The Microbiome Shifting Game

A Modified Chess Algorithm for Optimization of Dietary Nutrient Supplementation/ Depletion for Rational Modification of a Microbial Community Composition

1 Introduction

The human microbiome is comprised of bacteria, archaea, fungi, and viruses living on and inside the body. From the moment we are born, our microbiomes play a crucial role in healthy development. As the field progresses we are learning that the health outcomes of patients with dysbiosis-related syndromes such as IBD, Crohn's disease, obesity, and many others can be improved with dietary supplements. However, the link between nutritional intervention and a microbiome's taxonomic and functional profile are still poorly understood at the specific nutrient and species level[1]. To better understand this relationship, we aim to solve the Microbiome Shifting Game, which is loosely and biologically defined as determining the specific effects of dietary interventions on a microbiome's healthy vs. dysbiosis state.

For a given patient, an initial microbial makeup can be determined, along with the relative abundance of present microbes. To simplify these abundances, each microbe is considered to be present in a healthy or "normal" amount, overabundant or "high" amount, or underabundant or "low" amount.

The overarching goal is to maximize the number of microbes in a "normal" state and minimize the number of microbes in a state of dysbiosis. Certain bacteria are able to grow in the presence of given nutrients; for example, Phocaeicola

vulgatus grows in the presence of Fructose. Thus, if a patient with a low abundance of Phocaeicola vulgatus is administered Fructose, we can consider their resultant levels of Phocaeicola vulgatus to be normal. For the sake of simplicity, amounts of given nutrients are omitted in our model. With initial abundance information for a given patient and known nutrient sensitivities of microbes, an optimal "treatment", or group of nutrients to administer to a patient, can be compiled. How can such an optimal treatment be found?

2 Methods

Denote a **Binary Phenotype Matrix (BPM)** to be an $n \times m$ matrix where $n =$ number of microbes, and $m =$ number of nutrients, describing whether each microbe is reactive to a given nutrient.

Denote an **Abundance Qualifier A** as a vector of length n describing the initial state of each microbe in the BMP (overabundant or high: 1, normal or norm: 0, underrepresented or low: -1) where states 1 or -1 represent states of dysbiosis.

Denote a **treatment vector TREAT** as a $1 \times m$ vector representing the proposed nutrient intervention with values in 1: increasing the amount of a nutrient, 0: no change, -1: decreasing the amount of a nutrient.

Define an **outcome vector O**:

$$\mathbf{O} = Outcome(\mathbf{BPM}, \mathbf{A}, \mathbf{TREAT}) = sign(\mathbf{BPM} \cdot \mathbf{TREAT}^\top + \mathbf{A})$$

O is the outcome vector of length n representing the resultant states of each microbe after applying **TREAT** to **BPM** with initial abundances **A**, then taking the sign of each resultant entry. In taking the sign all values are mapped as follows:

$$> 0 \text{ as } +1$$

$$< 0 \text{ as } -1$$

$$= 0 \text{ as } 0$$

As the equation above implies, applying **TREAT** to **BPM** corresponds to multiplying the j^{th} row of **BPM** by the j^{th} entry in **TREAT** and taking the sum of each row, then adding this $n \times 1$ vector element-wise to the **A** $n \times 1$ vector yielding an outcome **O** for each of the n microbes.

Clarification: Let the confusion of number of $m$icrobes n, with number of $n$utrients m be explicitly clarified. Our **BPM** is represented as a matrix. Although variable names n and m are confusing, they are retained in order to stay consistent with the common mathematical variable names for number of rows n and columns m of a matrix.

Define a **score S** to be the proportion of elements in outcome vector **O** that are equal to 0.

