Artificial Intelligence
Uninformed Search (Ch. 3.4)
(and a little more formalization)

Questions?

- Bread-first, depth-first, uniform cost search
- Generation and expansion
- Goal tests
- Queueing function
- Complexity, completeness, and optimality
- Heuristic functions (for informed search)
- Admissibility

Formalizing Search: Review

- A state space is a graph \((V, E)\):
  - \(V\) is a set of nodes (states)
  - \(E\) is a set of arcs (agent operations/actions)
- State space contains all possible states

Formalizing Search: III

- Solution: a sequence of operators...
  - Giving a path
  - Through state space
  - From a start node to a goal node
- Solution cost: sum of arc costs on solution path
  - If all arcs have the same cost, then the solution cost = the length of the solution

Formalizing Search: IV

- State-space search: searching through a state space for a solution
- By making explicit a sufficient portion of an implicit state-space graph to find a goal node
  - Initially \(V = \{S\}\), where \(S\) is the start node
  - When \(S\) is expanded, its successors are generated; those nodes are added to \(V\) and the arcs are added to \(E\)
  - This process continues until a goal node is found
- It isn't usually practical to represent entire space

Homework 1

- Blackboard is open! Check access before tomorrow
- See corrections in Piazza:
  - Point values in III.2 should be 3, 6, and 9
  - Your PDF file should contain parts I, II, and IV
  - Example return in III.1.(b) should be in brackets:
    - \(\text{lottery()} = [75, 235, 7, 100]\)
- Common Mistakes:
  - Don't print additional information
  - Functions should return or print, not both
  - No extra arguments or return values
  - Return or output things in the order and format specified

Questions?

- Bread-first, depth-first, uniform cost search
- Generation and expansion
- Goal tests
- Queueing function
- Complexity, completeness, and optimality
- Heuristic functions (for informed search)
- Admissibility

Formalizing Search: Review

- A state space is a graph \((V, E)\):
  - \(V\) is a set of nodes (states)
  - \(E\) is a set of arcs (agent operations/actions)
- State space contains all possible states

Formalizing Search: III

- Solution: a sequence of operators...
  - Giving a path
  - Through state space
  - From a start node to a goal node
- Solution cost: sum of arc costs on solution path
  - If all arcs have the same cost, then the solution cost = the length of the solution

Formalizing Search: IV

- State-space search: searching through a state space for a solution
- By making explicit a sufficient portion of an implicit state-space graph to find a goal node
  - Initially \(V = \{S\}\), where \(S\) is the start node
  - When \(S\) is expanded, its successors are generated; those nodes are added to \(V\) and the arcs are added to \(E\)
  - This process continues until a goal node is found
- It isn't usually practical to represent entire space
Formalizing Search: V

- Each node implicitly or explicitly represents a **partial solution path** from start node to itself
  - (And a cost!)
  - In general, from a node there are many possible paths (and therefore solutions) that have this partial path as a prefix

State-Space Search Algorithm

```plaintext
function general-search (problem, QUEUEING-FUNCTION)
;; problem describes start state, operators, goal test, ;; and operator costs
;; queueing-function is a comparator function that
;; returns either a goal node or failure

nodes = MAKE-QUEUE(MAKE-NODE(problem.INITIAL-STATE))
loop
  if EMPTY(nodes) then return "failure"
  node = REMOVE-FRONT(nodes)
  if problem.GOAL-TEST(node.STATE) succeeds
    then return node
  nodes = QUEUEING-FUNCTION(nodes, EXPAND(node, problem.OPERATORS))
end
;; Note: The goal test is NOT done when nodes are generated
;; Note: This algorithm does not detect loops
```

Generation vs. Expansion

- **Selecting** a state means making that node current
- **Expanding** the current state means applying every legal action to the current state
- Which **generates** a new set of nodes

Key Procedures

- **EXPAND**
  - Generate all successor nodes of a given node
  - "What nodes can I reach from here (by taking what actions)?"
- **GOAL-TEST**
  - Test if state satisfies goal conditions
- **QUEUEING-FUNCTION**
  - Maintain a **ranked** list of nodes that are expansion candidates
  - "What should I explore next?"

