Search in AI

Chapters 3,4,&5 in your Text

Doing the Right Thing

"In order to cope, an organism must either armor itself (like a tree or a clam) and "hope for the best," or else develop methods for getting out of harm's way and into better neighborhoods of the vicinity. If you follow this later course, you are confronted with the primordial problem that every agent must continually solve: *Now what do I do?"*

– D.Dennett, *Consciousness Explained*, in Luger, p. 75

Now What Do I do?

- In our initial discussion, I'm going to use the term AGENT
- Likely a term you've heard before
- It's a systems metaphor: an agent is a system that perceives its environment and acts on those perceptions
- This doesn't imply much sophistication: the clock in my PC that adjusts for daylight savings time is an agent
- An intelligent agent is one that acts intelligently (see our discussion on intelligence!)

How Can an agent decide what to do?

- Think back to our water-jugs problem
- An agent can consider possibilities systematically until it comes up with a sequence that lets it do what it needs to achieve its goal
  - and use knowledge to reduce what it has to consider
- This is called *planning* – we're using a search process to construct a plan to achieve our desire
- Does every agent have to plan?

NO

- Of course not…plenty of simple agents don't do anything this complex
- It depends on the richness of the agent's domain and what it takes to be successful in it
- insects don't plan (giant bugs from outer space excluded). Neither do most animals.
- They don't operate in our world, but they DO find food, avoid enemies, and reproduce
- We generally want more than this (though you'd never know it from watching daytime TV…), which is why we DO plan
- But it IS possible to deal with simple situations without it

Reactive Agents

- Very simple agents react to their environment purely – they have no ability to remember previous situations or do any learning (at least what we'd call learning)
- Biologically, these are creatures like crickets, for example, whose responses to the world develop through evolutionary mechanisms
- have a built-in mapping (a function!) of perception to action
- Pure reaction good enough for many environments
- But not for those that require significant sequences of actions

Reactive Agents with State

- Adding the ability to remember a state makes an agent much more interesting
- We can do some learning, in that the agent can recall what it's seen before to some degree at least
- Still largely insect-level intelligence
- There are those that argue that even human level intelligence arises out of this, but it isn't demonstrable and it's complex enough to be hard to envision
- Certainly SOME of our behaviour does

Agents that can Search

- Assuming we have reaction too (we'll get to combining these later!) we have the best of both worlds
- Agents that can search can solve problems like the water-jugs and other puzzles, as well as putting together sophisticated sequences of physical actions
- Exploring issues and working out details before trying to do anything (searching a hypothetical space)
- This may be a physical search as well

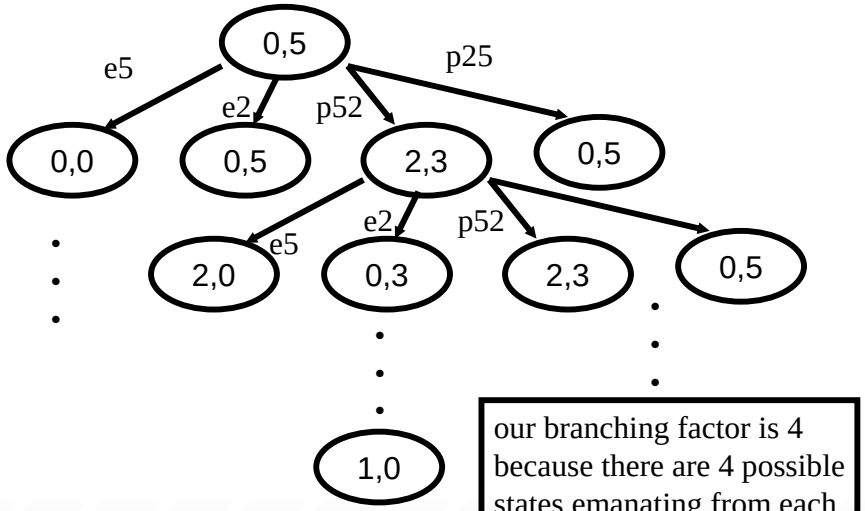
Problem States

- A problem state is a problem's configuration at a specific point in time (i.e. after a particular set of rules/operators has been applied)
- We have some representation for this, or construct one when confronted with the problem (e.g. 2 integers for the water-jugs)
- The initial state (or context) is the problem state before any operators have been applied
- The goal state is the desired problem state
- a Plan is a sequence of operations that takes us from the initial state to the goal state

Problem Space

- From the concept of a state we can talk about the problem space
- The problem space is the set of all valid problem states
- We can generate all the possible states in a problem using some sequence of operations
- so, the problem space can be represented by a graph (often a tree!) in which each node represents a problem state and each arc represents an operator to move to another state

Problem Space: Water Jugs



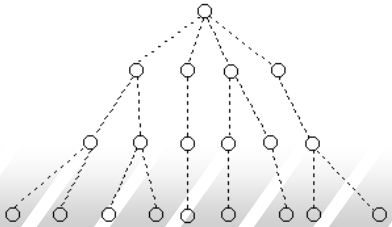
our branching factor is 4 because there are 4 possible states emanating from each existing node

Aside: Physical search vs. Graph Search

- Path planning is itself a vital part of robotics, and in virtual areas like gaming as well
- However the search we use here is equally applicable to searching through a *problem space* as opposed to searching a physical space
- a few issues are different (e.g. backing up – hard to do physically for emptying water!)
- Most are the same
  - problems like guaranteed state transitions
    - multiple state problems get nasty sometimes!

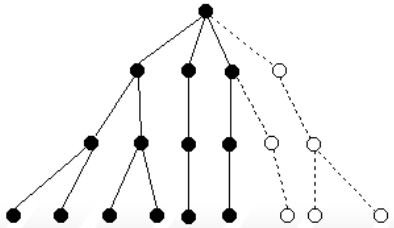
Problem Space

- The problem space is an abstract representation of the states associated with a particular problem
- The problem space does not physically exist when a problem is presented to the system
- i.e. a tree like the following is constructed BY searching – we're not traversing an existing structure (nor could we in most cases!)