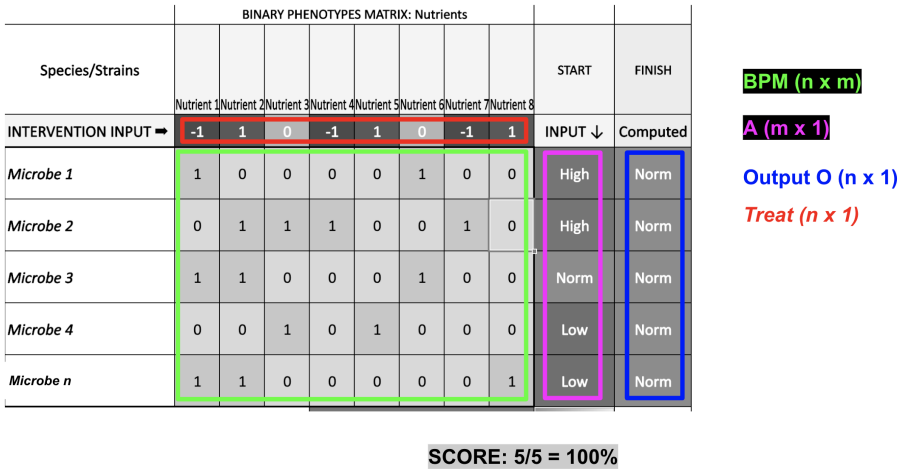

Figure 1: Visual representation of Binary Phenotype Matrix. Rows represent the reactivity (reactive: 1 or unreactive: 0) of a microbe to a given nutrient. Columns represent a number of different nutrients. The START input denotes initial abundance A. The FINISH Computed column denotes output O. The INTERVENTION INPUT row denotes the TREAT vector. The resulting score is 100% since all microbes are in a normal final state.

2.1 Challenge Statement

Challenge: Solve the Binary Microbiome Shifting (BMS) Problem

Given: A Binary Phenotype Matrix BPM and Abundance Qualifier vector A.
Return: A treatment vector **TREAT** that maximizes the score of $\mathbf{O} = Outcome$ among all possible vectors.

2.2 Simulating data

In order to test data quickly and for a variety of cases, values for filling **BPM** and **A** can be randomly drawn from distributions. Generating simulated data for **BPM** requires a simple draw from a uniform distribution on the set $\{0, 1\}$. Generating **A** requires drawing n values from a uniform distribution on the set $\{-1, 0, 1\}$. However, it may be desired to fix a specific score for the initial state vector **A**. This corresponds to fixing the number of microbes within the normal range (equal to zero), which is denoted k. In this case, **A** can be generated by randomly permuting an array which contains k zeroes and $n - k$ random values drawn from a uniform distribution on the set $\{-1, 1\}$.

3 Algorithmic approach

3.1 Brute force

The naive approach to solve this problem would be a brute force method which checks every possible **TREAT** vector and chooses the one which maximizes score, **S**. However, this quickly becomes implausible due to the fact that the ternary vector **TREAT** has 3^m possible states, making the brute force algorithm of runtime $O(3^m)$. Therefore, this approach is useful in cases with few number of nutrients (small m) and many microbes, but not useful in the more realistic case where many nutrients are considered(large m).

3.2 Chess Algorithm

In the case of a brute force algorithm, we can think of each intervention vector **TREAT** as a leaf of the tree generated by starting from a root node and branching into 3 nodes at each nutrient in **TREAT**. For the first nutrient, there are

three possible treatments, and three more possible treatments stemming from each of those, and so on. As **TREAT** has 3^m possible states, the tree we create in this case would have 3^m leaves.

The goal is then to eliminate many of these leaves, thus decreasing the search space. The termed **Chess Algorithm** is motivated by the basic idea behind finding a next best move in chess. In order to choose the next best possible move given any state in a game of chess, a player would *ideally* be able to consider all possible outcomes based on the next move and choose the best one. However, similar to the case of the Microbiome Shifting Game, it is computationally infeasible to play out every possible game for each next move. Instead, the chess player must consider only moves in the near future and prioritize the most promising moves.

In order to adapt this chess strategy and apply it to the Microbiome Shifting Game, first a random order of nutrients is generated. Note that this order could also possibly be determined not randomly, but by a biologically meaningful rank of nutrients. Then a next number of nutrients to fix is chosen, denoted **depth**. Each possible treatment for the next **depth** nutrients is then examined. For each treatment, the remaining unfixed nutrients are *randomly* permuted and each resultant treatment is scored. Whichever treatment results in the best maximum score is retained and the process is repeated until each shelf of length **depth** in the tree has been examined. In the tree, this is the translates to selecting the most promising node of the $3^{\mathbf{depth}}$ next nodes and preceding further down that branch.

