Problem Solving and Search

Ulle Endriss
Institute for Logic, Language and Computation
University of Amsterdam

[http://www.illc.uva.nl/~ulle/teaching/pss/]
Table of Contents

Lecture 8: State-Space Representation and Depth-first Search ................ 3
Lecture 9: Breadth-first Search and Iterative Deepening ....................... 33
Lecture 10: Heuristic Search with the A* Algorithm .......................... 57
Lecture 8: State-Space Representation and Depth-first Search
Search Techniques for Artificial Intelligence

Search is a central topic in AI. This part of the course will clarify why search is such an important topic, present a general approach to representing search problems, introduce several search algorithms, and demonstrate how to implement these algorithms in Prolog.

- Motivation: Applications and Toy Examples
- The State-Space Representation
- Basic (Uninformed) Search Techniques:
  - Depth-first Search (several variants)
  - Breadth-first Search
  - Iterative Deepening
- Heuristic-guided (Best-first) Search with the A* Algorithm
- Adversarial Search for Game Playing with the Minimax Algorithm
Plan for Today

In this first lecture on search techniques for AI, we are going to see:

- Motivation: Applications and Toy Examples
- The State-Space Representation (+ example: “Blocks World”)
- Three Depth-first Search Algorithms
Route Planning

Source: Google Maps
Robot Navigation

Source: http://www.ics.forth.gr/cvrl/
Planning in the Blocks World

How can we get from the situation on the left to the one on the right?
The Eight-Queens Problem

Arrange eight queens on a chess board in such a manner that none of them can attack any of the others!

The above is almost a solution, but not quite . . .
Eight-Puzzle

Yet another puzzle ...
Search and Optimisation Problems

All these problems have got a common structure:

- We are faced with an *initial situation* and we would like to achieve a certain *goal*.

- At any point in time we have different simple *actions* available to us (e.g., “turn left” vs. “turn right”). Executing a particular *sequence* of such actions may or may not achieve the goal.

- *Search* is the process of inspecting several such sequences and choosing one that achieves the goal.

- For some applications, each individual action has a certain cost. A search problem where we aim not only at reaching our goal but also at doing so at minimal cost is an *optimisation* problem.
The State-Space Representation

- **State space**: What are the possible states? Examples:
  - Route planning: positions on the map
  - Blocks World: configurations of blocks

  A concrete problem must also specify the *initial state*.

- **Moves**: What are legal moves between states? Examples:
  - Turning $45^\circ$ to the right could be a legal move for a robot.
  - Putting block $A$ on top of block $B$ is *not* a legal move if block $C'$ is currently on top of $A$.

- **Goal state**: When have we found a solution? Example:
  - Route planning: Position = ’Science Park 904’

- **Cost function**: How costly is a given move? Example:
  - Route planning: The cost of moving from position $X$ to position $Y$ could be the distance between the two.
Prolog Representation

For now, we are going to ignore the cost of moving from one node to the next. Thus, for now we only deal with pure search problems.

A problem specification has to include the following:

- The representation of states is problem-specific. In the simplest case, a state is represented by its name (e.g., a Prolog atom).

- move(+State, -NextState)
  Given the current State, instantiate the variable NextState with a possible next state (and all next states upon backtracking).

- goal(+State)
  Succeed in case State represents a goal state.
Example: Modelling the Blocks World

- **State representation**: We use a list of three lists with the atoms a, b, and c somewhere in these lists. Each sublist represents a stack. The first element in a sublist is the top block. The order of the sublists in the main list does not matter. Example:
  
  \[
  \left[ \left[ c, a \right], \left[ b \right], \left[ \right] \right]
  \]

- **Possible moves**: You can move the top block of any stack onto any other stack:

  \[
  \text{move} (\text{Stacks}, \text{NewStacks}) :-
  \]

  select([Top|Stack1], Stacks, Rest),
  select(Stack2, Rest, OtherStacks),
  NewStacks = [Stack1,[Top|Stack2]|OtherStacks].

- **Goal state**: We assume our goal is always to get a stack with a on top of b on top of c (other goals, of course, are possible):

  \[
  \text{goal} (\text{Stacks}) :- \text{member}([a,b,c], \text{Stacks}).
  \]
Searching the State Space

The possible sequences of legal moves together form a tree:

- The *nodes* of the tree are labelled with states (the same state could label many different nodes).
- The initial state is the *root* of the tree.
- For every legal follow-up move of a given state, any node labelled with that state will have a *child* labelled with the follow-up state.
- Every *branch* in the tree corresponds to a sequence of states (and thus also to a sequence of moves).

There are, at least, two ways of moving through such a tree: *depth-first* and *breadth-first* search . . .
**Depth-first Search**

In depth-first search, we start with the root node and completely explore the descendants of a node before exploring its siblings (with siblings being explored in a left-to-right fashion).

