Artificial Intelligence I

Dr Mateja Jamnik

Computer Laboratory, Room FC18

Telephone extension 63587

Email: mj201@cl.cam.ac.uk

http://www.cl.cam.ac.uk/users/mj201/

Notes II: problem solving by search, games (adversarial search), and constraint satisfaction problems.

Copyright ©Sean Holden 2002-2009.

Solving problems by search I: uninformed search

We now look at how an agent might achieve its goals using **search**.

Aims:

- to show how problem-solving can be modelled as the process of **searching** for a **sequence of actions** that **achieves a goal**;
- to introduce some basic algorithms for conducting the necessary search for a sequence of actions.

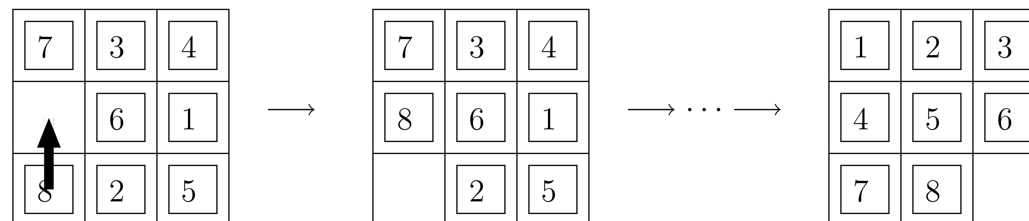
Reading: Russell and Norvig, chapter 3.

Problem solving by basic search

As usual: an *agent* exists within an *environment* and must *act* within this environment to achieve some desirable *goal*.

It has some means of knowing the *state* of its environment.

A simple example: the 8-puzzle.



Problem solving by basic search

Start state: a randomly-selected configuration of the numbers 1 to 8 arranged on a 3×3 square grid, with one square empty.

Goal state: the numbers in ascending order with the bottom right square empty.

Actions: `left`, `right`, `up`, `down`. We can move any square adjacent to the empty square into the empty square. (It's not always possible to choose from all four actions.)

Path cost: one per move.

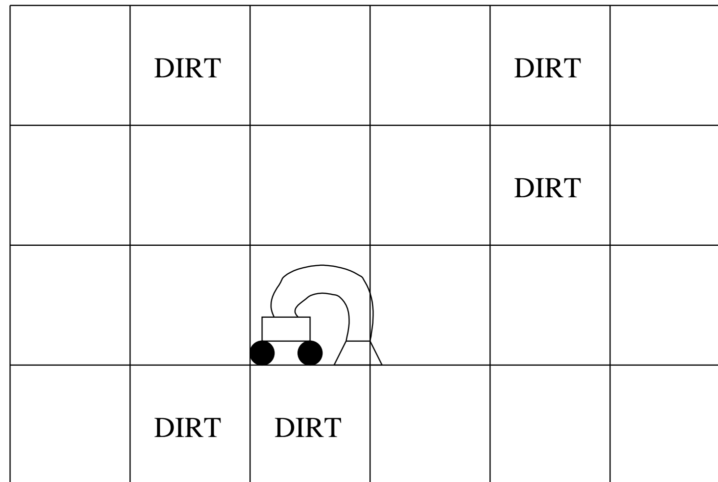
The 8-puzzle is very simple. However general sliding block puzzles are a good test case. The general problem is NP-complete. The 5×5 version has about 10^{25} states, and a random instance is in fact quite a challenge!

Another example:

- the agent is a robotic vacuum cleaner;
- the environment is a rectangular room with no obstacles, containing the cleaner and some dirt;
- the available actions are movement in four directions, switch on sucker, and switch off sucker;
- the cleaner can sense the presence or otherwise of dirt and knows it's own location;
- the goal is to have no dirt in the room.

Problem solving by basic search

The situation looks something like this:



Even this simple description hides a number of ambiguities and subtleties.

Problem solving by basic search

The examples given admit a simple solution strategy that is applicable to many simple problems in AI.

Initial state: the cleaner is at some position within the room and there is dirt in various locations.

Actions: the cleaner can alter the state of the environment by acting. In this case either by moving or using its sucker. By performing a sequence of actions it can move from state to state.

Aim: the cleaner wants to find a sequence of actions that achieves the goal state of having a dirt-free room.

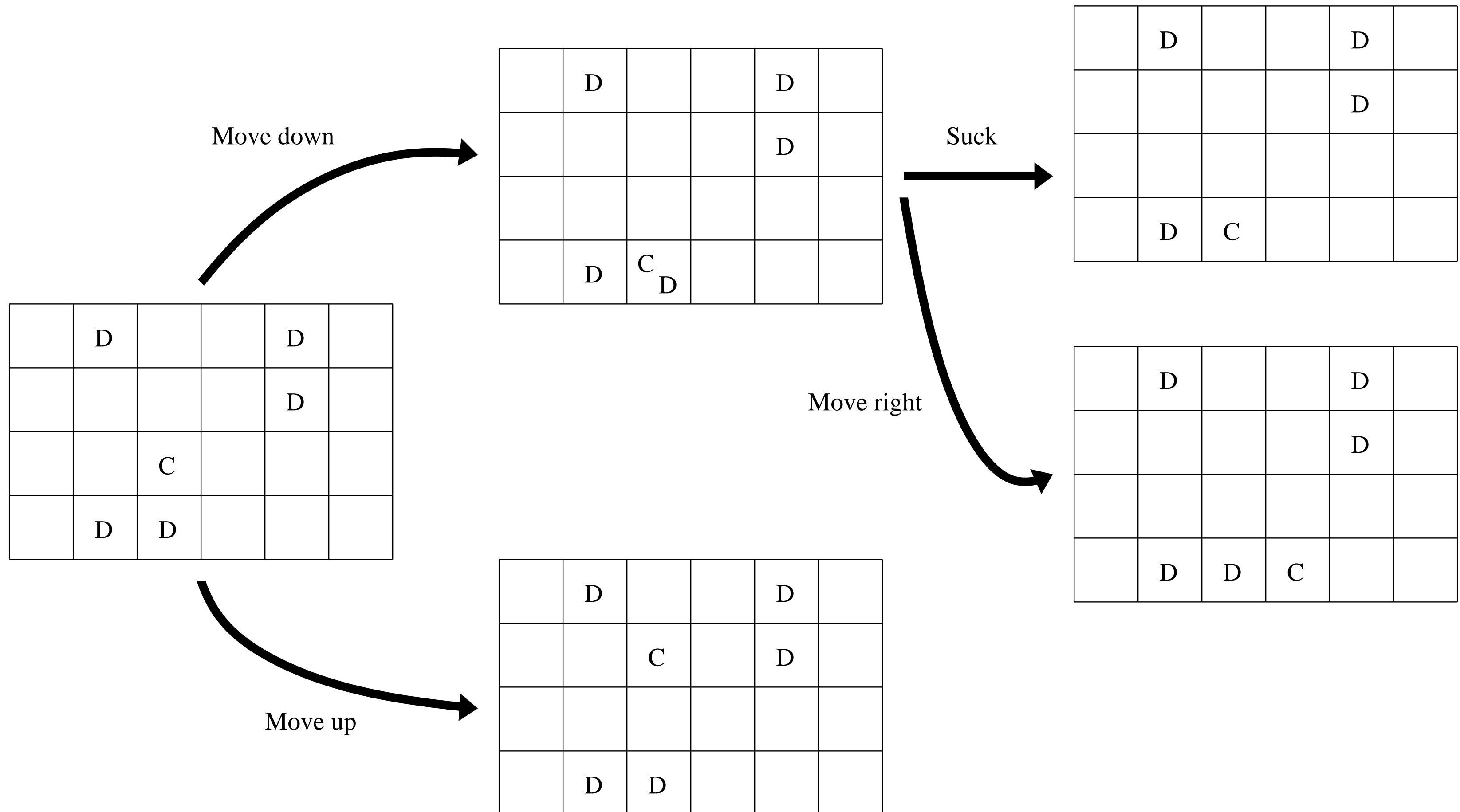
Problem solving by basic search

Other applications that can be addressed:

- route-finding
- tour-finding
- layout of VLSI systems
- navigation systems for robots
- sequencing for automatic assembly
- searching the internet
- design of proteins

and many others...

Problem solving by basic search



Problem solving by basic search

So what's ambiguous and subtle here?

1. Can the agent know it's current state in full?
   - It may only be able to sense dirt within a given radius.
   - It may not have a completely accurate position sensor.
   - It may not be able to distinguish between dirt and a stain on the carpet, and so on...
2. Can the agent know the outcome of its actions in full?
   - The sucker may not be completely reliable.
   - The sucker may occasionally deposit a little dirt.
   - The next door neighbour's child may sneak in and move it from one place to another while it thinks it's only moved a short way in one direction, and so on...

Depending on the answers to these questions we can identify four basic kinds of problem:

Single-state problems: the state is always known precisely, as is the effect of any action. There is therefore a single outcome state.

Multiple-state problems: The effect of any action is known, but the current state can not reliably be inferred. Hence we must reason about the set of states that we could be in. A similar situation arises if we know the current state but not necessarily the outcomes of the actions.

Single and multiple state problems can be handled using the search techniques to be discussed next.

Contingency problems:

In some situations it is necessary to perform sensing *while* the actions are being carried out in order to guarantee reaching a goal.

(It's good to keep your eyes open while you cross the road!)

This kind of problem requires **planning** and **acting**.

Sometimes it is actively beneficial to act and see what happens, rather than to try to consider all possibilities in advance in order to obtain a perfect plan.

Exploration problems:

Sometimes you have *no* knowledge of the effect that your actions have on the environment.

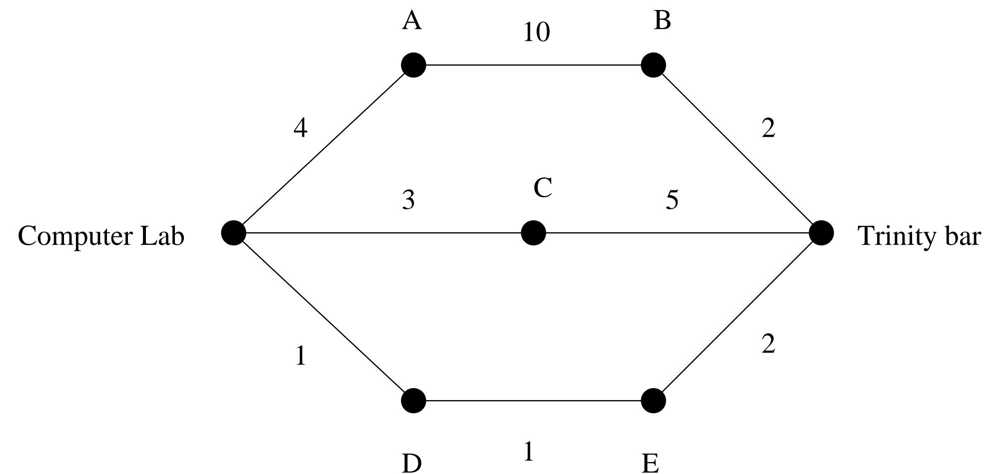
Babies in particular have this experience.

This means you need to experiment to find out what happens when you act.

This kind of problem requires **reinforcement learning** for a solution. We will not cover reinforcement learning in this course. (Although it is in AI II!)

Problem solving by basic search

Question: How much detail should the state description include?



To use a different example: considering your lecturer as an intelligent agent who wants to get from the Computer Lab to Trinity College before the bar closes.

Problem solving by basic search

The state of my environment could be said to include:

- the number and temperature of each hair on my head;
- the composition of the roads on all the potential routes;
- the current position of Saturn *etc*.

However for this problem a much simpler state description seems appropriate: “at the computer lab”, “in the computer lab bike stands” and so on.

Similarly for potential actions: although “remove wax from ears” is a perfectly valid action in state “at the computer lab” it’s clearly not very helpful.

Problem solving by basic search

Question: Are there conflicting goals or goals of varying importance?

Apart from getting to the College bar I might want to stop by the book signing at Waterstones, or drop by the language school to improve my Italian.

However we need to identify one specific goal.

Always in computer science, we need to do some **abstraction** to make a solution feasible—we need to remove all extraneous detail.

Note that in this example, if I have no internal map of Cambridge town centre I am stuck - I am doomed to try random actions. However if I have such a map I can try to **search** for a sequence of actions that achieves my goal.

Problem solving by basic search

We begin with (arguably) the simplest kind of scenario in which some form of computationally intelligent behaviour can be achieved. Namely, the single-state scenario.

To summarise, we have:

- **an initial state**: what is the agent's situation to start with;
- **a set of actions**: and we know what state will result on performing any available action from any known state;
- **a goal test**: we can tell whether or not the state we're in corresponds to the goal.

Note that the goal may be described by a property rather than an explicit state or set of states, for example ”checkmate”.

In addition, a **path** is a sequence of actions that lead from state to state.

We may also be interested in the **path cost** as some solutions might be better than others. Path cost will be denoted by p.

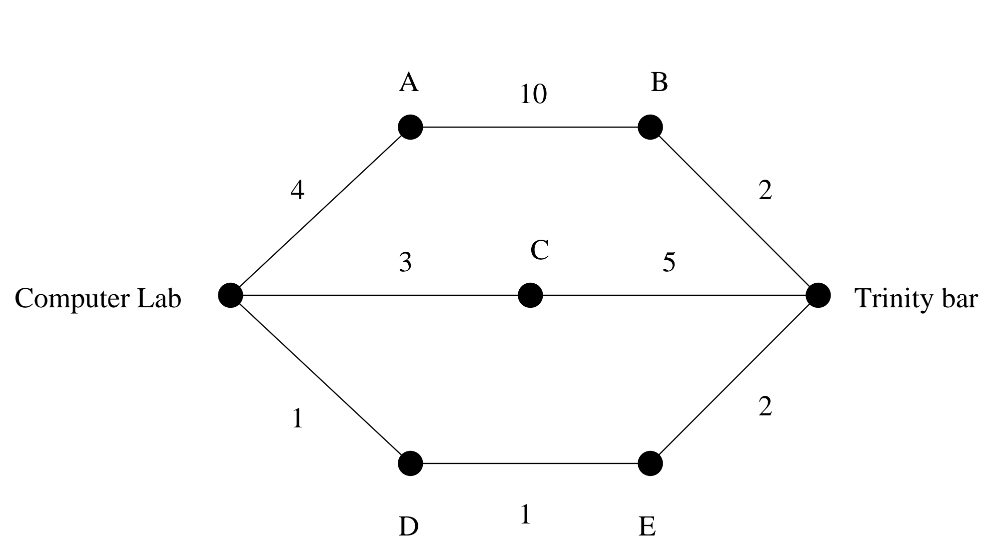
A **solution** is a path beginning with the initial state and ending in a goal state.

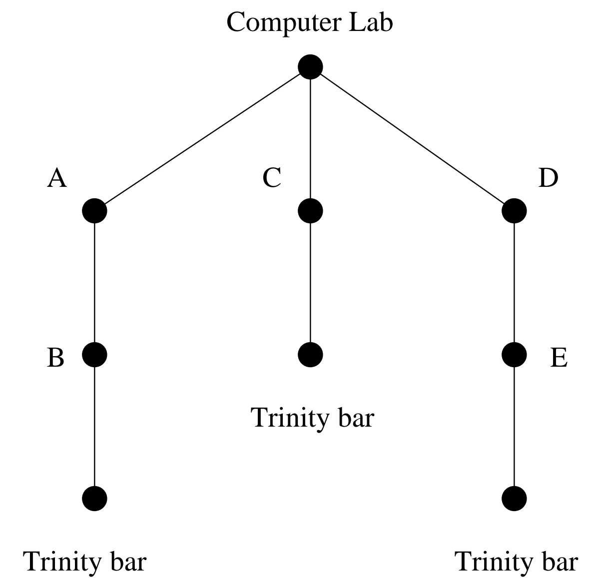
All of the search techniques to be presented can also be applied to multiple-state problems.

- In this case we have an initial *set* of states.
- Each action leads to a further set of states.
- The goal is a set of states *all* of which are valid goals.

Search trees

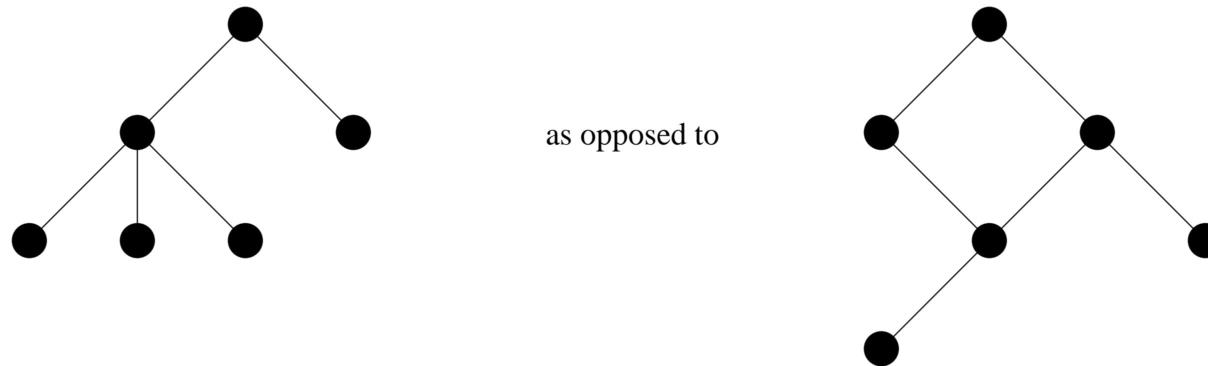
The basic method is familiar from your algorithms course.





Search trees versus search graphs

Note: for the time being we assume this is a **tree** as opposed to a **graph**.



In a **tree** only **one path** can lead to a given state. In a **graph** a **state** can be reached via possibly **multiple paths**.

There is a difference between a **state** and a **node**.

