3.4.3 Alpha-Beta Pruning

- Let us now turn to *pruning* away parts of the game tree, so that the minimax algorithm does not have to visit them, and still play optimally.
- Recall that A^* could also prune its search space: It avoided expanding some paths without missing the optimal path.
- The general idea is similar to the one we used in RBFS:
  - We maintain in the recursion stack extra information about the best alternative available to the currently explored branch.
  - But now we have two agents, and so we need two numbers as the extra information.
  - These two numbers are traditionally written as α and β, hence the name.
- Alpha-beta pruning assumes a two-player zero-sum game.
- The intuition is as follows:
  - Suppose we are computing the value for a MAX node...
  - ...and that we have already computed a value m for one child node c. Then we know that he value for this MAX node will be *at least* this m.
  - Let us then continue to the next sibling d of c. If d eventually gives some value $< m$, then we know that MAX will always choose to play c instead of d.
  - In that case, we *can* prune something from the subtree starting at d, if we pass this already computed MAX value m into its computation.
  - We can namely stop expanding a MIN node in that subtree as soon as we can see that its value will be $\leq m$...
  - ...that is, when this MIN node gets a child whose value is $\leq m$...
  - ...because this MIN node cannot help in getting the value of sibling d to become $> m$.

  Figure 44 shows this intuition, and Figure 45 shows how it applies by showng at each step the range where the node value is now known to be in.
- The same idea can be mirrored for MIN nodes by flipping the pruning condition around to '$\geq$' instead.
- We can realize this intuition with these two parameters:

  $\alpha =$ the best (= largest) value we have already found for MAX along the branch to the current *position*

  $\beta =$ the best (= smallest) value for MIN along the same branch.

  They are added to the minimax algorithm in Figure 41 together with their algorithm with the pruning conditions and the maintenance of these two parameters. Figure 46 shows the modified algorithm.

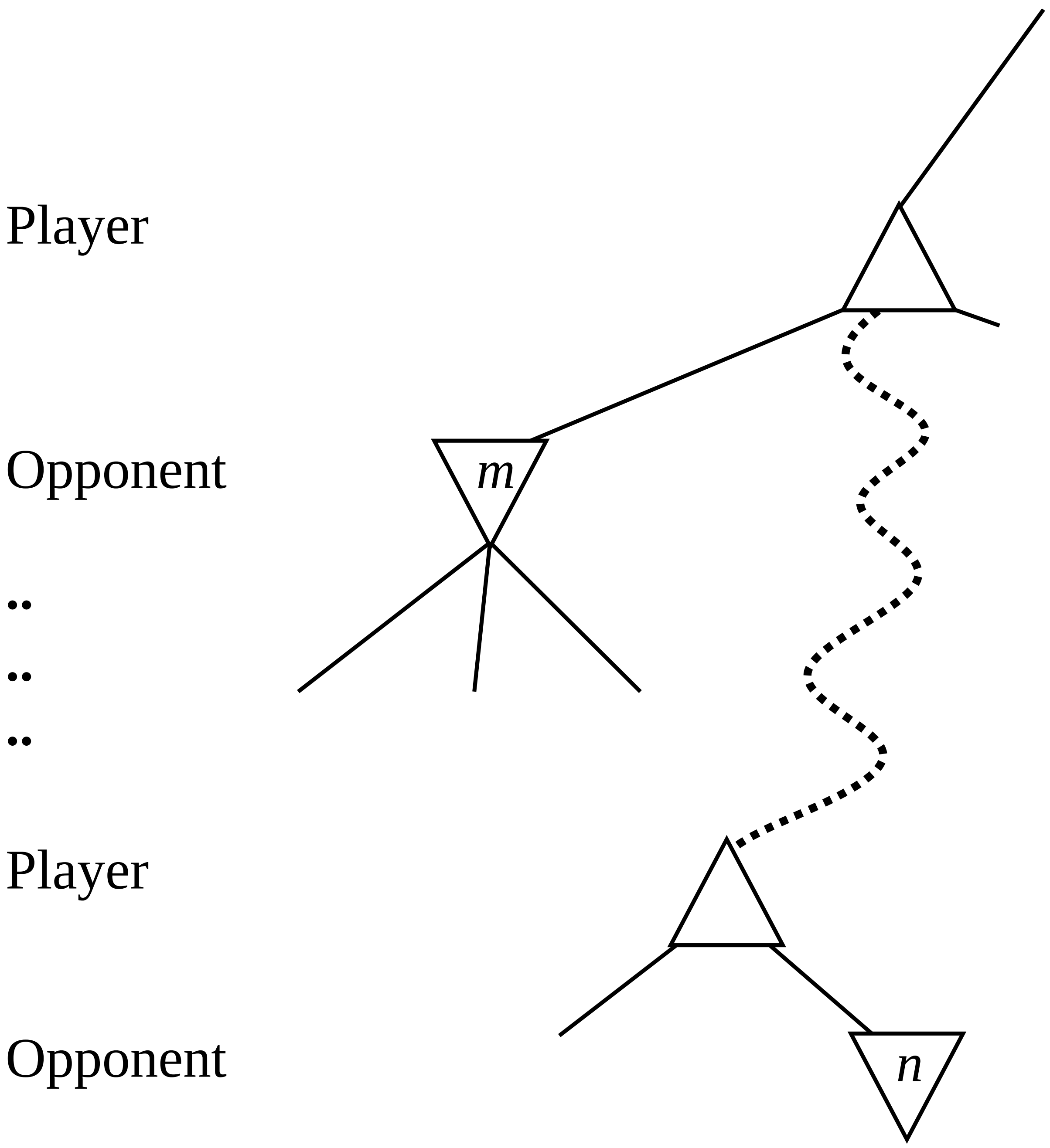


Figure 44: The general alpha-beta pruning principle. (Russell and Norvig, 2003, Figure 6.6)

