Slides for a Course on the Analysis and Design of Algorithms

Chapter 5: Basic Search Techniques for Graphs

Stephen J. Hegner
Department of Computing Science
Umeå University
Sweden
hegner@cs.umu.se
http://www.cs.umu.se/~hegner

5. Basic Search Techniques for Graphs

5.1 Binary Tree Traversal

5.1.1 A review of terminology for the traversal of binary trees

- The three basic traversal strategies are *preorder*, *inorder*, and *postorder*.
- The high-level control structures are described below.

  procedure $\mathit{preorder}(T : \mathit{bin\_tree});$
  $\langle$ **if** $T \neq \emptyset$
  then $\langle\ \mathit{visit}(\mathit{root}(T));\ \mathit{visit}(\mathit{left\_child}(T));\ \mathit{visit}(\mathit{right\_child}(T));$
  $\rangle$
  procedure $\mathit{inorder}(T : \mathit{bin\_tree});$
  $\langle$ **if** $T \neq \emptyset$
  then $\langle\ \mathit{visit}(\mathit{left\_child}(T));\ \mathit{root}(T));\ \mathit{visit}(\mathit{right\_child}(T));\ \rangle$
  $\rangle$
  procedure $\mathit{postorder}(T : \mathit{bin\_tree});$
  $\langle$ **if** $T \neq \emptyset$
  then $\langle\ \mathit{visit}(\mathit{left\_child}(T));\ \mathit{visit}(\mathit{right\_child}(T));\ \mathit{visit}(\mathit{root}(T));$
  $\rangle$

- There are also three mirror-image traversals with left and right reversed.

5.1.2 The complexity of tree traversal *Assume that the following conditions hold:*

(a) *The time required to reach a left or right child vertex, from its parent, is* $\Theta(1)$.

(b) *The time required to reach a parent vertex, from a left or right child vertex, is* $\Theta(1)$.

(c) *The time required for a visit operation is* $\Theta(f)$, *for some complexity function* f.

Then:

- *The time required for tree traversal in any of the above cases is* $\Theta(n \cdot f)$.
- *In particular, if* $\Theta(f) = \Theta(1)$, *then the time required is* $\Theta(n)$. $\square$

5.2 Searching Trees

5.2.1 Basic assumptions

- The following basic assumptions are made about the context of searching trees.
  - All trees are *rooted*.
  - The number of children of a vertex is arbitrary, but finite.
  - The children (and so subtrees) of each vertex are ordered and numbered, from left to right.

5.2.2 Depth-first search of trees

- The basic recursive algorithm is as follows:

```
/*  Recursive depth-first search  */
procedure RDFS(T : tree);
  ⟨ if T = ∅
       then failure
        else ⟨ visit(root(T));
                 if item found
                    then success
                     else foreach subtree S of T do
                              RDFS(S);
               ⟩
  ⟩
```

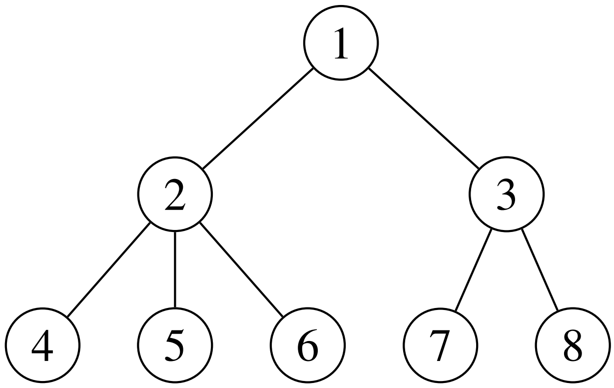
- While depth-first search is a naturally recursive process, it is instructive to remove the recursion.
- Assume that the abstract data type (ADT) *stack* is available:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\text{type } ST \; = \; \mathit{stack} \text{ of } T_0;} \\
\multicolumn{2}{l}{/* \;\; \text{The operations are the following:} \;\; */} \\
\mathit{insert}: & T_0 \times ST \rightarrow ST \\
\mathit{delete}: & ST \rightarrow T_0 (\times ST) \\
\mathit{is\_empty}: & ST \rightarrow \mathit{boolean} \\
\mathit{new}: & \mathbf{1} \rightarrow ST
\end{array}
$$

