Artificial Intelligence I

Dr Sean Holden

Notes on *problem solving by search*

Problem solving by search

We begin with what is perhaps the simplest collection of AI techniques: those allowing an *agent* existing within an *environment* to *search* for a *sequence of actions* that *achieves a goal*.

The algorithms can, crudely, be divided into two kinds: *uninformed* and *informed*.

Not surprisingly, the latter are more effective and so we'll look at those in more detail.

Reading: Russell and Norvig, chapters 3 and 4.

Problem solving by search

As with any area of computer science, some degree of *abstraction* is necessary when designing AI algorithms.

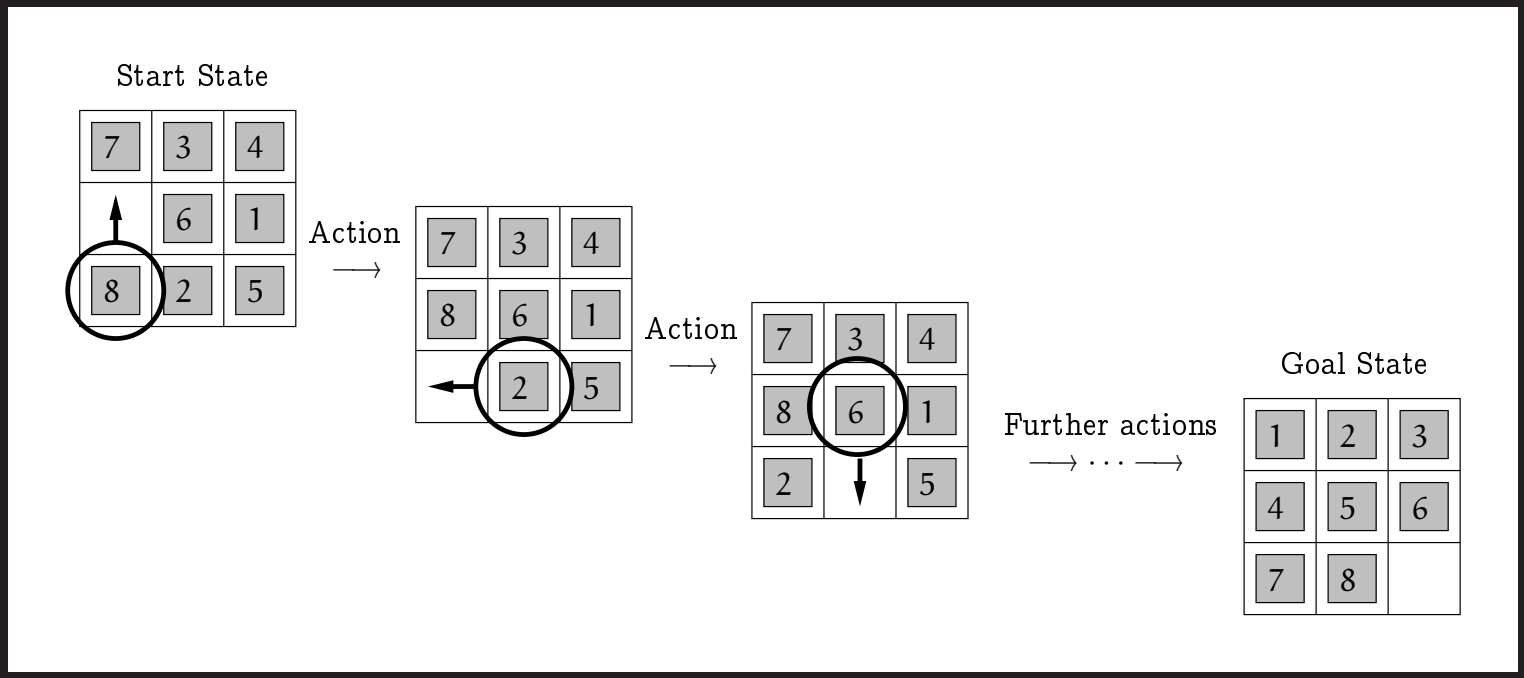
Search algorithms apply to a particularly simple class of problems—we need to identify:

- *An initial state*: what is the agent's situation to start with?
- *A set of actions*: these are modelled by specifying 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 a goal.

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

Problem solving by search

A simple example: *the 8-puzzle*.



(A good way of keeping kids quiet...)

Problem solving by 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.

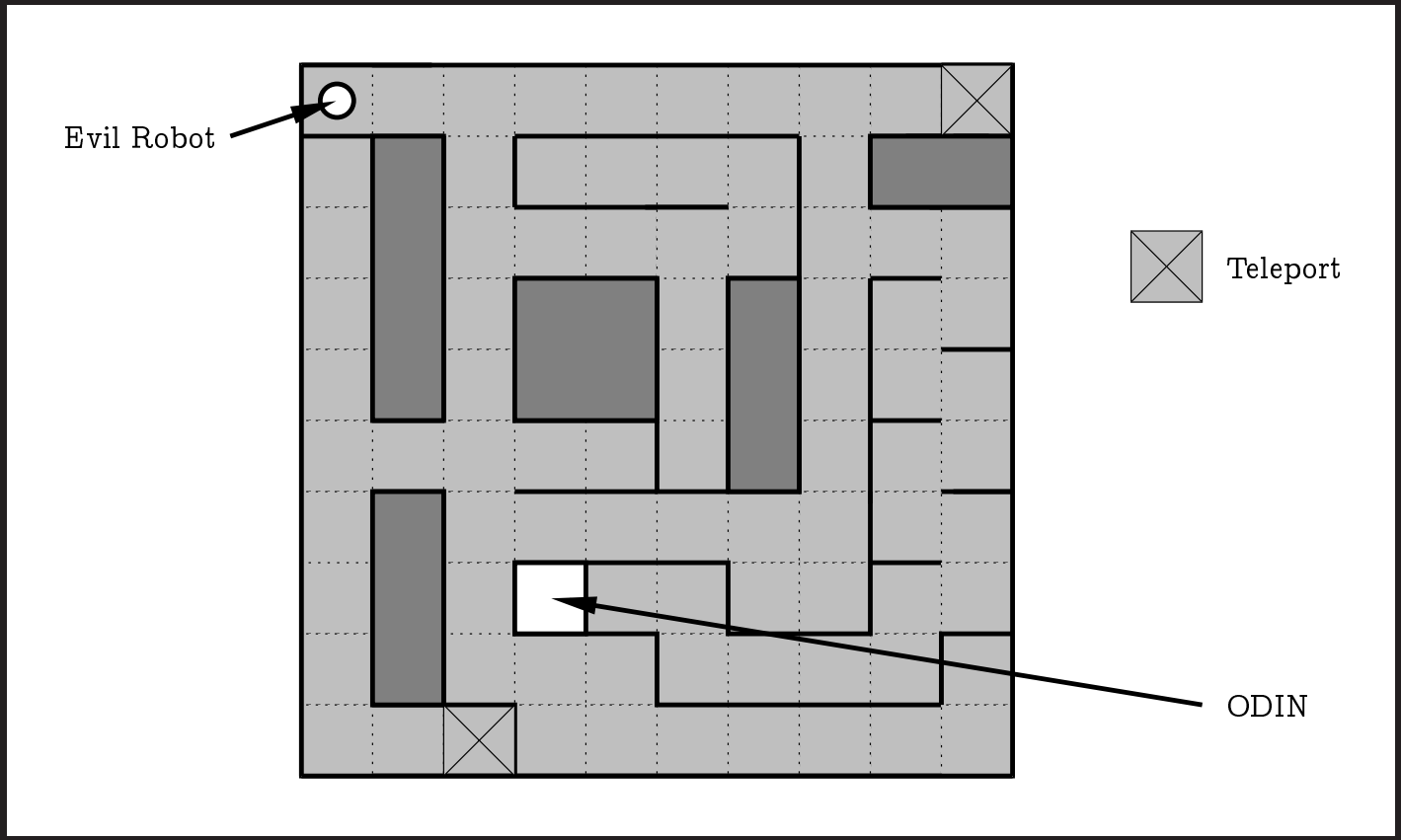
Actions: `left`, `right`, `up`, `down`. We can move any square adjacent to the empty square into the empty square. (It's not always possible to choose from all four actions.)

Path cost: 1 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.

Problem solving by basic search

EVIL ROBOT has found himself in an unfamiliar building:



He wants the *ODIN (Oblivion Device of Indescribable Nastiness)*.

