Comp24412: Symbolic AI
Lecture 4: Search

Ian Pratt-Hartmann

Room KB2.38: email: ipratt@cs.man.ac.uk

2016–17
Outline

Search algorithms

The travelling salesman problem

The Towers of Hanoi

Playing games
Typical planning problem

- Suppose we have a robot able to move blocks on a table
  - it can only pick up one block at a time
  - it can only pick up blocks with nothing on top
  - it can only put blocks down where there is a free space.

Initial situation:
- on(a, table)
- on(c, a)
- on(b, table)
- clear(b)
- clear(c)

Goal conditions:
- on(a, b)
- on(b, c)

- It helps to represent the problem in terms of a tree of possible states.
• In this tree, the nodes represent possible situations and the links represent actions.

\[
\text{move}(c,a,\text{table}) \quad \downarrow \quad [\text{on}(c,a),\text{on}(a,\text{table}),\text{on}(b,\text{table})]
\]

• Obviously, we do not consider paths with repeated states (going ’round in circles).

• The main problem now is that of how to search the tree.
• In **depth-first search**, we search the tree branch-by-branch:

• This search strategy is sometimes referred to as **chronological backtracking**.
• In **depth-first search**, we search the tree branch-by-branch:

![Tree Diagram]

• This search strategy is sometimes referred to as **chronological backtracking**.
• In **depth-first search**, we search the tree branch-by-branch:

• This search strategy is sometimes referred to as **chronological backtracking**.
• In **depth-first search**, we search the tree branch-by-branch:

• This search strategy is sometimes referred to as **chronological backtracking**.
• In **depth-first search**, we search the tree branch-by-branch:

![Depth-first search tree diagram]

• This search strategy is sometimes referred to as **chronological backtracking**.
• In **depth-first search**, we search the tree branch-by-branch:

• This search strategy is sometimes referred to as **chronological backtracking**.
• In **depth-first search**, we search the tree branch-by-branch:

![Diagram of depth-first search with nodes and edges]

• This search strategy is sometimes referred to as **chronological backtracking**.
• Pseudocode for depth-first search

set Queue to be the list [StartNode]

while Queue is non-empty
    let FirstNode be first element of Queue
    if FirstNode is a goal node
        return FirstNode
    else
        delete FirstNode from start of Queue
        add children of FirstNode to front of Queue
    return failure
• This is really easy to implement in Prolog:
  \[\text{dPlan}([\text{Item}|\text{RestofQueue}], \text{Item}) :\-  \\
   \text{success\_node(}\text{Item})\text{.}\]

  \[\text{dPlan}([\text{Item}|\text{RestOfQueue}], \text{Result}) :\-  \\
   \text{possible\_moves(}\text{Item, Children}),  \\
   \text{append(Children, RestOfQueue, NewQueue)},  \\
   \text{dPlan(}\text{NewQueue}, \text{Result})\text{.}\]

• We can then execute depth-first search with a call of the form
  \[\text{dplan([}\text{initial node}\text{], Result)}\]
where \textit{initial node} is a data structure representing the initial node.
• Knowledge about actions can be represented as a data-structure, thus:

\[
\text{action}(\text{move}(X,Y,Z),
    \begin{align*}
    \text{preconds} & (\text{[on}(X,Y),\text{clear}(X),\text{clear}(Z)]) \\
    \text{add} & (\text{[on}(X,Z),\text{clear}(Y)]), \\
    \text{delete} & (\text{[on}(X,Y),\text{clear}(Z)])
    \end{align*}
\]

• The effects of an action, such as move(c,a,b), can be easily computed:

\[
\begin{align*}
\text{[on}(c,a),\text{on}(a,\text{table}),\text{on}(b,\text{table}),\text{clear}(c),\text{clear}(b)] & \Rightarrow \\
\text{[on}(c,b),\text{on}(a,\text{table}),\text{on}(b,\text{table}),\text{clear}(c),\text{clear}(a)]
\end{align*}
\]
• All the work is involved in decisions such as:
  • how to represent the states and actions;
  • how to compute the possible actions and their effects;
  • eliminating already-encountered situations;
  • recovering the plan once a goal state has been found.

