CAP 4630
Artificial Intelligence

Instructor: Sam Ganzfried

sganzfri@cis.fiu.edu

- http://www.ultimateaiclass.com/
- https://moodle.cis.fiu.edu/
- HW1 out 9/5 today, due 10/3
  - Remember that you have up to 4 late days to use throughout the semester.
- HW2 will go out next week, due 10/17
- Midterm on 10/19
  - Review during half of class on 10/17
- TA office hours:
  - Thursday 3:15-4:15PM, ECS 254

Adversarial search

- This 8×8 variant of draughts (checkers) was **weakly solved** on April 29, 2007 by the team of Jonathan Schaeffer, known for Chinook, the "World Man-Machine Checkers Champion." From the standard starting position, both players can guarantee a draw with perfect play. Checkers is the largest game that has been solved to date, with a search space of 5×10^{20}. The number of calculations involved was 10^{14}, which were done over a period of 18 years. The process involved from 200 desktop computers at its peak down to around 50.

Weakly vs. strongly solved

- **Weak**: Provide an algorithm that secures a win for one player, or a draw for either, against any possible moves by the opponent, from the beginning of the game. That is, produce at least one complete ideal game (all moves start to end) with proof that each move is optimal for the player making it. It does not necessarily mean a computer program using the solution will play optimally against an imperfect opponent. For example, the checkers program Chinook will never turn a drawn position into a losing position (since the weak solution of checkers proves that it is a draw), but it might possibly turn a winning position into a drawn position because Chinook does not expect the opponent to play a move that will not win but could possibly lose, and so it does not analyze such moves completely.

Weakly vs. strongly solved

- **Strong**: Provide an algorithm that can produce perfect moves from any position, even if mistakes have already been made on one or both sides.

- **Ultra-weak**: Prove whether the first player will win, lose or draw from the initial position, given perfect play on both sides. This can be a non-constructive proof (possibly involving a strategy-stealing argument) that need not actually determine any moves of the perfect play.

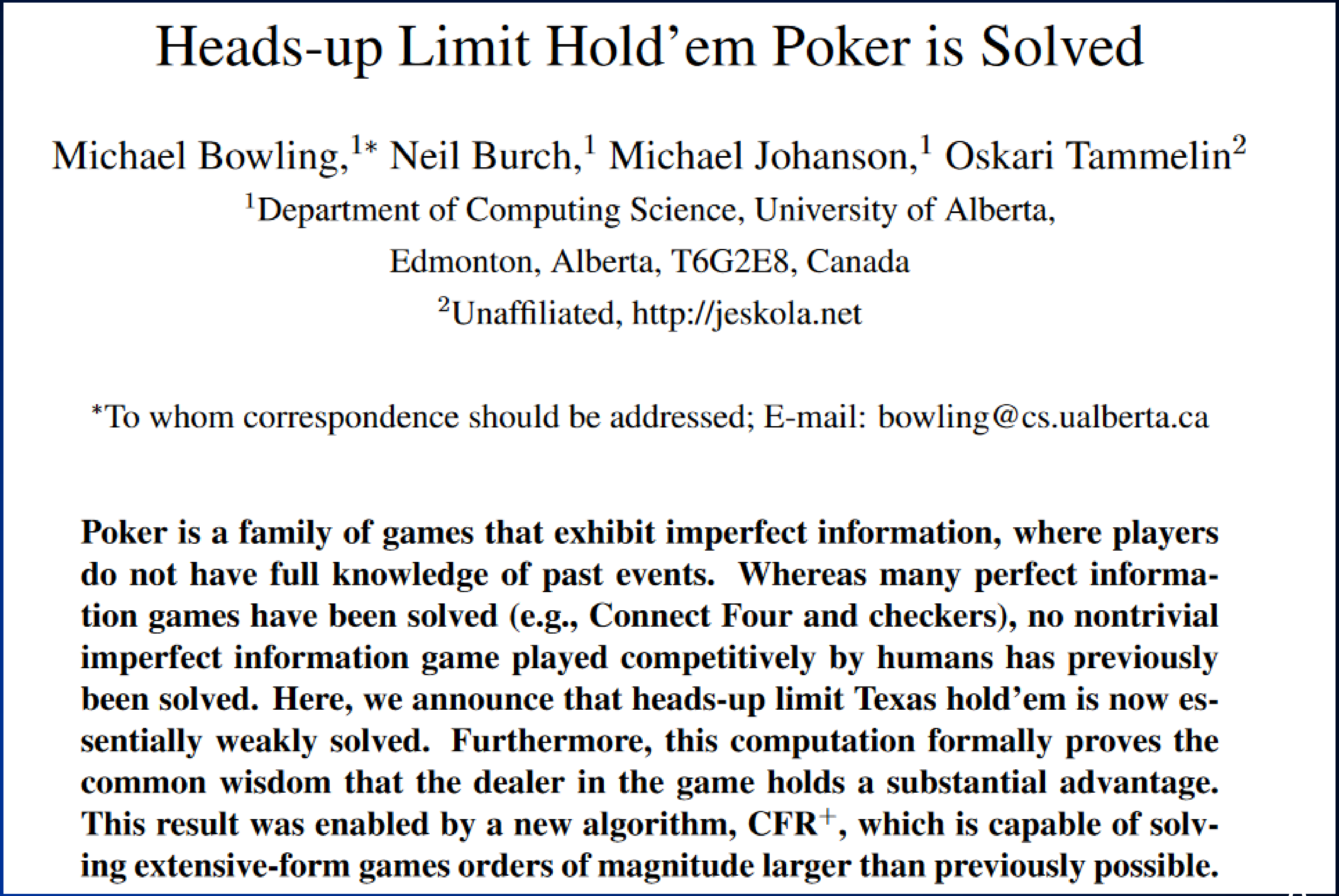
Connect Four

- Solved first by James D. Allen (Oct 1, 1988), and independently by Victor Allis (Oct 16, 1988). First player can force a win. Strongly solved by John Tromp's 8-ply database (Feb 4, 1995). Weakly solved for all boardsizes where width+height is at most 15 (as well as 8×8 in late 2015) (Feb 18, 2006).
- The artificial intelligence algorithms able to strongly solve Connect Four are minimax or negamax, with optimizations that include alpha-beta pruning, move ordering, and transposition tables.

Connect Four

- The solved conclusion for Connect Four is first player win. With perfect play, the first player can force a win, on or before the 41st move (ply) by starting in the middle column. The game is a theoretical draw when the first player starts in the columns adjacent to the center. For the edges of the game board, column 1 and 2 on left (or column 7 and 6 on right), the exact move-value score for first player start is loss on the 40th move, and loss on the 42nd move, respectively. In other words, by starting with the four outer columns, the first player allows the second player to force a win.

2-player limit Hold'em poker is solved (Science 2015)



Heads-up Limit Hold'em Poker is Solved

Michael Bowling,[1*] Neil Burch,[1] Michael Johanson,[1] Oskari Tammelin[2]

[1]Department of Computing Science, University of Alberta,
Edmonton, Alberta, T6G2E8, Canada

[2]Unaffiliated, http://jeskola.net

*To whom correspondence should be addressed; E-mail: bowling@cs.ualberta.ca

Poker is a family of games that exhibit imperfect information, where players do not have full knowledge of past events. Whereas many perfect information games have been solved (e.g., Connect Four and checkers), no nontrivial imperfect information game played competitively by humans has previously been solved. Here, we announce that heads-up limit Texas hold'em is now essentially weakly solved. Furthermore, this computation formally proves the common wisdom that the dealer in the game holds a substantial advantage. This result was enabled by a new algorithm, CFR$^+$, which is capable of solving extensive-form games orders of magnitude larger than previously possible.

Heads-up Limit Hold ‘em Poker is Solved

- Play against Cepheus here http://poker-play.srv.ualberta.ca/

Poker

- **Abstract:** Poker is a family of games that exhibit imperfect information, where players do not have full knowledge of past events. Whereas many perfect-information games have been solved (e.g., Connect Four and checkers), no nontrivial imperfect-information game played competitively by humans has previously been solved. Here, we announce that heads-up limit Texas hold’em is now **essentially weakly solved**. Furthermore, this computation formally proves the common wisdom that the dealer in the game holds a substantial advantage. This result was enabled by a new algorithm, CFR$^+$, which is capable of solving extensive-form games orders of magnitude larger than previously possible.

