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

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.

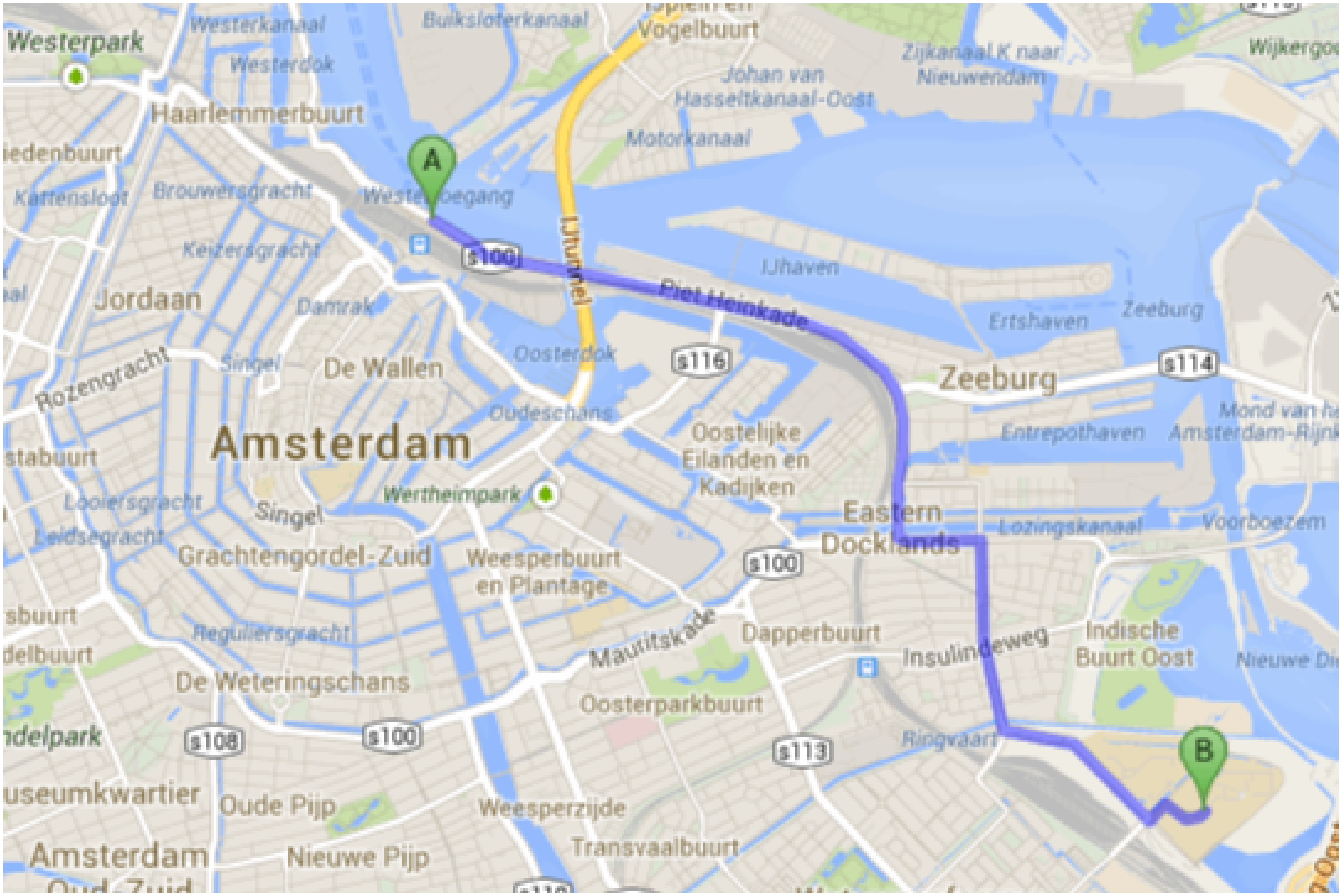
- Motivation: Applications and Toy Examples
- The State-Space Representation
- Basic (Uninformed) Search Techniques:
  - Depth-first Search (several variants)
  - Breadth-first Search
  - Iterative Deepening
- Heuristic-guided (Best-first) Search with the A* Algorithm
- Adversarial Search for Game Playing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

Plan for Today

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

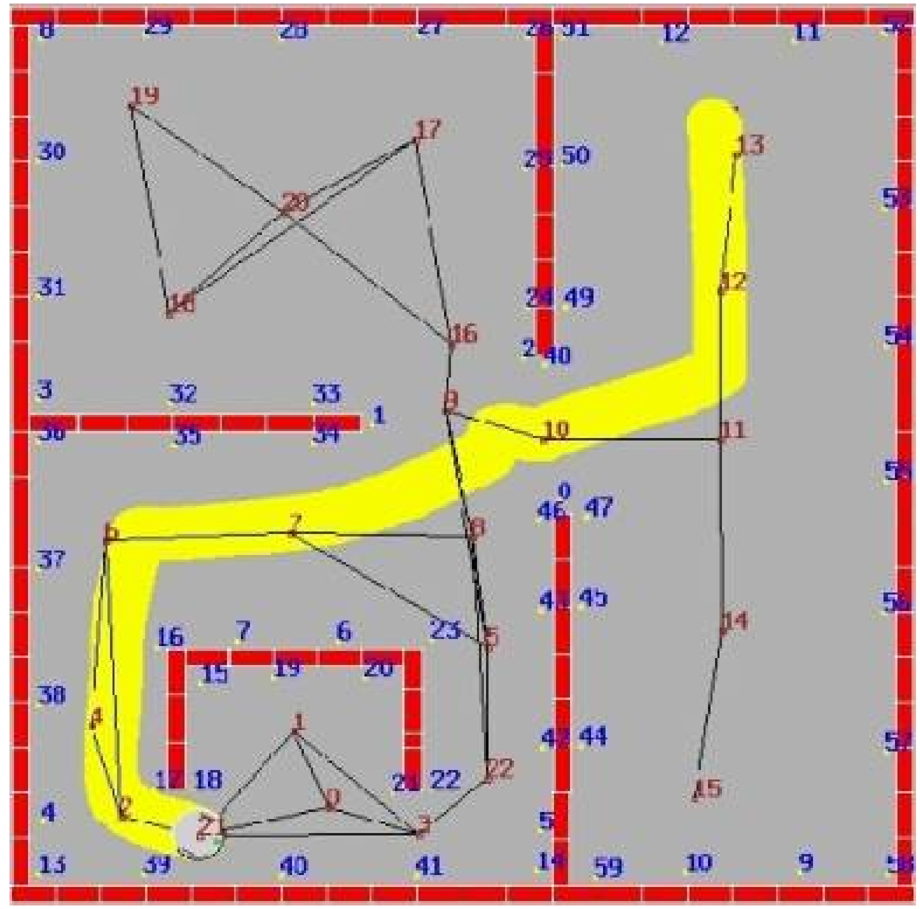
- Motivation: Applications and Toy Examples
- The State-Space Representation (+ example: "Blocks World")
- Three Depth-first Search Algorithms

