Project Weak-Schur: Algorithm Proposal

Prasanna M.S.S

March 2022

The proposal is to develop a generic selection and fitness evaluation loop to create weakly sum-free partitions. The naive, single-threaded algorithm is proposed first, then simply extended to a multi-threaded algorithm. The Fitness function is then defined and simple bounds are given for it. These motivate an out-of-order selection, which is then detailed.

1 Process

The core Process of the algorithm can be summarised as a simple selection and fitness evaluation loop.

1.1 Single-Thread

```
Data: n ≥ 0
Result: BestSolution
Solutions ← Dict(keys : {1 . . . n});
num ← 1;
while BreakCondition(n,num, Choice) do
    num ← num + 1;
    for sol_i ∈ Solutions do
        sol_i.append(num);
        Fitness[i] ← EvaluateFitness(sol_i);
    end
    bestSolution ←
      Choice(Solutions, Fitness);
end
```

Assume that the current search is for a Weak-Schur partition of n colours.

In the single-threaded implementation, Process will start by initialising the Dictionary of *Solutions*, and start a counter of the number to add in each iteration. Then, while *BreakCondition* is true, sol_i will add its number to the i^{th} colour, and evaluate the fitness. At the end of this evaluation, the best solution is chosen using a suitable *Choice* function.

Two remarks can be made here;

- The algorithm is general enough to permit us to change the *Choice* function easily to observe how the behaviour of the generated solutions change. This allows us to experiment with a varied set of potentially multi-threaded or multi-process genetic algorithms. The *Choice* function also changes the *BreakCondition* in obvious ways.
- The inner for-loop of the algorithm can be embarrassingly parallel, and therefore can be split over multiple threads. This allows the algorithm to scale directly with the size of the partition being sought.

1.2 Multi-Thread

Following from the second remark, Process will start n threads to evaluate the for loop, in effect, one thread per sol_i or per color.

2 Fitness Function

Any fitness function that we choose to work with, in the opinion of the author, should possess two properties:

- Be an monotonic function of the size of the partition i.e. follows Proposition 1 (increasing), or a similar decreasing property.
- Be easy to compute, either in $O(n)$ or $O(nlogn)$, but not worse.

Definition 1. *Let $S = \bigsqcup_i S_i$ be a weakly sum-free partition. The fitness of the partition is defined as*

$$fitness(S) := \sum_{i\in[n]} \left| \left\{ (a,b,c) \in S_i^3 : \; a+b=c, a \neq b \neq c \right\} \right| \tag{1}$$

This section is devoted to showing that Defn. 1 possesses both of these properties, making it ideally suited for out-of-order, iterative algorithms. These terms will be made clear in further sections.

2.1 Bounds

Proposition 1. *Let $S = \bigsqcup_i S_i$ be a weakly sum-free partition such that its size $|S| = \sum_i |S_i| = n$, and $fitness(S) = k$. Then, assume we add $N \in \mathbb{N} - S$ to S i.e. $\exists i \in [n] : S_i \leftarrow S_i \cup \{N\}$. Then, $fitness(S \cup \{N\}) \geq k$.*

Proof. **(Main)** Assume we add $N \in \mathbb{N} - S$ to S i.e. $\exists i \in [n] : S_i \leftarrow S_i \cup \{N\}$.

Case 1 $\forall (a, b, c) \in S_i^3 : a + b = c, a \neq b \neq c \neq N$. This is the trivial case where no new pairs violating the sum-free property have been added, and so, $fitness(S \cup \{N\}) = fitness(S) = k$.

Case 2 $\exists (a, b, c) \in S_i^3, a + b = c, a = N \neq b \neq c$. Case 2 implies that at least one new pair has been added, and we have $fitness(S \cup \{N\}) > k$. Note that we can assume $a = N$ without loss of generality.

Case 3 $\exists (a, b, c) \in S_i^3, a + b = c, a \neq b \neq c = N$. this also implies that at least one new pair has been added, and we have $fitness(S \cup \{N\}) > k$.

