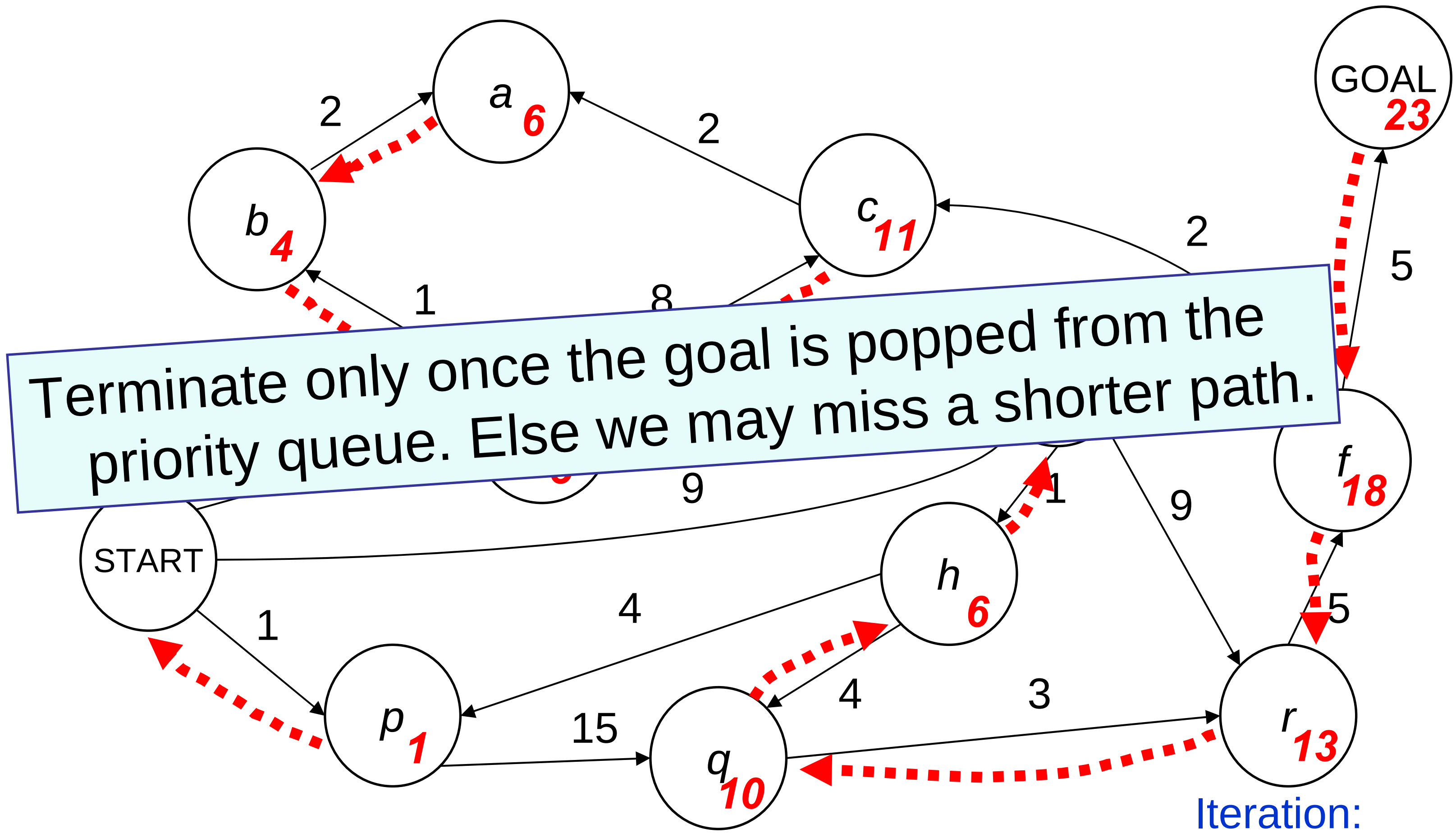


PQ = { (G,23) }

Iteration:

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

UCS Terminates



PQ = { }

Iteration:

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

Judging a Search Algorithm

- Completeness: is the algorithm guaranteed to find a solution if a solution exists?
- Guaranteed to find optimal? (will it find the least cost path?)
- Algorithmic time complexity
- Space complexity (memory use)

Variables:

N	number of states in the problem
B	the average branching factor (the average number of successors) (B>1)
L	the length of the path from start to goal with the shortest number of steps

How would we judge our algorithms?

