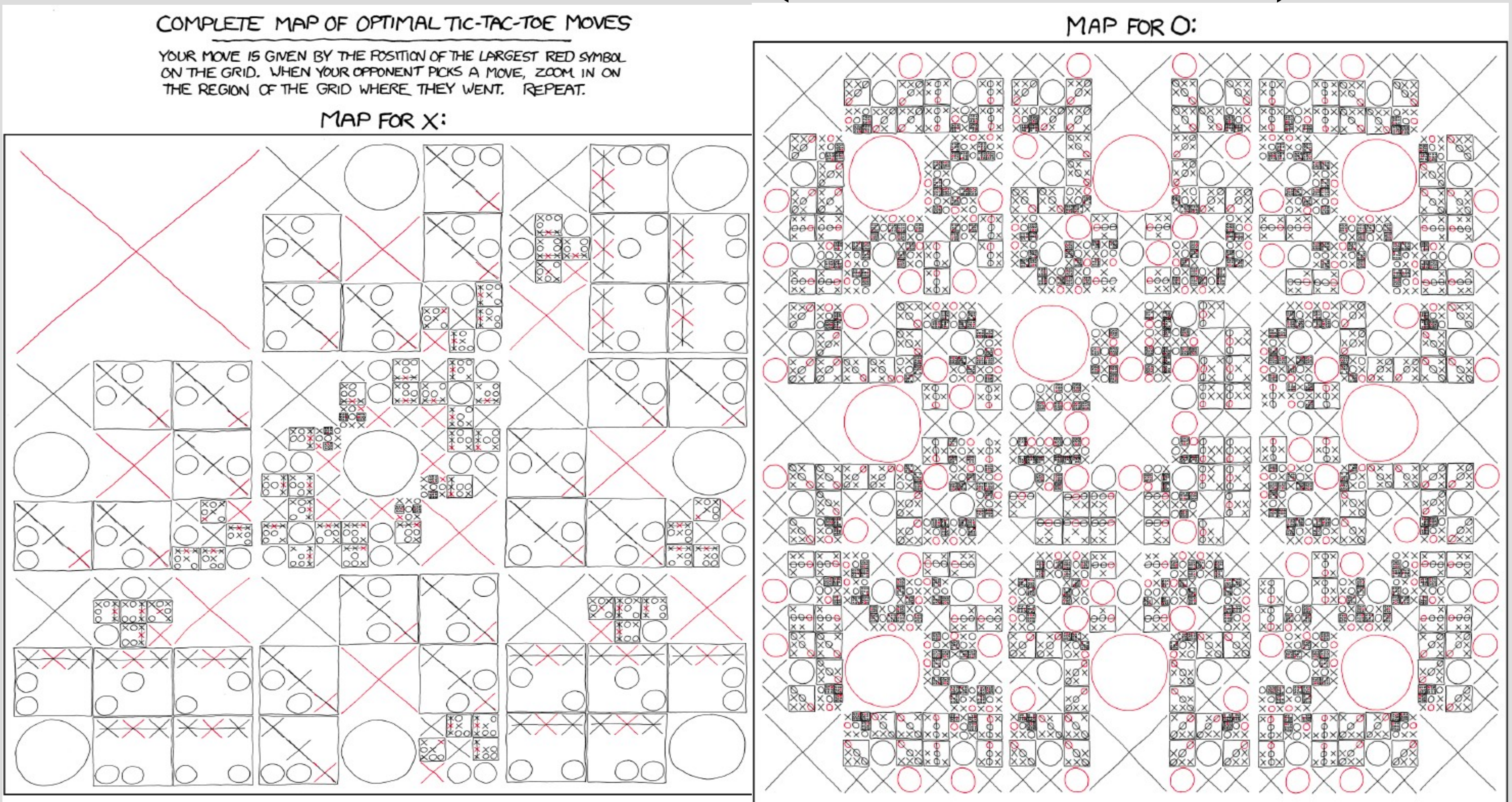
Minimax (Ch. 5-5.3)



Announcements

Homework 2 due Sunday
Midterm 1 next Tuesday

Homework 1 solutions posted
Previous midterm posted

Single-agent

So far we have look at how a single agent can search the environment based on its actions

Now we will extend this to cases where you are not the only one changing the state (i.e. multi-agent)

The first thing we have to do is figure out how to represent these types of problems

Multi-agent (competitive)

Most games only have a utility (or value) associated with the end of the game (leaf node)

So instead of having a “goal” state (with possibly infinite actions), we will assume:

(1) All actions eventually lead to terminal state
(i.e. a leaf in the tree)

(2) We know the value (utility) only at leaves

Multi-agent (competitive)

For now we will focus on zero-sum two-player games, which means a loss for one person is a gain for another

Betting is a good example of this: If I win I get $5 (from you), if you win you get $1 (from me). My gain corresponds to your loss

Zero-sum does not technically need to add to zero, just that the sum of scores is constant

Multi-agent (competitive)

Zero sum games mean rather than representing outcomes as:
[Me=5, You =-5]

We can represent it with a single number:
[Me=5], as we know: Me+You = 0 (or some c)

This lets us write a single outcome which "Me" wants to maximize and "You" wants to minimize

Minimax

Thus the root (our agent) will start with a maximizing node, the the opponent will get minimizing noes, then back to max... repeat...