Depth-first traversal: $A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$

Implementing depth-first search in Prolog is very easy, because Prolog itself uses depth-first search during backtracking.
Depth-first Search in Prolog

We are going to define a “user interface” like the following for each of our search algorithms:

```prolog
solve_depthfirst(Node, [Node|Path]) :-
    depthfirst(Node, Path).
```

Next the actual algorithm: Stop if the current Node is a goal state; otherwise move to the NextNode and continue to search. Collect the nodes that have been visited in Path.

```prolog
depthfirst(Node, []) :-
goal(Node).

depthfirst(Node, [NextNode|Path]) :-
    move(Node, NextNode),
    depthfirst(NextNode, Path).
```
Testing: Blocks World

It works pretty well for some problem instances . . .

?- solve_depthfirst([[c,b,a],[],[]], Plan).
Plan = [[[c,b,a], [], []],
        [[b,a], [c], []],
        [[a], [b,c], []],
        [[]], [a,b,c], []]]
Yes

... but not for others . . .

?- solve_depthfirst([[c,a],[b],[]], Plan).
ERROR: Out of local stack
Explanation

Debugging reveals that we are stuck in a loop:

?- spy(depthfirst).
[debug]  ?- solve_depthfirst([[c,a],[b],[[]], Plan).
    Call: (9) depthfirst([[c, a], [b], [[]], _G403) ? leap
    Redo: (9) depthfirst([[c, a], [b], [[]], _G403) ? leap
    Call: (10) depthfirst([a], [c, b], [[]], _G406) ? leap
    Redo: (10) depthfirst([a], [c, b], [[]], _G406) ? leap
    Call: (11) depthfirst([], [a, c, b], [[]], _G421) ? leap
    Redo: (11) depthfirst([], [a, c, b], [[]], _G421) ? leap
    Call: (12) depthfirst([[c, b], [a], [[]], _G436) ? leap
    Redo: (12) depthfirst([[c, b], [a], [[]], _G436) ? leap
    Call: (13) depthfirst([[b], [c, a], [[]], _G454) ? leap
    Redo: (13) depthfirst([[b], [c, a], [[]], _G454) ? leap
    Call: (14) depthfirst([], [b, c, a], [[]], _G469) ? leap
    Redo: (14) depthfirst([], [b, c, a], [[]], _G469) ? leap
    Call: (15) depthfirst([[c, a], [b], [[]], _G484) ?
Cycle Detection

The solution is simple: we need to disallow any moves that would result in a loop. Thus, if the next state is already present in the set of nodes visited so far, choose another follow-up state instead.

From now on we are going to use a “wrapper” around the move/2 predicate defined by the application (e.g., the Blocks World):

\[
\text{move-cyclefree}(\text{Visited}, \text{Node}, \text{NextNode}) :- \\
\text{move}(\text{Node}, \text{NextNode}), \\
\text{\texttt{\textbackslash + member}}(\text{NextNode}, \text{Visited}).
\]

\text{Visited} should be instantiated with the list of nodes visited already.

But note that we cannot just replace move/2 by move-cyclefree/3 in depthfirst/2, because Visited is not available where needed.
Cycle-free Depth-first Search in Prolog

Now the nodes will be collected as we go along, so we have to reverse the list of nodes in the end:

```
solve_depthfirst_cyclefree(Node, Path) :-
    depthfirst_cyclefree([Node], Node, RevPath),
    reverse(RevPath, Path).
```

The first argument is an accumulator collecting the nodes visited so far; the second argument is the current node; the third argument will be instantiated with the solution path (which equals the accumulator once we’ve hit a goal node):

```
depthfirst_cyclefree(Visited, Node, Visited) :-
    goal(Node).
```

```
depthfirst_cyclefree(Visited, Node, Path) :-
    move_cyclefree(Visited, Node, NextNode),
    depthfirst_cyclefree([NextNode | Visited], NextNode, Path).
```
Remark: Repetitions and Loops

Note that our “cycle-free” algorithm does not avoid all repetitions. It only avoids repetitions on the same branch, but if the same state occurs on two different branches, then both nodes might get visited.

As long as branching is finite, this still avoids looping.
Testing Again

With our new cycle-free algorithm, we can now get an answer to the query that did cause an infinite loop earlier:

?- solve_depthfirst_cyclefree([[c,a],[b],[[]], Plan).
Plan = [[[c,a],[b],[[]], [[a],[c,b],[[]], [[],[a,c,b],[[]], [[c,b],[a],[[]], [[b],[c,a],[[]], [[],[b],[c,a]], [[a],[c],[b]], [[],[a,c],[b]], [[c],[a],[b]], [[],[c,b],[a]], [[b],[c],[a]], [[],[b,c],[a]], [[c],[b],[a]], [[],[b,a],[c]], [[a],[b,c],[[]], [[],[a,b,c],[[]]]

Yes

But surely there must be a better solution than a path with 16 nodes!
Idea: Restricting Search to Short Paths

A possible solution to our problem of getting an unnecessarily long solution path is to restrict search to “short” paths:

Stop expanding the current branch once it has reached a certain maximal depth (the *bound*) and move on to the next.