Adversarial search

- We first consider games with two players, whom we call MAX and MIN. MAX moves first, and then they take turns moving until the game is over. At the end of the game, points are awarded to the winning player, and penalties given to the loser. A game can be formally defined as a kind of search problem with the following elements:

Search problem definition

- States
- Initial state
- Actions
- Transition model
- Goal test
- Path cost

Definition for 8-queens problem

- **States**: Any arrangement of 0 to 8 queens on the board is a state.
- **Initial state**: No queens on the board.
- **Actions**: Add a queen to any empty square.
- **Transition model**: Returns the board with a queen added to the specified square
- **Goal test**: 8 queens are on the board, none attacked
- **Path cost**: (Not applicable)

Game definition

- S_0: the **initial state**, which specifies how the game starts
- PLAYER(s): defines which player has the move in a state
- ACTIONS(s): Returns the set of legal moves in a state
- RESULT(s,a): The **transition model**, which defines the result of a move.
- TERMINAL-TEST(s): A **terminal** test, which is true when the game is over and false otherwise. States where the game has ended are called **terminal states**.
- UTILITY(s,p): A **utility function** (also called an objective function or payoff function), defines the final numeric value for a game that ends in terminal state s for a player p. In chess, the outcome is a win, loss, or draw, with values +1, 0, or ½. Some games have a wider variety of possible outcomes; the payoffs in backgammon range from 0 to +192.

Zero-sum games

- A **zero-sum** game is (confusingly) defined as one where the total payoff to all players is the same for every instance of the game.
- Is chess zero-sum?
- Checkers?
- Poker?

Zero-sum games

- Chess is zero-sum because every game has payoff of either 0 +1, 1+0, or ½ + ½
- “Constant-sum” would have been a better term, but zero-sum is traditional and makes sense if you imagine that each player is charged an entry fee of ½.

Game tree

- The initial state, ACTIONS function, and RESULT function define the **game tree** for the game—a tree where the nodes are game states and the edges are moves. The figure shows part of the game tree for tic-tac-toe. From the initial state, MAX has nine possible moves. Play alternates between MAX's placing an X and MIN's placing an O until we reach leaf nodes corresponding to terminal states such that one player has three in a row or all the squares are filled. The number on each leaf node indicates the utility value of the terminal state from the point of view of MAX; high values are assumed to be good for MAX and bad for MIN (which is how the players get their names).

Game trees

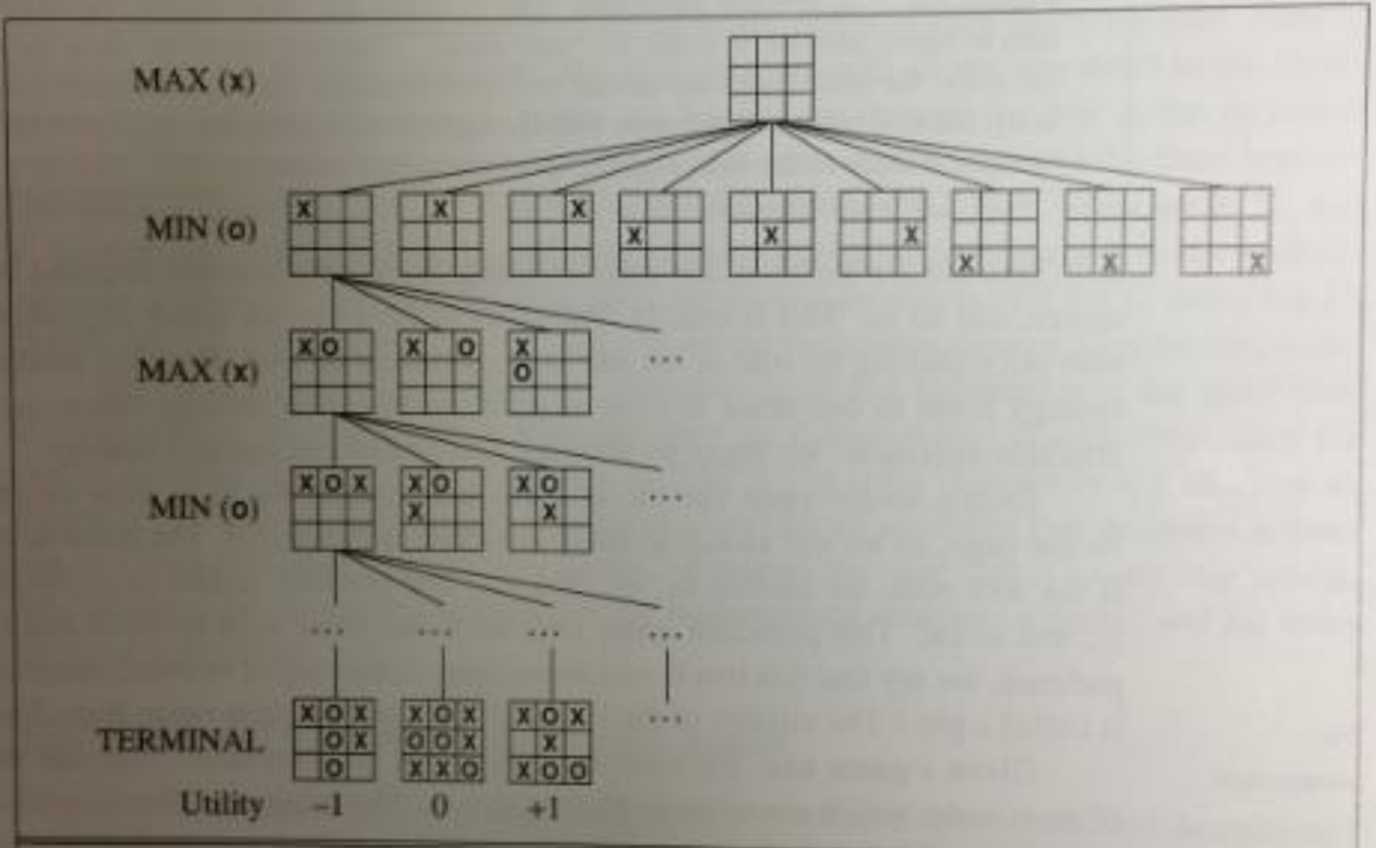


Figure 5.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

Game trees

- For tic-tac-toe the game tree is relatively small—fewer than 9! = 362,880 terminal nodes. But for chess there are over 10^40 nodes, so the game tree is best thought of as a theoretical construct that we cannot realize in the physical world. But regardless of the game tree, it is MAX's job to search for a good move. We use the term **search tree** for a tree that is superimposed on the full game tree, and examines enough nodes to allow a player to determine what move to make.

Optimal decisions in games

- In a normal search problem, the optimal solution would be a sequence of actions leading to a goal state—a terminal state that is a win. In adversarial search, MIN has something to say about it. MAX therefore must find a contingent **strategy**, which specifies MAX's move in the initial state, then MAX's moves in the states resulting from every possible response by MIN, then MAX's moves in the states resulting by every possible response by MIN to *those* moves, and so on. Roughly speaking, an optimal strategy leads to outcomes at least as good as any other strategy when one is playing an infallible opponent.

