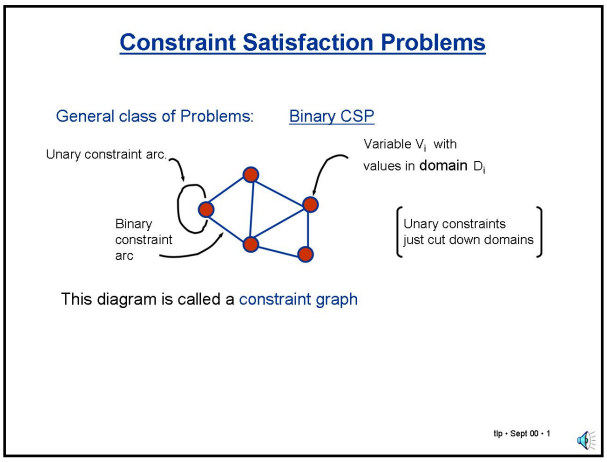
6.034 Notes: Section 6.1

Slide 6.1.1



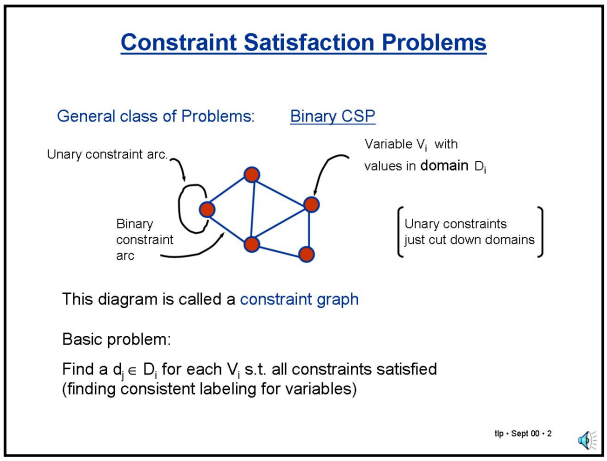
In this presentation, we'll take a look at the class of problems cal

led Constraint Satisfaction Problems (CSPs). CSPs arise in many applic

ation areas: they can be used to formulate scheduling tasks, robot planning tasks, puzzles, molecular structures, sensory interpretation tasks, etc.

In particular, we'll look at the subclass of Binary CSPs. A binary CSP is described in term of a set of Variables (denoted V_i), a domain of Values for each of the variables (denoted D_i) and a set of constraints involving the combinations of values for two of the variables (hence the name "binary"). We'll also allow "unary" constraints (constraints on a single variable), but these can be seen simply as cutting down the domain of that variable.

We can illustrate the structure of a CSP in a diagram, such as this one, that we call a **constraint graph** for the problem.



Slide 6.1.2

The solution of a CSP involves finding a value for each variable (drawn from its domain) such that all the constraints are satisfied. Before we look at how this can be done, let's look at some examples of CSP.

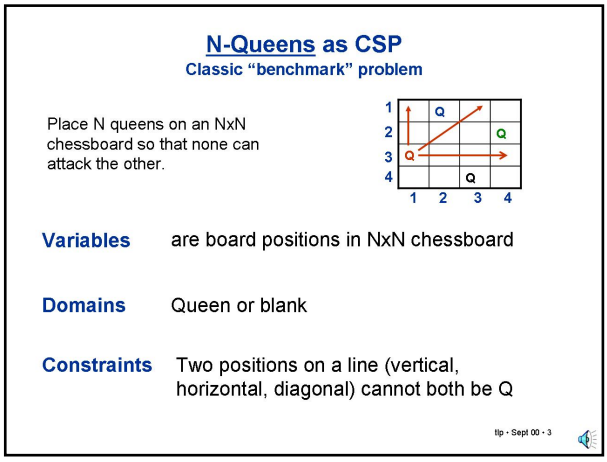
Slide 6.1.3

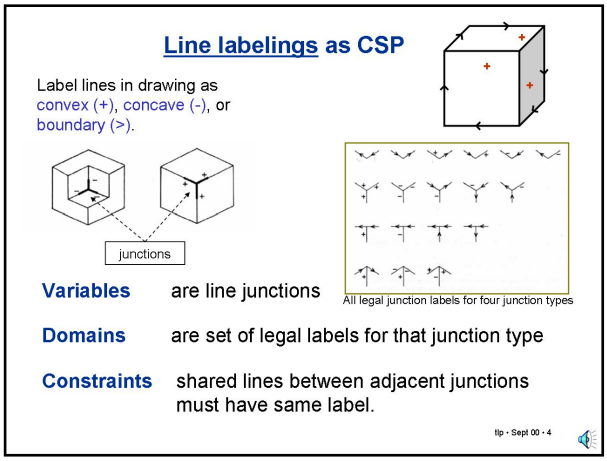
A CSP that has served as a sort of benchmark problem for the field is the so-called N-Queens problem, which is that of placing N queens on an NxN chessboard so that no two queens can attack each other.

One possible formulation is that the variables are the chessboard positions and the values are either Queen or Blank. The constraints hold between any two variables representing positions that are on a line. The constraint is satisfied whenever the two values are not both Queen.

This formulation is actually very wasteful, since it has N^2 variables. A better formulation is to have variables correspond to the columns of the board and values to the index of the row where the Queen for that column is to be placed. Note that no two queens can share a column and that every column must have a Queen on it. This choice requires only N variables and also fewer constraints to be checked.

In general, we'll find that there are important choices in the formulation of a CSP.





Slide 6.1.4

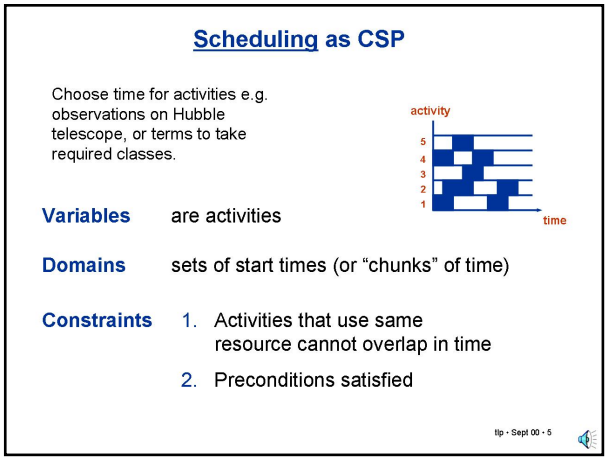
The problem of labeling the lines in a line-drawing of blocks as being either convex, concave or boundary, is the problem that originally brought the whole area of CSPs into prominence. Waltz's approach to solving this problem by propagation of constraints (which we will discuss later) motivated much of the later work in this area.

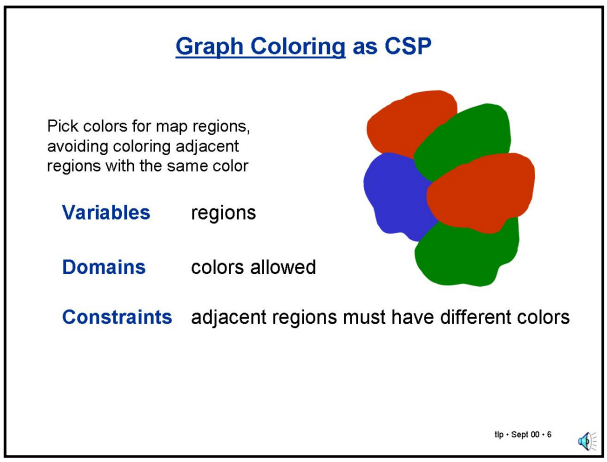
In this problem, the variables are the junctions (that is, the vertices) and the values are a combination of labels (+, -, >) attached to the lines that make up the junction. Some combinations of these labels are physically realizable and others are not. The basic constraint is that junctions that share a line must agree on the label for that line.

Note that the more natural formulation that uses lines as the variables is not a BINARY CSP, since all the lines coming into a junction must be simultaneously constrained.

Slide 6.1.5

Scheduling actions that share resources is also a classic case of a CSP. The variables are the activities, the values are chunks of time and the constraints enforce exclusion on shared resources as well as proper ordering of the tasks.





Slide 6.1.6

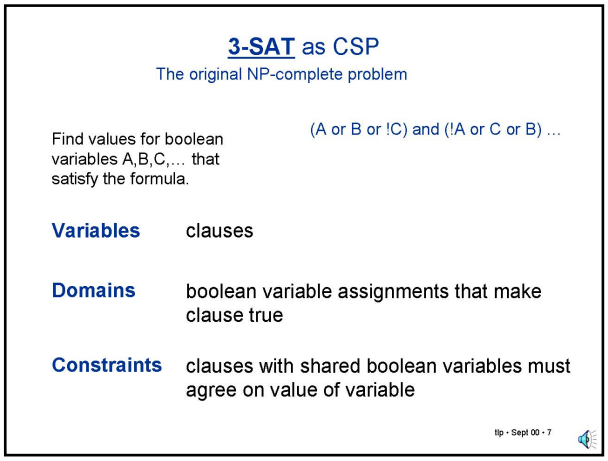
Another classic CSP is that of coloring a graph given a small set of colors. Given a set of regions with defined neighbors, the problem is to assign a color to each region so that no two neighbors have the same color (so that you can tell where the boundary is). You might have heard of the famous Four Color Theorem that shows that four colors are sufficient for any planar map. This theorem was a conjecture for more than a century and was not proven until 1976. The CSP is not proving the general theorem, just constructing a solution to a particular instance of the problem.

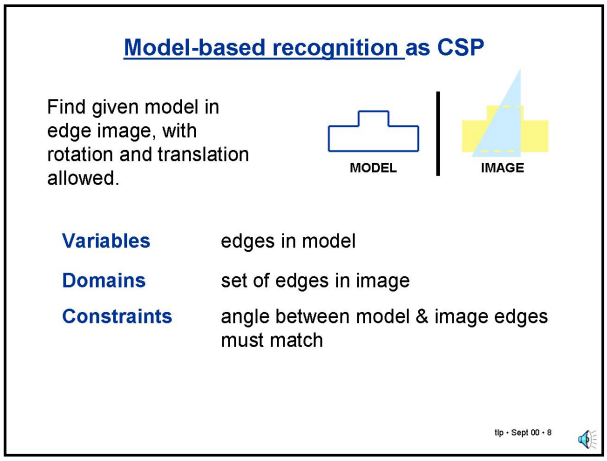
Slide 6.1.7

A very important class of CSPs is the class of boolean satisfiability problems. One is given a formula over boolean variables in conjunctive normal form (a set of ORs connected with ANDs). The objective is to find an assignment that makes the formula true, that is, a satisfying assignment.

SAT problems are easily transformed into the CSP framework. And, it turns out that many important problems (such as constructing a plan for a robot and many circuit design problems) can be turned into (huge) SAT problems. So, a way of solving SAT problems efficiently in practice would have great practical impact.

However, SAT is the problem that was originally used to show that some problems are NP-complete, that is, as hard as any problem whose solution can be checked in polynomial time. It is generally believed that there is no polynomial time algorithm for NP-complete problems. That is, that any guaranteed algorithm has a worst-case running time that grows exponentially with the size of the problem. So, at best, we can only hope to find a heuristic approach to SAT problems. More on this later.





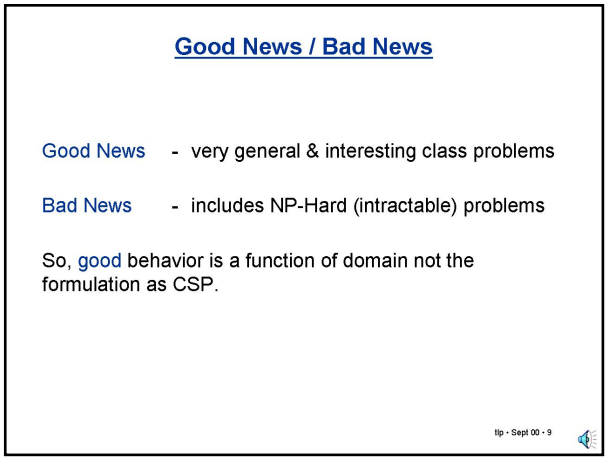
Slide 6.1.8

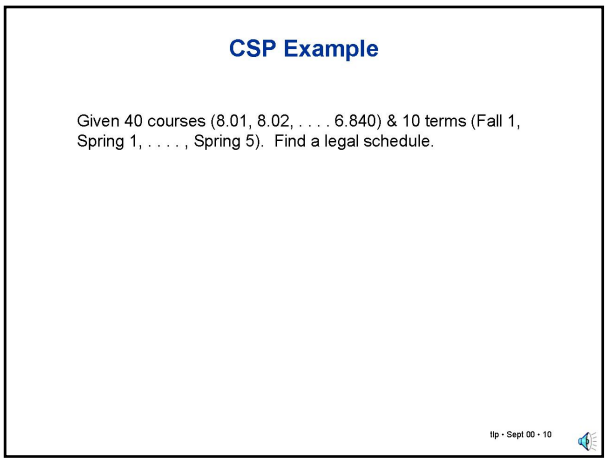
Model-based recognition is the problem of finding an instance of a known geometric model, described, for example, as a line-boundary in an image which has been pre-processed to identify and fit lines to the boundaries. The position and orientation of the instance, if any, is not known.

There are a number of constraints that need to be satisfied by edges in the image that correspond to edges in the model. Notably, the angles between pairs of edges must be preserved.

Slide 6.1.9

So, looking through these examples of CSPs we have some good news and bad news. The good news is that CSP is a very general class of problems containing many interesting practical problems. The bad news is that CSPs include many problem that are intractable in the worst case. So, we should not be surprised to find that we do not have efficient guaranteed solutions for CSP. At best, we can hope that our methods perform acceptably in the class of problems we are interested in. This will depend on the structure of the domain of applicability and will not follow directly from the algorithms.





Slide 6.1.10

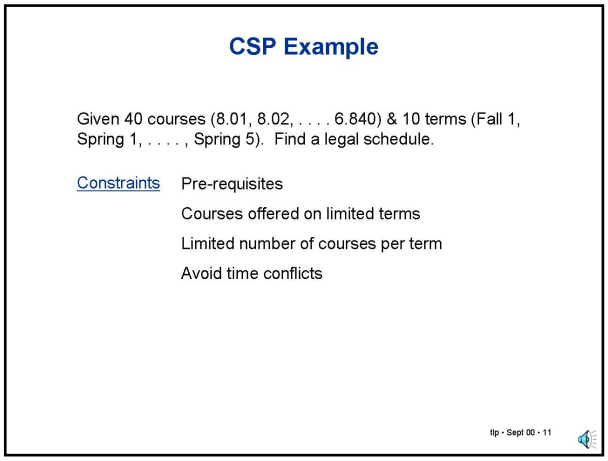
Let us take a particular problem and look at the CSP formulation in detail. In particular, let's look at an example which should be very familiar to MIT EECS students.

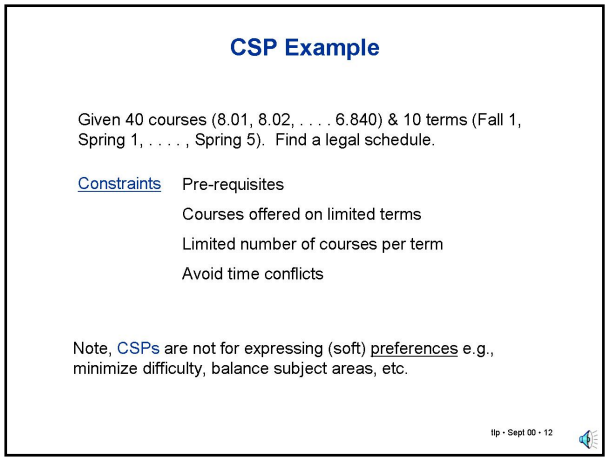
The problem is to schedule approximately 40 courses into the 10 terms for an MEng. For simplicity, let's assume that the list of courses is given to us.