3.2.1 Selecting the most promising node

Challenges arise in trying to select the most promising node at each step. A naive greedy algorithm could be implemented to proceed to the node with the best single score, under the hidden assumption that the remaining nodes are all 0. While this seems reasonable, it is unlikely to lead to vast improvement over the initial state. Changing one or even **depth** nutrients at a time without considering the impact of the remaining nutrients is shortsighted.

As an illustration of this challenge, consider a simple case in which the first

three nutrients each affect one distinct microbe. The first three values of these nutrients can be set to correct the imbalance and restore the three microbes individually to their normal states. However, if the next three nutrients affect many different microbes including the three that have already been remedied, it may be more reasonable to examine these nutrients further down the tree. Therefore, because the microbes' states will be constantly changing following our nutrient treatment selection, the most promising node should clearly take into account all nutrients and not just the ones being examined at any given step.

This presents an issue because considering all nutrients at the same time, as discussed previously, is not feasible due to the computational time scaling exponentially with the number of nutrients. A compromise can be made to consider a fixed number of states of **TREAT** for each node we are evaluating. We denote the number of fixed states considered $\mathbf{n}_{\text{samples}}$, which becomes and important input parameter to the algorithm. Therefore, in evaluating each node, we set the first nutrients in **TREAT** to those deterministically set according to the node and its preceding nodes. There will be different numbers of deterministically set nodes, depending on the **depth** and the position of the previous node. Then, for each sample, the remaining values of **TREAT** will be filled by randomly drawn variables from a uniform distribution over the values [-1, 0, 1]. We score each instance of **TREAT** and repeat the process $\mathbf{n}_{\text{samples}}$ times for each node.

From the $\mathbf{n}_{\text{samples}}$ scores for each node, now the best node must be chosen. The only metric that is important for the overall outcome of the algorithm is the max score. Therefore, it may make sense to take the maximum score from all of the samples. However, a node with many high scores may be seen as more promising than a node with one high score which is one point higher due to the fact that there may be more promising paths out of the node with many high scores. Therefore, we define another parameter of our model, $\mathbf{n}_{\text{keep}}$, which tells us how many of the top scores we will consider in choosing the most promising node. At present, the node with the highest sum of $\mathbf{n}_{\text{keep}}$ best nodes is considered most promising.

The algorithm depends on three parameters, **depth**, $\mathbf{n}_{\text{samples}}$, and $\mathbf{n}_{\text{keep}}$. We can run the algorithm many times within this parameter space to determine the best tradeoff between runtime and score.

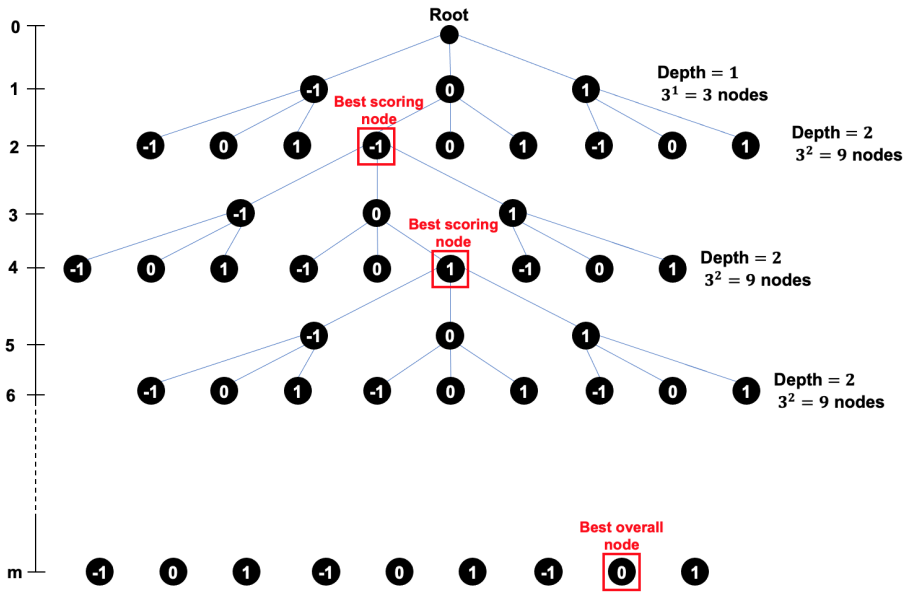

Figure 2: Chess algorithm schematic shown for a depth of 2 and m nutrients. The best node is calculated at the given depth and the algorithm proceeds generating the next 3^{depth} possible nodes starting from this previous best node.

