CompSci 370
Uninformed Search

Ron Parr
Department of Computer Science
Duke University

With thanks to Vince Conitzer for some slides and figures and thanks to Kris Hauser for many slides

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, possibly 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
    - May be constant or varying per edge
      (initially we assume constant)
    - Assumed to be > 0

- Optimal solution = lowest path cost to goal

Example: Path Planning, e.g. Google Maps

Find shortest source to destination using available roads
Other Search Problems

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

Robot Path Planning

What is the state space?
Formulation #1

Cost of one horizontal/vertical step = 1
Cost of one diagonal step = $\sqrt{2}$

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

Take Home Points

- States = modeling choice about the world

- Trade offs often exist:
  - Example 1: Discretization is easy to work with, but optimal solution to may be suboptimal in the real world
  - Example 2: More clever representations may require ingenuity to discover, or use, but may have benefits in real world

- Always keep modeling and solving distinct in your head
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

![Initial state and Goal state](image)

**State**: Arrangement of 8 numbered tiles & empty tile on a 3x3 board
15-Puzzle

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

Sam Loyd, Journalist and Advertising Expert,

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

```
1  2  3  4
5  6  7  8
9 10 11 12
13 14 15
```

```
1  2  3  4
5  6  7  8
9 10 11 12
13 15 14
```
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

- Key difference from algorithms class, where it is typically assumed that graph fits in memory

---

8-, 15-, 24-Puzzles

- 8-puzzle $\rightarrow$ 362,880 states
- 15-puzzle $\rightarrow$ $2.09 \times 10^{13}$ states
- 24-puzzle $\rightarrow$ $10^{15}$ states

- 0.036 sec
- ~ 55 hours
- > 10$^9$ years

100 million states/sec
Intractability

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

Searching the State Space

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

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)

Implementation

- Implementation of node data structure depends upon
  - Implementation
  - Programming language
  - Information expected by user

- Example 1: If all you care about is minimizing the number of steps, there is no need to store cost

- Example 2: Return path (sequence of states) associated with solution by storing path to reach a node as part of the node data structure

- Example 3: Avoid storing path explicitly, but reconstruct it using links from nodes to their parents.
Node expansion

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

node generation ≠ 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
- Implemented as a priority queue FRONTIER
  - INSERT(node, FRONTIER)
  - REMOVE(FRONTIER)
- The ordering of the nodes in FRONTIER defines the search strategy

Generic Tree Search (**assumes** state space is a tree)

**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)
Solution to the Search Problem

- A solution is a path connecting the initial node to a goal node (any goal)
- 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

Recall: Typically assume costs > 0
Breadth-First Search

• FRONTIER is a FIFO Queue

1
2
3
4
5
6
7
FRONTIER = (1)

Note: Typically assume that ties broken in left-to-right order.

Breadth-First Search

• FRONTIER is a FIFO Queue

1
2
3
4
5
6
7
FRONTIER = (2, 3)
Breadth-First Search

- FRONTIER is a FIFO Queue

FRONTIER = (3, 4, 5)

FRONTIER = (4, 5, 7)
BFS Properties

- Assume
  - Branching factor: \( b \)
  - Depth of shallowest goal: \( d \)
- Completeness: \( Y \)
- Optimality: \( (Y \) for constant cost, \( N \) for arbitrary cost) \)
- Time complexity (nodes generated): \( O(b^{d+1}) \)
- Space complexity: \( O(b^d) \)

Note: We are counting nodes generated in time complexity; textbook counts nodes expanded (so exponent can be 1 less).

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>( \sim 10^6 )</td>
<td>1 sec</td>
<td>100 Mb</td>
</tr>
<tr>
<td>8</td>
<td>( \sim 10^8 )</td>
<td>100 sec</td>
<td>10 Gbytes</td>
</tr>
<tr>
<td>10</td>
<td>( \sim 10^{10} )</td>
<td>2.8 hours</td>
<td>1 Tbyte</td>
</tr>
<tr>
<td>12</td>
<td>( \sim 10^{12} )</td>
<td>11.6 days</td>
<td>100 Tbytes</td>
</tr>
<tr>
<td>14</td>
<td>( \sim 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 BFS

\[ \frac{b^d}{2} + \frac{b^d}{2} \ll b^d \]

