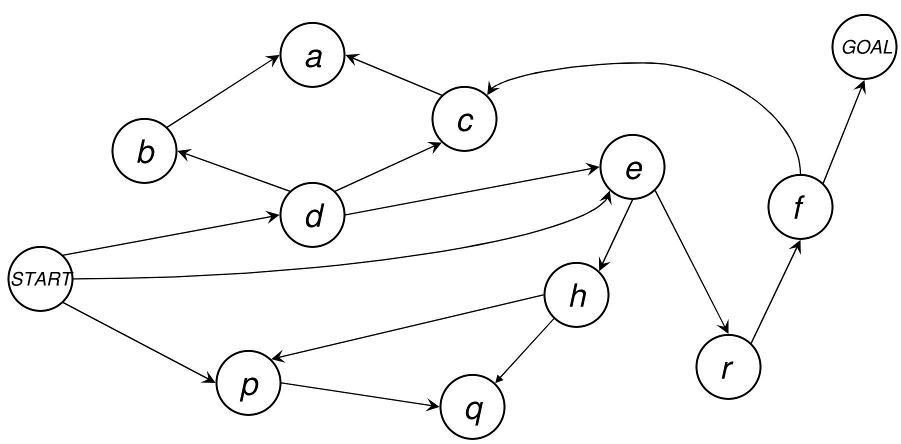
Search: Uninformed Search

Russel & Norvig Chap. 3

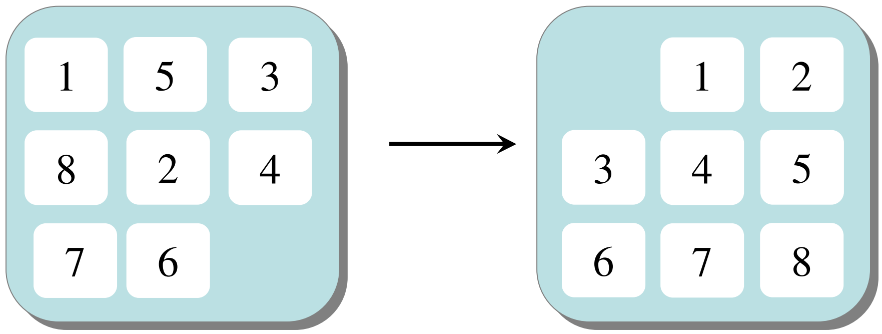
Material in part from http://www.cs.cmu.edu/~awm/tutorials

A Search Problem



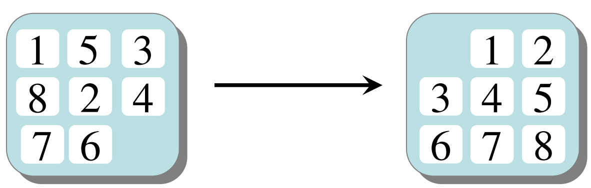
- Find a path from START to GOAL
- Find the minimum number of transitions

Example



START *GOAL*

Example

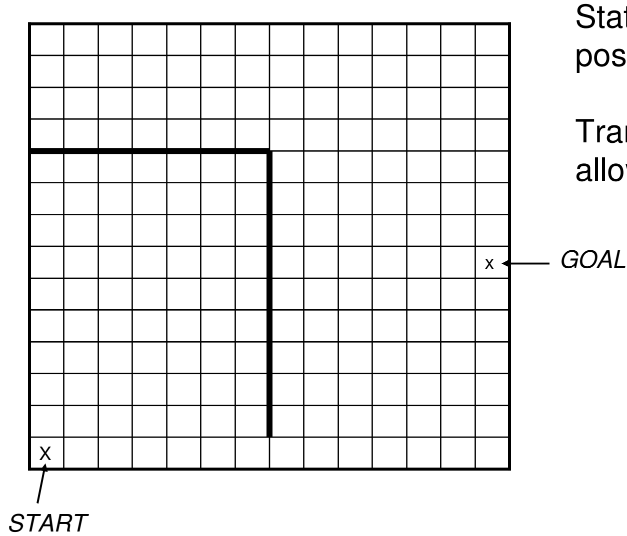


START *GOAL*

- State: Configuration of puzzle
- Transitions: Up to 4 possible moves (*up*, *down*, *left*, *right*)
- Solvable in 22 steps (average)
- But: 1.8 10^5 states (1.3 10^{12} states for the 15-puzzle)

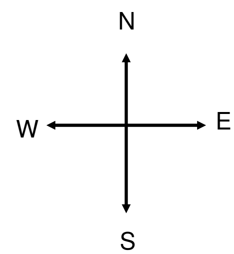
  → Cannot represent set of states explicitly

Example: Robot Navigation



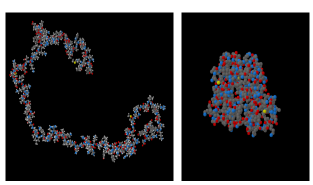
States =
positions in the map

Transitions =
allowed motions



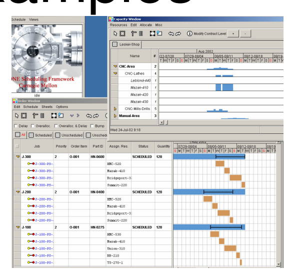
Navigation: Going from point START to point GOAL given a (deterministic) map

Other Real-Life Examples



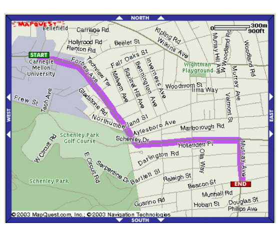
Protein design

http://www.blueprint.org/proteinfolding/trades/trades_problem.html

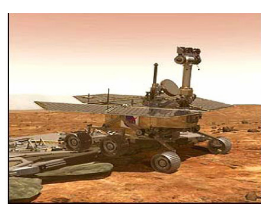


Scheduling/Manufacturing

http://www.ozone.ri.cmu.edu/projects/dms/dmsmain.html

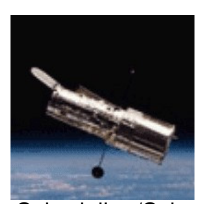


Route planning



Robot navigation

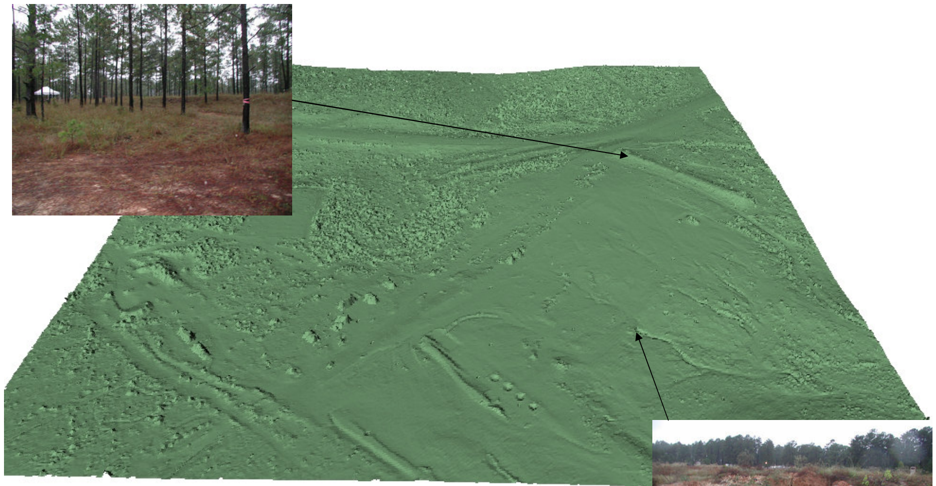
http://www.frc.ri.cmu.edu/projects/mars/dstar.html



