Artificial Intelligence
CS 6364

Professor Dan Moldovan

Section 3

Informed Search and Adversarial Search

Outline

- Best-first search
- Greedy best-first search
- A* search
- Heuristics revisited

Best-first search

- Idea: use an evaluation function *f(n)* for each node
  - estimate of "desirability"

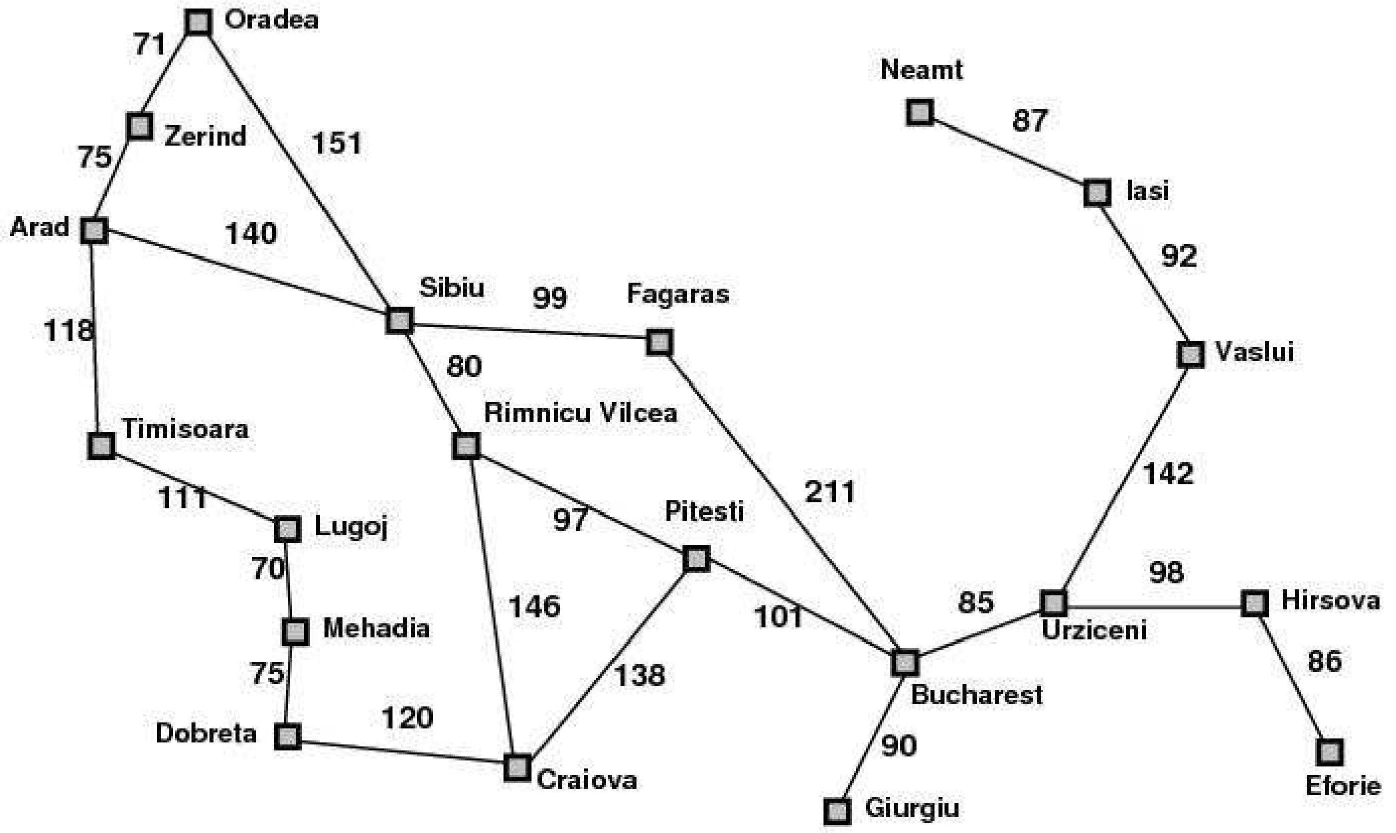
  → Expand most desirable unexpanded node

- Implementation:

  Order the nodes in fringe in decreasing order of desirability

- Special cases:
  - greedy best-first search
  - A^* search

Romania with step costs in km



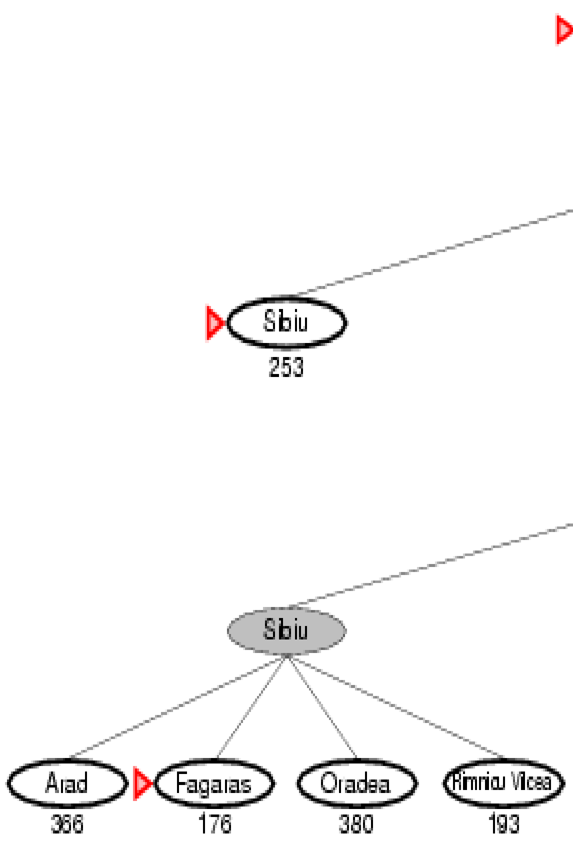
Straight−line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

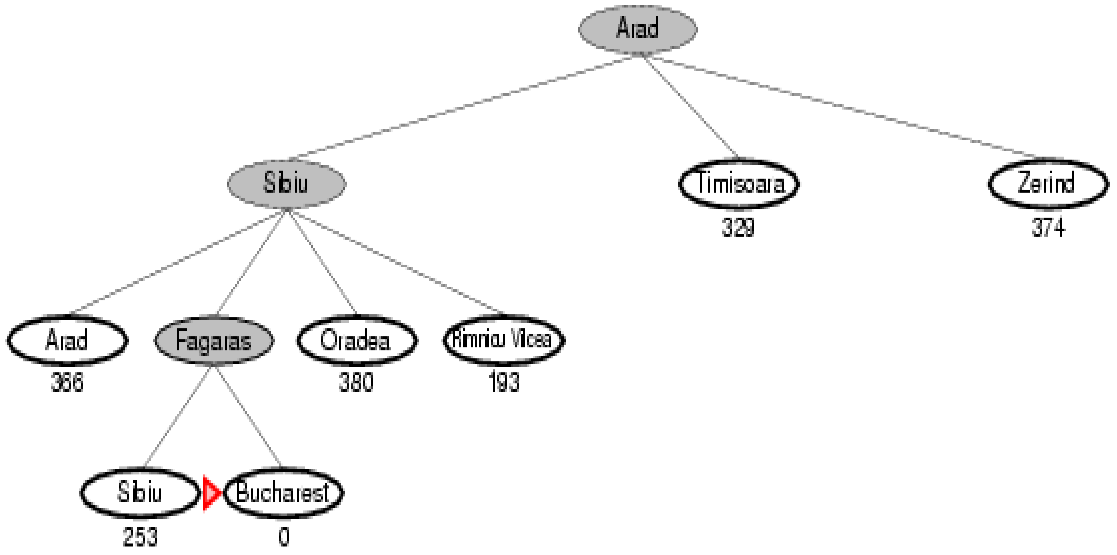
Greedy best-first search

- Evaluation function $f(n) = h(n)$ (heuristic)
  = estimate of cost from n to *goal*
  e.g., $h_{SLD}(n)$ = straight-line distance from n to Bucharest
- Greedy best-first search expands the node that appears to be closest to goal

Greedy best-first search example



Greedy best-first search example

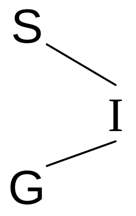


Properties of greedy best-first search

- Complete? No – can get stuck in loops, e.g., Iasi → Neamt → Iasi → Neamt →
- Time? $O(b^m)$, but a good heuristic can give dramatic improvement
- Space? $O(b^m)$ -- keeps all nodes in memory
- Optimal? No

A* Algorithm

Dynamic Programming Principle



The minimum distance from S to G is the sum from S to I and from I to G.

This helps to eliminate bad paths.

A* Algorithm

A* is a branch-and-bound search with an underestimate for remaining distance, combined with dynamic programming principle.

An evaluation function f* is used to evaluate nodes. It has two components.

$f^*(n) = g^*(n) + h^*(n)$

where n is a node

Idea: avoid expanding paths that are already expensive

A* Algorithm

g*(n) – estimates the minimum cost from starting node to n

h*(n) – estimates the minimum cost from node n to goal

Thus f*(n) estimates the minimum cost of a solution path passing through node n.

The actual costs are f, g, h (without *).

Note that g* = g for tree search spaces. g* gives the perfect estimate since only one path from S to n exists.

g* is either exact or overestimate.

h* is the carrier of heuristic information.

$h^* > 0$

$h^* \leq h$ h* needs to be an underestimate

These are the admissibility conditions

A* Algorithm

It can be proven that if h* satisfies the admissibility conditions, and all arc costs are positive, then A* is guaranteed to find optimum path if one exists.

Suppose we have two algorithms A_1 and A_2. We say that A_1 is more informed than A_2 if

$$h_1^*(n) > h_2^*(n)$$

for any node n other than the goal.

If we have two algorithms A and A*, such that A* is more informed, then A* never expands a node that is not also expanded by A.

