Binary tree optimization using evolutionary operators

Kuba Podgórski

Abstract

The research on binary trees and searching information to try answer the question on what is the best binary search tree. In other words, the tree where the *search cost* is minimal. In this case, the *search cost* refers to the number of needed comparisons, to find a certain element.
Deterministic algorithms like Knuth's dynamic programming algorithm is capable of constructing the statically optimal tree in $O(n^2)$ time, so it becomes impractical when the number of elements in the tree is very large. For the last couple of years new ideas related to genetics and evolution theory have gained in popularity. Genetic algorithms (GA) are based on natural selection mechanisms together with genetic operators make searching for results close to optimal.
Our work is kind of an experiment, checking for what cases optimization binary trees by genetic and evolutionary operators makes sense.

1 Definitions

First, we need to introduce basic terms related to genetic and evolutionary operators.

Chromosome In **GA**, a chromosome is often represented as a binary string (encoded by the alphabet of $\{0, 1\}$).

Crossover Genetic operator used to exchange genetic information between parents (randomly selected from a population), to generate a new offspring.

Mutation Genetic operator alters one or more gene values. A mutation occurs with very low probability (0.01). The purpose of mutation is to prevent the population of chromosomes from becoming too similar to each other.

Selection The stage in which genomes are chosen from the population for the next operation. Selection is based on pre evaluated fitness function for each genome. It can be implemented in many different ways. The most common are fitness proportionate selection or roulette-wheel selection (when we repeat the procedure until there are enough selected individuals), stochastic selection, tournament selection (when we select the best individual of a randomly chosen subset).

1.1 Termination condition

In **GA**, the optimization process is repeated until the termination condition has been reached. For instance, a process can be terminated if the solution found satisfies the predefined value, e.g.:

$$f(X(t)) \geq f_s$$

where: f_s is a predefined value, $X(t)$ is the best solution found in time t. This kind of condition is hard to apply, mainly because it is hard to define f_s value.
Another condition is based on the observation of how fast the evaluation improves. A process is terminated if in the last n generations the evaluation has not improved more than predefined ϵ:

$$|f(X(t-n)) - f(X(t))| \leq \epsilon$$

1.2 Difference between genetic and evolutionary algorithms

In classical **GA** it is common to assume that a chromosome is represented as a *fixed length* sequence of bits. Moreover, only two operators are allowed: *mutation* and *crossover*. Additionally, classical genetic operators only modify the sequence of bits and do not consider any properties of the encoded data structure.
Another difference between **GA** and **EA** is related to the selection process.

GA selects as many genomes as the size of the population. Genomes can be duplicated. In other words, *strong* genomes can be selected again to the new population however the weakest genome can also be selected. In evolutionary strategies selection is deterministic (compared to random selection which takes place in genetic algorithms). Only the best genomes (no duplication) are selected.
Normalization also works differently. In evolutionary strategies, the selection reduces the *middle-stage* population to the fixed number of genomes. In genetic algorithms, the *middle-stage* population will be selected first, then the genetic operators will be applied.

2 Encoding binary trees to sequences

Let T be a *complete* binary tree. For each *internal node* we assign label 1 and for each *leaf* a label 0.

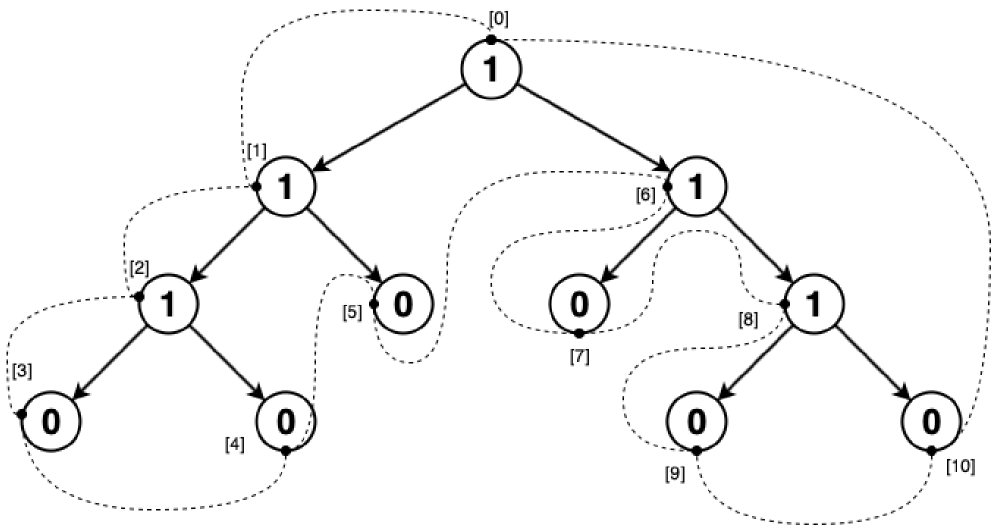


Figure 1: preorder traversal of complete binary tree T

If we traverse the tree T in *preorder* and read labels for every node, then we will get $x-sequence$ of T with n ones and $n+1$ zeros.
Let $\{x\}_T^{2n}$ be $x-sequence$ built from tree T with n *internal nodes*. And $\{x_i\}_T^{2n}$ is $i-th$ element of $x-sequence$. A $y-sequence$ is built from $x-sequence$ as a sequence of indexes with label zero ($\{y_i\}_T^n$ defines $i-th$ leaf in the tree

T). Similarly to $y-sequence$ a $z-sequence$ is built as a sequence of indexes with label one ($\{z_i\}_T^n$ defines $i-th$ internal node in the tree T). For instance, for the tree T (on Figure 1) we can build following sequences:

$\{x\}_T^{10} = 1\ 1\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 0$

$\{y\}_T^{5} = 3\ 4\ 5\ 7\ 9\ 10$

$\{z\}_T^{5} = 0\ 1\ 2\ 6\ 8$

Algorithm 1 Create a genome from binary tree

```
$genome(bt : BinaryTree) : []Gene$
    if $bt.node = NIL$ then
        return $NIL$
    end if
    $g \leftarrow []Gene((bt.node.key, bt.node.weight))$
    if $bt.left \neq NIL$ then
        if $genome(bt.left) \neq NIL$ then
            $g \leftarrow append(g, genome(bt.left)...)$
        else
            $g \leftarrow append(g, NIL)$
        end if
    end if
    if $bt.right \neq NIL$ then
        if $genome(bt.right) \neq NIL$ then
            $g \leftarrow append(g, genome(bt.right)...)$
        else
            $g \leftarrow append(g, NIL)$
        end if
    end if
    return $g$
```

3 Binary tree optimization

Let p_i ($\sum p_i = 1$) be a *visit probability* of *i-th* node in a binary tree. Our optimization task is to reorganize a tree in such a way that a sum of paths to each node is minimal. Every node keeps an extra weight w_i, which stands for frequency of visiting the *i-th* node. So, our fit function, which will be minimized may look like:

$$eval_T(n) = \sum_{i=0}^{n-1} w_i(h_i + 1) \quad , \tag{1}$$

where n is a number of nodes in a tree, w_i is a weight of *i-th* node, and h_i is a height of *i-th* node in a tree.

For example, we have three different binary trees with the following weights: $w_1 = 1$, $w_2 = 3$, $w_3 = 5$. Let's assign weights to nodes and evaluate trees.

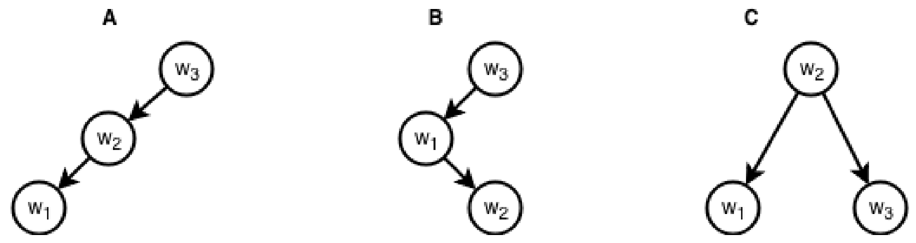


Figure 2: binary trees with three nodes

$$eval_A(3) = 5 \cdot (0+1) + 3 \cdot (1+1) + 1 \cdot (2+1) = 14 \tag{2}$$

$$eval_B(3) = 5 \cdot (0+1) + 1 \cdot (1+1) + 3 \cdot (2+1) = 16 \tag{3}$$

$$eval_C(3) = 3 \cdot (0+1) + 1 \cdot (1+1) + 5 \cdot (1+1) = 15 \tag{4}$$

In our example, the tree A was chosen as optimal, because the minimum value was returned by the fit function $eval^A$.

The most common optimal binary trees are used for dictionaries or lexical analyzers. In most cases, these are *binary search trees (BST).* We also know that a dictionary does not contain all words, so we should also consider *missed lookups.* The most important is to minimize the time to say the *key* does not exist in a tree. Let's adjust our previous fit function by considering not only hits (successes) but also failures (misses).

$$eval_T(n) = \underbrace{\sum_{i=0}^{n-1} w_i(h_i + 1)}_{hits} + \underbrace{\sum_{j=0}^{n} w'_j(h'_j + 1)}_{misses}$$

where w_i is a weight - how many times argument x was equal i-th key (a_i), w'_j - how many times x was between a_{j-1} and a_j. Moreover, w'_0 stands for number of times x was less than a_0, w'_n how many times x was greater than a_{n-1}, h_i is a height of i-th internal node and h'_j is a height of j-th external node (leaf).

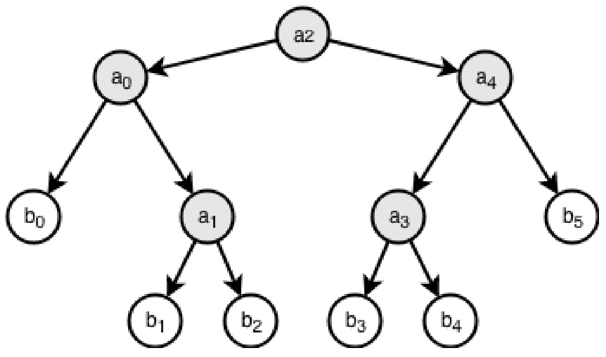


Figure 3: A binary search tree with visit frequencies

Algorithm 2 Evaluates a given genome

```
$eval(genome : []Gene) : Int$   {fitness function}
    $f \leftarrow func(h : Int) : Int$
        **if** $len(genome) = 0$ **then**
            **return** $0$
        **end if**
        **if** $empty(genome[0])$ **then**
            **return** $h * genome[0].weight$
        **end if**
        $s \leftarrow h * genome[0].weight$
        $genome \leftarrow genome[1:]$
        $s \leftarrow s + f(h+1)$
        $genome \leftarrow genome[1:]$
        $s \leftarrow s + f(h+1)$
        **return** $s$
    **return** $f(1)$
```

3.1 Encoding and representation

We chose $x - sequence$ method to encode binary trees because it uses the smallest alphabet. Chromosome indexes (represented as $x - sequence$) start with 0, so the example binary tree on figure 3 is represented by the following chromosome: (1_0 1_1 0_2 1_3 0_4 0_5 1_6 1_7 0_8 0_9 0_{10}).