Scheduling/Science

http://www.ozone.ri.cmu.edu/projects/hsts/hstsmain.html

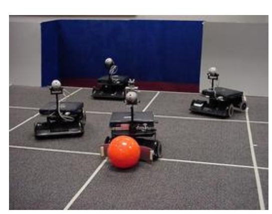
Don't necessarily know explicitly the structure of a search problem



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

What we are *not* addressing (yet)

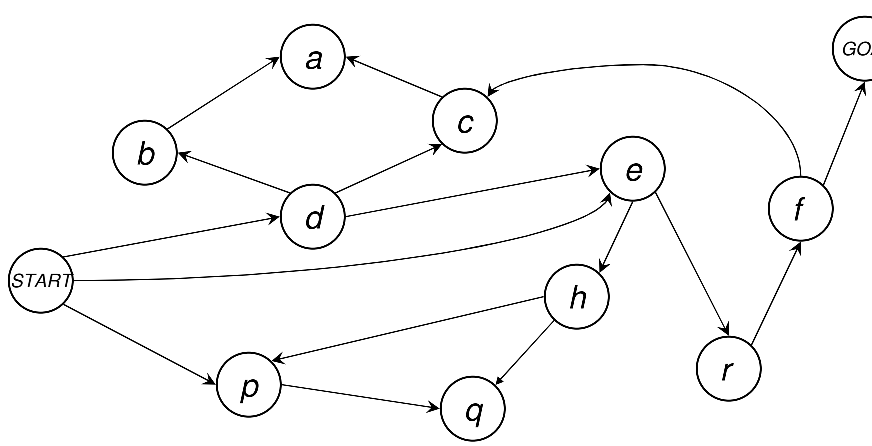
- Uncertainty/Chance → State and transitions are known and deterministic
- Game against adversary
- Multiple agents/Cooperation
- Continuous state space → For now, the set of states is discrete



Overview

- Definition and formulation
- Optimality, Completeness, and Complexity
- *Uninformed Search*
  - – Breadth First Search
  - – Search Trees
  - – Depth First Search
  - – Iterative Deepening
- *Informed Search*
  - – Best First Greedy Search
  - – Heuristic Search, A*

A Search Problem



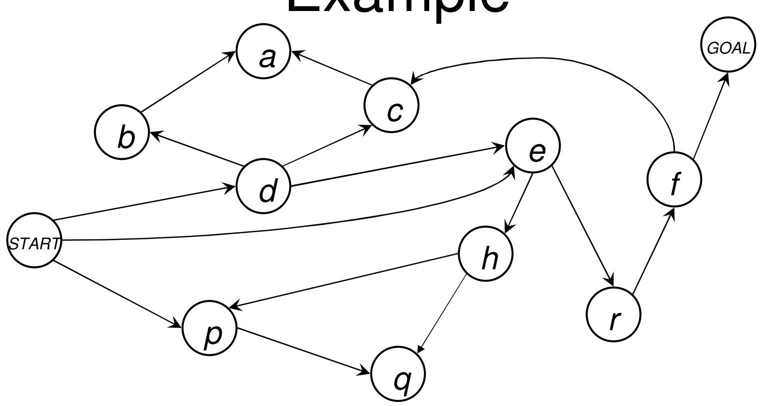
Formulation

- 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 state s to state s'

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

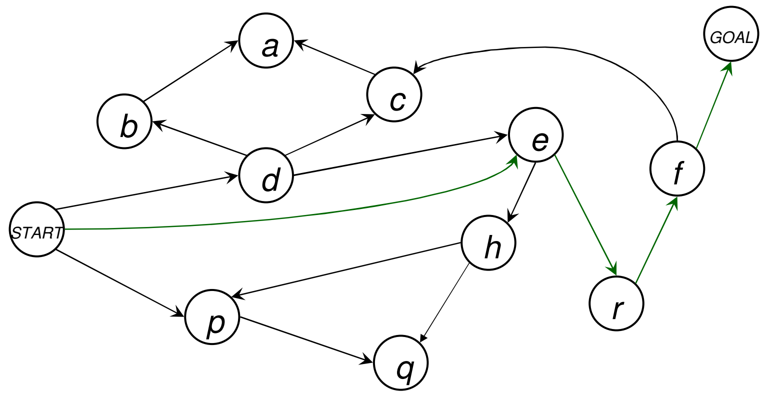
1. $s_1 \in S$
2. $s_K \in G$
3. $s_{i+1} \in$ **succs**(s_i)
4. $\sum$ **cost**(s_i, s_{i+1}) is the smallest among all possible sequences (desirable but optional)

Example



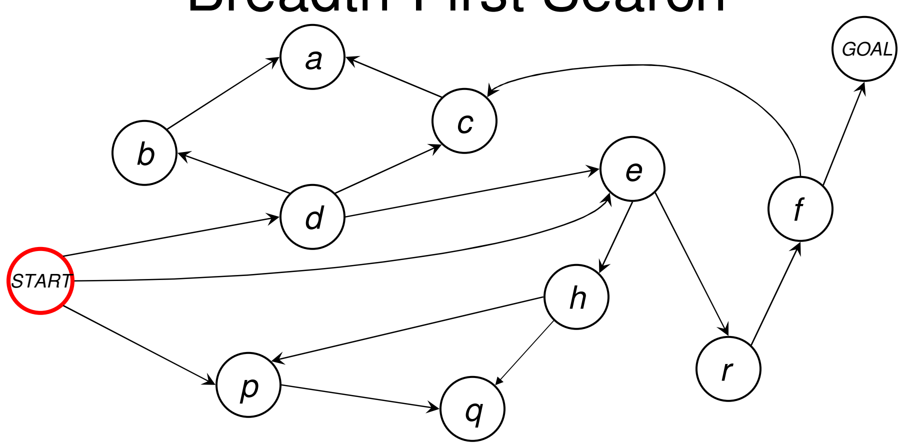
- $Q = \{START, GOAL, a, b, c, d, e, f, h, p, q, r\}$
- $S = \{START\}$ $G = \{GOAL\}$
- **succs**(d) = $\{b,c\}$
- **succs**($START$) = $\{p,e,d\}$
- **succs**(a) = NULL
- **cost**(s,s') = 1 for all transitions

Desirable Properties



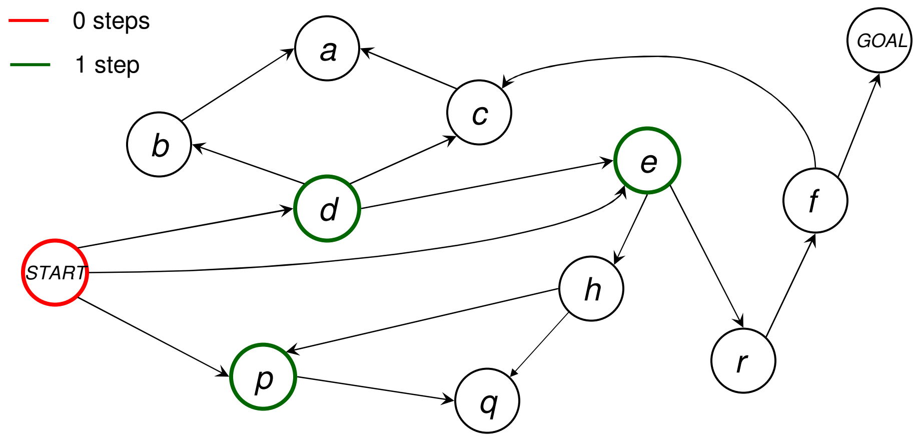
- ***Completeness*: An algorithm is complete if it is guaranteed to find a path if one exists**
- ***Optimality*: The total cost of the path is the lowest among all possible paths from start to goal**
- ***Time Complexity***
- ***Space Complexity***

