Lecture 2: Uninformed Search

(Russell and Norvig Chapter 3)

Hey guess what you're learning pseudoscience.

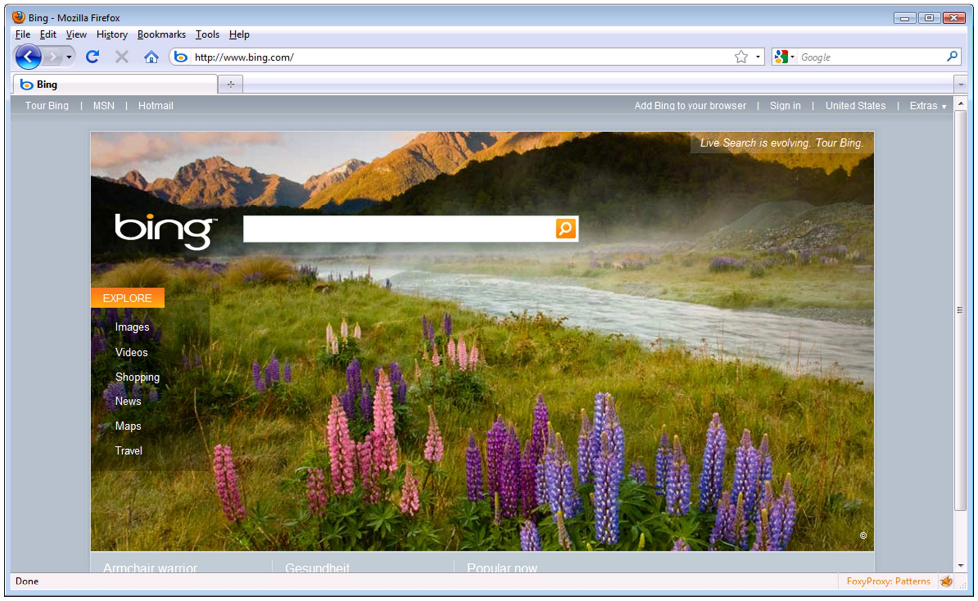
Correct URL of Class Website:

www.andrew.cmu.edu/course/15-381-f09/index.html

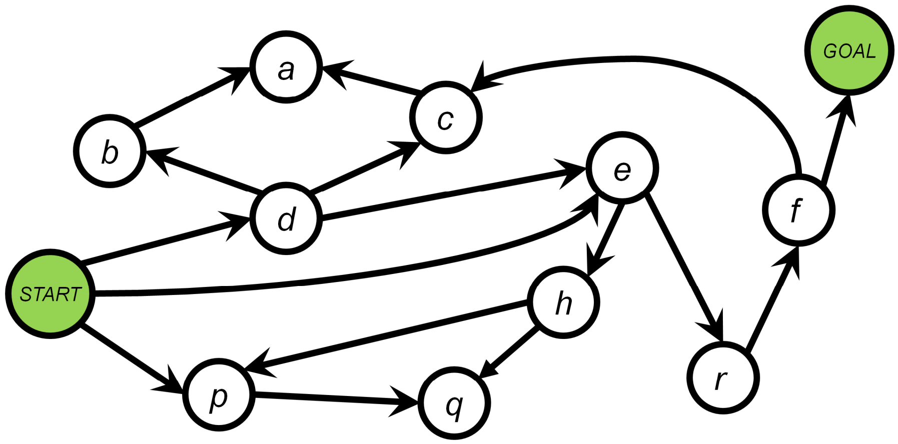
What happens when computers become as intelligent as humans?

Are they going to kill us? Will they be slaves? That's what we're working towards.

Lecture 2: Uninformed Search

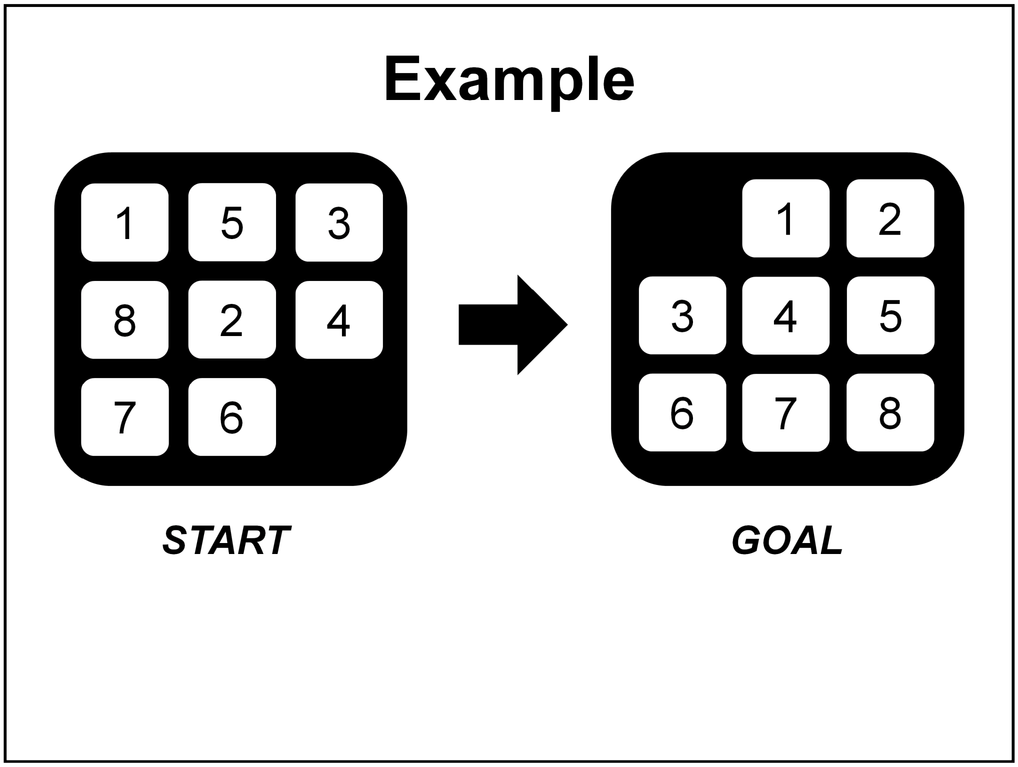

Bing - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://www.bing.com/
Google
Bing
Tour Bing | MSN | Hotmail
Add Bing to your browser | Sign in | United States | Extras
Live Search is evolving. Tour Bing.
bing
EXPLORE
Images
Videos
Shopping
News
Maps
Travel
Armchair warrior
Gesundheit
Popular now
Done
FoxyProxy: Patterns

A Search Problem



Find a path from START to GOAL
OR
Find the minimum number of transitions

Many problems can be encoded this way



How to encode as search problem?

Make a graph of possible states.

State: config of puzzle

Transition: up to 4 poss moves from each state

Solvable in 22 steps on average

But state space is really big. 2*1^5.

Example

State: Configuration of the puzzle

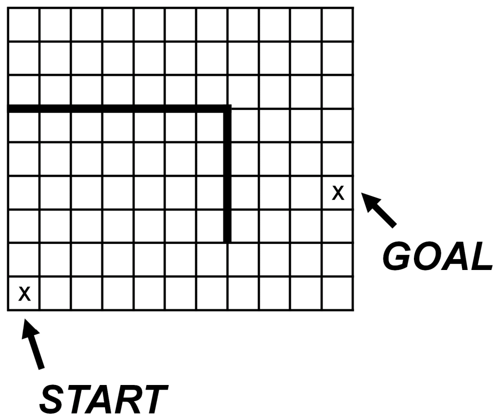
Transitions: Up to 4 possible moves from each states (up, down, left, right)

Solvable in 22 steps on average

But: 1.8 $\times 10^5$ states (1.3 $\times 10^{12}$ states for the 15-puzzle)

Cannot represent a set of states explicitly

Example: Robot Navigation



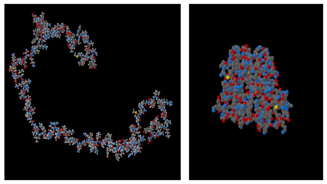
States = positions in the map

Transitions = allowed motions

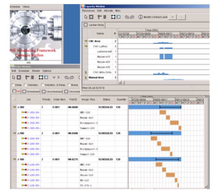
Navigation: Going from point START to point GOAL given a (deterministic) map

Allowed motions: You can move up, down, left, or right. You can't move through walls.

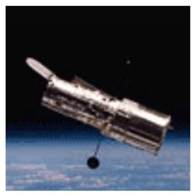
Other Real-Life Examples



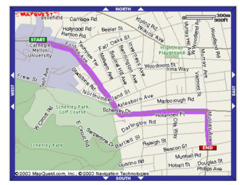
Protein design



Manufacturing



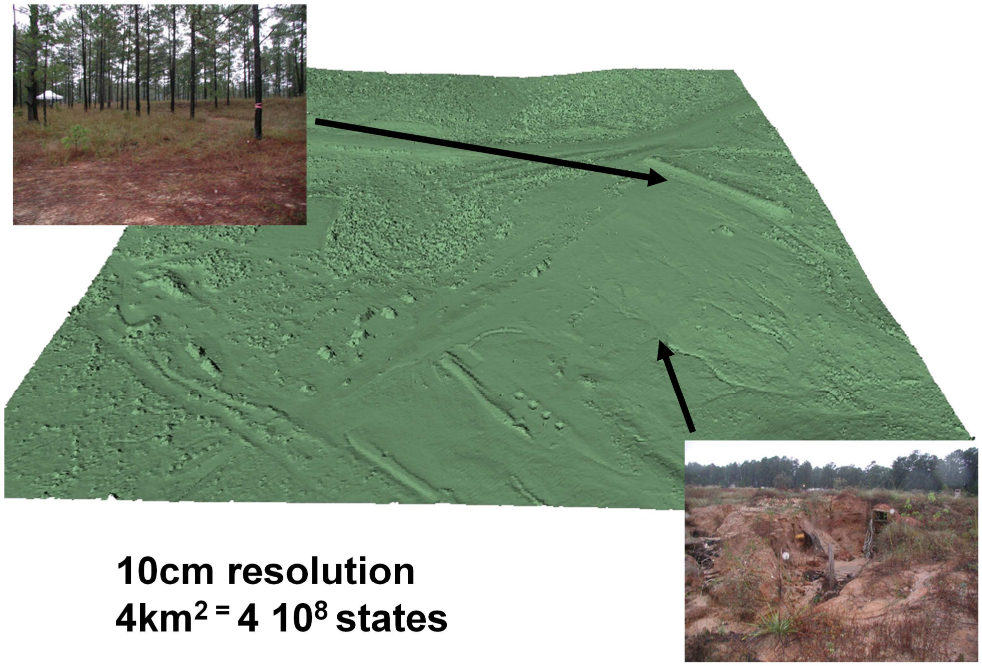
Scheduling/Science



Driving

Don’t necessarily know explicitly the structure of a search problem

In the real world the graph is really really big. V large # states



10cm resolution
$4 km^2 = 4 \; 10^8$ states

What We are Not Addressing (yet)