The preceding route-finding example is **not allowed** for now. (But the fix is simple and will be presented in a moment.)

Search trees

We form a **search tree** with the initial state as the root node.

Basic approach:

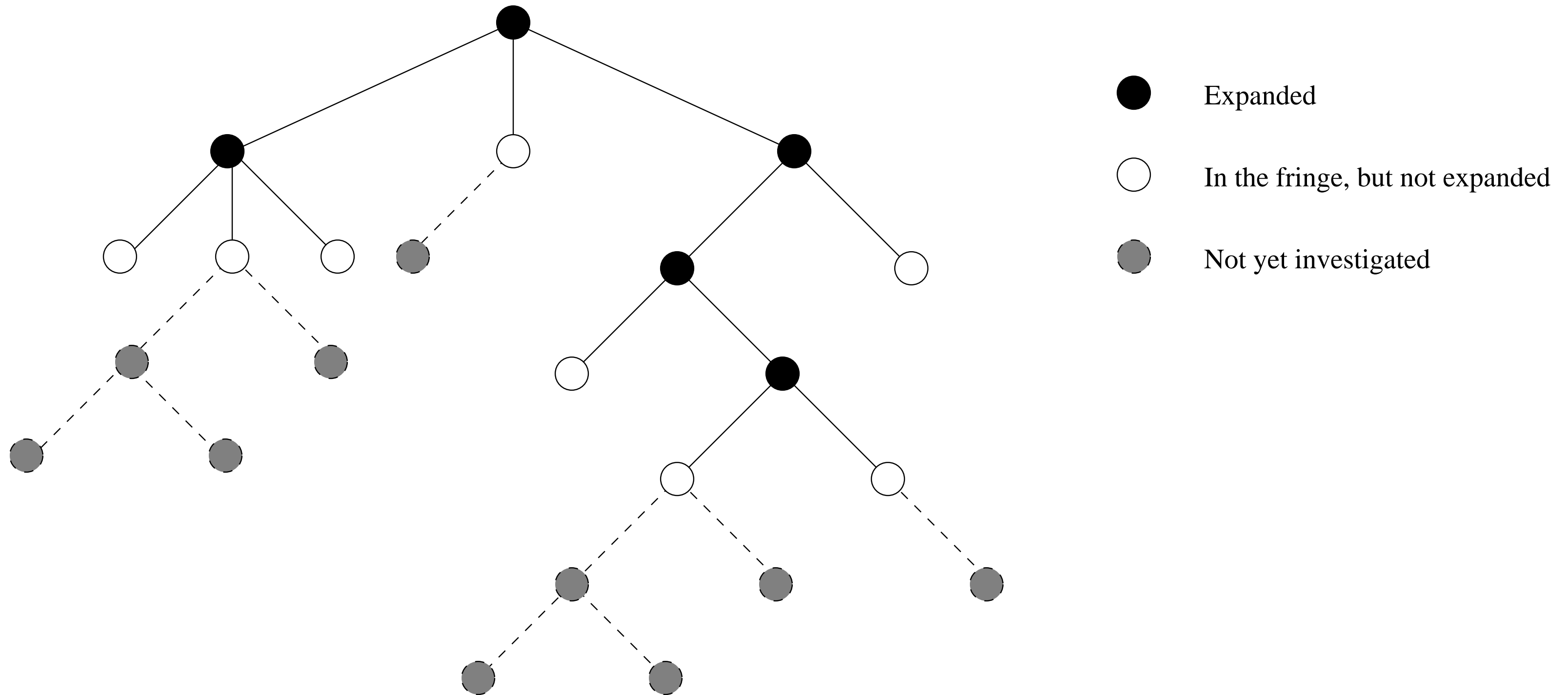
- Test the root to see if it is a goal.
- If not then **expand** it by generating all possible successor states according to the available actions.
- If there is only one outcome state then move to it. Otherwise choose one of the outcomes and expand it. The way in which this choice is made defines a **search strategy**.
- If a choice turns out to be no good then you can go back and try a different alternative.

The collection of states generated but not yet expanded is called the **fringe** or **frontier** and is generally stored as a queue.

The basic tree-search algorithm

```
function tree_search
{
  fringe = queue containing only the start state;
  while()
  {
    if (empty(fringe))
      return fail;
    node = head(fringe);
    if (goal(node))
      return solution(node);
    fringe = insert(expand(node), fringe);
  }
}
```

The basic tree-search algorithm



The performance of search techniques

We are interested in:

- whether a solution is found;
- whether the solution found is a good one in terms of path cost;
- the cost of the search in terms of time and memory.

$$\text{the total cost} = \text{path cost} + \text{search cost}$$

If a problem is highly complex it may be worth settling for a sub-optimal solution obtained in a short time.

Other characteristics of the problem may also be relevant. For example I may not want to spend a huge amount of time working out how to get to Trinity.

Evaluation of search strategies

We are also interested in:

Completeness: does the strategy guarantee a solution is found?

Time complexity

Space complexity

Optimality: does the strategy guarantee that the **best** solution is found?

Search trees

Two types of search:

- **Uninformed** or **blind** search is applicable when we *only* distinguish goal states from non-goal states.

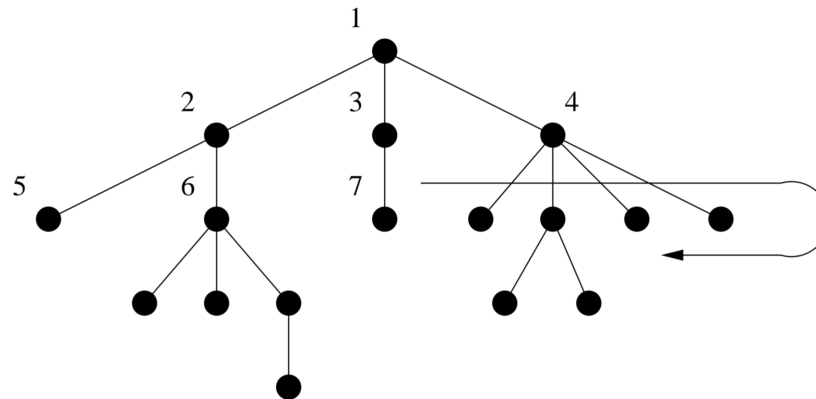
  Methods are distinguished by the order in which nodes in the search tree are expanded. These methods include: breadth-first, depth-first, depth-limited, iterative deepening, bidirectional.

- **Informed** or **heuristic** search is applied if we have some knowledge of the path cost or the number of steps between the current state and a goal.

  These methods include: best first, greedy, A*, iterative deepening A* (IDA*), SMA*.

Breadth-first search

Breadth-first search:



This is familiar from your algorithms courses.

Note:

- the procedure is *complete*: it is guaranteed to find a solution if one exists;
- the procedure is *optimal* under a simple condition: if the path cost is a non-decreasing function of node-depth;
- the procedure has exponential complexity for both memory and time. A branching factor x requires

$$1 + x + x^2 + x^3 + \cdots + x^n$$

nodes if the shortest path has depth n.

In practice: it is the **memory** requirement that is problematic.

Uniform-cost search

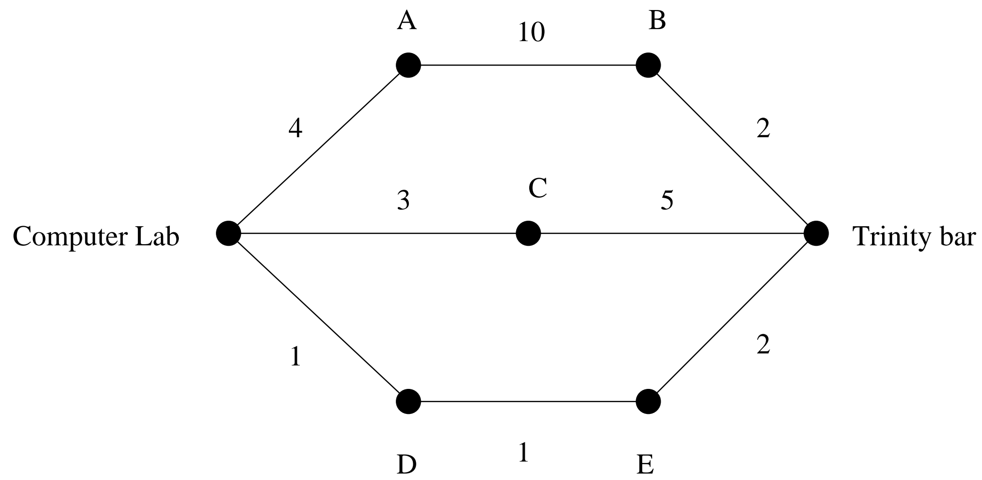
Breadth-first search finds the **shallowest** solution, but this is not necessarily the **best** one.

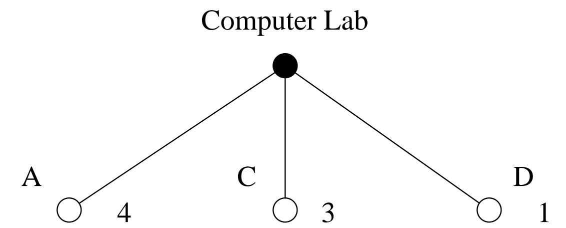
Uniform-cost search differs in that it always expands the node with the **lowest path-cost** $p(n)$ first.

The best solution will always be found if

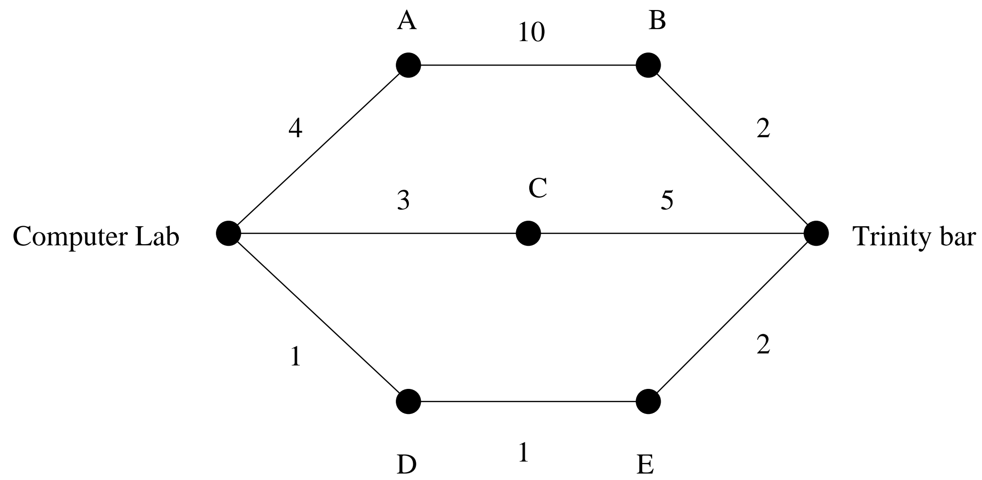
$$\forall \text{node } p(\text{node's successor}) \geq p(\text{node})$$

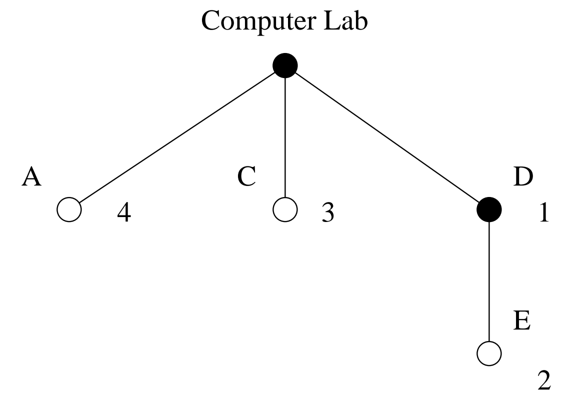
Uniform-cost search



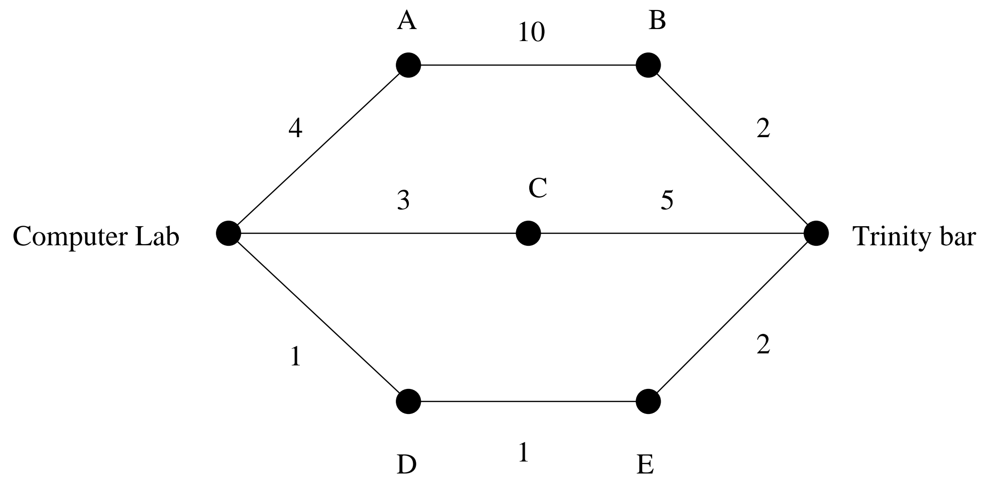


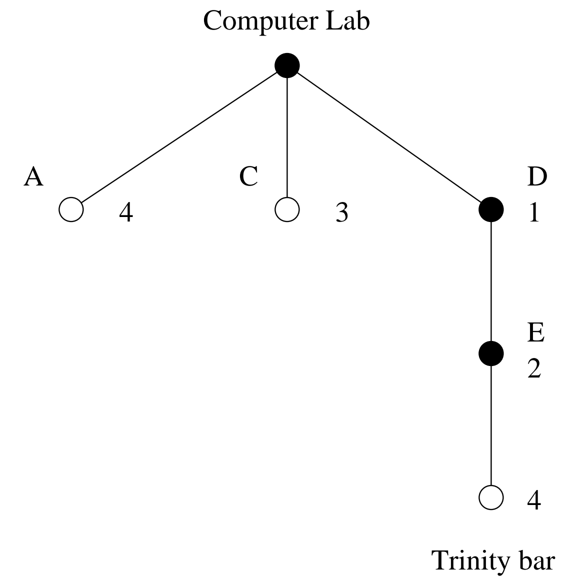
Uniform-cost search



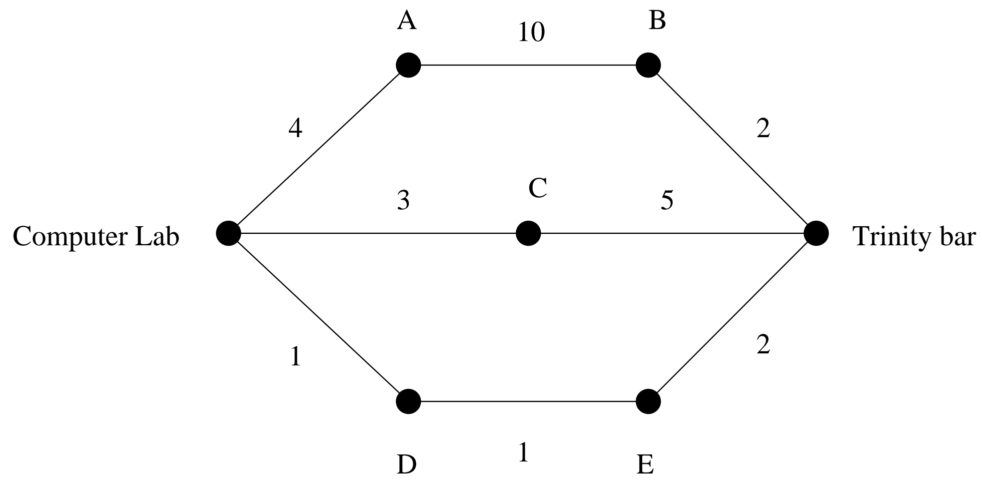


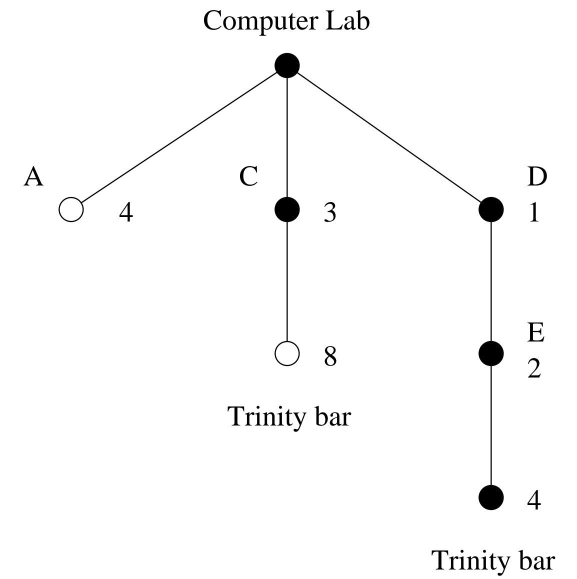
Uniform-cost search



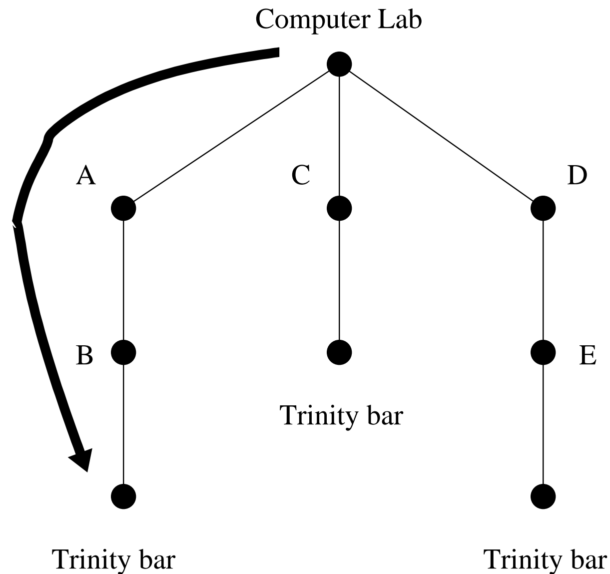


Uniform-cost search





Depth-first search

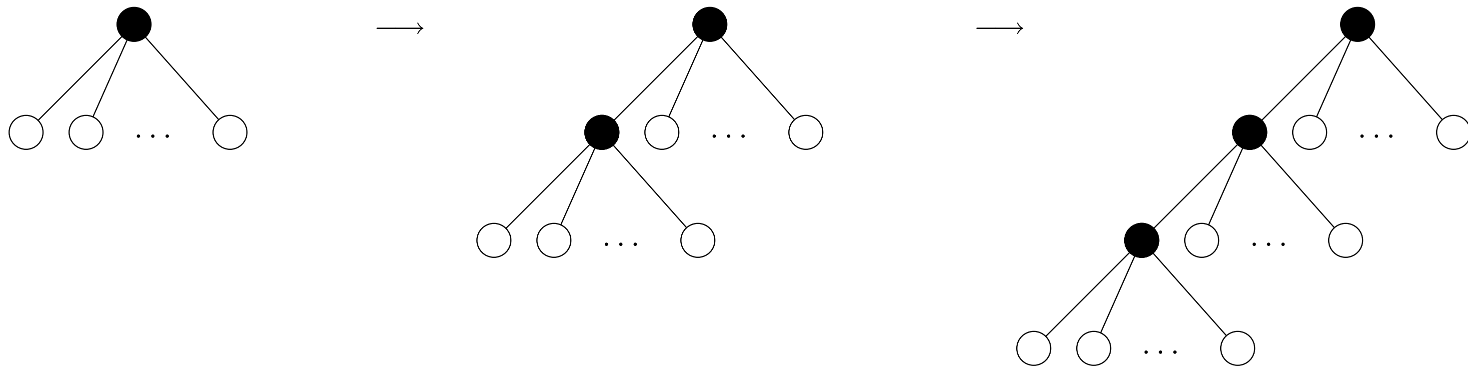


- nodes are expanded at the deepest existing part of the tree;
- for a given branching factor and depth the memory requirement is around branching $\times$ depth and the time $O(\text{branching}^{\text{depth}})$;
- despite the exponential time requirement, if there are **many solutions** this algorithm stands a chance of finding one quickly, compared with breadth-first search.