Some Issues

- Return a path or a node depending on problem
  - In 8-queens return a **node**
  - 8-puzzle return a **path**
  - What about Sheep & Wolves?
- **Changing definition of Queueing-Function ➔ different search strategies**
  - How do you choose what to expand next?*

Review: Characteristics

- **Completeness**: Is the algorithm guaranteed to find a solution (if one exists)?
- **Optimality**: Does it find the optimal solution?
  - (The solution with the lowest path cost of all possible solutions)
- **Time complexity**: How long does it take to find a solution? (# of nodes expanded/visited)
- **Space complexity**: How much memory is needed to perform the search? (max # of nodes in list)

* All of search is answering this question!
Uninformed vs. Informed Search

- Uninformed (aka “blind”) search
  - Use no information about the “direction” of the goal node(s)
  - No way to know if we’re “doing well so far”
  - Breadth-first, depth-first, depth-limited, uniform-cost, depth-first iterative deepening, bidirectional
- Informed (aka “heuristic”) search (next class)
  - Use domain information to (try to) (usually) head in the general direction of the goal node(s)
  - Hill climbing, best-first, greedy search, beam search, A, A*

Why Apply Goal Test Late?

- Why does it matter when the goal test is applied (expansion time vs. generation time)?
- Optimality and complexity of the algorithms are strongly affected!

Breadth-First

- Enqueue nodes in FIFO (first-in, first-out) order
- Characteristics:
  - Complete (meaning?)
  - Optimal (i.e., admissible) if all operators have the same cost
  - Otherwise, not optimal but finds solution with shortest path length
  - Exponential time and space complexity, $O(b^d)$, where:
    - $d$ is the depth of the solution
    - $b$ is the branching factor (number of children) at each node
- Takes a long time to find long-path solutions

BFS

- Breadth-first search
- Starts at the root node and explores all nodes at the current depth before moving on to the next depth level.
- Expands nodes in a breadth-first order, i.e., it explores all the nodes at the current depth before moving on to the next depth.
- It explores all the nodes at a given depth before moving on to the next depth.

BFS

- Breadth-first search
- Starts at the root node and explores all nodes at the current depth before moving on to the next depth.
- It explores all the nodes at a given depth before moving on to the next depth.
- This is done level by level, such that all nodes at a particular level are visited before any nodes at the next level.

**Breadth-First: Analysis**

- Takes a **long time to find long-path solutions**
  - Must look at all shorter length possibilities first
  - A complete search tree of depth \( d \) where each non-leaf node has \( b \) children:
    \[
    1 + b + b^2 + \ldots + b^d = \frac{(b^{d+1} - 1)}{(b-1)} \text{ nodes}
    \]
  - What if we expand nodes when they are selected?
  - **Checks a lot of short-path solutions quickly**

**Depth-First (DFS)**

- Enqueue nodes in \textit{LIFO} (last-in, first-out) order
  - That is, nodes used as a stack data structure to order nodes

- Characteristics:
  - ** Might not terminate** without a “depth bound”
    - I.e., cutting off search below a fixed depth \( D \) ("depth-limited search")
  - **Not complete**
    - With or without cycle detection, and with or without a cutoff depth
  - **Exponential time**, \( O(b^d) \), but only **linear space**, \( O(bd) \)

**Breadth-First: O(Example)**

\[
1 + b + b^2 + \ldots + b^d = \frac{(b^{d+1} - 1)}{(b-1)} \text{ nodes}
\]

- Tree where: \( d=12 \)
- Every node at depths 0, ..., 11 has 10 children (\( b=10 \))
- Every node at depth 12 has 0 children
- \( 1 + 10 + 100 + 1000 + \ldots + 10^{12} = (10^{13} - 1)/9 = \mathcal{O}(10^{12}) \) nodes in the complete search tree
- If BFS expands 1000 nodes/sec and each node uses 100 bytes of storage
  - Will take 35 years to run in the worst case
  - Will use 111 terabytes of memory
Depth-First (DFS): Analysis

• DFS:
  • Can find long solutions quickly if lucky
  • And short solutions slowly if unlucky

• When search hits a dead end
  • Can only back up one level at a time*
  • Even if the "problem" occurs because of a bad operator choice near the top of the tree
  • Hence, only does "chronological backtracking"

* Why?
Uniform-Cost (UCS)

- Enqueue nodes by path cost:
  - Let \( g(n) \) = cost of path from start node to current node \( n \)
  - Sort nodes by increasing value of \( g \)
  - Identical to breadth-first search if all operators have equal cost
- "Dijkstra's Algorithm" in algorithms literature
- "Branch and Bound Algorithm" in operations research literature
- Complete (*)
- Optimal/Admissible (*)
  - Admissibility depends on the goal test being applied when a node is removed from the nodes list, not when its parent node is expanded and the node is first generated
- Exponential time and space complexity, \( O(b^d) \)

UCS Implementation

- For each frontier node, save the total cost of the path from the initial state to that node
- Expand the frontier node with the lowest path cost
- Equivalent to breadth-first if step costs all equal
- Equivalent to Dijkstra's algorithm in general