Problem solving by search

Start state: EVIL ROBOT is in the top left corner.

Goal state: EVIL ROBOT is in the area containing the ODIN.

Actions: `left`, `right`, `up`, `down`. We can move as long as there's no wall in the way. (Again, it's not always possible to choose from all four actions.)

Path cost: 1 per move. If you step on a teleport then you move to the other one with a cost of 0.

Problem solving by search

Problems of this kind are very simple, but a surprisingly large number of applications have appeared:

- 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...

Problems of this kind continue to form an active research area.

Problem solving by search

It's worth emphasising that a lot of abstraction has taken place here:

- Can the agent know it's current state in full?
- Can the agent know the outcome of its actions in full?

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 effects of actions are known, but the state can not reliably be inferred, or the state is known but not the effects of the actions.

Problem solving by search

Single and multiple state problems can be handled using these search techniques.

In the latter, we must reason about the set of states that we could be in:

- 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.

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 will be dealt with later.

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.)

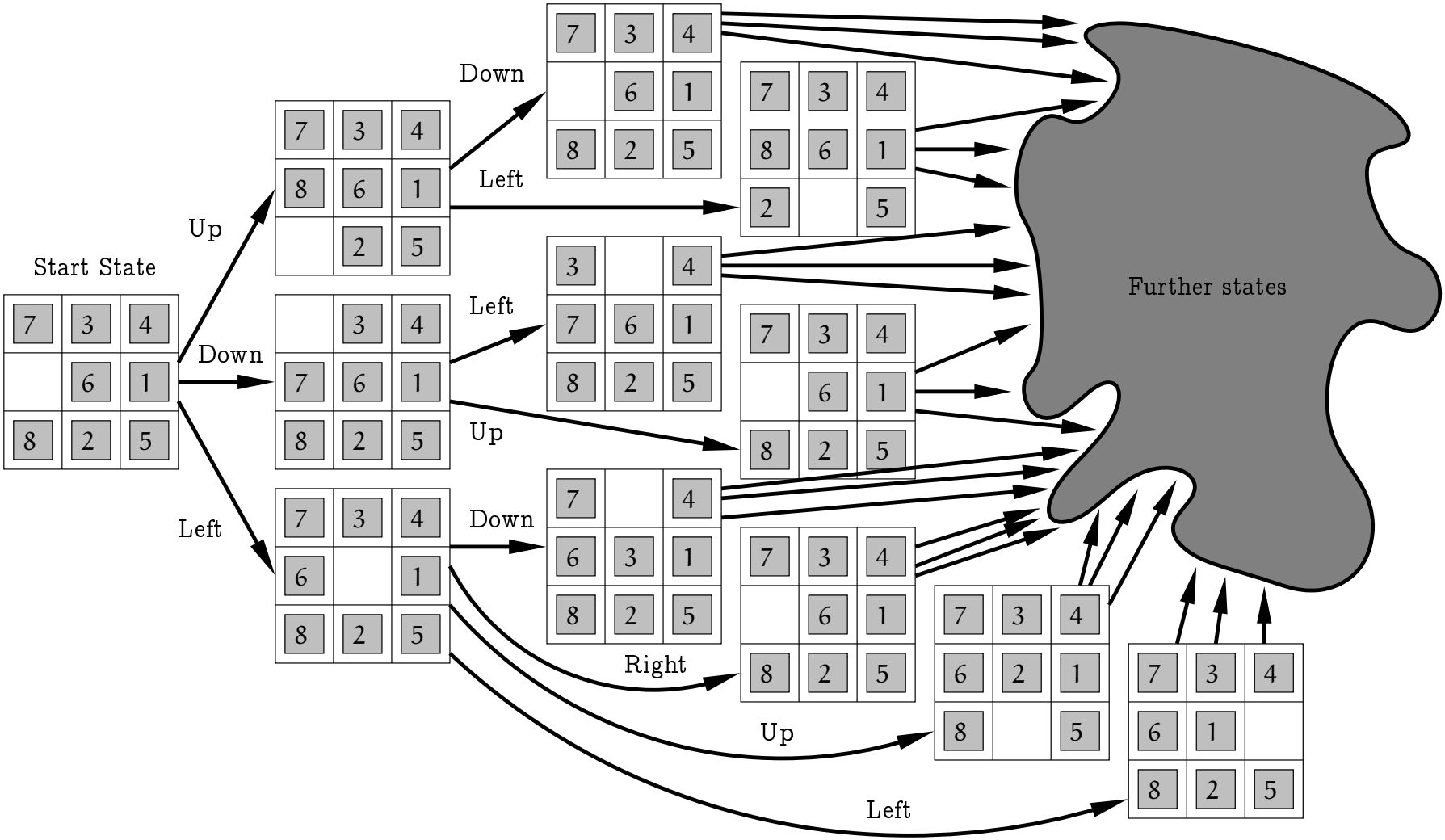
Search trees

The basic idea should be familiar from your (current) *Algorithms I* course, and also from *Foundations of Computer Science*.

- We build a *tree* with the start state as root node.
- A node is *expanded* by applying actions to it to generate new states.
- A *path* is a *sequence of actions* that lead from state to state.
- The aim is to find a *goal state* within the tree.
- A *solution* is a path beginning with the initial state and ending in a goal state.

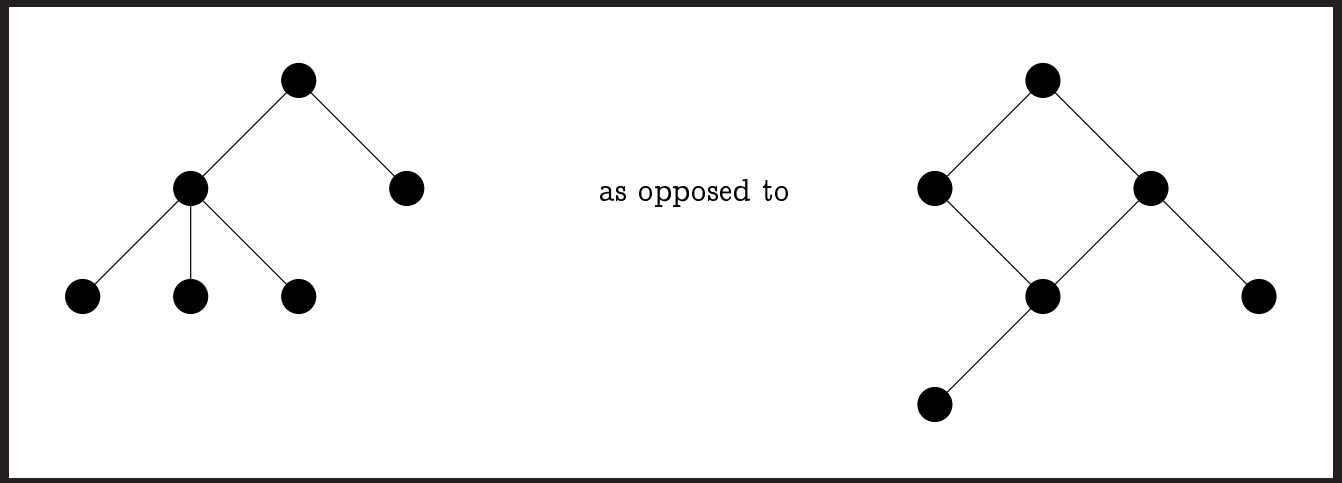
We may also be interested in the *path cost* as some solutions might be better than others.

Path cost will be denoted by p.


Start State
Up
Down
Left
Down
Left
Up
Left
Down
Right
Up
Left
Further states

Search trees versus search graphs

We need to make an important distinction between *search trees* and *search graphs*. For the time being we assume that it's a *tree* as opposed to a *graph* that we're dealing with.



(There is a good reason for this, which we'll get to in a moment...)

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

Search trees

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*.
- Repeat until you find a goal.