Judging a Search Algorithm

N	number of states in the problem
B	the average branching factor (the average number of successors) (*B*>1)
L	the length of the path from start to goal with the shortest number of steps
Q	the average size of the priority queue

Algorithm		Complete	Optimal	Time	Space
BFS	Breadth First Search				
LCBFS	Least Cost BFS				
UCS	Uniform Cost Search				

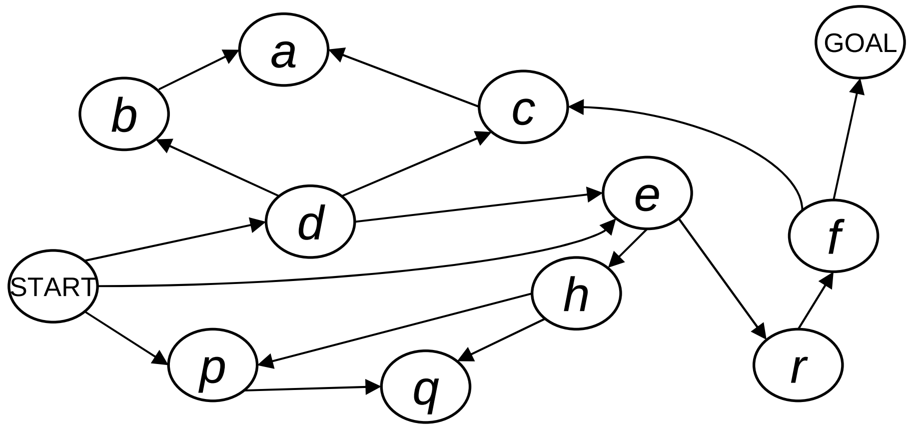
Judging a Search Algorithm

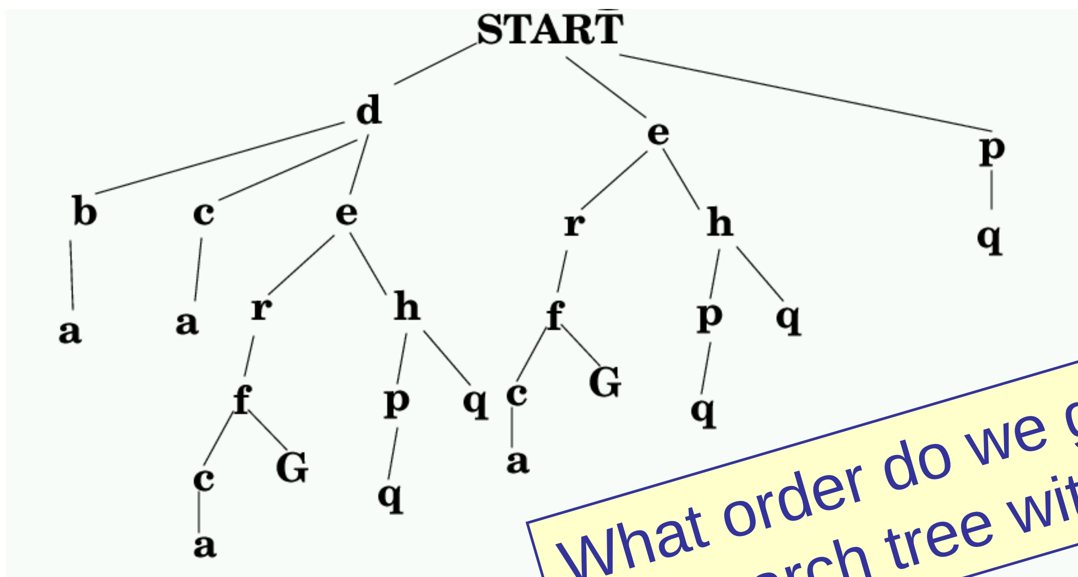
N	number of states in the problem
B	the average branching factor (the average number of successors) (*B*>1)
L	the length of the path from start to goal with the shortest number of steps
Q	the average size of the priority queue

Algorithm		Complete	Optimal	Time	Space
BFS	Breadth First Search	Y	if all transitions same cost	$O(min(N,B^L))$	$O(min(N,B^L))$
LCBFS	Least Cost BFS	Y	Y	$O(min(N,B^L))$	$O(min(N,B^L))$
UCS	Uniform Cost Search	Y	Y	$O(log(Q) * min(N,B^L))$	$O(min(N,B^L))$

Grad course:
"Algorithms and Computational Complexity"

