Optimizing Builds in Honkai: Star Rail using Branch and Bound

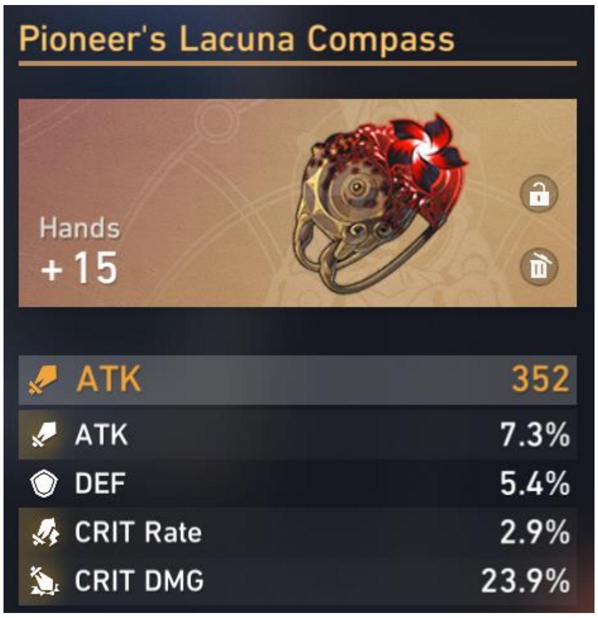
Muhammad Naufal Rayhannida - 10123006
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: naufalrayhannida@gmail.com , 10123006@mahasiswa.itb.ac.id

***Abstract*—Honkai: Star Rail is a popular turn-based video game developed by miHoYo. In this game, the players take turns fighting enemies using their abilities. Part of this game is gearing up your characters with "Relics" to increase their damage potential. This paper will discuss a branch-and-bound algorithm to help optimize the builds of characters.**

Keywords*—*Optimization; Branch and Bound; Honkai: Star Rail;

I. Introduction

Honkai: Star Rail is a turn-based video game developed miHoYo and published by Hoyoverse. In this game, players fight enemies by taking turns using abilities. Every time the player damages an enemy, the game will use the character and enemy's "stats" to calculate how much damage is dealt. However, before every fight starts, the player has the opportunity to gear up their characters with "relics". Each relic has a "main stat" and 4 possible "substats", each of them contributing to the character's stats.


Pioneer's Lacuna Compass
Hands
+15

ATK	352
ATK	7.3%
DEF	5.4%
CRIT Rate	2.9%
CRIT DMG	23.9%

Fig. 1. Example of a relic with ATK main stat and ATK, DEF, CRIT Rate, and CRIT DMG substats.

Each relic is also divided into 2 types, Cavern Relics and Planar Relics. Cavern Relics are divided into 4 types which are Head, Hands, Body, and Feet relics. Planar Relics are divided into 2 types, Planar Spheres and Link Rope. A character can wear 6 different relics, one for each type of Cavern and Planar Relics.