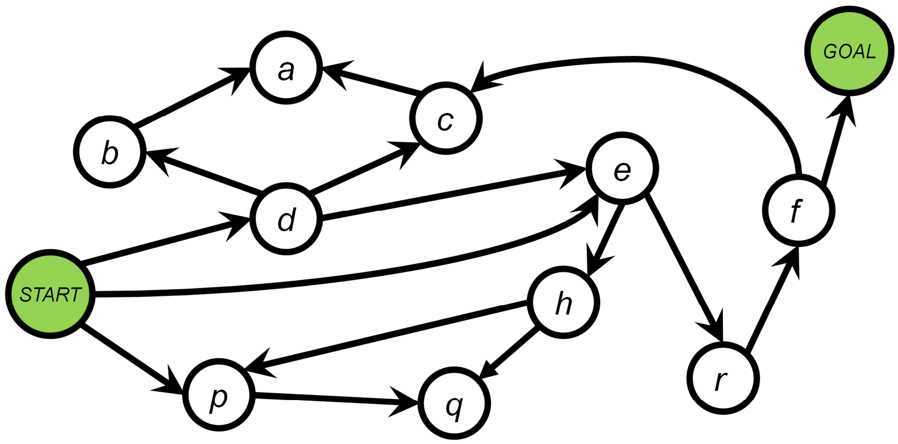
- Uncertainty/Chance: State and transitions are known and deterministic

- Game against an adversary

- Multiple agents and cooperation

- Continuous state space: For now, the set of states is discrete

A Search Problem



Formulation

Q: Finite set of states

$S \subseteq Q$: Non-empty set of start states

$G \subseteq Q$: Non-empty set of goal states

succs: Function $Q \rightarrow \mathcal{P}(Q)$

- succs(s) = Set of states that can be reached from s in one step

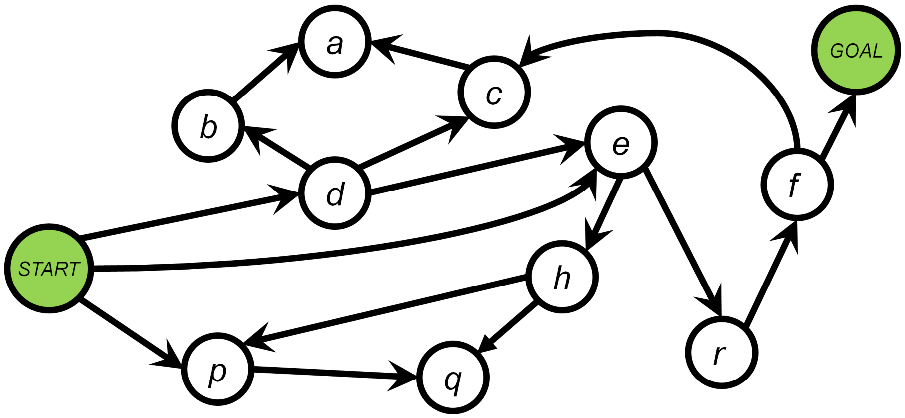
cost: function $Q \times Q \rightarrow$ Positive Numbers

- cost(s,s′) = Cost of taking a one-step transition from s to s′

Problem: Find a sequence $\{s_1,\ldots,s_K\}$ such that:

- $s_1 \in S$
- $s_K \in G$
- $s_{i+1} \in succs(s_i)$
- $\Sigma\, cost(s_i, s_{i+1})$ is the smallest among all possible sequences (desirable but optional)

Example



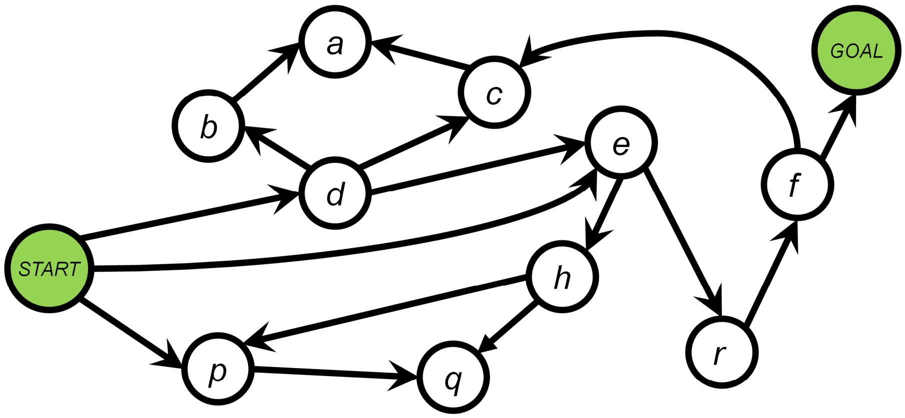
Q **= {*START*, *GOAL*, *a*, *b*, *c*, *d*, *e*, *f*, *h*, *p*, *q*, *r*}**

S **= {*START*}, *G* = {*GOAL*}**

succs(*d*) = {*b*,*c*}, succs(*START*) = {*p*,*e*,*d*}, succs(*a*) = NULL

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

Desirable Properties



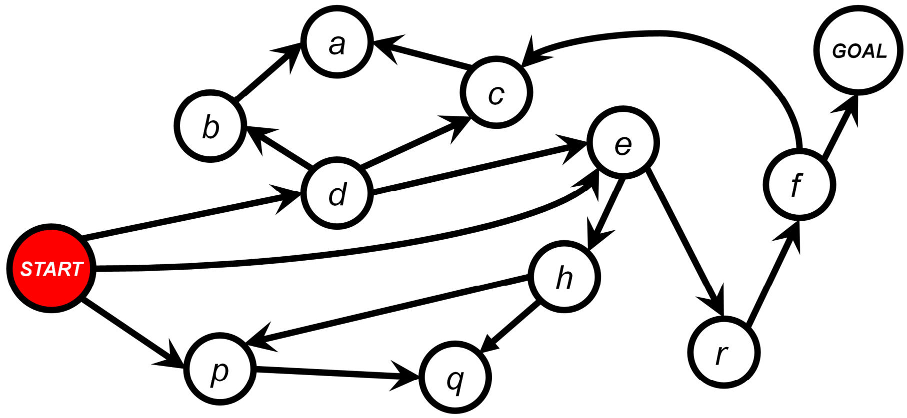
Completeness: An algorithm is complete if it is guaranteed to find a path if one exists

Optimality: The total cost of the path is the lowest among all possible paths from start to goal

Time and Space Complexity

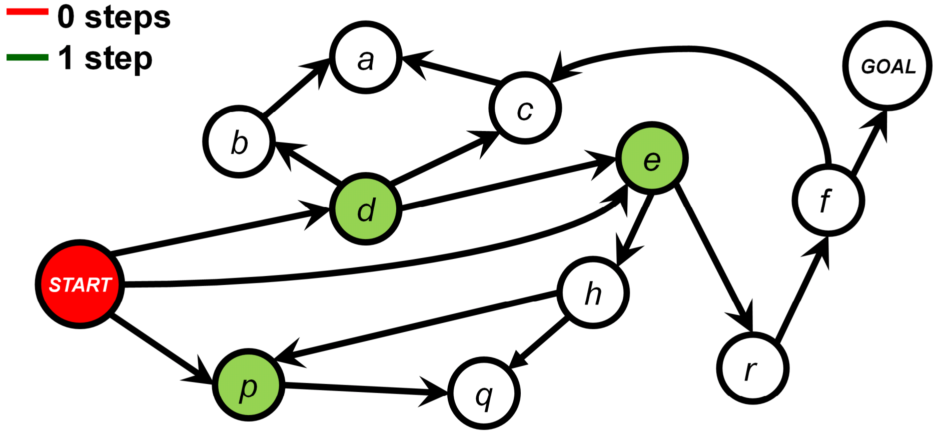
Keep in mind storing graph in mem is not an option.

Breadth-First Search



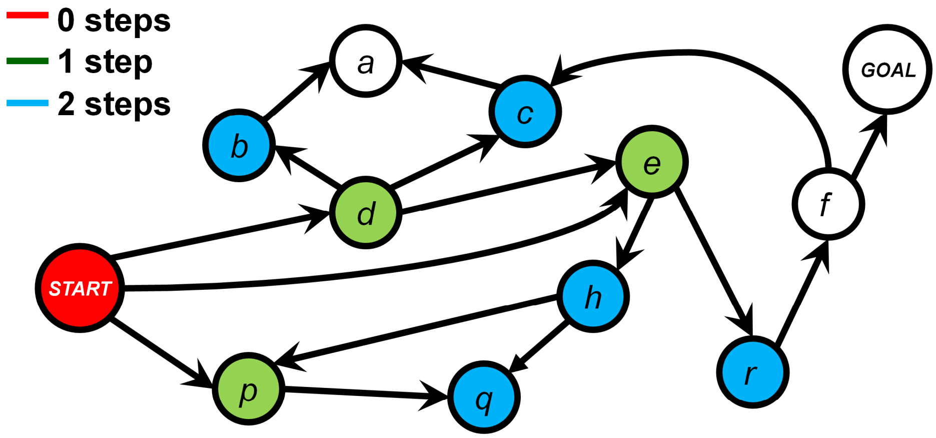
Label all states that are 0 steps from S. Call that set V_o

Breadth-First Search



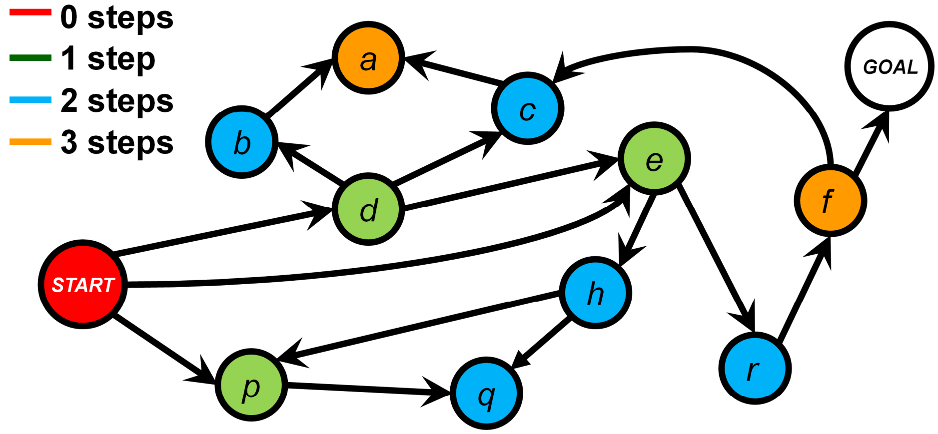
Label the successors of the states in V_o that are not yet labeled as the set V_1 of states that are 1 step away from the start

Breadth-First Search



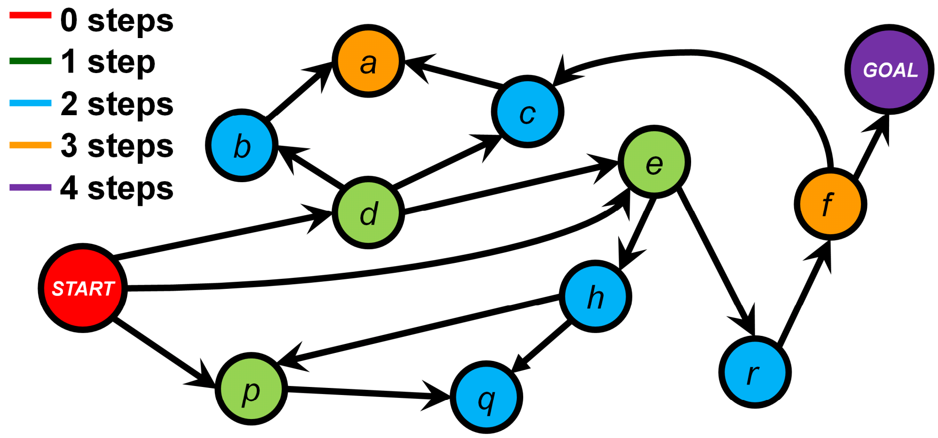
Label the successors of the states in V_1 that are not yet labeled as the set V_2 of states that are 2 steps away from the start

Breadth-First Search



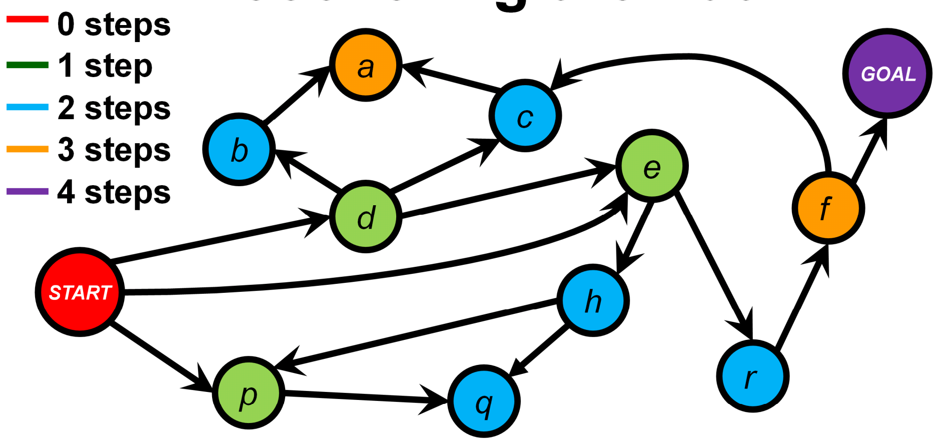
Label the successors of the states in V_2 that are not yet labeled as the set V_3 of states that are 3 steps away from the start

Breadth-First Search



Stop when goal is reached in the current expansion set. In this case goal can be reached in 4 steps.

