Where to Put A Rower: A Novel and Practical Solution to Dragon Boat Partition Problem

Brett Regnier
University of Lethbridge
Lethbridge, Alberta Canada, T1K 3M4
brett.regnier@uleth.ca

John Zhang
University of Lethbridge
Lethbridge, Alberta Canada, T1K 3M4
john.zhang@uleth.ca

***Abstract*—We present a novel and practical approach to the Dragon Boat Partition Problem (DBP), in which we need to assign rowers to the opposite sides of a dragon boat, such that certain requirements are satisfied. An example of the requirements is that weights on the right side and left side are balanced. While DBP is essentially an integer partition problem, it has its own unique characteristics. Our approach is greedy and heuristic and tackles DBP effectively, showing its practicality in real applications. The performance of our proposed approach is demonstrated through a set of simulated experiments and various related issued are discussed.**

I. Introduction

Throughout the world there are hard problems, many of which require great thought and enormous computations, given state of the art in the current computational techniques. It is safe to say that almost all the non-trivial problems, including satisfiability, independent Set, etc. [2], fall into the category. The goal of optimization, according to Pedregal et al. [12], is to find an optimal solution to a problem, by satisfying two important criteria for optimality: (1) the computational cost must be minimized; and (2) constraints must be explicitly enforced. The optimization problem we will be focusing on in this work is the Dragon Boat Partition problem (*DBP*, for short), a variant of the integer partition problem [2].

Before we discuss DBP, we first introduce the integer partition problem. Given a set of n positive integers, we try to find a partition of the set into two subsets such that the sum of the numbers in one subset is equal to the one of the other [4]. Formally, given $S \subset \mathbb{Z}^+$, find a partition $S_1 \subset S$, and $S_2 \subset S$ where $S_1 \cap S_2 = \emptyset$, such that the difference known as discrepancy,

$$E(A) = \sum_{s \in S_1} s - \sum_{s \in S_2} s \tag{1}$$

is minimized [10]. A perfect partition is defined as $E = 0$ for when $\sum_{s \in S} s$ is even, and $E = 1$ for when $\sum_{s \in S} s$ is odd. Despite its simple appearance, the partition problem is deceptively hard. The problem on a small set, such as equally spitting $S = \{1, 2, 3, 4, 5\}$ into two subsets, is trivial; However, splitting a set with, say, one hundred elements, starts to become hard to achieve efficiently, since essentially we need to check every $n!$ possible combinations, where n is the number of the integers in the set [4]. It has been shown that the partition problem is NP-hard and its decision version, i.e., whether such a partition exists, is NP-complete [2], [12]. Therefore, the use of fast approximate algorithms is needed, such as the greedy heuristic [4], [10] or the Karmarker and Karp's differencing method [10]. The partition problem has a number of real-world applications, such as multiprocessor scheduling, state asset partitioning, VLSI circuit size minimization, minimizing the delay for public-key cryptography [4], [10]. The DBP problem is just one of them. We will discuss the related work in Section V. It should be noted that the DBP problem further requires that the number of rowers on the left side be equal to the number of rowers on the right and also the number of rowers on the front half be equal to the number of rowers on the back half (as shown below).

A dragon boat is a long slender boat with 22 seating arrangements as shown in Figure 1. A boat is filled with a team of 22 members, with a drummer, who sets the pace and sits at the front of the boat facing the team, and a steersman, who directs the boat and sit at the back of the boat. The remaining are the 20 rowers that paddle, sitting side by side down the boat [5]. While the selection of the drummer and steersman is relatively easy, mainly depending on their special skills, configuring boat rowers such that some constraints, as shown below, are satisfied is hard. At present a practice of ad-hoc decisions by a leader based on rowers' physical features is employed.

Before the boat rowers get into the boat, they must first be approximately partitioned. Having a balanced dragon boat, i.e., both sides have roughly equal weights and the weight difference between the front half and back half is within certain range, ensures that the boat is parallel with the water, allowing maximum surface area and water flow around the boat and increasing boating efficiency and speed [14]. For example, if the back half is heavier it would cause the front half to lift and the back half to sink deeper into the water, effectively slowing down the boat.

To find an approximate optimal partitioning, multiple constraints must be considered. There are certain positions in the boat that each rower prefers to be in and/or is especially good at. After rowers are placed in their preferred position, the next step is to position the rowers by their dominant hand such

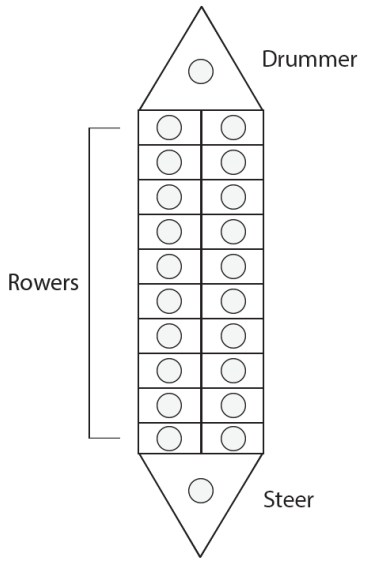

Fig. 1. Abstract view of a dragon boat

that the weight difference between the left side and right side is minimized. Then the rowers at the front and back halves are determined where the difference between the front half and back half must be close to 30lbs, with the front half being heavier. Lastly, the heaviest members of the team should be positioned in the center of the boat and the weights are gradient outwards towards the front and back [14]. While the DBP problem is not proposed by us, to our knowledge this is the first work that takes this problem seriously from a computational viewpoint.