3.3 Possible improvements

When progressing down the tree, the ideal outcome would be that the score increases as we explore more and more branches. Once a node is selected as the most promising, the algorithm explores more nodes following that most promising nodes. However, it is possible, and observed in simulation, that the score in selecting the next most promising node is not higher than the score found in selecting the previous most promising node. In this case, there are two options, retreat or continue. The current algorithm continues regardless of whether or not there is score improvement. One alteration to this current algorithm would be to retreat if the score doesn't improve. In this case, we might return to the level of the previous most promising node and select instead the second most promising node to then precede down that branch. This would inevitably slow down the computational time required, but may increase our score enough to justify the extra time.

It may make sense to set $\mathbf{n}_{\text{samples}}$ to be higher at the initial node selections,

when there are more possible states for the **TREAT** vector, and to then reduce the number of states considered further down the tree, when there are fewer remaining states to explore. This may allow for a more optimal tradeoff between computational time and maximum score.

Sorting the order at which we evaluate nutrients may change the effectiveness of our algorithm, though it is not immediately clear what impact this will have. If we sort the nutrients such that those affecting more microbes are fixed first, it may be easier to set the "most important" nutrients correctly, increasing our overall score. However, there is no guarantee that this would be true. Fortunately, it is something that we can evaluate and quantify.

Additionally, the algorithm lends itself well to parallelization. In this way, we could explore more than one node in parallel and compare the results before proceeding to the explore the next nodes. This should improve the maximum score without affecting the runtime significantly.

4 Results

After evaluating the Chess Algorithm's performance on simulated and real data, several conclusions can be made. When compared to a brute force algorithm, which is guaranteed to find the best solution at the cost of an infeasible runtime, the Chess Algorithm does not appear to perform significantly worse (when applied to simulated data with varying numbers of nutrients), while boasting a significant improvement in runtime (**Figure 3**).

A subset of datasets were examined due to time constraints. These datasets each contain varying numbers of nutrients and microbes each in the range of about 100 microbes and about 50 nutrients. As the algorithm traverses the tree, score improves, as illustrated by running the unparallelized implementation of the Chess Algorithm with **depth**=6, $\mathbf{n}_{\text{samples}} = 200$, $\mathbf{n}_{\text{keep}} = 1$ in (**Figure 4**).

A parallel implementation of the Chess Algorithm with (for example) 4 cores results in 4 chosen best nodes and a total of 16 candidate paths down which to traverse. The 4 best paths out of 16 are selected and iteratively processed again, on 4 different cores (**Figure 5**). As mentioned, such an approach aids in preventing the chess algorithm from traversing down a tree branch with a suboptimal score. This results in a significant improvement in score at a minimal cost in runtime in simulated data (**Figure 6**) and real data (**Figure 7**).

