Alpha-Beta Pruning: Algorithm and Analysis

Tsan-sheng Hsu

徐讚昇

tshsu@iis.sinica.edu.tw

http://www.iis.sinica.edu.tw/~tshsu

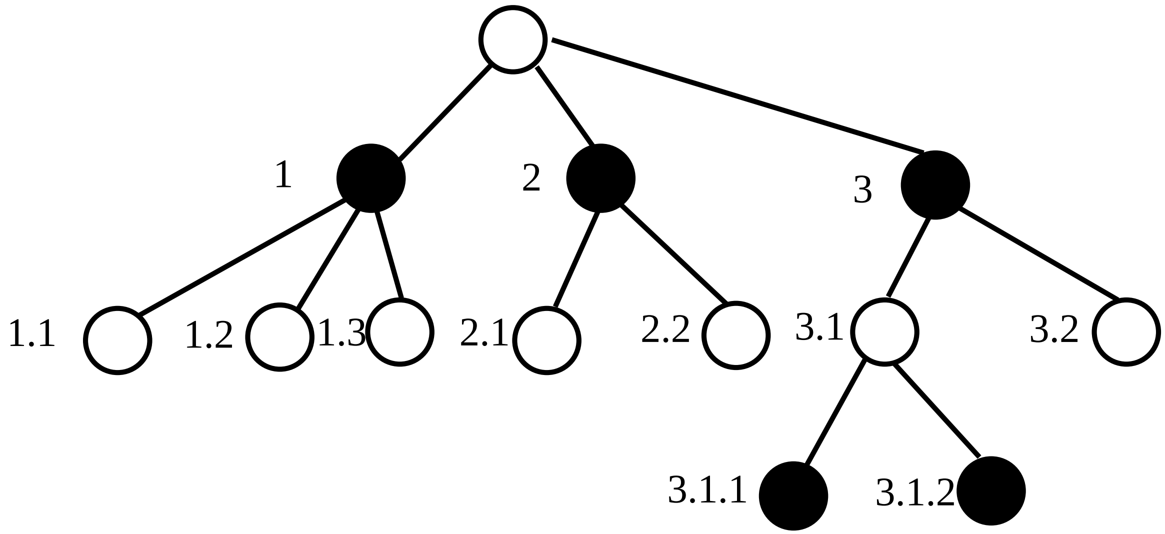
Abstract

- **Tree node numbering**
- **Exhaustive mini-max search and its negamax version**
- **Ideas for cut off**
  - **Alpha cut**
  - **Beta cut**
- **Alpha-beta cut off**
  - **Algorithm**
  - **Proof of performance**
    - *Categorize nodes of different cutting properties*
  - **Variations**
    - *Original (fail hard)*
    - *One-sided*
    - *Fail soft*

Introduction

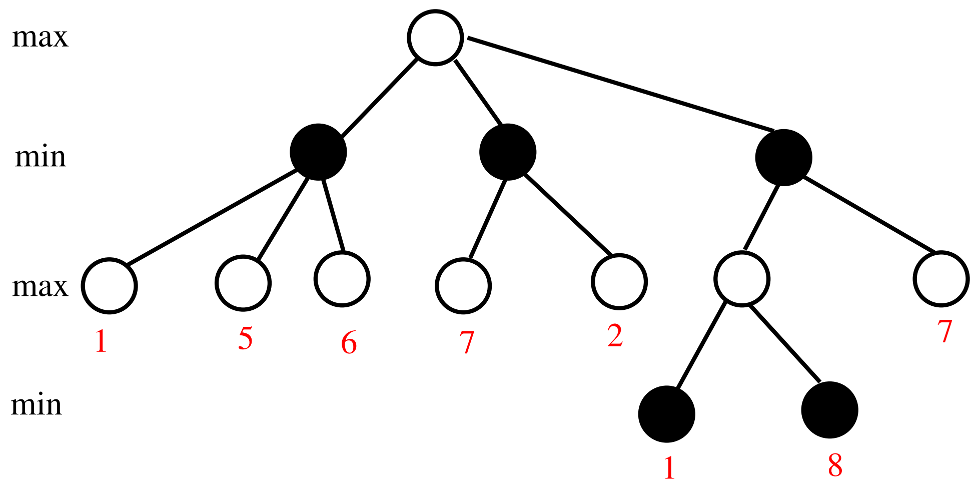
- **Alpha-beta pruning is the standard searching procedure used for solving 2-person perfect-information zero sum games exactly.**
- **Definitions:**
  - **A *position* p.**
  - **The value of a position p, $f(p)$, is a numerical value computed from evaluating p.**
    - *Value is computed from the root player's point of view.*
    - *Positive values mean in favor of the root player.*
    - *Negative values mean in favor of the opponent.*
    - *Since it is a zero sum game, thus from the opponent's point of view, the value can be assigned $-f(p)$.*
  - **A terminal position: a position whose value can be decided.**
    - *A position where win/loss/draw can be concluded.*
    - *In practice, we encounter a position where some constraints, e.g., time limit and depth limit, are met.*
  - **A position p has b legal moves $p_1, p_2, \ldots, p_b$.**

Tree node numbering



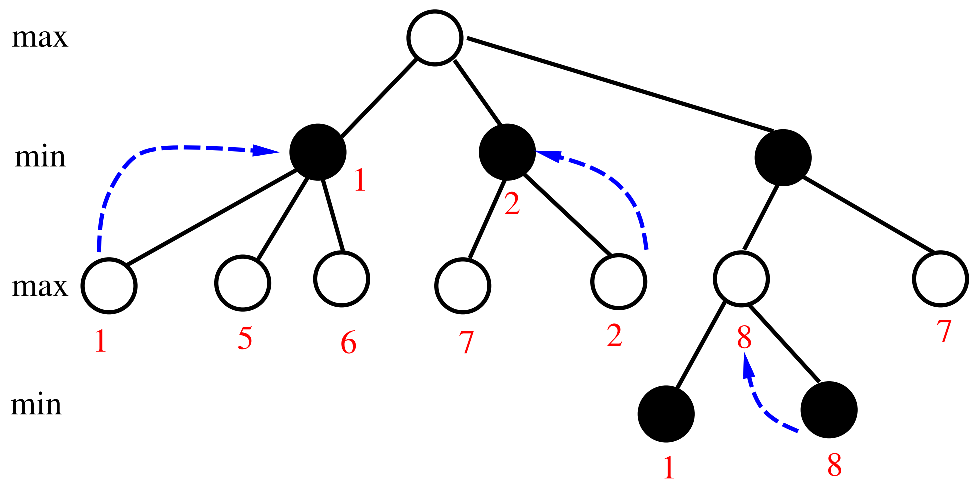
- **From the root, number a node in a search tree by a sequence of integers $a_1.a_2.a_3.a_4 \cdots$**
  - **Meaning from the root, you first take the a_1th branch, then the a_2th branch, and then the a_3th branch, and then the a_4th branch $\cdots$**
  - **The root is specified as an empty sequence.**
  - **The depth of a node is the length of the sequence of integers specifying it.**
- **This is called "Dewey decimal system."**

Mini-max formulation



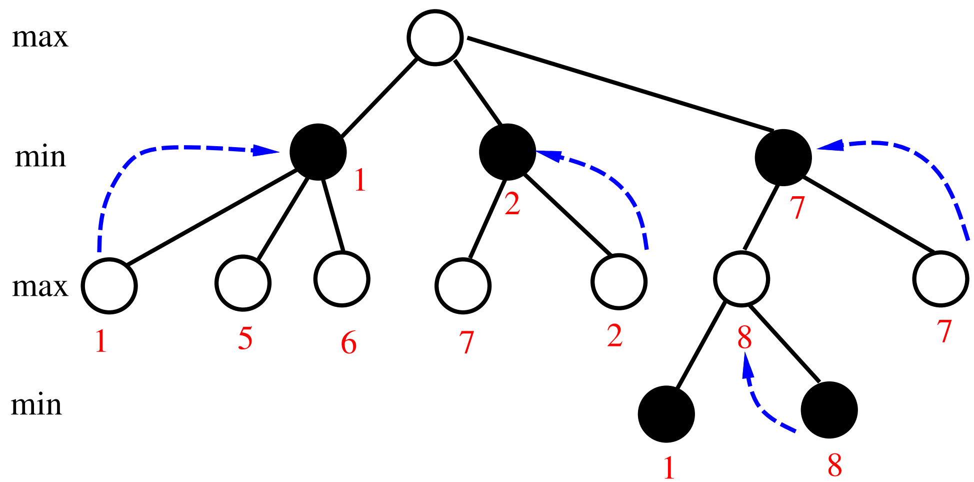
- **Mini-max formulation:**
  - $$F'(p) = \begin{cases} f(p) & \textbf{if } b = 0 \\ max\{G'(p_1), \ldots, G'(p_b)\} & \textbf{if } b > 0 \end{cases}$$
  - $$G'(p) = \begin{cases} f(p) & \textbf{if } b = 0 \\ min\{F'(p_1), \ldots, F'(p_b)\} & \textbf{if } b > 0 \end{cases}$$
  - **An indirect recursive formula with a bottom-up evaluation!**
  - **Equivalent to AND-OR logic.**

Mini-max formulation



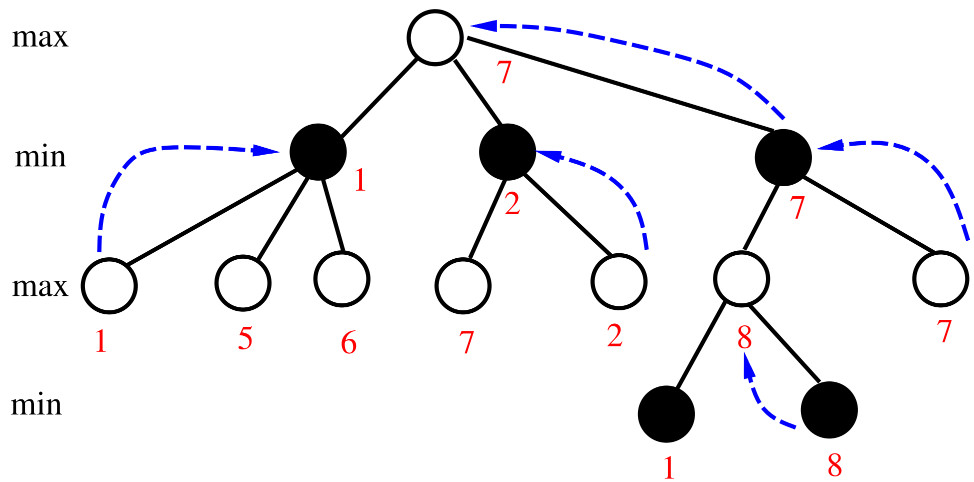
- **Mini-max formulation:**
  - $$F'(p) = \begin{cases} f(p) & \textbf{if } b = 0 \\ max\{G'(p_1), \ldots, G'(p_b)\} & \textbf{if } b > 0 \end{cases}$$
  - $$G'(p) = \begin{cases} f(p) & \textbf{if } b = 0 \\ min\{F'(p_1), \ldots, F'(p_b)\} & \textbf{if } b > 0 \end{cases}$$
  - **An indirect recursive formula with a bottom-up evaluation!**
  - **Equivalent to AND-OR logic.**

Mini-max formulation



- **Mini-max formulation:**
  - $$F'(p) = \begin{cases} f(p) & \textbf{if } b = 0 \\ max\{G'(p_1), \ldots, G'(p_b)\} & \textbf{if } b > 0 \end{cases}$$
  - $$G'(p) = \begin{cases} f(p) & \textbf{if } b = 0 \\ min\{F'(p_1), \ldots, F'(p_b)\} & \textbf{if } b > 0 \end{cases}$$
  - **An indirect recursive formula with a bottom-up evaluation!**
  - **Equivalent to AND-OR logic.**

Mini-max formulation



- **Mini-max formulation:**
  - $$F'(p) = \begin{cases} f(p) & \textbf{if } b = 0 \\ max\{G'(p_1), \ldots, G'(p_b)\} & \textbf{if } b > 0 \end{cases}$$
  - $$G'(p) = \begin{cases} f(p) & \textbf{if } b = 0 \\ min\{F'(p_1), \ldots, F'(p_b)\} & \textbf{if } b > 0 \end{cases}$$
  - **An indirect recursive formula with a bottom-up evaluation!**
  - **Equivalent to AND-OR logic.**

Algorithm: Mini-max (native)

- **Algorithm F'(position p)** // max node
  - determine the successor positions $p_1, \ldots, p_b$
  - if $b = 0$, then return $f(p)$ else begin
    - ▷ $m := -\infty$
    - ▷ *for* $i := 1$ *to* b *do*
    - ▷ $\quad t := G'(p_i)$
    - ▷ $\quad$ *if* $t > m$ *then* $m := t$ // *find max value*
  - end;
  - return m
- **Algorithm G'(position p)** // min node
  - determine the successor positions $p_1, \ldots, p_b$
  - if $b = 0$, then return $f(p)$ else begin
    - ▷ $m := \infty$
    - ▷ *for* $i := 1$ *to* b *do*
    - ▷ $\quad t := F'(p_i)$
    - ▷ $\quad$ *if* $t < m$ *then* $m := t$ // *find min value*
  - end;
  - return m

Mini-max: comments

- A brute-force method to try all possibilities!
  - May visit a position many times.
- Depth-first search
  - Move ordering is according to the order the successor positions are generated.
  - Bottom-up evaluation.
  - Post-ordering traversal.
- Q:
  - Iterative deepening?
  - BFS?
  - Other types of searching?

Mini-max: depth limited (1/2)

- **Search a max-node position p with a depth limit of $depth$.**
- **Algorithm $F0'$(position p, integer $depth$) // max node**
  - **determine the successor positions $p_1, \ldots, p_b$**
  - **if $b = 0$ // a terminal node**
    or $depth = 0$ // remaining depth to search
    or time is running up // from timing control
    or some other constraints are met // add knowledge here
    then return $f(p)$// current board value
    else begin
    - ▷ $m := -\infty$ // *initial value*
    - ▷ ***for*** $i := 1$ ***to*** b ***do*** // *try each child*
    - ▷ ***begin***
    - ▷ $t := G0'(p_i, depth - 1)$
    - ▷ ***if*** $t > m$ ***then*** $m := t$ // *find max value*
    - ▷ ***end***

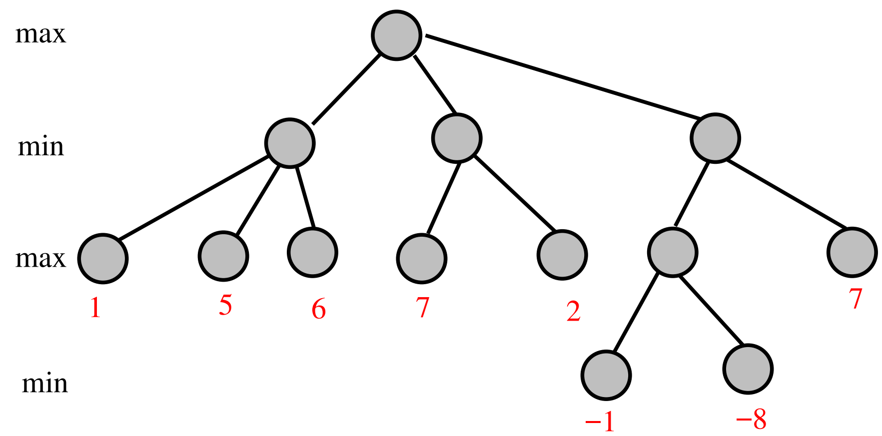
    end
  - **return m**

Mini-max: depth limited (2/2)

- **Search a min-node position p with a depth limit of $depth$.**
- **Algorithm $G0'$(position p, integer $depth$) // min node**
  - **determine the successor positions $p_1, \ldots, p_b$**
  - **if $b = 0$ // a terminal node**
    or $depth = 0$ // remaining depth to search
    or time is running up // from timing control
    or some other constraints are met // add knowledge here
    then return $f(p)$// current board value
    else begin
    - ▷ $m := \infty$ // *initial value*
    - ▷ ***for*** $i := 1$ ***to*** b ***do*** // *try each child*
    - ▷ ***begin***
    - ▷ $t := F0'(p_i, depth - 1)$
    - ▷ ***if*** $t < m$ ***then*** $m := t$ // *find min value*
    - ▷ ***end***

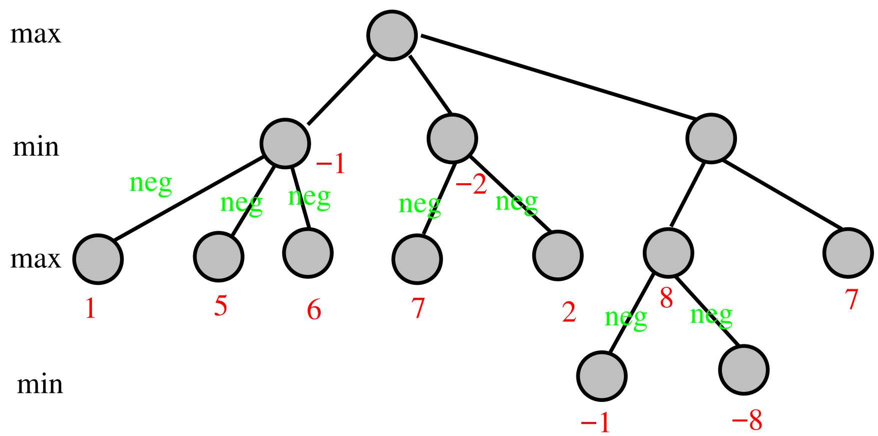
    end
  - **return m**

Nega-max formulation



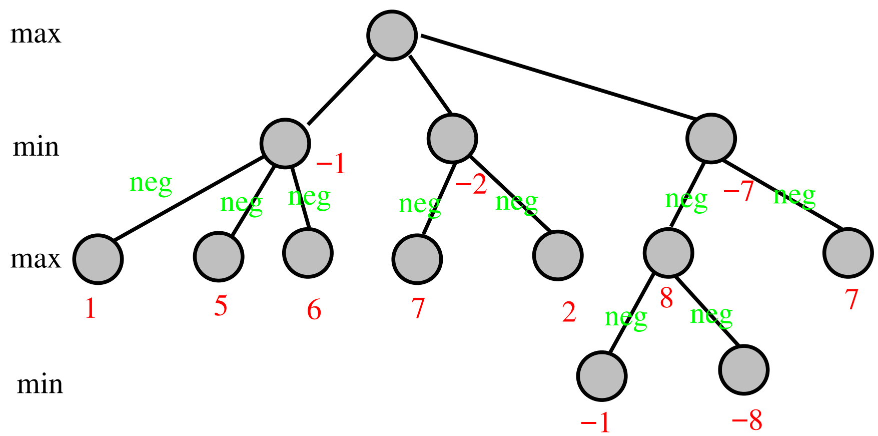
- **Nega-max formulation:**
  Let $F(p)$ be the greatest possible value achievable from position p against the optimal defensive strategy.
  - $$F(p) = \begin{cases} h(p) & \textbf{if } b = 0 \\ max\{-F(p_1), \ldots, -F(p_b)\} & \textbf{if } b > 0 \end{cases}$$
    - ▷ $$h(p) = \begin{cases} f(p) & \textit{if depth of } p \textit{ is 0 or even} \\ -f(p) & \textit{if depth of } p \textit{ is odd} \end{cases}$$
    - ▷ ***$h(p)$ is the position's value from the point of view of the player of p.***