- A high-level description of depth-first search now becomes:

```
/*  Non-recursive depth-first search  */
procedure DFS(T : tree)
   S : stack_of ptr_to(vertex);
   n : ptr_to(vertex);
   ⟨ new(S);
     if T = ∅
        then failure
         else ⟨ insert(ptr_to(root(T)), S);
                 while (¬is_empty(S)) do
                        ⟨ n ← delete(S);
                          visit(vertex_of(n));
                          if item found
                             then success
                              else ⟨ foreach d ∈ children(vertex_of(n)) do
                                          insert(ptr_to(d), S);
                                   ⟩
                        ⟩
     ⟩
```

- The following is a sample tree on which to run the algorithm.



- Note that if the child vertices are pushed onto the stack from left to right, then they are processed from right to left.

  Example: (of order of visit) $\langle 1, 3, 8, 7, 2, 6, 5, 4 \rangle$

- To process them from left to right, push them onto the stack from right to left.

  Example: (of order of visit) $\langle 1, 2, 4, 5, 6, 3, 7, 8 \rangle$

5.2.3 Breadth-first search of trees

- The algorithm for breadth-first search is *exactly* the same as that for (nonrecursive) depth-first search, save that S is taken to be a *queue* instead of a stack.
- Note that there is no problem of order reversal; children of a vertex are pushed onto the queue in left-to-right order.

5.2.4 Best-first search of trees

- In best-first search, it is assumed that there is some sort of evaluation function on the vertices which indicates which are most promising.
- The vertices are maintained in a priority queue.
- The algorithm for breadth-first search is *exactly* the same as that for (nonrecursive) depth-first search, save that S is taken to be a *priority queue* instead of a stack.

5.3 Searching and Traversing Directed Graphs

5.3.1 Assumptions and conventions

- All graphs to be searched have a distinguished *start vertex*.
- All vertices are reachable from the start vertex.
- Each vertex has a *mark* field, which is used to tag each vertex which is visited, so it is not processed repeatedly.

5.3.2 Depth-first traversal of general cyclic graphs

- The simplest and most general way to realize depth-first search is to use recursion.

$/*$ Assume that all vertices are initially unmarked. $*/$
$/*$ $graph\_of(v)$ the subgraph whose start vertex is v. $*/$
$/*$ Incoming edges to v are ignored. $*/$
procedure $DFGS(G : directed\_graph)$;
  $\langle$ $visit(start\_vertex(G))$;
   $mark(start\_vertex(G))$;
   foreach $v \in adjacent(start\_vertex(G))$ **do**
    if $(\neg marked(v))$
     then $DFGS(graph\_of(v))$;
  $\rangle$

- Note that the order of search is dependent upon the order in which the vertices are selected in the foreach statement.
- It is easy to convert this to a search.
- Just terminate when the desired element is found.

5.3.3 Stack-based depth-first traversal of general cyclic graphs

$/*$ Assume that all vertices are initially unmarked. $*/$
procedure *NRDFGS*$(G : \mathit{directed\_graph})$;
  $S : \mathit{stack}$ of $\mathit{ptr\_to}(\mathit{vertex})$;
  $n : \mathit{ptr\_to}(\mathit{vertex})$;
  $\langle$ $\mathit{new}(S)$;
    $\mathit{insert}(\mathit{ptr\_to}(\mathit{start\_vertex}(G), S))$;
    while $(\neg \mathit{is\_empty}(S))$ **do**
        $\langle$ $n \leftarrow \mathit{delete}(S)$;
          if $(\mathit{unmarked}(\mathit{vertex\_of}(n))$
            then $\langle$ $\mathit{mark}(\mathit{vertex\_of}(n))$;
                $\mathit{visit}(\mathit{vertex\_of}(n))$;
                foreach $m \in \mathit{children}(\mathit{vertex\_of}(n))$ **do**
                  if $\mathit{unmarked}(m)$
                    then $\mathit{insert}(\mathit{ptr\_to}(m), S)$;
              $\rangle$
    $\rangle$

- Note that vertices are marked as they are visited, and not as they are pushed onto the stack.
- This approach is taken because a vertex may be reached in many different ways, and so pushed onto the stack several times.
- It is possible to avoid pushing a vertex onto the stack more than once.
- To do so, backpointers from the vertices to their entries in the stack must be maintained.