Breadth-First Search



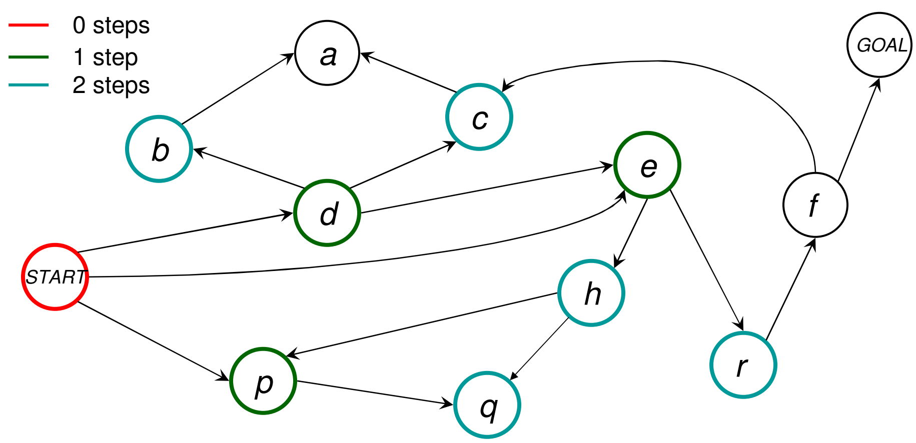
- Label all states that are 0 steps from S → Call that set V_o

Breadth-First Search



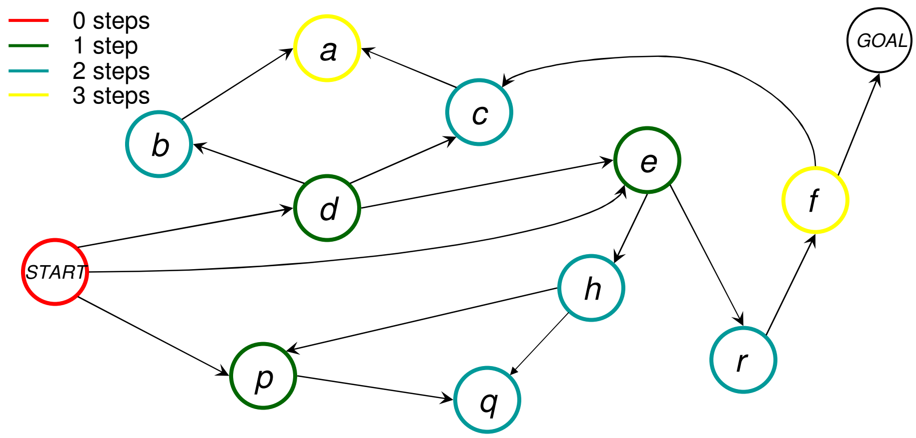
- Label the successors of the states in V_0 that are not yet labelled →Set V_1 of states that are 1 step away from the start

Breadth-First Search



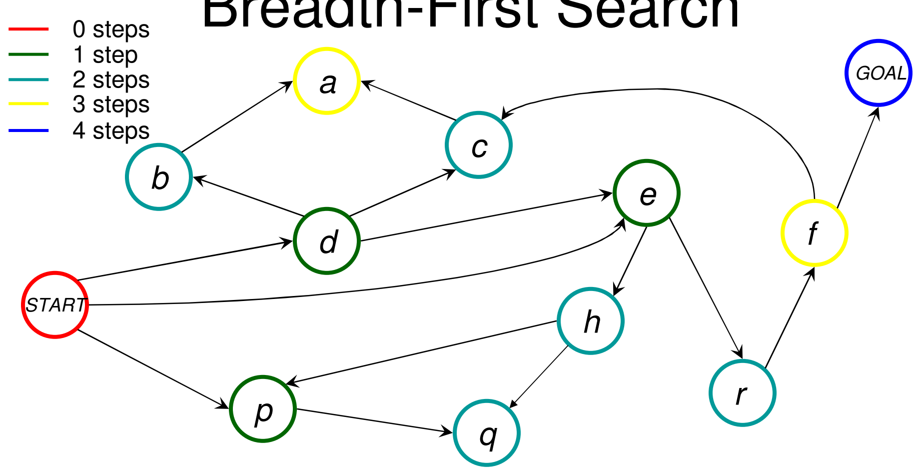
- Label the successors of the states in V_1 that are not yet labelled →Set V_2 of states that are 1 step away from the start

Breadth-First Search



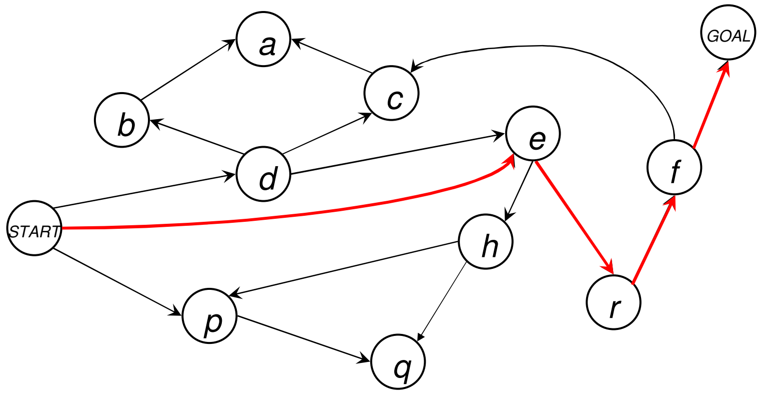
- Label the successors of the states in V_2 that are not yet labelled → Set V_3 of states that are 1 step away from the start

Breadth-First Search



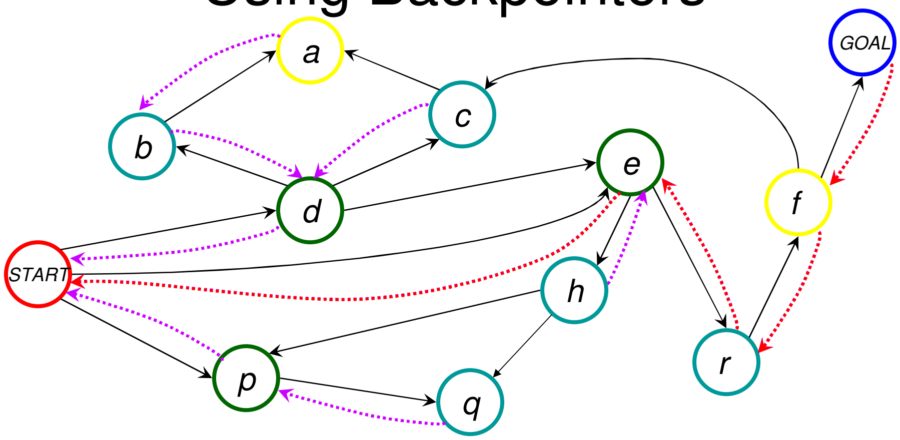
- Stop when goal is reached in the current expansion set → goal can be reached in 4 steps

