Uninformed search methods

Milos Hauskrecht
milos@cs.pitt.edu
5329 Sennott Square

Announcements

• Homework 1
  – Access through the course web page
    http://www.cs.pitt.edu/~milos/courses/cs2710/
  – Two things to download:
    • Problem statement
    • C/C++ programs you will need for the assignment
• Due date: September 14, 2004 before the lecture
• Submission:
  – Reports: on the paper at the lecture
  – Programs: electronic submissions
    Submission guidelines:
    http://www.cs.pitt.edu/~milos/courses/cs2710/program-submissions.html
Formulating a search problem

Many challenging problems in practice require search

• **Search (process)**
  – The process of exploration of the search space

• **Search space:**
  – alternatives (objects) among which we search for the solution

• **The efficiency of the search depends on:**
  – The search space and its size
  – Method used to explore (traverse) the search space
  – Condition to test the satisfaction of the search objective
    (what it takes to determine I found the desired goal object)

• **Think twice before solving the problem by search:**
  – Choose the search space and the exploration policy

Uninformed search methods

• Many different ways to explore the state space (build a tree)

**Uninformed search methods:**

  – use only information available in the problem definition

• **Breadth first search**
• **Depth first search**
• **Iterative deepening**
• **Bi-directional search**

**For the minimum cost path problem:**

• **Uniform cost search**
Search methods

Properties of search methods:

- **Completeness.**
  - Does the method find the solution if it exists?

- **Optimality.**
  - Is the solution returned by the algorithm optimal? Does it give a minimum length path?

- **Space and time complexity.**
  - How much time it takes to find the solution?
  - How much memory is needed to do this?

---

Parameters to measure complexities.

- **Space and time complexity.**
  - **Complexities** are measured in terms of parameters:
    - $b$ – maximum branching factor
    - $d$ – depth of the optimal solution
    - $m$ – maximum depth of the state space

**Branching factor**

![Branching factor diagram](image)
Breadth first search (BFS)

- The shallowest node is expanded first

Breadth-first search

- Expand the shallowest node first
- Implementation: put successors to the end of the queue (FIFO)
Breadth-first search

queue
Zerind
Sibiu
Timisoara

CS 2710 Foundations of AI

Breadth-first search

queue
Sibiu
Timisoara
Arad
Oradea

CS 2710 Foundations of AI
Breadth-first search

1. Start with the initial node (Arad).
2. Explore all the neighbors of the current node (Zerind and Sibiu) and add them to the queue.
3. Repeat the process: explore the neighbors of the nodes in the queue, adding them to the queue as they are discovered.

The queue initially contains: Timisoara, Arad, Oradea, Arad, Oradea, Fagaras, Rimnicu Vilcea.

The final state: Arad, Oradea, Arad, Oradea, Fagaras, Rimnicu Vilcea, Arad, Lugoj.
Properties of breadth-first search

• Completeness: Yes. The solution is reached if it exists.

• Optimality: Yes, for the shortest path.

• Time complexity: ?

• Memory (space) complexity: ?
**BFS – time complexity**

<table>
<thead>
<tr>
<th>depth</th>
<th>number of nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>$2^1=2$</td>
</tr>
<tr>
<td>2</td>
<td>$2^2=4$</td>
</tr>
<tr>
<td>3</td>
<td>$2^3=8$</td>
</tr>
</tbody>
</table>

Expanded nodes: \( O(b^d) \)

Total nodes: \( O(b^{d+1}) \)

---

**Properties of breadth-first search**

- **Completeness:** Yes. The solution is reached if it exists.

- **Optimality:** Yes, for the shortest path.

- **Time complexity:**

  \[
  1 + b + b^2 + \ldots + b^d = O(b^d)
  \]

  exponential in the depth of the solution \( d \)

- **Memory (space) complexity:** ?
**BFS – memory complexity**

- Count nodes kept in the tree structure or in the queue

<table>
<thead>
<tr>
<th>depth</th>
<th>number of nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>$2^1 = 2$</td>
</tr>
<tr>
<td>2</td>
<td>$2^2 = 4$</td>
</tr>
<tr>
<td>3</td>
<td>$2^3 = 8$</td>
</tr>
<tr>
<td>d</td>
<td>$2^d$</td>
</tr>
<tr>
<td>d+1</td>
<td>$2^{d+1}$</td>
</tr>
</tbody>
</table>

