What is Search?

- Search is a basic problem-solving method
- We start in an initial state
- We examine states that are (usually) connected by a sequence of actions to the initial state
- Note: Search is (usually) a thought experiment (separate topic: Real Time Search)

- We aim to find a solution, which is a sequence of actions that brings us from the initial state to the goal state, minimizing cost
Search vs. Web Search

- When we issue a search query using Google, does Google really go poking around the web for us?
  - Not in real time!
  - Google spiders the web continually, caches results
  - Uses page rank algorithm to find the most “popular” web pages that are consistent with your query

Overview

- Problem Formulation

- Uninformed Search – constant cost
  - DFS, BFS, IDDFS, etc.

- Non-constant cost
Problem Formulation

- Components of a search problem
  - State space & initial state
  - Actions
  - Goal Test
  - Edge costs (constant or varying per edge?)

- Optimal solution = lowest path cost to goal

Example: Path Planning

Find shortest route from one city to another using highways.
Example Search Problems

- Drug design
- Logistics
  - Route planning
  - Tour Planning
- Assembly sequencing
- Internet routing
- Robot motion/path planning

Path Planning

What is the state space?
Formulation #1

Cost of one horizontal/vertical step = 1
Cost of one diagonal step = sqrt(2)

Optimal Solution

This path is the shortest in the discretized state space, but not in the original continuous space
Formulation #2

Cost of one step: length of segment

Visibility graph

Formulation #2

Cost of one step: length of segment
Solution Path

The shortest path in this state space is also the shortest in the original continuous space

Basic Search Concepts

- **Search tree**: Internal representation of our progress
- **Nodes**: Places in search tree
  (states exist in the problem space)
- **Actions**: Connect states to next states (nodes to nodes)
- **Expansion**: Generation of next states (nodes)
- **Arc cost**: Cost of moving from one state to another
- **Frontier**: Set of nodes visited, but not expanded
- **Branching factor**: Max no. of successors = b
- **Goal depth**: Depth of shallowest goal = d
  (root is depth 0, possibility of multiple goal states!)
Example: 8-Puzzle

<table>
<thead>
<tr>
<th>8</th>
<th>2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>

Initial state

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Goal state

State: Arrangement of 8 numbered tiles & empty tile on a 3x3 board

http://mypuzzle.org/sliding

15-Puzzle

- Introduced (?) in 1878 by Sam Loyd, who dubbed himself “America’s greatest puzzle-expert”
15-Puzzle

- Sam Loyd offered $1,000 of his own money to the first person who would solve the following problem:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

How big is the state space of the \((n^2-1)\)-puzzle?

- 8-puzzle \(\rightarrow 9! = 362,880\) states
- 15-puzzle \(\rightarrow 16! \approx 2.09 \times 10^{13}\) states
- 24-puzzle \(\rightarrow 25! \approx 10^{25}\) states

- But only half of these states are reachable from any given state (but you may not know that in advance)
• No one ever won the prize !!

Searching the State Space

• Often infeasible (or too expensive) to build complete representation of the state graph
8-, 15-, 24-Puzzles

- **8-puzzle** → 362,880 states
- **15-puzzle** → 2.09 x 10^{13} states
- **24-puzzle** → 10^{25} states

100 millions states/sec

- Constructing the full state graph is intractable for many interesting problems
- n-puzzle: (n+1)! states

**Intractability**

Tractability of search hinges on the ability to explore only a tiny portion of the state graph!
Searching the State Space

Search tree

Searching the State Space

Search tree
Searching the State Space

Search tree
Searching the State Space

Search tree

Searching the State Space

Search tree
If states are allowed to be revisited, the search tree may be infinite even when the state space is finite.