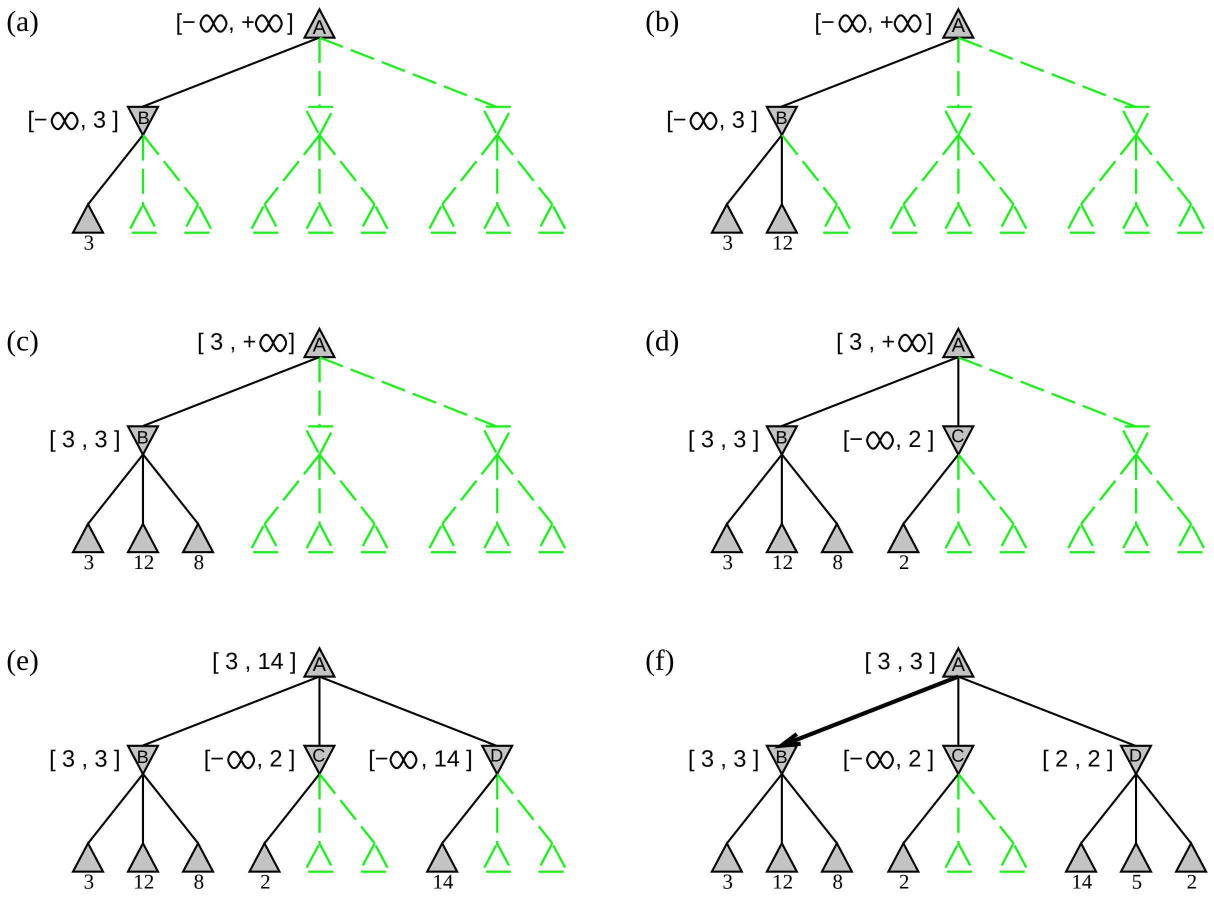


Figure 45: An example of pruning the game tree in Figure 42. (Russell and Norvig, 2003, Figure 6.5)

```
$MaxValue(position, \alpha, \beta)$
1  if $terminal(position)$
2      then return $utility(position, \textsc{Max})$
3  $v \leftarrow -\infty$
4  for each $\langle action, next \rangle \in successors(position)$
5          do $v \leftarrow \max(v, MinValue(next, \alpha, \beta))$
6              if $v \geq \beta$
7                  then return $v$
8              $\alpha \leftarrow \max(\alpha, v)$
9  return $v$
```

```
$MinValue(position, \alpha, \beta)$
1  if $terminal(position)$
2      then return $utility(position, \textsc{Max})$
3  $v \leftarrow +\infty$
4  for each $\langle action, next \rangle \in successors(position)$
5          do $v \leftarrow \min(v, MaxValue(next, \alpha, \beta))$
6              if $v \leq \alpha$
7                  then return $v$
8              $\beta \leftarrow \min(\beta, v)$
9  return $v$
```

Figure 46: The alpha-beta algorithm. (Russell and Norvig, 2003, Figure 6.7)

- The best value which MAX can achieve in the *current position* in computed with the recursive call
$$MaxValue(current\ position, -\infty, +\infty)$$
and the corresponding best move as the child of the *current position* giving this value.

- This alpha-beta without further optimization examines only about
$$\mathcal{O}(b^{3d/4}) \text{ instead of the full } \mathcal{O}(b^d)$$
positions examined by minimax, when the search proceeds to depth d, for games with a moderate branching factor b.

  Stated another way, unoptimized alpha-beta search can reach $1/3$ deeper than minimax given the same execution time.

- The order in which the *successors(position)* are expanded determines how well alpha-beta prunes the game tree:
  - The pruning value m from a child c is passed into the siblings $d, e, f, \ldots$ following c.
  - Hence it is better to get a good m as early as possible. That is, if the player to move in this *position* is

MAX then its children $c, d, e, f, \ldots$ should be expanded in *descending* order wrt. their values

MIN then in *ascending* order.