Recovering the Path

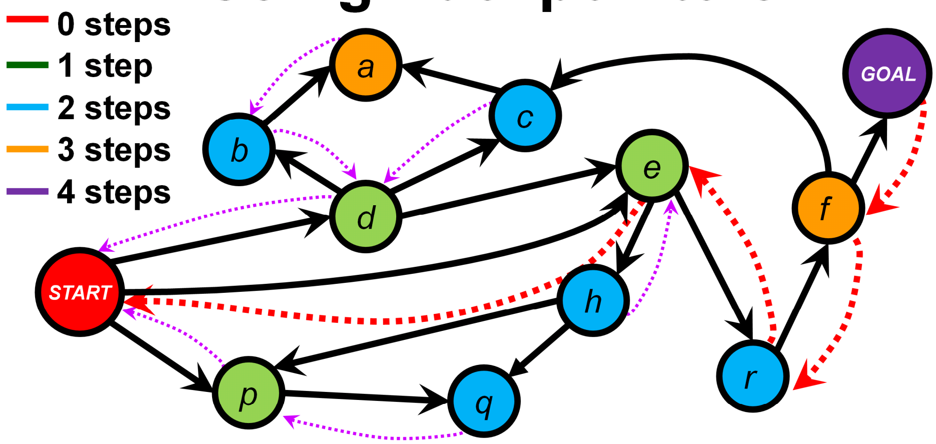


Record the predecessor when labeling a state

When labeling GOAL, I was expanding the neighbors of f, so f is the predecessor of GOAL

Final solution: {START, e, r, f, GOAL}

Using Backpointers

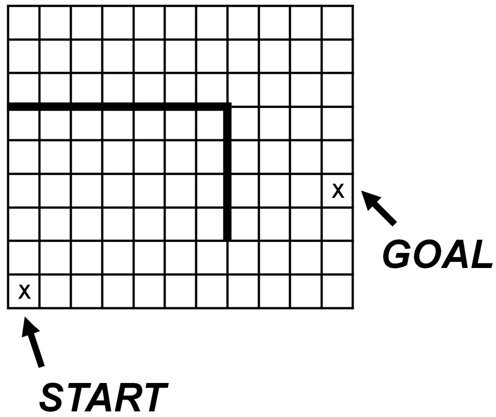


A backpointer previous(*s*) points to the node that stored the state that was expanded to label *s*

The path is recovered by following the backpointers starting at the goal state

The “right” way to implement

Example: Robot Navigation



States =
positions in the map

Transitions =
allowed motions

Navigation: Going from point START to point GOAL given a (deterministic) map

Breadth-First Search

V_0 = S (the set of start states)
previous(START) = NULL
k = 0

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

- 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 output FAILURE
else build the solution path thus:

- Define S_k = GOAL, and for all i ≤ k, define S_{i-1} = **previous**(S_i)
- Return path = {S_1,.., S_k}

Properties

- BFS can handle multiple start and goal states
- Can work either by searching forward from the start or backward for the goal (forward/backward chaining)
- (Which way is better?)
- Guaranteed to find the lowest-cost path in terms of number of transitions?

Which is better? Depends on graph. For example, for a dense tree with start at root and goal at leaf then backwards is better. Others may vary.

Guaranteed lowest cost IF all edges have uniform, nonnegative cost.

Complexity

B **= Average number of successors (branching factor)**

L **= Length from start to goal on shortest path**

	Algorithm	Complete	Optimal	Time	Space
BFS	Breadth First Search				

Complexity

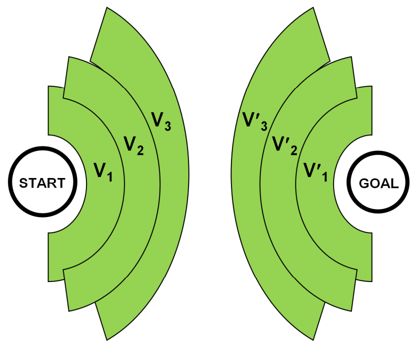
B **= Average number of successors (branching factor)**

L **= Length from start to goal on shortest path**

	Algorithm	Complete	Optimal	Time	Space
BFS	Breadth First Search	Y	Y, If all trans. have same cost	$O(B^L)$	$O(B^L)$

Bidirectional Search

- BFS search simultaneously forward from START and backward from GOAL
- What's the stopping criterion?
- Under what condition is it optimal?



Stopping: when intersect(V,V') nonempty

Optimal if all costs uniform

Complexity

B **= Average number of successors (branching factor)**
L **= Length from start to goal on shortest path**

	Algorithm	Complete	Optimal	Time	Space
BFS	Breadth First Search	Y	Y, if all trans. have same cost	O(B^L)	O(B^L)
BIBFS	Bi-directional BFS				

Complexity

***B* = Average number of successors (branching factor)**
***L* = Length from start to goal on shortest path**

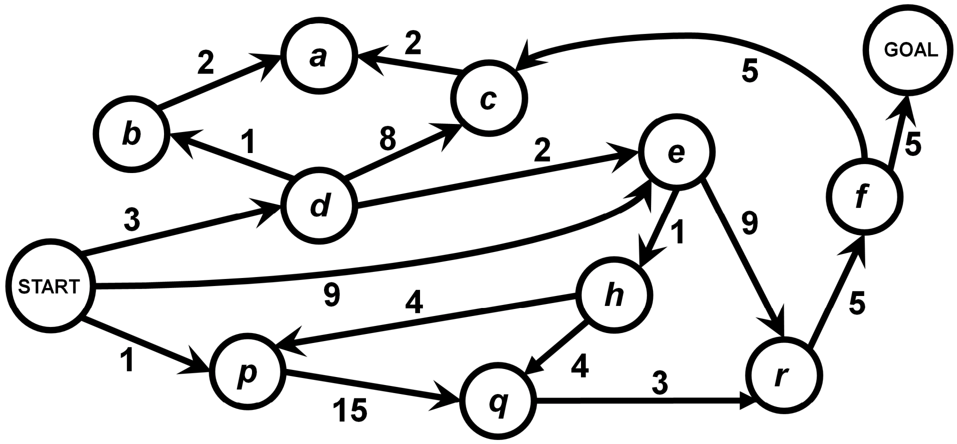
	Algorithm	Complete	Optimal	Time	Space
BFS	Breadth First Search	Y	Y, if all trans. have same cost	$O(B^L)$	$O(B^L)$
BIBFS	Bi-directional BFS	Y	Y, If all trans. have same cost	$O(2B^{L/2})$	$O(2B^{L/2})$

Major savings when bidirectional search is possible because $2B^{L/2} << B^L$

If B = 10, L = 6, then 22,200 states generated vs ~10^7

Note: Going backwards isn't always possible

Counting Transition Costs



BFS finds the shortest path in number of steps but does not take into account transition costs

New field to find least cost path: At iteration *k*, g(*s*) = least cost of path to *s* in *k* or fewer steps

Uniform Cost Search

- Strategy to select state to expand next: use the state with the smallest value of g( ) so far
- Use priority queue for efficient access to minimum g at every iteration

Priority Queue

- **Priority queue = data structure in which data of the form (*item*, *value*) can be inserted and the minimum value item can be retrieved efficiently**
- **Operations:**
  - **Init (*PQ*): Initialize empty queue**
  - **Insert (*PQ*, *item*, *value*): Insert a pair in the queue**
  - **Pop (*PQ*): Returns the pair with the minimum *value***
- **In our case:**
  - ***item* = state**
  - ***value* = current cost g( )**

Complexity: O(log(number of pairs in PQ)) for insertion and pop operations

Complexity depends on the implementation of the PQ. This complexity is for a min heap implementation. For a survey of different PQ implementations see http://www.theturingmachine.com/algorithms/heaps.html

Uniform Cost Search

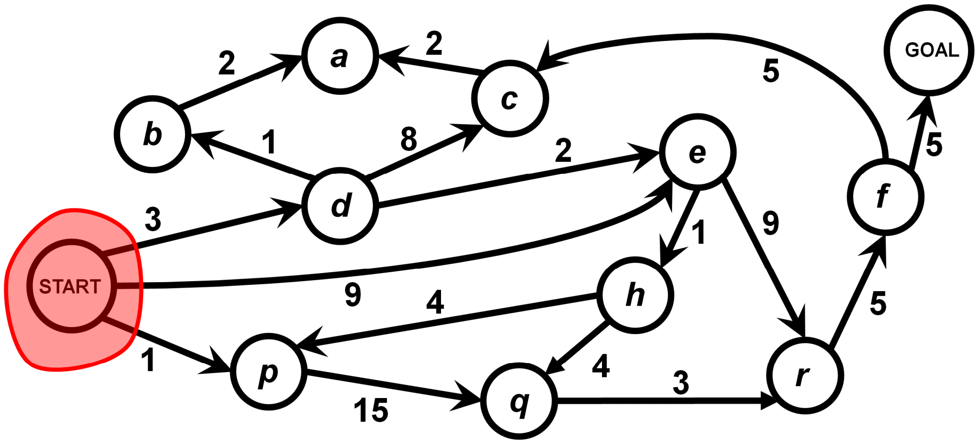
PQ = Current set of evaluated states

Value (priority) of state = g(*s*) = current cost of path to *s*

Basic iteration:

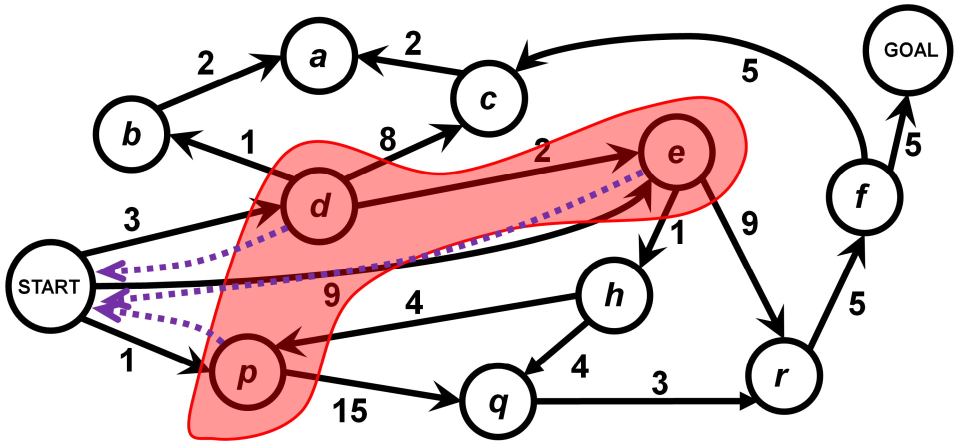
1. Pop the state *s* with t
   cost from *PQ*
2. Evaluate the path cos
   successors of *s*
3. Add the successors