- If a vertex is to be pushed onto the stack, a check to see whether or not it is already on the stack is made first.
- If it is already on the stack, then that entry is removed or disabled, and only the new, top one retained.

5.3.4 Breadth-first traversal of graphs

```
/*  Assume that all vertices are initially unmarked  */
procedure BFGS(G : directed_graph);
  Q : queue of ptr_to(vertex);
  n : ptr_to(vertex);
  ⟨ new(Q);
    insert(ptr_to(start_vertex(G), Q));
    mark(start_vertex(G));
    while (¬is_empty(Q)) do
          ⟨ n ← delete(Q);
            visit(vertex_of(n));
            foreach m ∈ children(vertex_of(n)) do
              if (unmarked(vertex_of(m)))
                then ⟨ insert(ptr_to(m), Q);
                       mark(vertex_of(m));
                     ⟩
          ⟩
  ⟩
```

- Note that vertices are marked as they are inserted into the queue.
- Thus, there is no need to move them within the queue, as there is within the stack with depth-first search.

5.3.5 Best-first search of graphs

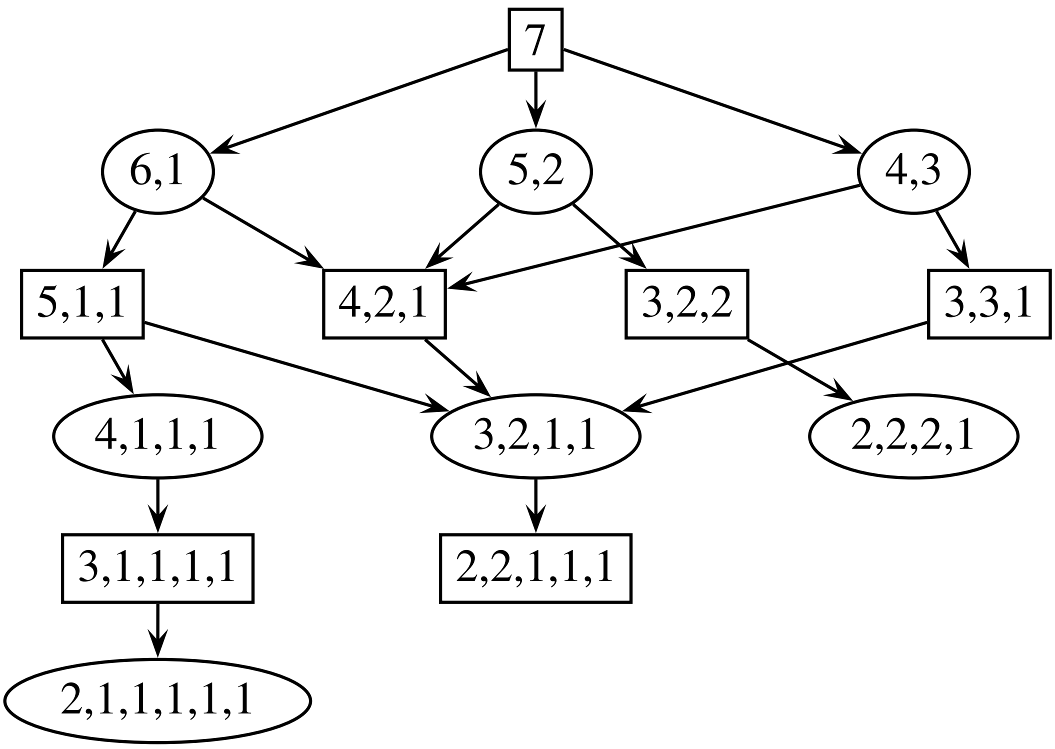
- The approach is exactly the same as for breadth-first search, save that a priority queue is used.

5.4 Game Graphs

- In this subsection, game graphs are presented as an application of searching.
- Attention is restricted to two-person games with perfect information.