Nega-max formulation



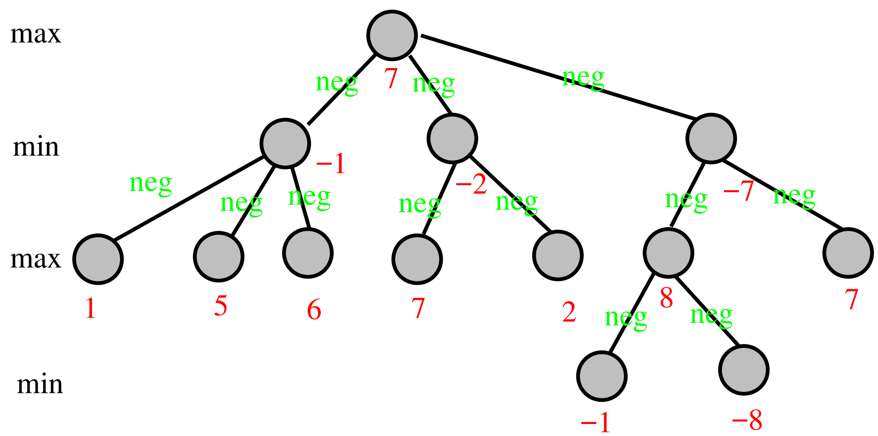
- **Nega-max formulation:**
  Let $F(p)$ be the greatest possible value achievable from position p against the optimal defensive strategy.
  - $$F(p) = \begin{cases} h(p) & \textbf{if } b = 0 \\ max\{-F(p_1), \ldots, -F(p_b)\} & \textbf{if } b > 0 \end{cases}$$
    - ▷ $$h(p) = \begin{cases} f(p) & \textit{if depth of } p \textit{ is 0 or even} \\ -f(p) & \textit{if depth of } p \textit{ is odd} \end{cases}$$
    - ▷ ***$h(p)$ is the position's value from the point of view of the player of p.***

Nega-max formulation



- **Nega-max formulation:**
  Let $F(p)$ be the greatest possible value achievable from position p against the optimal defensive strategy.
  - $$F(p) = \begin{cases} h(p) & \textbf{if } b = 0 \\ max\{-F(p_1), \ldots, -F(p_b)\} & \textbf{if } b > 0 \end{cases}$$
    - $$h(p) = \begin{cases} f(p) & \textit{\textbf{if depth of } } p \textit{\textbf{ is 0 or even}} \\ -f(p) & \textit{\textbf{if depth of } } p \textit{\textbf{ is odd}} \end{cases}$$
    - ***$h(p)$ is the position's value from the point of view of the player of p.***

Nega-max formulation



- **Nega-max formulation:**
  Let $F(p)$ be the greatest possible value achievable from position p against the optimal defensive strategy.
  - $$F(p) = \begin{cases} h(p) & \textbf{if } b = 0 \\ max\{-F(p_1), \ldots, -F(p_b)\} & \textbf{if } b > 0 \end{cases}$$
    - ▷ $$h(p) = \begin{cases} f(p) & \textit{\textbf{if depth of} } p \textit{\textbf{ is 0 or even}} \\ -f(p) & \textit{\textbf{if depth of} } p \textit{\textbf{ is odd}} \end{cases}$$
    - ▷ ***$h(p)$ is the position's value from the point of view of the player of p.***

Algorithm: Nega-max (native)

- **Algorithm F(position p)**
  - **determine the successor positions $p_1, \ldots, p_b$**
  - **if $b = 0$ // a terminal node**
  - **then return $h(p)$ else**
  - **begin**
    - ▷ $m := -\infty$
    - ▷ ***for*** $i := 1$ ***to*** b ***do***
    - ▷ ***begin***
    - ▷ $t := -F(p_i)$ ***// recursive call, the returned value is negated***
    - ▷ ***if*** $t > m$ ***then*** $m := t$ ***// always find a max value***
    - ▷ ***end***
  - **end**
  - **return m**

Algorithm: Nega-max (depth limited)

- **Algorithm** $F0$**(position** p**, integer** $depth$**)**
  - **determine the successor positions** $p_1, \ldots, p_b$
  - **if** $b = 0$ **// a terminal node**
    or $depth = 0$ **// remaining depth to search**
    or time is running up // from timing control
    or some other constraints are met // add knowledge here
  - **then return** $h(p)$ **else**
  - **begin**
    - $m := -\infty$
    - ***for*** $i := 1$ ***to*** b ***do***
    - ***begin***
    - $t := -F0(p_i, depth - 1)$ ***// recursive call, the returned value is negated***
    - ***if*** $t > m$ ***then*** $m := t$ ***// always find a max value***
    - ***end***
  - **end**
  - **return** m

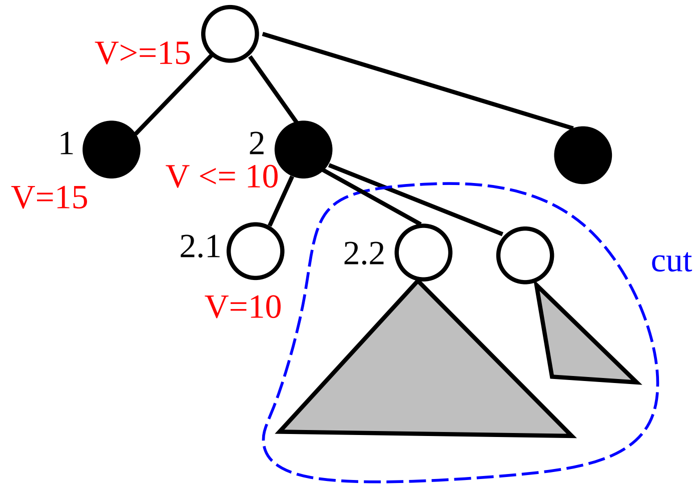
Nega-max: comments

- **Another brute-force method to try all possibilities.**
  - **Use $h(p)$ instead of $f(p)$.**
    - ▷ ***Zero-sum game: if one player thinks a position p has a value of w, then the other player thinks it is $-w$.***
  - **De Morgan's laws**
    - ▷ $\min\{x, y, z\} = -max\{-x, -y, -z\}$.
    - ▷ $\max\{x, y, z\} = -min\{-x, -y, -z\}$.
  - **Watch out the code in dealing with search termination conditions.**
    - ▷ ***Leaf.***
    - ▷ ***Reach a given searching depth.***
    - ▷ ***Timing control.***
    - ▷ ***Other constraints such as the score is good or bad enough.***
- **Notations:**
  - F' **means the Mini-max version.**
    - ▷ ***Need a G' companion.***
    - ▷ ***Easy to explain.***
  - F **means the Nega-max version.**
    - ▷ ***Simpler code.***
    - ▷ ***May be difficult to explain.***

Intuition for improvements

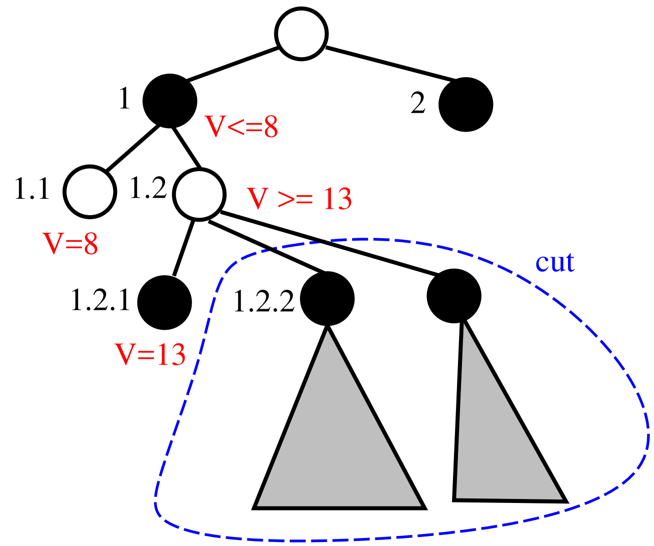
- **Branch-and-bound: using information you have so far to cut or prune branches.**
  - **A branch is cut means we do not need to search it anymore.**
  - **If you know for sure or almost sure the value of your result is more than x and the current search result for this branch so far can give you no more than x,**
    - ▷ ***then there is no/almost no need to search this branch any further.***
- **Two types of approaches**
  - **Exact algorithms: through mathematical proof, it is guaranteed that the branches pruned won't contain the solution.**
    - ▷ ***Alpha-beta pruning: reinvented by several researchers in the 1950's and 1960's.***
    - ▷ ***Scout.***
    - ▷ $\cdots$
  - **Approximated heuristics: with a high probability that the solution won't be contained in the branches pruned.**
    - ▷ ***Obtain a good estimation on the remaining cost.***
    - ▷ ***Cut a branch when it is in a very bad position and there is little hope to gain back the advantage.***

Alpha cut-off



- **On the max node which is the root:**
  - ***Assume you have finished exploring the branch at 1 and obtained the best value from it as $bound$.***
  - ***You now search the branch at 2 by first searching the branch at 2.1.***
  - ***Assume branch at 2.1 returns a value that is $\leq bound$.***
  - ***Then no need to evaluate the branch at 2.2 and all later branches of 2, if any, at all.***
  - ***The best possible value for the branch at 2 must be $\leq bound$.***
  - ***Hence we should take value returned from the branch at 1 as the best possible solution.***

Beta cut-off



- **On the min node 1:**
  - ***Assume you have finished exploring the branch at 1.1 and obtained the best value from it as $bound$.***
  - ***You now search the branch at 1.2 by first exploring the branch at $1.2.1$.***
  - ***Assume the branch at $1.2.1$ returns a value that is $\geq bound$.***
  - ***Then no need to evaluate the branch at $1.2.2$ and all later branches of 1.2, if any, at all.***
  - ***The best possible value for the branch at 1.2 is $\geq bound$.***
  - ***Hence we should take value returned from the branch at 1.1 as the best possible solution.***

Alpha and Beta cut-off

- **Alpha cut-off for a min node u:**
  - **An elder brother w of u produces a lower bound V_l.**
  - **A branch (descendant) of u produces an upper bound V_u for u.**
  - **If $V_l \geq V_u$, then there is no need to evaluate all later branches (descendants) of u.**
- **Beta cut-off for a max node v:**
  - **An elder brother y produces an upper bound V_u.**
  - **A branch (descendant) of u produces a lower bound V_l for u.**
  - **If $V_l \geq V_u$, then there is no need to evaluate all later branches (descendant) of v.**

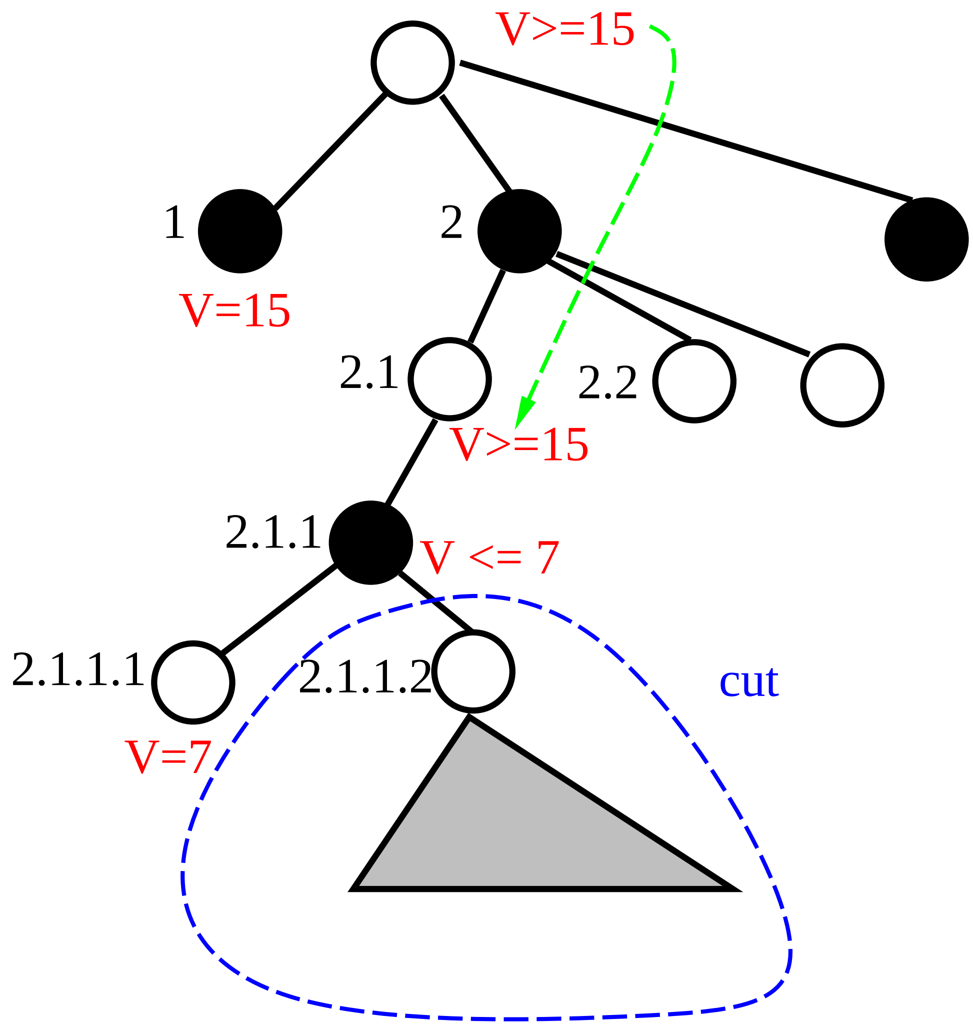
Degenerated case: direct alpha/beta cut-off

- Assume in the case of zero sum two-player games, the maximum value is max and the minimum value is $min = -max$.
- Direct alpha cut-off
  - A branch of a min node u produces an upper bound V_u for u.
  - If $V_u = -max$, then there is no need to evaluate all later branches of u.
  - Note when $V_u = -max$, then $V_l \geq V_u$ for all V_l since $-max$ is the minimum possible value.
- Direct beta cut-off
  - A branch of a max node v produces a lower bound V_l for v.
  - If $V_l = max$, then there is no need to evaluate all later branches of v.
  - Note when $V_l = max$, then $V_l \geq V_u$ for all V_u since max is the maximum possible value.
- Rationality: When one finds a way to win, stop thinking other alternatives.

Deep alpha/beta cut-off

- **For alpha cut-off:**
  - ▷ ***For a min node u, an elder brother w produces a lower bound V_l.***
  - ▷ ***A branch of u produces an upper bound V_u for u.***
  - ▷ ***If $V_l \geq V_u$, then there is no need to evaluate all later branches of u.***
- **Definition: For a node u in a tree and a positive integer g, Ancestor(g, u) is the ancestor of u by tracing the parent's link g times.**
- **Deep alpha cut-off:**
  - **When a lower bound V_l is produced at and propagated from u's great grand parent, i.e., Ancestor(3,u), or any Ancestor($2i+1$,u), $i \geq 1$.**
  - **When an upper bound V_u is returned from the a branch of u and $V_l \geq V_u$, then there is no need to evaluate all later branches of u.**
- **Deep beta cut-off:**
  - **When an upper bound V_u is produced at and propagated from u's great great grand parent, i.e., Ancestor(4,u), or any Ancestor($2i$,u), $i > 1$.**
  - **When a lower bound V_l is returned from the a branch of u and $V_l \geq V_u$, then there is no need to evaluate all later branches of u.**

Illustration — Deep alpha cut-off



Meanings of the two bounds

- **During searching, maintain two values $alpha$ and $beta$ for a node u so that**
  - $alpha$ **is the current lower bound of the possible returned value;**
    - ▷ ***This means you have known a way to achieve the value $alpha$ from searching a max node that is u or an ancestor of u.***
    - ▷ ***This will be a pre-condition set for every min node v that is a descendent of u.***
    - ▷ ***Node v lowers its $beta$ value after searching a child.***
    - ▷ ***When v's $beta$ is lower than u's $alpha$, we have an alpha cut.***
  - $beta$ **is the current upper bound of the possible returned value.**
    - ▷ ***This means your opponent have known a way to to achieve the value $beta$ from searching a min node that is u or an ancestor of u.***
    - ▷ ***This will be a pre-condition set for every max node v that is a descendent of u.***
    - ▷ ***Node v hightens its $alpha$ value after searching a child.***
    - ▷ ***When v's $alpha$ is higher than u's $beta$, we have a beta cut.***
- **Q: Does it help at all to record how "bad" this pre-condition is violated?**

Ideas for refinements

- **If $alpha = beta = val$, then we have found the solution which is val.**
- **If during searching, we know for sure $alpha > beta$, then there is no need to search any more in this branch.**
  - **The returned value cannot be in this branch.**
  - **Backtrack until it is the case $alpha < beta$.**
- **The two values $alpha$ and $beta$ are called the ranges of the current search window.**
  - **These values are dynamic.**
  - **Initially, $alpha$ is $-\infty$ and $beta$ is ∞.**

Alpha-beta pruning: Mini-Max (1/2)

- **Algorithm** $F1'$**(position** p**, value** $alpha$**, value** $beta$**, integer** $depth$**)**
  - **// max node**
  - **determine the successor positions** $p_1, \ldots, p_b$
  - **if** $b = 0$ **// a terminal node**
    or $depth = 0$ **// remaining depth to search**
    or time is running up // from timing control
    or some other constraints are met // add knowledge here
  - **then return** $f(p)$ **else**
    - ▷ $m := alpha$
    - ▷ ***for*** $i := 1$ ***to*** b ***do***
    - ▷ $t := G1'(p_i, m, beta, depth - 1)$
    - ▷ ***if*** $t > m$ ***then*** $m := t$ ***// improve the current best value***
    - ▷ ***if*** m ***is max or*** $m \geq beta$ ***then return***$(beta)$ ***// beta cut off***
  - **end;**
  - **return** m
- **"m is max" refers to m is the maximum possible value, which triggers a direct beta cut-off.**

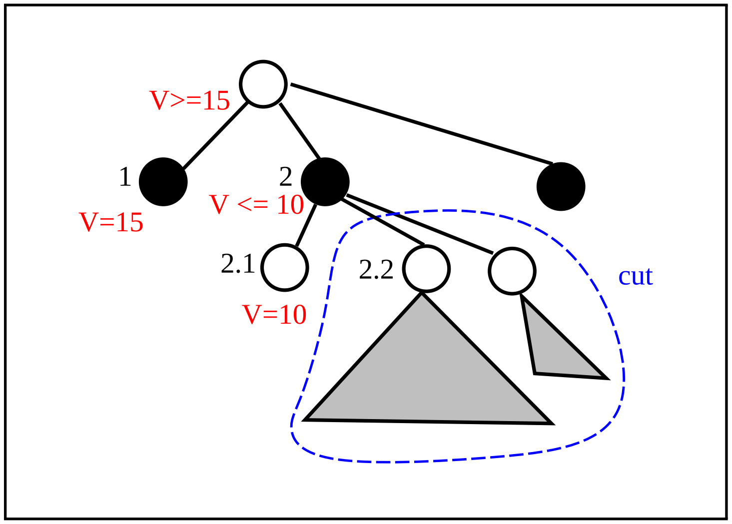
Alpha-beta pruning: Mini-Max (2/2)

- **Algorithm** $G1'$**(position** p**, value** $alpha$**, value** $beta$**, integer** $depth$**)**
  - **// min node**
  - **determine the successor positions** $p_1, \ldots, p_b$
  - **if** $b = 0$ **// a terminal node**
    or $depth = 0$ **// remaining depth to search**
    or time is running up // from timing control
    or some other constraints are met // add knowledge here
  - **then return** $f(p)$ **else**
    - ▷ $m := beta$
    - ▷ ***for*** $i := 1$ ***to*** b ***do***
    - ▷ $\quad t := F1'(p_i, alpha, m, depth - 1)$
    - ▷ $\quad$ ***if*** $t < m$ ***then*** $m := t$ ***// improve the current best value***
    - ▷ $\quad$ ***if*** m ***is min or*** $m \leq alpha$ ***then return(***$alpha$***) // alpha cut off***
  - **end;**
  - **return** m
- **"**m **is min" refers to** m **is the minimum possible value, which triggers a direct alpha cut-off.**