Game tree

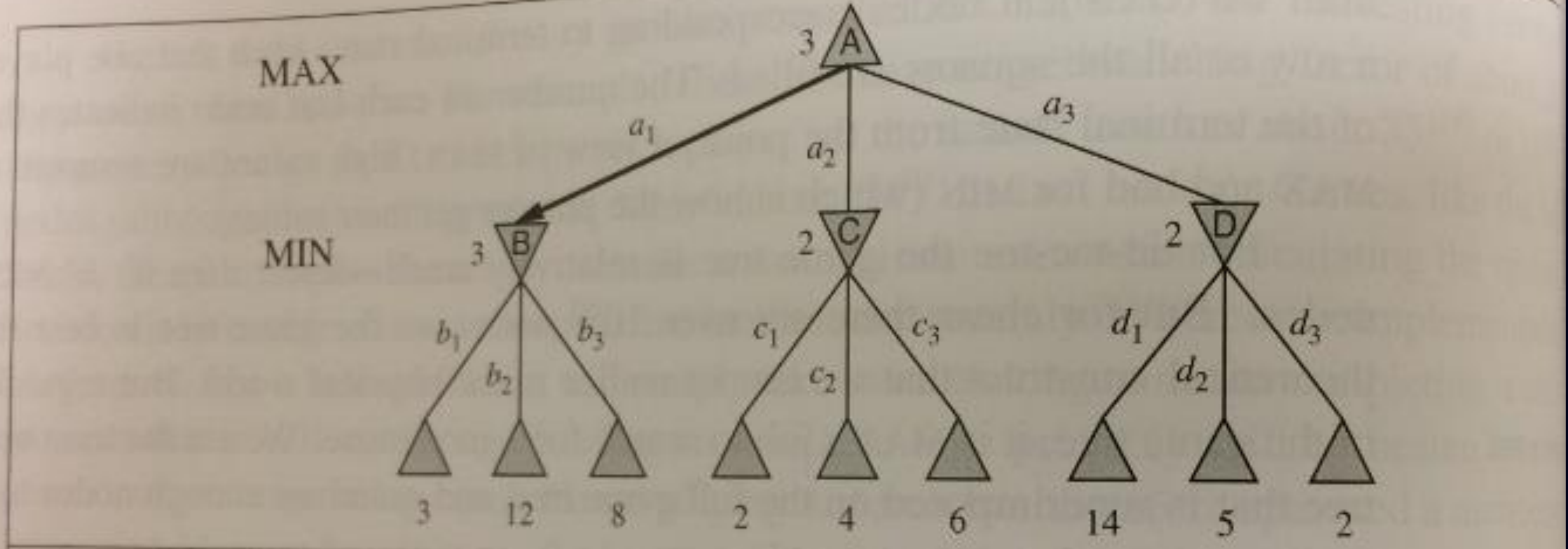


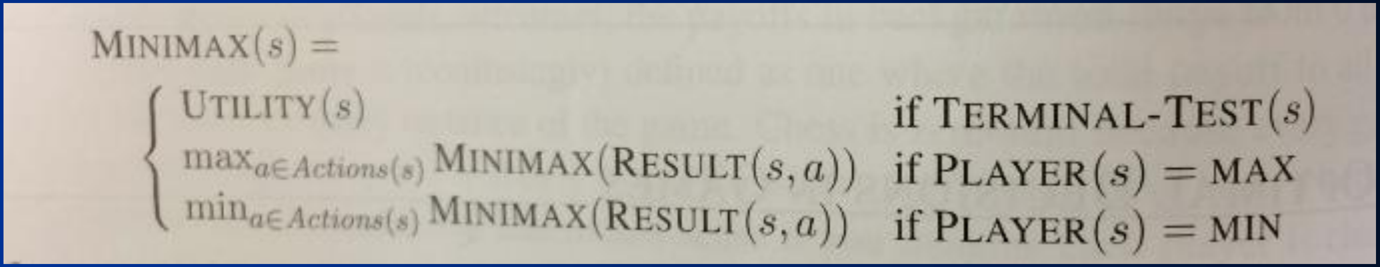
Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

Optimal decisions in games

- Even a simple game like tic-tac-toe is too complex for us to draw the entire game tree on one page, so we will instead examine a “trivial” game. The possible moves for MAX at the root node are labeled a1, a2, and a3. The possible replies to a1 for MIN are b1, b2, b3, and so on. This particular game ends after one move each by MAX and MIN. (We say that this tree is one move deep, consisting of two half-moves, each of which is called a **ply**.) The utilities of the terminal states in this game range from 2 to 14.

Optimal decisions in games

- Given a game tree, the optimal strategy can be determined from the **minimax value** of each node, which we write as MINIMAX(n). The minimax value of a node is the utility (for MAX) of being in the corresponding state, *assuming that both players play optimally* from there to the end of the game. Obviously, the minimax value of a terminal state is just its utility. Furthermore, given a choice, MAX prefers to move to a state of maximum value, whereas MIN prefers a state of minimum value. So we have:



$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

Optimal decisions in games

- Let us apply these definitions to the game tree considered above. The terminal nodes on the bottom level get their utility values from the game's UTILITY function. The first MIN node, labeled B, has three successor states with values 3, 12, and 8, so its minimax value is 3. Similarly, the other two MIN nodes have minimax value 2. The root node is a MAX node; its successor states have minimax values 3, 2, and 2; so it has a minimax value of 3. We can also identify the **minimax decision** at the root: action a1 is the optimal choice for MAX because it leads to the state with the highest minimax value.

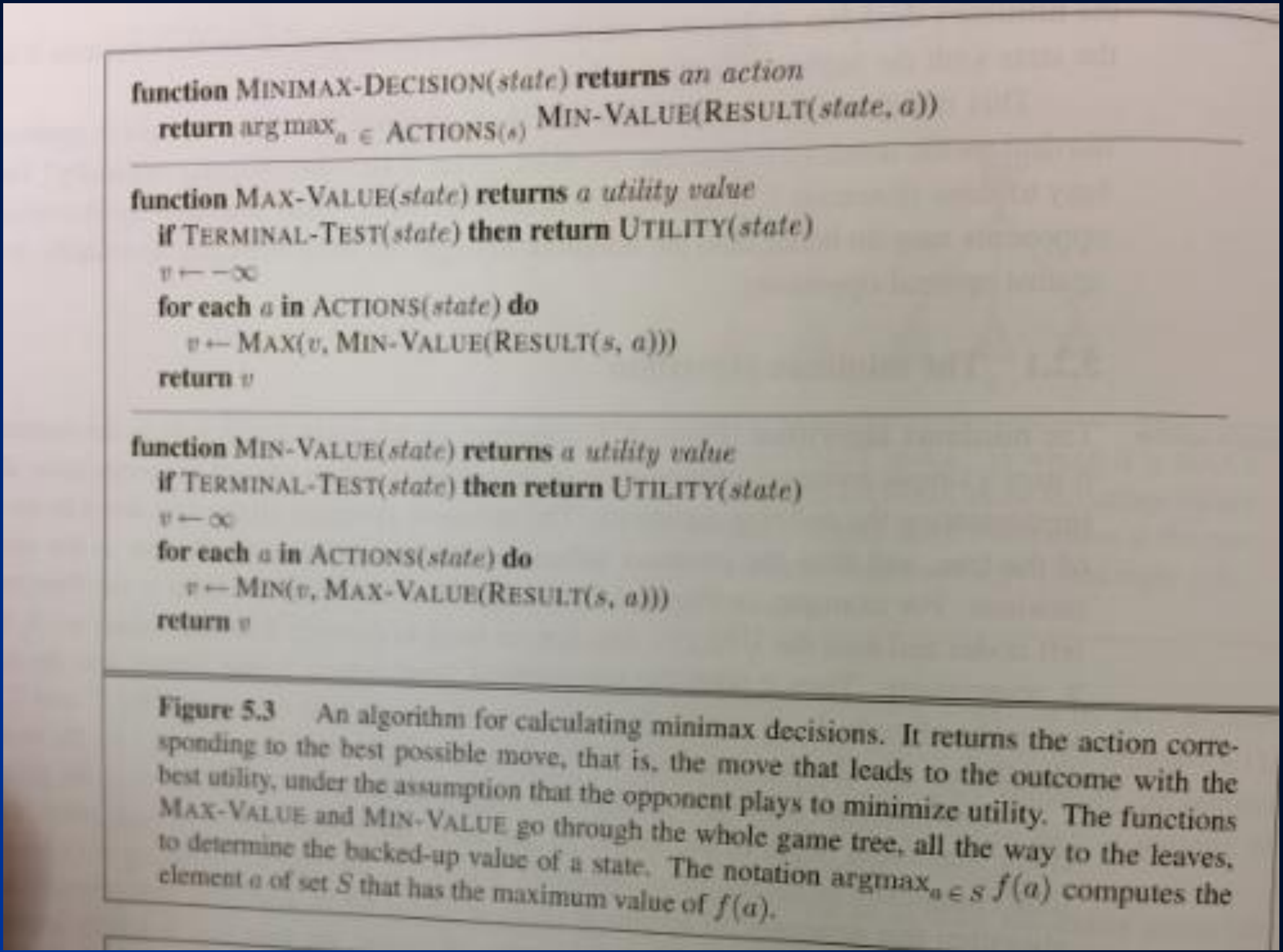
Optimal decisions in games

- This definition of optimal play for MAX assumes that MIN also plays optimally—it maximizes the *worst-case* outcome for MAX. What if MIN does not play optimally? Then it is easy to show (homework exercise) that MAX will do even better. Other strategies against suboptimal opponents may do better than the minimax strategy, but these strategies necessarily do worse against optimal opponents.

The minimax algorithm

- The **minimax algorithm** computes the minimax decision from the current state. It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations. The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are **backed up** through the tree as the recursion unwinds. For example, in the figure the algorithm first recurses down to the three bottom-left nodes and uses the UTILITY function on them to discover that their values are 3, 12, and 8, respectively. Then it takes the minimum of these values, 3, and returns it as the backed-up value of node B. A similar process gives the backed-up values of 2 for C and 2 for D. Finally, we take the maximum of 3, 2, and 2 to get the backed-up value of 3 for the root node.

