A Quadratic Time Locally Optimal Algorithm for NP-hard Equal Cardinality Partition Optimization
Kaan Gokcesu, Hakan Gokcesu

Abstract—We study the optimization version of the equal cardinality set partition problem (where the absolute difference between the equal sized partitions’ sums are minimized). While this problem is NP-hard and requires exponential complexity to solve in general, we have formulated a weaker version of this NP-hard problem, where the goal is to find a locally optimal solution. The local optimality considered in our work is under any swap between the opposing partitions’ element pairs. To this end, we designed an algorithm which can produce such a locally optimal solution in $O(N^2)$ time and $O(N)$ space. Our approach does not require positive or integer inputs and works equally well under arbitrary input precisions. Thus, it is widely applicable in different problem scenarios.

I. INTRODUCTION
A. Preliminaries

In the traditional set partition problem [11–4], the goal is to decide whether a given input set $X$ of positive integers can be partitioned into two disjoint complementary subsets $X_1$ and $X_2$ such that the sum of the elements in $X_1$ equals to the sum of the elements in $X_2$. A closely related problem to the set partitioning is the subset-sum problem [5], where the goal is to find a subset, sum of which equals to a target value $T$ (which can be solved with a set partition solver by adding a dummy sample). In our paper, we investigate the equal cardinality partition problem, which is a variant where both partitions should have an equal number of items on top of having an equal sum. Specifically, we deal with the optimization version, which is to partition the set $X$ into two disjoint subsets $X_1$ and $X_2$ of equal size such that the absolute difference between the sum of elements in $X_1$ and the sum of elements in $X_2$ is minimized.

The traditional problem is one of Karp’s 21 NP-complete combinatorial problems [6] and one of Garey and Johnson’s 6 NP-complete fundamental problems [7]. The equal cardinality variant is also NP-complete [7] and the optimization version is NP-hard [8]. [9]. The traditional one has also been dubbed "the easiest hard problem" in literature because of its limited structure in comparison to other NP-complete problems [10], [11]; because, despite its NP-completeness, there exist methods that can solve it optimally, approximately or locally in many instances (moreover, the set of viable solutions can be exponentially large). There exists pseudo-polynomial time algorithms, where the run time is polynomially dependent on the numeric values of the inputs in contrast to the length of the input [7]. There exists efficient locally optimal solvers for the traditional set partition [12]. All in all, it is a well-studied branch of combinatorial optimization with many important developments [4], [9], [13–19] and may imply similar possible improvements for harder NP-complete problems [19].

B. Notable Applications

The set partitioning has a lot of related applications to problems in learning, optimization and decision theory, e.g., scheduling, allocation, classification and regression [20–49].

The knapsack problem [31] is one such closely related application, which governs the following problem: given a set of items with varying weights and values, determine the number of each said item in a collection so that the total weight is constrained by an upper bound and the total value is maximized. This problem often arises in resource allocation settings where we need to choose from a set of non-divisible tasks (or goods) under a time (or budget) constraint. The knapsack problem is a century-old problem [33] and its name dates back to the early works of Dantzig [34]. Its applications are numerous in real-world decision-making processes in a wide variety of fields, such as finding the least wasteful way to cut raw materials, selection of investments and portfolios, selection of assets for securitization, and generating keys for the Merkle–Hellman and other knapsack cryptosystems [31].

The bin packing optimization problem [35] is another closely related one, where we need to pack certain items with varying sizes into a minimum number of bins with a fixed capacity. It has a myriad of applications from filling containers to loading trucks and creating file backups.

Another related problem is the fair division [36], which is the problem of fairly dividing a set of resources among a certain number of recipients. It arises in many real-world scenarios such as division of inheritance, dissolution of partnerships, settlements in divorces, frequency allocation in band usage, traffic management of airports or harbors, satellite allocations and social choice theory [35], [37].

Moreover, in some works, the partition problem is synonymous with parallel program scheduling [45], where the goal is to assign various tasks to the parallel working machines while minimizing the total amount of time it takes to their completion [7], [44–46]. A similar application is in berth allocation [47], which aims to optimally allocate berth space for the incoming vessel in container terminals while minimizing the total time it takes to serve all the vessels [47], [48].

