CS 188: Artificial Intelligence
Spring 2006

Lecture 3: Search
1/24/2006

Dan Klein – UC Berkeley
Many slides from either Stuart Russell or Andrew Moore

Administrivia

- Final Exam Issues
- Sections: starting 1/30
  - 101 (M 10am): B0051 Hildebrand
  - 104 (M 4pm): 002 Evans
- Written Assignment 1
  - Reminder: can solve in groups
  - But: individual write ups

Today

- Formulating Search Problems
- Uniformed Search
  - Depth First Search
  - Breadth First Search
  - Uniform Cost Search
- Properties of Search Algorithms

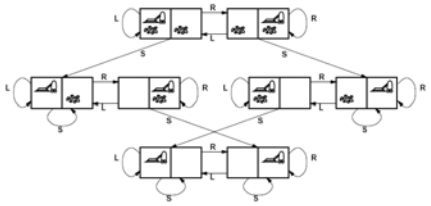
Example: Romania

Single State Problems

- A search problem is defined by four items:
  - Initial state: e.g. Arad
  - Successor function S(x) = set of action–state pairs:
    e.g., S(Arad) = {<Arad → Zerind, Zerind>, … }
  - Goal test, can be
    - explicit, e.g., x = Bucharest
    - implicit, e.g., Checkmate(x)
  - Path cost (additive)
    - e.g., sum of distances, number of actions executed, etc.
    - c(x,a,y) is the step cost, assumed to be ≥ 0
- A solution is a sequence of actions leading from the initial state to a goal state

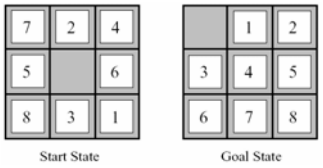
Search Gone Wrong?

Example: Vacuum World



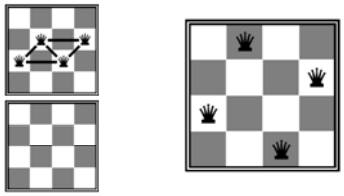
- Can represent problem as a state graph
  - Nodes are states
  - Arcs are actions
  - Arc weights are costs

Example: 8-Puzzle



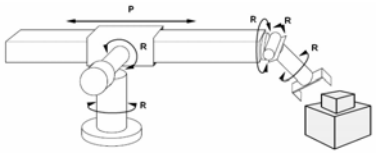
- What are the states?
- What are the actions?
- What states can I reach from the start state?
- What should the costs be?

Example: N-Queens



- What are the states?
- What is the start?
- What is the goal?
- What are the actions?
- What should the costs be?

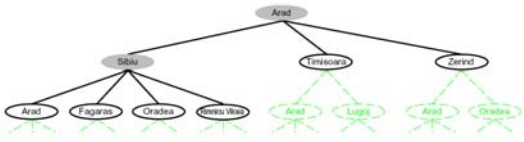
Example: Assembly



- What are the states?
- What is the goal?
- What are the actions?
- What should the costs be?

Example: Romania

A Search Tree



- Search trees:
  - Represent the branching paths through a state graph.
  - Usually **much** larger than the state graph.
  - Can a finite state graph give an infinite search tree?

Tree Search

- Basic solution method for graph problems
  - Offline simulated exploration of state space
  - Searching a model of the space, not the real world

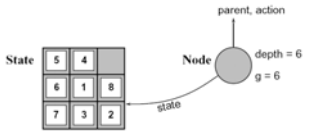
```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

Tree Search

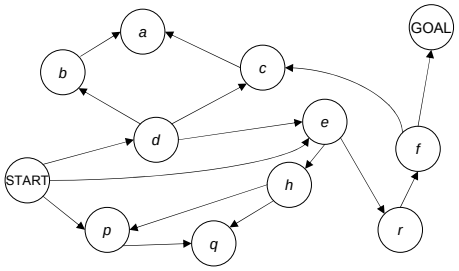
```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE(node)) then return SOLUTION(node)
        fringe ← INSERTALL(EXPAND(node, problem), fringe)

function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set; state ← STATE[node]
    for each action, result in SUCCESSOR-FN(problem, state) do
        s ← a new NODE
        PARENT-NODE[s] ← node; ACTION[s] ← action; STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node]+STEP-COST(state, action, result)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

States vs. Nodes

- Problem graphs have problem states
  - Have successors
- Search trees have search nodes
  - Have parents, children, depth, path cost, etc.
  - Expand uses successor function to create new search tree nodes
  - The same problem state may be in multiple search tree nodes



A Search Graph

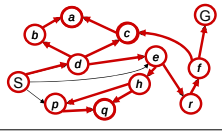


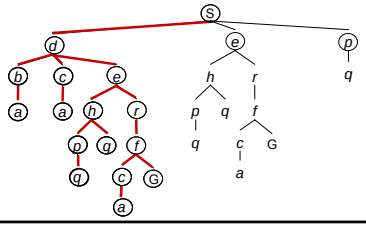
How do we get from S to G? And what's the smallest possible number of transitions?