Problem Space

- The problem space is searched by the search process
- The dark lines indicate paths that have been examined, the dotted lines paths that have not been examined

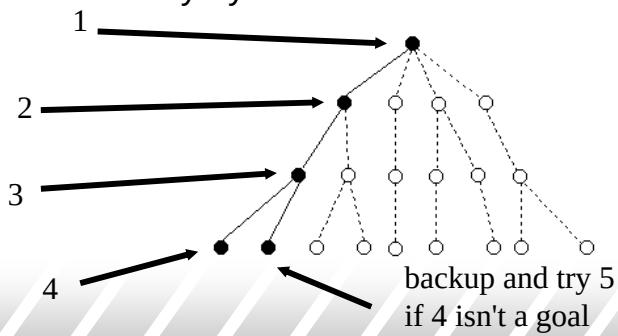


State Space Problem Solving

- state space problem solving involves creating complete problem states and searching for a solution within the space of all possible problem states (i.e. the problem space)
- Each of the states is a complete description of the problem (e.g. the level of water in 2 jugs) at a specific point in time (after some specific set of rules/operators has been applied)
- There are many approaches to search in state-space problem solving – you know some of these already

Depth-First Search

- Depth-first search involves examining the descendants of a node before examining the siblings of a node
- Apply the first possible rule to the initial state, and proceed downward from that using the first available rule – we only try alternatives if the

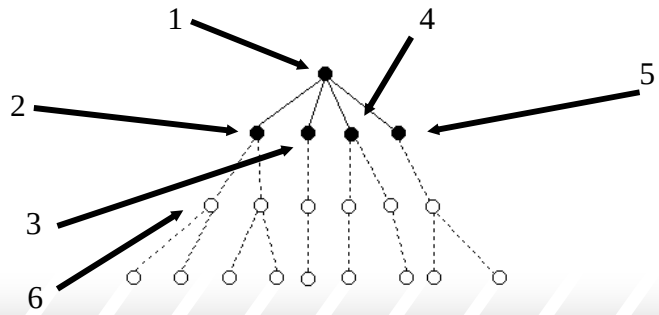


Depth-First Search

- Good:
  - Uses a minimal amount of storage space : O(bd) where b=branching factor, d=depth
    - store path and unexpanded nodes on that path
  - Good if the tree is relatively uniform and there are several paths to a goal state (time O(b^d))
- Bad:
  - Can get lost in a deep subtree
  - Need to worry about infinite loops from cycles
  - Must backtrack if a dead end is encountered

Breadth-First Search

- Breadth-first search involves examining all siblings of a node before examining the descendants of a node



Breadth-First Search

- Good
  - Does not get lost in deep subtrees
  - will find shallowest solution 1st (not nec. the best!)
  - No need to backtrack
  - Good if there are few paths to a goal state
- Bad
  - all nodes must be stored – time and space O(b^d)
  - If several paths lead to a goal state, time is wasted by examining them all
  - e.g. say branching factor of 10, 1000 nodes checked/second, 100 bytes of storage/node

BFS Performance

depth	nodes	time	memory
0	1	1 millisec	100b
2	111	.1sec	11k
4	11,111	11 sec	1 meg
6	10^6	18 min	111 meg
8	10^8	31 hrs	11 gig
10	10^10	128 days	1 terabyte
12	10^12	35 years	111 ter
14	10^14	3500 yrs	11,111 ter

remember, time is also the same for basic bfs too – it just saves space

Uninformed Searches

- These are called uninformed searches because they require no knowledge whatsoever of the domain
- They're weak methods – they work for anything we can define in a graph like structure with transitions based on operations (water-jugs to path planning!)
- There are ways of improving each of these to minimize their weaknesses – they don't change the crappy complexity results though!

Improving DFS

- The main problem with DFS is getting lost in a deeeeeeeeeeep subtree
- We can impose a depth limit to stop that: specify some number K to backtrack after going that deep, rather than continuing
- What does it mean when we chop the space like this?
- Our search is no longer complete – we don't search everything so we're not guaranteed to find the answer!
- Here we know when we won't find it: when it lies below our cutoff depth!

Improving DFS

- What can we do?
- If goal state can't be found within the depth bound, depth bound must be increased and search restarted from the beginning
- Depth First Iterative Deepening
- This sounds incredibly wasteful
- It actually ISN'T for many problems – think of how the problem space grows with the # of operators – retracing the first few levels is trivial compared to the work in adding a level
- we have to think that we're usually working with BIG problems

Iterative Deepening Overhead

- Say the depth of the tree is d, and each node has b branches coming off of it
- The number of nodes in a full tree is:
  - 1 + b + b^2 + b^3+…+ b^(d-1)+ b^d
- And the repeated parts? Well, the deepest stuff is generated once, with one repeat for the level before, and one more (repeatedly) for each level above that. Total nodes are:
  - (d+1)1+(d)b+(d-1)(b^2)+…
    ...+(2)(b^(d-1)) + 1(b^d)
  - The bigger the size of the level, , the fewer repeats!

Making This Concrete

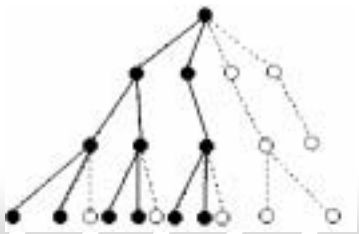
- suppose we had a tree of depth 5 with a branching factor of 10:
- plugging these numbers in, we get:
  - 6+50+400+3000+20000+100000 = 123,456
  - as opposed to 111,111 for the full tree with no repeats
  - so the repeated portions comprise only about 11% more nodes when b=10
  - even if b is only 2, still only 2x as long with repeats
- So this isn't so bad! In fact -

Iterative Deepening

- time O(b^d), space O(bd), guaranteed to find the optimal solution eventually
  - and no getting lost in subtrees
  - the bd space is because we're storing the path to a node and the unexpanded nodes for each node on that path
- Iterative deepening is actually the preferred method when there is a large search space and the solution depth isn't known
  - *assuming we're stuck with an uninformed method*

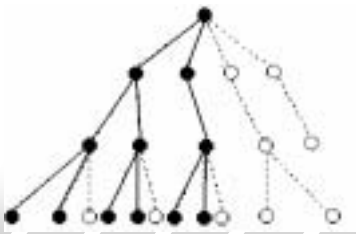
Iterative Broadening

- Breadth-first search can also be performed iteratively
- Instead of examining every node under the current node, only the first **n** nodes under each node we look at are examined – still using a BFS but only looking at and storing some of the nodes



Iterative Broadening

- Usually, the search begins with **n=2**
- If a goal state is not found, the search is repeated with **n=3**
- This process continues until a goal state is found
- rarely used because advantages of iterative deepening aren't usually there – rarer that eliminating some of the breadth is as useful as eliminating some of the depth



Another Approach

- Problem Reduction Problem Solving
- When you have a big problem, break it up and solve the pieces. Keep doing that and the pieces will eventually be trivial!
- a Divide and Conquer technique
- It's possible to do this in a forward direction (break down our problem description)
- But it is more natural do do this in a backward direction (start with our goal and break achieving it down into pieces)
- Before we get to PRPS, let's talk about why backward reasoning is useful in general

Search Direction

- Until now, we have assumed that search always proceeds in a forward direction, from the initial state towards the goal state
- However, the number of choices may be significantly smaller if the search proceeds backwards, from the goal state towards the initial state
  - not always, but often - and often quite dramatically

Backward Search

- Backward search can be used if the description of the goal state is explicit (e.g. not just to find a solution to a crossword puzzle)
  - there's another restriction too, that's coming up…
- Backward search is often used when solving planning problems, putting together a sequence of actions that achieve a specific goal
- e.g. if goal is in(me,berlin). I hypothesize I'm in berlin, and say ok, how did I get here?

