Analysis of Uninformed Search Methods

Brian C. Williams
16.410-13
Sep 15th, 2010

Assignments

• Assignment:
  – Problem Set #2: Uninformed Search out today, due Wednesday, September 22nd, 2010.

• Reading:
  – Monday: Proofs & Induction, Lectures 2 and 3 of 6.042J.
Outline

• Review
• Analysis
  – Depth-first search
  – Breadth-first search
• Iterative deepening

Autonomous Systems:
  • Plan complex sequences of actions
  • Schedule tight resources
  • Monitor and diagnose behavior
  • Repair or reconfigure hardware.

⇒ formulate as state space search.
Formalizing Graph Search

**Input:** A search problem $SP = <g, S, G>$ where
- graph $g = <V, E>$,
- start vertex $S$ in $V$, and
- goal vertex $G$ in $V$.

**Output:** A simple path $P = <S, v_2, \ldots, G>$ in $g$ from $S$ to $G$. 
(i.e., $<v_i, v_{i+1}> \in E$, and $v_i \neq v_j$ if $i \neq j$).

Graph Search is a Kind of State Space Search

Graph Search is a Kind Of Tree Search
Solution: Depth First Search (DFS)

Solution: Breadth First Search (BFS)
**Pseudo Code For Simple Search**

Let $g$ be a Graph
$S$ be the Start vertex of $g$
$G$ be the Goal vertex of $g$
$Q$ be a list of simple partial paths in $G$,

1. Initialize $Q$ with partial path $(S)$ as only entry; set Visited = ( );
2. If $Q$ is empty, fail. Else, pick some partial path $N$ from $Q$;
3. If head$(N) = G$, return $N$; (goal reached!)
4. Else
   a) Remove $N$ from $Q$;
   b) Find all children of head$(N)$ (its neighbors in $g$) not in Visited and create a one-step extension of $N$ to each child;
   c) Add to $Q$ all the extended paths;
   d) Add children of head$(N)$ to Visited;
   e) Go to step 2.

**Solution: Depth First Search (DFS)**

Depth-first:
Add path extensions to front of $Q$
Pick first element of $Q$

**Solution: Breadth First Search (BFS)**

Breadth-first:
Add path extensions to back of $Q$
Pick first element of $Q
Outline

• Review
• Analysis
  – Depth-first search
  – Breadth-first search
• Iterative deepening

Elements of Algorithm Design

Description: (last Monday)
  – Problem statement.
  – Stylized pseudo code, sufficient to analyze and implement the algorithm.
  – Implementation (last Wednesday).

Analysis: (today)
• Performance:
  – Time complexity:
    • how long does it take to find a solution?
  – Space complexity:
    • how much memory does it need to perform search?

• Correctness: (next Monday)
  – Soundness:
    • when a solution is returned, is it guaranteed to be correct?
  – Completeness:
    • is the algorithm guaranteed to find a solution when there is one?
Performance Analysis

Analysis of run-time and resource usage:
- Helps to understand *scalability*.
- Draws line between *feasible* and *impossible*.

- A function of program input.
- Parameterized by input size.
- Seeks upper bound.

Types of Analyses

**Worst-case:**
- \( T(n) = \text{maximum time of algorithm on any input of size } n \).

**Average-case:**
- \( T(n) = \text{expected time of algorithm over all inputs of size } n \).
  - Requires statistical distribution on inputs.

**Best-case:**
- \( T(n) = \text{minimum time of algorithm on any input} \).
Analysis uses *Machine-independent* Time and Space

Performance depends on computer speed:
- Relative speed (run on same machine)
- Absolute speed (on different machines)

Big idea:
- Ignore machine-dependent constraints
- Look at growth of \( T(n) \) as \( n \to \infty \)

“Asymptotic Analysis”

---

Asymptotic notation

O-notation (upper bounds):
- \( 2n^2 = O(n^3) \)
  
  means \( 2n^2 \leq cn^3 \) for *sufficiently large* \( c \) & \( n \)

- \( f(n) = O( g(n) ) \)
  
  if there exists constants \( c > 0, n_0 > 0 \)
  
  such that \( 0 \leq f(n) \leq c g(n) \) for all \( n \geq n_0 \).
Set definition of O-notation

\[ O(n^3) = \{ \text{all functions bounded by } cn^3 \} \]

\[ 2n^2 \in O(n^3) \]

\[ O(g(n)) = \{ f(n) \mid \text{there exists constants } c > 0, n_0 > 0 \text{ such that } 0 \leq f(n) \leq c g(n) \text{ for all } n \geq n_0 \} \]

Performance and Resource Usage

Which is better, depth-first or breadth-first?

