**General Tree Search**

```haskell
function Tree-search(problem) returns a solution, or failure

fringe = new Set();
fringe.put(problem.initialState)

loop do
  if fringe.isEmpty() then return failure
  node ← fringe.get()
  if problem.isGoalState(node)
    then return node;
  fringe.putAll(problem.expand(node))
```

A problem is specified by:
- `initialState`
- `actions/results` ➔ `expand()`
- `isGoalState`

*Which node in the fringe does `get()` return?*
Measuring Search Performance

- **Completeness**
  - Is the algorithm guaranteed to find a solution if it exists?

- **Optimality**
  - Does the strategy find the minimum path cost solution?

- **Time Complexity**
  - How long does it take to find a solution?

- **Space Complexity**
  - How much memory is needed to perform the search?

_Our search strategy will affect all of the above!_

Breadth-First Search (BFS)

- **General tree-search where queue is first-in-first-out (FIFO)**
  - `get` operation returns oldest item on fringe
  - corresponds to a _level-order_ traversal of search tree
  - All nodes at level $d$ expanded before any at $d+1$
BFS in Vacuum World

Actions:
L – move Left
S – Suck up the dirt
R – move Right

Starting in state 1, generate search tree.

Vacuum World BFS

Are we done? Hurray!
BFS again

- What does BFS look like from the perspective of General Tree Search?

Search Performance Criteria: BFS

- **Completeness**: Guaranteed to find a solution if it exists?
  - **YES**

- **Optimality**: Does the strategy find the minimum path cost solution?
  - **YES if uniform action cost**

- **Time Complexity**: How long does it take to find a solution?

- **Space Complexity**: How much memory is needed to perform the search?
BFS Time Complexity

- Assume uniform search space
  - Each node has same number of successors
  - Branching factor, $b$
- $d$: Depth of shallowest solution
- BFS generates
  - Complete search trees at depth $\leq d$
  - Worst case: $b^{d+1}$ nodes at depth $d+1$
  - Total nodes:
    \[ b + b^2 + b^3 + \ldots + b^d + b^{d+1} = O(b^{d+1}) \]

BFS Space Complexity

- Same as time complexity: $O(b^{d+1})$
- Must store entire fringe: all nodes at deepest level
- In typical computer configurations, will run out of space before running out of time
BFS and sub-optimal solutions

Uniform-Cost Search (UCS)

- General tree-search where queue is priority-first
  - get operation returns least-cost item on fringe
  - All nodes at cost less than $c$ expanded before any at cost $c$
    - Same as BFS if all actions have same cost, different otherwise
UCS Properties

- Complete?
  - Yes

- Time and space complexity
  - Uniform cost:
    - Same as BFS: $O(b^{d+1})$
    - In general, with minimal path cost $C^*$ and minimal action cost $\epsilon$:
      - $O(b^{C^*/\epsilon + 1})$

- Optimal as long as cost monotonic along path
  - Guaranteed by each edge cost $\geq 0$

Depth-First Search (DFS)

- General tree-search where queue is last-in-first-out (LIFO, aka stack)
  - `get` operation returns newest item on fringe

- Often implemented recursively using function-call stack
- Always expand deepest node on fringe
NOTE REGARDING NEXT SLIDE

- The nodes should really be expanded on the right side of the tree, so that the most recently added node is expanded.

Vacuum World DFS

Uh oh!
Recursive implementation

- function search_recursive(state)
  - if (is_goal(state))
    - return state;
  - for (child : children(state))
    - v = search_recursive(child)
    - if (v != null)
      - return v;
  - return null;

Performance of Depth-First

- Completeness
  - Only for finite depth trees, and so in general: No

- Optimality
  - No

- Time Complexity
  - $O(b^m)$; where $m$ is the maximum depth of tree

- Space Complexity
  - $O(b^m)$
Depth-Limited Search

- Problem: unbounded trees in DFS
- Solution:
  - Predetermine a depth limit \( L \)
  - Run DFS, cut off search at depth \( L \)
- Complete iff exists solution within \( L \) steps
- Optimal?
- Time complexity: \( O(b^L) \)
- Space complexity: \( O(bL) \)

- How do we choose \( L \)?

