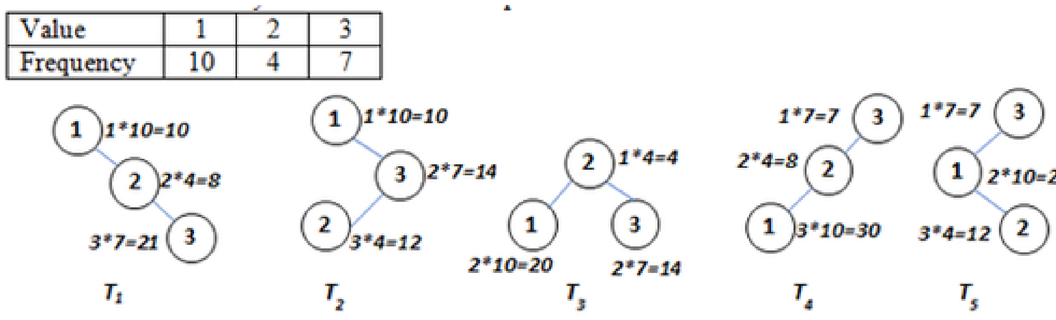
An Evolutionary Algorithm for Near-Optimal Binary Search Tree Construction

Evan Hataishi
ICS 674: Evolutionary Computation
Prof. L. Altenberg

May 15, 2020

1 Introduction

Finding the optimal Binary Search Tree (BST) is a somewhat classical programming problem. Given a set of keys and a probability that each key will be looked up i.e. frequency, the optimal BST is structured s.t. the expected search time over all keys is minimized. The expected cost of searching for any arbitrary key is $frequency \times (depth + 1)$. The expected search time of the entire BST is simply the sum of the search costs of all keys. Specifically, the goal is to construct a BST that minimizes the expected search time, while maintaining BST properties given the key values. Thus, it is preferable to have keys with higher frequencies at a shorter depth. Figure 1 shows 5 different BSTs that can be constructed given a set of keys (value row).



Value	1	2	3
Frequency	10	4	7

Figure 1: Various BST structures constructed from the same keys (reproduced from [1]).

A solution to the optimal BST problem is to simply try each key as root and recursively build each subtree. After exhaustively constructing every valid BST structure, the one with the minimum cost can be chosen. Given a tree with n nodes, this would take exponential time to solve. However, this naive solution can be further improved to $O(n^3)$ with dynamic programming. Although finding the optimal BST structure for any arbitrary set of keys and frequencies has been bounded, we would like to explore the application of an evolutionary algorithm to this problem.

1.1 Problem Statement

Given a set of keys $k_1, k_2, \ldots, k_n$ and a set of frequencies $f_1, f_2, \ldots, f_n$ s.t. f_i is the expected search frequency of k_i, we would like to use an evolutionary algorithm to construct a BST with minimal expected search time over all keys given the frequencies.

2 Related Work

The question of constructing optimal BSTs is a well-known problem with well-known classical and dynamic programming solutions as briefly discussed in Section 1. For this reason (we assume), we find only two significant works that propose evolutionary approaches to the question. The work in [4] proposed an evolutionary algorithm, but the main focus is on parallelization of the algorithm rather than the algorithm itself. The work in [3] also proposes an evolutionary algorithm to efficiently produce near-optimal BSTs. The maid idea of this work is the way in which crossover is implemented by directly merging the structure of two BSTs.

3 Evolutionary Algorithm

Search Space All valid BST structures given a set of keys.

Representation An N node BST constructed from a set of N keys and a corresponding set of N frequencies. All BSTs in the search space have the same keys with corresponding frequencies.

Objective Function Let k be the set of N keys and f be the set of N frequencies. The expected cost (and, consequently, fitness function) of the BST is calculated as follows:

$$f(k, f) = \sum_{i=1}^{N}(depth(k_i) + 1) \times f_i.$$

Therefore, we aim to minimize the expected cost, which will be some non-zero value depending on keys and frequencies.

Selection Operator We use elitism to pick the top 30% of a generation of trees to move on automatically to the next generation. The other 70% is formed from recombination.

Recombination Operator Only the top 50% fittest BSTs are used for recombination. Recombination is done by randomly picking two trees to combine into a single new tree. This process is described in the following paragraph.

Mutation Operator Mutation has a 10% chance of occurring on any tree being moved on to a subsequent generation after selection and recombination. The keys are simply shuffled and re-inserted to generate a new BST structure.

3.1 Recombination

The main novelty of this algorithm is the way in which trees do recombination. We first reduce each tree to its original list of keys. The list of keys is ordered by insertion (to the BST) order. Afterall, the tree structure is directly dependent on the order of insertion of the keys. Given the two lists of keys, we must merge the lists and remove all duplicates, so the resulting list maintains the same original set of keys. This is done based on the index of each key. Given two lists of keys x and y, we determine a *new* index for every key by randomly choosing between the key's respective index in x and respective index in y. Specifically, if key k_1 is at index 3 in x and at index 7 in y, then k_1 will be inserted into the "recombined" list at either index 3 or index 7. In cases where different keys are chosen to be inserted at the same index, they simply get inserted sequentially.

4 Code

To avoid having code within this report, all code used for this project can be found on my GitHub in a public repository [2]. `BST.py` contains a class `BST`. All methods related to my genome representation are implemented within that class. `main.py` is a driver and runs the genetic algorithm. Within `main.py`, we first define some hyper-parameters: `TREE_SIZE=100, POP_SIZE=45, GENERATIONS=300, SELECT_PERC=0.3`.