Of course, we may miss some solutions further down the current path. On the other hand, we increase the chance of finding a short solution on another branch within a reasonable amount of time.
Depth-bounded Depth-first Search in Prolog

The program is basically the same as for cycle-free depth-first search. We have one additional argument, the Bound (set by the user).

```
solve_depthfirst_bound(Bound, Node, Path) :-
    depthfirst_bound(Bound, [Node], Node, RevPath),
    reverse(RevPath, Path).

depthfirst_bound(_, Visited, Node, Visited) :-
    goal(Node).

depthfirst_bound(Bound, Visited, Node, Path) :-
    Bound > 0,
    move_cyclefree(Visited, Node, NextNode),
    NewBound is Bound - 1,
    depthfirst_bound(NewBound, [NextNode|Visited], NextNode, Path).
```
Testing Again

Now we can generate a short plan for our Blocks World problem, at least if we can guess a suitable value for the bound required as input to the depth-bounded depth-first search algorithm:

?- solve_depthfirst_bound(2, [[c,a],[b],[]], Plan).
No

?- solve_depthfirst_bound(3, [[c,a],[b],[]], Plan).
Plan = [[[c,a], [b], []],
          [[a], [c], [b]],
          [[]], [b, c], [a]],
          [[]], [a, b, c], []]]
Yes
Complexity of Depth-first Search

We want to analyse the complexity of our search algorithms . . .

As there can be infinite loops, in the worst case, the plain depth-first algorithm will never stop. So analyse depth-bounded depth-first search.

Two assumptions:

- Let $d$ be the \textit{maximal depth} allowed. (If we happen to know that no branch in the tree can be longer than $d$, then our analysis will also apply to the other two depth-first algorithms.)

- For simplicity, assume that for every possible state there are \textit{exactly} $b$ possible follow-up states. So $b$ is the \textit{branching factor} of the search tree.

We think of $d$ as the parameter determining the \textit{size} of our problem, and of $b$ as a \textit{constant}. 
Complexity of Depth-first Search (continued)

• What is the worst case?

In the worst case, every branch has length $d$ (or more) and the only node labelled with a goal state is the last node on the rightmost branch. Hence, depth-first search will visit all the nodes in the tree (up to depth $d$) before finding a solution.

• So: how many nodes in a tree of height $d$ with branching factor $b$?

$$\Rightarrow 1 + b + b^2 + b^3 + \cdots + b^d < 2 \cdot b^d$$

Example: $b = 2$ and $d = 2$

$$1 + 2^1 + 2^2 = 2^{2+1} - 1 = 7$$
Recap: The Big-O Notation

Let $n$ be the problem size and let $f(n)$ be the precise complexity.

Suppose $g$ is a “nice” function that is a “good approximation” of $f$. The Big-O Notation is a way of making this mathematically precise.

We say that $f(n)$ is in $O(g(n))$ if and only if there exist an $n_0 \in \mathbb{N}$ and a $c \in \mathbb{R}^+$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

Thus, from some $n_0$ onwards, the difference between $f$ and $g$ will be at most some constant factor $c$.

We have shown that the worst-case time complexity of depth-bounded depth-first search is in $O(b^d)$. We also say that the complexity of this algorithm is exponential in $d$. 
Exponential Complexity

In general, in Computer Science, anything exponential is considered bad news. Indeed, our simple search techniques will usually not work very well (or at all) for larger problem instances.

Suppose the branching factor is $b = 4$ and suppose it takes us 1 millisecond to check one node. What kind of depth bound would be feasible to use in depth-first search?

<table>
<thead>
<tr>
<th>Depth</th>
<th>Nodes</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>21</td>
<td>0.021 seconds</td>
</tr>
<tr>
<td>5</td>
<td>1365</td>
<td>1.365 seconds</td>
</tr>
<tr>
<td>10</td>
<td>1398101</td>
<td>23.3 minutes</td>
</tr>
<tr>
<td>15</td>
<td>1431655765</td>
<td>16.6 days</td>
</tr>
<tr>
<td>20</td>
<td>1466015503701</td>
<td>46.5 years</td>
</tr>
</tbody>
</table>
Space Complexity of Depth-first Search

The good news is that depth-first search is very efficient in view of its memory requirements:

- At any given time, we only need to keep the path from the root to the current node in memory, and—depending on implementation details—possibly also the sibling nodes of each node on that path.
- The length of the path is at most $d + 1$ and each of the nodes on the path will have at most $b - 1$ siblings left to consider.
- Thus, (as $b$ is constant) the worst-case space complexity is $O(d)$. That is, the complexity is linear in $d$.

In fact, because Prolog uses backtracking, sibling nodes do not need to be kept in memory explicitly.
Summary: Depth-first Search Algorithms

We have seen three variants of the basic depth-first search algorithm:

- *plain* depth-first search: sometimes just what you want
- *cycle-free* depth-first search: remember which states you have seen already on the current branch to avoid loops
- *depth-bounded* depth-first search: only explore branches up to a given maximum depth $d$ (our implementation also is cycle-free)

The *time complexity* of depth-first search is *exponential* in the exploration depth $d$ (bad!). The *space complexity* is *linear* (good!).