Fair team selection is a famous problem related to set partitions, where the goal is to create fair teams in regards to the total skill level of the players, i.e., minimizing the difference between the cumulative skill in different teams [10]. Another famous example is for the manipulation of veto elections, where each elector has different veto weights and the candidate with the smallest total veto wins the election. Using set partitioning, a coalition can maximize their candidate’s chances [39].
C. Related Works

Given an input set \( X \) of size \( N \), the most straightforward algorithm for the set partition problem is the brute force approach, which has \( O(2^N) \) time complexity. The problem is NP-hard and its solution takes exponential time to find, however, there exist more efficient approaches. An example is the algorithm in [50], which finds an optimal solution in \( O(2^{N/2}) \) time but also uses \( O(2^{N/2}) \) memory. While it has better run time than brute force, it also uses exponential memory. The algorithm in [51] addresses this issue and decreases the space complexity to \( O(2^{N/4}) \) while keeping the time complexity the same. Even though the problem is NP-hard, there are dynamic programming approaches as well [7], [52], [53], which can find an optimal solution in pseudo-polynomial time and space. Specifically, their complexities are polynomially dependent on the sum of inputs \( S \). Note that, in such approaches the inputs are assumed to be positive and integer. Thus, their performance is highly dependent on the precision of the inputs. There also exists sub-optimal algorithms like the greedy algorithm and the Karmarkar-Karp set differencing algorithm, which run in \( O(N \log N) \) time and \( O(N) \) space [13], [31], [54], [55]. The complete anytime algorithm in [4] can transform such efficient sub-optimal algorithms into optimal ones. It creates a binary tree from the sub-optimal algorithm selections (much like a modification of the brute force approach) to create anytime algorithms with linear memory usage. However, its worst-case time complexity is \( O(2^N) \). There are also algorithms which can produce locally optimal solutions for the set partition problem in \( O(N \log N) \) time and \( O(N) \) space [12]. Instead of a seemingly arbitrary sub-optimal heuristic, a locally optimal solver may prove to be more useful in many scenarios.

D. Contributions and Organization

Even though there exists optimal algorithms, they have exponential time (albeit better than brute force) and space complexities. There also exists an optimal solver with linear \( O(N) \) memory, however, its time complexity is the same as brute force in the worst case. For efficient approaches, dynamic programming methods find an optimal one with pseudo-polynomial complexity. However, it has limited use for high precision or non-integer inputs. Although there are linear-time complexity algorithms, they generally produce sub-optimal solutions. While there exists a locally optimal algorithm for the set partitioning, it is not applicable to the equal cardinality variant. Improving the exponential complexity is unfruitful because of the NP-hardness of the problem but fast algorithms are always desired especially with the emergence of big data. To this end, much like [12], we postulate that finding a ‘locally’ optimal solution to the equal cardinality set partition problem is nowhere near as hard as finding a ‘globally’ optimal one. In Section II, we mathematically formulate the problem definition. In Section III, we design an algorithm and prove that its locally optimal solution takes \( O(N^2) \) time and \( O(N) \) space to find. In Section IV, we provide some important discussions, show the extension to the traditional set partition problem and conclude with a few remarks.

II. Locally Optimal Equal Cardinality Partition

In this section, we formally define the equal cardinality set partition problem. As an input, we have the set of numbers

\[ X = \{x_1, x_2, \ldots, x_N\}, \]

\[ = \{x_n\}_{n=1}^N, \]

where \( N \) is an even number.

We partition \( X \) into two equally sized subsets \( X_1 \) and \( X_2 \) such that they are disjoint and their union is \( X \), i.e.,

\[ X_1 \cap X_2 = \emptyset, \]

\[ X_1 \cup X_2 = X, \]

\[ |X_1| = |X_2| = \frac{N}{2}, \]

where \(| \cdot |\) is the cardinality. The goal is to create the sets \( X_1 \) and \( X_2 \) such that their individual sums are as close as possible to each other, where the set sums are given by

\[ S_1 = \sum_{x \in X_1} x, \]

\[ S_2 = \sum_{x \in X_2} x, \]

Hence, \( S_1 \) and \( S_2 \) are each other’s complements, i.e., \( S_1 = S - S_2 \) and \( S_2 = S - S_1 \), where

\[ S = \sum_{x \in X} x. \]

We consider the formulation where we want to minimize the absolute difference between the sums \( S_1 \) and \( S_2 \), i.e.,

\[ \min_{X_1,X_2} (|S_1 - S_2|). \]

However, we point out that the formulation of minimizing the maximum sum, i.e.,

\[ \min_{X_1,X_2} (\max(S_1, S_2)), \]

or the formulation of maximizing the minimum sum, i.e.,

\[ \max_{X_1,X_2} (\min(S_1, S_2)), \]

are all equivalent. Since we are only dealing with two sets and their corresponding sums, in general, all such formulations become equivalent to each other [12].