This alternation of maximums and minimums is called minimax

I will use $\triangle$ to denote nodes that try to maximize and $\bigtriangledown$ for minimizing nodes

Minimax

Let's say you are treating a friend to lunch.
You choose either: Shuang Cheng or Afro Deli

The friend always orders the most inexpensive item, you want to treat your friend to best food

Which restaurant should you go to?
Menus:
Shuang Cheng: Fried Rice=$10.25, Lo Mein=$8.55
Afro Deli: Cheeseburger=$6.25, Wrap=$8.74

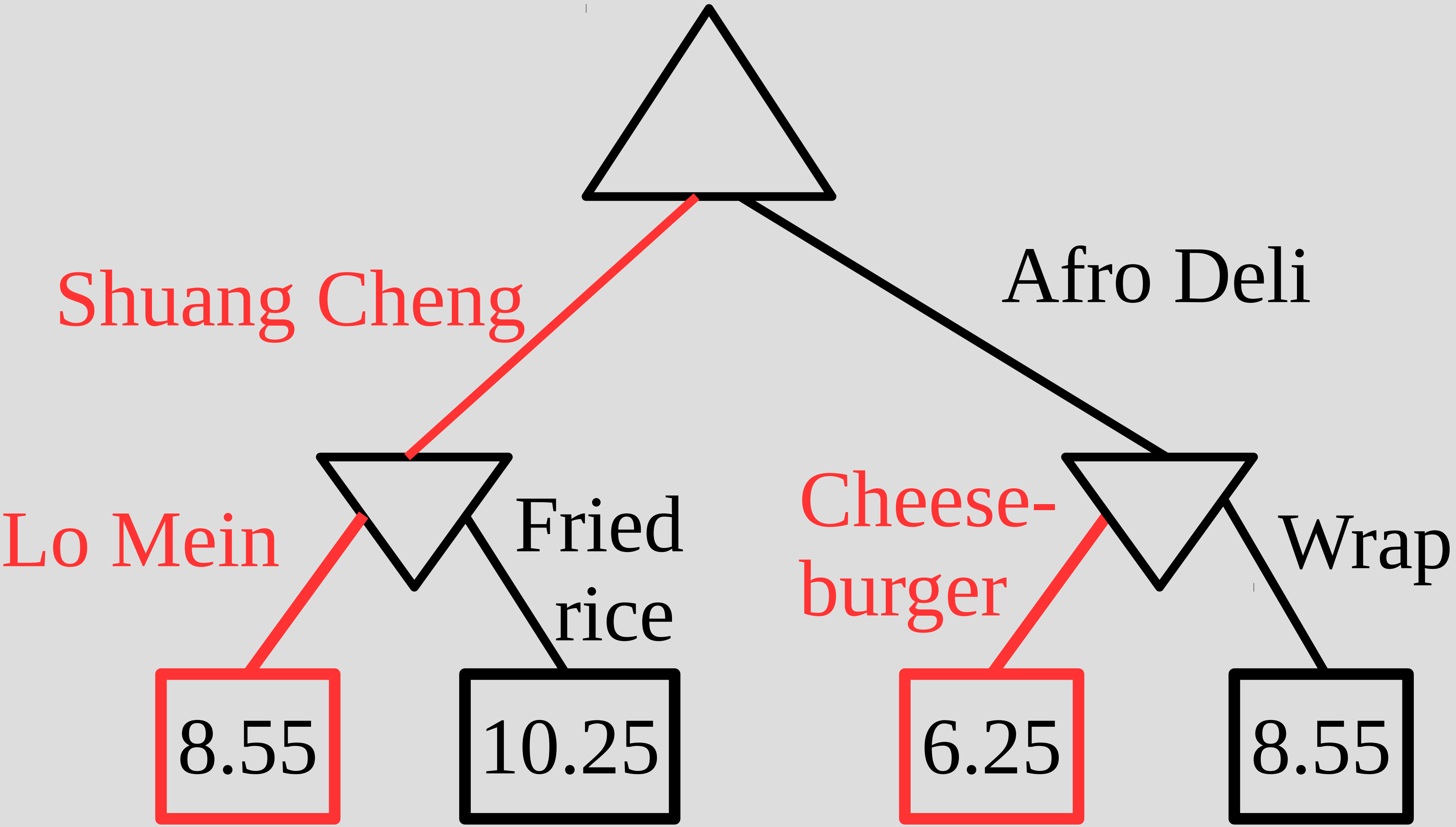
Minimax

Let's say you are treating a friend to lunch.
You choose either: Shuang Cheng or Afro Deli

The friend always orders the most inexpensive item, you want to treat your friend to best food

Which restaurant should you go to?
Menus:
Shuang Cheng: Fried Rice=$10.25, Lo Mein=$8.55
Afro Deli: Cheeseburger=$6.25, Wrap=$8.74

Minimax


Shuang Cheng
Afro Deli
Lo Mein
Fried rice
Cheese-burger
Wrap
8.55
10.25
6.25
8.55

Minimax

You could phrase this problem as a set of maximum and minimums as:

max(min(8.55, 10.25), min(6.25, 8.55))

