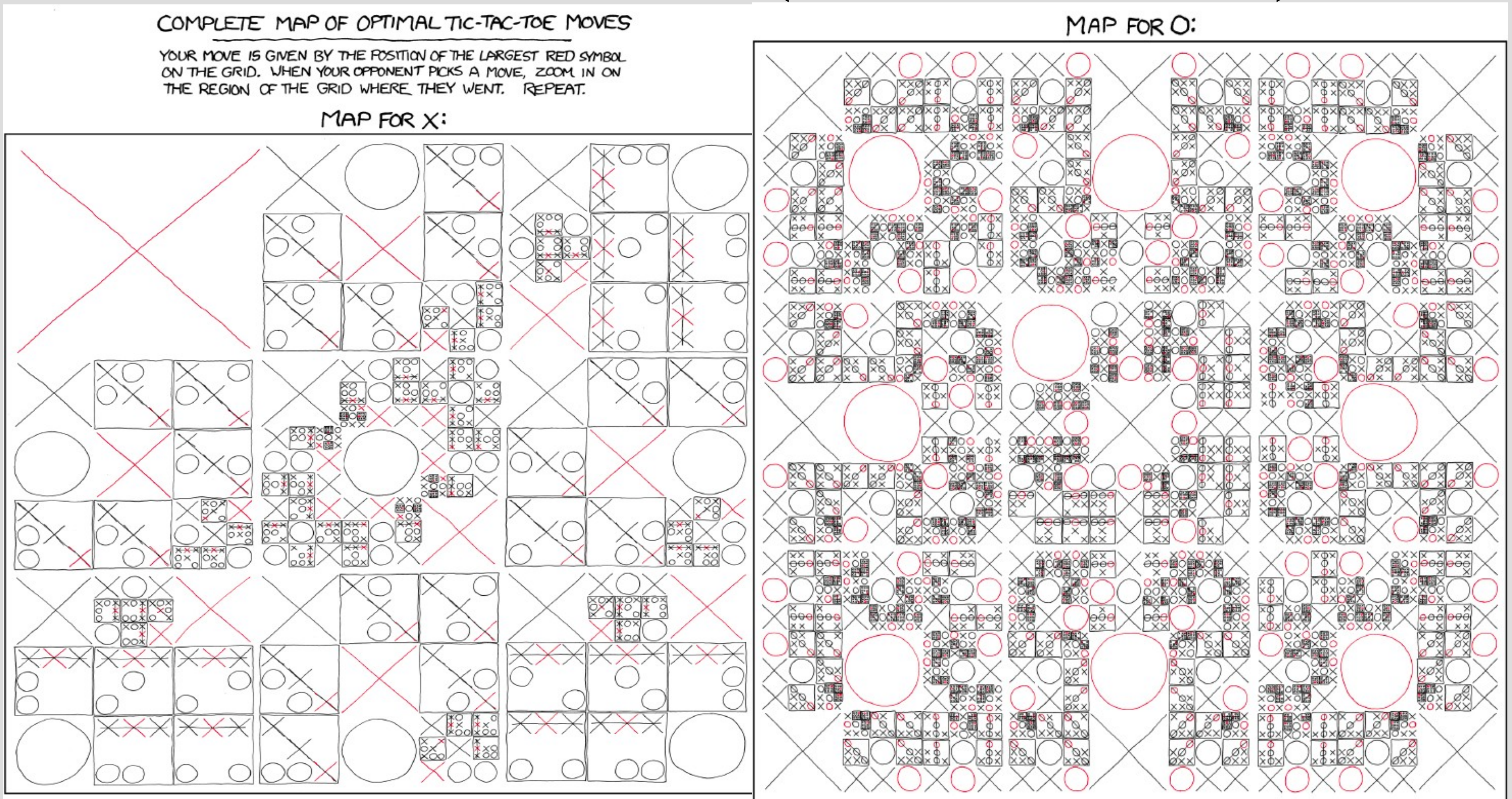
Minimax (Ch. 5-5.3)


COMPLETE MAP OF OPTIMAL TIC-TAC-TOE MOVES
YOUR MOVE IS GIVEN BY THE POSITION OF THE LARGEST RED SYMBOL ON THE GRID. WHEN YOUR OPPONENT PICKS A MOVE, ZOOM IN ON THE REGION OF THE GRID WHERE THEY WENT. REPEAT.
MAP FOR X:
MAP FOR O:

Local beam search

Beam search is similar to hill climbing, except we track multiple states simultaneously

Initialize: start with K random nodes
1. Find all children of the K nodes
2. Add children and K nodes to pool, pick best
3. Repeat...

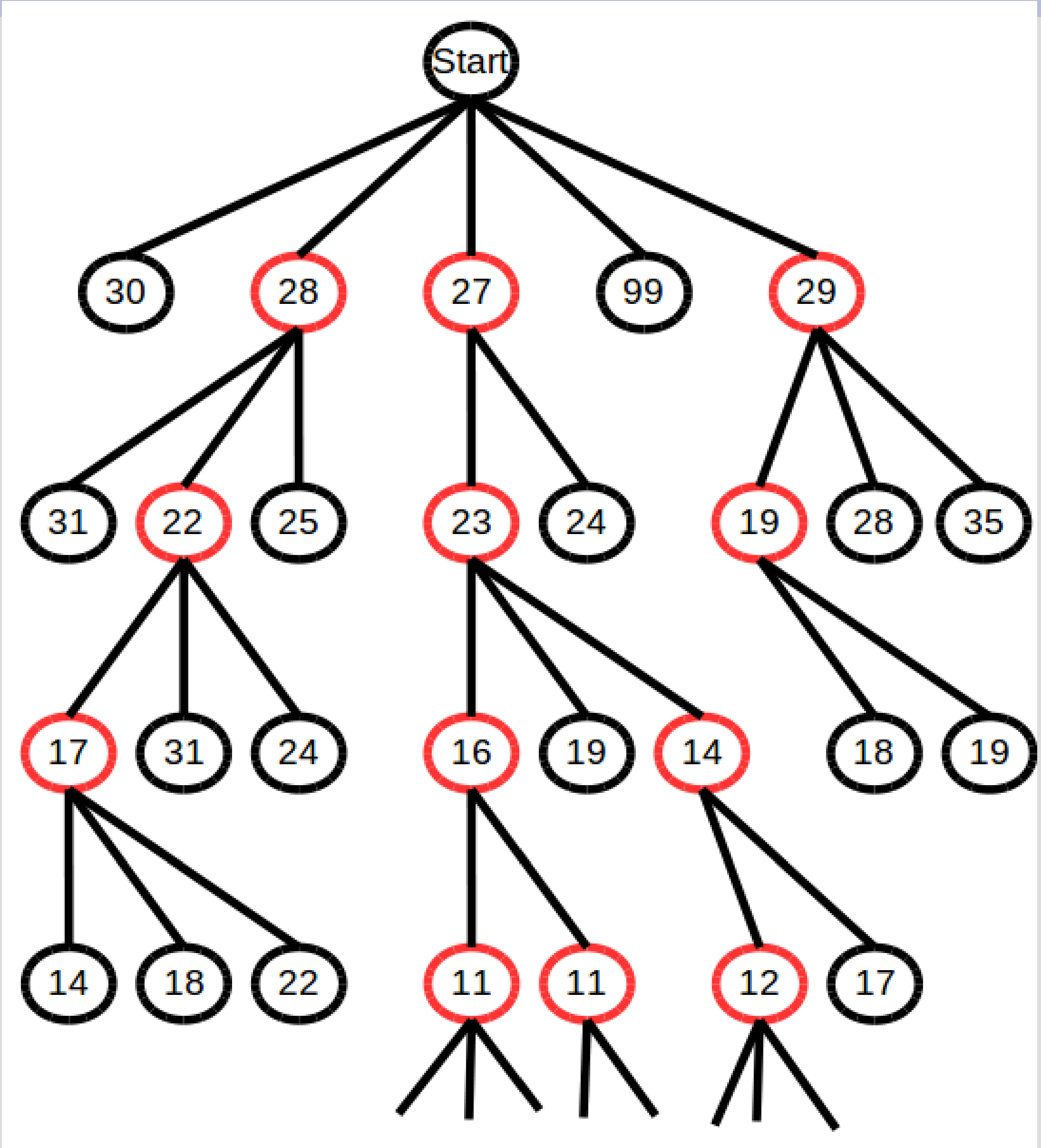
Unlike previous approaches, this uses more memory to better search “hopeful” options

Local beam search

Beam search with 3 beams

Pick best 3 options at each stage to expand

Stop like hill-climb (next pick is same as last pick)



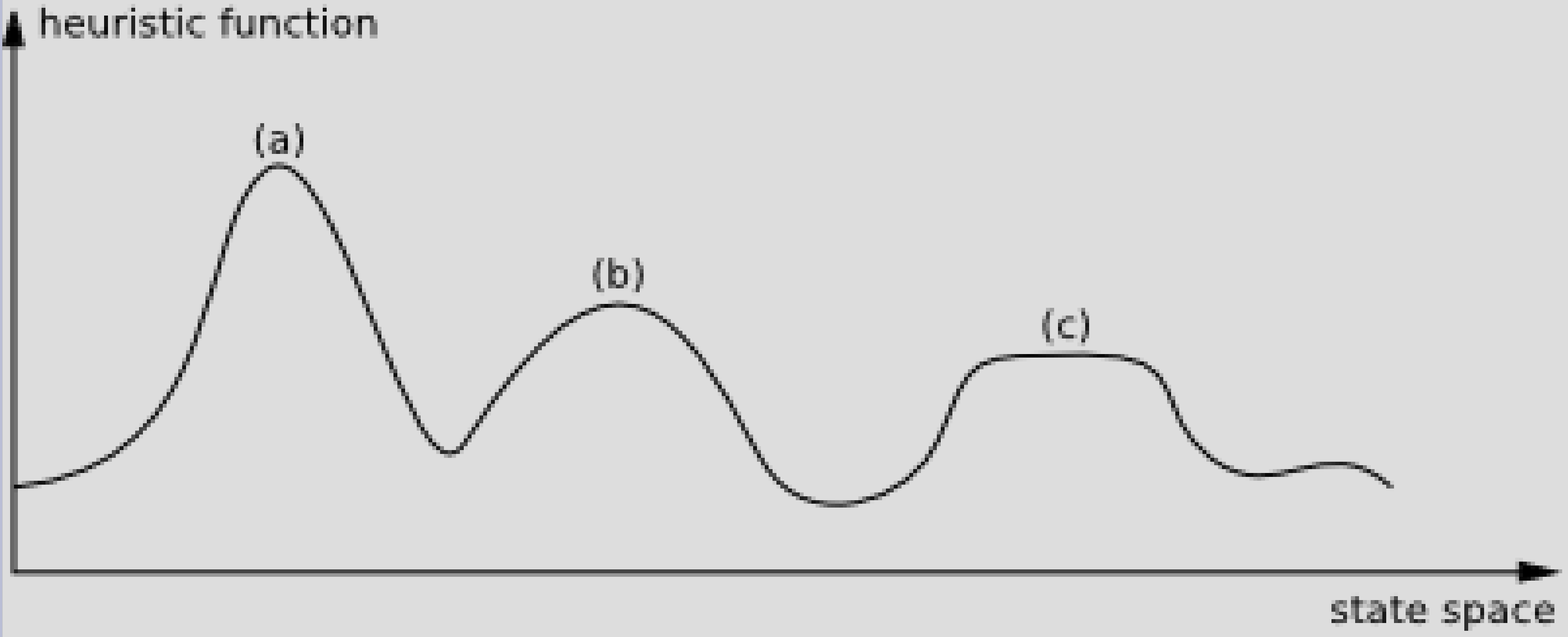
Local beam search

However, the basic version of beam search can get stuck in local maximum as well

To help avoid this, stochastic beam search picks children with probability relative to their values

This is different that hill climbing with K restarts as better options get more consideration than worse ones

Local beam search

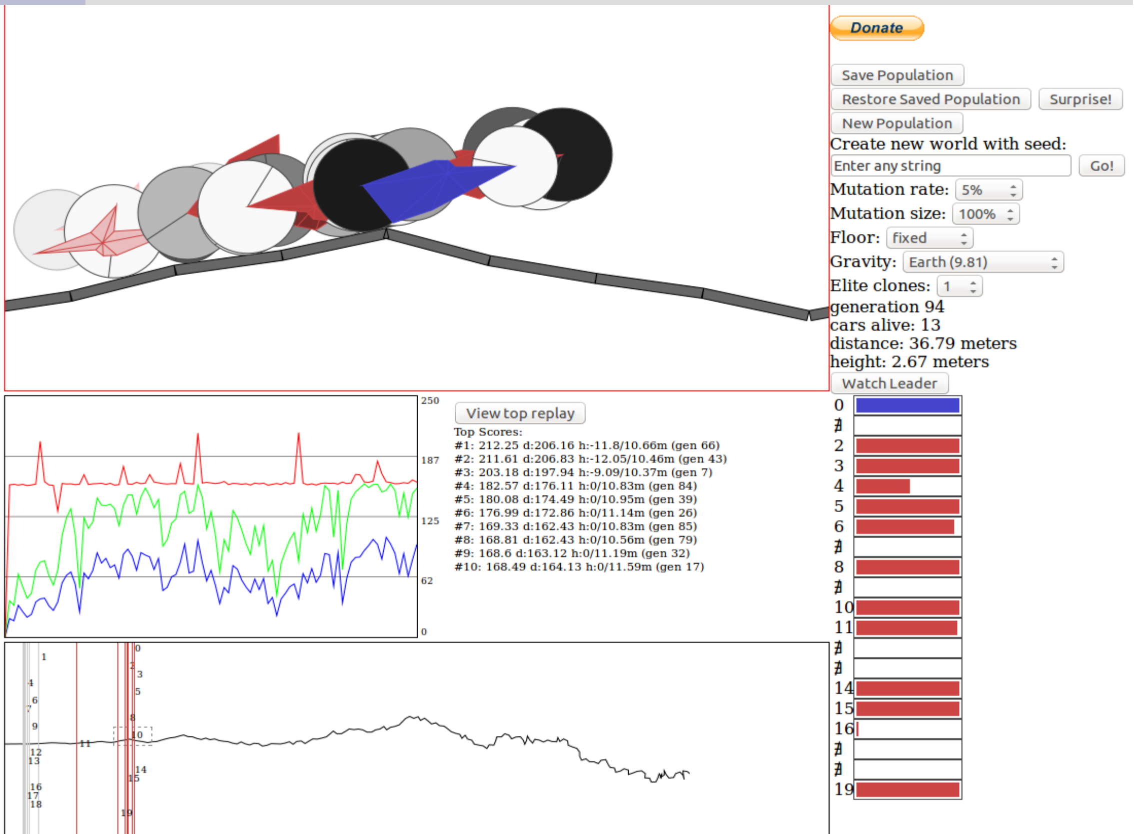


Genetic algorithms

Nice examples of GAs:

http://rednuht.org/genetic_cars_2/

http://boxcar2d.com/



Create new world with seed: Enter any string

Mutation rate: 5%

Mutation size: 100%

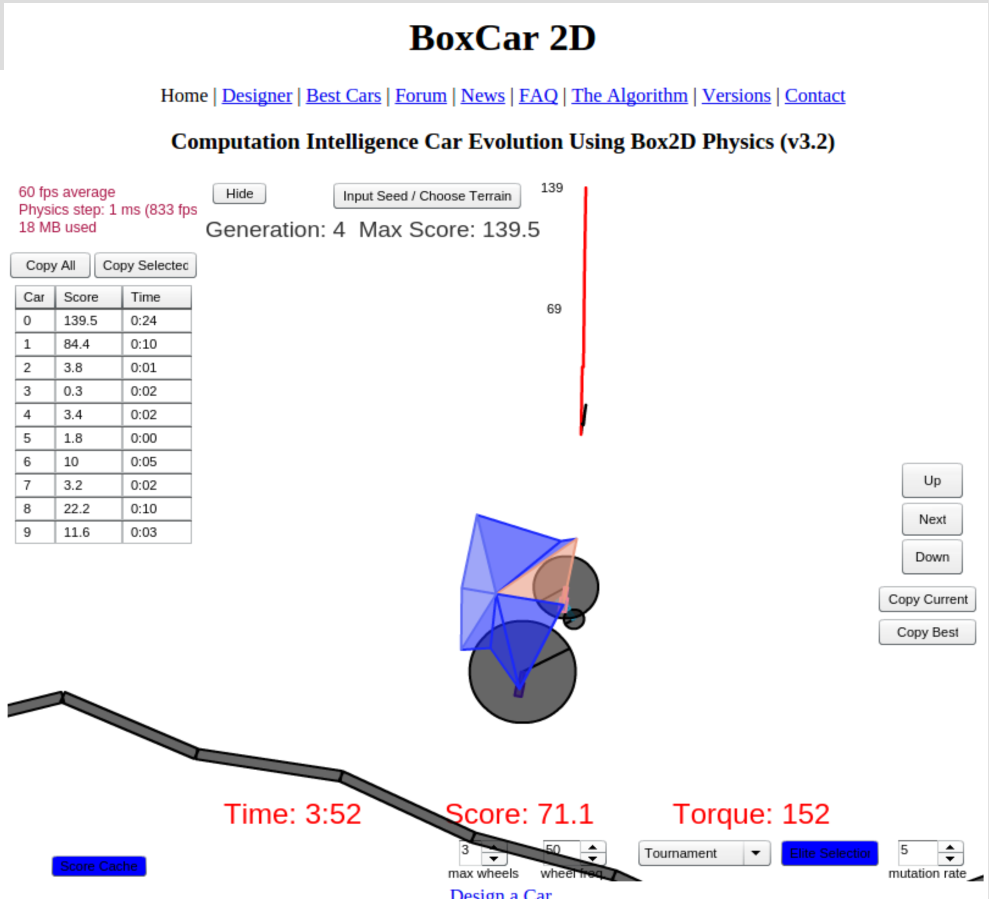
Floor: fixed

Gravity: Earth (9.81)