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

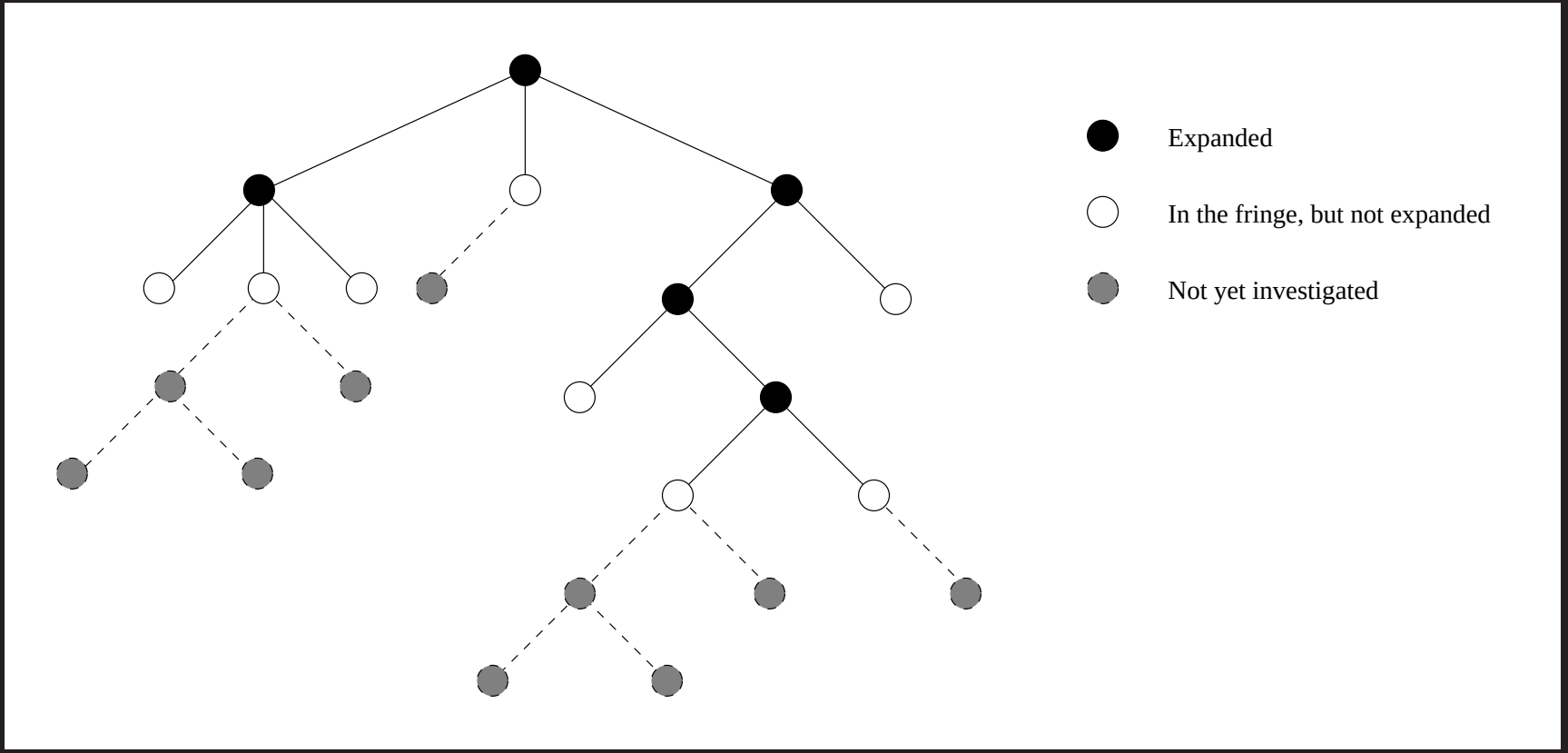
In pseudo-code, the algorithm looks like this:

```
function treeSearch
{
  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 *search strategy* is set by using a *priority queue*.

The definition of *priority* then sets the way in which the tree is searched.

The basic tree-search algorithm



The basic tree-search algorithm

We can immediately define some familiar tree search algorithms:

- New nodes are added to the *head of the queue*. This is *depth-first search*.
- New nodes are added to the *tail of the queue*. This is *breadth-first search*.

We will not dwell on these, as they are both *completely hopeless* in practice.

Why is that?

The performance of search techniques

How might we judge the performance of a search technique?

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*.

Evaluation of search strategies

We are also interested in:

Completeness: does the strategy *guarantee* a solution is found?

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

Once we start to consider these, things get a lot more interesting...

Breadth-first search

Why is breadth-first search hopeless?

- The procedure is *complete*: it is guaranteed to find a solution if one exists.
- The procedure is *optimal* if the path cost is a non-decreasing function of node-depth. (Exercise: why is this?)
- The procedure has *exponential complexity for both memory and time*. A branching factor b requires

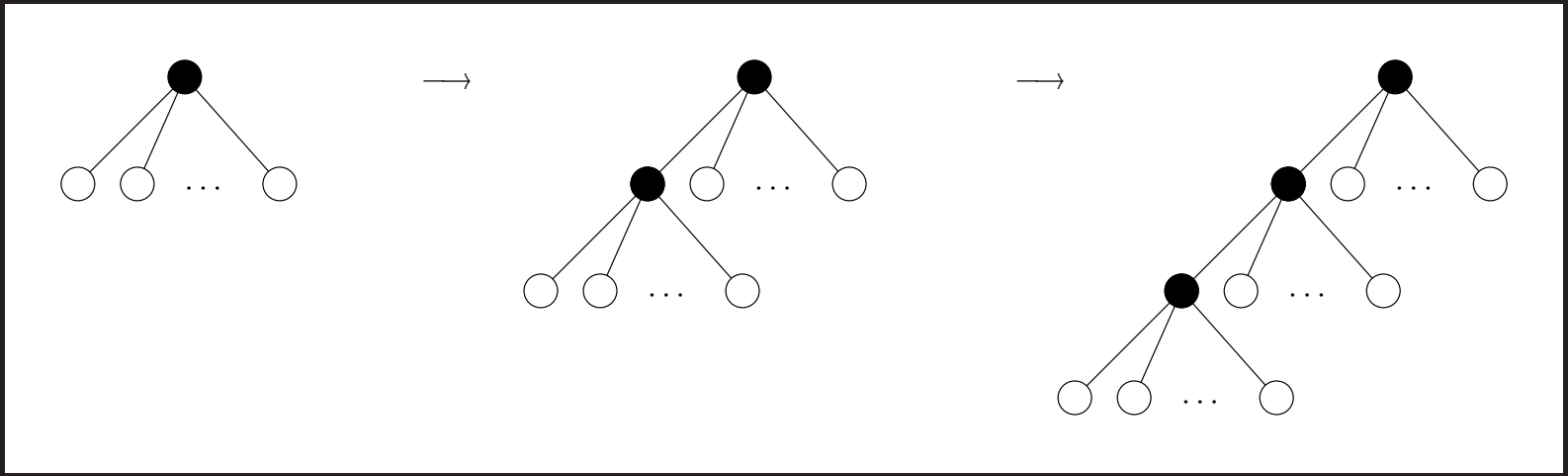
$$1 + b + b^2 + b^3 + \cdots + b^d = \frac{b^{d+1} - 1}{b - 1}$$

nodes if the shortest path has depth d.

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

Depth-first search

With depth-first search: for a given branching factor b and depth d the memory requirement is $O(bd)$.



This is because we need to store *nodes on the current path* and *the other unexpanded nodes*.

The time complexity is $O(b^d)$. Despite this, if there are *many solutions* we stand a chance of finding one quickly, compared with breadth-first search.

Backtracking search

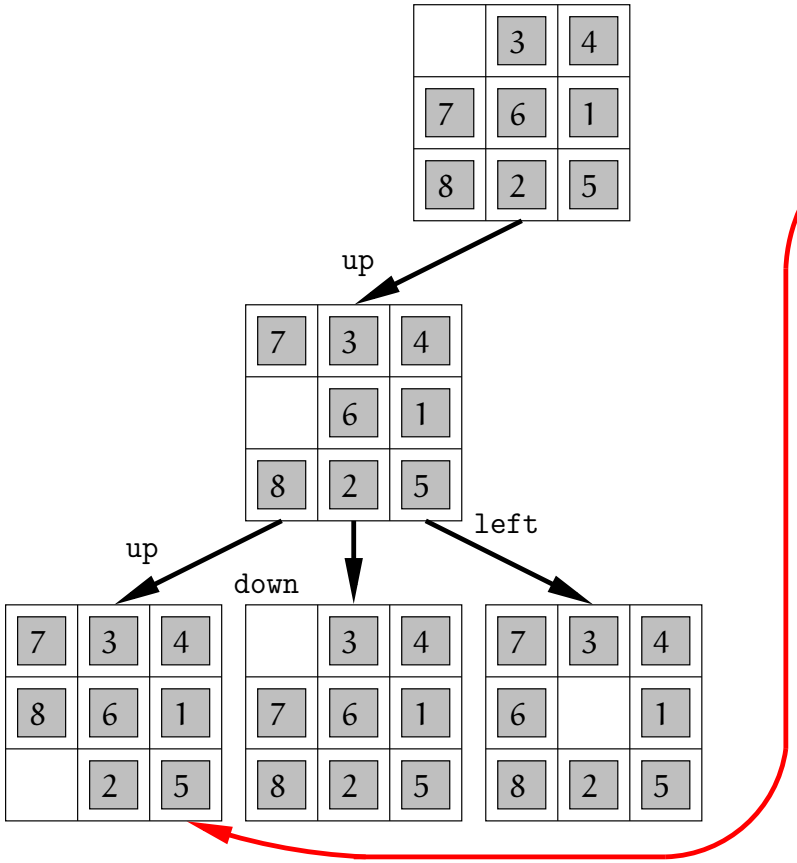
We can sometimes improve on depth-first search by using *backtracking search*.

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

How does this work?

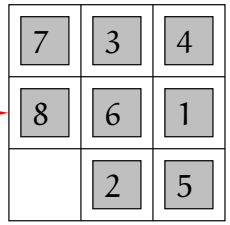
No backtracking

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



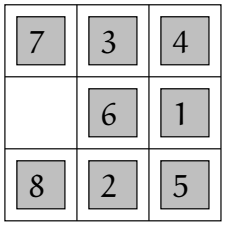
With backtracking

If we have:



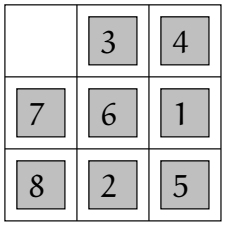
+ `[up, up]`

we can undo this to obtain



+ `[up]`

and apply `down` to get



+ `[up, down]`

and so on...

Depth-first, depth-limited, and iterative deepening search

Depth-first search is clearly dangerous if the tree is *very deep or infinite*.

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.

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

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

Iterative deepening search

Iterative deepening search:

- Essentially combines the advantages of depth-first and breadth-first search.
- It is complete and optimal.
- It has a memory requirement 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.