5.4.1 Grundy's game — a simple example

- Begin with a stack of *n* coins.
- Each of two players moves, in turn.
- A move consists of splitting some pile of coins into two unequally sized piles, each nonempty.
- The player who cannot continue loses.
- The graph below illustrates the situation for an initial stack of seven coins.

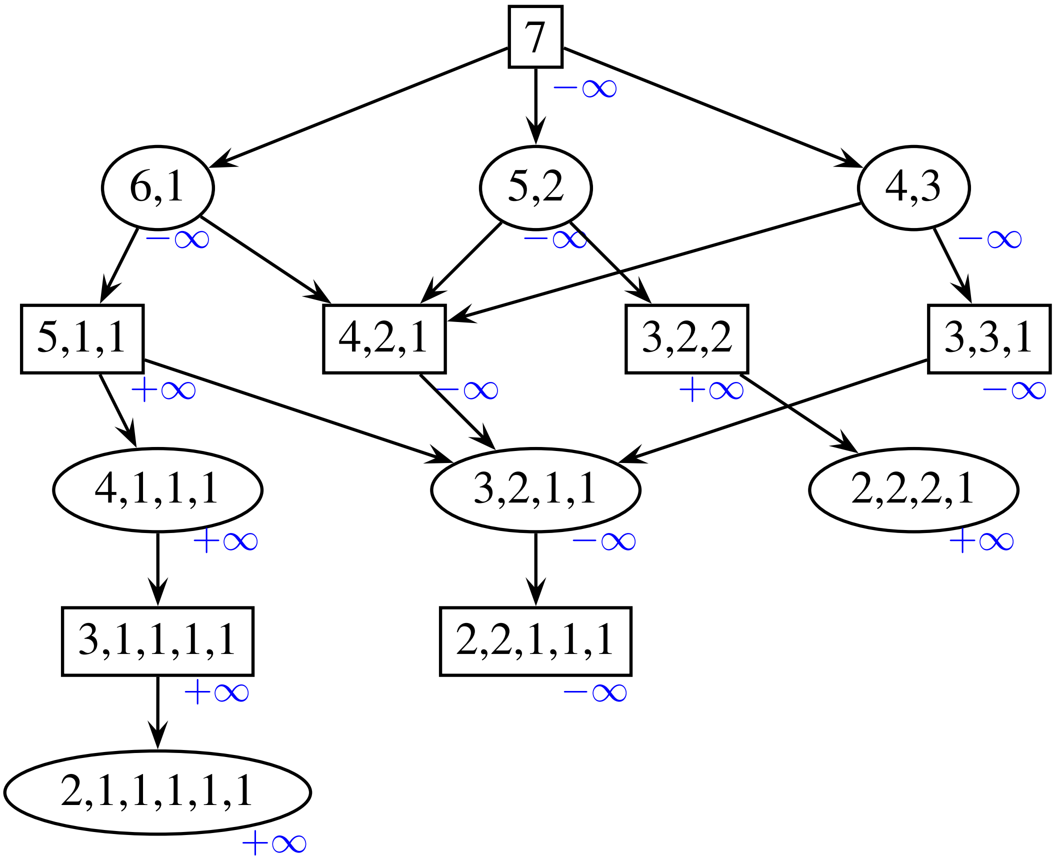


- Define the value of a leaf vertex to be:
  - $+\infty$ if it is an oval vertex (player 1 wins);
  - $-\infty$ if it is a rectangular vertex (player 2 wins);
- For interior vertices, define the values as follows:

$$\text{oval vertices} = \begin{cases} -\infty & \text{if the value of at least one of its successors is } -\infty \\ +\infty & \text{if the value of each of its successors is } +\infty. \end{cases}$$

$$\text{rect vertices} = \begin{cases} +\infty & \text{if the value of at least one of its successors is } +\infty \\ -\infty & \text{if the value of each of its successors is } -\infty. \end{cases}$$

- For the game graph of the previous slide, the values of the vertices are as follows.