Depth-first search

The memory requirement is about branching $\times$ depth as we need to store:

- nodes on the current path, and;
- the other unexpanded nodes.



So memory = $O(\text{branching} \times \text{depth})$.

Backtracking search

We can sometimes however improve on this by using **backtracking search**.

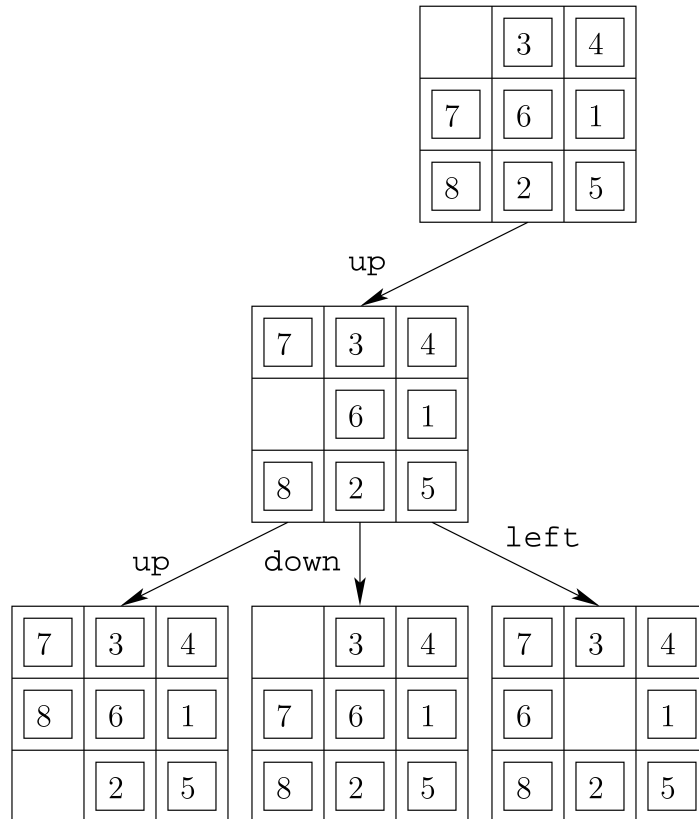
- If each node knows how to **generate the next possibility** then memory is $O(\text{depth})$.
- Even better, if we can work by **making modifications** to a **state description** then the memory requirement is:
  - One full state description, plus...
  - ... $O(\text{depth})$ actions (in order to be able to **undo** actions).

Let's see a simple example...

Backtracking search

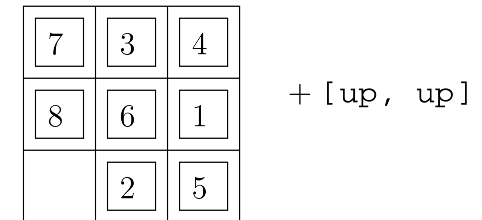
No backtracking

Trying: `up`, `down`, `left`, `right`:

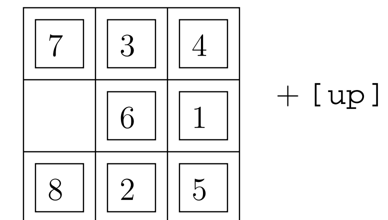


With backtracking

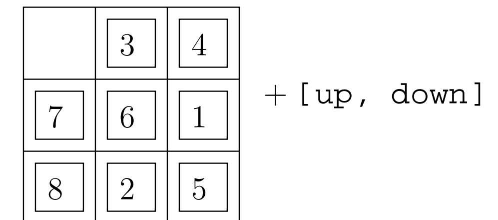
If we have:



we can undo this to obtain



and apply `down` to get



and so on...

Depth-first and depth-limited search

Depth-first search is clearly dangerous if the tree is either very deep or infinite:

- if the tree is very deep we risk finding a suboptimal solution;
- if the tree is infinite we risk an infinite loop.

Depth-limited search simply imposes a limit on depth. For example if we're searching for a route on a map with n cities we know that the maximum depth will be n. However:

- we still risk finding a suboptimal solution;
- the procedure becomes problematic if we impose a depth limit that is too small.

Iterative deepening search

Usually we do not know a reasonable depth limit in advance.

Iterative deepening search repeatedly runs depth-limited search for increasing depth limits $0, 1, 2, \ldots$

- this essentially combines the advantages of depth-first and breadth-first search;
- the procedure is complete and optimal;
- the memory requirement is similar to that of depth-first search;

Importantly, the fact that you're repeating a search process several times is less significant than it might seem.

Iterative deepening search

Intuitively, this is because *the vast majority of the nodes in a tree are in the bottom level*:

- in a tree with branching factor x and depth n the number of nodes is

$$f_1(x, n) = 1 + x + x^2 + x^3 + \cdots + x^n$$

- a complete iterative deepening search of this tree generates the final layer once, the penultimate layer twice, and so on down to the root, which is generated $n + 1$ times. The total number of nodes generated is therefore

$$f_2(x, n) = (n + 1) + nx + (n - 1)x^2 + (n - 2)x^3 + \cdots + 2x^{n-1} + x^n$$

Iterative deepening search

Example:

- for $x = 20$ and $n = 5$ we have

$$f_1(x, n) = 3,368,421$$

$$f_2(x, n) = 3,545,706$$

  which represents a 5 percent increase with iterative deepening search;

- the overhead gets *smaller* as x increases. However the time complexity is still exponential.

For problems where the search space is large and the solution depth is not known, this is the preferred method.

Bidirectional search

We can simultaneously search:

forward from the **start** state

backward from the **goal** state

until the searches meet.

This is potentially a very good idea:

- if the search methods have complexity $O(x^n)$ then...
- ...we are converting this to $O(2x^{n/2}) = O(x^{n/2})$.

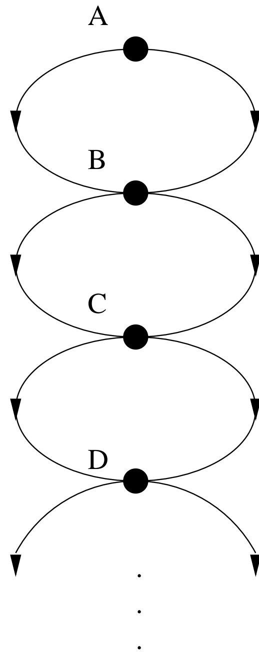
(Here, we are assuming the branching factor is x in both directions.)

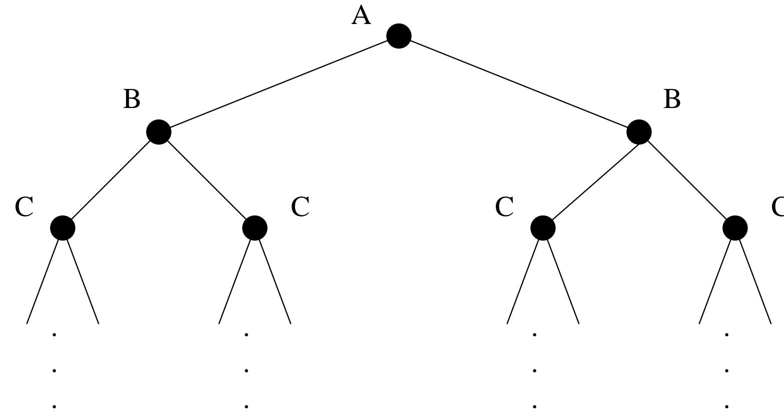
Bidirectional search

- It is not always possible to generate efficiently *predecessors* as well as successors.
- If we only have the *description* of a goal, not an explicit goal, then generating predecessors can be hard. (For example, consider the concept of *checkmate*.)
- We need a way of checking whether or not a node appears in the other search.
- We need to decide what kind of search to use in each half. For example, would depth-first search be sensible?
- The figure of $O(x^{n/2})$ hides the assumption that we can do constant time checking for intersection of the frontiers. Often this is possible using a hash table.
- To guarantee that the searches meet, we need to store all the nodes of at least one of the searches. Consequently the memory requirement is $O(x^{n/2})$.

Repeated states

With many problems it is easy to waste time by expanding nodes that have appeared elsewhere in the tree. For example:





Repeated states

For example, in a problem such as finding a route in a map, where all of the operators are **reversible**, this is inevitable.

There are three basic ways to avoid this, depending on how you trade off effectiveness against overhead.

- never return to the state you came from;
- avoid cycles: never proceed to a state identical to one of your ancestors;
- do not generate *any* state that has previously appeared.

Graph search

Graph search is a standard approach to dealing with the situation:

```
function graph_search
{
  closed = {};
  fringe = queue containing only start state;
  while ()
  {
    if (empty(fringe))
      return fail;
    node = head(fringe);
    if goal(node)
      return solution(node);
    if (node not a member of closed)
    {
      closed = closed + node;
      fringe = insert(expand(node), fringe);
    }
  }
}
```

Graph search

There are several points to note regarding graph search:

1. The **closed list** contains all the expanded nodes.
2. The closed list can be implemented using a hash table.
3. Both worst case time and space are now proportional to the size of the state space.
4. **Memory:** depth first and iterative deepening search are no longer linear space as we need to store the closed list.
5. **Optimality:** when a repeat is found we are discarding the new possibility even if it is better than the first one.
   - This never happens for uniform-cost or breadth-first search with constant step costs, so these remain optimal.
   - Iterative deepening search needs to check which solution is better and if necessary modify path costs and depths for descendants of the repeated state.

Solving problems by search II: informed search

We now look at how an agent might achieve its goals using more sophisticated search techniques.

Aims:

- to introduce the concept of a **heuristic** in the context of search problems;
- to introduce some further algorithms for conducting the necessary search for a sequence of actions, which are able to make use of a heuristic.

Reading: Russell and Norvig, chapter 4.

Problem solving by informed search

Basic search methods make limited use of any **problem-specific knowledge** we might have.

- Use of the available knowledge is limited to the **formulation** of the problem as a search problem.
- We have already seen the concept of **path cost** $p(n)$

  $p(n) =$ cost of any path (sequence of actions) in a state space

- We can now introduce an **evaluation function**. This is a function that attempts to measure the **desirability of each node**.

The evaluation function will clearly not be perfect. (If it is, there is no need to search!)

Best-first search and greedy search

Best-first search simply expands nodes using the ordering given by the evaluation function.

- We could just use path cost, but this is misguided as path cost is not in general **directed** in any sense **toward the goal**.
- A **heuristic function**, usually denoted $h(n)$ is one that **estimates** the cost of the best path from any node n to a goal.
- If n is a goal then $h(n) = 0$.

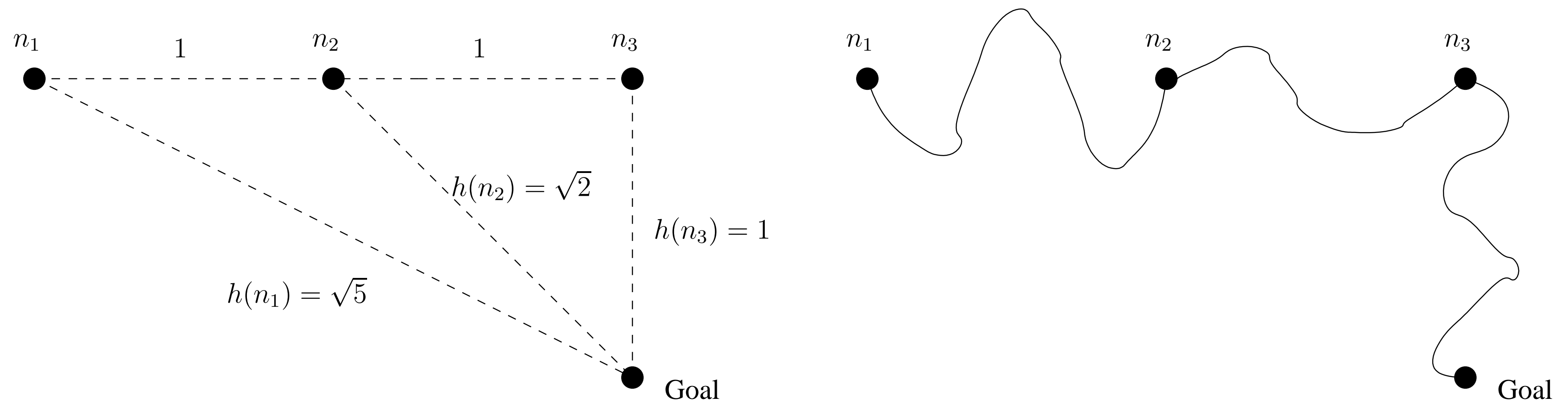
Using a heuristic function along with best-first search gives us the **greedy search** algorithm.

Example: route-finding

A reasonable heuristic function here is

$$h(n) = \text{straight line distance from } n \text{ to the nearest goal}$$

Example:



Example: route-finding

Greedy search suffers from some problems:

- its time complexity is $O(\text{branching}^{\text{depth}})$;
- it is not optimal or complete;
- its space-complexity is $O(\text{branching}^{\text{depth}})$.

BUT: greedy search is often very effective, provided we have a good $h(n)$.

$A^\star$ search

$A^\star$ **search** combines the good points of:

- greedy search—by making use of $h(n)$;
- uniform-cost search—by being optimal and complete.

It does this in a very simple manner: it uses path cost $p(n)$ and also the heuristic function $h(n)$ by forming

$$f(n) = p(n) + h(n)$$

where

$$p(n) = \text{cost of path } \textit{to } n$$

and

$$h(n) = \text{estimated cost of best path } \textit{from } n$$

So: $f(n)$ is the estimated cost of a path **through** n.

$A^\star$ search

$A^\star$ search:

- a best-first search using $f(n)$;
- it is both complete and optimal...
- ...provided that h obeys some simple conditions.

Definition: an **admissible heuristic** $h(n)$ is one that **never overestimates** the cost of the best path from n to a goal.

If $h(n)$ is admissible then tree-search $A^\star$ is optimal.

$A^\star$ tree-search is optimal for admissible $h(n)$

To see that $A^\star$ search is optimal we reason as follows.

Let Goal_{opt} be an optimal goal state with

$$f(\text{Goal}_{\text{opt}}) = p(\text{Goal}_{\text{opt}}) = f_{\text{opt}}$$

(because $h(\text{Goal}_{\text{opt}}) = 0$). Let Goal_2 be a suboptimal goal state with

$$f(\text{Goal}_2) = p(\text{Goal}_2) = f_2 > f_{\text{opt}}$$

We need to demonstrate that the search can never select Goal_2.

$A^\star$ tree-search is optimal for admissible $h(n)$

Let n be a leaf node in the fringe on an optimal path to Goal_{opt}. So

$$f_{\text{opt}} \geq p(n) + h(n) = f(n)$$

because h is admissible.