- If we could always generate $c, d, e, f, \ldots$ in the exactly correct order, then we would not need to search at all:

  Taking the first child c would always be an optimal choice.
- In practice, we sort $c, d, e, f, \ldots$ into some *roughly* correct order.
- If all these rough orderings happen to be the best possible, then alpha-beta is at its best, and it examines only

  $$\mathcal{O}(b^{d/2}) \text{ instead of the full } \mathcal{O}(b^d)$$

  positions, when it proceeds to depth d.

  Stated another way, alpha-beta at its best can reach *twice* deeper than minimax given the same amount of execution time.
- E.g. in Chess, roughly sorting the available moves so that
  1. captures are tried first,
  2. then threats,
  3. then forward moves,
  4. and then backward moves

  gets us to within about a factor of 2 of this best possible behaviour.

3.4.4 Cutting off Search

- Alpha-beta needs at least some values m to pass around before it can start pruning the game tree.
  - That is, it needs to play at least some possible — and hopefully good! — games all the way through before it can start pruning away games which would turn out worse.
  - This is of course unreasonable for most games.
- Hence we must develop alpha-beta further into an algorithm which first

  cuts off the current game before it has ended using some *CutoffTest*(*position*) function, and then

  estimates the utility m' of the remaining game based on the information available at this cutoff *position* using some *Eval*(*position*) function, so that it

  prunes the tree using alpha-beta with these estimates m' instead of exact values m.
- A good *Eval*uation function must estimate

  terminal *position*s so that their *order* is preserved:

  $$\begin{aligned} &\text{if } \mathit{utility}(\mathit{terminal}_1) < \mathit{utility}(\mathit{terminal}_2) \\ &\text{then also } \mathit{Eval}(\mathit{terminal}_1) < \mathit{Eval}(\mathit{terminal}_2). \end{aligned} \tag{18}$$

  Otherwise alpha-beta might choose a wrong ending for the game!

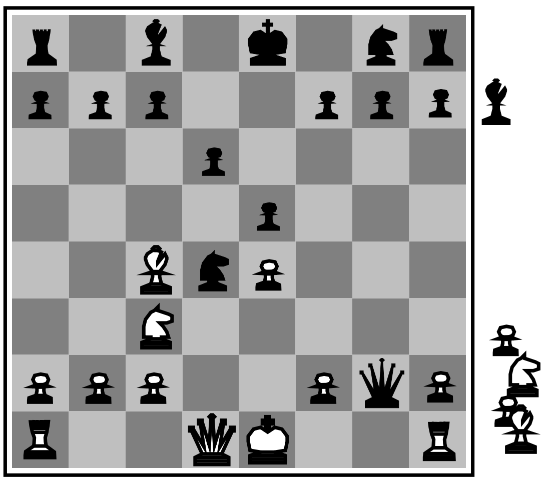

(a) White to move (b) White to move

Figure 47: Two similar chess positions, where (a) is very good for Black because it has many more pieces, but (b) is very bad because the black queen is about to be captured. (Russell and Norvig, 2003, Figure 6.8)

other *positions* so that

$$\mathit{Eval}(\mathit{this}) \approx \text{Max's “chances of winning” after } \mathit{this}$$

because cutting of search at *this* node lumps together all the leaves below it, and some of these leaves are better than others, so

$$\approx \text{expected } \mathit{Utility} \text{ when Max picks any such } \mathit{leaf} \text{ blindly.}$$

- An *Eval*uation function is usually built by

  defining various numerical *features* of game *positions*.
  E.g. introductory Chess books give “rules of thumb” for assessing the strength of a given *position* like “a knight or bishop is worth 3 pawns, a rook 5, the queen 9, ‘good pawn structure’ or ‘king safety’ $\frac{1}{2}$ pawn,...”
  Of course, an *Eval*uation function may fail to distinguish *positions* which look very similar but aren’t, as in Figure 47.

  combining the values of these features somehow into one function, e.g. by just adding them together. In

  theory this combining should be done according to the “expected *Utility*” ideal, but in

  practice this would be too much work for the human expert.

- A solution is to augment the human expert with *machine learning:*

  1. First the expert chooses the features and the *overall form* of the *Eval*uation function. E.g.

     $$\mathit{Eval}(\mathit{position}) = \mathit{weight}_1 \cdot \mathit{feature}_1(\mathit{position}) + \cdots + \mathit{weight}_n \cdot \mathit{feature}_n(\mathit{position})$$

     with an initial value for each weight_i.

2. Then the computer studies a lot of already played games, and tweaks each weight_i so that the values given by this *Eval*uation function get close enough to the actual values observed in the training games.
   - This is an example of a learning agent desribed in Section 2.7.
   - The computer can also play more games against itself, and learn about them too.

- Another use for the *Eval*uation function is improving alpha-beta:
  1. Generate all the $\langle \mathit{action}_i, \mathit{next}_i \rangle \in \mathit{successors}(\mathit{position})$.
  2. Compute $\mathit{Eval}(\mathit{next}_i)$ for each of them.
  3. Sort these next_i wrt. these computed values to get the rough order.