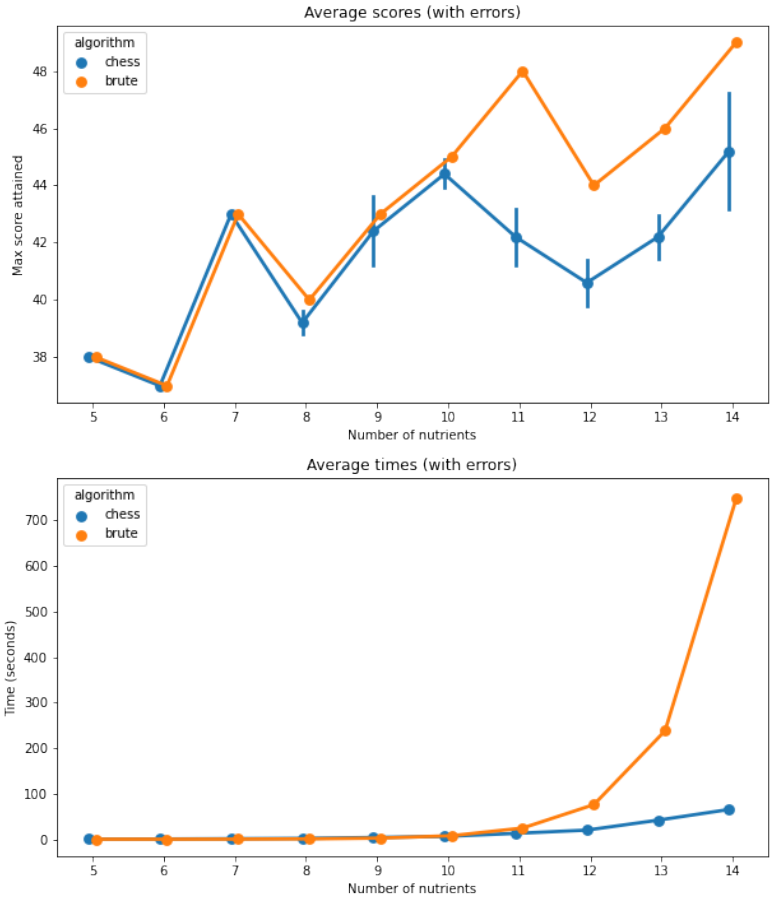

Figure 3: Average scores and average times from running the algorithm on simulated datasets with varying numbers of nutrients. All initial abundances were simulated to have initial scores of 20%. Each number of nutrient was sampled 5 times and variation bars are shown for these 5 runs. Number of nutrients terminated at 14 due to time constraints.

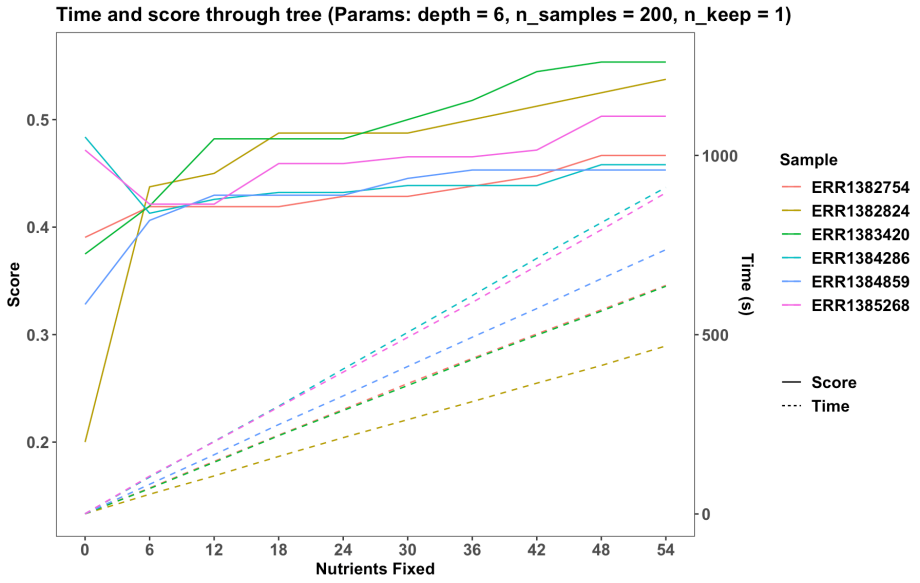

Figure 4: Score and time as the algorithm fills values in **TREAT**, for **depth** = 6 nutrients at a time. In five of six cases, score improvement over initial state is observed. Time is relatively short for this parameter set, and improvement in maximum score would be expected for higher **depth** and $\mathbf{n}_{\text{samples}}$, at the expense of longer runtime.