Minimax algorithm



```
function MINIMAX-DECISION(state) returns an action
  return $\arg\max_{a \in \text{ACTIONS}(s)}$ MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  $v \leftarrow -\infty$
  for each a in ACTIONS(state) do
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  $v \leftarrow \infty$
  for each a in ACTIONS(state) do
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

Figure 5.3 An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state. The notation $\operatorname{argmax}_{a \in S} f(a)$ computes the element a of set S that has the maximum value of $f(a)$.

Minimax algorithm

- Does the minimax algorithm resemble any algorithms we have seen previously?

- How does it rate on the “big 4”?
  - Recall that game-tree search is still a form of search.

Minimax algorithm

- The minimax algorithm performs a complete depth-first exploration of the game tree. If the maximum depth of the tree is m and there are b legal moves at each point, then the time complexity of the minimax algorithm is $O(b^m)$. The space complexity is O(bm) for an algorithm that generates all actions at once, or O(m) for an algorithm that generates actions one at a time. For real games, of course, the time cost is totally impractical, but this algorithm serves as the basis for the mathematical analysis of games and for more practical algorithms.

Optimal decisions in multiplayer games

- Many popular games allow more than two players. Let us examine how to extend the minimax idea to multiplayer games. This is straightforward from the technical viewpoint, but raises some interesting conceptual issues.
- First, we need to replace the single value for each node with a *vector* of values. For example, in a three-player game with players A, B, and C, a vector (vA,vB,vC) is associated with each node. For terminal states, this vector gives the utility of the state from each player's viewpoint. (In two-player zero-sum games, the two-element vector can be reduced to a single value because the values are always opposite.) The simplest way to implement this is to have the UTILITY function return a vector of utilities.

Multiplayer minimax algorithm

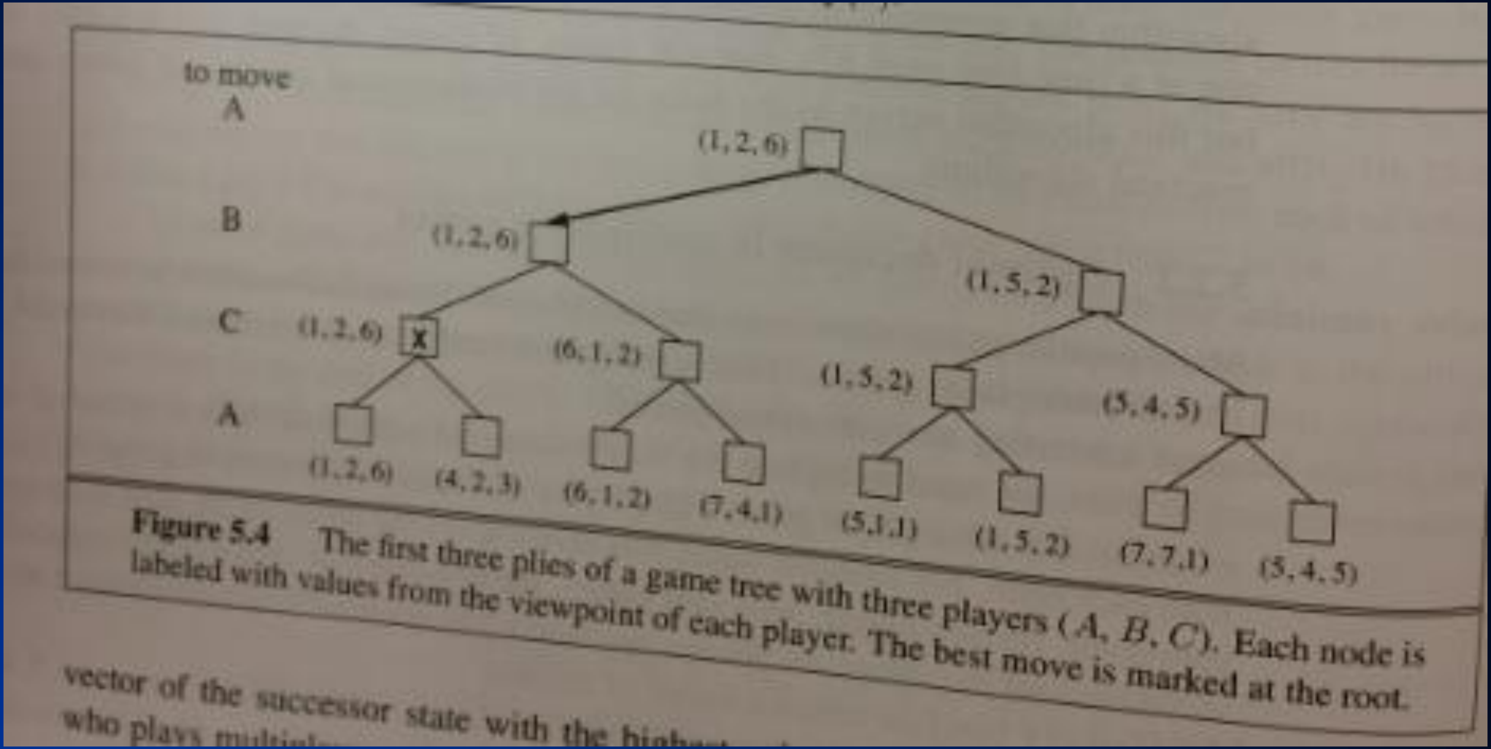


Figure 5.4 The first three plies of a game tree with three players (A, B, C). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

vector of the successor state with the high
who plays multipl

Multiplayer minimax

- Now we have to consider nonterminal states. Consider the node marked X in the game tree. In that state, player C chooses what to do. The two choices lead to terminal states with utility vectors (vA=1,vB=2,vC=6) and (vA=4,vB=2,vC=3). Since 6 is bigger than 3, C should choose the first move. This means that if state X is reached, subsequent play will lead to a terminal state with utilities (vA=1,vB=2,vC=6). Hence, the backed-up value of X is this vector.

Multiplayer minimax