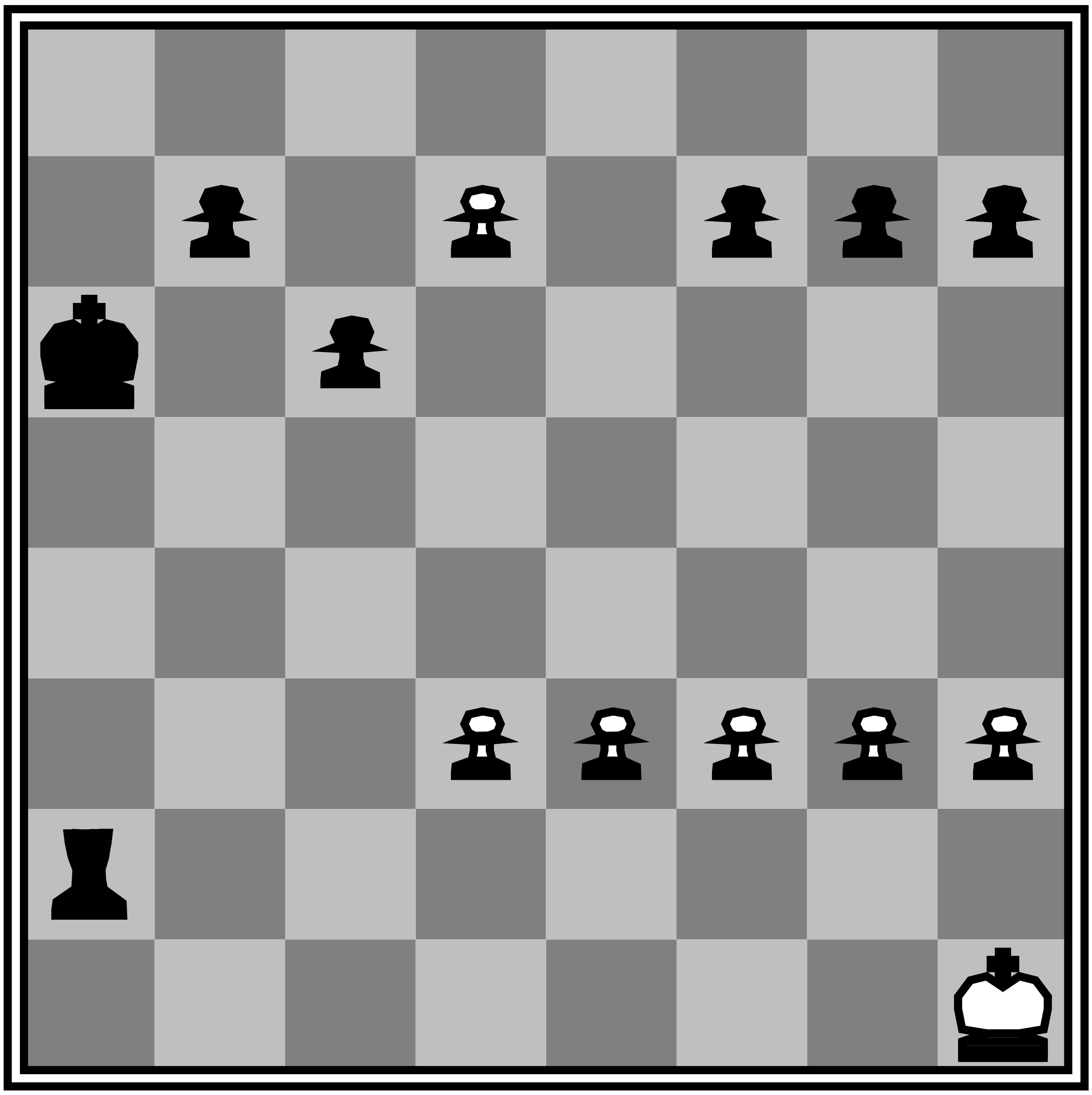
- The *CutoffTest*(*position*) can be designed according to several principles, e.g.

  Fixed depth: Cut off the search when *position.Depth* reaches some fixed *limit* so that our program "looks *limit* moves ahead".
    - But stopping at this fixed *limit* might settle for an imprecise *Eval*uation value, when a much more precise value could have been found by looking (just a little or much) more ahead, as in Figures 47 and 48.
    - This might not use the *time* allowed for thinking about the next move efficiently:
      The *limit* must be set low enough to avoid exceeding the *time*, and then the rest is unused.

  Iterative Deepening: Or we can keep increasing this *limit* and redoing the search, until we have used all the given *time*.
    - Then we return the best move found during the last completed search.
    - Most games have a large branching factor b, so the overhead is small, as in Eq. (6).
    - The "lookahead problem" of a fixed *limit* is lessened but not eliminated.

  Quiescence Search: A fixed depth *limit* can be replaced with a condition to stop when the *position* has become "quiet".
    - That is, this *position* does not suggest any dramatic next moves, which should be examined further.
    - E.g. when the *Eval*uation values in this *position*, its *position.ParentNode* and in its *successors*(*position*) are sufficiently "close to each other".

  Singular Extensions: The *position* is almost quiet, but there is one successor, whose *Eval*uation value is very different (better or worse) from all the others.
    - Then this *position* has one dramatic next move to examine further.
    - The other moves are not examined, because the dramatic move is the one likely to be played.
    - This covers the *position*s where there is just one "obvious" move.



Black to move

Figure 48: The horizon effect: The black rook can check repeatedly, but eventually White will get a queen, so the position looks good for Black for a long time but is really a win for White. (Russell and Norvig, 2003, Figure 6.9)

3.4.5 Including an Element of Chance

- Let us next add an element of *chance* to our games, but keep this element

  controlled in the sense that the rules of the game tell when chance appears into the gameplay and how it affects the rest of the game

  known in the sense that its probability distribution is given beforehand.

  An example is *dice* in board games: rules say when it is rolled, and what happens, and each roll has probability 1/6.

- E.g. in *backgammon* shown in Figure 49 Max starts his turn by rolling two dice. Their outcome determines his available moves:
  - Max must move one of his pieces as many steps as given by one dice.
  - This piece is allowed to end up in a place only if that place has at most one Min's piece.
  - This Min's piece returns to its starting place.
  - Then Max must move similarly according to the other dice.