Above algorithms can be applied to *any* search problem modelled using the *state-space representation* (so far just one example: Blocks World).
Lecture 9: Breadth-first Search and Iterative Deepening
Plan for Today

We are going to introduce two further basic search algorithms:

- Breadth-first Search
- Iterative Deepening

We are also going to see a further example (besides the Blocks World) for modelling a search problem using the search-space representation:

- Solving the Eight-Queens Problem
Breadth-first Search

The problem with (plain and cycle-free) depth-first search is that we may get lost in a very long (or even infinite branch), while there could be another branch leading to a short solution.

The problem with depth-bounded depth-first search is that it can be difficult to correctly estimate a good value for the bound.

Such problems can be overcome by using \textit{breadth-first} search, where we explore (righthand) siblings before children.

![Breadth-first tree](image)

Breadth-first traversal: A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D $\rightarrow$ E $\rightarrow$ F $\rightarrow$ G
Breadth-first Search: Implementation Difficulties

How do we keep track of which nodes we have already visited and how do we identify the next node to go to?

Recall: For depth-first search, in theory, you have to keep track of the current branch, but in Prolog we actually get this functionality for free (Prolog keeps the current branch on its recursion stack).

For breadth-first search, we are going to have to take care of the memory management ourselves.
**Breadth-first Search: Implementation Idea**

The algorithm will maintain a *list of the currently active paths*. Each round of the algorithm consists of three steps:

1. Remove the first path from the list of paths.

2. Generate a new path for every possible follow-up state of the state labelling the last node in the selected path.

3. Append the list of newly generated paths to the end of the list of paths (to ensure paths are really visited breadth-first).
Breadth-first Search in Prolog

Our usual “user interface” takes care of initialising the list of active paths and of reversing the solution path in the end:

\[
\text{solve_breadthfirst(Node, Path) :-}
\]
\[
\text{breadthfirst([[Node]], RevPath),}
\]
\[
\text{reverse(RevPath, Path).}
\]

And here is the actual algorithm:

\[
\text{breadthfirst([[Node|Path]|_], [Node|Path]) :-}
\]
\[
\text{goal(Node).}
\]

\[
\text{breadthfirst([Path|Paths], SolutionPath) :-}
\]
\[
\text{expand_breadthfirst(Path, ExpPaths),}
\]
\[
\text{append(Paths, ExpPaths, NewPaths),}
\]
\[
\text{breadthfirst(NewPaths, SolutionPath).}
\]

Still to do: implement expand_breadthfirst/2
Expanding Branches

Given a Path (in reverse order), generate the list of expanded paths we get by making a single move from the last Node in the input path.

\[
\text{expand\_breadthfirst}([\text{Node}|\text{Path}], \text{ExpPaths}) :- \\
\quad \text{findall}([\text{NewNode}, \text{Node}|\text{Path}], \\
\quad \quad \text{move\_cyclefree}(\text{Path}, \text{Node}, \text{NewNode}), \\
\quad \quad \text{ExpPaths}).
\]
We are now able to find the shortest possible plan for our Blocks World scenario, without having to guess a suitable bound first:

?- solve_breadthfirst([[c,a],[b],[]], Plan).

Plan = [[[c,a], [b], []],
        [[a], [c], [b]],
        [[]],
        [[b,c], [a]],
        [[], [a,b,c], []]]

Yes
Completeness and Optimality

Some good news about breadth-first search:

- Breadth-first search guarantees *completeness*: if there exists a solution, it will be found eventually.
- Breadth-first search also guarantees *optimality*: the first solution returned will be as short as possible.

**Remark:** This interpretation of optimality presupposes that every move has a cost of 1. Proper cost functions to be discussed later.

**Recall:** Depth-first search does *not* ensure optimality (and only the cycle-free variant without depth bound can ensure completeness).
Complexity Analysis of Breadth-first Search

*Time complexity*: In the worst case, we have to search through the entire tree for any search algorithm. Both depth-first and breadth-first search visit each node exactly once, so time complexity is the same.

Let $d$ be the depth of the first solution and let $b$ be the branching factor (again, assumed to be constant for simplicity). Then worst-case time complexity is $O(b^d)$. Bad! (just as before)

*Space complexity*: Big difference. Now we have to store every path visited before, while for depth-first search we only had to keep a single branch in memory. Hence, space complexity is also $O(b^d)$. Bad!

So there is a *trade-off* between memory-requirements on the one hand and completeness/optimality considerations on the other.
Best of Both Worlds

Would like: an algorithm that, like breadth-first search,

(1) ensures *completeness* by visiting every node eventually and
(2) ensures *optimality* by returning the shortest possible solution.

But at the same time, like depth-first search, it should

(3) have very *low memory requirements* (linear space complexity).

Observation: Depth-bounded depth-first search *almost* fits the bill. The only problem is that we may choose the bound either

- *too low* (losing completeness by stopping early) or
- *too high* (becoming too similar to normal depth-first with the danger of getting lost in a single deep branch).

Idea: Run depth-bounded depth-first search again and again, with increasing values for the bound! This is called *iterative deepening*. 
Iterative Deepening

We can specify the iterative deepening algorithm as follows:

(1) Set $n$ to 0.

(2) Run depth-bounded depth-first search with bound $n$.

(3) Stop and return answer in case of success;
   increment $n$ by 1 and go back to (2) otherwise.