Now say Goal_2 is chosen for expansion *before* n. This means that

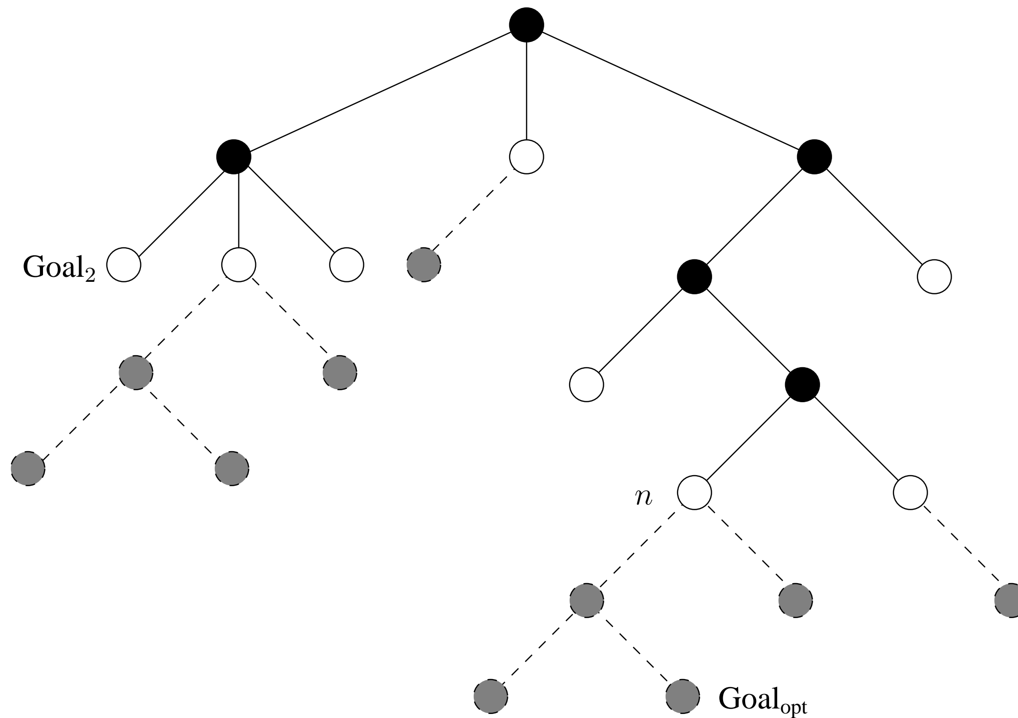
$$f(n) \geq f_2$$

so we've established that

$$f_{\text{opt}} \geq f_2 = p(\text{Goal}_2).$$

But this means that Goal_{opt} is not optimal! A contradiction.

$A^{\star}$ tree-search is optimal for admissible $h(n)$



At some point Goal$_2$ is in the fringe.

Can it be selected before n?

$A^\star$ graph search

Of course, we will generally be dealing with graph search.

Unfortunately the proof breaks in this case.

- Graph search can discard an optimal route if that route is not the first one generated.
- We could keep only the least expensive path. This means updating, which is extra work, not to mention messy, but sufficient to insure optimality.
- Alternatively, we can impose a further condition on $h(n)$ which forces the best path to a repeated state to be generated first.

Monotonicity

Assume h is admissible. Remember that $f(n) = p(n) + h(n)$ so if n' follows n

$$p(n') \geq p(n)$$

and we expect that

$$h(n') \leq h(n)$$

although this does not have to be the case. The possibility remains that $f(n')$ might be *less* than $f(n)$.

- if it is always the case that $f(n') \geq f(n)$ then $h(n)$ is called **monotonic**;
- $h(n)$ is monotonic if and only if it obeys the **triangle inequality**.

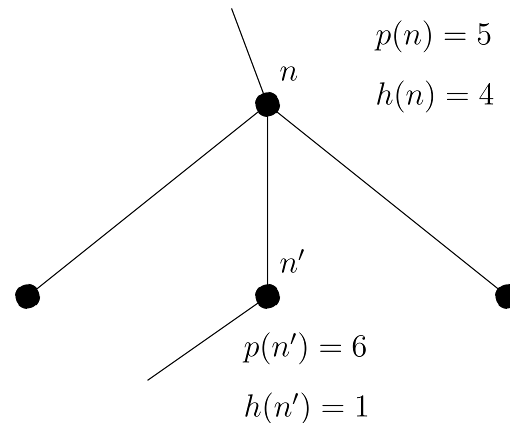
If $h(n)$ is *not* monotonic we can make a simple alteration and use

$$f(n') = \max\{f(n), p(n') + h(n')\}$$

This is called the **pathmax** equation.

The pathmax equation

Why does the pathmax equation make sense?



So here $f(n) = 9$ and $f(n') = 7$.

The fact that $f(n) = 9$ tells us the cost of a path through n is **at least** 9 (because $h(n)$ is admissible).

But n' is **on a path through** n. So to say that $f(n') = 7$ makes no sense.

Monotonic heuristics

A heuristic will be monotonic if

$$h(n) \leq \textsf{cost}(n \xrightarrow{a} n') + h(n')$$

As luck would have it

$$\text{monotonicity} \longrightarrow \text{admissibility}$$

$A^{\star}$ graph search is optimal for monotonic heuristics.

$A^\star$ graph search is optimal for monotonic heuristics

The crucial fact from which optimality follows is that if $h(n)$ is monotonic then the values of $f(n)$ along any path are non-decreasing.

Assume we move from n to n' using action a. Then

$$\forall a, p(n') = p(n) + \textsf{cost}(n \xrightarrow{a} n')$$

and from the last slide

$$h(n) \leq \textsf{cost}(n \xrightarrow{a} n') + h(n') \tag{1}$$

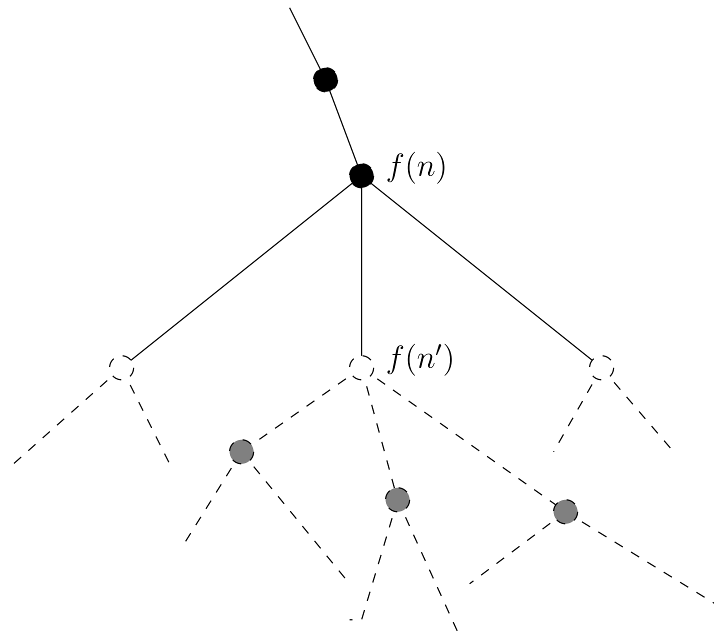
Thus

$$\begin{aligned} f(n') &= p(n') + h(n') \\ &= p(n) + \textsf{cost}(n \xrightarrow{a} n') + h(n') \\ &\geq p(n) + h(n) \\ &= f(n) \end{aligned}$$

where the inequality follows from equation 1.

$A^{\star}$ graph search is optimal for monotonic heuristics

We therefore have the following situation:



You can't deal with n' until **everything** with $f(n'') < f(n')$ has been dealt with.

Consequently everything with $f(n'') < f_{\text{opt}}$ gets explored. Then one or more things with f_{opt} get found (not necessarily all goals).

$A^\star$ search is complete

$A^\star$ search is complete provided:

1. the graph has finite branching factor;
2. there is a finite, positive constant c such that each operator has cost at least c.

Why is this?

$A^\star$ search is complete

The search expands nodes according to increasing $f(n)$. So: the only way it can fail to find a goal is if there are infinitely many nodes with $f(n) < f(\text{Goal})$.

There are two ways this can happen:

1. there is a node with an infinite number of descendants;
2. there is a path with an infinite number of nodes but a finite path cost.

Complexity

- $A^\star$ search has a further desirable property: it is *optimally efficient*.
- This means that no other optimal algorithm that works by constructing paths from the root can guarantee to examine fewer nodes.
- BUT: despite its good properties we're not done yet!
- $A^\star$ search unfortunately still has exponential time complexity in most cases unless $h(n)$ satisfies a very stringent condition that is generally unrealistic:

$$|h(n) - h'(n)| \leq O(\log h'(n))$$

  where $h'(n)$ denotes the *real* cost from n to the goal.
- As $A^\star$ search also stores all the nodes it generates, once again it is generally memory that becomes a problem before time.

IDA* - iterative deepening $A^\star$ search

Iterative deepening search used depth-first search with a limit on depth that gradually increased.

- IDA* does the same thing **with a limit on f cost**.
- It is complete and optimal under the same conditions as $A^\star$.
- Often good if we have step costs equal to 1.
- Does not require us to maintain a sorted queue of nodes.
- It only requires space proportional to the longest path.
- The time taken depends on the number of values h can take.

If h takes enough values to be problematic we can increase f by a fixed ϵ at each stage, guaranteeing a solution at most ϵ worse than the optimum.

IDA* - iterative deepening $A^\star$ search

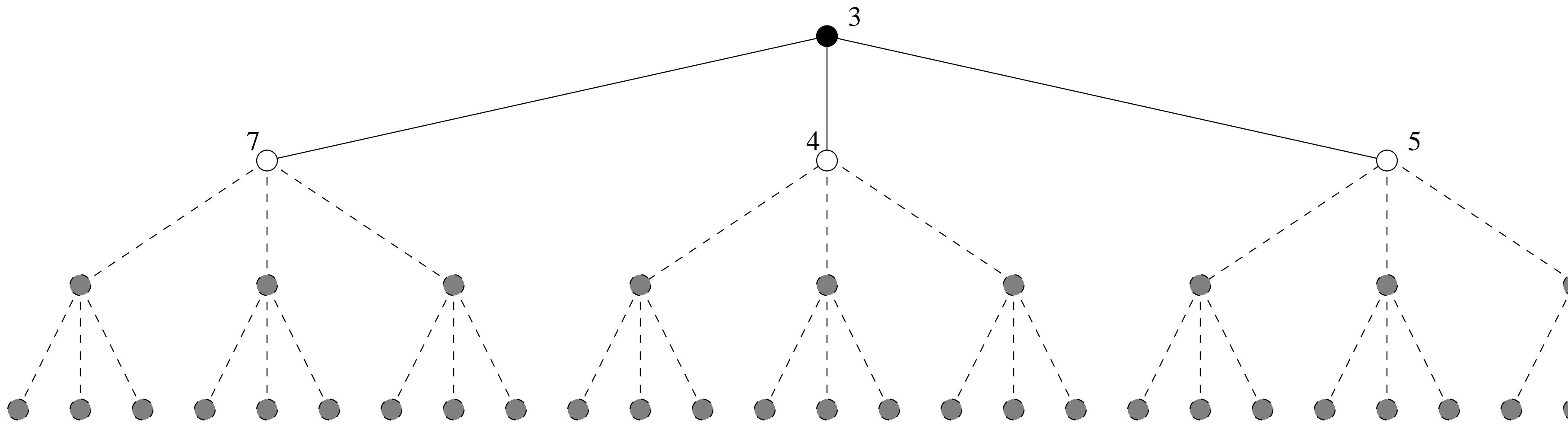
```
Action_sequence ida()
{
   float f_limit = f(root);
   Node root = root node for problem;

   while(true)
   {
      (sequence,f_limit) = contour(root,f_limit);
      if (sequence != empty_sequence)
        return sequence;
      if (f_limit == infinity)
        return empty_sequence;
   }
}
```

IDA* - iterative deepening $A^\star$ search

```
(Action_sequence,float) contour(Node node, float f_limi
{
   float next_f = infinity;
   if (f(node) > f_limit)
      return (empty_sequence,f(node));
   if (goaltest(node))
      return (node,f_limit);
   for (each successor s of node)
   {
      (sequence,new_f) = contour(s,f_limit);
      if (sequence != empty_sequence)
         return (sequence,f_limit);
      next_f = minimum(next_f,new_f);
   }
   return (empty_sequence,next_f);
}
```

IDA* - iterative deepening $A^\star$ search

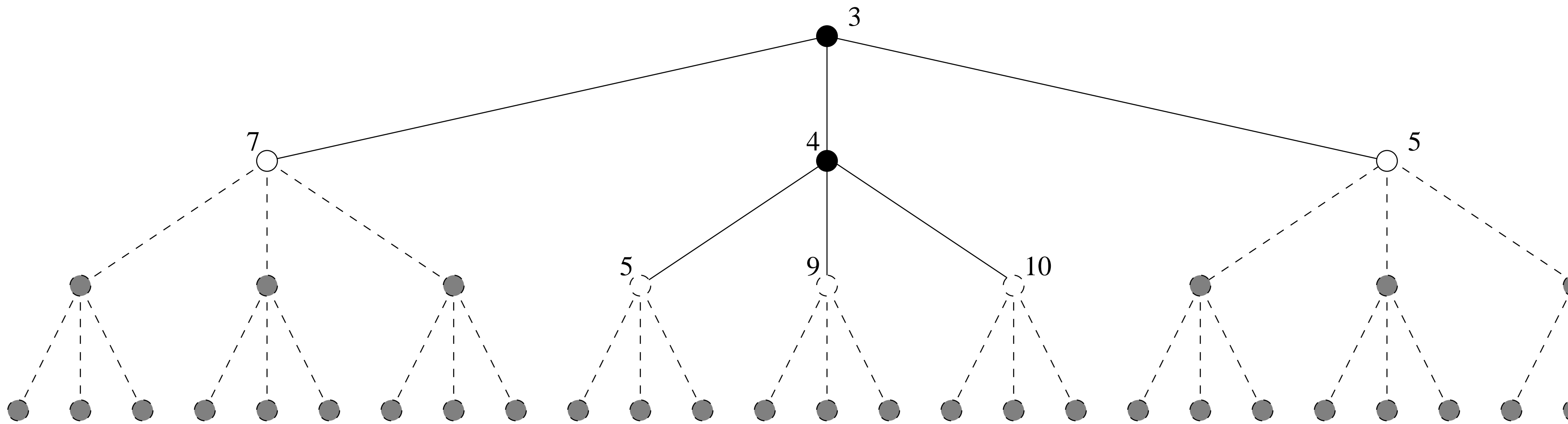
This is a little tricky to unravel, so here is an example:



Initially, the algorithm looks ahead and finds the **smallest** f cost that is **greater than** its current f cost limit. The new limit is 4.

IDA* - iterative deepening $A^\star$ search

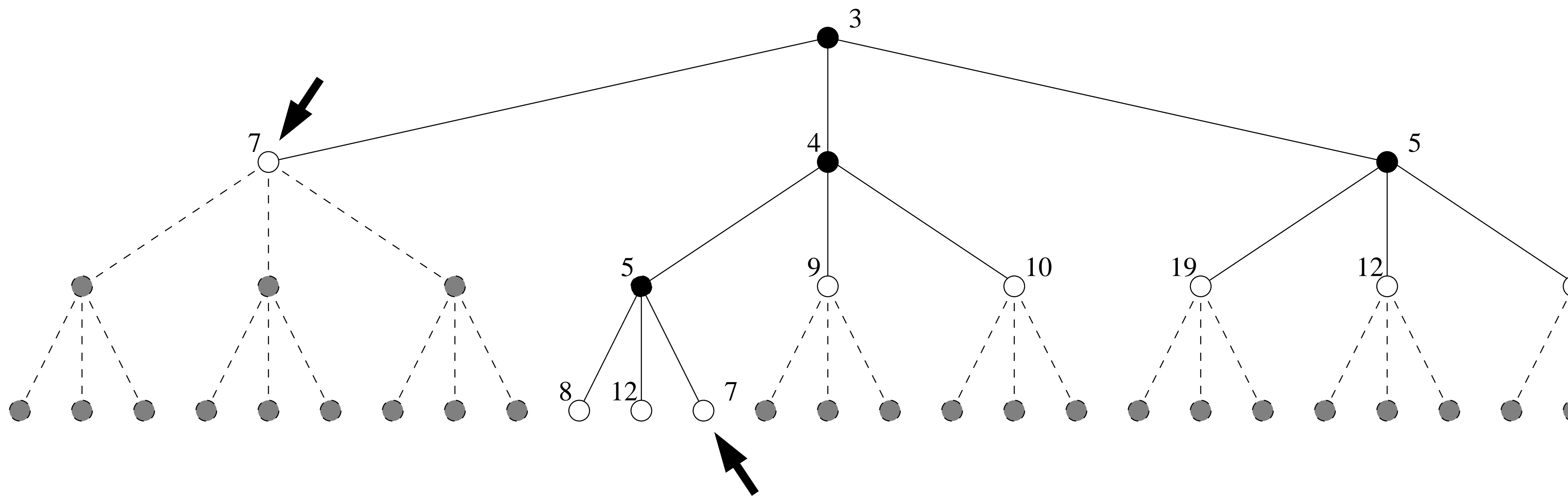
It now does the same again:



Anything with f cost **at most** equal to the current limit gets explored, and the algorithm keeps track of the **smallest** f cost that is **greater than** its current limit. The new limit is 5.

IDA* - iterative deepening $A^\star$ search

And again:



The new limit is 7, so at the next iteration the three arrowed nodes will be explored.

Recursive best-first search (RBFS)

Another method by which we can attempt to overcome memory limitations is the **Recursive best-first search (RBFS)**.

Idea: basically, we try to do a standard best-first search, but in such a way that the space requirement is only linear.