Route Planning



Source: Google Maps

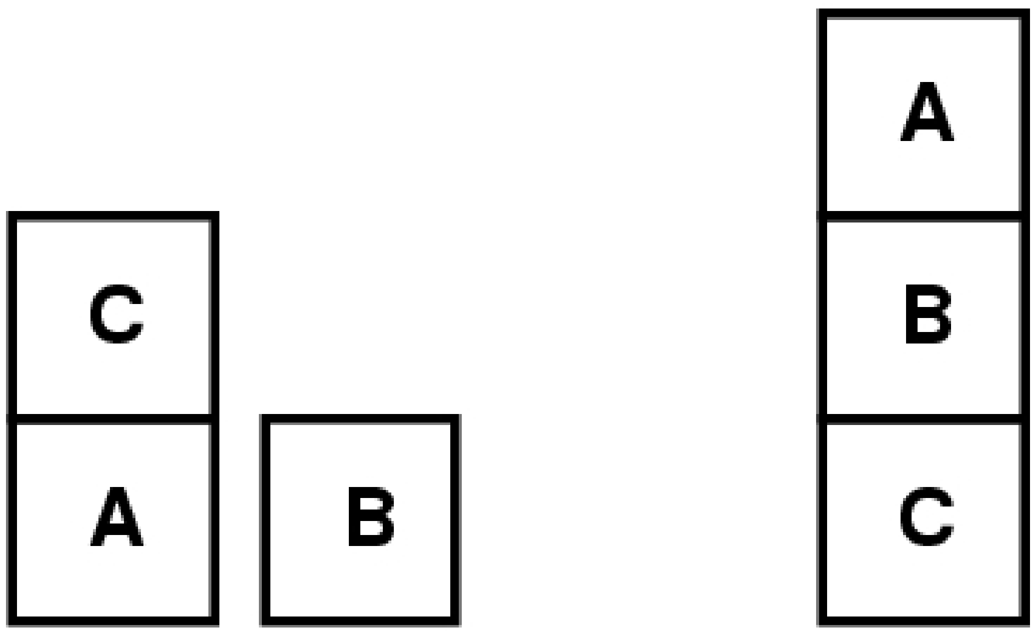
Robot Navigation



Source: `http://www.ics.forth.gr/cvrl/`

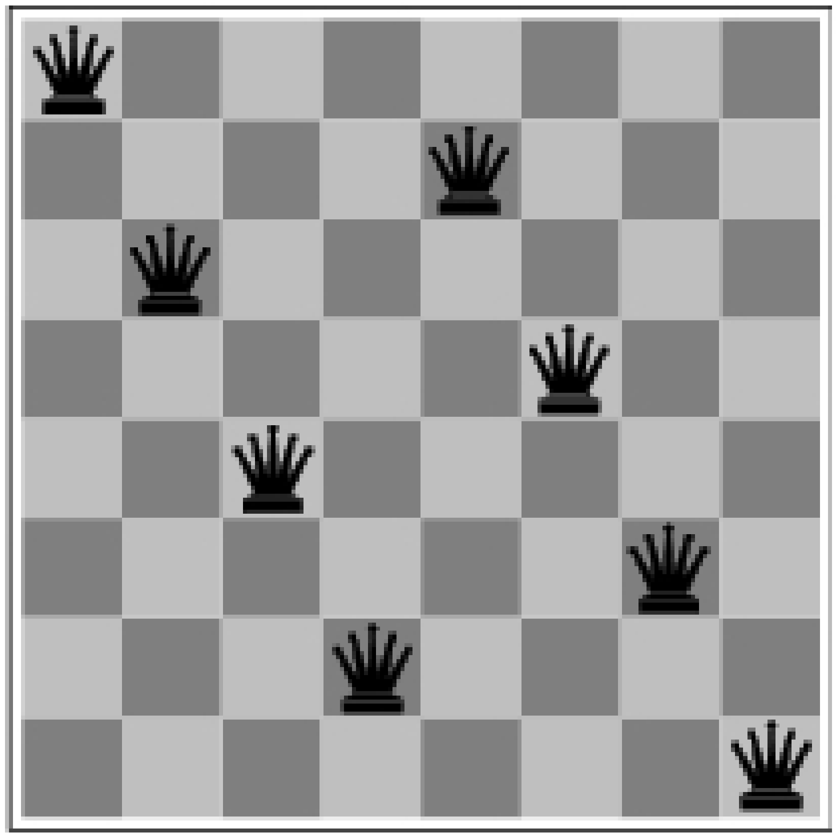
Planning in the Blocks World

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!

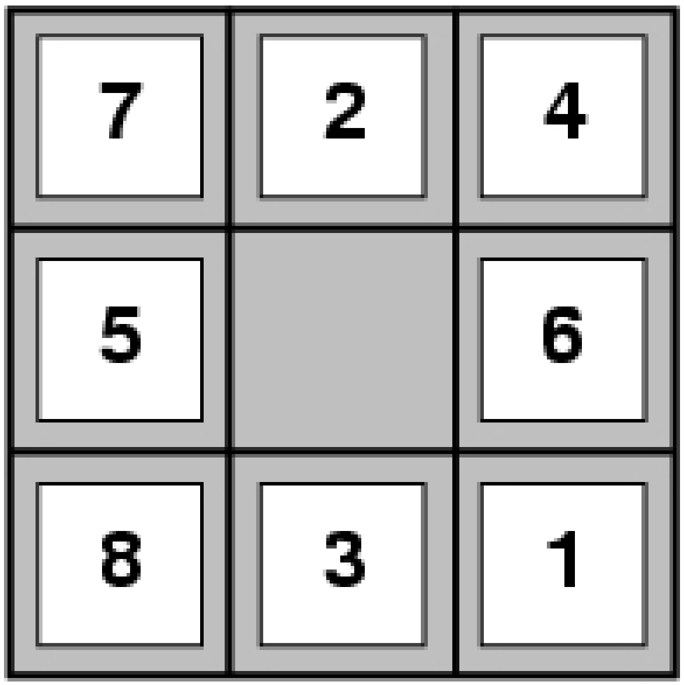


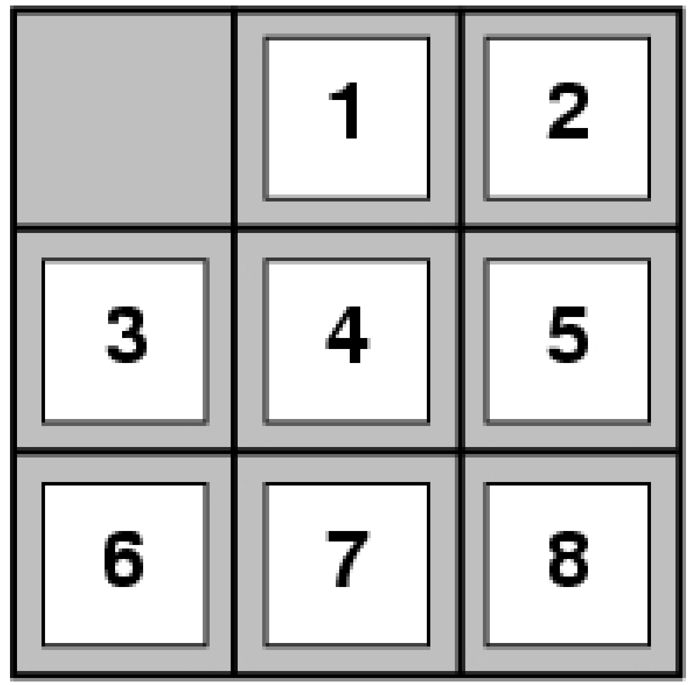
Source: Russell & Norvig, *Artificial Intelligence*

The above is *almost* a solution, but not quite . . .

Eight-Puzzle

Yet another puzzle . . .





Source: Russell & Norvig, *Artificial Intelligence*

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° 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: $\texttt{Position} = \texttt{'Science Park 904'}$

- *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:

  ```
  [ [c,a], [b], [] ]
  ```

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

  ```
  move(Stacks, 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):

  ```
  goal(Stacks) :- member([a,b,c], Stacks).
  ```

Searching the State Space

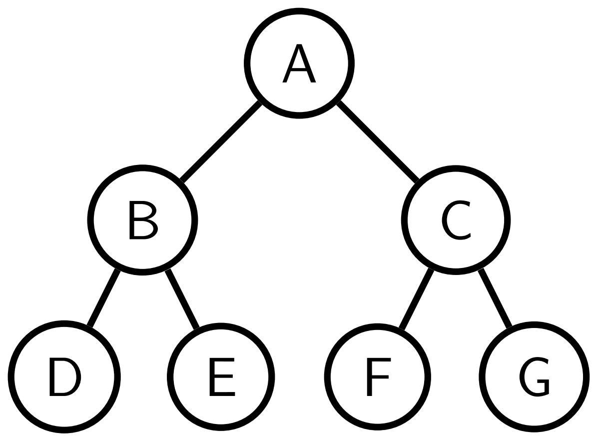
The possible sequences of legal moves together form a *tree:*

- 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:

```
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`.

```
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):

```
move_cyclefree(Visited, Node, NextNode) :-
  move(Node, NextNode),
  \+ member(NextNode, Visited).
```

`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 *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 *exactly* b possible follow-up states. So b is the *branching factor* of the search tree.

We think of d as the parameter determining the *size* of our problem, and of b as a *constant*.

Complexity of Depth-first Search (continued)

- *What is the worst case?*

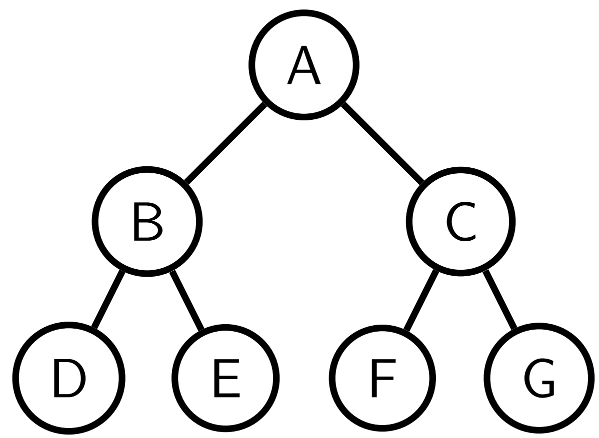
  In the worst case, every branch has length d (or more) and the only node labelled with a goal state is the last node on the rightmost branch. Hence, depth-first search will visit *all* the nodes in the tree (up to depth d) before finding a solution.

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

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

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 $f(n) \leqslant c \cdot g(n)$ for all $n \geqslant n_0$.

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?

Depth	Nodes	Time
2	21	0.021 seconds
5	1365	1.365 seconds
10	1398101	23.3 minutes
15	1431655765	16.6 days
20	1466015503701	46.5 years

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 state-space representation:

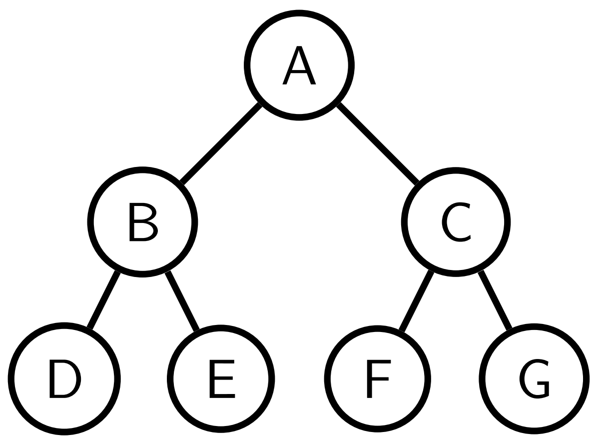
- Solving the Eight-Queens Problem

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 *breadth-first* search, where we explore (righthand) siblings before children.



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:

```
solve_breadthfirst(Node, Path) :-
  breadthfirst([[Node]], RevPath),
  reverse(RevPath, Path).
```

And here is the actual algorithm:

```
breadthfirst([[Node|Path]|_], [Node|Path]) :-
  goal(Node).