Elite clones: 1

generation 94
cars alive: 13
distance: 36.79 meters
height: 2.67 meters

Top Scores:
#1: 212.25 d:206.16 h:-11.8/10.66m (gen 66)
#2: 211.61 d:206.83 h:-12.05/10.46m (gen 43)
#3: 203.18 d:197.94 h:-9.09/10.37m (gen 7)
#4: 182.57 d:176.11 h:0/10.83m (gen 84)
#5: 180.08 d:174.49 h:0/10.95m (gen 39)
#6: 176.99 d:172.86 h:0/11.14m (gen 26)
#7: 169.33 d:162.43 h:0/10.83m (gen 85)
#8: 168.81 d:162.43 h:0/10.56m (gen 79)
#9: 168.6 d:163.12 h:0/11.19m (gen 32)
#10: 168.49 d:164.13 h:0/11.59m (gen 17)


Home | Designer | Best Cars | Forum | News | FAQ | The Algorithm | Versions | Contact
Hide
Input Seed / Choose Terrain
Copy All
Copy Selected
Up
Next
Down
Copy Current
Copy Best
Score Cache
Tournament
Elite Selection
Design a Car

BoxCar 2D

Computation Intelligence Car Evolution Using Box2D Physics (v3.2)

60 fps average
Physics step: 1 ms (833 fps
18 MB used

Generation: 4 Max Score: 139.5

Car	Score	Time
0	139.5	0:24
1	84.4	0:10
2	3.8	0:01
3	0.3	0:02
4	3.4	0:02
5	1.8	0:00
6	10	0:05
7	3.2	0:02
8	22.2	0:10
9	11.6	0:03

Time: 3:52 Score: 71.1 Torque: 152

max wheels: 3 wheel freq: 50 mutation rate: 5

Genetic algorithms

Genetic algorithms are based on how life has evolved over time

They (in general) have 3 (or 5) parts:

1. Select/generate children
    1a. Select 2 random parents
    1b. Mutate/crossover
2. Test fitness of children to see if they survive
3. Repeat until convergence

Genetic algorithms

Genetic algorithms are based on how life has evolved over time

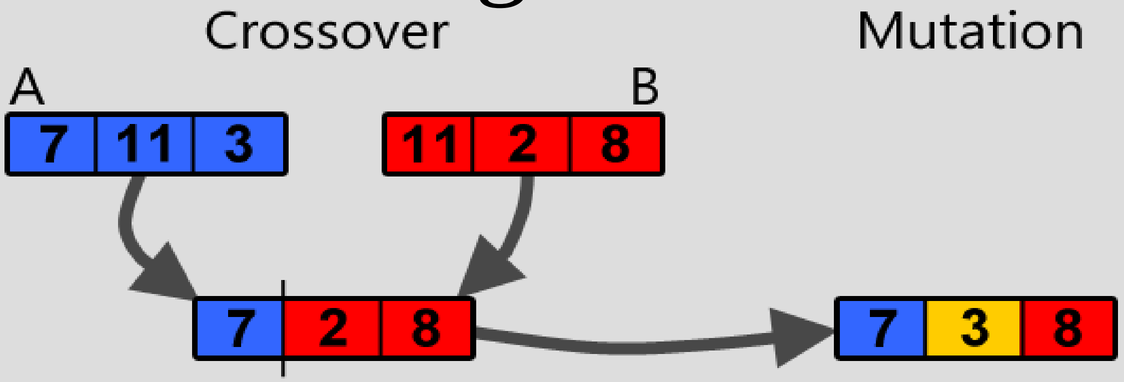
They (in general) have 3 (or 5) parts:

1. Select/generate children
    1a. Select 2 random parents
    1b. Mutate/crossover
2. Test fitness of children to see if they survive
3. Repeat until convergence

Genetic algorithms

- Selection/survival:

Typically children have a probabilistic survival rate (randomness ensures genetic diversity)



- Crossover:

Split the parent's information into two parts, then take part 1 from parent A and 2 from B

- Mutation:

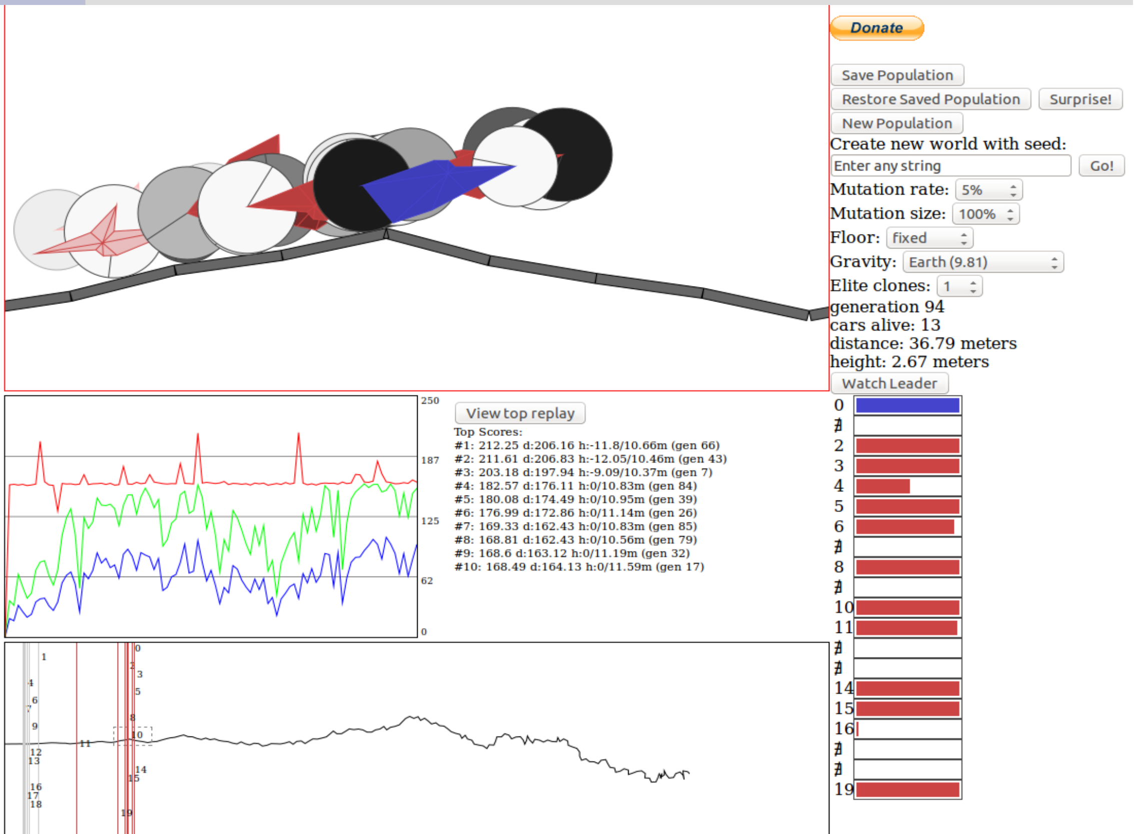
Change a random part to a random value

Genetic algorithms

Nice examples of GAs:

http://rednuht.org/genetic_cars_2/

http://boxcar2d.com/



Donate

Save Population

Restore Saved Population | Surprise!

New Population

Create new world with seed: Enter any string | Go!

Mutation rate: 5%

Mutation size: 100%

Floor: fixed

Gravity: Earth (9.81)

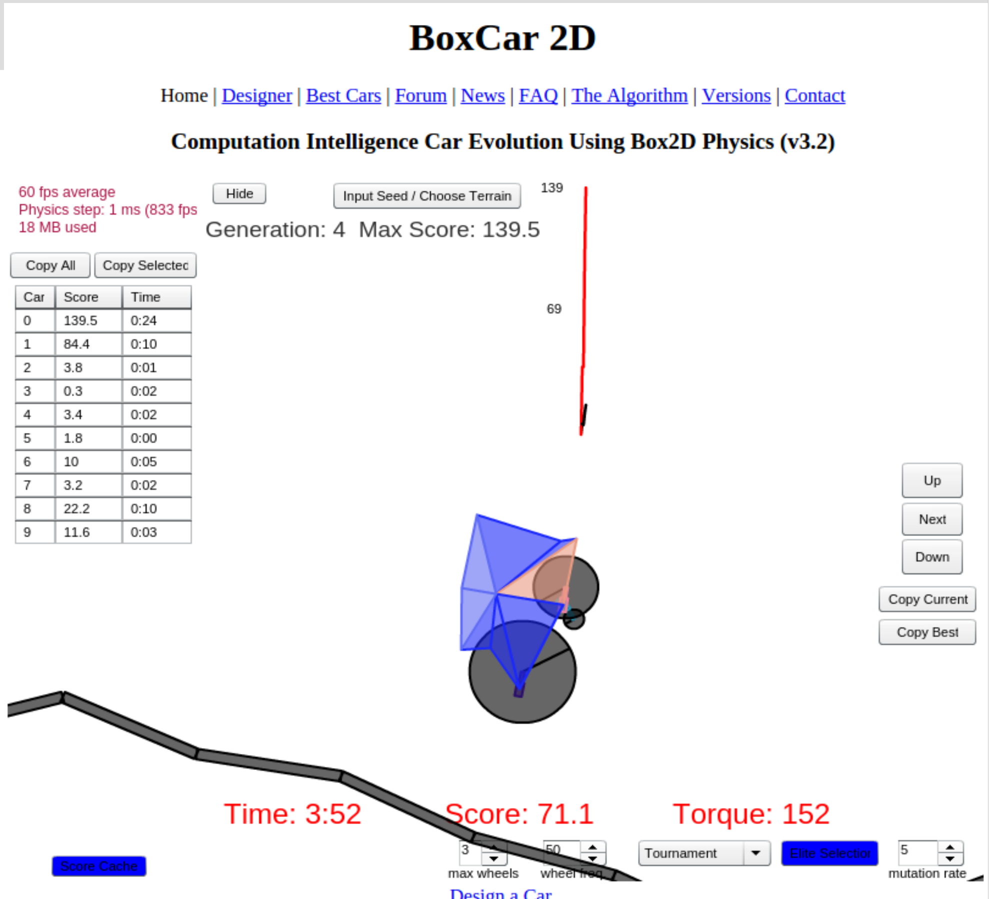
Elite clones: 1

generation 94
cars alive: 13
distance: 36.79 meters
height: 2.67 meters

Watch Leader

View top replay

Top Scores:
#1: 212.25 d:206.16 h:-11.8/10.66m (gen 66)
#2: 211.61 d:206.83 h:-12.05/10.46m (gen 43)
#3: 203.18 d:197.94 h:-9.09/10.37m (gen 7)
#4: 182.57 d:176.11 h:0/10.83m (gen 84)
#5: 180.08 d:174.49 h:0/10.95m (gen 39)
#6: 176.99 d:172.86 h:0/11.14m (gen 26)
#7: 169.33 d:162.43 h:0/10.83m (gen 85)
#8: 168.81 d:162.43 h:0/10.56m (gen 79)
#9: 168.6 d:163.12 h:0/11.19m (gen 32)
#10: 168.49 d:164.13 h:0/11.59m (gen 17)



BoxCar 2D

Home | Designer | Best Cars | Forum | News | FAQ | The Algorithm | Versions | Contact

Computation Intelligence Car Evolution Using Box2D Physics (v3.2)

60 fps average
Physics step: 1 ms (833 fps
18 MB used

Hide | Input Seed / Choose Terrain

Generation: 4 Max Score: 139.5

Copy All | Copy Selectec

Car	Score	Time
0	139.5	0:24
1	84.4	0:10
2	3.8	0:01
3	0.3	0:02
4	3.4	0:02
5	1.8	0:00
6	10	0:05
7	3.2	0:02
8	22.2	0:10
9	11.6	0:03

Time: 3:52 Score: 71.1 Torque: 152

max wheels: 3 wheel freq: 50 Tournament mutation rate: 5

Genetic algorithms

Genetic algorithms are very good at optimizing the fitness evaluation function (assuming fitness fairly continuous)

While you have to choose parameters (i.e. mutation frequency, how often to take a gene, etc.), typically GAs converge for most