However, in Prolog we can find a more compact implementation . . .
Finding a Path from A to B

A central idea in our implementation of iterative deepening in Prolog will be to provide a predicate that can compute a path of moves from a given start node to some end node.

\[
\text{path}(\text{Node}, \text{Node}, [\text{Node}]).
\]

\[
\text{path}(\text{FirstNode}, \text{LastNode}, [\text{LastNode}|\text{Path}]) :-
\]

\[
\quad \text{path}(\text{FirstNode}, \text{PenultimateNode}, \text{Path}),
\]

\[
\quad \text{move\_cyclefree}(\text{Path}, \text{PenultimateNode}, \text{LastNode}).
\]
Iterative Deepening in Prolog

The implementation of iterative deepening now becomes surprisingly easy. We can rely on the fact that Prolog will enumerate candidate paths, of increasing lengths, from the initial node to a goal node.

solve_iterative_deepening(Node, Path) :-
    path(Node, GoalNode, RevPath),
    goal(GoalNode),
    reverse(RevPath, Path).
Example

And it really works:

?- solve_iterative_deepening([[[a,c,b],[],[]], Plan).
Plan = [[[a,c,b], [], []],
         [[c,b], [a], []],
         [[b], [c], [a]],
         [[], [b,c], [a]],
         [[], [a,b,c], []]]
Yes

Note: Iterative deepening will go into an infinite loop when there are no more answers (even when the search tree is finite). Of course, a more sophisticated implementation could avoid this problem.
Complexity Analysis of Iterative Deepening

*Space complexity:* As for depth-first search, at any moment in time we only keep a single path in memory $\sim O(d)$.

*Time complexity:* This seems worse than for the other algorithms, because the same nodes will get generated again and again.

However, time complexity is of the same order of magnitude as before. If we add the complexities for depth-bounded depth-first search for maximal depths $0, 1, \ldots, d$ (somewhat abusing notation), we still get:

$$O(b^0) + O(b^1) + O(b^2) + \cdots + O(b^d) = O(b^d)$$

This follows from the following inequality (we have seen already):

$$b^0 + b^1 + b^2 + \cdots + b^d < 2 \cdot b^d$$

In practice, memory issues are often the greater problem, and iterative deepening is typically the best of the algorithms considered so far.
Comparison of Basic Search Algorithms

Let $b$ be the maximal branching factor in the search tree (taken to be constant) and $d$ the maximal depth of the search tree explored.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time Complexity</th>
<th>Space Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth-first Search</td>
<td>$O(b^d)$</td>
<td>$O(d)$</td>
</tr>
<tr>
<td>Breadth-first Search</td>
<td>$O(b^d)$</td>
<td>$O(b^d)$</td>
</tr>
<tr>
<td>Iterative Deepening</td>
<td>$O(b^d)$</td>
<td>$O(d)$</td>
</tr>
</tbody>
</table>

Note that for plain depth-first search, depth $d$ may be undefined (as branches could be of infinite length).

Both breadth-first search and iterative deepening are complete (no solution is missed) and optimal (the shortest solution is found first).

None of our three depth-first search algorithms is optimal.

Only cycle-free depth-first search is complete.
Summary: Basic Search Algorithms

We have introduced the following general-purpose algorithms:

- **Depth-first search:**
  - Plain version: `solve_depthfirst/2`
  - Cycle-free version: `solve_depthfirst_cyclefree/2`
  - Depth-bounded version: `solve_depthfirst_bound/3`

- **Breadth-first search:** `solve_breadthfirst/2`

- **Iterative deepening:** `solve_iterative_iterative_deepeening/2`

These algorithms (and their implementations, as given on these slides) are applicable to any problem that can be formalised using the state-space approach. The Blocks World is just one example!

Next we will see how to model a second (very different) problem. (We won't have to change our algorithms at all!)
Recall the Eight-Queens Problem

Arrange eight queens on a chess board in such a manner that none of them can attack any of the others!

The above is almost a solution, but not quite . . .
Modelling the Eight-Queens Problem

Imagine you are trying to solve the problem by going through the columns one by one (we’ll do it right-to-left), placing a queen in an appropriate row for each column.

- **States:** States are partial solutions, with a queen placed in columns $n$ to 8, but not 1 to $n - 1$. We represent them as lists of pairs (abusing the built-in infix operator `/`). Example:

  $[4/2, 5/7, 6/5, 7/3, 8/1]$

  The initial state is the empty list: `[]`

- **Moves:** A move amounts to adding a queen in the rightmost empty column. Moves are only legal if the new queen does not attack any of the queens already present on the board.