Backward Search

- Backward search is often much more efficient than forward search because *it constrains the search process to working with relevant operations* (i.e. focuses the attention on the goal)
- e.g. if I'm trying to get a robot to the doorway, I start there and say what would precisely get me here, rather than starting at the beginning and asking "what can I do?"
- everything I consider is *directly relevant to being at the door*, rather than looking at anything I could do at any point

Resolution Refutation

- can be a backward search, if we use the goal or a resolution from it each time
- The reason we use a negated hypothesis is to have a specific goal to attempt to generate a contradiction from, rather than having to look everywhere
- Recall if we attempt to generate a contradiction from a positive hypothesis, we have to go thru the whole space to show that there isn't one
- with RR, every step we take can involve our goal if we so choose
- Attempting to generate our H from initial facts would be forward reasoning, with much potential for doing useless deductions!

Backward Search

- Not every problem can be tackled this way!
- The water-jugs problem cannot be solved using backward search
- because the operators aren't reversible – e.g. if I empty a jug, I can't guess how much water was in it before to reason about the previous state
- You can do backward search in a state-space manner, *provided you can reverse the operators*
- However, backward search is most often carried out with problem-reduction search

Problem-Reduction Search

- State-space search involves applying rules/operators to problem states to create new, complete problem states
- Problem reduction problem solving involves breaking a problem down into smaller subproblems that can be solved individually

Problem-Reduction Search

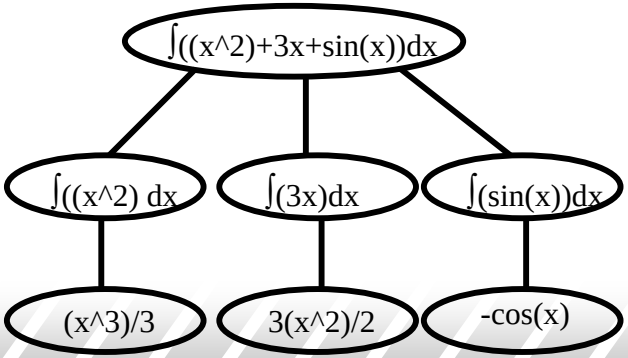
- For example, problem solving in integral calculus frequently uses a problem reduction approach

  e.g. $\int (x^2 + 3x + \sin(x))dx$
- This problem can be reduced to the subproblems

  $\int x^2dx + \int 3xdx + \int \sin(x)dx$
- The solution to the problem is the sum of the solutions to the three subproblems

Problem-Reduction Search

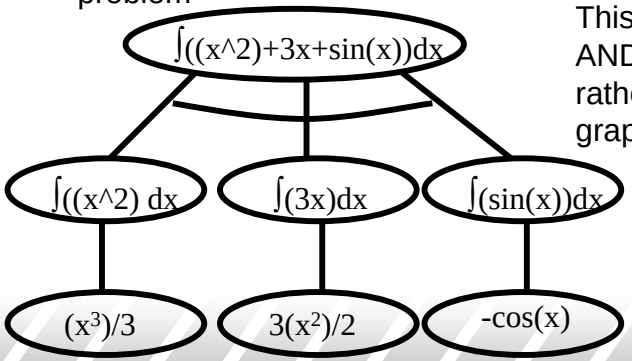
- Like state-space search, PR search can be easily represented by a graph
- the nature of the graph is different though. Consider our calculus problem



Problem-Reduction Search

- What's different about this graph compared to our usual state space graph?
  - A path to a leaf alone isn't our solution- there are parts where we have to do ALL of the parts of the problem

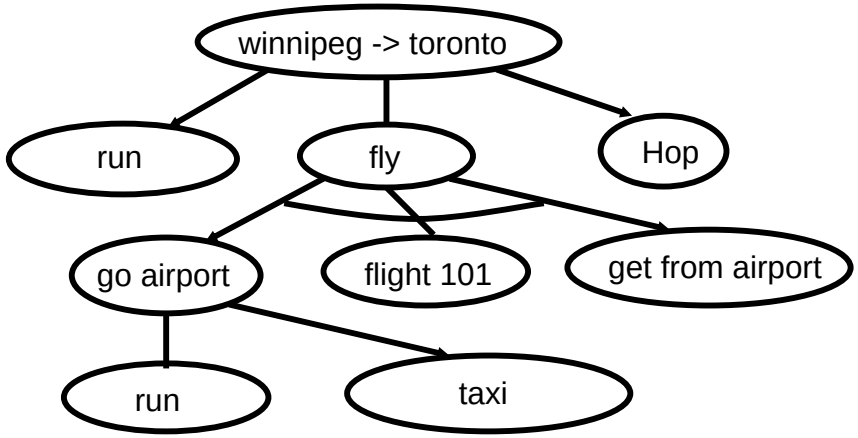
This is an AND-OR graph rather than an OR graph



Problem Reduction Search

- State-space problem-solving involved an OR graph – we could take any path at any point, and we backtrack if we get to a dead end
- In a Problem-reduction search is represented by and AND/OR graph
- The arcs indicate subproblems that must all be solved
- Not every node has an arc – that is, we still have alternatives we can explore
- Each node is now not necessarily a complete description of the problem anymore!

Problem-Reduction Problem Solving



Direction vs. Problem Solving Type

- State Space Search and Problem Reduction are two types of problem solving strategies
  - See Polya, *How to Solve It* for many more
- The direction you operate in is independent of the strategy
- You can have backward state space search, forward problem reduction search

Other Techniques

- Bidirectional search
- It may be possible to start from the goal and from the initial state, and meet in the middle
- Still uninformed, but it helps us try to eliminate some of the huge bottom areas in a search tree!
- Requires operators that can work backwards (that have inverses), just as regular backward search
- This STILL only gets us so far – weak techniques are…weak!

Summary

- Choice of forward/backward search
- Choice of state-space/problem reduction
- These choices are independent
- The correct choice depends on the characteristics of the domain
- **All of these are weak techniques:** they're very broadly applicable, but not particularly powerful, because they don't take the domain into account

Uninformed Search

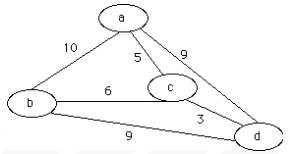
- Depth-first search and breadth-first search are *uninformed* search strategies
- The strategy has no knowledge of how to solve problems **efficiently**
- Uninformed search is acceptable only if the problem space is relatively small
- This will obviously not be the case in most interesting problems!
- Problems for which the number of possibilities increases exponentially (or worse!) suffer from a *combinatorial explosion*: the branching factor causes an enormous number of potential states after a few levels

Combinatorial Explosion

- e.g, Chess, after 40 moves
  - 10^{120} ways that board state could have been generated
- e.g. Go
  - an average game has 10^{800} possible outcomes
- And these are finite games with rigid rules
  - How many outcomes in a 10 minute trip driving?
  - How many outcomes of attempting to cook breakfast?
  - and more importantly, how many outcomes do you ***expect*** from these?