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

```
expand_breadthfirst([Node|Path], ExpPaths) :-
  findall([NewNode,Node|Path],
          move_cyclefree(Path,Node,NewNode),
          ExpPaths).
```

Example

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 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.

```
path(Node, Node, [Node]).

path(FirstNode, LastNode, [LastNode|Path]) :-
  path(FirstNode, PenultimateNode, Path),
  move_cyclefree(Path, PenultimateNode, 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 $\rightsquigarrow O(d)$.

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:

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

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

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

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.

Algorithm	Time Complexity	Space Complexity
Depth-first Search	$O(b^d)$	$O(d)$
Breadth-first Search	$O(b^d)$	$O(b^d)$
Iterative Deepening	$O(b^d)$	$O(d)$

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:

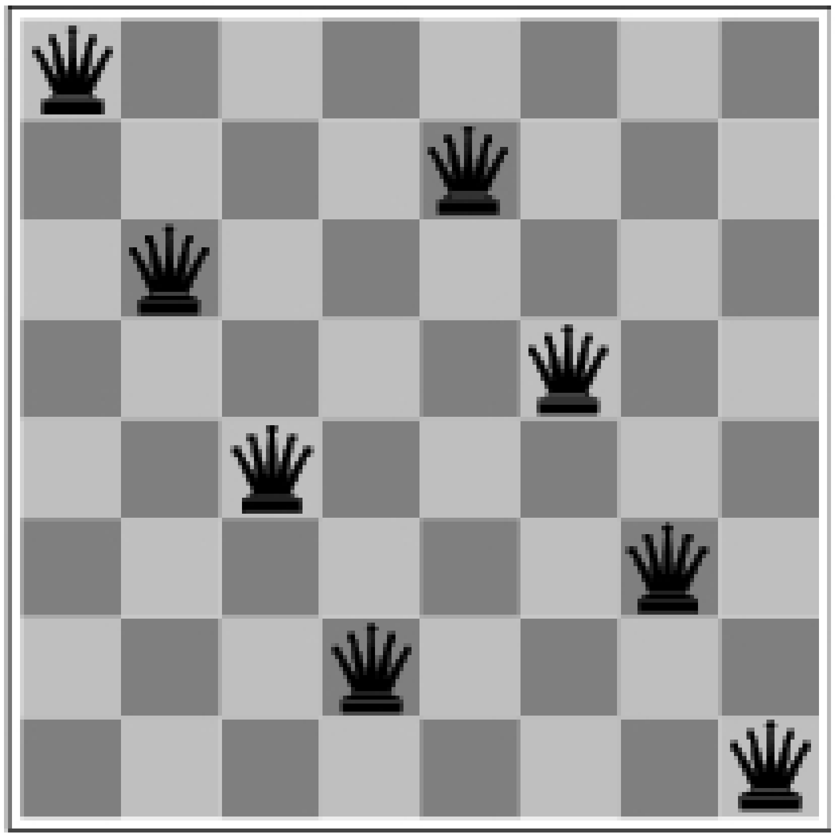
- Depth-first search:
  - Plain version: `solve_depthfirst/2`
  - Cycle-free version: `solve_depthfirst_cyclefree/2`
  - Depth-bounded version: `solve_depthfirst_bound/3`
- Breadth-first search: `solve_breadthfirst/2`
- Iterative deepening: `solve_iterative_deepening/2`

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!



Source: Russell & Norvig, *Artificial Intelligence*

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.

```
noattack(_, []).

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

Examples:

```
?- noattack(3/4, [1/8,2/6]).   ?- noattack(2/7, [1/8]).
Yes                            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:

  ```
  move(Queens, [X/Y|Queens]) :-
    length(Queens, Length),
    X is 8 - Length,
    member(Y, [1,2,3,4,5,6,7,8]),
    noattack(X/Y, Queens).
  ```

- *Goal state*. We have achieved our goal once we have placed 8 queens on the board:

  ```
  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.
<u>But:</u> sometimes can use *heuristics* to figure out which parts of the search space to focus on and get workable algorithms that way.

Topics to be covered today:

- *optimisation problems:* now every 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*

Optimisation Problems

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:

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

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.

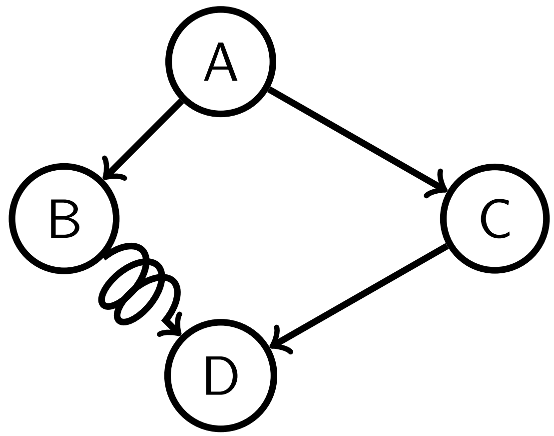
One option is *greedy* best-first search:

> 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:

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

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:

```
move_astar([Node|Path]/Cost/_, [NextNode,Node|Path]/NewCost/Est) :-
  move(Node, NextNode, StepCost),
  \+ member(NextNode, Path),
  NewCost is Cost + StepCost,
  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:

```
expand_astar(Path, ExpPaths) :-
  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.

```
get_best([Path], Path) :- !.

get_best([Path1/Cost1/Est1,_/Cost2/Est2|Paths], BestPath) :-
  Cost1 + Est1 =< Cost2 + Est2, !,
  get_best([Path1/Cost1/Est1|Paths], BestPath).

get_best([_|Paths], BestPath) :-
  get_best(Paths, 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

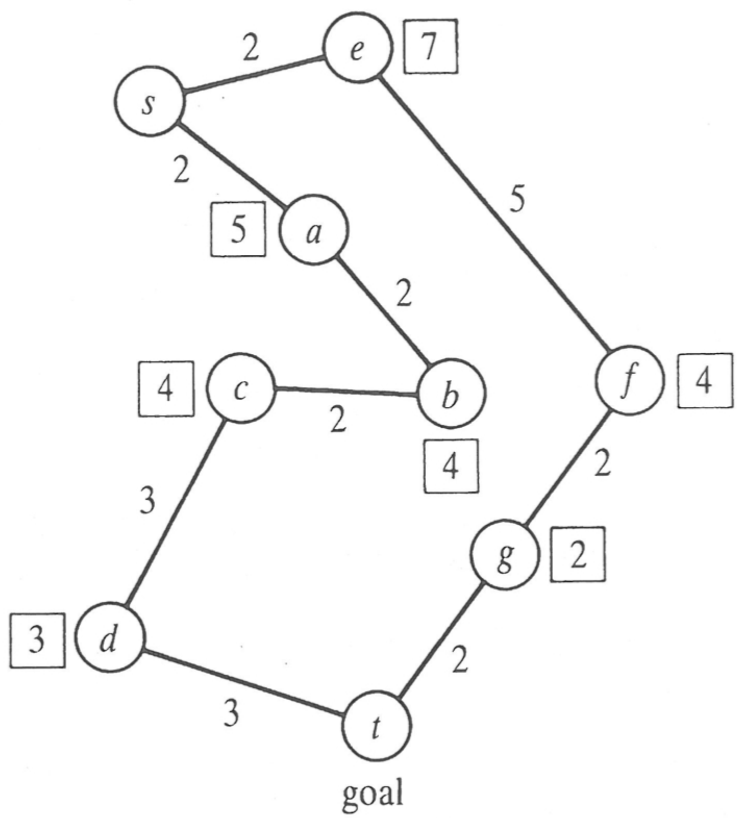
Stop in case the best path ends in a goal node:

```
astar(Paths, Path) :-
  get_best(Paths, Path),
  Path = [Node|_]/_/_,
  goal(Node).
```

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

```
astar(Paths, SolutionPath) :-
  get_best(Paths, BestPath),
  select(BestPath, Paths, OtherPaths),
  expand_astar(BestPath, ExpPaths),
  append(OtherPaths, ExpPaths, NewPaths),
  astar(NewPaths, SolutionPath).
```

Example



Source: Bratko, *Prolog Programming for AI*

```
move(s, a, 2).     estimate(a, 5).
move(a, b, 2).     estimate(b, 4).
move(b, c, 2).     estimate(c, 4).
move(c, d, 3).     estimate(d, 3).
move(d, t, 3).     estimate(e, 7).
move(s, e, 2).     estimate(f, 4).
move(e, f, 5).     estimate(g, 2).
move(f, g, 2).
move(g, t, 2).     estimate(s, 100).
goal(t).           estimate(t, 0).
```

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:

```
move(Node, NextNode) :- move(Node, NextNode, _).
```

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

```
?- solve_depthfirst(s, Path).
Path = [s, a, b, c, d, t] ;    % [Cost = 12]
Path = [s, e, f, g, t] ;       % [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 $c^\star$ be the *cost of the optimal solution*. Then we get (1) $f(y) = g(y) + h(y) = g(y) + 0 > c^\star$ and, due to admissibility of h, (2) $f(x) = g(x) + h(x) \leqslant c^\star$. 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.