Slide 6.1.11

The constraints we need to represent and enforce are as follows:

- The pre-requisites of a course were taken in an earlier term (we assume the list contains all the pre-requisites).
- Some courses are only offered in the Fall or the Spring term.
- We want to limit the schedule to a feasible load such as 4 courses a term.
- And, we want to avoid time conflicts where we cannot sign up for two courses offered at the same time.





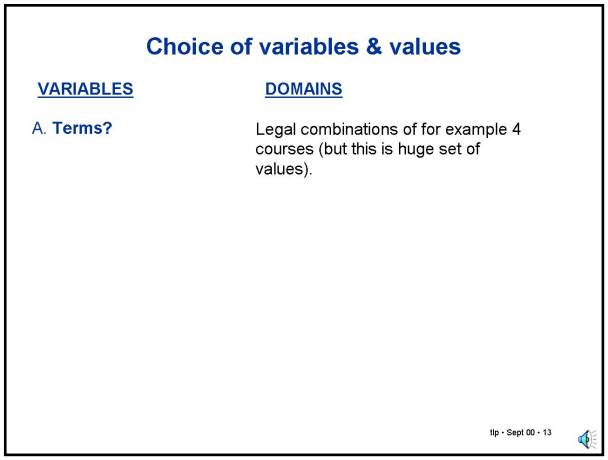
Slide 6.1.12

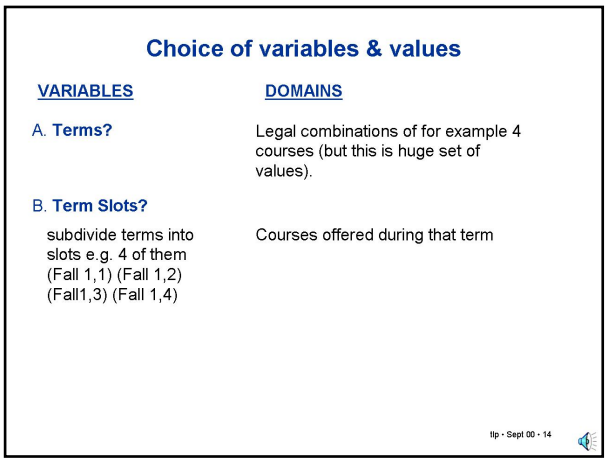
Note that all of these constraints are either satisfed or not. CSPs are not typically used to express preferences but rather to enforce hard and fast constraints.

Slide 6.1.13

One key question that we must answer for any CSP formulation is "What are the variables and what are the values?" For our class scheduling problem, a number of options come to mind. For example, we might pick the terms as the variables. In that case, the values are combinations of four courses that are **consistent**, meaning that they are offered in the same term and whose times don't conflict. The pre-requisite constraint would relate every pair of terms and would require that no course appear in a term before that of any of its pre-requisite course.

This perfectly valid formulation has the practical weakness that the domains for the variables are huge, which has a dramatic effect on the running time of the algorithms.



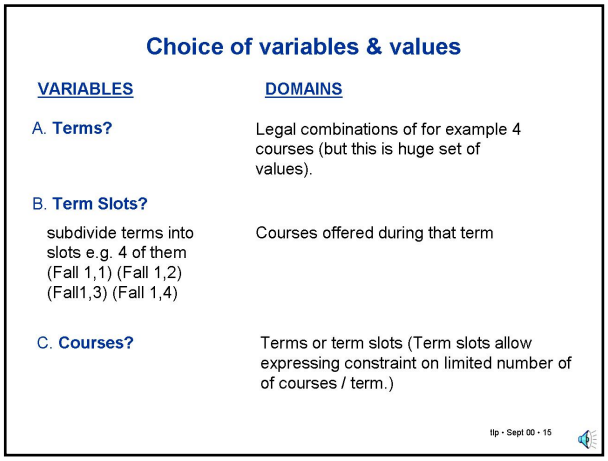


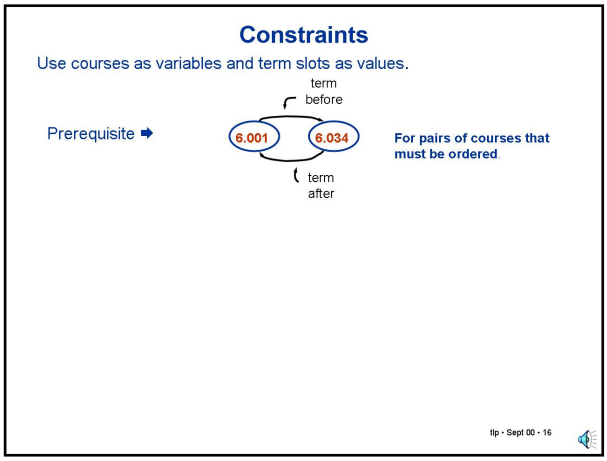
Slide 6.1.14

One way of avoiding the combinatorics of using 4-course schedules as the values of the variables is to break up each term into "term slots" and assign to each term-slot a single course. This formulation, like the previous one, has the limit on the number of courses per term represented directly in the graph, instead of stating an explicit constraint. With this representation, we will still need constraints to ensure that the courses in a given term do not conflict and the pre-requisite ordering is enforced. The availability of a course in a given term could be enforced by filtering the domains of the variables.

Slide 6.1.15

Another formulation turns things around and uses the courses themselves as the variables and then uses the terms (or more likely, term slots) as the values. Let's look at this formulation in greater detail.





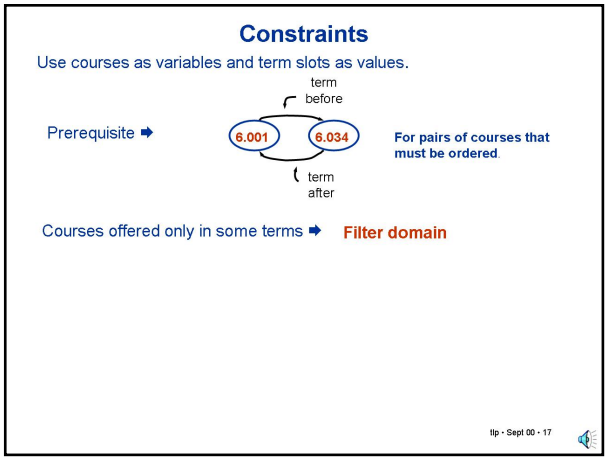
Slide 6.1.16

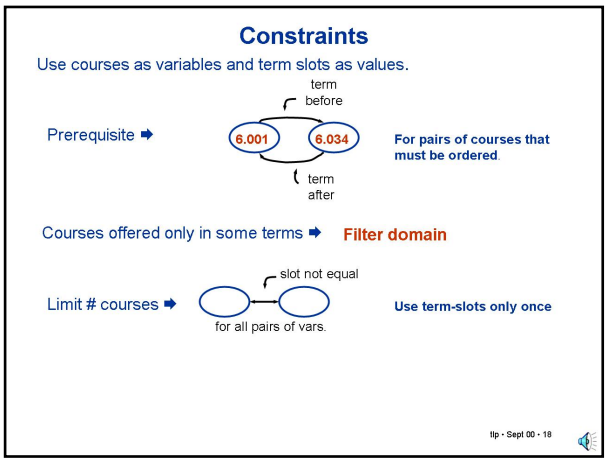
One constraint that must be represented is that the pre-requisites of a class must be taken before the actual class. This is easy to represent in this formulation. We introduce types of constraints called "term before" and "term after" which check that the values assigned to the variables, for example, 6.034 and 6.001, satisfy the correct ordering.

Note that the undirected links shown in prior constraint graphs are now split into two directed links, each with complementary constraints.

Slide 6.1.17

The constraint that some courses are only offered in some terms simply filters illegal term values from the domains of the variables.



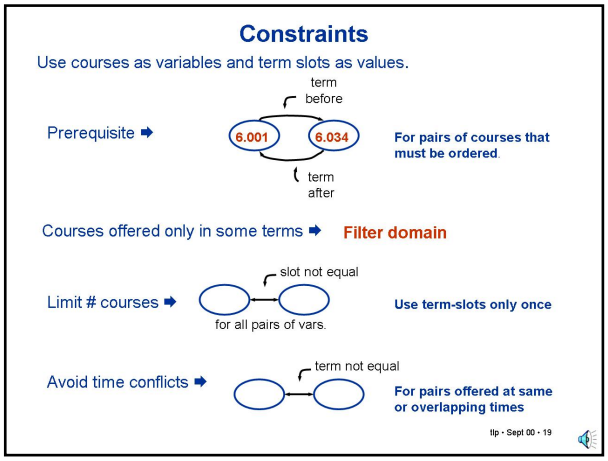


Slide 6.1.18

The limit on courses to be taken in a term argues for the use of term-slots as values rather than just terms. If we use term-slots, then the constraint is implicitly satisfied.

Slide 6.1.19

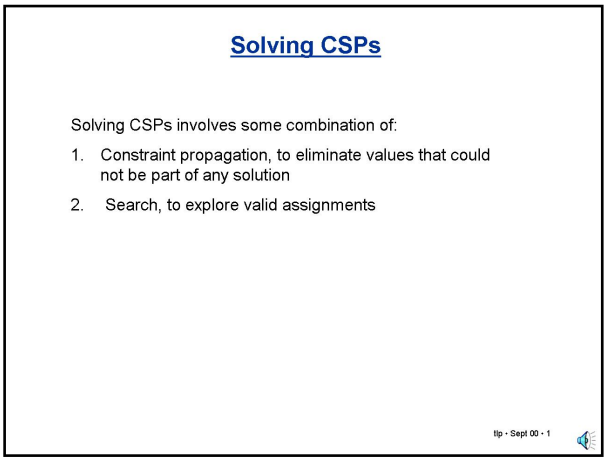
Avoiding time conflicts is also easily represented. If two courses occur at overlapping times then we place a constraint between those two courses. If they overlap in time every term that they are given, we can make sure that they are taken in different terms. If they overlap only on some terms, that can also be enforced by an appropriate constraint.

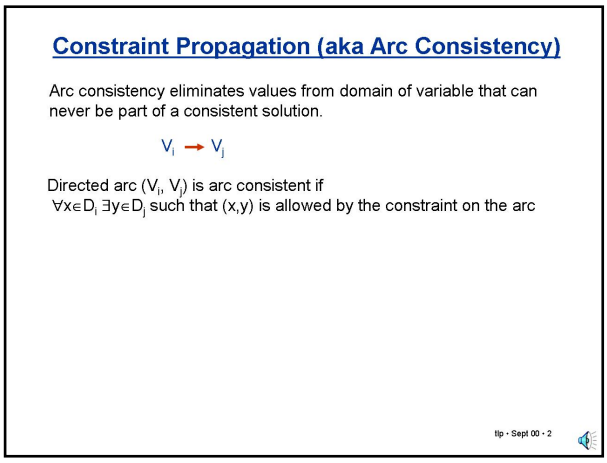


6.034 Notes: Section 6.2

Slide 6.2.1

We now turn our attention to solving CSPs. We will see that the approaches to solving CSPs are some combination of constraint propagation and search. We will look at these in turn and then look at how they can be profitably combined.



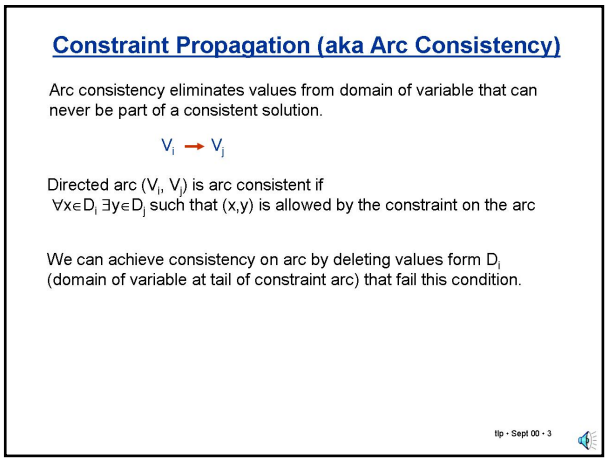


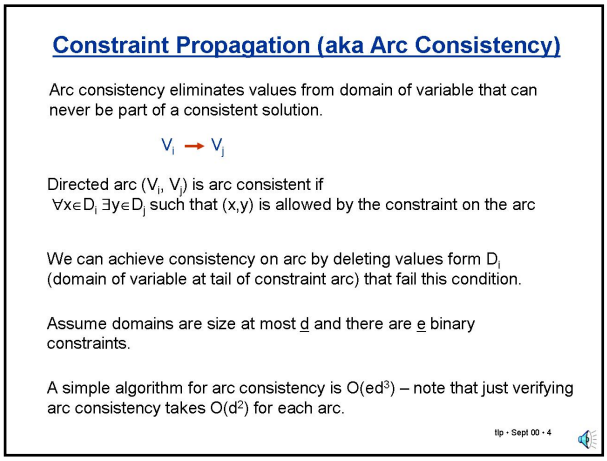
Slide 6.2.2

The great success of Waltz's constraint propagation algorithm focused people's attention on CSPs. The basic idea in constraint propagation is to enforce what is known as "ARC CONSISTENCY", that is, if one looks at a directed arc in the constraint graph, say an arc from V_i to V_j, we say that this arc is consistent if for **every** value in the domain of V_i, there exists **some** value in the domain of V_j that will satisfy the constraint on the arc.

Slide 6.2.3

Suppose there are some values in the domain at the tail of the constraint arc (for V_i) that do not have any consistent partner in the domain at the head of the arc (for V_j). We achieve arc consistency by dropping those values from D_i. Note, however, that if we change D_i, we now have to check to make sure that any other constraint arcs that have D_i at their head are still consistent. It is this phenomenon that accounts for the name "constraint propagation".





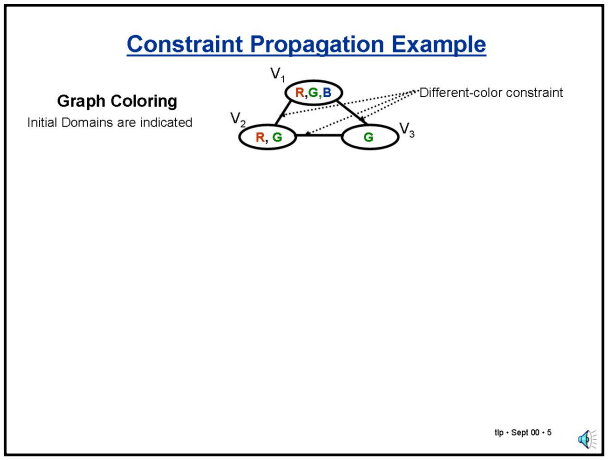
Slide 6.2.4

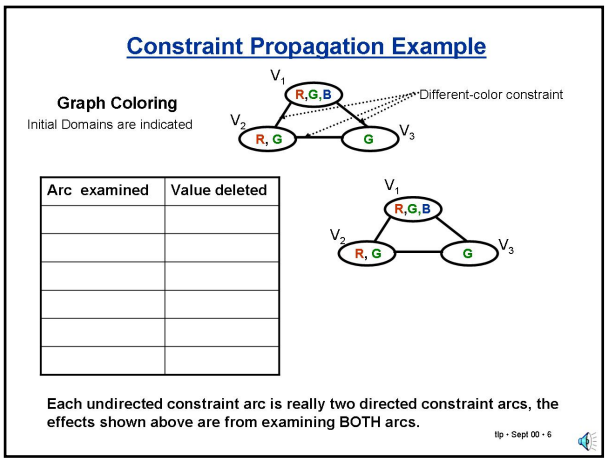
What is the cost of this operation? In what follows we will reckon cost in terms of "arc tests": the number of times we have to check (evaluate) the constraint on an arc for a pair of values in the variable domains of that arc. Assuming that domains have at most *d* elements and that there are at most *e* binary constraints (arcs), then a simple constraint propagation algorithm takes $O(ed^3)$ arc tests in the worst case.