II. The Dragon Boat Partition Problem Statement

The Dragon Boat Partition problem (*DBP*, for short) is a variant of the partition problem. If we only require that the left-hand side and right-hand side have equal weight, then we reduce the partition problem to the DBP problem through restriction and this reduction takes obviously polynomial time. Since the partition problem is NP-hard, so is the DBP problem.

The objective of the DBP problem is to partition the rowers for a dragon boat with four constraints. (1) Each rower must be placed based on their dominant hand; (2) The total weight of the rowers on the left-hand side must be approximately equal to the total weight of those on the right-hand side; (3) The total weight of the rowers on the front half of the boat must be approximately heavier by a certain threshold (in practice, 30lbs is selected) than the total weight of those on the back half of the boat; and (4) The number of rowers on the left side should be the equal to the number of the right side and so should the number of rowers on the front half be equal to the number of the back site.

We define the left-handedness as a rower's ability to row effectively with their left hand and refer to them as *left rowers*, and similarly for the right-handedness (*right rowers*). A rower may also have the ability to row both left-handed and right-handed and we refer to them as *ambidextrous rowers*. The left-handedness is denoted as a Boolean value $p_l \in \{0, 1\}$ and similarly $p_r \in \{0, 1\}$ denotes the right-handedness. As such, if a rower has a $p_l = 1$ and $p_r = 1$, then they are an ambidextrous rower. Finally, a rower's weight is a positive integer number in pounds (lbs) denoted by $p_w \in \mathbb{Z}^+$.

The DBP problem can now be formally defined as follows. Assume that there is a set P representing boat rowers, where rower $p \in P$ is a 3-tuple (p_w, p_l, p_r). We assume that $|P|$ is even.

Let $L \subset P$ where $|L| = |P|/2$ and $\forall p \in L, p_l = 1$ denote the left group, and let $R \subset P$ where $|R| = |P|/2$ and $\forall p \in R, p_r = 1$ denote the right group. It should be noted that an ambidextrous rower can be either in L or R, since its $p_l = 1$ and $p_r = 1$. Similarly, let $F \subset P$, where $|F| = |P|/2$, denote the front group, and $B \subset P$, where $|B| = |P|/2$, denote the back group.

Let $L_w = \sum p_w \ \forall p \in L$ be the total weight of the rowers in the left-group, $R_w = \sum p_w \ \forall p \in R$ of the rowers in the right-group, $F_w = \sum p_w \ \forall p \in F$ of the rowers the front-group, and $B_w = \sum p_w \ \forall p \in B$ of the rowers in the back-group.

Let V_{lr} and V_{fb} represent the left-right and front-back optimal *discrepancy relaxation* values, respectively. $W_{lr} = |L_w - R_w|$ is the left-right weight discrepancy and $W_{fb} = F_w - B_w$ is the front-back weight discrepancy. The reason we do not use an absolute value for W_{fb} is that F_w is bigger than B_w, i.e., $W_{fb} > 0$, as discussed before.

We use W^*_{lr} to denote the optimal weight discrepancy between the left and right and W^*_{fb} the optimal weight discrepancy between the front and back. Note the difference between V_{lr} and W^*_{lr} and the difference between V_{fb} and W^*_{fb}. It is desirable that W_{lr} and W_{fb} be as close to W^*_{lr} and W^*_{fb}, respectively, as possible. Typically $W^*_{lr} = 0$ while $W^*_{fb} = 30$ with the front heavier. It is obvious that the rowers on the left side are disconnected from the right side group but can be in either the front half or back half, depending on the seat they are assigned to. The same applies to the right side as well. The same concept applies to the front half and back half, as any rower assigned to the front half cannot exist in the back half and vise versa, but can be in either the left-side or the right-side.

A. Rules

In the DBP problem, we have the following rules to follow, when assigning rowers to the boat.

1) Handedness Rule: There cannot be any left-rowers on the right-side, nor any right-rowers on the left-side.
2) Left-Right Balance Rule: The left-right discrepancy must be close to the left-right optimal weight within the range of the left-right relaxation.
3) Front-Back Balance Rule: The front-back discrepancy be as close to the front-back optimal weight (with the front heavier) within the range of the front-back relaxation.
4) Weight Gradient Rule: The rowing participants should be graded by heaviest to lightest from the center of the board outward.

Our proposed approach, as presented in Section III, terminates when the first three rules are fulfilled. We do not enforce the last rule in our algorithm, since once the first three constraints are satisfied, it is just a sorting procedure that is applied the participants on the left side and right side. But we include it here for the sake of completeness and include it in our proposed algorithm in Section III.

Now the DBP problem seeks a partition where $|L| = |R|$, $|F| = |B|$, $L \cap R = \emptyset$, $F \cap B = \emptyset$, $L \cap (F \cup B) = L$, $R \cap (F \cup B) = R$, $(F \cap L) \cup (F \cap R) = F$, $(B \cap L) + (B \cap R) = B$ in which W_{lr} is close to W^*_{lr}, such that $W^*_{lr} - V_{lr} \leq W_{lr} \leq W^*_{lr} + V_{lr}$, and W_{fb} is close to W^*_{fb}, such that $B_w < F_w$ and $W^*_{fb} - V_{fb} \leq W_{fb} \leq W^*_{fb} + V_{fb}$, where typically $W^*_{lr} = 0$ and $W^*_{fb} = 30$.