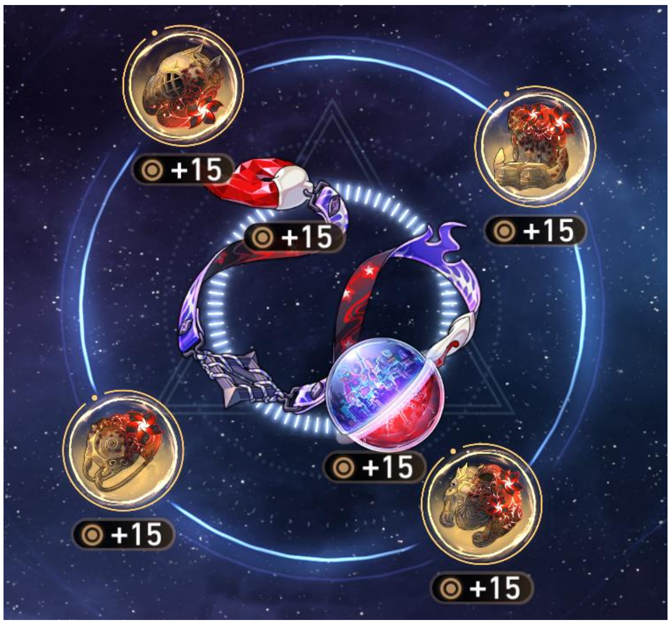
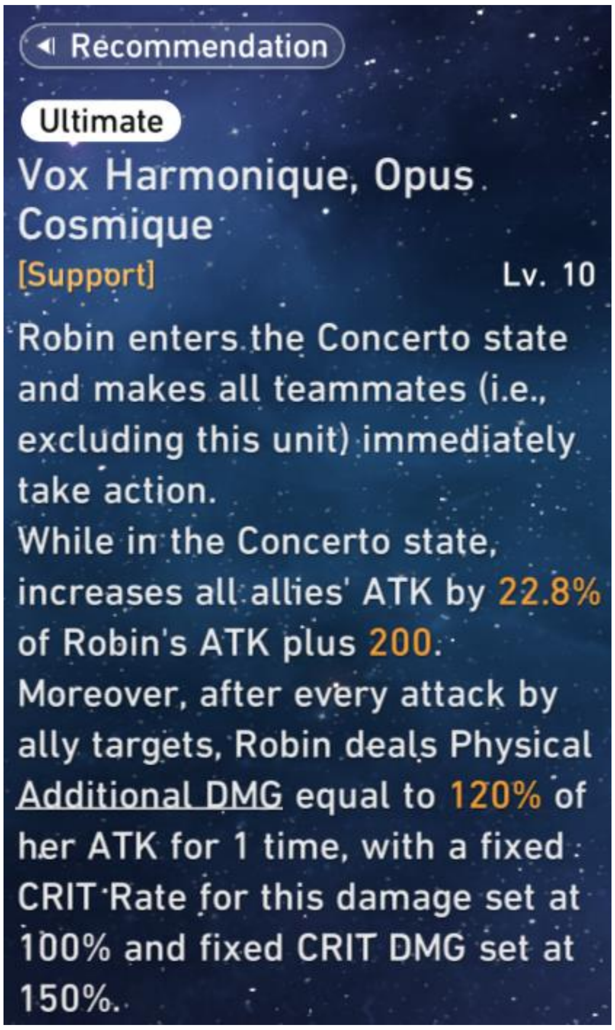


Fig. 2. Example of a character wearing all 6 different relics, with Planar Relics being the 2 in the middle.

A relic is also a part of a "set". When a character uses 2 or 4 relics of the same set, the character will get additional stats as a bonus. During the fight, characters can also get "buffed" either temporarily or permanently. These buffs can drastically change what builds the characters want as they can saturate a single stat. For the sake of simplicity, this paper will only include main stats, substats and assume any additional buffs from outside sources are always active.



◀ Recommendation

Ultimate

Vox Harmonique, Opus Cosmique

[Support] Lv. 10

Robin enters the Concerto state and makes all teammates (i.e., excluding this unit) immediately take action.

While in the Concerto state, increases all allies' ATK by 22.8% of Robin's ATK plus 200.

Moreover, after every attack by ally targets, Robin deals Physical Additional DMG equal to 120% of her ATK for 1 time, with a fixed CRIT Rate for this damage set at 100% and fixed CRIT DMG set at 150%.

Fig. 3. Example of a character giving buffs to other characters.

As there are 6 possible relic slots, by using a naïve brute force approach and iterating through the entire list of relics will have a time complexity of $O(n^6)$. While not exponential, it is still a high degree polynomial which does not scale well with more inputs. The Branch and Bound algorithm does not improve on the worst case time complexity, however it does improve on the average time immensely. Analyzing the exact time complexity for this specific problem is a daunting task [2] that this paper will not get into.

II. Theoretical Analysis

A. *Branch and Bound*

Branch and Bound (BnB) is an optimization algorithm where the problem is broken down into subproblems. Each subproblem would be evaluated by a bounding function and if the subproblem cannot result in an optimal solution, the subproblem will not be evaluated any longer.