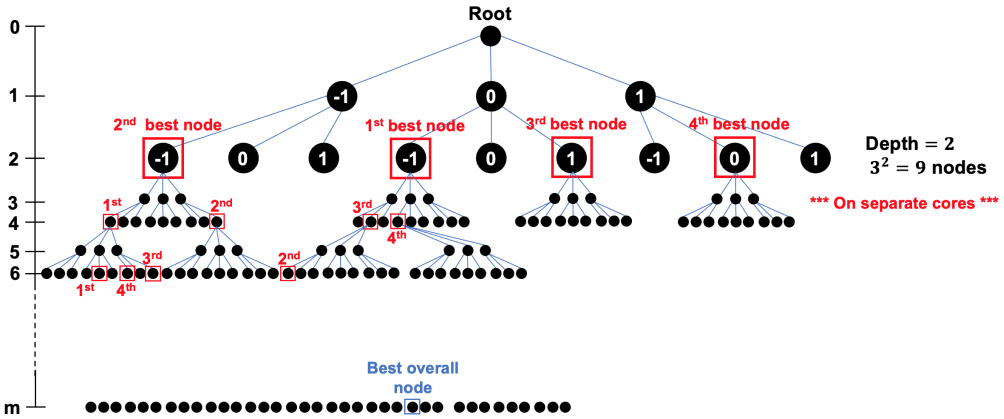

Figure 5: A schematic showing a possible implementation of a parallel chess pruning algorithm with **depth** = 2 and four cores. A parallel implementation should have a similar runtime to selecting the top single node with the expectation that the maximum score improves significantly.

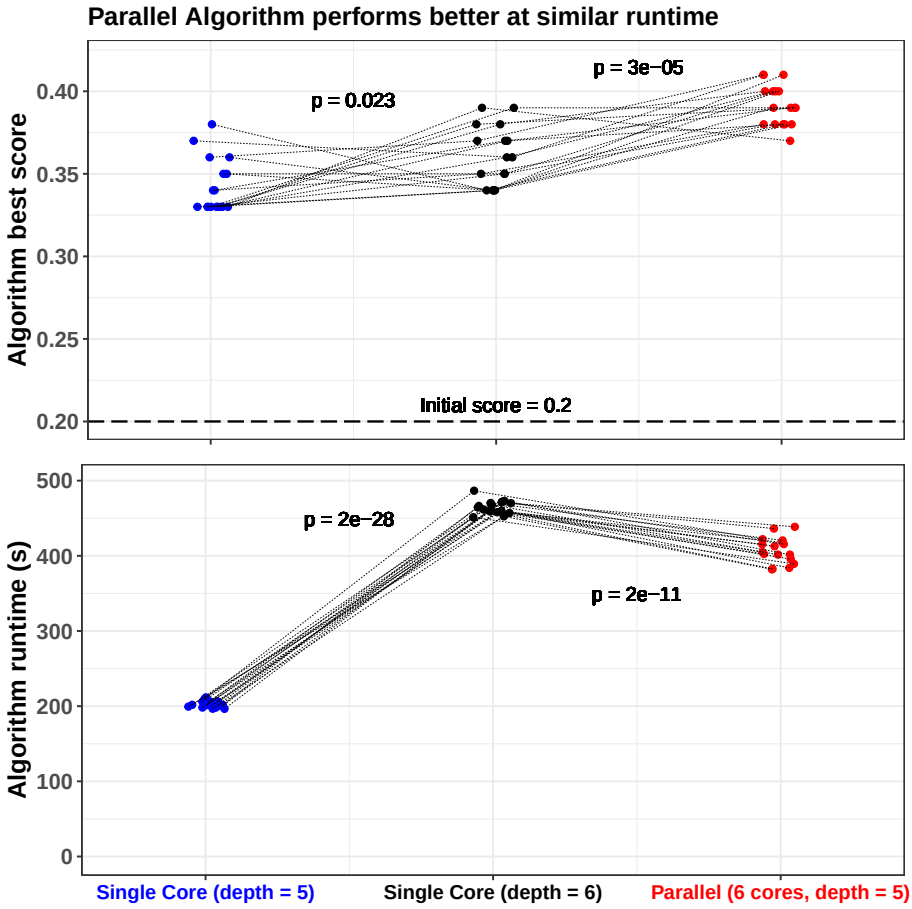

Figure 6: Parallel implementation significantly improves score at a roughly 2X cost to runtime on a local machine. In order to determine if this is a valuable tradeoff, the parallel implementation must be compared to a single core implementation at higher depth. It is observed to perform significantly better than a single core function at a higher depth, with the added benefit of slightly reduced runtime. Parameters for this run: $\mathbf{n} = 100$ microbes, $\mathbf{m} = 40$ nutrients, $\mathbf{n}_{\text{samples}} = 300$, $\mathbf{n}_{\text{keep}} = 3$. Each algorithm run for 17 randomly initialized **BPM** and **A**. Paired t-test used for statistical comparisons.