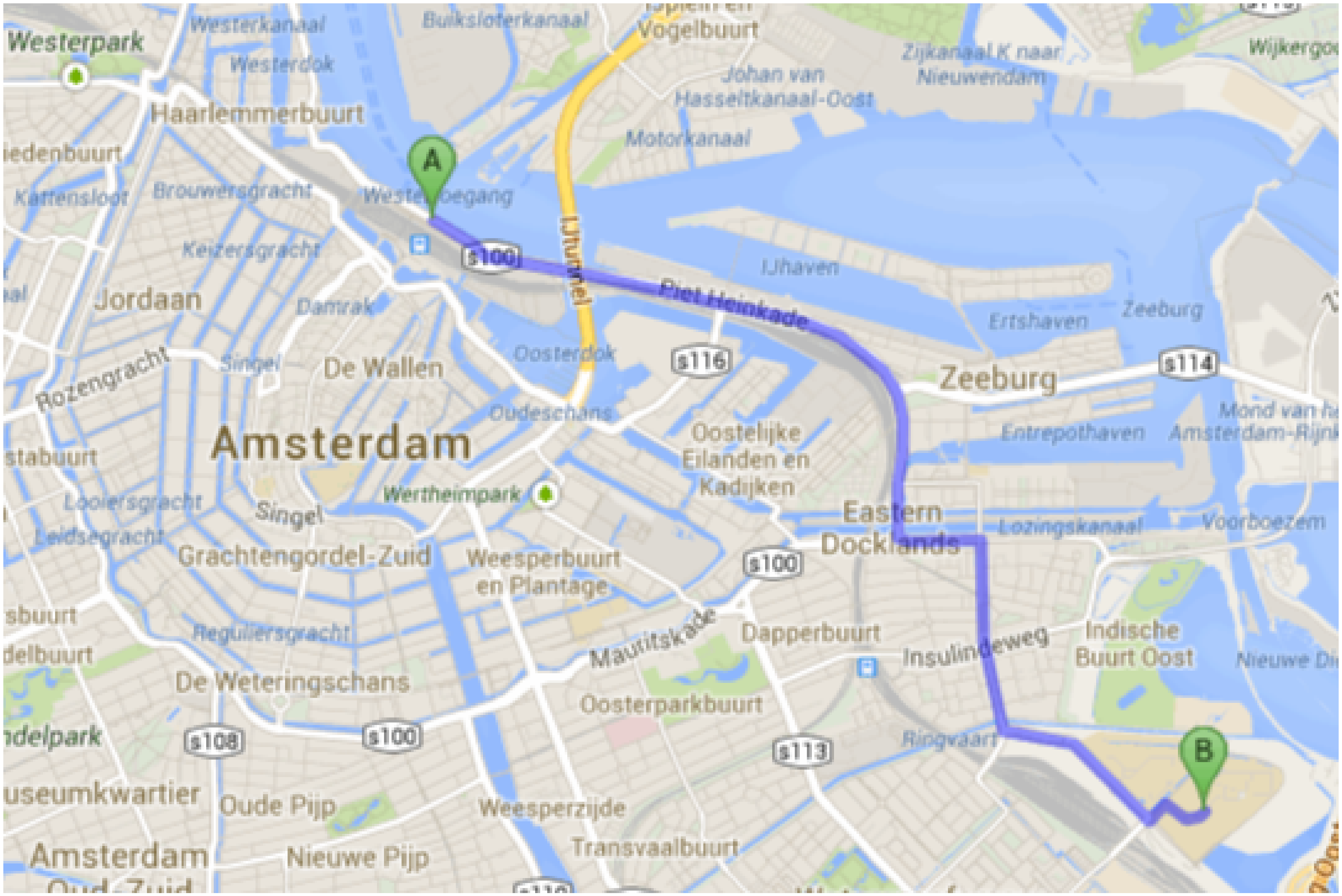
  It guarantees optimality, but it is of no help whatsoever in focusing the search. So using h_0 is *not efficient*.

- The *perfect heuristic function* $h^\star$, 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 $h^\star$ 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.

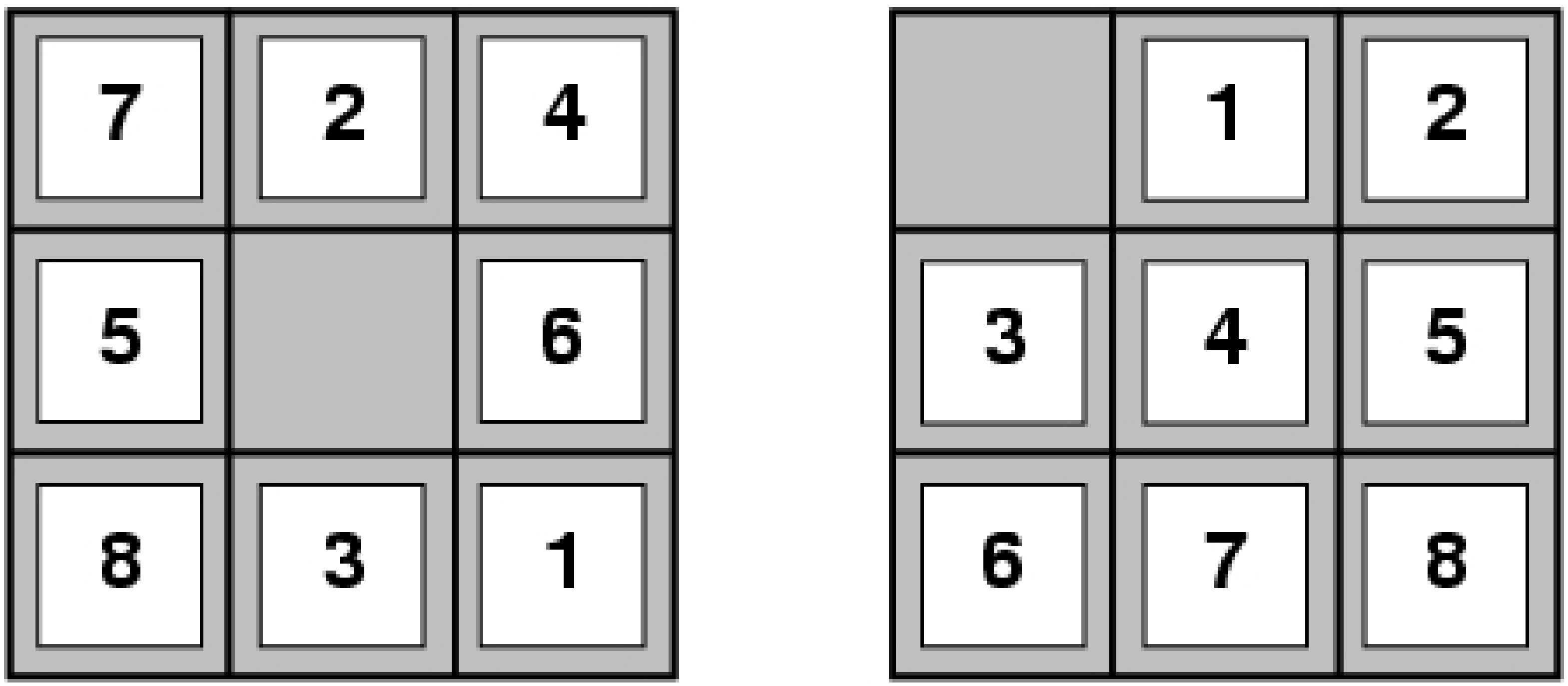
- Let $h_2(x) = 1.2 \cdot h_1(x)$ (adding 20% to the straight-line distance).

  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 $h_3(x) \leqslant h_4(x)$ for all nodes x.

Complexity Analysis of A*

Both *worst-case* time and space complexity are *exponential* in the depth of the search tree (as for breadth-first search): in the worst case, we still have to visit *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 *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).

Lecture 11: Adversarial Search with the Minimax Algorithm

Plan for Today

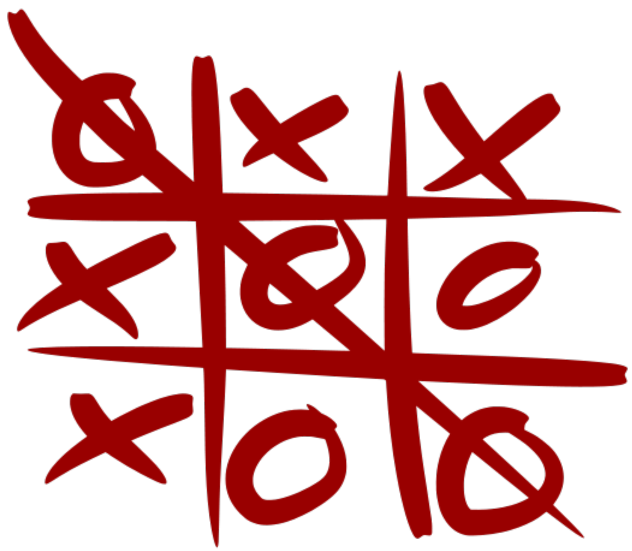
The "games" we have discussed so far (such as the Eight-Puzzle) were not really games but rather puzzles: there is no opponent.

Today we introduce *adversarial search*, where you play a game against an opponent and search the space defined by the moves permitted:

- definition of *two-player perfect-information zero-sum games*
- variant of the *state-space representation* for modelling games and in-depth discussion of applying it to one specific game
- the *minimax algorithm* to search for optimal moves in a game

Tic-Tac-Toe

Also known as *Noughts-and-Crosses* or *Boter-Kaas-en-Eieren* . . .



Many people know how to play this game optimally and—at least when they focus—will never lose a game. *We'll also solve it today.*

Chess

No different from Tic-Tac-Toe in principle: both players have perfect information about the state of the game at every point in time.

<u>But:</u> the search space is much, much bigger than for Tic-Tac-Toe.

Still, the techniques we'll see in this and the next lecture are at the core of all successfull Chess playing programs.

Poker

Yet another popular game that has been tackled by AI researchers . . .

Caveat: Poker (like most other card games) is qualititatively different from perfect-information games such as Tic-Tac-Toe and Chess.

You don't know your opponent's hand, so you cannot simulate in your mind what moves your opponent will consider optimal.

What Games?

We consider games for *two players*. We call them *Max* and *Mindy*.

The players *take turns* making moves on a *board*. We only consider *perfect-information games:* all relevant information is on this board.

Examples: Chess, Go, Tic-Tac-Toe are two-player perfect-information games, but most card games (even those for two players) are not.

Given outcome x, Max gets utility $u^+(x)$ and Min gets utility $u^-(x)$. We only consider *zero-sum games:* $u^+(x) + u^-(x) = 0$ for all x.