There are difficulties in:

--finding a good h*

--finding a good h* which requires a few computations

A* Algorithm

A. Generalization of A* is:

$f^* = (1-w)g^* + wh^*$

a) where w is a constant $0 \le w \le 1$

for w = 1 heuristic search or greedy search

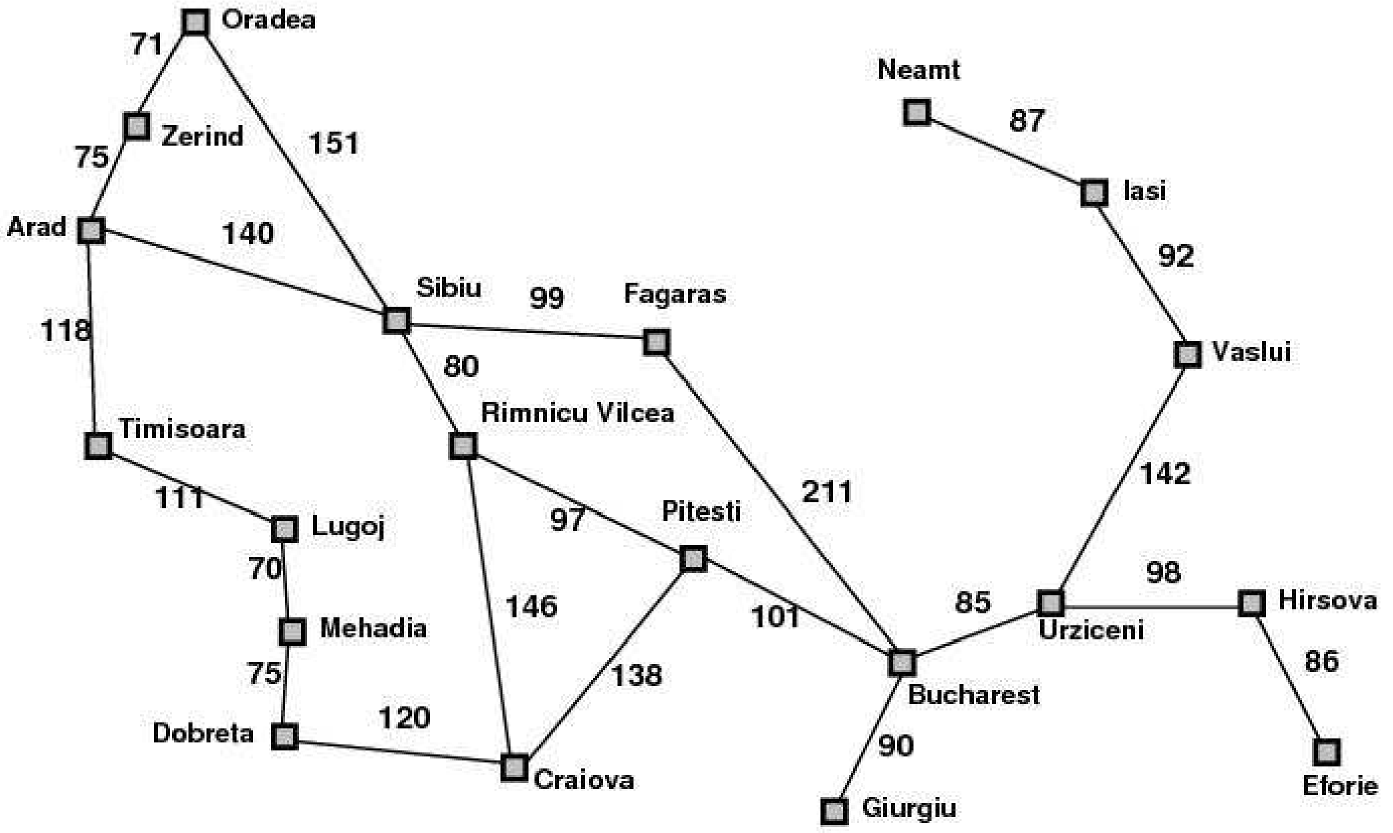
for w = 0 branch and bound

for w = 0.5 A*

b) w may not be constant. This is called dynamic weighting w(n) The weight is adjusted at every node.

It can be proven that A* is complete—meaning that it finds a solution and that for any given admissible heuristic is optimal—meaning that it expands the path with highest-quality solution.

Romania with step costs in km



Straight−line distance
to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Values of h_{SLD}-straight-line distance to Bucharest

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

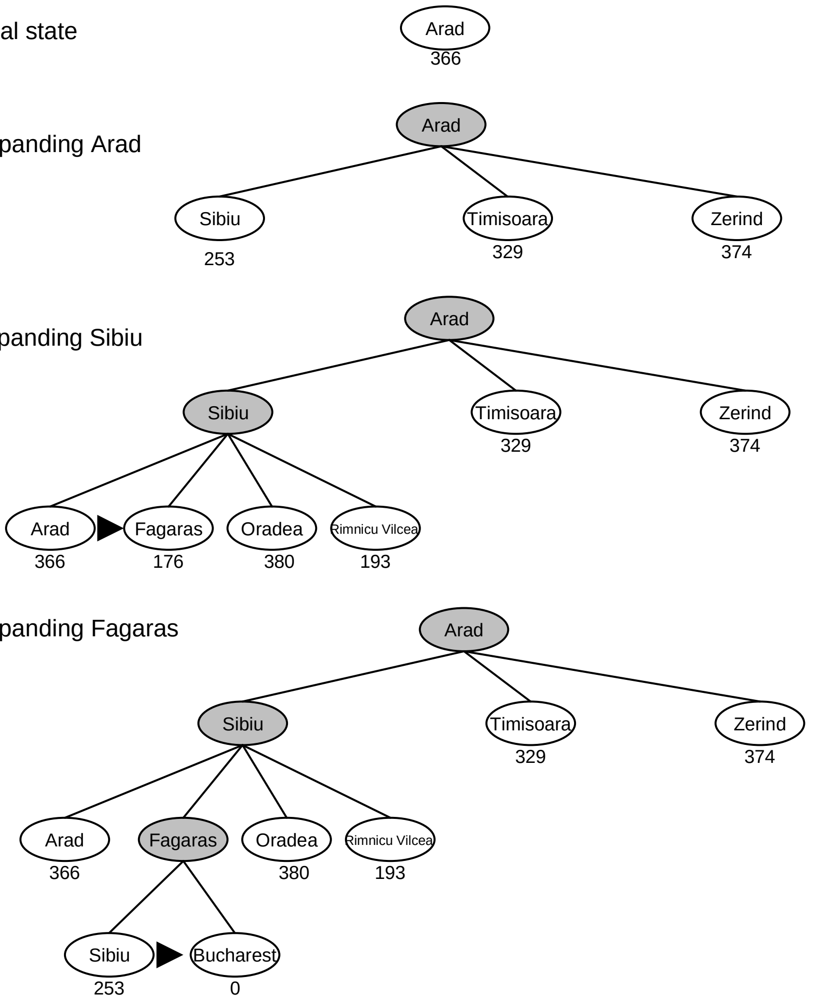
A* Search

(a) The initial state

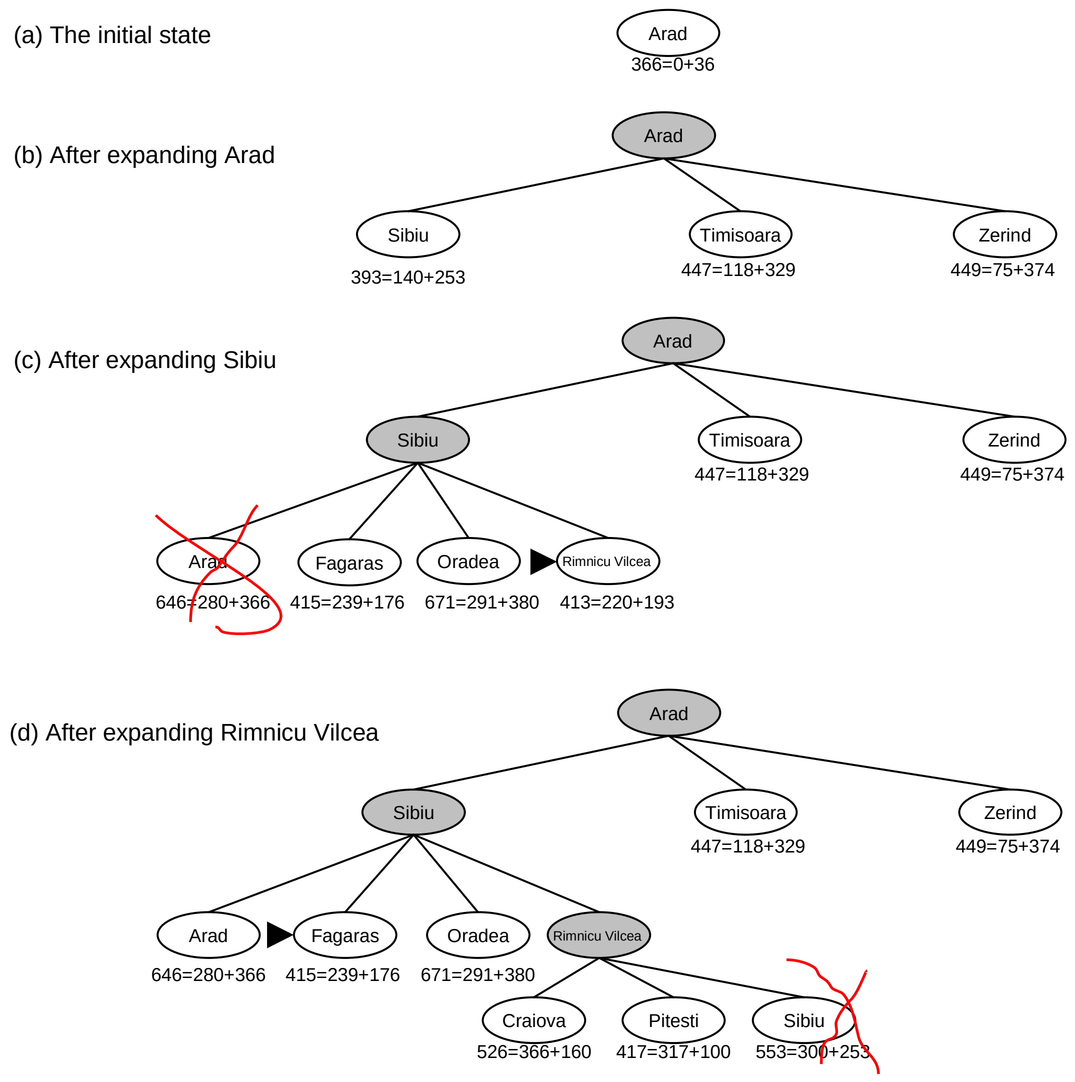
(b) After expanding Arad

(c) After expanding Sibiu

(d) After expanding Fagaras

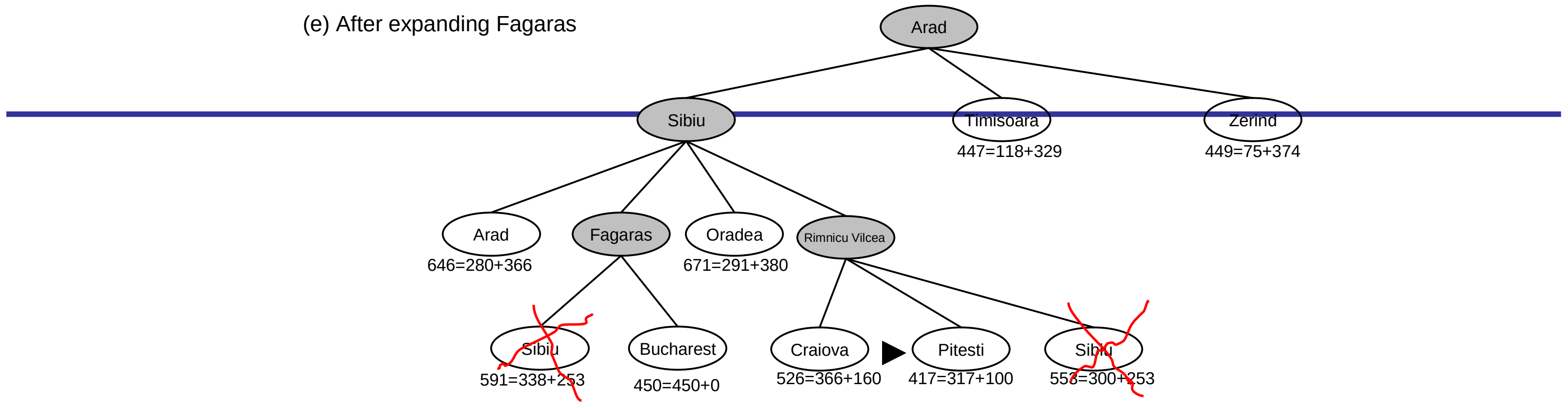


Stages in a greedy best-first search for Bucharest using the straight-line distance heuristic h_{SLD}. Nodes are labeled with the *h*-values.