- The backed-up value of a node n is always the utility vector of the successor state with the highest value for the player choosing at n. Anyone who plays multiplayer games, such as Diplomacy, quickly becomes aware that much more is going on than in two-player games. Multiplayer games usually involve **alliances**, whether formal or informal, among the players. Alliances are made and broken as the game proceeds. How are we to understand such behavior? Are alliances a natural consequence of optimal strategies for each player in a multiplayer game?

Multiplayer minimax

- It turns out that they can be. For example, suppose A and B are in weak positions and C is in a stronger position. Then it is often optimal for both A and B to attach C rather than each other, lest C destroy each of them individually. In this way, collaboration emerges from purely selfish behavior. Of course, as soon as C weakens under the joint onslaught, the alliance loses its value, and either A or B could violate the agreement. In some cases, a social stigma attaches to breaking an alliance, so players must balance the immediate advantage of breaking an alliance against the long-term disadvantage of being perceived as untrustworthy.

Multiplayer minimax

- If the game is not zero-sum, then collaboration can also occur with just two players. Suppose, for example, that there is a terminal state with utilities (vA=1000,vB=1000) and that 1000 is the highest possible utility for each player. Then the optimal strategy is for both players to do everything possible to reach this state—that is, the players will automatically cooperate to achieve a mutually desirable goal.

Game-tree search pruning

- The problem with minimax search is that the number of game states it has to examine is exponential in the depth of the tree. Unfortunately, we can’t eliminate the exponent, but it turns out that we can effectively cut it in half. The trick is that it is possible to compute the correct minimax decision without looking at every node in the game tree. That is, we can borrow the idea of **pruning** from the search section (recall that A* pruned the subtree following below Timisoara) to eliminate large parts of the tree from consideration. The particular technique we consider is **alpha-beta pruning**. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

Game tree

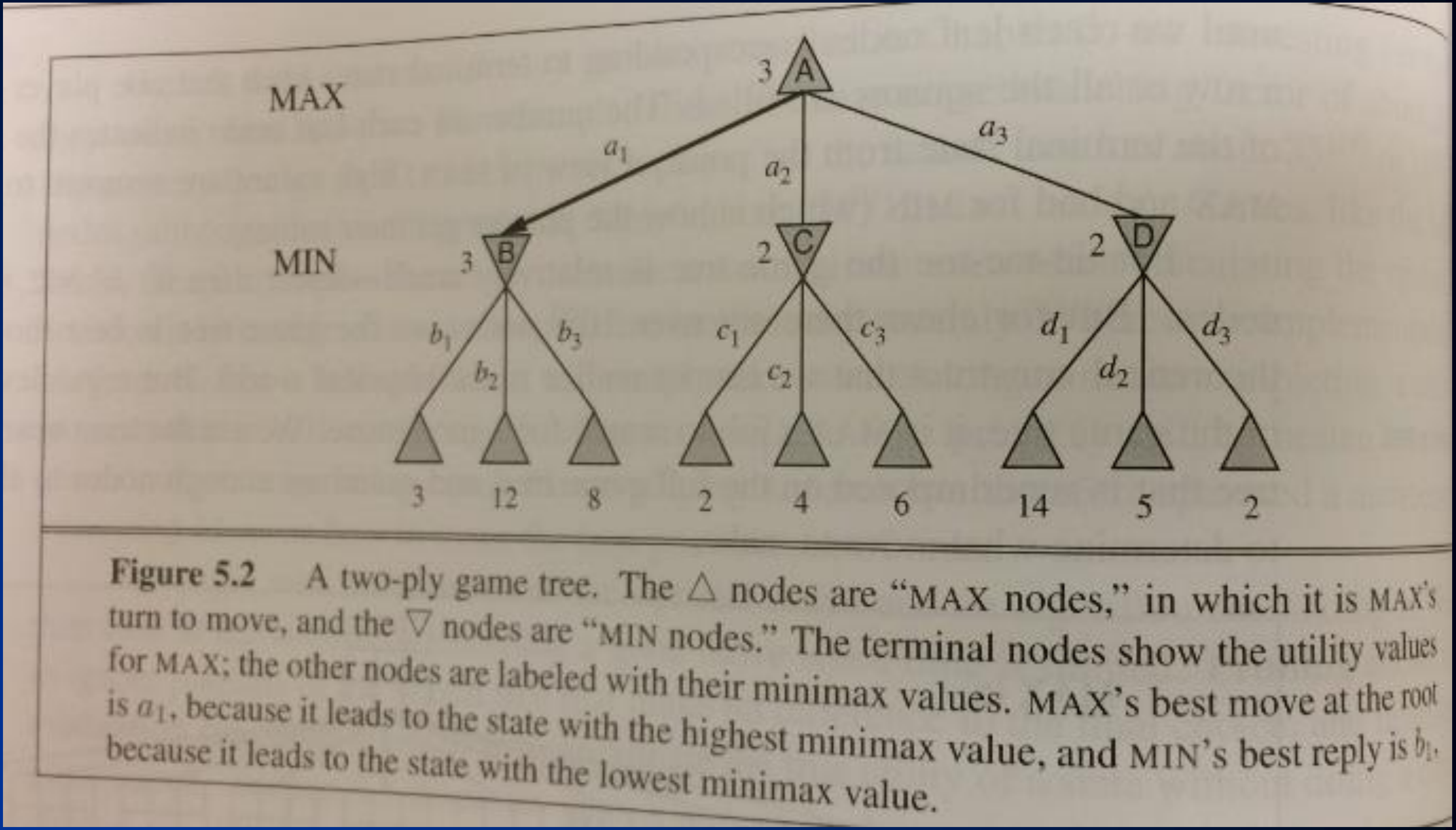


Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

Alpha-beta pruning

- Consider again the two-play game tree. Let’s go through the calculation of the optimal decision once more, this time paying careful attention to what we know at each point in the process. The steps are explained in the figure on the next page. The outcome is that we can identify the minimax decision without ever evaluating two of the leaf nodes.

Alpha-beta pruning

Chapter

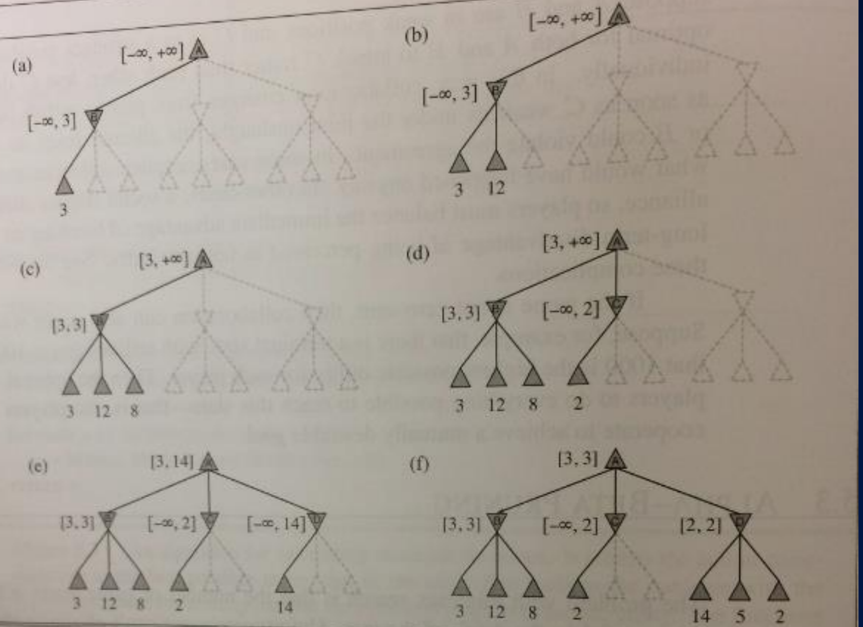