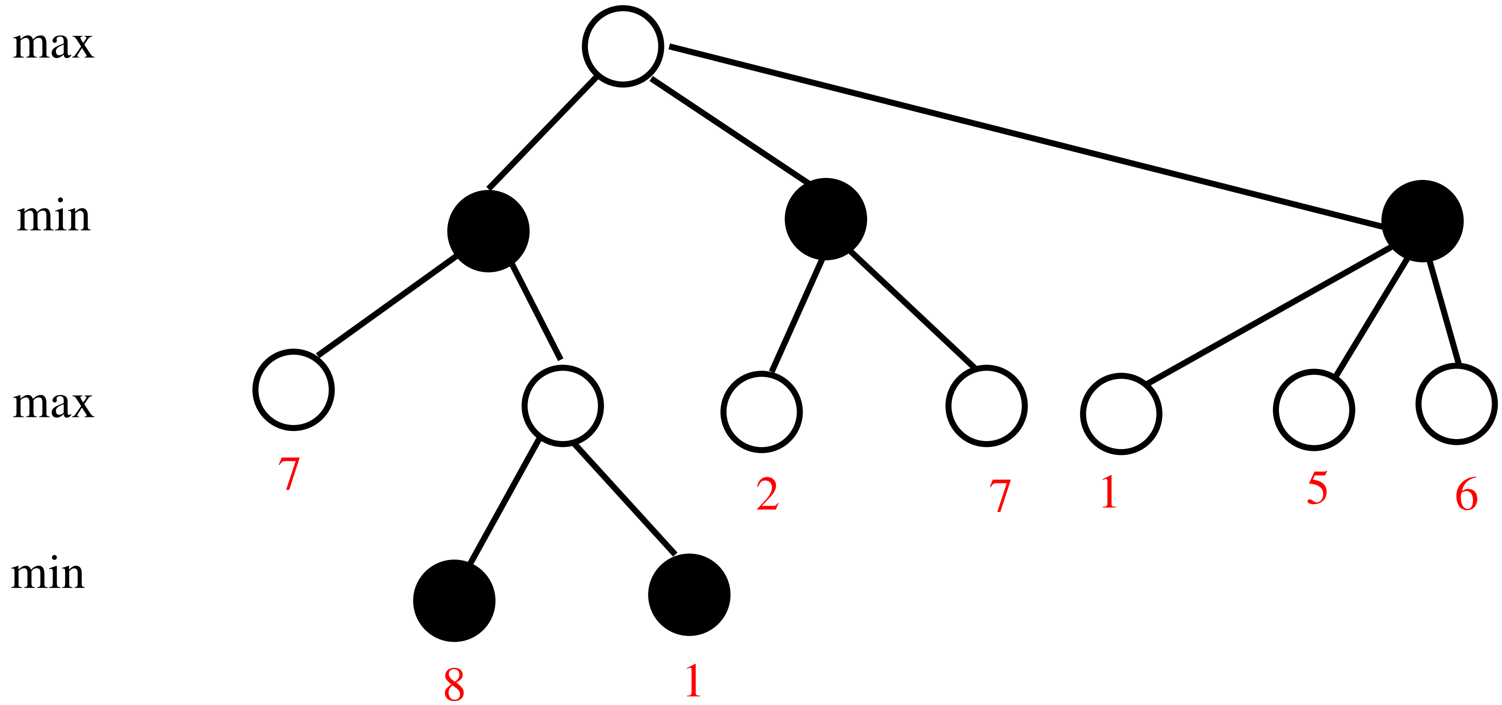
Example

Initial call: $F1'$(root,$-\infty$,∞,$depth$)

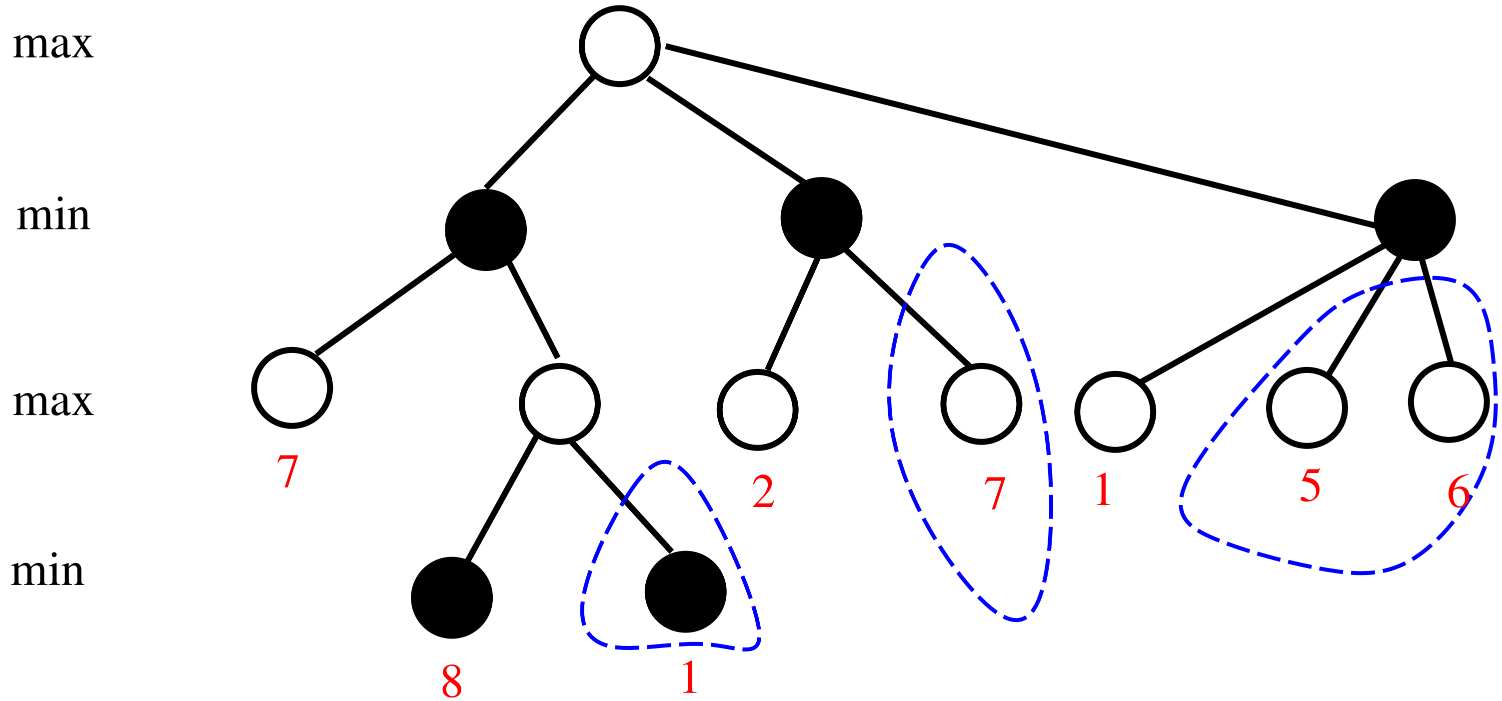
- $m = -\infty$
- **call** $G1'$**(node 1,**$-\infty$**,**∞**,**$depth - 1$**)**
  - ▷ ***it is a terminal node***
  - ▷ ***return value*** 15
- $t = 15$**;**
  - ▷ ***since*** $t > m$***,*** m ***is now*** 15
- **call** $G1'$**(node 2,**15**,**∞**,**$depth - 1$**)**
  - ▷ ***call*** $F1'$***(node 2.1,***15***,***∞***,***$depth - 2$***)***
  - ▷ ***it is a terminal node; return*** 10
  - ▷ $t = 10$***; since*** $t < \infty$***,*** m ***is now*** 10
  - ▷ $alpha$ ***is*** 15***,*** m ***is*** 10***, so we have an alpha cut off,***
  - ▷ ***no need to call*** $F1'$***(node 2.2,15,10,***$depth - 2$***)***
  - ▷ ***return 15***
  - ▷ $\cdots$



A complete example



A complete example

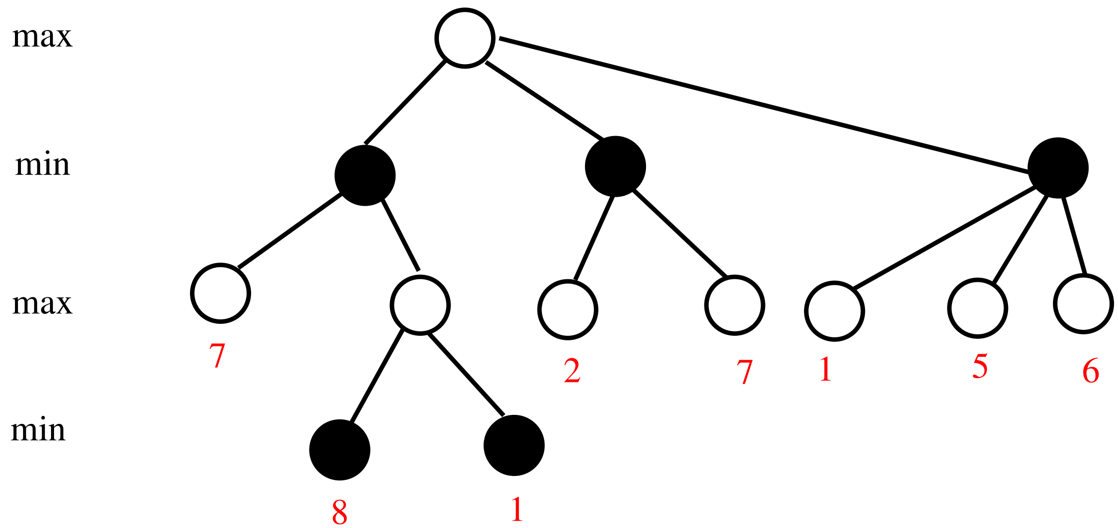


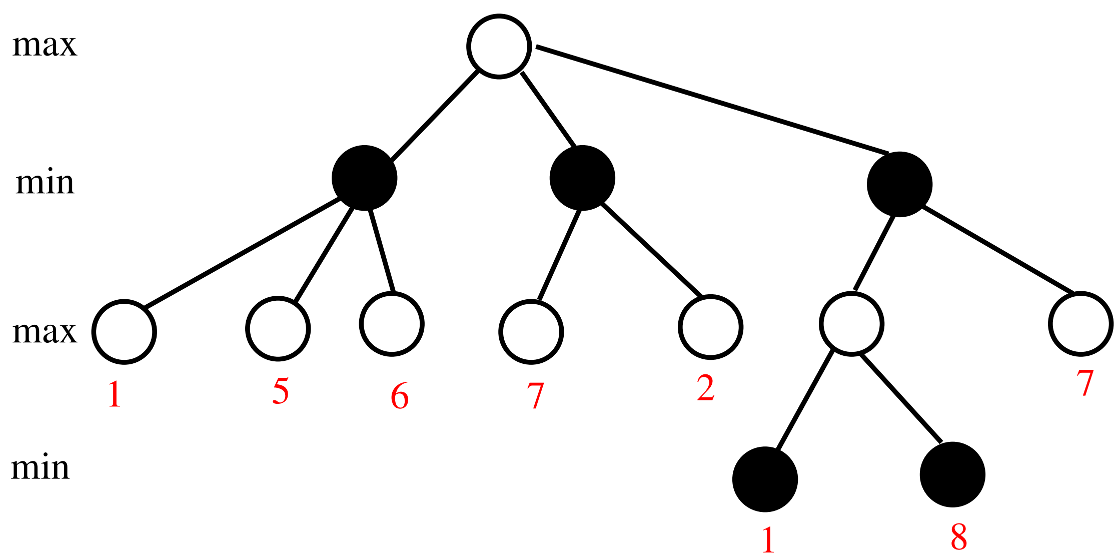
- **The solution is the same with or without the cuts as circled by dashed lines.**

Alpha-beta pruning algorithm: Nega-max

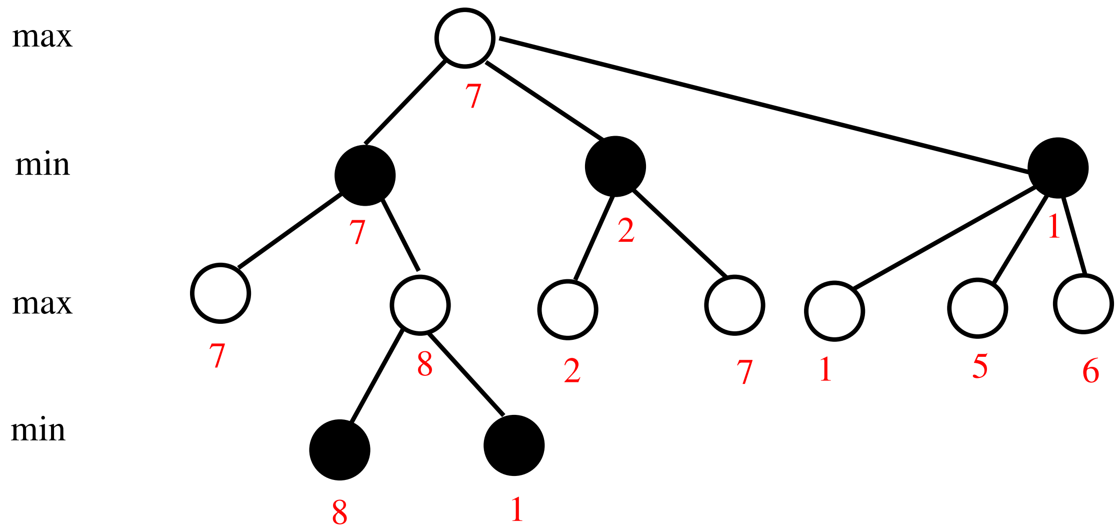
- **Algorithm $F1$(position p, value $alpha$, value $beta$, integer $depth$)**
  - determine the successor positions $p_1, \ldots, p_b$
  - if $b = 0$ // a terminal node
    or $depth = 0$ // remaining depth to search
    or time is running up // from timing control
    or some other constraints are met // add knowledge here
  - then return $h(p)$ else
  - begin
    - ▷ $m := alpha$
    - ▷ ***for*** $i := 1$ ***to*** b ***do***
    - ▷ ***begin***
    - ▷ $t := -F1(p_i, -beta, -m, depth - 1)$
    - ▷ ***if*** $t > m$ ***then*** $m := t$ // ***improve the current best value***
    - ▷ ***if*** m ***is max or*** $m \geq beta$ ***then return***$(beta)$ // ***cut off***
    - ▷ ***end***
  - end
  - return m
- **Comment: Watch out the changes of the bounds in the recursive call.**

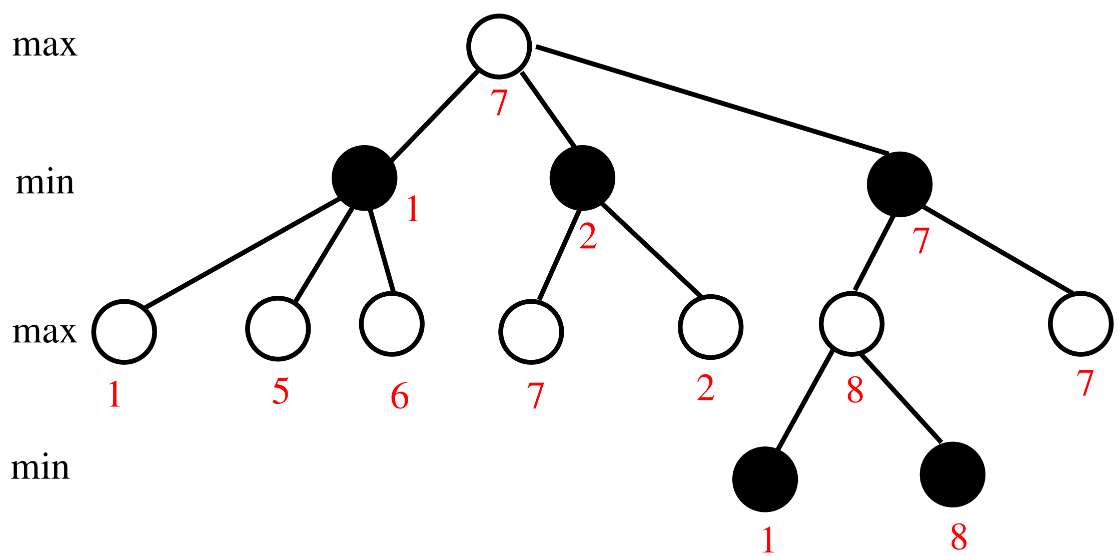
Examples (1/4)



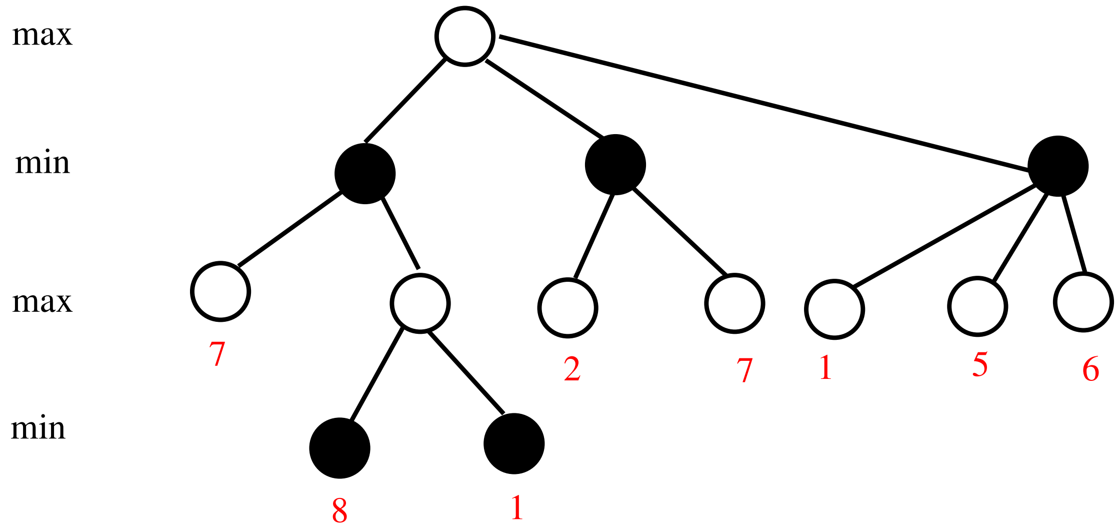


Examples (2/4)

