Today’s Class

- Uninformed search
  - What does that mean?

- Specific algorithms
  - Breadth-first search
  - Depth-first search
  - Uniform cost search
  - Depth-first iterative deepening

- Example search problems revisited

"This is the essence of search—following up one option now and putting the others aside for later, in case the first choice does not lead to a solution."

- R&N pg. 75

Key Procedures to Define

- EXPAND
  - Generate all successor nodes of a given node

- GOAL-TEST
  - Test if state satisfies all goal conditions

- QUEUEING-FUNCTION
  - Used to maintain a ranked list of nodes that are candidates for expansion

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?

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

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 ranks two states
    ;; 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
```

Bookkeeping

- Piazza
  - Thank you all for using Piazza!
  - Reminder:
    - [posts] on Piazza must follow the academic integrity guidelines
    - So post about clarifications, resources, or debugging help, but not (for example) hints about specific answers, code examples

- HW 1
- Guest lecturer next Tuesday
### 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

### Pre-Reading Quiz

- How does **breadth-first search** instantiate the EXPAND, GOAL-TEST, and QUEUING-FUNCTION components of state space search?
  - EXPAND: always expand shallowest unexpanded node
  - GOAL-TEST: test a node when it is expanded
  - QUEUING-FUNCTION: FIFO
  - What does breadth-first search remind you of? (A simple abstract data type)

- How does **uniform-cost search** instantiate these search components?
  - EXPAND: always expand lowest path cost unexpanded node
  - GOAL-TEST: test a node when it is selected for expansion
  - QUEUING-FUNCTION: priority queue
  - First generated node may not be on optimal path

- How does **depth-first search** instantiate these search components?
  - EXPAND: always expand deepest unexpanded node
  - GOAL-TEST: test a node when it is expanded
  - QUEUING-FUNCTION: LIFO

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

• A heuristic function is **admissible** if it never overestimates the cost of reaching the goal.
• The estimated cost it estimates is not higher than the lowest possible cost from the current point in the path.

Uninformed vs. Informed Search

• Uninformed search strategies
  - Use no information about the “direction” of the goal node(s).
  - Also known as “blind search”.
  - Methods: Breadth-first, depth-first, depth-limited, uniform-cost, depth-first iterative deepening, bidirectional.

• Informed search strategies (**next class…**)
  - Use information about the domain to (try to) (usually) head in the general direction of the goal node(s).
  - Also known as “heuristic search”.

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

- $A$...
- $B$...
- $C$...
- $D$...
- $E$...
- $F$...
- $G$...
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?

Depth-First (DFS)

- Enqueue nodes on nodes in **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, $O(bd)$
  - Can find long solutions quickly if lucky
  - And short solutions slowly if unlucky

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
  - \[ 1 + 10 + 100 + 1000 + \ldots + 10^{12} = \frac{(10^{13} - 1)}{9} = O(10^{12}) \text{ nodes} \]
- 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 \( g \)
  - Equivalent 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

Depth-First Iterative Deepening (DFID)

1. DFS to depth 0 (i.e., treat start node as having no successors)
2. If no solution found, 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 because nodes near the top of the search tree are generated multiple times
   - Because most nodes are near the bottom of a tree, worst case time complexity is still exponential, \( O(b^d) \)

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 \( b^d + 2b^{d-1} + \ldots + db <= b^d \cdot (1 - 1/b)^2 = O(b^d) \).
  - If \( b=4 \), then worst case is \( 1.78 \cdot 4^d \), 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
Iterative deepening search (c=1)

Depth-First Search

Expanded node | Nodes list
--- | ---
S* | \{ S* \}
A* | \{ A* B* C* \}
D* | \{ D* E* G* B* C* \}
E** | \{ E** G** B* C* \}
G** | \{ G** B* C* \}

Solution path found is S A G, cost 18
Number of nodes expanded (including goal node) = 10

Breadth-First Search

Expanded node | Nodes list
--- | ---
S* | \{ S* \}
A* | \{ A* B* C* \}
B* | \{ C* D* E* G* G* \}
C* | \{ D* E* G* G* G* \}
E** | \{ E** G** G* G* \}
G** | \{ G** G* G* \}

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 }

{ B¹ A² C³ }

{ A³ C⁶ G¹² }  

{ D⁶ C¹⁰ G¹⁰ G¹⁶ }  

{ C⁷ E⁹ G¹⁸ G²¹ }  

{ E¹⁰ G¹⁸ G²¹ }  

{ G¹³ G¹³ G²¹ }  

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:
  - Nodes expanded: S S A B C S A D E G

Bi-directional Search

• Alternate searching from
  - start state → goal
  - goal state → 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?

Comparing Search Strategies

<table>
<thead>
<tr>
<th>Search</th>
<th>Breadth-First</th>
<th>Uniform-Cost</th>
<th>Depth-First</th>
<th>Depth-Limited</th>
<th>Iterative-Deepening</th>
<th>Bi-directional</th>
<th>Applicable</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>Space</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>Optimal</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>Complete</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

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.
A State Space that Generates an Exponentially Growing Search Space

Holy Grail Search

Why not go straight to the solution, without any wasted detours off to the side?

<foreshadowing> If only we knew where we were headed... </foreshadowing>

8-Puzzle Revisited

“Satisficing”

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

• Contrast with optimality
  * Satisficiable problems do not get more benefits from finding an optimal solution
  * A combination of satisfy and suffice
  * Introduced by Herbert A. Simon in 1956