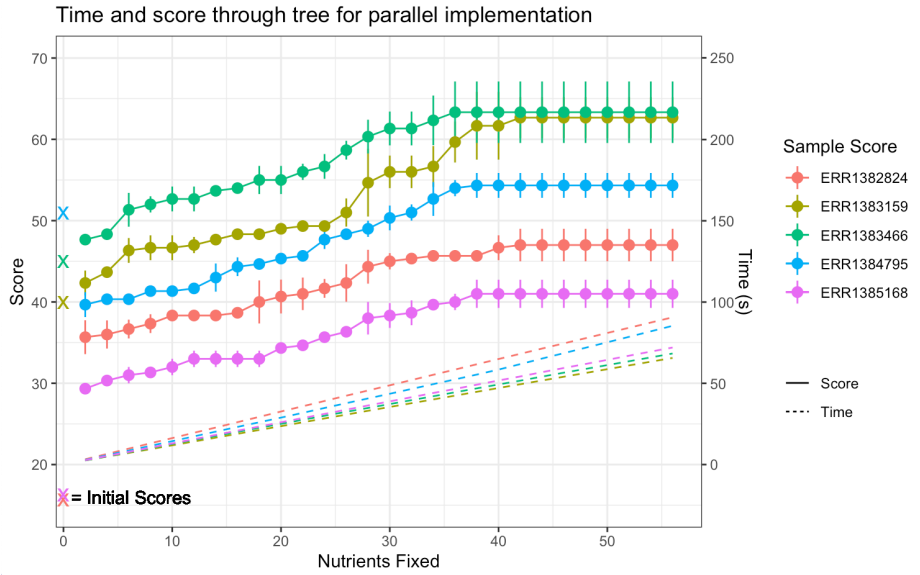

Figure 7: Parallel algorithm run for parameters: **depth** $= 2$, $\mathbf{n}_{\text{samples}} = 500$, $\mathbf{n}_{\text{keep}} = 3$, $\mathbf{n}_{\text{cores}} = 8$. The algorithm clearly outperforms the initial states.

5 Discussion

Developing an algorithm for solving the microbiome shifting game remains elusive. Unlike some problems, it is not easy to verify that any solution produces the optimal score, or even a score remotely near the optimal score. If a brute force algorithm is infeasible, we cannot determine the best possible score for a given **BPM** and **A**. On a standard CPU, this problem becomes relevant above 15 or so nutrients. Massive parallelization of the brute force algorithm may allow for improvements that make a calculation feasible for slightly higher numbers of nutrients, but there remains a limit due to the inevitable $O(3^m)$ scaling. Therefore, the current algorithm can be measured against brute force only for low values of m. The algorithm, even in a single core implementation, is observed to only slightly reduce the maximum score with a significantly shorter runtime, though it is not clear how to measure the algorithm compared to the optimal solution for larger values of m.

There are two remaining benchmarks that can be used to evaluate the currently implemented algorithm. One is relative performance compared to other algorithms attempting to solve the same computational problem. The other measures the maximum score found by the algorithm compared to the initial state. For a high-scoring initial state, it may be difficult to improve the maximum score even when randomly generating many possible **TREAT** vectors.

Fortunately, the current implementation of our algorithm can be used as a baseline for future improvements. In this way, while still lacking information on the optimal maximum score, it can be determined whether certain alterations to the existing algorithm progress in the direction of the optimal solution. We have shown that an algorithm exploring multiple nodes simultaneously is able to outperform the single node algorithm. For any further alterations, similar comparisons can be made to determine whether improvement has occurred.