B. Assumptions

Usually dragon boat teams have 22 participants in total, consisting of 20 rowers, one (1) steersman, and one (1) drummer [5]. We do not take into consideration benched rowers in our approaches. Therefore we assume that the rowers provided is the complete list and will be the only data to be manipulated. We make a rough estimate of human-like characteristics and attributes of a rower. Therefore, we assume that each rower weighs between 110lbs to 270lbs, rounded to an integer number. We also assume that most of the rowers are able to row on either side. It is easy to see that having many rowers that are only able to row left or right makes the problem trivial to approximate. Furthermore, we assume that a rower's height does not affect the partition process. Lastly, we assume that no rowers have a preferred position to be assigned to.

C. Input Features

An instance of the DBP problem can be represented and organized into a 3-dimensional array. The first dimension denotes the weight of an individual rower while the second and third dimension denote their handedness, as shown in Table I. Note that an ambidextrous rower has both $left = 1$ and $right = 1$.

TABLE I. INPUT FEATURES OF AN INSTANCE OF THE DBP PROBLEM.

Feature	*Data type*	*Description*
Weight	Integer	The weight of the participant
Left	Boolean	The ability to row left handed
Right	Boolean	The ability to row right handed

III. A NOVEL AND PRACTICAL APPROACH TO DBP

A. Our Proposed Heuristic Algorithm

With the appropriate notation in hand, we now present a simple but practical heuristic approach to the DBP problem. Some initial attempts of the proposed approach can be found in our work [3].

Our algorithm follows a series of optimizing operations, in which if it fails to find a suitable action given the current assignment, i.e., the arrangement of the rowers on the left/right sides and on the front/back halves, it will begin by searching with a wider range of discrepancy by relaxing the search size sz, which is defined as how far from the optimal weight difference the algorithm will accept. This will repeat until a suitable assignment is found. Upon sz reaching the sz_m, the maximum search size, our algorithm will try to revert to the best-known previous assignment and stop. Note that sz_m is a user controlled parameter. Sometimes a call to the reverting phase is necessary to revert to a previous best assignment.

It is obvious that the weight discrepancy between the left side (the front half) and the right side (the back half) is due to the fact that one side is heavier while the other is lighter. In the following description of our algorithm, we denote the heavier side as *HS* and denote lighter side as *LS*, both of which fail one of the rules as discussed in Section II,

B. Algorithm

```
/* Heuristic Algorithm */
Step 1: Cleaning Phase
while a rower P_1 breaking Handedness rule {
    Find a rower P_2 with opposite handed capability
    or with either-handed capability.
    Swap (P_1, P_2)
}
Calculate W_lr
Calculate W_fb
sz = 0

Step 2: Left-Right Partitioning Phase
while W_lr ≤ W*_lr − V_lr or W_lr ≥ W*_lr + V_lr {
  /* find the best discrepancy change */
  bestdiff = 0
  find next lightest ambidextrous rower on {LS} as P^x {
    find next heaviest ambidextrous rower on {HS} as P^y {
      diff = P^y_w - P^x_w
      if |W_lr − 2 * diff| < |W_lr − 2 * bestdiff| {
        bestdiff = diff
        if bestdiff == |W*_lr − W_lr| + sz
          goto checkswap
      }
    }
  }

  checkswap:
  if (P^x, P^y) in SwapMemory {
    /* if the same pair has been swapped repeatedly, increase
       the search size. */
    sz++
    if sz == sz_m {
      /* No perfect discrepancy can be found.
         Recall to last best assignment. */
      start Reverting Phase
      start Front-Back Partitioning Phase
    }
  } else {
      swap P^x and P^y /* swap function also adjusts
                          W_lr and W_fb. */
      save (P^x, P^y) to SwapMemory
      W_lr = abs(W_lr − 2 * curr_diff)
    }
  }

    if W_fb ! = W*_fb {
        start Front-Back Partitioning Phase
    }
}

/* When balancing the front half and back half, a
   left- (right-) rower can be swapped with a
   left- (right-) rower in order to maintain
   the balance between the left side and right
   side. We call such a pair a valid pair. */
Step 3: Front-Back Partitioning Phase
```

```
while W_fb < W*_fb − V_fb or W_fb > W*_fb + V_fb {
  /* find the best discrepancy change */
  bestdiff = 0
  if W_fb < W*_fb − V_fb {
    {LS} = front half
    {HS} = back half
  else /* W_fb > W*_fb + V_fb */ {
    {LS} = back half
    {HS} = front half
  }
  find next lightest rower on {LS} as P^x {
    find next heaviest rower on {HS} as P^y {
    if (P^x, P^y) is valid {
      diff = P^y_w − P^x_w
        if W_fb − 2 * diff ≤ W_fb − 2 * bestdiff {
          bestdiff = diff
          /* stop front-back partition phase, if the current
      swap will result
              in a perfect discrepancy between the front and
      the back. */
          if bestdiff == abs(W*_fb − W_fb) + sz
            goto checkswap
        }
      }
    }
  }

  checkswap:
  if (P^x, P^y) in SwapMemory {
    /* If the same pair has been swapped repeatedly, increase
       the search size. */
    sz++
    if sz == sz_m {
      /* No perfect discrepancy can be found.
         Recall to last best assignment. */
      start Reverting Phase
      /* After reverting to best known phase
         begin the next phase. */
      start Weight Gradient Partitioning Phase
    }
  } else {
    swap P^x and P^y /* swap function also adjusts W_fb */
    save (P^x, P^y) to SwapMemory
    W_fb = W_fb − 2 * bestdiff // W_fb > 0
  }

  if W_fb != W*fb
        Weight Gradient Partitioning Phase or Stop
}

Step 4: Weight Gradient Partitioning Phase (optional)
Call QuickSort with front-left group
Call QuickSort with front-right group
Call QuickSort with back-left group and Reverse
Call QuickSort with back-right group and Reverse

Step 5: Reverting Phase (if needed)
while SwapMemory is not empty {
  Pop most recent pair (P^y, P^x)
  Swap (P^x, P^y)
}
```