Figure 5.5 Stages in the calculation of the optimal decision for the game tree in Figure 5.2. At each point, we show the range of possible values for each node. (a) The first leaf below *B* has the value 3. Hence, *B*, which is a MIN node, has a value of *at most* 3. (b) The second leaf below *B* has a value of 12; MIN would avoid this move, so the value of *B* is still at most 3. (c) The third leaf below *B* has a value of 8; we have seen all *B*'s successor states, so the value of *B* is exactly 3. Now, we can infer that the value of the root is *at least* 3, because MAX has a choice worth 3 at the root. (d) The first leaf below *C* has the value 2. Hence, *C*, which is a MIN node, has a value of *at most* 2. But we know that *B* is worth 3, so MAX would never choose *C*. Therefore, there is no point in looking at the other successor states of *C*. This is an example of alpha–beta pruning. (e) The first leaf below *D* has the value 14, so *D* is worth *at most* 14. This is still higher than MAX's best alternative (i.e., 3), so we need to keep exploring *D*'s successor states. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14. (f) The second successor of *D* is worth 5, so again we need to keep exploring. The third successor is worth 2, so now *D* is worth exactly 2. MAX's decision at the root is to move to *B*, giving a value of 3.

somewhere in the tree (see Figure 5.6), such th

Alpha-beta pruning

- Another way to look at this is as a simplification of the formula for MINIMAX. Let the two unevaluated successors of node C in the figure have values x and y. Then the value of the root node is given by:

  MIMIMAX(root)

  = max(min(3,12,8),min(2,x,y),min(14,5,2)

  = max(3,min(2,x,y),2)

  = max(3,z,2) where z = min(2,x,y) <= 2

  = 3.
- In other words, the value of the root and hence the minimax decision are *independent* of the values of the pruned leaves x and y.

Alpha-beta pruning

- Alpha-beta pruning can be applied to trees of any depth, and it is often possible to prune entire subtrees rather than just leaves. The general principle is this: consider a node n somewhere in the tree (see next figure) such that Player has a choice of moving to that node. If Player has a better choice m either at the parent node of n or at any choice point further up, then *n will never be reached in actual play*. So once we have found out enough about n (by examining some of its descendants) to reach this conclusion, we can prune it.

General alpha-beta pruning

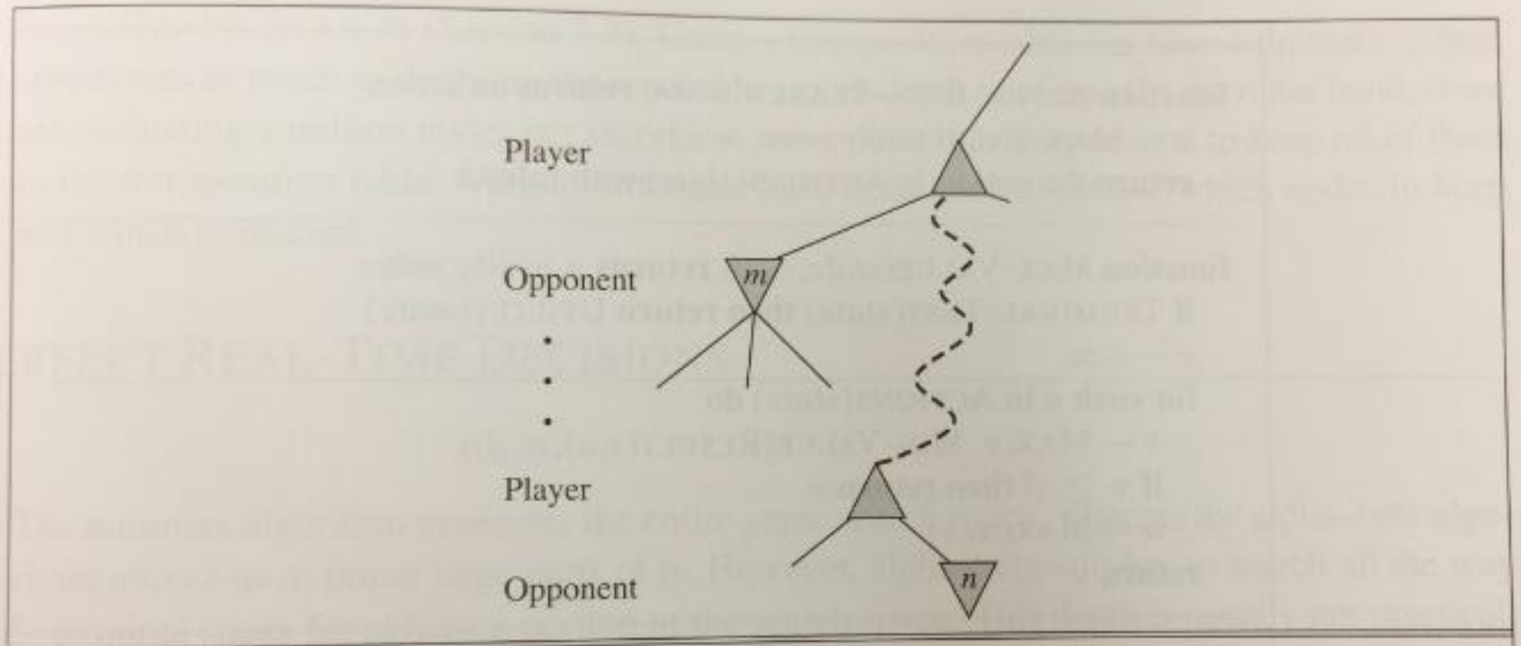


Figure 5.6 The general case for alpha–beta pruning. If m is better than n for Player, we will never get to n in play.

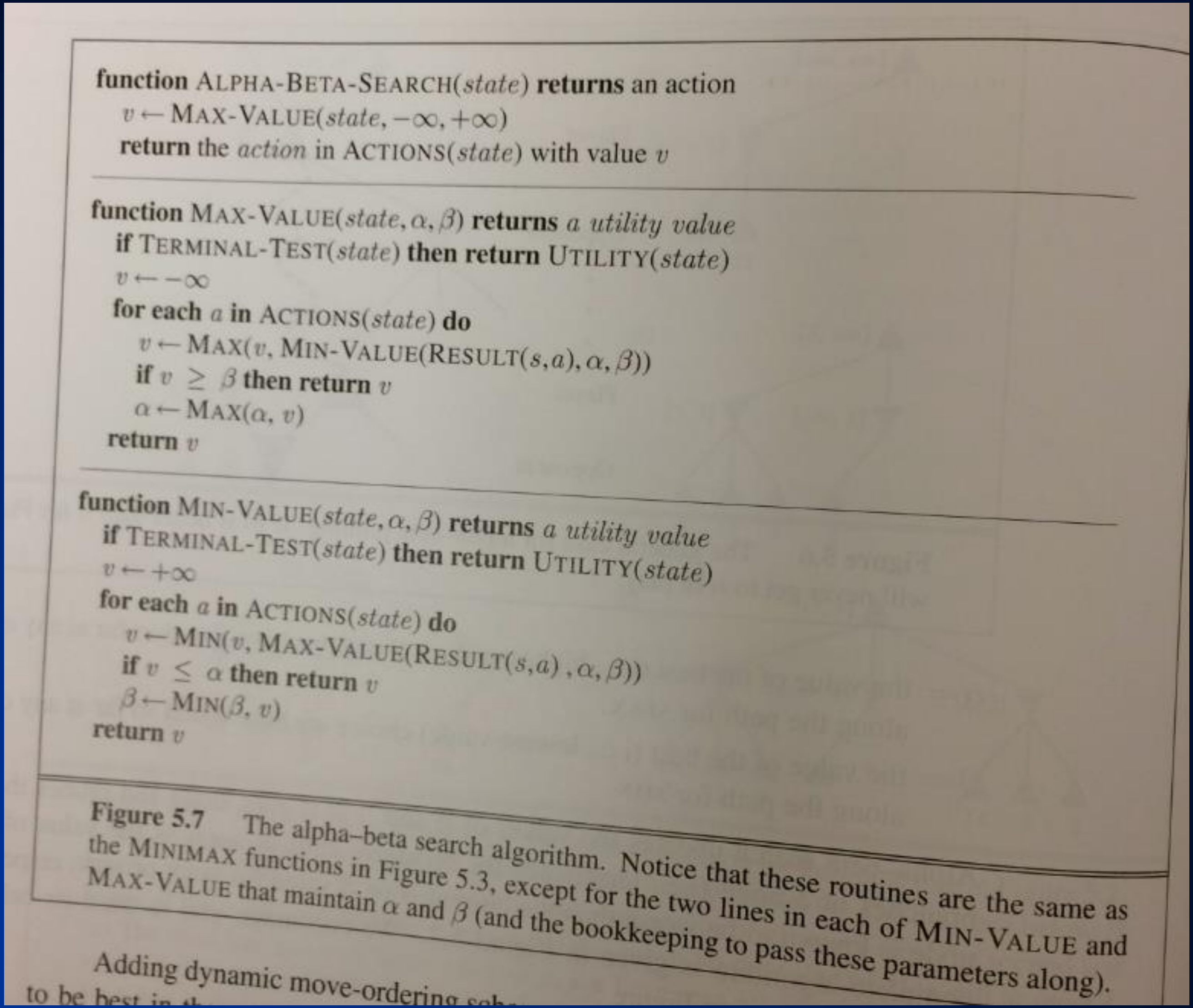
Alpha-beta search

- Remember that minimax search is depth-first, so at any one time we just have to consider the nodes along a single path in the tree. Alpha-beta pruning gets its name from the following two parameters that describe bounds on the backed-up values that appear anywhere along the path:
  - α = the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.
  - β = the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.

Alpha-beta search algorithm

- Alpha-beta search updates the values of α and β as it goes along and prunes the remaining branches at a node (i.e., terminates the recursive call) as soon as the values of the current node is known to be worse than the current α or β value for MAX or MIN, respectively. The complete algorithm is given on the next slide. We can trace its behavior when applied to the example.

Alpha-beta search algorithm