This problem is NP-hard [6] and impossible to solve with an efficient method. To this end, instead of this NP-hard problem, we consider a ‘weaker’ version of the equal cardinality set partition problem. Instead of a global optimal solution, we are after a ‘locally’ optimal one, which is defined as follows:

Definition 1. A partition \((X_1, X_2)\) of \( X \) is locally optimal if there is no single swap of element pairs (belonging to different sets) that can decrease the absolute difference between the set sums \( S_1 \) and \( S_2 \), i.e.,

\[ |(S_1 - x_1 + x_2) - (S_2 + x_1 - x_2)| \geq |S_1 - S_2|, \]

for all \( x_1 \in X_1 \) and \( x_2 \in X_2 \).

Note that this local optimality definition is fundamentally different from [12]. Theirs was under any single element transfer between the partitions, which is not meaningful for equal cardinality case.
III. FINDING A LOCALLY OPTIMAL SOLUTION TO EQUAL CARDINALITY SET PARTITION IN QUADRATIC TIME

A. Iterative Algorithm

Before we propose the algorithm, we make some initial assumptions.

Assumption 1. Let the set $\mathcal{X}$ be composed of only positive elements, i.e.,
\[ x > 0, \quad \forall x \in \mathcal{X}. \]

Assumption 2. Let the set $\mathcal{X} = \{x_n\}_{n=1}^N$ be in ascending order, i.e.,
\[ x_n \leq x_{n+1}, \quad \forall n \in \{1, 2, \ldots, N-1\}. \]

Thus, our input $\mathcal{X}$ is a positive ordered set. However, we point out that there is no requirement for the elements to be integers, i.e., they can be reals with arbitrary precisions.

Given the input set $\mathcal{X}$, the algorithm works as follows:

1) We start with an arbitrary partition of $\mathcal{X}$: $\mathcal{X}_1$ and $\mathcal{X}_2$ with the initial sums $S_1$ and $S_2$ respectively.

2) Set $n = 1$

3) Check if swapping $x_n$ with any element $x_{n'}$ $(n' < n)$ from the opposing set strictly decreases the absolute sum difference $|S_1 - S_2|$.

4) If yes, continue;
   - else if $n = N$, STOP;
   - or else, set $n \leftarrow n + 1$ and return to Step 3.

5) Swap $x_n$ with the element in the other set that decreases the absolute sum difference. For example, if $x_n \in \mathcal{X}_1$, swap $x_n$ with $x_{n^*}$, where
\[ n^* = \arg \min_{n' < n : x_{n'} \in \mathcal{X}_2} |S_1 - S_2 - 2x_n + 2x_{n'}|, \]
and make the necessary updates
\begin{align}
\mathcal{X}_1 &\leftarrow (\mathcal{X}_1 \setminus \{x_n\}) \cup \{x_{n^*}\}, \\
\mathcal{X}_2 &\leftarrow (\mathcal{X}_2 \setminus \{x_{n^*}\}) \cup \{x_n\}, \\
S_1 &\leftarrow S_1 - x_n + x_{n^*}, \\
S_2 &\leftarrow S_2 - x_{n^*} + x_n.
\end{align}

6) If the sign of $S_1 - S_2$ remains unchanged,
   - if $n = N$, STOP;
   - or else, set $n \leftarrow n + 1$ and return to Step 3;
   - or else, return to Step 2.

Remark 1. Heuristics can be used in Step 4 to speed up the algorithm. Initial state is inconsequential in the worst-case.

Proposition 1. The algorithm definitely terminates.

Proof. The algorithm terminates, i.e., stops, only when $n = N$ in the algorithm. When $n < N$, it either returns to Step 3 or it returns to Step 2 if the sign of $S_1 - S_2$ changes after a swap.

We know that with each swap the absolute difference $|S_1 - S_2|$ strictly decreases because of Step 3. Since there are a finite number of elements, hence, finite possible swaps; the sign of $S_1 - S_2$ cannot keep changing indefinitely. Thus, the return to Step 2 happens a finite amount of times. Together with the fact that whenever we return to Step 3, we have already increased $n$ by 1; the algorithm reaches $n = N$ after a finite number of iterations and definitely terminates.

B. Local Optimality

In this section, we prove the local optimality of our algorithm, where we start with a few useful results.

Proposition 2. The absolute sum difference can only be decreased by swapping two elements, where the larger one belongs to the set with the larger sum.

