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: $n \geq 0$
Result: *Solutions*
$Solutions \leftarrow Dict(keys : \{1 \dots n\})$;
$num \leftarrow 1$;
while *BreakCondition(n,num, Choice)* **do**
  $num \leftarrow num + 1$;
  for $sol_i \in Solutions$ **do**
    sol_i.append(num);
    $Fitness[i] \leftarrow EvaluateFitness(sol_i)$;
  end
  $bestSolution \leftarrow$
   $Choice(Solutions, Fitness)$;
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

2.1 Bounds

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

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

Lemma 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 $S_i \cup \{N\}$. □

Lemma 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, assume we add $N \in \mathbb{N} - S$ to S i.e. $\exists i \in [n] : S_i \leftarrow S_i \cup \{N\}$. Then, $fitness(S \cup \{N\}) \leq k +^{|S_i+1|} P_3 - fitness(S_i)$.*

Proof. Consider the naive bound that adding N creates all possible triplets i.e. any triplet $(a,b,c) \in S_i^3$ such that $a \neq b \neq c$ now violates the weakly sum-free property. Then, there are ${}^{|S_i+1|}P_3$ total pairs and the number of new pairs is:

$$\begin{aligned} & fitness(S \cup \{N\}) - fitness(S) \\ & \qquad \leq {}^{|S_i+1|} P_3 - fitness(S_i) \\ \implies & fitness(S \cup \{N\}) \leq fitness(S) + \\ & \qquad {}^{|S_i+1|}P_3 - fitness(S_i) \quad (2) \end{aligned}$$

as desired. □

Since ${}^nP_3 = O(n^3)$, this bounds the growth of $fitness$; however, this may still be improved.

2.2 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 Lemma 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.

2.3 Multi-Process Algorithm

After motivating an out-of-order selection, Alg. 2 details the multi-process algorithm with $getNumber$. It still remains to discuss the interaction of each Process's solutions, after their computation.

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