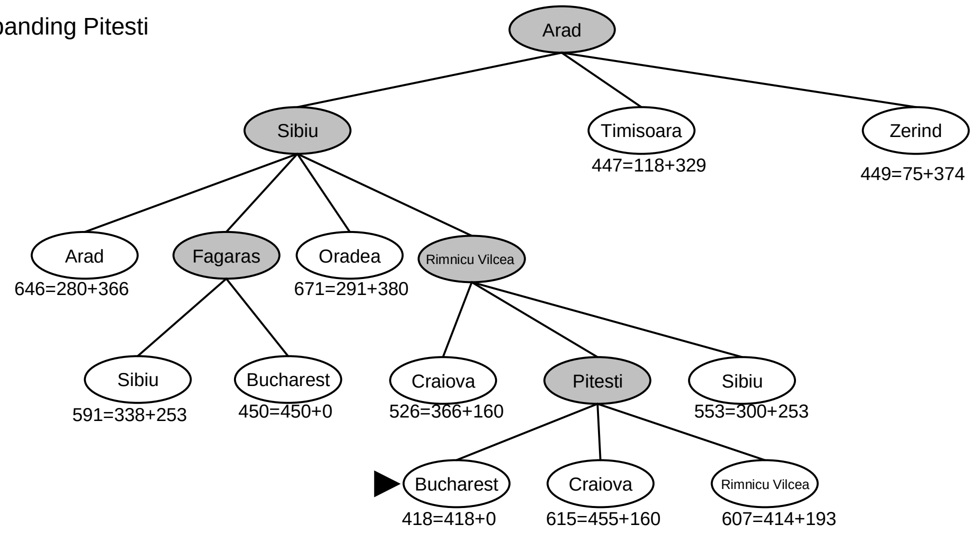


Stages in a A* search for Bucharest. Nodes are labeled with *f* = *g* + *h*. The *h* values are the straight-line distances to Bucharest

(e) After expanding Fagaras



(f) After expanding Pitesti



Stages in a A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest

A* Algorithm

For this example the cost f increases from a node to the next. This property is true for almost all admissible heuristics. The heuristic function decreases, so that at goal is zero. This is called <u>monotonicity</u> property of the heuristic.

If the heuristic function does not have this property—we still can pick a function—called pathmax that is always increasing.

$f^*(n') = \max [f^*(n), g^*(n') + h^*(n')]$

where n is the parent of n′

where heuristic did not respect the monotonicity principle.

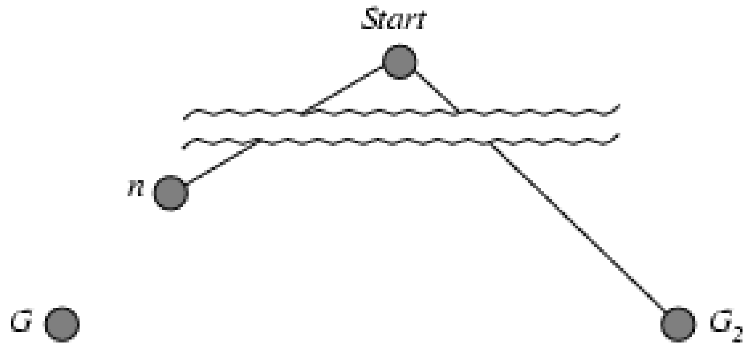
Thus, we can always think of f* as representing some contours in the state space.

Admissible heuristics

- A heuristic h*(n) is admissible if for every node n,
  $h^*(n) \leq h(n)$, where h(n) is the true cost to reach the goal state from n.
- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic
- Example: $h_{SLD}(n)$ (never overestimates the actual road distance)
- Theorem: If h*(n) is admissible, A^* using TREE-SEARCH is optimal

Optimality of A* (proof)

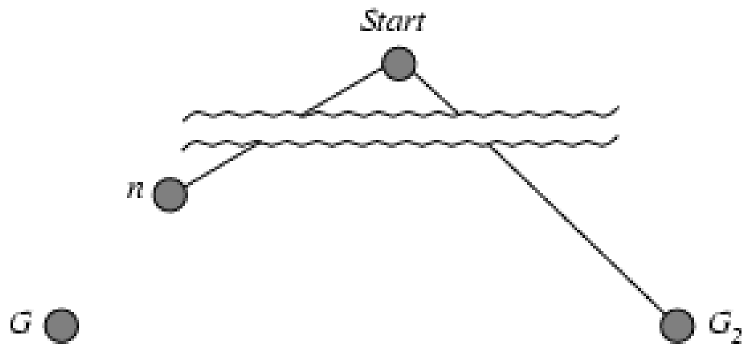
- Suppose some suboptimal goal G_2 has been generated and is in the fringe. Let n be an unexpanded node in the fringe such that n is on a shortest path to an optimal goal G.



$f(G_2) = g(G_2)$	since $h(G_2) = 0$
$g(G_2) > g(G)$	since G_2 is suboptimal
$f(G) = g(G)$	since $h(G) = 0$
$f(G_2) > f(G)$	from above

Optimality of A* (proof)

- Suppose some suboptimal goal G_2 has been generated and is in the fringe. Let n be an unexpanded node in the fringe such that n is on a shortest path to an optimal goal G.



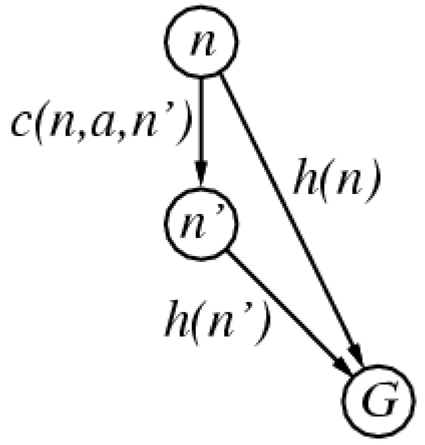
■ $f(G_2)$	$> f(G)$	from above
■ $h^*(n)$	$\leq h(n)$	since h* is admissible
■ $g(n) + h^*(n)$	$\leq g(n) + h(n)$	
■ $f(n)$	$\leq f(G)$	

Hence $f(G_2) > f(n)$, and A* will never select G_2 for expansion

Consistent heuristics

- A heuristic is consistent if for every node *n*, every successor *n'* of *n* generated by any action *a*,

  $h(n) \leq c(n,a,n') + h(n')$



- If *h* is consistent, we have

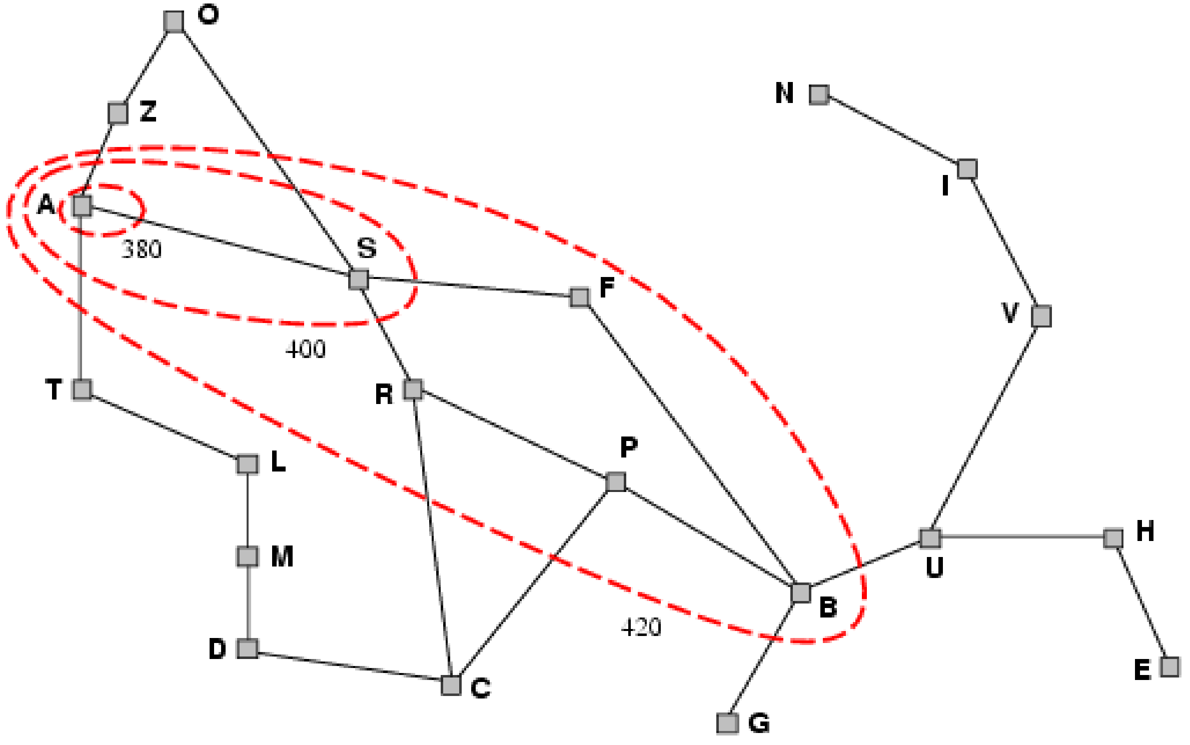
  $$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n,a,n') + h(n') \\ &\geq g(n) + h(n) \\ &= f(n) \end{aligned}$$

- i.e., *f(n)* is non-decreasing along any path.
- Theorem: If *h(n)* is consistent, A* using GRAPH-SEARCH is optimal