Let all encoded binary trees be *complete binary trees*. In other words, for each node which does not have two sons (leaf or internal node with just one son), we will add *empty nodes* (with no information). We will work with a complete binary tree with $2N + 1$ nodes. After that, we will encode the complete tree to the chromosome with length $2N + 1$. Genes marked as 1 are internal nodes (with information), genes marked as 0 are leaves (with no information). Every gene contains *weight* (visit frequency) and *is empty* (boolean information). It is easy to notice that all chromosomes generated from non-empty binary trees match a pattern $P_T = (1\underbrace{\star\star\star\ldots\star}_{2N-2}00)$, where $N > 0$ is a number of non-empty nodes.

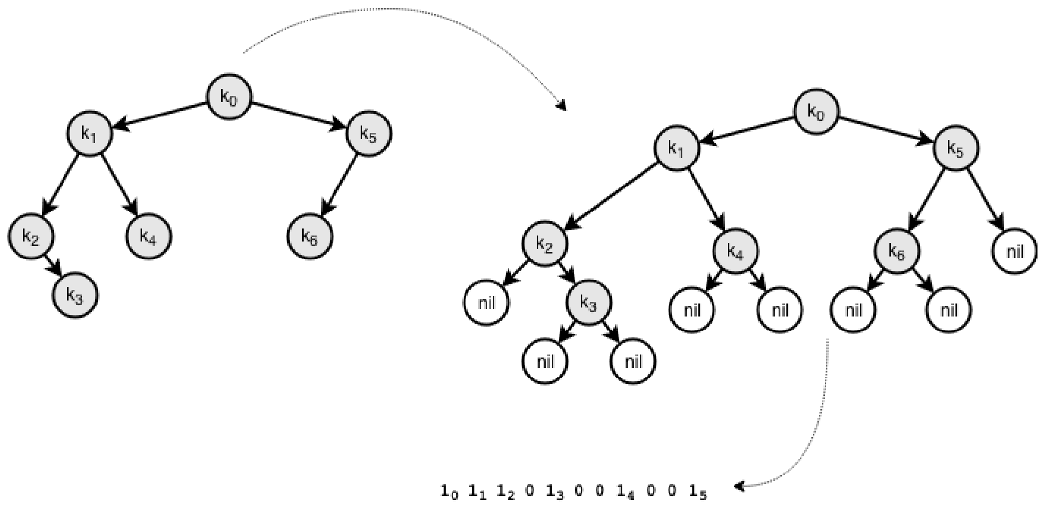
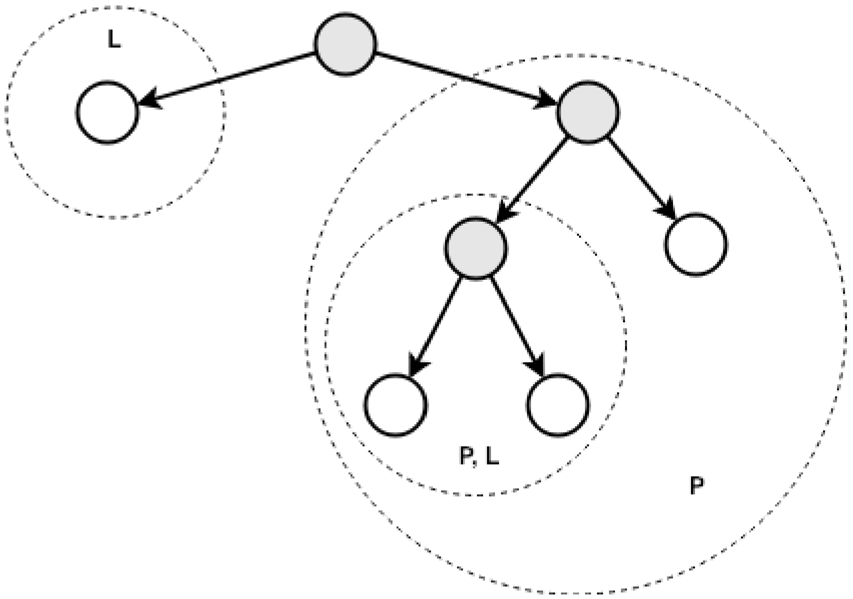


Figure 4: Conversion into a complete binary tree and encoded to $x-sequence$

All optimal binary trees have one very important property - all sub-trees are also optimal. This property lets us compose multiple optimal trees into one big optimal tree. The problem of how to find a sub-tree in a chromosome can be solved by finding the first gene from which the number of zeros is greater than the number of ones (counting from the next first non-root node, e.g.:



$$(1 \quad \underbrace{0}_{L} \quad \underbrace{1 \quad \underbrace{1 \quad 0 \quad 0}_{P,L} \quad 0)}_{P}$$

3.2 Optimization algorithm

Our optimization algorithm is a simple iteration with *(1+1) selection strategy*. In every iteration, the algorithm selects the most optimal chromosome and executes one of five operators (here we use the *roulette selection*): *inversion*, *swap*, *crossover*, *splay left*, *splay right*. After that we apply validation heuristics to check if the new chromosome is a correctly encoded binary tree. If the test passes, the algorithm will evaluate a fitness function and check if we get a *new minimum*.

Let's define evolutionary operators used by our algorithm. The mutation operator was excluded because it changes a random gene. Because we assumed that genes with label 1 are *non-empty* genes (with information), thus a mutation would remove or insert a new node.