□

Proof. **(Alternate)** Assume we add $N \in \mathbb{N} - S$ to S i.e. $\exists i \in [n] : S_i \leftarrow S_i \cup \{N\}$, and that $fitness(S \cup \{N\}) < k$. Then, there must exist at least one triplet $(a, b, c) \in S_i^3$ such that $a + b = c$, but $a + b \neq c$ when $(a, b, c) \in S_i^3 \cup \{N\}$. This is not possible when $a \neq b \neq c \neq N$, since no number was removed. However, supposing that $a = N$ or $c = N$ cannot be possible as $N \notin S_i$, but only in $S_i \cup \{N\}$. □

Proposition 2. *Let $S = \bigsqcup_i S_i$ be a weakly sum-free partition such that its size $|S| = \sum_i |S_i| = n$, and $fitness(S) = k$. Then, suppose we add $N \in \mathbb{N} - S$ to S i.e. $\exists i \in [n] : S_i \leftarrow S_i \cup \{N\}$. Then, $fitness(S \cup \{N\}) \leq k + 2^{|S_i+1|}P_2$.*

Proof. Adding N creates triplets of two types: 1) $(a, b, N) \in S_i^3$ such that $a \neq b \neq N$ and (2) $(N, b, c) \in S_i^3$ such that $N \neq b \neq c$ now violates the weakly sum-free property. There are at most $2 \cdot {}^{|S_i|+1}P_2$ pairs of this form and we bound the number of pairs after the addition of $\{N\}$ as :

$$\begin{aligned} fitness(S \cup \{N\}) - fitness(S) \\ \leq 2 \cdot {}^{|S_i|+1}P_2 \\ \implies fitness(S \cup \{N\}) \leq fitness(S) + \\ 2 \cdot (|S_i| + 1)(|S_i|) \end{aligned} \quad (2)$$

as desired. □

Since $\forall m, {}^mP_2 = O(m^2)$, this gives a good bound on the growth of $fitness$; however, this may still be improved.

In summary, we have

$$\begin{aligned} 0 \leq fitness(S \cup \{N\}) - fitness(S) \\ \leq g(k_i) = O(k_i^2) \end{aligned} \quad (3)$$

where $k_i = |S_i|$ so that g is a polynomial of order 2.

2.2 Linear $O(k_i)$ Implementation

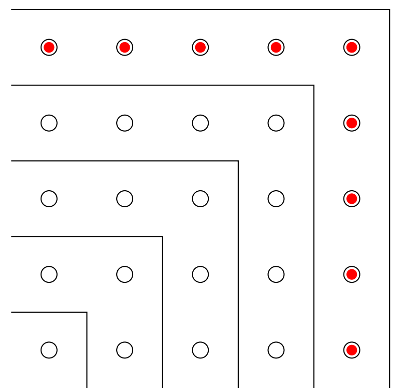

Figure 1: Domain division to compute $fitness$ iteratively, where each point is a possible pair $(a, b) \in S_i$.

The naive computation of $fitness$ computes all possible pairs of distinct values to count the pairs that violate the sum-free property, which can be seen as the map $S_i \times S_i \to \mathbf{N}$ as $(a, b) \to a + b, a \neq b$, and we verify if $a + b \in S_i$. We can use the symmetry of the map to reduce the number of computations by half, since addition is commutative. This still gives $O(k_i^2)$ comparisons to make.

This is not suited for an iterative algorithm, as all $a, b \in S_i$ are repeatedly verified. Therefore, there is a need to avoid re-computation, and thus, re-partition the domain better.

To this end, we notice from the proof of Prop. 2 that each addition creates pairs of two types: 1) $a, b \in S_i : a + b = N$ and 2) $a, b \in S_i : a + N = b$, where N is the number added in the current iterations. Furthermore, the type of pair that can occur necessarily depends on N. This is the subject of Prop. 3.

Proposition 3. *Let $S = \bigsqcup_i S_i$ such that $S_i \leftarrow S_i \cup \{N\}, N \in \mathbf{N} - S$. Then, for S_i can be partitioned into $S_i(N) = \{a \in S_i : a \leq N\}$ and $S_i^c(N) = S_i - S_i(N)$, for all pairs of the form $a + b = N$, $a, b \in S_i(N)$, and pairs of the form $a + N = b$, $b \in S_i^c(N)$, $a \in S_i$.*

Proof. The proof can be carried out by enumerating the cases as follows.

