15-150 Fall 2013
Homework 11

Out: Thursday, 14 November 2013
Due: Tuesday, 26 November 2013 at 23:59 EDT

1 Introduction

In this homework you will write a game and a game player using the techniques you have been learning in class. There are two major parts: writing the representation of the game, and writing a parallel version of the popular alpha-beta pruning algorithm. This will also continue to test your ability to use and understand the module system.

1.1 Getting The Homework Assignment

The starter files for the homework assignment have been distributed through our `git` repository as usual.

1.2 Submitting The Homework Assignment

Submissions will be handled through Autolab, at

`https://autolab.cs.cmu.edu`

To submit your solutions, run `make` from the `hw/11` directory (which should contain a `code` folder). This should produce a file `hw11.tar`, containing the files that should be handed in for this homework assignment. Open the Autolab web site, find the page for this assignment, and submit your `hw11.tar` file via the "Handin your work" link.

The Autolab handin script does some basic checks on your submission: making sure that the file names are correct; making sure that no files are missing; making sure that your PDF is valid; making sure that your code compiles cleanly. Note that the handin script is *not* a grading script—a timely submission that passes the handin script will be graded, but will not necessarily receive full credit. You can view the results of the handin script by clicking the number (usually either 0.0 or 1.0) corresponding to the "check" section of your latest handin on the "Handin History" page. If this number is 0.0, your submission failed the check script; if it is 1.0, it passed.

Your `graphseq.sml`, `riskless.sml`, `alphabeta.sml`, and `jamboree.sml` files must contain all the code that you want to have graded for this assignment, and must compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded.

1.3 Due Date

This assignment is due on Tuesday, 26 November 2013 at 23:59 EDT. Remember that you may use a maximum of one late day per assignment, and that you are allowed a total of three late days for the semester.

1.4 Methodology

You must use the five step methodology discussed in class for writing functions, for **every** function you write in this assignment. Recall the five step methodology:

1. In the first line of comments, write the name and type of the function.
2. In the second line of comments, specify via a `REQUIRES` clause any assumptions about the arguments passed to the function.
3. In the third line of comments, specify via an `ENSURES` clause what the function computes (what it returns).
4. Implement the function.
5. Provide testcases, generally in the format

```
val <return value> = <function> <argument value>
```

For example, for the factorial function presented in lecture:

```
(*  fact : int -> int
 *  REQUIRES:  n >= 0
 *  ENSURES: fact(n) ==> n!
*)

fun fact (0 : int) : int = 1
  | fact (n : int) : int = n * fact(n-1)

(* Tests: *)

val 1 = fact 0
val 720 = fact 6
```

2 Views

2.1 Introduction to Views

As we have discussed many times, lists operations have bad parallel complexity, but the corresponding sequence operations are much better.

However, sometimes you want to write a **sequential** algorithm (e.g. because the inputs aren't very big, or because no good parallel algorithms are known for the problem). Given the sequence interface so far, it is difficult to decompose a sequence as "either empty, or a cons with a head and a tail." To implement this using the sequence operations we have provided, you have to write code that would lose style points, such as:

```
case Seq.length s of
     0 =>
   | _ => ... uses (Seq.hd s) and (Seq.tl s) ...
```

We can solve this problem using a *view*. This means we put an appropriate datatype in the signature, along with functions converting sequences to and from this datatype. This allows us to pattern-match on an abstract type, while keeping the actual representation abstract. For this assignment, we have extended the `SEQUENCE` signature with the following components to enable viewing a sequence like a list:

```
datatype 'a lview = Nil | Cons of 'a * 'a seq
val showl : 'a seq -> 'a lview
val hidel : 'a lview -> 'a seq
(* invariant: showl (hidel v) ==> v *)
```

Because the datatype definition is in the signature, the constructors `Nil` and `Cons` can be used outside the abstraction boundary. The `showl` and `hidel` functions convert between sequences and list views. The following is an example of using this view to perform list-like pattern matching:

```
case Seq.showl s of
    Seq.Nil => ... (* Nil case *)
  | Seq.Cons (x, s') => ... uses x and s' ... (* Cons case *)
```

Note that the second argument to `Cons` is another `'a seq`, *not* an `lview`. Thus, `showl` lets you do one level of pattern matching at a time: you can write patterns like `Seq.Cons(x,xs)` but not `Seq.Cons(x,Seq.Nil)` (to match a sequence with exactly one element). We have also provided `hidel`, which converts a view back to a sequence—`Seq.hidel (Seq.Cons(x,xs))` is equivalent to `Seq.cons(x,xs)` and `Seq.hidel Seq.Nil` is equivalent to `Seq.empty()`.

Task 2.1 (2%). In `graphseq.sml`, write a function:

```
to_list : 'a Seq.seq -> 'a list
```

that uses listviews to convert a sequence into a list. Your algorithm should run in $O(n)$ work and span.

3 Graphs

3.1 Introduction to Graphs

Graphs are a powerful data structure used to represent relationships between pairs of items. Many real-world problems can be reduced to graphs; social networks, links between documents, and transportation routes can all be represented by graphs.

On this homework, we will be using graphs to represent territories and the paths between them. Our maps will be represented as **simple**, **undirected**, and **connected** graphs.

Consider the picture below, which we can think of as representing the routes of a relatively dysfunctional airline. In this picture, the *vertices* (or *nodes*) in this graph represent cities, and the *edges* between the vertices represent flight paths between these cities.

We will start by numbering the vertices from 0 to $n-1$, where n is the number of vertices in the graph.