Exponential Problems

- It actually doesn't even take much to get an exponential (or worse!) problem… e.g. the Traveling Salesman Problem
- Given a list of cities and the distance between each pair of cities, find the shortest path that visits each city exactly once and then returns to the starting city



Here, The minimal tour is
a, b, d, c, a with a path length of 27

Travelling Salesman Problem

- The problem space is proportional to N!

  10! = 3,628,800
- For even small values of N, the problem space is too large to be able to search it completely

Combinatorial Explosion

- time for growth rates vs. problems size at 1 microsecond per operation

	n					
	10	**20**	**30**	**40**	**50**	**60**
				time		
n	.00001s	.00002s	.00003s	.00004s	.00005s	.00006s
n^2	.0001s	.0004s	.0009s	.0016s	.0025s	.0036s
n^3	.001s	.008s	.027s	.064s	.125s	.216s
n^5	.1s	3.2s	24.3s	1.7min	5.2min	13.0min
2^n	.001s	1.0s	17.9m	12.7days	35.7yrs	366 centuries
3^n	.059 sec.	58 min.	6.5 years	3,855 centuries	$2x10^8$ centuries	$1.3x10^{13}$ centuries
n!	3.6 sec.	78.5 centuries	??	??	??	??

  - Garey and Johnson, *Computers and Intractability*

Informed Search

- How do we deal with these numbers?
- Literally: because most of the problems we deal with daily are of this kind of complexity!
- We use knowledge of the problem to not bother expanding parts of the tree
  - e.g. hopping to toronto!
- In general trying to decide whether to expand a node or not is a significant problem in itself
- But our experiences teach us situations to be wary of, or ignore, that work **most** of the time

Thinking Computationally

- When the problem space is too large to be examined exhaustively, the search process needs to identify the states that appear to be more plausible and examine these states first
- As I said, we have pieces of knowledge that let us know – not perfectly, but MOST of the time – whether a state is worth pursuing or not
- There's a proper word for this knowledge:
- A **heuristic** is a technique (usually consisting of domain-specific knowledge) that identifies plausible from implausible problem states

Deciding Based on Heuristics

- If we're using heuristics to exclude some states,what does that say about our search process?
- Like when using a depth bound in a DFS, it's no longer complete: we don't know that we're searching everything!
- we might not find the optimal path – but in most problems we don't care about optimality – we care that our solution is good enough given the resources expended to find it
- There's a word for that too: **Satisficing**
- **it's what we do most of the time**

Heuristic Search

- Heuristic solutions are not guaranteed to be correct
- Sounds useless…
- But what in life is guaranteed to be correct?
- The only things that are guaranteed are hard answers to simple problems – problems where we can search the entire space (or have a hard algorithm!). Anything much more complex likely requires universe-lifetimes to guarantee
- So we live our lives without guarantees!

Satisficing

- Of course the decision that is optimal in the simplified model will seldom be optimal in the real world. The decision maker has a choice between optimal decisions for an imagined simplified world or decisions that are good enough, that satisfice, for a world approximating the complex real one more closely.
- We use heuristic solutions to hard problems not because we prefer less to more but because we have no choice!

– H. Simon, *The Sciences of the Artificial*, p 35

Search and Intelligence

- "The amount of intelligence in a search strategy is measured not by the amount of search actually performed, but by the amount that *would have been necessary* had an intelligent method not been used (i.e. by *the amount of search avoided*)."

– Newell and Simon

Travelling Salesman Problem Heuristic

- In order to reduce the number of states in the problem space that must be examined, the search process requires some domain knowledge (a heuristic)
- For example, the nearest neighbour heuristic performs a local minimization at each step
  - select the closest city that has not yet been visited

Travelling Salesman Problem Heuristic

- This heuristic returns a path of a, c, d, b, a with a path length of 27
- Turns out the minimal tour has a path length of 27 – so this *just happens* to be optimal too (remember we aren't guaranteed to do this well at all!)
- This heuristic search would execute in time proportional to N^2 instead of N!
- Remember we want to balance the *cost of the heuristic* against the work it saves us and the quality of the solution
- I.e. if we care a little less about optimality maybe there's something even cheaper..?

Distance/Cost Functions

- Heuristics are often described in terms of functions that produce the goodness of a state in terms of how close to a goal it is, an actual estimated distance to a goal, or the remaining cost to get to the goal from there
- Goodness increases as we move toward a goal, while distance and cost decreases
- For example, the function h(N) might returns an estimate of the *distance* from a node/state N to a goal state. Decreasing h is therefore better here
- In other domains an increasing h is better, and this will be noted as an exception when it occurs
- We will note in any given domain which of these we are using

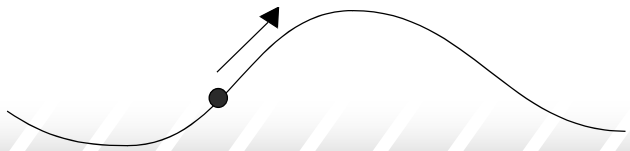
Accurate H

- Heuristics are often described in terms of functions that produce the goodness of a state in terms of how close to a goal it is, an actual estimated distance to a goal, or the remaining cost to get to the goal from there
- Goodness increases as we move toward a goal, while distance and cost decreases
- For example, the function h(N) might returns an estimate of the *distance* from a node/state N to a goal state. Decreasing h is therefore better here
- In other domains an increasing h is better (e.g. a game where we are talking about the general goodness of a move)
- We will note in any given domain which of these we are using

Hill-Climbing Search

- Any state can be evaluated for goodness
- Suppose we plot that – we end up with a three dimensional landscape with height based on goodness

- The goal will obviously be the highest point



Hill Climbing Algorithm

- Example of a class of algorithms called *Iterative Improvement Algorithms*
- In these algorithms, we always have *a* solution – it's just not a very *good* one (we start off far from our goal)
- At each step we move closer – climbing the hill
- We begin at the initial state (black=visited, white = don't know about it yet):