Algorithm 1. Dragon Boat Heuristic Partitioning Algorithm.

The above is a step-wise view of our algorithm. Each step/phase is implemented as a procedure. Some discussions are in order. A misplaced rower is the one who is a left rower is on the right side or vise versa. *SwapMemory* is a stack data structure that stores each swap's assignment, action, and weights. When it is called upon to revert, the most recent saved memory is the first to be removed and reverted to if necessary. Note that *swap* function adjusts W_{lr} and W_{fb} necessarily.

Cleaning Phase has only one objective, to satisfy the Handedness Rule in Section II. This phase finds any misplaced rower on one side and swaps them with a rower on the other side. If there is no such rower, the offending rower will simply be swapped with an ambidextrous one. This phase continues until there are no more misplaced rowers. We assume that there is at most an equal number of left rowers to the number of seats on the left side, and so is there the number of right rowers to the number of seats on the right side. Thus, there are at most $n/2$ left rowers, and at most $n/2$ right rowers. But in order to make the problem more interesting, the number of the ambidextrous is much more than the number of either left rowers or right rowers.

After the Cleaning Phase, the Left-Right Partition Phase can now ignore the issues related to handedness of rowers the left rowers, as they are on their proper respective sides. During this phase, the only eligible rowers for swapping positions are ambidextrous rowers. If there are no ambidextrous rowers to swap, then this phase will move to the next phase. Otherwise, our algorithm searches for an eligible swap that will minimize the left-right discrepancy.

The algorithm achieves this by first deciding the desirable discrepancy. Next, it performs, in the worst case, a $O(n^2)$ search to find two rowers whose swapping minimizes the left-right discrepancy. The algorithm will retrieve the lightest rower on the LS and retrieve the heaviest rower on the HS. In detail, when a new pair (P^y, P^x), where ${P^y}_w \geq {P^x}_w$, is found, we need to check whether the difference between them is better than the previous best one. Suppose that at this moment the total weight of LS is w_{ls} and the total weight of HS is w_{hs}. Then it is easy to see-https://www.overleaf.com/project/61a1152dd8a7f969b930c6c7 that $W_{lr} = w_{hs} - w_{ls}$. To check whether the difference is better, we need to calculate the current discrepancy as $(w_{hs} - {P^y}_w + {P^x}_w) - (w_{ls} + {P^y}_w - {P^x}_w) = w_{hs} - w_{ls} - 2*({P^y}_w - {P^x}_w)$, which is equal to $W_{lr} - 2*diff$. If the current difference results in a better discrepancy it is selected. Otherwise, the searching process repeats until each rower has been checked. If there is no swap that results in a perfect discrepancy then the best minimized discrepancy seen so far is selected. Each swap and current assignment is stored in the swap memory. If it is found that the swapping that results in the best discrepancy has occurred before, then the search size is incremented and the search is performed again which could result in a larger acceptable difference range. This repeats until a perfect or near-perfect left-right partition is found before proceeding to the front-back partition phase. Otherwise, the algorithm continues into the reversion phase, and restores the last best known assignment, and then continues to the next phase.

Note that each swap performed is stored in the swap memory in a situation where no acceptable solution can be found. At this moment the past must be recalled and reapplied. This occurs where the algorithm is not able to find a perfect partition, i.e. $E = 0$ or $E = 1$. The reverting phase will be discussed shortly.

The Front-Back Partition Phase follows directly after the Left-Right Partition Phase and is very similar. The strategy of selecting which two rowers to swap is the same, except the groups used are the front and back halves. The only real difference in these two phases is that in the Front-Back Partition

Phase rowers must remain on the left-side or right-side they were assigned in the previous phase. Thus, any swap involving rowers not on the same side then they can heavily affect the left-right balance rule. Therefore, only rowers positioned in the front-left and back-left can be swapped, and the same applies to the front-right and back-right. In addition, the same argument in the Left-Right Partitioning Phase applies to the discrepancy calculations in this phase, though there is still some difference. In Front-Back Partition Phase, we need the front half to be always heavier than the back half. This is different from the balancing between the left side and right side, where we do not mind which side is heavier as long as the discrepancy between them is within an acceptable range. This difference is reflected in the algorithm shown above. Note that this phase also saves each swap into memory and can move into the Reverting Phase in a situation where no acceptable assignment is found. The phase ends after reverting or if a perfect assignment is achieved. Finally, when this phase ends the algorithm moves into Weight Gradient Phase, which simply runs the Quicksort algorithm [13] on each of the four groups. In particular, the front-left and front-right rowers are sorted from the lightest to the heaviest, while the back-left and back-right are sorted in from the heaviest to the lightest.