Uniform-cost search example

Example: Path Costs

Depth-First Iterative Deepening (DFID)

1. DFS to depth 0 (i.e., treat start node as having no successors)
2. If no solution, do DFS to depth 1
   - Complete
   - Optimal/Admissible if all operators have the same cost
     - Otherwise, not optimal, but guarantees finding solution of shortest length
   - Time complexity is a little worse than BFS or DFS
   - Nodes near the top of the tree are generated multiple times
     - Because most nodes are near the bottom of a tree, worst case time complexity is still exponential, \( O(b^d) \)
Iterative deepening search (c=1)

Nodes visited: 3

Iterative deepening search (c=2)

Nodes visited: 3+4 = 7

Iterative deepening search (c=3)

Nodes visited: 3+4+8 = 15

The point: because cost is exponential, you're not really redoing that much work!

Depth-First Iterative Deepening

- If branching factor is $b$ and solution is at depth $d$, then nodes at depth $d$ are generated once; nodes at depth $d-1$ are generated twice, etc.
- Hence $b^d + 2b^{d-1} + ... + db / (1 - 1/b) = O(b^d)$.
- If $b=4$, then worst case is $1.78 * 4^d$, i.e., 78% more nodes searched than exist at depth $d$ (in the worst case).
- **Linear space complexity**, $O(bd)$, like DFS
- Has advantage of both BFS (completeness) and DFS (limited space, finds longer paths more quickly)
- Generally preferred for **large state spaces where solution depth is unknown**

Example for Illustrating Search Strategies
**Depth-First Search**

**Expanded node** | **Nodes list**
--- | ---
S₀ | { S₀ }  
A³ | { G₁₈ B₁₈ C₈ }  
D₈ | { E₁₀ G₁₈ B₁₈ C₈ }  
E₁₀ | { G₁₈ B₁₈ C₈ }  
G₁₈ | { B₁₈ C₈ }  

Solution path found is S A G, cost 18  
Number of nodes expanded (including goal node) = 5

**Breadth-First Search**

**Expanded node** | **Nodes list**
--- | ---
S₀ | { S₀ }  
A³ | { A³ B³ C³ }  
B¹ | { C³ D³ G¹₈ G¹₃ }  
C³ | { D³ E³ G¹₈ G¹₃ }  
D₈ | { E¹₈ G¹₈ G¹₃ }  
E¹₈ | { G¹₈ G¹₃ }  
G¹₈ | { G¹₃ G¹₁₈ }  

Solution path found is S A G, cost 18  
Number of nodes expanded (including goal node) = 7

**Uniform-Cost Search**

**Expanded node** | **Nodes list**
--- | ---
S₀ | { S₀ }  
B¹ | { A³ B³ C³ }  
A³ | { D³ C³ E¹₈ G¹₈ G¹₃ }  
D₈ | { C³ E¹₈ G¹₈ G¹₃ }  
E¹₈ | { G¹₈ G¹₈ G¹₃ }  
G¹₈ | { G¹₈ G¹₃ }  

Solution path found is S C G, cost 13  
Number of nodes expanded (including goal node) = 7

**How they Perform**

- **Depth-First Search**
  - Expanded nodes: S A D E G  
  - Solution found: S A G (cost 18)

- **Breadth-First Search**
  - Expanded nodes: S A B C D E G  
  - Solution found: S A G (cost 18)

- **Uniform-Cost Search**
  - Expanded nodes: S A D B C E G  
  - Solution found: S C G (cost 13)

  This is the only uninformed search that worries about costs.

- **Iterative-Deepening Search**
  - Expanded nodes: S, S A, S A B, S A B C, S A B C D E G

**Comparing Search Strategies**

<table>
<thead>
<tr>
<th>Search Strategy</th>
<th>Complete</th>
<th>Optimal</th>
<th>Time Complexity</th>
<th>Space Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadth-first search</td>
<td>yes</td>
<td>yes</td>
<td>O(bᵈ)</td>
<td>O(b²)</td>
</tr>
<tr>
<td>Depth-first search</td>
<td>no</td>
<td>no</td>
<td>O(bᵐ)</td>
<td>O(bm)</td>
</tr>
<tr>
<td>Depth limited search</td>
<td>if f₁ &gt; d</td>
<td>no</td>
<td>O(bᵈ)</td>
<td>O(b)</td>
</tr>
<tr>
<td>Depth-first iterative deepening search</td>
<td>yes</td>
<td>yes</td>
<td>O(bᵈ)</td>
<td>O(bd)</td>
</tr>
<tr>
<td>Bi-directional search</td>
<td>yes</td>
<td>yes</td>
<td>O(bᵈ)</td>
<td>O(b²)</td>
</tr>
</tbody>
</table>

b is branching factor, d is depth of the shallowest solution, m is the maximum depth of the search tree, l is the depth limit

**Avoiding Repeated States**

- **Ways to reduce size of state space (with increasing computational costs)**
  - In increasing order of effectiveness:
    1. Do not return to the state you just came from.
    2. Do not create paths with cycles in them.
    3. Do not generate any state that was ever created before.