Proof. Without loss of generality let $S_1 - S_2 > 0$. For the absolute sum difference $|S_1 - S_2|$ to decrease the swapped elements $x_1$ and $x_2$ should have the inequality $x_1 > x_2$. Otherwise, $S_1 - S_2$ will get larger and so is $|S_1 - S_2|$. □

Proposition 3. If swapping elements in different sets $x_{n_1}$ and $x_{n_2}$ for some $n_1 > n_2$ does not decrease the absolute difference of set sums $|S_1 - S_2|$, neither does swapping $x_{n_1}'$ (for some $n_1' \geq n_1$) with $x_{n_2}'$ (for some $n_2' \leq n_2$).

Proof. The proof is straightforward from the fact that $\{x_n\}_{n=1}^N$ is in ascending order, since the difference $x_{n_1} - x_{n_2}$ is less than or equal to $x_{n_1}' - x_{n_2}'$ for any $n_1' \geq n_1$ and $n_2' \leq n_2$. □

Proposition 4. Without loss of generality let $S_1^{old} - S_2^{old} = \Delta^{old} > 0$. If swapping elements in different sets $x_1 \in \mathcal{X}_1$ and $x_2 \in \mathcal{X}_2 (x_1 > x_2)$ did not decrease the absolute difference of set sums; neither does swapping them when $S_1^{new} - S_2^{new} = \Delta^{new} > 0$, where $\Delta^{new} \leq \Delta^{old}$.

Proof. The proof comes from the fact that the pair $x_{n_1}$ and $x_{n_2}$ is a minimizer swap over the swaps including $x_{n_1}$. If there indeed were a feasible $x_{n_3}$, the minimizer swap would simply be between $x_{n_1}$ and $x_{n_3}$.

Lemma 1. At the stop, we have a local optimal solution.

Proof. At the algorithm’s termination, we make a final traverse from $n = 1$ to $n = N$ all the while never returning to Step 2 from Step 6. In this traverse, $n$ increases by 1 if swapping $x_n$ with any $x_{n'}$ $(n' < n)$ is unfruitful or we make a swap and the sign of $S_1 - S_2$ remains unchanged. Without loss of generality let $S_1 - S_2 > 0$. If there are no swaps in the traverse, that means there are no pairs $x_{n_1} \in \mathcal{X}_1$ and $x_{n_2} \in \mathcal{X}_2 (n_1 > n_2)$ whose swap decreases the absolute difference of set sums, which implies the local optimality. Suppose there is a swap between $x_{n_1} \in \mathcal{X}_1$ and $x_{n_2} \in \mathcal{X}_2 (n_1 > n_2)$ whose swap decreases the absolute difference of set sums. From Proposition 4, we have that $x_{n_1} \in \mathcal{X}_1$ for $n < n_1$ and $n_1 \neq n_2$ still has no viable swaps. Moreover, from Proposition 5, we have that $x_{n_2}$ has no viable swap. Thus, $x_n \in \mathcal{X}_1$ for $n < n_1$ has no viable swap. Henceforth, in the final traverse, the swaps eliminate any remaining viable swaps while not affecting the smaller elements. At $n = N$ we will end up with a local optimal solution. □
C. Complexity Analysis

Since our algorithm only keeps track of which elements are in which partition, it uses \( O(N) \) space.

Lemma 2. In the algorithm, whenever a swap happens between \( x_{n_1} \in X_1 \) and \( x_{n_2} \in X_2 \) (without loss of generality let \( S_1 - S_2 > 0 \) and \( n_1 > n_2 \)), we have before the swap:\n\[ x_{n_1 - 1} \in X_2 \text{ and } \max_{n < n_1} x_n \in X_1, n < n_2. \]

Proof. The algorithm checks \( x_{n_1} \in X_1 \) only if \( x_{n < n_1} \in X_1 \) has no viable swaps. Together with Proposition 3 concludes the proof.

Lemma 3. Any swap in the algorithm can be expressed as a series of swaps between consecutive elements.

Proof. From Lemma 2 we have that whenever there is a swap between \( x_{n_1} \in X_1 \) and \( x_{n_2} \in X_2 \) (without loss of generality let \( S_1 - S_2 > 0 \) and \( n_1 > n_2 \)), all the elements between them are in \( X_2 \), i.e., \( x_{n_1 > n_2} \in X_2 \). Thus, this swap is equivalent to swapping \( x_{n_1 - 1} \) and \( x_{n_1 - 1} \) consecutively for \( i \in \{0, 1, \ldots, n_1 - n_2 - 1\} \), which concludes the proof.

Lemma 4. In the algorithm, whenever a sign change happens on \( S_1 - S_2 \) after a swap between \( x_{n_1} \in X_1 \) and \( x_{n_2} \in X_2 \) (without loss of generality let \( S_1 - S_2 > 0 \) and \( n_1 > n_2 \)), we have \( |S_1 - S_2 - 2x_{n_1} + 2x_{n_2}| \leq |x_{n_1 - 1} - x_{n_2}|. \)

Proof. The lemma comes from Lemma 3. Since a series of swaps between consecutive elements results in the sign change. The last swap in the series results in the sign change, which concludes the proof.