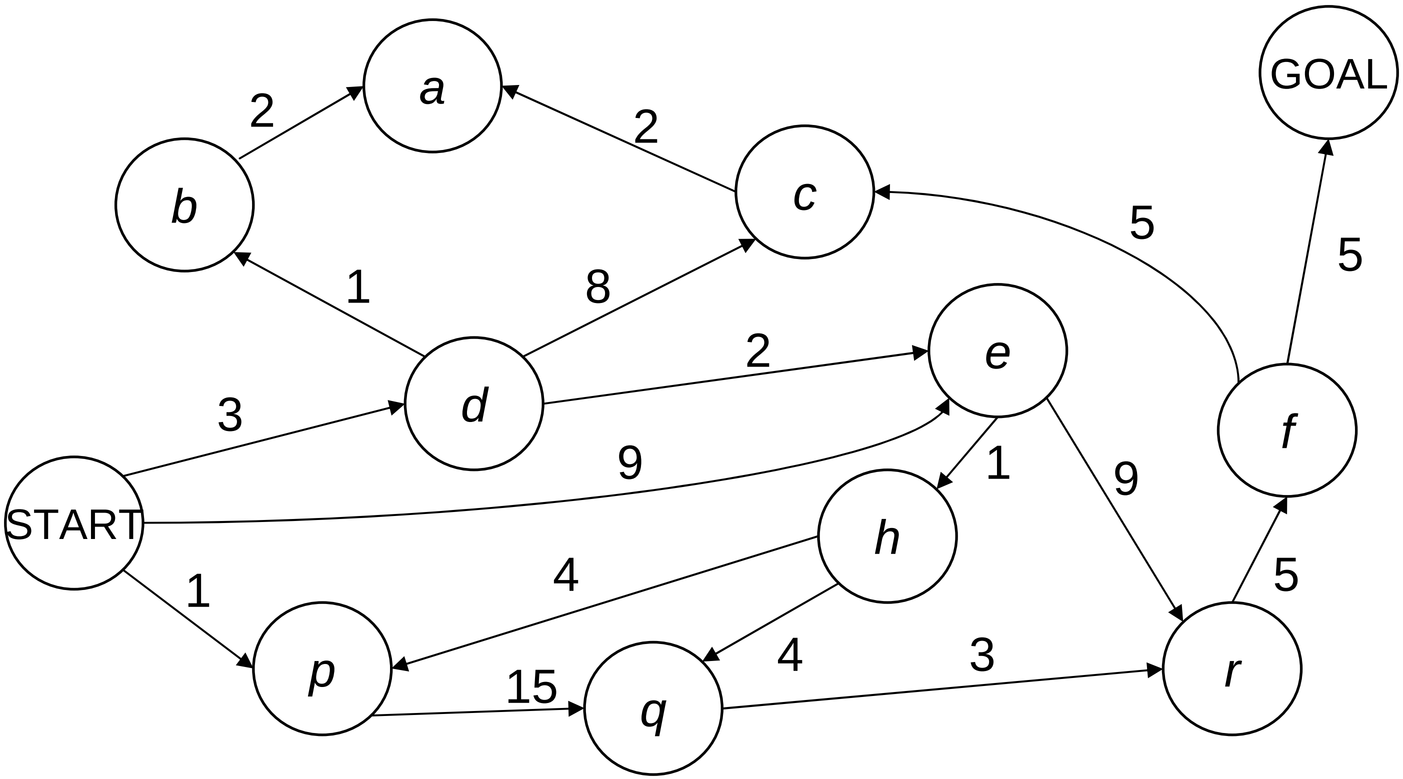
Search Tree Representation





What order do we go through the search tree with BFS?

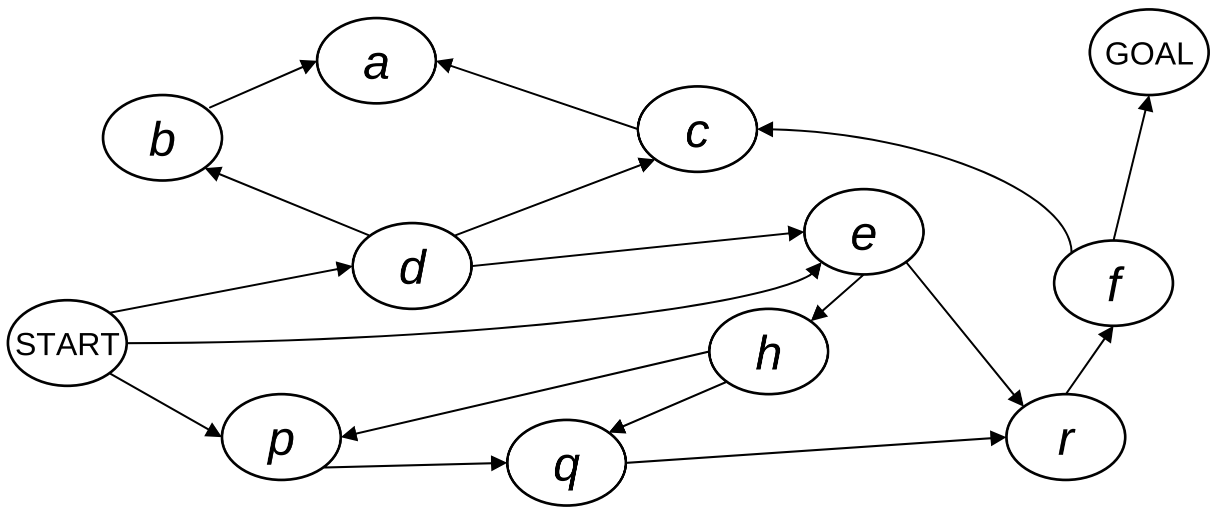
Depth First Search



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.