We add the successors of *s* that have not yet been visited and we update the cost of those currently in the queue



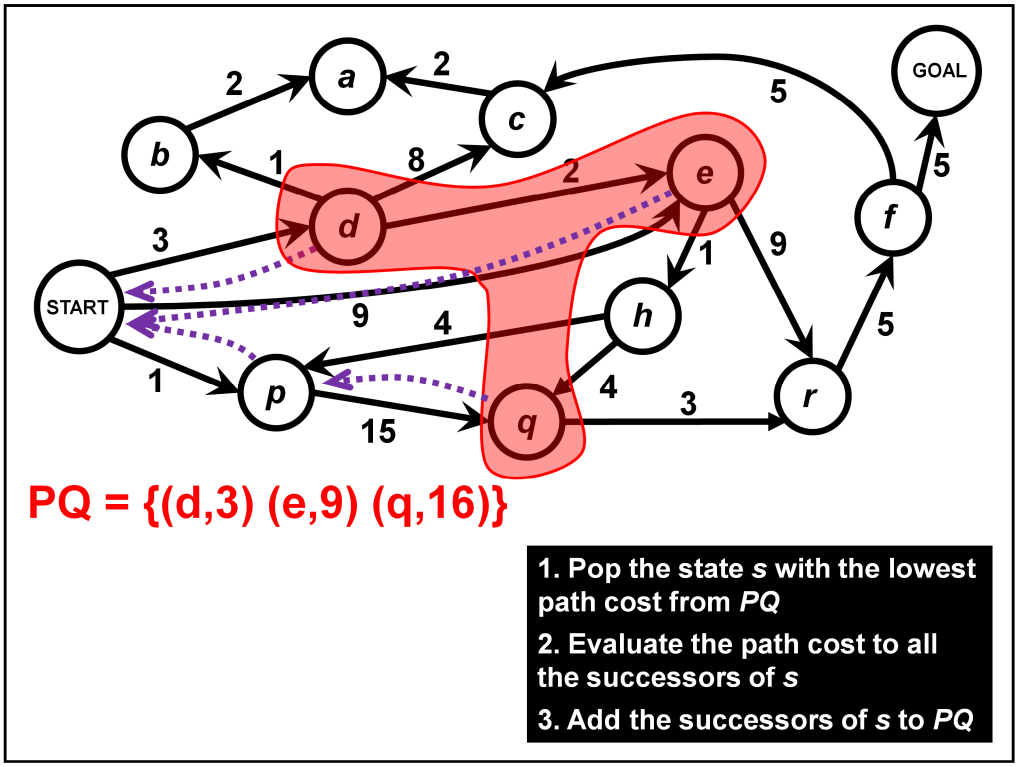
PQ = {(START,0)}

1. Pop the state *s* with the lowest path cost from *PQ*
2. Evaluate the path cost to all the successors of *s*
3. Add the successors of *s* to *PQ*

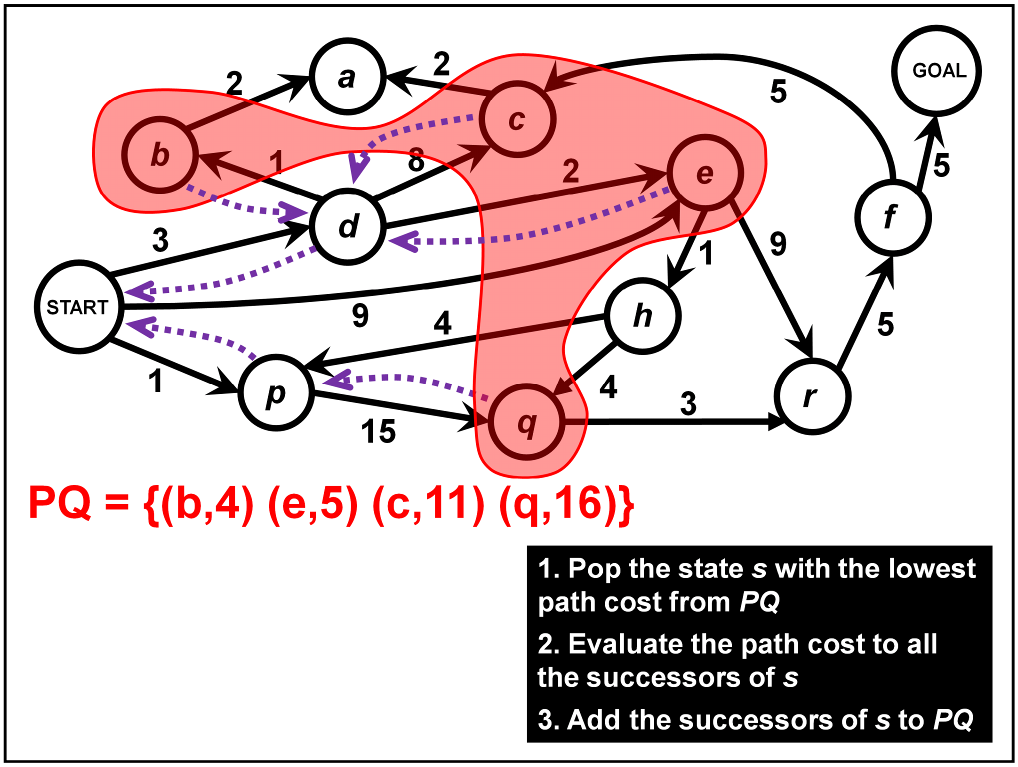


PQ = {(p,1) (d,3) (e,9)}

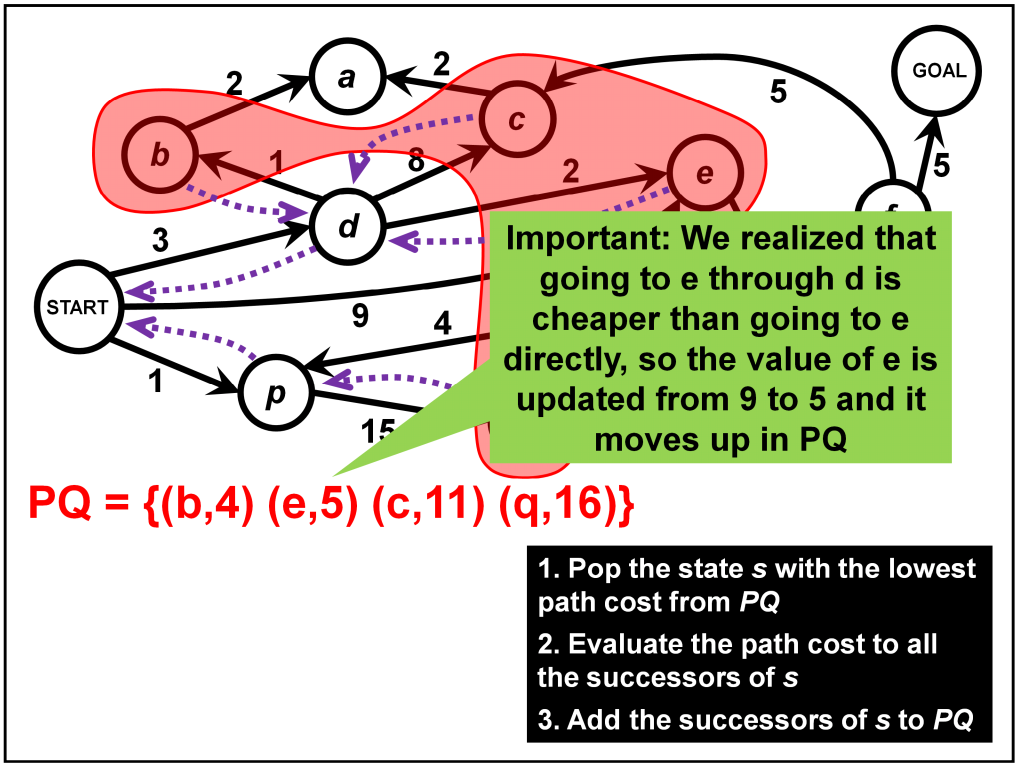
1. Pop the state *s* with the lowest path cost from *PQ*
2. Evaluate the path cost to all the successors of *s*
3. Add the successors of *s* to *PQ*



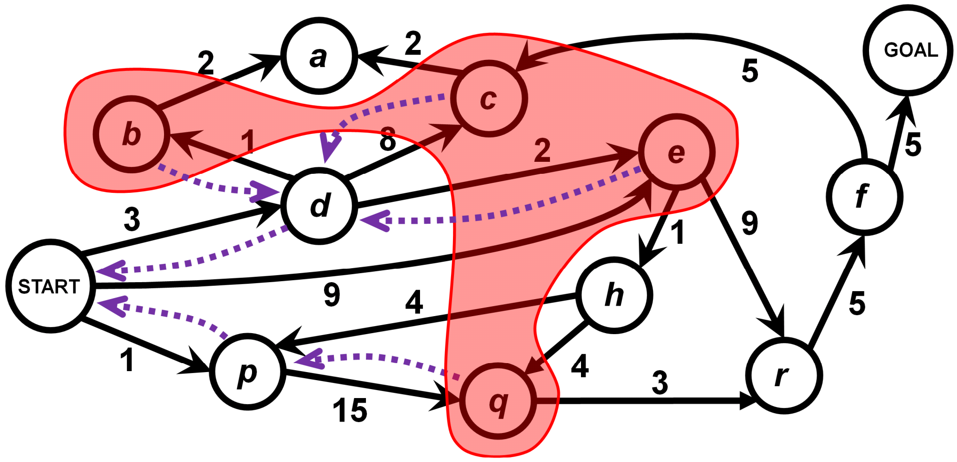
Add cost from p to q to the value of p



The value for item e has been updated from 9 to 5. This is generally not supported by a PQ, but there are ways to fake it.