4.1 Search Space Representation

`BST` has an attribute called `nodes`, which represents the order in which each node was inserted to construct the BST (used for recombination). The constructor: `__init__(self, data)` takes a list of pairs of keys and frequencies, creates a node for each pair, and inserts each node into the BST. Both keys and frequencies should take arbitrary values, so (unique) keys are simply generated from a range given the tree size, and a frequency for each key is a randomly generated integer between 0 and 10 inclusive. The set of keys is then re-shuffled (to create a random insertion order) to generate each new BST in the population.

4.2 Variation Operator Encoding

Crossover is implemented as a static method: `crossover(p1, p2)`. Given two parents, it produces a single child by the crossover process described in Section 3.

4.3 Selection Operator Encoding

Selection is done in `main.py`. All agents are sorted by fitness, and the top 30% are selected to move on to the next generation.

4.4 Termination Criterion

We only terminate when either the optimal cost is reached. The actual optimal cost is calculated beforehand using a proven classical approach. However, the optimal is usually never found given the hyper-parameters, so we terminate at 300 generations.

5 Results

Figure 2 shows the best, average, and worst genome fitness in the population over 300 generations for a single run of the algorithm. Since the optimal fitness value is a minimum, and larger fitness values mean a "worse" genome, we negate every fitness value to obtain a more intuitive graph with the minimum at the top. Results seem fairly typical for a genetic algorithm. Worst fitness values are messy and have a high variance across generations. The best fitness values converge fairly quickly and do not vary much. We assume any dips in the best fitness values are due to mutation. At the end of 300 generations, the best genome obtained a fitness of -2734. Using the classical DP algorithm given this set of keys and frequencies, the actual optimal value is -2837, so this seems fairly close to optimal.

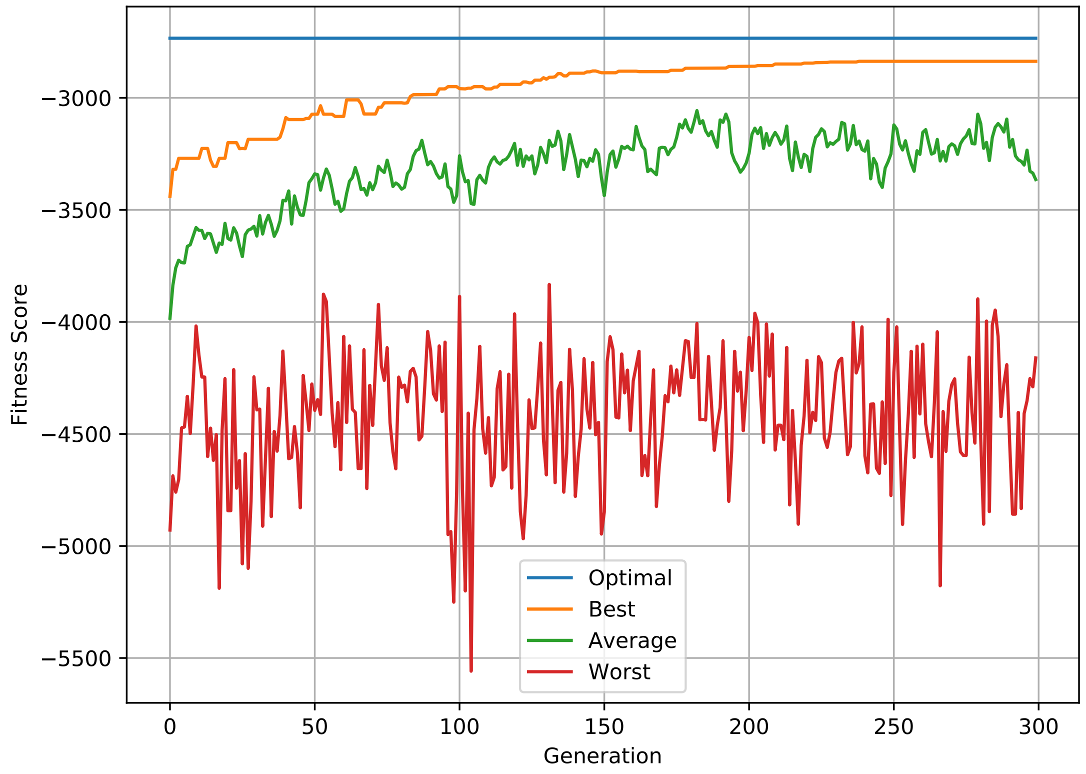


Figure 2: Fitness results over 300 generations. The actual optimal (calculated via DP algorithm) is plotted in blue.

6 Conclusion

We have proposed an evolutionary approach for constructing near-optimal binary search trees. Like some of the previous work discussed in Section 2, we use classical evolutionary techniques to setup the problem. The novelty of the algorithm proposed in this work is the way in which crossover is implemented. Rather than attempting to combine two BSTs based directly on their structures, we consider reducing each BST to a list of keys sorted by insertion order then doing crossover. We find that the algorithm converges fairly quickly (within a few hundred generations) and is able to construct a near-optimal BST.

6.1 Future Work

We can propose two main directions for future work. The first is to tune the algorithm parameters in the hopes of seeing slightly better convergence. A more major direction is to reconsider the way in which crossover is implemented.

References

[1] Optimal bst figure. `https://qph.fs.quoracdn.net/main-qimg-75a50568e29dd3ddbebf4fa80b4e85d8`.

[2] Project repository containing all relevant code. `https://github.com/evan-hataishi/ics-674/tree/master/final_project`.

[3] Afsaneh Fatemi, Kamran Zamanifar, and Naser NematBakhsh. A new genetic approach to construct near-optimal binary search trees. *Applied mathematics and computation*, 190(2):1514–1525, 2007.

[4] K Zamanifar and M Koorangi. Designing optimal binary search tree using parallel genetic algorithms. *International Journal of Computer Science and Network Security*, 7(1):138–146, 2007.