Data Structure of a Node

Depth of a node N = length of path from root to N (depth of the root = 0)
Node expansion

- The expansion of a node $N$ of the search tree consists of:
  - Evaluating the successor function on $\text{STATE}(N)$
  - Generating a child of $N$ for each state returned by the function

- node generation $\neq$ node expansion

Frontier of Search Tree

- The frontier is the set of all search nodes that haven’t been expanded yet
Search Strategy

- The frontier is the set of all search nodes that haven’t been expanded yet
- The fringe is implemented as a priority queue \text{FRONTIER}
  - \text{INSERT}(\text{node},\text{FRONTIER})
  - \text{REMOVE}(\text{FRONTIER})
- The ordering of the nodes in \text{FRONTIER} defines the search strategy
Generic Tree Search

**TREE-SEARCH**(initial-state)

1. If GOAL?(initial-state) then return initial-state
2. INSERT(initial-node, FRONTIER)
3. Repeat:
   4. If empty(FRONTIER) then return failure
   5. N ← REMOVE(FRONTIER)
   6. s ← STATE(N)  
      
      **Expansion of N**
   7. For every state s' in SUCCESSORS(s)
   8. Create a new node N' as a child of N
   9. If GOAL?(s') then return path or goal state
10. INSERT(N', FRONTIER)

Search Strategy

- **The frontier** is the set of all search nodes that haven’t been expanded yet
- The FRONTIER is implemented as a priority queue FRONTIER
  - INSERT(node,FRONTIER)
  - REMOVE(FRONTIER)
- The ordering of the nodes in FRONTIER defines the search strategy
**Solution to the Search Problem**

- A **solution** is a path connecting the initial node to a goal node (any one)
- The **cost** of a path is the sum of the arc costs along this path
- An **optimal** solution is a solution path of minimum cost
- There might be no solution!

---

**Algorithm Performance Measures**

- **Completeness:**
  - Does it find a solution when one exists?

- **Optimality:**
  - Does it return a min cost path whenever solution exists?

- **Complexity (space or time):**
  - Resources required by the algorithm
Breadth-First Search

- FRONTIER is a FIFO Queue

FRONTIER = (1)

FRONTIER = (2, 3)
Breadth-First Search

- FRONTIER is a FIFO Queue

FRONTIER = (3, 4, 5)

FRINGE = (4, 5, 6, 7)
BFS Properties

- Completeness:
- Optimality:
- Time complexity:
- Space complexity:

How bad is exponential in d?

<table>
<thead>
<tr>
<th>d</th>
<th># Nodes</th>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>111</td>
<td>.01 msec</td>
<td>11 Kbytes</td>
</tr>
<tr>
<td>4</td>
<td>11,111</td>
<td>1 msec</td>
<td>1 Mbyte</td>
</tr>
<tr>
<td>6</td>
<td>~10^6</td>
<td>1 sec</td>
<td>100 Mb</td>
</tr>
<tr>
<td>8</td>
<td>~10^8</td>
<td>100 sec</td>
<td>10 Gbytes</td>
</tr>
<tr>
<td>10</td>
<td>~10^10</td>
<td>2.8 hours</td>
<td>1 Tbyte</td>
</tr>
<tr>
<td>12</td>
<td>~10^12</td>
<td>11.6 days</td>
<td>100 Tbytes</td>
</tr>
<tr>
<td>14</td>
<td>~10^14</td>
<td>3.2 years</td>
<td>10,000 Tbytes</td>
</tr>
</tbody>
</table>

Assumptions: b = 10; 1,000,000 nodes/sec; 100 bytes/node
Bi-directional Search

Issues with Bi-directional Search

- Uniqueness of goal
  - Suppose goal is parking your car
  - Huge no. of possible goal states
    (configurations of other vehicles)

- Invertability of actions
Depth-First Search

- FRONTIER is a LIFO Queue

### First Iteration

- FRONTIER = (1)

### Second Iteration

- FRONTIER = (2, 3)
Depth-First Search

• FRONTIER is a LIFO Queue

1

2

FRONTIER = (4, 5, 3)

3

4

5
Depth-First Search

- FRONTIER is a LIFO Queue
Depth-First Search

- FRONTIER is a LIFO Queue
Depth-First Search

- FRONTIER is a LIFO Queue

Depth-First Search

- FRONTIER is a LIFO Queue
Depth-First Search

• FRONTIER is a LIFO Queue

DFS Properties

• Completeness:
• Optimality:
• Time complexity:
• Space complexity:
Iterative Deepening