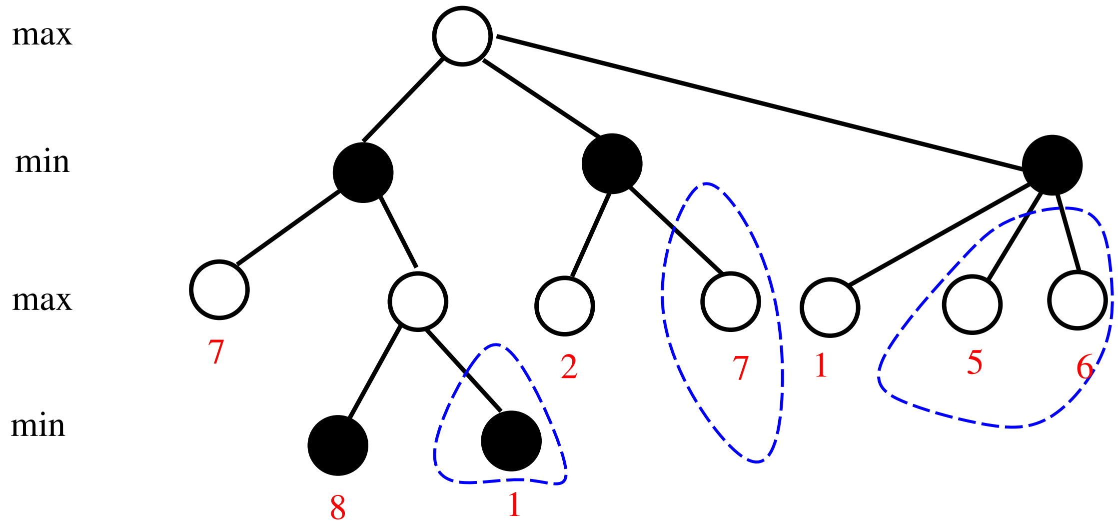

max
min
max
min
7
7
2
1
7
8
2
7
1
5
6
8
1


max
min
max
min
7
1
2
7
1
5
6
7
2
8
7
1
8

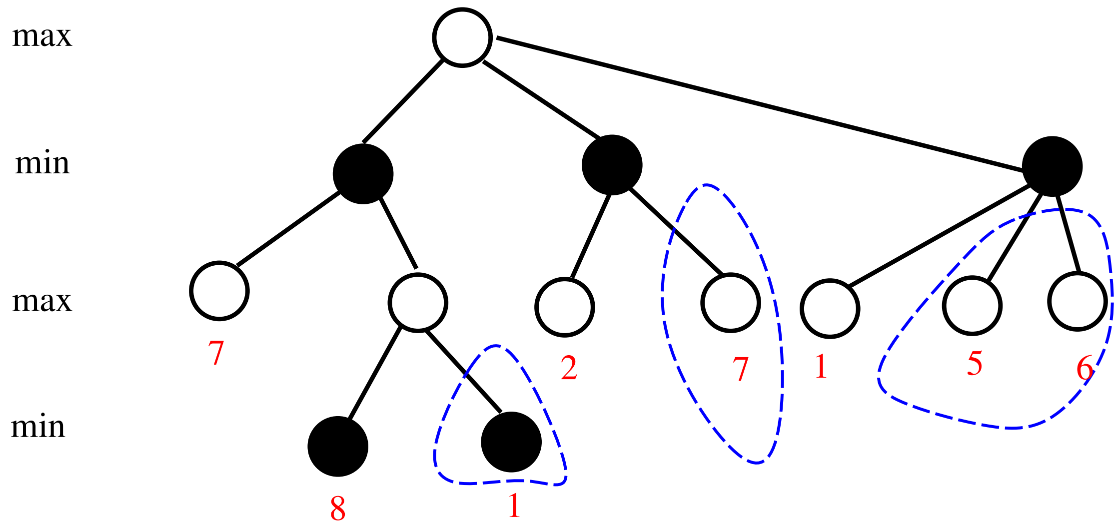
Examples (3/4)

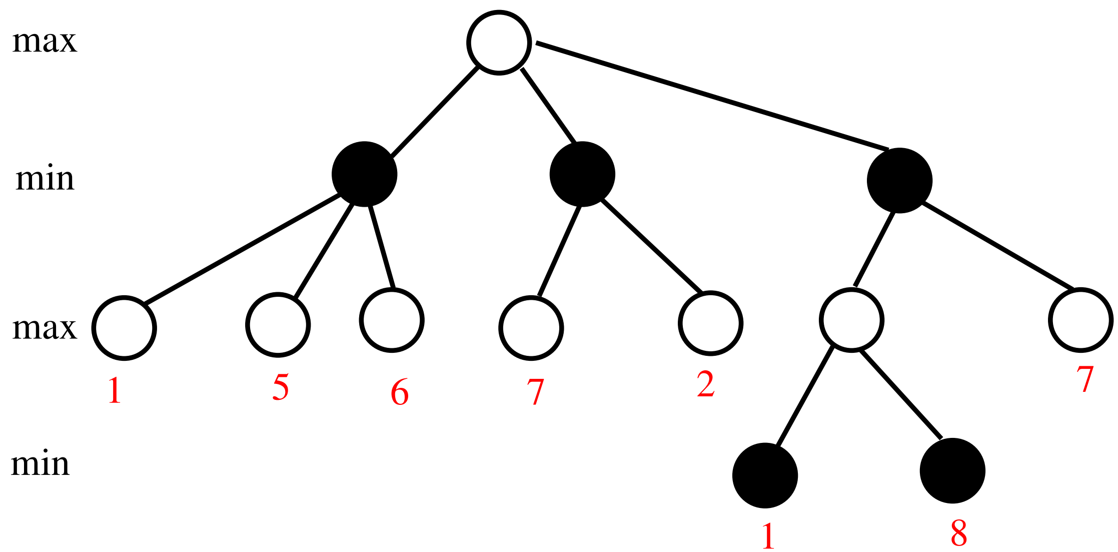


Examples (3/4)



Examples (4/4)





What happened in the last examples

- **Assume we run $F1'$ and $G1'$ in the order of from left to right in a game tree.**
- **The tree on the top and the tree on the bottom are the same game tree with different search orderings.**
  - **A tree has a fixed searched value no matter what search orderings used.**
- **We can prune 4 nodes in the tree on the top, but cannot prune any node in the tree on the bottom.**

Lessons from the previous examples

- It looks like for the same tree, different move orderings give very different cut branches.
- It looks like if a node can evaluate a child with the best possible outcome earlier, then it has a chance to cut earlier.
  - For a min node, this means to search the child branch that gives the lowest value first.
  - For a max node, this means to search the child branch that gives the highest value first.
- Comments:
  - Watch out the returned value v for a node p when alpha or beta cut-off happens.
    - ▷ *It is a bound for p, not its best possible value.*
  - It is impossible to always know which the best branch is; otherwise we need to always do a brute-force exhaustive search.
- Q: In the best case scenario, how many nodes can be cut?

Analysis of a possible best case

- **Definitions:**
  - **A path in a search tree is a sequence of numbers indicating the branches selected in each level using the Dewey decimal system.**
  - **A position is denoted as a path $a_1.a_2.\cdots.a_\ell$ from the root.**
  - **A position $a_1.a_2.\cdots.a_\ell$ is critical if**
    - ▷ *$a_i = 1$ for all even values of i or*
    - ▷ *$a_i = 1$ for all odd values of i or*
    - ▷ *it is the root.*
  - **Note: as a special case, the root is critical.**
  - **Examples:**
    - ▷ *2.1.4.1.2, 1.3.1.5.1.2, 1.1.1.2.1.1.1.3 and 1.1 are critical*
    - ▷ *1.2.1.1.2 is not critical*
  - **The number of 1's in a path has little to do with whether it is critical or not.**
    - ▷ *A critical node has at least $\lfloor \ell/2 \rfloor$ 1's, but the reverse is not true.*
- **Q: Why does the root need to be critical?**

Perfect-ordering tree

- **A perfect-ordering tree:**

$$F(a_1.\cdots.a_\ell) = \begin{cases} h(a_1.\cdots.a_\ell) & \textbf{if } a_1.\cdots.a_\ell \textbf{ is a terminal} \\ -F(a_1.\cdots.a_\ell.1) & \textbf{otherwise} \end{cases}$$

  - **The first successor of every non-terminal position gives the best possible value.**

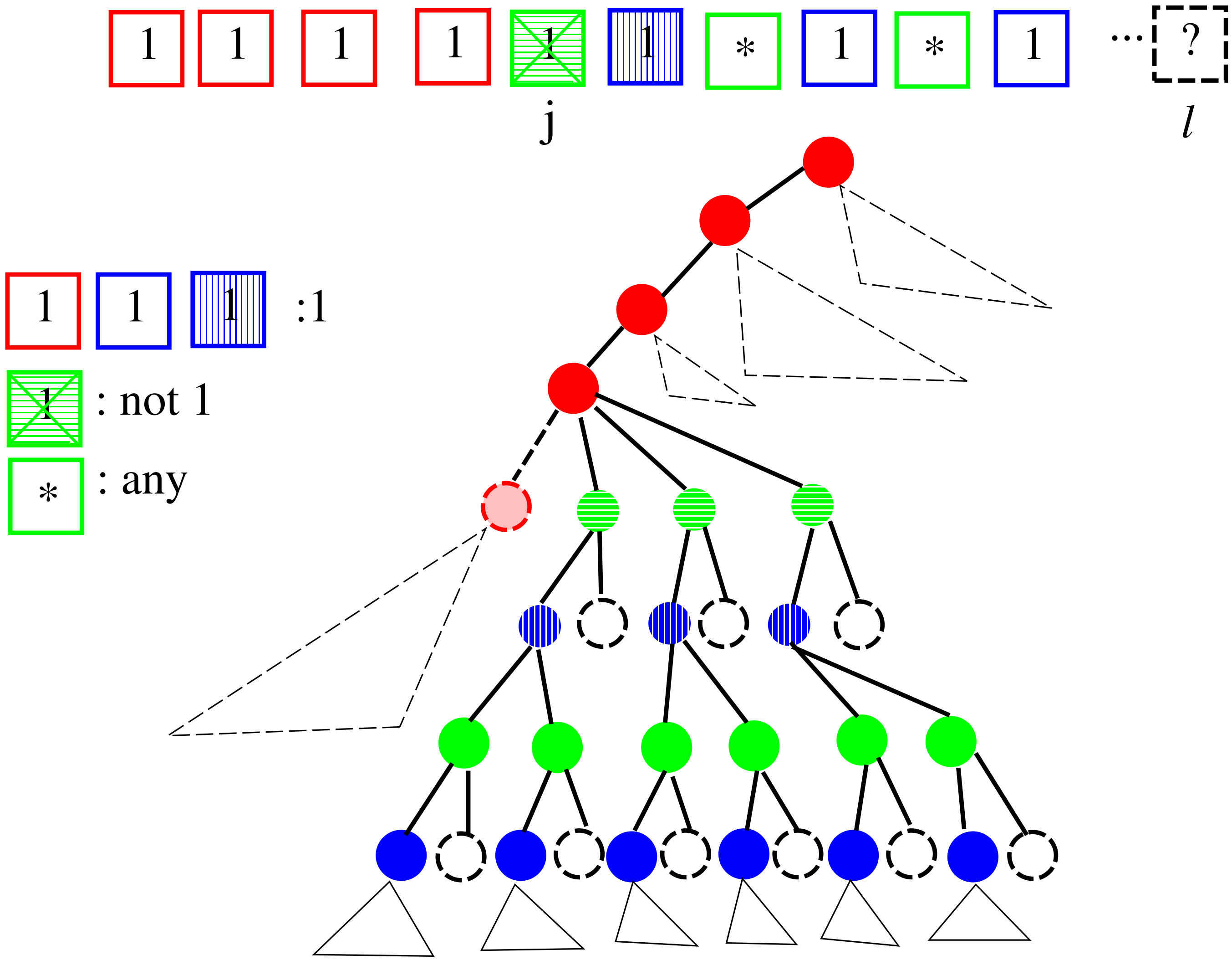
Theorem 1

- **Theorem 1: $F1$ examines precisely the critical positions of a perfect-ordering tree.**
- **Proof sketch:**
  - **Classify the critical positions, a.k.a. nodes, into different types.**
    - ▷ ***You must evaluate the first branch from the root to the bottom.***
    - ▷ ***Alpha cut off happens at odd-depth nodes as soon as the first branch of this node is evaluated.***
    - ▷ ***Beta cut off happens at even-depth nodes as soon as the first branch of this node is evaluated.***
  - **For nodes of the same type, find common characteristics causing or not causing prunings to happen.**

Types of nodes

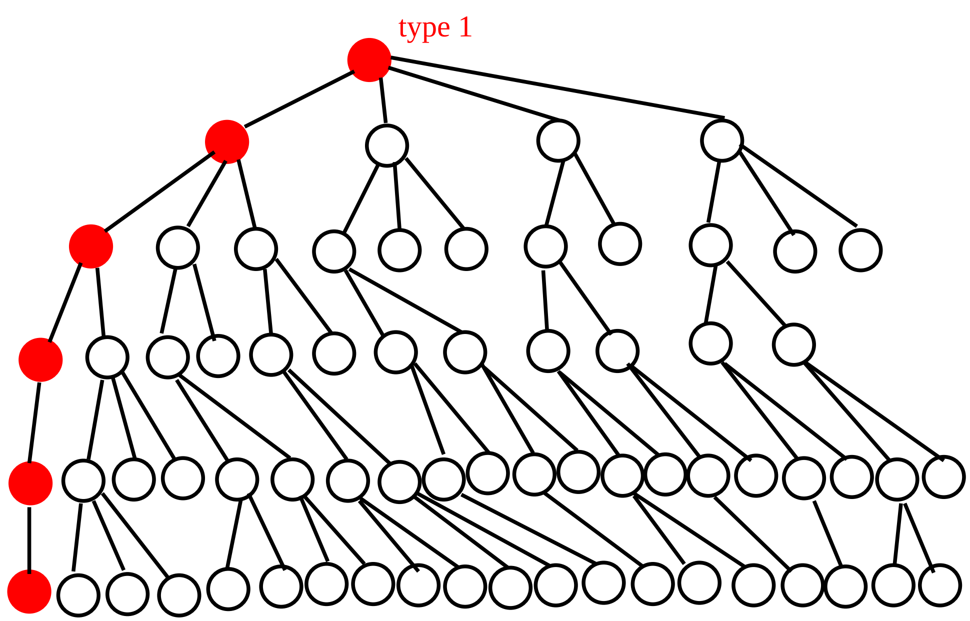
- Classification of critical positions $a_1.a_2.\cdots.a_j.\cdots.a_\ell$ where j is the least index, if exists, such that $a_j \neq 1$ and ℓ is the last index.
  - j is the anchor in the analysis.
  - Definition: let $IS1(a_i)$ be a boolean function so that it is 0 if it is not the value 1 and it is 1 if it is.
    - ▷ *We call this $IS1$ parity of a number.*
  - If j exists and $\ell > j$, then
    - ▷ *$a_{j+1} = 1$ because this position is critical and thus the $IS1$ parities of a_j and a_{j+1} are different.*
  - Since this position is critical, if $a_j \neq 1$, then $a_h = 1$ for any h such that $h - j$ is odd.
    - ▷ *a_{j+1} must be 1.*
- We now classify critical nodes into three types.
  - Nodes of the same type share some common properties.

Illustration — critical nodes



Type 1 nodes

- type 1: the root, or a node with all the a_i are 1;
  - This means the anchor j does not exist.
  - Nodes on the leftmost branch.
  - The leftmost child of a type 1 node except the root.
- In a DFS-like searching, type 1 nodes are examined first.



Type 2 nodes

- **Classification of critical positions $a_1.a_2.\cdots.a_j.\cdots.a_\ell$ where j is the least index such that $a_j \neq 1$ and ℓ is the last index.**
- **The anchor j exists.**
- **Type 2: $\ell - j$ is zero or even;**
  - **type 2.1: $\ell - j = 0$ which means $\ell = j$.**
    - ▷ ***It is in the form of $\underline{1.1.1.\cdots.1.1.1}.a_\ell$ and $a_\ell \neq 1$.***
    - ▷ ***The non-leftmost children of a type 1 node.***
  - **type 2.2: $\ell - j > 0$ and is even.**
    - ▷ ***It is in the form of $\underline{1.1.\cdots.1.1.a_j.1.a_{j+2}.\cdots.a_{\ell-2}.1}.a_\ell$.***
    - ▷ ***Note, we will define $1.1.\cdots.1.1.a_j.1.a_{j+2}.\cdots.a_{\ell-2}.1$ to be a type 3 node. This means all of the children of a type 3 node.***
- **Q:**
  - **Can a_ℓ be 1 or non-1 for a type 2 node?**
  - **Can a_ℓ be 1 or non-1 for a type 2.1 node?**
  - **Can a_ℓ be 1 or non-1 for a type 2.2 node?**

Type 3 nodes

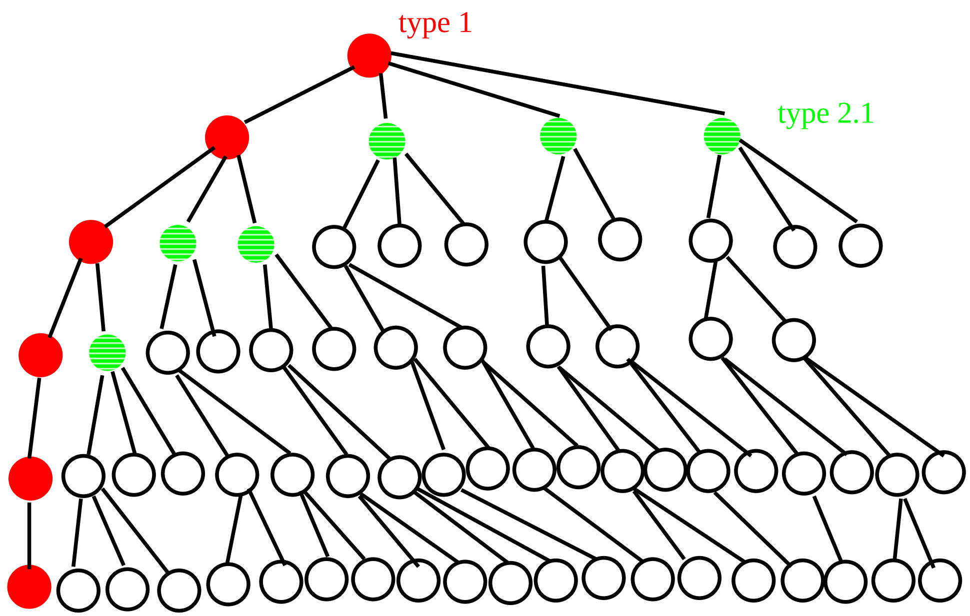
- **Classification of critical positions $a_1.a_2.\cdots.a_j.\cdots.a_\ell$ where j is the least index such that $a_j \neq 1$ and ℓ is the last index.**
- **The anchor j exists.**
- **Type 3: $\ell - j$ is odd;**
  - $a_j \neq 1$ **and** $\ell - j$ **is odd**
    - ▷ ***Since this position is critical, the*** $IS1$ ***parities of*** a_j ***and*** a_ℓ ***are different.***
      $\Longrightarrow a_\ell = 1$
      $\Longrightarrow a_{j+1} = 1$
  - **It is in the form of**
    - ▷ $1.1.\cdots.1.a_j.1.a_{j+2}.1.\cdots.1.a_{\ell-1}.1.$
  - **The leftmost child of a type 2 node.**
  - **type 3.1:** $\ell - j = 1$.
    - ▷ ***It is of the form*** $\underline{1.1.\cdots.1.a_j}.1$
    - ▷ ***The leftmost child of a type 2.1 node.***
  - **type 3.2:** $\ell - j > 1$.
    - ▷ ***It is of the form*** $\underline{1.1.\cdots.1.a_j.1.a_{j+2}.1.\cdots.1.a_{\ell-1}}.1$
    - ▷ ***The leftmost child of a type 2.2 node.***
- **Q: Can a_ℓ be 1 or non-1 for a type 3 node?**

Comments

- **Nodes of the same type have common properties.**
- **These properties can be used in solving other problems.**
  - **Example: Efficient parallelization of alpha-beta based searching algorithms.**
- **Main techniques used:**
  - **For each non-1 number, any number appeared later and is odd distance away must be 1.**
    - ▷ ***You cannot have two consecutive non-1 numbers in the ID of a critical node.***