It's *still* not a good practical method, but it does point us in the direction of one...

Iterative deepening search

Iterative deepening depends on the fact that *the vast majority of the nodes in a tree are in the bottom level*:

- In a tree with branching factor b and depth d the number of nodes is
$$f_1(b, d) = 1 + b + b^2 + b^3 + \cdots + b^d = \frac{b^{d+1} - 1}{b - 1}$$
- 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 $d + 1$ times. The total number of nodes generated is therefore
$$f_2(b, d) = (d + 1) + db + (d - 1)b^2 + (d - 2)b^3 + \cdots + 2b^{d-1} + b^d$$

Iterative deepening search

Example:

- For $b = 20$ and $d = 5$ we have

$$f_1(b, d) = 3,368,421$$

$$f_2(b, d) = 3,545,706$$

  which represents a 5 percent increase with iterative deepening search.

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

For problems where the search space is large and the solution depth is not known, this can be a good method.

Iterative deepening search

Further insight can be gained if we note that

$$f_2(b,d) = f_1(b,0) + f_1(b,1) + \cdots + f_1(b,d)$$

as we generate the root, then the tree to depth 1, and so on. Thus

$$f_2(b,d) = \sum_{i=0}^{d} f_1(b,i) = \sum_{i=0}^{d} \frac{b^{i+1}-1}{b-1}$$

$$= \frac{1}{b-1}\sum_{i=0}^{d} b^{i+1} - 1 = \frac{1}{b-1}\left[\left(\sum_{i=0}^{d} b^{i+1}\right) - (d+1)\right]$$

Noting that

$$bf_1(b,d) = b + b^2 + \cdots + b^{d+1} = \sum_{i=0}^{d} b^{i+1}$$

we have

$$f_2(b,d) = \frac{b}{b-1} f_1(b,d) - \frac{d+1}{b-1}$$

so $f_2(b,d)$ is about equal to $f_1(b,d)$ for large b.

Bidirectional search

In some problems 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 $O(b^d)$ then...
- ...we are converting this to $O(2b^{d/2}) = O(b^{d/2})$.

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

Bidirectional search - beware!

- 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*...
- ... and the figure of $O(b^{d/2})$ hides the assumption that we can do *constant time* checking for intersection of the frontiers. (This may be possible using a hash table).
- We need to decide what kind of search to use in each half. For example, would *depth-first search* be sensible? Possibly not...
- ...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 $O(b^{d/2})$.

Uniform-cost search

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

Uniform-cost search is a variant. It uses the *path cost* $p(n)$ as the priority for the priority queue.

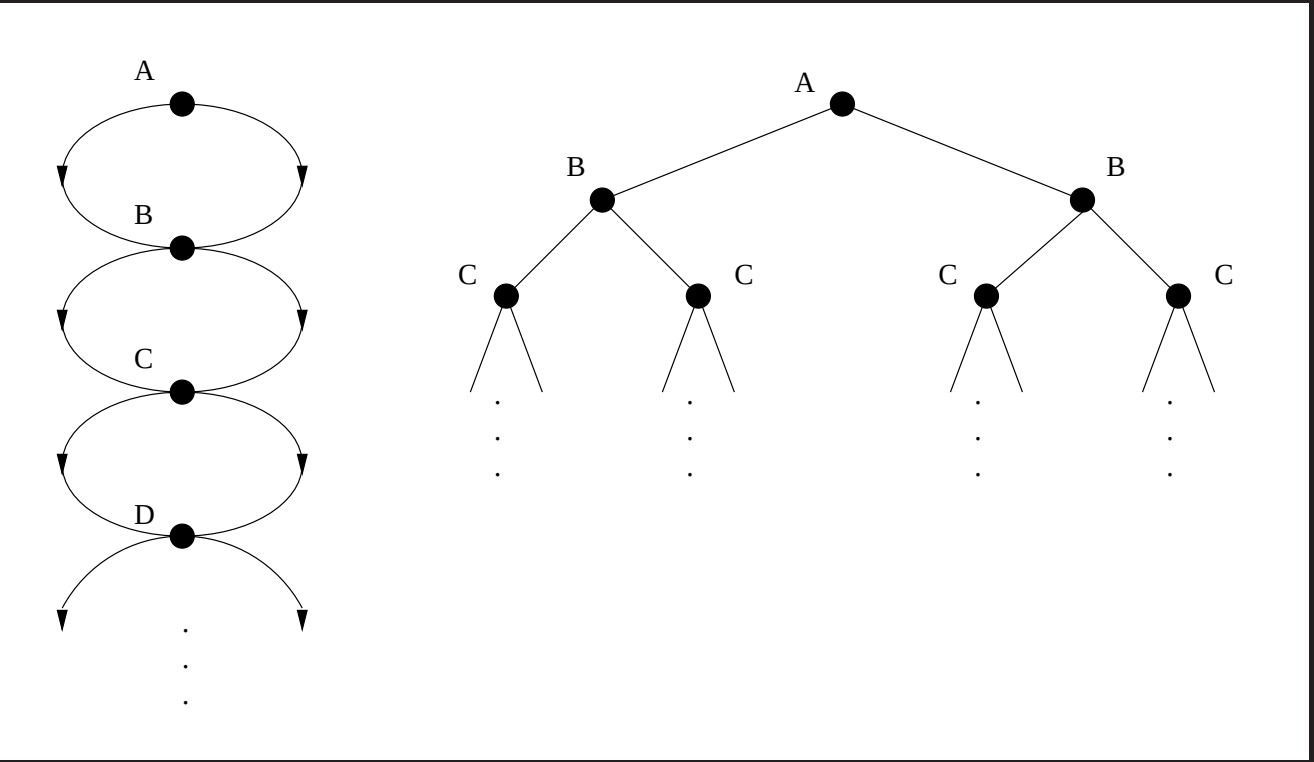
Thus, the paths that are apparently best are explored first, and the best solution will always be found if

$$\forall n \, (\forall n' \in \text{successors}(n) \, . \, p(n') \geq p(n))$$

Although this is still not a good practical algorithm, it does point the way forward to *informed search...*

Repeated states

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



The sliding blocks puzzle for example suffers this way.

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 expand *any state that has previously appeared*.

Graph search is a standard approach to dealing with the situation. It uses the last of these possibilities.

Graph search

In pseudocode:

```
function graphSearch()
{
  closed = {};
  fringe = queue containing only the 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.

Search trees

Everything we've seen so far is an example of *uninformed* or *blind* search—we only distinguish goal states from non-goal states.

(Uniform cost search is a slight anomaly as it uses the path cost as a guide.)

To perform well in practice we need to employ *informed* or *heuristic* search.

This involves exploiting knowledge of the *distance between the current state and a goal*.

Problem solving by informed search

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

- We have already seen the concept of *path cost* $p(n)$

  $p(n) =$ cost of path (sequence of actions) from the start state to n

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

Greedy search

We've already seen *path cost* used for this purpose.

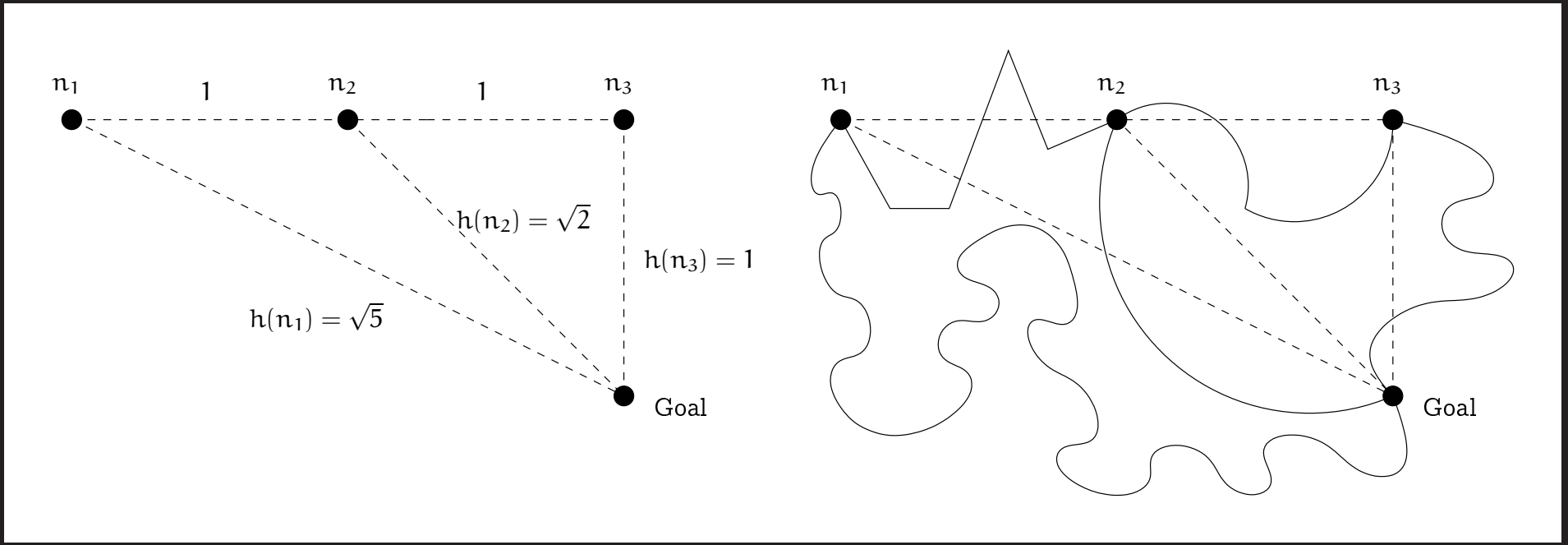
- This is misguided as path cost is not in general *directed* in any sense *toward the goal*.
- A *heuristic function*, usually denoted $h(n)$ is one that *estimates* the cost of the best path from any node n to a goal.
- If n is a goal then $h(n) = 0$.

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

Example: route-finding

Example: for route finding a reasonable heuristic function is

$h(n) =$ straight line distance from n to the nearest goal



Accuracy here obviously depends on what the roads are really like.

Example: route-finding

Greedy search suffers from some problems:

- Its time complexity is $O(b^d)$.
- Its space-complexity is $O(b^d)$.
- It is not optimal or complete.

BUT: greedy search *can* be effective, provided we have a good $h(n)$.

Wouldn't it be nice if we could improve it to make it optimal and complete?

$A^\star$ search

Well, we can.

$A^\star$ 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

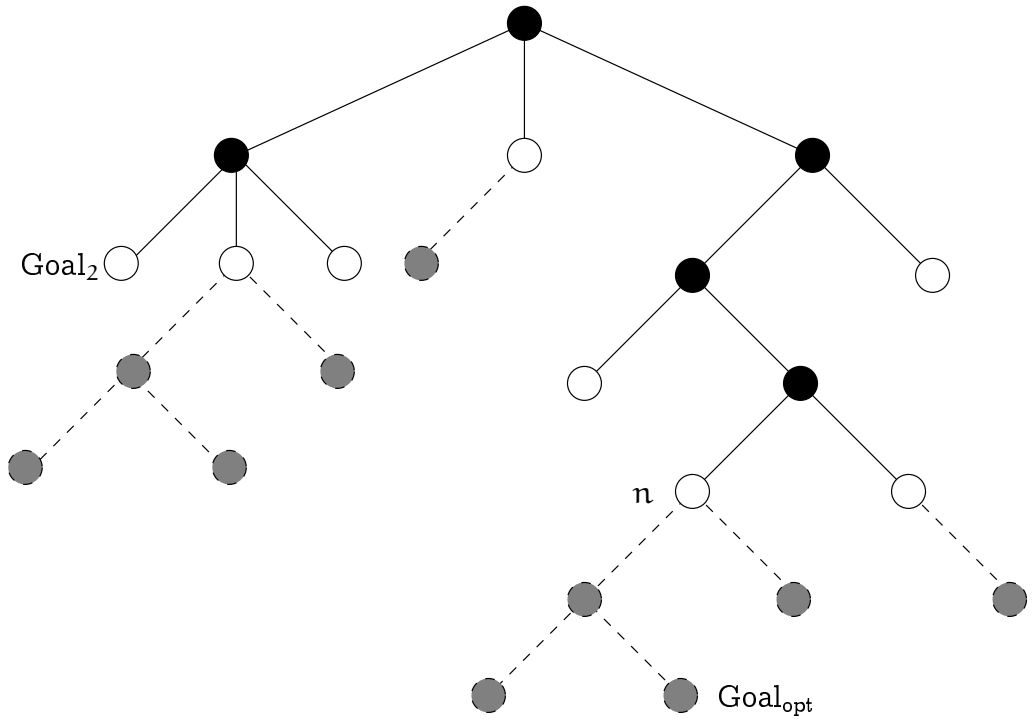
$$f(\text{Goal}_{\text{opt}}) = p(\text{Goal}_{\text{opt}}) = f_{\text{opt}}$$

(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 $h(n)$



At some point Goal_2 is in the fringe.

Can it be selected before n?

$A^\star$ 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.

$A^\star$ 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*.

The required condition is called *monotonicity*. As

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

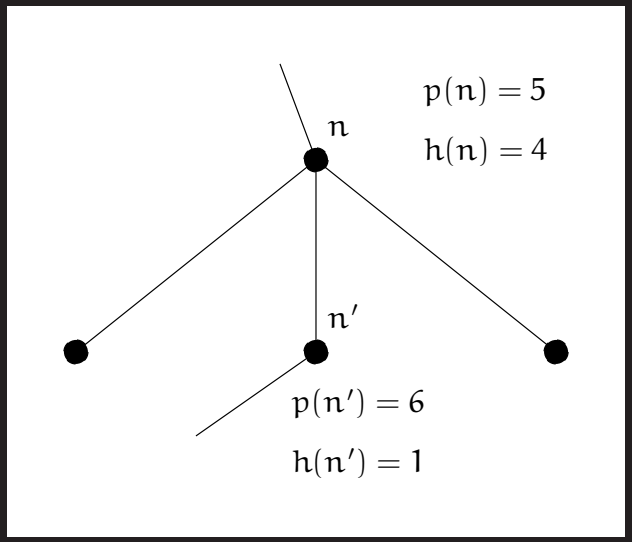
this is an important property.

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.



Here $f(n) = 9$ and $f(n') = 7$ so $f(n') < f(n)$.

Monotonicity

Monotonicity:

- 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*.