5.4.2 Theorem *In a two-person game with perfect information, if both players make optimal moves, player 1 will win iff the root is a* $+\infty$ *vertex.* $\square$

5.4.3 Max-min graphs

(a) An *max-min graph* is a directed, acyclic graph $G = (V, E, g)$ with a distinguished start vertex $s \in V$ and two classes of non-leaf vertices:

- max vertices;
- min vertices;

subject to the constraints:

- Every successor of a max vertex is a min vertex;
- Every successor of a min vertex is a max vertex.

(b) A *value function* for G is a function

$$p : V \to \mathbb{Z} \cup \{-\infty, +\infty\}$$

subject to the following constraints:

(i) For a leaf vertex v, $p(v)$ is arbitrary.

(ii) For a max vertex v,

$$p(v) = \max(\{p(w) \mid w \text{ is a direct successor of } v\}$$

(iii) For a min vertex v,

$$p(v) = \min(\{p(w) \mid w \text{ is a direct successor of } v\}$$

(c) The *value* of (G, p) is $p(s)$.

5.4.4 The need for approximation

- A general top-down strategy for determining $p(s)$ begins with s and recursively evaluates each subgraph.
- For graphs arising from larger games, the cost becomes prohibitive.
- Examples sizes for the full game graph, using symmmetries:

  <u>tic-tac-toe</u>: 10^5 vertices.

  <u>checkers</u>: 10^{40} vertices.

  <u>chess</u>: 10^{120} vertices.

- In the latter two cases, it is impossible to generate the entire game tree.
- Approximation schemes are clearly necessary.

5.4.5 The evaluation-function strategy

- In the *evaluation function strategy*, the entire game tree is not generated.
- Rather, it is only generated to a predetermined level.
- At the leaf level, an estimate of the quality of each vertex is made.
- Based upon these estimates, a heuristic procedure for obtaining values at deeper levels is employed.
- As an example, consider the simple game of tic-tac-toe.
- This game may result in a draw, so there are three “final” values:

$$\begin{array}{cl} +\infty & \text{if X wins} \\ -\infty & \text{if O wins} \\ 0 & \text{for a draw} \end{array}$$

- The max/min strategy may still be applied.
- A simple evaluation function for a board configuration b is the following:

$$e(b) = \begin{cases} +\infty & \text{if X has won} \\ -\infty & \text{if O has won} \\ \#_b(\text{X}) - \#_b(\text{O}) & \text{otherwise} \end{cases}$$

in which

$$\#_b(Z) = \text{number of rows, columns, and diagonals open for } Z$$

with $Z \in \{X, O\}$.

- The following configuration b has $\#(X) = 6$ and $\#(O) = 4$, so $e(b) = 2$.

	O	
	X	

- Note that $e(b) = 0$ for a draw configuration under this definition.
- On the next slide, the expansion of the first two levels of a game is shown.
- Configurations which are equivalent under rotation or reflection are not repeated.
- The following two configurations are equivalent under a rotation of 90°.