The algorithm traverses the solution space as a tree and evaluates every child along with its estimated bound. If one of the nodes have their estimated bound lower than the current maximum, then that node would be pruned and would no longer be expanded. The algorithm would then take the node with the highest potential according to its estimated bound and evaluate its children, and so on until every possible node has been evaluated and the maximum has been found.

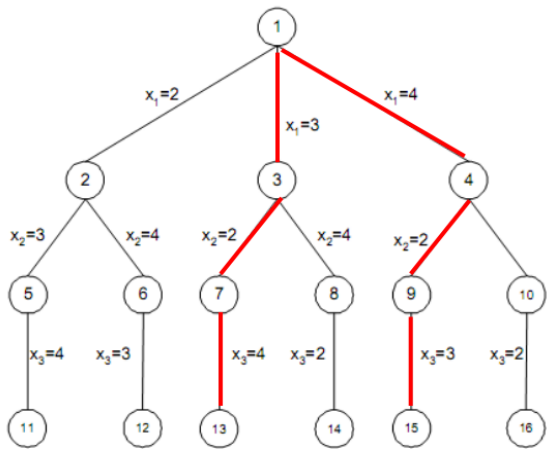


Fig. 4. Example of solution space and traversal. Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/18-Algoritma-Branch-and-Bound-(2025)-Bagian2.pdf, accessed on 24 June 2025

In practice, this algorithm uses a priority queue to list out all the live nodes. When a node is being traversed, it is evaluated and compared against the current best maximal node. If it's higher than the current maximum, then it updates the current maximum and prunes all nodes in the priority queue that's lower than the node's value. After that, the node's children would be added to the priority queue with the priority being the result of the bounding function, which would estimate the value of the children. After the priority queue has been fully exhausted, the current maximum is the optimal solution and the algorithm can be terminated.

B. *Mathematical Analysis*

This section will derive the approximation function used to bound a node in the BnB algorithm. The factors used in the damage formula for Honkai: Star Rail goes as follows:

$$\mathit{DMG} = \mathit{Base\ DMG} \cdot \mathit{Crit} \cdot \mathit{DMG\ Boost} \cdot \mathit{Weaken} \cdot \mathit{DEF} \cdot \mathit{RES} \cdot \mathit{Vulnerability} \cdot \mathit{DMG\ Mitigation} \cdot \mathit{Broken\ Multiplier}$$

However, many of the factors are enemy specific and aren't affected by the builds of characters. Therefore, we can simplify the damage formula. The simplified formula goes as follows:

$$\mathit{DMG}_{simple} = \mathit{Base\ DMG} \cdot \mathit{Crit} \cdot \mathit{DMG\ Boost}$$

where,

$$\mathit{Base\ DMG} = \mathit{Ability\ Multiplier} \cdot \mathit{Scaling\ Stat}$$
$$\mathit{Crit} = 1 + \mathit{Crit\ Rate} \cdot \mathit{Crit\ Damage}$$
$$\mathit{DMG\ Boost} = 1 + \Sigma\ \mathit{DMG\ Boost}_{stat}$$

Because *Ability Multiplier* is a constant that's tied to the character and not the build, we can ignore it from now on.

From here, we can expand the of the crit multiplier to get the following

$$DMG_{simple} = Scaling\ Stat \cdot DMG\ Boost + Scaling\ Stat \cdot Crit\ Rate \cdot Crit\ Damage \cdot DMG\ Boost$$