Algorithm 3 Optimization

```
$optimize(i : Individual, maxiter : Int) : Int$
    $n \leftarrow 0$
    $min \leftarrow eval(i.genome)$
    for $n < maxiter$ do
        $op \leftarrow i.selectOperator()$
        $g \leftarrow op(i.selectGenome())$
        if not $feasible(g, i.constraints)$ then
            continue
        end if
        $v \leftarrow eval(g)$
        if $v < min$ then
            $i.genome \leftarrow g$
            $min \leftarrow v$ {found a new minimum - reset number of iterations}
            $n \leftarrow 0$
        else
            $n \leftarrow n + 1$
        end if
    end for
    return $min$
```

3.2.1 Inversion operator

The inversion operator changes an ordering of genes in the chromosome, and it works as follows:

- We generate two *uniformly-distributed* numbers k_1, k_2 from a range $[0, n)$, where n is a length of the chromosome and $k_1 \leq k_2$.
- A new chromosome x', generated from x can be defined as:

$$x'_i = \begin{cases} x_{k_2 - i + k_1} & \text{for } i \in [k_1, k_2] \\ x_i & \text{otherwise} \end{cases}$$

Example: for $k_1 = 2, k_2 = 6$ and

$$x = (1_0, 0_1, \mathbf{1_2}, \mathbf{1_3}, \mathbf{0_4}, \mathbf{0_5}, \mathbf{1_6}, 1_7, 0_8, 0_9, 0_{10})$$

as a result of inversion, we will get

$$x' = (1_0, 0_1, \mathbf{1_{2(6)}}, \mathbf{0_{3(5)}}, \mathbf{0_{4(4)}}, \mathbf{1_{5(3)}}, \mathbf{1_{6(2)}}, 1_7, 0_8, 0_9, 0_{10})$$

Algorithm 4 Inversion operator

```
$inversion(genome : []Gene) : []Gene$
    $n \leftarrow len(genome)$
    $k_1 \leftarrow random(n)$
    $k_2 \leftarrow random(n)$
    **if** $k_1 > k_2$ **then**
        $k_1, k_2 \leftarrow k_2, k_1$
    **end if**
    **for** $k1 < k2$ **do**
        $genome[k_1], genome[k_2] \leftarrow genome[k_2], genome[k_1]$
        $k_1 \leftarrow k_1 + 1$
        $k_2 \leftarrow k_2 + 1$
    **end for**
    **return** $genome$
```

3.2.2 Swap operator

The swap operator looks like a simplified version of inversion because of it just swaps information between two random genes. The main difference is in the distribution algorithm which picks random indexes. For inversion we used *uniformly-distributed* numbers. Here indexes are pseudo-random numbers with *normal distribution*. A swap operation is *the lowest cost operation*, because it works in constant time $O(1)$, however, it can destabilize the whole structure due to modifications of single nodes (instead of entire blocks). We can interpret a swap operation as an exchange of two nodes in a binary tree.

Algorithm 5 Swap operator

```
$swap(genome : []Gene) : []Gene$
    $n \leftarrow len(genome)$
    $k_1 \leftarrow random(n)$
    $k_2 \leftarrow random(n)$
    $genome[k_1], genome[k_2] \leftarrow genome[k_2], genome[k_1]$
    **return** $genome$
```

3.2.3 Crossover operator

The crossover operator does a kind of reflection on a chromosome, and it works as follows:

- Generate a random *reference point* k, from a range $[0, n-3)$ with *uniform distribution*, where n is a length of the chromosome.
- Reflect the chromosome over the $k = 5$ point, without last two genes.

$$\mathbf{1_a} \quad \mathbf{0} \quad \mathbf{1_b} \quad \mathbf{1_c} \quad \mathbf{0} \quad \mathbf{0} \quad |_{k=5} \quad \mathbf{1_d} \quad \mathbf{1_e} \quad \mathbf{0} \quad 0 \quad 0$$

$$\mathbf{1_d} \quad \mathbf{1_e} \quad \mathbf{0} \quad |_{k} \quad \mathbf{1_a} \quad \mathbf{0} \quad \mathbf{1_b} \quad \mathbf{1_c} \quad \mathbf{0} \quad \mathbf{0} \quad 0 \quad 0$$

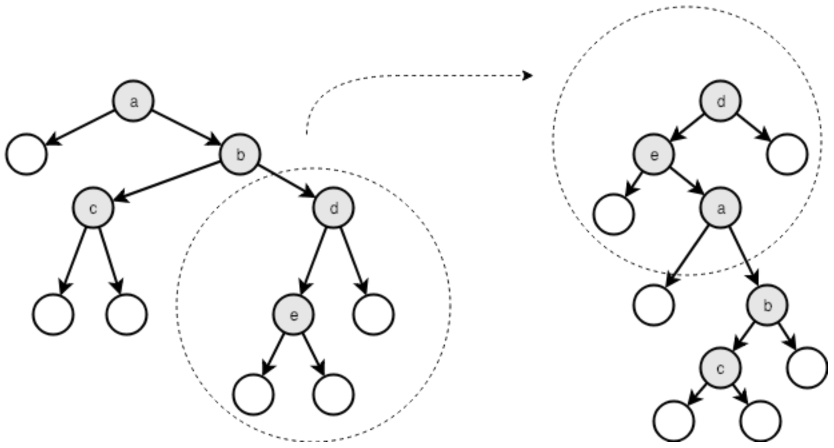


Figure 5: Crossover binary tree

Algorithm 6 Crossover operator

```
$crossover(genome : []Gene) : []Gene$
    $n \leftarrow len(genome)$
    $k \leftarrow random(n-3)$
    $head \leftarrow clone(genome[0 : k+1])$
    $tail \leftarrow clone(genome[k+1 : n-2])$
    $copy(genome[0 : len(tail)], tail)$
    $copy(genome[len(tail) :], head)$
    return $genome$
```

3.2.4 Splay Left operator

The splay left operator simulates a single *left rotation* of a binary tree. Splay operations are very common for self-adjusting binary search trees. Single left rotation transforms the tree T into binary tree T' as follows:

- The node x replaces the node y (where y is a parent for x)

- The node y becomes a left child of x and left sub-tree of node x becomes a right sub-tree of node y (by keeping all properties of binary search tree).