From the point of view of a search tree

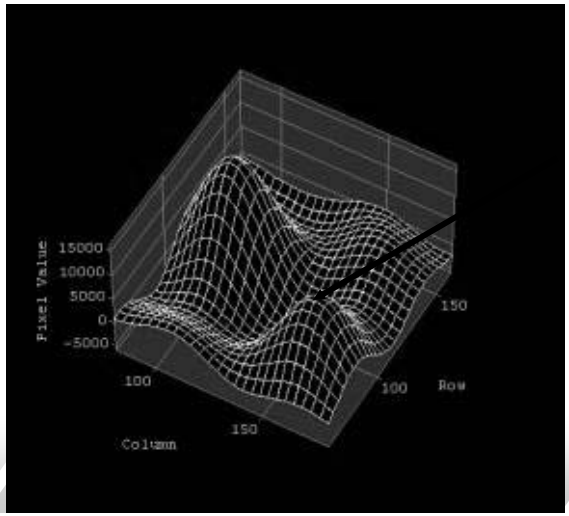
- Expand this state's children to determine the distance of each successor state (i.e. call heuristic fn) – we have to generate (visit) these nodes to get the heuristic value

- Select the ***successor*** with the best h value (higher is good here) and continue, always taking the best of the successors as the next move up the hill (i.e. all children visited, one chosen to keep following)

- What's the obvious problem with hill-climbing search (really iterative improvement alg's in general)??

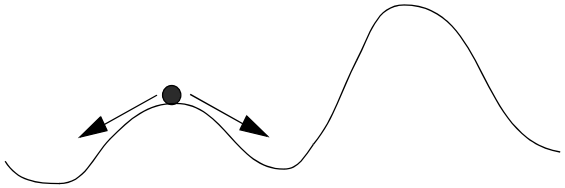
Local Maxima / Mimima

- Its possible to get to a point in the search space where every successor state is worse:



You Are Here

Local Maxima/Minima



- We think this is our goal because it's the best point around, but it's not *globally* the best: we're stuck on a ***Local Maximum*** and can't get off
- ***Minima*** occur if we're talking about minimizing h(n) values rather than maximizing

Curing local Maxima?

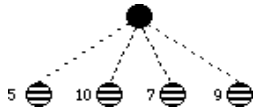
- Option 1: Try elsewhere!
- *Random Restart Hill Climbing*: Start again from somewhere else in the search space (the idea being if we start someplace else and don't get to the same place, we've found a different (possibly better maximum)
- Option 2: step downward occasionally to see if it gets you restarted up again quickly or not!
- *Simulated Annealing* – gradually cooling off over time (from metallurgy). we have some given likelihood of choosing a downward move at each point (also considering how bad that move would be), and that likelihood decreases with a time factor (the cooling)

Beam Search

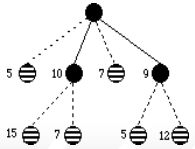
- Expand all successor nodes of the nodes at the current level
- Calculate h(N) for each of the successor nodes
- Create a list with the best *m* nodes (i.e. the *m* nodes with the best h(N) values) *at the current level*, and ignore the remaining nodes. we can increase m if there's no solution found
- Similar to iterative broadening, but we're not being arbitrary, we're actually working with what we *think* are the best!

Beam Search

- For example assume >h is good, expand (visit) children to get h value, then select the m=2 best nodes for processing

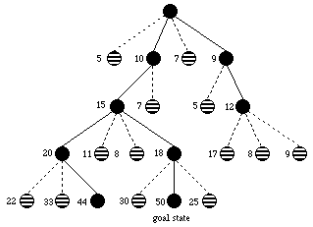


- Evaluate these nodes and generate their successors



Beam Search

- Continue generating successors and selecting the m best nodes at each level until a goal state is generated



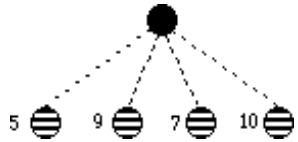
- If the extra nodes are discarded, the search process may not find a goal state: restart with > m value

Best-First Search

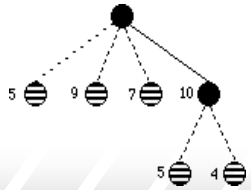
- Expand (visit) all nodes under the current node
- Calculate the h(N) value for each of the new nodes
- Select the node with the best h(N) value ***in the entire tree*** (as opposed to best child)
- This technique follows the most promising path until it becomes less promising than some other path
- Doing this with just an h value is also called:
- ***Greedy search***

Best-First Search

- Generate (visit in the graph) the successors of the initial state (>h = good again here)

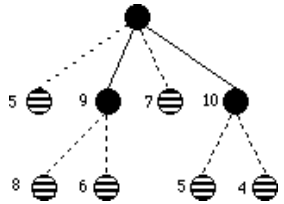


- Select the most promising state and continue



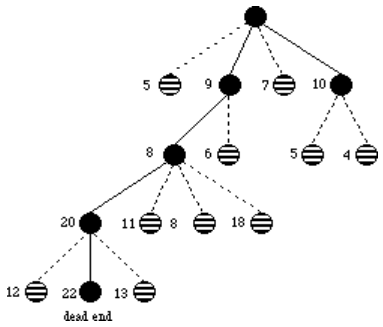
Best-First Search

- At each stage, select the most promising state in the entire tree and generate its successors



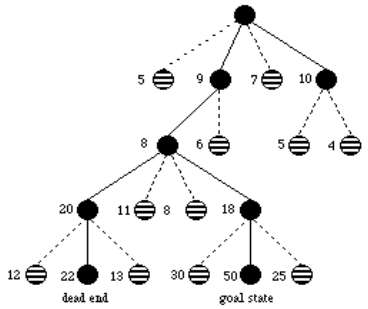
Best-First Search

- If there is a dead end, select the next most promising state



Best-First Search

- Stop when a goal node has been reached



Getting Better?

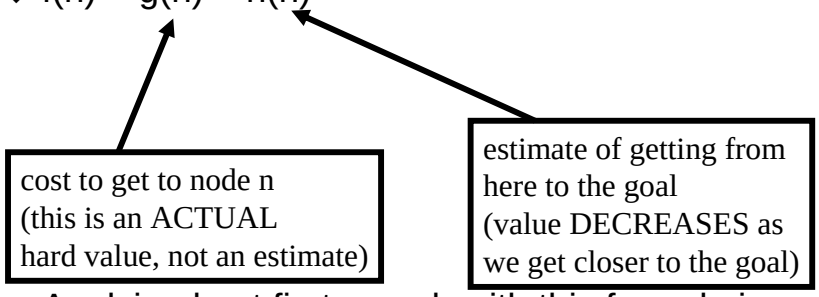
- There are lots of variations on best first search. For example, my heuristic is measuring goodness or closeness to the goal – I'm choosing the best node based on that, hoping to reduce my workload
- Remember though that this is an estimate… It's all based on how far I ***think*** I am from a goal
- it's possible to have a heuristic lead us down the wrong path too, remember. So -

Thinking Further

- Because of this, when performing a best-first search, it would be nice if when we have a choice between two nodes, we chose the shallower node
- That would reflect on the fact that it's likely less work to get to the shallower one (a heuristic in itself!)
- Even better, we could keep track of the **actual** work (terrain, resources, whatever, instead of using shallowness) – note that this is not hard to do, and it's exact – not a guess!
- This means that our overall heuristic could be made up of two parts:

A Two-Part Heuristic

- f(n) = g(n) + h(n)



- Applying best first search with this formula is called algorithm A - inherently better than heuristic alone, because we consider the (real!) cost so far too.