It is easy to see that checking for consistency of each arc for all the values in the corresponding domains takes $O(d^2)$ arc tests, since we have to look at all pairs of values in two domains. Going through and checking each arc once requires $O(ed^2)$ arc tests. But, we may have to go through and look at the arcs more than once as the deletions to a node's domain propagate. However, if we look at an arc only when one of its variable domains has changed (by deleting some entry), then no arc can require checking more than *d* times and we have the final cost of $O(ed^3)$ arc tests in the worst case.

Slide 6.2.5

Let's look at a trivial example of graph coloring. We have three variables with the domains indicated. Each variable is constrained to have values different from its neighbors.



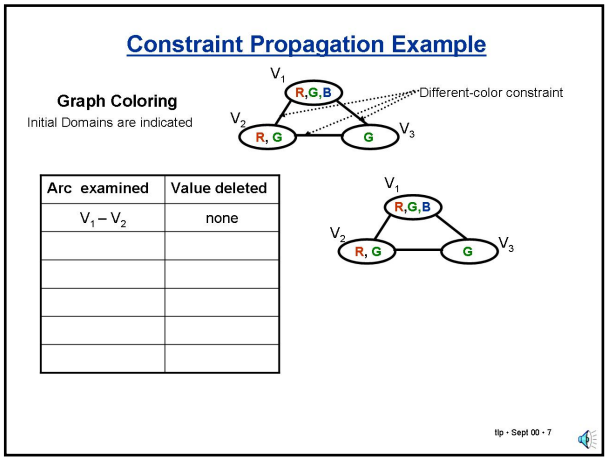


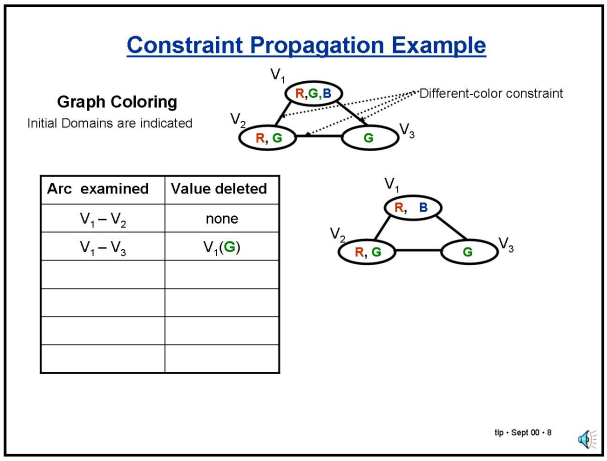
Slide 6.2.6

We will now simulate the process of constraint propagation. In the interest of space, we will deal in this example with undirected arcs, which are just a shorthand for the two directed arcs between the variables. Each step in the simulation involves examining one of these undirected arcs, seeing if the arc is consistent and, if not, deleting values from the domain of the appropriate variable.

Slide 6.2.7

We start with the V_1-V_2 arc. Note that for every value in the domain of V_1 (R, G and B) there is some value in the domain of V_2 that it is consistent with (that is, it is different from). So, for R in V_1 there is a G in V_2, for G in V_1 there is an R in V_2 and for B in V_1 there is either R and G in V_2. Similarly, for each entry in V_2 there is a valid counterpart in V_1. So, the arc is consistent and no changes are made.



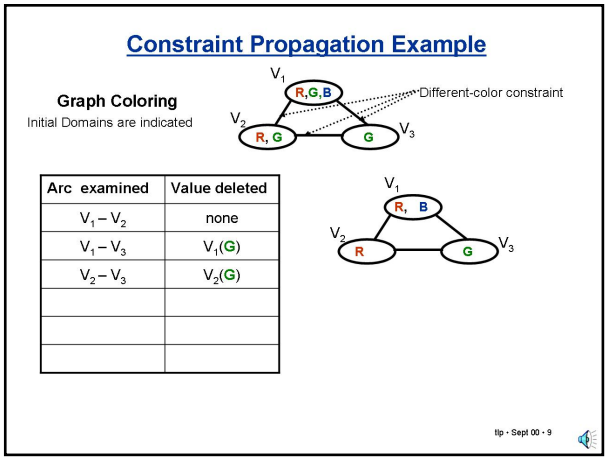


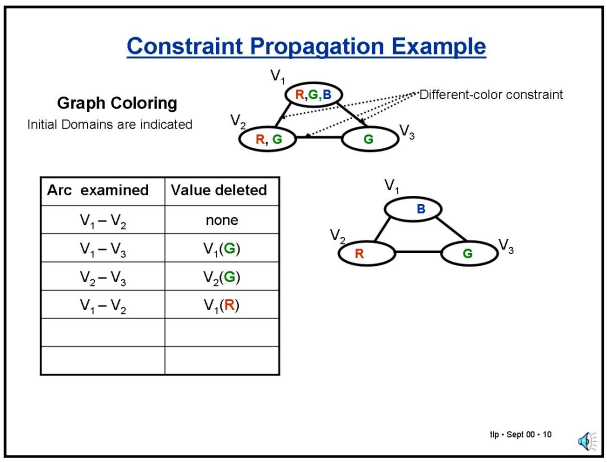
Slide 6.2.8

We move to V_1-V_3. The situation here is different. While R and B in V_1 can co-exist with the G in V_3, not so the G in V_1. And, so, we remove the G from V_1. Note that the arc in the other direction is consistent.

Slide 6.2.9

Moving to V_2-V_3, we note similarly that the G in V_2 has no valid counterpart in V_3 and so we drop it from V_2's domain. Although we have now looked at all the arcs once, we need to keep going since we have changed the domains for V_1 and V_2.



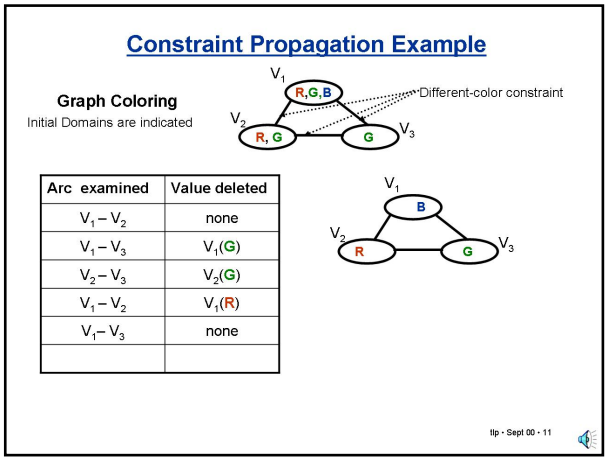


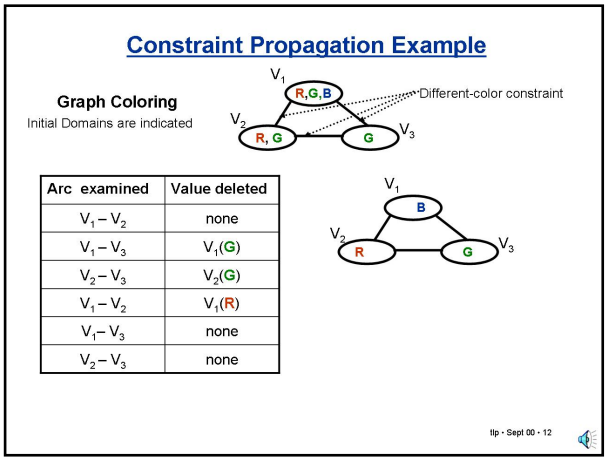
Slide 6.2.10

Looking at V_1-V_2 again we note that R in V_1 no longer has a valid counterpart in V_2 (since we have deleted G from V_2) and so we need to drop R from V_1.

Slide 6.2.11

We test V_1-V_3 and it is consistent.





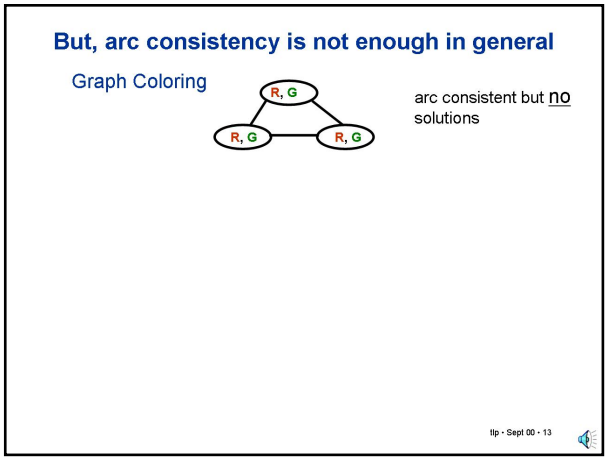
Slide 6.2.12

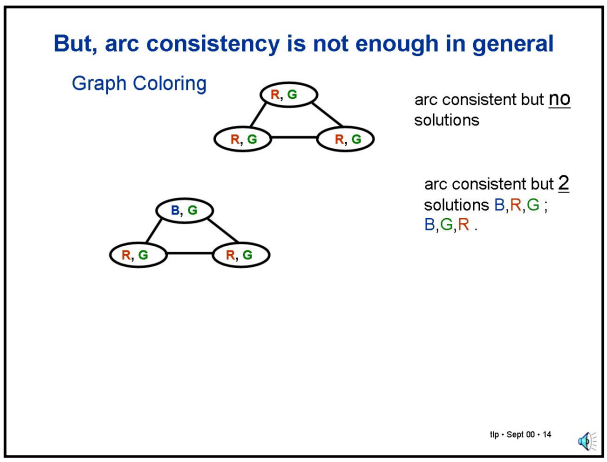
We test V_2-V_3 and it is consistent.

We are done; the graph is arc consistent. In general, we will need to make one pass through any arc whose head variable has changed until no further changes are observed before we can stop. If at any point some variable has an empty domain, the graph has no consistent solution.

Slide 6.2.13

Note that whereas arc consistency is required for there to be a solution for a CSP, having an arc-consistent solution is not sufficient to guarantee a unique solution or even any solution at all. For example, this first graph is arc-consistent but there are NO solutions for it (we need at least three colors and have only two).



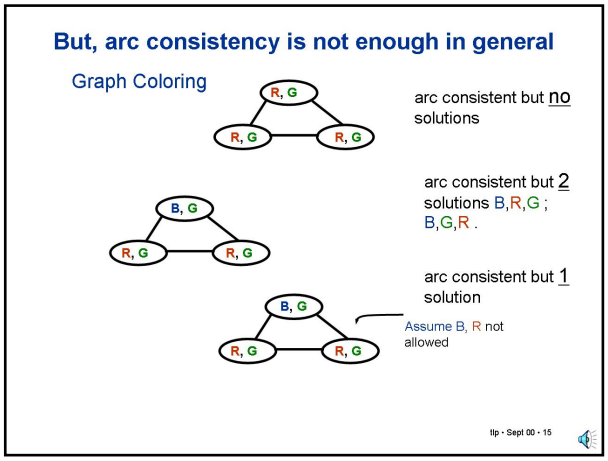


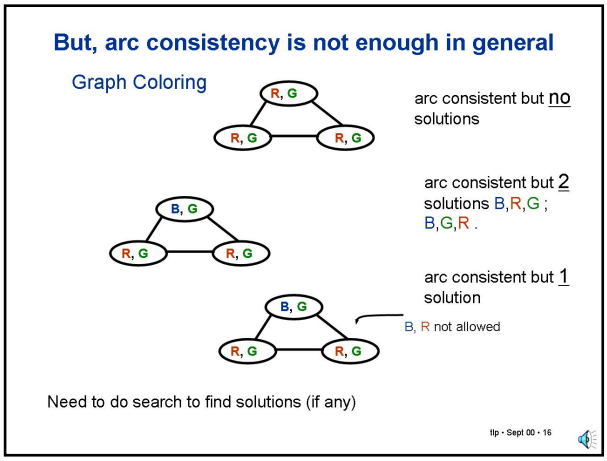
Slide 6.2.14

This next graph is also arc consistent but there are 2 distinct solutions: BRG and BGR.

Slide 6.2.15

This next graph is also arc consistent but it has a unique solution, by virtue of the special constraint between two of the variables.





Slide 6.2.16

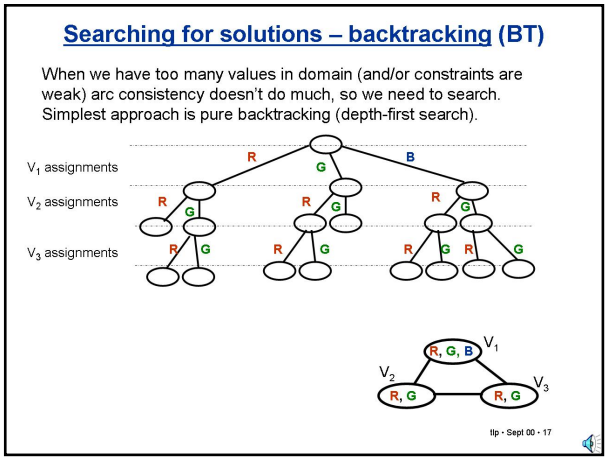
In general, if there is more than one value in the domain of any of the variables, we do not know whether there is zero, one, or more than one answer that is globally consistent. We have to search for an answer to actually know for sure.

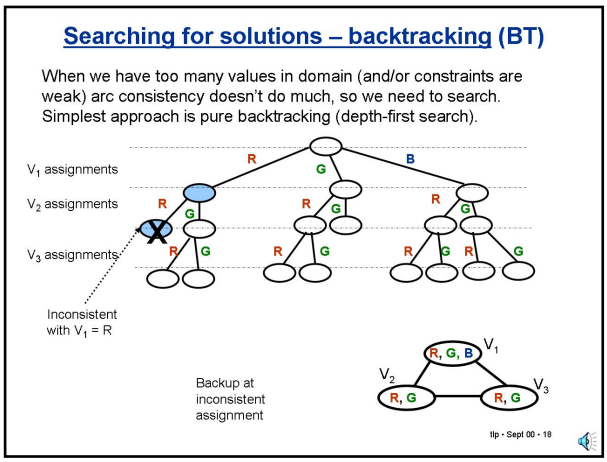
Slide 6.2.17

How does one search for solutions to a CSP problem? Any of the search methods we have studied is applicable. All we need to realize is that the space of assignments of values to variables can be viewed as a tree in which all the assignments of values to the first variable are descendants of the first node and all the assignments of values to the second variable form the descendants of those nodes and so forth.

The classic approach to searching such a tree is called "backtracking", which is just another name for depth-first search in this tree. Note, however, that we could use breadth-first search or any of the heuristic searches on this problem. The heuristic value could be used to either guide the search to termination or bias it to a desired solution based on preferences for certain assignments. Uniform-Cost and A* would make sense also if there were a non-uniform cost associated with a particular assignment of a value to a variable (note that this is another (better but more expensive) way of incorporating preferences).

However, you should observe that these CSP problems are different from the graph search problems we looked at before, in that we don't really care about the path to some state but just the final state itself.