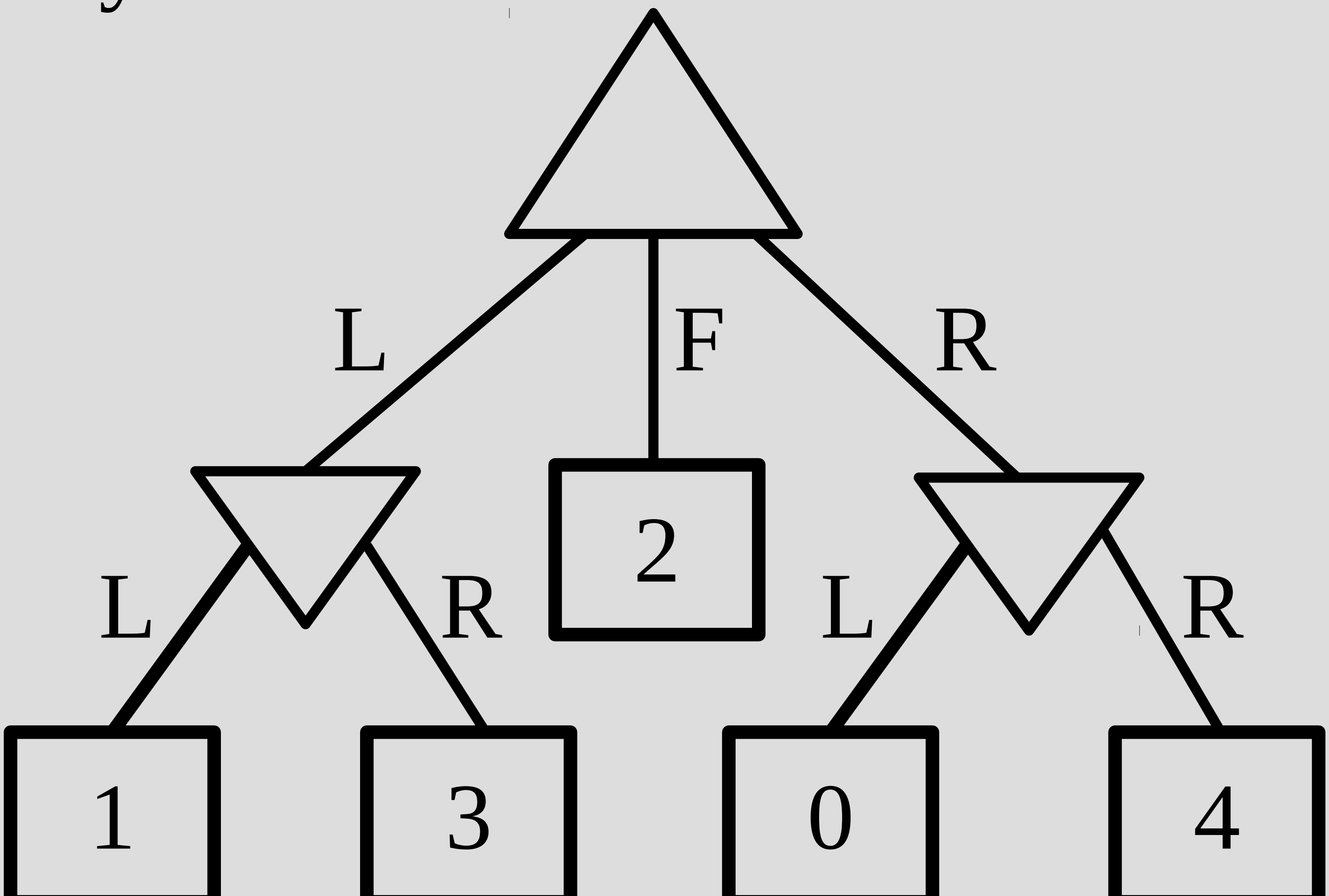
... which corresponds to:

max(Shuang Cheng choice, Afro Deli choice)

If our goal is to spend the most money on our friend, we should go to Shuang Cheng

Minimax

One way to solve this is from the leaves up:



Minimax

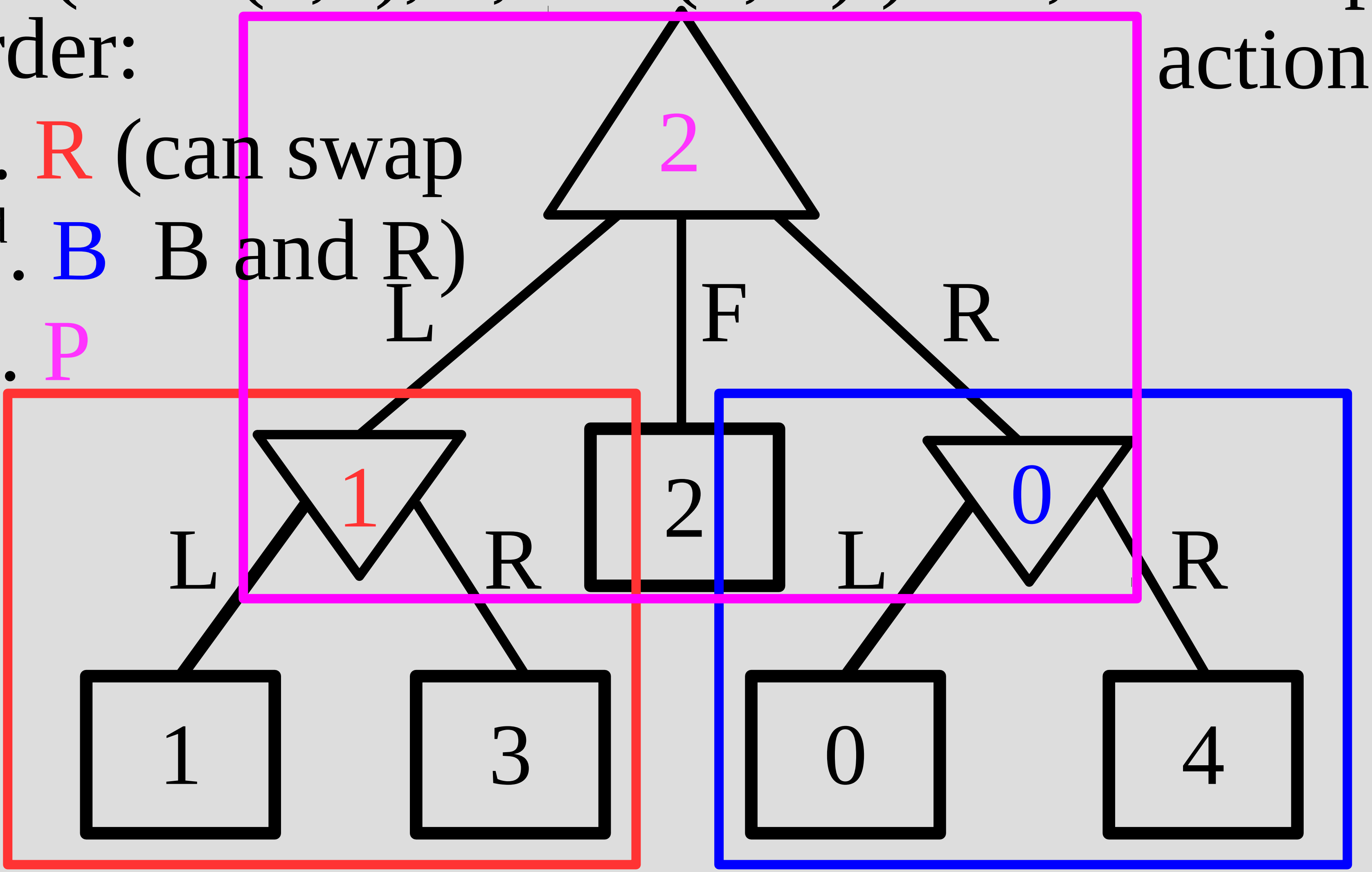
max(min(1,3), 2, min(0, 4)) = 2, should pick action F