We can see that DMG_{simple} is a sum of 2 monomial terms. Unfortunately, getting the maximum of the monomial terms with 4 different relic slots is hard. We would need to iterate over every combination of relics to get the bound of that node. Therefore, we need to construct an approximation that is an upper bound to the DMG_{simple} formula. By getting the possible domain of each stat, we can construct a linear function

$$f(x, y, z, w) = ax + by + cz + dw + e$$

where *x, y, z, and w* are the character's stats, and $f(x, y, z, w)$ is greater than DMG_{simple} for every *x, y, z, w* in the domain. To get the constants *a, b, c, d,* and *e,* we only need to check the corners of a hypercube that are defined by the domain of each stat as the function we're working with is linear. We can solve this by using linear programming by minimizing the errors on each corner and constraining them to be positive. Linear programming can be solved by using algorithms such as the simplex algorithm or the interior point method. The approximation for the damage formula now looks like

$$DMG_{simple,\ approx.} = k_1\ Scaling\ Stat + k_2\ Crit\ Damage + k_3\ Crit\ Rate + k_4\ DMG\ Boost + k_5$$

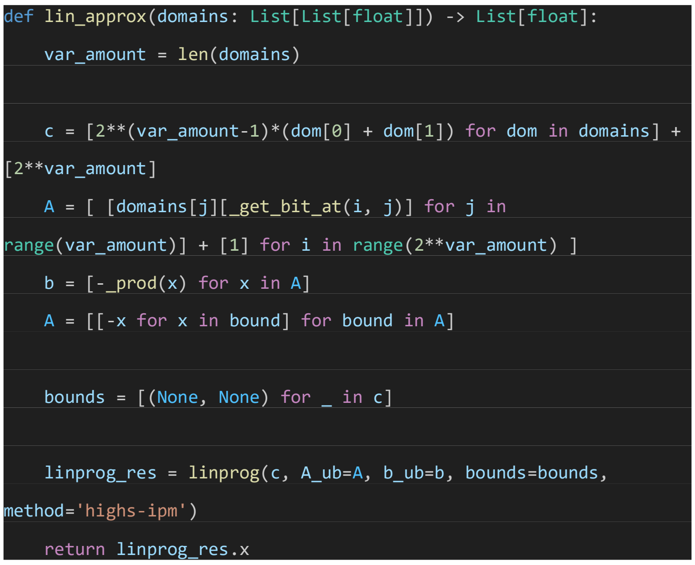
for some constants k_1, k_2, k_3, k_4 and k_5. With this linear approximation, we no longer need to check every combination of relics. We only need to take the maximum for each slot to get the upper bound, thereby getting a major speed up to computational complexity.

C. Algorithm Implementation Details

With the approximation function defined, we can now start with the details of the BnB algorithm implementation. We start by creating a node with no relic data and 0 cost and append it to a priority queue. From here, we will need to generate the possible bounds for each stat and create the approximation function for the first node. After constructing the approximation function, we can now expand the node and append it to the priority queue. Each of the node's children will have Head piece relic appended to its relic data and given a cost using the approximation function. The rest of the algorithm will be the same. We pop out the node with the highest cost, generate the bounds, get the approximation function, and expand the node by giving its children the next piece. The order of the pieces is Head, Hands, Body, Feet, Planar Sphere, and Link Rope. Once a node has all 6 pieces, the node is then calculated using the DMG_{simple} formula. Every other node with cost less than the true value of the node evaluated will be pruned. After the priority queue has been exhausted, the node with the highest value will be the optimal node.

III. Implementation

This paper implemented this algorithm using Python 3.13 and SciPy for solving the linear programming problem. The full source code is available in the GitHub repository linked at the end of the paper, however some of the snippets regarding the approximation function will be discussed in this section.



```python
def lin_approx(domains: List[List[float]]) -> List[float]:
    var_amount = len(domains)

    c = [2**(var_amount-1)*(dom[0] + dom[1]) for dom in domains] +
[2**var_amount]
    A = [ [domains[j][_get_bit_at(i, j)] for j in
range(var_amount)] + [1] for i in range(2**var_amount) ]
    b = [-_prod(x) for x in A]
    A = [[-x for x in bound] for bound in A]

    bounds = [(None, None) for _ in c]

    linprog_res = linprog(c, A_ub=A, b_ub=b, bounds=bounds,
method='highs-ipm')
    return linprog_res.x
```

