Uninformed search methods

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

CS 2710 Foundations of AI
Lecture 3

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, 2005 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

Problem-solving as search

- Many search problems can be converted to graph search problems
- **A graph search problem can be described in terms of:**
  - A set of states representing different world situations
  - Initial state
  - Goal condition
  - Operators defining valid moves between states
- **Two types of search:**
  - Path search: solution is a path to a goal state
  - Configuration search: solution is a state satisfying the goal condition
- **Optimal solution** = a solution with the optimal value
  - shortest path between the two cities, or
  - a desired n-queen configuration
Searching for the solution

**Search**: exploration of the state space through successive application of operators from the initial state and goal testing

**Search tree**: represents a trace of the search process and its exploration fringe, branches of the tree correspond to paths from the initial state that has been explored so far

A branch in the search tree = a path in the graph
General search algorithm

**General-search** \((\text{problem}, \text{strategy})\)
initializethe search tree with the initial state of \text{problem}

\textbf{loop}
  - if there are no candidate states to explore \textbf{return} failure
  - \textbf{choose} a leaf node of the tree to expand next according to \text{strategy}
    - if the node satisfies the goal condition \textbf{return} the solution
    - \textbf{expand} the node and add all of its successors to the tree

\textbf{end loop}
General search algorithm

**General-search** *(problem, strategy)*

initialize the search tree with the initial state of problem

loop
  if there are no candidate states to explore return failure
  choose a leaf node of the tree to expand next according to strategy
  if the node satisfies the goal condition return the solution
  expand the node and add all of its successors to the tree

end loop

- Search methods differ in how they explore the space, that is how they choose the node to expand next !!!!!

Implementation of search

- Search methods can be implemented using the queue structure

**General search** *(problem, Queuing-fn)*

nodes ← Make-queue(Make-node(Initial-state(problem)))

loop
  if nodes is empty then return failure
  node ← Remove-node(nodes)
  if Goal-test(problem) applied to State(node) is satisfied then return node

  nodes ← Queuing-fn(nodes, Expand(node, Operators(node)))

end loop

- Candidates are added to nodes representing the queue structure
Implementation of the search tree structure

- A search tree node is a data-structure constituting part of a search tree

Other attributes:
- state value (cost)
- depth
- path cost

Expand node function – applies Operators to the state represented by the search tree node.

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**

Number of operators
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

queue -> Sibiu
Timisoara
Arad
Oradea
Breadth-first search

queue

Timisoara
Arad
Oradea
Arad
Oradea
Fagaras
Romanic Vilcea

Arad
Zerind
Sibiu

Arad
Oradea
Arad
Oradea
Fagaras
Romanic Vilcea

Arad
Lugoj

Breadth-first search

queue

Arad
Oradea
Arad
Oradea
Fagaras
Romanic Vilcea
Arad
Lugoj
Properties of breadth-first search

- Completeness: ?
- Optimality: ?
- Time complexity: ?
- Memory (space) complexity: ?

- For complexities use:
  • $b$ – maximum branching factor
  • $d$ – depth of the optimal solution
  • $m$ – maximum depth of the search tree

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: ?
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:** ?
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:**
  \[ O(b^d) \]
  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

Depth-first search
Depth-first search

Note: Arad – Zerind – Arad cycle

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>
<tr>
<td>(d)</td>
<td>(2^d)</td>
</tr>
<tr>
<td>(m)</td>
<td>(2^m, 2^{m-d})</td>
</tr>
</tbody>
</table>

Complexity: \(O(b^m)\)
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**

<table>
<thead>
<tr>
<th>depth</th>
<th>number of nodes kept</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>$(b-1)$</td>
</tr>
<tr>
<td>2</td>
<td>$(b-1)$</td>
</tr>
<tr>
<td>3</td>
<td>$(b-1)$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$m$</td>
<td>$2^m$</td>
</tr>
</tbody>
</table>

Complexity: $O(bm)$
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

- **Time complexity:**
  
  \[ O(b^l) \]

- **Memory complexity:**
  
  \[ O(bl) \]

  \( l \) - is the given limit

CS 2710 Foundations of AI
**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?

- **Assume:** we have a traveler problem with 20 cities
  - How to pick the maximum tree depth?
  - **We need to consider only paths of length** < 20

- Limited depth DFS
- **Time complexity:** \( O(b^l) \) \( l \) - is the limit
- **Memory complexity:** \( O(b^l) \)
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

CS 2710 Foundations of AI
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
Iterative deepening

Cutoff depth = 2
Properties of IDA

• Completeness: ?

• Optimality: ?

• Time complexity: 
  
• Memory (space) complexity: 
  
Cutoff depth = 2
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 showing IDA’s time complexity](image)
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>

\[ O(1) \quad O(b) \quad O(2b) \quad O(db) \]

\[ O(db) \]
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?

???

---

**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 (general)

- nodeB-2 may be 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.)