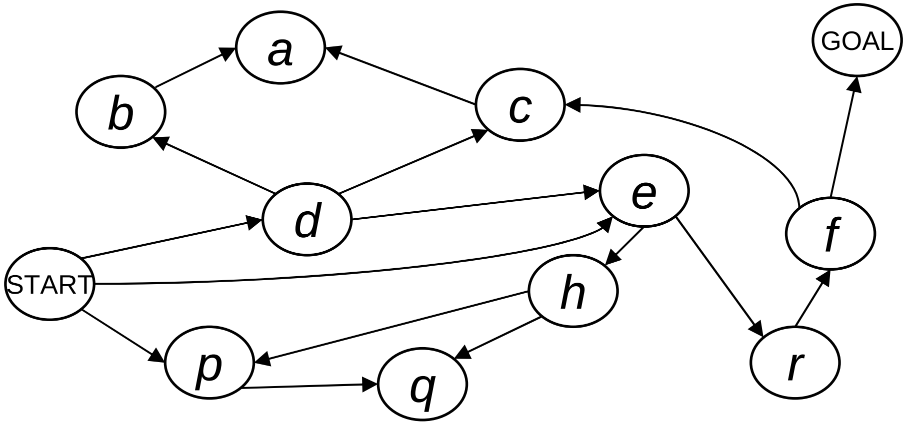
DFS in Action

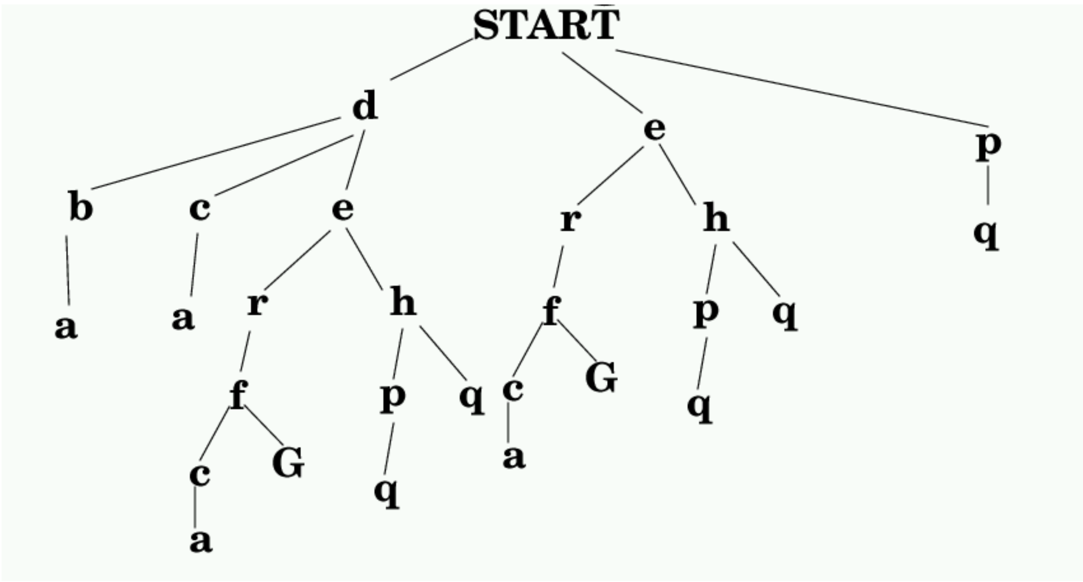
START
START *d*
START *d b*
START *d b a*
START *d c*
START *d c a*
START *d e*
START *d e r*
START *d e r f*
START *d e r f c*
START *d e r f c a*
START *d e r f* GOAL



DFS Search Tree Traversal

Can you draw in the order in which the search-tree nodes are visited?



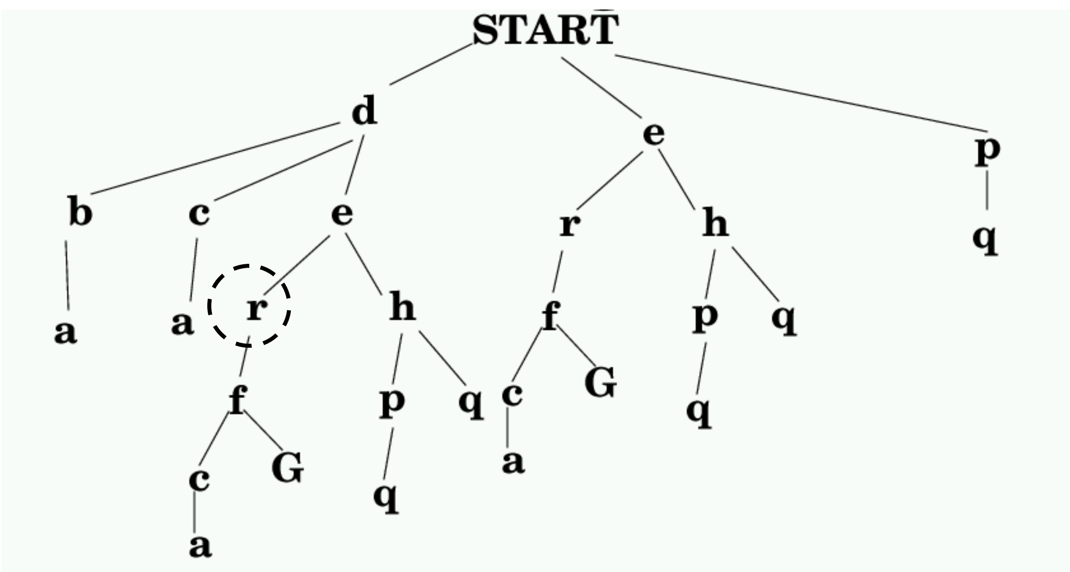


DFS Algorithm

We use a data structure we'll call a **Path** to represent the path from the START to the current state.