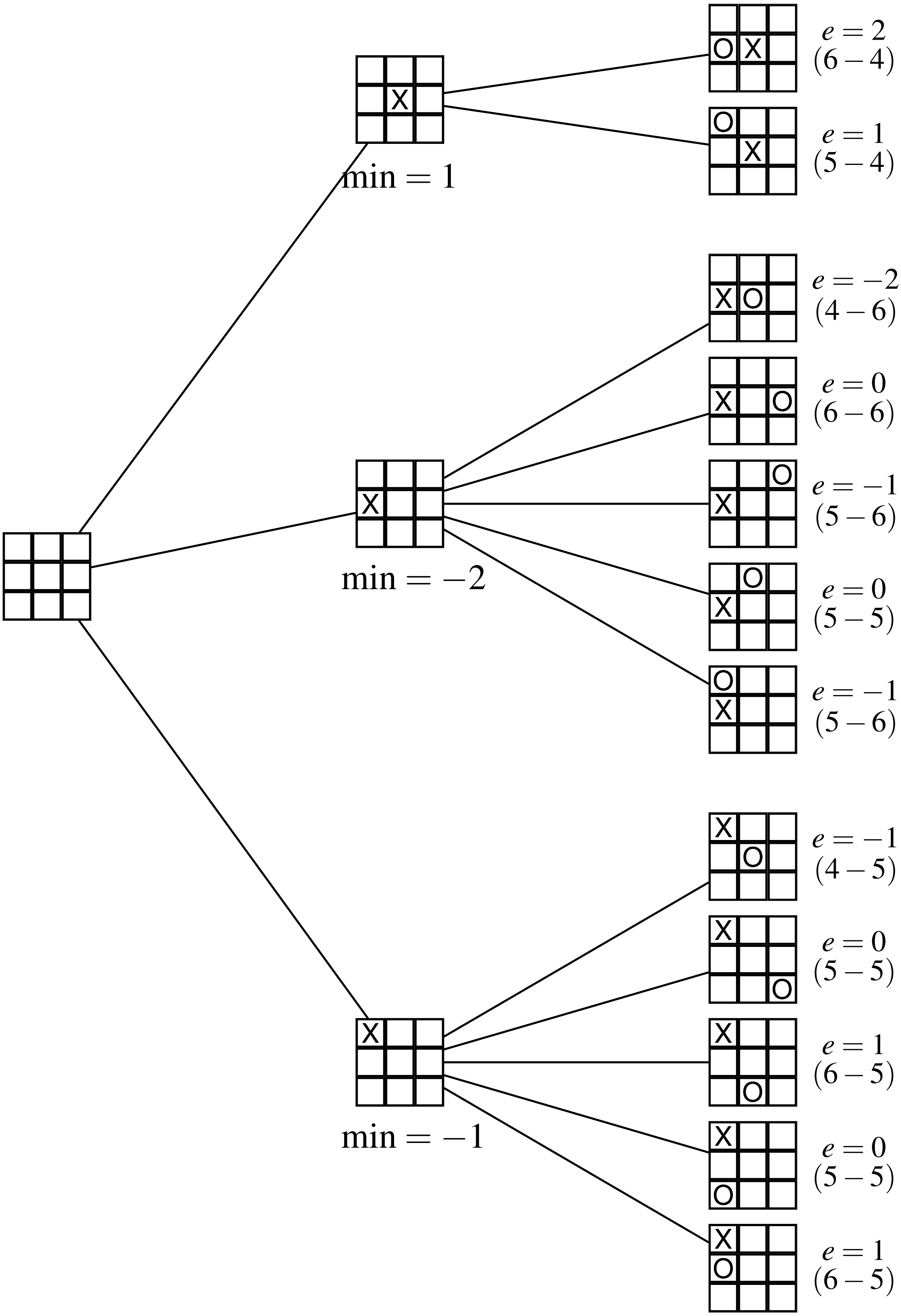
	O	
	X	

	X	O

- The following two configurations are equivalent under a reflection.

X	O	

	O	X

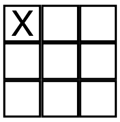


X
O
$e = 2$
$(6-4)$
$e = 1$
$(5-4)$
$\min = 1$
$e = -2$
$(4-6)$
$e = 0$
$(6-6)$
$e = -1$
$(5-6)$
$e = 0$
$(5-5)$
$e = -1$
$(5-6)$
$\min = -2$
$e = -1$
$(4-5)$
$e = 0$
$(5-5)$
$e = 1$
$(6-5)$
$e = 0$
$(5-5)$
$e = 1$
$(6-5)$
$\min = -1$

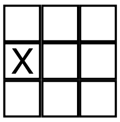
- Decision is via a so-called "min-max" procedure.
- Player X is a maximizer, since a higher $e(-)$ score on a vertex favors X.
- In particular, a score of $+\infty$ means a victory for player X.
- Thus, a vertex for which X has the next move is a max vertex.
- Similarly, player O is a minimizer, and so a vertex for which O has the next move is a min vertex.
- Formally, to expand upon the definitions of 5.4.3:
    (a) A *max vertex* in a two-person game tree is a vertex in which the player who is trying to maximize the score (*i.e.*, who wins with $+\infty$) has the next move.
    (a) A *min vertex* in a two-person game tree is a vertex in which the player who is trying to minimize the score (*i.e.*, who wins with $-\infty$) has the next move.
- In the graph, player X chooses the move b which yields the best *guaranteed* value of $e(b)$.
- Player X assumes that player O will make the best move.
- Thus, player X chooses the move which will be the least damaging, in the case that player O makes an optimal move.
- Player X choose the vertex at level one whose whose minimum-value child has the maximum value.
- Once player X makes a move, this process is repeated, with the new configuration as root vertex.