• These things vary from problem to problem, but the basic search strategy is completely general.
• Important point: the goals here are considered all together.
• In practical planning applications in AI, the aim is to work on goals independently, and somehow combine the results into a single plan.
• This is non-trivial, because planning for one goal typically undoes previously satisfied goals.
• Our present example is a case in point:

Initial situation:
- on(a, table)
- on(c, a)
- on(b, table)
- clear(b)
- clear(c)

Goal conditions:
- on(a, b)
- on(b, c)
In **breadth-first search**, we search the tree row-by-row:
Breadth-first search

- In **breadth-first search**, we search the tree row-by-row:
Breadth-first search

- In **breadth-first search**, we search the tree row-by-row:
Breadth-first search

- In **breadth-first search**, we search the tree row-by-row:
In **breadth-first search**, we search the tree row-by-row:
- Pseudocode for breadth-first search
  
  set Queue to be the list [StartNode]
  
  while Queue is non-empty
    let FirstNode be first element of Queue
    if FirstNode is a goal node
      return FirstNode
    else
      delete FirstNode from start of Queue
      add children of FirstNode to end of Queue
  
  return failure
Hill-climbing search

- Usually we want to find a plan sooner rather than later.
- In hill-climbing search, we consider the children in order of nearness to goals.

```
set Queue to be the list [StartNode]
while Queue is non-empty
    let FirstNode be first element of Queue
    if FirstNode is a goal node
        return FirstNode
    else
        delete FirstNode from start of Queue
        order children of FirstNode with most promising first
        add ordered children to start of Queue
    return failure
```

- Note that the measure of nearness to goals will be problem-dependent.
Best-first search

- The strategy of **best-first search** is like hill-climbing, except that we order **all** nodes in the queue by nearness to goals.

```plaintext
set Queue to be the list [StartNode]
while Queue is non-empty
    let FirstNode be first element of Queue
    if FirstNode is a goal node
        return FirstNode
    else
        delete FirstNode from start of Queue
        add children of FirstNode to Queue
        re-order Queue with most promising first
return failure
```
A* search

- Often, we want to find a *good* plan, not just *any* plan.
- The A*-search method gives us a simple way to do this.
- Suppose each node has an associated
  - Cost so far (csf)
  - Underestimate of remaining cost (urc)
- The idea is then to examine nodes in increasing order of csf+urc.
• Pseudocode for A* search

set Queue to be the list [StartNode]

while Queue is non-empty
    let FirstNode be first element of Queue
    if FirstNode is a goal node
        return FirstNode
    else
        delete FirstNode from start of Queue
        add children of FirstNode to Queue
        re-order Queue by csf + urc

return failure

• This algorithm is guaranteed to produce an optimal solution if a solution exists. (Assumption: finiteness.)
Outline

Seach algorithms

The travelling salesman problem

The Towers of Hanoi

Playing games
• Here is a famous problem involving the search for an optimal plan.
• Suppose a salesman has to tour each one of \( n \) cities, with given distances between them.
• We would like to find a complete tour which will minimize the total distance travelled.

![Graph](image)

• More formally:

<table>
<thead>
<tr>
<th>TSP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Given:</strong></td>
</tr>
<tr>
<td><strong>Return:</strong></td>
</tr>
</tbody>
</table>
• The time taken to run an exhaustive search procedure for TSP increases exponentially with \( n \).
• In fact, TSP is a **combinatorially hard optimization problem**.
• Specifically, the problem

  \[
  \text{TSP-feasibility} \\
  \text{Given: an} \ n \times n \ \text{symmetric matrix} \ \{c_{i,j}\} \ \text{over} \ \mathbb{N} \ \text{and} \ C \in \mathbb{N}. \\
  \text{Return: Yes if there is a tour of cost} \ < C \ \text{through this matrix;} \\
  \ \text{No otherwise.}
  \]