This function interfaces with *linprog* from the SciPy library to get the constants for the approximation function. The coefficients for the objective function can be immediately found by multiplying the sum of the domain endpoints by 2^{n-1} where n is the amount of variables, except the coefficient for the constant where it's 2^n instead.

```
current_stats = calculate_stats(current_node.data, char,
buffs)
coeffs_1 = lin_approx([get_stat_bound(current_stats,
next_relic_groups, char, stat, buffs),
get_stat_bound(current_stats, next_relic_groups, char,
char.element, buffs)])
coeffs_2 = lin_approx([get_stat_bound(current_stats,
next_relic_groups, char, stat, buffs),
get_stat_bound(current_stats, next_relic_groups, char,
char.element, buffs), \
get_stat_bound(current_stats, next_relic_groups, char, 'cr_',
buffs), get_stat_bound(current_stats, next_relic_groups, char,
'cd_', buffs)])

        # Combine them both by adding the same stat to
eachother
coeffs = [coeffs_2[0] + coeffs_2[0], coeffs_2[1] +
coeffs_1[1], coeffs_2[2], coeffs_2[3], coeffs_2[4] +
coeffs_1[2]]
```

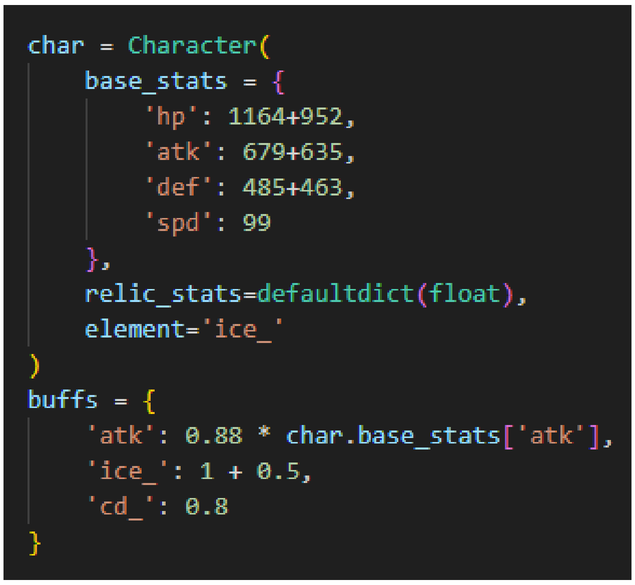
The coefficient from the previous function has to be added correctly according to each stat it represents.

```
def get_stat_bound(current_stats: Dict[FlatStats, float],
new_relics_per_type: List[List[Relic]], char: Character,
scaling_stat: FlatStats, buffs: Dict[FlatStats, float]) -> float:
    best_inc, worst_inc =
get_best_worst_bounding_increases(new_relics_per_type, char,
scaling_stat)

    if scaling_stat in ['ice_', 'fire_', 'lightning_', 'wind_',
'physical_', 'quantum_', 'imaginary_']:
        best_inc += 1
        worst_inc += 1

    best = current_stats.get(scaling_stat, 0) + best_inc +
buffs.get(scaling_stat, 0)
    worst = current_stats.get(scaling_stat, 0) + worst_inc +
buffs.get(scaling_stat, 0)

    if scaling_stat == 'cr_':
        best = min(1, best)

    return [worst, best]
```

The bounds are treated as flat increases that change the current stat of a character. The *DMG Bonus* multiplier is treated as a single variable, as it's just the *DMG Bonus* stat added by one.