Iterative Deepening DFS

```
function Iterative-deeping-search(problem)
returns a solution, or failure

loop for depth from 0 to infinity
  if Depth-limited-search(problem, depth) succeeds
    then return its result
  end loop
return failure
```
IDS Properties

- Complete?
- Optimal?
- Time Complexity?
- Space Complexity?

IDS Complexity

- Time is same as running DLS for depth = 1,...,d

\[ \sum_{l=1}^{d} O(b^l) = O(b) + O(b^2) + \cdots + O(b^d) = O(b^d) \]

- Space is same as DLS for depth d: $O(bd)$
## Analysis Summary

<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</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Time complexity</td>
<td>$O(b^{d+1})$</td>
<td>$O(b^{C^{e+1}})$</td>
<td>$O(b^m)$</td>
<td>$O(b^L)$</td>
<td>$O(b^d)$</td>
</tr>
<tr>
<td>Space complexity</td>
<td>$O(b^{d+1})$</td>
<td>$O(b^{C^{e+1}})$</td>
<td>$O(bm)$</td>
<td>$O(bL)$</td>
<td>$O(bd)$</td>
</tr>
</tbody>
</table>

Exponential in depth!  \hspace{1cm} Linear in depth!

## Your turn!

- **Show that:**
  \[ b + b^2 + b^3 + \ldots + b^n = O(b^n) \]
  - (by finding a polynomial of order $b^n$ with constant coefficients that is greater than the sum)

- **7 Queens:**
  - **Goal:** Place 7 queens on 7x7 chess board so that no two attack each other
  - **Formulate the problem carefully**
    - What state space?
    - What actions do you consider at each step?
  - **Which search strategy to use?**
    - Find a solution using your strategy.
The importance of problem formulation

- Problem formulation has huge impact on complexity
  - State space
  - Actions

- Consider 7-queens problem:
  - Naïve state space: branching factor of roughly ~49
  - One queen per column: branching factor of 7.
Path Planning Example

- Consider an agent trying to find the best route from one place to another.
  - Actions = \{N, S, E, W\}
  - DFS is out. (Why?)

- The shortest path is 50 moves.
  - Complexity of BFS/IDS?

- How many distinct states are there?
  - \(25^2 << 4^{50}\) (by a factor of \(10^{27}\))
  - What are we doing wrong?
    - Repeated states!

Avoiding repeated states

- Idea: don’t re-expand nodes that we’ve already expanded.
  - Closed list: set of all states previously visited
  - Memory usage?
General Graph Search

function Tree-search(problem)
  returns a solution, or failure

  closed = new Set();
closed.add(problem.initialState);

  fringe = new Set();
fringe.put(problem.initialState)

  loop do
    if fringe.isEmpty() then return failure
    node ← fringe.get()
    if problem.isGoalState(node)
      then return node;

    for each child in problem.expand(node)
      if !set.contains(child)
        closed.add(child)
        fringe.put(child)

General Graph Search: Analysis

- Time complexity?
  - $O(|V| + |E|)$

- Memory complexity?
  - $O(|V|)$

- Optimality?
  - BFS/IDS?
  - DFS?

- What about DFS's infinite loop problem?
  - Fixed for finite worlds!

- Can we guarantee optimality, regardless of the policy?
  - For finite worlds, if we replace old bad paths with new good paths rather than discarding the new good paths.
  - Some bookkeeping...
Graph Search: Summary

- Graph Search
  - Maintains list of states already visited
  - Terminates searches that revisit the same states.

- When do we want to use graph search?
  - When repeated states are likely
  - We can afford the memory

Next Time: Informed Search

- Employ additional information about node states in deciding which to expand

- Questions:
  - What criteria?
  - How to exploit?
  - Properties?