We perform a depth-first search, with a few modifications:

Recursive best-first search (RBFS)

```
function RBFS(Node node, float f_limit)
{
  if (goaltest(node))
    return node;
  if (node has no successors)
    return (fail, infinity);
  for (each successor s of node)
  {
    f(s) = maximum(f(s), f(node))
  }
  while()
  {
    best = successor of node that has the smallest f(s);
    if (f(best) > f_limit)
      return (fail, f(best));
    next_best = second smallest f(s) value for successors of node;
    (result, f(best)) = RBFS(best, minimum(f_limit, next_best));
    if (result is not fail)
      return result;
  }
}
```

Recursive best-first search (RBFS)

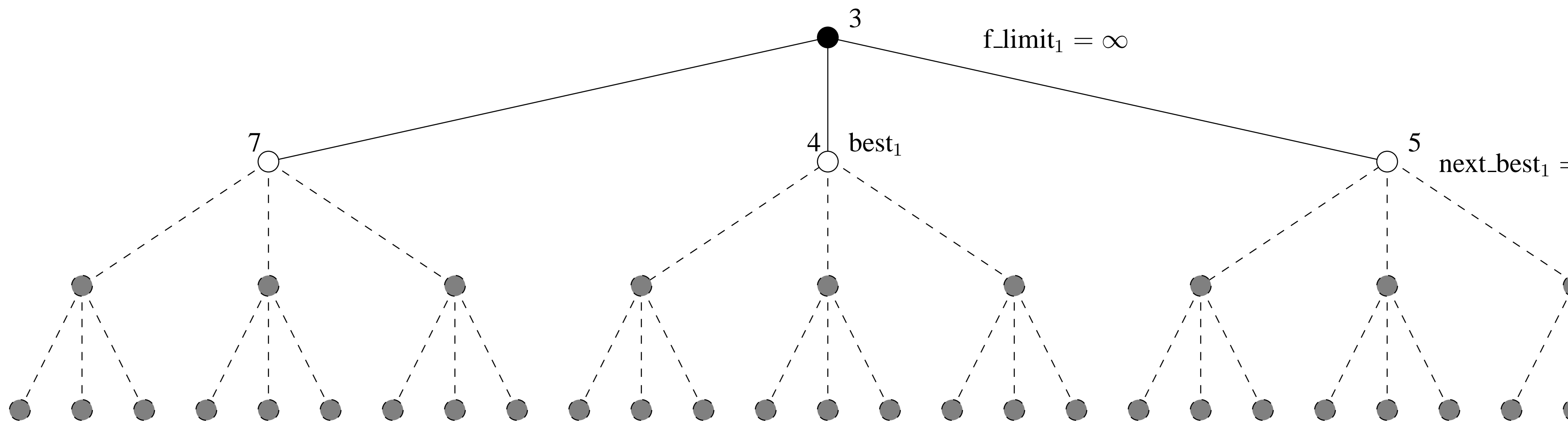
This function is called using `RBFS(start_state, infinity)` to begin the process.

1. We remember the $f(n)$ for the best alternative node n we've seen so far on the way to the node n' we're currently considering.
2. if n' has $f(n') > f(n)$:
   - we go back and explore the best alternative...
   - ...and as we retrace our steps we replace the f cost of every node we're seen in the current path with $f(n')$.

The replacement of f values as we retrace our steps provides a means of remembering how good a discarded path might be, so that we can easily return to it later.

Recursive best-first search (RBFS): an example

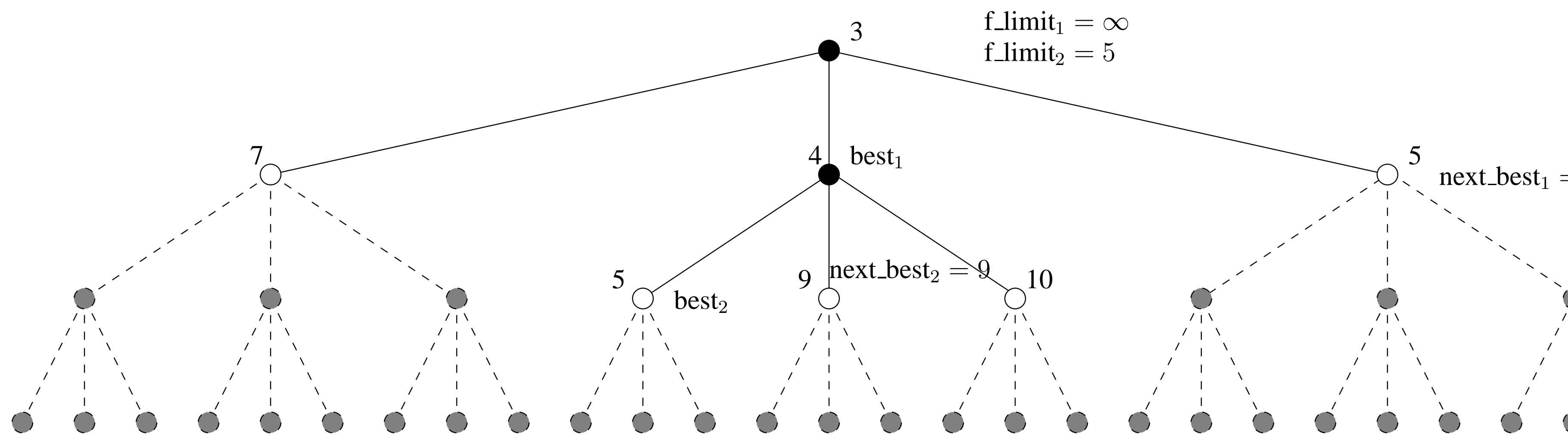
Function call number 1:



Now perform the recursive function call $(\text{result}_2, f(\text{best}_1)) = \text{RBFS}(\text{best}_1, 5)$

Recursive best-first search (RBFS): an example

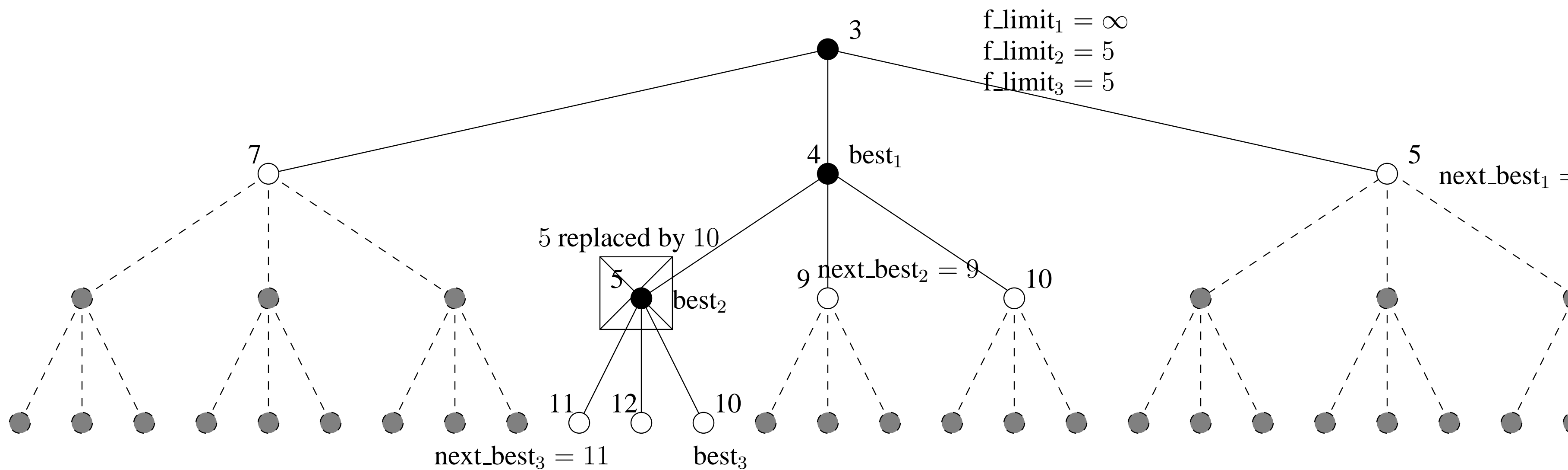
Function call number 2:



Now perform the recursive function call $(\text{result}_3, f(\text{best}_2)) = \text{RBFS}(\text{best}_2, 5)$

Recursive best-first search (RBFS): an example

Function call number 3:



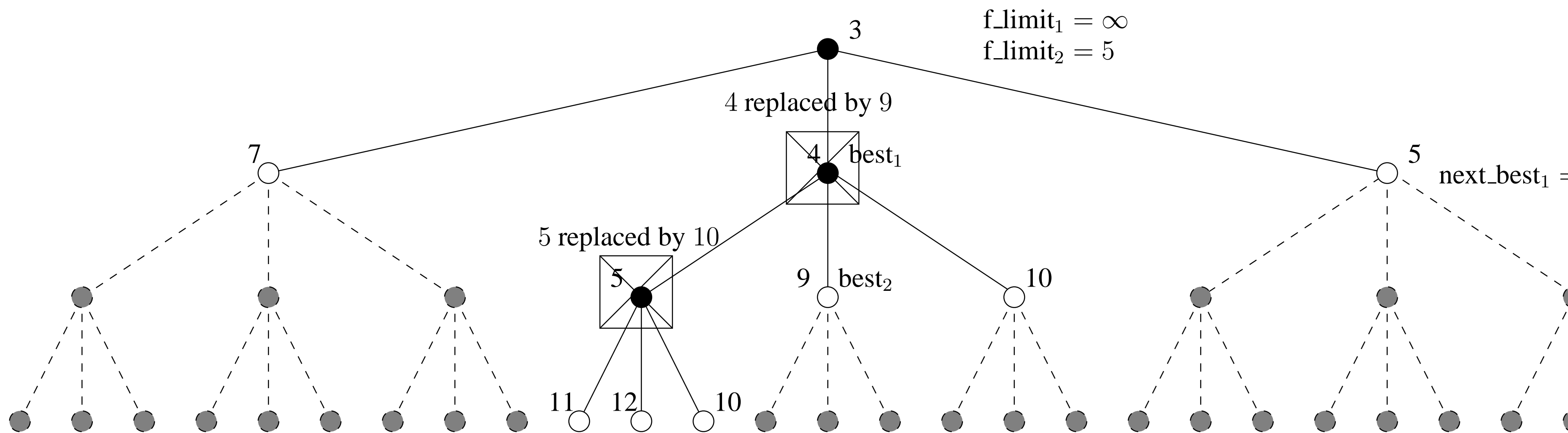
Now

$$f(\text{best}_3) > \text{f_limit}_3$$

so the function call returns $(\text{fail}, 10)$ into $(\text{result}_3, f(\text{best}_2))$.

Recursive best-first search (RBFS): an example

The while loop for function call 2 now repeats:



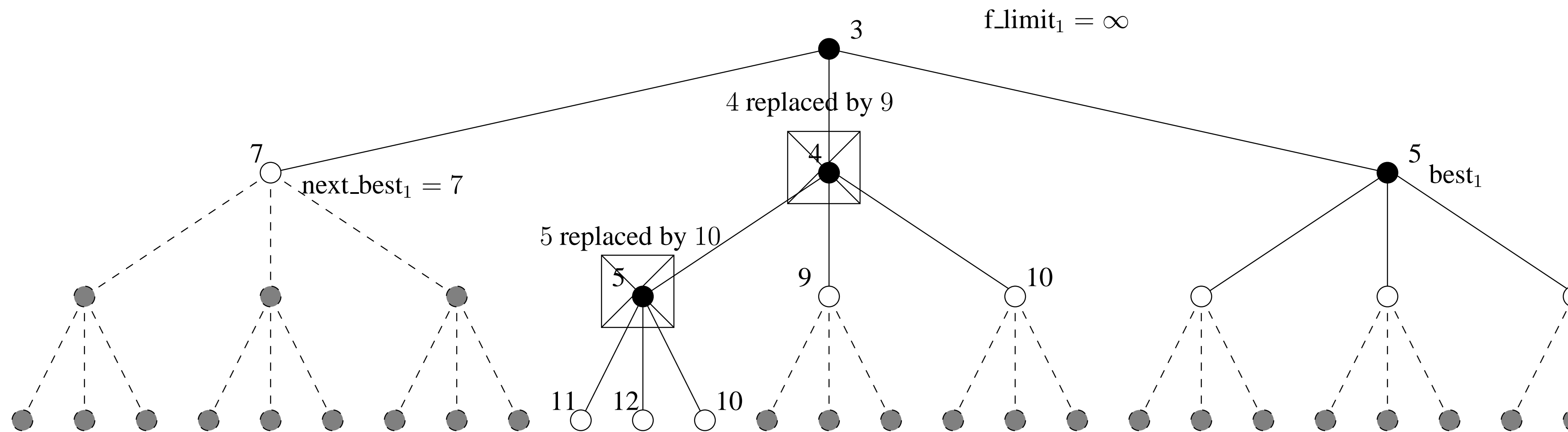
Now

$$f(\text{best}_2) > \text{f_limit}_2$$

so the function call returns $(\text{fail}, 9)$ into $(\text{result}_2, f(\text{best}_1))$.

Recursive best-first search (RBFS): an example

The while loop for function call 1 now repeats:



We do a further function call to expand the new best node, and so on...

Recursive best-first search (RBFS)

Some nice properties:

- If h is admissible then RBFS is optimal.
- Memory requirement = $O(\text{branching} \times \text{depth})$
- Generally more efficient than IDA*.

And some less nice ones:

- Time complexity is hard to analyse, but can be exponential.
- Can spend a lot of time re-generating nodes.

Other methods for getting around the memory problem

To some extent, IDA* and RBFS throw the baby out with the bathwater.

- They limit memory too harshly, so...
- ...we can try to use **all available memory**.

MA* and SMA* will not be covered in this course...

Solving problems by search III: playing games

We now look at how an agent might act when the outcomes of its actions are not known because an **adversary** is trying to hinder it. We look specifically at the example of **playing games**.

Aims:

- to show how game-playing can be modelled as search;
- to introduce the **minimax** algorithm for game-playing;
- to look at some problems inherent in the use of minimax and to introduce methods for their solution;
- to introduce the concept of $\alpha - \beta$ **pruning**.

Reading: Russell and Norvig, chapter 6.

Playing games: search against an adversary

Something is missing from our existing formulation of problem-solving by search.

- What if we do not know the exact outcome of an action?
- Game playing is a good example: in chess, drafts, and so on an opponent **responds** to our moves.
- We don't know what their response will be, and so the outcome of our moves is not clear.

Game playing has traditionally been of interest in AI because it provides an **idealisation** of a world in which two agents act to **reduce** each other's well-being.

Playing games: search against an adversary

Nonetheless, game playing can be an excellent source of hard problems.

For instance with chess:

- the average branching factor is roughly 35;
- games can reach 50 moves per player;
- so a rough calculation gives the search tree 35^{100} nodes;
- even if only different, legal positions are considered it's about 10^{40}.

Playing games: search against an adversary

As well as dealing with uncertainty due to an opponent:

- we can't make a complete search to find the best move...
- ... so we have to act even though we're not sure about the best thing to do.

It therefore seems that games are a step closer to the complexities inherent in the world around us than are the standard search problems considered so far.

Playing games: search against an adversary

Note:

- "Go" is *much* harder than chess!
- The branching factor is about 360.

If you want to make yourself:

- rich (there's a $2,000,000$ dollar prize if your program can beat a top-level player), and;
- famous (nobody is anywhere near winning the prize);

then you should get to work.

Perfect decisions in a two-person game

Say we have two players, called Maxwell and Minny - Max and Min for short.

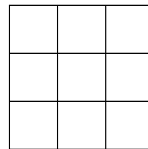
(Yes, there is a reason for this.)

- We'll use noughts and crosses as an initial example.
- Max moves first.
- The players alternate until the game ends.
- At the end of the game, prizes are awarded. (Or punishments administered.)

Perfect decisions in a two-person game

Games like this can be modelled as search problems.

- There is an **initial state**.

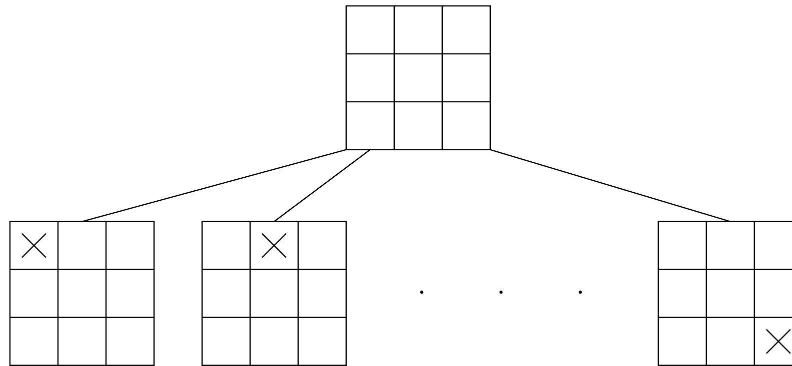


Max to move

- There is a set of **operators**. Here, Max can place a cross in any empty square, or Min a nought.
- There is a **terminal test**. Here, the game ends when three noughts or three crosses are in a row, or there are no unused spaces.
- There is a **utility** or **payoff** function. This tells us, numerically, what the outcome of the game is.

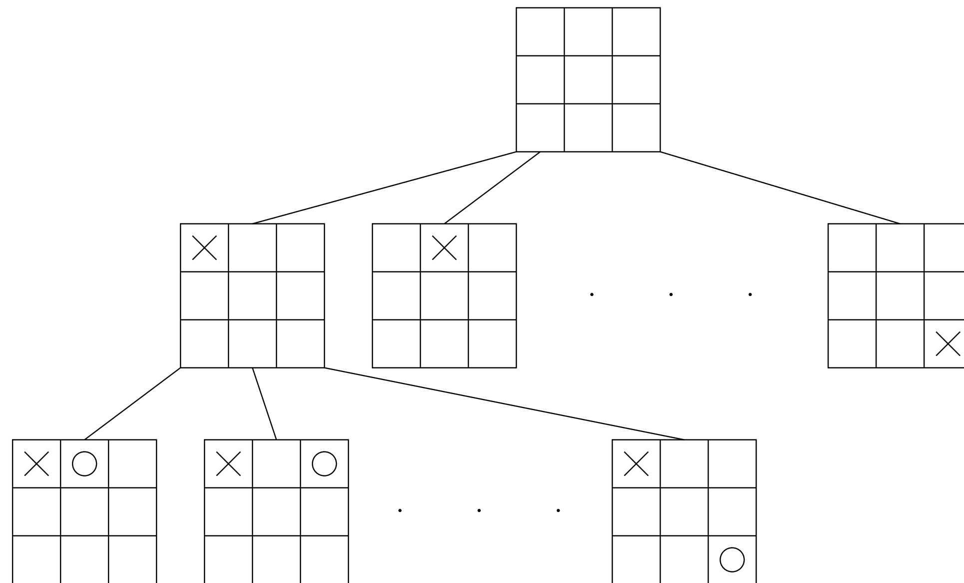
Perfect decisions in a two-person game

We can construct a tree to represent a game. From the initial state, Max can make nine possible moves:



Perfect decisions in a two-person game

In each case, Min has eight replies:



And so on.