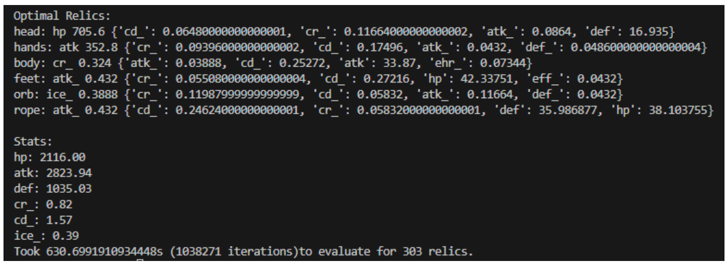
IV. Results and Analysis

The following results were taken using this base case



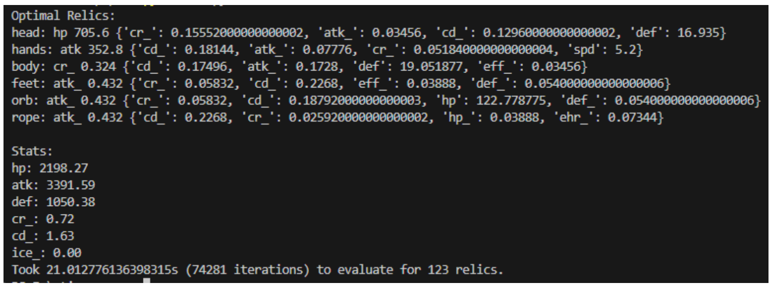
```
char = Character(
    base_stats = {
        'hp': 1164+952,
        'atk': 679+635,
        'def': 485+463,
        'spd': 99
    },
    relic_stats=defaultdict(float),
    element='ice_'
)
buffs = {
    'atk': 0.88 * char.base_stats['atk'],
    'ice_': 1 + 0.5,
    'cd_': 0.8
}
```

This base case is modelled after the character "The Herta" with some indeterminate external buffs. The relic pool is randomized with for each run, with differing qualities and counts.



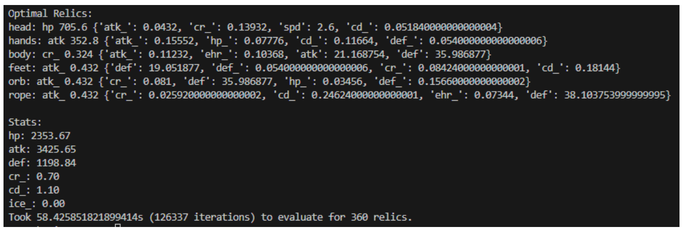
```
Optimal Relics:
head: hp 705.6 {'cd_': 0.06480000000000001, 'cr_': 0.11664000000000002, 'atk_': 0.0864, 'def': 16.935}
hands: atk 352.8 {'cr_': 0.09396000000000002, 'cd_': 0.17496, 'atk_': 0.0432, 'def_': 0.04860000000000004}
body: cr_ 0.324 {'atk_': 0.03888, 'cd_': 0.25272, 'atk': 33.87, 'ehr_': 0.07344}
feet: atk_ 0.432 {'cr_': 0.05508000000000004, 'cd_': 0.27216, 'hp': 42.33751, 'eff_': 0.0432}
orb: ice_ 0.3888 {'cr_': 0.11987999999999999, 'cd_': 0.05832, 'atk_': 0.11664, 'def_': 0.0432}
rope: atk_ 0.432 {'cd_': 0.24624000000000001, 'cr_': 0.05832000000000001, 'def': 35.986877, 'hp': 38.103755}

Stats:
hp: 2116.00
atk: 2823.94
def: 1035.03
cr_: 0.82
cd_: 1.57
ice_: 0.39
Took 630.6991910934448s (1038271 iterations)to evaluate for 303 relics.
```