The reverting phase uses a simple arrangment of memory, where the assignments and actions of all swaps performed to reach moment are saved. When the Reverting Phase occurs it recalls each previous swap, in the reverse order, until the best assignment for the given calling phase is found.

IV. Simulation and Experiments

Given that there is limited available resource, to our knowledge, for the Dragon Boat Partition Problem, we simulate and experiment our proposed algorithm on generated datasets. The algorithm we have designed is largely inspired by the rules and preferences by Stickels [14]. There are different online sources that set up different preferences for the problem [1], [14]. For instance, in [1], it is preferred to have the same weight for the front half and back half, in contrast to the scenario with a heavier front side [14]. For consistency, we have used the rules and preferences by Stickels [14] as our original source.

A. Dataset Generation

To generate a dragon boat environment we first need to determine the number of rowers. We always assume the drummer and steer are ignored and only consider the number of left, right, and ambidextrous rowers. Two rowers are randomly generated at the same time, as a way to keep the left-right weight steady. To ensure to have the required numbers of left rowers and right rowers we generate left and right rowers first, and then generate either-handed rowers to fill in the rest of the positions available. In addition, any rowers appended to the left and right groups will be added to the front and back groups with respect to their position in the boat.

After the rowers' data of the entire boat have been generated, we try to balance out the left-right weight, by repeatedly adding small to moderate amount of weight to a randomly selected rower, on the side that is causing imbalance, until the left-right discrepancy is equal to the optimal weight discrepancy. By this point, we have completed two of the three rules required of a dragon boat environment, namely, that only left rowers can be on the left side and right rowers on the right-side, and the left-right discrepancy is equal to the optimal weight.

Next, we move into balancing the weights on the front and back halves. Since the front-half and back-half groups are coupled to the rowers on the left side as well as on the right side, changes made to the left-side and right-side groups will heavily affect the front half and the back half groups and vise versa. On the other hand, any changes to balance the front-back weight will also affect the left-right weight balances as well. We proceed with a similar fashion with the left-right balance, where we add small to moderate amount of weight to a randomly selected rower on the side that is causing the imbalance, until the front-back balance reaches the desired goal. This process has an extra stipulation that any changes made to the front-back weight must be equally distributed in the left-side and right-side weights.

In our experiments we skip the Weight Gradient Partition phase, as it does not affect the overall balancing process of assigning rowers and will not add any other complexity to our implementation. Now, our experiment involves randomly generating 10,000 perfect dragon boat environments (i.e., 10,000 datasets) according to the aforementioned approach and then shuffling the rowers in each environment. This is followed by finding an approximation, using our proposed algorithm as discussed in Section III, of the optimal solutions for each generated environment per parameter combination.

We have a total of three (3) difference versions of our testing environment - the left-right test where only the cleaning and the left-right phase are considered, the front-back test involving only the cleaning and the front-back phase, and lastly, the full test where the three phases are involved. Our approach has three (3) important variables - left-right relaxation V_{lr}, front-back relaxation V_{fb}, and the maximum search size sz_m. Each of them significantly affects, as shown shortly, the ability of our algorithm to find an approximation of rower assignment. It is easy to see that increasing the relaxation values would result in non-perfect discrepancies to be accepted. Moreover, in the full test V_{lr} and V_{fb} will be increased at the same time. It is noted that in our experiments, *accuracy* of our proposed algorithm is simply defined to be the percentage of the resultant assignments for a randomly generated dragon boat environment for which the corresponding discrepancy is equal to the goal weight or falls within the specified relaxations.

B. Balance between Left and Right

In Table II on column $V_{lr} = 0$ we can see that decreasing sz_m results in a significant impact on finding perfect discrepancies and in turn affects the accuracy. However, increasing sz_m eventually hits an upper limit and lacks improvement, as we observe in rows for $sz_m = 150$ and $sz_m = 200$,

respectively. This is because sz_m will eventually be equal or greater than the largest weight difference among the rowers, which allows no more margin of error between the lightest rower and heaviest rower. Thus a maximum search size greater than or equal to the largest weight difference cannot push our algorithm for further improvements. We also observe that by increasing V_{lr} and V_{fb} we approximate more often within the optimal weight limit with relaxation. Continuing to increase the relaxation values eventually leads to almost any action, resulting in a "close enough" approximation, which could require no action at all. Thus, we only consider small relaxation values of 0, 1, 2, and 5.

TABLE II.
PERCENTAGE OF PERFECT DISCREPANCY APPROXIMATE W^*_{lr}.

	$V_{lr}=0$	$V_{lr}=1$	$V_{lr}=2$	$V_{lr}=5$
$sz_m=0$	52.03%	84.87%	83.99%	97.11%
$sz_m=10$	77.38%	94.09%	94.57%	99.02%
$sz_m=50$	95.77%	99.34%	99.53%	99.93%
$sz_m=100$	97.02%	99.59%	99.62%	99.97%
$sz_m=150$	97.20%	99.53%	99.56%	99.94%
$sz_m=200$	97.03%	99.57%	99.65%	99.95%

We note that by increasing the search value sz we in turn will naturally increase the amount of time required to search for an approximation. This is expected. As shown in Table III we observe that even with a larger sz_m, the average elapsed time per episode is quite minimal and appears to stagnate at $sz_m = 150$. On the other hand, by increasing V_{lr}, we will naturally lower the time to find an approximate assignment, since a larger error margin is accepted.