- We add this into our game tree thinking with levels of CHANCE nodes **C**, as in Figure 50.
  - In their successors, the *actions* are replaced with *probabilities:*

    $$\langle probability_i, position_i\rangle \in successors(\mathbf{C}).$$

    Here this $probability_i$ tells the odds that this random act **C** (like die rolls) comes out so that the game continues from this $position_i$.
  - E.g. in backgammon, the odds that Max rolls x and y is

    $$probability = \begin{cases} \frac{1}{36} & \text{if } x = y \\ \frac{1}{18} & \text{if } x \neq y \text{ — here } x, y \text{ and } y, x \text{ are the same roll} \end{cases}$$

    and they determine his possible moves.
  - Again we assume that the branching factor b remains finite.
  - Moreover the sum of all probabilities in $successors(\mathbf{C})$ is

    $$probability_1 + probability_2 + probability_3 + \cdots + probability_b = 1$$

    since they are exactly the b possible outcomes of this random act **C**.

- The minimax approach extends to these game trees with CHANCE nodes as follows:
  - A new CHANCE node **C** evaluates into the *expected* (value of the) utility, given its probabilities and the computed utilities in its children:

    $$TheValue(\mathbf{C}) = probability_1 \cdot Max\,Value(position_1) + \cdots + probability_b \cdot Max\,Value(position_b) \quad (19)$$

    when the $successors(\mathbf{C})$ are Max-nodes, as in Figure 50, and vice versa.

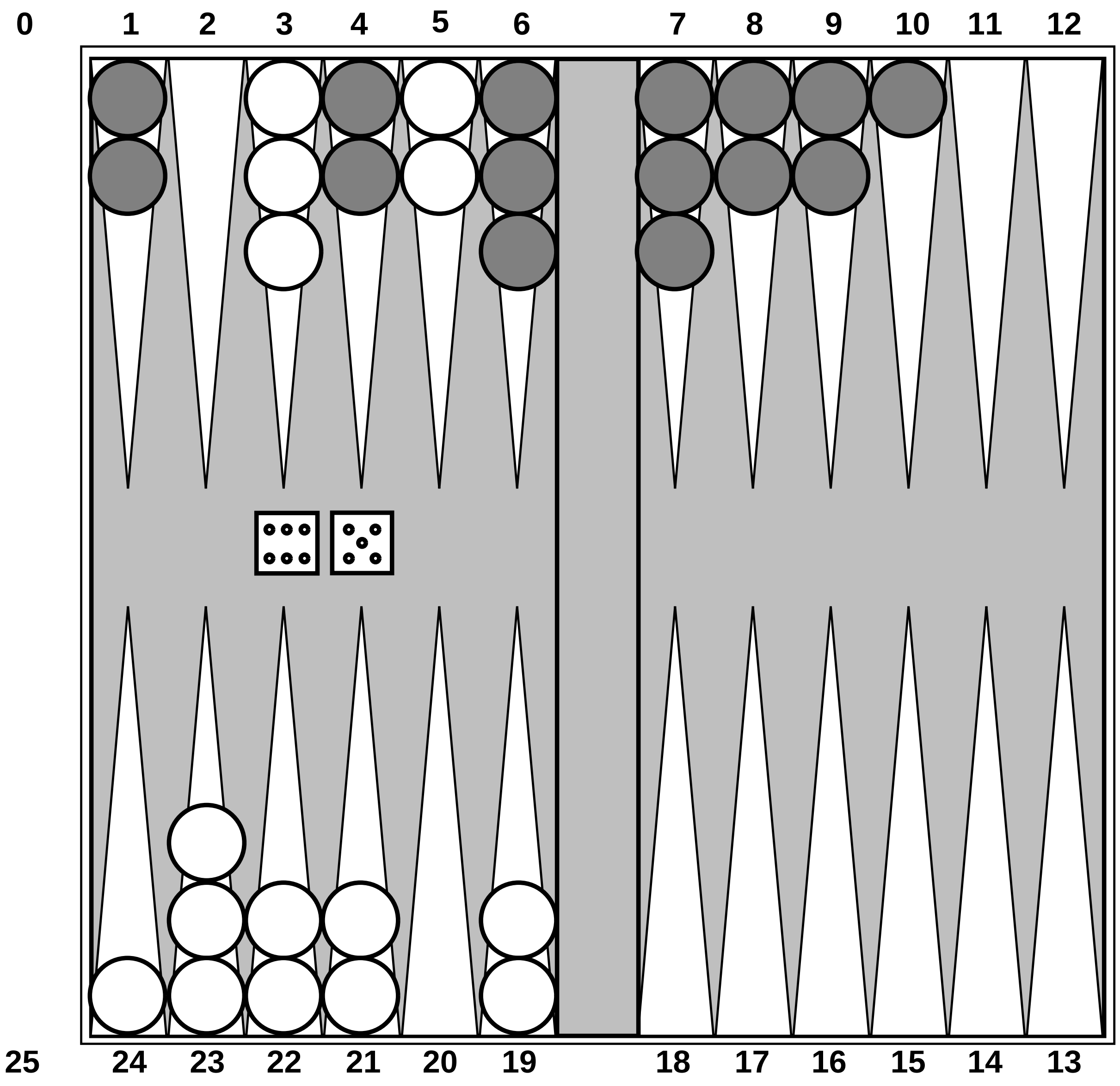


Figure 49: A backgammon position. (Russell and Norvig, 2003, Figure 6.10)

