CS 771
Artificial Intelligence
Problem Solving by Searching
Uninformed search
Complete architectures for intelligence?

• Search?
  – Solve the problem of what to do.

• Learning?
  – Learn what to do.

• Logic and inference?
  – Reason about what to do.
  – Encoded knowledge/”expert” systems?
    • Know what to do.

• Modern view: It’s complex & multi-faceted.
Search

• Formulate “What to do?” as a search problem.
  – Solution to the problem tells agent what to do.
• What if no solution in the current search space?
  – Formulate and solve the problem of finding a search space that does contain a solution.
  – Solve original problem in the new search space.
• Many powerful extensions to these ideas.
  – Constraint satisfaction; means-ends analysis; planning; game playing; etc.
• Human problem-solving often looks like search.
Why search?

• Suppose the agent has a map of Romania
  – Wants to travel from one city to another

• What properties does the task environment have?
  – Observable
    • Agent always knows the current state
  – Discrete
    • At any given state there are only finite actions to choose from
  – Known
    • Agent knows which states are reached from each state
  – Deterministic
    • Each action has only one outcome
Why search?

• To achieve goals or to maximize our utility we need to predict what the result of our actions in the future will be

• There are many sequences of actions, each with their own utility

• We want to find, or search for, the best one
Defining a problem

A problem can be defined by five components

• **Initial state**
  – agent starts from initial state

• **Actions**
  – once in a state, these are the various options that an agent can choose and execute

• **Transition model**
  – describes what happens when an agent in a particular state executes a particular action

• **Goal test**
  – checks if the current state is a goal state

• **Path cost**
  – assigns a numeric cost to each path
Example: Romania

• On holiday in Romania; currently in Arad
• Flight leaves tomorrow from Bucharest
• **Formulate goal:**
  – be in Bucharest
• **Formulate problem:**
  – **states:** various cities
  – **actions:** drive between cities or choose next city
• **Find solution:**
  – sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest
Example: Romania
Why not Dijkstra’s Algorithm?

• Dijkstra’s algorithm inputs the entire graph
  – we want to search in unknown spaces
  – essentially, we combine search with exploration

• D’s algorithm takes connections as given
  – we want to search based on agent’s actions
  – the agent may not know the result of an action in a state before trying it

• D’s algorithm won’t work on infinite spaces
  – we want to search in infinite spaces
  – e.g., the logical reasoning space is infinite
Example: vacuum world

• States
  – \(2 \times 2^2 = 8\) possible states

• Initial state
  – Any of these 8 states
  – Say in 5

• Actions
  – Each state has 3 actions, Left, Right and Suck

• Transition model
  – Actions have their expected effect
  – Moving Left in the leftmost square, Right in the rightmost square, Suck in the clean square has no effect

• Goal test
  – Check whether all the squares are clean

• Path cost
  – Each step costs 1, path cost is number of steps in the path

Solution: Right, Suck
Vacuum world state space graph
Example: The 8-puzzle

- states?
- initial state?
- actions?
- goal test?
- path cost?
Example: The 8-puzzle

- **states?** locations of tiles
- **initial state?** given
- **actions?** move blank left, right, up, down
- **goal test?** goal state (given)
- **path cost?** 1 per move

[Note: optimal solution of $n$-Puzzle family is NP-complete]
8-puzzle

• 8-puzzle belongs to the family of sliding block puzzles
• This family is know to be NP-complete
  – One doesn’t expect to find methods significantly better in the worst case than the search algorithms to be described net
  – Any given goal states can be reached from exactly half of the possible initial states
• How hard is it to solve sliding block puzzles?
  – 8-puzzle have 9!/2=181,440 reachable states and can be solved easily
  – 15-puzzle (4 x 4 block) has around 1.3 million states and random problem instances can be solved optimally in a few milliseconds by the best search algorithms
  – The 24-puzzle (5 x 5 block) has around $10^{25}$ states and random instances may take several hours to solve optimally
Example: 8-Queens

- **states?**
  - any arrangement of $n \leq 8$ queens
  - *or* arrangements of $n \leq 8$ queens in leftmost $n$ columns, 1 per column, such that no queen attacks any other.