Order:

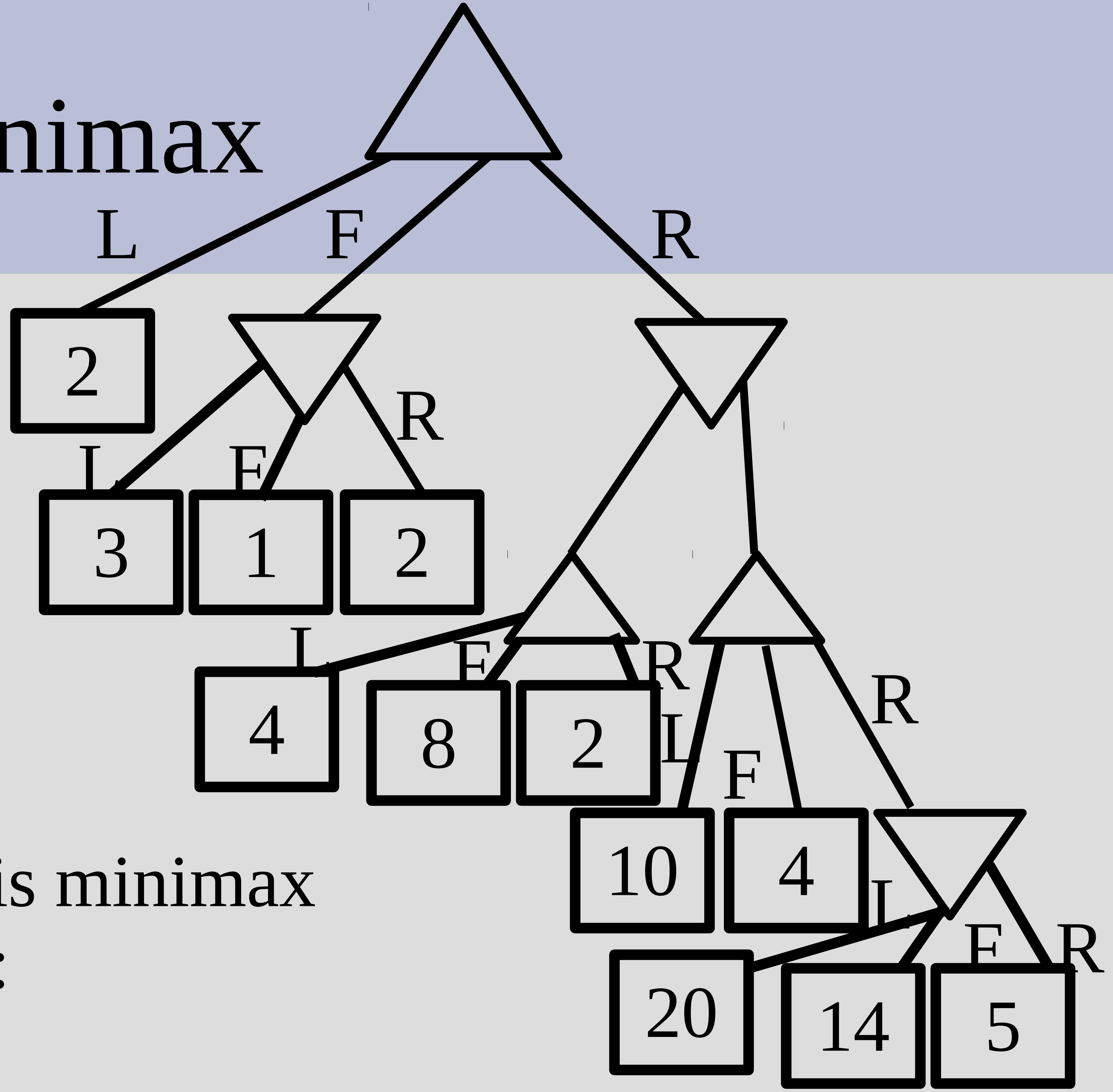
1st. R (can swap

2nd. B B and R)

3rd. P



Minimax



Solve this minimax problem:

Minimax

This representation works, but even in small games you can get a very large search tree

For example, tic-tac-toe has about 9! actions to search (or about 300,000 nodes)