Examples: Suppose your utility for a *win* is $+1$, for a *loss* -1, and for a *draw* 0. So most games (e.g., Chess, Go, Tic-Tac-Toe) are zero-sum.

As $u^-(x) = -u^+(x)$, we can model games using just *one* number per outcome. So associate every possible *terminal state* with a *value:*

$$+1 : \text{Max wins} \qquad 0 : \text{draw} \qquad -1 : \text{Min wins}$$

Other numbers are also possible, but we won't use them here.

Representation of Games

We will use a variant of the *state-space representation* to model games.

To specify a game you need to fix the following parameters:

- *States*. The description of a state has two parts:
  - the *board configuration* (representation depends on the game)
  - the name of the *player* to move next (i.e., whose *turn* it is)

  In Prolog: terms of the form `(Board,P)` with `P` $\in \{\texttt{max}, \texttt{min}\}$
- *Moves*. Given the current state, specify possible follow-up states.

  In Prolog: `move(+State, -NextState)`
- *Terminal states* and their *values*. Given the current state, specify whether it is a terminal state and, if so, what its value is.

  In Prolog: `terminal(+State, -Value)` with `Value` $\in [-1, +1]$

To play a full game (rather than just compute the next move), we also need a specification of the *initial state* (in Prolog: `initial(-State)`).

Two Useful Auxiliary Predicates

We represent player Max using atom `max` and player Min using `min`.

The following auxiliary predicates are useful when modelling a variety of different games using the state-space representation for games.

- Given the name of one player, return the name of the *other player:*

```
other(max, min).
other(min, max).
```

- Given the name of a player, return the *value* associated with the terminal states where that player is *winning:*

```
value(max, +1).
value(min, -1).
```

Example: Modelling Tic-Tac-Toe

States are of the form `(Board,Player)` with `Player` $\in$ {`max`, `min`}.

A *board configuration* is a list of three three-element lists, chosen from `o` (Max's symbol) `x`, (Min's symbol), and `-` (empty). Example:

```
Board = [ [x, -, o],
          [-, x, o],
          [-, o, x] ]
```

In the *initial state* the entire board is empty and it is Max's turn:

```
initial((Board,max)) :-
  Board = [ [-,-,-], [-,-,-], [-,-,-] ].
```

Still to do:

- `move/2`: to specify what makes a legal move
- `terminal/2`: to specify terminal states and their values

Another Useful Auxiliary Predicate

Specifically about Tic-Tac-Toe (`other/2` and `value/2` are general).

Associate each player with the symbol they use to make marks:

```
symbol(max, o).
symbol(min, x).
```

Modelling Moves in Tic-Tac-Toe

Suppose we are in state `(Board,Player)`. Then `Player` has to pick a cell on the `Board` that is still empty and replace the `-` found there with his/her symbol (`o` or `x`). And then it is the other player's turn.

In Prolog, all of this can be done with a smart use of `append/3`:

```
move((Board,Player), (NewBoard,OtherPlayer)) :-
  append(TopRows, [Row|BottomRows], Board),
  append(LeftCells, [-|RightCells], Row),
  symbol(Player, Symbol),
  append(LeftCells, [Symbol|RightCells], NewRow),
  append(TopRows, [NewRow|BottomRows], NewBoard),
  other(Player, OtherPlayer).
```

Modelling Terminal States in Tic-Tac-Toe

Suppose we are in state `(Board,Player)`. Note that the identity of the `Player` is irrelevant to determining whether this state is terminal.

There are two kinds of *terminal states:*

- one of the players has completed a "*line*" $\leadsto$ value is $+1$ or -1
- there are *no empty cells* left (and there is no "line") $\leadsto$ value is 0

In Prolog, we can implement this as follows:

```
terminal((Board,_), Value) :-
  symbol(Player, Symbol),
  line(Board, [Symbol,Symbol,Symbol]), !,
  value(Player, Value).

terminal((Board,_), 0) :-
  \+ ( member(Row, Board), member(-, Row) ).
```

Still to do: implement `line/2` to extract "lines" from a given `Board`.

Lines in a Tic-Tac-Toe Board

There are 3 horizontal, 3 vertical, and 2 diagonal “lines” in a board. We need a predicate to get all of them in turn through backtracking. This is a straightforward programming exercise. One solution:

```
line([[A,B,C], [_,_,_], [_,_,_]], [A,B,C]).
line([[_,_,_], [A,B,C], [_,_,_]], [A,B,C]).
line([[_,_,_], [_,_,_], [A,B,C]], [A,B,C]).

line([[A,_,_], [B,_,_], [C,_,_]], [A,B,C]).
line([[_,A,_], [_,B,_], [_,C,_]], [A,B,C]).
line([[_,_,A], [_,_,B], [_,_,C]], [A,B,C]).

line([[A,_,_], [_,B,_], [_,_,C]], [A,B,C]).
line([[_,_,A], [_,B,_], [C,_,_]], [A,B,C]).
```

More elegant solutions that generalise beyond 3×3 grids are possible. Advantage of this solution: fast!

Testing the Line-Extraction Predicate

This works as it should (also for nonstandard "boards" with numbers):

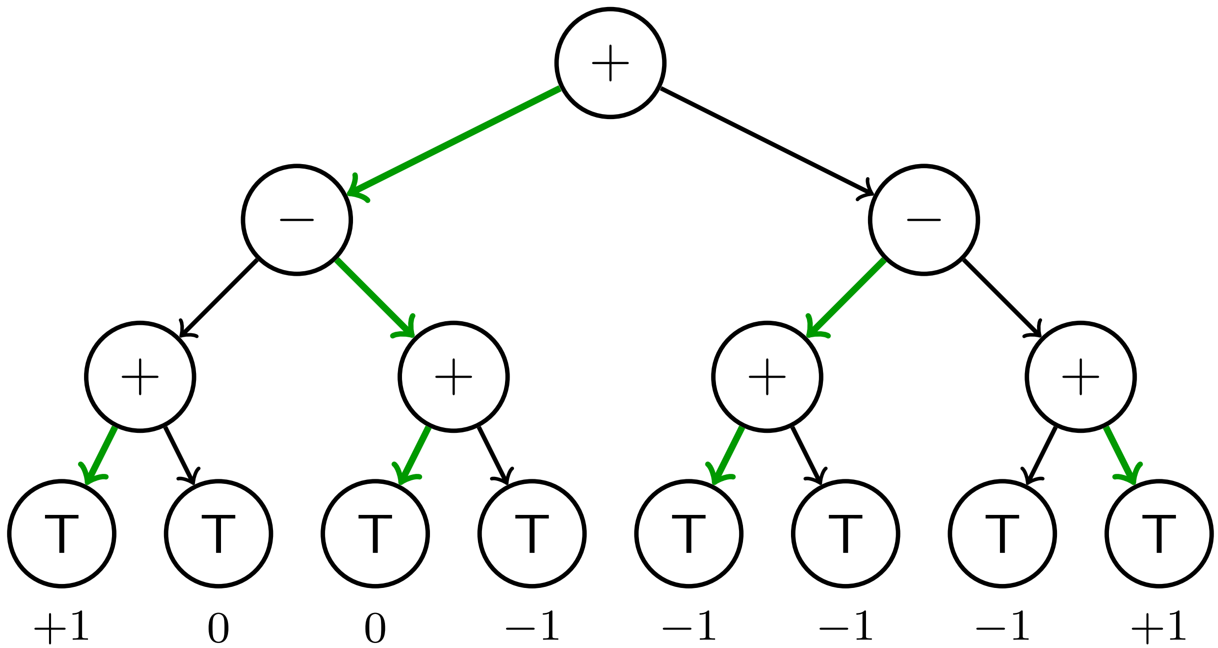
```
?- line([[1,2,3], [4,5,6], [7,8,9]], L).
L = [1, 2, 3] ;
L = [4, 5, 6] ;
L = [7, 8, 9] ;
L = [1, 4, 7] ;
L = [2, 5, 8] ;
L = [3, 6, 9] ;
L = [1, 5, 9] ;
L = [3, 5, 7] ;
No
```

```
?- line([[x,-,o], [-,x,o], [-,o,x]], L).
L = [x, -, o] ;
L = [-, x, o] ;
L = [-, o, x] ;
L = [x, -, -] ;
L = [-, x, o] ;
L = [o, o, x] ;
L = [x, x, x] ;
L = [o, x, -] ;
No
```

Finding an Optimal Move to Play

Now step back and consider the problem of finding an *optimal move* to play in a *given state* for *any* game (not just Tic-Tac-Toe).

Consider the tree of all possible plays from the current state. Example:



Can reason backwards (recursively!) to find *optimal move* in any state.

The Minimax Algorithm

Given a *game tree*, the root of which is the current state and the leaves of which are the terminal states, the *minimax algorithm* uses what is called *backward induction* (in other words: recursion) to compute a value for every state on this game tree:

- If x is a *terminal state*, then the value of x is defined by the game.
- If x is a state in which it is *Max's turn* to move, then the value of x is the *maximum* of the values of the children of x.
- If x is a state in which it is *Min's turn* to move, then the value of x is the *minimum* of the values of the children of x.

Once we have evaluated all states on the game tree in this manner, we immediately obtain *optimal strategies* for both players:

- Max should always pick a child with maximal value.
- Min should always pick a child with minimal value.

The Finiteness Assumption

The minimax algorithm only works when the game tree is *finite:* for an infinite branch you would not be able to "start at the bottom".

But is it ok to assume that all branches have finite length?

- Some (simple) games clearly satisfy this finiteness assumption.

  Example: In *Tic-Tac-Toe* we only *add* (but never *remove*) marks, so any game must end after at most 9 moves.

- Some games allow for loops and thus fail this assumption.

  Example: A game of *Chess* can continue forever.