- **initial state?**
  - no queens on the board

- **actions?**
  - add queen to any empty square *or* add queen to leftmost empty square such that it is not attacked by other queens.

- **goal test?**
  - 8 queens on the board, none attacked.

- **path cost?**
  - 1 per move
8-queens

• 8-queens problem has $64.63 \ldots 57 = 1.8 \times 10^{14}$ possible sequences to investigate
• A better formulation would prohibit placing a queen in a square that is already attacked
  – States: all possible arrangements of $n$ queens ($0 \leq n \leq 8$) one per column in the leftmost $n$ columns, with no queen attacking another
  – Action: add a queen to the leftmost empty column such that it is not attacked by any other queen
  – This formulation reduces state space from $1.8 \times 10^{14}$ to 2057 and solutions are easy to find
• For 100-queen problems this reduction is from $10^{400}$ states to $10^{52}$ states (not enough to make the problem tractable)
• Later, we will study algorithms that can solve even the million-queen problem with ease
Tree search algorithms

• Basic idea:
  – Having formulate the problem we need to solve it
  – A solution is an action sequence
  – The possible action sequence starting at the initial state forms a search tree
  – Root node of this tree corresponds to the initial state
  – Tree search proceeds by
    • Exploring the state space by generating successors of already-explored states (a.k.a. expanding states)
    • Evaluating every generated state if it is a goal state
Tree search example
Tree search example
Tree search example

```
function TREE-SEARCH(problem) returns a solution, or failure
initialize the frontier using the initial state of problem
loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier
```
Repeated states

- Failure to detect repeated states can turn a linear problem into an exponential one!

- Test is often implemented as a hash table.
Solutions to Repeated States

- Graph search
  - never generate a state generated before
    - must keep track of all possible states (uses a lot of memory)
    - e.g., 8-puzzle problem, we have $9! = 362,880$ states
    - approximation for DFS/DFS: only avoid states in its (limited) memory: avoid looping paths.
    - Graph search optimal for BFS and UCS, not for DFS.
Solutions to Repeated States

function TREE-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        expand the chosen node, adding the resulting nodes to the frontier

function GRAPH-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    initialize the explored set to be empty
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        add the node to the explored set
        expand the chosen node, adding the resulting nodes to the frontier
            only if not in the frontier or explored set

Figure 3.7 An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.
Implementation: states vs. nodes

- A state is a (representation of) a physical configuration

- A node is a data structure constituting part of a search tree contains info such as: state, parent node, action, path cost $g(x)$, depth

- The **CHILD-NODE** function takes a parent node and an action and returns the resulting child node
Implementation: states vs. nodes