- **Goal state:** The goal has been achieved once there are 8 queens on the board. By construction, no queen will attack any other.
Specifying the Attack-Relation

The predicate noattack/2 succeeds if the queen given in the first argument position does not attack any of the queens in the list given as the second argument.

\[
\text{noattack}(X/Y, [X1/Y1|Queens]) :-
\text{X} =\neq X1, \quad \% \text{not in same column}
\text{Y} =\neq Y1, \quad \% \text{not in same row}
\text{Y1-Y} =\neq X1-X, \quad \% \text{not on ascending diagonal}
\text{Y1-Y} =\neq X-X1, \quad \% \text{not on descending diagonal}
\text{noattack}(X/Y, Queens).
\]

Examples:

\[
?- \text{noattack}(3/4, [1/8,2/6]). \quad ?- \text{noattack}(2/7, [1/8]).
\]

Yes \hspace{1cm} No
Representing Moves and Goal States

We are now in a position to define the predicates `move/2` and `goal/1` for the eight-queens problem:

- **Moves.** Making a move means adding one more queen \(X/Y\), where \(X\) is the next column and \(Y\) could be anything, such that the new queen does not attack any of the old ones:
  
  \[
  \text{move(Queens, [X/Y|Queens]) :-}
  \]
  
  \[
  \text{length(Queens, Length),}
  \]
  
  \[
  X \text{ is } 8 - \text{Length,}
  \]
  
  \[
  \text{member(Y, [1,2,3,4,5,6,7,8]),}
  \]
  
  \[
  \text{noattack(X/Y, Queens).}
  \]

- **Goal state.** We have achieved our goal once we have placed 8 queens on the board:
  
  \[
  \text{goal(Queens) :- length(Queens, 8).}
  \]
Solution

What is special about (our formalisation of) the eight-queens problem is that there are no cycles or infinite branches in the search tree. Therefore, all of our search algorithms will work.

Here’s the (first) solution found by the plain depth-first algorithm:

?- solve_depthfirst([], Path), last(Path, Solution).
Path = [[], [8/1], [7/5, 8/1], [6/8, 7/5, 8/1], ...]
Solution = [1/4, 2/2, 3/7, 4/3, 5/6, 6/8, 7/5, 8/1]
Yes

Note that here we are not actually interested in the path to the final state, but only the final state itself (hence the use of last/2).
Summary: Modelling Search Problems

We have now seen two examples for modelling basic search problems:

- Blocks World
- Eight-Queens Problem

Although they look very different, they can be modelled using the same general approach, namely the *state-space representation*:

- **States**: define what the set of all possible states is
- **Moves**: for any given state, define the possible next states
- **Goals**: for any given state, define whether it is a goal state

Once modelled this way, all our basic search algorithms can be used. Which works best depends on problem features and our requirements.
Lecture 10: Heuristic Search with the A* Algorithm
Plan for Today

Our complexity analysis of basic search algorithms showed that these algorithms are unlikely to work well for more complex problems.

No way around this: cannot exhaustively inspect huge search space. But: sometimes can use heuristics to figure out which parts of the search space to focus and get workable algorithms that way.

Topics to be covered today:

• optimisation problems: now every each move has a cost
• heuristic functions to estimate cost to reach closest goal state
• family of best-first search algorithms, including the A* algorithm
• implementation and theoretical analysis of A*
Today we consider *optimisation problems* (not plain *search problems*):

- Now every move is associated with a *cost*.
- We look for solution paths that *minimise* overall cost.
- For our implementations, we use `move/3` instead of `move/2`. The third argument is used for the cost of an individual move.
Best-first Search and Heuristic Functions

Recall: For depth-first and breadth-first search, which node in the search tree is explored next only depends on the structure of the tree.

The rationale in best-first search is to expand those paths next that seem the most “promising”. Making the vague idea of what may be promising precise means defining heuristics.

We fix heuristics by means of a heuristic function $h$ that is used to estimate the “distance” of the current node $x$ to a goal node:

$$h(x) = \text{estimated cost from node } x \text{ to closest goal node}$$

The definition of $h$ is highly application-dependent. Examples:

- **Route planning**: straight-line distance to destination, 
- **Eight-puzzle**: number of misplaced tiles, 

Best-first Search Algorithms

There are many different ways of defining a heuristic function $h$. But there are also different ways of using $h$ to decide which path to expand next, giving rise to different best-first search algorithms.

One option is greedy best-first search:

- expand a path with an end node $x$ such that $h(x)$ is minimal
Example: Greedy Best-first Search

Greedy best-first search means always trying to continue with the node that seems (according to $h$) closest to the goal.

This can work well in some case, but in other cases it does not:

Suppose you want to go from $A$ to $D$. Greedy best-first search would move to $B$ first, as it appears to be closer to the goal than $C$, but in fact the path via $C$ is shorter.

Thus, greedy best-first search is not optimal.

Like depth-first search, it is also not complete. (Can you see why?)
The A* Algorithm

The central idea underlying the so-called A* algorithm is to guide best-first search by two parameters:

- the estimated cost to the goal (as given by $h$)
- the actual cost of the path developed so far

Let $x$ be a node, $g(x)$ the actual cost of moving from the initial node to $x$ along the current path, and $h(x)$ the estimated cost of reaching a goal node from $x$. Define $f(x)$ as follows:

$$f(x) = g(x) + h(x)$$

This is the estimated cost of the cheapest path through $x$ leading from the initial node to a goal node. A* is defined as the best-first search algorithm that always expands a node $x$ such that $f(x)$ is minimal.
A* in Prolog

We now give an implementation of A*. Users of this algorithm will have to implement these application-dependent predicates themselves:

- **move(+State, -NextState, -Cost)**
  Given the current State, instantiate NextState with a possible follow-up state and Cost with the associated cost (all possible follow-up states should get generated through backtracking).