Admissible Heuristics

- We say an heuristic is admissible if it never overestimates the cost to a goal
- admissible heuristics are optimistic! They'll tell you something is easier or cheaper than it really will be, but never that it's harder or more expensive!
  - My plumber and my car repairman are both like this ☺
- straight-line distance to a city on a map is an admissible heuristic
- What's so cool about these?

A* Search

- Algorithm A (f(n)=g(n)+h(n)) + ***admissible H*** =
- ***Algorithm A****!
- i.e. best first search counting cost and assuming an admissible heuristic
- this is a VERY special algorithm – this isn't algorithm analysis so I won't prove it for you formally but you can look the proof up if you are interested
- A* is complete (guaranteed to find a solution) and also guaranteed to find the OPTIMAL solution (that's what makes it special!)

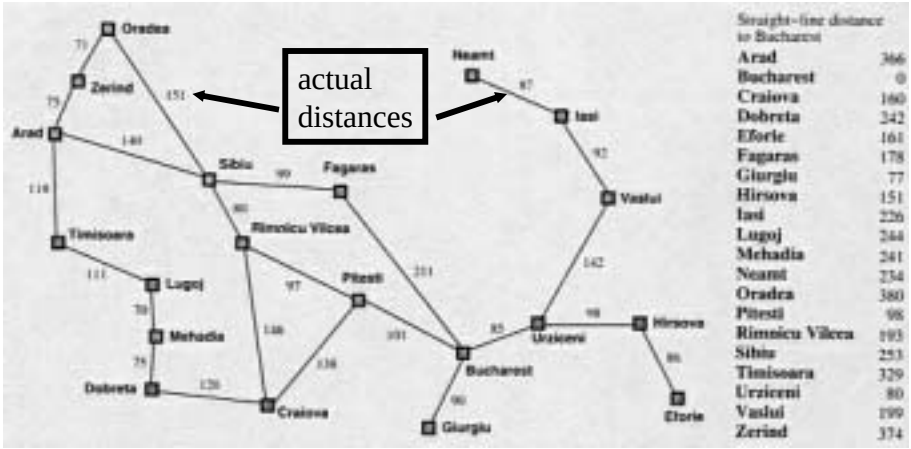
Sketch of Proof

- As we go further down the search tree, a greater portion of the f(n)=g(n)+h(n) comes from g rather than h
- Thus our estimates become more accurate as we go along – we rely on heuristic less
- if we're doing best first search, we're only getting BETTER as we go along this way
- *Provided we have an admissible heuristic*
- If it's not admissible, we might suddenly get worse, which means we'd never be sure it's optimal

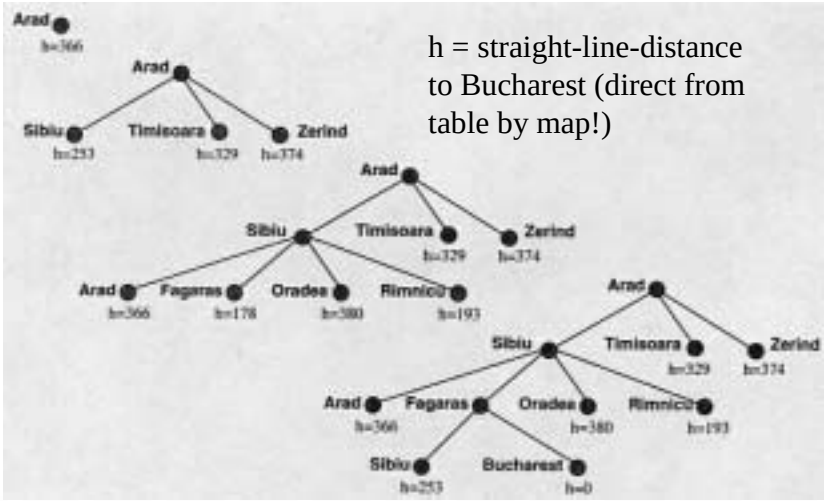
Example: Travelling in Romania

- No, it's not mine. It's from the book "AI: A Modern Approach", by Russell & Norvig (4th year text) which is in the library. This book also has a reasonably basic formal proof of A*'s optimality if you're interested (this proof turns out not to be easy to show for all cases)
- We see a map of Romania, which is not a flat country. Straight line distance between cities is our heuristic, and we want to plot routes from place to place

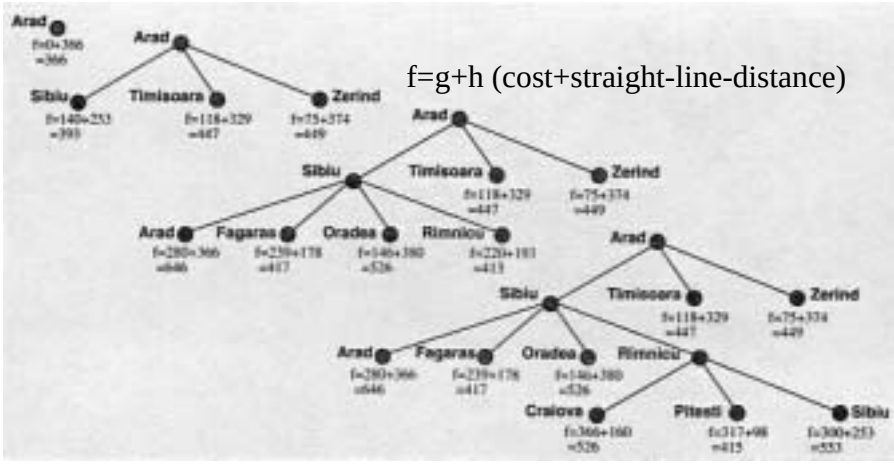
Travelling in Romania



Greedy (best first) search



A* search



The last step from Pitesti is not shown – but you can easily see what it will be!