Lemma 5. In the algorithm, each traverse over \( n \) (i.e., from \( n = 1 \) to again \( n = 1 \)) takes \( O(N) \) time.

Proof. Without loss of generality let \( S_1 - S_2 > 0 \) in our traverse (note that whenever the sign changes we begin a new traverse). Suppose we are at some time iteration \( n_1 \) and we have \( \max_{n < n_1} x_n \in X_1 \) has no viable swaps. From Lemma 2, we know that \( x_{n_1} \) can only be swapped with \( x_{n_1 < n_1} \) (all of which are in \( X_2 \)). We start by checking the swap with \( x_{n_1 = n_1 + 1} \) and do a simple linear search to find the best swap, which takes \( O(n_1 - n_1) \) time. Since we iteratively do this for all elements in \( X_1 \) in a single traverse, it takes at most \( O(N) \) time, which concludes the proof.

Lemma 6. In the algorithm, we have at most \( O(N) \) traverses.

Proof. A traverse end after a swap which causes a sign change in \( S_1 - S_2 \). From Lemma 4 we have that whenever the sign changes, the resulting absolute difference between the set sums will be less than the difference between some consecutive elements in \( X \). Since with each swap, the absolute difference decreases, this consecutive swap can never happen again after it has happened once. Since there are \( N - 1 \) possible consecutive element pairs, the number of traverse can never exceed \( O(N) \), which concludes the proof.

Theorem 1. Our algorithm takes \( O(N^2) \) time.

Proof. The result comes from combining the results of Lemma 5 and Lemma 6.

IV. Discussions and Conclusion

Our algorithm does not use the fact that \( x_n \) are positive, hence, it is redundant. However, since we are dealing with the equal cardinality set partition problem, note that, any arbitrary scaling or translation has no effect on the partitioning. Thus, any affine transform of \( x_n \) in the form of \( \alpha x_n + \beta \) for nonzero \( \alpha \) produces the same partitioning. The invariance under arbitrary scalings is true for any possible set partition not just equal cardinality. However, note that, the translation invariance only exists for the partitions with equal cardinality in general, since the effect of \( \beta \) will be \( \frac{\beta}{2} \) on both partitions.

If \( X \) is unordered, it takes \( O(N \log N) \) time and \( O(N) \) space to order it, which does not change our complexity since our algorithm runs in \( O(N^2) \) time and \( O(N) \) space.

Note that our local optimality definition is fundamentally different from [12], which studies general set partitions. The results in [12] are locally optimal under any element transfers between the partitions, which is not applicable in our problem, since the equal cardinality partition does not allow an element to transfer between partitions by itself. To this end, our solutions are locally optimal under any swap between elements.

We point out that our algorithm can be straightforwardly extended to the case where we search for a local optimal partitioning with possibly different cardinalities as in [12]. We first observe there is a simple reduction from the equal cardinality variant to the traditional set partitioning. Any algorithm that can solve the equal cardinality problem can be used for the traditional set partition problem by addition of some dummy zero elements. Hence, the equal cardinality variant is at least as hard as the traditional set partition problem. To apply our algorithm in the traditional set partition problem, we need to extend the input set \( X \) of size \( N \) to a set of size \( 2N \), which includes all the elements in \( X \) and \( N \) zeros. After running our algorithm on this set, we will end up with a partitioning that has a stronger local optimality than [12]. Since in this case, the solution will be not only optimal with regards to any single element transfers but also any swap between element pairs in different sets. Note that, the stronger optimality comes together with an increase in the complexity of the algorithm.

Moreover, it is straightforward to apply our algorithm to not just the equal cardinality setting but any predefined cardinalities. Since our algorithm works by swapping element between partitions, the initial cardinalities will be preserved.

In conclusion, we have studied the optimization version of the equal cardinality set partition problem (where the absolute difference between the partition sums are minimized and the sets have equal cardinality). While this problem is NP-hard and requires exponential complexity to solve in general, we have formulated a weaker version of this NP-hard problem, where the goal is to find a locally optimal solution. The local optimality considered in our work is under any single swap between the opposing partition elements. To this end, we designed an algorithm which can produce such a locally optimal solution in \( O(N^2) \) time and \( O(N) \) space. Our approach does not require positive or integer inputs. Our algorithms work equally well under arbitrary input precisions, hence, widely applicable in difference problem scenarios and settings.