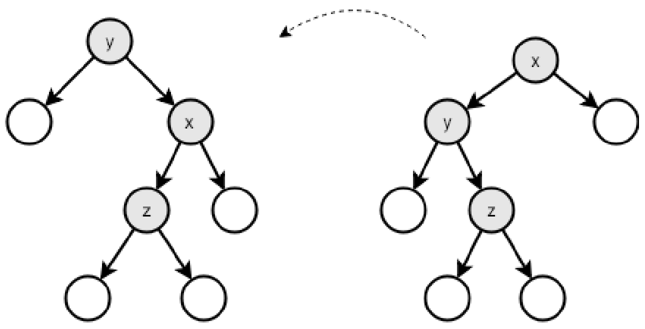


Figure 6: Single splay left operation

For chromosome representation we will get the following transformation:

$$T = (1_y \quad 0 \quad \mathbf{1_x} \quad 1_z \quad 0 \quad 0 \quad 0)$$

$$T' = (\mathbf{1_x} \quad 1_y \quad 0 \quad 1_z \quad 0 \quad 0 \quad 0)$$

It is easy to notice that the left rotation for $x - sequence$ is just a move of the root of right sub-tree T to the beginning. Because our chromosomes are $x - sequences$ then the rest (tail) of our sequence will keep the structure of a binary tree.

An algorithm to find a node which will be moved works as follows:

- Start from index 1 and find the index from which the number of empty nodes is greater than the number of non-empty nodes.
- If the index was found, then check if the next node is a non empty node. If yes then we found a root of sub-tree, and we move it to the beginning, otherwise the *splay left* is not doable.

In our example, the algorithm will stop after the first step, because of the node with index 1 is an empty node. And the next one is x which will be moved to the beginning. In the end, we get chromosome T'.

Algorithm 7 Splay Left operator

```
$splayLeft(genome : []Gene) : []Gene$
    $n \leftarrow len(genome)$
    $cnt0, cnt1 \leftarrow 0, 0$
    for $i \leftarrow 1$ to $i < n$ do
        if $empty(genome[i])$ then
            $cnt0 \leftarrow cnt0 + 1$
        else
            $cnt1 \leftarrow cnt1 + 1$
        end if
        if $cnt0 > cnt1$ then
            $k \leftarrow i + 1$
            if $k < n$ and not $empty(genome[k])$ then
                $g \leftarrow genome[k]$
                $copy(genome[1 : k + 1], genome[0 : k])$
                $genome[0] = g$
                return $genome$
            end if
        end if
    end for
    return $genome$
```

3.2.5 Splay Right operator

The operator *splay right* is analogous to *splay left* operator. The only difference is a single rotation which for *splay right* goes right. Single right rotation transforms the tree T into binary tree T' as follows:

- The node y replaces the node x (where y is a left child of x).
- The node x becomes a right child of y and right sub-tree of node y becomes a left sub-tree of node x (by keeping all properties of binary search tree).

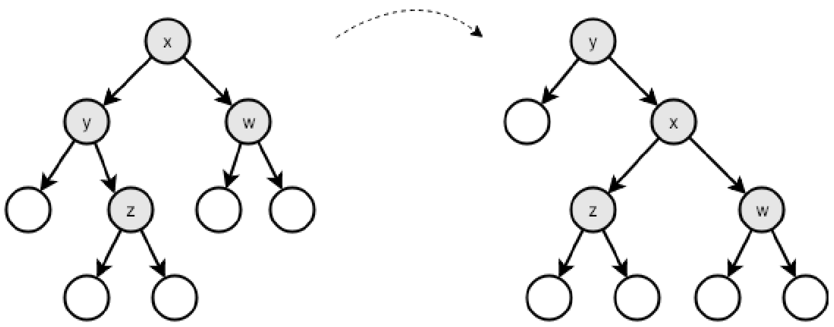


Figure 7: Single splay right operation

For chromosome representation we will get the following transformation:

$$T = (1_x \quad 1_y \quad 0 \quad 1_z \quad 0 \quad 0 \quad 1_w \quad 0 \quad 0)$$

$$T' = (1_y \quad 0 \quad 1_x \quad 1_z \quad 0 \quad 0 \quad 1_w \quad 0 \quad 0)$$

The algorithm for *splay right* slightly differs from *splay left* version. The main difference is instead of moving just one element we move entire sub-sequence. In other words, we find boundary nodes in sub-sequence and move the whole block to the beginning of the chromosome.

- Start from index 1 as a root of left sub-tree.
- If a gene with index 1 is an empty node, it means the left sub-tree is empty, so we cannot make a right rotation. Otherwise, we found the left boundary.
- Iterate through the rest of nodes (from index 2) and count number of empty and non-empty nodes. If the number of empty nodes is greater than number of nodes with information, then we found the right boundary.
- Move the found sub-sequence to the beginning of the chromosome.