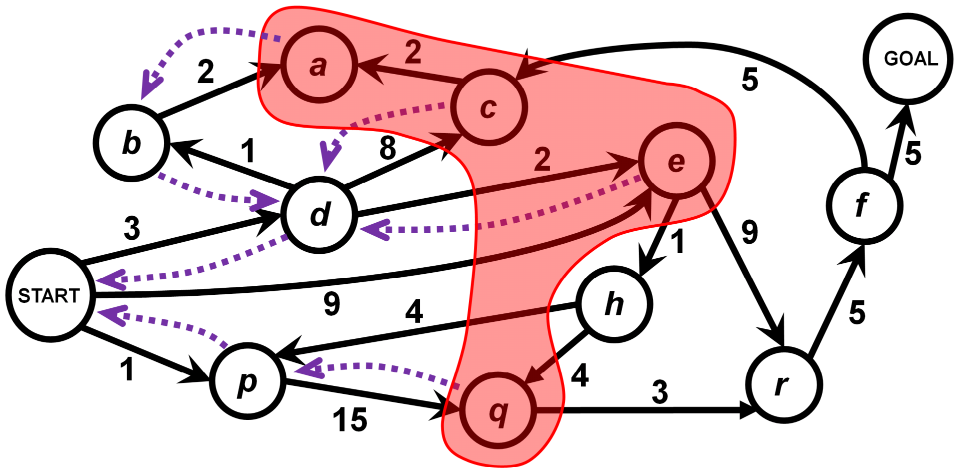


a
b
c
d
e
f
p
START
GOAL
2
2
1
8
2
5
5
3
9
4
1
15
Important: We realized that going to e through d is cheaper than going to e directly, so the value of e is updated from 9 to 5 and it moves up in PQ
PQ = {(b,4) (e,5) (c,11) (q,16)}
1. Pop the state s with the lowest path cost from PQ
2. Evaluate the path cost to all the successors of s
3. Add the successors of s to PQ



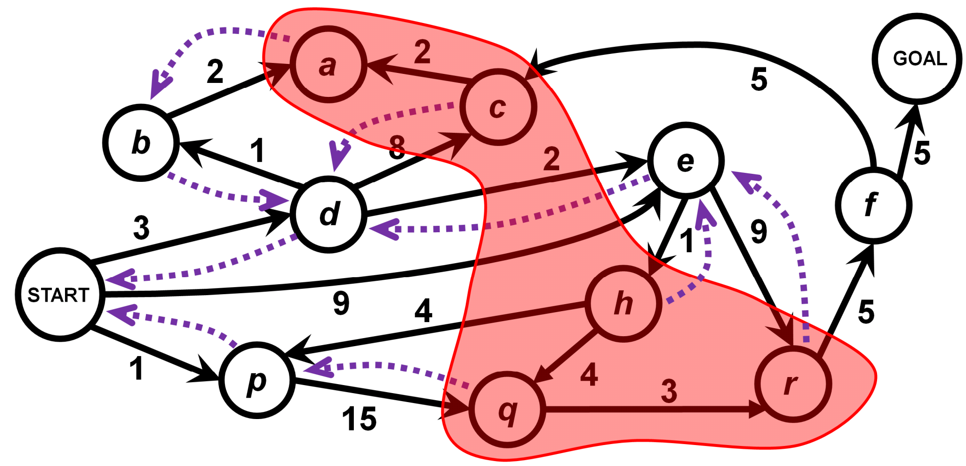
PQ = {(b,4) (e,5) (c,11) (q,16)}

1. Pop the state *s* with the lowest path cost from *PQ*
2. Evaluate the path cost to all the successors of *s*
3. Add the successors of *s* to *PQ*



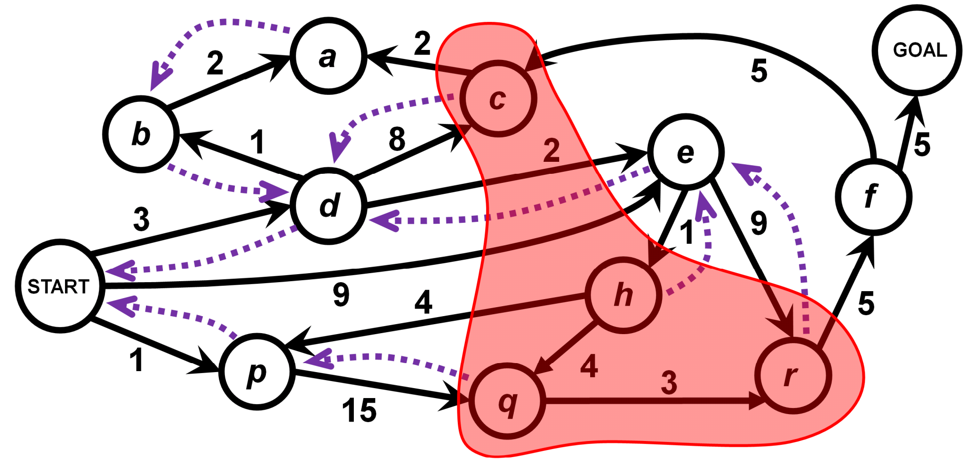
PQ = {(e,5) (a,6) (c,11) (q,16)}

1. Pop the state *s* with the lowest path cost from *PQ*
2. Evaluate the path cost to all the successors of *s*
3. Add the successors of *s* to *PQ*



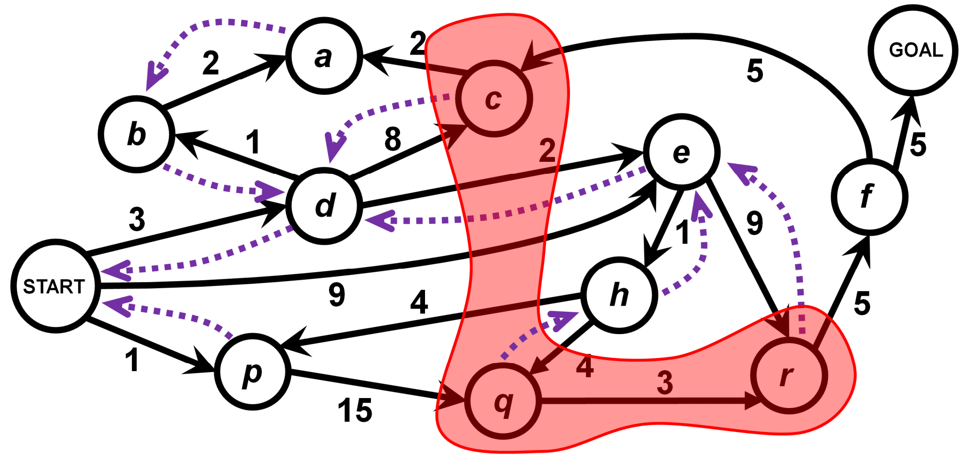
PQ = {(a,6) (h,6) (c,11) (r,14) (q,16)}

1. Pop the state *s* with the lowest path cost from *PQ*
2. Evaluate the path cost to all the successors of *s*
3. Add the successors of *s* to *PQ*



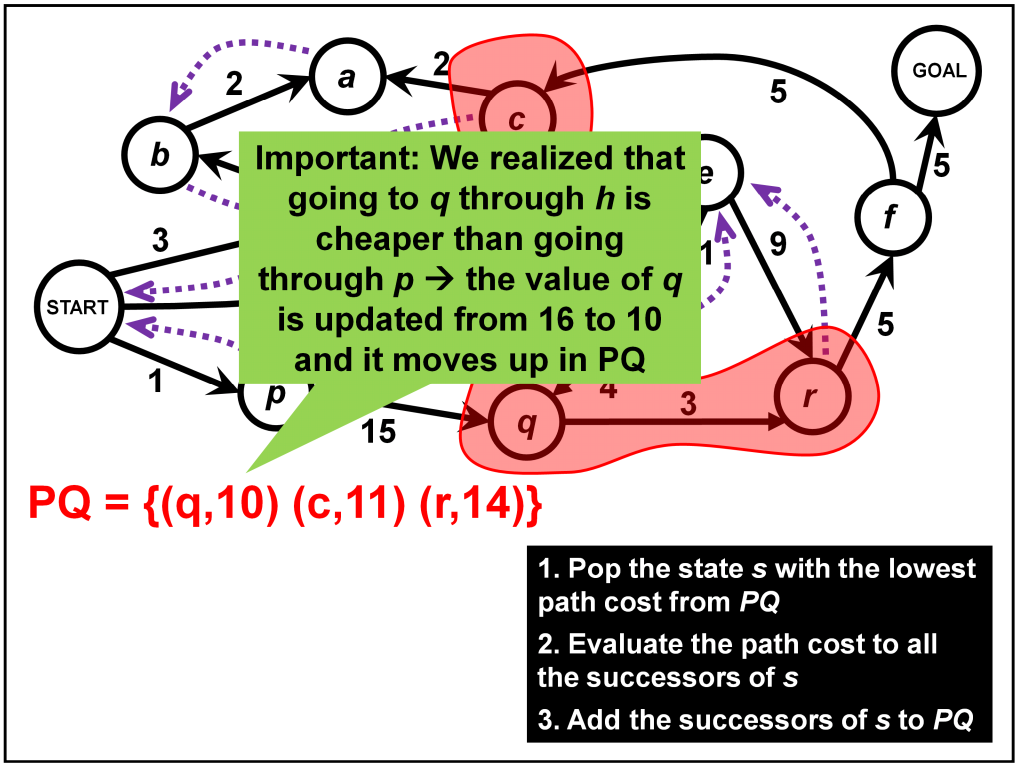
PQ = {(h,6) (c,11) (r,14) (q,16)}

1. Pop the state *s* with the lowest path cost from *PQ*
2. Evaluate the path cost to all the successors of *s*
3. Add the successors of *s* to *PQ*



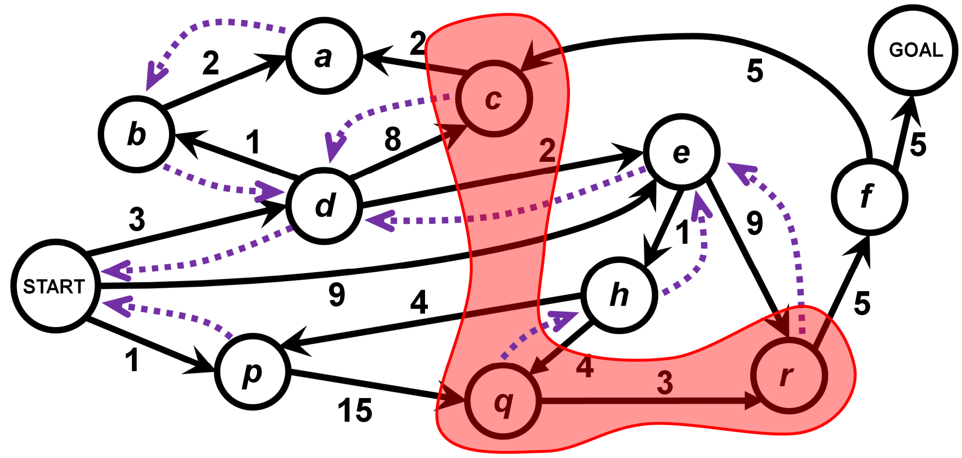
PQ = {(q,10) (c,11) (r,14)}

1. Pop the state *s* with the lowest path cost from *PQ*
2. Evaluate the path cost to all the successors of *s*
3. Add the successors of *s* to *PQ*



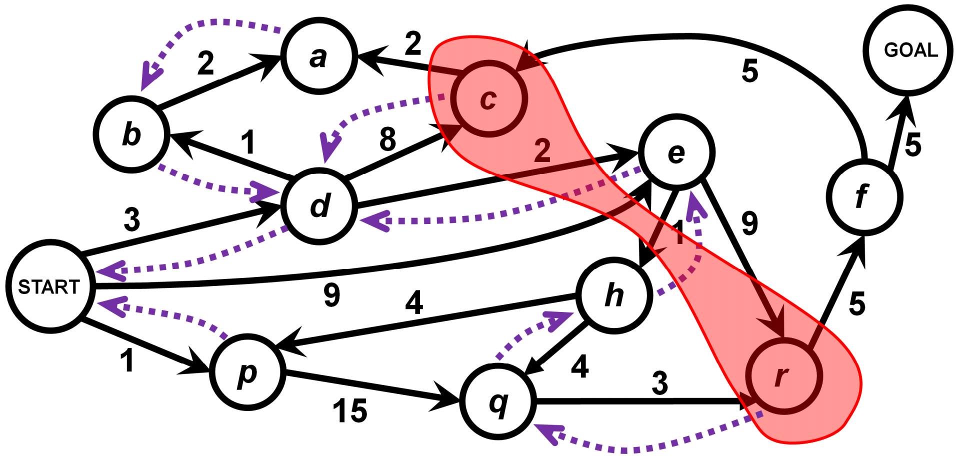
If p is not in the queue but visited, we already visited it.