• A state is a (representation of) a physical configuration

• A node is a data structure constituting part of a search tree contains info such as: state, parent node, action, path cost $g(x)$, depth

```python
function CHILD-NODE(problem, parent, action) returns a node
    return a node with
    STATE = problem.RESULT(parent.STATE, action),
    PARENT = parent, ACTION = action,
    PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```
Search strategies

• A search **strategy** is defined by picking the **order of node expansion**

• Strategies are evaluated along the following dimensions:
  - **completeness**: does it always find a solution if one exists?
  - **time complexity**: number of nodes generated
  - **space complexity**: maximum number of nodes in memory
  - **optimality**: does it always find a least-cost solution?

• Time and space complexity are measured in terms of
  - **$b$**: maximum branching factor of the search tree
  - **$d$**: depth of the least-cost solution
  - **$m$**: maximum depth of any path in the state space (may be $\infty$)
Uninformed search strategies

• **Uninformed (blind):**
  – You have no clue whether one non-goal state is better than any other. Your search is blind. You don’t know if your current exploration is likely to be fruitful.

• **Various blind strategies:**
  – Breadth-first search
  – Uniform-cost search
  – Depth-first search
  – Iterative deepening search (generally preferred)
  – Bidirectional search (preferred if applicable)
Uninformed search strategies

• **Queue for Frontier:**
  – FIFO? LIFO? Priority?

• **Goal-Test:**
  – When inserted into *Frontier*? When removed?

• **Tree Search or Graph Search:**
  – Forget *Explored* nodes? Remember them?
Frontier

- **Explored nodes**
  - Black
- **Unexplored nodes**
  - Gray
- **Frontier**
  - white

**Figure 3.9** The separation property of **GRAPH-SEARCH**, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been expanded. In (c), the remaining successors of the root have been expanded in clockwise order.
Queue for Frontier

• FIFO (First In, First Out)
  – Results in Breadth-First Search

• LIFO (Last In, First Out)
  – Results in Depth-First Search

• Priority Queue sorted by path cost so far
  – Results in Uniform Cost Search

• Iterative Deepening Search uses Depth-First

• Bidirectional Search can use either Breadth-First or Uniform Cost Search
When to do Goal-Test? When generated? When popped?

• Do Goal-Test when node is popped from queue
  IF you care about finding the optimal path
  AND your search space may have both short expensive and long cheap paths to a goal.
    – Guard against a short expensive goal.
    – E.g., Uniform Cost search with variable step costs.

• Otherwise, do Goal-Test when is node inserted.
  – E.g., Breadth-first Search, Depth-first Search, or Uniform Cost search when cost is a non-decreasing function of depth only (which is equivalent to Breadth-first Search).

• REASON ABOUT your search space & problem.
  – How could I possibly find a non-optimal goal?
Breadth-first Search

function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure

    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier) /* chooses the shallowest node in frontier */
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier ← INSERT(child, frontier)

Figure 3.11 Breadth-first search on a graph.
Breadth-first search

• Expand shallowest unexpanded node
• **Frontier** (or fringe): nodes in queue to be explored
• **Frontier** is a first-in-first-out (FIFO) queue, i.e., new successors go at end of the queue.
• **Goal-Test** when inserted.

Initial state = A
Is A a goal state?

Put A at end of queue.
frontier = [A]
Breadth-first search

• Expand shallowest unexpanded node
• *Frontier* is a FIFO queue, i.e., new successors go at end

Expand A to B, C.
Is B or C a goal state?

Put B, C at end of queue. frontier = [B,C]
Breadth-first search

• Expand shallowest unexpanded node

• *Frontier* is a FIFO queue, i.e., new successors go at end

Expand B to D, E
Is D or E a goal state?

Put D, E at end of queue
frontier=[C,D,E]
Breadth-first search

• Expand shallowest unexpanded node
• *Frontier* is a FIFO queue, i.e., new successors go at end

Expand C to F, G.
Is F or G a goal state?

Put F, G at end of queue.
frontier = [D,E,F,G]
Breadth-first search

• Expand shallowest unexpanded node

• *Frontier* is a FIFO queue, i.e., new successors go at end