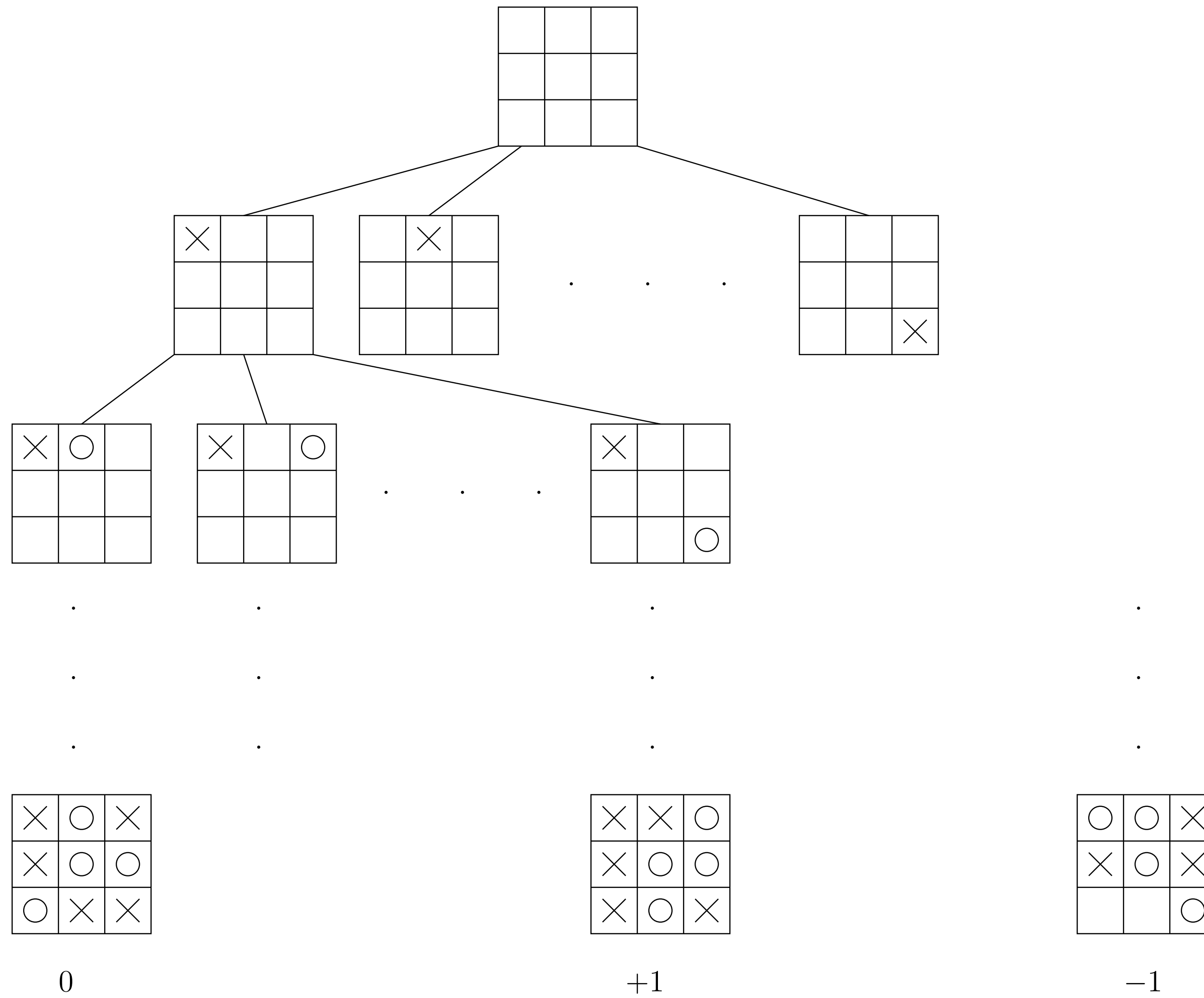
Perfect decisions in a two-person game

This construction can in principle be continued to represent *all* possibilities for the game.

The leaves are situations where a player has won, or there are no spaces.

Each leaf is labelled using the utility function.

Perfect decisions in a two-person game

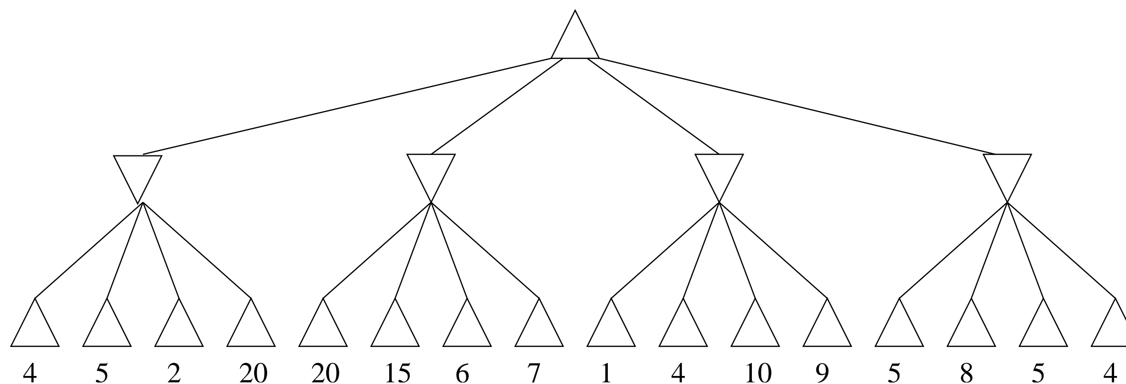


Perfect decisions in a two-person game

How can Max use this tree to decide on a move?

- if he is rational he will play to reach a position with the biggest utility possible;
- bit if Min is rational, she will play to *minimise* the utility available to Max.

Consider a much simpler tree:

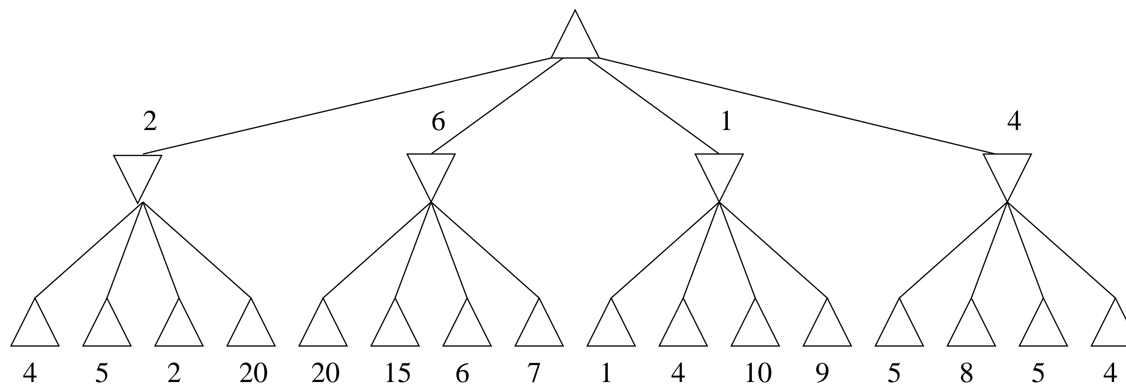


The minimax algorithm

There are only two moves—one for each player. Game theorists would call this one move, or two *ply* deep.

- Max's utility is labelled for each terminal state.
- The **minimax algorithm** allows us to infer the best move that the current player can make, given the utility function.

We work backward from the leaves. In the current example:



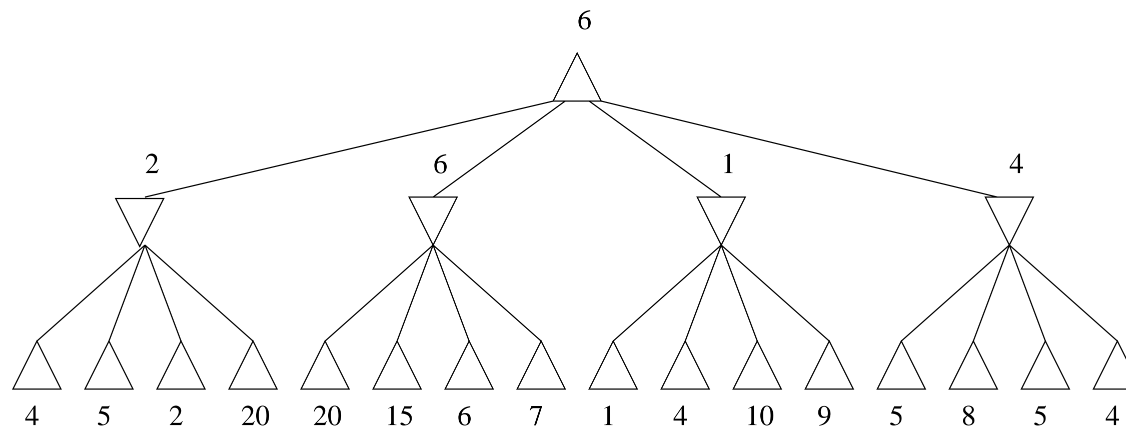
The minimax algorithm

Min takes the final move:

- If Min is in game position 1, her best choice is move 3. So from Max's point of view this node has a utility of 2.
- If Min is in game position 2, her best choice is move 3. So from Max's point of view this node has a utility of 6.
- If Min is in game position 3, her best choice is move 1. So from Max's point of view this node has a utility of 1.
- If Min is in game position 4, her best choice is move 4. So from Max's point of view this node has a utility of 4.

The minimax algorithm

Moving one further step up the tree:



We can see that Max's best opening move is move 2, as this leads to the node with highest utility.

The minimax algorithm

In general:

- generate the complete tree and label the leaves according to the utility function;
- working from the leaves of the tree upward, label the nodes depending on whether Max or Min is to move;
- if Min is to move label the current node with the *minimum* utility of any descendant;
- if Max is to move label the current node with the *maximum* utility of any descendant.

If the game is p ply and at each point there are q available moves then this process has $O(q^p)$ time complexity and space complexity linear in p and q.

Making imperfect decisions

We need to avoid searching all the way to the end of the tree. So:

- we generate only part of the tree: instead of testing whether a node is a leaf we introduce a **cut-off** test telling us when to stop;
- instead of a utility function we introduce an **evaluation function** for the evaluation of positions for an incomplete game.

The evaluation function attempts to measure the expected utility of the current game position.

Making imperfect decisions

How can this be justified?

- This is a strategy that humans clearly sometimes make use of.
- For example, when using the concept of **material value** in chess.
- The effectiveness of the evaluation function is **critical**...
- ... but it must be computable in a reasonable time.
- (In principle it could just be done using minimax!)

The evaluation function

Designing a good evaluation function can be extremely tricky:

- let's say we want to design one for chess by giving each piece its material value: pawn = 1, knight/bishop = 3, rook = 5 and so on;
- define the evaluation of a position to be the difference between the material value of black's and white's pieces

$$\text{eval}(\text{position}) = \sum_{\text{black's pieces } p_i} \text{value of } p_i - \sum_{\text{white's pieces } q_i} \text{value of } q_i$$

This seems like a reasonable first attempt. Why might it go wrong?

The evaluation function

Consider what happens at the start of a game:

- until the first capture the evaluation function gives 0, so in fact we have a *category* containing many different game positions with equal estimated utility.
- For example, all positions where white is one pawn ahead.
- The evaluation function for such a category should represent the probability that a position chosen at random from it leads to a win.

The evaluation function

Considering individual positions.

If on the basis of past experience a position has 50% chance of winning, 10% chance of losing and 40% chance of reaching a draw, we might give it an evaluation of

$$\text{eval}(\text{position}) = (0.5 \times 1) + (0.1 \times -1) + (0.4 \times 0) = 0.4.$$

Extending this to the evaluation of categories, we should then weight the positions in the category according to their likelihood of occurring.

The evaluation function

Using material advantage as suggested gives us a *weighted linear evaluation function*

$$\text{eval}(\text{position}) = \sum_{i=1}^{n} w_i f_i$$

where the w_i are *weights* and the f_i represent *features* of the position. In this example

$$f_i = \text{value of the } i\text{th piece}$$

$$w_i = \text{number of } i\text{th pieces on the board}$$

where black and white pieces are regarded as different and the f_i are positive for one and negative for the other.

The evaluation function

Evaluation functions of this type are very common in game playing.

There is no systematic method for their design.

Weights can be chosen by allowing the game to play itself and using *learning* techniques to adjust the weights to improve performance.

$\alpha - \beta$ pruning

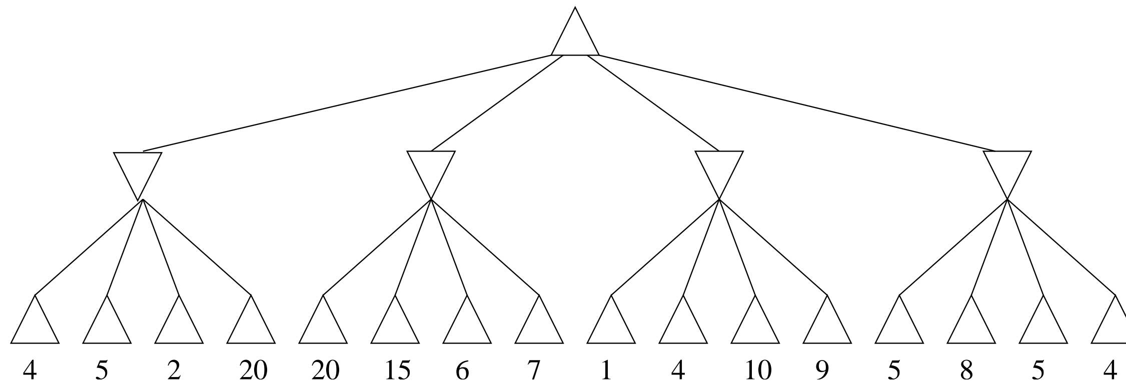
Even with a good evaluation function and cut-off test, the time complexity of the minimax algorithm makes it impossible to write a good chess program without some further improvement.

- Assuming we have 150 seconds to make each move, for chess we would be limited to a search of about 3 to 4 ply whereas...
- ...even an average human player can manage 6 to 8.

Luckily, it is possible to prune the search tree without affecting the outcome and without having to examine all of it.

$\alpha - \beta$ pruning

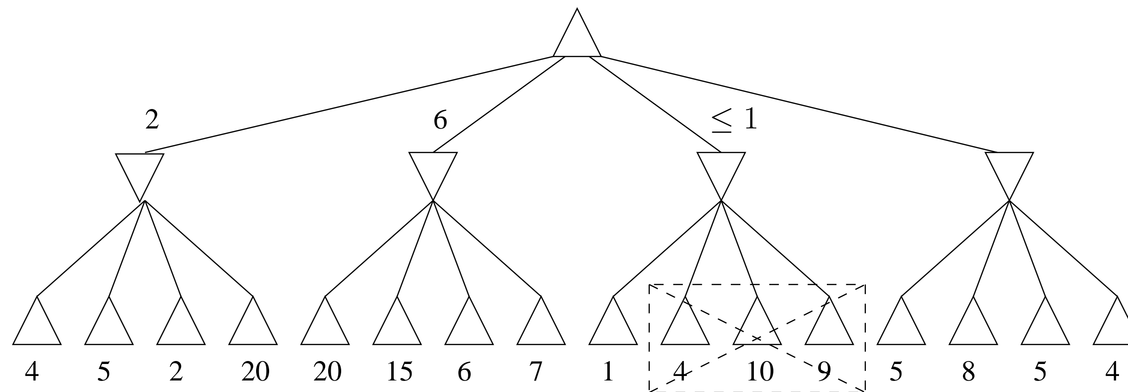
Returning for a moment to the earlier, simplified example:



The search is depth-first and left to right.

$\alpha - \beta$ pruning

The search continues as previously for the first 8 leaves.