Recovering the Path



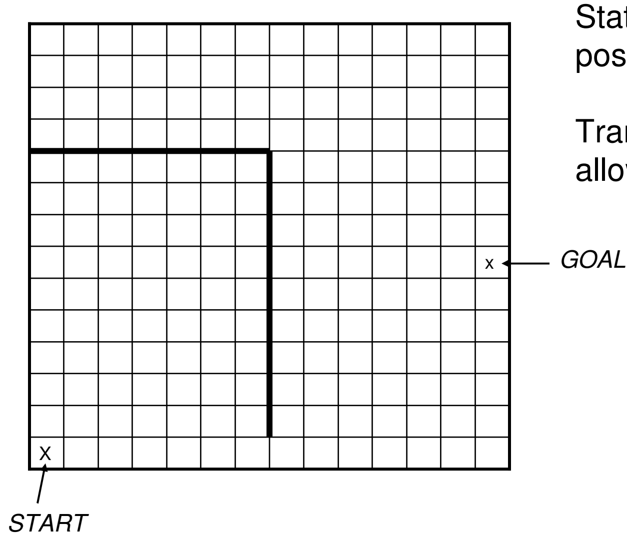
- Record the predecessor state when labeling a new state
- When I labeled *GOAL*, I was expanding the neighbors of *f* → *f* is the predecessor of *GOAL*
- When I labeled *f*, I was expanding the neighbors of *r* → *r* is the predecessor of *f*
- Final solution: {*START, e, r, f, GOAL*}

Using Backpointers



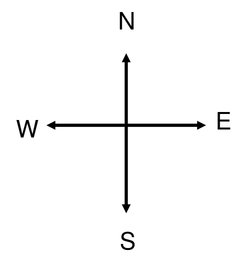
- A backpointer **previous**(*s*) point to the node that stored the state that was expanded to label *s*
- The path is recovered by following the backpointers starting at the goal state

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

$V_0 \leftarrow S$ (the set of start states)
previous(*START*) := *NULL*
$k \leftarrow 0$

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

- $V_{k+1} \leftarrow$ 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') $\leftarrow s$
      - Add s' into V_{k+1}
- $k \leftarrow k+1$

if V_k is empty signal FAILURE
else build the solution path thus:

- Define S_k = *GOAL*, and forall $i <= k$, 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??

See maze example

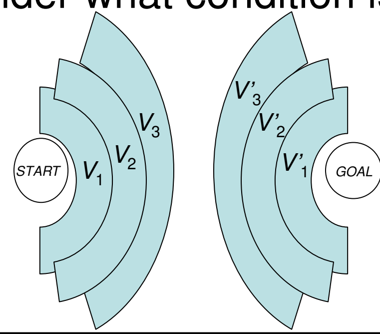
Complexity

- N = Total number of states
- B = Average number of successors (branching factor)
- L = Length from start to goal with smallest number of steps

	Algorithm	Complete	Optimal	Time	Space
BFS	Breadth First Search				

Bidirectional Search

- BFS search simultaneously forward from *START* and backward from *GOAL*
- When do the two search meet?
- What stopping criterion should be used?
- Under what condition is it optimal?



Complexity

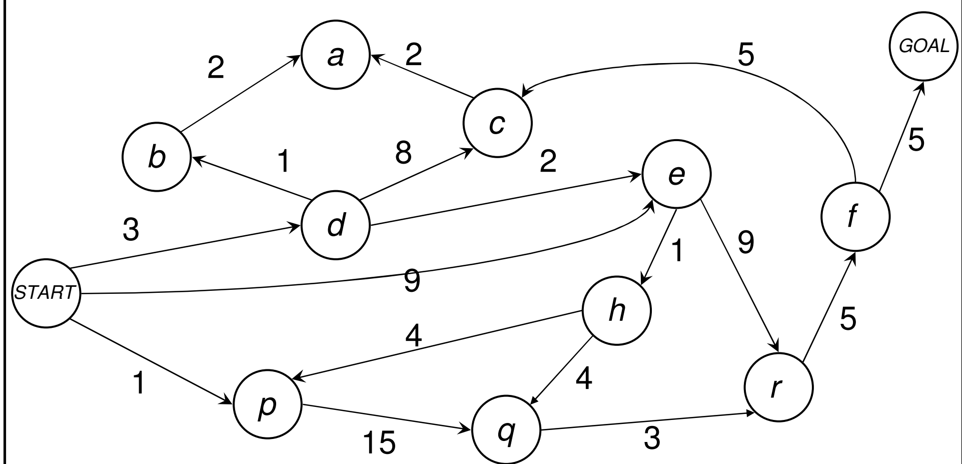
- N = Total number of states
- B = Average number of successors (branching factor)
- L = Length for start to goal with smallest number of steps

	Algorithm	Complete	Optimal	Time	Space
BFS	Breadth First Search				
BIBFS	Bi-directional Breadth First Search				

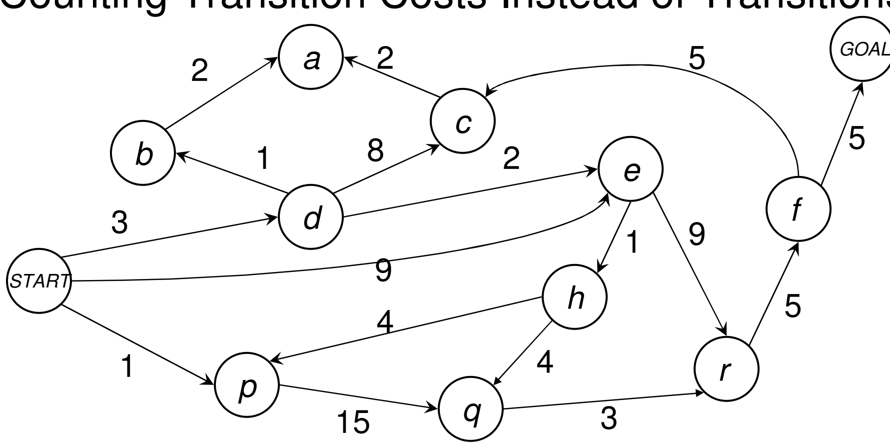
Major savings when bidirectional search is possible because
$2B^{L/2} << B^L$

B = 10, L = 6 → 22,200 states generated vs. $\sim 10^7$

Counting Transition Costs Instead of Transitions



Counting Transition Costs Instead of Transitions



- BFS finds the shortest path in number of steps but does not take into account transition costs
- Simple modification finds the least cost path
- New field: At iteration k, $\mathbf{g}(s)$ = least cost 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 item of minimum value 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**()

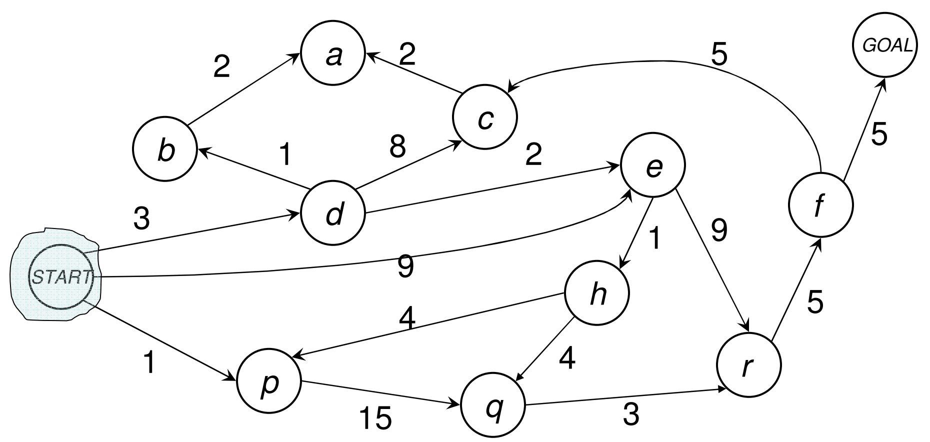
Complexity: O(log(number of pairs in PQ)) for insertion and pop operations → very efficient

http://www.leekillough.com/heaps/ Knuth&Sedwick ….