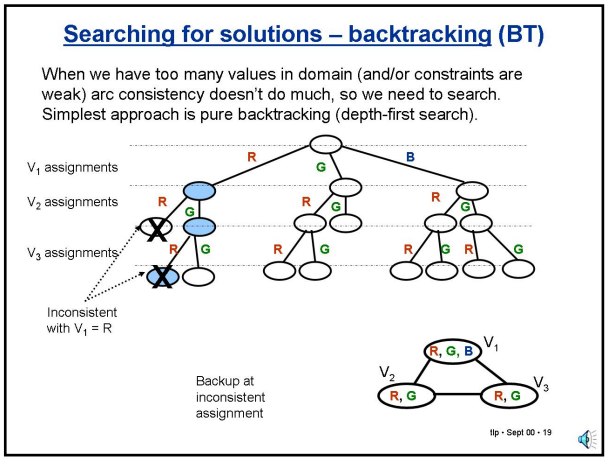


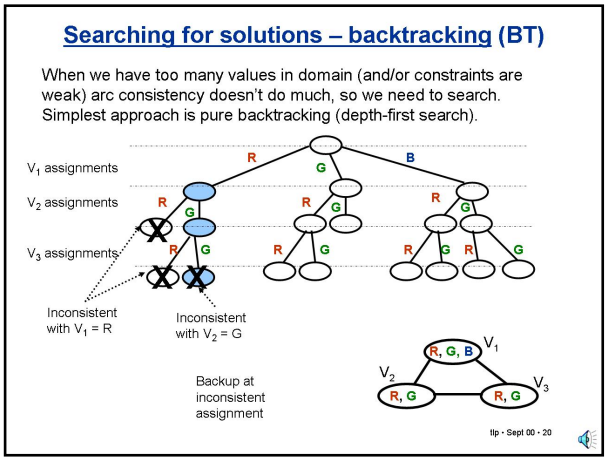
Slide 6.2.18

If we undertake a DFS in this tree, going left to right, we first explore assigning R to V_1 and then move to V_2 and consider assigning R to it. However, for any assignment, we need to check any constraints involving previous assignments in the tree. We note that V_2=R is inconsistent with V_1=R and so that assignment fails and we have to backup to find an alternative assignment for the most recently assigned variable.

Slide 6.2.19

So, we consider assigning V_2=G, which is consistent with the value for V_1. We then move to V_3=R. Since we have a constraint between V_1 and V_3, we have to check for consistency and find it is not consistent, and so we backup to consider another value for V_3.



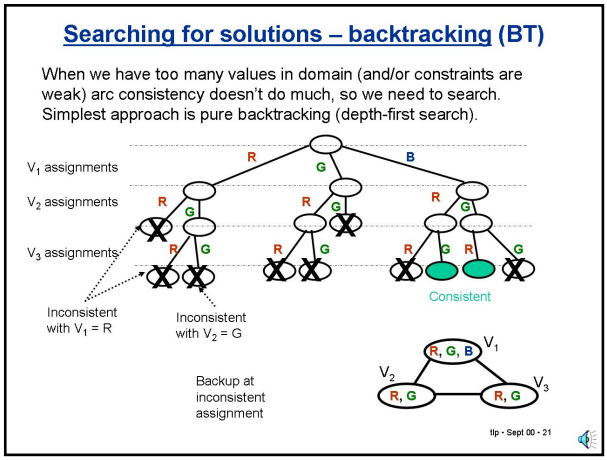


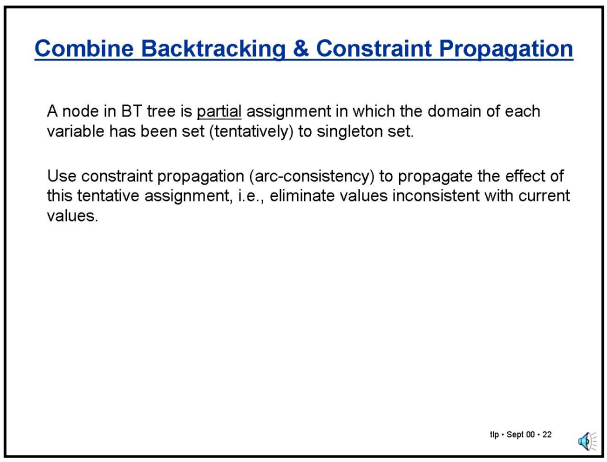
Slide 6.2.20

But V_3=G is inconsistent with V_2=G, and so we have to backup. But there are no more pending values for V_3 or for V_2 and so we fail back to the V_1 level.

Slide 6.2.21

The process continues in that fashion until we find a solution. If we continue past the first success, we can find all the solutions for the problem (two in this case).



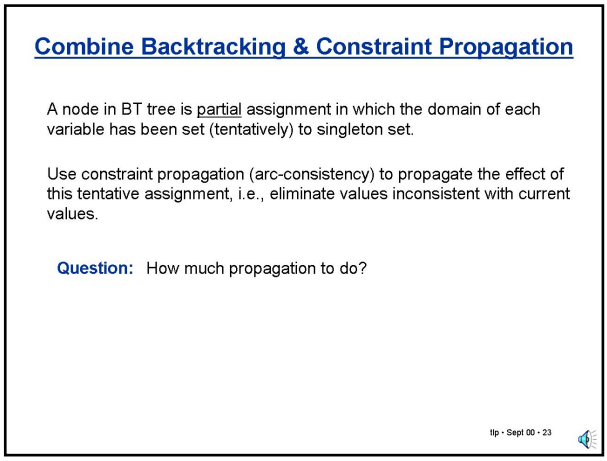


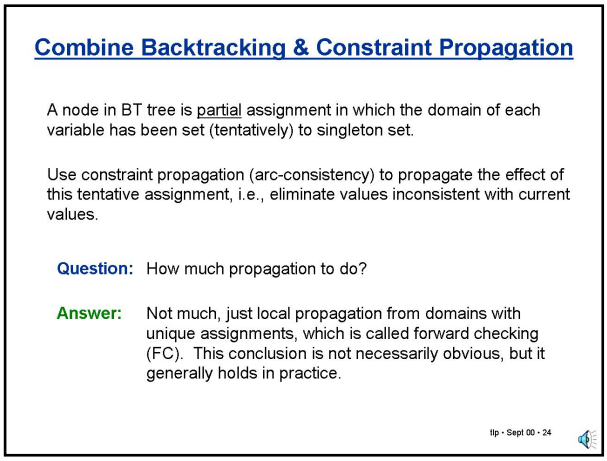
Slide 6.2.22

We can use some form of backtracking search to solve CSP independent of any form of constraint propagation. However, it is natural to consider combining them. So, for example, during a backtracking search where we have a partial assignment, where a subset of all the variables each has unique values assigned, we could then propagate these assignments throughout the constraint graph to obtain reduced domains for the remaining variables. This is, in general, advantageous since it decreases the effective branching factor of the search tree.

Slide 6.2.23

But, how much propagation should we do? Is it worth doing the full arc-consistency propagation we described earlier?





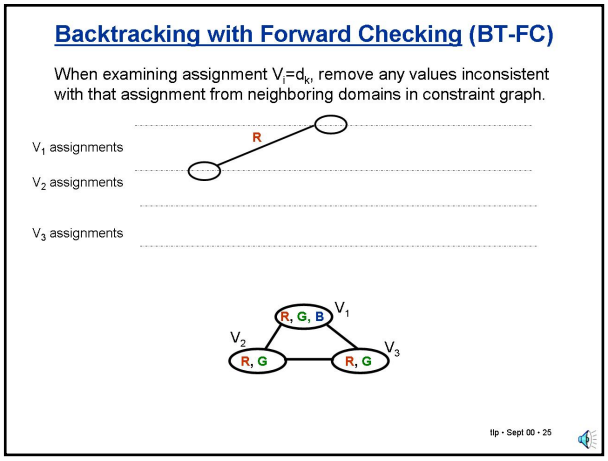
Slide 6.2.24

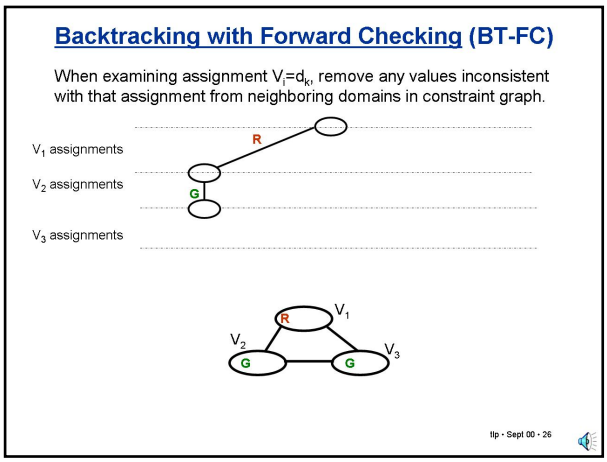
The answer is USUALLY no. It is generally sufficient to only propagate to the immediate neighbors of variables that have unique values (the ones assigned earlier in the search). That is, we eliminate from consideration any values for future variables that are inconsistent with the values assigned to past variables. This process is known as **forward checking** (FC) because one checks values for future variables (forward in time), as opposed to standard backtracking which checks value of past variables (backwards in time, hence back-checking).

When the domains at either end of a constraint arc each have multiple legal values, odds are that the constraint is satisfied, and so checking the constraint is usually a waste of time. This conclusion suggests that forward checking is usually as much propagation as we want to do. This is, of course, only a rule of thumb.

Slide 6.2.25

Let's step through a search that uses a combination of backtracking with forward checking. We start by considering an assignment of V_1=R.



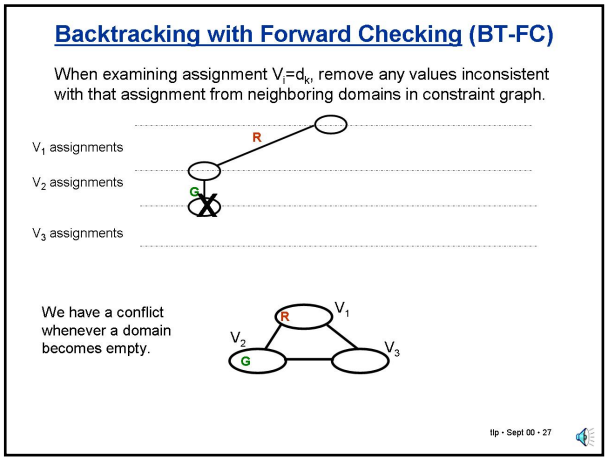


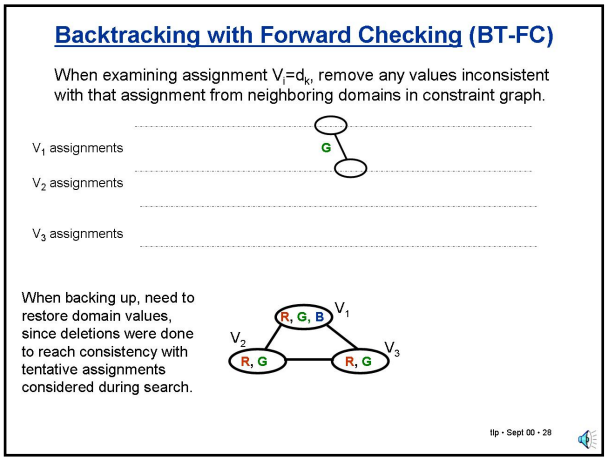
Slide 6.2.26

We then propagate to the neighbors of V_1 in the constraint graph and eliminate any values that are inconsistent with that assignment, namely the value R. That leaves us with the value G in the domains of V_2 and V_3. So, we make the assignment V_2=G and propagate.

Slide 6.2.27

But, when we propagate to V_3 we see that there are no remaining valid values and so we have found an inconsistency. We fail and backup. Note that we have failed much earlier than with simple backtracking, thus saving a substantial amount of work.



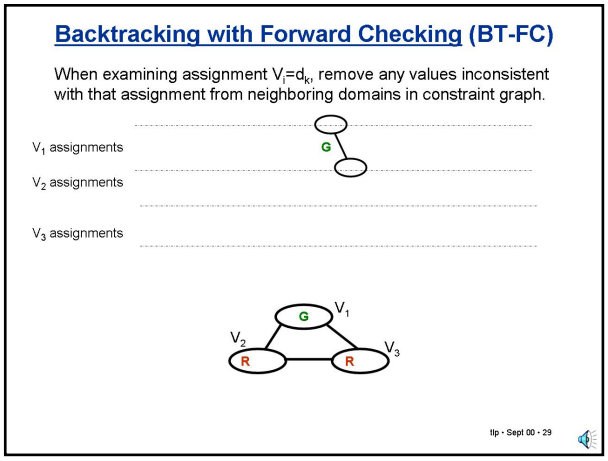


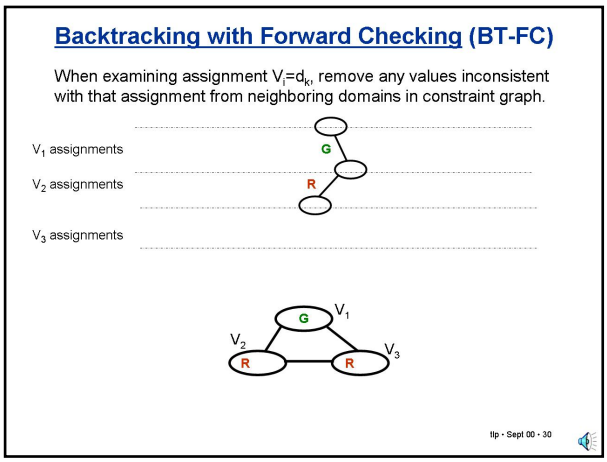
Slide 6.2.28

We now consider V_1=G and propagate.

Slide 6.2.29

That eliminates G from V_2 and V_3.



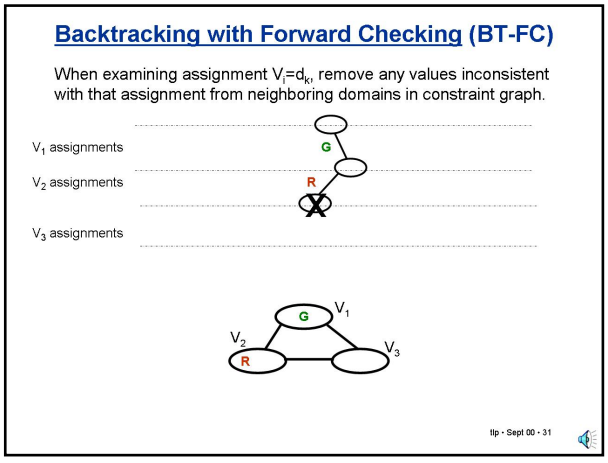


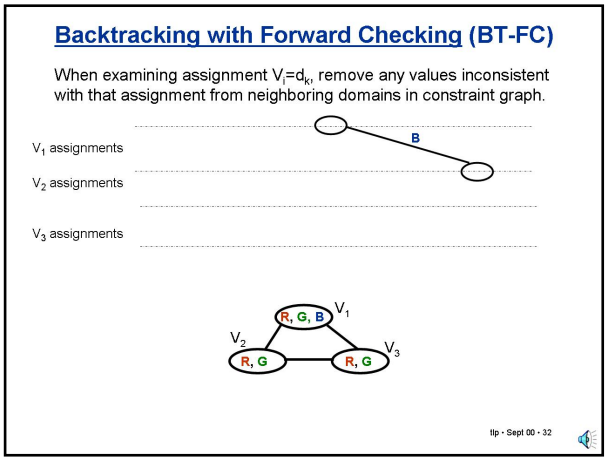
Slide 6.2.30

We now consider V_2=R and propagate.

Slide 6.2.31

The domain of V_3 is empty, so we fail and backup.



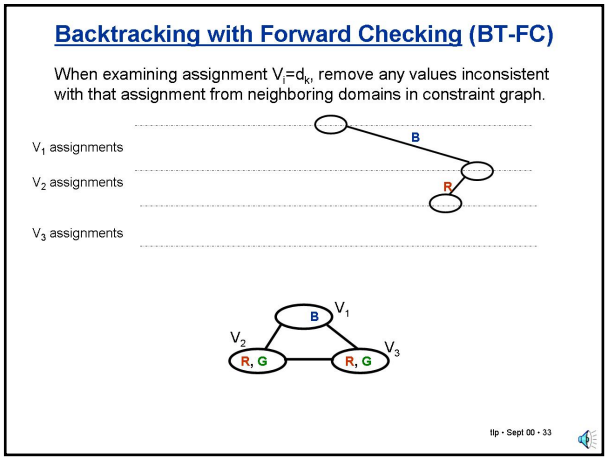


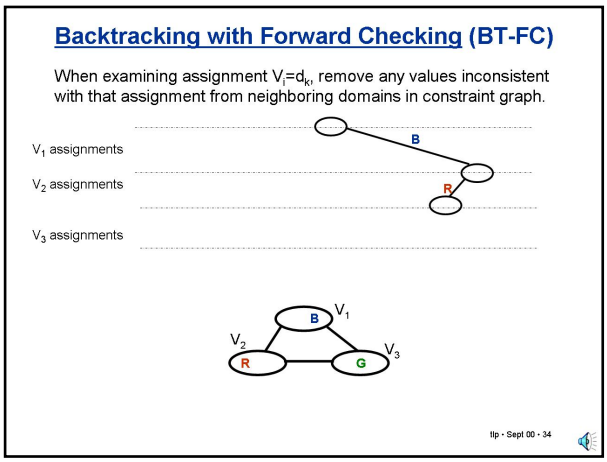
Slide 6.2.32

So, we move to consider V_1=B and propagate.