The downside is that often it takes many generations to converge to the optimal

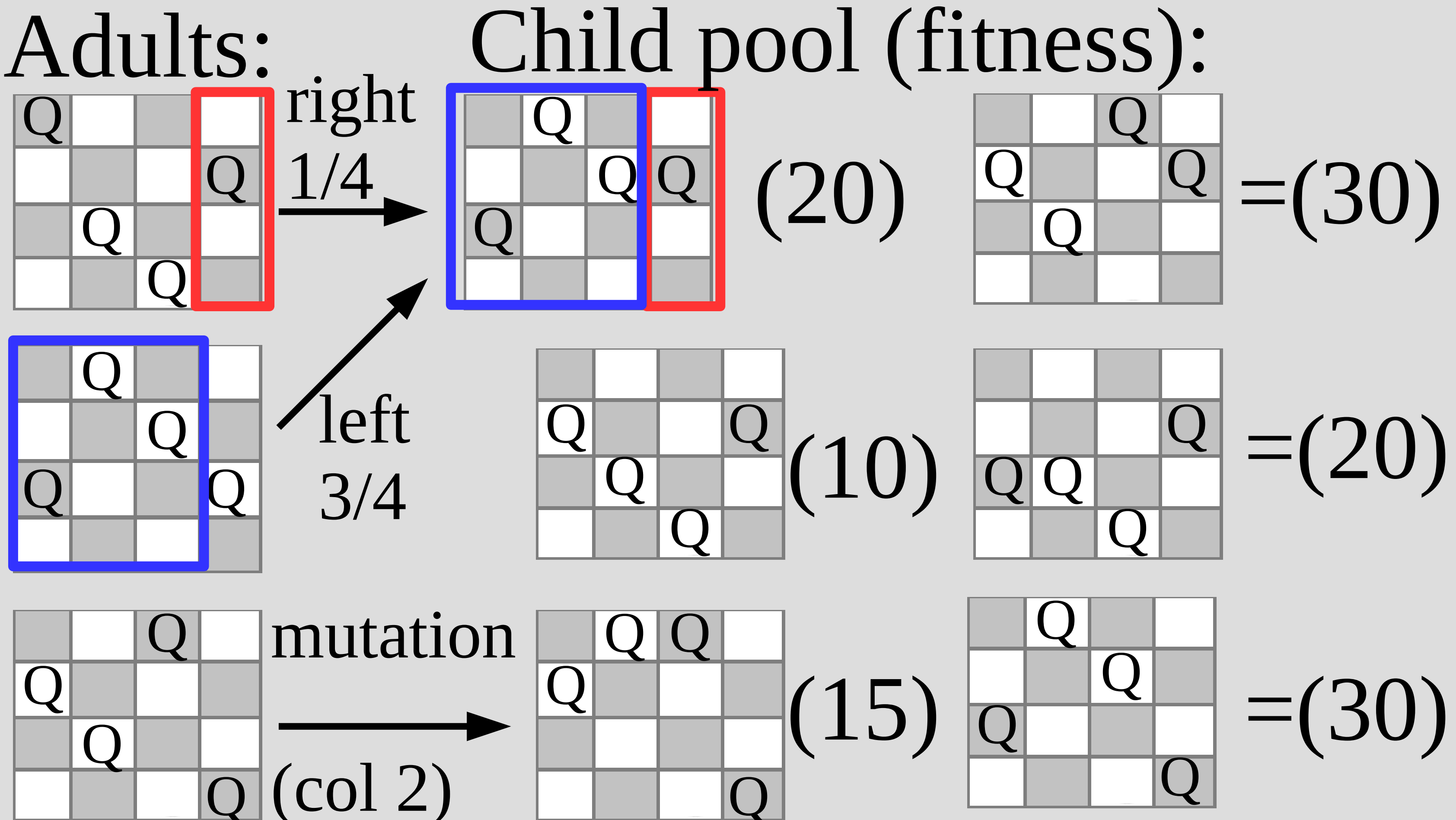
Genetic algorithms

There are a wide range of options for selecting who to bring to the next generation:
- always the top people/configurations (similar to hill-climbing... gets stuck a lot)
- choose purely by weighted random (i.e. 4 fitness chosen twice as much as 2 fitness)
- choose the best and others weighted random

Can get stuck if pool's diversity becomes too little (hope for many random mutations)

Genetic algorithms

Let's make a small (fake) example with the 4-queens problem

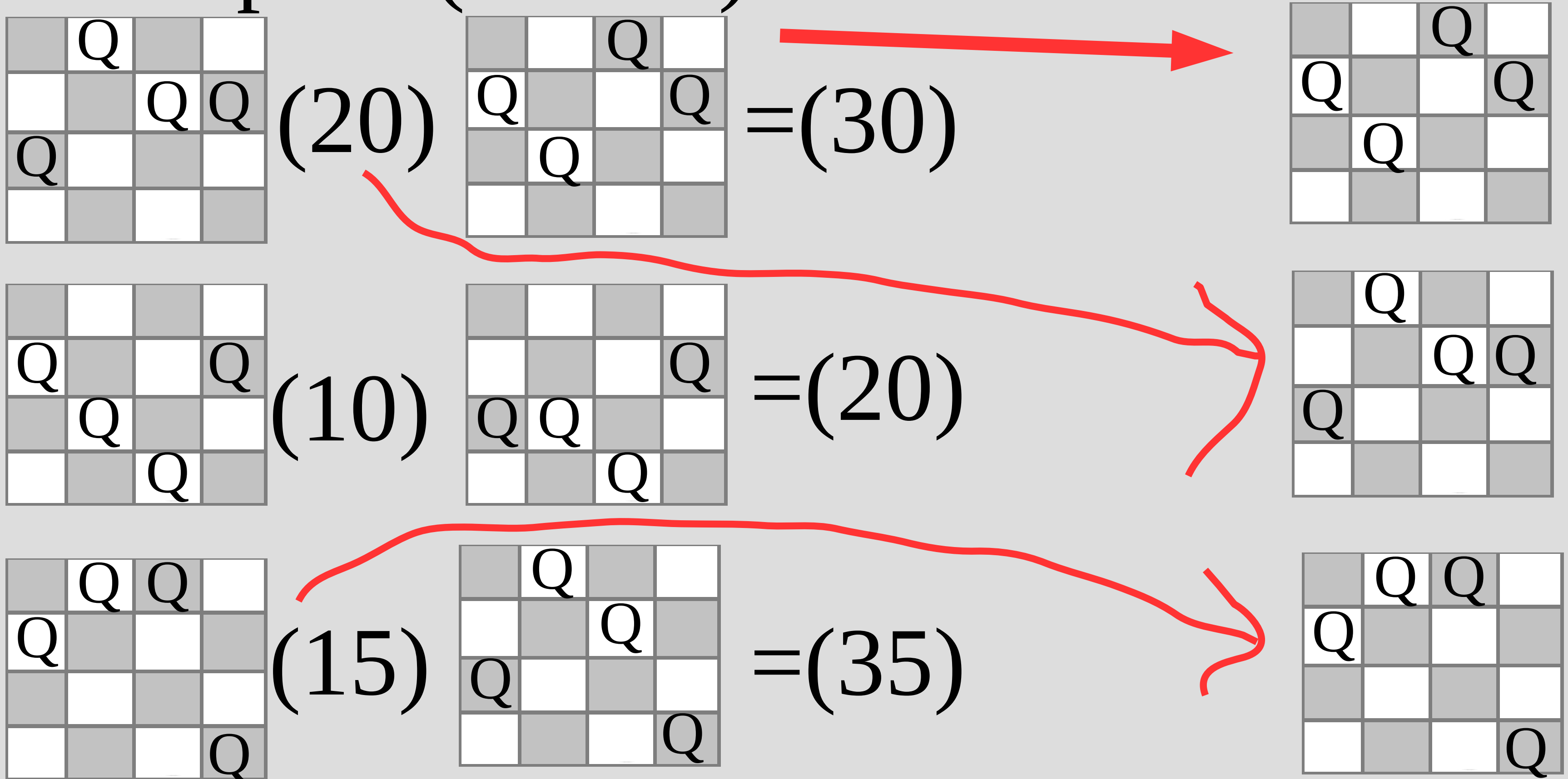


Genetic algorithms

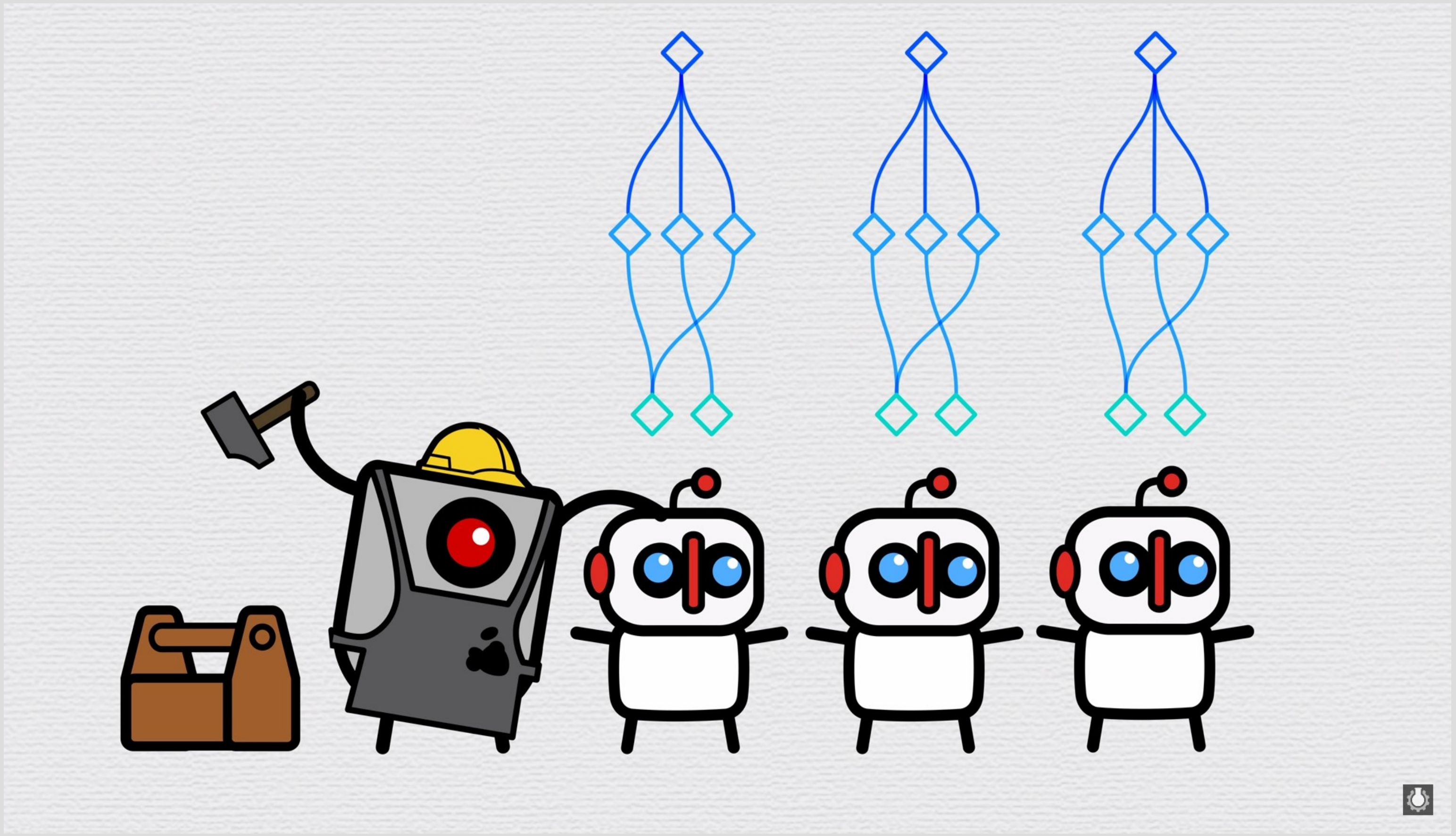
Let's make a small (fake) example with the 4-queens problem

Child pool (fitness):

Weighted random selection:



Genetic algorithms



https://www.youtube.com/watch?v=R9OHn5ZF4Uo

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 <u>minimax</u>

I will use △ to denote nodes that try to maximize and ▽ for minimizing nodes

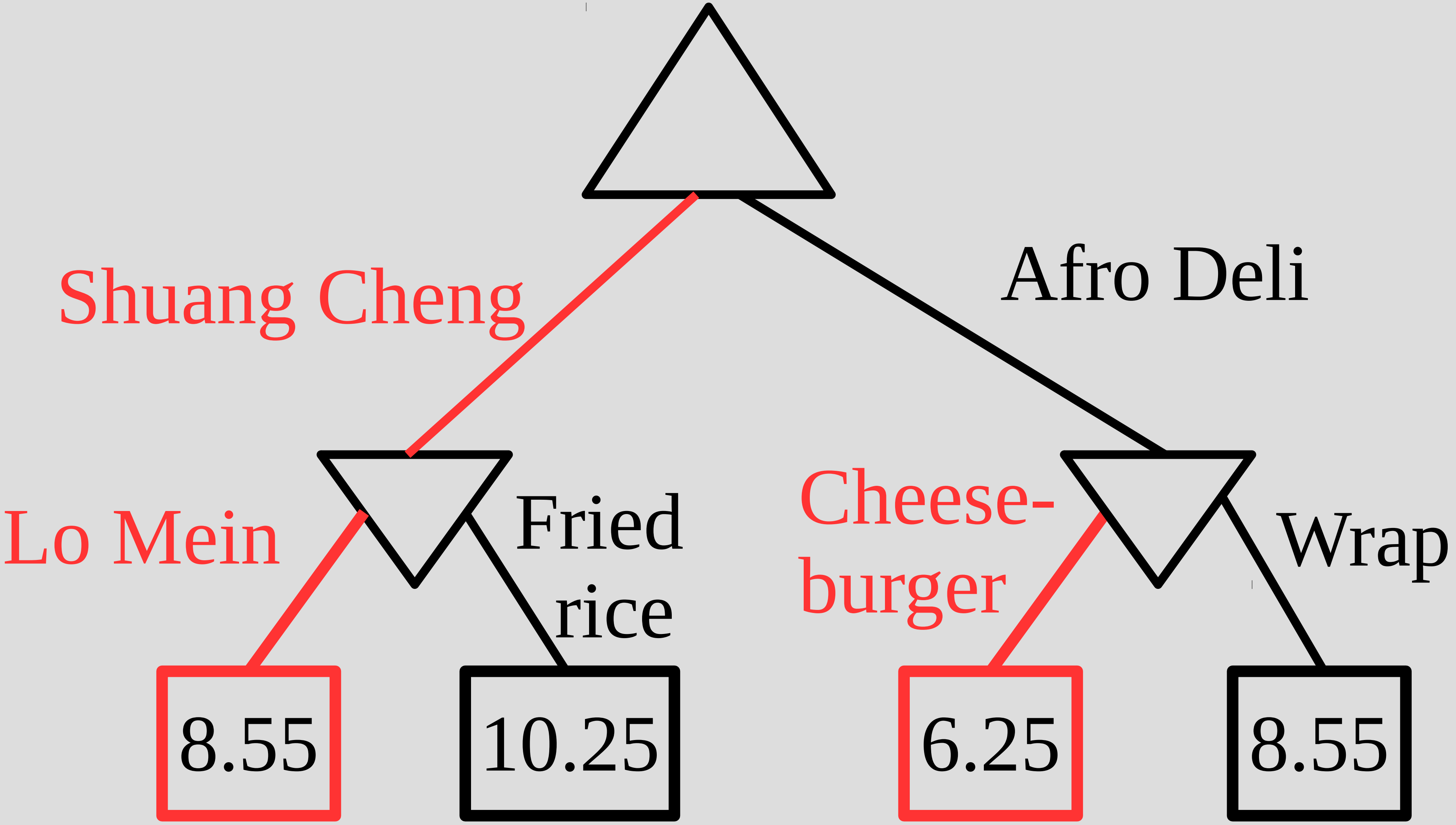
Minimax

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


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

Minimax

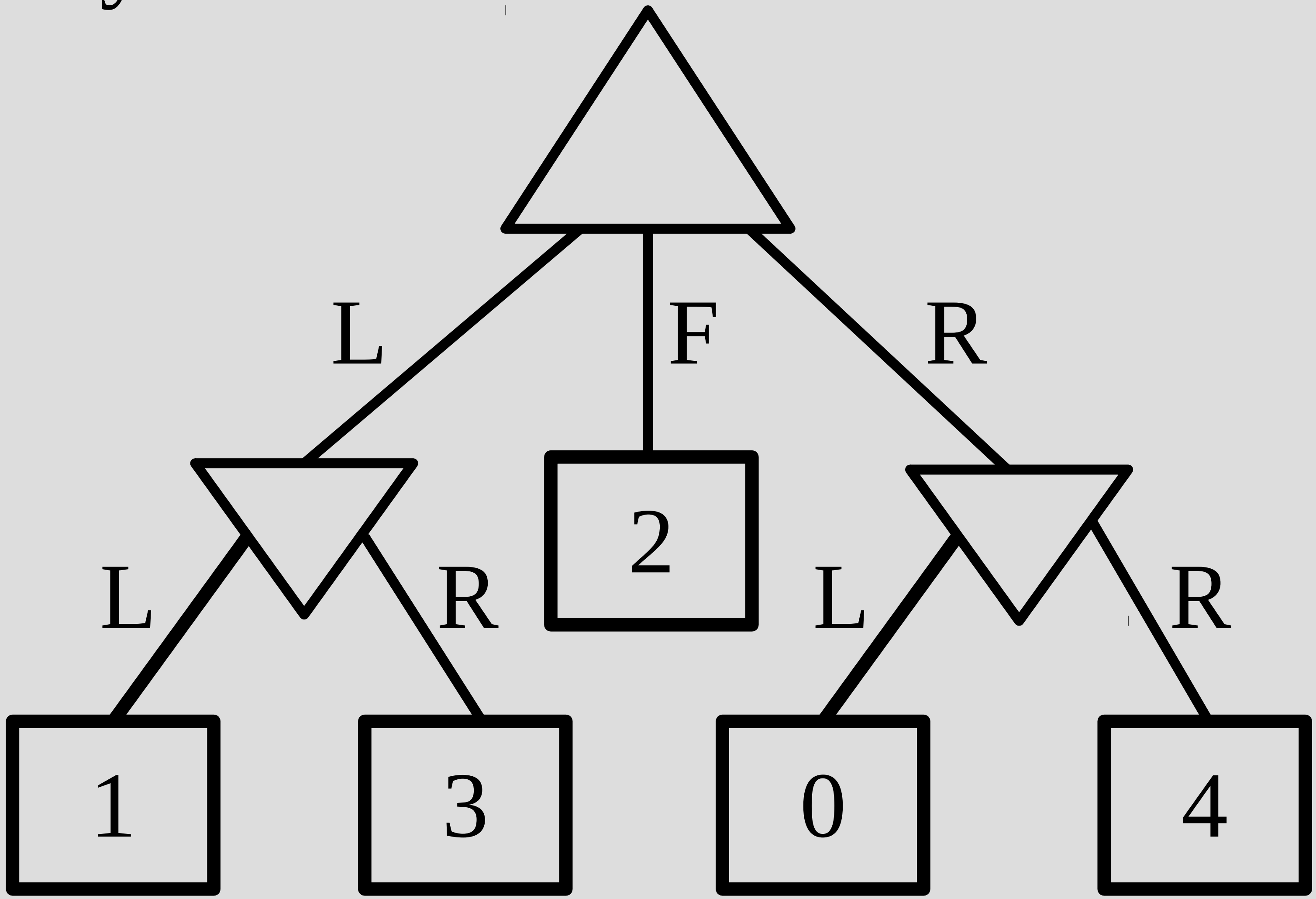
You could phrase this problem as a set of maximum and minimums as:
max(min(8.55, 10.25), min(6.25, 8.55))

... 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

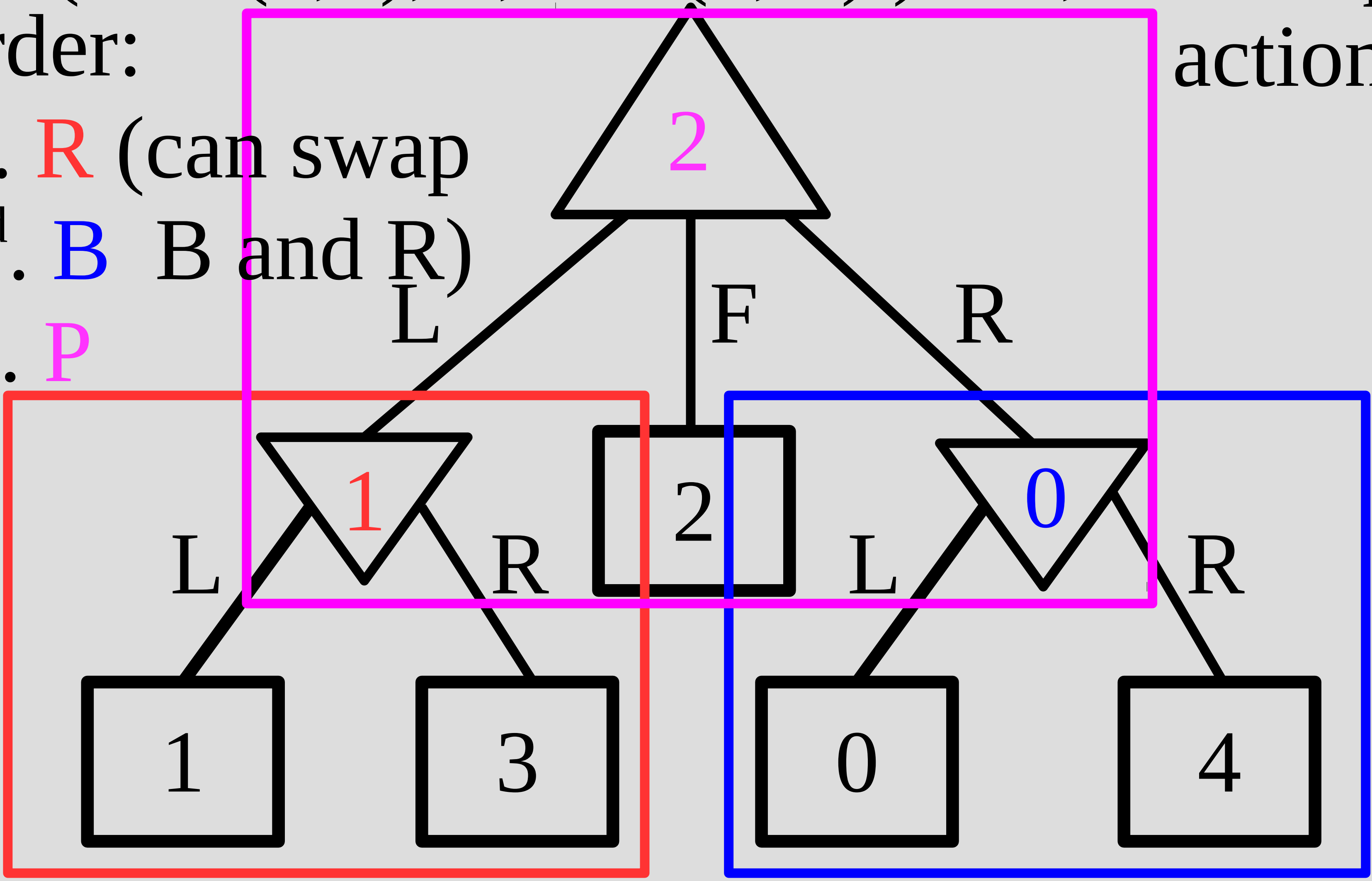
max(min(1,3), 2, min(0, 4)) = 2, should pick action F

Order:

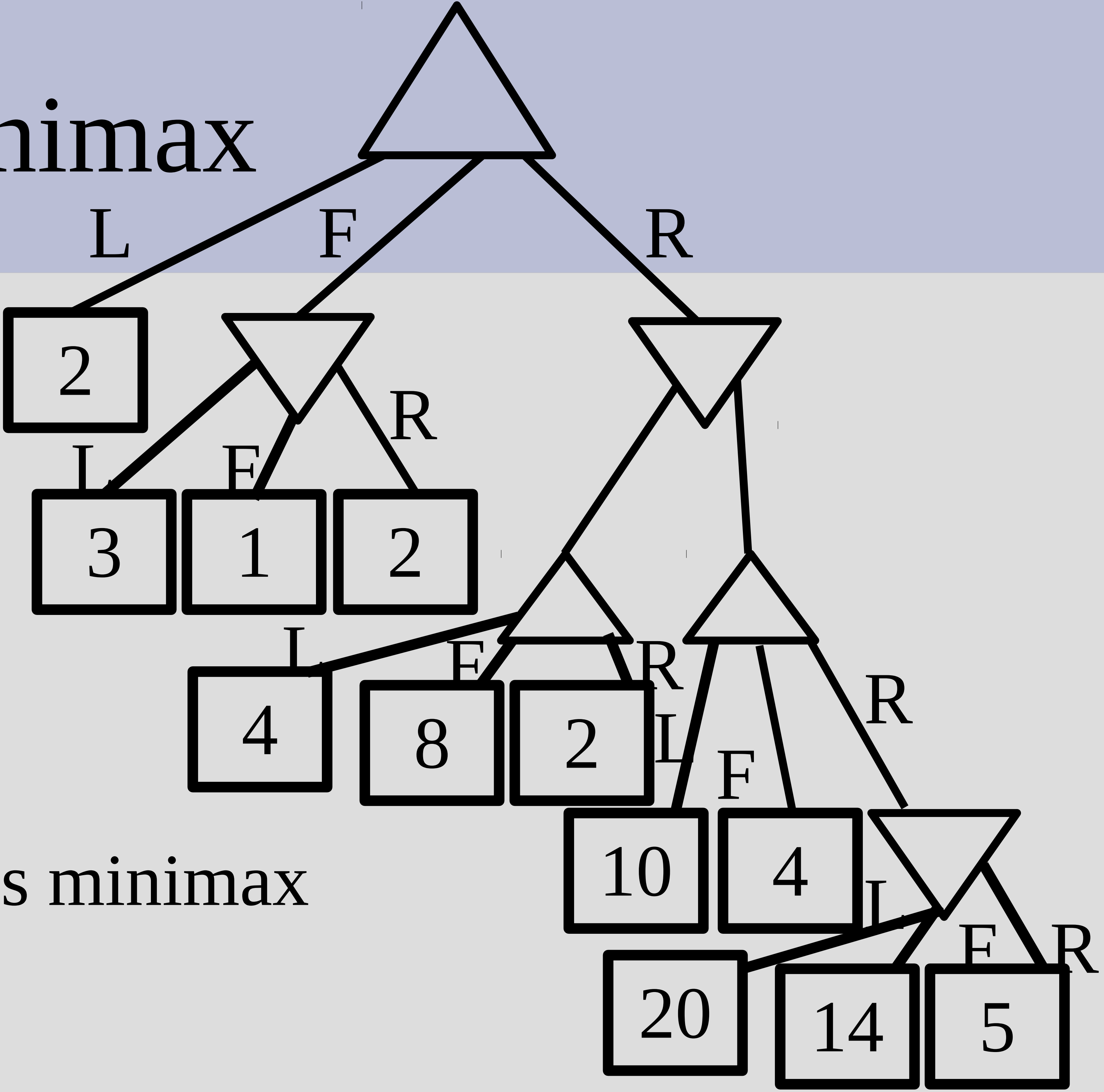
1st. R (can swap

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)

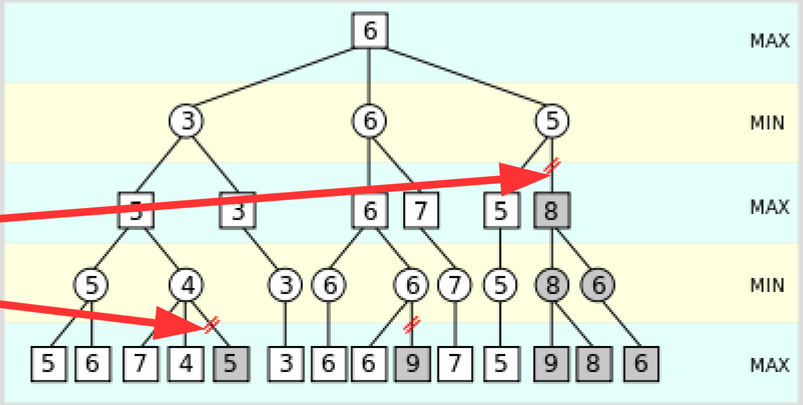
Minimax

"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

This can apply to max nodes as well, so we propagate the best values for max/min in tree

Alpha-beta pruning algorithm:

Do minimax as normal, except:

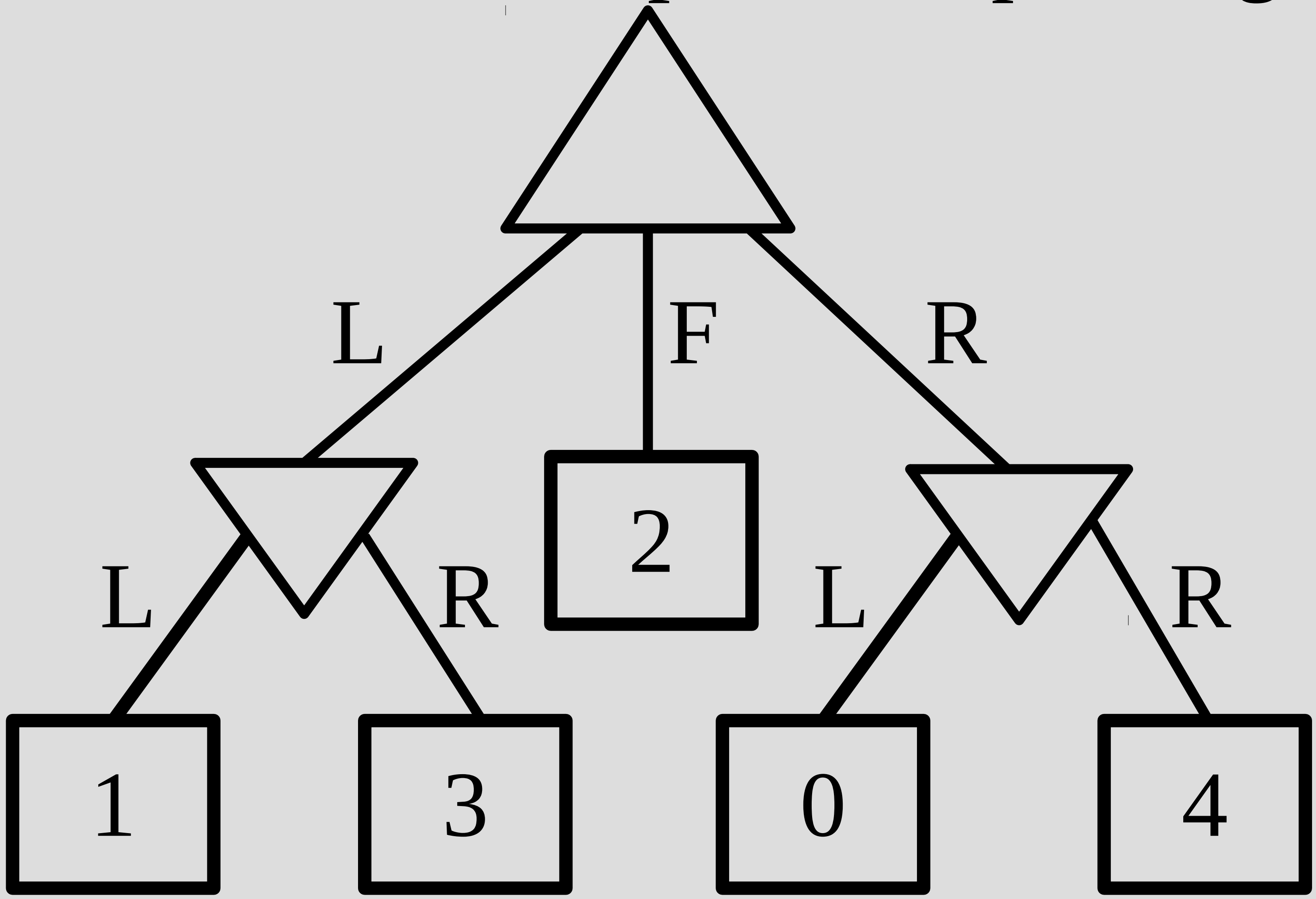
Going down tree: pass “best max/min” values

min node: if parent's “best max” greater than current node, go back to parent immediately

max node: if parent's “best min” less than current node, go back to parent immediately