  But for such (complex) games the real challenge is a different one. Even if repetitions were not allowed in Chess, the sheer size of the search space would still be too much for (plain) minimax and we will need additional techniques (discussed in the next lecture).

So in practice our finiteness assumption does not restrict us a lot.

The Minimax Algorithm in Prolog

At the core of our implementation will be the following predicate:

- `eval(+State, -Value)`

  Given a `State`, compute and return its `Value`.

We will implement it with the help of two further predicates:

- `maxeval(+States, -State, -Value)`

  Given a list of `States`, compute their values and return a `State` with maximal value amongst them, together with its `Value`.

- `mineval(+States, -State, -Value)`

  Given a list of `States`, compute their values and return a `State` with minimal value amongst them, together with its `Value`.

Remark: To *evaluate* states we only need to know the values returned, while to *recommend* a move we only need to know the states returned.

Auxiliary Predicate: Collect All Next States

Recall: the programmer modelling the specific game we want to play has to provide `move/2` to compute possible follow-up states.

This predicate can then be used to collect *all follow-up states* in a list:

```
moves(State, NextStates) :-
  findall(NextState, move(State, NextState), NextStates).
```

Evaluation of States

If the given state is a *terminal state*, just look up its value (base case):

```
eval(State, Value) :-
  terminal(State, Value), !.
```

Otherwise, first compute the *list of follow-up states* for the given state.

Then, if it is *Max's turn*, select the *max-value state* amongst them and return the *value of that state* (*not* the state itself: anonymous variable).

```
eval((Board,max), Value) :-
  moves((Board,max), NextStates),
  maxeval(NextStates, _, Value).
```

Proceed analogously in case it is *Min's turn:*

```
eval((Board,min), Value) :-
  moves((Board,min), NextStates),
  mineval(NextStates, _, Value).
```

Auxiliary Predicate: Making a Choice

Purely technical programming device, to be used on the next slide . . .

Given a *condition* and two terms `X` and `Y`, if the condition holds, then bind the *final argument* to `X`, otherwise bind it to `Y`:

```
choose(Condition, X, _, X) :- call(Condition), !.
choose(_, _, Y, Y).
```

Examples:

```
?- choose(42 > 0, alpha, beta, Result).
Result = alpha
Yes

?- choose(42 < 0, alpha, beta, Result).
Result = beta
Yes
```

Selecting a State with a Maximal Value

Given a list of states, want to find the max-value state and its value.

If there is just one input state, return it and its value (base case):

```
maxeval([State], State, Value) :- !,
  eval(State, Value).
```

Otherwise, compute the value of the first state as well as the max-value state and *its* value for the tail of the list (recursion) and choose the better state/value pair of the two:

```
maxeval([State1|States], MaxState, MaxValue) :-
  eval(State1, Value1),
  maxeval(States, State, Value),
  choose(Value > Value1,
         (State,Value),
         (State1,Value1),
         (MaxState,MaxValue)).
```

Note: If `Value =:= Value1`, `State1/Value1` is returned.

Selecting a State with a Minimal Value

Picking a state with minimal value from a given list works analogously:

```
mineval([State], State, Value) :- !,
  eval(State, Value).

mineval([State1|States], MinState, MinValue) :-
  eval(State1, Value1),
  mineval(States, State, Value),
  choose(Value < Value1,
         (State,Value),
         (State1,Value1),
         (MinState,MinValue)).
```

Recommending a Best Move via Minimax

It is now straightforward to write a predicate that, for any given state of the form (Board,Player), will return the best follow-up state the Player whose turn it is can possibly move to:

```
minimax((Board,max), MaxState) :-
  moves((Board,max), NextStates),
  maxeval(NextStates, MaxState, _).

minimax((Board,min), MinState) :-
  moves((Board,min), NextStates),
  mineval(NextStates, MinState, _).
```

The value of the state returned is irrelevant (anonymous variable).

Testing

It works! If you use `minimax/2` to compute your moves in a game of Tic-Tac-Toe, you will never lose, whatever your opponent may do.

Here minimax recommends to Min (playing `x`) a real killer move:

```
?- minimax( ([[x,o,o], [-,x,o], [-,-,-]],min), (Board,_) ).
Board =      [[x,o,o], [-,x,o], [-,-,x]]
Yes
```

But it's slow! The most expensive move to compute is the first one:

```
?- initial(State), time( minimax(State, NextState) ).
% 19,170,066 inferences, 6.519 CPU in 6.524 seconds [...]
State     = ([[-,-,-], [-,-,-], [-,-,-]], max),
NextState = ([[o,-,-], [-,-,-], [-,-,-]], min)
Yes
```

We will address this efficency issue in the next lecture.

Discussion: Optimal = Optimal ?

When minimax recommends a move, then that move is optimal under the assumption that your opponent will also play optimally.

But if your opponent does not always play optimally, then that may not be the best strategy. For example, there may be no strategy that guarantees you a win (only a draw), but along one branch chances are higher your opponent will overlook a good move, allowing you to win. Minimax cannot recognise such opportunities.

Summary: Adversarial Search with Minimax

We have introduced the topic of *adversarial search* to compute optimal moves in *two-player perfect-information zero-sum games:*

- *state-space representation* to model any such game
  - describe states of the form `(Board,Player)`
  - define legal moves using `move/2`
  - define values of terminal states using `terminal/2`
- *minimax algorithm* to evaluate states (so you can pick the best)
  - compute values for nodes on the game tree recursively, starting with leaves (terminal states) and simulating what each player (Max and Min) would do when it is their turn

In theory this approach can solve games such as Chess and Go, but in practice it won't scale much beyond small games such as Tic-Tac-Toe.

Lecture 12: Alpha-Beta Pruning and Heuristic Evaluation

Plan for Today

We have seen that the *minimax algorithm* can be used to solve (= compute optimal moves for) any two-player perfect-information zero-sum game in principle, but that in practice it will fail to do so for somewhat larger games, due to the *explosion of the search space*.

Today is about techniques for *speeding up* the minimax algorithm:

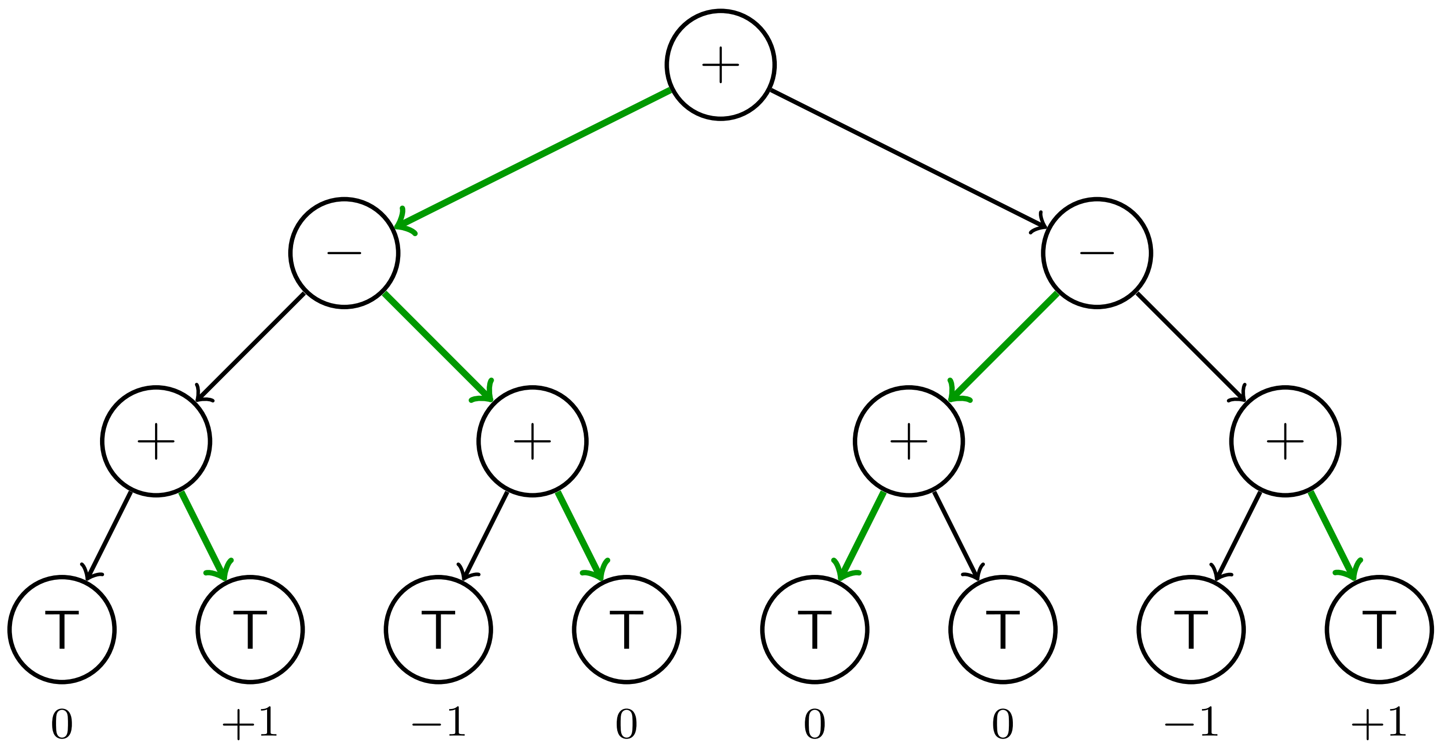
- *Alpha-beta pruning:* smart exploration of the search space that avoids evaluating moves that have no chance of being optimal
- *Heuristic evaluation:* making informed guesses for the values of certain states, rather than computing them exactly using minimax

We will conclude with a brief review of famous examples for the use of adversarial search techniques in AI (to tackle Chess, Go, and Poker).