Optimality of A*

- A* expands nodes in order of increasing *f* value

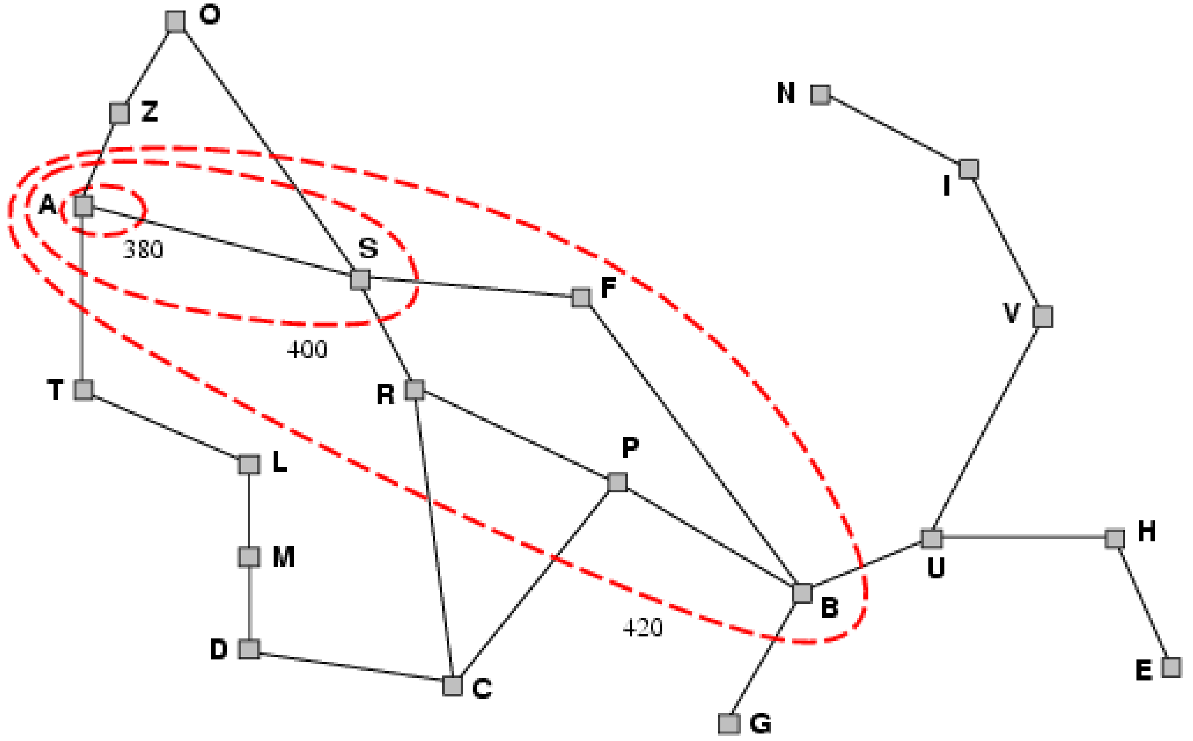
- Gradually adds "*f*-contours" of nodes
- Contour *i* has all nodes with $f=f_i$, where $f_i < f_{i+1}$



The behavior of A* Algorithm

Note that for w = 1 (without heuristic,—the contours are circles around the start state.

For w = 0, the contours degenerate in straight lines but for w = 0.5 the contours are ellipse-like with the long axis from start to goal.



Properties of A*

- Complete? Yes (unless there are infinitely many nodes with f ≤ *f(G)*)
- Time? Exponential
- Space? Keeps all nodes in memory
- Optimal? Yes

A* Algorithm

If f is the cost of the optimal path, A* expands all nodes $f^*(n) < f$

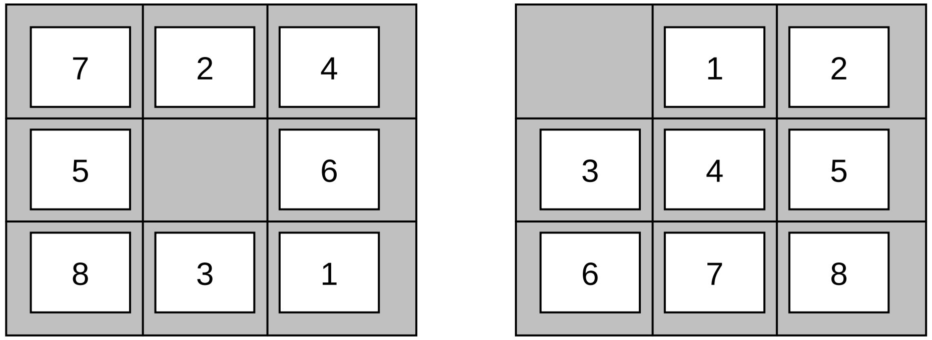
A* may expand some of the nodes right on the contour for which $f^*(n) = f$

The contours suggest that A* reaches the goal—since contours expand, so solution is always found.

The solution is optimal because nodes in subsequent contours have higher cost and the first solution found must be the optimal one.

Heuristic Functions

8-puzzle example



Start State   Goal State

A typical instance of the 8-puzzle. The solution is 26 steps long.

Two heuristics that underestimate the number of steps to goal are:

h_1 = number of tiles in wrong position. This Heuristic is intuitively correct since our goal is to put tiles in correct position, so h_1 is monotonically decreasing. Initially $h_1 = 8$.

h_2 = is the sum of Manhattan distances of all tiles from their goal positions. Here h_2 is initially

$h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$

Dominance

- If $h_2(n) \geq h_1(n)$ for all n (both admissible)
- then h_2 dominates h_1
- h_2 is better for search

- Typical search costs (average number of nodes expanded):

- *d=12* IDS = 3,644,035 nodes
  $A^*(h_1)$ = 227 nodes
  $A^*(h_2)$ = 73 nodes
- *d=24* IDS = too many nodes
  $A^*(h_1)$ = 39,135 nodes
  $A^*(h_2)$ = 1,641 nodes

IDS – iterative-deepening search

Inventing heuristic functions

Heuristic functions can be found by relaxing the constraints of the problem.

A moves to B if (A adjacent B) & (B=blank)

or in general

O operator if c_1 & c_2 & ... & c_n

(Perform operator O if all conditions c_1 and c_2 and ...c_n are satisfied).

By relaxing (violating) each condition c_i we may obtain a heuristic function.

This process may be automated, devise a program to generate and test heuristics.

In our example:

a) A moves to B if A adjacent to B

b) A moves to B if B is blank

c) A moves to B

The costs of exact solutions for these relaxed problems provide heuristics.

a) $\rightarrow h_2$; c) $\rightarrow h_1$

Game Trees

- Game trees
- Minimax
- Alpha-Beta Pruning
- Heuristic

Game trees

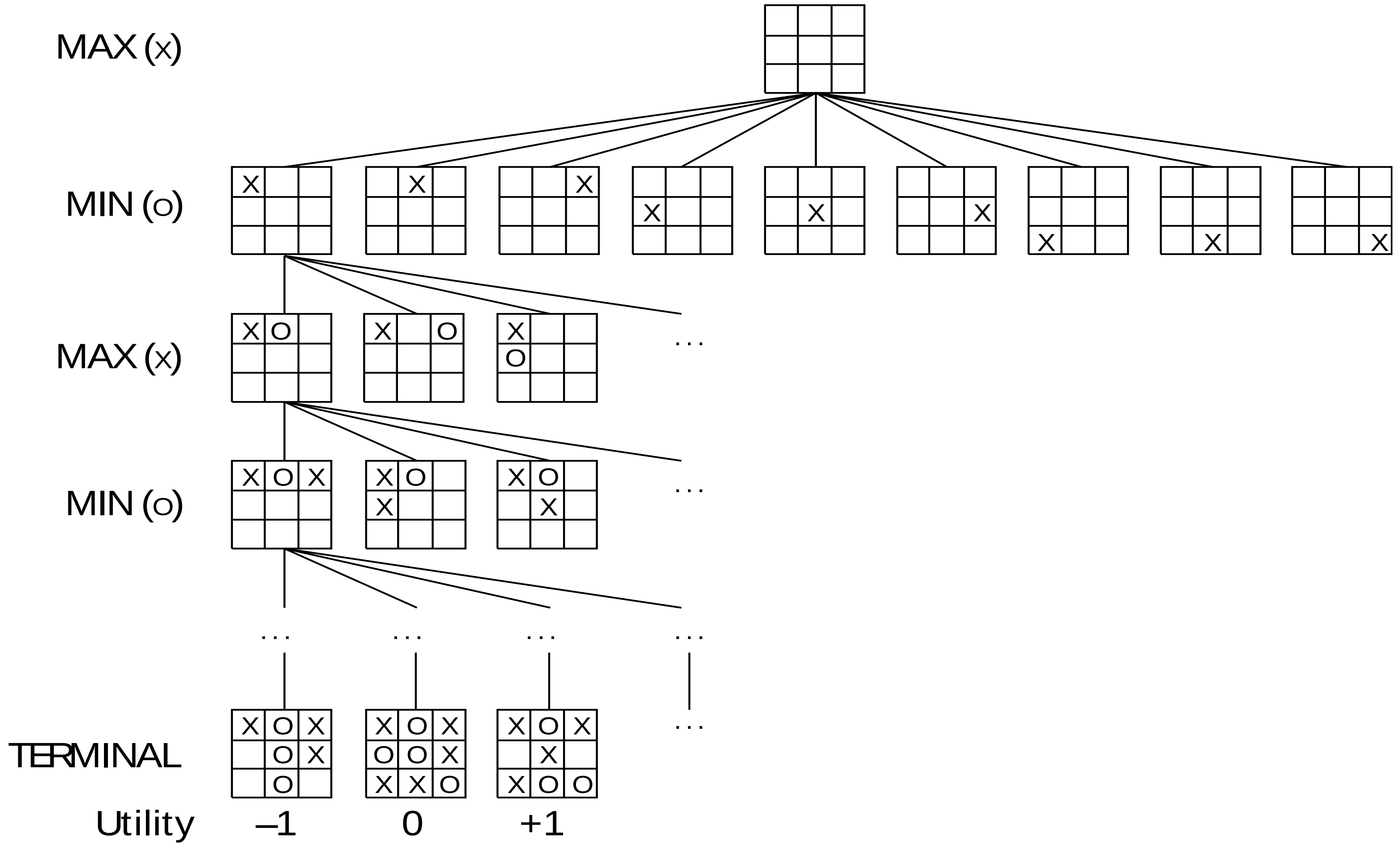
We consider games with two players who alternate in making moves. Each player has information about opponent options. (This is like in chess, and unlike in card games).

The game begins from a specified state and ends in a win for one player and loss for the other, or a draw.

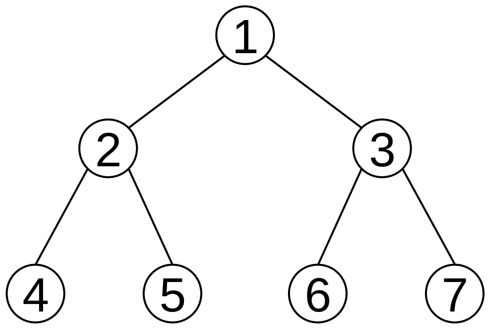
A game tree is a representation of all possible states of such game.

Nodes represent possible states (board situations) and branches represent how a state transforms into another as result of a player’s move.

Tic-tac-toe



Game Trees



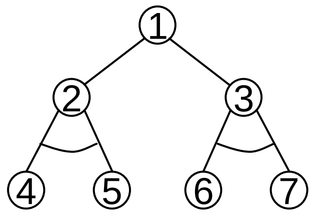
Player A

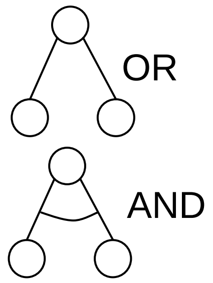
Player B

Player A – is the maximizer

Player B – is the minimizer

Often a game tree is drawn as an AND/OR tree.





From A’s point of view, in node 1, he has two choices to select from when maximizing the score. So node 1 is an OR node from A’s point of view.

However, nodes 2 and 3 from A’s point of view are AND nodes since they represent nodes where B makes moves and to which A responds.

Some issues

1. Plausible move generator

Move generator produces all legal situations derived from any given situation.

We are interested in good moves, so called plausible moves which are likely to advance situation towards the goal.