(If costs aren't negative or 0, this works. In fact this is why it won't work if costs are negative or 0) Note keep in memory everything we've seen.



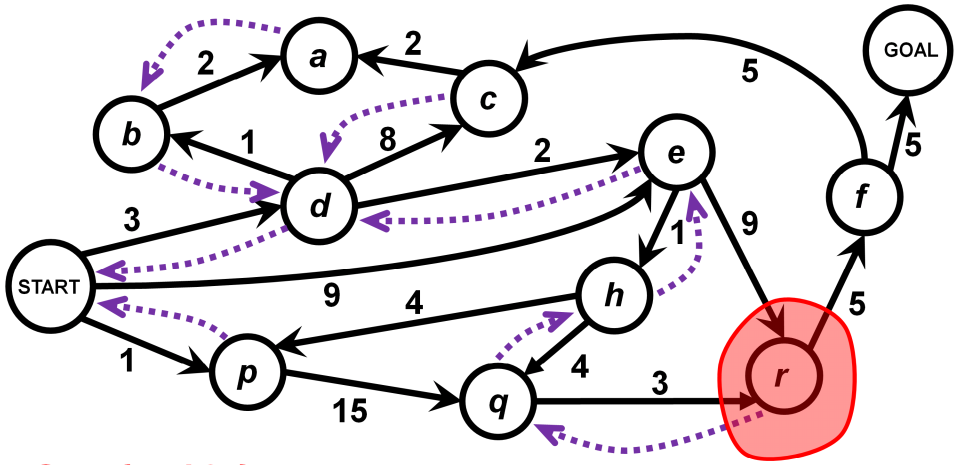
PQ = {(q,10) (c,11) (r,14)}

1. Pop the state *s* with the lowest path cost from *PQ*
2. Evaluate the path cost to all the successors of *s*
3. Add the successors of *s* to *PQ*



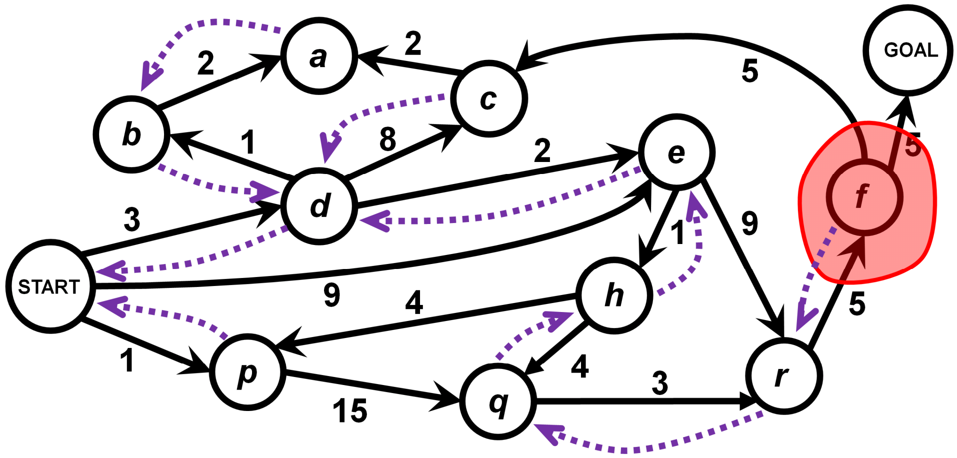
PQ = {(c,11) (r,13)}

1. Pop the state *s* with the lowest path cost from *PQ*
2. Evaluate the path cost to all the successors of *s*
3. Add the successors of *s* to *PQ*



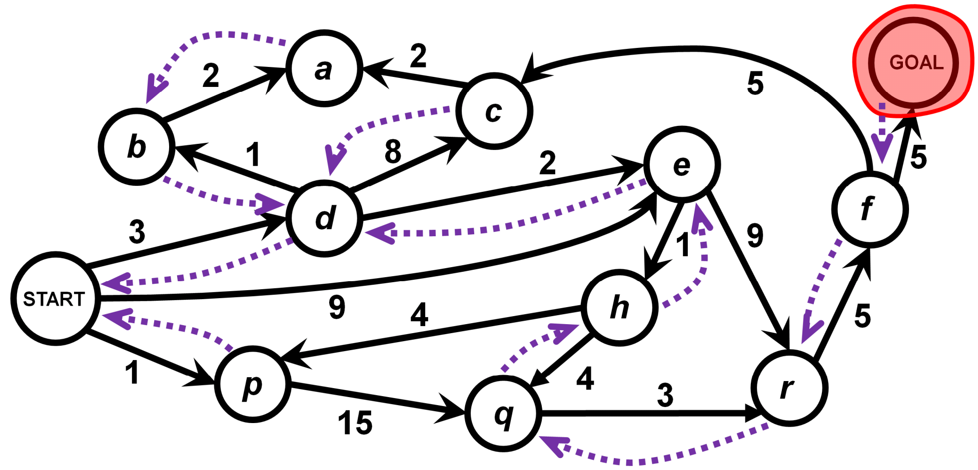
PQ = {(r,13)}

1. Pop the state *s* with the lowest path cost from *PQ*
2. Evaluate the path cost to all the successors of *s*
3. Add the successors of *s* to *PQ*



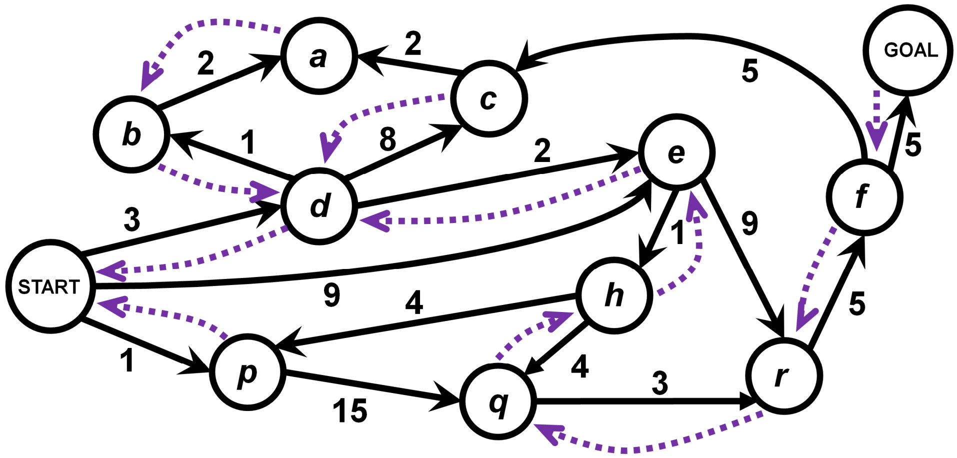
PQ = {(f,18)}

1. Pop the state *s* with the lowest path cost from *PQ*
2. Evaluate the path cost to all the successors of *s*
3. Add the successors of *s* to *PQ*



PQ = {(GOAL,23)}

1. Pop the state *s* with the lowest path cost from *PQ*
2. Evaluate the path cost to all the successors of *s*
3. Add the successors of *s* to *PQ*



Final path: {START, d, e, h, q, r, f, GOAL}

- **This path is optimal in total cost even though it has more transitions than the one found by BFS**
- **What should be the stopping condition?**
- **Under what conditions is UCS complete/optimal?**

Can't stop until you POP the goal!

Complexity

B **= Average number of successors (branching factor)**
L **= Length from start to goal on shortest path**
C **= Cost of optimal path**
Q **= Average size of the priority queue**

	Algorithm	Complete	Optimal	Time	Space
BFS	Breadth First Search	Y	Y, If all trans. have same cost	$O(B^L)$	$O(B^L)$
BIBFS	Bi-directional Breadth First Search	Y	Y, If all trans. have same cost	$O(2B^{L/2})$	$O(2B^{L/2})$
UCS	Uniform Cost Search				

Complexity

B **= Average number of successors (branching factor)**
L **= Length from start to goal on shortest path**
C **= Cost of optimal path**
Q **= Average size of the priority queue**

	Algorithm	Complete	Optimal	Time	Space
BFS	Breadth First Search	Y	Y, If all trans. have same cost	$O(B^L)$	$O(B^L)$
BIBFS	Bi-directional Breadth First Search	Y	Y, If all trans. have same cost	$O(2B^{L/2})$	$O(2B^{L/2})$
UCS	Uniform Cost Search	Y, if cost > ε > 0	Y, if cost > 0	$O(\log(Q)*B^{C/\varepsilon})$	$O(B^{C/\varepsilon})$

Why is this correct?

Log(Q) is due to priority queue

Each edge cost >= epsilon, therefore B^(C/epsilon) is upper bound on number of steps.

Why no log in space? Log has to do with popping priority queue, but space is just storing the thing (so multiply space by 2)

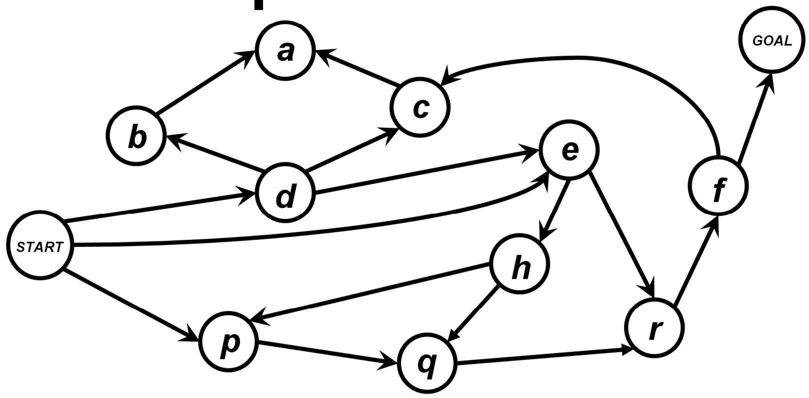
Limitations of BFS

Limitations of BFS

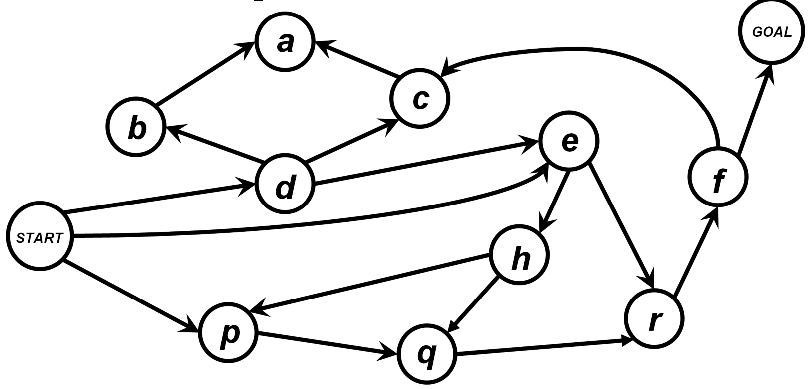
- Memory usage is $O(B^L)$ in general
- Limitation in many problems in which the states cannot be enumerated or stored explicitly, e.g., large branching factor
- Alternative: Find a search strategy that requires little storage for use in large problems

You could visit a lot of nodes (ex driving)