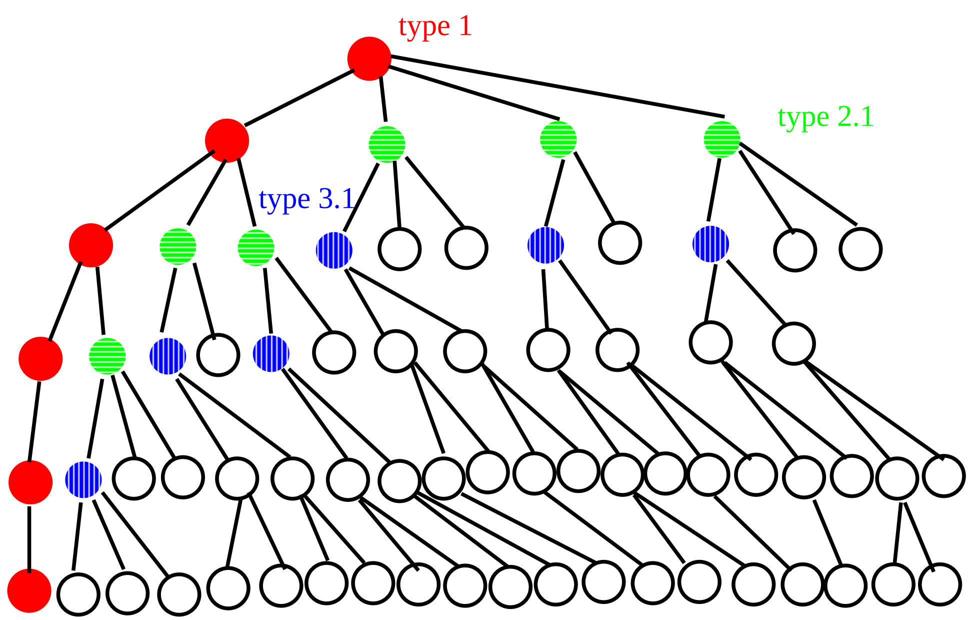
Type 2.1 nodes

- **Classification of critical positions $a_1.a_2.\cdots.a_j.\cdots.a_\ell$ where j is the least index such that $a_j \neq 1$ and ℓ is the last index.**
- **type 2: $\ell - j$ is zero or even;**
  - **type 2.1: $\ell - j = 0$.**
    - ▷ ***Then $\ell = j$.***
    - ▷ ***It is of the form of $\underline{1.1.1.\cdots.1.1.1}.a_\ell$ and $a_\ell \neq 1$.***
    - ▷ ***The non-leftmost children of a type 1 node.***



Type 3.1 nodes

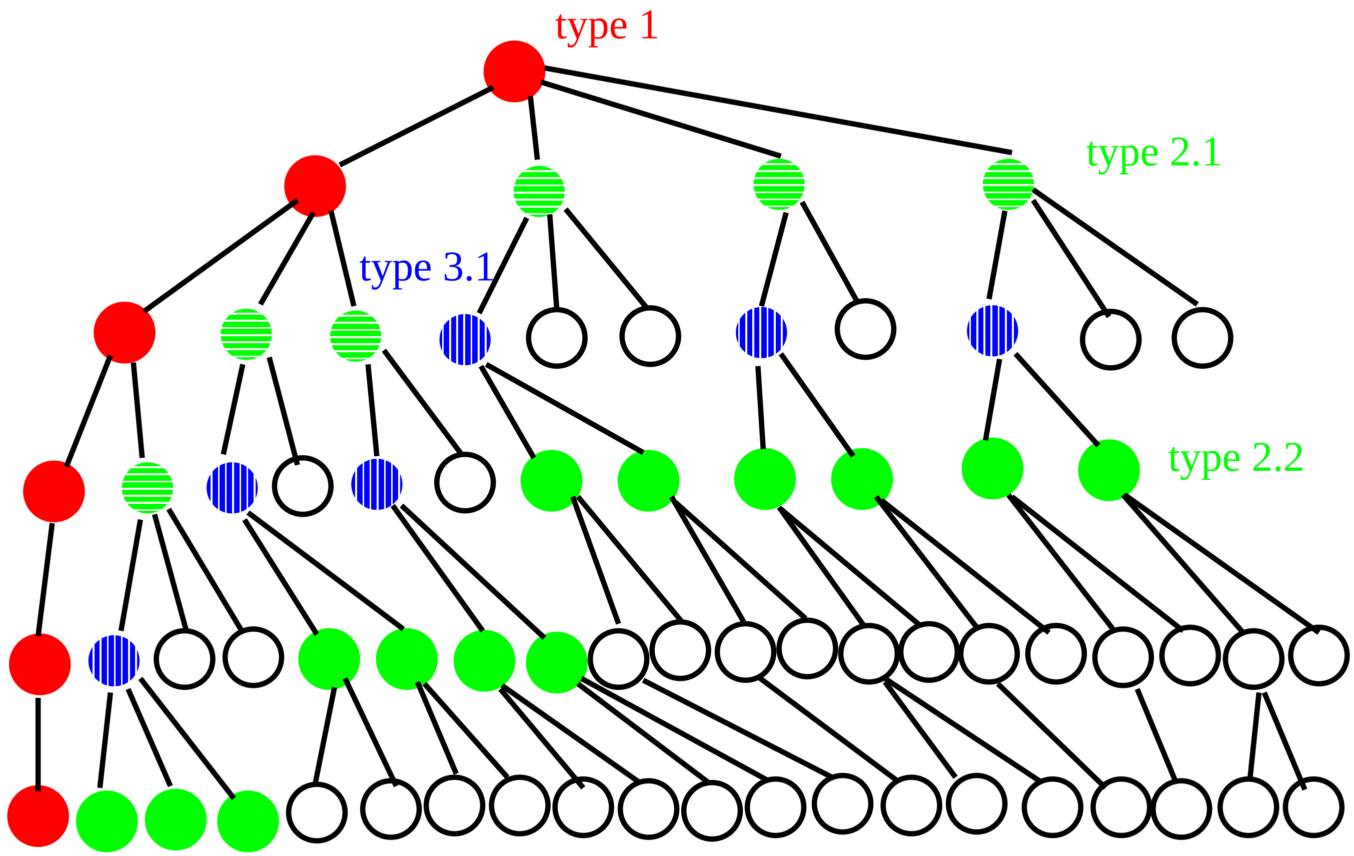
- Classification of critical positions $a_1.a_2.\cdots.a_j.\cdots.a_\ell$ where j is the least index such that $a_j \neq 1$ and ℓ is the last index.
- type 3: $\ell - j$ is odd;
  - type 3.1: $\ell - j = 1$.
    - ▷ *Then $\ell = j + 1$.*
    - ▷ *It is of the form $\underline{1.1.\cdots.1.a_j}.1$ and $a_\ell \neq 1$.*
    - ▷ *The leftmost child of a type 2.1 node.*



Type 2.2 nodes

- **Classification of critical positions $a_1.a_2.\cdots.a_j.\cdots.a_\ell$ where j is the least index such that $a_j \neq 1$ and ℓ is the last index.**
- **type 2: $\ell - j$ is zero or even;**
  - **type 2.2: $\ell - j > 0$ and is even.**
    - ▷ ***The $IS1$ parties of a_j and a_{j+1} are different.***
      $\Longrightarrow$ ***Since $a_j \neq 1$, $a_{j+1} = 1$.***
    - ▷ ***$(\ell - 1) - j$ is odd:***
      $\Longrightarrow$ ***The $IS1$ parties of $a_{\ell-1}$ and a_j are different.***
      $\Longrightarrow$ ***Since $a_j \neq 1$, $a_{\ell-1} = 1$.***
    - ▷ ***It is in the form of $\underline{1.1.\cdots.1.1.a_j.1.a_{j+2}.\cdots.a_{\ell-2}.1}.a_\ell$.***
    - ▷ ***Note, we will show $1.1.\cdots.1.1.a_j.1.a_{j+2}.\cdots.a_{\ell-2}.1$ is a type 3 node later.***
    - ▷ ***All of the children of a type 3 node.***

Illustration: Type 2.2 nodes



Type 3.2 nodes

- **Classification of critical positions $a_1.a_2.\cdots.a_j.\cdots.a_\ell$ where j is the least index such that $a_j \neq 1$ and ℓ is the last index.**
- **type 3: $\ell - j$ is odd;**
  - **type 3.2: $\ell - j > 1$.**
    - ▷ ***It is of the form*** $\underline{1.1.\cdots.1.a_j.1.a_{j+2}.1.\cdots.1.a_{\ell-1}}.1$
    - ▷ ***The leftmost child of a type 2.2 node.***