TABLE III.
AVERAGE ELAPSED TIME IN MILLISECONDS TO APPROXIMATE W^*_{lr}.

	$V_{lr}=0$	$V_{lr}=1$	$V_{lr}=2$	$V_{lr}=5$
$sz_m=0$	0.37ms	0.22ms	0.22ms	0.16ms
$sz_m=10$	1.61ms	0.45ms	0.46ms	0.18ms
$sz_m=50$	4.41ms	0.84ms	0.77ms	0.23ms
$sz_m=100$	5.93ms	0.95ms	1.00ms	0.23ms
$sz_m=150$	6.68ms	1.15ms	1.06ms	0.27ms
$sz_m=200$	7.28ms	1.08ms	1.09ms	0.25ms

While we can see that the executing times increase steadily until $sz_m = 100$, where it plateaus, we can see it takes more time. We conjecture that the differences in time for $sz_m = 100$, $sz_m = 150$, and $sz_m = 200$ can be attributed to the fluctuations in hardware. However, the more important point, as shown in Table IV, that we need to discuss is the average steps taken to complete an assignment.

As shown in Table IV, the average number of steps required to approximate an instance of DBP is heavily affected by the maximum search size sz_m. This reflects in perfect and imperfect discrepancies. We also need to look at the best-case and worst-case scenarios. We only observe the search size that offers the best accuracy while keeping the least amount of relaxation. We have found that in the worst case over all of our instances for the left-right partitioning phase is 213 steps and the best case is 1 step.

TABLE IV.
AVERAGE STEPS (SP) TAKEN TO APPROXIMATE W^*_{lr}.

	$V_{lr}=0$	$V_{lr}=1$	$V_{lr}=2$	$V_{lr}=5$
$sz_m=0$	5.21sp	4.28sp	4.30sp	3.65sp
$sz_m=10$	8.59sp	4.93sp	4.91sp	3.79sp
$sz_m=50$	11.67sp	5.43sp	5.31sp	3.82sp
$sz_m=100$	12.04sp	5.37sp	5.43sp	3.81sp
$sz_m=150$	12.42sp	5.44sp	5.41sp	3.82sp
$sz_m=200$	12.72sp	5.35sp	5.48sp	3.83sp

C. Balance between Front and Back

Next we consider the front-back partitioning phase separately. We observe that the results in general are similar to the left and right partitioning phase, with a slight improvement, as shown in Table V. We believe that this is due to the reduced number of constraints placed upon this phase. Particularly the Handedness Rule is not required in this phase as only rowers can be swapped between the front and back on their respective left or right side.

TABLE V.
PERCENTAGE OF PERFECT DISCREPANCY APPROXIMATE W^*_{fb}.

	$V_{fb}=0$	$V_{fb}=1$	$V_{fb}=2$	$V_{fb}=5$
$sz_m=0$	60.44%	78.34%	87.94%	97.79%
$sz_m=10$	82.17%	92.45%	96.72%	99.50%
$sz_m=50$	99.02%	99.66%	99.92%	99.99%
$sz_m=100$	99.76%	99.95%	100.00%	100.00%
$sz_m=150$	99.79%	99.94%	100.00%	100.00%
$sz_m=200$	99.81%	99.98%	100.00%	100.00%

As the front-back partitioning phase is expected to have a goal weight of $W^*_{fb} = 30$, we observe a slight increase in accuracy. We are convinced that the performance improvement is due to the less number of constraints, as the number of eligible rowers is half of the number of rowers for both the left and right sides during the front-back partitioning phase.

TABLE VI.
AVERAGE ELAPSED TIME IN MILLISECONDS TO APPROXIMATE W^*_{fb}.

	$V_{fb}=0$	$V_{fb}=1$	$V_{fb}=2$	$V_{fb}=5$
$sz_m=0$	0.32ms	0.26ms	0.22ms	0.16ms
$sz_m=10$	1.33ms	0.68ms	0.41ms	0.18ms
$sz_m=50$	3.17ms	1.28ms	0.58ms	0.20ms
$sz_m=100$	3.58ms	1.28ms	0.59ms	0.20ms
$sz_m=150$	3.46ms	1.38ms	0.57ms	0.20ms
$sz_m=200$	3.82ms	1.23ms	0.56ms	0.21ms

With a larger pool of eligible swappable rowers the performance has significantly improved. Additionally, as shown in Table VI and Table VII, there is a large decrease in average steps per instance. Lastly, we have found in the worst-case instance the algorithm would take 149 steps, whilst in best case instance only one (1) step would be required. As both phases are shown to perform well, we suspect that there will be a significant drop on perfect partition accuracy, and a significant increase in time and steps to the same rate as the left-right partitioning phase when both phases are in tandem.

TABLE VII.
AVERAGE STEPS (SP) TAKEN TO APPROXIMATE W^*_{fb}.