Depth First Search

Expand deepest node first:

Fringe is a LIFO stack





Search Algorithm Properties

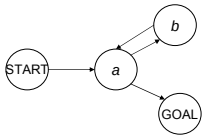
- Complete? Guaranteed to find a solution if one exists?
- Optimal? Guaranteed to find the least cost path?
- Time complexity?
- Space complexity?

Variables:

n	Number of states in the problem
b	The average branching factor B (the average number of successors)
C^*	Cost of least cost solution
s	Depth of the shallowest solution
m	Max depth of the search tree

DFS

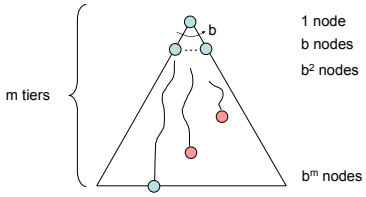
Algorithm		Complete	Optimal	Time	Space
DFS	Depth First Search	N	N	Infinite	Infinite



- Infinite paths make DFS incomplete…
- How can we fix this?

DFS

- With cycle checking, DFS is complete.



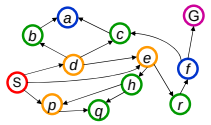
Algorithm		Complete	Optimal	Time	Space
DFS	w/ Path Checking	Y	N	$O(b^{m+1})$	$O(bm)$

- When is DFS optimal?

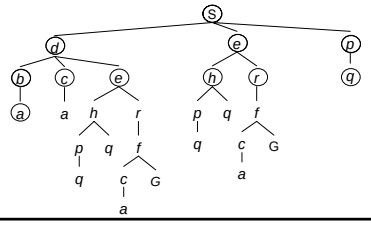
Breadth First Search

Expand shallowest node first:

Fringe is a FIFO queue

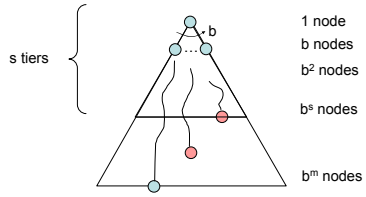


Search Tiers



BFS

Algorithm		Complete	Optimal	Time	Space
DFS	w/ Path Checking	Y	N	$O(b^{m+1})$	$O(bm)$
BFS		Y	N*	$O(b^{s+1})$	$O(b^s)$

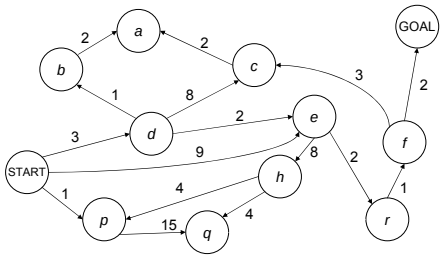


- When is BFS optimal?

Comparisons

- When will BFS outperform DFS?
- When will DFS outperform BFS?

Costs on Actions



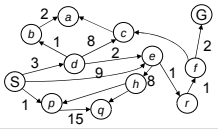
Notice that BFS finds the shortest path in terms of number of transitions. It does not find the least-cost path.

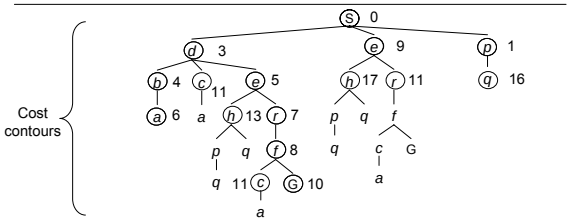
We will quickly cover an algorithm which does find the least-cost path.

Uniform Cost Search

Expand cheapest node first:

Fringe is a priority queue





Priority Queue Refresher

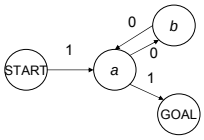
- A priority queue is a data structure in which you can insert and retrieve (key, value) pairs with the following operations:

pq.setPriority(key, value)	inserts *(key, value)* into the queue.
pq.dequeue()	returns the key with the lowest value, and removes it from the queue.