Slide 6.2.33

This propagation does not delete any values. We pick V_2=R and propagate.



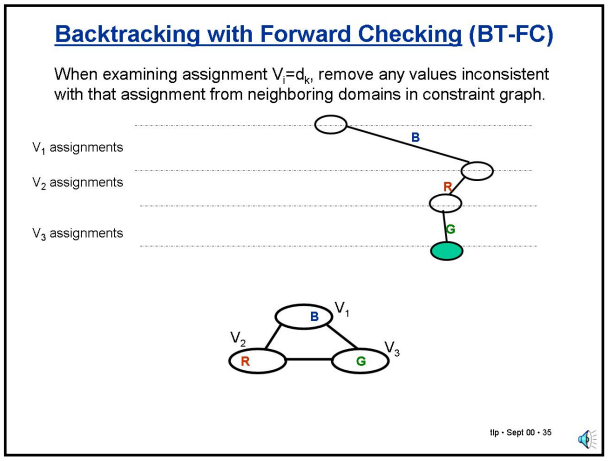


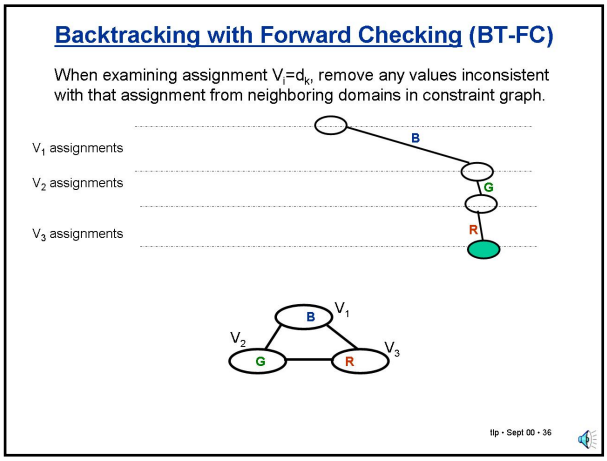
Slide 6.2.34

This removes the R values in the domains of V_1 and V_3.

Slide 6.2.35

We pick V_3 = G and have a consistent assignment.



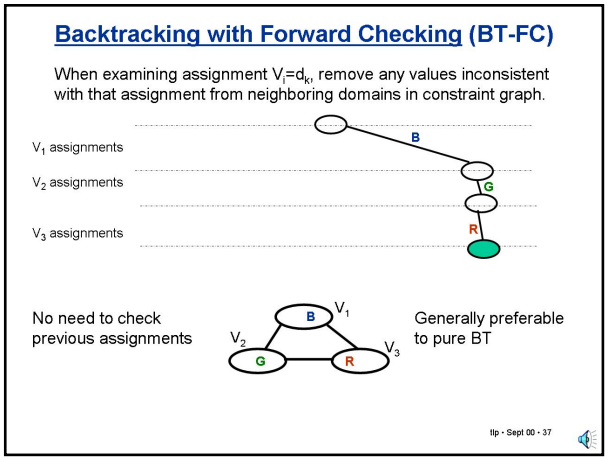


Slide 6.2.36

We can continue the process to find the other consistent solution.

Slide 6.2.37

Note that when doing forward checking there is no need to check new assignments against previous assignments. Any potential inconsistencies have been removed by the propagation. BT-FC is usually preferable to plain BT because it eliminates from consideration inconsistent assignments once and for all rather than discovering the inconsistency over and over again in different parts of the tree. For example, in pure BT, an assignment for V_3 that is inconsistent with a value of V_1 would be "discovered" independently for every value of V_2. Whereas FC would delete it from the domain of V_3 right away.

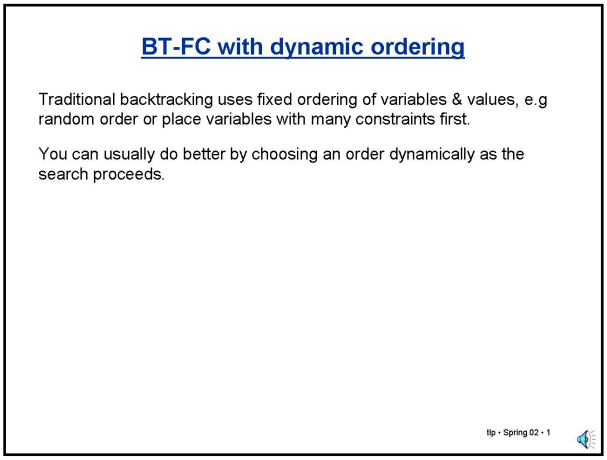


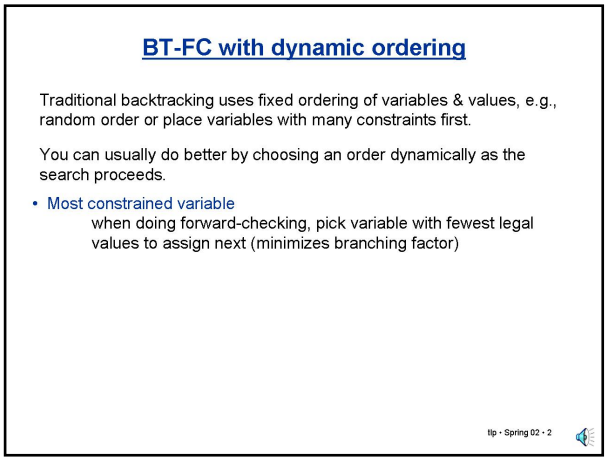
6.034 Notes: Section 6.3

Slide 6.3.1

We have been assuming that the order of the variables is given by some arbitrary ordering. However, the order of the variables (and values) can have a substantial effect on the cost of finding the answer. Consider, for example, the course scheduling problem using courses given in the order that they should ultimately be taken and assume that the term values are ordered as well. Then a depth first search will tend to find the answer very quickly.

Of course, we generally don't know the answer to start off with, but there are more rational ways of ordering the variables than alphabetical or numerical order. For example, we could order the variables before starting by how many constraints they have. But, we can do even better by dynamically re-ordering variables based on information available during a search.





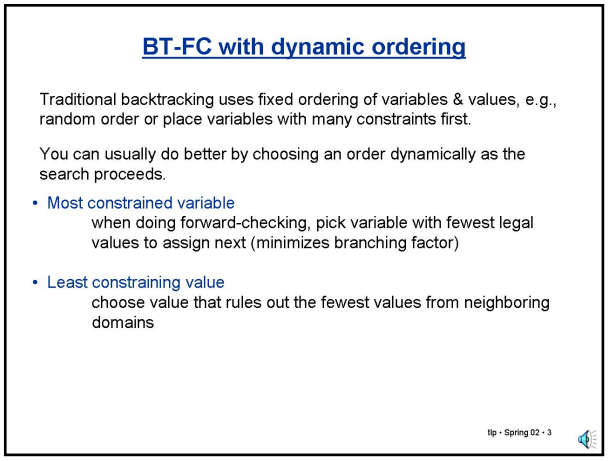
Slide 6.3.2

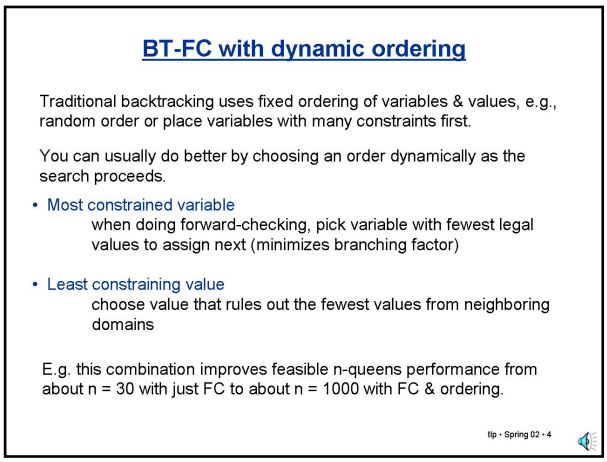
For example, assume we are doing backtracking with forward checking. At any point, we know the size of the domain of each variable. We can order the variables below that point in the search tree so that the most constrained variable (smallest valid domain) is next. This will have the effect of reducing the average branching factor in the tree and also cause failures to happen sooner.

Slide 6.3.3

Furthermore, we can count for each value of the variable the impact on the domains of its neighbors, for example the minimum of the resulting domains after propagation. The value with the largest minimum resulting domain size (or average value or sum) would be one that least constrains the remaining choices and is least likely to lead to failure.

Of course, value ordering is only worth doing if we are looking for a single answer to the problem. If we want all answers, then all values will have to be tried eventually.



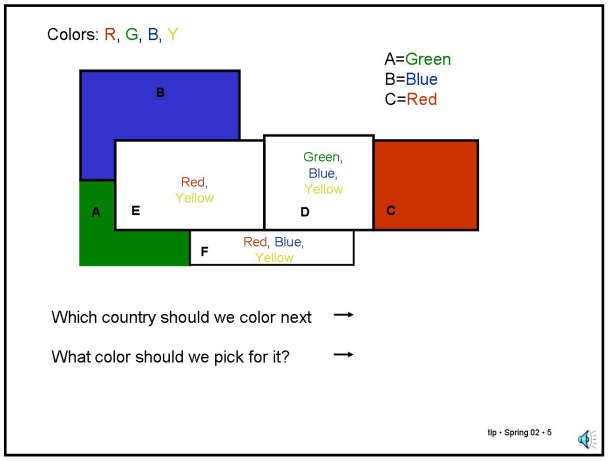


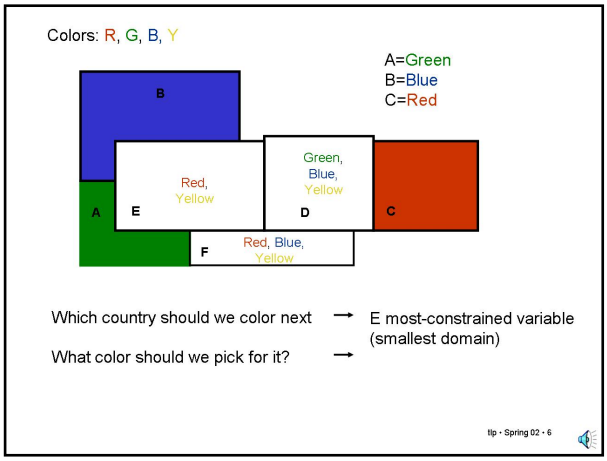
Slide 6.3.4

This combination of variable and value ordering can have dramatic impact on some problems.

Slide 6.3.5

This example of the 4-color map-coloring problem illustrates a simple situation for variable and value ordering. Here, A is colored Green, B is colored Blue and C is colored Red. What country should we color next, D or E or F?



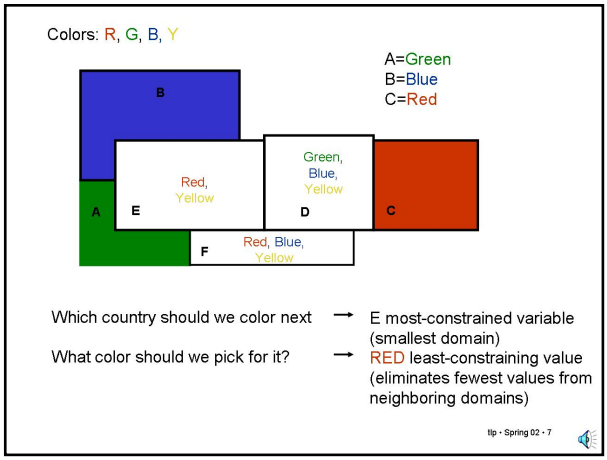


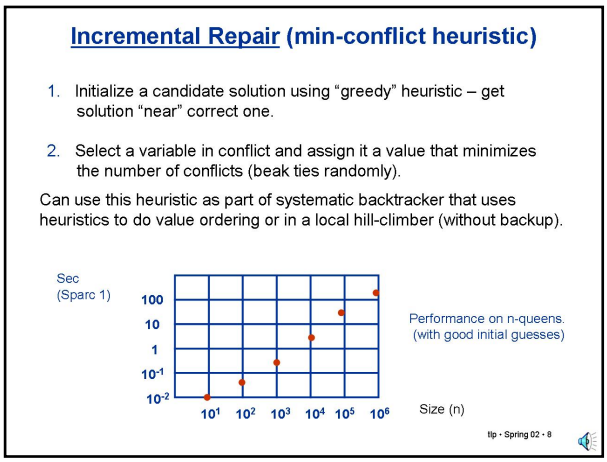
Slide 6.3.6

Well, E is more constrained (has fewer) legal values so we should try it next. Which of E's values should we try next?

Slide 6.3.7

By picking RED, we keep open the most options for D and F, so we pick that.





Slide 6.3.8

All of the methods for solving CSPs that we have discussed so far are systematic (guaranteed searches). More recently, researchers have had surprising success with methods that are not systematic (they are randomized) and do not involve backup.

The basic idea is to do incremental repair of a nearly correct assignment. Imagine we had some heuristic that could give us a "good" answer to any of the problems. By "good" we mean one with relatively few constraint violations. In fact, this could even be a randomly chosen solution.

Then, we could take the following approach. Identify a random variable involved in some conflict. Pick a new value for that variable that minimizes the number of resulting conflicts. Repeat.

This is a type of local "greedy" search algorithm.

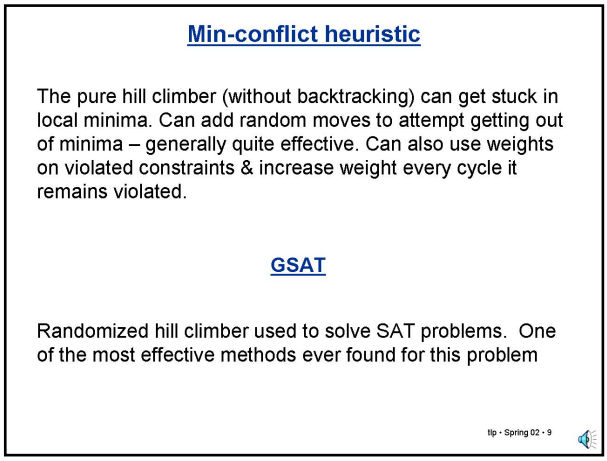
There are variants of this strategy that use this heuristic to do value ordering within a backtracking search. Remarkably, this type of ordering (in connection with a good initial guess) leads to remarkable behavior for benchmark problems. Notably, the systematic versions of this strategy can solve the million-queen problem in minutes. After this, people decided N-queens was not interesting...

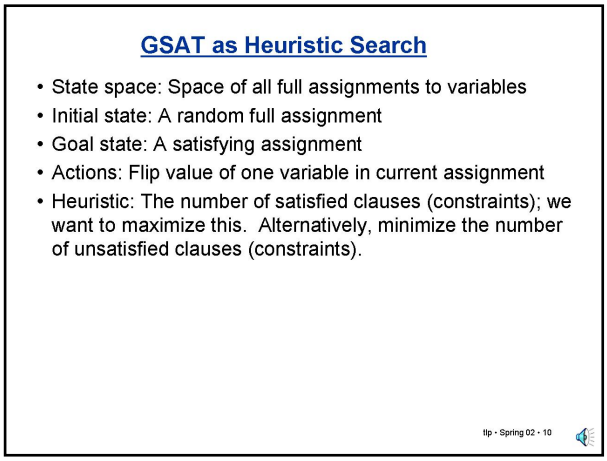
Slide 6.3.9

The pure "greedy" hill-climber can readily fail on any problem (by finding a local minimum where any change to a single variable causes the number of conflicts to increase). We'll look at this a bit in the problem set.