Reminder: Minimax Algorithm

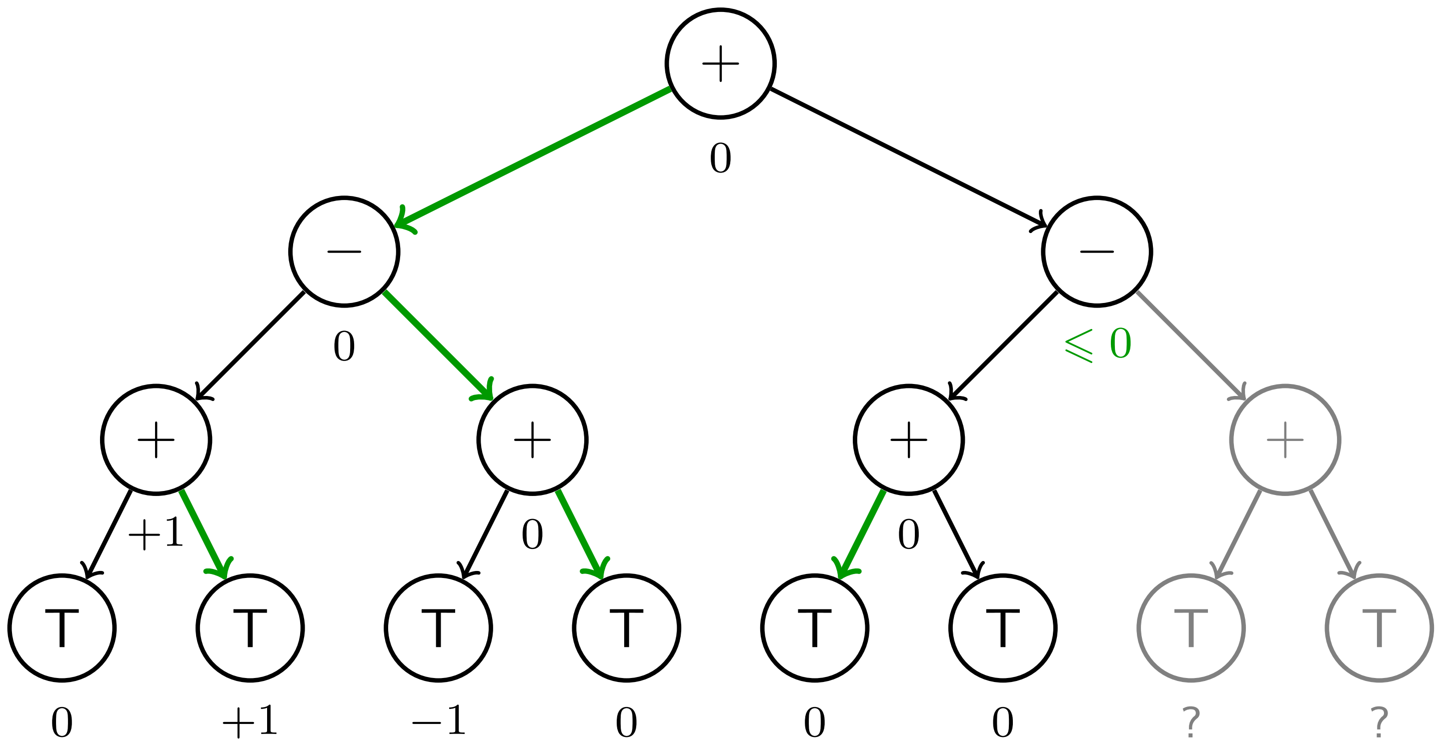
Recall how using the *minimax algorithm* we compute the value of each state by starting from the terminal states (for which values are given).

- Max ($+$) states: take the maximum of the values of the children.
- Min ($-$) states: take the minimum of the values of the children.



Pruning Irrelevant Parts of the Tree

Sometimes you do not need to inspect the full tree to evaluate the root of the game tree. Max knows that if he moves right, then Min can force at least a draw ($\leqslant 0$). As Max can get a draw also by moving left, there is no need to further explore the rightmost part of the tree.



Minimax with Alpha-Beta Pruning

Suppose you want to evaluate state x. Its (unknown) value is $v_x^\star \in [-1, +1]$. Suppose you are satisfied if you can compute $v_x \in [-1, +1]$ such that these conditions hold (for some numbers $-1 \leqslant \alpha_x \leqslant \beta_x \leqslant +1$):

- $v_x = v_x^\star$ in case $\alpha_x \leqslant v_x^\star \leqslant \beta_x$ (so: precise within critical interval)
- $v_x = \alpha_x$ in case $v_x^\star < \alpha_x$ (so: low default for very low values)
- $v_x = \beta_x$ in case $v_x^\star > \beta_x$ (so: high default for very high values)

Algorithm to compute v_x (with $v_x = v_x^\star$ for $\alpha_x = -1$ and $\beta_x = +1$):

- If x is *terminal*, look up $v_x^\star$ and set v_x to the *closest point* in $[\alpha_x, \beta_x]$.
- If x has *children* $y_1, \ldots, y_k$ and it is *Max's turn*, first compute v_{y_1} with $\alpha_{y_1} = \alpha_x$ and $\beta_{y_1} = \beta_x$. Then v_{y_2} with $\alpha_{y_2} = v_{y_1}$ and $\beta_{y_2} = \beta_x$. Then v_{y_3} with $\alpha_{y_3} = \max\{v_{y_1}, v_{y_2}\}$ and $\beta_{y_2} = \beta_x$. And so forth. Finally, set $v_x = \max\{v_{y_1}, \ldots, v_{y_k}\}$.
- If x has *children* $y_1, \ldots, y_k$ and it is *Min's turn*, proceed accordingly (now always *updating* β) and set $v_x = \min\{v_{y_1}, \ldots, v_{y_k}\}$.

Note: for $\alpha_x = \beta_x$ you immediately get $v_x = \alpha_x = \beta_x$ (*no computation!*).

Auxiliary Predicate: Rounding

Given numbers X, A, and B with $A \leqslant B$, find the number closest to X within the interval $[A, B]$:

```
round(X, A, _, A) :- X < A, !.
round(X, _, B, B) :- X > B, !.
round(X, _, _, X).
```

Plan for Implementation in Prolog

Same basic setup as before, but now with two extra input arguments:

- `eval(+State, +Alpha, +Beta, -Value)`

  Given a `State` (with initially unknown exact value $v^\star$) and two numbers `Alpha` and `Beta`, return a number `Value` such that:

  - `Value` $=$ $v^\star$ in case `Alpha` $\leqslant v^\star \leqslant$ `Beta`
  - `Value` $=$ `Alpha` in case $v^\star <$ `Alpha`
  - `Value` $=$ `Beta` in case $v^\star >$ `Beta`

- `maxeval(+States, +Alpha, +Beta, -State, -Value)`

  Given a list of `States` and numbers `Alpha` and `Beta`, return a `State` (in `States`) with maximal value and a number `Value` that satisfies the same condition as above w.r.t. the value $v^\star$ of `State`.

- `mineval(+States, +Alpha, +Beta, -State, -Value)`

  As above, but now with `State` having minimal value.

Evaluation of States

If `Alpha` = `Beta`, then this must also be the exact value (base case):

```
eval(_, Value, Value, Value) :- !.
```

If we are in a *terminal state*, look up its value and round (base case):

```
eval(State, Alpha, Beta, RoundedValue) :-
  terminal(State, Value), !,
  round(Value, Alpha, Beta, RoundedValue).
```

Otherwise, from the list of *follow-up states* return the (approximate) value of the state chosen by the player whose turn it is (recursion):

```
eval((Board,max), Alpha, Beta, Value) :-
  moves((Board,max), NextStates),
  maxeval(NextStates, Alpha, Beta, _, Value).

eval((Board,min), Alpha, Beta, Value) :-
  moves((Board,min), NextStates),
  mineval(NextStates, Alpha, Beta, _, Value).
```

Selecting a State with a Maximal Value

Getting the max-value state from a list of states together with its value (or an approximation thereof, in case it falls outside the α-β interval).

If the list has just one element, relegate to `eval/4` (base case):

```
maxeval([State], Alpha, Beta, State, Value) :- !,
  eval(State, Alpha, Beta, Value).
```

Else: evaluate head and tail (recursion). Return better state/value. But use *first value* found as *new lower bound* during recursion:

```
maxeval([State1|States], Alpha, Beta, MaxState, MaxValue) :-
  eval(State1, Alpha, Beta, Value1),
  maxeval(States, Value1, Beta, State, Value),
  choose(Value > Value1, (State,Value), (State1,Value1),
         (MaxState,MaxValue)).
```

Selecting a State with a Minimal Value

Implementing `mineval/5` is essentially the same as for `maxeval/5`:

```
mineval([State], Alpha, Beta, State, Value) :- !,
  eval(State, Alpha, Beta, Value).

mineval([State1|States], Alpha, Beta, MinState, MinValue) :-
  eval(State1, Alpha, Beta, Value1),
  mineval(States, Alpha, Value1, State, Value),
  choose(Value < Value1, (State,Value), (State1,Value1),
         (MinState,MinValue)).
```

But note that this time we use the *first value* found (for the head of the list of states) as a *new upper bound* during the recursive call.

Reason: Min knows she can achieve `Value1`, so when searching through `States`, she is only interested in states with even lower value.

Recommending a Best Move

Now the predicate `alphabeta/2` can be used to recommend a best move for a given board configuration for a given player:

```
alphabeta((Board,max), MaxState) :-
  moves((Board,max), NextStates),
  maxeval(NextStates, -1, +1, MaxState, _).

alphabeta((Board,min), MinState) :-
  moves((Board,min), NextStates),
  mineval(NextStates, -1, +1, MinState, _).
```

<u>Remark:</u> Observe how we initialise `Alpha` with -1 and `Beta` with $+1$. This is ok, as (for us) no terminal state has a value outside of $[-1, +1]$. For more general games, you have to initialise with $-\infty$ and $+\infty$.