Illustration: Type 3.2 nodes

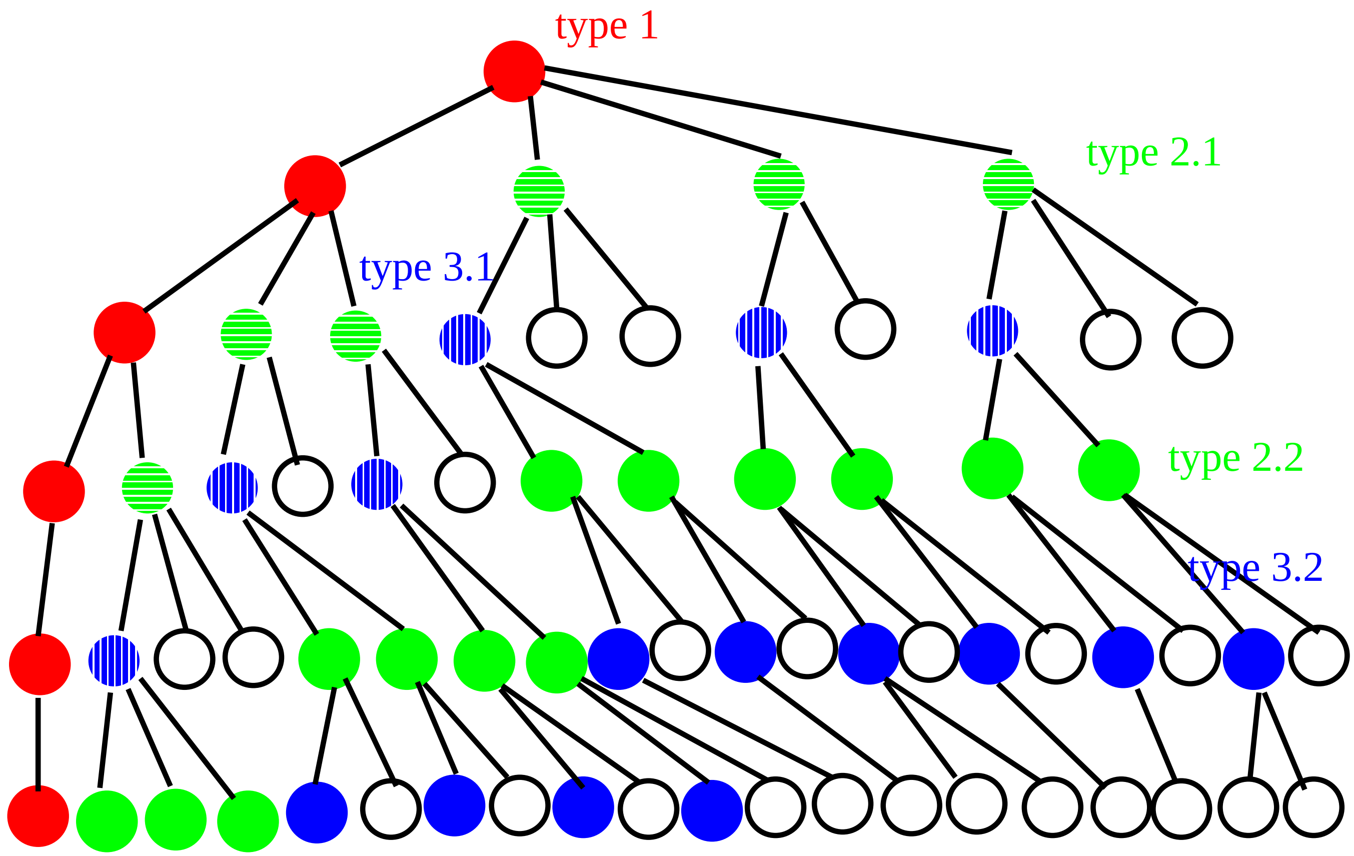


Illustration of all nodes

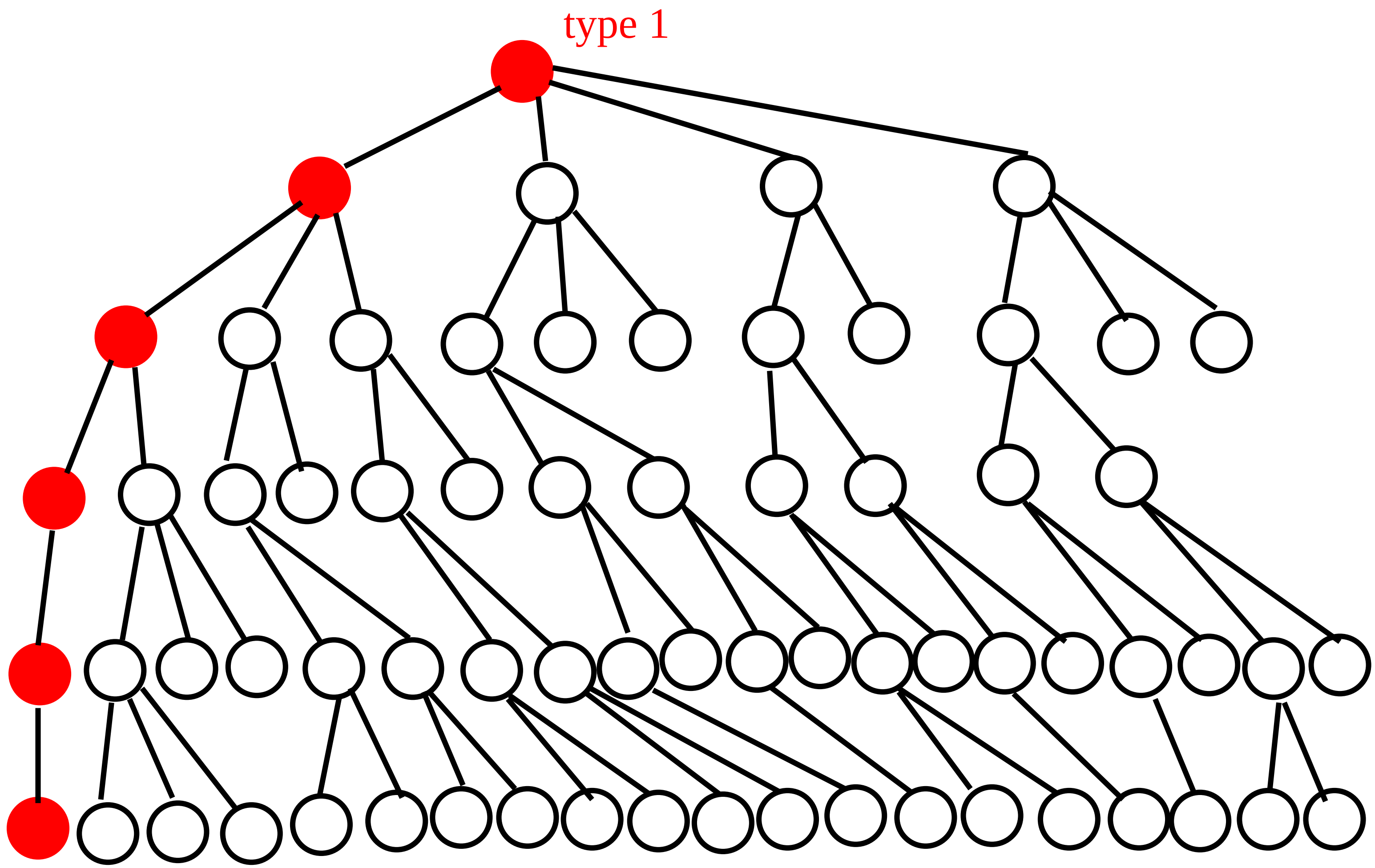


Illustration of all nodes

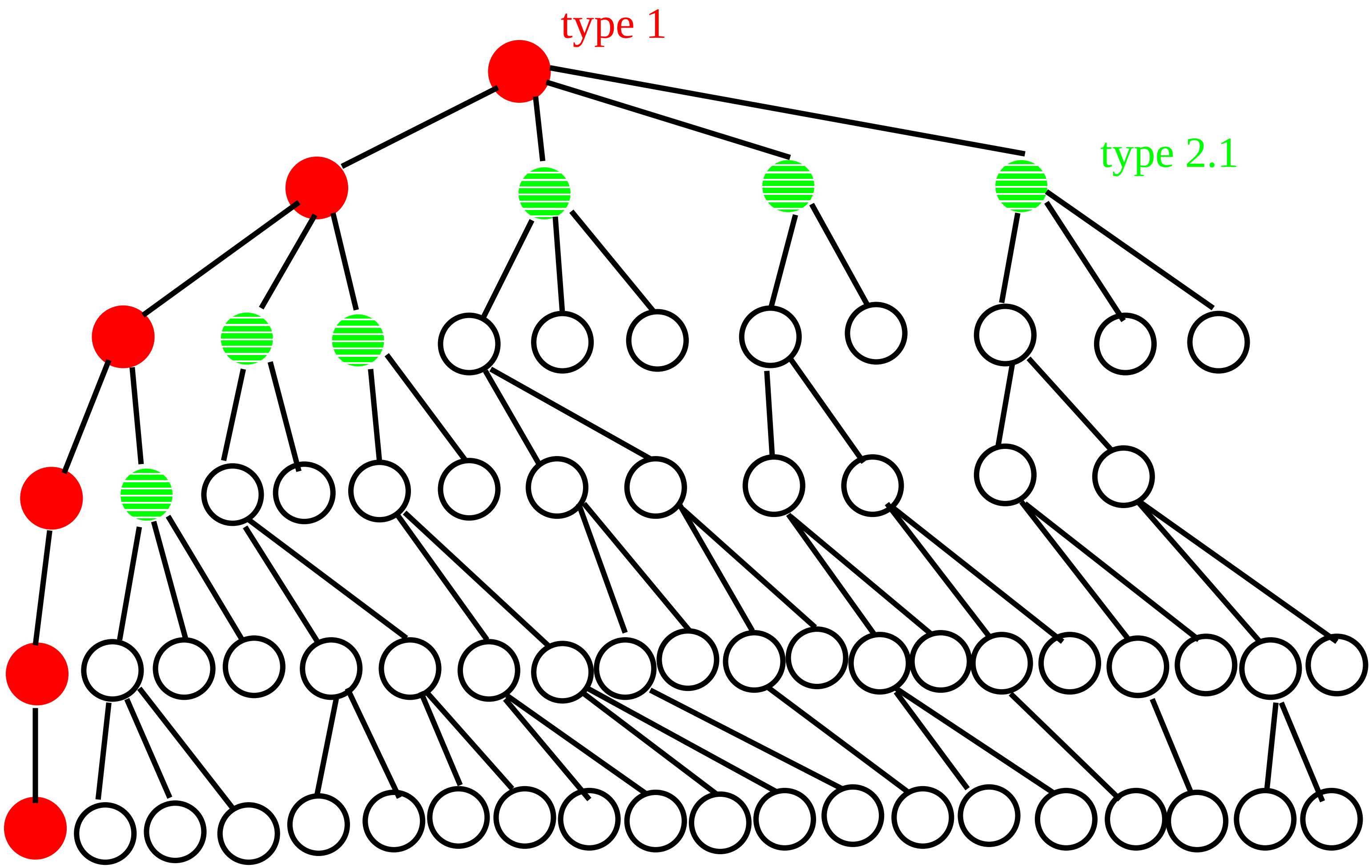


Illustration of all nodes

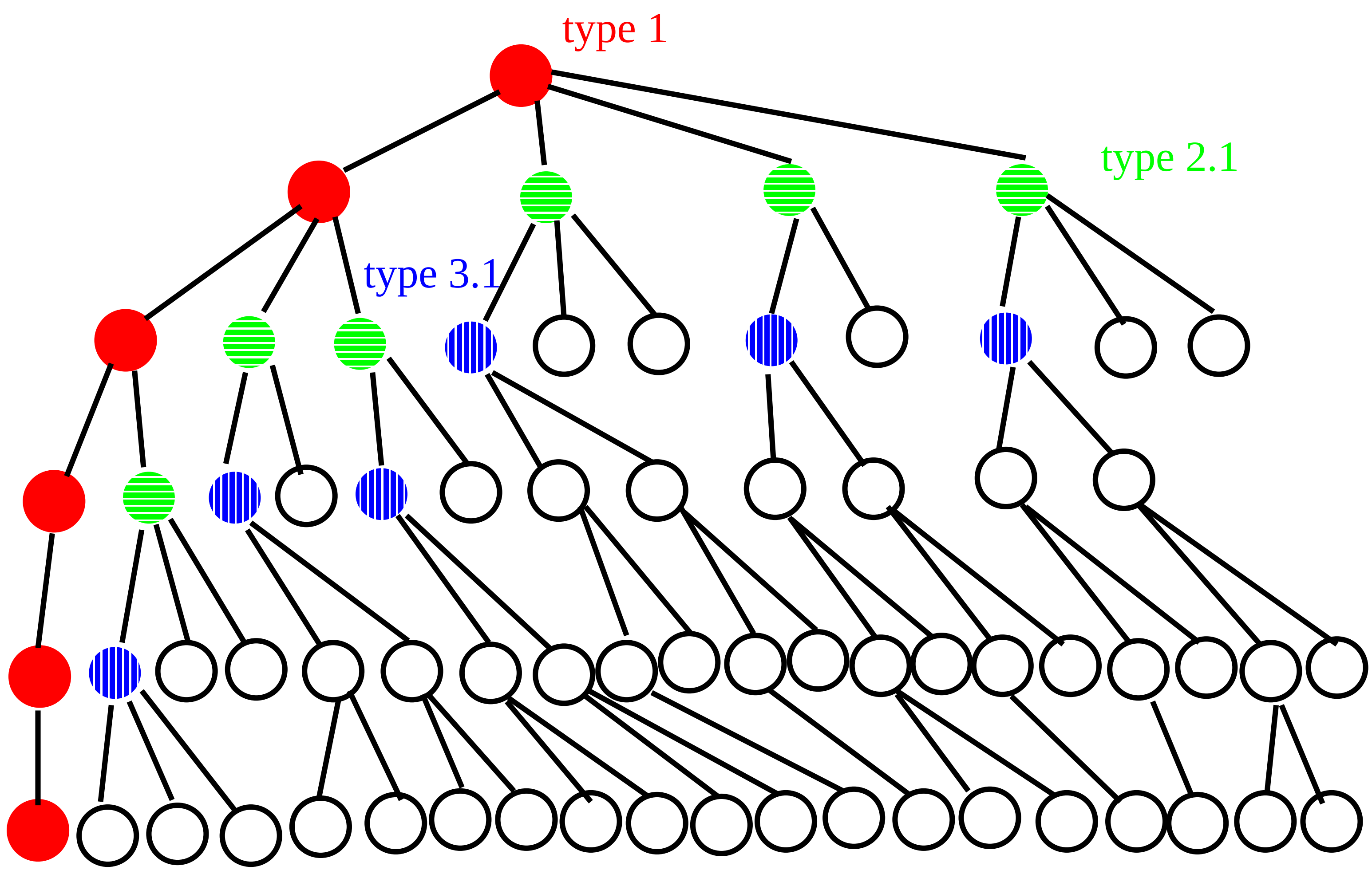


Illustration of all nodes

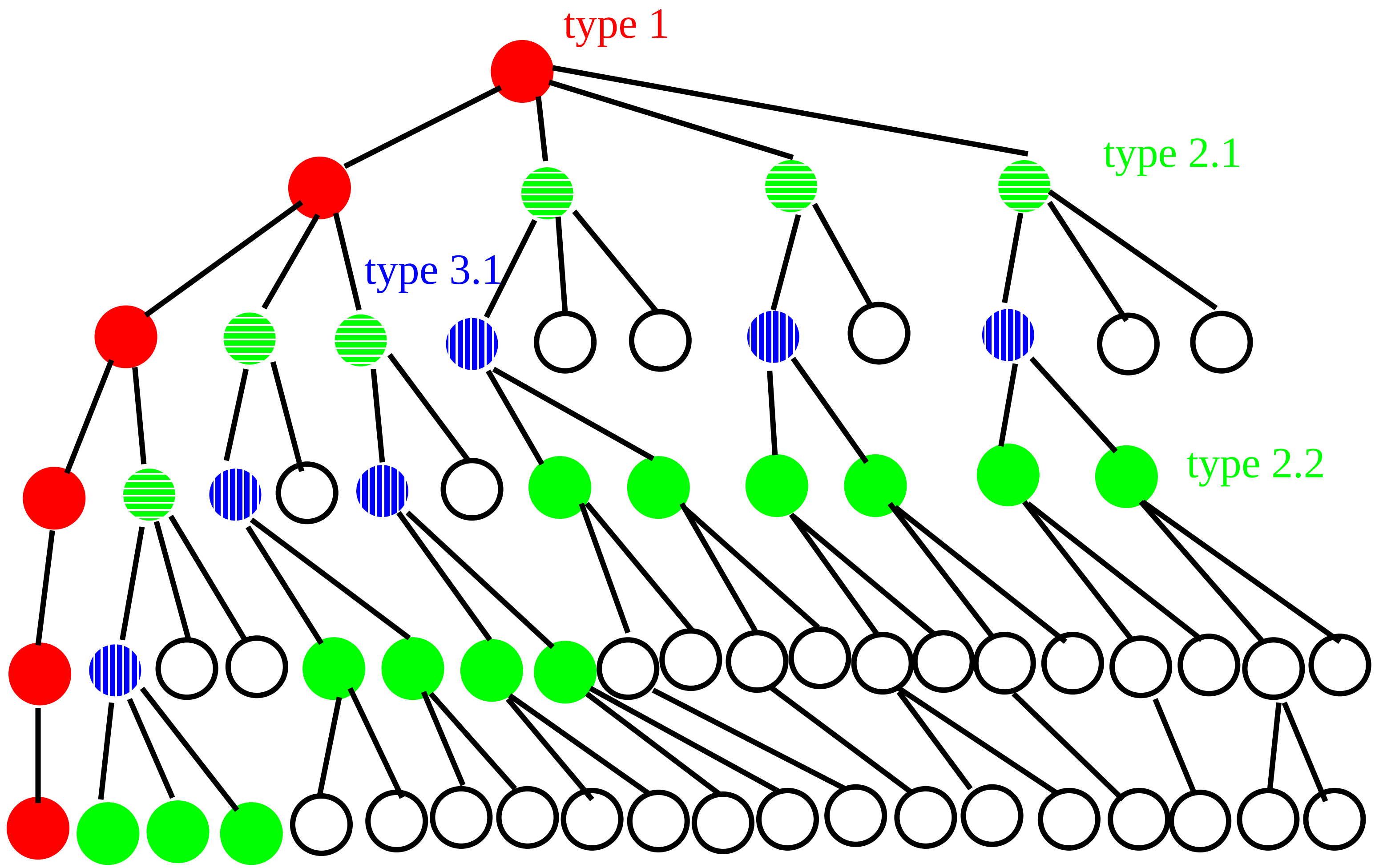


Illustration of all nodes

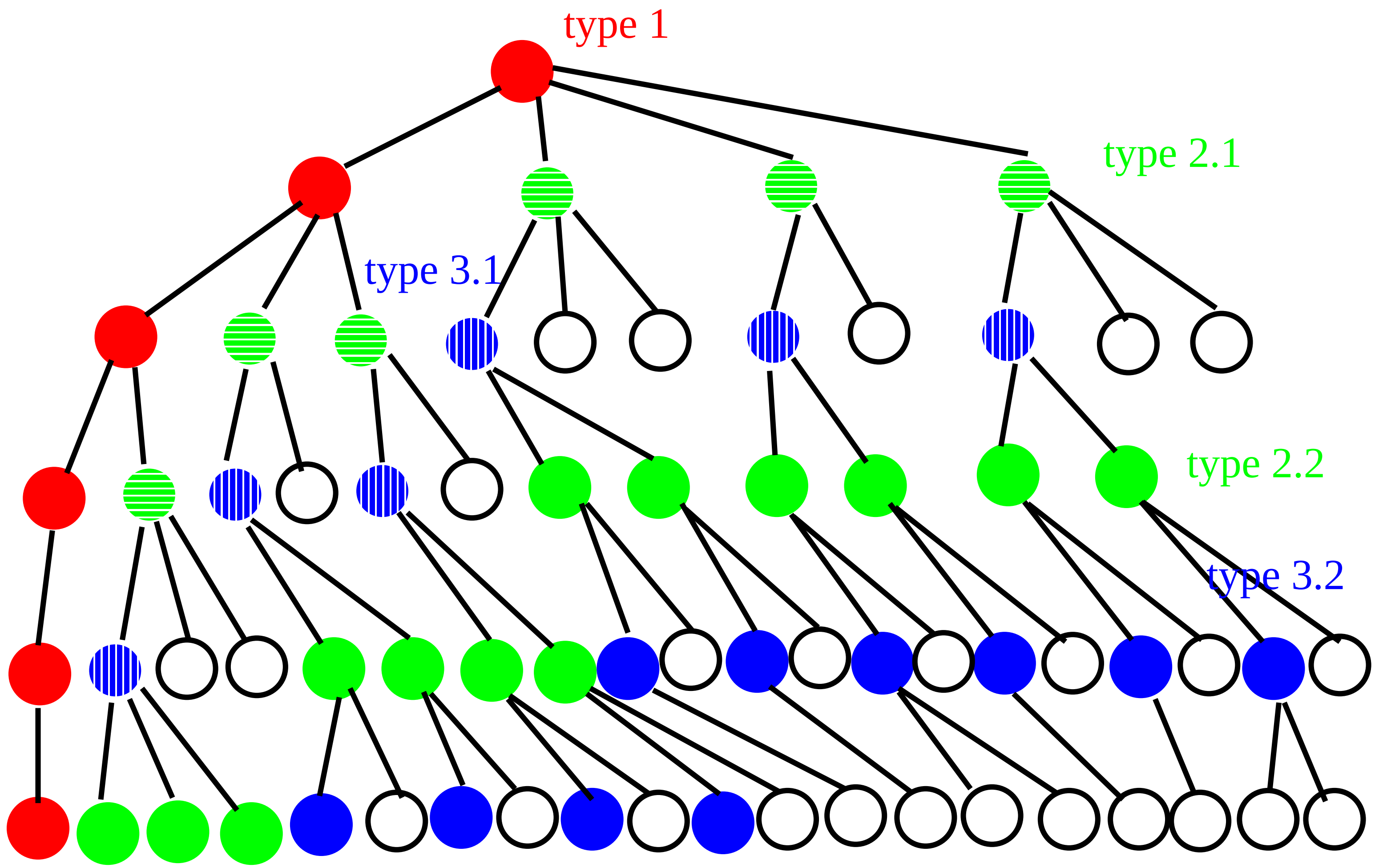


Illustration of all nodes

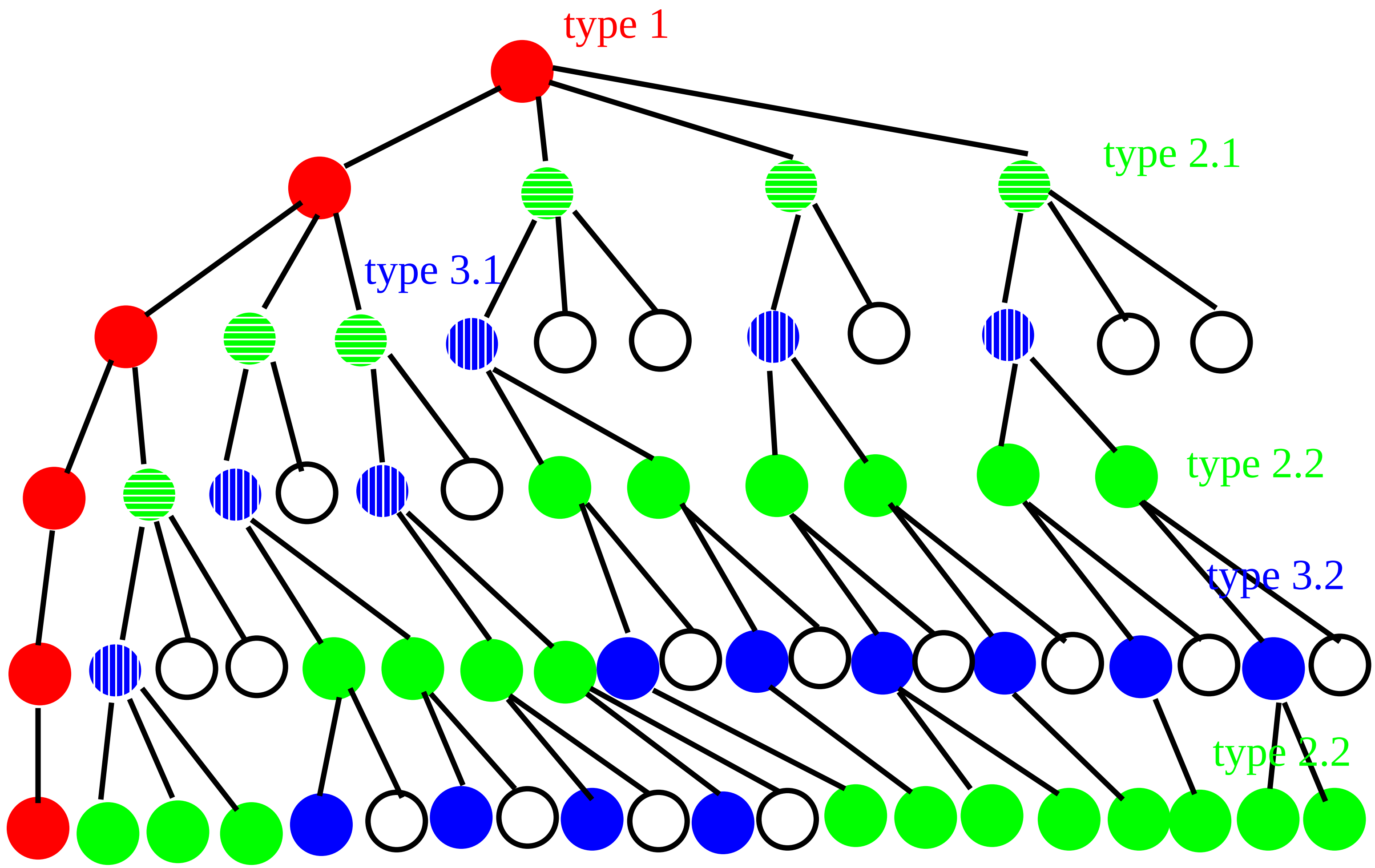
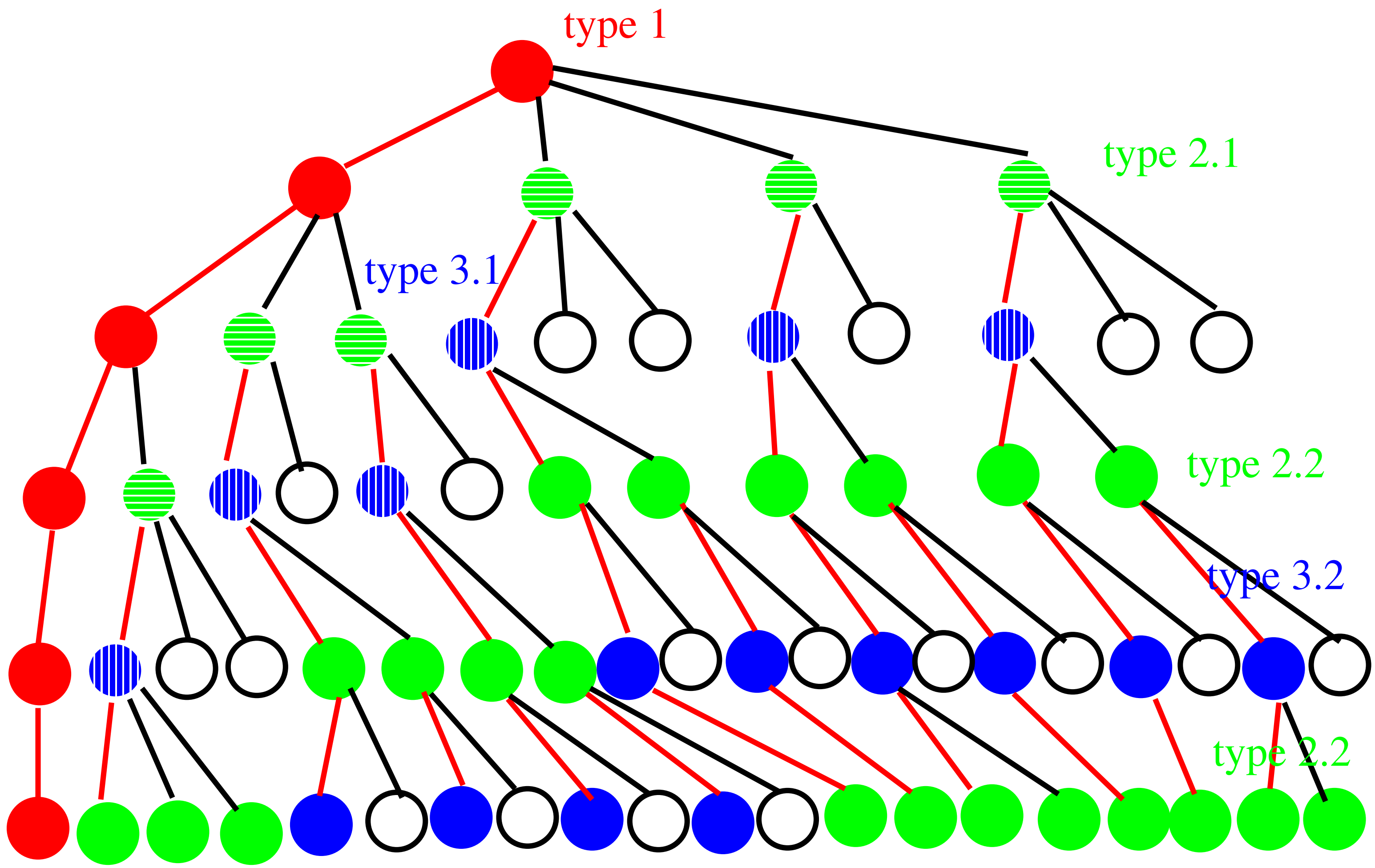