**Expanded nodes:** $O(b^d)$

**Total nodes:** $O(b^{d+1})$

---

**Properties of breadth-first search**

- **Completeness:** Yes. The solution is reached if it exists.
- **Optimality:** Yes, for the shortest path.
- **Time complexity:**
  \[
  1 + b + b^2 + \ldots + b^d = O(b^d) \\
  \text{exponential in the depth of the solution } d
  \]
- **Memory (space) complexity:**
  \[
  O(b^d) \\
  \text{nodes are kept in the memory}
  \]
**Depth-first search (DFS)**

- The deepest node is expanded first
- Backtrack when the path cannot be further expanded

---

**Depth-first search**

- The deepest node is expanded first
- Implementation: put successors to the beginning of the queue
Depth-first search

CS 2710 Foundations of AI
**Properties of depth-first search**

- **Completeness:** Does it always find the solution if it exists?
- **Optimality:** ?
- **Time complexity:** ?
- **Memory (space) complexity:** ?
Properties of depth-first search

• Completeness: No. Infinite loops can occur.
  Infinite loops imply -> Infinite depth search tree.

• Optimality: does it find the minimum length path?

• Time complexity: ?

• Memory (space) complexity: ?
Properties of depth-first search

- **Completeness:** No. Infinite loops can occur.

- **Optimality:** No. Solution found first may not be the shortest possible.

- **Time complexity:** ?

- **Memory (space) complexity:** ?

---

DFS – time complexity

<table>
<thead>
<tr>
<th>depth</th>
<th>number of nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>$2^1 = 2$</td>
</tr>
<tr>
<td>2</td>
<td>$2^2 = 4$</td>
</tr>
<tr>
<td>3</td>
<td>$2^3 = 8$</td>
</tr>
</tbody>
</table>

Total nodes: $O(b^m)$
Properties of depth-first search

- **Completeness:** No. Infinite loops can occur.

- **Optimality:** No. Solution found first may not be the shortest possible.

- **Time complexity:**
  \[ O(b^m) \]
  exponential in the maximum depth of the search tree \( m \)

- **Memory (space) complexity:** ?

---

DFS – memory complexity

**Count nodes kept in the tree structure or the queue**

<table>
<thead>
<tr>
<th>depth</th>
<th>number of nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2 = b</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>( m )</td>
<td>2</td>
</tr>
</tbody>
</table>

**Total nodes:** \( O(bm) \)
Properties of depth-first search

- **Completeness:** No. Infinite loops can occur.
- **Optimality:** No. Solution found first may not be the shortest possible.
- **Time complexity:**
  \[ O(b^m) \]
  exponential in the maximum depth of the search tree \( m \)
- **Memory (space) complexity:**
  \[ O(bm) \]
  the tree size we need to keep is linear in the maximum depth of the search tree \( m \)

Limited-depth depth first search

- The limit \( l \) on the depth of the depth-first exploration

\[ \text{Limit } l=2 \]

\[ \begin{align*}
\text{Time complexity: } O(b^l) \\
\text{Memory complexity: } O(bl)
\end{align*} \]

- \( l \) - is the given limit

Not explored
Limited depth depth-first search

• Avoids pitfalls of depth first search
• Use cutoff on the maximum depth of the tree
• How to pick the maximum depth?

• Assume: we have a traveler problem with 20 cities
• How to pick the maximum tree depth?

- How to pick the maximum tree depth?
  – We need to consider only paths of length < 20

• Limited depth DFS
• Time complexity: \( O(b^l) \)  
• Memory complexity: \( O(bl) \)
Iterative deepening algorithm (IDA)

- Based on the idea of the limited-depth search, but
- It resolves the difficulty of knowing the depth limit ahead of time.

Idea: try all depth limits in an increasing order.
That is, search first with the depth limit \( l=0 \), then \( l=1, l=2 \), and so on until the solution is reached

Iterative deepening combines advantages of the depth-first and breadth-first search with only moderate computational overhead
Iterative deepening

Cutoff depth = 0

CS 2710 Foundations of AI
Iterative deepening

Cutoff depth = 1
Iterative deepening

Cutoff depth = 1

CS 2710 Foundations of AI
Iterative deepening

Cutoff depth = 1

Iterative deepening

Cutoff depth = 2
Iterative deepening

Cutoff depth = 2

CS 2710 Foundations of AI
Iterative deepening

Cutoff depth = 2

CS 2710 Foundations of AI
Iterative deepening

Cutoff depth = 2

Properties of IDA

- Completeness: ?
- Optimality: ?
- Time complexity: ?
- Memory (space) complexity: ?
Properties of IDA

- **Completeness:** Yes. The solution is reached if it exists. (the same as BFS when limit is always increased by 1)
- **Optimality:** Yes, for the shortest path. (the same as BFS)
- **Time complexity:** ?
- **Memory (space) complexity:** ?