2. Static evaluation function

In order to choose the best move a player needs to quantify the "goodness" of the situation. Static evaluation function estimates how likely a situation leads to a win.

This function is similar to the heuristic function h in A*.

For example for chess we may have

--add the values of black pieces B

and the values of white pieces W

and take ratio W/B

(this was proposed by Turing).

Some issues

--capability for advancement

--control of center

--threat of fork, etc.

We may put weights to each component to have a more balanced function.

$f = c_1 \times \text{piece.advantage} + c_2 \times \text{center\_control} + \_\ \_\ \_\ \_$

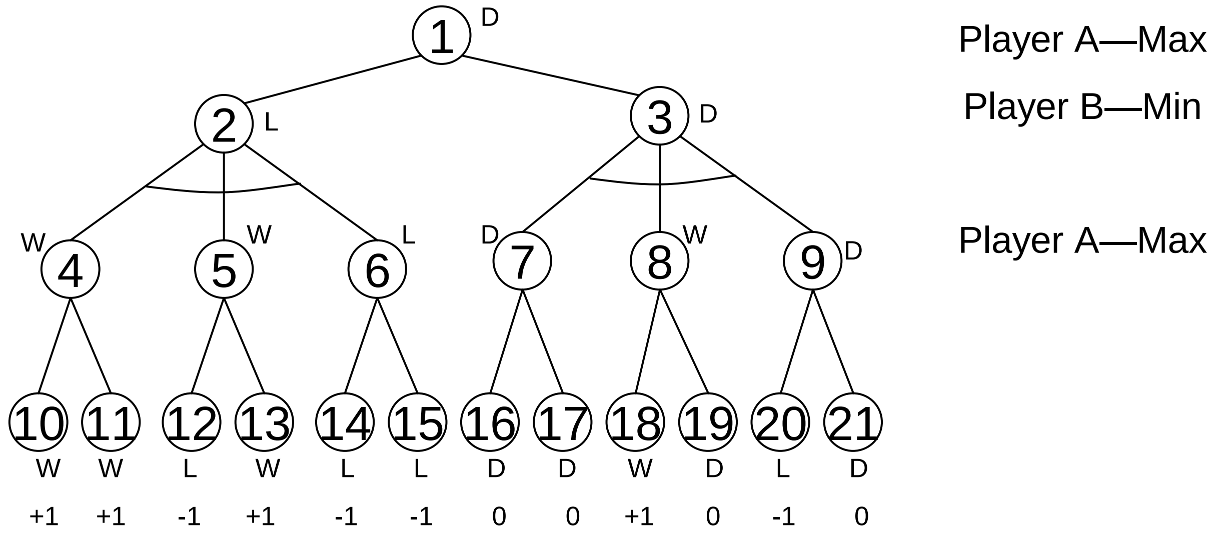
constants c_i may be learned.

Evaluation function needs to be computed only at leaf nodes.

3. Search method

The number of nodes is very large, so we need a search procedure. Here the situation is complicated because there are two players.

Minimax Search Procedure



W—win; +1

L—lose; -1

D—draw; 0

This represents a tree from MAX's point of view.

According to the minimax procedure, MAX player (starting player) should move to node 2 or 3, whichever maximizes his gain.

Max player expects that at nodes 2 or 3, Min player minimizes his own loss. And so on.

Minimax Search Procedure

Given the values of terminal/leaf nodes, the values of the other nodes are computed.

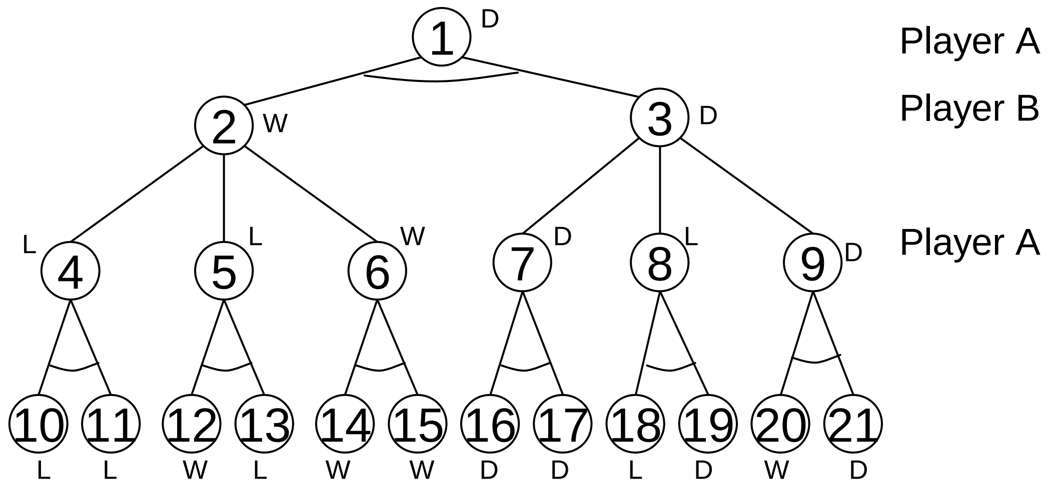
Node 2 evaluates to a loss for A and node 3 evaluates to a draw. So A moves to 3, and B moves to 7, and A to either 16 or 17 ending in a draw.

Rules

1. The value to player A (max) of a node with OR successors (a node from where A chooses next move) is the maximum value of any of its successors.
2. The value to player A of a node with AND successors (a node where B chooses next move) is the minimum value of any of its successors.

Minimax Search Procedure

From B's (Min) point of view the game tree is:



Note that the values of the leaf nodes are reversed.

OR and AND nodes are reversed.

Here Player B maximizes and Player A minimizes.

The result is the same

Minimax Continued

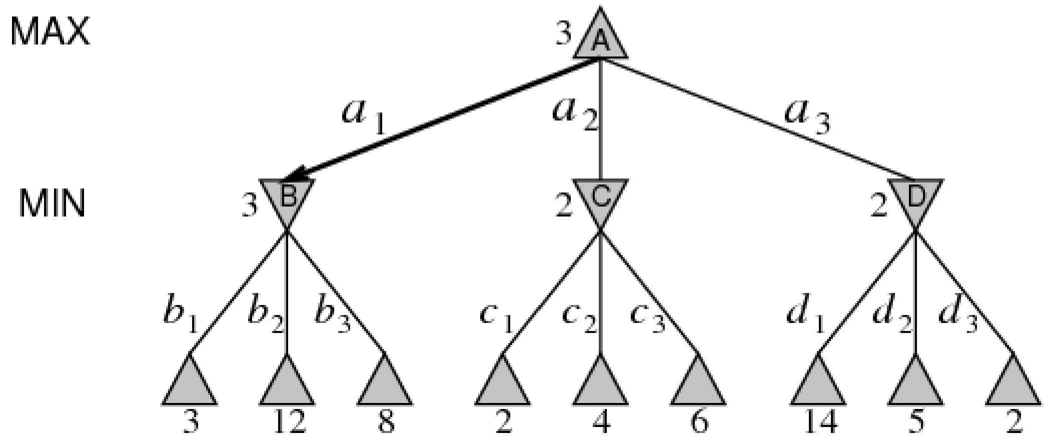


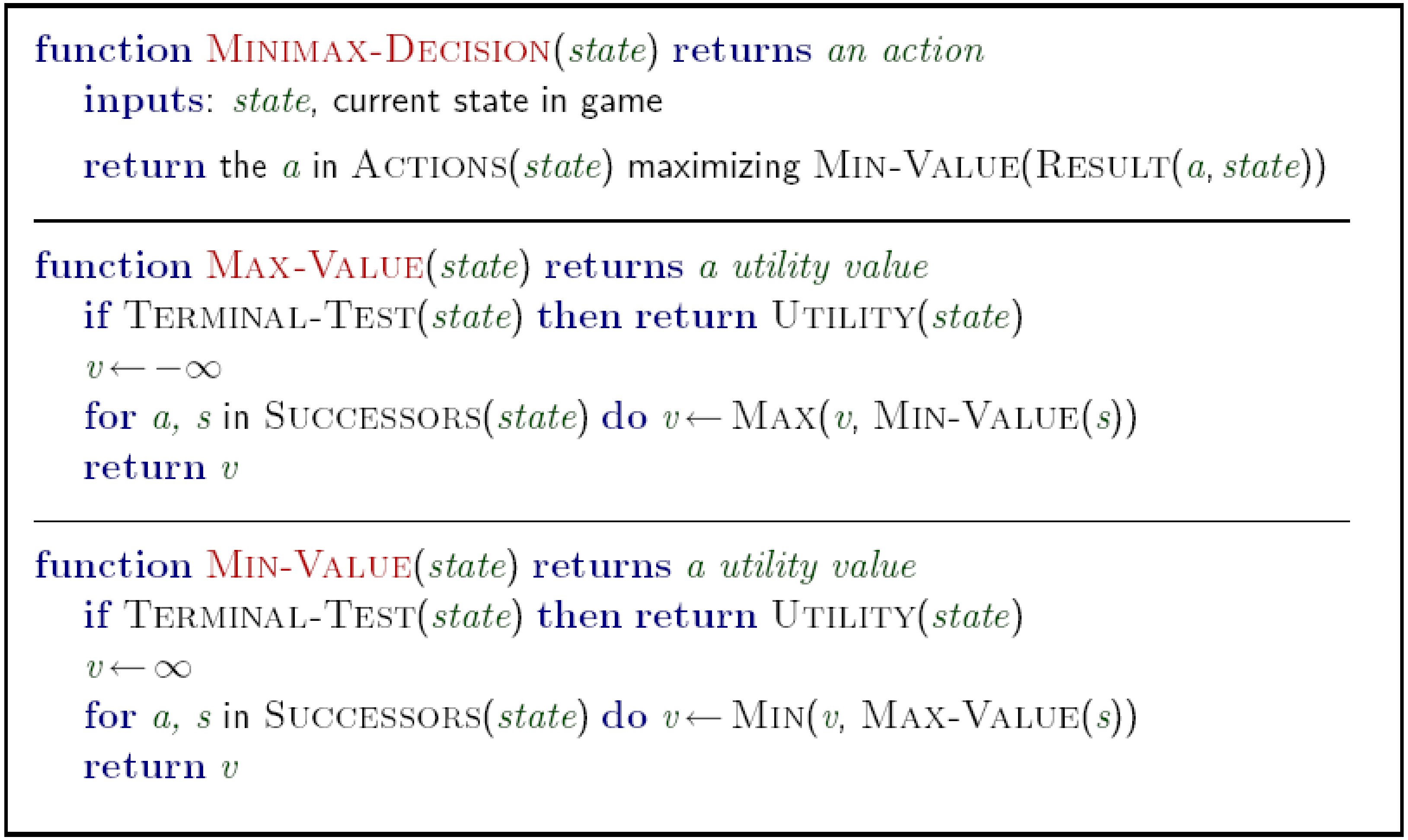
Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX'S turn to move, and the ▽ nodes are "MIN nodes." the terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the successor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