Uniform Cost Search

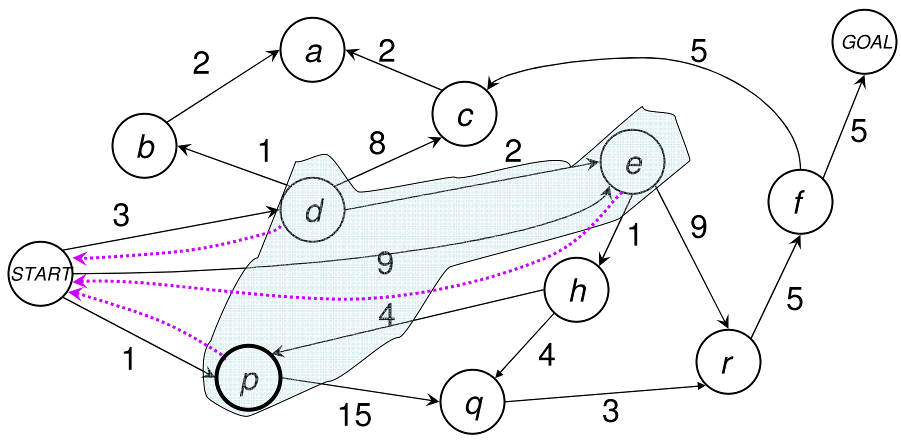
- *PQ* = Current set of evaluated states
- Value (priority) of state = **g**(*s*) = current cost of path to *s*
- Basic iteration:
  1. Pop the state *s* with the lowest path cost from *PQ*
  2. Evaluate the path cost to all the successors of *s*
  3. Add the successors of *s* to *PQ*

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



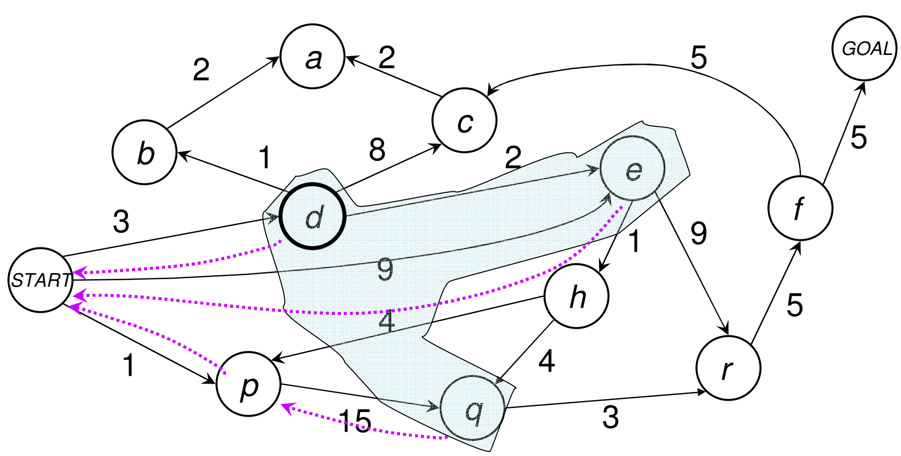
PQ = {(START,0)}

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*



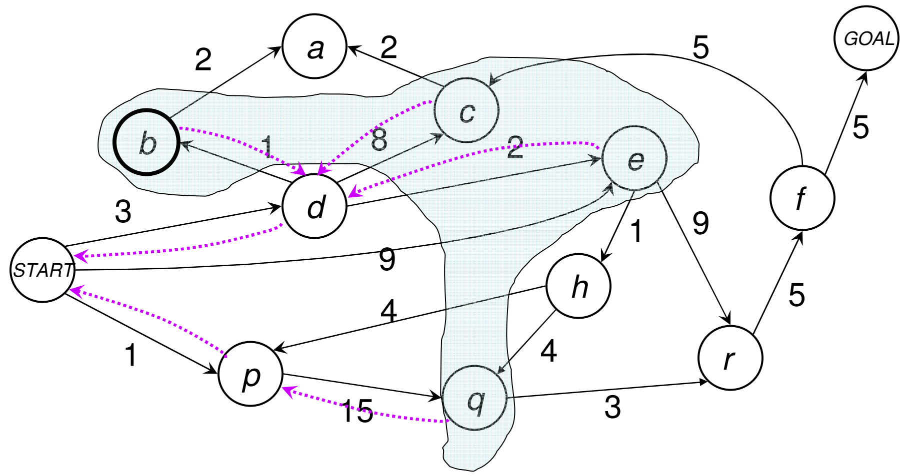
PQ = {(p,1) (d,3) (e,9)}

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* t *PQ*



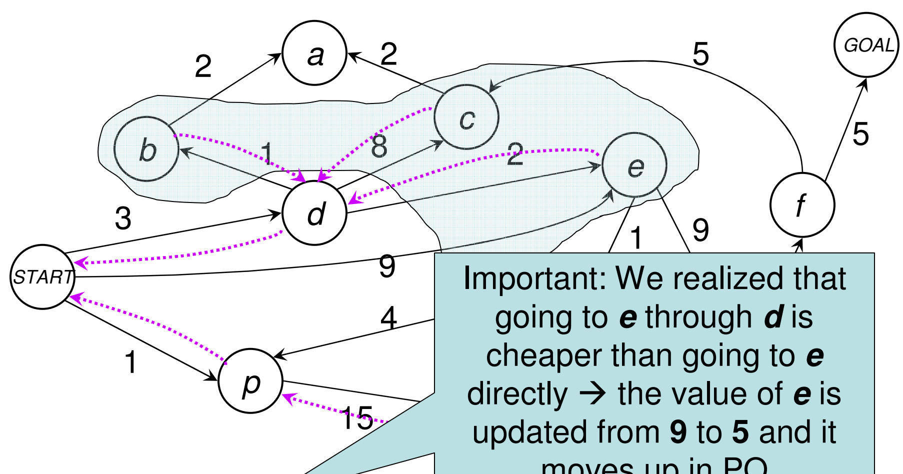
PQ = {(d,3) (e,9) (q,16)}

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* t *PQ*



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

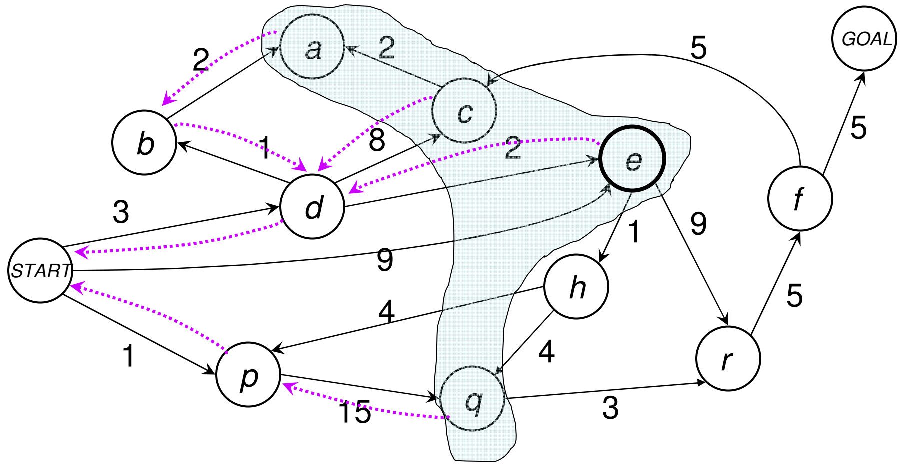
1. Pop the state *s* with the lowest path cost from *PQ*
2. Evaluate the path cost to all the successors of *s*
3. Add the successors of *s* to *PQ*