E.G. Path P = <START, d, e, r >

Along with each node on the path, we must remember which successors we still have available to expand. E.G. at the following point, we'll have



P = <START (expand=e , p) ,

d (expand = NULL) ,

e (expand = h) ,

r (expand = f) >

DFS Algorithm

```
Let P = <START (expand = succs(START))>
While (P not empty and top(P) not a goal)
        if expand of top(P) is empty
        then
                remove top(P) ("pop the stack")
        else
                let s be a member of expand of top(P)
                remove s from expand of top(P)
                make a new item on the top of path P:
                        s (expand = succs(s))
If P is empty
        return FAILURE
Else
        return the path consisting of states in P
```

This algorithm can be written neatly with recursion, i.e. using the program stack to implement P.

Judging a Search Algorithm

N	number of states in the problem
B	the average branching factor (the average number of successors) (B>1)
L	the length of the path from start to goal with the shortest number of steps
Q	the average size of the priority queue

Algorithm		Complete	Optimal	Time	Space
BFS	Breadth First Search	Y	if all transitions same cost	$O(min(N,B^L))$	$O(min(N,B^L))$
LCBFS	Least Cost BFS	Y	Y	$O(min(N,B^L))$	$O(min(N,B^L))$
UCS	Uniform Cost Search	Y	Y	$O(log(Q) * min(N,B^L))$	$O(min(N,B^L))$
DFS	Depth First Search				

Judging a Search Algorithm

N	number of states in the problem
B	the average branching factor (the average number of successors) (*B*>1)
L	the length of the path from start to goal with the shortest number of steps
Q	the average size of the priority queue

Algorithm		Complete	Optimal	Time	Space
BFS	Breadth First Search	Y	if all transitions same cost	$O(min(N,B^L))$	$O(min(N,B^L))$
LCBFS	Least Cost BFS	Y	Y	$O(min(N,B^L))$	$O(min(N,B^L))$
UCS	Uniform Cost Search	Y	Y	$O(log(Q) * min(N,B^L))$	$O(min(N,B^L))$
DFS	Depth First Search	N	N	*N/A*	*N/A*

Judging a Search Algorithm

N	number of states in the problem
B	the average branching factor (the average number of successors) (*B*>1)
L	the length of the path from start to goal with the shortest number of steps
Q	the average size of the priority queue

Algorithm		Complete	Optimal	Time	Space
BFS	Breadth First Search	Y	if all transitions same cost	$O(min(N,B^L))$	$O(min(N,B^L))$
LCBFS	Least Cost BFS	Y	Y	$O(min(N,B^L))$	$O(min(N,B^L))$
UCS	Uniform Cost Search	Y	Y	$O(log(Q) * min(N,B^L))$	$O(min(N,B^L))$
DFS**	Depth First Search				

Assuming Acyclic Search Space

Judging a Search Algorithm

N	number of states in the problem
B	the average branching factor (the average number of successors) (*B*>1)
L	the length of the path from start to goal with the shortest number of steps
LMAX	Length of longest path from start to anywhere
Q	the average size of the priority queue

Algorithm		Complete	Optimal	Time	Space
BFS	Breadth First Search	Y	if all transitions same cost	$O(min(N,B^L))$	$O(min(N,B^L))$
LCBFS	Least Cost BFS	Y	Y	$O(min(N,B^L))$	$O(min(N,B^L))$
UCS	Uniform Cost Search	Y	Y	$O(log(Q) * min(N,B^L))$	$O(min(N,B^L))$
DFS**	Depth First Search	Y	N	$O(B^{LMAX})$	$O(LMAX)$

Assuming Acyclic Search Space

Questions to Ponder

- How would you prevent DFS from looping?

- How could you force it to give an optimal solution?

Questions to P

- How would you prevent DFS from looping?

- How could you force it to give an optimal solution?

Answer 1:

PC-DFS (Path Checking DFS):

Answer 2:

MEMDFS (Memoizing DFS):

Questions to P

- How would you prevent DFS from looping?

- How could you force it to give an optimal solution?

Answer 1:

PC-DFS (Path Checking DFS):

Don't recurse on a state if that state is already in the current path

Answer 2:

MEMDFS (Memorizing DFS):

Remember all states expanded so far. Never expand anything twice.

Questions to P

- How would you prevent DF... looping...
- How could you fix... to give an ...imal solution?

Answer 1:

PC-DFS (Path Checking DFS):

Don't recurse on a state if that state is already in the current path

Answer 2:

MEMDFS (Memoizing DFS):

Remember all states expanded so far. Never expand anything twice.

Are there occasions when PCDFS is better than MEMDFS?

Are there occasions when MEMDFS is better than PCDFS?