- Effect depends on frequency of loops in state space.
  - Worst case, storing as many nodes as exhaustive search!
A State Space that Generates an Exponentially Growing Search Space

Bi-directional Search
- Alternate searching from
  - start state $\rightarrow$ goal
  - goal state $\rightarrow$ start
- Stop when the frontiers intersect.
- Works well only when there are unique start and goal states
- Requires ability to generate "predecessor" states.
- Can (sometimes) find a solution fast

For next time: What’s a real world problem where you can’t generate predecessors?

Holy Grail Search
- Why not go straight to the solution, without any wasted detours off to the side?
- If we knew where the solution was we wouldn’t be searching!
  If only we knew where we were headed...

8-Puzzle Revisited
- What’s a good algorithm?
  - Depth-first search?
  - Breadth-first search?
  - Uniform-cost?
  - Iterative deepening?
Sudoku, Naïvely

- **State space:** 4x4 matrix, divided into four 2x2 matrices:
  A, B, C, D, cells containing values [1-4]

- **Operators:**
  - Put a 2 in square <x,y>
  - Preconditions:
    - <x,y> is empty
    - <x, (y±1)> ≠ 2; <x, (y±2)> ≠ 2; 1
    - <(x±1), y> ≠ 2; <(x±3), y> ≠ 2
    - if <x,y> in A, then 3 ∉ A; ...

- **Goal:** all blocks are filled

---

Sudoku, Naïvely

- **State space:** 4x4 matrix, divided into four 2x2 matrices:
  A, B, C, D, cells containing values [1-4]

- **Operators:**
  - Put a 2 in square <x,y>
  - Preconditions:
    - <x,y> is empty
    - <x, (y±1)> ≠ 2; <x, (y±2)> ≠ 2; 1
    - <(x±1), y> ≠ 2; <(x±3), y> ≠ 2
    - if <x,y> in A, then 3 ∉ A; ...

- **Goal:** all blocks are filled

---

Sudoku, Naïvely

- **State space:** 4x4 matrix, divided into four 2x2 matrices:
  A, B, C, D, cells containing values [1-4]

- **Operators:**
  - Put a 2 in square <x,y>
  - Preconditions:
    - <x,y> is empty
    - <x, (y±1)> ≠ 2; <x, (y±2)> ≠ 2; 1
    - <(x±1), y> ≠ 2; <(x±3), y> ≠ 2
    - if <x,y> in A, then 3 ∉ A; ...

- **Goal:** all blocks are filled

---

Sudoku, Naïvely

- **State space:** 4x4 matrix, divided into four 2x2 matrices:
  A, B, C, D, cells containing values [1-4]

- **Operators:**
  - Put a 2 in square <x,y>
  - Preconditions:
    - <x,y> is empty
    - <x, (y±1)> ≠ 2; <x, (y±2)> ≠ 2; 1
    - <(x±1), y> ≠ 2; <(x±3), y> ≠ 2
    - if <x,y> in A, then 3 ∉ A; ...

- **Goal:** all blocks are filled
Sudoku, Naïvely

- **State space**: 4x4 matrix, divided into four 2x2 matrices: A, B, C, D, cells containing values [1-4]

- **Operators**:
  - Put a 2 in square <x,y>
  - Preconditions:
    - <x,y> is empty
    - <x, (y±1)> ≠ 2; <x, (y±2)> ≠ 2; ... 
    - <(x±1), y> ≠ 2; ... 
    - \( X \) if <x,y> in A, then 3 ∈ A

- **How many operators is that?**

- **Goal**: all blocks are filled

---

“Satisficing”

- Wikipedia: “Satisficing is … searching until an acceptability threshold is met”

- **Contrast with optimality**
  - Satisfiable problems do not get more benefit from finding an optimal solution

- Ex: You have an A in the class. Studying for four hours will get you a 95 on the final. Studying for four more (eight hours) will get you a 99 on the final. What to do?

- A combination of satisfy and suffice

- Introduced by Herbert A. Simon in 1956