MINIMAX-VALUE(s)=

$$\begin{cases} \text{UTILITY}(s) & \text{if Terminal-test}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX(Result}(s,a)) & \text{if Player}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX(Result}(s,a)) & \text{if Player}(s) = \text{MIN} \end{cases}$$

where s is the state that corresponds to node n
a is one of actions from state s

Minimax Algorithm

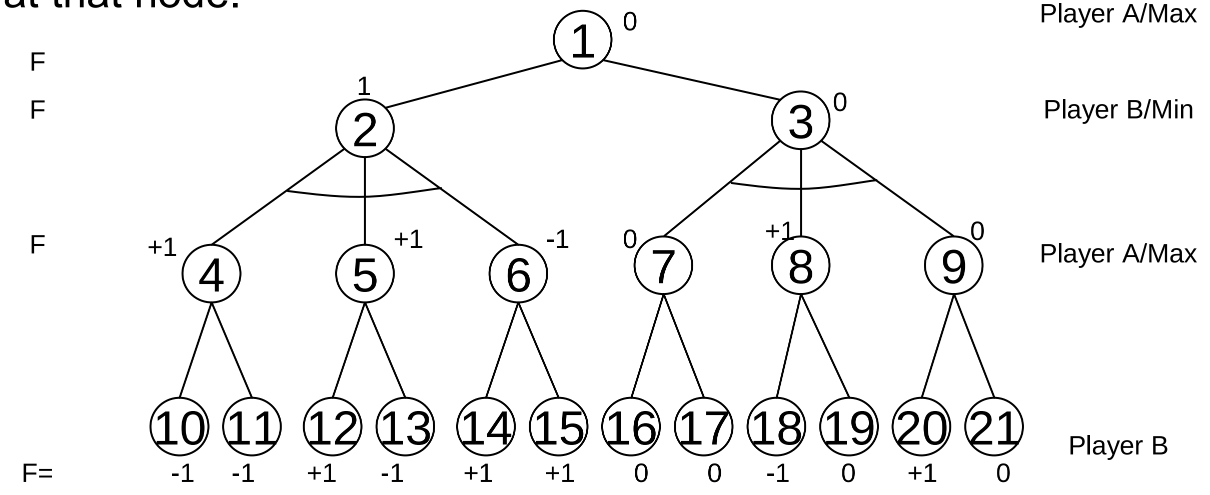


```
function MINIMAX-DECISION($state$) returns $an\ action$
    inputs: $state$, current state in game
    return the $a$ in ACTIONS($state$) maximizing MIN-VALUE(RESULT($a$, $state$))

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

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

Negmax (Knuth and Moore Algorithm, 1975)

Looks at the tree from both players point of view, thus unifies the previous two figures.

Rule 1. The value given to each node is the value to the player whose turn is to make a move at that node.



Rule 2. $F(n) = \max\{-F(n_1),\ldots -F(n_k)\}$

where n is a node, and

$n_1,\ldots n_k$ are its successors.

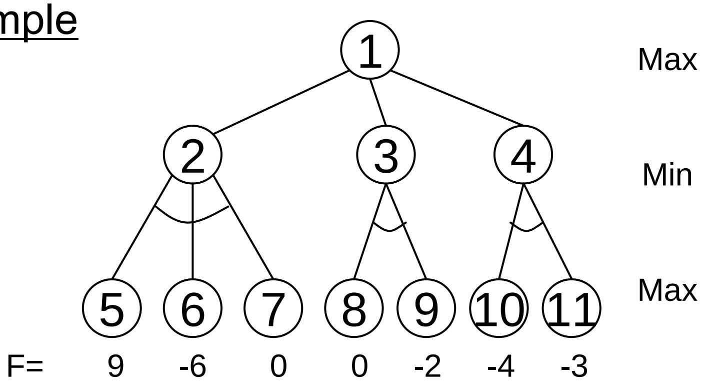
The value of F at a node is the maximum of the negative functions of successor nodes derived from that node.

Note that we start with F set for Player B.

Negmax (Knuth and Moore Algorithm, 1975)

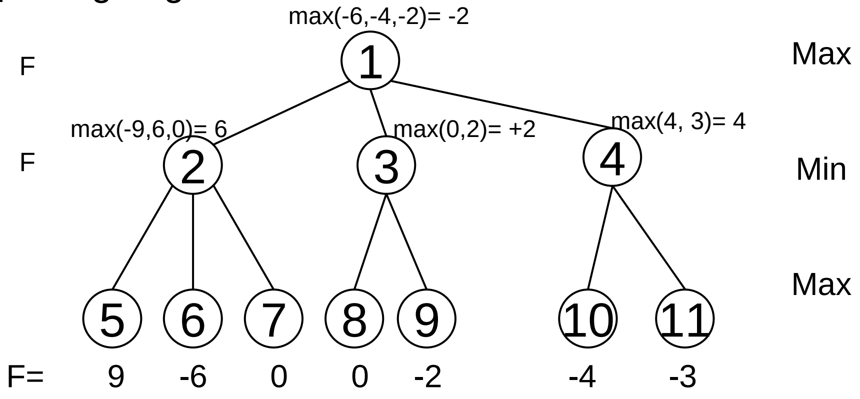
Observe that negmax is equivalent to minimax

Another example



Method 1. using minimax: max selects node 3 and min selects 9 with value -2. This is the best max can do.

Method 2. using negmax:



Note that the leaf values remain unchanged because they are at Max level.

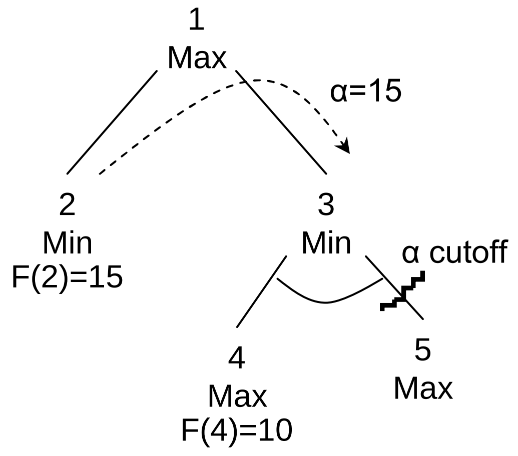
Alpha-Beta Pruning Procedure

The idea is to use the branch and bound strategy to eliminate a path that is worse than the one already found.

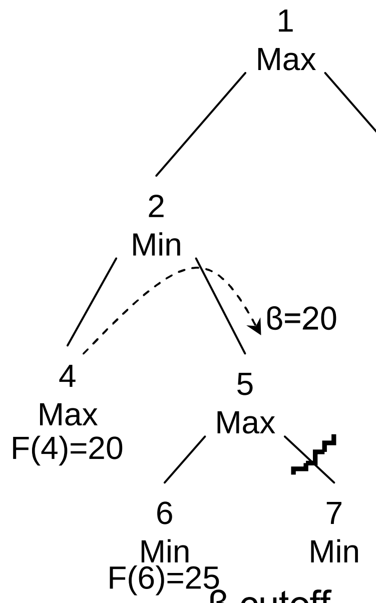
Here it is necessary to modify the branch and bound to include two bounds; one for each player.

α—is the lower bound of maximizing player

β—is the upper bound of minimizing player



α cutoff
(pruning)



ß cutoff
(pruning)

Alpha-Beta Pruning Procedure