Judging a Search Algorithm

N	number of states in the problem
B	the average branching factor (the average number of successors) (*B*>1)
L	the length of the path from start to goal with the shortest number of steps
LMAX	Length of longest **cycle-free** path from start to anywhere
Q	the average size of the priority queue

Algorithm		Complete	Optimal	Time	Space
BFS	Breadth First Search	Y	if all transitions same cost	$O(min(N,B^L))$	$O(min(N,B^L))$
LCBFS	Least Cost BFS	Y	Y	$O(min(N,B^L))$	$O(min(N,B^L))$
UCS	Uniform Cost Search	Y	Y	$O(log(Q) * min(N,B^L))$	$O(min(N,B^L))$
PCDFS	Path Check DFS				
MEMDFS	Memoizing DFS				

Judging a Search Algorithm

N	number of states in the problem
B	the average branching factor (the average number of successors) (*B*>1)
L	the length of the path from start to goal with the shortest number of steps
LMAX	Length of longest **cycle-free** path from start to anywhere
Q	the average size of the priority queue

Algorithm		Complete	Optimal	Time	Space
BFS	Breadth First Search	Y	if all transitions same cost	$O(min(N,B^L))$	$O(min(N,B^L))$
LCBFS	Least Cost BFS	Y	Y	$O(min(N,B^L))$	$O(min(N,B^L))$
UCS	Uniform Cost Search	Y	Y	$O(log(Q) * min(N,B^L))$	$O(min(N,B^L))$
PCDFS	Path Check DFS	Y	N	$O(B^{LMAX})$	$O(LMAX)$
MEMDFS	Memoizing DFS	Y	N	$O(min(N,B^{LMAX}))$	$O(min(N,B^{LMAX}))$

Judging a Search Algorithm

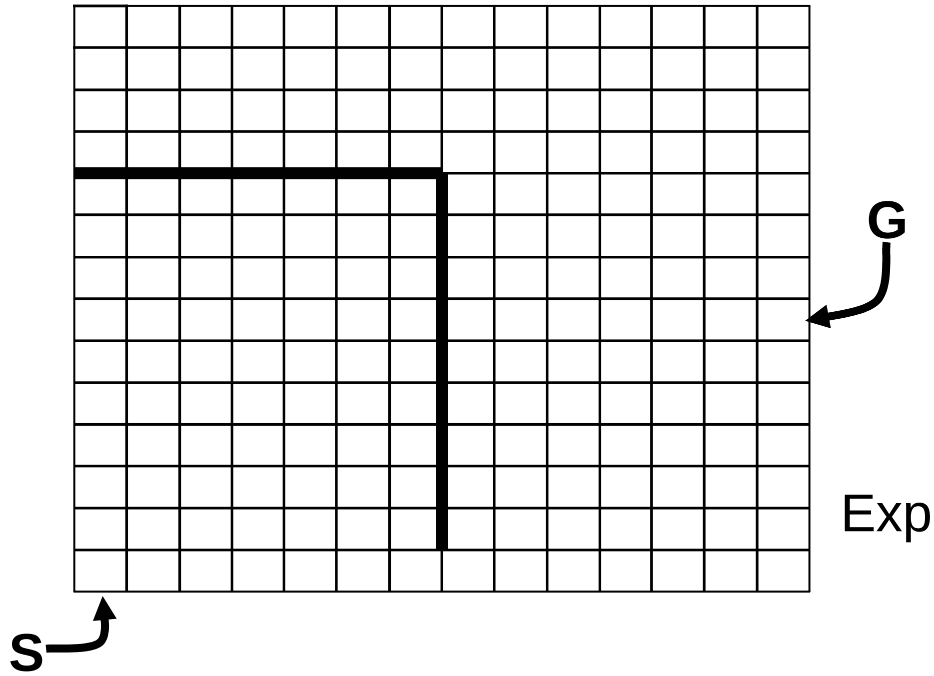
N	number of states in the problem
B	the average branching factor (the average number of successors) (*B*>1)
L	the length of the path from start to goal with the shortest number of steps
LMAX	Length of longest **cycle-free** path from start to anywhere
Q	the average size of the priority queue

Algorithm		Complete	Optimal	Time	Space
BFS	Breadth First Search	Y	if all transitions same cost	$O(min(N,B^L))$	$O(min(N,B^L))$
LCBFS	Least Cost BFS	Y	Y	$O(min(N,B^L))$	$O(min(N,B^L))$
UCS	Uniform Cost Search	Y	Y	$O(log(Q) * min(N,B^L))$	$O(min(N,B^L))$
PCDFS	Path Check DFS	Y	N	$O(B^{LMAX})$	$O(LMAX)$
MEMDFS	Memoizing DFS	Y	N	$O(min(N,B^{LMAX}))$	$O(min(N,B^{LMAX}))$

Why?