- **Want:**
  - DFS memory requirements
  - BFS optimality, completeness
- **Idea:**
  - Do a depth-limited DFS for depth $m$
  - Iterate over $m$
Iterative Deepening

Iterative Deepening
IDDFS Properties

- Completeness:
- Optimality:
- Time complexity:
- Space complexity:

Non-constant Costs

- Arrows between states can have variable costs
- The cost of the path to each node $N$ is $g(N) = \Sigma$ costs of arcs
- Breadth-first is no longer optimal
Uniform-Cost Search

- Expand node in FRONTIER with the cheapest path, i.e., frontier is a priority queue prioritized on $g(N)$

Suboptimal path!

Search Algorithm #2

**TREE-SEARCH2(initial-state)**

1. If GOAL?(initial-state) then return initial-state
2. INSERT(initial-node,FRONTIER)
3. Repeat:
   - If empty(FRONTIER) then return failure
   - $N \leftarrow$ REMOVE(FRONTIER)
   - $s \leftarrow$ STATE($N$)
   - If GOAL?($s$) then return path or goal state
   - For every state $s'$ in SUCCESSORS($s$)
     - Create a new node $N'$ as a child of $N$
   - INSERT($N'$,FRONTIER)

The goal test is applied to a node when this node is expanded, not when it is generated.
Avoiding Revisited States

• Requires comparing state descriptions
• Breadth-first search:
  – Store all states associated with generated nodes in VISITED
  – If the state of a new node is in VISITED, then discard the node

Implemented as hash-table or as explicit data structure with flags
Explicit Data Structures

- Robot navigation
- VISITED: array initialized to 0, matching grid
- When grid position \((x,y)\) is visited, mark corresponding position in VISITED as 1
- **Size of the entire state space!**

Hash Tables

VISITED: Hash table of size \(N\)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>●</th>
<th>●</th>
<th>●</th>
<th>N-1</th>
</tr>
</thead>
</table>

```
8 2
3 4 7
5 1 6
```

```
1 2 3
4 5
7 8 6
```

Hash function: map from \(S\) to \(\{0,\ldots,N-1\}\)
Avoiding Revisited States

- Depth-first search:
  - Solution 1:
    - Store all states in current path in VISITED
    - If the state of a new node is in VISITED, then discard the node
  - ??

Avoiding Revisited States in Uniform-Cost Search

- For any state S, when the first node N such that STATE(N) = S is expanded, the path to N is the best path from the initial state to S

- So:
  - When a node is expanded, store its state into VISITED
  - When a new node N is generated:
    - If STATE(N) is in VISITED, discard N
    - If there exits a node N’ in the frontier such that STATE(N’) = STATE(N), discard the node -- N or N’ – w/highest cost
Search Algorithm #3

**GRAPH-SEARCH(initial-state)**
1. If GOAL?(initial-state) then return initial-state
2. INSERT(initial-node,FRONTEXIER)
3. Repeat:
4. If empty(FRONTEXIER) then return failure
5. \( N \leftarrow \text{REMOVE}(\text{FRONTEXIER}) \)
6. \( s \leftarrow \text{STATE}(N) \)
7. Add \( s \) to VISITED
8. If GOAL?(\( s \)) then return path or goal state
9. For every state \( s' \) in SUCCESSORS( )
10. Create a new node \( N' \) as a child of \( N \)
11. If \( s' \) is in VISITED then discard \( N' \)
12. If there is \( N'' \) in FRONTEXIER with STATE(\( N' \))=STATE(\( N'' \))
13. If g(\( N'' \)) is lower than g(\( N' \)) then discard \( N' \)
14. Otherwise discard \( N'' \)
15. INSERT(\( N', \text{FRONTEXIER} \))

Uninformed Search Summary

- Many variations on same basic algorithm

- Key differences:
  - How \text{frontier} is implemented (FIFO, LIFO, priority queue)
  - When \text{goal test} is applied
  - Whether \text{visited} list is maintained

- Big impact on:
  - Completeness
  - Optimality
  - Complexity