There are several ways of trying to deal with local minima. One is to introduce weights on the violated constraints. A simpler one is to re-start the search with another random initial state. This is the approach taken by GSAT, a randomized search process that solves SAT problems using a similar approach to the one described here.

GSAT's performance is nothing short of remarkable. It can solve SAT problems of mind-boggling complexity. It has forced a complete reconsideration of what it means when we say that a problem is "hard". It turns out that for SAT, almost any randomly chosen problem is "easy". There are really hard SAT problems but they are difficult to find. This is an area of active study.



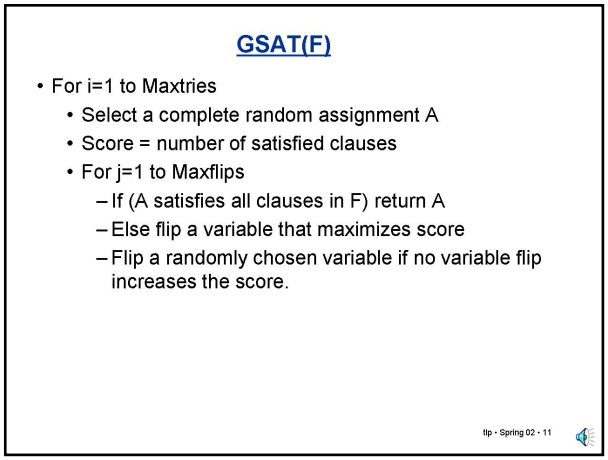


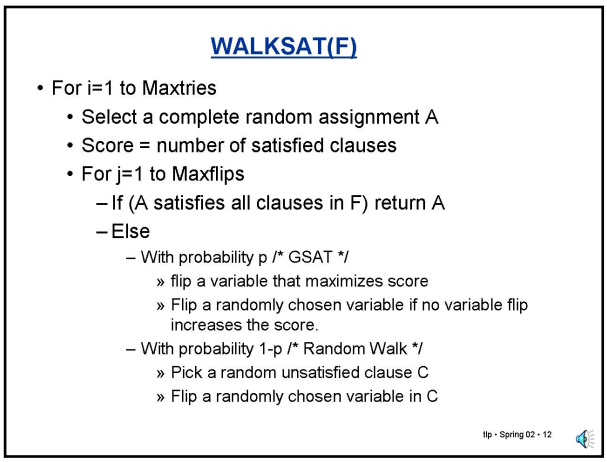
Slide 6.3.10

GSAT can be framed as a heuristic search strategy. Its state space is the space of all full assignments to the variables. The initial state is a random assignment, while the goal state is any assignment that satisfies the formula. The actions available to GSAT are simply to flip one variable in the assignment from true to false or vice-versa. The heuristic value used for the search, which GSAT tries to maximize, is the number of satisfied clauses (constraints). Note that this is equivalent to minimizing the number of conflicts, that is, violated constraints.

Slide 6.3.11

Here we see the GSAT algorithm, which is very simple in sketch. The critical implementation challenge is that of finding quickly the variable whose flip maximizes the score. Note that there are two user-specified variables: the number of times the outer loop is executed (MAXTRIES) and the number of times the inner loop is executed (MAXFLIPS). These parameters guard against local minima in the search, simply by starting with a new, randomly chosen assignment and trying a different sequence of flips. As we have mentioned, this works surprisingly well.





Slide 6.3.12

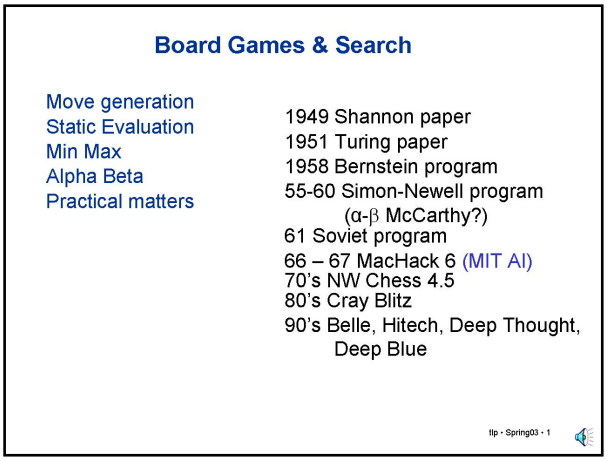
An even more effective strategy turns out to add even more randomness. WALKSAT basically performs the GSAT algorithm some percentage of the time and the rest of the time it does a random walk in the space of assignments by randomly flipping variables in unsatisfied clauses (constraints).

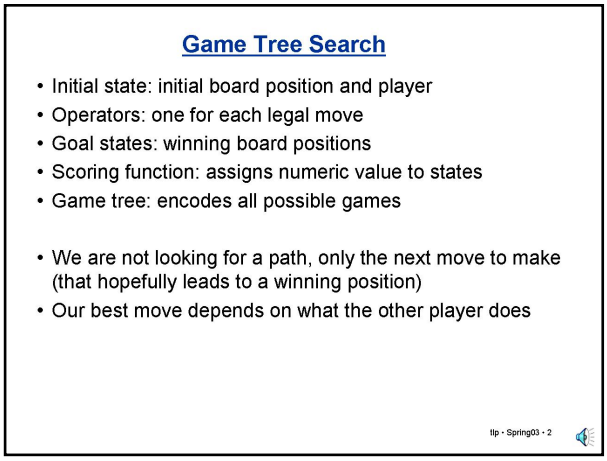
It's a bit depressing to think that such simple randomized strategies can be so much more effective than clever deterministic strategies. There are signs at present that some of the clever deterministic strategies are becoming competitive or superior to the randomized ones. The story is not over.

6.034 Notes: Section 6.4

Slide 6.4.1
In this section, we will look at some of the basic approaches for building programs that play two-person games such as tic-tac-toe, checkers and chess.

Much of the work in this area has been motivated by playing chess, which has always been known as a "thinking person's game". The history of computer chess goes way back. Claude Shannon, the father of information theory, originated many of the ideas in a 1949 paper. Shortly after, Alan Turing did a hand simulation of a program to play checkers, based on some of these ideas. The first programs to play real chess didn't arrive until almost ten years later, and it wasn't until Greenblatt's MacHack 6 that a computer chess program defeated a good player. Slow and steady progress eventually led to the defeat of reigning world champion Garry Kasparov against IBM's Deep Blue in May 1997.

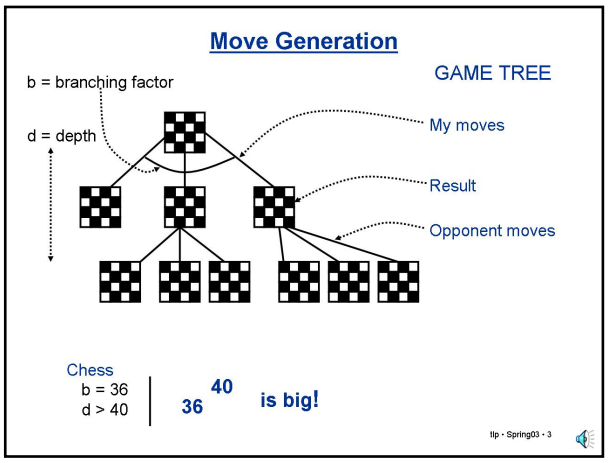


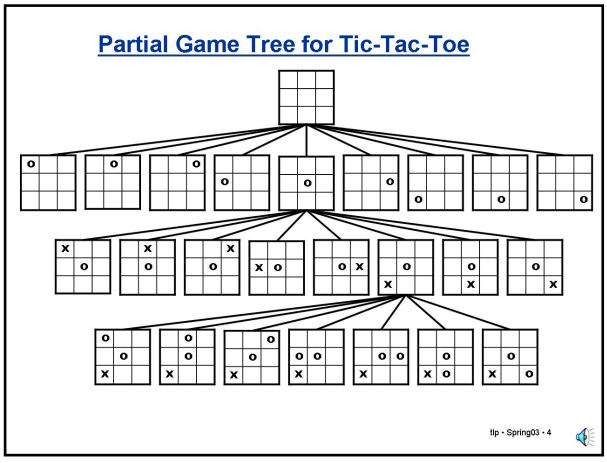


Slide 6.4.2
Game playing programs are another application of search. The states are the board positions (and the player whose turn it is to move). The operators are the legal moves. The goal states are the winning positions. A scoring function assigns values to states and also serves as a kind of heuristic function. The game tree (defined by the states and operators) is like the search tree in a typical search and it encodes all possible games.

There are a few key differences, however. For one thing, we are not looking for a path through the game tree, since that is going to depend on what moves the opponent makes. All we can do is choose the best move to make next.

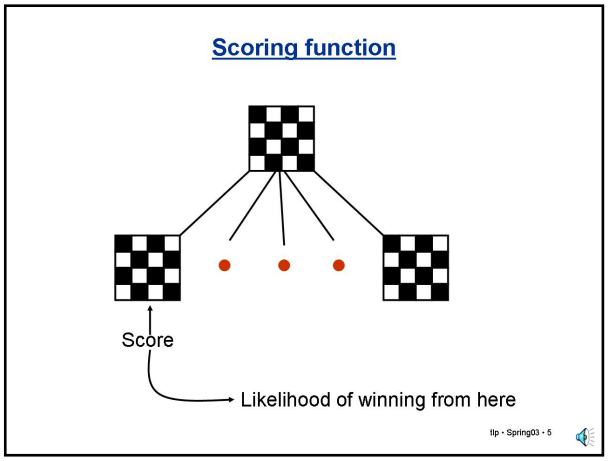
Slide 6.4.3
Let's look at the game tree in more detail. Some board position represents the initial state and it's now our turn. We generate the children of this position by making all of the legal moves available to us. Then, we consider the moves that our opponent can make to generate the descendants of each of these positions, etc. Note that these trees are enormous and cannot be explicitly represented in their entirety for any complex game.

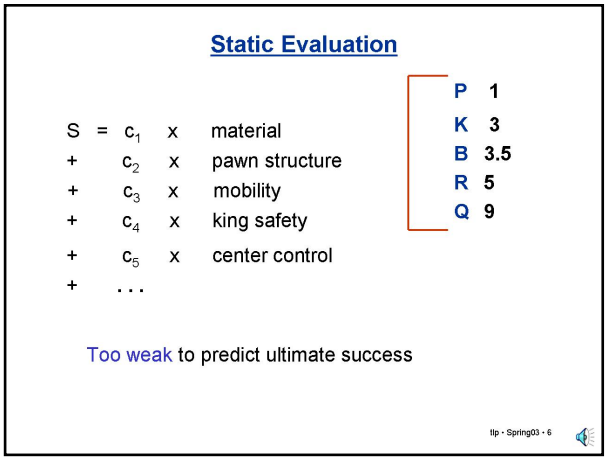




Slide 6.4.4
Here's a little piece of the game tree for Tic-Tac-Toe, starting from an empty board. Note that even for this trivial game, the search tree is quite big.

Slide 6.4.5
A crucial component of any game playing program is the scoring function. This function assigns a numerical value to a board position. We can think of this value as capturing the likelihood of winning from that position. Since in these games one person's win is another's person loss, we will use the same scoring function for both players, simply negating the values to represent the opponent's scores.





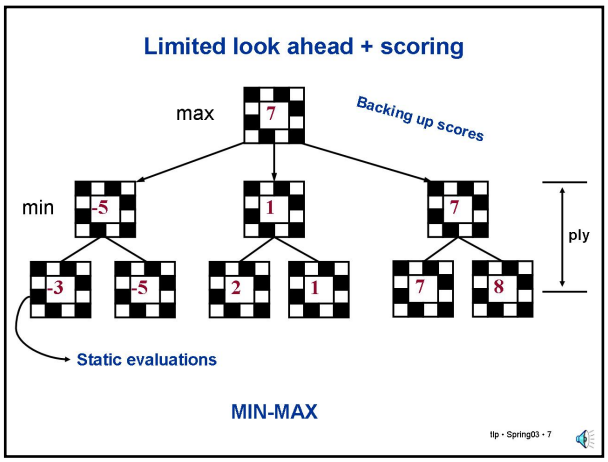
Slide 6.4.6
A typical scoring function is a linear function in which some set of coefficients is used to weight a number of "features" of the board position. Each feature is also a number that measures some characteristic of the position. One that is easy to see is "material", that is, some measure of which pieces one has on the board. A typical weighting for each type of chess piece is shown here. Other types of features try to encode something about the distribution of the pieces on the board.

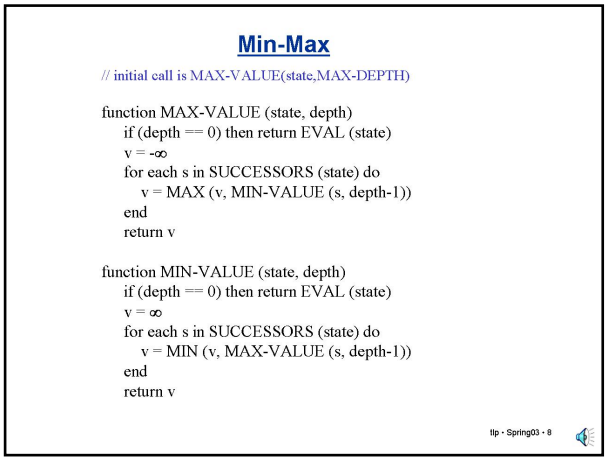
In some sense, if we had a perfect evaluation function, we could simply play chess by evaluating the positions produced by each of our legal moves and picking the one with the highest score. In principle, such a function exists, but no one knows how to write it or compute it directly.

Slide 6.4.7
The key idea that underlies game playing programs (presented in Shannon's 1949 paper) is that of limited look-ahead combined with the Min-Max algorithm.

Let's imagine that we are going to look ahead in the game-tree to a depth of 2 (or 2 ply as it is called in the literature on game playing). We can use our scoring function to see what the values are at the leaves of this tree. These are called the "static evaluations". What we want is to compute a value for each of the nodes above this one in the tree by "backing up" these static evaluations in the tree.

The player who is building the tree is trying to maximize their score. However, we assume that the opponent (who values board positions using the same static evaluation function) is trying to minimize the score (or think of this is as maximizing the negative of the score). So, each layer of the tree can be classified into either a maximizing layer or a minimizing layer. In our example, the layer right above the leaves is a minimizing layer, so we assign to each node in that layer the minimum score of any of its children. At the next layer up, we're maximizing so we pick the maximum of the scores available to us, that is, 7. So, this analysis tells us that we should pick the move that gives us the best guaranteed score, independent of what our opponent does. This is the MIN-MAX algorithm.



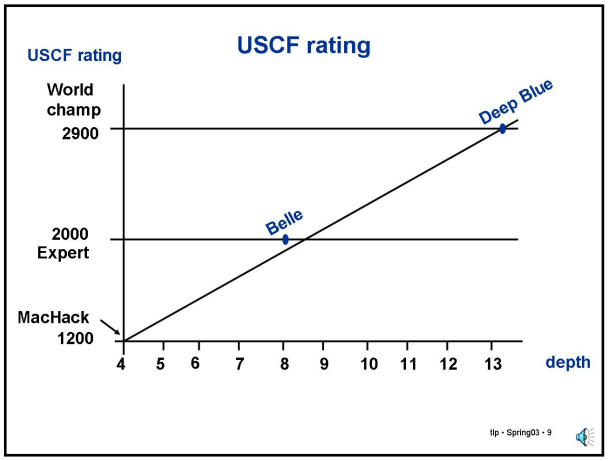


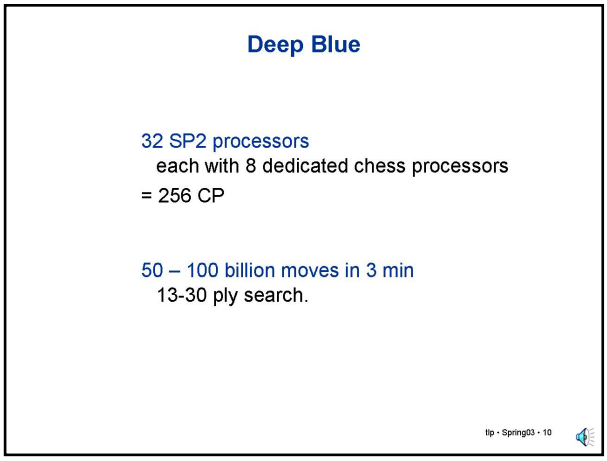
Slide 6.4.8
Here is pseudo-code that implements Min-Max. As you can see, it is a simple recursive alternation of maximization and minimization at each layer. We assume that we count the depth value down from the max depth so that when we reach a depth of 0, we apply our static evaluation to the board.