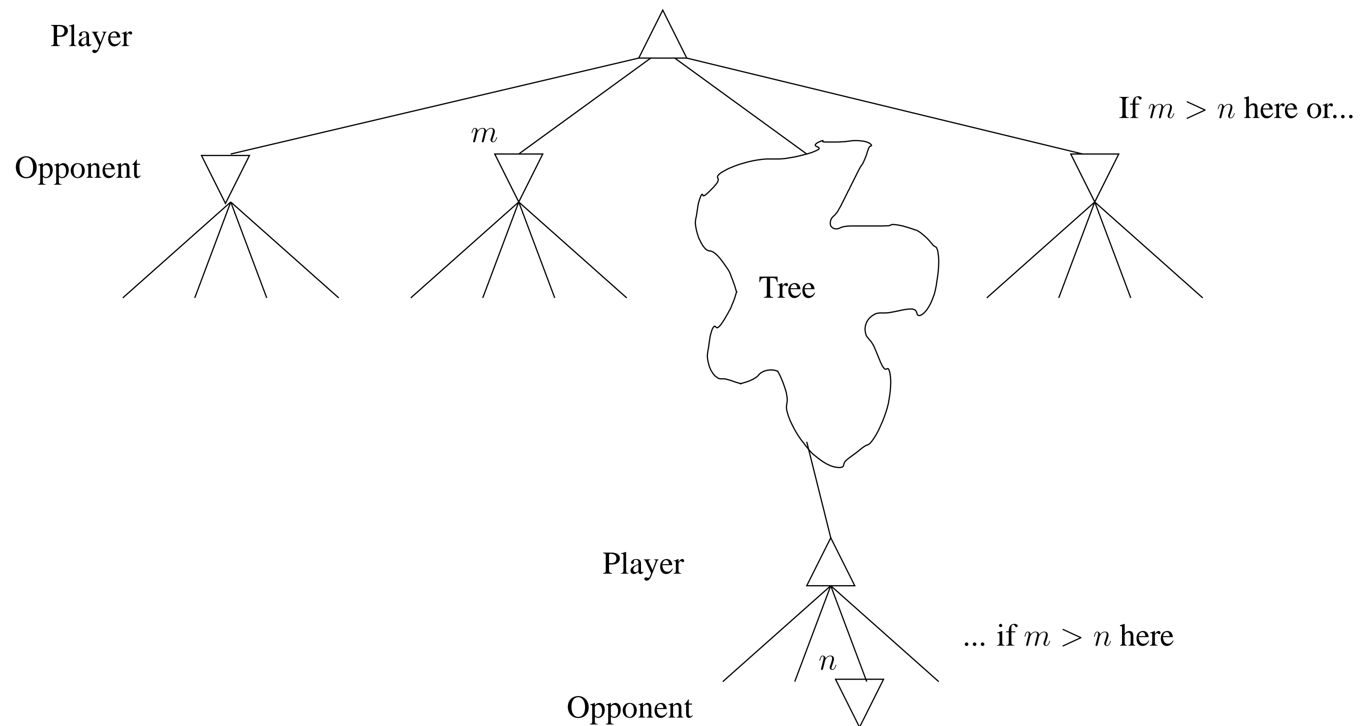
We then discover that should Max play move 3 as an opening, Min has the option of reaching a leaf with utility at most 1!

So: **we don't need to search any further under Max's opening move 3**.

This is because the search has **already established** that Max can do better by making opening move 2.

$\alpha - \beta$ pruning in general

If...



... then n will *never actually be reached*.

$\alpha - \beta$ pruning in general

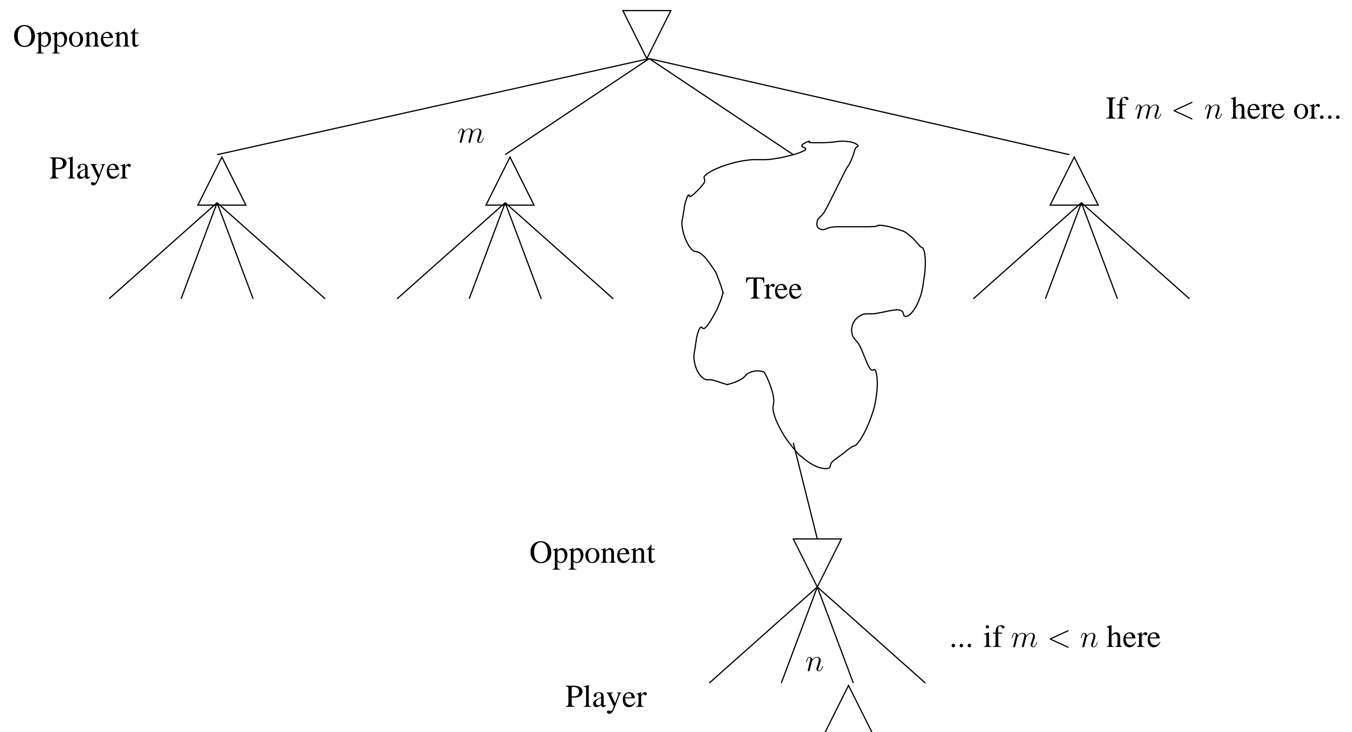
The search is depth-first, so we're only ever looking at one path through the tree.

We need to keep track of the value α where

$$\alpha = \text{the highest utility seen so far on the path for Max}$$

$\alpha - \beta$ pruning in general

Similarly, if...



...then again n will never actually be reached. We keep track of $\beta =$ the lowest utility seen so far on the path for Min.

$\alpha - \beta$ pruning in general

Assume Max begins.

Initial values for α and β are

$$\alpha = -\infty$$

and

$$\beta = +\infty.$$

So: we call the function $\max(-\infty, +\infty, \texttt{root})$.

$\alpha - \beta$ pruning in general

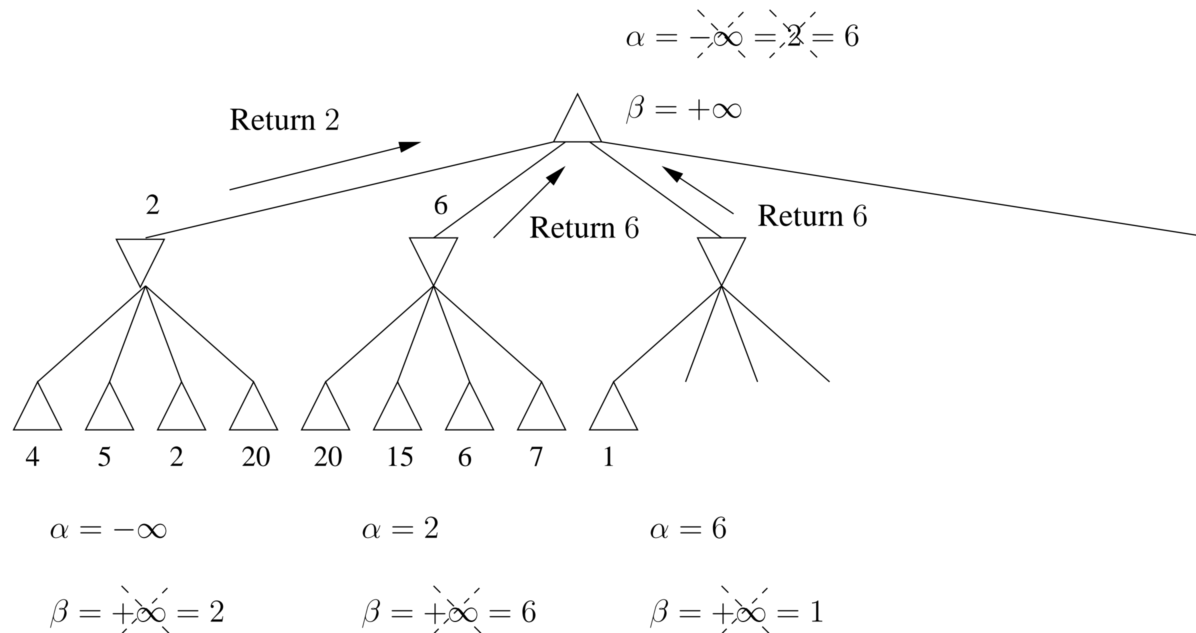
```
max(alpha,beta,node)
{
  if (node is at cut-off)
    return evaluation(node);
  else
    {
      for (each successor s of node)
        {
          alpha = maximum(alpha,min(alpha,beta,s));
          if (alpha >= beta)
            return beta; // pruning happens here.
        }
      return alpha;
    }
}
```

$\alpha - \beta$ pruning in general

```
min(alpha,beta,node)
{
  if (node is at cut-off)
    return evaluation(node);
  else
    {
      for (each successor s of node)
        {
          beta = minimum(beta,max(alpha,beta,s));
          if (beta <= alpha)
            return alpha; // pruning happens here.
        }
      return beta;
    }
}
```

$\alpha - \beta$ pruning in general

Applying this to the earlier example and keeping track of the values for α and β you should obtain:



How effective is $\alpha - \beta$ pruning?

(Warning! The theoretical results that follow are somewhat idealised.)

A quick inspection should convince you that the *order* in which moves are arranged in the tree is critical.

So, it seems sensible to try good moves first:

- if you were to have a perfect move-ordering technique the $\alpha - \beta$ pruning would be $O(q^{p/2})$ as opposed to $O(q^p)$;
- so the branching factor would effectively be $\sqrt{q}$ instead of q;
- and we would expect to be able to search ahead twice as many moves as before.

However, this is not realistic: if you had such an ordering technique you'd be able to play perfect games!

How effective is $\alpha - \beta$ pruning?

If moves are arranged at random then $\alpha - \beta$ pruning is:

- $O((q/\log q)^p)$ asymptotically when $q > 1000$ or;
- about $O(q^{3p/4})$ for reasonable values of q.

In practice simple ordering techniques can get close to the best case. For example, if we try captures, then threats, then moves forward *etc*.

Alternatively, we can implement an iterative deepening approach and use the order obtained at one iteration to drive the next.

Introduction to constraint satisfaction problems

We now return to the idea of problem solving by search and examine it from a slightly different perspective.

Aims:

- To introduce the idea of a **constraint satisfaction problem (CSP)** as a general means of representing and solving problems by search.
- To look at the basic **backtracking algorithm** for solving CSPs.
- To look at some basic **heuristics** for solving CSPs.

Reading: Russell and Norvig, chapter 5.

Constraint satisfaction problems

The search scenarios examined so far seem in some ways unsatisfactory.

- States were represented using an **arbitrary** and **problem-specific** data structure.
- Heuristics, similarly, were problem-specific.

Constraint satisfaction problems

CSPs **standardise** the manner in which states and goal tests are represented.

- As a result we can devise **general purpose** algorithms and heuristics.
- The form of the goal test can tell us about the structure of the problem.
- Consequently it is possible to introduce techniques for decomposing problems.
- We can also try to understand the relationship between the **structure** of a problem and the **difficulty of solving it**.

Constraint satisfaction problems

We have:

- A set of n **variables** $V_1, V_2, \ldots, V_n$.
- For each V_i, a **domain** D_i specifying the values that V_i can take.
- A set of m **constraints** $C_1, C_2, \ldots, C_m$.

Each constraint C_i involves a set of variables and specifies an allowable collection of values.

- A **state** is an assignment of specific values to some or all of the variables.
- An assignment is **consistent** if it violates no constraints.
- An assignment is **complete** if it gives a value to every variable.

A **solution** is a consistent and complete assignment.

Formulation of CSPs as standard search problems

Clearly a CSP can be formulated as a search problem in the familiar sense:

- **Initial state**: $\{\}$—no variables are assigned.
- **Successor function**: assigns value(s) to currently unassigned variable(s) provided constraints are not violated.
- **Goal**: reached if all variables are assigned.
- **Path cost**: constant c per step.

In addition:

- The tree is limited to depth n so depth-first search is usable.
- We don't mind what path is used to get to a solution, so it is feasible to allow every state to be a complete assignment whether consistent or not. (Local search is a possibility.)

Varieties of CSP

The simplest possible CSP will be *discrete* with *finite domains* and we will concentrate on these.

1. Discrete CSPs with *infinite domains*:
   - will need a *constraint language*. For example
     $$V_3 \leq V_{10} + 5$$
   - Algorithms are available for integer variables and linear constraints.
   - There is *no algorithm* for integer variables and nonlinear constraints.
2. Continuous domains:
   - Using linear constraints defining convex regions we have *linear programming*.
   - This is solvable in polynomial time in n.

Types of constraint

We will concentrate on **binary constraints**.

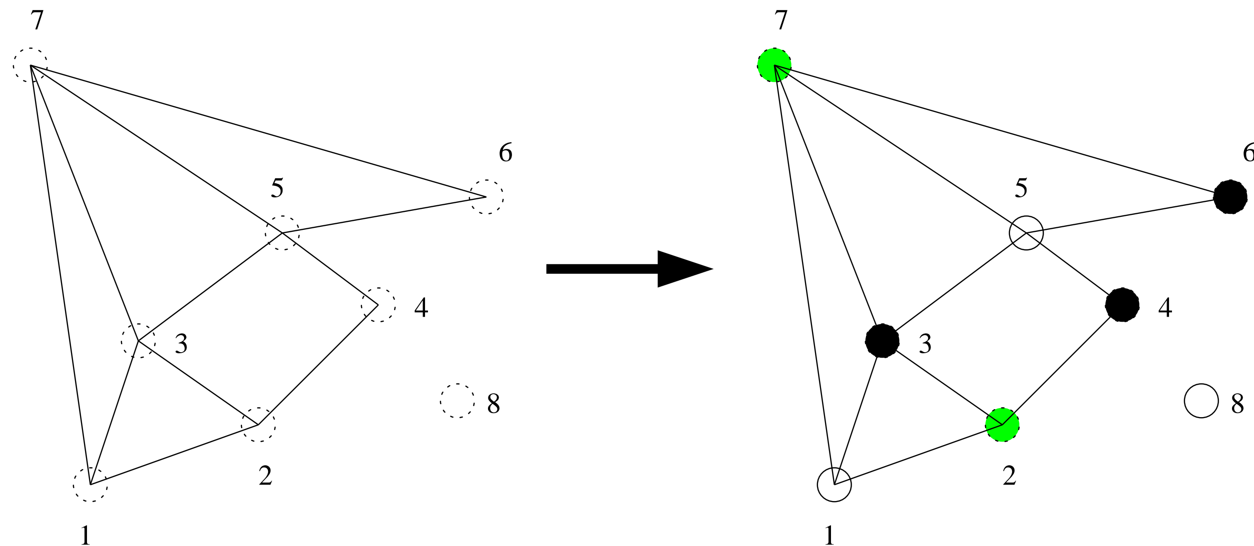
- **Unary constraints** can be removed by adjusting the domains.
- **Higher-order constraints** applying to three or more variables can certainly be considered, but...
- ...when dealing with finite domains they can always be converted to sets of binary constraints by introducing extra **auxiliary variables**.

It is also possible to introduce **preference constraints** in addition to **absolute constraints**.

We may sometimes also introduce an **objective function**.

Example

We will use the problem of colouring the nodes of a graph as an example.



We have three colours and directly connected nodes should have different colours.

Example

This translates easily to a CSP formulation:

- The variables are the nodes

$$V_i = \text{node } i$$

- The domain for each variable contains the values black, white and green (or grey on the printed handout)

$$D_i = \{B, W, G\}$$

- The constraints enforce the idea that directly connected nodes must have different colours. For example, for 1 and 2 the constraints specify

$$(B, W), (B, G), (W, B), (W, G), (G, B), (G, W)$$

Backtracking search

Consider what happens if we try to solve a CSP using a simple technique such as *breadth-first search*.

The branching factor is nd at the first step, for n variables each with d possible values.

$$\left.\begin{array}{ll}\text{Step 2:} & (n-1)d \\ \text{Step 3:} & (n-2)d \\ & \vdots \\ \text{Step } n\text{:} & 1\end{array}\right\} \quad \text{Number of leaves} = nd \times (n-1)d \times \cdots \times$$

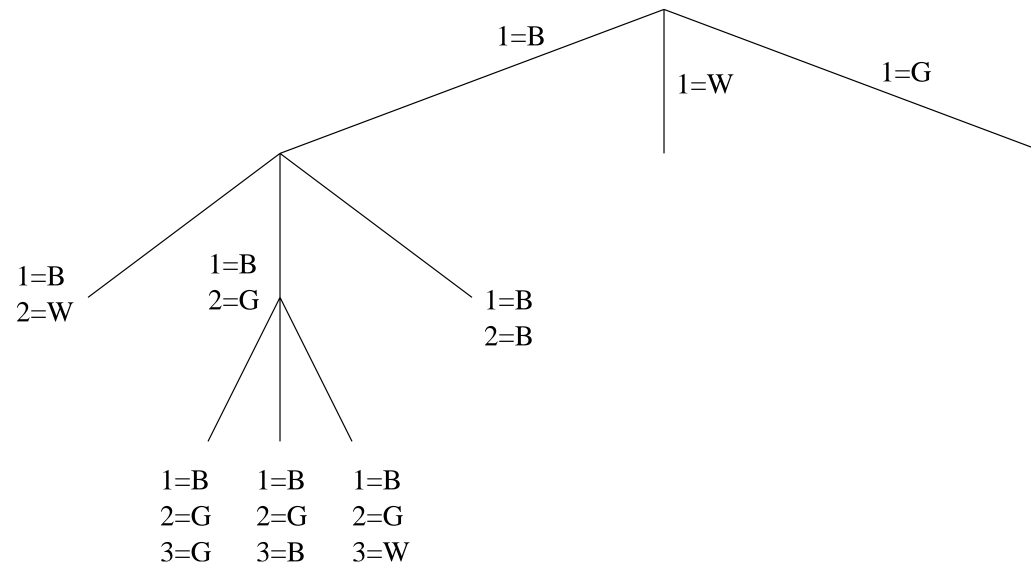
$$= n!d^n$$

BUT: only d^n assignments are possible.