Depth First Search



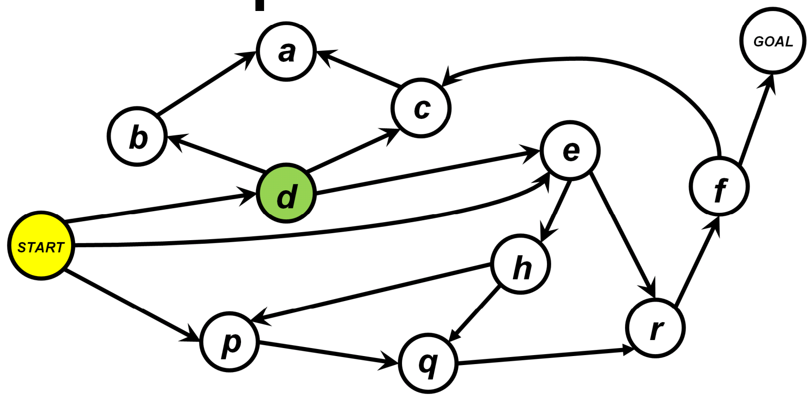
Depth First Search



General idea:

- Expand the most recently expanded node if it has successors
- Otherwise backup to the previous node on the current path

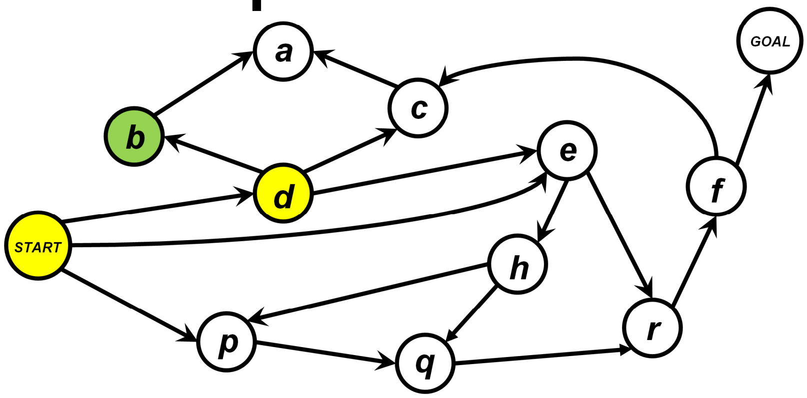
Depth First Search



General idea:

- Expand the most recently expanded node if it has successors
- Otherwise backup to the previous node on the current path

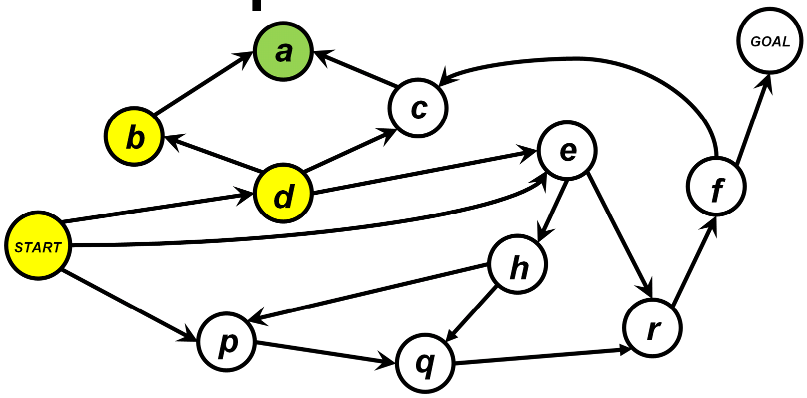
Depth First Search



General idea:

- **Expand the most recently expanded node if it has successors**
- **Otherwise backup to the previous node on the current path**

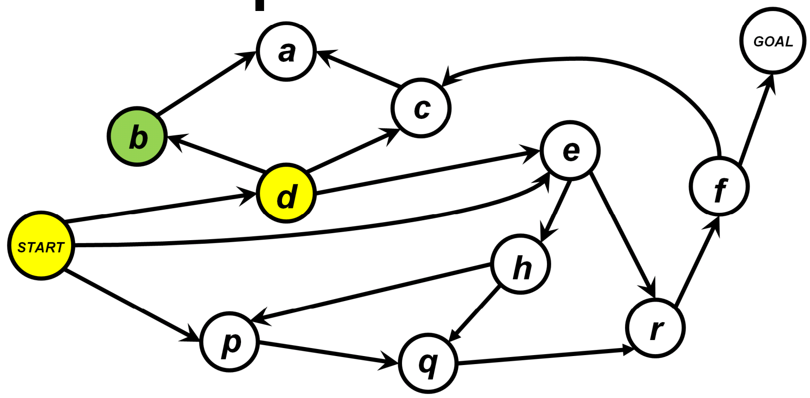
Depth First Search



General idea:

- **Expand the most recently expanded node if it has successors**
- **Otherwise backup to the previous node on the current path**

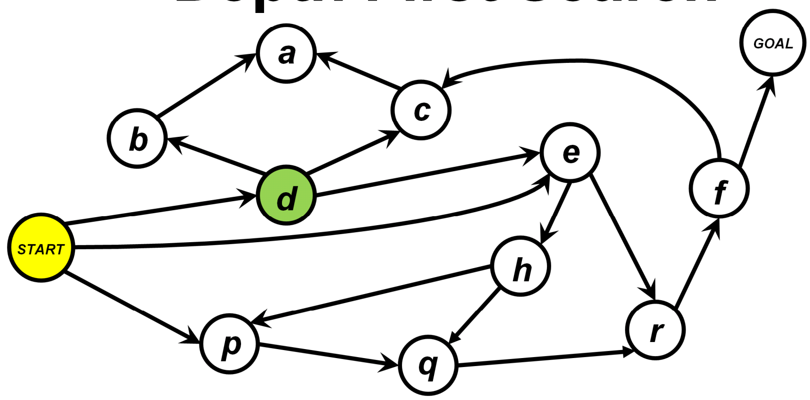
Depth First Search



General idea:

- **Expand the most recently expanded node if it has successors**
- **Otherwise backup to the previous node on the current path**

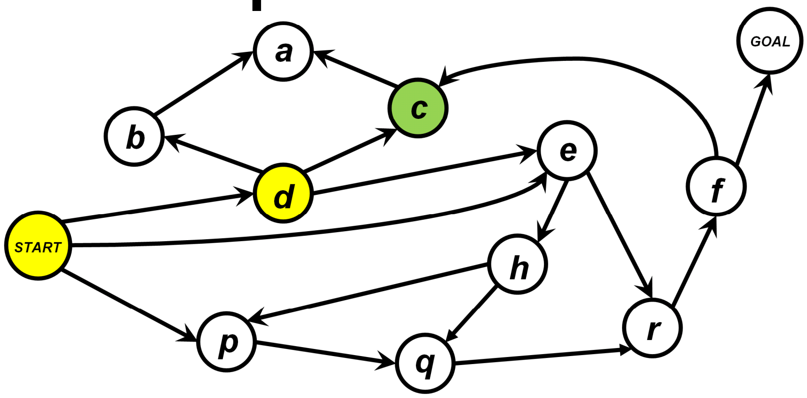
Depth First Search



General idea:

- Expand the most recently expanded node if it has successors
- Otherwise backup to the previous node on the current path

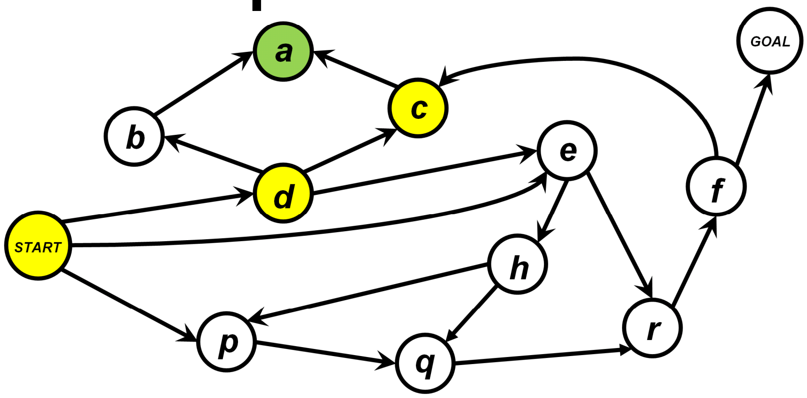
Depth First Search



General idea:

- **Expand the most recently expanded node if it has successors**
- **Otherwise backup to the previous node on the current path**

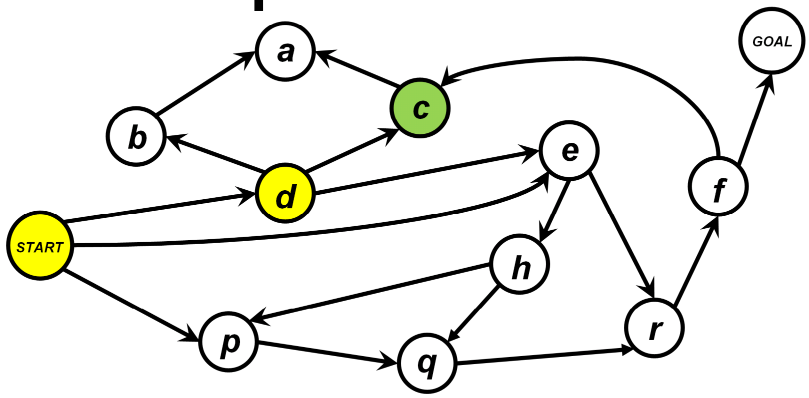
Depth First Search



General idea:

- **Expand the most recently expanded node if it has successors**
- **Otherwise backup to the previous node on the current path**

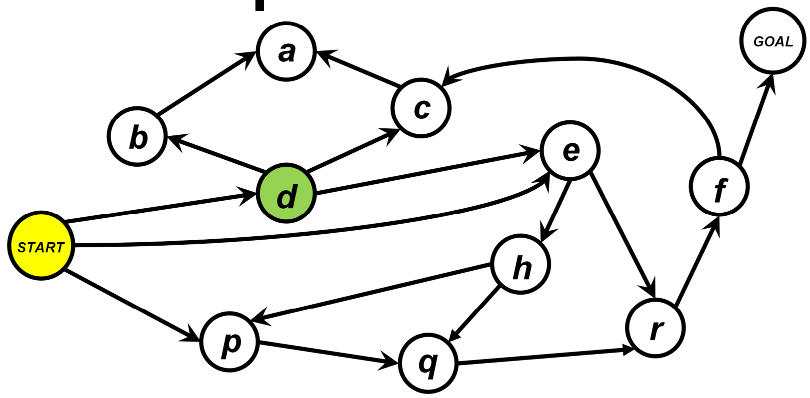
Depth First Search



General idea:

- Expand the most recently expanded node if it has successors
- Otherwise backup to the previous node on the current path

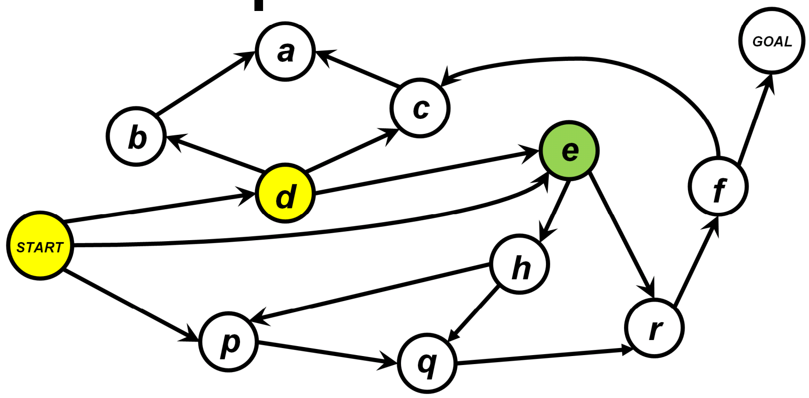
Depth First Search



General idea:

- **Expand the most recently expanded node if it has successors**
- **Otherwise backup to the previous node on the current path**