Slide 6.4.9
The key idea is that the more lookahead we can do, that is, the deeper in the tree we can look, the better our evaluation of a position will be, even with a simple evaluation function. In some sense, if we could look all the way to the end of the game, all we would need is an evaluation function that was 1 when we won and -1 when the opponent won.

The truly remarkable thing is how well this idea works. If you plot how deep computer programs can search chess game trees versus their ranking, we see a graph that looks something like this. The earliest serious chess program (MacHack6), which had a ranking of 1200, searched on average to a depth of 4. Belle, which was one of the first hardware-assisted chess programs doubled the depth and gained about 800 points in ranking. Deep Blue, which searched to an average depth of about 13 beat the world champion with a ranking of about 2900.

At some level, this is a depressing picture, since it seems to suggest that brute-force search is all that matters.



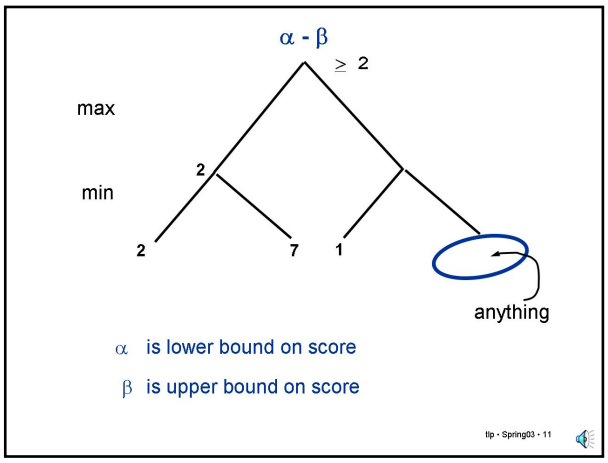


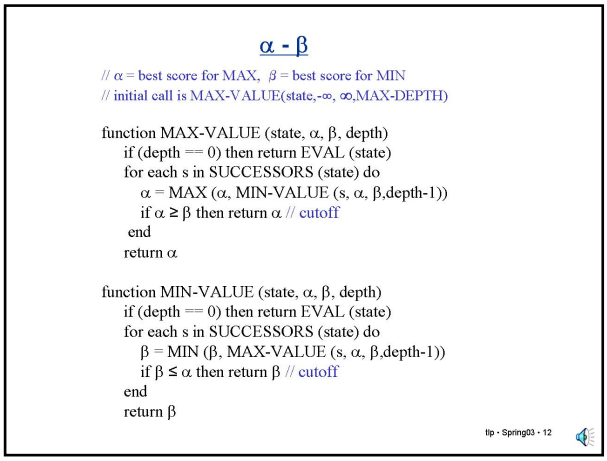
Slide 6.4.10
And Deep Blue is brute indeed... It had 256 specialized chess processors coupled into a 32 node supercomputer. It examined around 30 billion moves per minute. The typical search depth was 13-ply, but in some dynamic situations it could go as deep as 30.

Slide 6.4.11
There's one other idea that has played a crucial role in the development of computer game-playing programs. It is really only an optimization of Min-Max search, but it is such a powerful and important optimization that it deserves to be understood in detail. The technique is called alpha-beta pruning, from the Greek letters traditionally used to represent the lower and upper bound on the score.

Here's an example that illustrates the key idea. Suppose that we have evaluated the sub-tree on the left (whose leaves have values 2 and 7). Since this is a minimizing level, we choose the value 2. So, the maximizing player at the top of the tree knows at this point that he can guarantee a score of at least 2 by choosing the move on the left.

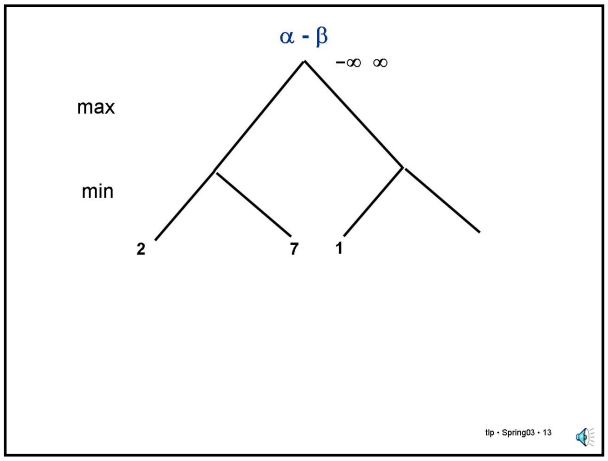
Now, we proceed to look at the subtree on the right. Once we look at the leftmost leaf of that subtree and see a 1, we know that if the maximizing player makes the move to the right then the minimizing player can force him into a position that is worth no more than 1. In fact, it might be much worse. The next leaf we look at might bring an even nastier surprise, but it doesn't matter what it is: we already know that this move is worse than the one to the left, so why bother looking any further? In fact, it may be that this unknown position is a great one for the maximizer, but then the minimizer would never choose it. So, no matter what happens at that leaf, the maximizer's choice will not be affected.

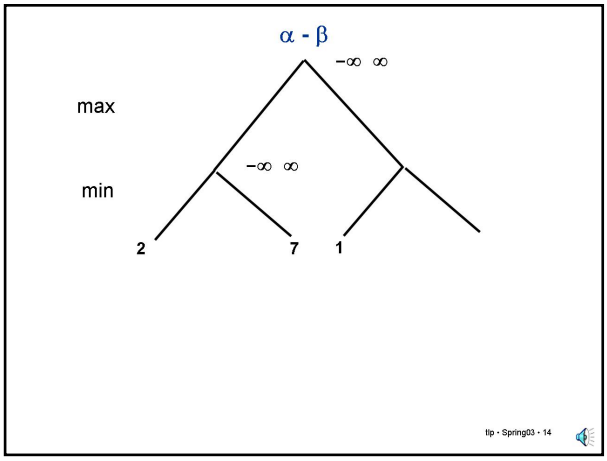




Slide 6.4.12
Here's some pseudo-code that captures this idea. We start out with the range of possible scores (as defined by alpha and beta) going from minus infinity to plus infinity. Alpha represents the lower bound and beta the upper bound. We call Max-Value with the current board state. If we are at a leaf, we return the static value. Otherwise, we look at each of the successors of this state (by applying the legal move function) and for each successor, we call the minimizer (Min-Value) and we keep track of the maximum value returned in alpha. If the value of alpha (the lower bound on the score) ever gets to be greater or equal to beta (the upper bound) then we know that we don't need to keep looking - this is called a cutoff - and we return alpha immediately. Otherwise we return alpha at the end of the loop. The Minimizer is completely symmetric.

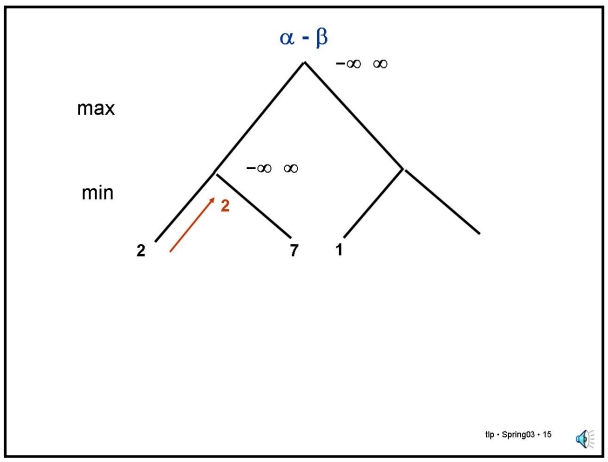
Slide 6.4.13
Lets look at this program in operation on our previous example. We start with an initial call to Max-Value with the initial infinite values of alpha and beta, meaning that we know nothing about what the score is going to be.

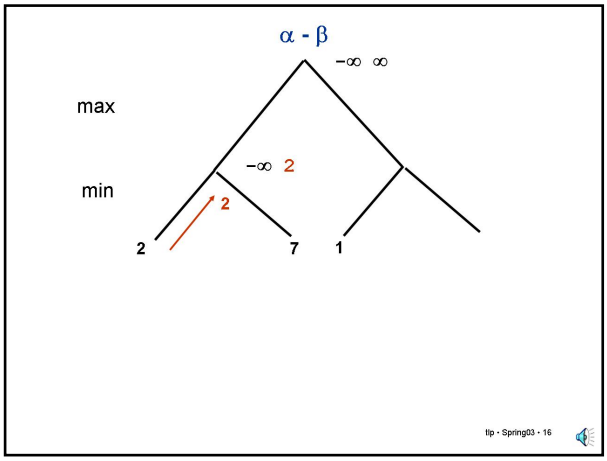




Slide 6.4.14
Max-Value now calls Min-Value on the left successor with the same values of alpha and beta. Min-Value now calls Max-Value on its leftmost successor.

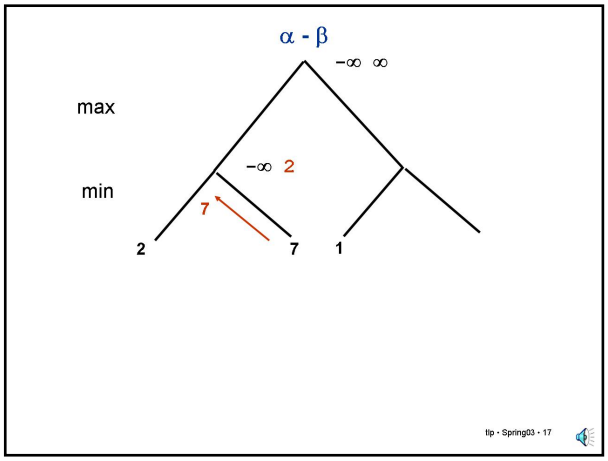
Slide 6.4.15
Max-Value is at the leftmost leaf, whose static value is 2 and so it returns that.

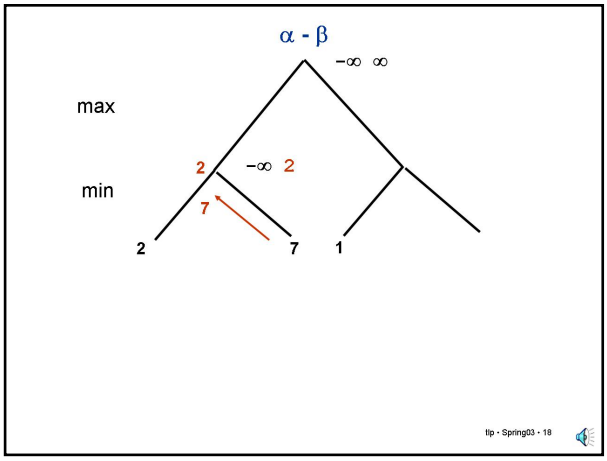




Slide 6.4.16
This first value, since it is less than infinity, becomes the new value of beta in Min-Value.

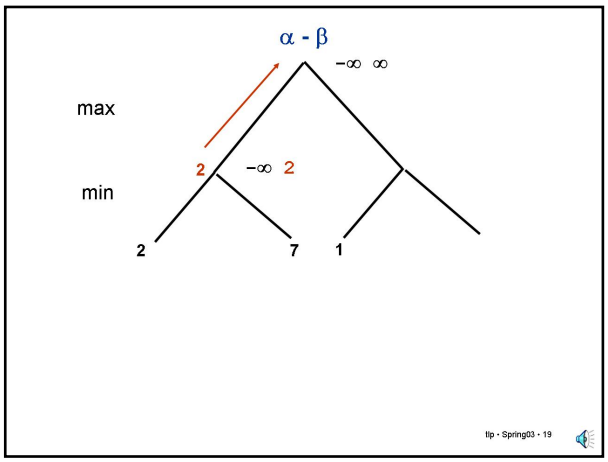
Slide 6.4.17
So, now we call Max-Value with the next successor, which is also a leaf whose value is 7.

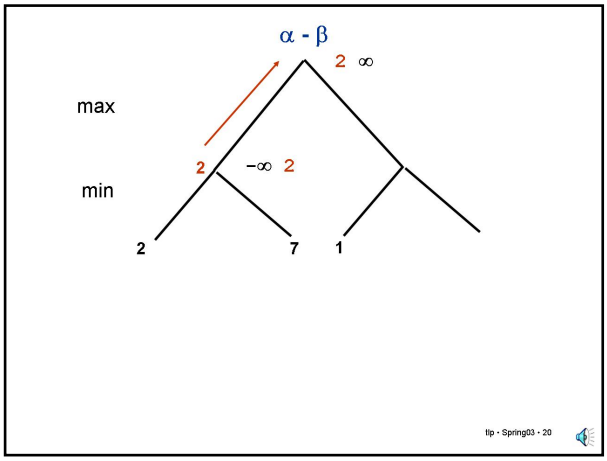




Slide 6.4.18
7 is not less than 2 and so the final value of beta is 2 for this node.

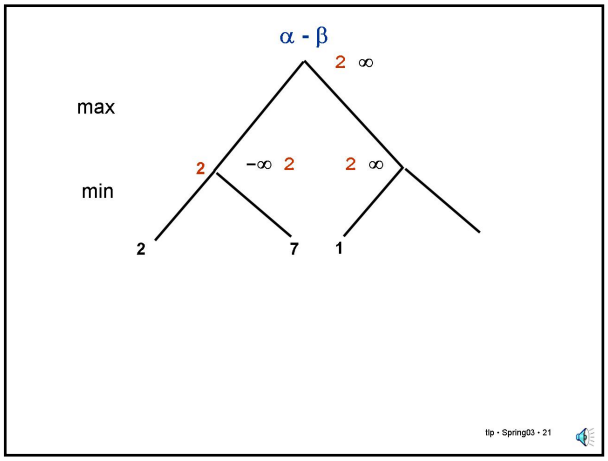
Slide 6.4.19
Min-Value now returns this value to its caller.

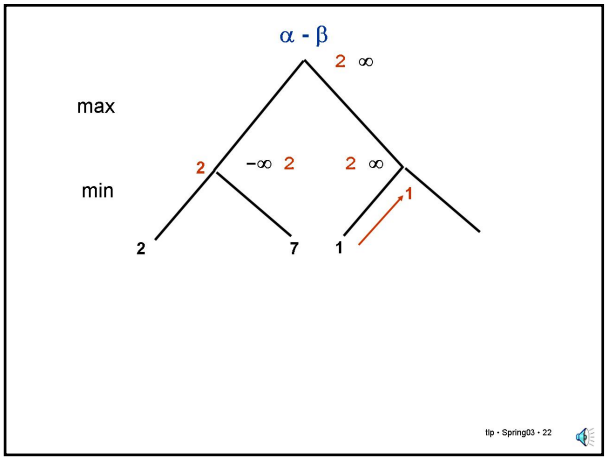




Slide 6.4.20
The calling Max-Value now sets alpha to this value, since it is bigger than minus infinity. Note that the range of [alpha beta] says that the score will be greater or equal to 2 (and less than infinity).