<table>
<thead>
<tr>
<th>Search Method</th>
<th>Worst Time</th>
<th>Worst Space</th>
<th>Shortest Path?</th>
<th>Guaranteed to find path?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth-first</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Breadth-first</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Analyzing Time and Space Complexity of Search in Terms of Trees

- **b** = maximum branching factor, number of children
- **d** = depth of the shallowest goal node
- **m** = maximum length of any path in the state space

Worst Case Time for Depth-first

worst case time \( T \) is proportional to number of nodes visited

\[
T_{dfs} = \frac{[b^m + \ldots + b + 1] \times c_{dfs}}{[b - 1] \times c_{dfs}} \text{ where } c_{dfs} \text{ is time per node}
\]

Solve recurrence

Brian Williams, Fall 2016

10
Cost Using Order Notation

Worst case time $T$ is proportional to number of nodes visited

Order Notation

- $T(n) = O(e(n))$ if $T \leq c \cdot e$ for sufficiently large $c$ & $n$

$$T_{dfs} = \frac{b^{m+1} - 1}{b - 1} \cdot c_{dfs}$$

- $= O(b^{m+1})$
- $\sim O(b^m)$ as $b \to \infty$ (used in some texts)

Performance and Resource Usage

Which is better, depth-first or breadth-first?

<table>
<thead>
<tr>
<th>Search Method</th>
<th>Worst Time</th>
<th>Worst Space</th>
<th>Shortest Path?</th>
<th>Guaranteed to find path?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth-first</td>
<td>$\sim b^m$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Breadth-first</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Worst case time is proportional to number of nodes visited
Worst case space is proportional to maximal length of Q
Worst Case Space for Depth-first

Worst case space $S_{dfs}$ is proportional to maximum length of Q

- If a node is queued its parent and siblings have been queued, and its parent dequeued.

$S_{dfs} = [(b-1) \times m + 1] \times c_{dfs}$ where $c_{dfs}$ is space per node.

- At most one sibling of a node has its children queued.

$S_{dfs} = [(b-1) \times m + 1] \times c_{dfs}$

$S_{dfs} = O(b^m) + \text{add visited list}$
Performance and Resource Usage

Which is better, depth-first or breadth-first?

<table>
<thead>
<tr>
<th>Search Method</th>
<th>Worst Time</th>
<th>Worst Space</th>
<th>Shortest Path?</th>
<th>Guaranteed to find path?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth-first</td>
<td>$\sim b^m$</td>
<td>$b^*m$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Breadth-first</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Worst case time is proportional to number of nodes visited
Worst case space is proportional to maximal length of Q

Brian Williams, Fall 10

Performance and Resource Usage

Which is better, depth-first or breadth-first?

<table>
<thead>
<tr>
<th>Search Method</th>
<th>Worst Time</th>
<th>Worst Space</th>
<th>Shortest Path?</th>
<th>Guaranteed to find path?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth-first</td>
<td>$\sim b^m$</td>
<td>$b^*m$</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>Breadth-first</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Worst case time is proportional to number of nodes visited
Worst case space is proportional to maximal length of Q

Brian Williams, Fall 10
Which is better, depth-first or breadth-first?

<table>
<thead>
<tr>
<th>Search Method</th>
<th>Worst Time</th>
<th>Worst Space</th>
<th>Shortest Path?</th>
<th>Guaranteed to find path?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth-first</td>
<td>$\sim b^m$</td>
<td>$b^*m$</td>
<td>No</td>
<td>Yes for finite graph</td>
</tr>
<tr>
<td>Breadth-first</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Worst case time is proportional to number of nodes visited
Worst case space is proportional to maximal length of Q
Worst Case Time for Breadth-first

Worst case time $T$ is proportional to number of nodes visited

Consider case where solution is at level $d$ (absolute worst is $m$):

$$T_{bfs} = \left[b^{d+1} + b^d + \ldots + b + 1 - b\right] * c_{bfs}$$

$$= \left[b^{d+2} - b^2 + b - 1\right] / [b - 1] * c_{bfs}$$

$$= O(b^{d+2})$$

$$~ O(b^{d+1})$$ for large $b$
Performance and Resource Usage

Which is better, depth-first or breadth-first?