	$V_{fb}=0$	$V_{fb}=1$	$V_{fb}=2$	$V_{fb}=5$
$sz_m=0$	2.75sp	2.28sp	1.93sp	1.47sp
$sz_m=10$	5.42sp	3.35sp	2.42sp	1.54sp
$sz_m=50$	7.38sp	3.99sp	2.63sp	1.57sp
$sz_m=100$	7.60sp	3.89sp	2.64sp	1.58sp
$sz_m=150$	7.44sp	3.99sp	2.60sp	1.58sp
$sz_m=200$	7.74sp	3.90sp	2.62sp	1.57sp

TABLE X.
AVERAGE STEPS (SP) TAKEN TO APPROXIMATE W^*_{lr} AND W^*_{fb} TOGETHER.

	$V_{lr,fb}=0$	$V_{lr,fb}=1$	$V_{lr,fb}=2$	$V_{lr,fb}=5$
$z_m=0$	7.72sp	6.68sp	6.05sp	5.17sp
$z_m=10$	12.75sp	8.94sp	7.00sp	5.32sp
$z_m=50$	16.71sp	10.28sp	7.64sp	5.41sp
$z_m=100$	17.66sp	10.39sp	7.49sp	5.42sp
$z_m=150$	17.69sp	10.63sp	7.68sp	5.42sp
$z_m=200$	17.35sp	10.58sp	7.57sp	5.46sp

D. Combining Left-Right and Front-Back Balance

Let us see if our findings match our intuition when we combine the left-right and front-back partitioning together, which we refer to as the full test. We first examine the data involving the accuracy for the full boat in Table VIII. As is shown, the full-test's accuracy is heavily affected by the ability to find a perfect partition during the left-right phase as seen in the previous discussions. This is followed by investigating the number of average steps taken. We can see that it is heavily affected by the left-right phase as well. Lastly, we observe that it is obvious to have a higher average from both phases.

TABLE VIII.
PERCENTAGE OF PERFECT DISCREPANCY APPROXIMATE $W^*_{lr,fb}$.

	$V_{lr,fb}=0$	$V_{lr,fb}=1$	$V_{lr,fb}=2$	$V_{lr,fb}=5$
$sz_m=0$	39.54%	63.32%	77.07%	94.05%
$sz_m=10$	68.67%	84.20%	92.44%	98.61%
$sz_m=50$	96.11%	98.29%	99.50%	99.90%
$sz_m=100$	97.46%	99.14%	99.71%	99.91%
$sz_m=150$	97.68%	99.10%	99.67%	99.97%
$sz_m=200$	97.67%	99.07%	99.70%	99.96%

While there is a small increase in the accuracy in the full environment experiments over the left-right one, we conjecture that this is caused by random variance, since we are randomly generating our boat environments for every instance of the DBP problem.

TABLE IX.
AVERAGE ELAPSED TIME IN MILLISECONDS TO APPROXIMATE W^*_{lr} AND W^*_{fb} TOGETHER.

	$V_{lr,fb}=0$	$V_{lr,fb}=1$	$V_{lr,fb}=2$	$V_{lr,fb}=5$
$sz_m=0$	0.64ms	0.50ms	0.42ms	0.31ms
$sz_m=10$	2.49ms	1.32ms	0.78ms	0.37ms
$sz_m=50$	6.12ms	2.61ms	1.24ms	0.43ms
$sz_m=100$	8.21ms	3.06ms	1.36ms	0.48ms
$sz_m=150$	8.85ms	3.39ms	1.43ms	0.45ms
$sz_m=200$	8.43ms	3.58ms	1.44ms	0.46ms

As shown in Table IX and Table X, the full test has a higher overall average number of steps taken, as it has the potential of both worst case scenarios of the left-right and front-back phases. It results in a total of 348 steps for the worst-case instance. Consequently, with both phases included, we believe that we have the potential for the best case instance to happen, i.e., swapping one pair of rowers can lead to improvements for both the left-right and front-back partitioning. The best case of such a situation can result in the number of steps being 1. As an example, suppose that there is a situation where $W_{lr}=20$, and $W_{fb}=10$, and the left side is heavier than the right side by 20lb while the front half is heavier than the back by 10lb. Therefore, if we find two participants, one in the back-left, and the other in the front-right, with a weight difference of $10lbs$, we can see that swapping them would result in $W_{lr}=0$ and $W_{fb}=30$, requiring only a single step to fulfill both requirements.

An important distinction among the left-right, front-back, and full-boat environments, in particular, is the front-back partitioning phase in the full-boat does not start from a randomized perfect discrepancy. Rather, the phase begins only after the left-right phase has completed. Conversely, the front-back boat environment only performs random front-back swapping actions, leaving the left-right discrepancy unaltered. Therefore, we can conclude that the particular order of performing left-right partitioning phase first has an impact on the front-back phase.