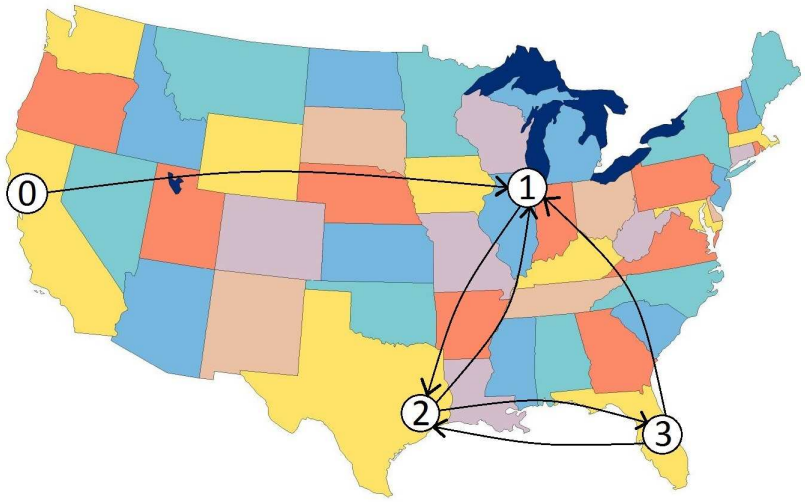


Figure 1: Example graph of flight paths in the USA

We call a connection between two vertices an **edge**. We will represent an edge as a pair of vertices; the first vertex in the pair represents the source, and the second vertex in the pair represents the destination. For given vertices u and v, (u, v) represents a *directed* edge from u to v.

We will be representing our undirected graphs as directed graphs, such that $\forall$ edges (u, v), there is also an edge (v, u). For example, in the figure above, the graph is undirected if for every flight path from u to v, there exists a flight path from v to u.

Since we know the nodes are numbered 0 through $n-1$, the graph could be completely described by a sequence of edges, and the number of vertices in the graph.

3.2 Adjacency List Representation of Graphs

In this assignment, we will be representing graphs as **adjacency lists**; that is, for each vertex, we maintain a list of all of the vertices that our vertex is adjacent to.

For the purposes of this section, we will represent the graph using the following type:

```
type vertex = int
type graph = vertex seq seq
```

In this representation, each index in the outer sequence of some graph `G` corresponds to the source vertex, and the *vertex sequence* at that index represents the list of destination vertices. Finally, we know that the number of vertices in the graph is `length G`.

The following is an example of the graph in Figure 1, represented as an adjacency list.

```
The graph in Figure 1, as a mapping from source to destination:
       0 -> 1
       1 -> 2
       2 -> 1, 3
       3 -> 1, 2
The graph in Figure 1, as a vertex seq seq:
<<1>, <2>, <1, 3>, <1, 2>>
```

3.3 Graph Warmup

For the purposes of this assignment, assume all graphs are simple and connected. "Simple" means that for all vertices u and v (where $u \neq v$), there is at most one edge (u, v). In addition, for any vertex u, there is no edge (u, u) (or self-loop), where a single vertex is both the source and the destination.

In an undirected graph, "connected" means that there is a path from each vertex in the graph to every other vertex in the graph. We will consider a directed graph "weakly connected" if, when we replace every directed edge with an undirected edge, the resulting graph is connected.

In the context of this assignment, by guaranteeing connectedness, we are ensuring that there are no unreachable, unconquerable "islands" in our graph.

Task 3.1 (3%). In `graphseq.sml`, write a function

```
get_neighbors: vertex seq seq -> vertex -> vertex seq
```

such that `get_neighbors G source` evaluates to a sequence containing only the neighbors of vertex `source` in an undirected, connected graph `G` (where a neighbor is any vertex in an edge with `source`). If the requested vertex is not in the graph, raise **not_in_graph**.

For example, `get_neighbors <<1>, <0,3>, <3>, <1,2>> 1` should evaluate to `<0,3>`.

Task 3.2 (5%). Write a function

```
from_edge_seq : (vertex * vertex) seq -> int -> vertex seq seq
```

where `from_edge_seq E n`, given the n, number of vertices in the graph, and E, a list of valid edges for the graph of n vertices, constructs the adjacency list representation of the graph. Note that your algorithm must have $O(|E|^2)$ work, and $O(|E|)$ span (but it can be faster).

For example, `from_edge_seq <(0,1), (1,0), (1,2), (2,1)> 3` should evaluate to `<<1>, <0,2>, <1>>`.

Task 3.3 (5%). Write a function

```
is_undirected : vertex seq seq -> bool
```

that returns true if all edges in a graph `G` are undirected (i.e. for each edge (u, v) in `G`, (v, u) is also in `G`), and false otherwise.

3.4 Weighted Nodes

Suppose that in addition to edges, you want to store information on each vertex. To do so, we consider an implementation using *weighted nodes*:

```
type weight = 'a
type graph = (weight * vertex seq) seq
```

In this representation, we store both the "weight" of each vertex `i` and its neighbors (adjacency sequence) at the $\texttt{i}^{th}$ index. As you will see in the next section, a weight can represent anything from an integer label to a datatype storing data about the vertex.

4 Riskless

4.1 Rules and Strategy

4.1.1 Description

Riskless is a strategic game of military might and territorial domination.[1] Two players–black and white, or Maxie and Minnie–take turns mobilizing armies to empty or occupied territories. The first player to mobilize at least X armies (for some $X \in \mathbb{N}$ determined by the map), or the last player standing, wins.