In <u>α-cutoff</u> nodes 2 and 4 have been evaluated—either by the static function or by backing up from descendents (not shown here).

For player Max, the maximum of 2 is 15. However, node 3 can offer a maximum of 10. To understand consider two situations:

- Node 5 is smaller than 10, player Min will select node 5 but is irrelevant since Max player is sure to get 15 from node 2
- Node 5 is larger than 10, in which case player Min disregards it in favor of node 4.

So, node 3 which receives the lower bound α = 15 does not need to evaluate node 5 and any subsequent nodes. Node 5 is α cutoff, or α pruning.

Alpha-Beta Pruning Procedure

In β-cutoff example, the minimum value of node 2 is 20 as given by node 4. Any other value at node 5 larger than 20 is not going to be selected by Min player at node 2.

The maximizer at node 5 gets the maximum of at least 25 from node 6.

This value is larger than 20, that already Minimizer player at node 2 can have, so node 7 is not explored, is a β cutoff.

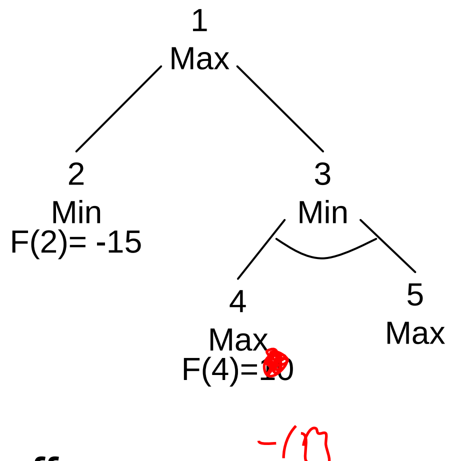
The Alpha-Beta procedure starts from node 1 where two parameters are set

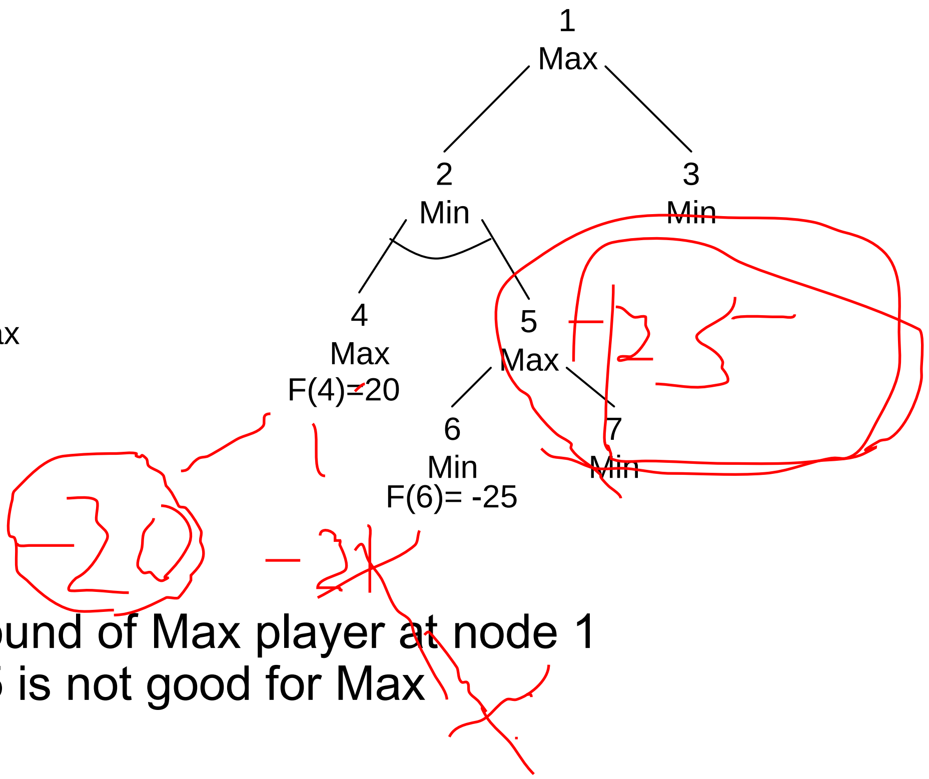
$\alpha = -\infty$

$\beta = +\infty$

As nodes are visited, α and β may be increased or decreased respectively, and branches cutoff.

Alpha-Beta in Negmax representation





α cutoff

- F(3) = max(-10,...)
- F(1) = max(-15, -10,...)
- α = -15 is the lower bound of Max player at node 1
- Anything less than -15 is not good for Max

β cutoff

- F(5) = max(-25,...)
- F(2) = max(-20, -25, ...)
- β = 20 is the upper bound of Min player at 2

Note: α comes from F at Min and
β comes from F at Max player.

Alpha-Beta continued

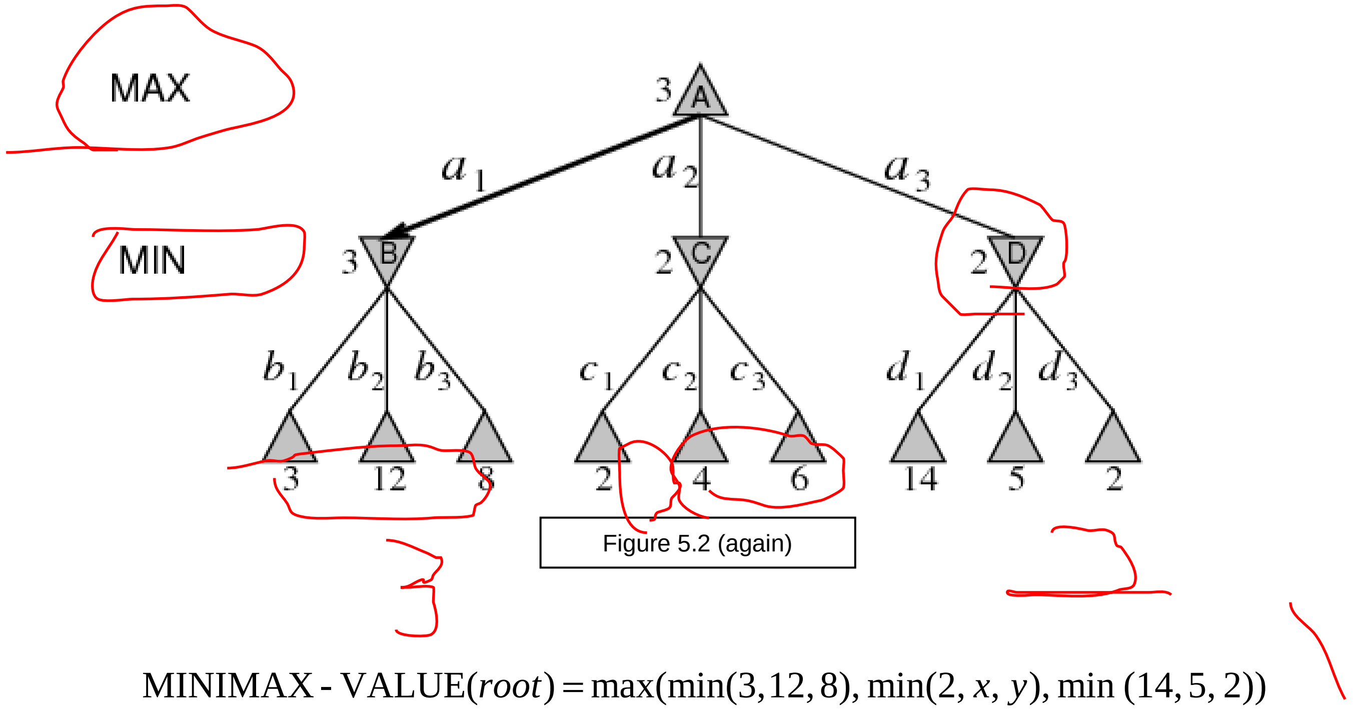
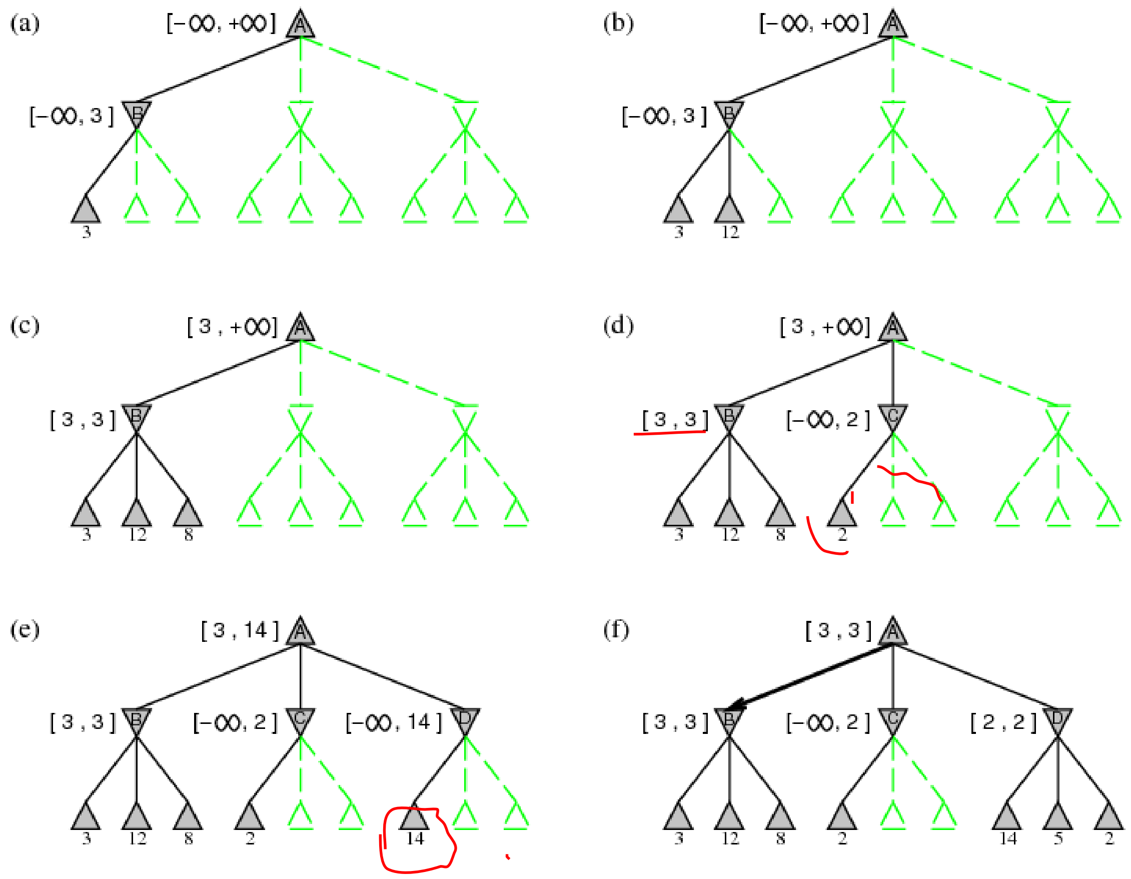


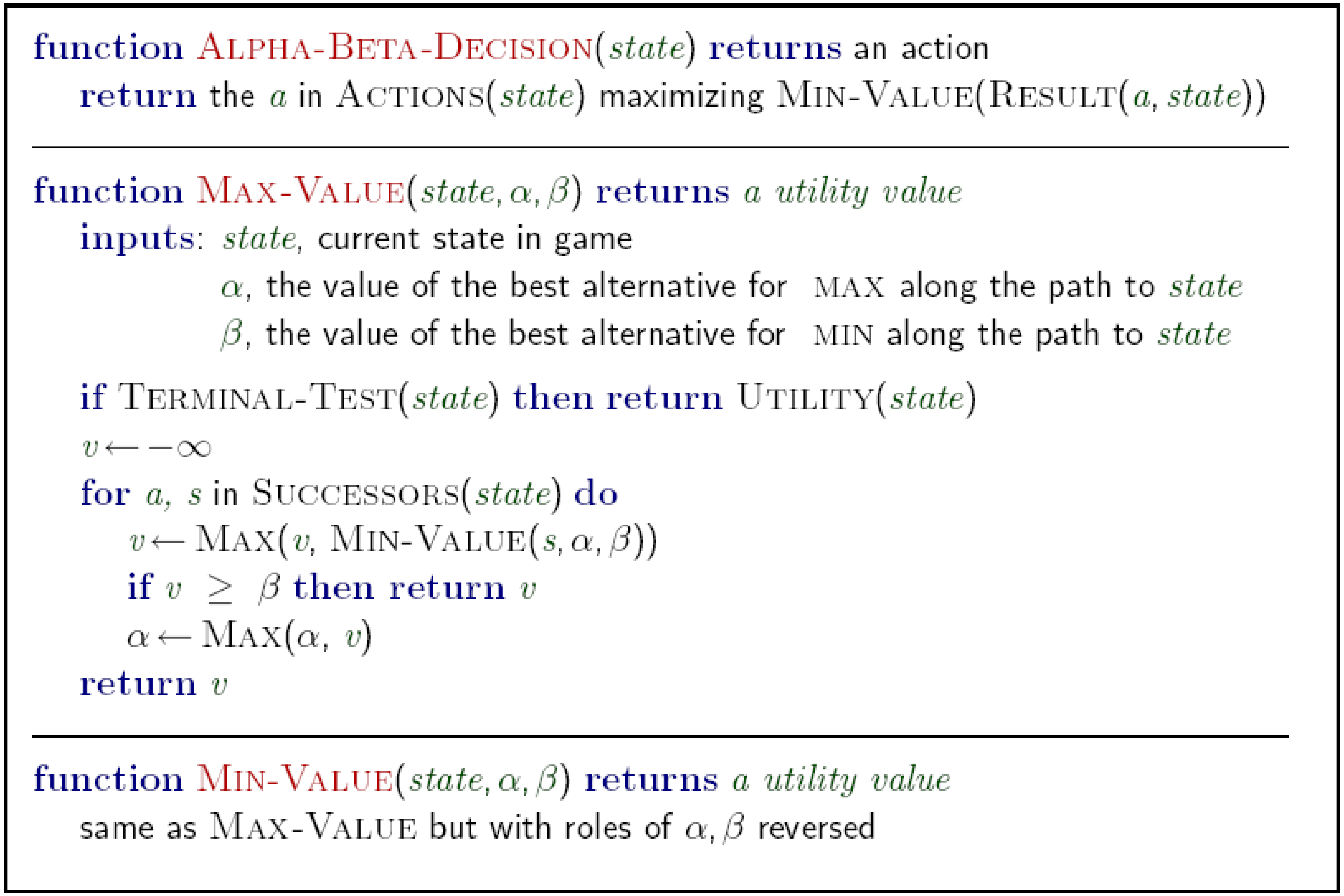
Figure 5.2 (again)

$$
\begin{aligned}
\text{MINIMAX-VALUE}(root) &= \max(\min(3,12,8), \min(2,x,y), \min(14,5,2)) \\
&= \max(3, \min(2, x, y), 2) \\
&= \max(3, z, 2) \qquad \text{where } z \le 2 \\
&= 3.
\end{aligned}
$$

Alpha-Beta continued



Alpha-Beta Procedure



```
function ALPHA-BETA-DECISION($state$) returns an action
    return the $a$ in ACTIONS($state$) maximizing MIN-VALUE(RESULT($a$, $state$))
```