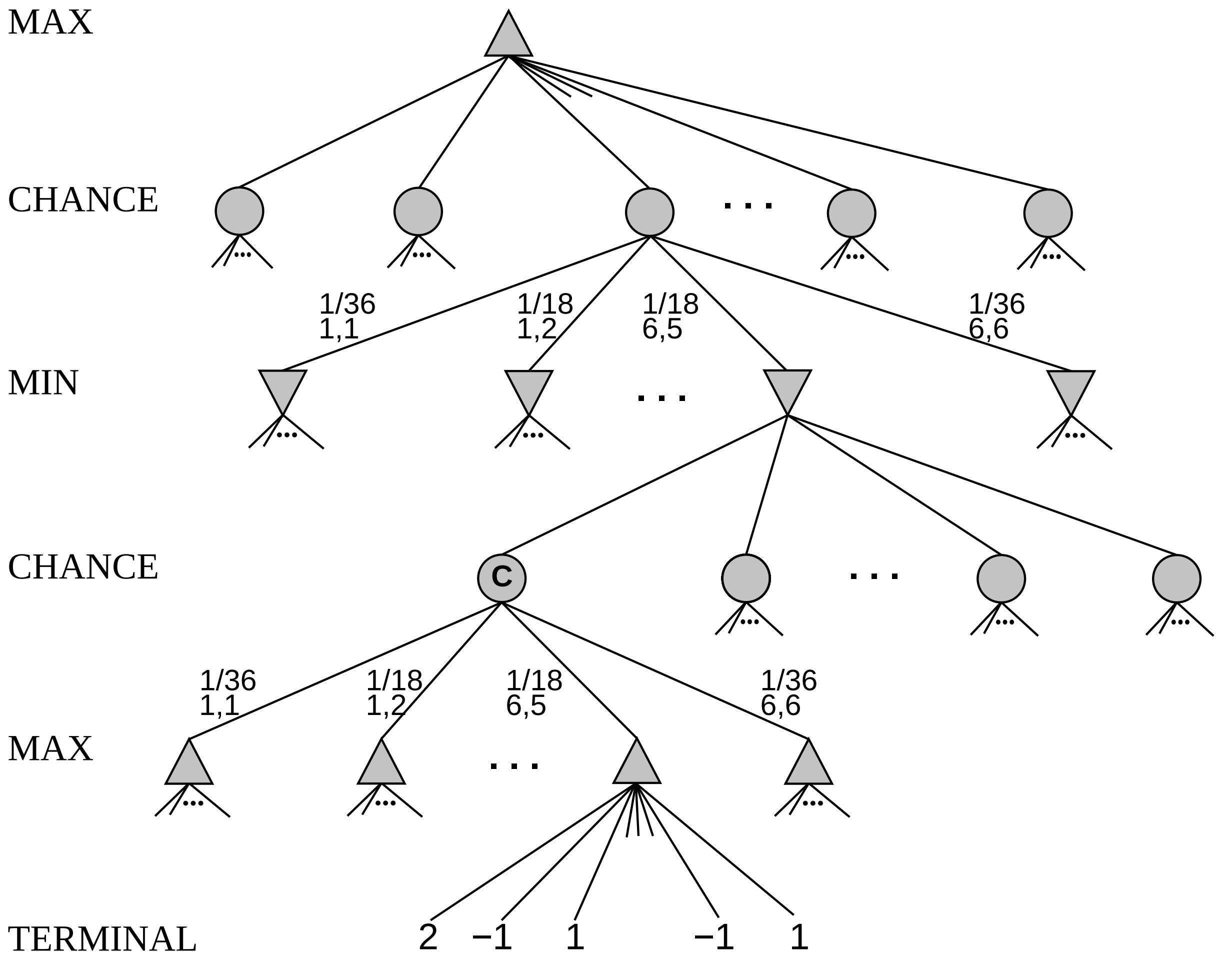


Figure 50: A part of a backgammon game tree. (Russell and Norvig, 2003, Figure 6.11)

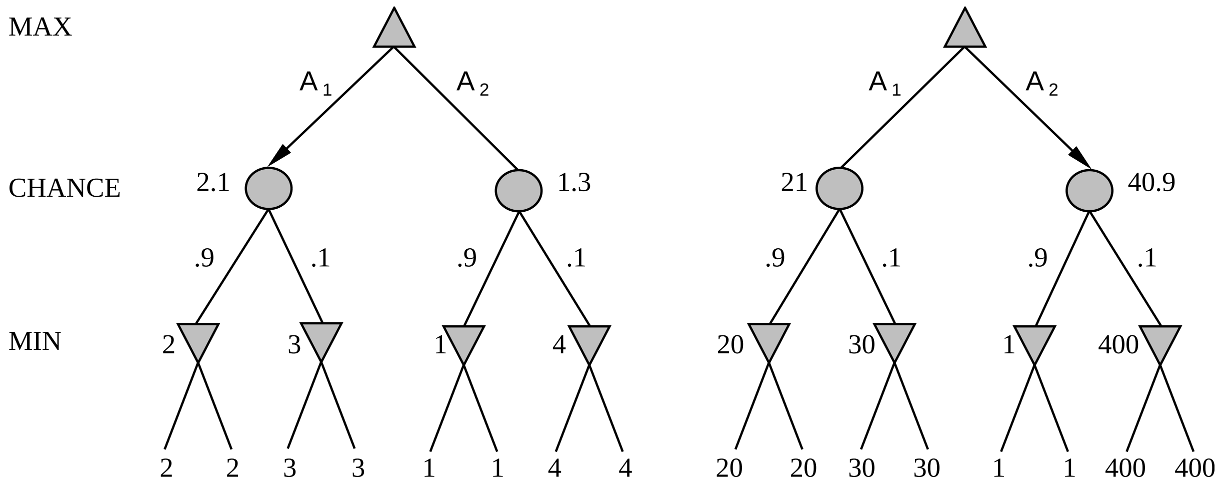


Figure 51: An order- but not action-preserving transformation. (Russell and Norvig, 2003, Figure 6.12)

- MAX and MIN nodes are evaluated as before:
  - Now they optimize the *chances* of winning the game, as determined by these expected utilities Eq (19).
  - The best *action* is still chosen according to the same principle — by each player optimizing his own chances.