```
function ALPHA-BETA-SEARCH(state) returns an action
  $v \leftarrow$ MAX-VALUE(state, $-\infty, +\infty$)
  return the action in ACTIONS(state) with value $v$

function MAX-VALUE(state, $\alpha, \beta$) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  $v \leftarrow -\infty$
  for each $a$ in ACTIONS(state) do
    $v \leftarrow$ MAX($v$, MIN-VALUE(RESULT($s,a$), $\alpha, \beta$))
    if $v \geq \beta$ then return $v$
    $\alpha \leftarrow$ MAX($\alpha$, $v$)
  return $v$

function MIN-VALUE(state, $\alpha, \beta$) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  $v \leftarrow +\infty$
  for each $a$ in ACTIONS(state) do
    $v \leftarrow$ MIN($v$, MAX-VALUE(RESULT($s,a$), $\alpha, \beta$))
    if $v \leq \alpha$ then return $v$
    $\beta \leftarrow$ MIN($\beta$, $v$)
  return $v$
```

Figure 5.7 The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure 5.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).

Adding dynamic move-ordering sch
to be best in th

Move ordering

- The effectiveness of alpha-beta pruning is highly dependent on the order in which the states are examined. For example, in the figure we could not prune any successors of D at all because the worst successors (from the point of view of MIN) were generated first. If the third successor of D had been generated first, we would have been able to prune the other two. This suggests that it might be worthwhile to try to examine first the successors that are likely to be best.

Alpha-beta move ordering

- If this can be done, then it turns out that alpha-beta needs to examine only O(b^(m/2)) nodes to pick the best move, instead of O(b^m) for minimax. This means that the effective branching factor becomes sqrt(b) instead of b – for chess, about 6 instead of 35. Put another way, alpha-beta can solve a tree roughly twice as deep as minimax in the same amount of time. If successors are examined in random order rather than best-first, the total number of nodes examined will be roughly O(b^(3m/4)) for moderate b. For chess, a fairly simple ordering function (such as trying captures first, then threats, then forward moves, and then backward moves) gets to within about a factor of 2 of the best-case O(b^(m/2)) result.

Alpha-beta move ordering

- Adding dynamic move-ordering schemes, such as trying the moves that were found to be best in the past, brings us quite close to the theoretical limit. The past could be the previous move—often the same threats remain– or it could come from previous exploration of the current move. One way to gain information from the current move is with iterative deepening search. First, search 1 ply deep and record the best path of moves. Then search 1 ply deeper, but use the recorded path to inform move ordering. As we saw in the search module, iterative deepening on an exponential game three adds only a constant fraction to the total search time, which can be more than made up from better move ordering. The best moves are often called **killer moves** and to try them first is called the **killer move heuristic**.

Alpha-beta move ordering

- In the search module, we noted that repeated states in the search tree can cause an exponential increase in search cost. In many games, repeated states occur frequently because of **transpositions**—different permutations of the move sequence that end up in the same position. For example, if White has one move, a1, that can be answered by Black with b1 and an unrelated move a2 on the other side of the board that can be answered by b2, then the sequences [a1,b1,a2,b2] and [a2,b2,a1,b1] both end up in the same position. It is worthwhile to store the evaluation of the resulting position in a hash table the first time it is encountered so that we don't have to recompute it on subsequent occurrences. The hash table of previously seen positions is called a **transposition table**; it is analogous to the *explored* list in GRAPH-SEARCH.

Transposition table

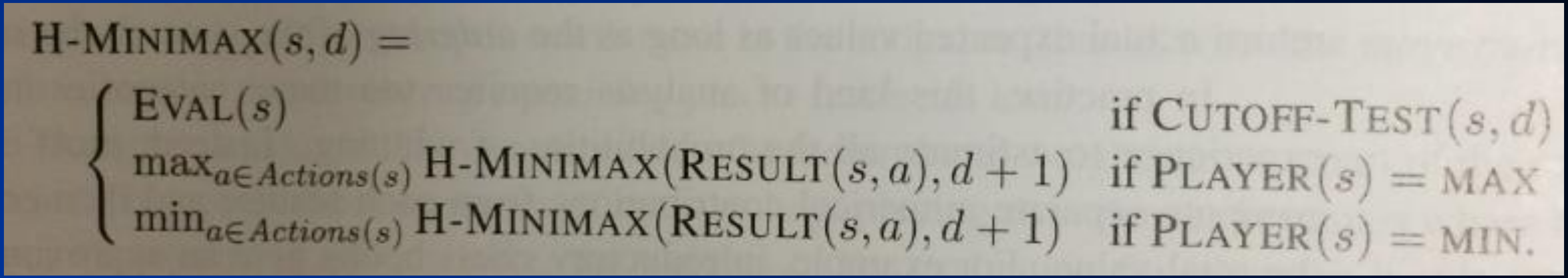
- Using a transposition table can have a dramatic effect, sometimes as much as doubling the reachable search depth in chess. On the other hand, if we are evaluating a million nodes per second, at some point it is not practical to keep *all* of them in the transposition table. Various strategies have been used to choose which nodes to keep and which to discard.

Evaluation function

- The minimax algorithm generates the entire game search space, whereas the alpha-beta algorithm allows us to prune large parts of it. However, alpha-beta still has to search all the way to terminal states for at least a portion of the search space. This depth is usually not practical, because moves must be made in a reasonable amount of time—typically a few minutes at most. Claude Shannon’s paper *Programming a Computer for Playing Chess* (1950) proposed instead that programs should cut off the search earlier and apply a heuristic **evaluation function** to states in the search, effectively turning nonterminal nodes into terminal leaves.

Evaluation function

- In other words, the suggestion is to alter minimax or alpha-beta in two ways:
  - Replace the utility function by a heuristic evaluation function EVAL, which estimates the position's utility
  - Replace the terminal test by a **cutoff test** that decides when to apply EVAL.
- This gives the following for heuristic minimax for state s and maximum depth d:



$$\text{H-MINIMAX}(s,d) = \begin{cases} \text{EVAL}(s) & \text{if CUTOFF-TEST}(s,d) \\ \max_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MIN}. \end{cases}$$

Adversarial search summary

- A game can be defined by the **initial state**, legal actions at each state, the result of each **action**, a **terminal test**, and a **utility function** that applies to terminal states.
- In two-player zero-sum games with **perfect information**, the **minimax** algorithm can select optimal moves by a depth-first enumeration of the game tree.
- The **alpha-beta** search algorithm computes the same optimal move as minimax, but achieves much greater efficiency by eliminating subtrees that are provably irrelevant.
- Usually it is not feasible to consider the whole game tree (even with alpha-beta), so we need to cut the search off at some point and apply a heuristic **evaluation function** that estimates the utility of a state.

Adversarial search extensions

- Many game programs precompute tables of best opening and endgame moves so they can look up a move rather than search.
- Games of chance can be handled by an extension to the minimax algorithm that evaluates a **chance node** by taking the average utility of all children, weighted by the probability of each child.
- Optimal play in games of **imperfect information**, such as Kriegspiel and bridge, requires reasoning about the current and future belief states of each player. A simple approximation can be obtained by averaging the value of an action over each possible configuration of missing information.
- Programs have bested even champion human players at games such as chess, checkers, and Othello. Humans retain the edge in several games of imperfect information, such as poker, bridge, and Kriegspiel, and in games with very large branching factors and little good heuristic knowledge, such as Go (outdated).

Constraint satisfaction

- In the first portion of the search module, we explored the idea that problems can be solved by searching in a space of **states**. These states can be evaluated by domain-specific heuristics and tested to see whether they are goal states. From the point of view of the search algorithm, however, each state is atomic, or divisible—a black box with no internal structure.
- We now describe a way to solve a wide variety of problems more efficiently. We use a **factored representation** for each state: a set of variables, each of which has a value. A problem is solved when each variable has a value that satisfies all the constraints on the variable. A problem describe this way is called a **constraint satisfaction problem**, or CSP.