In our implementation, we will model the game board (a political map of Earth) as a simple, connected, undirected graph.[2] Each node will represent a territory (empty or occupied), and each edge a road connecting two territories. The players begin with their armies distributed on the occupied territories (Minnie with one extra army), and gameplay commences turn-by-turn (Maxie going first):

1. On a player's turn, he or she mobilizes all the armies on any occupied territory to an adjacent territory.
   (a) The (now vacated) source territory remains occupied by the same player with 0 units.
   (b) The target territory updates as follows:
      i. If the target is empty, the player occupies the territory, and all armies on the source are moved to the target.
      ii. If the target is occupied by the same player, all armies on the source are moved (added) to the target territory.
      iii. If the target is occupied by the opponent, the player with more armies wins the battle and reoccupies the territory with the (absolute) difference of armies remaining. In the case of a tie, the attacking player wins.
   (c) A player cannot abstain from moving.
2. At the end of every move, both players train one additional army on each of their occupied territories up to a set cap (determined by the map). Any territories that exceed the cap are reduced to the limit.

See Figure 2 below for an example game on a graph of four vertices; an arrow represents a move, and "x" represents a battle, as mentioned above. Each graph shows the state after a move and after each territory has trained an additional unit.

[1] A deterministic variation of its namesake Risk, the game has been simplified for ease of coding. More information on the board game can be found at http://en.wikipedia.org/wiki/Risk_(game)

[2] Remember that a simple graph has no self-loops and at most one edge between any two nodes.

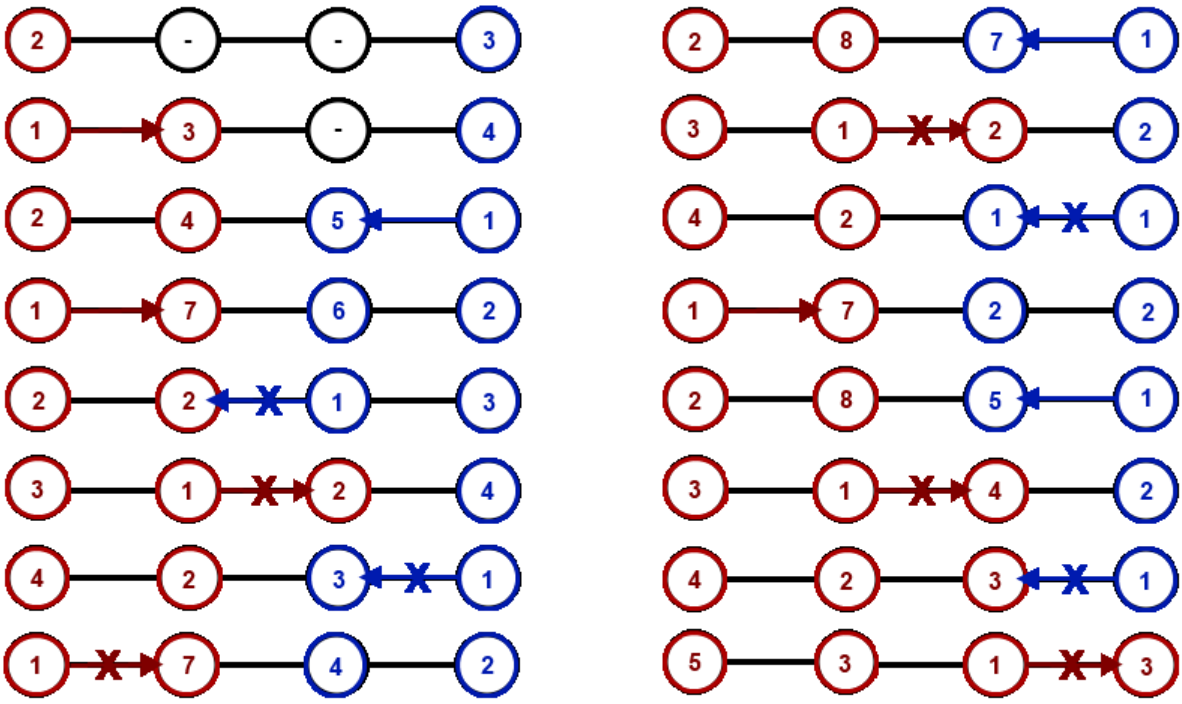


Figure 2: Example game of Riskless; red wins!

4.1.2 Strategy

There are a number of useful strategies to ensure success on your journey to world domination. A good tactician should consider, among other factors, the following:

1. The more territories a player controls, the more armies the player can train in one turn.

2. Players should watch borders (nodes adjacent to opposing territories) for a build up of armies: this could signal a potential attack!

3. Players should amass armies on their borders for better defense.

4.1.3 Evaluation

To implement Riskless for a computer player, we need to evaluate different game states. A very naïve static evaluation algorithm would be to award points for the total number of armies on the map. For instance, one could sum the number of armies, with positive points for the "Max" player and negative points for the "Min" player.

More sophisticated static evaluations might take other factors into account, such as:

- the number of territories the player has conquered

- the importance of amassing larger armies (as opposed to stretching out one's troops)
- the value of defending borders
- the value of safe territories (nodes surrounded by the player's own territories)

4.2 Implementation

Part of an implementation of the `GAME` signature for Riskless is provided in `riskless.sml`. It uses the following abstract representation of game states:

```
datatype position = Min of int | Max of int | Empty
datatype territory = T of string * position
type vertex = int
datatype gstate = B of (territory * vertex seq) seq * player
type state = gstate
type move = vertex * vertex
```

Here, we have provided the `Sequence` library. We represent the game board as a weighted graph, where each vertex in the sequence contains a territory and its adjacency sequence. Each territory has a label (name) and a position, indicating whether the node is occupied and how many armies are currently occupying the territory. Each game state is represented by a board, as well as the current player making the move. Finally, a move is a tuple of the source and destination vertex.