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

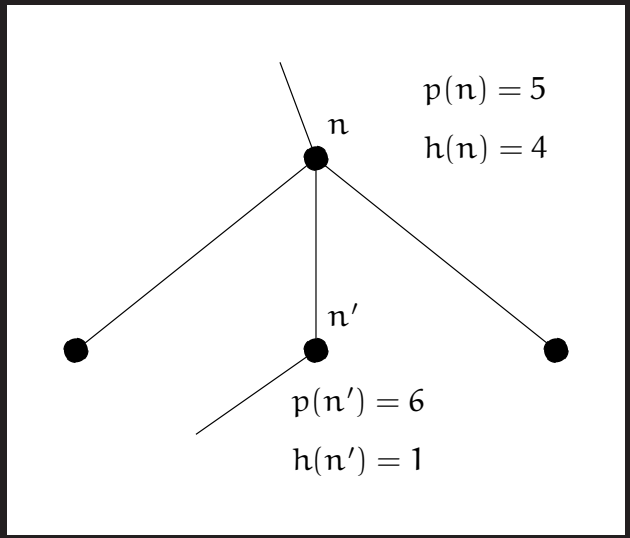
If $h(n)$ is *not* monotonic we can make a simple alteration and use

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

This is called the *pathmax* equation.

The pathmax equation

Why does the pathmax equation make sense?



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.

$A^\star$ 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

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

and using the triangle inequality

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

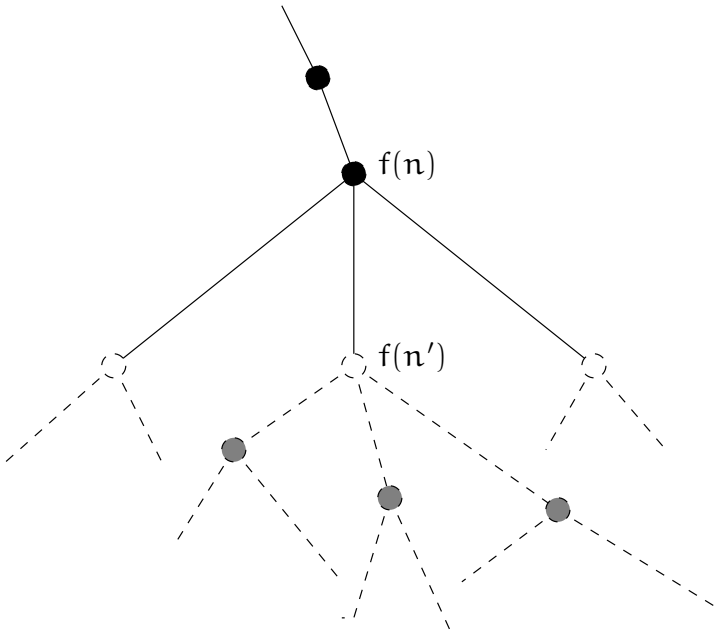
Thus

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

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 $f(n'') < f_{opt}$ 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

How might we improve the way in which $A^\star$ search uses memory?

- 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*.

```
ActionSequence ida()
{
   float fLimit = f(root);
   root = root node for problem;
   while()
   {
      (sequence, fLimit) = contour(root,fLimit,emptySequence);
      if (sequence != emptySequence)
        return sequence;
      if (fLimit == infinity)
        return emptySequence;
   }
}
```

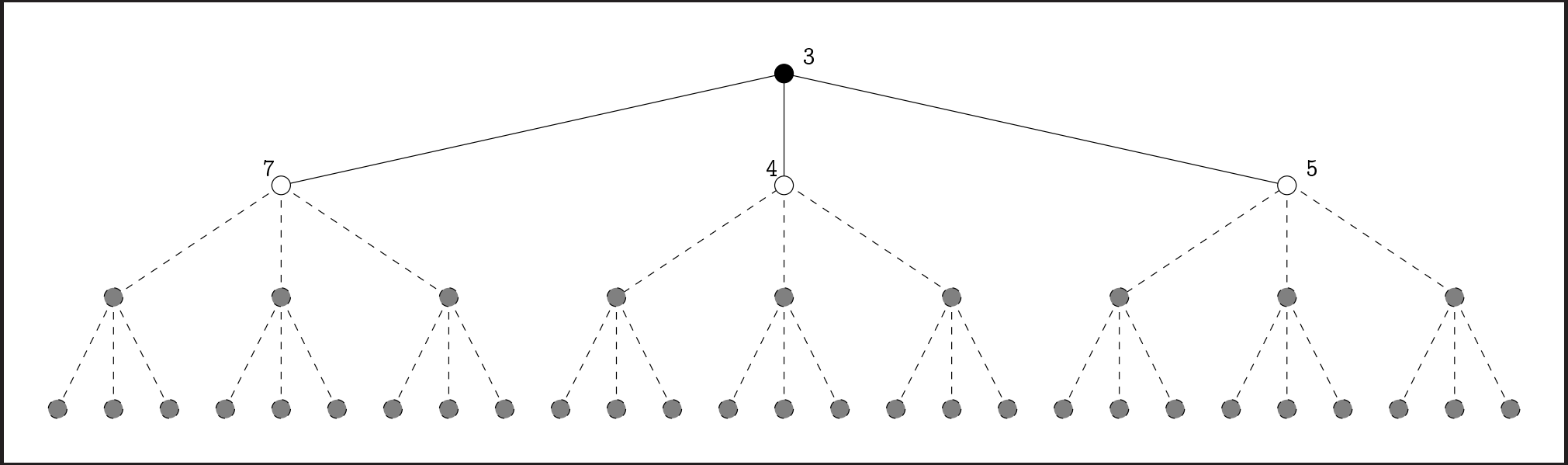
IDA* - iterative deepening $A^\star$ search

The function contour searches from a given node, *as far as the specified f limit*. It returns either a solution, or the *next biggest* value of f to try.

```
(ActionSequence,Float) contour(Node node, Float fLimit, ActionSequence s)
{
   Float nextF = infinity;
   if (f(node) > fLimit)
      return (emptySequence,f(node));
   ActionSequence s' = addToSequence(node,s);
   if (goalTest(node))
      return (s',fLimit);
   for (each successor n' of node)
   {
      (sequence,newF) = contour(n',fLimit,s');
      if (sequence != emptySequence)
         return (sequence,fLimit);
      nextF = minimum(nextF,newF);
   }
   return (emptySequence,nextF);
}
```

IDA* - iterative deepening A* search

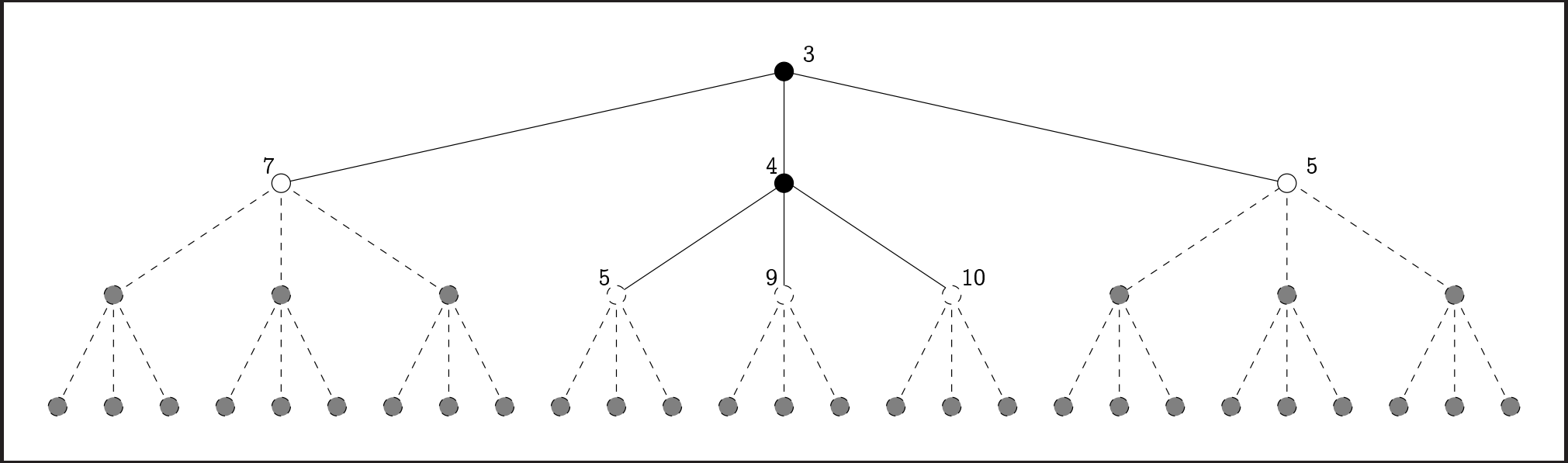
This is a little tricky to unravel, so here is an example:



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

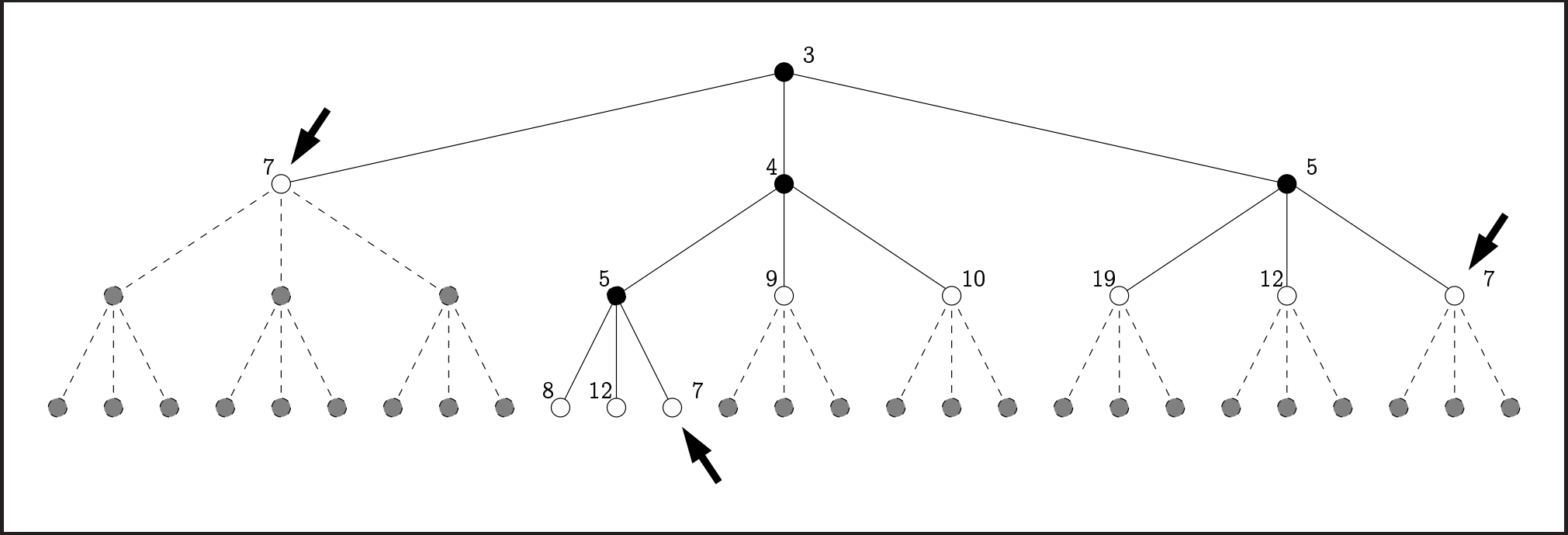
It now does the same again:



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.

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

Properties of IDA*:

- It is complete and optimal under the same conditions as $A^\star$.
- It is often good if we have step costs equal to 1.
- It 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.

Recursive best-first search (RBFS)

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

Idea: try to do a best-first search, but only use *linear space* by doing a depth-first search with a few modifications:

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've 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)

Note: for simplicity a parameter for the path has been omitted.

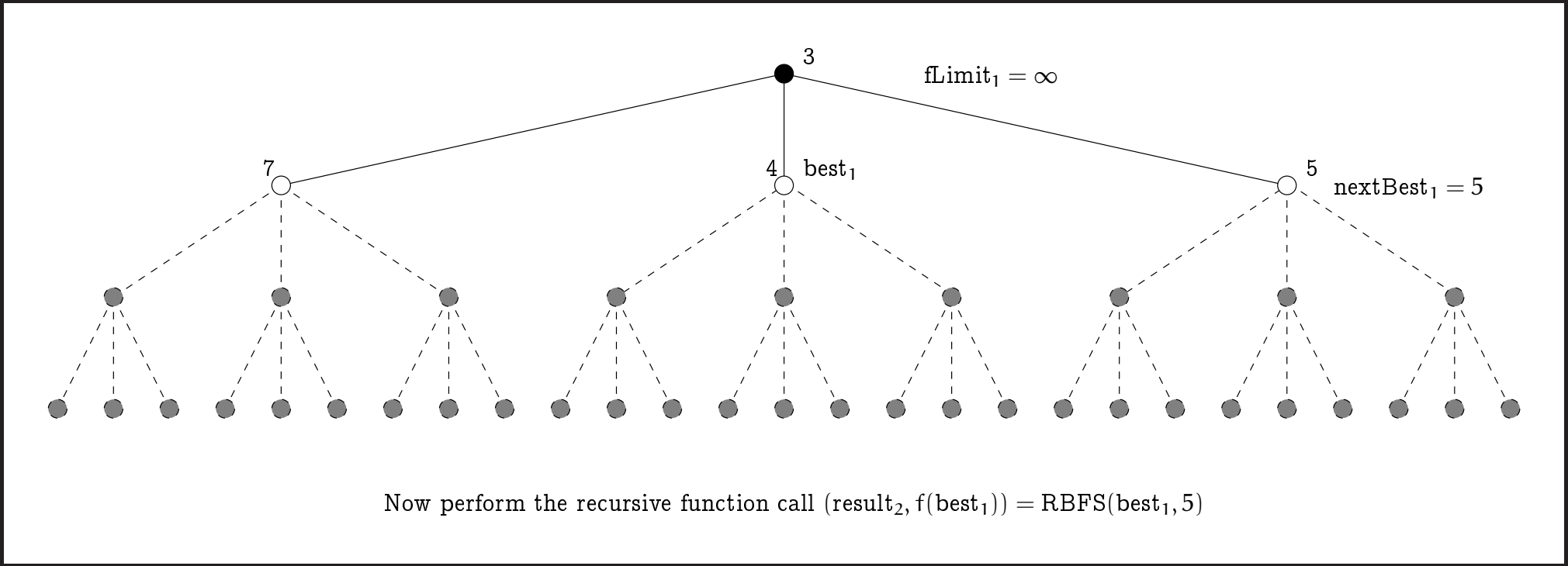
```
function RBFS(Node n, Float fLimit) {
  if (goaltest(n))
    return n;
  if (n has no successors)
    return (fail, infinity);
  for (each successor n' of n)
    f(n') = maximum(f(n'), f(n));
  while() {
    best = successor of n that has the smallest f(n');
    if (f(best) > fLimit)
      return (fail, f(best));
    nextBest = second smallest f(n') value for successors of n;
    (result, f(best)) = RBFS(best, minimum(fLimit, nextBest));
    if (result != fail)
      return result;
  }
}
```

IMPORTANT: `f(best)` is *modified* when RBFS produces a result.

Recursive best-first search (RBFS): an example

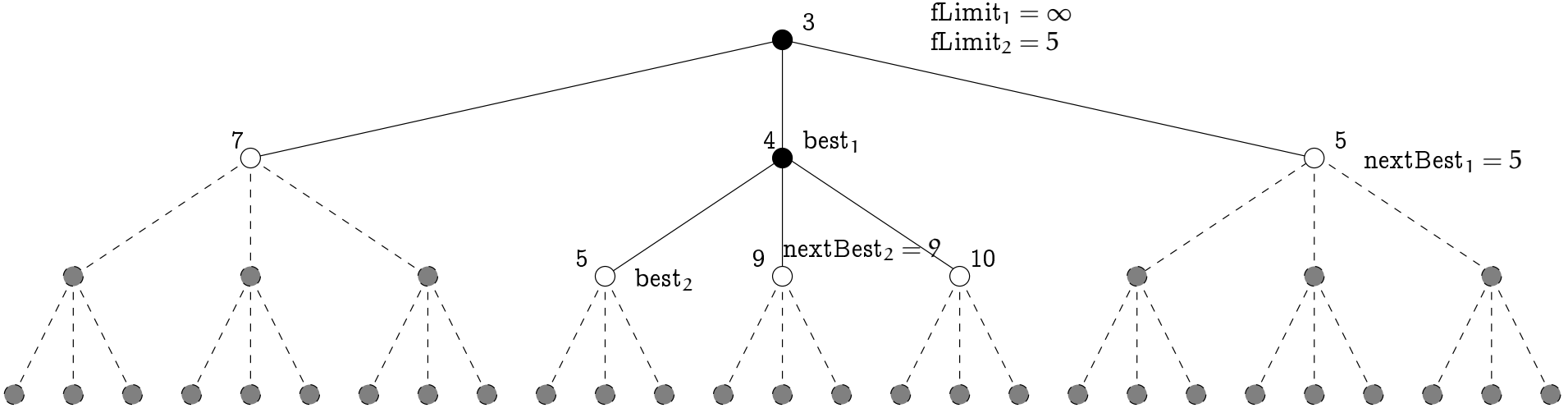
This function is called using `RBFS(startState, infinity)` to begin the process.