In our example, we start from the chromosome:

$$T = (1_x \quad \mathbf{1_y} \quad \mathbf{0} \quad 1_z \quad 0 \quad 0 \quad 1_w \quad 0 \quad 0)$$

The node y is a root of left sub-tree and the left boundary of our sub-sequence. Next node is an empty node, so from the index 2, a number of empty nodes is greater than number of nodes with information. So index 2 is our right boundary. After that, we move our sub-sequence $chromosome[1, 2]$ to the beginning. As a result, we get the chromosome:

$$T' = (\mathbf{1_y} \quad \mathbf{0} \quad 1_x \quad 1_z \quad 0 \quad 0 \quad 1_w \quad 0 \quad 0)$$

Algorithm 8 Splay Right operator

```
$splayRight(genome : []Gene) : []Gene$
    $n \leftarrow len(genome)$
    **if** $n > 1$ **and** $genome[1] \neq NIL$ **then**
        $k_1 \leftarrow 1$
        $cnt0, cnt1 \leftarrow 0, 0$
        **for** $i \leftarrow 2$ **to** $i < n$ **do**
            **if** $empty(genome[i])$ **then**
                $cnt0 \leftarrow cnt0 + 1$
            **else**
                $cnt1 \leftarrow cnt1 + 1$
            **end if**
            **if** $cnt0 > cnt1$ **then**
                $k_2 \leftarrow i$
                $g \leftarrow genome[0]$
                $copy(genome[0:], genome[k_1 : k_2 + 1])$
                $genome[k_2] = g$
                **return** $genome$
            **end if**
        **end for**
    **end if**
    **return** $genome$
```

4 Monte Carlo tree search

Monte Carlo tree search (MCTS) is a heuristic search algorithm for some kinds of decision processes. It works in four phases:

- Tree traversal phase using following formula:

  $$UCB1(S_i) = \overline{V_i} + C\sqrt{\frac{\ln N}{n_i}},$$

  where:

  S_i is a $i - th$ state,

  C is a constant in order to balance between the exploitation term $\overline{V_i}$ and exploration term $\sqrt{\frac{\ln N}{n_i}}$ (the larger the constant, the more the algorithm will favor exploration over exploitation),

  $\overline{V_i}$ is an average value of $i - th$ state,

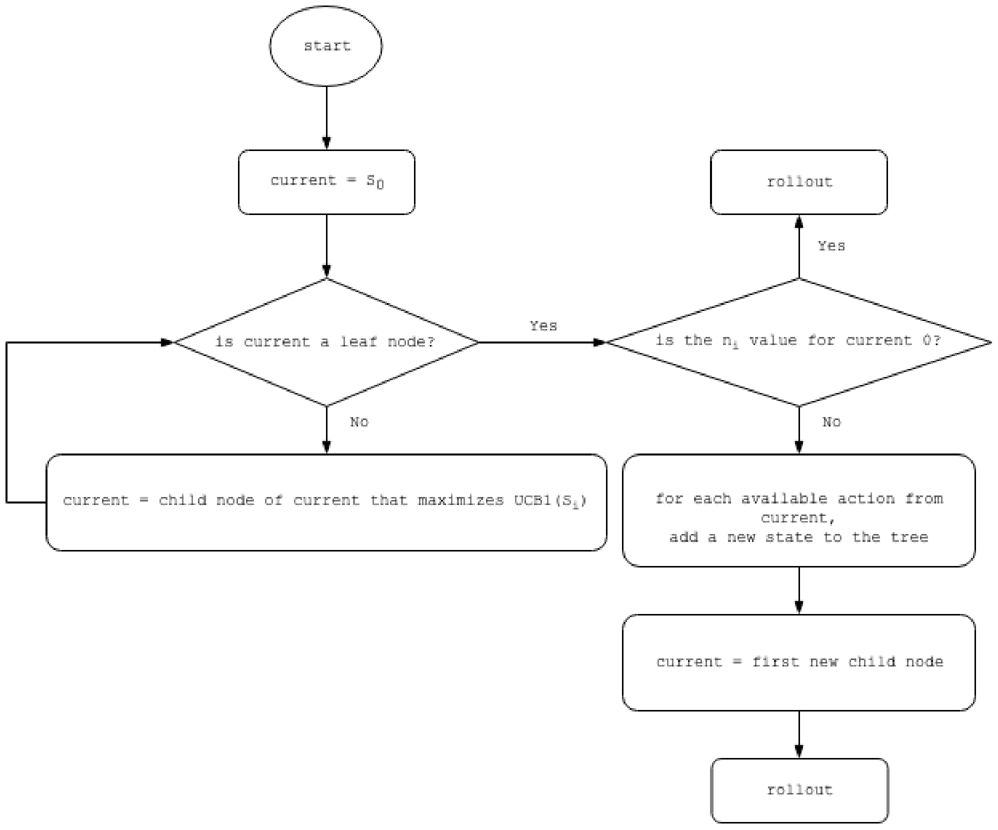
  N is a total number of all visits,

  n_i is a number of visits of $i - th$ node.

- Node expansion phase where you add extra nodes into the tree.
- Rollout phase which is a random simulation of the problem we are solving to find a value.
- Backpropagation phase which is where you take a value found from a *rollout* and put that in the appropriate places in the tree.