Recall the `GAME` signature we discussed in class. We have extended it to include the following:

```
datatype player = Minnie | Maxie
datatype outcome = Winner of player | Draw
datatype status = Over of outcome | In_play
```

Each game state will store both the current player and have a status, represented as `Over` with an outcome or `In_play`.

`structure Est : EST`

The `EST` signature contains estimated values and functions, notably `minnie_wins` and `maxie_wins`. See `estimate.sig` for more information.

`val status : state -> status`

Given a game state `s`, `status s` evaluates to the current status of the game. Remember that a game of Riskless ends when either one player is out of territories, or the other has reached the target number of units, represented by `Dim.max_units`. A `Draw` state occurs when both players have exceeded the target number of units.

`val moves : state -> move Seq.seq`

If a state `s` is still `In_play`, `moves s` evaluates to a non-empty sequence of valid moves for `s`. Otherwise, `moves s` returns an empty sequence.

`val player : state -> player`

Returns the current player making the move in the current state.

`val make_move : (state * move) -> state`

Assuming a valid move for the current state, evaluates to the state after the move has been applied. Remember to update the number of armies, and to account for the maximum cap per territory (defined by `Dim.max_terri`).

`val estimate : state -> Est.est`

Returns an estimated score for a given state. You will receive full credit as long as `estimate` performs no worse than the naïve version above. Of course, a more refined evaluation algorithm will yield a more competent and challenging computer opponent! Your implementation of estimate should work on all valid game states, whether it is in play or it is a terminal state.

Please describe your implementation and the strategies you considered in a few sentences in a comment.

Task 4.1 (40%). Complete the implementation of the `Riskless` functor in `riskless.sml`. This takes a structure ascribing to the `MAP` signature specifying the layout of the board.

4.3 Running Your Game

The file `runriskless.sml` uses the `Referee` to construct a Riskless game, with Maxie as a human player and Minnie as Minimax. Thus, you can run your game with:

```
- CM.make "sources.cm";
- Risk_HvMM.go();
```

You can write other referee applications to test different variations on the game: different graph sizes, different search depths, different game tree search algorithms (such as Jamboree below...).

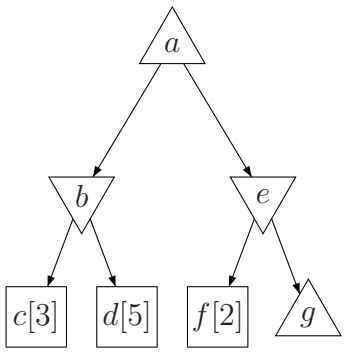
4.4 Testing Your Game

Task 4.2 (5%). The test structure `TestRiskless` has been provided in `riskless.sml`. Please fill out the test structure with appropriate tests.

5 Alpha-Beta Pruning

In lecture, we implemented the minimax game tree search algorithm: each player chooses the move that, assuming optimal play of both players, gives them the best possible score (highest for Maxie, lowest for Minnie). This results in a relatively simple recursive algorithm, which searches the entire game tree up to a fixed depth. However, it is possible to do better than this!

Consider the following game tree:



Maxie nodes are drawn as an upward-pointing triangle; Minnie nodes are drawn as a downward-pointing triangle. Each node is labelled with a letter, for reference below. The leaves, drawn as squares, are Maxie nodes and are also labelled with their values (e.g. given by the estimator).

Let's search for the best move from left to right, starting from the root a. If Maxie takes the left move to b, then Maxie can achieve a value of 3 (because Minnie has only two choices, node c with value 3 and node d with value 5). If Maxie takes the right move to e, then Minnie can take the her left move to f, yielding a value of 2. But this is worse than what Maxie can already achieve by going left at the top! So Maxie already knows to go left to b rather than right to e. So, there is no reason to explore the tree g (which might be a big tree and take a while to explore) to find out what Minnie's value for e would actually be: we already know that the value, whatever it is, will be less than or equal to 2, and this is enough for Maxie not to take this path.

$\alpha\beta$-pruning is an improved search algorithm based on this observation. In $\alpha\beta$-pruning, the tree g is *pruned* (not explored). This lets you explore more of the relevant parts of the game tree in the same amount of time.

5.1 Setup

In $\alpha\beta$-pruning, we keep track of two values, representing the best (that is, highest for Maxie, and lowest for Minnie) scores that can be guaranteed for each player based on the parts of the tree that have been explored so far. α is the highest guaranteed score for Maxie, and β is the lowest guaranteed score for Minnie.

For each node, $\alpha\beta$-pruning computes a *candidate value* which represents the best score (in particular, `Game.est`imate) of that node. The values are labeled according to the following spec:

> **Spec for $\alpha\beta$-pruning:** Fix a search depth and an estimation function, so that both minimax and $\alpha\beta$-pruning are exploring a tree with the same leaves. Fix bounds α and β such that such that $\alpha < \beta$. Let MM be the minimax value of a node s. Then the $\alpha\beta$-pruning result (i.e., the candidate value) for that node, AB, satisfies the following:
>
> - If s is a Maxie node and $MM \leq \alpha$, then stop evaluating any siblings (the parent of s should be pruned).
> - If s is a Minnie node and $\beta \leq MM$, then stop evaluating any siblings (the parent of s should be pruned).
> - Otherwise, $AB = MM$.