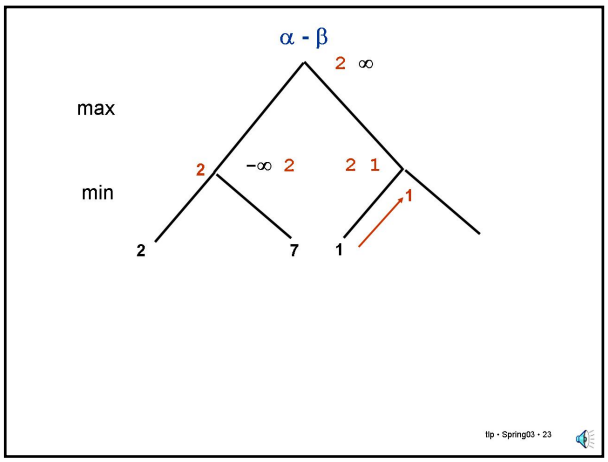
Slide 6.4.21
Max-Value now calls Min-Value with the updated range of [alpha beta].

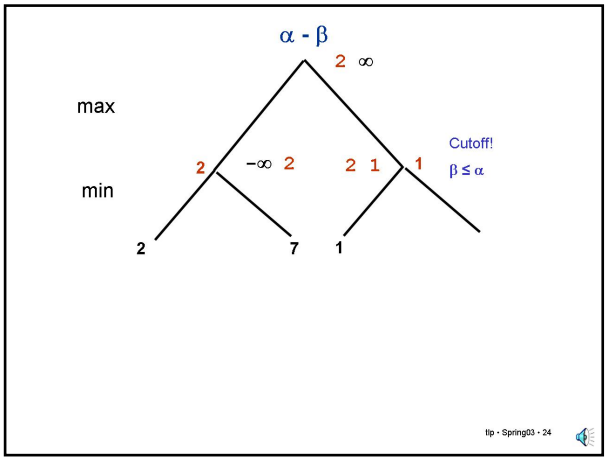




Slide 6.4.22
Min-Value calls Max-Value on the left leaf and it returns a value of 1.

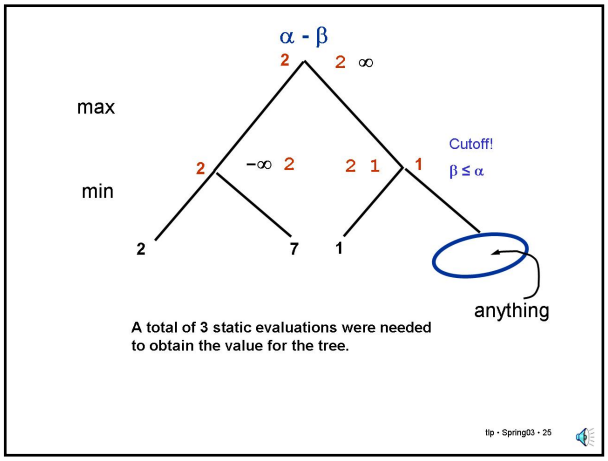
Slide 6.4.23
This is used to update beta in Min-Value, since it is less than infinity. Note that at this point we have a range where alpha (2) is greater than beta (1).

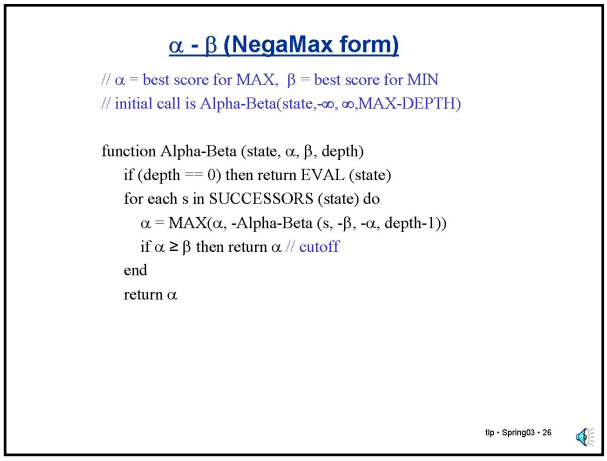




Slide 6.4.24
This situation signals a cutoff in Min-Value and it returns beta (1), without looking at the right leaf.

Slide 6.4.25
So, basically we had already found a move that guaranteed us a score greater or equal to 2 so that when we got into a situation where the score was guaranteed to be less than or equal to 1, we could stop. So, a total of 3 static evaluations were needed instead of the four we would have needed under pure Min-Max.



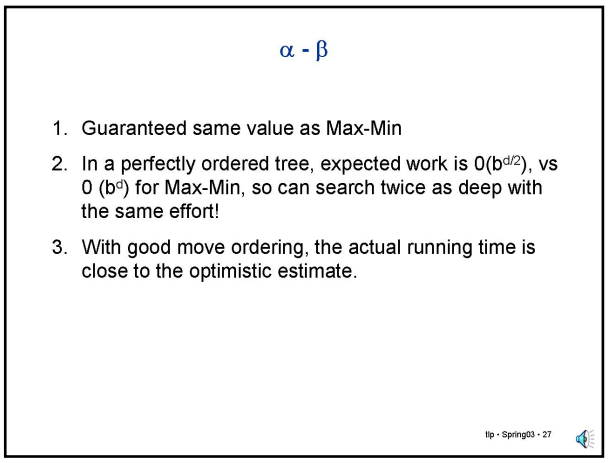


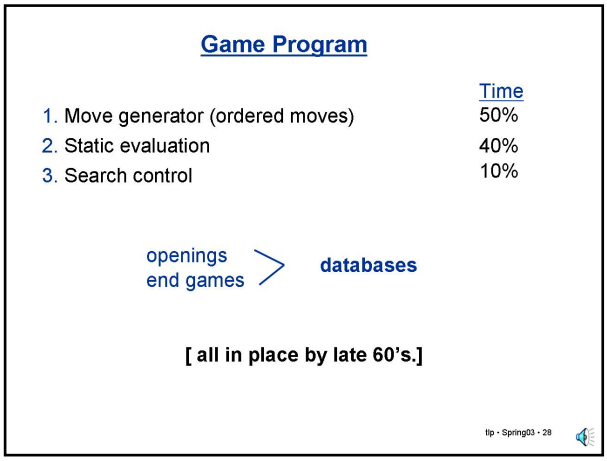
Slide 6.4.26
We can write alpha-beta in a more compact form that captures the symmetry between the Max-Value and Min-Value procedures. This is sometimes called the NegaMax form (instead of the Min-Max form). Basically, this exploits the idea that minimizing is the same as maximizing the negatives of the scores.

Slide 6.4.27
There are a couple of key points to remember about alpha-beta pruning. It is guaranteed to return exactly the same value as the Min-Max algorithm. It is a pure optimization without any approximations or tradeoffs.

In a perfectly ordered tree, with the best moves on the left, alpha beta reduces the cost of the search from order b^d to order $b^{(d/2)}$, that is, we can search twice as deep! We already saw the enormous impact of deeper search on performance. So, this one simple algorithm can almost double the search depth.

Now, this analysis is optimistic, since if we could order moves perfectly, we would not need alpha-beta. But, in practice, performance is close to the optimistic limit.





Slide 6.4.28
If one looks at the time spent by a typical game program, about half the time goes into generating the legal moves ordered (heuristically) in such a way to take maximal advantage of alpha-beta. Most of the remaining time is spent evaluating leaves. Only about 10% is spent on the actual search.

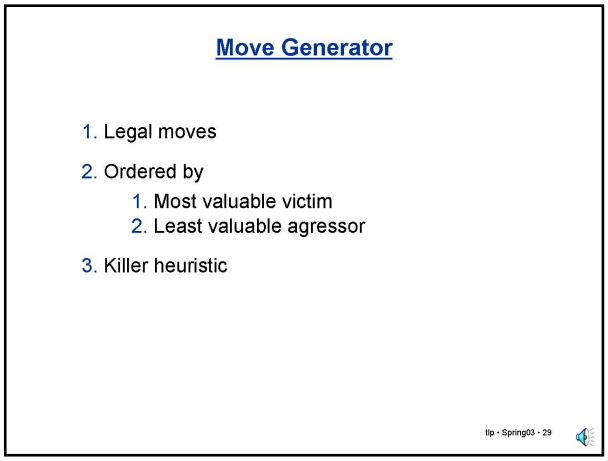
We should note that, in practice, chess programs typically play the first few moves and also complex end games by looking up moves in a database.

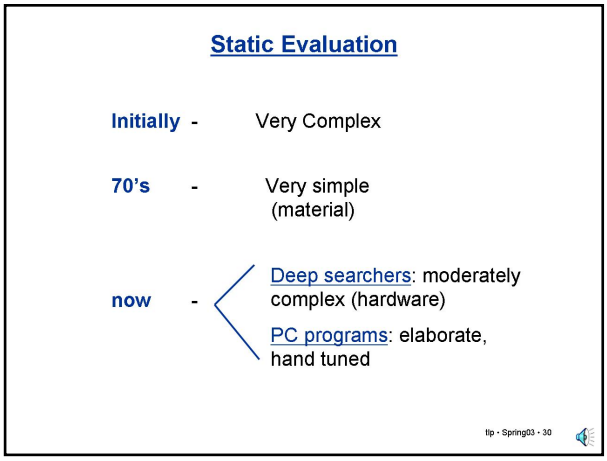
The other thing to note is that all these ideas were in place in MacHack6 in the late 60's. Much of the increased performance has come from increased computer power. The rest of the improvements come from a few other ideas that we'll look at later. First, let's look a bit more of two components that account for the bulk of the time.

Slide 6.4.29
The Move Generator would seem to be an unremarkable component of a game program, and this would be true if its only function were to list the legal moves. In fact, it is a crucial component of the program because its goal is to produce ordered moves. We saw that if the moves are ordered well, then alpha-beta can cutoff much of the search tree without even looking at it. So, the move generator actually encodes a fair bit of knowledge about the game.

There are a few criteria used for ordering the moves. One is to order moves by the value of the captured piece minus the value of the attacker, this is called the "Most valuable victim/Least valuable attacker" ordering, so obviously "pawn-takes-Queen" is the highest ranked move in chess under this ordering.

For non-capture moves, we need other ways of ordering them. One such strategy is known as the "killer heuristic". The basic idea is to keep track of a few moves at each level of the search that cause cutoffs (killer moves) and try them first when considering subsequent moves at that level. Imagine a position with white to move. After white's first move we go into the next recursion of Alpha-Beta and find a move K for black which causes a beta cutoff for black. The reasoning is then that move K is a good move for black, a 'killer'. So when we try the next white move it seems reasonable to try move K first, before all others



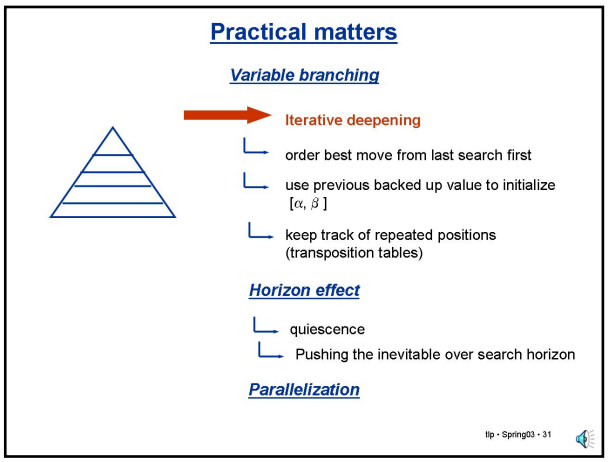


Slide 6.4.30
The static evaluation function is the other place where substantial game knowledge is encoded. In the early chess players, the evaluation functions were very complex (and buggy). Over time it was discovered that using a simple, reliable evaluator (for example, just a weighted count of pieces on the board) and deeper search provided better results. Today, systems such as Deep Blue use static evaluators of medium complexity implemented in hardware. Not surprisingly, the "cheap" PC programs, which can't search as deeply as Deep Blue rely on quite complex evaluation functions. In general, there is a tradeoff between the complexity of the evaluator and the depth of the search.

Slide 6.4.31
As one can imagine in an area that has received as much attention as game playing programs, there are a million and one techniques that have been tried and which make a difference in practice. Here we touch on a couple of the high points.

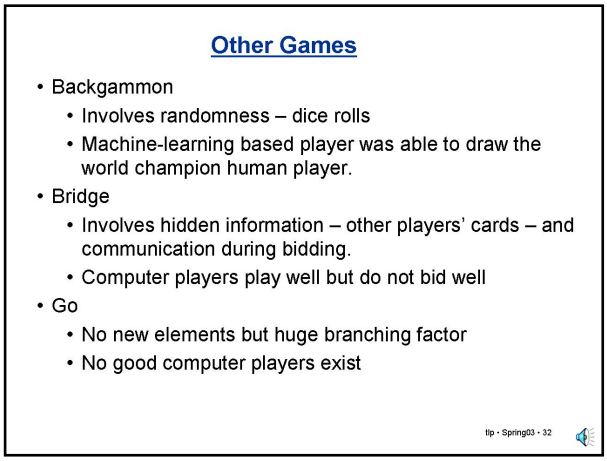
Chess and other such games have incredibly large trees with highly variable branching factor (especially since alpha-beta cutoffs affect the actual branching of the search). If we picked a fixed depth to search, as we've suggested earlier, then much of the time we would finish too quickly and at other times take too long. A better approach is to use iterative deepening and thus always have a move ready and then simply stop after some allotted time.

One of the nice side effects of iterative deepening is that the results of the last iteration of the search can be used to help in the next iteration. For example, we can use the last search to order the moves. A somewhat less obvious advantage is that we can use the previous results to pick an initial value of alpha and beta. Instead of starting with alpha and beta at minus and plus infinity, we can start them in a small window around the values found in the previous search. This will help us cutoff more irrelevant moves early. In fact, it is often useful to start with the tightest possible window, something like [alpha, alpha +epsilon] which is simply asking "is the last move we found still the best move"? In many cases, it is.



Another issue in fixed depth searches is known as the "horizon effect". That is, if we pick a fixed depth search, we could miss something very important right over the horizon. So, it would not do to stop searching right as your queen is about to be captured. Most game programs attempt to assess whether a "leaf" node is in fact "static" or "quiescent" before terminating the search. If the situation looks dynamic, the search is continued. In Deep Blue, as I mentioned earlier, some moves are searched to a depth of 30 ply because of this.

Obviously, Deep Blue makes extensive use of parallelization in its search. This turns out to be surprisingly hard to do effectively and was probably the most significant innovation in Deep Blue.



Slide 6.4.32
In this section, we have focused on chess. There are a variety of other types of games that remain hard today, in spite of the relentless increase in computing power, and other games that require a different treatment.

Backgammon is interesting because of the randomness introduced by the dice. Humans are not so good at building computer players for this directly, but a machine learning system (that essentially did a lot of search and used the results to build a very good evaluation function) was able to draw the human world-champion.

Bridge is interesting because it has hidden information (the other players' cards) and communication with a partner in a restricted language. Computer players, using search, excel now in the card-play phase of the game, but are still not too good at the bidding phase (which involves all the quirks of communication with another human).

Go is actually in the same class of games as chess: there is no randomness, hidden information, or communication. But the branching factor is enormous and it seems not to be susceptible to search-based methods that work well in chess. Go players seem to rely more on a complex understanding of spatial patterns, which might argue for a method that is based more strongly on a good evaluation function than on brute-force search.

Slide 6.4.33
There are a few observations about game playing programs that actually are observations about the whole symbolic approach to AI. The great successes of machine intelligence come in areas where the rules are clear and people have a hard time doing well, for example, mathematics and chess. What has proven to be really hard for AI are the more nebulous activities of language, vision and common sense, all of which evolution has been selecting for. Most of the research in AI today focuses on these less well defined activities.

The other observation is that it takes many years of gradual refinement to achieve human-level competence even in well-defined activities such as chess. We should not expect immediate success in attacking any of the grand challenges of AI.