Constraint satisfaction

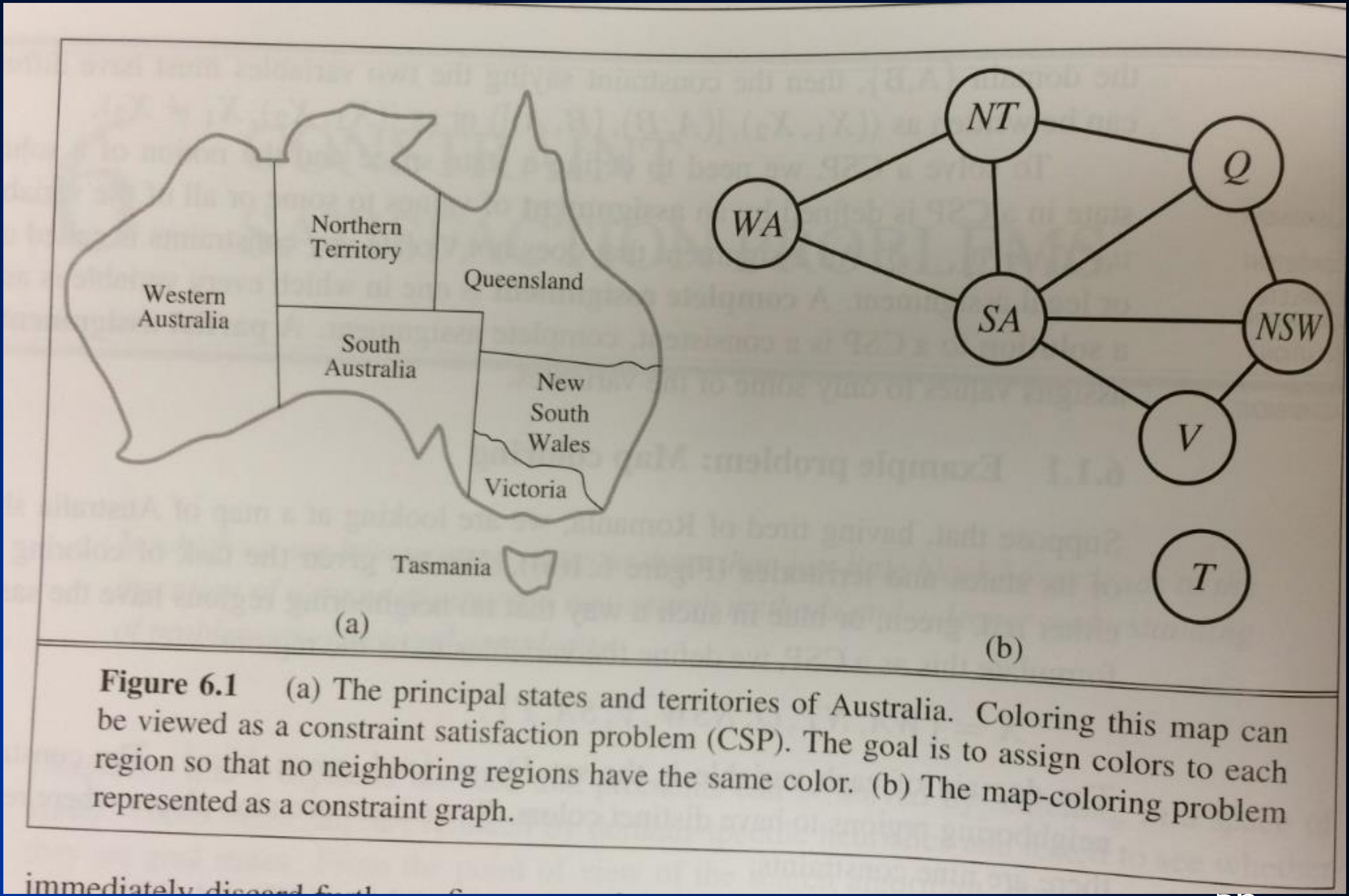


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

Constraint satisfaction problems

- A constraint satisfaction problem consists of three components, X, D, and C:
  - X is a set of variables, $\{X_1,\ldots,X_n\}$.
  - D is a set of domains, $\{D_1,\ldots,D_n\}$, one for each variable.
  - C is a set of constraints that specify allowable combinations of values.

Example problem: Map coloring

- Suppose that, having tired of Romania, we are looking at a map of Australia showing each of its states and territories. We are given the task of coloring each region either red, green, or blue in such a way that no neighboring regions have the same color.
- To formulate this as a CSP, we define the variables to be the regions: X = {WA, NT, Q, NSW, V, SA, T}
- The domain of each variable is the set D_i = {red, green, blue}.
- The constraints require neighboring regions to have distinct colors. Since there are nine places where regions border, there are nine constraints: C = {SA!=WA, SA!=NT,SA!=Q, etc.}
- SA!=WA is shortcut for ((SA,WA),SA!=WA), where SA!=WA can be fully enumerated in turn as {(red,green),(red,blue),…}

Example problem: Map coloring

- There are many possible solutions to this problem, such as …

Example problem: Map coloring

- There are many possible solutions to this problem, such as ...

{WA=red, NT=green, Q=red, NSW=green, V=red, SA=blue, T=red}

- It can be helpful to visualize a CSP as a **constraint graph**. The nodes of the graph correspond to variables of the problem, and a link connects any two variables that participate in a constraint.

Constraint satisfaction problem

- Each domain D_i consists of a set of allowable values, $\{v_1,\ldots,v_n\}$ for variable X_i. Each constraint consists of a pair (*scope*, *rel*), where *scope* is a tuple of variables that participate in the constraint and *rel* is a relation that defines the values that those variables can take on. A relation can be represented as an explicit list of all tuples of values that satisfy the constraint, or as an abstract relation that supports two operations: testing if a tuple is a member of the relation and enumerating the members of the relation. For example, if X1 and X2 both have the domain {A,B}, then the constraint saying the two variables must have different values can be written as ((X1,X2),[(A,B),(B,A)]) or as ((X1,X2),X1 != X2).

CSP

- To solve a CSP, we need to define a state space and the notion of a solution. Each state in a CSP is defined by an **assignment** of values to some or all of the variables, {X1=v1,X2=v2,…} An assignment that does not violate any constraints is called a **consistent** or legal assignment. A **complete assignment** is one in which every variable is assigned, and a **solution** to a CSP is a consistent, complete assignment. A **partial assignment** is one that assigns values to only some of the variables.

Why formulate a problem as a CSP?

- One reason is that the CSPs yield a natural representation for a wide variety of problems; if you already have a CSP-solving system, it is often easier to solve a problem using it than to design a custom solution using another search technique. In addition, CSP solvers can be faster than state-space searchers because the CSP solver can quickly eliminate large swatches of the search space. For example, once we have chosen {SA=blue} in the Australia problem, we can conclude that none of the five neighboring variables can take on the value *blue*. Without taking advantage of constraint propagation, a search procedure would have to consider 3^5=243 assignments for the five neighboring variables; with constraint propagation we never have to consider *blue* as a value, so we have only 2^5=32 assignments to look at, a reduction of 87%.

Why formulate a problem as a CSP?

- In regular state-space search we can only ask: is this specific state a goal? No? What about this one? With CSPs, once we find out that a partial assignment is not a solution, we can immediately discard further refinements of the partial assignment. Furthermore, we can see *why* the assignment is not a solution—we see which variables violate a constraint—so we can focus attention on the variables that matter. As a result, many problems that are intractable for regular state-space search can be solved quickly when formulated as a CSP.

Example problem: Job-shop scheduling

- Factories have the problem of scheduling a day’s worth of jobs, subject to various constraints. In practice, many of these problems are solved with CSP techniques. Consider the problem of scheduling the assembly of a car. The whole job is composed of tasks, and we can model each task as a variable, where the value of each variable is the time that the task starts, expressed as an integer number of minutes. Constraints can assert that one task must occur before another—for example, a wheel must be installed before the hubcap is put on—and that only so many tasks can go on at once. Constraints can also specify that a task takes a certain amount of time to complete.

- We consider a small part of the car assembly, consisting of 15 tasks: install axles (front and back), affix all four wheels (right and left, front and back), tighten nuts for each wheel, affix hubcaps, and inspect the final assembly. We can represent the tasks with 15 variables:

- X = {AxleF, AxleB, WheelRF,…,NutsRF,…, CapRF,…,Inspect}
- The value of each variable is the time that the task starts. Next we represent **precedence constraints** between individual tasks. Whenever a task T1 must occur before task T2, and task T1 takes duration d1 to complete, we add an arithmetic constraint of the form…

- T1 + d1 <= T2
- In our example, the axles have to be in place before the wheels are put on, and it takes 10 minutes to install an axle, so we write:
- AxleF + 10 <= WheelRF;
- AxleF + 10 <= WheelLF;
- AxleB + 10 <= WheelRB;
- AxleB + 10 <= WheelLB.

- Next we say that, for each wheel, we must affix the wheel (which takes 1 minute), then tighten the nuts (2 minutes), and finally attach the hubcab (1 minute, but not represented yet):

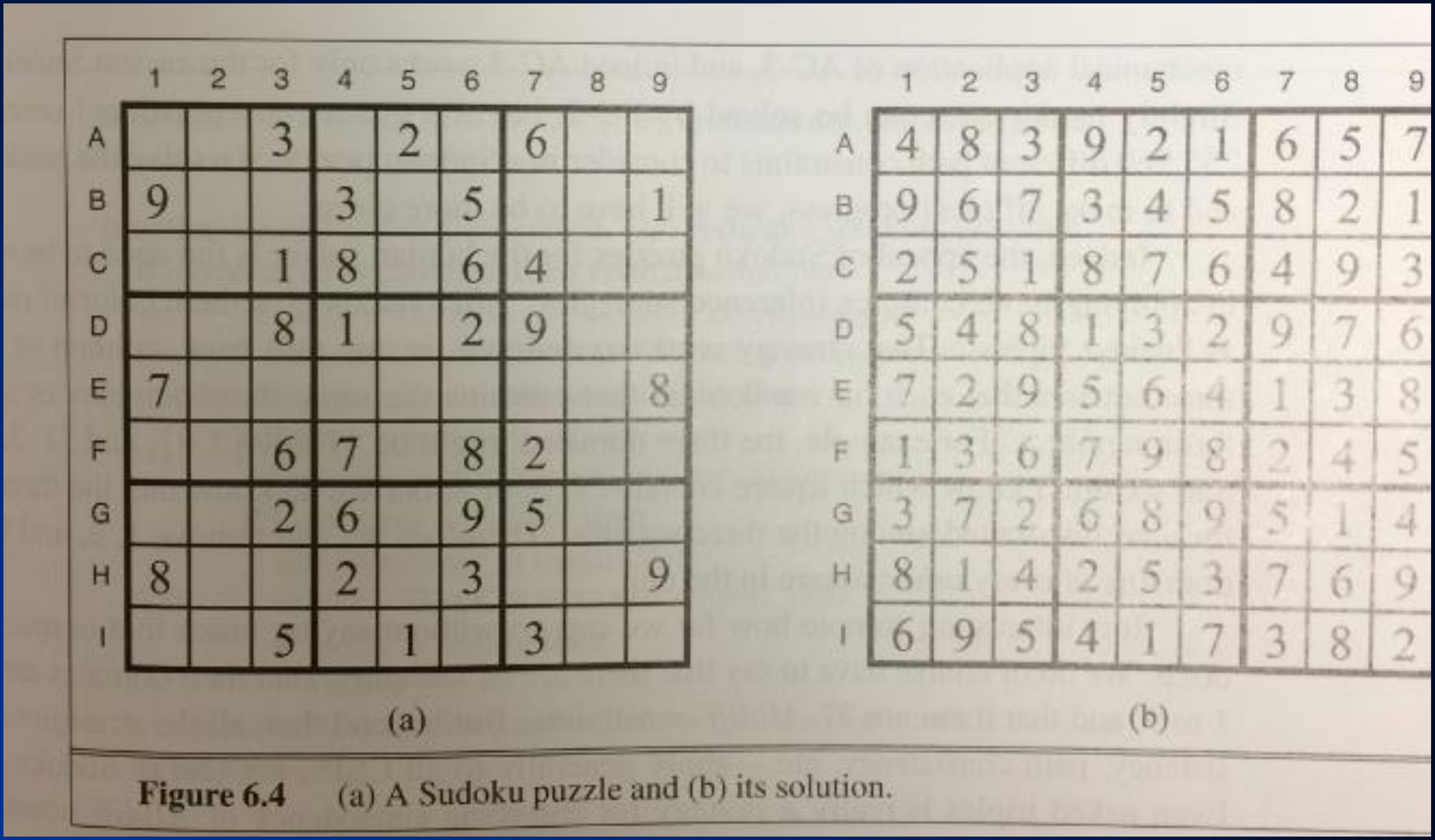
- WheelRF + 1 <= NutsRF;
- WheelLF + 1 <= NutsLF;
- WheelRB + 1 <= NutsRB;
- WheelLB + 1 <= NutsLB;
- NutsRF + 2 <= CapRF;
- NutsLF + 2 <= CapLF;
- NutsRB + 2 <= CapRB;
- NutsLB + 2 <= CapLB.

- Suppose we have four workers to install wheels, but they have to share one tool that helps put the axle in place. We need a **disjunctive constraint** to say that AxleF and AxleB must not overlap in time; either one comes first or the other does:
- (AxleF + 10 <= AxleB) OR (AxleB + 10 <= AxleF)
- This looks like amore complicated constraint, combining arithmetic and logic. But it still reduces to a set of pairs of values that AxleF and AxleB can take on.

- We also need to assert that the inspection comes last and takes 3 minutes. For every variable except *Inspect* we add a constraint of the form X + dX <= Inspect.
- Finally, suppose there is a requirement to get the whole assembly done in 30 minutes. We can achieve that by limiting the domain of all variables: Di = {1,2,3,…,27}

- This particular problem is trivial to solve, but CSPs have been applied to job-shop scheduling problems like this with thousands of variables. In some cases, there are complicated constraints that are difficult to specify in the CSP formalism, and more advanced planning techniques are used, which will discuss in the Planning Module of the course.

Example problem: Sudoku



	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

(a)

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

(b)

Figure 6.4 (a) A Sudoku puzzle and (b) its solution.

Homework for next class

- Chapters 7-8 from Jensen textbook.
- HW1: out 9/5 due 10/3