This is called the *expectiminimax* algorithm.

- Designing an *Eval*uation function for expectiminimax is even more difficult than for minimax:
  - Eq. (18) showed that this *Eval*uation function must preserve the ordering between the leaves to preserve the *action* chosen.
  - However, this is not enough any more, since we are now computing with expected values instead:

    An *Eval*uation function which preserves the order *but not the relative sizes* of the leaf values can cause another *action* to be chosen instead, as shown in Figure 51!
  - Hence the *Eval*uation function must now also "respect the expected value computation" in Eq. (19).
  - This in turn is guaranteed only if

    $$\mathit{Eval}(\mathit{leaf}) = \mathit{constant}_1 \cdot \mathit{utility}(\mathit{leaf}, \textsc{Max}) + \mathit{constant}_2$$

    so that the only allowed thing is to multiply every correct leaf value with the same $\mathit{constant}_1 > 0$ and adding another $\mathit{constant}_2$.
  - That is, the *Eval*uation function should assess the *leaf* nodes "essentially perfectly" to guarantee choosing the best *action*. . .

- Alpha-beta pruning can often be modified into a form which works in expectiminimax.

- The idea behind the α and β parameters is the same as before…
  - except that now they are bounds for the *expected* instead of optimal values
  - so the pruning rules for MAX and MIN nodes remain the same as before.
- Consider then a CHANCE node **C** whose parent is a MIN node, as in Figure 50. How could we prune any of its children, given that they are all *possible* continuations of the game?
- MIN will *not* need node **C** (because MIN has some other move than **C** which is at least as good) if its

$$\mathit{TheValue}(\mathbf{C}) \geq \beta$$

so we can stop expanding its *successors* as soon as we can be certain that this is true: as soon as

$$\text{the best possible value for } \mathit{TheValue}(\mathbf{C}) \geq \beta. \tag{20}$$

- How can we estimate the best possible value for *TheValue*(**C**) in Eq. (20) without knowing the values of all the *successors*(**C**)?
  - Assume that we are now about to expand its ith successor.
  - Then this estimate is

$$\left(\sum_{j=1}^{i-1} \mathit{probability}_j \cdot \mathit{MaxValue}(\mathit{position}_j)\right) + \\ \left(\sum_{k=i}^{b} \mathit{probability}_k \cdot \underbrace{\text{the best possible } \mathit{leaf} \text{ value for MIN}}_{\text{if this is a constant determined by the game rules}}\right) \tag{21}$$

where the

first sum is the already computed correct value for the already expanded successors $j = 1, 2, 3, \ldots, i-1$

second sum is the best possible value MIN can get from the still unexpanded successors $k = i, i+1, i+2, i+3, \ldots, b$.

- We can derive the symmetric condition to Eqs. (20) and (21) for α as well, and stop expanding as soon as either starts to hold.
- Eqs. (20) and (21) also show that alpha-beta pruning in a CHANCE node could be improved by expanding its *most likely successors* first = by sorting them wrt. their *probability* values into descending order.
- Pruning is less effective now:
  - Without randomness, alpha-beta prunes away those *position*s which are never going to happen, because either *player* will never play so.
  - With randomness, alpha-beta can only prune away those *position*s which this *player* can avoid even when the dice rolls come out badly for him.
- *CutoffTest*s and *Eval*uations are not used in a CHANCE node, because it inherits its *position* from its parent.

3.4.6 Including Imperfect Information

- Let us then consider briefly games of *imperfect* information, where...
  - the game rules still tell what are all the possible *positions* are, but
  - the players don't know what the actual *position* is.

  Examples are *card* games such as bridge:

  The rules tell that a deck of cards is dealt among the players, but they don't know which player got which cards.

- One alluring idea is to consider them as including randomness instead of missing information:
  1. The root of the game tree is one huge random node representing shuffling the deck and dealing the hands.
  2. Then the game continues with these hands dealt.

  - This is *wrong!*
    Because it suggests that the next player should choose his card to play
    - by thinking only about the probabilities of the deal
    - without thinking about what becomes *known* about the deal as the game progresses.
  - However, it simplifies the game search significantly, and often gives good enough results...

    perhaps because humans are not very good either in reasoning about missing information.
  - This is called "averaging over clairvoyancy":
    1. The random node representing the shuffling and dealing takes the expected value ("average") of the deals.
    2. Each deal is "played" as a game of *perfect* information ("clairvoyant" ("selvänäkijä" in Finnish)).

- E.g. let us consider the following simple card game:
  - The rule is: "The player whose turn it is to lead the next trick ("tikki" in Finnish) plays a card. To win this trick, the other player must play a higher card of the same suit ("maa" in Finnish). The winner of the previous trick leads the next."
  - The hands are as follows *and known to both:* [1em] MAX: ♡6, ♢6, ♣9, ♣8. MIN: ♠2, ♣10, ♣5, ♡4. [1em] Now the play is as follows:
    1. MAX leads this 1st trick with his best choice ♣9.
       MIN plays ♣10 and wins this trick.
    2. MIN leads this 2nd trick with ♠2.
       MAX must throw away some card and lose this trick. He throws away ♢6, because both ♡6 and ♣8 are trick winning cards.
    3. MIN leads this 3rd trick with either ♣5 or ♡4.
       MAX wins this trick, no matter which one MIN plays.

4. MAX leads this 4th and final trick with his only remaining card. MIN plays also his only remaining card, and loses.

   The score is a tie: 2–2.