Important: We realized that going to ***e*** through ***d*** is cheaper than going to ***e*** directly → the value of ***e*** is updated from **9** to **5** and it moves up in PQ

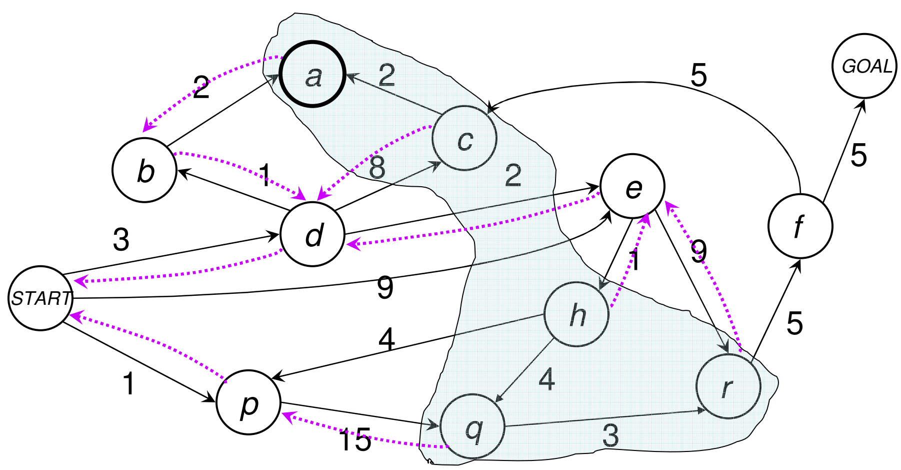
***PQ* = {(*b*,4) (*e*,5) (*c*,11) (*q*,16)}**

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*



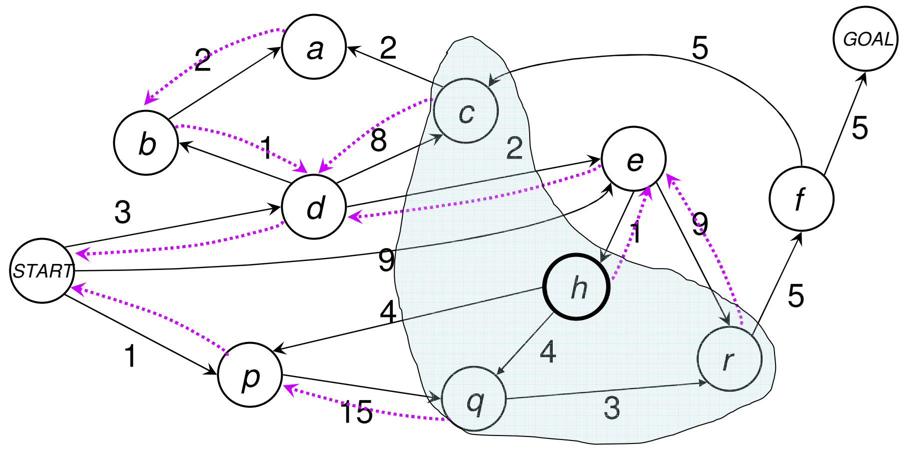
PQ = {(*e*,5) (*a*,6) (*c*,11) (*q*,16)}

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*



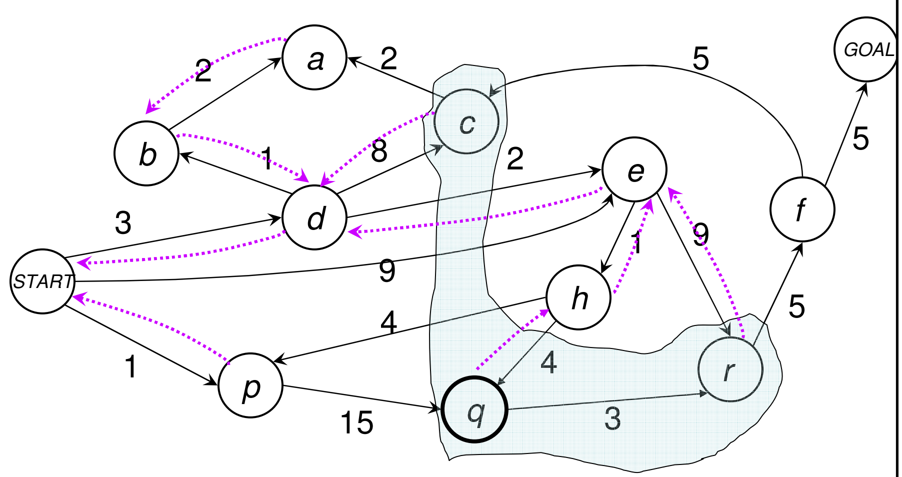
PQ = {(*a*,6) (*h*,6) (*c*,11) (*r*,14) (*q*,16)}

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*



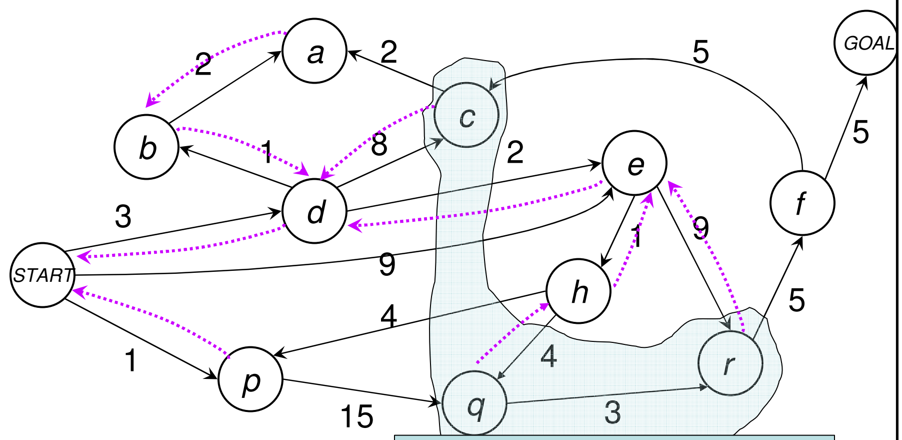
***PQ* = {(*h*,6) (*c*,11) (*r*,14) (*q*,16)}**