4.1 Tree traversal and node expansion

Let have a look at the algorithm for the first two phases (Figure 8). We start with S_0. If it is not a leaf node then you have to choose which child of the current states we are going to explore. We do this by calculating $UCB1$ value of each of the states and we choose the one which maximizes that value. We keep doing this until we get to a leaf node in the tree. Then we check how many times the node has been sampled. If it has never been sampled before, we do not expand it - we just do a rollout from there. However, if it has been sampled before, we add new nodes into the tree. For each available action from the current state, add a new state to the tree, and then let current be the first of those new child nodes. Then we do a rollout from this new child node.


start
current = S_0
is current a leaf node?
Yes
No
current = child node of current that maximizes UCB1(S_i)
is the n_i value for current 0?
Yes
rollout
No
for each available action from current, add a new state to the tree
current = first new child node
rollout
Figure 8: Monte Carlo tree search (tree traversal and node expansion)

4.2 Rollout

The rollout process is very simple:

Algorithm 9 Taking the state we got to, at the end of previous phase

```
$rollout(s_i : State)$
loop
    {loop forever until we get to terminal state}
    if $s_i$ is terminal state then
        return $value(s_i)$
    else
        {choose a random action}
        $a_i \leftarrow random(actions(s))$
        {simulate the action on the current state}
        $s_i \leftarrow simulate(a_i, s_i)$
    end if
end loop
```

4.3 Selection strategy

In order to decrease the probability factor in the optimization process, we decided to adopt **MCTS** as an experimental selection strategy (instead of *(1+1) strategy*). In other words, we still encode binary trees into chromosomes, and our evolutionary operators are available actions used in the *rollout* algorithm. But our selection process is mainly based on $UCB1$ formula and *negative reward* which is evaluation value of binary tree: $-\sum_{i=0}^{n-1} w_i(h_i + 1)$. Why do we use a *negative reward*? Because our task is to find a minimal tree evaluation, but at the same time we follow **MCTS** rules and expand nodes with the highest $UCB1$ value. If we look again into our custom $UCB1$ formula:

$$UCB1(S_i) = \overline{V_i} + C\sqrt{\frac{\ln N}{n_i}}, \quad C = 5$$

where C is set to 5 (by default), because we test five evolutionary operators and we assign each operator to a new node (after expansion). Of course, we can adjust this constant in case we want to favor more exploitation or exploration. $\overline{V_i}$ is equal to $\frac{-eval_T(S_i)}{n_i}$, where $eval_T(S_i)$ is T tree evaluation after applying operator assigned to state S_i. And n_i is a number of visits of state S_i. Generally, *selection strategy* prefers states with better reward, but at the same time non visited states have higher priority.

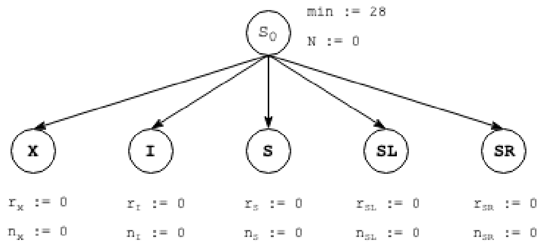


Figure 9: Monte Carlo tree search (initial state)

4.4 Simulation

The best way to explain our custom **MCTS** implementation is by simulating a few iterations of the selection strategy and optimization process. Let's assume our binary tree is represented by chromosome:

$$T = (1_1 \quad 1_1 \quad 1_2 \quad 1_3 \quad 0_0 \quad 0_0 \quad 0_1 \quad 0_1 \quad 0_0)$$

so, the start evaluation value is equal:

$$eval(T) = 1 \cdot 1 + 2 \cdot (1 + 0) + 3 \cdot (2 + 1) + 4 \cdot (3 + 1) = 28$$

We will try to optimize the tree (T) to get better evaluation. As you remember, we try to find a minimal evaluation which will satisfy binary tree constraint. We start with initial state S_0 (Figure 9) and the first thing we need to do is to add available actions (*crossover*, *inversion*, *swap*, *splay left* and *splay right*).

After a couple of iterations, we see that the best evaluation and the best reward got *splay right* action. Our current minimum is 20 and the best optimal tree, so far is:

$$T = (1_2 \quad 1_3 \quad 0_0 \quad 0_0 \quad 1_1 \quad 0_1 \quad 1_1 \quad 0_1 \quad 0_0)$$

In the next step, based on $UCB1$ formula, we choose and expand *splay right* state (Figure 10).

A *rollout* mechanism applies one of the evolutionary operators, we check if the new chromosome is *feasible* (satisfies all constraints), we evaluate it, and if we found a new minimum for that state we assign a new reward. We repeat this process until a chromosome is not feasible, or we have not found a better value for a certain number of iterations.

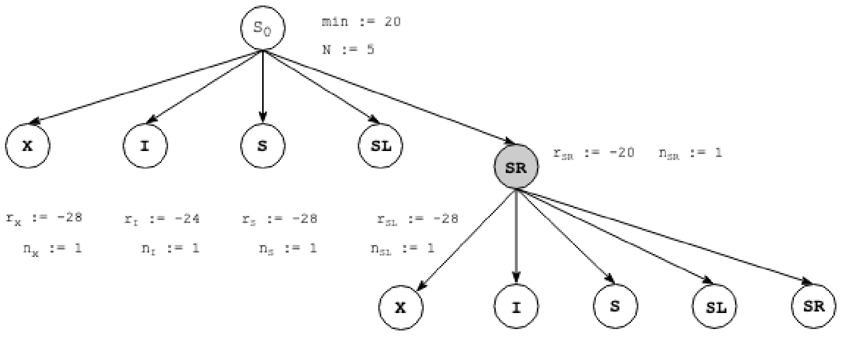


Figure 10: Monte Carlo tree search (expand *splay right* state)