### IDA – time complexity

![Diagram of IDA time complexity]

- Level 0: $O(1)$
- Level 1: $O(b)$
- Level 2: $O(b^2)$
- Level $d$: $O(b^d)$

$O(b^d)$
Properties of IDA

- **Completeness:** Yes. The solution is reached if it exists.
  (the same as BFS)
- **Optimality:** Yes, for the shortest path.
  (the same as BFS)
- **Time complexity:**
  \[ O(1) + O(b^1) + O(b^2) + \ldots + O(b^d) = O(b^d) \]
  exponential in the depth of the solution \( d \)
  worse than BFS, but asymptotically the same
- **Memory (space) complexity:**
  ?

---

### IDA – memory complexity

<table>
<thead>
<tr>
<th>Level 0</th>
<th>Level 1</th>
<th>Level 2</th>
<th>\ldots</th>
<th>Level ( d )</th>
</tr>
</thead>
</table>

\[ O(1) \quad O(b) \quad O(2b) \quad O(db) \quad O(db) \]
Properties of IDA

• **Completeness:** Yes. The solution is reached if it exists.
  (the same as BFS)
• **Optimality:** Yes, for the shortest path.
  (the same as BFS)
• **Time complexity:**
  \[ O(1) + O(b^1) + O(b^2) + \ldots + O(b^d) = O(b^d) \]
  exponential in the depth of the solution \( d \)
  worse than BFS, but asymptotically the same
• **Memory (space) complexity:**
  \[ O(db) \]
  much better than BFS

---

Elimination of state repeats

While searching the state space for the solution we can encounter the same state many times.

**Question:** Is it necessary to keep and expand all copies of states in the search tree?

**Two possible cases:**

(A) **Cyclic state repeats**

(B) **Non-cyclic state repeats**
Elimination of cycles

**Case A:** Corresponds to the path with a cycle

Can the branch (path) in which the same state is visited twice ever be a part of the optimal (shortest) path between the initial state and the goal? **No!!**

Branches representing cycles cannot be the part of the shortest solution and can be eliminated.
Elimination of cycles

How to check for cyclic state repeats:
• Check ancestors in the tree structure
• Caveat: we need to keep the tree.
Do not expand the node with the state that is the same as the state in one of its ancestors.

Elimination of non-cyclic state repeats

Case B: nodes with the same state are not on the same path from the initial state
Is one of the nodes nodeB-1, nodeB-2 better or preferable?
Elimination of non-cyclic state repeats

Case B: nodes with the same state are not on the same path from the initial state
Is one of the nodes nodeB-1, nodeB-2 better or preferable? Yes. nodeB-1 represents the shorter path between the initial state and B

Since we are happy with the optimal solution nodeB-2 can be eliminated. It does not affect the optimality of the solution.

Problem: Nodes can be encountered in different order during different search strategies.
Elimination of non-cyclic state repeats with BFS

Breadth FS is well behaved with regard to non-cyclic state repeats: nodeB-1 is always expanded before nodeB-2
- Order of expansion determines the correct elimination strategy
- we can safely eliminate the node that is associated with the state that has been expanded before

Elimination of state repeats for the BFS

For the breadth-first search (BFS)
- we can safely eliminate all second, third, fourth, etc. occurrences of the same state
- this rule covers both cyclic and non-cyclic repeats !!!

Implementation of all state repeat elimination through marking:
- All expanded states are marked
- All marked states are stored in a hash table
- Checking if the node has ever been expanded corresponds to the mark structure lookup
Elimination of non-cyclic state repeats with DFS

**Depth FS:** nodeB-2 is expanded before nodeB-1
- The order of node expansion does not imply correct elimination strategy
- We need to remember the length of the path between nodes to safely eliminate them

Elimination of all state redundancies

- **General strategy:** A node is redundant if there is another node with exactly the same state and a shorter path from the initial state
  - Works for any search method
  - Uses additional path length information

**Implementation: marking with the minimum path value:**
- The new node is redundant and can be eliminated if
  - It is in the hash table (it is marked), and
  - Its path is longer or equal to the value stored.
- Otherwise, the new node cannot be eliminated and it is entered together with its value into the hash table. (If the state was in the hash table the new path value is better and needs to be overwritten.)
Bi-directional search

- In some search problems we want to find the path from the initial state to the unique goal state (e.g. traveler problem).

**Bi-directional search:**

- Search both from the initial state and the goal state;
- Use inverse operators for the goal-directed search.

When does it help?
- It cuts the size of the search tree by half.

What is necessary?
- Merge the solutions.