In other words, α is what we know Maxie can achieve, and β is what we know Minnie can achieve. If the true minimax value of a node is between these bounds, then $\alpha\beta$-pruning computes that value. If it is outside these bounds, then $\alpha\beta$-pruning signals this in one of two ways, depending on which side the actual value falls on, and whose turn it is. Suppose that it's Maxie's turn:

> If the actual minimax value is less than α, the subtrees rooted at this node's parent can be pruned. The reason: this Maxie node's value is worse for Maxie than what we already know Maxie can achieve, so it gives the enclosing Minnie node an option that Maxie doesn't want it to have. So the Maxie grand-parent should ignore this branch, independently of what the other siblings are. Node f in the above tree is an example.

A symmetric argument can be applied to Minnie.

Sometimes, no bounds on α and β are known (e.g., at the initial call, when you have not yet explored any of the tree). To account for this, we typically set the initial α and β to the smallest and largest possible estimated values, such that any minimax score is equal or better for either player. This can be modeled with $-\infty$ and $+\infty$, or in our implementation, `Game.Est.minnie_wins` and `Game.Est.maxie_wins`, the estimated values for Minnie's victory and Maxie's victory. In other words, we assume the worst, and only update the bounds when we find a state with a better score.

5.2 The Algorithm

The above spec doesn't entirely determine the algorithm (a trivial algorithm would be to run minimax and then adjust the label at the end). The extra ingredient is how the values of α and β are adjusted as the algorithm explores the tree. The key idea here is *horizontal propagation.*

Consider the case where we want to compute the value of a *parent* node (so it is not a leaf), given search bounds α and β based on the portion of the tree seen so far.

- To calculate the result for the parent node, you scan across the children of the node, recursively calculating the result of each child from left to right, using the $\alpha\beta$ for the parent as the initial $\alpha\beta$ for the first child. Suppose that after considering a child, we find it has the estimated value v:

  - Update the appropriate bounds depending the player. If the parent node is a Maxie node, update $\alpha_{new} = \max(\alpha_{old}, v)$ (since the new value can only be better). Dually, if the parent is a Minnie node, update $\beta_{new} = \min(\beta_{old}, v)$.
  - Next, we check if the new values are within the bounds.
    * If $\alpha_{new} \geq \beta$ for Maxie or $\beta_{new} \leq \alpha$ for Minnie, we have found a value that will not be considered (since the opposing player will simply choose the tighter value). In this case, we prune the subtree by ignoring the remaining children.
    * Otherwise, we continue processing the remaining children using the updated bounds.

  Note : Most labs presented checking the bounds, and then doing the update. This is equivalent to doing the update, and then checking the bounds. The first is a little easier to explain why it is correct, but the latter results in slightly cleaner code.

- Once all children have been processed, the parent node is labeled using the final updated α (for Maxie) or β (for Minnie) as a *candidate value*. Realize that this value is equivalent to the minimax value even if we were to have processed the entire subtree.

  Note that, except via the returned value of a node, the updates to α and β made in children do not affect the α and β for the parent node—they are based on different information.

Once we reach a terminal node, we label it using the estimated or actual value as a candidate value. This base case can be reached in two ways: either we have reached a leaf node with no children, or we have reached the maximum search depth ($d = 0$).

5.3 Extended Example

Figure 4 shows a trace of this algorithm running on the tree in Figure 3. As explained above, we use the values `minnie_wins` and `maxie_wins` in the initial call to represent "no bound for α/β." The $c - d - g$ subtree is equivalent to the above example tree, and shows how the algorithm runs on it.