Illustration of all nodes



Theorem 1: Proof sketch

- **Properties (invariants)**
  - **A type 1 position p is examined by calling $F1(p, -\infty, \infty, depth)$**
    - ***p's first successor p_1 is of type 1***
    - $F(p) = -F(p_1) \neq \pm\infty$
    - ***p's other successors $p_2, \ldots, p_b$ are of type 2***
    - ***p_i, $i > 1$, are examined by calling $F1(p_i, -\infty, F(p_1), depth)$***
  - **A type 2 position p is examined by calling $F1(p, -\infty, beta, depth)$ where $-\infty < beta \leq F(p)$**
    - ***p's first successor p_1 is of type 3***
    - $F(p) = -F(p_1)$
    - ***p's other successors $p_2, \ldots, p_b$ are not examined***
  - **A type 3 position p is examined by calling $F1(p, alpha, \infty, depth)$ where $\infty > alpha \geq F(p)$**
    - ***p's successors $p_1, \ldots, p_b$ are of type 2***
    - ***they are examined by calling $F1(p_1, -\infty, -alpha, depth)$, $F1(p_2, -\infty, -\max\{m_1, alpha\}, depth), \ldots,$ $F1(p_i, -\infty, -\max\{m_{i-1}, alpha\}, depth)$ where $m_i = F1(p_i, -\infty, -\max\{m_{i-1}, alpha\}, depth)$***
- **Using an inductive argument to prove.**

Properties of Theorem 1

- **To cut off a subtree rooted at a node u entirely using alpha-beta based algorithms, at the very least, we need to know the values of**
  - **one of u's elder sibling, and**
  - **one of v' elder sibling where v is the parent of u.**
- **To know the value of a node rooted at a subtree, the subtree's left-most branch must be examined at the very least.**
- **Branches of a vertex that are examined**
  - **leftmost branch only**
    - ▷ ***type 2.1, whose leftmost child is type 3.1***
    - ▷ ***type 2.2, whose leftmost child is type 3.2***
  - **all branches**
    - ▷ ***type 1***
    - ▷ ***type 3.1***
    - ▷ ***type 3.2***

Analysis: best case

- **Corollary 1: Assume each position has exactly b successors**
  - **The number of positions examined by the alpha-beta procedure on level i is exactly**

$$b^{\lceil i/2 \rceil} + b^{\lfloor i/2 \rfloor} - 1.$$

- **Proof:**
  - **There are $b^{\lfloor i/2 \rfloor}$ sequences of the form $a_1. \cdots .a_i$ with $1 \leq a_i \leq b$ for all i such that $a_i = 1$ for all odd values of i.**
  - **There are $b^{\lceil i/2 \rceil}$ sequences of the form $a_1. \cdots .a_i$ with $1 \leq a_i \leq b$ for all i such that $a_i = 1$ for all even values of i.**
  - **We subtract 1 for the sequence $1.1. \cdots .1.1$ which are counted twice.**
- **Total number of nodes visited is**

$$\sum_{i=0}^{\ell} b^{\lceil i/2 \rceil} + b^{\lfloor i/2 \rfloor} - 1.$$

Comments for the best case

- **Assume we can afford to spend T time in searching a game tree with an average branching factor b.**
- **From T and the speed of your implementation, you can estimate the total number of nodes N that can be searched.**
- **From b and N, you can set the search depth limit d as follows**

$$b^d = N.$$

- **This means you can search to the depth of d using a brute force algorithm.**
- **Using alpha-beta pruning in the best case you can afford to search up to a depth of about $2 \cdot d - 1$ within the time T.**

Analysis: average case

- **Assumptions: Let a random game tree be generated in such a way that each position on level j has**
  - **a probability q_j of being nonterminal and**
  - **an average of b_j successors.**
- **Properties of the above random game tree**
  - **Expected number of positions on level ℓ is $b_0 \times b_1 \times \cdots \times b_{\ell-1}$**
  - **Expected number of positions on level ℓ examined by an alpha-beta procedure assumed the random game tree is perfectly ordered is**

$$b_0 q_1 b_2 q_3 \cdots b_{\ell-2} q_{\ell-1} + q_0 b_1 q_2 b_3 \cdots q_{\ell-2} b_{\ell-1} - q_0 q_1 \cdots q_{\ell-1} \text{ if } \ell \text{ is even;}$$

$$b_0 q_1 b_2 q_3 \cdots q_{\ell-2} b_{\ell-1} + q_0 b_1 q_2 b_3 \cdots b_{\ell-2} q_{\ell-1} - q_0 q_1 \cdots q_{\ell-1} \text{ if } \ell \text{ is odd}$$

- **Proof sketch:**
  - **If x is the expected number of positions of a certain type on level j, then $x \times b_j$ is the expected number of successors of these positions, and $x \times q_j$ is the expected number of "numbered 1" successors.**
  - **The above numbers equal to those of Corollary 1 when $q_j = 1$ and $b_j = b$ for $0 \leq j < \ell$.**

Comments for the average case (1/2)

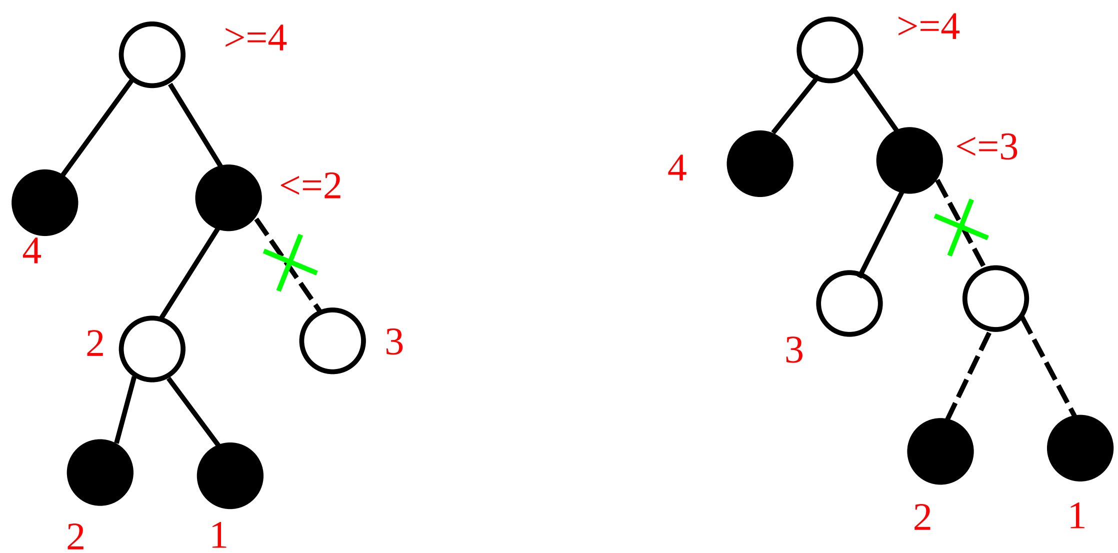
- [Knuth & Moore 1975] proved that with only the normal alpha-beta pruning across two adjacent levels, the effective branching factor in the average case is $O(b/\log b)$ where b is the average branching factor.
  - That is, in average, alpha-beta only searches one branch every $\log b$ branches encountered.
- [Fuller et al 1975] proved that together with deep alpha-beta pruning, the effective branching factor in the average case is $\sim b^{0.75}$ where b is the average branching factor.
- Direct alpha-beta pruning has more cuts in the endgame phase than in the open game phase.

Comments for the average case (2/2)

- **Assume you can afford to seraph b^d nodes in time T using brute force methods.**
  - **Note: given a tree of depth d and branching factor b, it has b^d nodes.**
- **In average, alpha-beta only searches one branch for every $b^{0.25}$ branches encountered.**
  - **Using alpha-beta pruning in the average case you can afford to search up to a depth of about $\frac{4}{3} \cdot d$ within the time T.**
- **However, within time T,**
  - **without deep alpha-beta pruning, the searching depth is only about $\frac{\log b}{\log b - \log\log b} \cdot d$, which means a lot of cut offs come from deep prunings;**
  - **in the best case, you can search up to the depth of $2 \cdot d - 1$.**
- **In practice, using a good move ordering heuristic plus other heuristics and techniques, Chinese chess programs can almost achieve a constant effective branching factor of about 3.**

Perfect ordering is not always the best

- **Intuitively, we may "think" alpha-beta pruning would be most effective when a game tree is perfectly ordered.**
  - **That is, when the first successor of every position is the best possible move.**
  - **This is not always the case!**



- **Truly optimum order of game trees traversal is not obvious.**

When is a branch pruned?

- **Assume a node r has two children u and v with u being visited before v using some move ordering.**
  - **Further assume u produced a new bound $bound$.**
- **Assume node v has a child w.**
  - **If the value new returned from w can cause a range conflict with $bound$, then branches of v later than w are cut.**
- **This means as long as the "relative" ordering of u and v is good enough, then we can have a cut-off.**
  - **There is no need to have a perfect ordering to enable cut-off to happen.**

Theorem 2

- **Theorem 2: Alpha-beta pruning is optimum in the following sense:**
  - **Given any game tree and any algorithm which computes the value of the root position, there is a way to permute the tree**
    - *by reordering successor positions if necessary;*
  - **so that every terminal position examined by the alpha-beta method under this permutation is examined by the given algorithm.**
  - **Furthermore if the value of the root is not ∞ or $-\infty$, the alpha-beta procedure examines precisely the positions which are critical under this permutation.**

Variations of alpha-beta search

- **Initially, to search a tree with the root r by calling $F1(r, -\infty, +\infty, depth)$.**
  - **What does it mean to search a tree with the root r by calling $F1(r, alpha, beta, depth)$?**
    - ▷ *To search the tree rooted at r requiring that the returned value to be within $alpha$ and $beta$.*
- **In an alpha-beta search with a pre-assigned window $(alpha, beta)$:**
  - **Failed-high means the correct value is larger than or equal to its upper bound $beta$.**
  - **Failed-low means the correct value is smaller than or equal to its lower bound $alpha$.**
- **Variations:**
  - **Brute force Nega-Max version: $F/F0$**
    - ▷ *Always finds the correct answer according to the Nega-Max formula.*
  - **Original (fail hard) alpha-beta cut (Nega-Max) version: $F1$**
  - **One-sided alpha-beta cut (Nega-Max) version: $F2$**
  - **Fail soft alpha-beta cut (Nega-Max) version: $F3$**

Original version (fail hard)

- **Requiring** $alpha \leq beta$; **nega-max version**
- **Algorithm** $F1$(**position** p, **value** $alpha$, **value** $beta$, **integer** $depth$)
  - determine the successor positions $p_1, \ldots, p_b$
  - if $b = 0$ // a terminal node
    or $depth = 0$ // remaining depth to search
    or time is running up // from timing control
    or some other constraints are met // add knowledge here
  - then return $h(p)$ else
  - begin
    - ▷ $m := alpha$ // *hard initial value*
    - ▷ ***for*** $i := 1$ ***to*** b ***do***
    - ▷ ***begin***
    - ▷ $t := -F1(p_i, -beta, -m, depth - 1)$
    - ▷ ***if*** $t > m$ ***then*** $m := t$ // ***the returned value is "used"***
    - ▷ ***if*** m ***is max or*** $m \geq beta$ ***then return***$(beta)$ // ***cut off and return the hard bound***
    - ▷ ***end***
  - end
  - return m // if nothing is over alpha, then alpha is returned

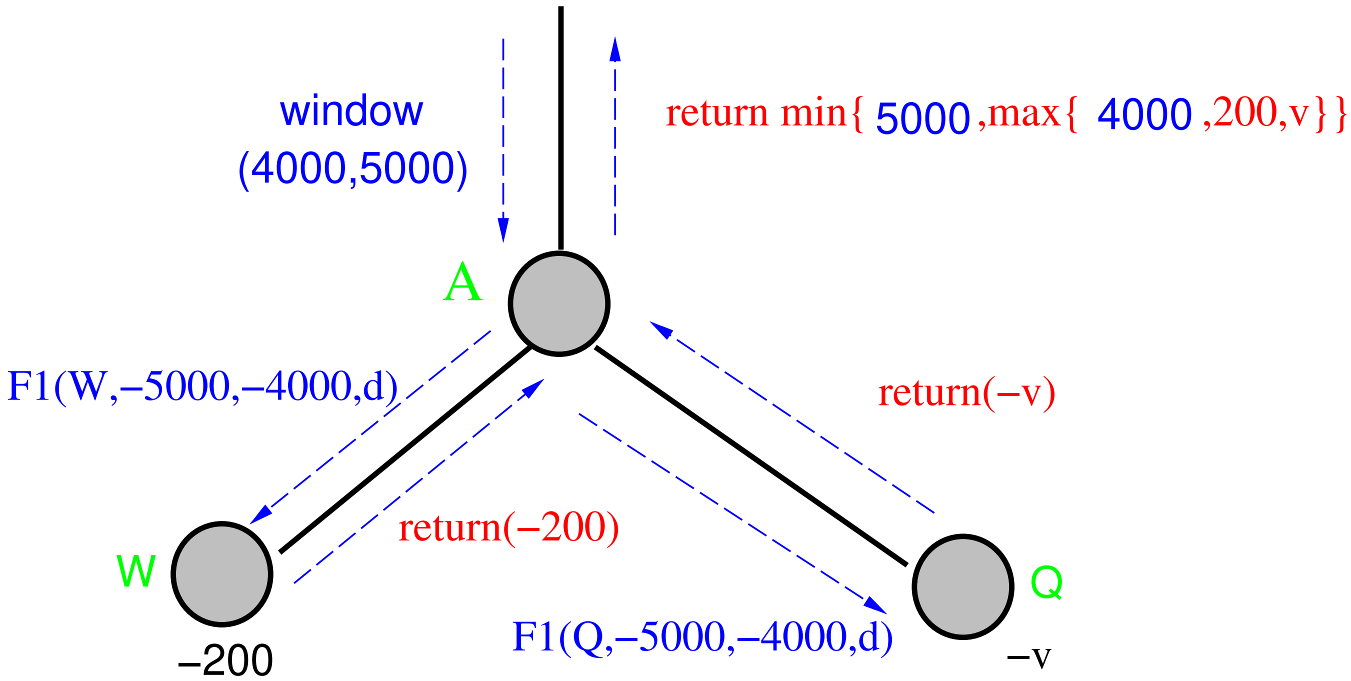
Properties of $F1$

- **Assumptions:**
  - $alpha \leq beta$
  - p **is not a leaf**
  - $depth = \infty$
  - **there is no additional resource or knowledge constraints**
- **Recall that** $F(p)$ **is the true value of** p**.**
- $F1(p, alpha, beta, depth) = alpha$ **if** $F(p) \leq alpha$
- $F1(p, alpha, beta, depth) = F(p)$ **if** $alpha < F(p) < beta$
- $F1(p, alpha, beta, depth) = beta$ **if** $F(p) \geq beta$
- $F1(p, -\infty, +\infty, depth) = F(p)$

Comments

- $F1(p, alpha, beta, depth)$: **find the best possible value according to a nega-max formula for the position p with the constraints that**
  - ***If $F(p) \leq alpha$, then $F1(p, alpha, beta, depth)$ returns with the value $alpha$ from a terminal position whose value is $\leq alpha$.***
  - ***If $F(p) \geq beta$, then $F1(p, alpha, beta, depth)$ returns the value $beta$ from a terminal position whose value is $\geq beta$.***
- **The meanings of $alpha$ and $beta$ during searching:**
  - ***For a max node: the current best value is at least $alpha$.***
  - ***For a min node: the current best value is at most $beta$.***
- $F1$ **always finds a value that is within $alpha$ and $beta$.**
  - ***Both bounds are hard, i.e., cannot be violated.***

$F1$: Example



- **As long as the value of the leaf node W is less than the current $alpha$ value, the returned value of A will be $alpha$.**
- **If the value of the leaf node W is greater than the current $beta$ value, the returned value of A will be $beta$.**

Version $F2$

- **Intuition**
  - **When something is over expected, then return this unexpected value the moment it appears.**
  - **When something is less expected, then continue searching.**
  - **MAX node:**
    - $(alpha, beta) = (-\infty, beta)$.
  - **MIN node:**
    - $(alpha, beta) = (alpha, \infty)$.
- **Deep alpha-beta cut-offs are not possible!**

Alpha-beta pruning: one-sided, Mini-Max (1/2

- **Note: one-sided bound.**
- **Algorithm $F2'$(position p, value $beta$, integer $depth$)**
  - // max node
  - determine the successor positions $p_1, \ldots, p_b$
  - if $b = 0$ // a terminal node
    or $depth = 0$ // remaining depth to search
    or time is running up // from timing control
    or some other constraints are met // add knowledge here
  - then return $f(p)$ else
    - ▷ $m := -\infty$
    - ▷ ***for*** $i := 1$ ***to*** b ***do***
    - ▷ $\quad t := G2'(p_i, m, depth - 1)$
    - ▷ $\quad$ ***if*** $t > m$ ***then*** $m := t$ ***// improve the current best value***
    - ▷ $\quad$ ***if*** m ***is max or*** $m \geq beta$ ***then return***(m) ***// beta cut off, return*** m
  - end;
  - return m // if nothing is over beta, then the largest one is returned

Alpha-beta pruning: one-sided, Mini-Max (2/2)

- **Note: one-sided bound.**
- **Algorithm $G2'$(position p, value $alpha$, integer $depth$)**
  - **// min node**
  - **determine the successor positions $p_1, \ldots, p_b$**
  - **if $b = 0$ // a terminal node**
    or $depth = 0$ // remaining depth to search
    or time is running up // from timing control
    or some other constraints are met // add knowledge here
  - **then return $f(p)$ else**
    - ▷ $m := \infty$
    - ▷ ***for*** $i := 1$ ***to*** b ***do***
    - ▷ $t := F2'(p_i, m, depth - 1)$
    - ▷ ***if*** $t < m$ ***then*** $m := t$ ***// improve the current best value***
    - ▷ ***if*** m ***is min or*** $m \leq alpha$ ***then return(m) // alpha cut off, return*** m
  - **end;**
  - **return m // if nothing is below alpha, then the smallest one is returned**

Alpha-beta pruning: one-sided, Nega-Max

- **Note: one-sided bound.**
- **Algorithm $F2$(position p, value $bound$, integer $depth$)**
  - **determine the successor positions $p_1, \ldots, p_b$**
  - **if $b = 0$ // a terminal node**
    or $depth = 0$ // remaining depth to search
    or time is running up // from timing control
    or some other constraints are met // add knowledge here
  - **then return $h(p)$ else**
  - **begin**
    - ▷ $m := -\infty$
    - ▷ ***for*** $i := 1$ ***to*** b ***do***
    - ▷ ***begin***
    - ▷ $\quad t := -F2(p_i, -m, depth - 1)$
    - ▷ $\quad$ ***if*** $t > m$ ***then*** $m := t$ ***// improve the current best value***
    - ▷ $\quad$ ***if*** m ***is max or*** $m \geq bound$ ***then return***(m) ***// cut off, return*** m ***that is*** $\geq bound$
    - ▷ ***end***
  - **end**
  - **return m**

Properties of $F2$

- **Assumptions:**
  - p **is not a leaf**
  - $depth = \infty$
  - **there is no additional resource or knowledge constants**
- **Recall that** $F(p)$ **is the true value of** p**.**
- $F2(p, bound, depth) = F(p)$ **if** $F(p) < bound$
- $F2(p, bound, depth) \geq bound$ **if** $F(p) \geq bound$
  - **Note that** $F(p) \geq F2(p, bound, depth)$ **in this case.**
- $F2(p, \infty, depth) = F(p)$