<table>
<thead>
<tr>
<th>Search Method</th>
<th>Worst Time</th>
<th>Worst Space</th>
<th>Shortest Path?</th>
<th>Guaranteed to find path?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth-first</td>
<td>~b^m</td>
<td>b^*m</td>
<td>No</td>
<td>Yes for finite graph</td>
</tr>
<tr>
<td>Breadth-first</td>
<td>~b^{d+1}</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Worst case time is proportional to number of nodes visited
Worst case space is proportional to maximal length of Q

Worst Case Space for Breadth-first

Worst case space S_{dfs} is proportional to maximum length of Q
Worst Case Space for Breadth-first

Worst case space $S_{dfs}$ is proportional to maximum length of $Q$

$$S_{dfs} = [b^{d+1} - b + 1]c_{dfs} = O(b^{d+1})$$

Performance and Resource Usage

Which is better, depth-first or breadth-first?

<table>
<thead>
<tr>
<th>Search Method</th>
<th>Worst Time</th>
<th>Worst Space</th>
<th>Shortest Path</th>
<th>Guaranteed to find path?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth-first</td>
<td>$b^m$</td>
<td>$b^m$</td>
<td>No</td>
<td>Yes for finite graph</td>
</tr>
<tr>
<td>Breadth-first</td>
<td>$b^{d+1}$</td>
<td>$b^{d+1}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Worst case time is proportional to number of nodes visited
Worst case space is proportional to maximal length of $Q$
Breadth-first Finds Shortest Path

Assuming each edge is length 1, other paths to G must be at least as long as first found.

Performance and Resource Usage

Which is better, depth-first or breadth-first?

<table>
<thead>
<tr>
<th>Search Method</th>
<th>Worst Time</th>
<th>Worst Space</th>
<th>Shortest Path?</th>
<th>Guaranteed to find path?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth-first</td>
<td>$b^m$</td>
<td>$b^*m$</td>
<td>No</td>
<td>Yes for finite graph</td>
</tr>
<tr>
<td><strong>Breadth-first</strong></td>
<td>$b^{d+1}$</td>
<td>$b^{d+1}$</td>
<td>Yes unit length</td>
<td></td>
</tr>
</tbody>
</table>

Worst case time is proportional to number of nodes visited
Worst case space is proportional to maximal length of Q
Perfromance and Resource Usage

Which is better, depth-first or breadth-first?

<table>
<thead>
<tr>
<th>Search Method</th>
<th>Worst Time</th>
<th>Worst Space</th>
<th>Shortest Path?</th>
<th>Guaranteed to find path?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth-first</td>
<td>$\sim b^m$</td>
<td>$b^m$</td>
<td>No</td>
<td>Yes for finite graph</td>
</tr>
<tr>
<td>Breadth-first</td>
<td>$\sim b^{d+1}$</td>
<td>$b^{d+1}$</td>
<td>Yes with length</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Worst case time is proportional to number of nodes visited
Worst case space is proportional to maximal length of Q

The Worst of The Worst

Which is better, depth-first or breadth-first?

- Assume $d = m$ in the worst case, and call both $m$.
- Best-first can’t expand to level $m+1$, just $m$.

<table>
<thead>
<tr>
<th>Search Method</th>
<th>Worst Time</th>
<th>Worst Space</th>
<th>Shortest Path?</th>
<th>Guaranteed to find path?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth-first</td>
<td>$\sim b^m$</td>
<td>$b^m$</td>
<td>No</td>
<td>Yes for finite graph</td>
</tr>
<tr>
<td>Breadth-first</td>
<td>$\sim b^m$</td>
<td>$b^m$</td>
<td>Yes with length</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Worst case time is proportional to number of nodes visited
Worst case space is proportional to maximal length of Q
For best first search, which runs out first – time or memory?