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* t
PQ



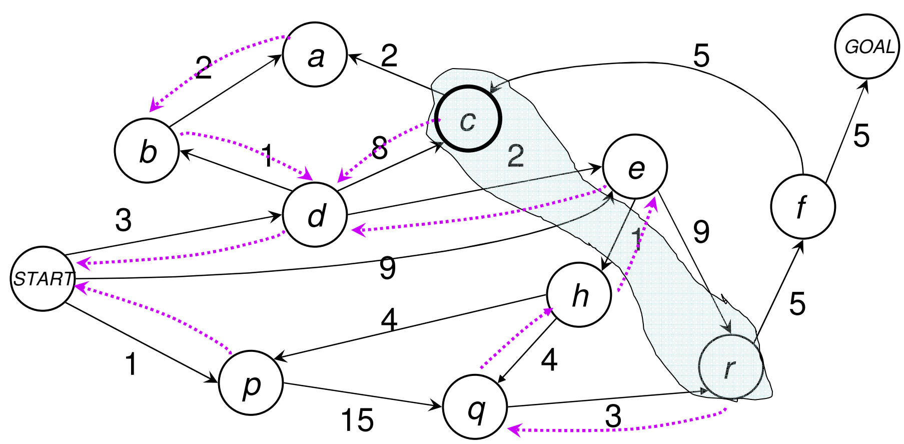
***PQ* = {(*q*,10) (*c*,11) (*r*,14)}**

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*



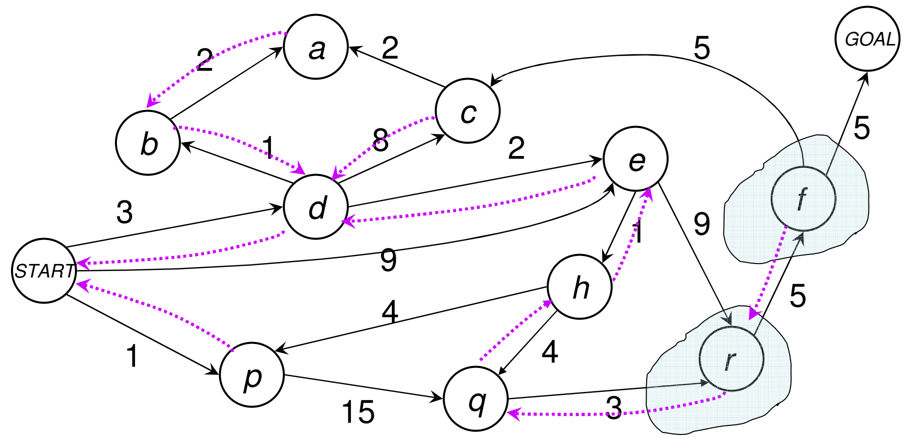
***PQ* = {(*q*,10) (*c*,11) (...**

Important: We realized that going to ***q*** through ***h*** is cheaper than going through ***p*** → the value of ***q*** is updated from **16** to **10** and it moves up in PQ



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

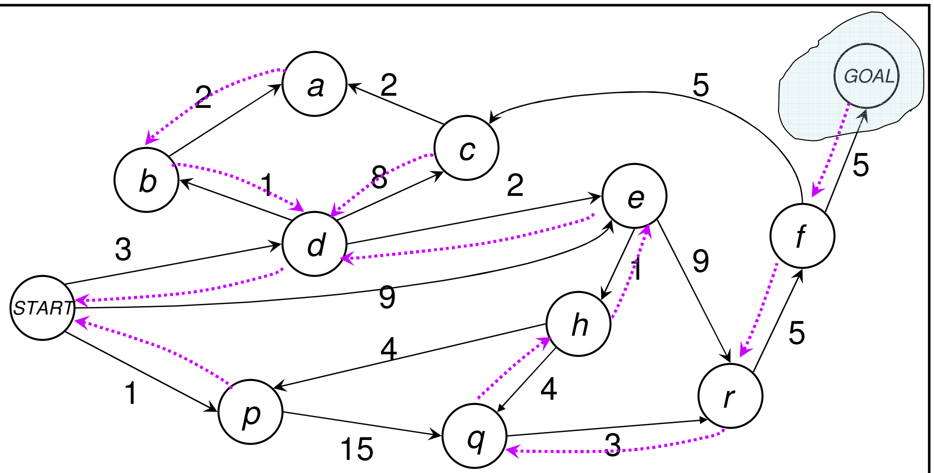
1. Pop the state *s* with the lowest path cost from *PQ*
2. Evaluate the path cost to all the successors of *s*
3. Add the successors of *s* to *PQ*



PQ = {(*r*,13)}

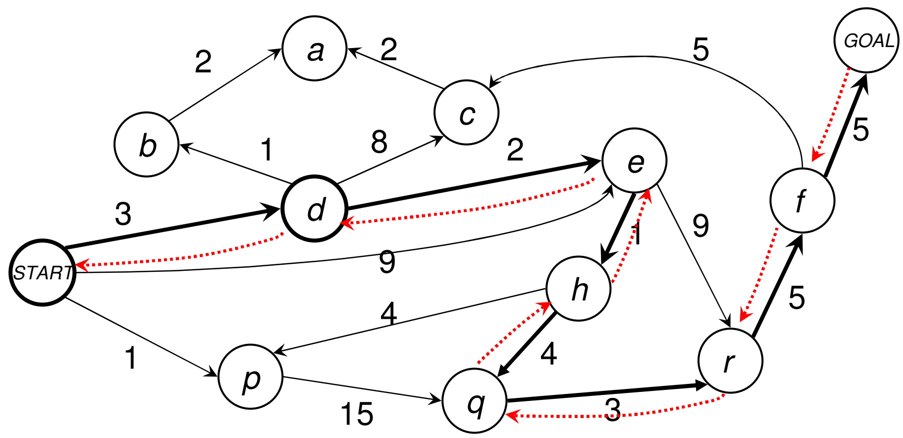
PQ = {(*f*,18)}

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)}

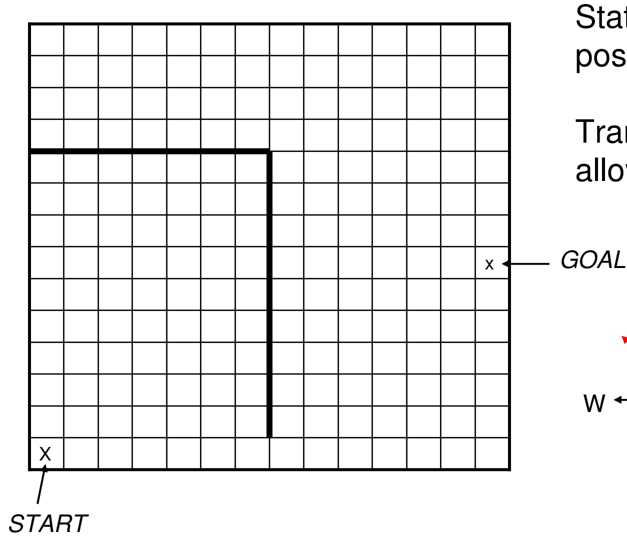
1. Pop the state *s* with the lowest path cost from *PQ*
2. Evaluate the path cost to all the successors of *s*
3. Add the successors of *s* to *PQ*



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

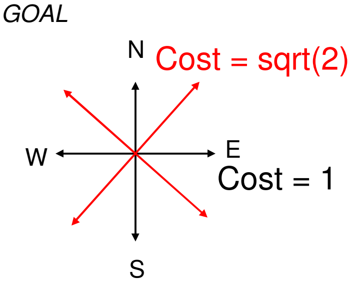
- This path is optimal in total cost even though it has more transitions than the one found by BFS
- What should be the stopping condition?
- Under what conditions is UCS complete/optimal?