Alpha-beta pruning

Let's solve this with alpha-beta pruning



Alpha-beta pruning

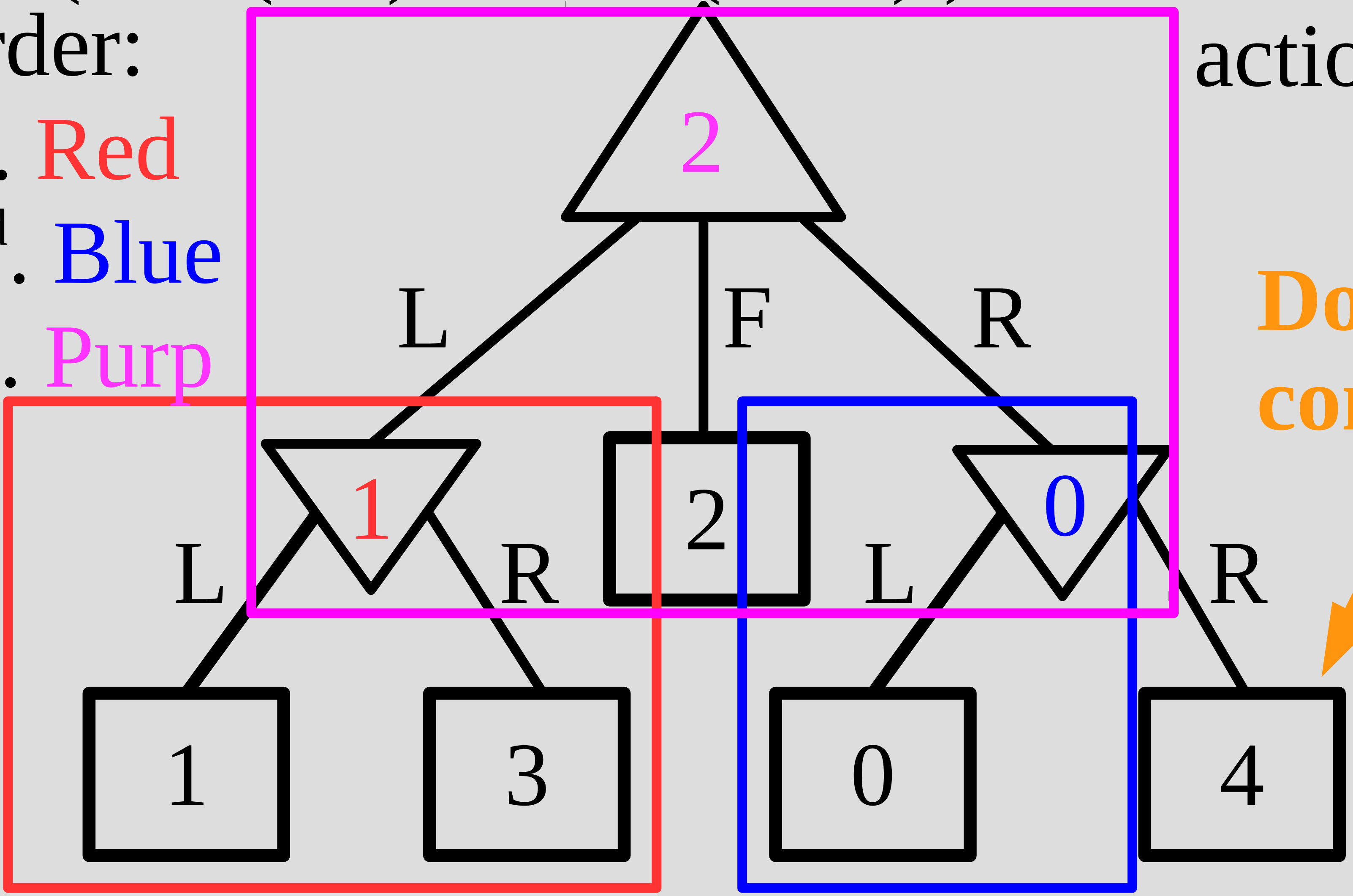
max(min(1,3), 2, min(0, ??)) = 2, should pick action F

Order:

1st. Red

2nd. Blue

3rd. Purp

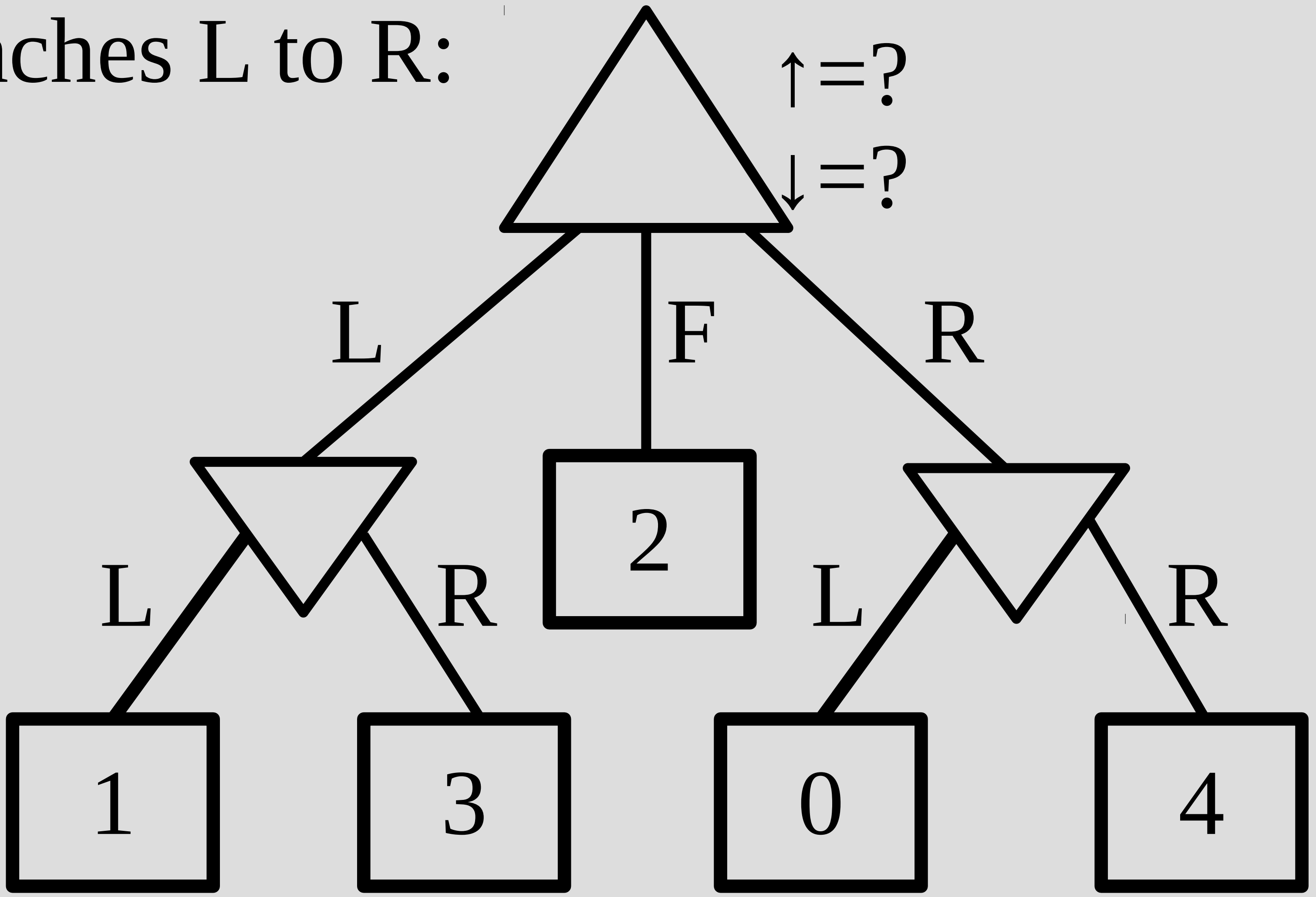


Do not consider

Alpha-beta pruning

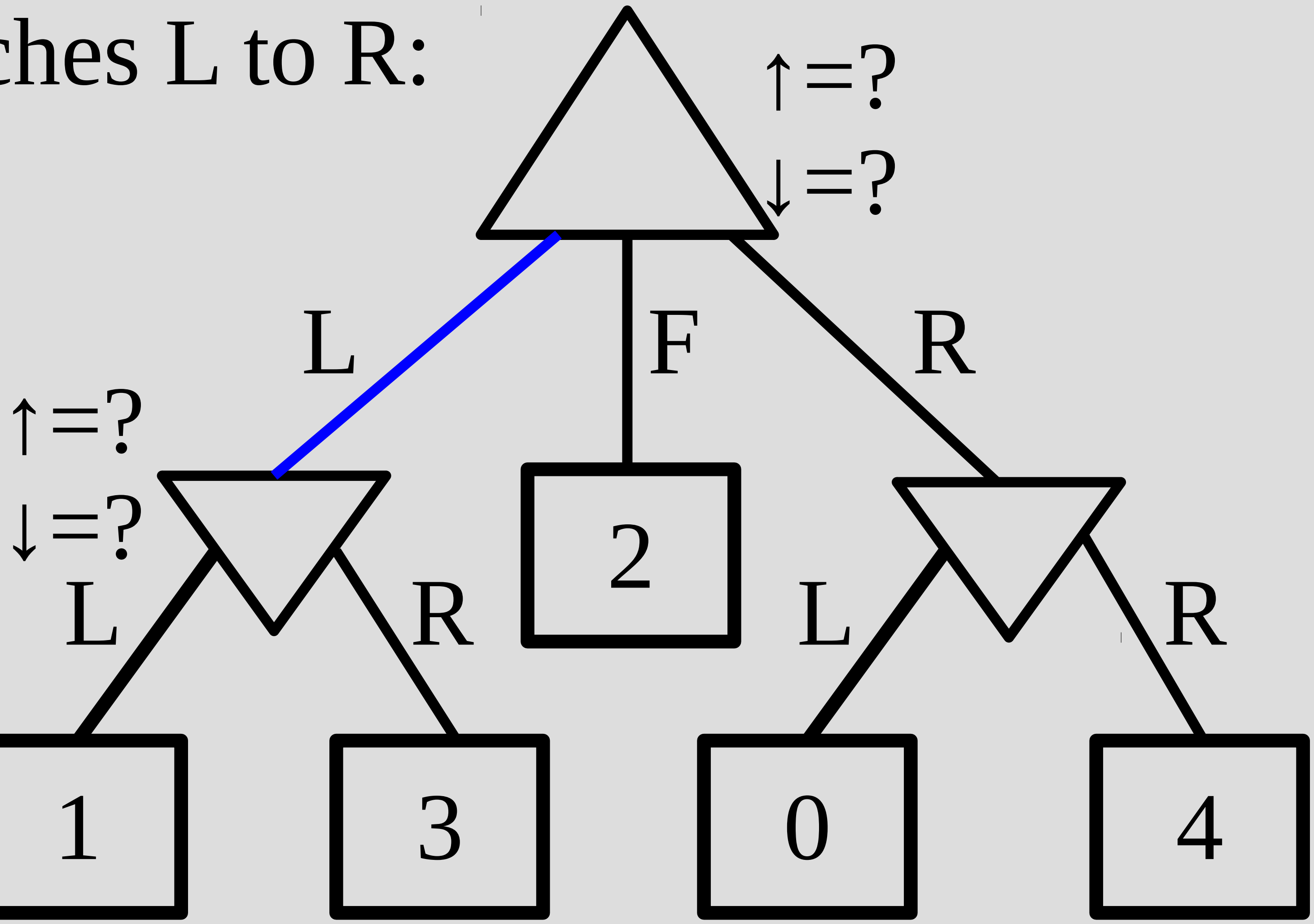
Let best max be "↑" and best min be "↓"

Branches L to R:



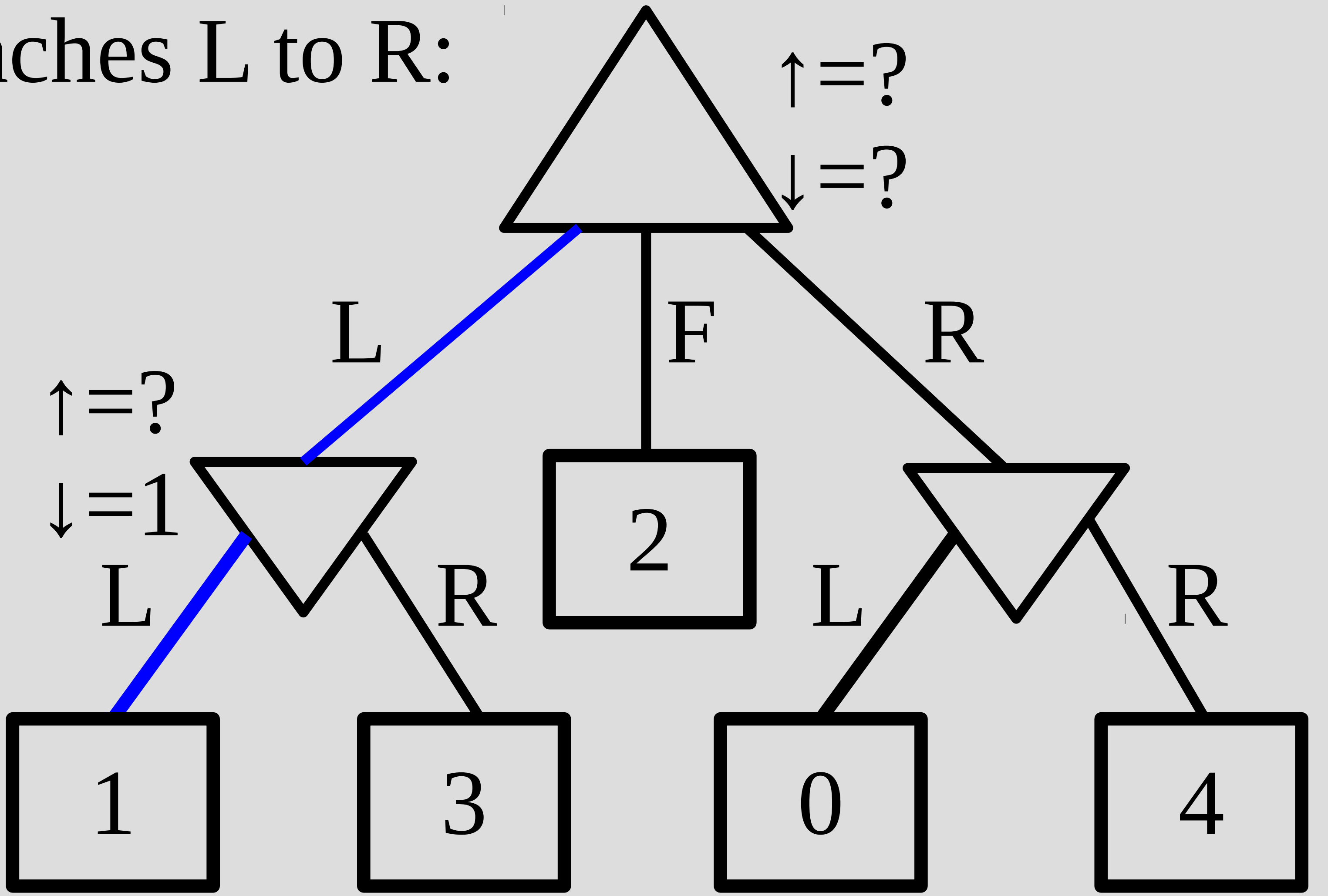
Alpha-beta pruning

Let best max be "↑" and best min be "↓"
Branches L to R:



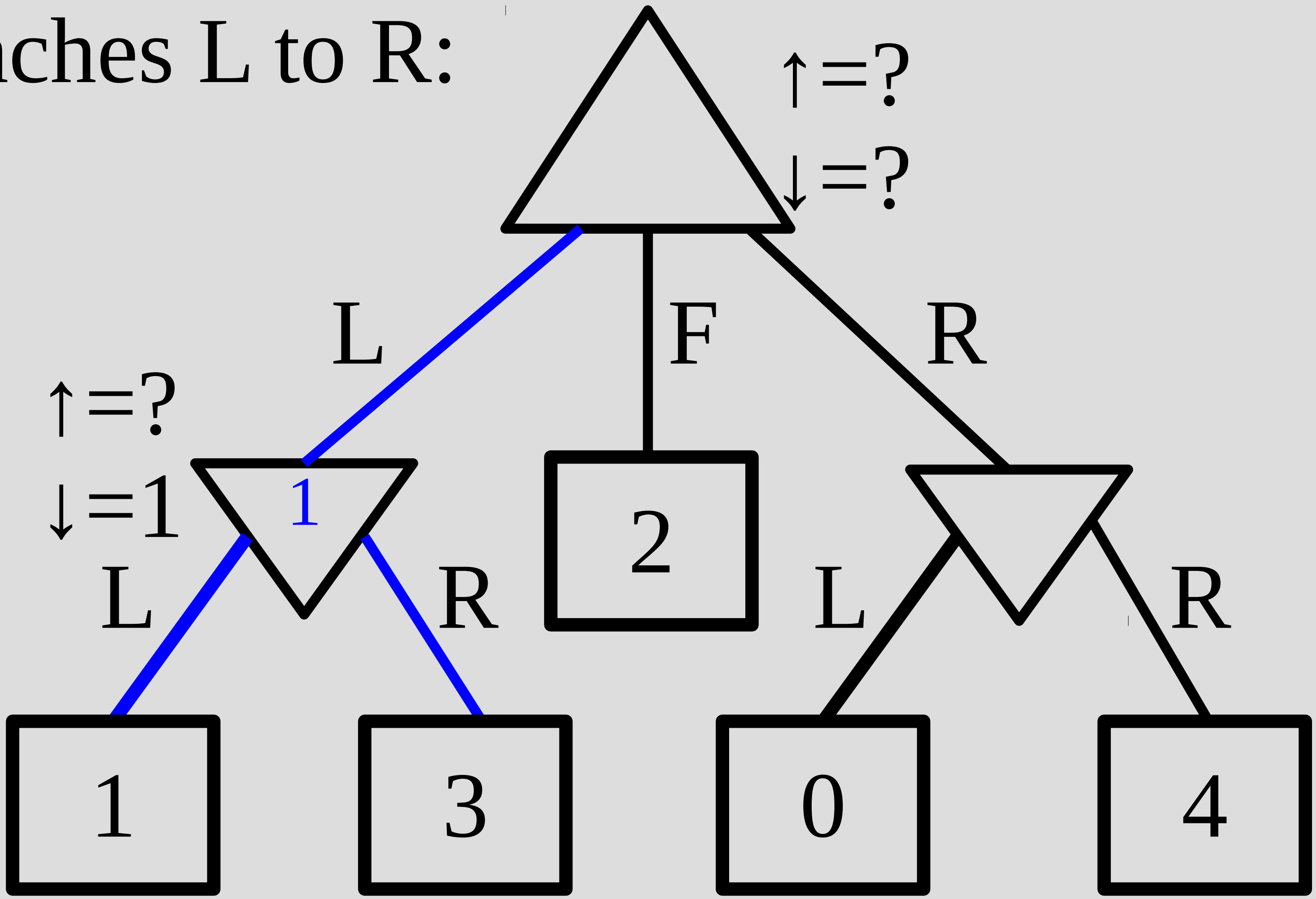
Alpha-beta pruning

Let best max be "↑" and best min be "↓"
Branches L to R:



Alpha-beta pruning

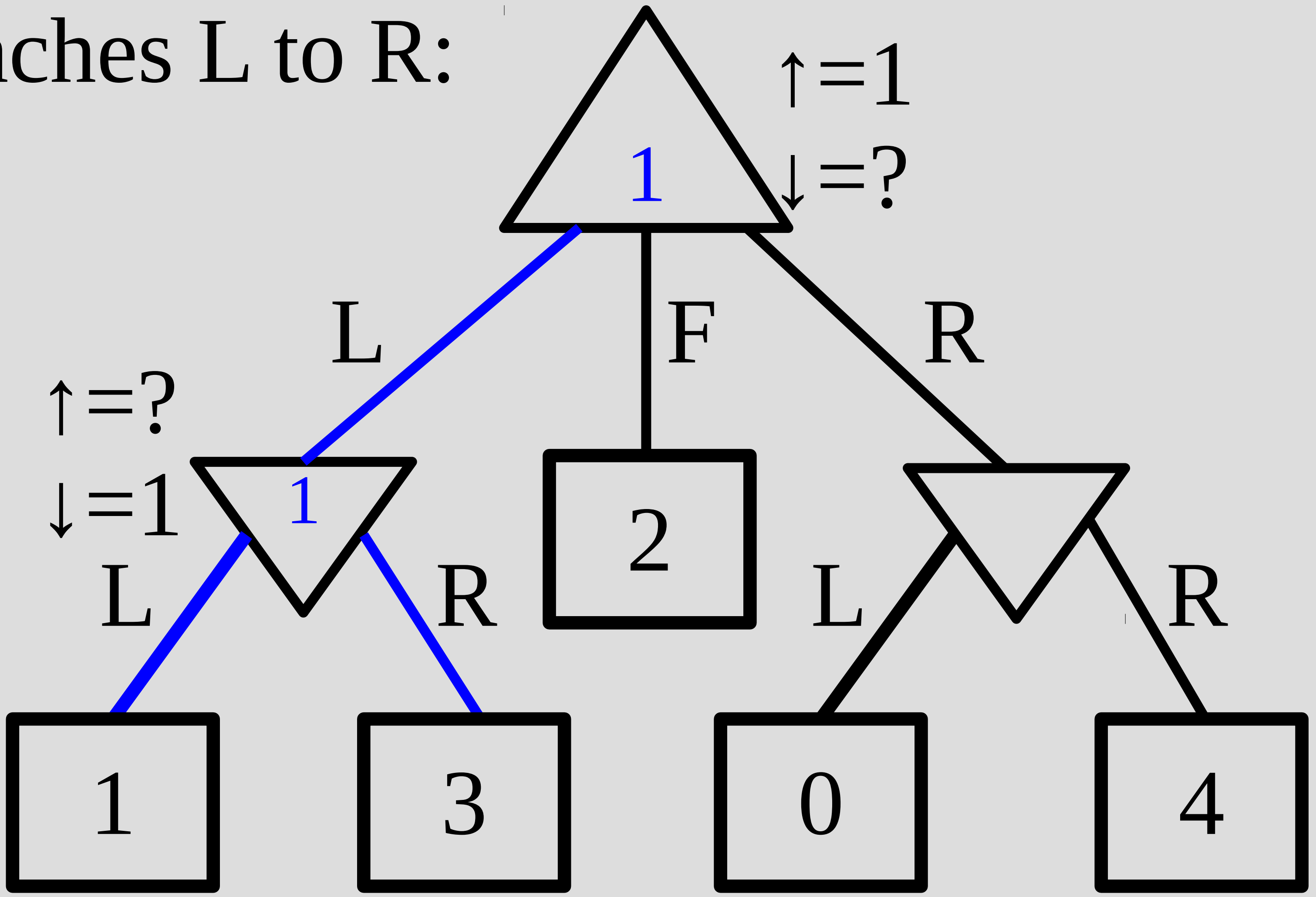
Let best max be "↑" and best min be "↓"
Branches L to R:



Alpha-beta pruning

Let best max be "↑" and best min be "↓"

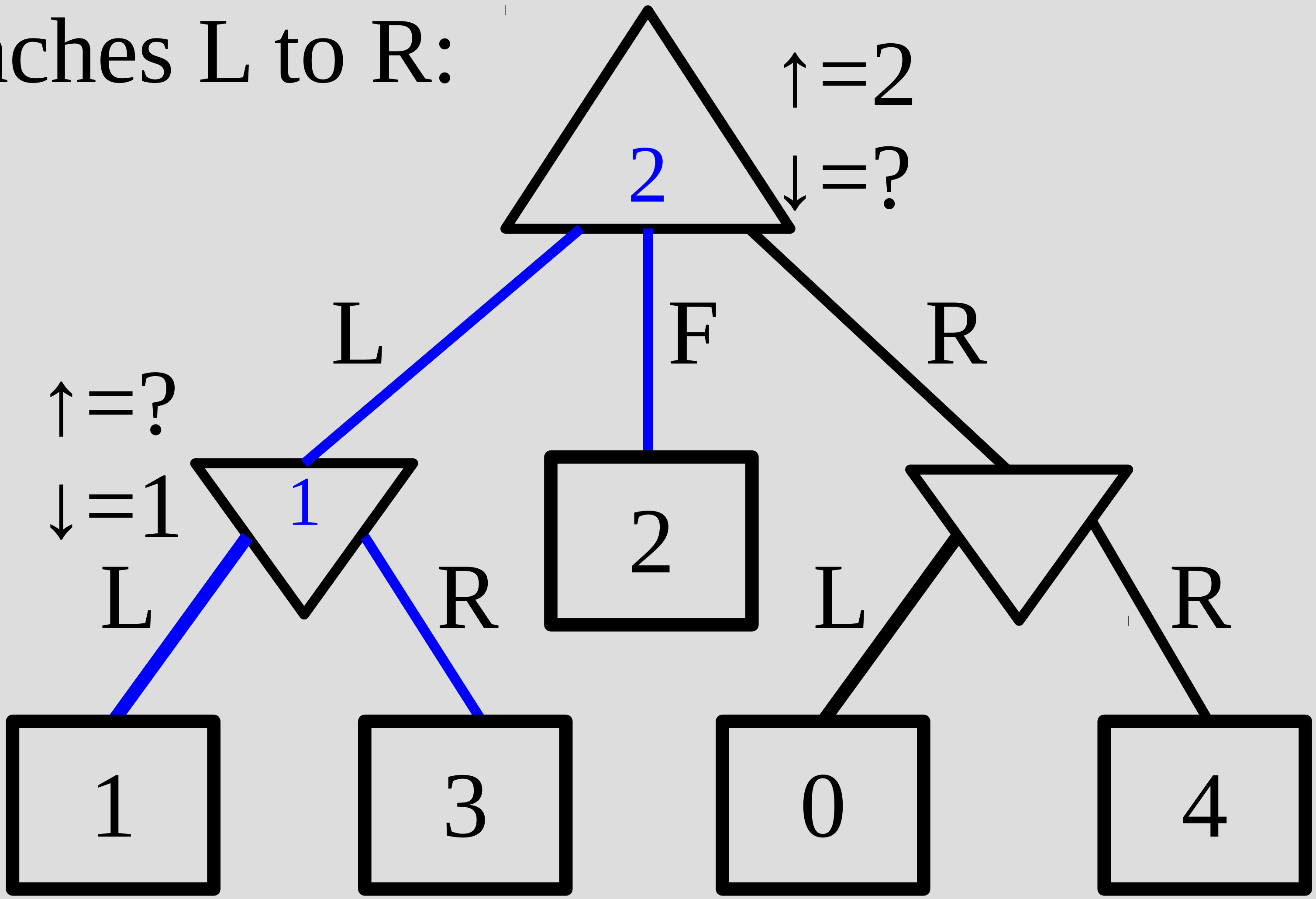
Branches L to R:



Alpha-beta pruning

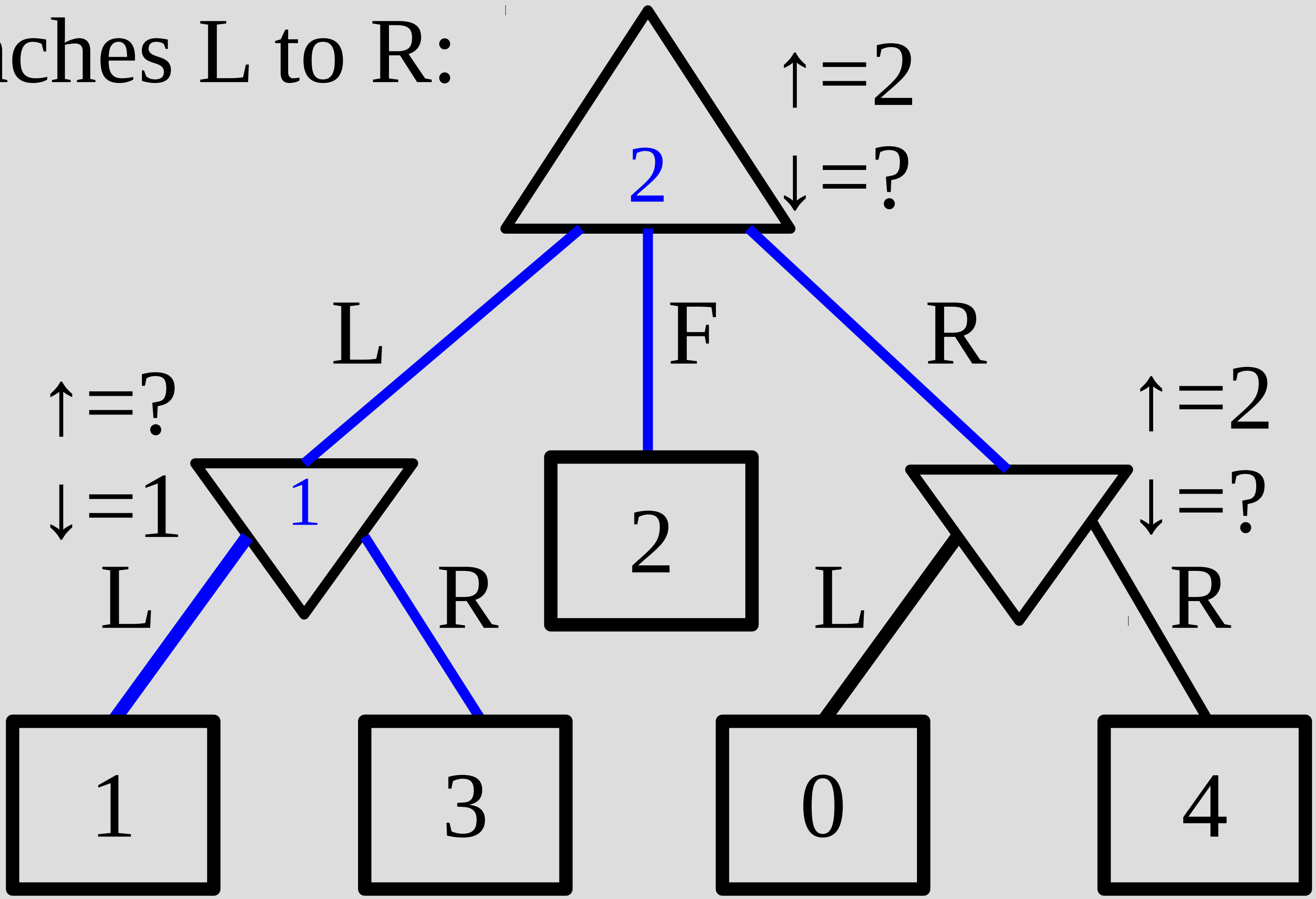
Let best max be "↑" and best min be "↓"

Branches L to R:



Alpha-beta pruning

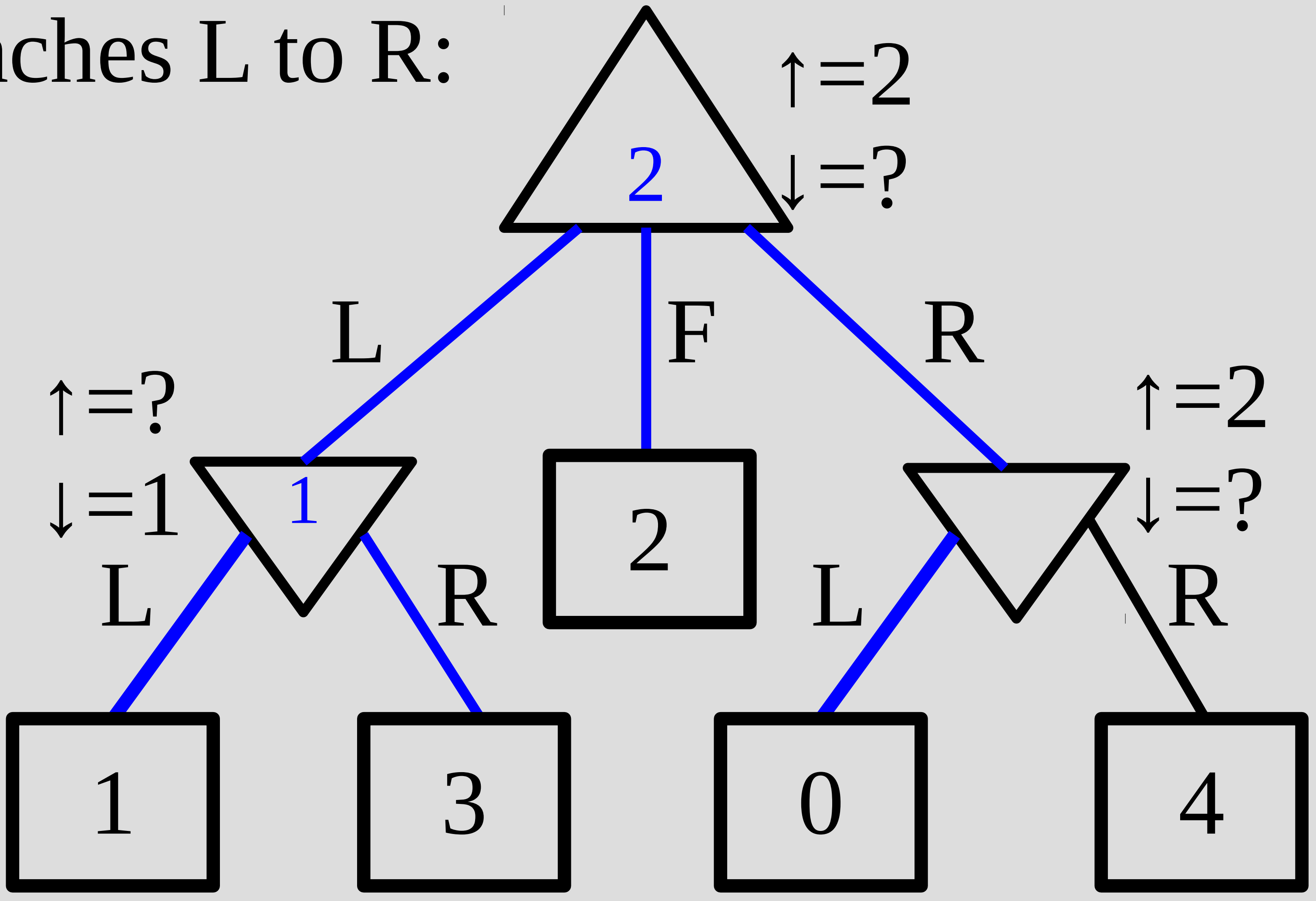
Let best max be "↑" and best min be "↓"
Branches L to R:



Alpha-beta pruning

Let best max be "↑" and best min be "↓"

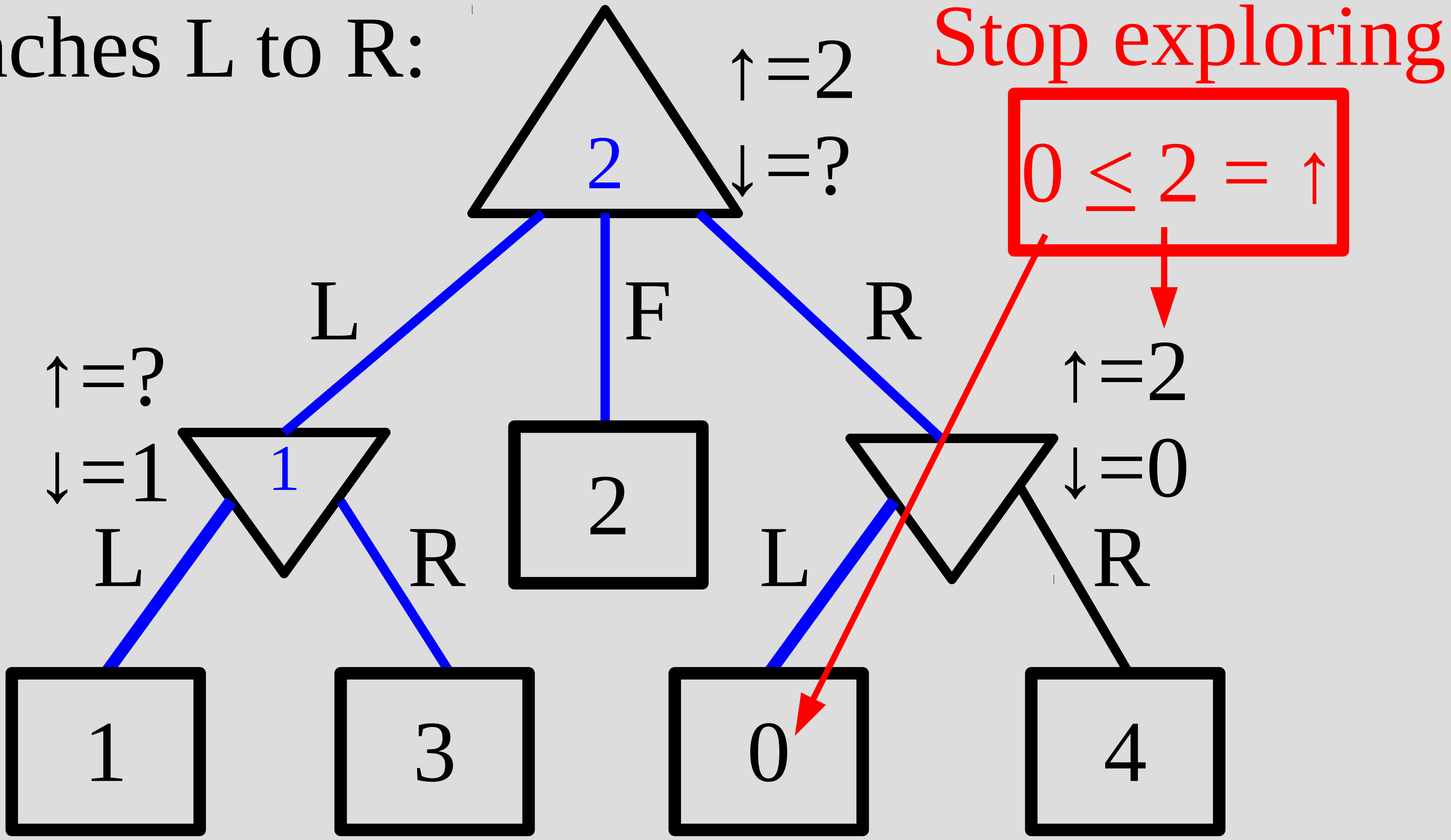
Branches L to R:



Alpha-beta pruning

Let best max be "↑" and best min be "↓"

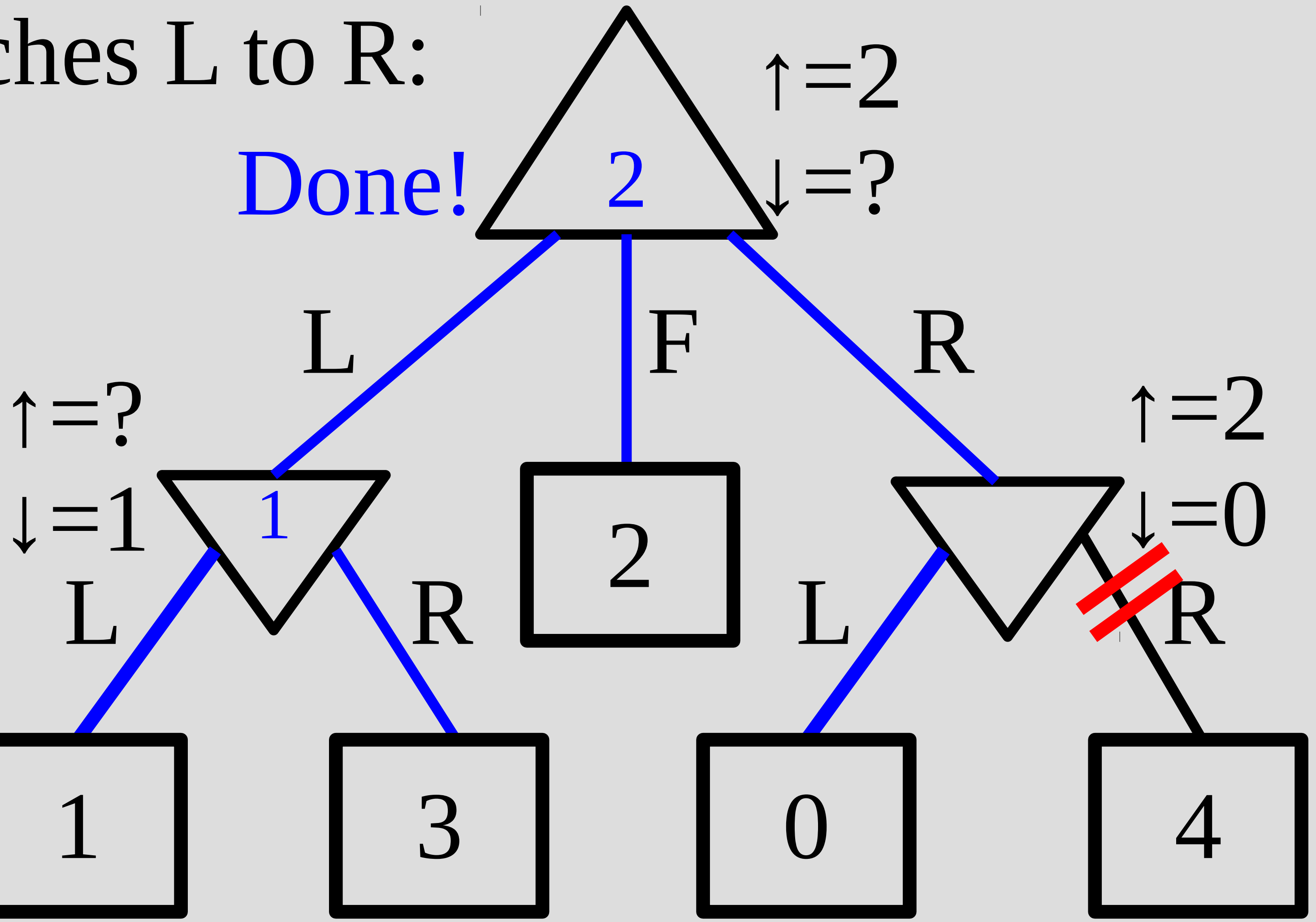
Branches L to R:



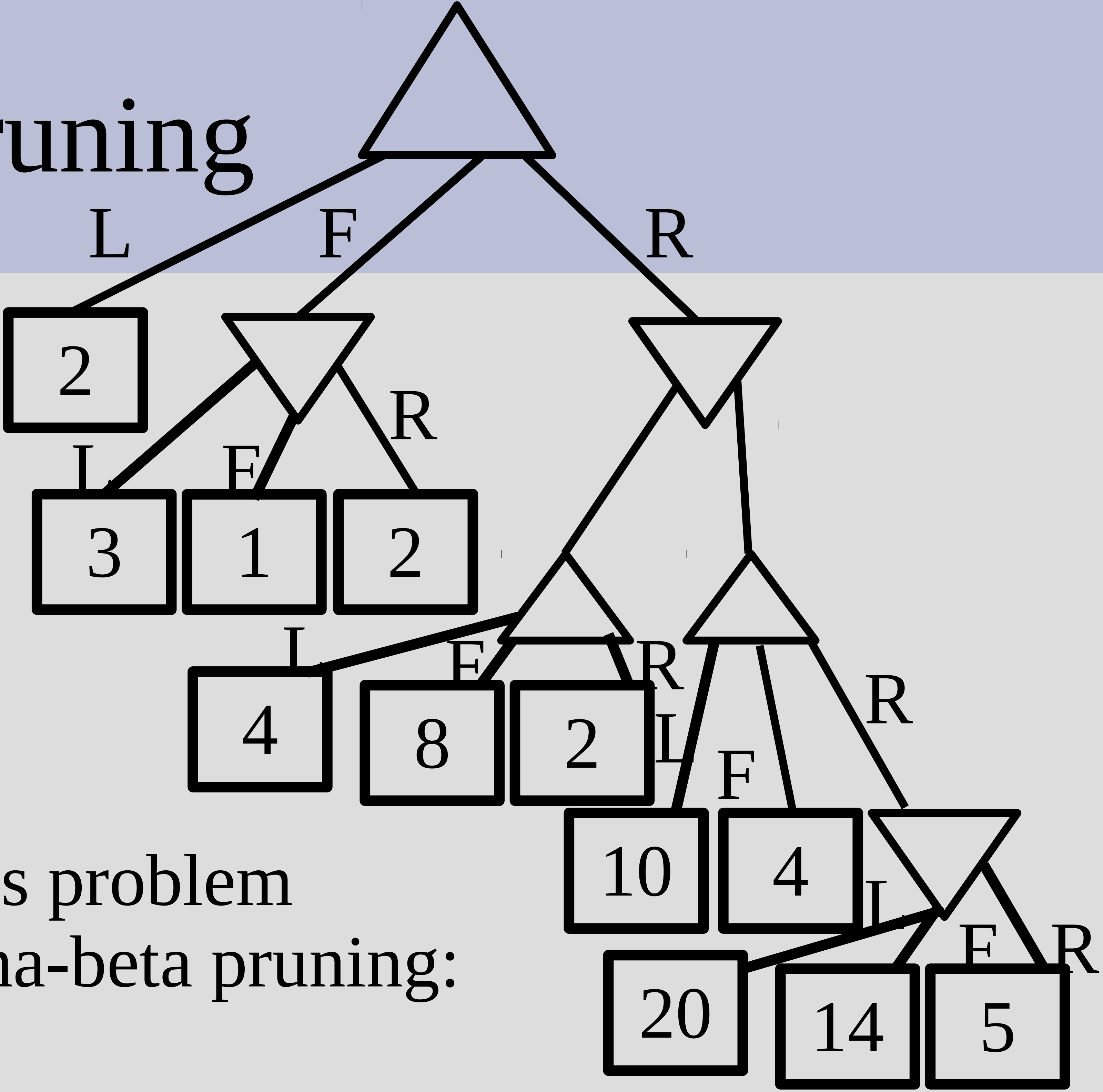
Alpha-beta pruning

Let best max be "↑" and best min be "↓"

Branches L to R:



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: b^m
Then, alpha-beta searches: $b^{m/2}$

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)

Alpha-beta pruning

This rule of checking your parent's best/worst with the current value in the child only really works for two player games...

What about 3 player games?