- **goal(+State)**
  Succeed in case State represents a goal state.

- **estimate(+State, -Estimate)**
  Given a State, instantiate Estimate with an estimate of the cost of reaching a goal state. This implements the heuristic function $h$. 
A* in Prolog: User Interface

Now we do not maintain a list of paths (as for breadth-first search), but a *list of (reversed) paths* labelled with the current *cost* \( g(x) \) and the current *estimate* \( h(x) \):

General form: Path/Cost/Estimate
Example: \([c, b, a, s]/6/4\)

Our usual “user interface” initialises the list of labelled paths with the path consisting of just the initial node, labelled with cost 0 and the appropriate estimate:

```
solve_astar(Node, Path/Cost) :-
estimate(Node, Estimate),
astar([[Node]/0/Estimate], RevPath/Cost/_),
reverse(RevPath, Path).
```

So for the final output we are not interested in the estimate anymore, but we do report the cost of solution paths.
A* in Prolog: Moves

This predicate serves as a "wrapper" around the `move/3` predicate supplied by the application developer:

\[
\text{move\_astar([Node|Path]/Cost/_, [NextNode,Node|Path]/NewCost/Est)} :- \\
\text{move(Node, NextNode, StepCost),} \\
\text{\+ member(NextNode, Path),} \\
\text{NewCost is Cost + StepCost,} \\
\text{estimate(NextNode, Est).}
\]

After calling `move/3` itself, the predicate (1) checks for cycles, (2) updates the cost of the current path, and (3) labels the new path with the estimate for the new node.

We can now generate all expansions of a given path by a single state:

\[
\text{expand\_astar(Path, ExpPaths) :-}
\text{findall(NewPath, move\_astar(Path,NewPath), ExpPaths).}
\]
**A* in Prolog: Getting the Best Path**

The following predicate implements the *search strategy* of A*:
from a list of labelled paths, select one that minimises the sum of
current cost and current estimate.

\[
\text{get\_best}([\text{Path}], \text{Path}) :- !.
\]

\[
\text{get\_best}([\text{Path1/Cost1/Est1}, _/\text{Cost2/Est2}|\text{Paths}], \text{BestPath}) :-
\]
\[
\text{Cost1 + Est1 =< Cost2 + Est2}, !,
\]
\[
\text{get\_best}([\text{Path1/Cost1/Est1}|\text{Paths}], \text{BestPath}).
\]

\[
\text{get\_best}([_|\text{Paths}], \text{BestPath}) :-
\]
\[
\text{get\_best}(\text{Paths}, \text{BestPath}).
\]

**Remark:** Implementing a different best-first search algorithm only
involves changing `get_best/2`. The rest can stay the same.
A* in Prolog: Main Algorithm

Stop in case the best path ends in a goal node:

\[
\text{astar(Paths, Path) :- }
\text{get\_best(Paths, Path),}
\text{Path = [Node|\_]_/\_/},
\text{goal(Node).}
\]

Otherwise, extract the best path, generate all its expansions, and continue with the union of the remaining and the expanded paths:

\[
\text{astar(Paths, SolutionPath) :-}
\text{get\_best(Paths, BestPath),}
\text{select(BestPath, Paths, OtherPaths),}
\text{expand\_astar(BestPath, ExpPaths),}
\text{append(OtherPaths, ExpPaths, NewPaths),}
\text{astar(NewPaths, SolutionPath).}
\]
Example

\[
\begin{align*}
\text{move}(s, a, 2). & \quad \text{estimate}(a, 5). \\
\text{move}(a, b, 2). & \quad \text{estimate}(b, 4). \\
\text{move}(b, c, 2). & \quad \text{estimate}(c, 4). \\
\text{move}(c, d, 3). & \quad \text{estimate}(d, 3). \\
\text{move}(d, t, 3). & \quad \text{estimate}(e, 7). \\
\text{move}(s, e, 2). & \quad \text{estimate}(f, 4). \\
\text{move}(e, f, 5). & \quad \text{estimate}(g, 2). \\
\text{move}(f, g, 2). & \\
\text{move}(g, t, 2). & \quad \text{estimate}(s, 100). \\
\text{goal}(t). & \quad \text{estimate}(t, 0). 
\end{align*}
\]

Source: Bratko, *Prolog Programming for AI*
Example (continued)

If we run A* on this problem specification, we first obtain the optimal solution path and then one more alternative path:

?- solve_astar(s, Path).
Path = [s, e, f, g, t]/11 ;
Path = [s, a, b, c, d, t]/12 ;
No
Debugging

We can use debugging to reconstruct the workings of A* for this example (trace edited for readability):