Depth First Search



General idea:

- **Expand the most recently expanded node if it has successors**
- **Otherwise backup to the previous node on the current path**

DFS Implementation

DFS (*s*)

if *s* = *GOAL*

return *SUCCESS*

else

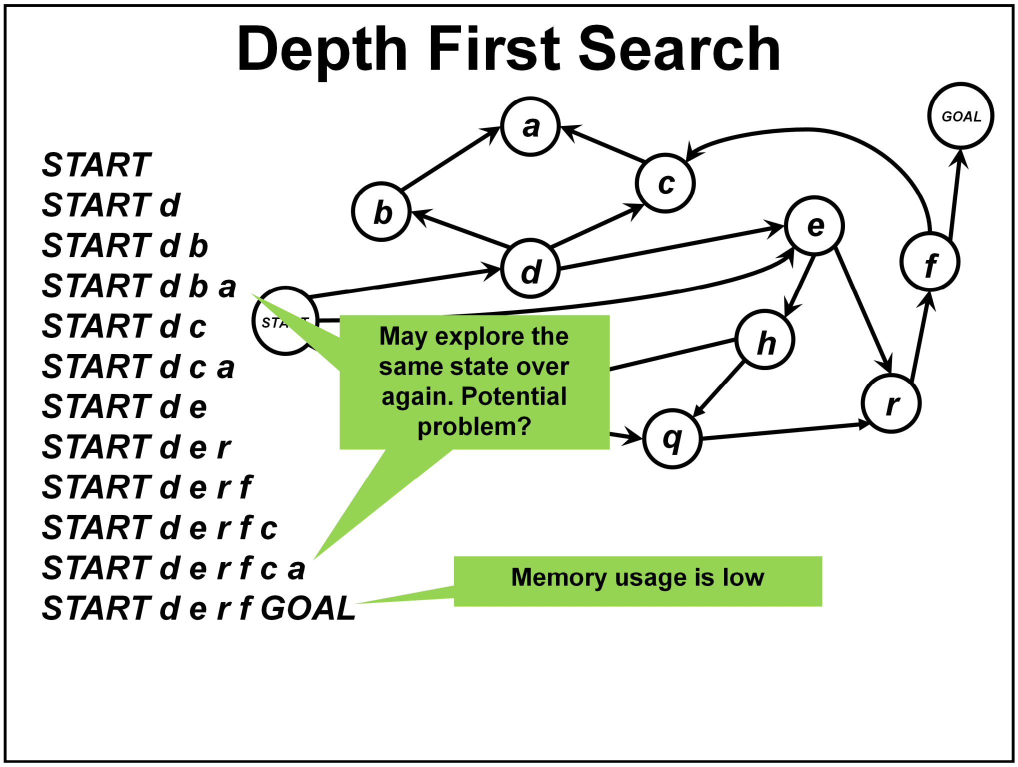
For all *s*′ in succs(*s*)

DFS (*s*′)

return *FAILURE*

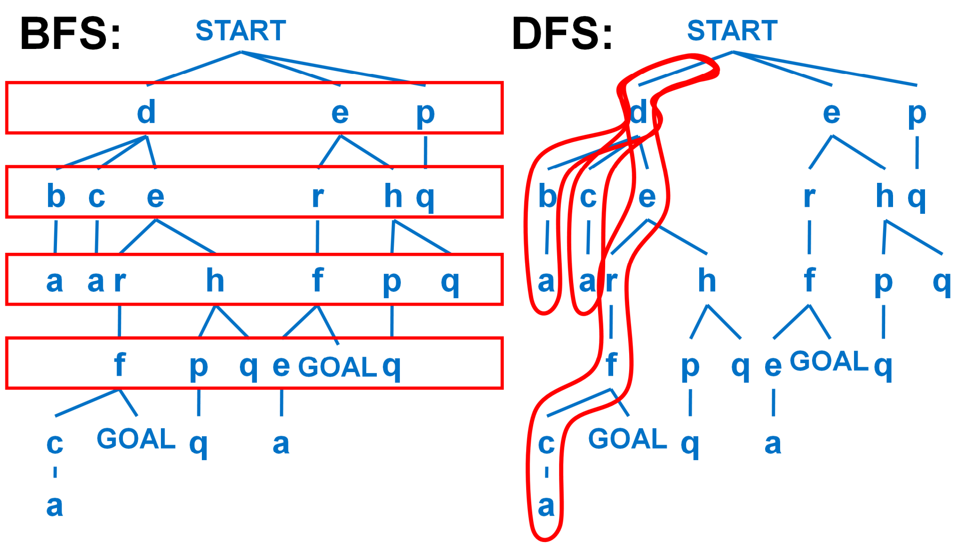
In a recursive implementation, the program stack keeps track of the states in the current path

s* is current state being expanded, starting with *START



What’s the problem? Cycles.

Didn’t just remember path (in the stack), also remembered which successor each node has already tried.



- Root: ***START*** **state**
- **Children of node *s*: All states in succs(*s*)**
- **In the worst case the entire tree is explored → O($B^{L_{max}}$)**
- **Infinite branches if there are loops in the graph!**

Complexity

N **= Total number of states**
B **= Average number of successors (branching factor)**
L **= Length from start to goal on shortest path**
C **= Cost of optimal path**
Q **= Average size of the priority queue**
Lmax **= Length of longest path from *START* to any state**

	Algorithm	Complete	Optimal	Time	Space
BFS	Breadth First Search	Y	Y, if all trans. have same cost	$O(B^L)$	$O(B^L)$
BIBFS	Bi-directional Breadth First Search	Y	Y, if all trans. have same cost	$O(2B^{L/2})$	$O(2B^{L/2})$
UCS	Uniform Cost Search	Y, if cost > 0	Y, if cost > 0	$O(\log(Q)*B^{C/\varepsilon})$	$O(B^{C/\varepsilon})$
DFS	Depth First Search				

Complexity

N **= Total number of states**
B **= Average number of successors (branching factor)**
L **= Length from start to goal on shortest path**
C **= Cost of optimal path**
Q **= Average size of the priority queue**
Lmax **= Length of longest path from *START* to any state**

	Algorithm	Complete	Optimal	Time	Space
BFS	Breadth First Search	Y	Y, if all trans. have same cost	$O(B^L)$	$O(B^L)$
BIBFS	Bi-directional Breadth First Search	Y	Y, if all trans. have	$O(2B^{L/2})$	$O(2B^{L/2})$
UCS	Uniform Cost Search	Y, if cost 0		og(Q)*$B^{C/\varepsilon}$)	$O(B^{C/\varepsilon})$
DFS	Depth First Search	Y	N	$O(B^{Lmax})$	$O(BL_{max})$

For graphs without cycles

DFS Limitation 1

- Need to prevent DFS from looping
- Avoid visiting the same states repeatedly
  - PC-DFS (Path Checking DFS):
    - Don’t use a state that is already in the current path
  - MEMDFS (Memorizing DFS):
    - Keep track of all the states expanded so far. Do not expand any state twice
  - Comparison PC-DFS vs. MEMDFS?

Which should you do?

Current path: at least we won’t loop forever, and better for memory usage.

Memdfs means you’d have to keep track of whole graph, and that’s a lot.

Complexity

N **= Total number of states**
B **= Average number of successors (branching factor)**
L **= Length from start to goal on shortest path**
C **= Cost of optimal path**
Q **= Average size of the priority queue**
Lmax **= Length of longest path from *START* to any state**

	Algorithm	Complete	Optimal	Time	Space
BFS	Breadth First Search	Y	Y, if all trans. have same cost	$O(B^L)$	$O(B^L)$
BIBFS	Bi- Direction. BFS	Y	Y, if all trans. have same cost	$O(2B^{L/2})$	$O(2B^{L/2})$
UCS	Uniform Cost Search	Y, if cost > 0	Y, if cost > 0	$O(\log(Q)*B^{C/\varepsilon})$	$O(B^{C/\varepsilon})$
PCDFS	Path Check DFS				
MEMDFS	Memorizing DFS				

Complexity

N **= Total number of states**
B **= Average number of successors (branching factor)**
L **= Length from start to goal on shortest path**
C **= Cost of optimal path**
Q **= Average size of the priority queue**
Lmax **= Length of longest path from *START* to any state**

	Algorithm	Complete	Optimal	Time	Space
BFS	Breadth First Search	Y	Y, if all trans. have same cost	$O(B^L)$	$O(B^L)$
BIBFS	Bi- Direction. BFS	Y	Y, if all trans. have same cost	$O(2B^{L/2})$	$O(2B^{L/2})$
UCS	Uniform Cost Search	Y, if cost > 0	Y, if cost > 0	$O(\log(Q)*B^{C/\varepsilon})$	$O(B^{C/\varepsilon})$
PCDFS	Path Check DFS	Y	N	$O(B^{Lmax})$	$O(BL_{max})$
MEMDFS	Memorizing DFS	Y	N	$O(B^{Lmax})$	$O(B^{Lmax})$

In PCDFS you can visit same node twice (though not in the same path). So no luxury of N

PCDFS is same storage as DFS (we're already storing the path in the stack)

MEMDFS you would have to store whole graph. However, the time complexity could be as low as N.

DFS Limitation 2

- **Need to make DFS optimal**

DFS Limitation 2

- Need to make DFS optimal

"Depth-Limited Search"

- IDS (Iterative Deepening Search):
  - Run DFS by searching only paths of length 1 (DFS stops if length of path is greater than 1)
  - If that doesn't find a solution, try again by running DFS on paths of length 2 or less
  - If that doesn't find a solution, try again by running DFS on paths of length 3 or less
  - …
  - Continue until a solution is found

Iterative Deepening Search

Iterative Deepening Search

- Sounds horrible: We need to run DFS many times
- Actually not a problem:

$$O(LB^1+(L-1)B^2+\ldots+B^L) = O(B^L)$$

Nodes generated at depth 1

Nodes generated at depth 2

Nodes generated at depth *L*

- Compare B^L and B^{Lmax}
- Optimal if transition costs are equal

Why does it not matter to do the same stuff over and over? Exponential growth means the first levels aren't all that bad.

Iterative Deepening Search

- Memory usage same as DFS
- Computation cost comparable to BFS even with repeated searches, especially for large *B*.
- Example:
  - *B* = 10, *L* = 5
  - BFS: 111,111 expansions
  - IDS: 123,456 expansions

N **= Total number of states**
B **= Average number of successors (branching factor)**
L **= Length from start to goal on shortest path**
C **= Cost of optimal path**
Q **= Average size of the priority queue**
Lmax **= Length of longest path from *START* to any state**

	Algorithm	Complete	Optimal	Time	Space
BFS	Breadth First Search	Y	Y, if all trans. have same cost	$O(B^L)$	$O(B^L)$
BIBFS	Bi- Direction. BFS	Y	Y, if all trans. have same cost	$O(2B^{L/2})$	$O(2B^{L/2})$
UCS	Uniform Cost Search	Y, if cost > 0	Y, If cost > 0	$O(\log(Q)*B^{C/\varepsilon})$	$O(B^{C/\varepsilon})$
PCDFS	Path Check DFS	Y	N	$O(B^{Lmax})$	$O(BL_{max})$
MEMD FS	Memorizing DFS	Y	N	$O(B^{Lmax})$	$O(B^{Lmax})$
IDS	Iterative Deepening	Y	Y, If all trans. have same cost	$O(B^L)$	$O(BL)$

Summary

- Basic search techniques: BFS, UCS, PCDFS, MEMDFS, …
- Property of search algorithms: Completeness, optimality, time and space complexity
- Iterative deepening and bidirectional search ideas
- Trade-offs between the different techniques and when they might be used