Control Knowledge

- In many applications, it is not possible to define a function that evaluates the goodness of each state
- However, it may be possible to define symbolic knowledge that guide the search process
- For example, consider a board game – we may not have a quantitative estimate of the distance to a winning position, but we could have something like:

Control Knowledge

If (number_of_moves <= 10) Then try opening_moves
If (number_of_moves > 10) Then try middle_moves
If (number_of_pieces < 10) Then try closing_moves

- This type of knowledge is control knowledge – knowledge that helps us control a search. It's one type of meta-knowledge: knowledge about knowledge, in this case when types of moves are better than others
- Used in most large expert systems: often no obvious H, but we can identify situations where some knowledge is more useful than others

Adversarial Search

- Most often thought of in Games
- But really, any time where our agent is not the only one affecting the environment
- Really, we're never the only ones affecting the environment (others move things, the wind blows things over,…)
- But we'll start with the idea that someone else's decisions are involved too

Search in 2-Player Games

- We take turns making moves, mine depends not only on what I did but what you might do
- With >2 players what I have to consider is even more complicated
- We'll just talk about 2-player games here
- We give our players names, for convenience: *max* and *min*
- A game consists of an initial state and operators, conditions for a goal (termination test - what it takes to win!), and some concept of the utility of a goal (binary win/lose, or points, $…)

Searching in 2-player Games

- If this were a "normal" search problem, all max would have to do is find a sequence leading to a goal state
  - like solving a maze
- But Min's choice-making ability leads to uncertainty
  - We don't know what choices min will make, so it's hard to decide what to do
  - What DO we know about min?

Searching in 2-player Games

- Min's object is to win the game
  - therefore, their goal is to make the best choices from their perspective
  - i.e. MINimize Max's likelihood of winning
- Given this, what is our (Max's) goal?
- To search and find a path to a terminal state REGARDLESS of what min does

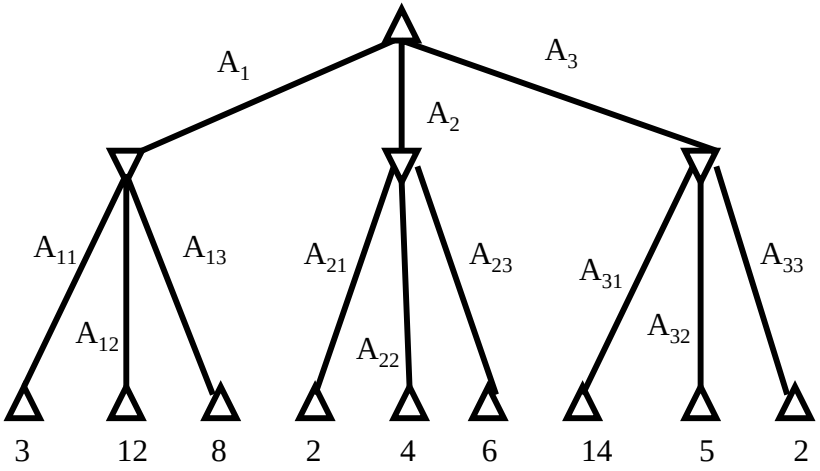
Trivial Example: TicTacToe

- From an initial state, Max has 9 possible moves
- Min responds with a move from each of these and so on
- If we are considering all possible moves on both sides, the tree becomes very large very quickly
- To consider the details of this search process, we'll make up a trivial game

The Trivial Game

- There are only two moves: Max moves, Min responds, and the game is over
  - Wow, fun game, let's get Regis to host
- Draw states where it's Max's turn with upward-pointing triangles
  - Max's Choices: A1-A3
- Min's with downward-pointing triangles
- The depth of the entire search space is one move, consisting of two half-moves or two ply
- Any terminal score results in some point score for Max

The Trivial Game

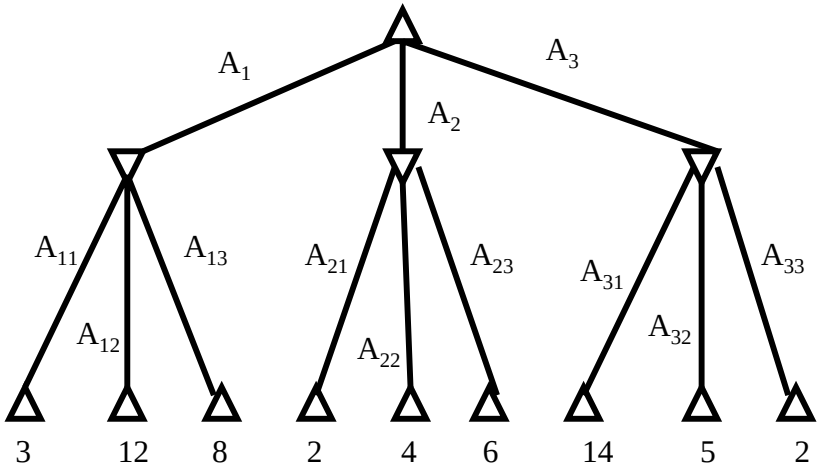


Any 2 player game is like this, the tree is just bigger…what we as Max want is some way to decide which move to take…

Minimax

- An algorithm to determine the optimal strategy for Max and thus the best first move:
  - Generate the entire tree down to all terminal states, and find the utilities of each
  - Apply the utilities of terminal states to determine the utility of nodes (decisions) one level higher
    - How do we do this? We don't know the moves min will chose?
    - But by knowing Min's motivations, we can assume the move Min will choose!
    - i.e. we can assume min will move to maximize his or her chances of winning
    - And if they don't, we'll be doing even better!

The Trivial Game

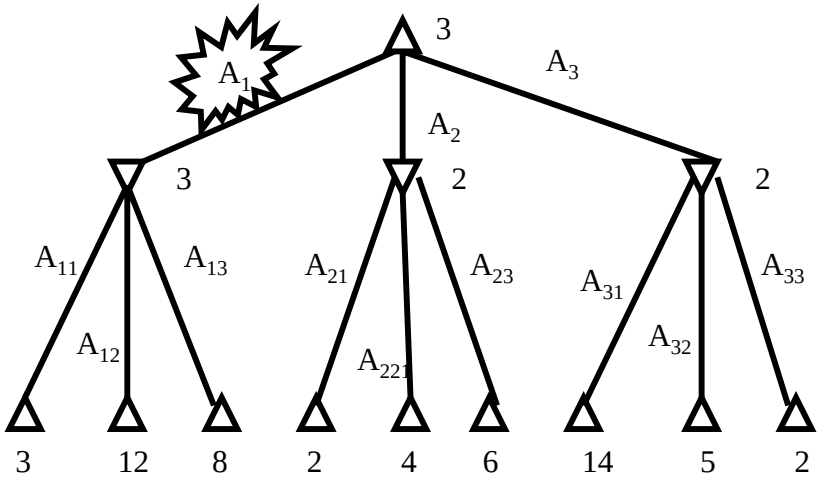


- For example, from Min's moves, we know that the WORST max could do using A1 is 3

Minimax

- i.e., Max MIGHT do better, if Min doesn't play very well
- but knowing Min's motivations, Min will likely confine us to the worst outcome from out point of view (if Min doesn't though, who cares – things are even better then!!)
- We can do this to all branches one level up from the bottom and get the worst Max could do from each…

The Trivial Game



What's Max's move????