?- spy(expand_astar).
Yes

[debug]  ?- solve_astar(s, Path).
Call: (10) expand_astar([s]/0/100, _L233) ? leap
Call: (11) expand_astar([a, s]/2/5, _L266) ? leap
Call: (12) expand_astar([b, a, s]/4/4, _L299) ? leap
Call: (13) expand_astar([e, s]/2/7, _L353) ? leap
Call: (14) expand_astar([c, b, a, s]/6/4, _L386) ? leap
Call: (15) expand_astar([f, e, s]/7/4, _L419) ? leap
Call: (16) expand_astar([g, f, e, s]/9/2, _L452) ? leap

Path = [s, e, f, g, t]/11
Yes
Aside: Using Basic Search Algorithms

To test our basic (uninformed) search algorithms with this data, we can introduce the following rule to map problem descriptions involving a cost function to simple problem descriptions:

\[
\text{move(Node, NextNode)} \leftarrow \text{move(Node, NextNode, _)}. \]

We can now use, say, depth-first search as well:

\[
?- \text{solve_depthfirst}(s, \text{Path}).
\]
\[
\text{Path} = [s, a, b, c, d, t] ; \quad \% \ [\text{Cost} = 12]
\]
\[
\text{Path} = [s, e, f, g, t] ; \quad \% \ [\text{Cost} = 11]
\]

No

Now we (obviously) cannot guarantee the best solution is found first.
Properties of A*

A heuristic function $h$ is called **admissible** if $h(x)$ is never more than the actual cost of the best path from $x$ to a goal node.

An important theoretical result is the following:

*A* with an admissible heuristic function guarantees optimality: the first solution found has minimal cost.

Proof: Let $x$ be any node on an optimal solution path and let $y$ be any non-optimal goal node. We need to show that A* will correctly pick $x$ over $y$. Let $c^*$ be the cost of the optimal solution. Then we get

1. $f(y) = g(y) + h(y) = g(y) + 0 > c^*$ and, due to admissibility of $h$,
2. $f(x) = g(x) + h(x) \leq c^*$. Hence, $f(x) < f(y)$, which means that A* will correctly pick $x$ over $y$. This completes the proof. ✓
Admissible Heuristic Functions

How do we choose a “good” admissible heuristic function?

Two general examples:

- The *trivial heuristic function* \( h_0(x) = 0 \) (for all \( x \)) is admissible. It guarantees optimality, but it is of no help whatsoever in focusing the search. So using \( h_0 \) is *not efficient*.

- The *perfect heuristic function* \( h^* \), mapping any given \( x \) to the *actual* cost of reaching a goal node from \( x \), is also admissible. This function would lead us straight to the best solution (but, of course, *we don’t know* what \( h^* \) is!).

Finding a good heuristic function is often a challenging problem ...
Recall the Route Planning Problem

Source: Google Maps
Examples for Admissible Heuristics

For the route planning domain, here are two heuristic functions:

- Let $h_1(x)$ be the straight-line distance to the goal location. This is an admissible heuristic, because no solution path will ever be shorter than the straight-line connection.

- Let $h_2(x) = 1.2 \cdot h_1(x)$ (adding 20% to the straight-line distance). An intuitive justification would be that there are no completely straight streets, so this would be a better estimate than $h_1(x)$. Indeed, $h_2$ may often work better (be more efficient) than $h_1$. But $h_2$ generally is not admissible, because there could be two locations connected by a street that is almost straight. So $h_2$ does not guarantee optimality.
Recall the Eight-Puzzle

Source: Russell & Norvig, *Artificial Intelligence*
Examples for Admissible Heuristics

For the eight-puzzle, here are two admissible heuristic functions:

- Let $h_3(x)$ be the number of misplaced tiles.
  So $h_3(x)$ will always be a number between 0 and 8.
  This is clearly a lower bound for the number of moves to the goal, so $h_3$ is an admissible heuristic.

- Assume we could freely move tiles without regard for other tiles.
  Let $h_4(x)$ be the number of 1-step moves required to get to the goal configuration under this assumption.
  This is also an admissible heuristic, because in reality we will always need at least $h_4(x)$ moves (and typically more, because other tiles will be in the way). Furthermore, $h_4$ is better than $h_3$, because we have $h_3(x) \leq h_4(x)$ for all nodes $x$. 
Complexity Analysis of A*

Both \textit{worst-case} time and space complexity are \textit{exponential} in the depth of the search tree (as for breadth-first search): in the worst case, we still have to visit \textit{all} the nodes on the tree and ultimately keep the full tree in memory.

The reason why, in spite of the above, A* usually works much better than basic breadth-first search is that the heuristic function will \textit{typically} guide us to the solution much more directly.
Summary: Best-first Search with A*

• Heuristics can be used to guide a search algorithm in a large search space. The central idea of best-first search is to expand the path that seems “most promising”.

• There are different ways of defining a heuristic function $h$ to estimate how far off the goal a given node is, and there are different ways of using $h$ to decide which node is “best”.

• In the A* algorithm, the node $x$ minimising the sum of the cost $g(x)$ to reach the current node $x$ and the estimate $h(x)$ of the cost to reach a goal node from $x$ is chosen for expansion.

• A heuristic function $h$ is called admissible if it never over-estimates the true cost of reaching a goal node.

• If $h$ is an admissible heuristic function, then A* guarantees that an optimal solution will be found (first).
[[slides for final week still to be added]]