Example: Robot Navigation



States = positions in the map

Transitions = allowed motions



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

Complexity

- N = Total number of states
- B = Average number of successors (branching factor)
- L = Length for start to goal with smallest number of steps
- Q = Average size of the priority queue

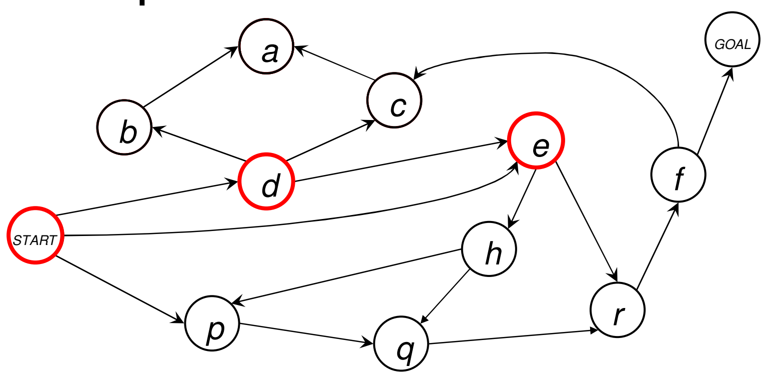
	Algorithm	Complete	Optimal	Time	Space
BFS	Breadth First Search				
BIBFS	Bi-directional Breadth First Search				
UCS	Uniform Cost Search				

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

Depth First Search

START
START *d*
START *d b*
START *d b a*
START *d c*
START *d c a*
START *d e*



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

DFS Implementation

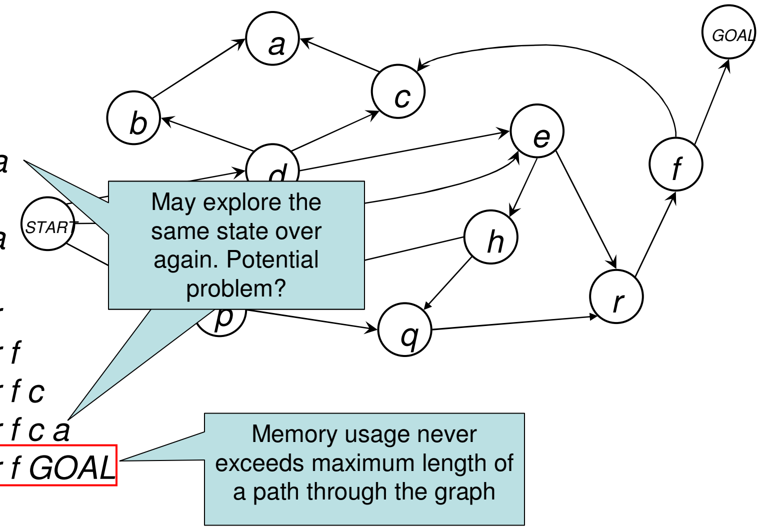
DFS (*s*)
if *s* = *GOAL*
return *SUCCESS*
else
For all *s'* in **succs**(*s*)
DFS (*s'*)
return *FAILURE*

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

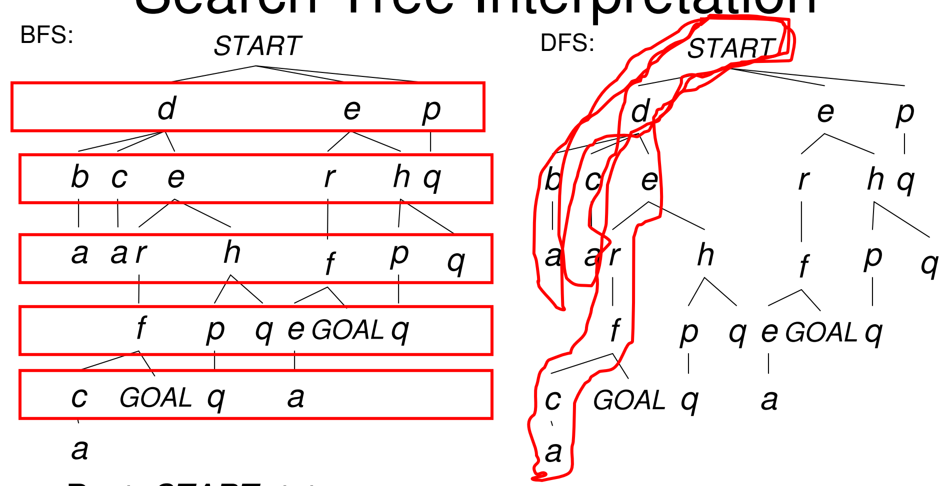
s is current state being expanded, starting with *START*

Depth First Search

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



Search Tree Interpretation



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

Complexity

- N = Total number of states
- B = Average number of successors (branching factor)
- L = Length for start to goal with smallest number of steps
- 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				
BIBFS	Bi-directional Breadth First Search				
UCS	Uniform Cost Search				
DFS	Depth First Search				

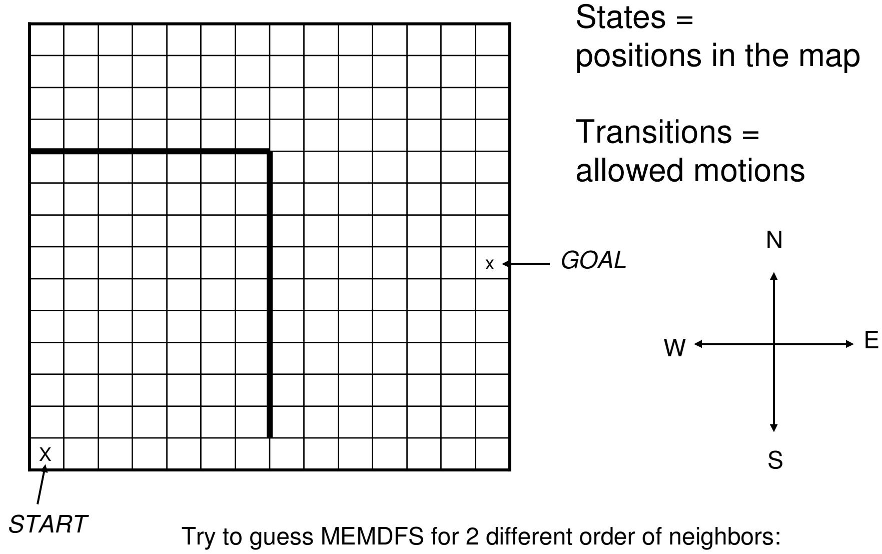
DFS Limitation 1

- Need to prevent DFS from looping
- Avoid visiting the same states repeatedly

Because B^d may be much larger than the number of states d steps away from the start

- 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?

Example: Robot Navigation



States =
positions in the map

Transitions =
allowed motions

Try to guess MEMDFS for 2 different order of neighbors:
E, N, W, S
W, E, N, S

Complexity

- *N* = Total number of states
- *B* = Average number of successors (branching factor)
- *L* = Length for start to goal with smallest number of steps
- *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				
BIBFS	Bi- Direction. BFS				
UCS	Uniform Cost Search				
PCDFS	Path Check DFS				
MEMD FS	Memorizing DFS				

DFS Limitation 2

- Need to make DFS optimal

"Depth-Limited Search"

- IDS (Iterative Deepening Search):
  - Run DFS by searching only path 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

- 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

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

Complexity

- *N* = Total number of states
- *B* = Average number of successors (branching factor)
- *L* = Length for start to goal with smallest number of steps
- *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				
BIBFS	Bi- Direction. BFS				
UCS	Uniform Cost Search				
PCDFS	Path Check DFS				
MEMDFS	Memorizing DFS				
IDS	Iterative Deepening				

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