- You can promote or demote keys by resetting their priorities
- Unlike a regular queue, insertions into a priority queue are not constant time, usually O(log *n*)
- We'll need priority queues for most cost-sensitive search methods.

Uniform Cost Search

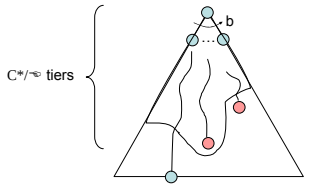
- What will UCS do for this graph?



- What does this mean for completeness?

Uniform Cost Search

Algorithm		Complete	Optimal	Time	Space
DFS	w/ Path Checking	Y	N	$O(b^{m+1})$	$O(bm)$
BFS		Y	N	$O(b^{s+1})$	$O(b^s)$
UCS		Y*	Y	$O(C^* b^{C^*/\varepsilon})$	$O(b^{C^*/\varepsilon})$



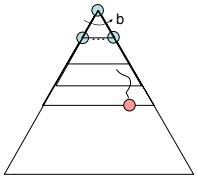
We'll talk more about uniform cost search's failure cases next class…

Iterative Deepening

Iterative deepening uses DFS as a subroutine:

1. Do a DFS which only searches for paths of length 1 or less. (DFS gives up on 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.

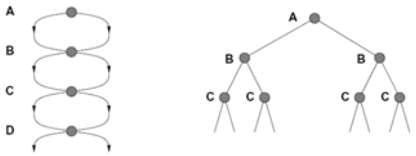
….and so on.



Algorithm		Complete	Optimal	Time	Space
DFS	w/ Path Checking	Y	N	$O(b^{m+1})$	$O(bm)$
BFS		Y	N*	$O(b^{s+1})$	$O(b^s)$
ID		Y	N*	$O(b^{d+1})$	$O(bd)$

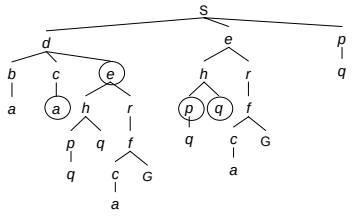
Extra Work?

- Failure to detect repeated states can cause exponentially more work. Why?



Graph Search

- In BFS, for example, we shouldn't bother expanding the circled nodes (why?)



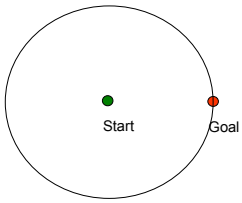
Graph Search

- Very simple fix: never expand a node twice

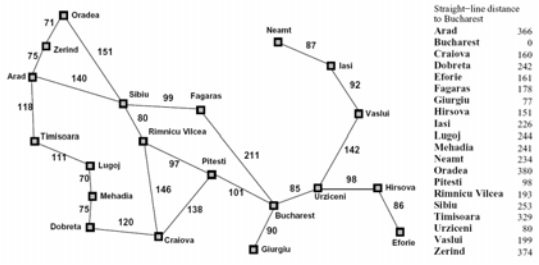
```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
    end
```

Uniform Cost Issues

- Where will uniform cost explore?
- Why?
- What is wrong here?

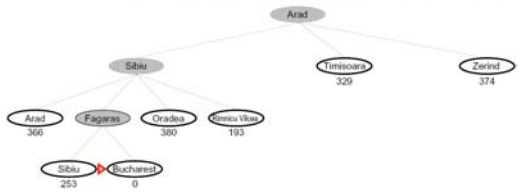


Greedy Search



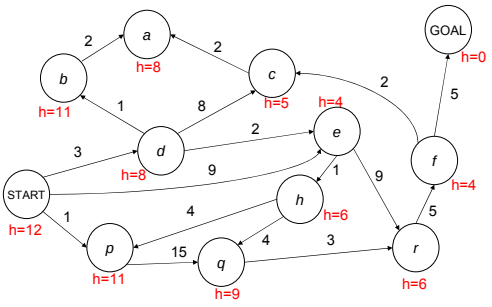
Greedy Search

- Expand the node that seems closest…



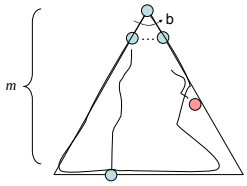
- What can go wrong?

Euclidian Heuristic



Best First Greedy Search

Algorithm	Complete	Optimal	Time	Space
Greedy Best-First Search	Y*	N	$O(b^m)$	$O(b^m)$



- Can we make it optimal? Next class!