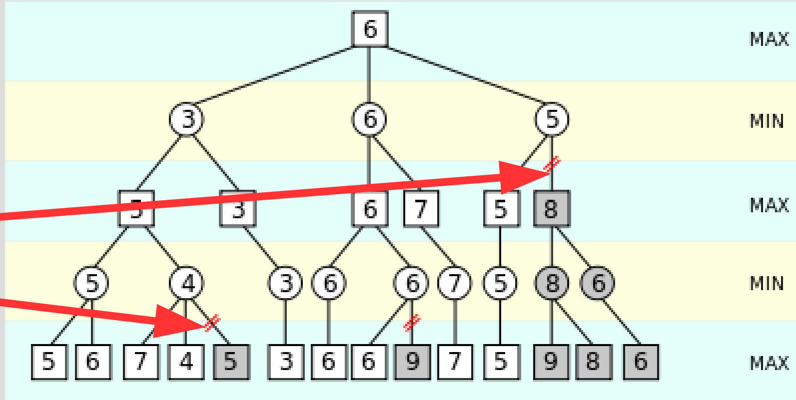
Larger problems (like chess or go) are not feasible for this approach (more on this next class)

Minimax

“Pruning” in real life:

Snip branch

“Pruning” in CSCI trees:



Snip branch

Alpha-beta pruning

However, we can get the same answer with searching less by using efficient “pruning”

It is possible to prune a minimax search that will never “accidentally” prune the optimal solution

A popular technique for doing this is called alpha-beta pruning (see next slide)

Alpha-beta pruning

Consider if we were finding the following:

max(5, min(3, 19))

There is a “short circuit evaluation” for this, namely the value of 19 does not matter

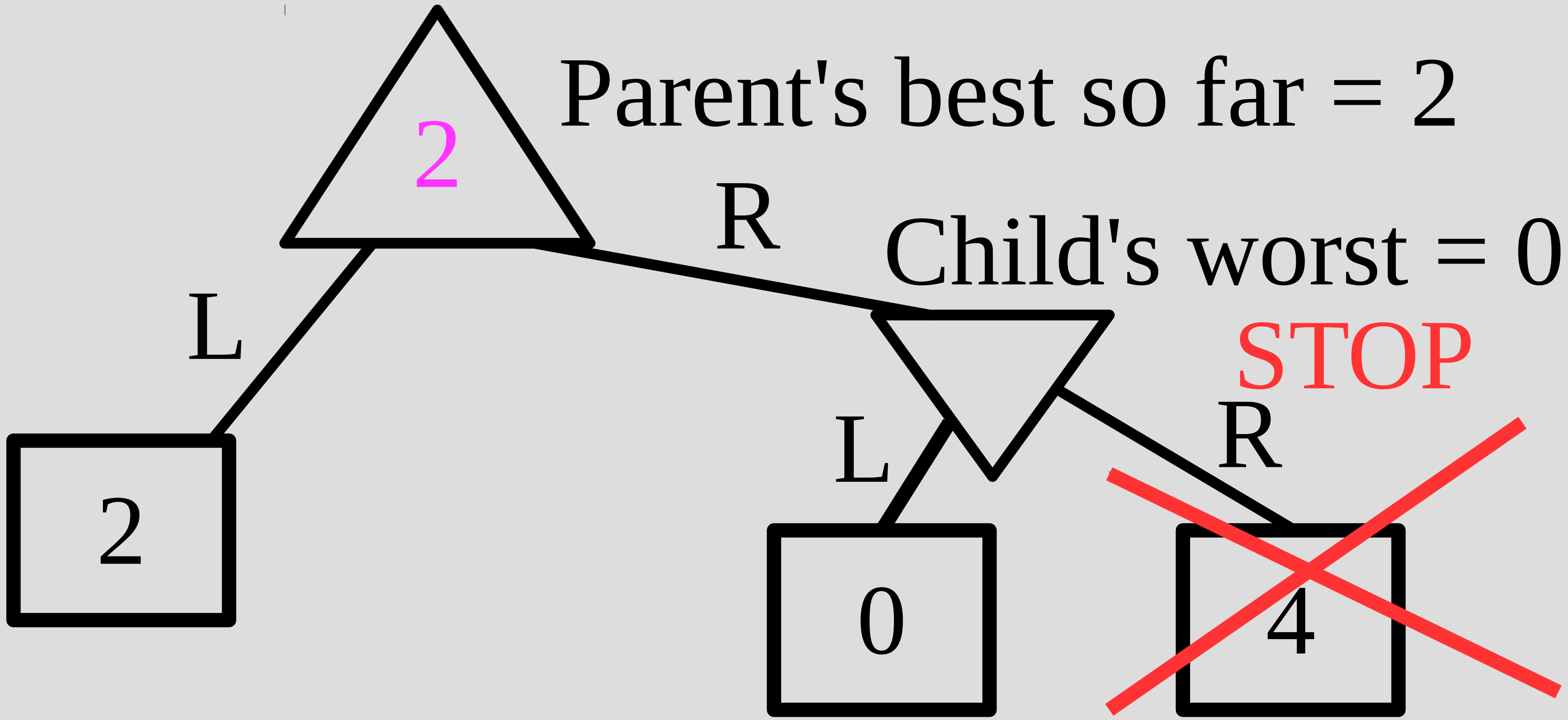
min(3, x) $\leq$ 3 for all x

Thus max(5, min(3,x)) = 5 for any x

Alpha-beta pruning would not search x above

Alpha-beta pruning

If when checking a min-node, we ever find a value less than the parent's “best” value, we can stop searching this branch



Alpha-beta pruning

In the previous slide, “best” is the “alpha” in the alpha-beta pruning
(Similarly the “worst” in a min-node is “beta”)