Issues with Bi-directional BFS

- Uniqueness of goal
  - Suppose goal is parking your car at airport
  - Huge no. of possible goal states
    - Configurations of other vehicles
    - Which space you use

- Invertability of actions
Depth-First Search

- FRONTIER is a LIFO Queue

FRONTIER = (1)

FRONTIER = (2, 3)
Depth-First Search

- FRONTIER is a LIFO Queue

FRONTIER = (4, 5, 3)
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: (Y for finite trees, N for infinite trees)
• Optimality: N
• Time complexity: \(O(b^{m+1})\) (\(m = \text{depth we hit}, m>d?\))
• Space complexity: \(O(bm)\) (bounded for trees)
Iterative Deepening

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

Note: The IDDFS slides are animated, showing DFS running down to the red line on each slide.
IDDFS Properties

- Completeness: $\gamma$
- Optimality: (whenever BFS is optimal)
- Time complexity: $O(b^{d+2})$
- Space complexity: $O(bd)$

**IDDFS vs. BFS**

Theorem: IDDFS generates no more than twice as many nodes for a binary tree as BFS.

Proof: Assume the tree bottoms out at depth $d$, BFS generates:

$$2^{d+1} - 1$$

In the worst case, IDDFS does no more than:

$$\sum_{i=0}^{d} (2^{i+1} - 1) = \sum_{i=0}^{d} 2^{i+1} - \sum_{i=0}^{d} 1 = 2(2^{d+1} - 1) - (d + 1) < 2(2^{d+1} - 1) = 2 \times BFS(d)$$

What about $b$-ary trees? IDDFS relative cost is lower!
Non-constant Costs

- Arcs between states can have variable costs

- The cost of the path to each node $N$ is $g(N) = \sum \text{costs of arcs from root to } N$

- Breadth-first is no longer optimal with variable arc costs!

Uniform-Cost Search (UCS)

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

Suboptimal path!
(how to fix this?)
Search Algorithm #2

**TREE-SEARCH2**
- If GOAL?(initial-state) then return initial-state
- Insert(initial-node,FRONTIER)
- Repeat:
  - If empty(FRONTIER) then return failure
  - **N** ← REMOVE(FRONTIER)
  - **s** ← 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.

Now, UCS is optimal!

Avoiding Revisited States

- Requires comparing state descriptions
- Applied to 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 (e.g. python dictionary) or as explicit data structure with flags
Avoiding Revisited States in DFS

• Depth-first search:
  – Solution 1 (similar to BFS approach):
    • Store all states in current path in VISITED
    • If the state of a new node is in VISITED, then discard the node
      • Only avoids loops
  – Solution 2:
    • Store all generated states in VISITED
    • If the state of a new node is in VISITED, then discard the node
    • Avoids ever revisiting the same state twice
    • Same space complexity as breadth-first!

Avoiding Revisited States in Uniform-Cost Search

• UCS property: For any state $S$, when the first node $N$ such that $\text{STATE}(N) = S$ 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 $\text{STATE}(N)$ is in VISITED, discard $N$
    • If there exists a node $N'$ in the frontier such that $\text{STATE}(N') = \text{STATE}(N)$, 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,FRON TIER)
3. Repeat:
4. If empty(FRON TIER) then return failure
5. \( N \leftarrow \text{REMO VE(FRON TIER)} \)
6. \( s \leftarrow \text{STATE}(N) \)
7. Add \( s \) to VIS ITED
8. For every state \( s' \) in SUCCESSORS(\( s \))
9. Create a new node \( N' \) as a child of \( N \)
10. If \( s' \) is in VIS ITED then discard \( N' \)
11. else if there is \( N'' \) in FRON TIER with \( \text{STATE}(N') = \text{STATE}(N'') \)
12. If \( g(N') \) is lower than \( g(N'') \) then discard \( N' \)
13. else discard \( N'' \)
14. INSERT(\( N' \),FRON TIER) (if not discarded above)

Uninformed Search Summary

- Many variations on same basic algorithm

- Key differences:
  - How **frontier** is implemented (FIFO, LIFO, priority queue)
  - When **goal test** is applied
  - Whether and how assiduously **visited** list is maintained

- Big impact on:
  - Completeness
  - Optimality
  - Complexity