```
function MAX-VALUE($state, \alpha, \beta$) returns $a$ $utility$ $value$
    inputs: $state$, current state in game
            $\alpha$, the value of the best alternative for MAX along the path to $state$
            $\beta$, the value of the best alternative for MIN along the path to $state$

    if TERMINAL-TEST($state$) then return UTILITY($state$)
    $v \leftarrow -\infty$
    for $a$, $s$ in SUCCESSORS($state$) do
        $v \leftarrow$ MAX($v$, MIN-VALUE($s, \alpha, \beta$))
        if $v \geq \beta$ then return $v$
        $\alpha \leftarrow$ MAX($\alpha$, $v$)
    return $v$
```

```
function MIN-VALUE($state, \alpha, \beta$) returns $a$ $utility$ $value$
    same as MAX-VALUE but with roles of $\alpha, \beta$ reversed
```

Multiplayer Games

- Consider now there are 3 players A, B, C
- Minimax values are replaced by vector values for each node. (v_A, v_B, v_C)

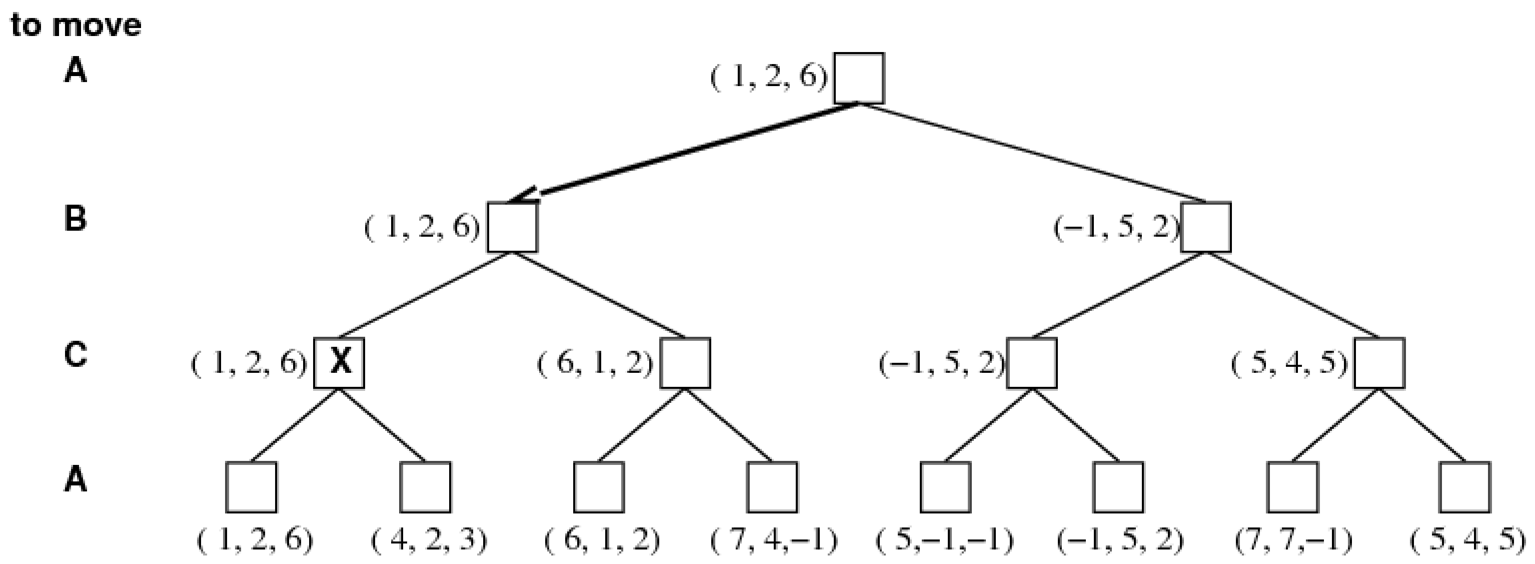


Figure 5.4 The first three ply of a game tree with three players (*A, B, C*). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

Games that include the element of chance

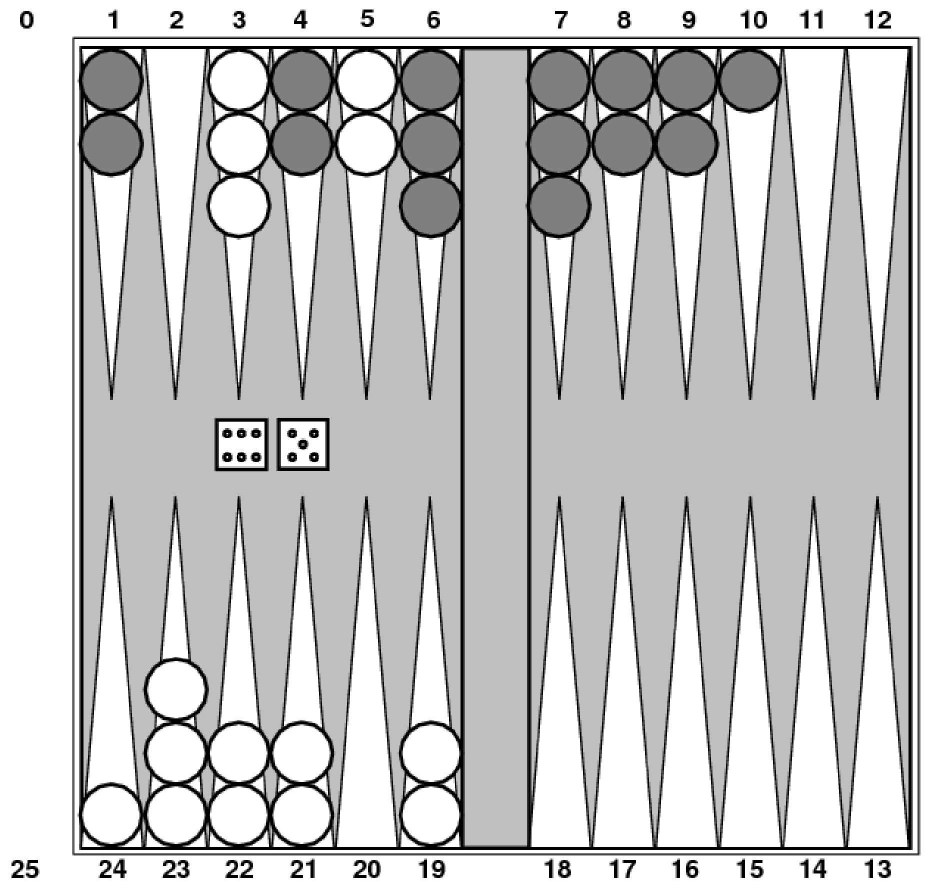


Figure 5.10 A typical backgammon position. The goal of the game is to move all one's pieces off the board. White moves clockwise toward 25, and black moves counterclockwise toward 0. A piece can move to any position unless there are multiple opponent pieces there; if there is one opponent, it is captured and must start over. In the position shown, White has rolled 6-5 and must choose among four legal moves: (5-10, 5-11), (5-11, 19-24), (5-10, 10-16), and (5-11, 11-16).

Games that include the element of chance

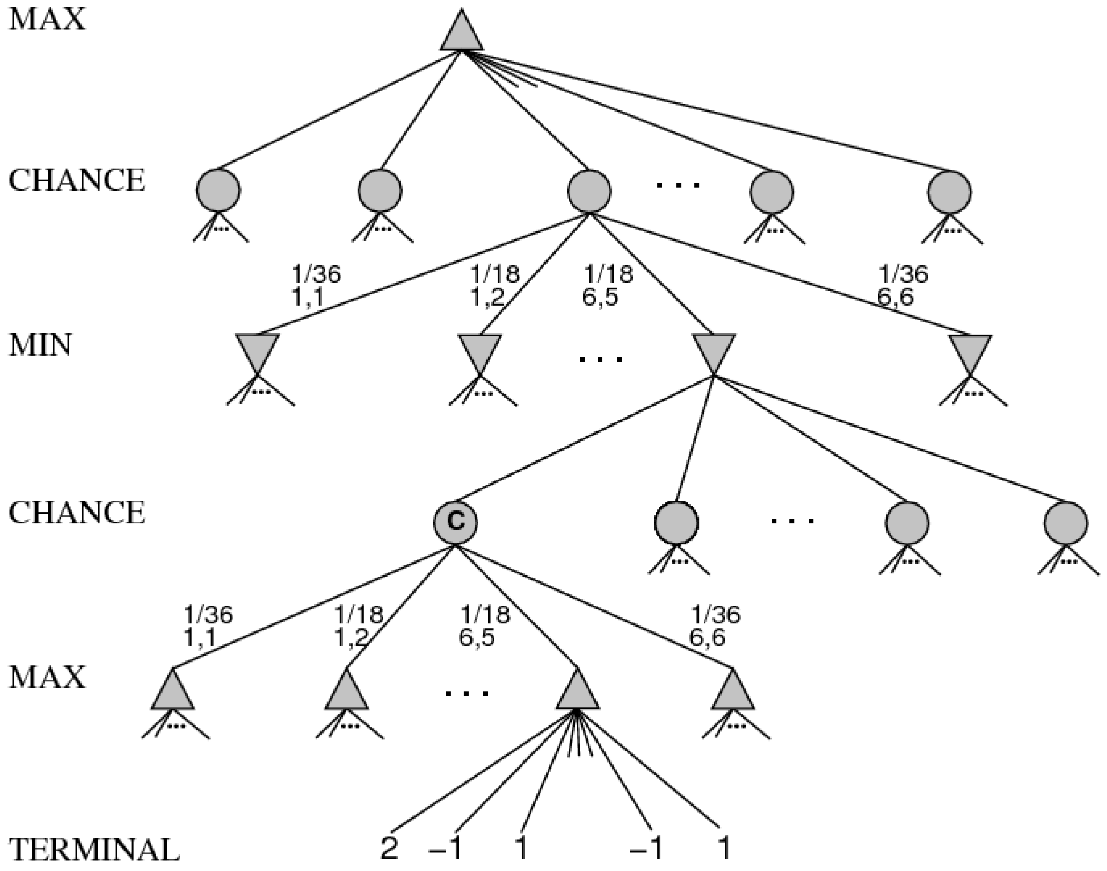


Figure 5.11 Schematic game tree for a backgammon position.

Games that include the element of chance

- Here we can not calculate the minmax value, but have expected value instead.
- Minimax function for deterministic games is generalized to expectiminimax function for games with chance nodes.

$$\text{EXPECTIMINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if Terminal-Test}(s) \\ \max_a \text{EXPECTIMINIMAX}(\text{Result}(s,a)) & \text{if Player}(s)\text{=MAX} \\ \min_a \text{EXPECTIMINIMAX}(\text{Result}(s,a)) & \text{if Player}(s)\text{=MIN} \\ \sum_r P(r) \cdot \text{EXPECTIMINIMAX}(\text{Result}(s,r)) & \text{if Player}(s)\text{=CHANCE} \end{cases}$$

Where *r* represents possible dice roll

Result(*s*,*r*) is the same state as *s*, with the additional fact that the result of the dice roll is *r*.

a is successor of state *s*