Expand D to no children.
Forget D.

frontier = [E,F,G]
Breadth-first search

- Expand shallowest unexpanded node
- *Frontier* is a FIFO queue, i.e., new successors go at end

Expand E to no children.
Forget B,E.

frontier = [F,G]
Example BFS
Properties of breadth-first search

- **Complete?**
  - Yes, it always reaches a goal (if $b$ is finite)

- **Time?**
  - $1 + b + b^2 + b^3 + ... + b^d = O(b^d)$
  - (this is the number of nodes we generate)

- **Space?**
  - $O(b^d)$ (keeps every node in memory, either in frontier or on a path to frontier)
  - $O(b^{d-1})$ nodes in explored set and $O(b^d)$ nodes in frontier

- **Optimal?**
  - No, for general cost functions.
  - Yes, if cost is a non-decreasing function only of depth with $f(d) \geq f(d-1)$, e.g., step-cost = constant:
    - All optimal goal nodes occur on the same level
    - Optimal goal nodes are always shallower than non-optimal goals
    - An optimal goal will be found before any non-optimal goal
Properties of breadth-first search

• Assume 1 million nodes can be generated per sec and a node requires 1000 bytes of storage

<table>
<thead>
<tr>
<th>Depth</th>
<th>Nodes</th>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>110</td>
<td>1.1 milliseconds</td>
<td>107 kilobytes</td>
</tr>
<tr>
<td>4</td>
<td>11,110</td>
<td>111 milliseconds</td>
<td>10.6 megabytes</td>
</tr>
<tr>
<td>6</td>
<td>$10^6$</td>
<td>11 seconds</td>
<td>1 gigabyte</td>
</tr>
<tr>
<td>8</td>
<td>$10^8$</td>
<td>19 minutes</td>
<td>103 gigabytes</td>
</tr>
<tr>
<td>10</td>
<td>$10^{10}$</td>
<td>31 hours</td>
<td>10 terabytes</td>
</tr>
<tr>
<td>12</td>
<td>$10^{12}$</td>
<td>129 days</td>
<td>1 petabyte</td>
</tr>
<tr>
<td>14</td>
<td>$10^{14}$</td>
<td>35 years</td>
<td>99 petabytes</td>
</tr>
<tr>
<td>16</td>
<td>$10^{16}$</td>
<td>3,500 years</td>
<td>10 exabytes</td>
</tr>
</tbody>
</table>

*Figure 3.13* Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 100,000 nodes/second; 1000 bytes/node.

• **Space** is the bigger problem (more than time)
Uniform-cost search

• Breadth-first is only optimal if path cost is a non-decreasing function of depth,
  – i.e., \( f(d) \geq f(d-1) \); e.g., constant step cost, as in the 8-puzzle.
• Can we guarantee optimality for variable positive step costs \( \geq \varepsilon \)
  – Why \( \geq \varepsilon \)?
  – Uniform step cost doesn’t care about number of steps a path has but only about total path cost. Therefore, it will get stuck in an infinite loop if there is a path with infinite sequence of zero cost actions. (e.g., NoOp action)
• **Uniform-cost Search:**
  – Expand node with smallest path cost \( g(n) \).
  – Frontier is a priority queue, i.e., new successors are merged into the queue sorted by \( g(n) \).
  – Can remove successors already on queue w/higher \( g(n) \).
  – Saves memory, costs time; another space-time trade-off.
  – Goal-Test when node is popped off queue.
Uniform-cost search

function UNIFORM-COST-SEARCH(problem) returns a solution, or failure

node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
frontier ← a priority queue ordered by PATH-COST, with node as the only element
explored ← an empty set
loop do
  if EMPTY?(frontier) then return failure
  node ← POP(frontier) /* chooses the lowest-cost node in frontier */
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  add node.STATE to explored
  for each action in problem.ACTIONS(node.STATE) do
    child ← CHILD-NODE(problem, node, action)
    if child.STATE is not in explored or frontier then
      frontier ← INSERT(child, frontier)
    else if child.STATE is in frontier with higher PATH-COST then
      replace that frontier node with child
Uniform-cost search

• **Uniform-cost Search**
  – Expand node with smallest path cost $g(n)$.

• **Proof of completeness**
  – Given that every step will cost more than 0
    – Assume a finite branching factor
    – There is a finite number of expansions required before the total path cost is equal to the path cost of the state
    – Hence, we will reach it

• **Proof of optimality given completeness**
  – Assume UCS is not optimal.
  – Then there must be an (optimal) goal state with path cost smaller than the found (suboptimal) goal state (invoking completeness).
  – However, this is impossible because UCS would have expanded that node first by definition.
  – **Contradiction.**
Uniform-cost search

- **Implementation:**
  - *Frontier* = queue ordered by path cost.