>2 ply?

- We picked the best move by choosing the best of the options that max had available.
- If we have >2ply, we repeat in this fashion: assuming scores are from Max's standpoint, we MINimize on each of Min's turns, and MAXimize on each of Max's.
- We keep repeating this as many times as necessary until we hit the top of the tree
- Minimax maximizes utility based on the assumption Min will play perfectly to minimize it – again, if Min doesn't, so what, we do even better!

Minimax

- If depth of the tree is M, and there are B legal moves at each point, time complexity is O(b^m)
  - Bad, but that's because it's a hard problem!
  - what BAD thing does this process so far assume?
- Assumes we can search to the bottom, which we clearly can't do in real time (or even at all, often!).
- Alternatives?
  - Depth bound; precomputation (see Chinook)
  - We're looking at the interesting stuff, so we'll assume a depth bound
  - As soon as we have one, we don't know the utility…

Depth Bounded Minimax

- Instead of going to the bottom and using a termination test to decide if a state is a goal, we instead use a cutoff test to decide if we've gone far enough (easiest = constant function = depth limit)
- We don't know the utility of the outcome anymore, so instead of a utility function we have a heuristic evaluation function of the states where we cut off
- i.e. how likely is the state to lead us to our goal

Heuristics in Games

- Obviously highly domain specific
  - # pieces left
  - Giveaways of known strategies
  - Aspects of board position
  - Strength of pieces left
  - …
- We're relying heavily on the computability and decency of these heuristics!

Cutting off Search

- We could use a fixed depth
- Or time, with some variations on this algorithm
- Both these strict bounds are a problem
  - e.g. in chess, may be white's move, and white may be ahead by a piece - sounds good
  - but may lose a queen in the next move ahead - suddenly much poorer
    - but only if we look one ply ahead
- A problem no matter how far we go - Horizon Problem

Variable Cutoffs

- It would be nice to have something more sophisticated
  - i.e. don't stop at a specific point, stop when it looks like we've gone far enough - i.e. a heuristic for stopping!
- We want to stop and evaluate at moves where further lookahead wouldn't mean much
  - Quiescent state: unlikely to vary much
- We expand nonquiescent states until quiescence is reached
- May consider only certain types of moves (captures, check, etc.)

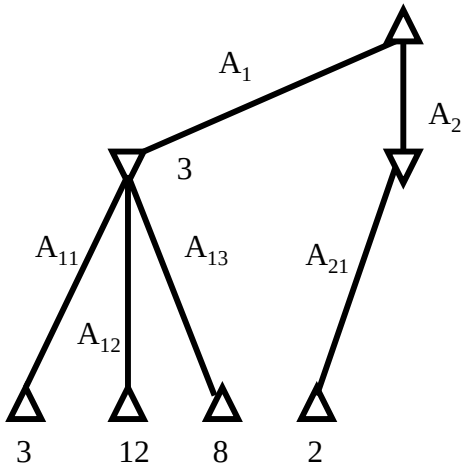
Improving Minimax

- suppose we're playing chess: even a mediocre implementation of minimax will allow 1000 moves/second to be considered
- at around 150 seconds/move, we get to look at ~150,000 board positions
- at ~35 branches/node, this means we have 3-4 ply
- Pretty bad!
  - A decent human player can look at least 6-8 ply ahead

Improving Minimax

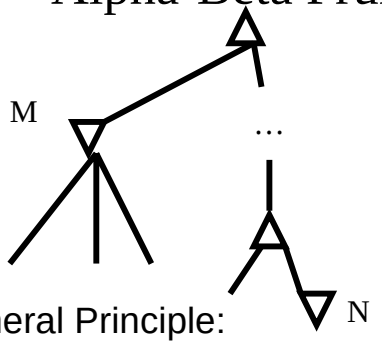
- We'd like to avoid looking at nodes that don't help us
  - Prune the search tree
- A common technique in minimax search is alpha-beta pruning, which ends up giving the same results as standard minimax, but does not look at any nodes that couldn't possibly influence the decision
- Consider our original "trivial game" game tree

The Trivial Game



- Search starts with A1, then A11-13 (DFS), then A2 and A21
- Here we have utility 2
- This means that A2 has at MOST a value of 2
- But we already have a value better than that
- So don't bother with the rest of branch A2!!!

Alpha-Beta Pruning



- General Principle:
  - Consider a node N
  - If we have a better choice M at the parent of N or anywhere further up in the tree, N will **never** be chosen
- Where does the alpha-beta part come in?

Alpha-Beta Pruning

- If we're doing DFS, we are maintaining only one path at a time
  - **Alpha** is the best choice we have found so far for max
  - **Beta** is the best choice for min (the lowest value - worst for max)
  - We update the values of alpha and beta as we go
  - And we can prune any subtree worse than the current alpha or beta value
- So what does this actually give us in practice?

Results

- The effectiveness depends a lot on the ordering in the tree
  - look at the third sub-branch in our example – worst last!
  - It's a good idea to examine descendents in their worst possible order if we can
- If we can do this, we end up needing to examine O(b^(d/2)) nodes rather than O(b^(d))
- our effective branching factor becomes root-b rather than b - pretty darned good heuristic!
- for chess, 6 rather than 35 branches
- we effectively double the number of nodes we can look at given the same processing power

Summary

- General-purpose techniques are referred to as "weak" techniques because they lack detailed domain knowledge
- Weak techniques may be adequate for solving problems when the problem domain has been significantly simplified; however, these techniques will not be adequate when the simplifying assumptions are removed

Summary

- The current view of AI researchers is that an intelligent system requires a large amount of knowledge and a smaller amount of search
- Techniques that rely on extensive knowledge of a domain ("strong" techniques) must be used to solve hard problems
- Unfortunately, adding more knowledge does not always solve the problem

Summary

- For example, in chess, there are only a few dozen possible moves; these moves can be defined in a knowledge base and an inference engine (search process) can then search for the best move
- However, there are approximately 10^{120} states that must be considered so a chess program can not be based solely on a brute-force search of the problem space
- Additional specialized knowledge that is specific to particular situations can be added (e.g. book openings)

Summary

- The amount of this specialized knowledge grows very quickly and soon additional (meta-) knowledge is required to locate the knowledge that is relevant to each board position
- Thus, adding additional knowledge may not reduce the amount of search that is required in a complex domain
- We need to develop specialized mechanisms for representation and search in different types of problems, rather than just adding heuristics!

AI Methodologies

- There are many such specialized mechanisms – many for special areas
- Still there are many more specialized mechanisms that are still useful in many domains
- One of the best examples of these are production systems, more commonly known as rule-based systems
- These focus on the limitations of FOL computationally, and attempt to build a more practical and extendible formalism