3-player games

For more than two player games, you need to provide values at every state for all the players

When it is the player's turn, they get to pick the action that maximizes their own value the most

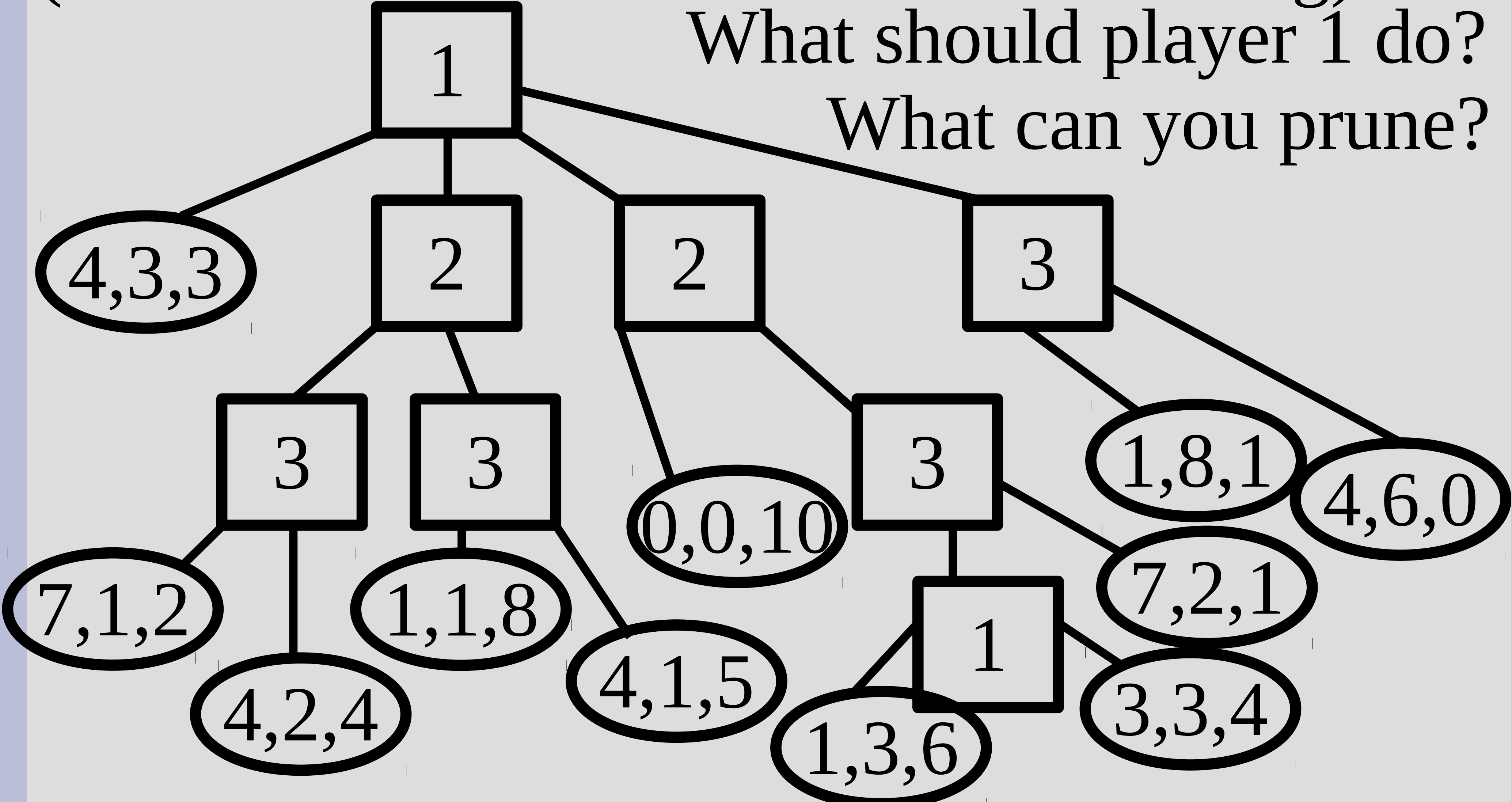
(We will assume each agent is greedy and only wants to increase its own score... more on this next time)

3-player games

(The node number shows who is max-ing)

What should player 1 do?

What can you prune?



3-player games

How would you do alpha-beta pruning in a 3-player game?

3-player games

How would you do alpha-beta pruning in a 3-player game?

TL;DR: Not easily

(also you cannot prune at all if there is no range on the values even in a zero sum game)

This is because one player could take a very low score for the benefit of the other two

Mid-state evaluation

So far we assumed that you have to reach a terminal state then propagate backwards (with possibly pruning)

More complex games (Go or Chess) it is hard to reach the terminal states as they are so far down the tree (and large branching factor)

Instead, we will estimate the value minimax would give without going all the way down

Mid-state evaluation

By using <u>mid-state evaluations</u> (not terminal) the “best” action can be found quickly

These mid-state evaluations need to be:

1. Based on current state only
2. Fast (and not just a recursive search)
3. Accurate (represents correct win/loss rate)

The quality of your final solution is highly correlated to the quality of your evaluation

Mid-state evaluation

For searches, the heuristic only helps you find the goal faster (but A* will find the best solution as long as the heuristic is admissible)

There is no concept of “admissible” mid-state evaluations... and there is almost no guarantee that you will find the best/optimal solution

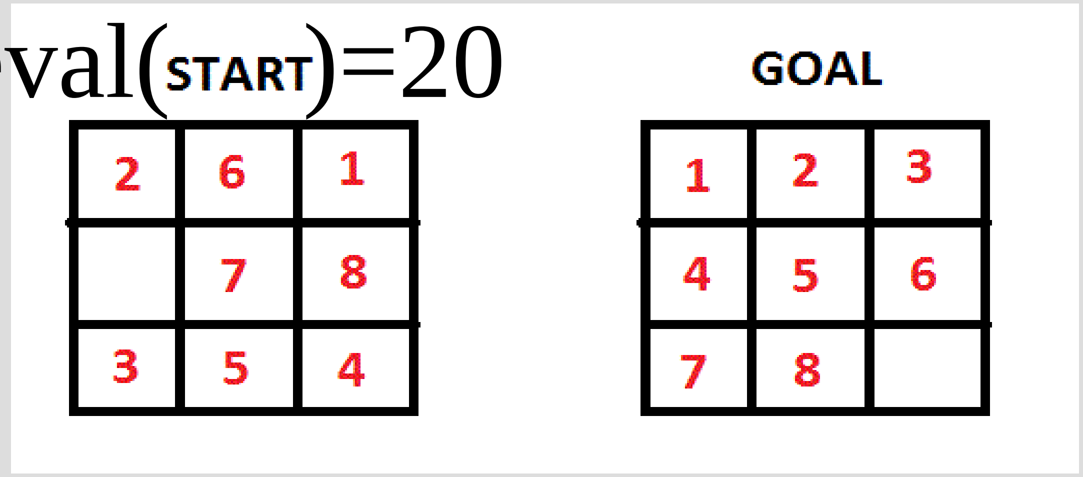
For this reason we only apply mid-state evals to problems that we cannot solve optimally

Mid-state evaluation

A common mid-state evaluation adds features of the state together

(we did this already for a heuristic...)

eval(START)=20



We summed the distances to the correct spots for all numbers

Mid-state evaluation

We then minimax (and prune) these mid-state evaluations as if they were the correct values

You can also weight features (i.e. getting the top row is more important in 8-puzzle)

A simple method in chess is to assign points for each piece: pawn=1, knight=4, queen=9... then sum over all pieces you have in play

Mid-state evaluation

What assumptions do you make if you use a weighted sum?

Mid-state evaluation

What assumptions do you make if you use a weighted sum?

A: The factors are independent
(non-linear accumulation is common if the relationships have a large effect)

For example, a rook & queen have a synergy bonus for being together is non-linear, so queen=9, rook=5... but queen&rook = 16

Mid-state evaluation

There is also an issue with how deep should we look before making an evaluation?

Mid-state evaluation

There is also an issue with how deep should we look before making an evaluation?

A fixed depth? Problems if child's evaluation is overestimate and parent underestimate (or visa versa)

Ideally you would want to stop on states where the mid-state evaluation is most accurate

Mid-state evaluation

Mid-state evaluations also favor actions that “put off” bad results (i.e. they like stalling)

In go this would make the computer use up ko threats rather than give up a dead group

By evaluating only at a limited depth, you reward the computer for pushing bad news beyond the depth (but does not stop the bad news from eventually happening)

Mid-state evaluation

It is not easy to get around these limitations:

1. Push off bad news
2. How deep to evaluate?

A better mid-state evaluation can help compensate, but they are hard to find

They are normally found by mimicking what expert human players do, and there is no systematic good way to find one

Forward pruning

You can also use mid-state evaluations for alpha-beta type pruning

However as these evaluations are estimates, you might prune the optimal answer if the heuristic is not perfect (which it won't be)

In practice, this prospective pruning is useful as it allows you to prioritize spending more time exploring hopeful parts of the search tree

Forward pruning

You can also save time searching by using “expert knowledge” about the problem

For example, in both Go and Chess the start of the game has been very heavily analyzed over the years

There is no reason to redo this search every time at the start of the game, instead we can just look up the “best” response

Random games

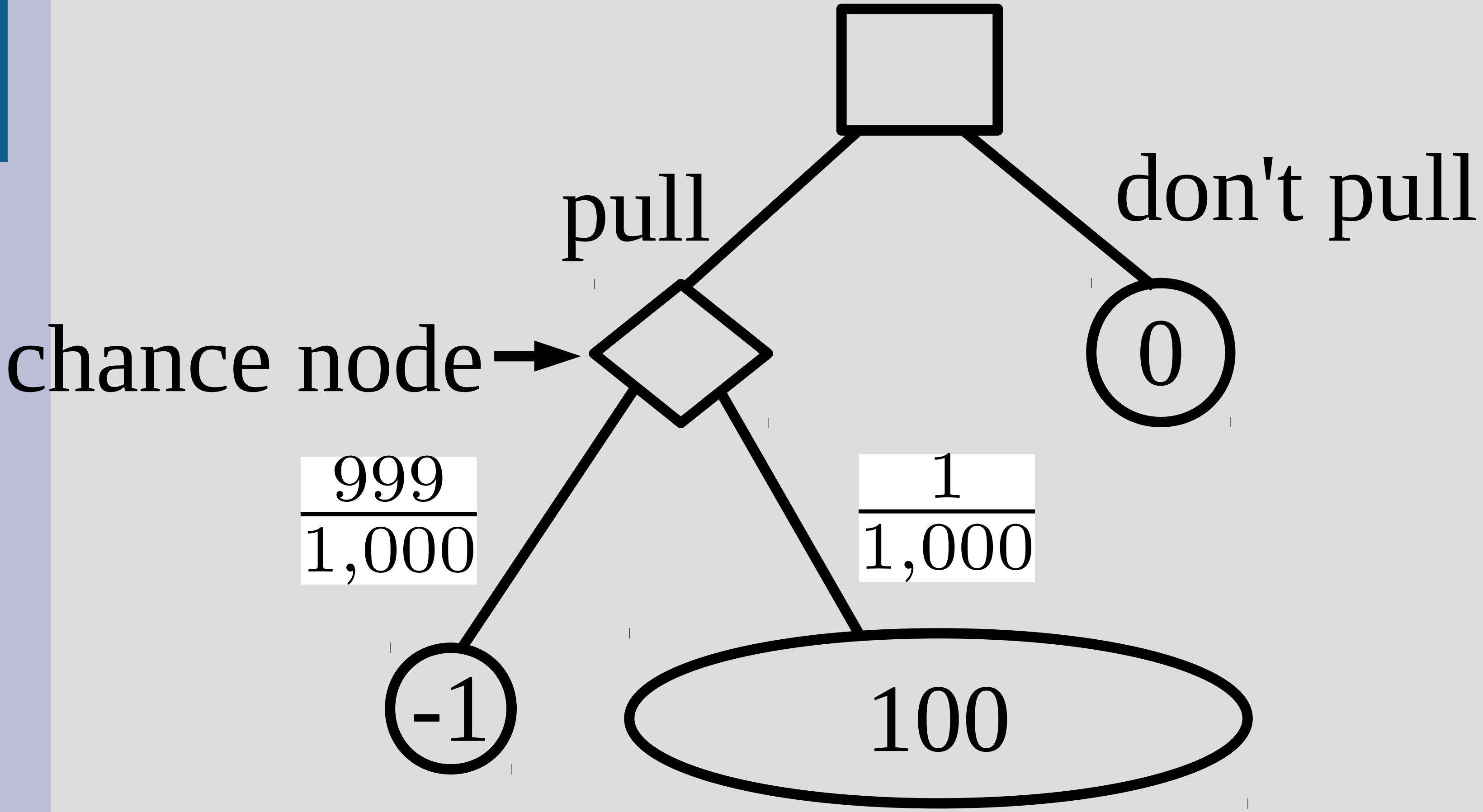
If we are playing a “game of chance”, we can add <u>chance nodes</u> to the search tree

Instead of either player picking max/min, it takes the expected value of its children

This expected value is then passed up to the parent node which can choose to min/max this chance (or not)

Random games

Here is a simple slot machine example:

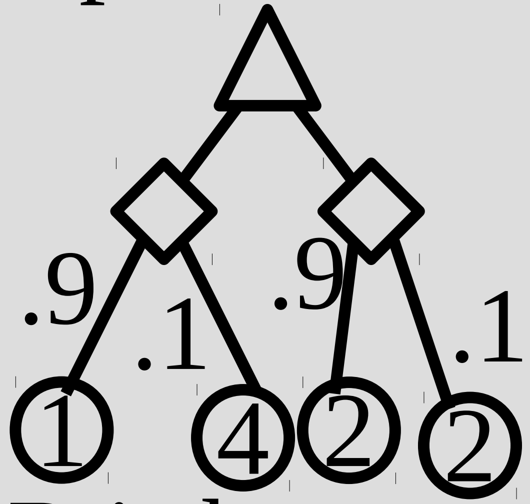


V(chance) = $-1 \cdot \frac{999}{1{,}000} + 100 \cdot \frac{1}{1{,}000} = -0.899$

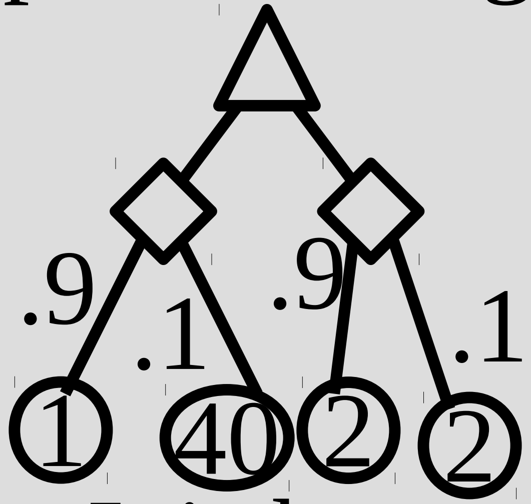
Random games

You might need to modify your mid-state evaluation if you add chance nodes

Minimax just cares about the largest/smallest, but expected value is an implicit average:



R is better



L is better

Random games

Some partially observable games (i.e. card games) can be searched with chance nodes

As there is a high degree of chance, often it is better to just assume full observability (i.e. you know the order of cards in the deck)

Then find which actions perform best over all possible chance outcomes (i.e. all possible deck orderings)

Random games

For example in blackjack, you can see what cards have been played and a few of the current cards in play

You then compute all possible decks that could lead to the cards in play (and used cards)

Then find the value of all actions (hit or stand) averaged over all decks (assumed equal chance of possible decks happening)

Random games

If there are too many possibilities for all the chance outcomes to “average them all”, you can sample

This means you can search the chance-tree and just randomly select outcomes (based on probabilities) for each chance node

If you have a large number of samples, this should converge to the average