Alpha-beta pruning algorithm:

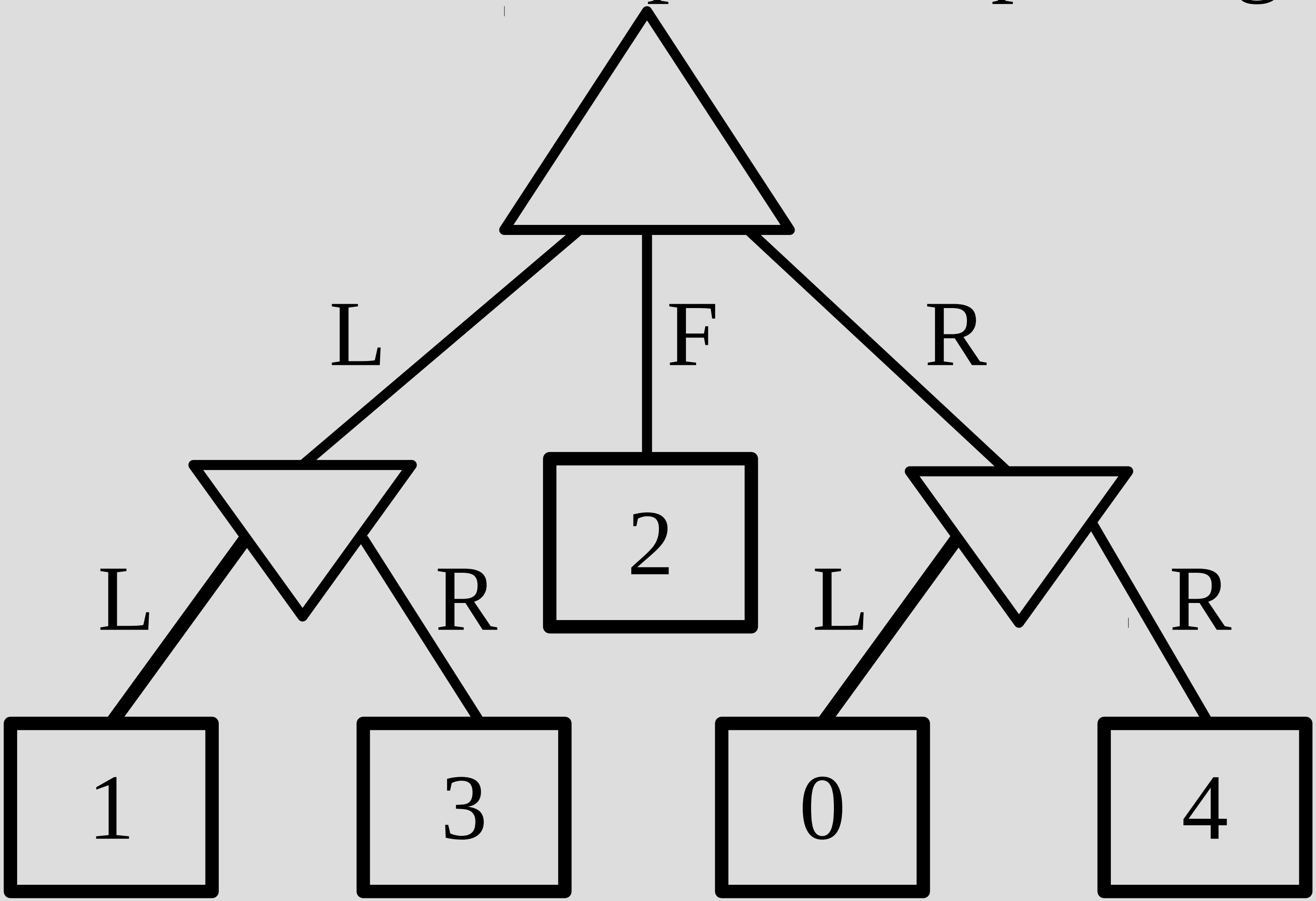
Do minimax as normal, except:

min node: if parent's “best” value greater than current node, stop & tell parent current value

max node: if parent's “worst” value less than current node, stop search and return current