5.4.6 The α-β pruning strategy

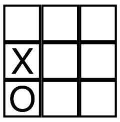
- It is not always necessary to generate the full tree to *n* levels in an *n*-level evaluation strategy.
- Some subtrees may be eliminated via a *pruning* strategy.
- In the example of 5.4.5, with depth-first expansion, after evaluating the vertex

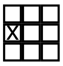
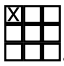
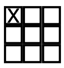
  

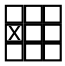
  at the second level of the tree, and determining its min-value to be -1, the vertex

  is expanded.
- Note that its first descendant

  evaluates to -1.
- Since  is a min vertex, its value cannot exceed -1.
- Thus, it cannot be a better choice for X than , since X seeks to maximize.
- Hence, evaluation of the subtree rooted at  need not continue.

- The value -1 generated at  is called an α-value for the root vertex. Formally:

  (a) An α-*value* for a max vertex is a known lower bound on the ultimate value of that vertex.

  (b) The *simple* α-*pruning rule* states that if the value of a min vertex is found to be less than or equal to an α-value for its parent vertex, then it is not necessary to determine its value further.

- The dual concepts are as follows

  (c) A β-*value* for a min vertex is a known upper bound on the ultimate value of that vertex.

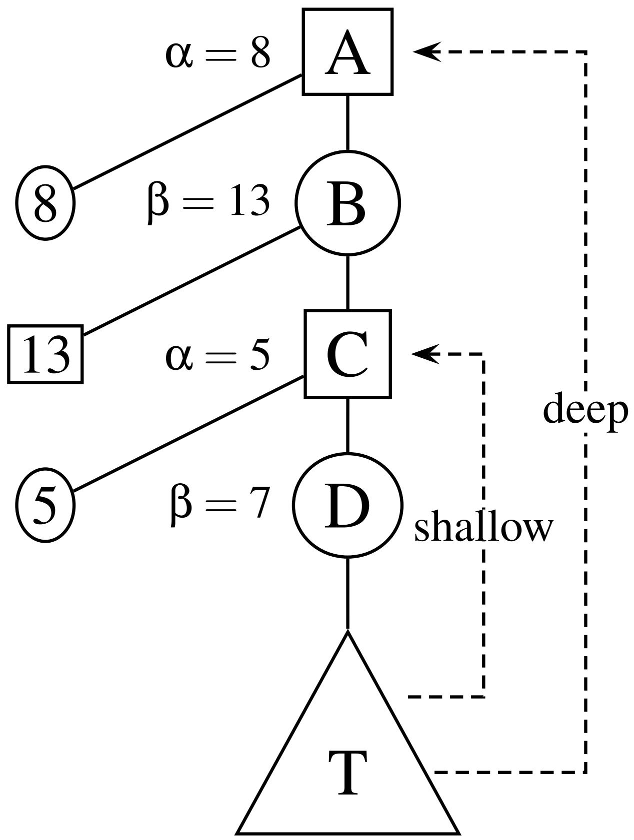
  (d) The *simple* β-*pruning rule* states that if the value of a max vertex is found to be greater than or equal to an β-value for its parent vertex, then it is not necessary to determine its value further.

- Finally, these two may be combined:

  (c) The technique of α-β-*pruning* combines these two ideas.

5.4.7 The deep α-β pruning rule

- In *deep* α-β *pruning*, α and β values of all ancestors of a vertex, rather than just those of its parent, are used to direct the pruning process.
- The idea is sketched with a simple example.



- Expansion of the subtree T is not stopped by α pruning, since $7 > 5$.
- However, 7 is a maximum value for vertex C.
- Thus, in fact, 7 is a β value for vertex B.
- Hence, the α value of the root cannot change, and so the evaluation may be halted.