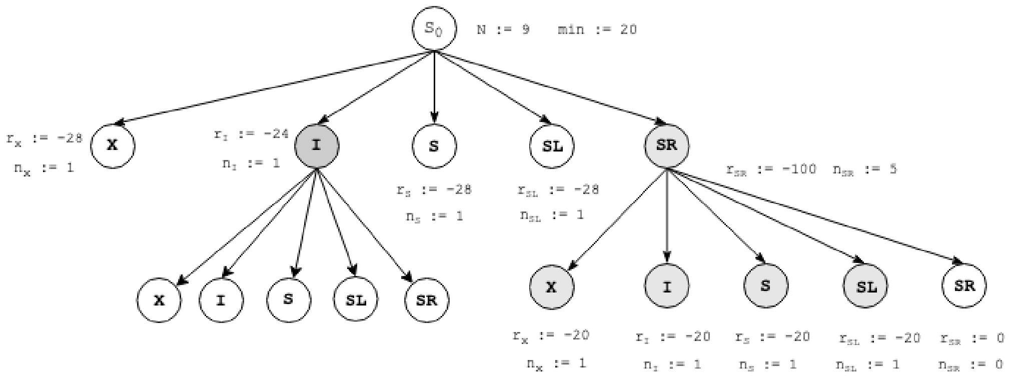


Figure 11: Monte Carlo tree search (expand *inversion* state)

After *splay right* rollout, the selection process chose the *splay right* state to expand. We expanded *SR* state, but most of the actions did not improve the reward, so at some point, the algorithm picked *inversion state* to expand (Figure 11).

We continued the process till we could not find better evaluation for N iterations (by default we ran our benchmarks with $N = 1000$). Our *Monte Carlo tree* was expanded up to five levels deep, and the best found chromosome was:

$$T = (1_3 \quad 1_2 \quad 1_1 \quad 0_0 \quad 0_0 \quad 0_1 \quad 1_1 \quad 0_1 \quad 0_0)$$

with evaluation:

$$eval(T) = 3 + 2 \cdot (2 + 1) + 3 \cdot (1 + 1 + 1) = 18$$

5 Benchmarking

We tested our optimization algorithms against three types of *binary search trees*. It is worth noting here that our *feasibility* function tested our chromosomes against two constraints. The first one checked if an encoded chromosome is a binary tree. The second constraint assumes that the chromosome is already a valid binary tree and checks if the chromosome is a correct binary search tree.

List BST is a binary search tree where all internal nodes are linked like in a list. Weights in all nodes were generated randomly with uniform distribution.

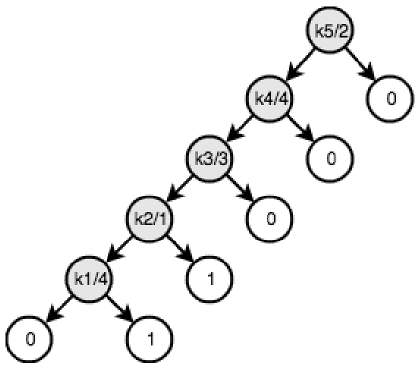


Figure 12: Randomly generated *List BST* with 5 internal nodes ($k1..k5$) in format (key/weight)

Non optimal BST is similar to **List BST**, but additionally the highest weights are at the bottom of a tree.

Balanced BST is a balanced binary search tree. Weights in all nodes were generated randomly with uniform distribution.

Every tested tree had 1500 *internal nodes* (nodes with information - a key). Rest of 1501 nodes (leaves) are either empty node (0) or *miss* node (no key, just weight).

For testing purposes, we tried adjust *selection probability* for each evolutionary operator. The best results we got after disabling all pure genetic operators (inversion, swap and crossover) and used only *splay* operators. Both *splay left* and *splay right* have the same, equal *selection probability* (0.5).

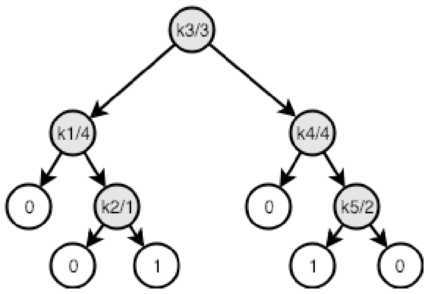


Figure 13: Randomly generated *Balanced BST* with 5 internal nodes ($k1..k5$) in format (key/weight)

We compared our results with a deterministic algorithm (based on *Knuth algorithm*) with time complexity $O(n^3)$ and memory $O(n^2)$. For $n = 1500$ our algorithm gave optimal results in over four seconds.

An evolutionary optimization time and memory complexity are linear, but it does not give us optimal results.

5.0.1 Results

	original value	evop value (time)	mcts value (time)	optimal value (time)
List BST	852.98	19.46 (2 s)	420.62 (2 s)	10.24 (4.90 s)
Non optimal BST	1130.03	18.18 (2 s)	333.30 (2 s)	10.55 (4.82 s)
Balanced BST	11.11	10.91 (2 s)	11.11 (0.7 s)	10.40 (4.75 s)

For an already balanced BST we did not notice much improvement, but for *pathological trees* (*List BST* and *Non optimal BST*) we can quickly get better evaluation than the original value.

Conclusion - if we want to get results quickly with very low memory footprint and we do not need super optimal values, then evolutionary operators can be a good choice. Especially for a large numbers of nodes ($n > 1000$) where deterministic algorithms may work for minutes and where our binary trees are not balanced. For instance for $n = 2000$ the deterministic algorithm ran for about *14 seconds*.

	original value	evop value (time)	mcts value (time)	optimal value (time)
List BST	2018.30	48.61 (3 s)	999.27 (3 s)	19.07 (13.98 s)
Non optimal BST	2675.86	58.16 (3 s)	787.75 (3 s)	19.54 (14.47 s)
Balanced BST	20.32	20.14 (3 s)	20.32 (0.9 s)	19.35 (14.75 s)

Moreover, if we do not evalutate *misses* (only *hits* - nodes with information) then evolutionary operators may give us results close to optimal ones.

	original value	evop value	optimal value
List BST	1982.23	22.85	18.65
Non optimal BST	2666.67	21.88	19.43
Balanced BST	20.51	20.34	19.48