Alpha-beta pruning

Let's solve this with alpha-beta pruning



Alpha-beta pruning

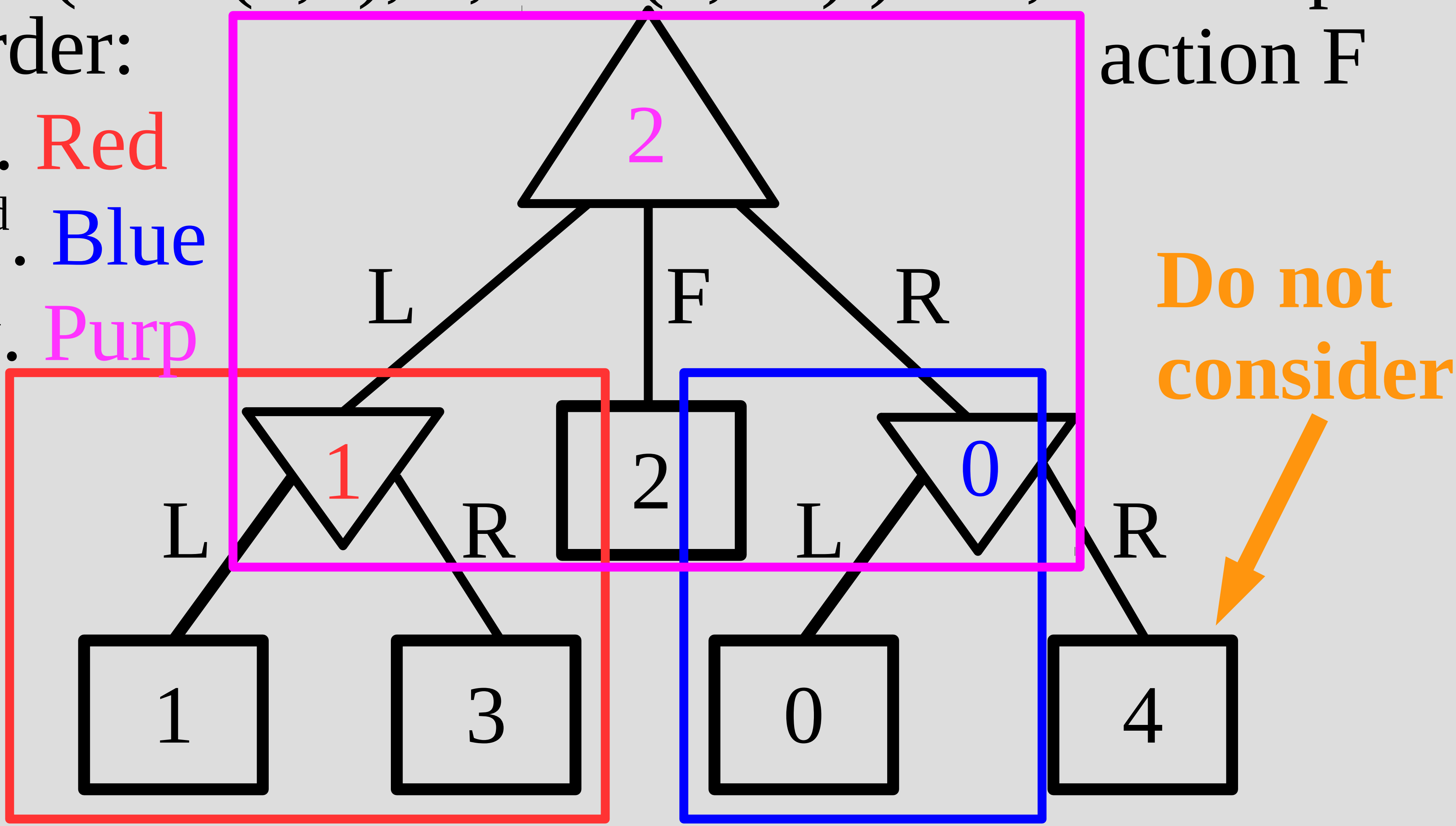
max(min(1,3), 2, min(0, ??)) = 2, should pick action F

Order:

1st. Red

2nd. Blue

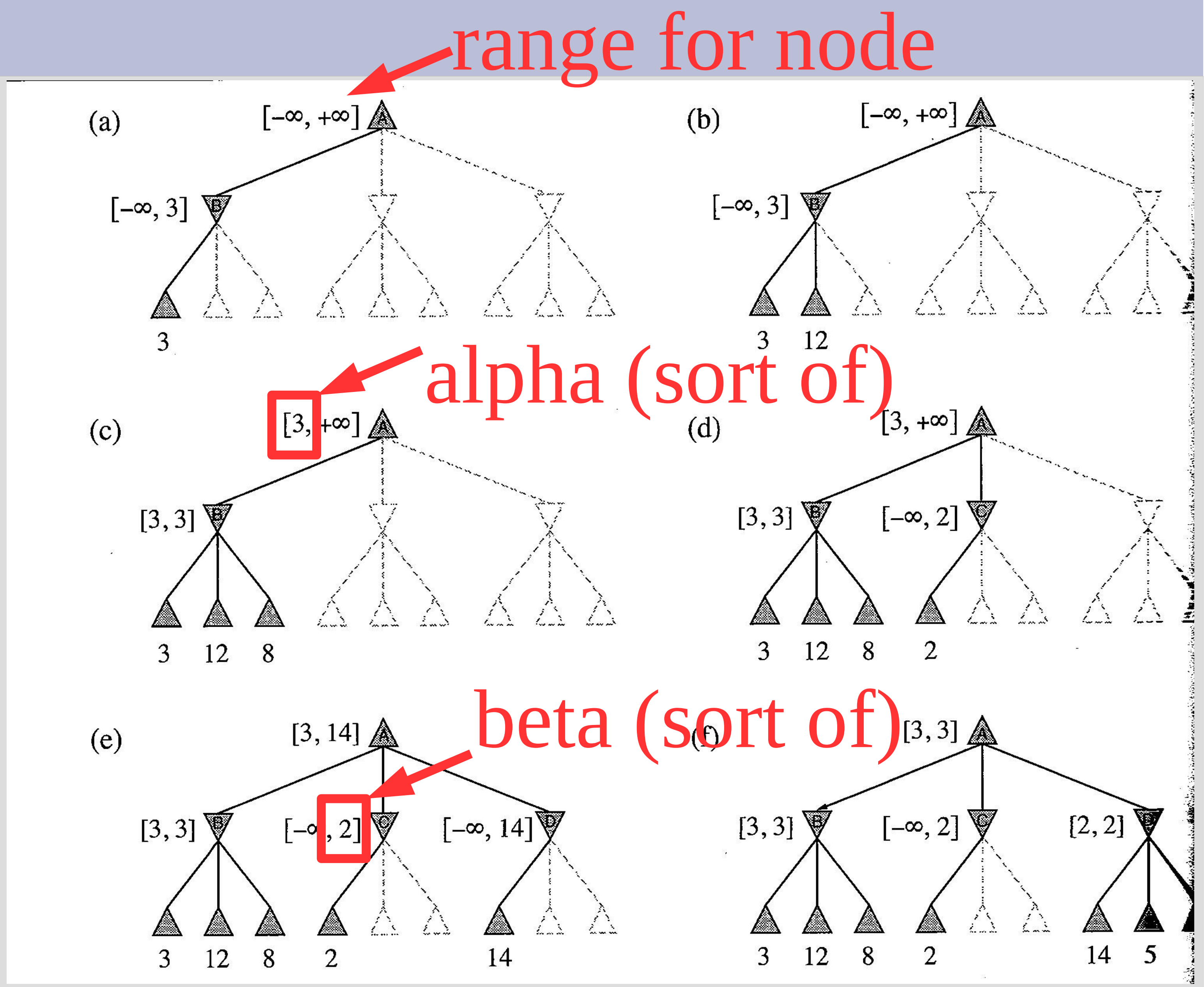
3rd. Purp



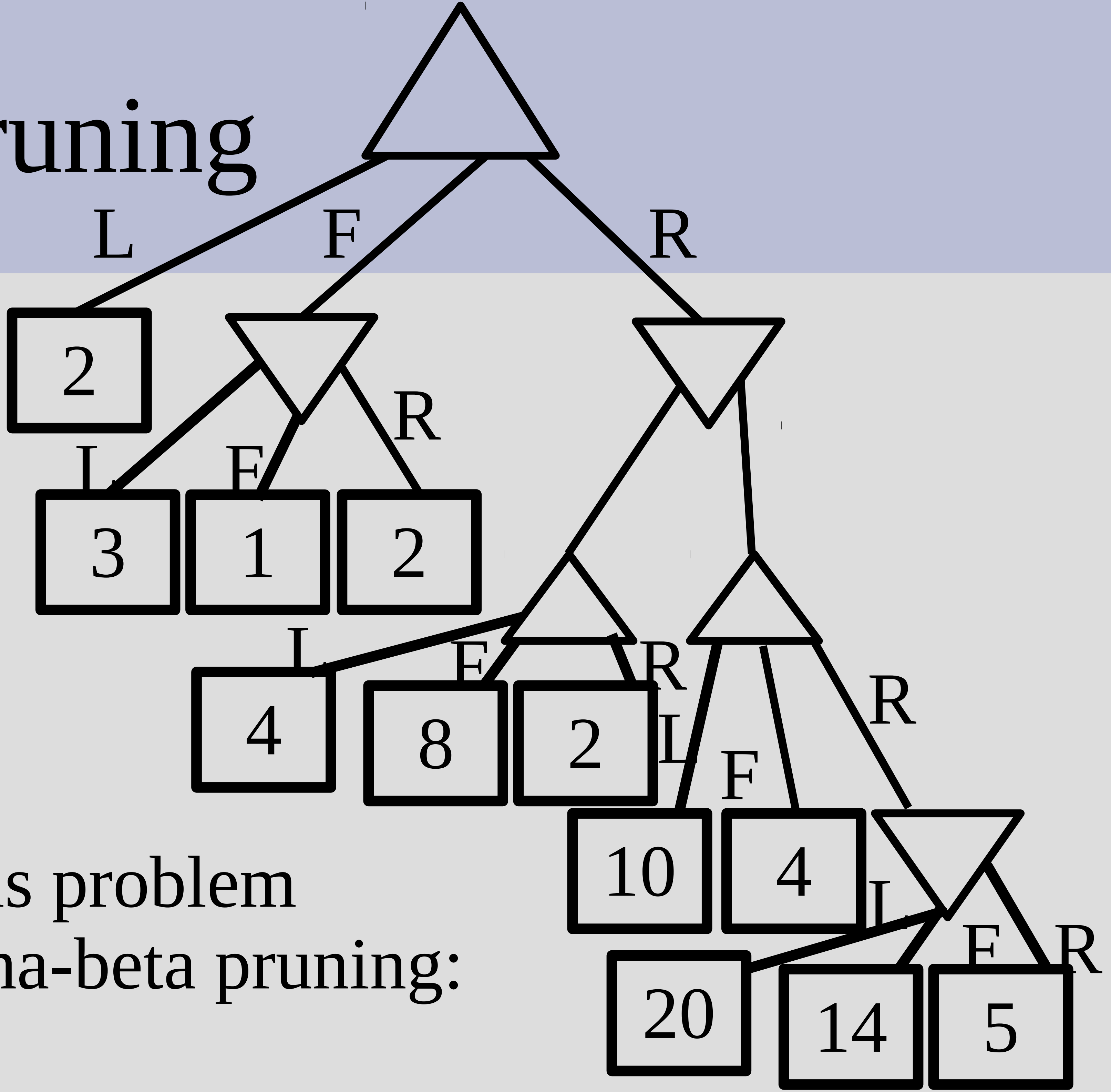
Alpha-beta pruning

\rantOn

I think the book is confusing about alpha-beta, especially Figure 5.5



αβ pruning



Solve this problem
with alpha-beta pruning:

Alpha-beta pruning

In general, alpha-beta pruning allows you to search to a depth 2d for the minimax search cost of depth d

So if minimax needs to find: $O(b^m)$
Then, alpha-beta searches: $O(b^{m/2})$

This is exponentially better, but the worst case is the same as minimax

Alpha-beta pruning

Ideally you would want to put your best (largest for max, smallest for min) actions first

This way you can prune more of the tree as a min node stops more often for larger “best”

Obviously you do not know the best move, (otherwise why are you searching?) but some effort into guessing goes a long way (i.e. exponentially less states)

Side note:

In alpha-beta pruning, the heuristic for guess which move is best can be complex, as you can greatly effect pruning

While for A* search, the heuristic had to be very fast to be useful
(otherwise computing the heuristic would take longer than the original search)