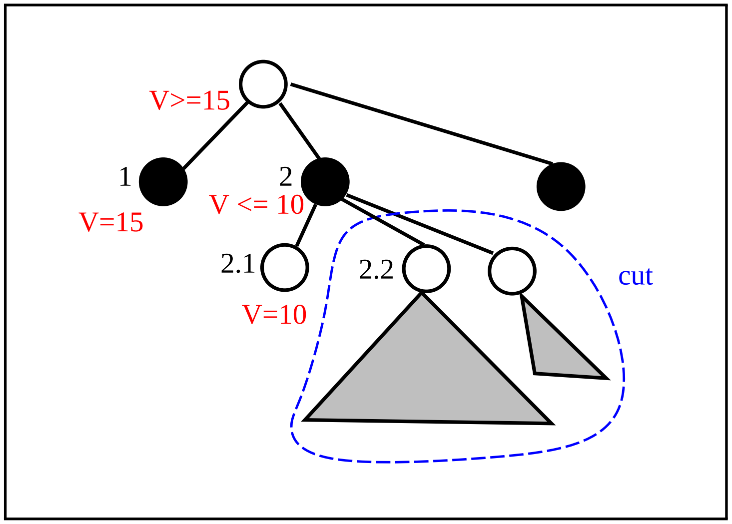
Comments: $F2$

- $F2(p, bound, depth)$: **find the best possible value according to a nega-max formula for the position p with the constraints that**
  - ▷ ***If*** $F(p) \leq bound$***, then*** $F2(p, bound, depth)$ ***returns*** $F(p)$***.***
  - ▷ ***If*** $F(p) \geq bound$***, then*** $F2(p, bound, depth)$ ***returns a value*** $\geq bound$ ***from a terminal position whose value is*** $\geq bound$***.***
- **An intermediate version.**
  - ▷ ***One-sided bounded.***
  - ▷ ***Always return something better than expected, but never something worse!!***
  - ▷ ***Easier to find the branch where the returned value is coming from.***
- **Can be treated as**
  - ▷ $F2'(p, -\infty, beta, depth)$
  - ▷ $G2'(p, alpha, \infty, depth)$
- **For historical reason [Fishburn 1983], this is called fail hard.**

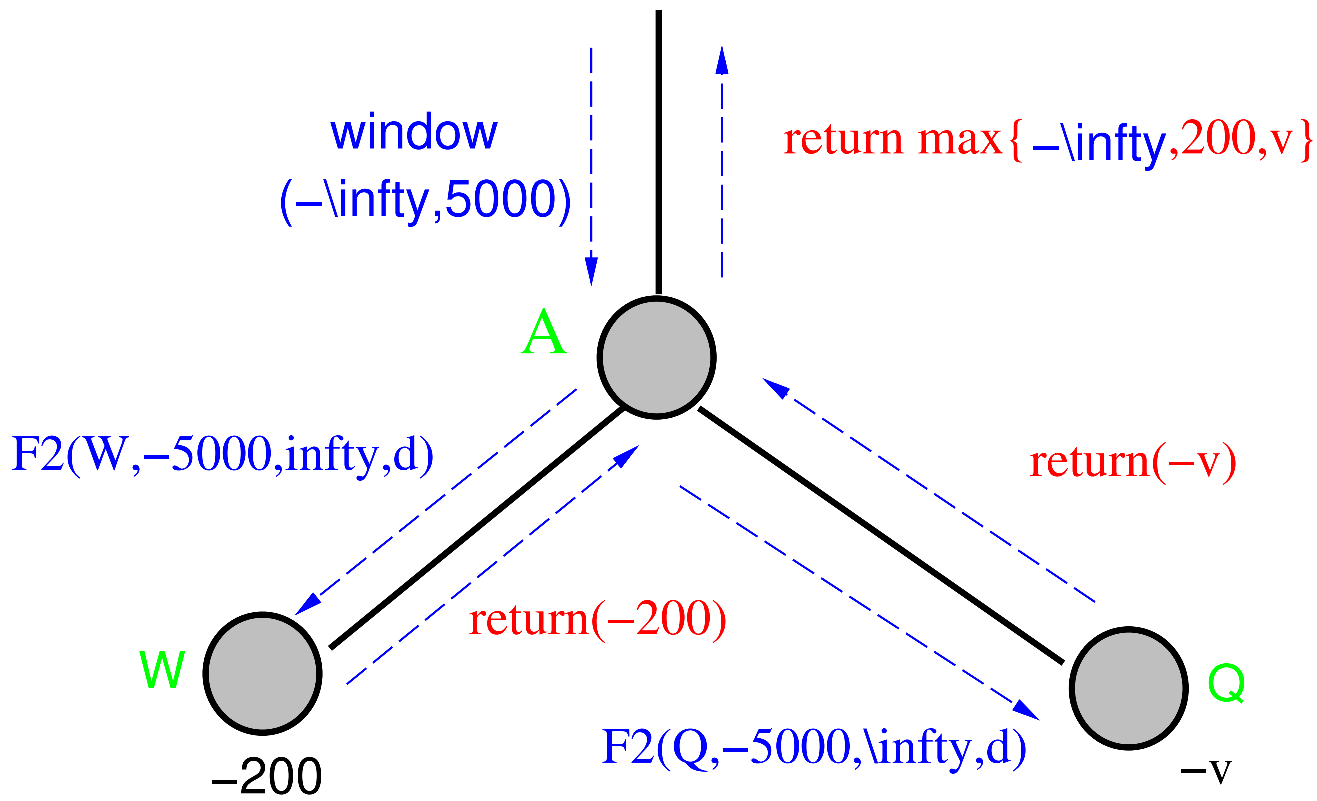
Example

Initial call: $F2'(\text{root},\infty,depth)$

- $m = -\infty$
- **call** $G2'$**(node 1,**∞**,**$depth - 1$**)**
  - ▷ ***it is a terminal node***
  - ▷ ***return value*** 15
- $t = 15$**;**
  - ▷ ***since*** $t > m$***,*** m ***is now*** 15
- **call** $G2'$**(node 2,**15**,**$depth - 1$**)**
  - ▷ ***call*** $F2'$***(node 2.1,***15***,***$depth - 2$***)***
  - ▷ ***it is a terminal node; return*** 10
  - ▷ $t = 10$***; since*** $t < \infty$***,*** m ***is now*** 10
  - ▷ $bound$ ***is*** 15***,*** m ***is*** 10***, so we have an alpha cut off,***
  - ▷ ***no need to call*** $F2'$***(node 2.2,10,***$depth - 2$***)***
  - ▷ ***return 10***
  - ▷ $\cdots$



$F2$: Example



- **As long as the value v of the leaf node W is less than the current $beta$ value, the returned value of A will be v.**
- **If the value of the leaf node W is greater than the current $beta$ value, the returned value of A will be the returned value of W.**

Version $F3$

- **Intuition**
  - **MAX node:**
    - ***Same with $F2$: when the value is more than $beta$, report this value, not just $beta$.***
    - ***Additional: if the value is less than $alpha$, report his value being a very bad node for a max node.***
    - ***Next time, this fact can be used to have a faster cut off.***
  - **MIN node:**
    - ***Same with $F2$: when the value is less than $alpha$, try to report this value, not just $alpha$.***
    - ***Additional: if the value is more than $beta$, report his value being a very bad node for a min node.***
    - ***Next time, this fact can be used to have a faster cut off.***

Alpha-beta pruning: Fail soft, Mini-Max (1/2)

- **Algorithm $F3'$(position p, value $alpha$, value $beta$, integer $depth$)**
  - // max node
  - determine the successor positions $p_1, \ldots, p_b$
  - if $b = 0$ // a terminal node
    or $depth = 0$ // remaining depth to search
    or time is running up // from timing control
    or some other constraints are met // add knowledge here
  - then return $f(p)$ else
  - begin
    - ▷ $m := -\infty$ *// soft initial value*
    - ▷ ***for*** $i := 1$ ***to*** b ***do***
    - ▷ ***begin***
    - ▷ $t := G3'(p_i, \max\{m, alpha\}, beta, depth - 1)$
    - ▷ ***if*** $t > m$ ***then*** $m := t$ ***// the returned value is "used"***
    - ▷ ***if*** m ***is max or*** $m \geq beta$ ***then return***(m) ***// beta cut off***
    - ▷ ***end***
  - end
  - return m

Alpha-beta pruning: Fail soft, Mini-Max (2/2)

- **Algorithm** $G3'$**(position** p**, value** $alpha$**, value** $beta$**, integer** $depth$**)**
  - **// min node**
  - **determine the successor positions** $p_1, \ldots, p_b$
  - **if** $b = 0$ **// a terminal node**
    or $depth = 0$ **// remaining depth to search**
    or time is running up // from timing control
    or some other constraints are met // add knowledge here
  - **then return** $f(p)$ **else**
  - **begin**
    - ▷ $m := \infty$ ***// soft initial value***
    - ▷ ***for*** $i := 1$ ***to*** b ***do***
    - ▷ ***begin***
    - ▷ $t := F3'(p_i, alpha, \min\{m, beta\}, depth - 1)$
    - ▷ ***if*** $t < m$ ***then*** $m := t$ ***// the returned value is "used"***
    - ▷ ***if*** m ***is min or*** $m \leq alpha$ ***then return(***m***) // alpha cut off***
    - ▷ ***end***
  - **end**
  - **return** m

Alpha-beta pruning: Fail soft, Nega-Max

- **Algorithm $F3$(position p, value $alpha$, value $beta$, integer $depth$)**
  - determine the successor positions $p_1, \ldots, p_b$
  - if $b = 0$ // a terminal node
    or $depth = 0$ // remaining depth to search
    or time is running up // from timing control
    or some other constraints are met // add knowledge here
  - then return $h(p)$ else
  - begin
    - ▷ $m := -\infty$ // *soft initial value*
    - ▷ ***for*** $i := 1$ ***to*** b ***do***
    - ▷ ***begin***
    - ▷ $t := -F3(p_i, -beta, -\max\{m, alpha\}, depth - 1)$
    - ▷ ***if*** $t > m$ ***then*** $m := t$ // ***the returned value is "used"***
    - ▷ ***if*** m ***is max or*** $m \geq beta$ ***then return***(m) // ***cut off***
    - ▷ ***end***
  - end
  - return m

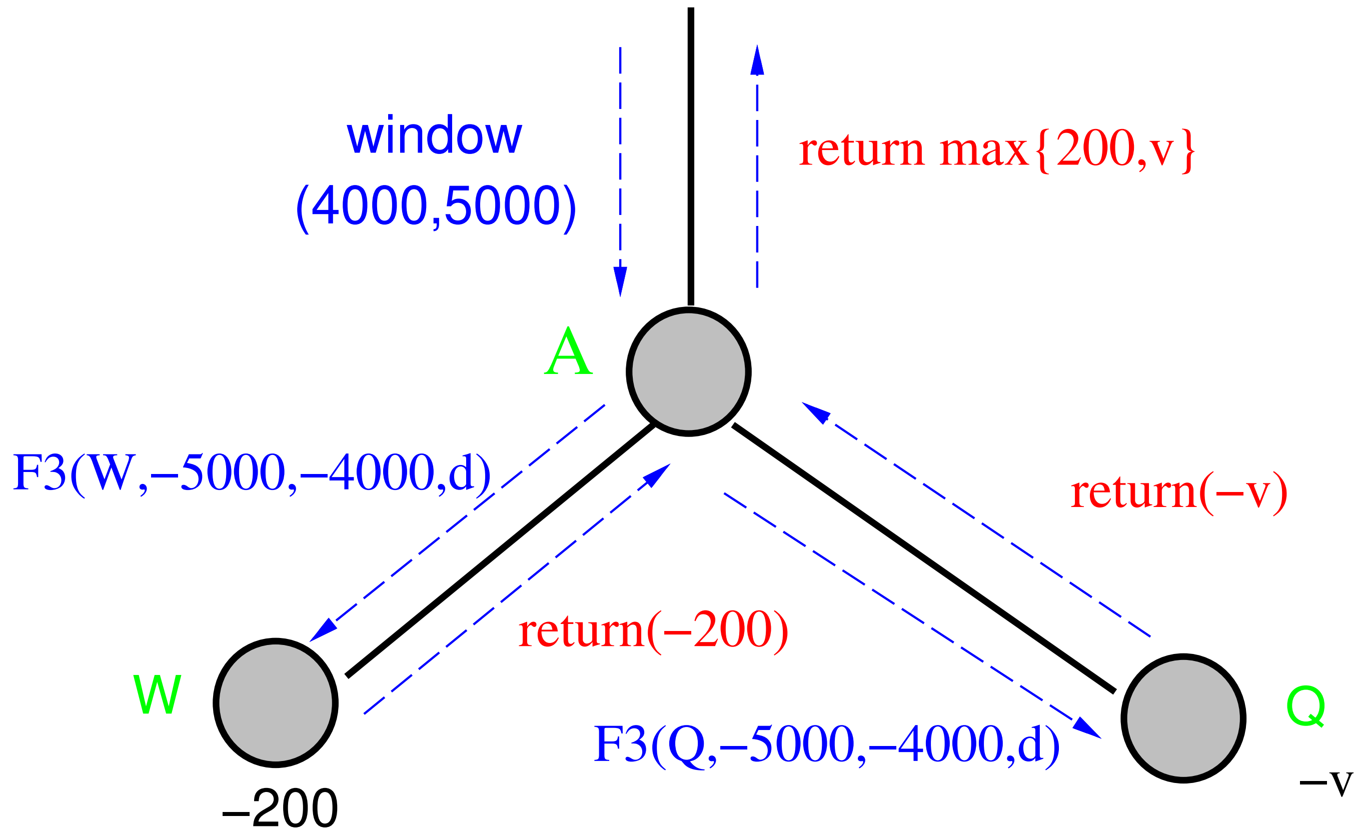
Properties of $F3$

- **Assumptions**
  - $alpha \leq beta$
  - p **is not a leaf**
  - $depth = \infty$
  - **there is no additional resource or knowledge constants**
- **Recall that** $F(p)$ **is the true value of** p**.**
- $F3(p, alpha, beta, depth) \leq alpha$ **if** $F(p) \leq alpha$
  - **Note that** $F(p) \leq F3(p, alpha, beta, depth)$ **in this case.**
- $F3(p, alpha, beta, depth) = F(p)$ **if** $alpha < F(p) < beta$
- $F3(p, alpha, beta, depth) \geq beta$ **if** $F(p) \geq beta$
  - **Note that** $F(p) \geq F3(p, alpha, beta, depth)$ **in this case.**
- $F3(p, -\infty, +\infty, depth) = F(p)$

Comments: $F3$

- $F3$ finds a "better" value when the value is out of the search window.
  - Better means a tighter bound.
    - ▷ *The bounds are soft, i.e., can be violated.*
  - When it is failed-high, $F3$ normally returns a value that is higher than that of $F1$ or $F2$.
    - ▷ *Never higher than that of F!*
  - When it is failed-low, $F3$ normally returns a value that is lower than that of $F1$ or $F2$.
    - ▷ *Never lower than that of F!*
- Example: assume you search the root r, a MAX node, with a very high $alpha$ value and actually $F(r) << alpha$.
  - $F2(r, alpha, beta, \infty)$ returns $alpha$.
  - $F3(r, alpha, beta, \infty)$ may return a value $< alpha$ which is more informatic than returning $alpha$.

Fail soft version (F3): Example

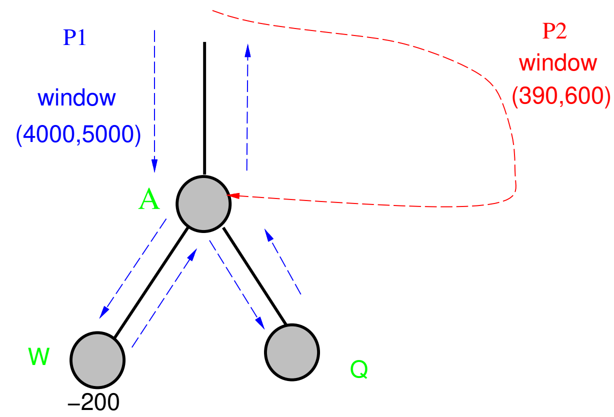


- **Let the value of the leaf node W be u.**
- **If $u < alpha$, then the returned value of A will be at least u.**

Comparisons between $F1$ and $F3$

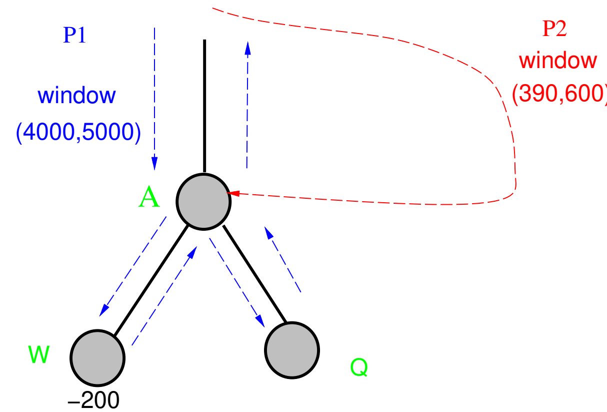
- Both versions find the corrected value v if v is within the window $(alpha, beta)$.
- Both versions scan the same set of nodes during searching.
  - ▷ *If the returned value of a subtree is decided by a cut, then $F1$ and $F3$ return the same value.*
- $F3$ provides more information when the true value is out of the pre-assigned search window.
  - Can provide a feeling on how bad or good the game tree is.
  - Use this "better" value to guide searching later on.
- $F3$ saves about 7% of time than that of $F1$ when a transposition table is used to save and re-use searched results [Fishburn 1983].
  - A transposition table is a data structure to record the results of previous searched results.
  - The entries of a transposition table can be efficiently accessed, i.e., read and write, during searching.
  - Need an efficient addressing scheme, e.g., hash, to translate between a position and its address.

$F1$ and $F3$: Example (1/2)



- **Assume the node A can be reached from the starting position using path P_1 and path P_2.**
  - **If W is visited first along P_1 with a window $(4000, 5000)$, and returns a value of 200, then**
    - ▷ *the returned value of W, 200, is stored into the transposition table.*
  - **If A is visited again along P_2 with the window $(390, 600)$, then a better value of previously stored value of W helps to decide whether the subtree rooted at W needs to be searched again.**

$F1$ and $F3$: Example (2/2)



- **Fail soft version has a chance to record a "better" value to be used later when this position is revisited.**
  - **If A is visited again along P_2 with the window $(390, 600)$, then**
    - ▷ ***it does not need to be searched again, since the previous stored value of W is -200.***
  - **However, if the value of W is 450, then it needs to be searched again.**
- **Fail hard version does not store the returned value of W after its first visit since this value is less than $alpha$.**

Concluding remarks

- We compare $F1$ and $F3$, and remember that $F2$ is the slowest one since it has no deep cut-offs.
  - To me, $F1$ fails really hard. $F2$ is only an intermediate version!
  - However, $F1$ is never a choice over $F2$ and $F3$ historically.
  - People first use $F2$, then $f3$, never $F1$.
  - Q: Is there any use to have a version that the upper bound cannot be violated, but the lower bound can?
- What move ordering is good?
  - It may not be good to search the best possible move first.
  - It may be better to cut off a branch with more nodes first.
- Q: How about the case when the tree is not uniform?
- Q: What is the effect of using iterative-deepening alpha-beta cut off?
- Q: How about the case for searching a game graph instead of a game tree?
  - Some nodes are visited more than once.

References and further readings

* D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6:293–326, 1975.
* John P. Fishburn. Another optimization of alpha-beta search. *SIGART Bull.*, (84):37–38, 1983.
- J. Pearl. The solution for the branching factor of the alpha-beta pruning algorithm and its optimality. *Communications of ACM*, 25(8):559–564, 1982.
- Fuller, S.H, Gaschnig, J.G. and Gillogly, J.J. Analysis of the Alpha-beta Pruning Algorithm Carnegie Mellon University. Computer Science Department https://books.google.com.tw/books?id=cOTmlwEACAAJ, 1973