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/cs1571/
  – Two things to download:
    • Problem statement
    • C/C++ programs you will need for the assignment
• Due date: September 10, 2002 before the lecture
• Submission:
  – Reports: on the paper at the lecture
  – Programs: electronic submissions
    Submission guidelines:
    http://www.cs.pitt.edu/~milos/courses/cs1571/program-submissions.html
Problem-solving as search

- Many search problems in practice 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:
  - Configuration search: solution is a state satisfying the goal condition
  - Path search: solution is a path to a goal state
- Optimal solution = a solution with the optimal value
  - E.g. 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 process can be though of as a process of building a search tree, with nodes corresponding to explored states
Search tree: represents all paths from the initial state that has been explored during the search so far
A branch in the search tree = path in the graph

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
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 the 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 can 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.

![State Transition Diagram](image)

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

<table>
<thead>
<tr>
<th>State</th>
<th>Other attributes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 4</td>
<td>- state value (cost)</td>
</tr>
<tr>
<td>6 1</td>
<td>- depth</td>
</tr>
<tr>
<td>7 3</td>
<td>- path cost</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

### 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

Arad
    /   \
Zerind  Sibiu  Timisoara

Breadth-first search

queue ⇒ Sibiu
      Timisoara
      Arad
      Oradea

Arad
    /   \
Zerind  Sibiu  Timisoara

Arad  Oradea
Breadth-first search

queue ➔

CS 1571 Intro to AI
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:
  \[O(b^d)\]
  every node is kept in the memory
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>

Total nodes: $O(b^d)$

BFS – memory 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: $O(b^d)$
**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 diagram]

**CS 1571 Intro to 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.

- **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:**
  \[ O(bm) \]
  linear in the maximum depth of the search tree \( m \)

---

**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>

Total nodes: \( O(b^m) \)
DFS – memory 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 = b</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>m</td>
<td>2</td>
</tr>
</tbody>
</table>

Total nodes: $O(bm)$

Limited-depth depth first search

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

- Time complexity: $O(b^l)$
- Memory complexity: $O(bl)$

Limit $l = 2$

Not explored

- $l$ - is the given limit
Limited depth depth-first search

- Avoids pitfalls of depth first search
- Cutoff on the maximum depth of the tree
- If we know that the solution length is within a limit the solution the algorithm gives is complete
- How to design the limit?
  - 20 cities in the travel problem
  - We need to consider only paths of length < 20
- Without the known limit the search may fail to find the solution
- **Time complexity:** $O(b^l)$ $l$ - is the limit
- **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 algorithm (IDA)

- Progressively increases the limit of the limited-depth depth-first search

Limit 0

Limit 1

Limit 2

Iterative deepening

Cutoff depth = 0
Iterative deepening

Cutoff depth = 1

CS 1571 Intro to AI
Iterative deepening

Cutoff depth = 1

CS 1571 Intro to AI
Iterative deepening

Cutoff depth = 1

Iterative deepening

Cutoff depth = 2
Iterative deepening

Cutoff depth = 2

CS 1571 Intro to AI
Iterative deepening

Cutoff depth = 2
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)
- **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

IDA – time complexity

![Diagram showing IDA's time complexity](image)
IDA – memory complexity

Level 0 Level 1 Level 2 \ldots Level d

\begin{align*}
O(1) & \quad O(b) & \quad O(2b) & \quad O(db) \\
\end{align*}

\[ O(db) \]

Elimination of state repeats

While searching the state space we can reach the same state in many possible ways.

Two cases:
- Cyclic state repeat
- Non-cyclic state repeat
Elimination of state repeats

Should we keep all copies of states around if we want the optimal solution?

State repeat eliminations:
1. Elimination of all cycles. Do not expand the state that is the same as one of its ancestors.
2. Elimination of all repeats. Do not expand the state that has ever been expanded before.

Implementation of state repeat eliminations:
• Case 1.
  – Check ancestors in the tree structure
• Case 2.
  – Check all expanded nodes (can be a very large number)
  – Use state “marking” schemes

Implementation of the case 2 through marking:
• All expanded states are marked
• All marked states are stored in a special structure (hash table)
• Checking if the node has ever been expanded corresponds to the mark structure lookup
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.

How does it helps?
• It cuts the size of the search tree by half.

What is necessary?
• To merge the solutions.
**Travel example with distances [km]**

**Optimal path:** the shortest distance path from Arad to Bucharest

---

**Finding the minimum cost path**

- Expansion of the search tree should be driven by the cost of the current (partially) built path
  
  \[ g(n) \] - cost function; path cost from the initial state to \( n \)

- Expand the node with the minimum cost path first.
  - This is what the breadth first search does when operator costs are all equal to 1.

- The basic algorithm for finding the minimum cost path:
  - **Dijkstra’s shortest path**

- In AI, the strategy goes under the name
  - **Uniform cost search**
**Uniform cost search**

- Expand the node with the minimum path cost first
- **Implementation**: priority queue

![Uniform cost search diagram](image)

1. **Expand the node with the minimum path cost first**
2. **Implementation**: priority queue

![Uniform cost search diagram](image)
Uniform cost search

queue

Arad
Zerind
Sibiu
Timisoara

0
140
118
118

Arad 150
Zerind 75
Sibiu 140
Timisoara 118

Arad
Zerind
Sibiu
Timisoara

0
140
118
118

Arad 150
Zerind 75
Sibiu 140
Timisoara 118

queue

Sibiu 140
Oradea 146
Arad 150
Lugoj 129
Arad 236

g(n)
Properties of the uniform cost search

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

---

**Properties of the uniform cost search**

- **Completeness**: **Yes**, assuming that operator costs are non-negative (the cost of path never decreases)

\[
g(n) \leq g(\text{successor}(n))
\]

- **Optimality**: **Yes**. Returns the least-cost path.

- **Time complexity**:

  number of nodes with the cost \( g(n) \) smaller than the optimal cost

- **Memory (space) complexity**:

  number of nodes with the cost \( g(n) \) smaller than the optimal cost