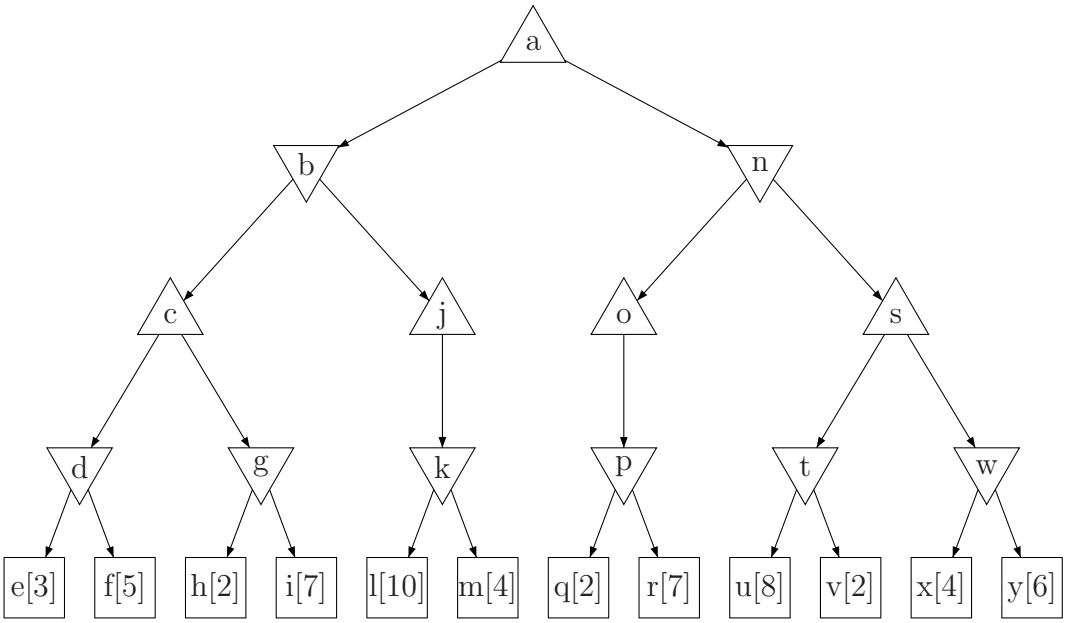


Figure 3: Extended $\alpha\beta$-pruning example

We start evaluating c with no bounds, so we recursively look at d, and then e, still with no bounds. The estimate for e is 3, which is trivially within the bounds, so the result of e is 3. Because d is a Minnie node, we update β to be min(`maxie_wins`,3), which is defined to be 3, for a recursive call on f. Since the estimate of f is 5, which is greater than the given β (which is 3), and it is a Maxie node, we take the min(3, 5) and update d to 3. Since c is a Maxie node, we update α to be max(`minnie_wins`,3), which is also 3. These bounds are passed down to g and then h. Since the actual value of h is $2 < \alpha$, and it is a Maxie node, we prune the remaining children: we know Maxie can get 3 in the left tree, and this branch alone gives Minnie the ability to get 2 here, so Maxie doesn't want to take it. Thus, we simply set g to $\beta = 3$, *without even looking at* i. Then we update α to be $min(3, 3)$, which is still 3. Since there are no other children, and since 3 is in the original bounds for c, it is the value of c.

```
F: state (Maxie,a) with ab=(Leaf(Minnie wins!),Leaf(Maxie wins!))
G: state (Minnie,b) with ab=(Leaf(Minnie wins!),Leaf(Maxie wins!))
F: state (Maxie,c) with ab=(Leaf(Minnie wins!),Leaf(Maxie wins!))
G: state (Minnie,d) with ab=(Leaf(Minnie wins!),Leaf(Maxie wins!))
F: state (Maxie,e) with ab=(Leaf(Minnie wins!),Leaf(Maxie wins!))
Estimating state e[3]
Result of (Maxie,e) is Leaf(Guess: 3)
F: state (Maxie,f) with ab=(Leaf(Minnie wins!),Step(0, Guess: 3))
Estimating state f[5]
Result of (Maxie,f) is Leaf(Guess: 5)
Result of (Minnie,d) is Step(0, Guess: 3)
G: state (Minnie,g) with ab=(Step(0, Guess: 3),Leaf(Maxie wins!))
F: state (Maxie,h) with ab=(Step(0, Guess: 3),Leaf(Maxie wins!))
Estimating state h[2]
Result of (Maxie,h) is Leaf(Guess: 2)
Result of (Minnie,g) is Step(0, Guess: 2) (* Pruned *)
Result of (Maxie,c) is Step(0, Guess: 3)
F: state (Maxie,j) with ab=(Leaf(Minnie wins!),Step(0, Guess: 3))
G: state (Minnie,k) with ab=(Leaf(Minnie wins!),Step(0, Guess: 3))
F: state (Maxie,l) with ab=(Leaf(Minnie wins!),Step(0, Guess: 3))
Estimating state l[10]
Result of (Maxie,l) is Leaf(Guess: 10)
F: state (Maxie,m) with ab=(Leaf(Minnie wins!),Step(0, Guess: 3))
Estimating state m[4]
Result of (Maxie,m) is Leaf(Guess: 4)
Result of (Minnie,k) is Step(0, Guess: 3)
Result of (Maxie,j) is Step(0, Guess: 3) (* Pruned *)
Result of (Minnie,b) is Step(0, Guess: 3)
G: state (Minnie,n) with ab=(Step(0, Guess: 3),Leaf(Maxie wins!))
F: state (Maxie,o) with ab=(Step(0, Guess: 3),Leaf(Maxie wins!))
G: state (Minnie,p) with ab=(Step(0, Guess: 3),Leaf(Maxie wins!))
F: state (Maxie,q) with ab=(Step(0, Guess: 3),Leaf(Maxie wins!))
Estimating state q[2]
Result of (Maxie,q) is Leaf(Guess: 2)
Result of (Minnie,p) is Step(0, Guess: 2) (* Pruned *)
Result of (Maxie,o) is Step(0, Guess: 3)
Result of (Minnie,n) is Step(0, Guess: 3) (* Pruned *)
Result of (Maxie,a) is Step(0, Guess: 3)

Therefore value of (Maxie,a) is Guess: 3, and best move is 0.
Terminals visited: 6
```

Figure 4: AB-Pruning trace for the above tree

5.4 Tasks

We have provided starter code in `alphabeta.sml`. As in minimax, we need to return not just the value of a node, but the move that achieves that value, so that at the top we can select the best move:

```
type edge = Game.move * Game.est
```

While it would be simple to let our $\alpha\beta$ `type value = edge`, we must remember to consider the special cases when a node is already at a terminal state (i.e. a leaf node, or one at search depth = 0). To account for these, we represent a terminal node as an `Leaf`, and all others as a `Step`:

```
datatype value =
    Step of edge
  | Leaf of Game.est
```

This way, we can represent α and β by a `value` consisting of the current best estimated candidate value, and the current best move (if there is one).

Note that `value`s are compared via their candidate values, where `minnie_wins` and `maxie_wins` represents the smallest and greatest values. The presence of a move does not affect the ordering, except in the case of equal values, where we must choose a `Step` over an `Leaf` to ensure we pass back a valid move.

We have provided three functions:

```
valuecmp : value * value -> order
maxalpha : value * value -> value
minbeta : value * value -> value
```

that will be helpful when implementing these comparisons.

To extract the candidate value or move from α or β, you may use

```
valueOf : value -> Game.est
moveOf : value -> Game.move
```

Finally, we abbreviate

```
(* invariant: (alpha, beta) with alpha < beta *)
type alphabeta = value * value
```

Task 5.1 (1%). Fill in the value

```
val initialAB : alphabeta = ...
```

for the initial α/β bounds to pass in to your implementation.