The order of assignment doesn't matter, and we should assign to one variable at a time.

Backtracking search

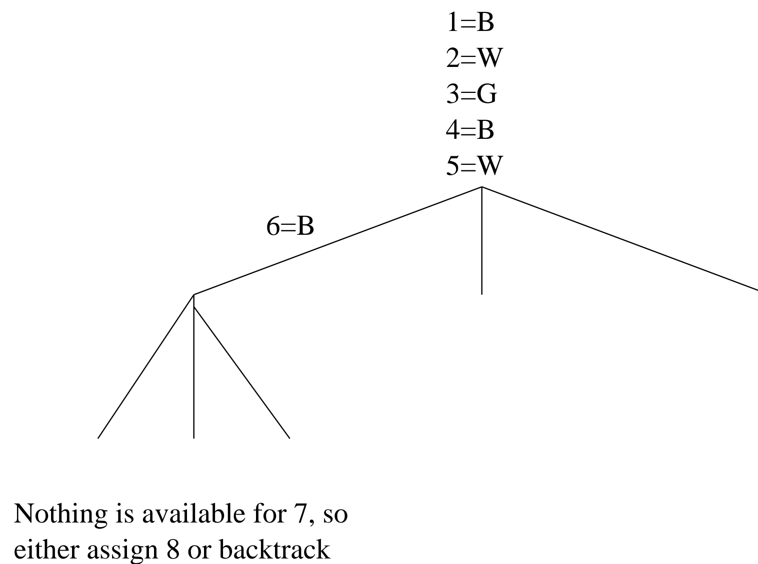
The search now looks something like this...

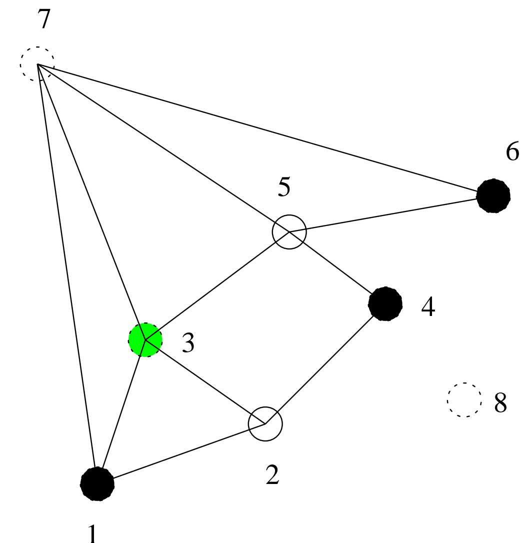


...and new possibilities appear.

Backtracking search

Backtracking search searches depth-first, assigning a single variable at a time, and backtracking if no valid assignment is available.





Rather than using problem-specific heuristics to try to improve searching, we can now explore heuristics applicable to *general* CSPs.

Backtracking search

```
result backtrack(problem)
{
  return bt ([],problem);
}

result bt(assignment_list, problem)
{
  if (assignment_list is complete)
    return assignment_list;
  next_var = get_next_var(assignment_list, problem);
  for (every value in order_variables(next_var, assignment_list, problem))
  {
    if (value is consistent with assignment_list)
    {
      add "next_var=value" to assignment_list;
      solution = bt(assignment_list, problem);
      if (solution is not "fail")
        return solution;
      remove "next_var=value" from assignment_list;
    }
  }
  return "fail";
}
```

Backtracking search: possible heuristics

There are several points we can examine in an attempt to obtain general CSP-based heuristics:

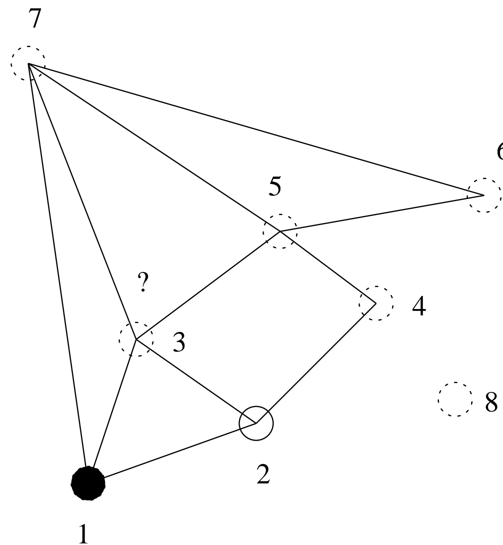
- In what order should we try to assign variables?
- In what order should we try to assign possible values to a variable?

Or being a little more subtle:

- What effect might the values assigned so far have on later attempted assignments?
- When forced to backtrack, is it possible to avoid the same failure later on?

Heuristics I: Choosing the order of variable assignments and values

Say we have $1 = B$ and $2 = W$

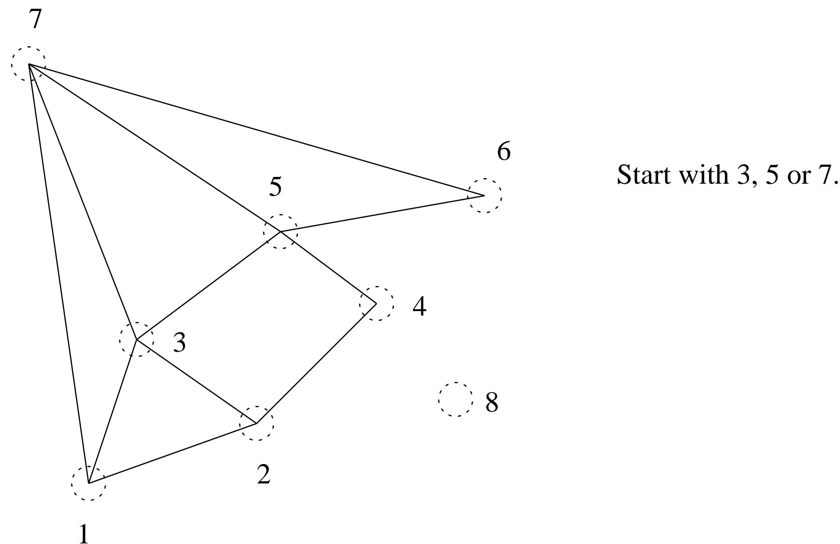


At this point there is only one possible assignment for 3, whereas the others have more flexibility. Assigning such variables **first** is called the **minimum remaining values (MRV)** heuristic. (Alternatively, the **most constrained variable** or **fail first** heuristic.

Heuristics I: Choosing the order of variable assignments and values

How do we choose a variable to begin with?

The **degree heuristic** chooses the variable involved in the most constraints on as yet unassigned variables.

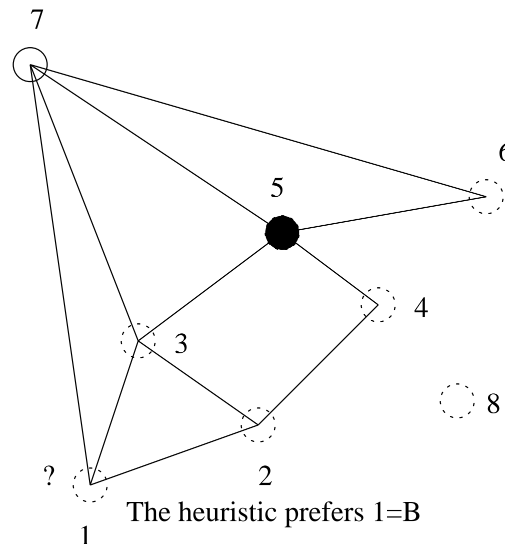


MRV is usually better but the degree heuristic is a good tie breaker.

Heuristics I: Choosing the order of variable assignments and values

Once a variable is chosen, in what order should values be assigned?

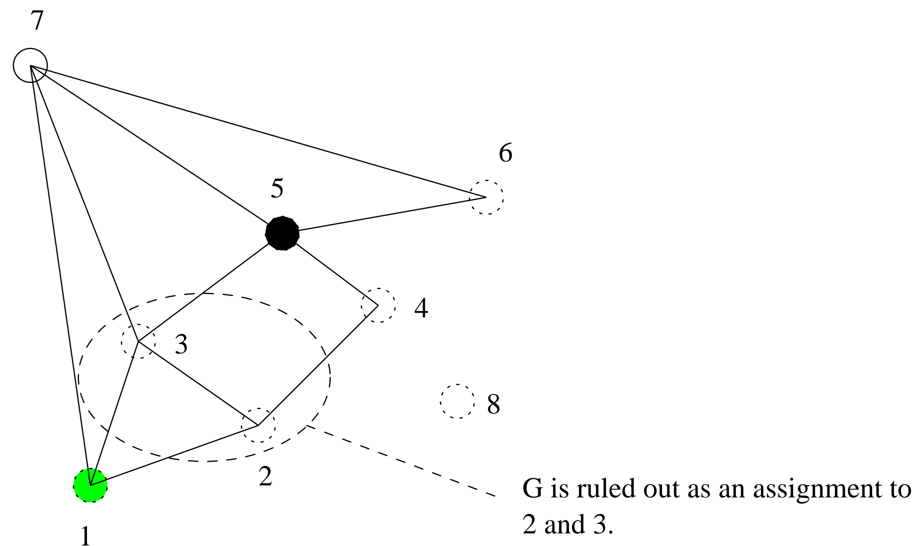
The **least constraining value** heuristic chooses first the value that leaves the maximum possible freedom in choosing assignments for the variable's neighbours.



Choosing $1 = G$ is bad as it removes the final possibility for 3.

Heuristics II: forward checking and constraint propagation

Continuing the previous slide's progress, now add $1 = G$.



Each time we assign a value to a variable, it makes sense to delete that value from the collection of **possible assignments to its neighbours**. This is called **forward checking**. It works nicely in conjunction with MRV.

Heuristics II: forward checking and constraint propagation

We can visualise this process as follows:

	1	2	3	4	5	6	7	8
Start	BWG	BWG	BWG	BWG	BWG	BWG	BWG	BWG
$2 = B$	WG	$= B$	WG	WG	BWG	BWG	BWG	BWG
$3 = W$	G	$= B$	$= W$	WG	BG	BWG	BG	BWG
$6 = B$	G	$= B$	$= W$	WG	G	$= B$	G	BWG
$5 = G$	G	$= B$	$= W$	W	$= G$	$= B$	$!$	BWG

At the fourth step, 7 has no possible assignments left.

However, we could have detected a problem a little earlier...

Heuristics II: forward checking and constraint propagation

...by looking at step three.

- At step three, 5 can be G only and 7 can be G only.
- But 5 and 7 are connected.
- So we can't progress, and this hasn't been detected.
- Ideally we want to do **constraint propagation**.

Trade-off: time to do the search, against time to explore constraints.

Constraint propagation

Arc consistency:

Consider a constraint as being **directed**. For example $4 \rightarrow 5$.

In general, say we have a constraint $i \rightarrow j$ and currently the domain of i is D_i and the domain of j is D_j.

$i \rightarrow j$ is *consistent* if

$$\forall d \in D_i, \exists d' \in D_j \text{ such that } i \rightarrow j \text{ is valid}$$

Constraint propagation

Example:

In step three of the table, $D_4 = \{W, G\}$ and $D_5 = \{G\}$.

- $5 \longrightarrow 4$ in step three of the table is consistent.
- $4 \longrightarrow 5$ in step three of the table is not consistent.

$4 \longrightarrow 5$ can be made consistent by deleting G from D_4.

Enforcing arc consistency

We can enforce arc consistency each time a variable i is assigned.

- We need to maintain a collection of arcs to be checked.
- Each time we alter a domain, we may have to include further arcs in the collection.

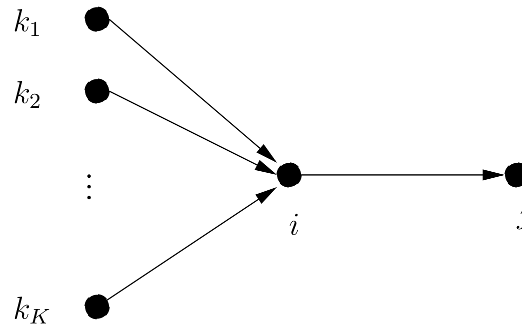
This is because if $i \rightarrow j$ is inconsistent, resulting in a deletion from D_i, we may as a consequence make some arc $k \rightarrow i$ inconsistent.

Enforcing arc consistency

Why is this?

- $i \rightarrow j$ inconsistent means removing a value from D_i.
- $\exists d \in D_i$ such that there is no valid $d' \in D_j$.
- So delete $d \in D_i$.

However some $d'' \in D_k$ may only previously have been pairable with d.



We need to continue until all consequences are taken care of.

Enforcing arc consistency

Complexity:

- A binary CSP with n variables can have $O(n^2)$ directional constraints $i \rightarrow j$.
- Any $i \rightarrow j$ can be considered at most d times where $d = \max_k |D_k|$ because only d things can be removed from D_i.
- Checking any single arc for consistency can be done in $O(d^2)$.

So the complexity is $O(n^2 d^3)$.

Note: this setup includes 3SAT.

Consequence: we can't check for consistency in polynomial time. Which suggests this doesn't guarantee to find all inconsistencies.

The AC-3 algorithm

```
new_domains AC-3 (problem)
{
  queue to_check = all arcs i->j;
  while (to_check is not empty)
  {
    i->j = next(to_check);
    if (remove_inconsistencies(Di,Dj))
    {
      for (each k that is a neighbour of i)
        add k->i to to_check;
    }
  }
}
```

The AC-3 algorithm

```
bool remove_inconsistencies (domain1, domain2)
{
  bool result = false;
  for (each d in domain1)
  {
    if (no d’ in domain2 valid with d)
    {
      remove d from domain1;
      result = true;
    }
  }
  return result;
}
```

A more powerful form of consistency

We can define a stronger notion of consistency as follows:

Given:

- Any $k-1$ variables and,
- any consistent assignment to these.

Then:

- We can find a consistent assignment to any kth variable.

This is known as k-*consistency*.

A more powerful form of consistency

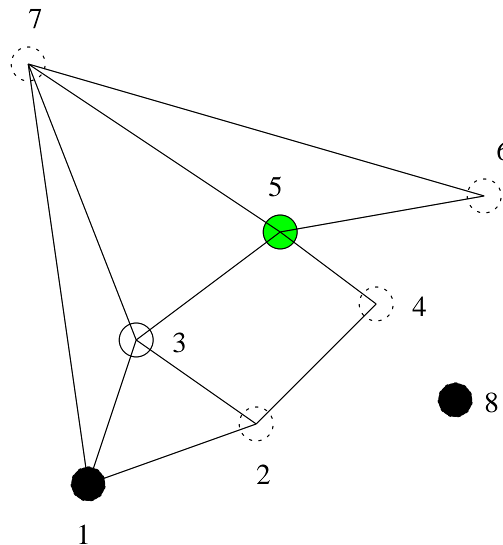
Strong k*-consistency* requires the we be k-consistent, $k-1$-consistent *etc* as far down as 1-consistent.

If we can demonstrate strong n-consistency (where as usual n is the number of variables) then an assignment can be found in $O(nd)$.

Unfortunately, demonstrating strong n-consistency will be worst-case exponential.

Backjumping I

The basic backtracking algorithm backtracks to the most recent assignment. This is known as **chronological backtracking**. It is not always the best policy:



Say we've done $1 = B$, $3 = W$, $5 = G$ and $8 = B$ and now we want to do 7. This isn't possible so we backtrack, however re-assigning 8 clearly doesn't help.

Backjumping I

Backjumping backtracks to the **conflict set**, which in this case is $\{1, 3, 5\}$:

conflict$(x) =$ set of currently assigned variables connected to x

This can be done by accumulating the sets conflict(x) as we make assignments.

Backjumping I

If forward checking is in operation it can be used to find conflict sets.

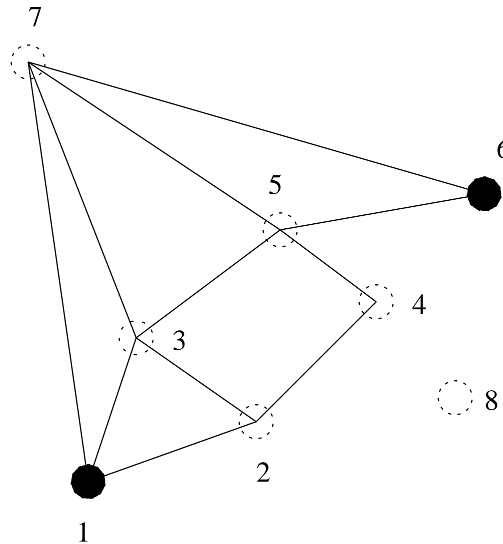
Say we're assigning to x, say $x = v$:

- Forward checking removes v from the D_i of all x_i connected to x.
- Then x needs to be added to conflict(x_i).
- If the *last* member of D_i is ever removed then we need to add *all* of conflict(x_i) to conflict(x).

In fact, use of forward checking turns out to make backjumping redundant.

Backjumping II

In the current example, only two assignments are needed to doom the process:



Next we can assign $8, 3, 7$ and 4, but then 5 fails.

This can never work because 1 and 6 prevent us from getting an assignment for $3, 7, 4$ and 5.

Backjumping II

In this example $\{3, 7, 4, 5\}$ as a *collection* are prevented by 1 and 6 from having an assignment.

We can redefine conflict(x) to be the collection of preceding variables causing x and any subsequent variables not to have a valid set of assignments.

Using the new concept for conflict(x) gives us **conflict-directed backjumping**:

When backtracking from x' to x:

$$\text{conflict}(x) = \text{conflict}(x) \cup (\text{conflict}(x') - x)$$

so that the causes of failure *after* x are maintained.