Function call number 1:



Recursive best-first search (RBFS): an example

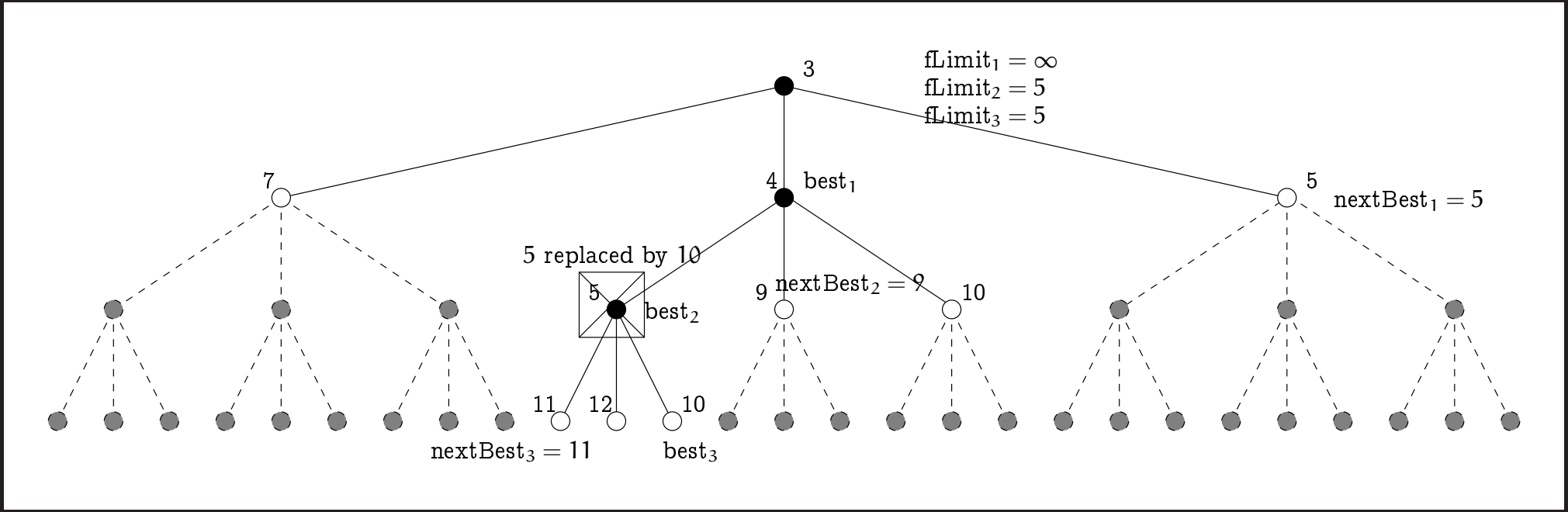
Function call number 2:



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

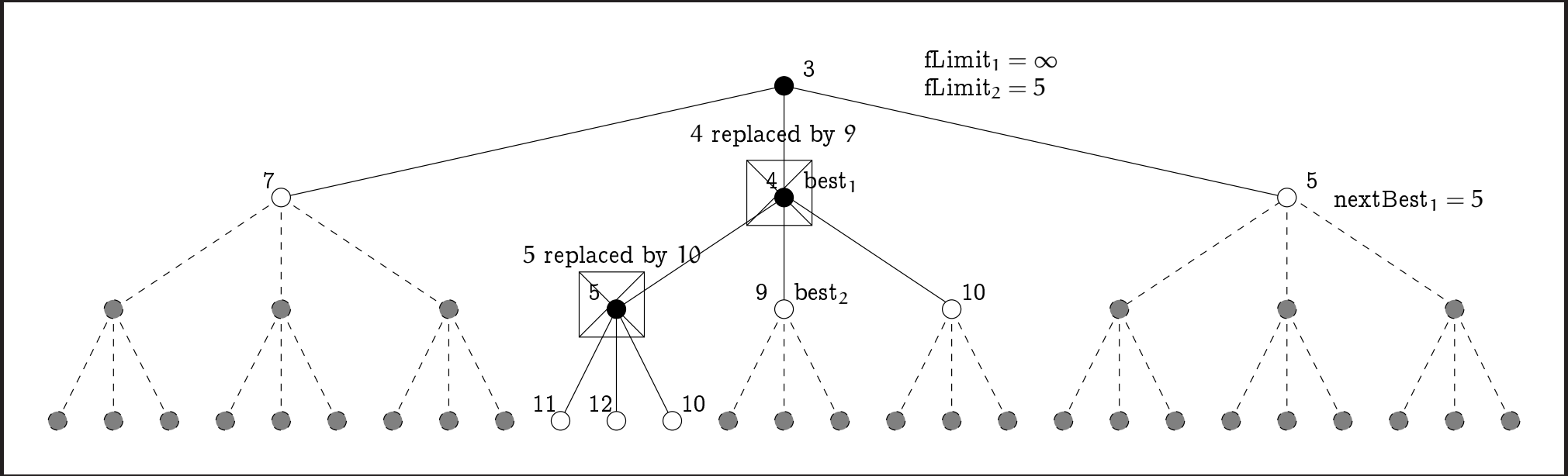
Function call number 3:



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

Recursive best-first search (RBFS): an example

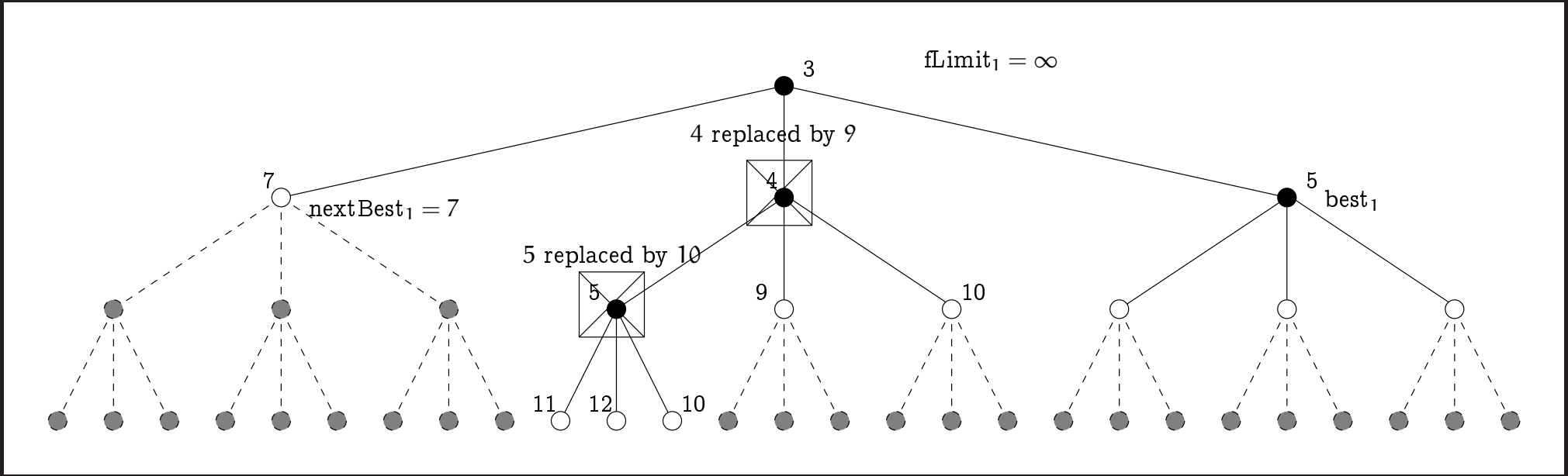
The while loop for function call 2 now repeats:



Now $f(best_2) > fLimit_2$ so the function call returns $(fail, 9)$ into $(result_2, f(best_1))$.

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 is $O(bd)$
- 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 bath-water.

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

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

Exercises

1. Exam question: paper 5, question 6, 2004.
2. Exam question: paper 3, question 8, 2007.
3. Exam question: paper 3, question 7, 2008.
4. Exam question: paper 4, question 3, 2009.