  - Equivalent to breadth-first if all step costs all equal.

- **Complete?**
  - Yes, if \( b \) is finite and step cost \( \geq \varepsilon > 0 \). (otherwise it can get stuck in infinite loops)

- **Time?**
  - # of nodes with *path cost* \( \leq \) cost of optimal solution. \( O(b^{\lceil 1+C*/\varepsilon \rceil}) \)
  - \( C^* \) is the cost of the optimal solution
  - Every action costs at least \( \varepsilon \)
  - For unit cost \( O(b^{\lceil 1+C*/\varepsilon \rceil}) \approx O(b^{d+1}) \)

- **Space?**
  - # of nodes with path cost \( \leq \) cost of optimal solution \( O(b^{\lceil 1+C*/\varepsilon \rceil}) \approx O(b^{d+1}) \)

- **Optimal?**
  - Yes, for any step cost \( \geq \varepsilon > 0 \).
Uniform-cost search

• Want to go from Sibiu to Bucharest
  • BFS does not find optimal path but UCS finds optimal path why?
The graph above shows the step-costs for different paths going from the start (S) to the goal (G).

Use uniform cost search to find the optimal path to the goal.

Exercise in class
Depth-first search

• Expand deepest unexpanded node
• **Frontier** = Last In First Out (LIFO) queue, i.e., new successors go at the front of the queue.
• **Goal-Test** when inserted.

Initial state = A
Is A a goal state?

Put A at front of queue.
frontier = [A]
Depth-first search

• Expand deepest unexpanded node
  – *Frontier* = LIFO queue, i.e., put successors at front

Expand A to B, C.
Is B or C a goal state?

Put B, C at front of queue.
frontier = [B,C]

Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes
Depth-first search

• Expand deepest unexpanded node
  – *Frontier* = LIFO queue, i.e., put successors at front

Expand B to D, E.
Is D or E a goal state?

Put D, E at front of queue.
frontier = [D,E,C]
Depth-first search

• Expand deepest unexpanded node
  – *Frontier* = LIFO queue, i.e., put successors at front

Expand D to H, I.
Is H or I a goal state?

Put H, I at front of queue.
frontier = [H,I,E,C]

Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes
Depth-first search

• Expand deepest unexpanded node
  – *Frontier* = LIFO queue, i.e., put successors at front

Expand H to no children.
Forget H.

frontier = [I,E,C]

Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes
Depth-first search

• Expand deepest unexpanded node
  – *Frontier* = LIFO queue, i.e., put successors at front

Expand I to no children.
Forget D, I.

frontier = [E,C]
Depth-first search

• Expand deepest unexpanded node
  – *Frontier* = LIFO queue, i.e., put successors at front

Expand E to J, K.
Is J or K a goal state?

Put J, K at front of queue.
frontier = [J,K,C]

Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes
Depth-first search

- Expand deepest unexpanded node
  - **Frontier** = LIFO queue, i.e., put successors at front

Expand I to no children.
Forget D, I.

frontier = [E,C]
Depth-first search

• Expand deepest unexpanded node
  – *Frontier* = LIFO queue, i.e., put successors at front

Expand K to no children.
Forget B, E, K.

frontier = [C]

Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes
Depth-first search

• Expand deepest unexpanded node
  – Frontier = LIFO queue, i.e., put successors at front

Expand C to F, G.
Is F or G a goal state?

Put F, G at front of queue.
frontier = [F,G]
Properties of depth-first search

- **Complete?**
  - Tree search
    - **NO**
  - Graph Search
    - Yes (for finite state space)
- **Time?** $O(b^m)$ for tree search, with $m =$maximum depth of state space
  - Terrible if $m$ is much larger than $d$
  - If solutions are dense, may be much faster than BFS
- **Space?** $O(bm)$, i.e., linear space (for tree search)!
  - Remember a single path from root to leaf+ expanded unexplored sibling nodes
- **Optimal?** No: It may find a non-optimal goal first
Properties of breadth-first search

<table>
<thead>
<tr>
<th>Depth</th>
<th>Nodes</th>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>110</td>
<td>1.1 milliseconds</td>
<td>107 kilobytes</td>
</tr>
<tr>
<td>4</td>
<td>11,110</td>
<td>111 milliseconds</td>
<td>10.6 megabytes</td>
</tr>
<tr>
<td>6</td>
<td>$10^6$</td>
<td>11 seconds</td>
<td>1 gigabyte</td>
</tr>
<tr>
<td>8</td>
<td>$10^8$</td>
<td>19 minutes</td>
<td>103 gigabytes</td>
</tr>
<tr>
<td>10</td>
<td>$10^{10}$</td>
<td>31 hours</td>
<td>10 terabytes</td>
</tr>
<tr>
<td>12</td>
<td>$10^{12}$</td>
<td>129 days</td>
<td>1 petabyte</td>
</tr>
<tr>
<td>14</td>
<td>$10^{14}$</td>