Maze Example

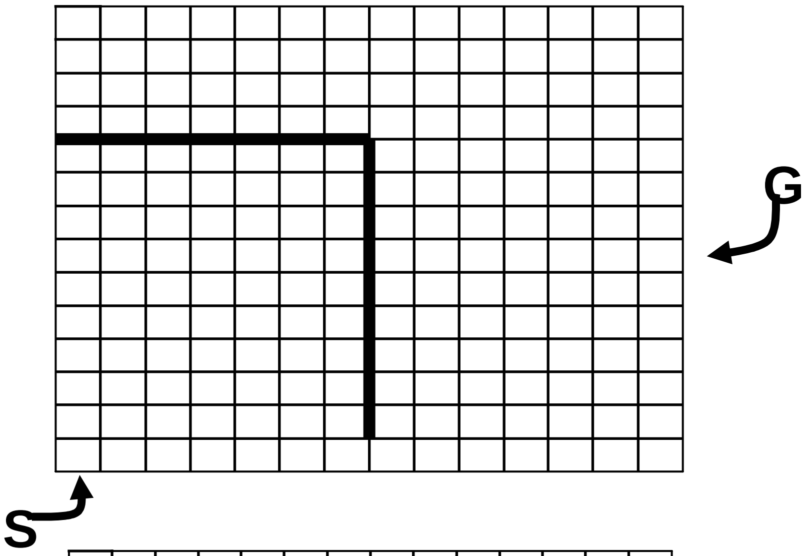
Imagine states are cells in a maze, you can move N, E, S, W. What would plain DFS do, assuming it always expanded the E successor first, then N, then W, then S?



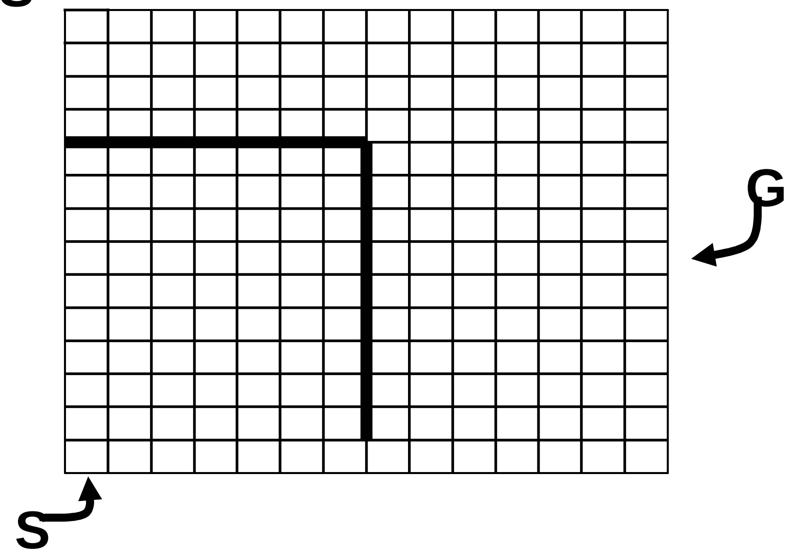
Expansion order E, N, W, S

Other questions:
- What would BFS do?
- What would PCDFS do?
- What would MEMDFS do?

Two Other DFS examples



Order: N, E, S, W?



Order: N, E, S, W

with loops prevented

Forward DFSearch and Backward DFSearch

If you have a predecessors() function as well as a successors() function you can begin at the goal and depth-first-search backwards until you hit a start.

Why/When might this be a good idea?

Invent an Algorithm!

Here’s a way to dramatically decrease costs sometimes. Bidirectional Search. Can you guess what this algorithm is, and why it can be a huge cost-saver?

N	number of states in the problem
B	the average branching factor (the average number of successors) (*B*>1)
L	the length of the path from start to goal with the shortest number of steps
LMAX	Length of longest **cycle-free** path from start to anywhere
Q	the average size of the priority queue

Algorithm		Complete	Optimal	Time	Space
BFS	Breadth First Search	Y	if all transitions same cost	$O(min(N,B^L))$	$O(min(N,B^L))$
LCBFS	Least Cost BFS	Y	Y	$O(min(N,B^L))$	$O(min(N,B^L))$
UCS	Uniform Cost Search	Y	Y	$O(log(Q) * min(N,B^L))$	$O(min(N,B^L))$
PCDFS	Path Check DFS	Y	N	$O(B^{LMAX})$	$O(LMAX)$
MEMDFS	Memoizing DFS	Y	N	$O(min(N,B^{LMAX}))$	$O(min(N,B^{LMAX}))$
BIBFS	Bidirection BF Search				

N	number of states in the problem
B	the average branching factor (the average number of successors) (*B*>1)
L	the length of the path from start to goal with the shortest number of steps
LMAX	Length of longest **cycle-free** path from start to anywhere
Q	the average size of the priority queue

Algorithm		Complete	Optimal	Time	Space
BFS	Breadth First Search	Y	if all transitions same cost	$O(min(N,B^L))$	$O(min(N,B^L))$
LCBFS	Least Cost BFS	Y	Y	$O(min(N,B^L))$	$O(min(N,B^L))$
UCS	Uniform Cost Search	Y	Y	$O(log(Q) * min(N,B^L))$	$O(min(N,B^L))$
PCDFS	Path Check DFS	Y	N	$O(B^{LMAX})$	$O(LMAX)$
MEMDFS	Memoizing DFS	Y	N	$O(min(N,B^{LMAX}))$	$O(min(N,B^{LMAX}))$
BIBFS	Bidirection BF Search	Y	All trans same cost	$O(min(N,2B^{L/2}))$	$O(min(N,2B^{L/2}))$