Task 5.2 (20%). Define the mutually recursive functions:

```
fun F (d : int) (s : Game.state) (ab : alphabeta) : value = ...
and G (d : int) (s : Game.state) (ab : alphabeta) : value = ...
```

which compute the best possible result (move and candidate value) using the $\alpha\beta$ algorithm, given a game state `s` at a depth `d`, for Maxie (`F`) and Minnie (`G`).

You may assume that the depth is non-zero, the status of `s` is `In_play`, and that `Game.moves s` evaluates to a non-empty sequence of valid moves.

Hint 1: Recall `Seq.showl`, which takes an `'a Seq.seq` and evaluates to a list view (`Seq.Nil` or `Seq.Cons('a * 'a Seq.seq)`). This and a recursive helper function may be useful while sequentially analyzing a parent node's children. You may also find the utility function `SeqUtils.null` helpful.

Hint 2: Remember to update the move after processing each child node! If your code is evaluating to invalid moves, or returning only `Leaf`s, check for this bug.

Task 5.3 (4%). Define `next_move : Game.state -> Game.move`, such that given a state `s`, `next_move s` evaluates to the best move the current player can choose.

Task 5.4 (0%). In `runriskless.sml`, use the `Referee` to define a structure `Risk_HvAB` that allows you to play Riskless against your $\alpha\beta$-pruning implementation.

For fun, you can also try pitting MiniMax and AlphaBeta against each other. A search depth of 4-6 should run relatively quickly.

Testing We have provided a functor `ExplicitGame` that makes a game from a given game tree, along with two explicit games, `HandoutSmall` (Figure 5) and `HandoutBig` (Figure 4). These can be used for testing as follows:

```
- structure TestBig =
    AlphaBeta(struct structure G = HandoutBig val search_depth = 4 end);
- TestBig.next_move HandoutBig.start;
Estimating state e[3]
Estimating state f[5]
Estimating state h[2]
Estimating state l[10]
Estimating state m[4]
Estimating state q[2]
val it = 0 : HandoutBig.move

structure TestSmall =
    AlphaBeta(struct structure G = HandoutSmall val search_depth = 2 end);
- TestSmall.next_move HandoutSmall.start;
Estimating state c[3]
Estimating state d[6]
```

```
Estimating state e[~2]
Estimating state g[6]
Estimating state h[4]
Estimating state i[10]
Estimating state k[1]
val it = 1 : HandoutSmall.move
```

The search depths of `2` and `4` here are important, because an explicit game causes errors if it tries to estimate in the wrong place.

For these explicit games, `estimate` prints the states it visits, so you can see what terminals are visited, as indicated above. You may additionally wish to annotate your code so that it prints out traces, as above.

6 Jamboree

$\alpha\beta$-pruning is entirely sequential, because you update α/β as you search across the children of a node, which creates a dependency between children. On the other hand, MiniMax is entirely parallel: you can evaluate each child in parallel, because there are no dependencies between them. This is an example of a *work-span tradeoff*: MiniMax does more work, but has a better span, whereas $\alpha\beta$-pruning does less work, but has a worse span.

The *Jamboree*[3] algorithm manages this tradeoff by evaluating *some* of the children sequentially, updating $\alpha\beta$, and then the remainder in parallel, using the updated information. Depending on the parallelism available in your execution environment, you can choose how many children to evaluate sequentially to prioritize work or span.

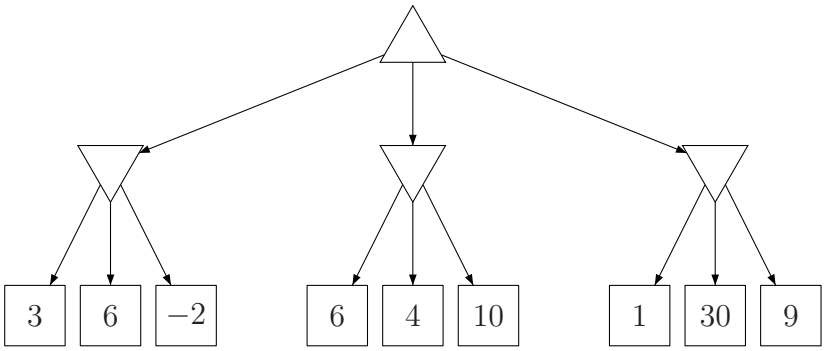
In the file `jamboree.sml`, you will implement a functor

```
functor Jamboree (Settings : sig
                               structure G : GAME
                               val search_depth : int
                               val prune_percentage : real
                             end) : PLAYER
```

`prune_percentage` is assumed to be a number between 0 and 1. For each node, `100.0 * prune_percentage` percent of the children are evaluated sequentially, updating $\alpha\beta$, and the remaining children are evaluated in parallel. For example, with `prune_percentage` $= 0$, Jamboree performs just like Minimax, and with `prune_percentage` $= 1$, Jamboree performs like completely sequential $\alpha\beta$-pruning.

Suppose `prune_percentage` is `0.5`. For the tree in Figure 3, Jamboree will explore the tree in the same way as in Figure 4: no node has more than 2 children, so the restriction on pruning never comes into play. However, on the following tree, the traces differ:

[3] A parallelized version of the original Scout $\alpha\beta$ algorithm in 1980, Jamboree is so named for running multiple "scouts" in parallel.



See Figure 5 for the traces.

6.1 Tasks

Task 6.1 Copy `initialAB` from your $\alpha\beta$-pruning implementation, as this will be unchanged.