Fig. 5. Results with 303 high quality relics



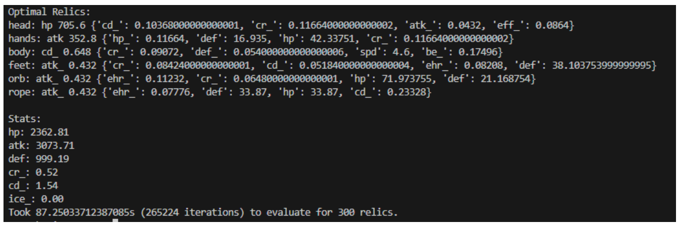
```
Optimal Relics:
head: hp 705.6 {'cr_': 0.15552000000000002, 'atk_': 0.03456, 'cd_': 0.12960000000000002, 'def': 16.935}
hands: atk 352.8 {'cd_': 0.18144, 'atk_': 0.07776, 'cr_': 0.05184000000000004, 'spd': 5.2}
body: cr_ 0.324 {'cd_': 0.17496, 'atk_': 0.1728, 'def': 19.051877, 'eff_': 0.03456}
feet: atk_ 0.432 {'cr_': 0.05832, 'cd_': 0.2268, 'eff_': 0.03888, 'def_': 0.05400000000000006}
orb: atk_ 0.432 {'cr_': 0.05832, 'cd_': 0.18792000000000003, 'hp': 122.778775, 'def_': 0.05400000000000006}
rope: atk_ 0.432 {'cd_': 0.2268, 'cr_': 0.02592000000000002, 'hp_': 0.03888, 'ehr_': 0.07344}

Stats:
hp: 2198.27
atk: 3391.59
def: 1050.38
cr_: 0.72
cd_: 1.63
ice_: 0.00
Took 21.012776136398315s (74281 iterations) to evaluate for 123 relics.
```

Fig. 6. Results with 123 high quality relics



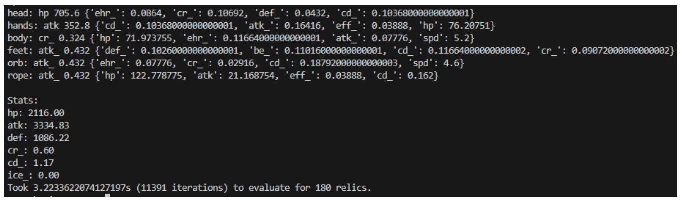
```
Optimal Relics:
head: hp 705.6 {'atk_': 0.0432, 'cr_': 0.13932, 'spd': 2.6, 'cd_': 0.05184000000000004}
hands: atk 352.8 {'atk_': 0.15552, 'hp_': 0.07776, 'cd_': 0.11664, 'def_': 0.05400000000000006}
body: cr_ 0.324 {'atk_': 0.11232, 'ehr_': 0.10368, 'atk': 21.168754, 'def': 35.986877}
feet: atk_ 0.432 {'def': 19.051877, 'def_': 0.05400000000000006, 'cr_': 0.08424000000000001, 'cd_': 0.18144}
orb: atk_ 0.432 {'cr_': 0.081, 'def': 35.986877, 'hp_': 0.03456, 'def_': 0.15660000000000002}
rope: atk_ 0.432 {'cr_': 0.025920000000000002, 'cd_': 0.24624000000000001, 'ehr_': 0.07344, 'def': 38.103753999999995}

Stats:
hp: 2353.67
atk: 3425.65
def: 1198.84
cr_: 0.70
cd_: 1.10
ice_: 0.00
Took 58.425851821899414s (126337 iterations) to evaluate for 360 relics.
```

Fig. 7. Results with 360 random relics



```
Optimal Relics:
head: hp 705.6 {'cd_': 0.10368000000000001, 'cr_': 0.11664000000000002, 'atk_': 0.0432, 'eff_': 0.0864}
hands: atk 352.8 {'hp_': 0.11664, 'def': 16.935, 'hp': 42.33751, 'cr_': 0.11664000000000002}
body: cd_ 0.648 {'cr_': 0.09072, 'def_': 0.05400000000000006, 'spd': 4.6, 'be_': 0.17496}
feet: atk_ 0.432 {'cr_': 0.08424000000000001, 'cd_': 0.05184000000000004, 'ehr_': 0.08208, 'def': 38.103753999999995}
orb: atk_ 0.432 {'ehr_': 0.11232, 'cr_': 0.06480000000000001, 'hp': 71.973755, 'def': 21.168754}
rope: atk_ 0.432 {'ehr_': 0.07776, 'def': 33.87, 'hp': 33.87, 'cd_': 0.23328}

Stats:
hp: 2362.81
atk: 3073.71
def: 999.19
cr_: 0.52
cd_: 1.54
ice_: 0.00
Took 87.25033712387085s (265224 iterations) to evaluate for 300 relics.
```