- Let $S_i(N) = \{a \in S_i : a \leq N\}$. Then, all pairs in $S_i(N)$ are of the type $a + b = N$. We see this by contradiction: if there exist a such that $a + N = b \in S_i(N)$, then N cannot be the maximal element of $S_i(N)$. Thus, for $S_i(N)$, the verification condition becomes $a = N - b \in S_i(N)$.
- Let $S_i^c(N) = S_i - S_i(N) = \{a \in S_i : a > N\}$ be non-empty. All pairs are now of the form $a + N = b \in S_i^c(N)$, so the verification condition passes to $\forall b \in S_i^c(N), b - N = a \in S_i$. This is shown as follows.
  - For $S_i^c(N)$, pairs are of the form $a+N = b \in S_i^c(N)$. No assumption is made on $a \in S_i$. By contradiction, suppose that $a + b = N$, such that $a, b \in S_i^c(N)$. Then $a < N, b < N \implies a, b \in S_i(N)$, which is false.
  - Lastly, suppose that $\exists a \in S_i(N), b \in S_i^c(N)$ such that $a+b = c \in S_i$. If $a \neq b \neq N$, then the pair does not involve N and is trivially not considered. Only the case $a = N$ remains. Then, $N + b = c$, where $b \in S_i^c(N)$ and so $c \in S_i^c(N)$.

□

The implementation of $fitness$ can now be split into the counting of the two possible types of pairs:

- **Type 1**: $a + b = N$, which can be formulated as the check $N - b = a \in S_i$, and the counting formula $|\{b : N - b \in S_i\}|$
- **Type 2**: $a + N = b$, which gives the check $b - N = a \in S_i$, and the counting formula $|\{b : b - N \in S_i\}|$.

Most importantly, this represents a reduction in time of $O(k_i^2)$ to $O(2k_i) = O(k_i)$, with implementations using hash maps or similar tools. Here. $k_i = |S_i|$ is the cardinality of the partition to which N is being added.

3 Extension to Multiple Processes

3.1 Out-of-order Selection

Out-of-order selection essentially presents the argument that in Alg. 1, $num \leftarrow num+1$ can be replaced by a generic $num \leftarrow getNumber(num)$ (This has the natural signature $getNumber : \mathbb{N} \to \mathbb{N}$).

This is immediately suggested by prop 1, where $fitness$, as shown to be a monotonically increasing function of the size of the partition n. Since no assumptions on the added number N were made, it holds true for any N that we add.

The proposal is to simulate and theoretically quantify the differences obtained when using such strategies. For example, $S = \{\{1, 2\}, \{3\}\}$ and $S = \{\{1, 3\}, \{2\}\}$ are both valid partitions using 2 colors, but may not lead to the same partitions over large iterations.

Therefore, if each coloring is a function $Col : \{1, \ldots, N\} \to \{1, \ldots, n\}$, where n is the number of colors, then understanding out-of-order selection can help us choose better maps Col.

3.2 Multi-Process Algorithm

After motivating an out-of-order selection, Alg. 2 details the multi-process algorithm with $getNumber$. **This is still a work in progress.**

```
Data: n ≥ 0, m ≥ 1
Result: BestSolution
Solutions ← Dict(keys : {1 ... m}, values :
  Dict(keys : {1 ... n}) );
while BreakCondition(n,num, Choice) do
    for Process_j in ProcessPool do
        num ← getNumber(num, j);
        for sol_i ∈ Solutions[j] do
            sol_i.append(num);
            Fitness[j][i] ←
              EvaluateFitness(sol_i);
        end
        BestSolutions[j] ←
          Choice(Solutions[j], Fitness[j]);
    end
    BestSolution ←
      Choice(BestSolutions, Fitness);
end
```

4 Current State and Further progress

The following have already been done:

- Implemented a correct, linear-order implementation of the fitness function, which can perform out-of-order addition correctly.

Currently, the following need to be implemented:

- Parallelizing the algorithm, either in Python or C++.

Further work is centered around the following axes:

- Characterise the out-of-order dynamics, possibly theoretically, generated by these algorithms, i.e. the effects of out-of-order addition over many iterations.
- Prove the algorithm is optimal, sub-optimal or worse.