Task 6.2 (3%). Define the function `splitMoves : Game.state -> (Game.move seq * Game.move seq)` which, given a game state, splits the possible moves into two sequences `(abmoves, mmmoves)` depending on the `prune_percentage`. In particular, if `s` has `n` moves, `abmoves` should contain the first `(Int.floor (prune_percentage * n))` moves, and `mmmoves` whatever is left over.

Task 6.3 (10%). Define the functions:

```
fun F (d : int) (s : Game.state) (ab : alphabeta) : value = ...
and G (d : int) (s : Game.state) (ab : alphabeta) : value = ...
```

using the Jamboree algorithm.

The specs for `F` and `G` are as above, except that during the processing of each parent's child nodes, it should divide up the `moves` from `s` into `abmoves` and `mmmoves`.

The algorithm should process `abmoves` sequentially, updating α/β as you go, as in your $\alpha\beta$-pruning implementation. When there are no more `abmoves`, you should then process `mmmoves` in parallel, and combine the max/min of their values with the incoming α/β to produce the appropriate value for `s`.

(Again, you may find the helper functions in `SeqUtils` useful.)

Task 6.4 (2%). Define `next_move`.

Jamboree with `prune_percentage` = 0.5, so $\alpha\beta$ are updated only after the first child (we round down):

```
F: state (Maxie,a) with ab=(Leaf(Minnie wins!),Leaf(Maxie wins!))
G: state (Minnie,b) with ab=(Leaf(Minnie wins!),Leaf(Maxie wins!))
F: state (Maxie,c) with ab=(Leaf(Minnie wins!),Leaf(Maxie wins!))
Result of (Maxie,c) is Leaf(Guess: 3)
F: state (Maxie,d) with ab=(Leaf(Minnie wins!),Step(0, Guess: 3))
Result of (Maxie,d) is Leaf(Guess: 6)
F: state (Maxie,e) with ab=(Leaf(Minnie wins!),Step(0, Guess: 3))
Result of (Maxie,e) is Leaf(Guess: ~2)
Result of (Minnie,b) is Step(2, Guess: ~2)
G: state (Minnie,f) with ab=(Step(0, Guess: ~2),Leaf(Maxie wins!))
F: state (Maxie,g) with ab=(Step(0, Guess: ~2),Leaf(Maxie wins!))
Result of (Maxie,g) is Leaf(Guess: 6)
F: state (Maxie,h) with ab=(Step(0, Guess: ~2),Step(0, Guess: 6))
Result of (Maxie,h) is Leaf(Guess: 4)
F: state (Maxie,i) with ab=(Step(0, Guess: ~2),Step(0, Guess: 6))
Result of (Maxie,i) is Leaf(Guess: 10)
Result of (Minnie,f) is Step(1, Guess: 4)
G: state (Minnie,j) with ab=(Step(0, Guess: ~2),Leaf(Maxie wins!))
F: state (Maxie,k) with ab=(Step(0, Guess: ~2),Leaf(Maxie wins!))
Result of (Maxie,k) is Leaf(Guess: 1)
F: state (Maxie,l) with ab=(Step(0, Guess: ~2),Step(0, Guess: 1))
Result of (Maxie,l) is Leaf(Guess: 30)
F: state (Maxie,m) with ab=(Step(0, Guess: ~2),Step(0, Guess: 1))
Result of (Maxie,m) is Leaf(Guess: 9)
Result of (Minnie,j) is Step(0, Guess: 1)
Result of (Maxie,a) is Step(1, Guess: 4)
Terminals visited: 9
Overall choice: move 1[middle]
```

$\alpha\beta$-pruning:

```
F: state (Maxie,a) with ab=(Leaf(Minnie wins!),Leaf(Maxie wins!))
G: state (Minnie,b) with ab=(Leaf(Minnie wins!),Leaf(Maxie wins!))
F: state (Maxie,c) with ab=(Leaf(Minnie wins!),Leaf(Maxie wins!))
Result of (Maxie,c) is Leaf(Guess: 3)
F: state (Maxie,d) with ab=(Leaf(Minnie wins!),Step(0, Guess: 3))
Result of (Maxie,d) is Leaf(Guess: 6)
F: state (Maxie,e) with ab=(Leaf(Minnie wins!),Step(0, Guess: 3))
Result of (Maxie,e) is Leaf(Guess: ~2)
Result of (Minnie,b) is Step(2, Guess: ~2)
G: state (Minnie,f) with ab=(Step(0, Guess: ~2),Leaf(Maxie wins!))
F: state (Maxie,g) with ab=(Step(0, Guess: ~2),Leaf(Maxie wins!))
Result of (Maxie,g) is Leaf(Guess: 6)
F: state (Maxie,h) with ab=(Step(0, Guess: ~2),Step(0, Guess: 6))
Result of (Maxie,h) is Leaf(Guess: 4)
F: state (Maxie,i) with ab=(Step(0, Guess: ~2),Step(1, Guess: 4))
Result of (Maxie,i) is Leaf(Guess: 10)
Result of (Minnie,f) is Step(1, Guess: 4)
G: state (Minnie,j) with ab=(Step(1, Guess: 4),Leaf(Maxie wins!))
F: state (Maxie,k) with ab=(Step(1, Guess: 4),Leaf(Maxie wins!))
Result of (Maxie,k) is Leaf(Guess: 1)
Result of (Minnie,j) is Step(0, Guess: 1) (* Pruned *)
Result of (Maxie,a) is Step(1, Guess: 4)
Terminals visited: 7
Overall choice: move 1[middle]
```


Figure 5: Traces for Tree 2