<td>35 years</td>
<td>99 petabytes</td>
</tr>
<tr>
<td>16</td>
<td>$10^{16}$</td>
<td>3,500 years</td>
<td>10 exabytes</td>
</tr>
</tbody>
</table>

Figure 3.13  Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 100,000 nodes/second; 1000 bytes/node.

- **DFS** would require 156KB storage instead of 10 exabytes at depth $d=16$!
Iterative deepening search

• To avoid the infinite depth problem of DFS, only search until depth L,
  – we don’t expand nodes beyond depth L.
  – This is known as **Depth-Limited Search**

• What if solution is deeper than L?
  – Increase L iteratively
  – This is known as **Iterative Deepening Search**
  – This inherits the memory advantage of Depth-first search
  – Better in terms of space complexity than Breadth-first search
Iterative deepening search \( L=0 \)
Iterative deepening search $L=1$

Limit = 1
Iterative deepening search $L=2$
Iterative Deepening Search $L=3$
Iterative deepening search

• Number of nodes generated in a breadth-first search to depth $d$ with branching factor $b$:
  - $N_{BFS} = b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$

• Number of nodes generated in an iterative deepening search to depth $d$ with branching factor $b$:
  - $N_{IDS} = d b^1 + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + b^d = O(b^d)$

• **For $b = 10$, $d = 5$,**
  - $N_{BFS} = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110$
  - $N_{IDS} = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$
Properties of iterative deepening search

• **Iterative deepening search combines the benefits of BFS and DFS**
  – Like DFS its memory requirement is modes $O(bd)$
  – Like BFS it is complete when branching factor is finite and optimal when path cost is non decreasing function of depth of the node
Properties of iterative deepening search

- **Complete?**
  - Yes

- **Time?**
  - $O(b^d)$

- **Space?**
  - $O(bd)$

- **Optimal?**
  - No, for general cost functions.
  - Yes, if cost is a non-decreasing function only of depth
Bidirectional Search

• Basic idea
  – simultaneously search forward from start S and backwards from goal G
  – stop when both “meet in the middle”
  – Motivation is that $b^{d/2} + b^{d/2}$ is much less than $b^d$
  – need to keep track of the intersection of 2 open sets of nodes
  – Search is implemented by replacing the goal test with a check to see whether frontier of the two searches intersect, if they do a solution has been found
  – The first solution found in such a way may not be optimal, even if both the searches are breadth first searches
Bi-Directional Search

Complexity: time and space complexity are: $O(b^{d/2})$
Bidirectional Search

• Reduction in time complexity makes bidirectional search attractive

• What does searching backwards from G mean
  – need a way to specify the predecessors of G
    • Let predecessor of a state x be all those states that have x as a successor
    • Specifying predecessor can be difficult if the goal is abstract
    • e.g., what is the predecessor of a goal state that “no queen attacks another queen”?  
  – which one to choose if there are multiple goal states?
## Summary of algorithms

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Breadth-First</th>
<th>Uniform-Cost</th>
<th>Depth-First</th>
<th>Depth-Limited</th>
<th>Iterative Deepening DLS</th>
<th>Bidirectional (if applicable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete?</td>
<td>Yes[a]</td>
<td>Yes[a,b]</td>
<td>No</td>
<td>No</td>
<td>Yes[a]</td>
<td>Yes[a,d]</td>
</tr>
<tr>
<td>Time</td>
<td>O(b^d)</td>
<td>O(b\lceil1+C*/\varepsilon\rceil)</td>
<td>O(b^m)</td>
<td>O(b^l)</td>
<td>O(b^d)</td>
<td>O(b^{d/2})</td>
</tr>
<tr>
<td>Space</td>
<td>O(b^d)</td>
<td>O(b\lceil1+C*/\varepsilon\rceil)</td>
<td>O(bm)</td>
<td>O(bl)</td>
<td>O(bd)</td>
<td>O(b^{d/2})</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes[c]</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes[c]</td>
<td>Yes[c,d]</td>
</tr>
</tbody>
</table>

There are a number of footnotes, caveats, and assumptions. See Fig. 3.21, p. 91.

[a] complete if b is finite
[b] complete if step costs \( \geq \varepsilon > 0 \)
[c] optimal if step costs are all identical
   (also if path cost non-decreasing function of depth only)
[d] if both directions use breadth-first search
   (also if both directions use uniform-cost search with step costs \( \geq \varepsilon > 0 \))

Note that \( d \leq \lceil 1 + C*/\varepsilon \rceil \)