Fig. 8. Results with 300 random relics



```
head: hp 705.6 {'ehr_': 0.0864, 'cr_': 0.10692, 'def_': 0.0432, 'cd_': 0.10368000000000001}
hands: atk 352.8 {'cd_': 0.10368000000000001, 'atk_': 0.16416, 'eff_': 0.03888, 'hp': 76.20751}
body: cr_ 0.324 {'hp': 71.973755, 'ehr_': 0.11664000000000001, 'atk_': 0.07776, 'spd': 5.2}
feet: atk_ 0.432 {'def_': 0.10260000000000001, 'be_': 0.11016000000000001, 'cd_': 0.11664000000000002, 'cr_': 0.09072000000000002}
orb: atk_ 0.432 {'ehr_': 0.07776, 'cr_': 0.02916, 'cd_': 0.18792000000000003, 'spd': 4.6}
rope: atk_ 0.432 {'hp': 122.778775, 'atk': 21.168754, 'eff_': 0.03888, 'cd_': 0.162}

Stats:
hp: 2116.00
atk: 3334.83
def: 1086.22
cr_: 0.60
cd_: 1.17
ice_: 0.00
Took 3.2233622074127197s (11391 iterations) to evaluate for 180 relics.
```

Fig. 9. Results with 180 random relics

The algorithm has improved the time complexity immensely from the naïve brute force algorithm of $O(n^6)$ to something more manageable. However, the better relics you have, the longer it will take, as it will take longer for the tree to reach its leaf nodes and prune out branches near the top. It can also be seen in the 300 and 360 random relic test cases, where different relics can be clearly seen to drastically affect the runtime.

Some improvements that can be made are the assumptions that are ignored, such as permanent buffs and the lack of relic sets. By making a smarter approximation function, it would be possible to account for non-permanent buffs and relic sets. The implementation in this paper can also be improved by using a language with better memory management as Python does not have a robust way of managing memory and making the most out of the CPU's cache. It is also possible to preprocess the relics first and pre-prune the relics that do not have any useful stats. However, that will require a more robust UX design than a simple Python script.

There is also the possibility of parallelization, as described by Vu and Derbel (2016) [3]. By effectively splitting up the work into different workers, there is a potential to get larger gains. However, as mentioned in the paper by Vu and Derbel, memory management is crucial as accessing memory is a major bottleneck in improvement. If memory management is done correctly, the speed-up is near-linear.

Another improvement that can be made for the user is optimizing for multiple attacks at once. Because each attack can have different scaling, it is important to make sure your stats are distributed equally.

ACKNOWLEDGMENT

The author would like to thank the lecturers of IF2211 for teaching the class of 2024/2025 with knowledge about algorithms and how to measure their speed.. The author would also like to thank the fellow mathematics major who took up the IF2211 class despite it being a hard subject to do.

SOURCE CODE

The source code for the algorithm can be found in the following GitHub repository.

REFERENCES

[1] R. Munir, Branch and Bound Algorithm (Part 1), 2025, https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/17-Algoritma-Branch-and-Bound-(2025)-Bagian1.pdf, accessed on 24 June 2025.

[2] R. Lipton, 2012, Branch And Bound—Why Does It Work?, https://rjlipton.com/2012/12/19/branch-and-bound-why-does-it-work/, accessed on 24 June 2025.

[3] Trong-Tuan Vu, Bilel Derbel. Parallel Branch-and-Bound in Multi-core Multi-CPU MultiGPU Heterogeneous Environments. Future Generation Computer Systems, 2016, 56, pp.95-109. ff10.1016/j.future.2015.10.009ff. ffhal-01067662.

[4] Honkai: Star Rail Wiki, 2025, https://honkai-star-rail.fandom.com/wiki/Damage, accessed on 24 June 2025.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 1 Juni 2025

[illegible]

Muhammad Naufal Rayhannida (10123006)