**Growth for Best First Search**

\( b = 10; 10,000 \text{ nodes/sec}; 1000 \text{ bytes/node} \)

<table>
<thead>
<tr>
<th>Depth</th>
<th>Nodes</th>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1,100</td>
<td>.11 seconds</td>
<td>1 megabyte</td>
</tr>
<tr>
<td>4</td>
<td>111,100</td>
<td>11 seconds</td>
<td>106 megabytes</td>
</tr>
<tr>
<td>6</td>
<td>(10^7)</td>
<td>19 minutes</td>
<td>10 gigabytes</td>
</tr>
<tr>
<td>8</td>
<td>(10^9)</td>
<td>31 hours</td>
<td>1 terabyte</td>
</tr>
<tr>
<td>10</td>
<td>(10^{11})</td>
<td>129 days</td>
<td>101 terabytes</td>
</tr>
<tr>
<td>12</td>
<td>(10^{13})</td>
<td>35 years</td>
<td>10 petabytes</td>
</tr>
<tr>
<td>14</td>
<td>(10^{15})</td>
<td>3,523 years</td>
<td>1 exabyte</td>
</tr>
</tbody>
</table>

How Do We Get The Best of Both Worlds?

<table>
<thead>
<tr>
<th>Search Method</th>
<th>Worst Time</th>
<th>Worst Space</th>
<th>Shortest Path?</th>
<th>Guaranteed to find path?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth-first</td>
<td>(\sim b^m)</td>
<td>(b^m)</td>
<td>No</td>
<td>Yes for finite graph</td>
</tr>
<tr>
<td>Breadth-first</td>
<td>(b^{d+1})</td>
<td>(b^{d+1})</td>
<td>Yes for unit leng</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Worst case time is proportional to number of nodes visited
Worst case space is proportional to maximal length of Q
Outline

• Analysis
• Iterative deepening

Iterative Deepening (IDS)

Idea:
• Explore tree in breadth-first order, using depth-first search.
⇒ Search tree to depth 1, ….

called depth-limited search
Iterative Deepening (IDS)

Idea:
• Explore tree in breadth-first order, using depth-first search.
  ➔ Search tree to depth 1, then 2, ....

called depth-limited search
Speed of Iterative Deepening

Compare speed of BFS vs IDS:
• $T_{\text{bfs}} = 1 + b + b^2 + \ldots + b^d + (b^{d+1} - b) \sim O(b^{d+1})$

• $T_{\text{ids}} = (d + 1)1 + (d)b + (d - 1)b^2 + \ldots + 2b^{d-1} + b^d$
  
  
  $= [b^{d+2} + d(b-1) + 1] / [b - 1]^2$

  $\sim O(b^d)$ for lrg $b$

$\Rightarrow$ Iterative deepening performs better than breadth-first!

Brian Williams, Fall 10

Speed of Iterative Deepening

$T_{\text{ids}} = (d + 1)1 + (d)b + (d - 1)b^2 + \ldots + 2b^{d-1} + b^d$

$bT_{\text{ids}} = (d + 1)b + (d)b^2 + (d - 1)b^3 + \ldots + 2b^{d-1} + b^{d+1}$

$(b-1)T_{\text{ids}} = (d + 1) + b + b^2 + b^3 + \ldots + b^d + b^{d+1}$

$(b-1)T_{\text{ids}} = d + \{[b^{d+2} + 1] / [b - 1]\}$

$= [b^{d+2} + d(b-1) + 1] / [b - 1]^2$

$\sim O(b^d)$ for lrg $b$

$\Rightarrow$ Iterative deepening performs better than breadth-first!
Soundness and Completeness
(next Monday)

Soundness:
• All returned solutions are correct.
• Returns only simple paths from S to G.

Completeness:
• Always returns a solution if one exists.
• Returns a simple path from S to G whenever S is connected to G.

Summary
• Most problem solving tasks may be encoded as state space search.
• Basic data structures for search are graphs and search trees.
• Depth-first and breadth-first search may be framed, as instances of a generic search strategy.
• Cycle detection is required to achieve efficiency and completeness.
• Complexity analysis shows that breadth-first is preferred in terms of optimality and time, while depth-first is preferred in terms of space.
• Iterative deepening draws the best from depth-first and breadth-first search.