V. DISCUSSIONS AND CONCLUSION

A novel yet practical approach to the Dragon Boat Partition problem has been presented. The performance of our approach has been demonstrated, and different combinations of our parameters have been assessed. Our approach is able to approximate close to a perfect discrepancy, without relaxation. For example in our experiments, when $sz_m=150$, the accuracy for left-right partitioning is 97.20%, front-back partitioning 99.79%, and full-boat balancing 97.67% in the 10,000 experiments. Our unique approach is able to find a perfect discrepancy for all experiments in each environment with the left-right within 12.42 steps on average, the front-back within 7.44 steps on average, and the full-boat with 17.69 steps on average. Additionally, in the worst case the number of steps for each environment took 213, 149, 348 steps in the left-right, front-back, and full-boat tests respectively. While we could reduce the search size to decrease the number of steps, there is a significant drop in perfect partitions found. It is easy to see that the problem we try to attack in this work is a variation of the classical Integer Partition Problem [2]. While there is a great number of works on finding an approximate solution to the problem, the most related work is from [7], which introduces two algorithms to tackle the multi-way number partitioning, i.e., partitioning numbers into multi-groups. The first is the sequential number partitioning. Built upon an extended Karmarkar-Karp algorithm [6], [8], the work generates

each subset that could possibly be part of an optimal partition of three groups, and for each such subset, uses the extended Karmarkar-Karp algorithm to optimally partition the remaining numbers into two groups to get a three-group partition. For problem of partitioning numbers into multiple groups other than three, the proposed approach calls so-called recursive number partitioning. The work first runs the Karmarkar-Karp algorithm to get an approximate four-way partition. Then all the numbers are divided into two groups, each of which will later be partitioned into two groups. The partitioning is done in every way that could possibly lead to a four-way partition better than the best one found so far. For such a given partition, the approach tries its best to perfectly partition each of the two subsets recursively. Our approach to the problem is different from the one in [7], due to the unique characteristics of the DBP problem, where we require the equal number of rowers both between the left side and right side and between the front half and end half. In essence, we greedily select two potential members from opposite sides to offset the discrepancy between them, while maintaining the equal number. We keep doing it in each iteration while keeping the best selection in each iteration. Given the circumstances of this rower assignment problem, we have room to have some difference between the two groups as long as it falls into an acceptable range. In addition, our problem has to consider the discrepancy situation between the front half and back half of a boat.

We are planning to analyze our proposed algorithm in terms of its time and space complexity. While we conjecture the analysis of the space complexity is relatively easier (the only hard part would be the analysis of the space requirement of the swap operations), the time complexity poses some great challenges. The major obstacle is in the stage of balancing the left and right side, due to its dynamic nature. (The situation of the front and back half, though different, is quite similar.) We will also look into the analysis of the performance ratio of our proposed approach, i.e., the gap between the result from our approach to the optimal assignment of rowers. At this moment, we are considering borrowing some analytic techniques in [7]. There is another subtle situation we may consider in our future work, where a rower rows with one hand with a different efficiency than with the other. For example, a rower rows with an efficiency of 70% with their right hand and 30% with their left hand. How we involve such a consideration in our approach would be a non-trivial problem.

The data structure of SwapMemory in our approach approach is implemented as a stack. Each element therein contains the information for P^x and P^y, the assignment of each rower, etc. As can be seen easily, such an arrangement leads to data redundancy. How to maintain the same amount information dynamically, while aiming reducing redundancy is the next task on our agenda. A crucial step in our approach is to find the best pair (P^x, P^y), since it is the one that consumes the large portion of the running time. In our implementation, we maintain the available rowers from both sides as two individual lists and conduct a linear search when needed. We are considering using a heap to speed up this search speed. We plan to experiment on the DBP instances with larger number of rowers and generalize our approach to other application scenarios, such as cargo arrangements in large shipments, passenger arrangements in airplanes, etc.

REFERENCES

[1] M. Fogliani. The Best Way to Balance Your Dragon Boat, 2011, unpublished, https://dragonanalytics.com.au/the-best-way-to-balance-your-dragon-boat/.

[2] M. Garey and D. Johnson. Computers and intractability, vol. 174, freeman. San Francisco, 1979.

[3] B. Regnier. Applying Deep Convolutional Neural Networks to the Dragon Boat Partition Problem. Master's Thesis, University of Lethbridge, 2021.

[4] B. Hayes. Computing science: the easiest hard problem, American Scientist, 90(2), pp. 113-117, 2002.

[5] What Is Dragon Boat Racing? https://www.dragonboatevents.com/what-is-dragon-boat-racing.

[6] N. Karmarkar and R. M. Karp. The differencing method of set partitioning. Technical Report UCB/CSD 82/113, Computer Science Division, University of California, Berkeley, 1982.

[7] R. E. Korf. Multi-Way Number Partitioning. In Proceedings of the International Joint Conference on Artificial Intellgience, pp. 583-543, 2009.

[8] R. E. Korf. A complete anytime algorithm for number partitioning. Artificial Intelligence, 106(2), pp 181–203, 1998.

[9] R. E. Korf. From approximate to optimal solutions: A case study of number partitioning. In Proceedings of the International Joint Conference on Artificial Intelligence, pages 266–272, 1995.

[10] S. Mertens. The Easiest Hard Problem: Number partitioning, Computational Complexity and Statistical Physics, 125(2), pp. 125-139, 2006.

[11] S. Mertens. Phase transition in the Number Partitioning Problem. Physical Review Letters, 81(20), pp. 4281–4284, 1998.

[12] P. Pedregal. Introduction to optimization, vol. 46. Springer Science & Business Media, 2006.

[13] R. Sedgewick. and K. Wayne, Algorithms. Addison-wesley professional, 2011

[14] K. Stickels. How to balance a dragon boat: tips for your most successful race boat layout, 2015, unpublished, http://paddlechica.com/how-to-balance-a-dragon-boat-tips-for-your-most-successful-race-boat-layout.