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

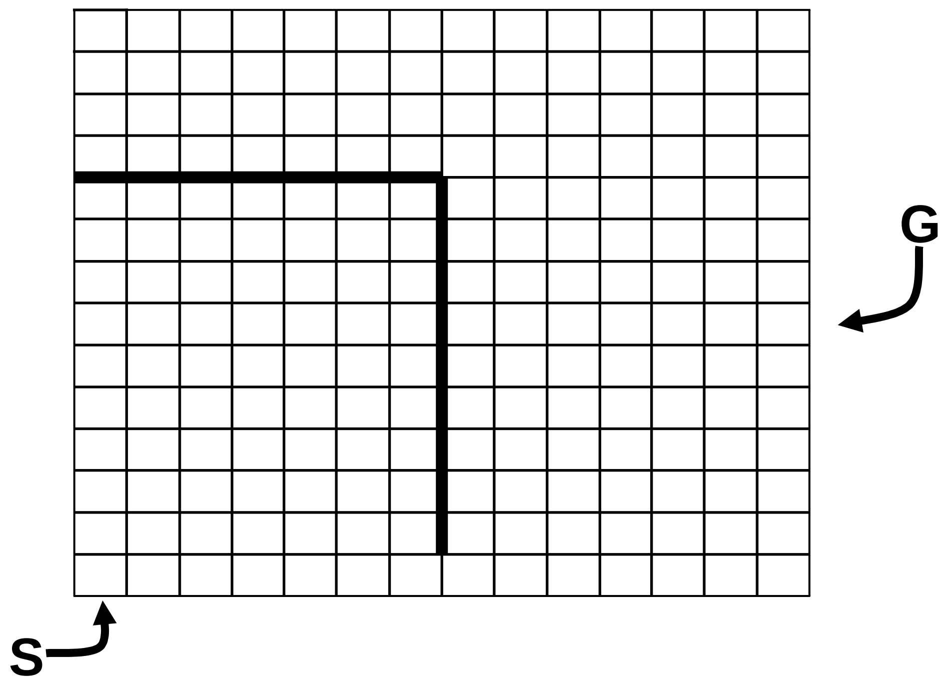
Cost is

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

Can be much better than regular DFS. But cost can be much greater than the number of states.

Maze Example Again

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



Expansion order E, N, W, S

N	number of states in the problem
B	the average branching factor (the average number of successors) (B>1)
L	the length of the path from start to goal with the shortest number of steps
LMAX	Length of longest **cycle-free** path from start to anywhere
Q	the average size of the priority queue

Algorithm		Complete	Optimal	Time	Space
BFS	Breadth First Search	Y	if all transitions same cost	$O(min(N,B^L))$	$O(min(N,B^L))$
LCBFS	Least Cost BFS	Y	Y	$O(min(N,B^L))$	$O(min(N,B^L))$
UCS	Uniform Cost Search	Y	Y	$O(log(Q) * min(N,B^L))$	$O(min(N,B^L))$
PCDFS	Path Check DFS	Y	N	$O(B^{LMAX})$	$O(LMAX)$
MEMDFS	Memoizing DFS	Y	N	$O(min(N,B^{LMAX}))$	$O(min(N,B^{LMAX}))$
BIBFS	Bidirection BF Search	Y	All trans same cost	$O(min(N,2B^{L/2}))$	$O(min(N,2B^{L/2}))$
ID	Iterative Deepening				

N	number of states in the problem
B	the average branching factor (the average number of successors) (B>1)
L	the length of the path from start to goal with the shortest number of steps
LMAX	Length of longest **cycle-free** path from start to anywhere
Q	the average size of the priority queue

Algorithm		Complete	Optimal	Time	Space
BFS	Breadth First Search	Y	if all transitions same cost	$O(min(N,B^L))$	$O(min(N,B^L))$
LCBFS	Least Cost BFS	Y	Y	$O(min(N,B^L))$	$O(min(N,B^L))$
UCS	Uniform Cost Search	Y	Y	$O(log(Q) * min(N,B^L))$	$O(min(N,B^L))$
PCDFS	Path Check DFS	Y	N	$O(B^{LMAX})$	$O(LMAX)$
MEMDFS	Memoizing DFS	Y	N	$O(min(N,B^{LMAX}))$	$O(min(N,B^{LMAX}))$
BIBFS	Bidirection BF Search	Y	All trans same cost	$O(min(N,2B^{L/2}))$	$O(min(N,2B^{L/2}))$
ID	Iterative Deepening	Y	if all transitions same cost	$O(B^L)$	$O(L)$

Searching with Partial Information

What happens if we relax the constraints:

Observable and Deterministic

- Sensorless (Conformant) Problems

- Contingency Problems

- Exploration Problems

Searching with Partial Information

What happens if we relax the constraints:

Observable and Deterministic

- Sensorless (Conformant) Problems

No sensors ⇨ Need to act in the "belief" space

- Contingency Problems

State space partially observable ⇨ New information after each action

- Exploration Problems

State space unknown ⇨ Deliberately act to acquire new information

Reading Assignment

- ***Read Chapters 1-3 in Russel & Norvig***

"Artificial Intelligence: A Modern Approach"

2^{nd} Ed

- Also read Bibliographical and Historical Notes!!
(They are interesting and make the connections with many other disciplines)

Next: Informed (Heuristic) Search