Regardless of any improvements made to the algorithm, the parameter space must be explored to maximimize current performance. We know that runtime scales linearly with $\mathbf{n}_{\text{samples}}$ and exponentially ($3^{\mathbf{depth}}$) with **depth**. Therefore, we seek an optimal tradeoff between runtime and score that has yet to be fully explored. Additionally, the optimal value of $\mathbf{n}_{\text{keep}}$ is not known and may depend on other parameters. Further work should elucidate these relationships and demonstrate the effects introduced by varying the $\mathbf{n}_{\text{keep}}$ parameter.

6 Conclusion

In brief, an adaptation of a chess algorithm has been presented that is able to propose a nutrient intervention to improve the healthy state of an individual's microbiome. This algorithm and its derivatives aim to help patients with a plethora of gut dysibiosis related syndromes.

Supplementary: Sorting does not seem to affect score or runtime

As discussed in section 3.3, intuition may suggest that the order of the nutrients in **BPM** is important. Testing this has shown that, at least for the parameter set shown below, sorting **BPM** appears to have no effect on the score or runtime. It remains possible that sorting may improve the score, specifically at different values of **depth**. This should be tested more extensively in future work.

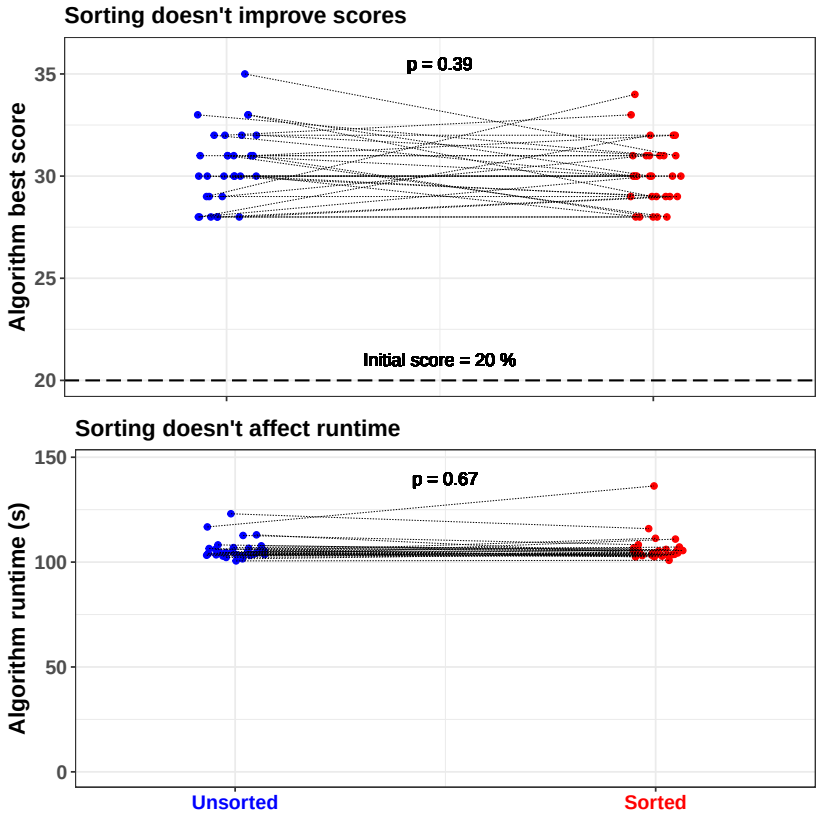

Figure 8: Sorting the **BPM** to select nutrients that affect more microbes first does not seem to affect the results. Parameters for this run: $\mathbf{depth} = 5$, $\mathbf{n} = 100$ microbes, $\mathbf{m} = 50$ nutrients, $\mathbf{n}_{\text{samples}} = 100$, $\mathbf{n}_{\text{keep}} = 1$. Each algorithm run for 30 randomly initialized **BPM** and **A**.

References

[1] Rodionov, D. A., Arzamasov, A. A., Khoroshkin, M. S., Iablokov, S. N., Leyn, S. A., Peterson, S. N., Novichkov, P. S., and Osterman, A. L., "Micronutrient requirements and sharing capabilities of the human gut microbiome," *Frontiers in Microbiology* **10** (2019).