is known to be **complete for nondeterministic polynomial time**, or, for short, **NP-complete**.
• TSP is a search problem, where we seek a least-cost path to the goal.
• (The need to loop back to the starting point does not really change anything.)
• The A*-method can be used for TSP:
  • the cost-so-far is easy to calculate;
  • various underestimates of remaining costs can be used, the most obvious being the smallest distance in the graph multiplied by the number of remaining nodes.
• However, A* tends to be breadth-first like in its performance (guzzles memory).
• As an exercise, you might try to write a modified depth-first search procedure that finds an optimal path.
• For large $n$, exhaustive search is impractical.
• The following approach may be used instead. Start with any old tour and improve it:

![Diagram](image)

• This transformation is seen to produce an improvement.
• We then do hill-climbing on this problem space until we reach a local optimum.
Outline

Search algorithms

The travelling salesman problem

The Towers of Hanoi

Playing games
• Another well-known example: the **Towers of Hanoi**

• Of course, the problem can be generalized to *n* discs (but always with three poles)
• The states of the TOH problem (with \( n \) discs) can be represented by a data-structure of the following form:

\[
a_1, \ldots, a_n
\]

where \( 1 \leq a_i \leq 3 \) for all \( i, \ (1 \leq i \leq n) \).

• If the discs are identified as 1, \ldots, \( n \) (in increasing order of size), the semantics (meaning) of this data-structure are as follows:

\[\text{For all } i \ (1 \leq i \leq n), \text{ disc } i \text{ is on pole } a_i\]

• Example:
Consider the graph of states for the 1-disc problem:
Here is the corresponding graph for the 2-disc problem:
• and for the 3-disc problem:
• and finally for the 4-disc problem:
• Every state is reachable from every other state, i.e. not like

• The length of the shortest path from the start state to a goal state is \(2^{n-1}\).
• The length of the longest path from the start state to a goal state without repetitions is \(3^{n-1}\).
Some important points about standard search methods (like dfs, bfs, ...) treated in this lecture:

- they are easy to implement;
- they are typically uncompetitive on large problem instances (cf. TSP);
- they use no special information about the structure of the problem (cf. TOH);
- sometimes, there is just no solution (cf. 9-puzzle).

There is a huge literature on TSP (see further reading on website).

There is a huge literature on planning in AI (specifically concerning goal interaction).
Outline

Search algorithms

The travelling salesman problem

The Towers of Hanoi

Playing games
• Search in Game-Playing

- Nodes of the tree represent possible board positions
- Links represent possible moves

white move
black move
white move
black move
• Nodes are either “white” or “black” corresponding to control of moves by different players.

• At some point, board positions must be evaluated. This may involve cut-off.

• The values can be taken to represent utilities for the white player. Thus the white player (the maximizer) wants to maximize them, and the black player (the minimizer) to minimize them.
The **minimax** procedure determines the best move to make assuming optimal playing by the adversary

```plaintext
begin minimax(node)
    if node is at limit of search tree then
        let utility be the utility of node
        return [node, utility]
    else if node is a white node, then
        apply minimax to children of node
        return result which has maximum utility
    else if node is a black node, then
        apply minimax to children of node
        return result which has minimum utility
    end if
end minimax
```
The minimax procedure, as just described, invokes unnecessary searching. Consider the following search tree:
• Implementing $\alpha$-$\beta$ minimax
• At maximizing nodes
  • Get value of $\beta$ from parent
  • Keep a parameter $\alpha$ set $\alpha$ to maximum so far returned by children
  • Return final value of $\alpha$
• At minimizing nodes
  • Get value of $\alpha$ from parent
  • Keep a parameter $\beta$ set $\alpha$ to minimum so far returned by children
  • Return final value of $\beta$
• When processing any node
  • Stop if $\alpha > \beta$. 