Testing

The *basic minimax algorithm* was pretty slow for Tic-Tac-Toe:

```
?- initial(State), time( minimax(State, NextState) ).
% 19,170,066 inferences, 6.519 CPU in 6.524 seconds [...]
State     = ([[-,-,-], [-,-,-], [-,-,-]], max),
NextState = ([[o,-,-], [-,-,-], [-,-,-]], min)
Yes
```

If we use *minimax with alpha-beta pruning* instead, we are more than *30 times faster* when computing Max's first move:

```
?- initial(State), time( alphabeta(State, NextState) ).
% 732,054 inferences, 0.198 CPU in 0.199 seconds [...]
State     = ([[-,-,-], [-,-,-], [-,-,-]], max),
NextState = ([[o,-,-], [-,-,-], [-,-,-]], min)
Yes
```

Great! But still not enough for larger games, such as Chess or Go.

Using Heuristics to Evaluate States

Idea: When you do not manage to inspect all of the game tree to evaluate states, *estimate values* (for some of them) instead!

When do you stop searching and start estimating instead?

Here are some possible approaches (we will only explore the first):

- once you have reached a certain *depth* in the tree
- once you have spent a certain amount of *time* searching
- whenever you encounter a state you feel *confident* evaluating

How do you actually estimate the value of a given state?

- very much *depends* on the game you are playing

Example: A Heuristic for Tic-Tac-Toe

Recall that there are *8 "lines"* on the board. Consider a give state:

- Let A be the number of lines that Max could still complete.
- Let B be the number of lines that Min could still complete.
- Estimate the *value* of the state as $(A - B)\,/\,8$.

This can be implemented in Prolog as follows:

```
estimate((Board,_), Value) :-
  symbol(max, MaxSym),
  symbol(min, MinSym),
  findall(L, (line(Board,L), \+ member(MinSym,L)), MaxLines),
  findall(L, (line(Board,L), \+ member(MaxSym,L)), MinLines),
  length(MaxLines, MaxChances),
  length(MinLines, MinChances),
  Value is (MaxChances - MinChances) / 8.
```

Caveat: Not a great heuristic. But useful for experimentation.

Heuristic Minimax: Implementation in Prolog

Suppose a heuristic evaluation function is provided by `estimate/2`.

Next: an implementation of the minimax algorithm with alpha-beta pruning that stops search at a given depth and instead consults this heuristic evaluation function.

The code is almost exactly the same as before. Two refinements:

- We need to carry along an extra variable `Depth` everywhere and decrement it by 1 in every recursive step (as we go down the tree).
- We need to add a *third base case* to our basic evaluation predicate (now `eval/5`) that calls `estimate/2` once `Depth` is 0.

Evaluation of States

```
eval(_, _, Value, Value, Value) :- !.

eval(State, _, Alpha, Beta, RoundedValue) :-
  terminal(State, Value), !,
  round(Value, Alpha, Beta, RoundedValue).

eval(State, 0, Alpha, Beta, RoundedValue) :- !, % new base case
  estimate(State, Value),
  round(Value, Alpha, Beta, RoundedValue).

eval((Board,max), Depth, Alpha, Beta, Value) :-
  moves((Board,max), NextStates),
  NewDepth is Depth - 1,            % decementing depth counter
  maxeval(NextStates, NewDepth, Alpha, Beta, _, Value).

eval((Board,min), Depth, Alpha, Beta, Value) :-
  moves((Board,min), NextStates),
  NewDepth is Depth - 1,            % decrementing depth counter
  mineval(NextStates, NewDepth, Alpha, Beta, _, Value).
```

Selecting a State with a Maximal Value

Note: The only change in this part of the code is that we need to carry along the additional argument `Depth`.

```
maxeval([State], Depth, Alpha, Beta, State, Value) :- !,
  eval(State, Depth, Alpha, Beta, Value).

maxeval([State1|States], Depth, Alpha, Beta, MaxState, MaxValue) :-
  eval(State1, Depth, Alpha, Beta, Value1),
  maxeval(States, Depth, Value1, Beta, State, Value),
  choose(Value > Value1, (State,Value), (State1,Value1),
         (MaxState,MaxValue)).
```

Selecting a State with a Minimal Value

Note: The only change in this part of the code is that we need to carry along the additional argument `Depth`.

```
mineval([State], Depth, Alpha, Beta, State, Value) :- !,
  eval(State, Depth, Alpha, Beta, Value).

mineval([State1|States], Depth, Alpha, Beta, MinState, MinValue) :-
  eval(State1, Depth, Alpha, Beta, Value1),
  mineval(States, Depth, Alpha, Value1, State, Value),
  choose(Value < Value1, (State,Value), (State1,Value1),
         (MinState,MinValue)).
```

Recommending a Best Move

Also for the predicate to recommend a move in a given state, all we need to change is to include an argument to allow the user to specify the `Depth` at which she wants to switch to heuristic evaluation:

```
alphabeta((Board,max), Depth, MaxState) :-
  moves((Board,max), NextStates),
  maxeval(NextStates, Depth, -1, +1, MaxState, _).

alphabeta((Board,min), Depth, MinState) :-
  moves((Board,min), NextStates),
  mineval(NextStates, Depth, -1, +1, MinState, _).
```

Testing

Suppose you are Min (playing `x`) and it is your turn:

```
 o | 2 | 3
---+---+---
 4 | x | 6
---+---+---
 7 | 8 | o
```

You can force a draw by marking 2, 4, 6, or 8.

The other two moves are bad!

For a *high bound* (so: don't use heuristic!), minimax gets it right:

```
?- time( alphabeta( ([[o,-,-],[-,x,-],[-,-,o]],min), 100, S) ).
% 11,462 inferences, 0.007 CPU in 0.007 seconds [...]
S = ([[o, x, -], [-, x, -], [-, -, o]], max)
```

For *very low bounds*, we get bad recommendations (but it's faster!):

```
?- time( alphabeta( ([[o,-,-],[-,x,-],[-,-,o]],min), 1, S) ).
% 2,064 inferences, 0.001 CPU in 0.001 seconds [...]
S = ([[o, -, x], [-, x, -], [-, -, o]], max)
?- time( alphabeta( ([[o,-,-],[-,x,-],[-,-,o]],min), 0, S) ).
% 727 inferences, 0.000 CPU in 0.000 seconds [...]
S = ([[o, -, x], [-, x, -], [-, -, o]], max)
```

Monte Carlo Search

Designing heuristics is hard! *Monte Carlo search* is an approach to *automatically generate* heuristic evaluation functions.

Here's a sketch of the basic idea. To estimate the value of *state x:*

- *Simulate 100 game continuations* ("rollouts") from state x, assuming that both players always choose their moves *at random*.
- Let A be the number of simulations in which Max wins.
- Let B be the number of simulations in which Min wins.
- Estimate the value of state x as $(A - B) / 100$.

In other words: estimate the value of state x as the *average value* of the terminal states reached in, say, 100 *random rollouts* from x.

Complexity Analysis

The minimax algorithm (with or w/o any of the refinements discussed) is essentially a *depth-first* search algorithm. Thus:

- *Space comlexity* is very good: *linear* in the depth explored.
- *Time complexity* is problematic: *exponential* in the depth explored. Alpha-beta pruning great in practice, yet exponential in worst case.

Above analysis concerns the search component only. Need to multiply this with complexity of computing the *heuristic evaluation function*. Idea here is that heuristics should be *extremely fast* to compute.

Playing Chess: Deep Blue (1997)

In May 1997 IBM's *Deep Blue* beat world champion Garry Kasparov under standard tournament conditions (3.5 to 2.5).

Deep Blue was based on *minimax* with *alpha-beta pruning*, as well as:

- *specialised hardware* to run minimax on, heavily *parallelised*
- sophisticated *heuristic evaluation function* using *domain knoweldge*
  - function depending on ∼8,000 features of board configurations
  - partly handcrafted with help of Chess grandmaster
  - partly based on analysis of database of grandmaster games

Playing Go: AlphaGo (2016)

Go has an even larger search space than Chess.

In March 2016 Google DeepMind's *AlphaGo* beat Lee Sedol, one of the world's best players, under standard tournament conditions (4 to 1).

AlphaGo was based on *minimax* with *alpha-beta pruning*, as well as:

- *Monte Carlo* approach to generate heuristic evaluation function, but with higher probabilities for more promising moves
- machine learning with *deep neural networks* to learn a policy for selecting those promising moves from expert plays
- *reinforcement learning:* refinements via simulated self-play

Extremely strong heuristics: even zero-depth minimax is competitive.

Remark: Neither Go nor Chess are "solved". We do not yet know how to compute the best move in every situation. These programs "only" are good enough to beat the best human players most of the time.

Playing Poker: Libratus (2017)

Poker is a zero-sum game and it can be restricted to two players. But is is *not a perfect-information game*.

In January 2017 Carnegie Mellon University's *Libratus* beat four top human players at a 20-day contest at the Rivers Casino in Pittsburgh.

Requires more *sophisticated search and optimisation techniques* than covered in this course, as well as insights from *game theory*.

Summary: Alpha-Beta Pruning and Heuristics

We have discussed two refinements of the *minimax algorithm* for computing the optimal move in a game:

- *Alpha-beta pruning:* smart propagation of bounds on possible values of states, to cut out parts of the game tree to be searched.
- *Heuristic evaluation* of states to terminate search early, including *Monte Carlo* techniques to automatically generate heuristics.

These techniques form the basis for building competitive game-playing systems. Examples include some of the best known milestones in AI.