  – Then consider another game, where MIN has ♢4 instead of ♡4:

    The only difference is that MAX throws away ♡6 instead of ♢6 in the 2nd trick.

  – Then consider a third game, where MAX knows that MIN has either ♡4 or ♢4, but does not know which one.

    * Now averaging over clairvoyancy still suggests that MAX should lead the 1st trick with ♣9, because is the best choice in both deals.
    * But MAX does not know which of ♡6 or ♢6 he should throw away in the 2nd trick!
    * If MAX chooses the wrong suit, then MIN wins with a score of 1–3!
    * Hence MAX is now in a *position* where the best he can hope for is a tie but he might also lose. . .
    * MAX should have started the game with ♡6 and ♢6 (in either order) instead of ♣9 since this would get a certain tie instead.

- MAX could — and should! — have found this out by building instead a game tree, whose nodes are *belief* states instead of actual *positions*:

  – Its root would have been a MAX node telling what he knows in the beginning of the game: [1em] MAX: ♡6, ♢6, ♣9, ♣8. MIN: ♠2, ♣10, ♣5, either ♡4 or ♢4. [1em]

  – MIN has both possible *actions* "play ♡4" and "play ♢4" in this tree, because he does know which card he has in his hand.

  – The tree must also take into account how the game rules specify the leading *player* for the next trick, but this is a minor detail.

  Then standard (and not expecti)minimax in this tree finds the correct strategy for MAX, because it models correctly the lack of information in the beginning of the game and its effects to the moves.

- Of course, a game might have both randomness and imperfect information. Then a correct — but computationally *very* heavy! — solution would be expectiminimax in belief states.

- Adding imperfect information and/or randomness makes also *one*-player games interesting, e.g.

  solitaires with shuffled card decks

  Yatzy when your aim is to beat your own previous personal high score.

  Since the only *player* is MAX, then

  – alpha-beta pruning drops β

  – (expecti)minimax drops the MIN nodes.

- More generally, games are a model for an intelligent agent reasoning about what it should do in a sequential, static and discrete environment E, which may otherwise be complicated (maybe only partially observable, stochastic, and/or multi-agent).
- In this view, game search optimizations such as alpha-beta pruning, *CutoffTest*s and *Eval*uation functions can be seen as tools for trying to cope with a dynamic E by calculative rationality.

4 Logic

The material in this section is from the course book Russell and Norvig (2003, Chapters 7–9) unless indicated otherwise.

- We have already mentioned "logic" as a possible way to
  - give a finite description for a potentially infinite belief set
  - express general background information

  but it has may other uses in AI as well.

  Hence the rest of this course will concentrate on logical AI.

- There are now very many different logics, each targeted for certain uses (in AI, philosophy, linguistics, mathematics,...), and researchers develop new ones all the time.

- This course and its book Russell and Norvig (2003) considers only the two most common logics:
  1. *classical* *propositional* and
  2. *classical* *predicate* logic.

- Here "*classical*" means that these two logics speak about the *two truth values* FALSE and TRUE.

- Another choice would have been *constructive* or *intuitionistic* logic, which furthermore requires that a TRUE sentence must also be *justified* somehow.

- They are separated by the *Rule of Excluded Middle* ("kolmannen poissuljetun laki" in Finnish)(also known as "Tertium Non Datur")

  $$\phi \textbf{ or not } \phi \tag{22}$$

  which holds classically for *every* sentence ϕ:

if ϕ is	then it is TRUE because...
TRUE	its *left* part is TRUE
FALSE	its *right* part is TRUE instead.

- But if

  $$\phi = \text{"there is life on Mars"}$$

  then an intuitionist says that the truth value of Eq. (22) is still UNKNOWN instead, because we have not been able to construct any explicit proof for either part yet.

- Such *constructive* or *intuitionistic* logics are used e.g. in "formal programming": programming language theory, program verification, program synthesis from specifications,...

- The underlying idea is the *Curry-Howard or "proofs-as-programs" correspondence:*

intuitionistic logic concept	**programming concept**
sentence ϕ	type ϕ in the programming language sense
a proof $\mathcal{P}_\phi$ of ϕ	program code $\mathcal{P}_\phi$ having type ϕ
constructing a proof $\mathcal{P}_\psi$ for the result ψ from the proof $\mathcal{P}_\phi$ for ϕ and the proof $\mathcal{P}_{\phi\Rightarrow\psi}$ for the implication $\phi \Rightarrow \psi$	calling the function $\mathcal{P}_{\phi\Rightarrow\psi}$ with the argument value $\mathcal{P}_\phi$ of type ϕ to get the return value $\mathcal{P}_\psi$ of type ψ

  (Neither this course nor Russell and Norvig (2003) discuss this correspondence, because it is outside AI. If you get interested, then Sørensen and Urzyczyn (2006) give a formal exposition.)

- We also omit considering *modal* logics.
  - In the non-modal logics we will consider, checking whether a given sentence ϕ is TRUE or FALSE considers only *one viewpoint:* ours/the agent's current viewpoint.
  - Adding *modalities* into a logic *changes* the viewpoint during this checking.
  - *Time* is one thing which could be treated modally:
    * "Tomorrow he comes" is TRUE *today* if "Here he comes now!" will be true when we switch our viewpoint into *tomorrow.*
    * Such *temporal* logics are used in the *verification* of digital hardware, data transmission protocols,...
  - Another is *knowledge.*
    * We might reason during a card game of imperfect information as follows: "If my *opponent* did have ♡K in his hand, then he would have played it already in that previous trick. Therefore I *do know* that he doesn't have it!"
    * This involves changing to my *opponent*'s viewpoint.

  Logics about knowledge are called *epistemic. Dynamic* epistemic logics consider also how our knowledge changes. They would be interesting when many agents reason with incomplete information.