CS 124 Programming Assignment 3: Spring 2023

Your name(s) (up to two): Gabriel Chiong

Collaborators: None

No. of late days used on previous psets: 0
No. of late days used after including this pset: 0

This document contains the accompanying write-up to our implementation of several heuristic algorithms to search for an approximate solution to the NP-complete problem, Number Partition. Formally, the number partition problem takes a sequence $A = (a_1, a_2, \ldots, a_n)$ of non-negative integers as input. The desired output is a sequence $S = (s_1, s_2, \ldots, s_n)$ of signs $s_i \in -1, +1$ that minimizes the *residue*:

$$u = \left| \sum_{i=1}^{n} s_i a_i \right|$$

This experiment first provides a solution to the Number Partition problem using a dynamic programming approach. Then the deterministic algorithm Karmarkar-Karp algorithm which uses a differencing method is discussed briefly, along with the different forms of solutions to the Number Partition problem and how to convert them into comparable forms. We then introduce and describe the implementation details of the randomized heuristic algorithms used in this study, as well as with the Karmarkar-Karp algorithm and its randomized variant, BubbleSearch. The randomized heuristic algorithms we investigate in this experiment include: Repeated Random, Hill Climbing, and Simulated Annealing, with further details being provided in later sections. Finally, we present our results and hypothesize about the differences, and also discuss potential optimizations that could be useful for future investigations. To obtain our results, we use fifty randomly generated input sequences, with each value chosen uniformly at random.

Overview

The complete implementation for this experiment is separated into the main program (`main.cpp`), with several sets of header and their accompanying implementation files. The Karmarkar-Karp algorithm is defined as a separate class within the `karmarkar_karp.h` and `karmarkar_karp.cpp` files. The implementation of randomized heuristic algorithms share a significant amount of common code, which is stored in a single base class `heuristic_algorithms.h` and `heuristic_algorithms.cpp`. The differences lie in how the different solution forms calculate their residue, generate random initial starting solutions, and generate random neighbor solutions. Therefore, these different functionalities are captured in their individual child classes, both inheriting from the `heuristic_algorithms.h` parent class. The `algorithm_standard.h` and `algorithm_standard.cpp` provide an implementation for the algorithms in standard form, while the `algorithm_prepartitioned.h` and `algorithm_prepartitioned.cpp` provide an implementation for the algorithms in the prepartitioned form.

In order to provide support for this experiment, several utility files are now described. Instructions for running the experiment are contained within `README.md`, including instructions for unit testing and running other extensions of this investigation. A suite of unit test that cover all algorithm implementations used in this experiment are provided in the `tests.cpp` file (although it is somewhat difficult to test the randomized heuristic algorithms exactly). The randomly generated input sequences are stored in the directory `input/`, along with a Python script for generating all fifty input files. The `output/` directory contains the results from running all our algorithms on the input files and averaging over 100 trials. Finally the `run.sh` script automates the entire experimentation process.

Dynamic Programming Approach

In this section, we present an exact solution to the Number Partition problem using the dynamic programming paradigm. Our solution is similar in spirit to the 0-1 Knapsack problem, in the sense that we are trying to fill one of the partition sets with elements that make up half of the total sum of all elements (in our case, as close as possible), which is akin to filling a knapsack with as many items as possible while trying to get as close to the knapsack capacity of value equal to half the sum of all elements. This minimizes the residue of the Number Partition problem, since the other set will contain the other half (or as close as possible) of the values. More formally, given an input sequence of integers $A = (a_1, \dots, a_n)$ of total sum $b = \sum_{i=1}^{n} a_i$, we seek to find the subset of elements which result in the closest sum to $\lfloor b/2 \rfloor$.

We define our subproblems $OPT(i, j)$ to be "true" if we can construct a subset of elements with a sum equal to j using the elements in A from indices $1, \dots, i$, and "false" otherwise. Since we do not know whether taking the current element at index i should be in the subset we are constructing, we check both possibilities - with and without element i. In the case of the latter, we still need to know if we can construct a subset that sums to j, but with one less element (i.e. $i = 1, \dots, i-1$). In the case of the former, picking element i means that we are now checking if it is possible to construct a subset that sums up to $j - A[i]$ with the remaining $i - 1$ elements. So our relation is $OPT(i, j) = OPT(i-1, j) \vee OPT(i-1, j - A[i])$.

We have two base cases for our recurrence relation. Firstly, when $j = 0$, we always return true since we can choose not to pick any element within the given indices. This includes the case of $OPT(0, 0)$ since no elements are needed to sum to 0. Secondly, when $i = 0$ and $j > 0$, we return false since there are no elements for us to choose from, and it is impossible to obtain a sum of a positive integer j. Our recurrence relation can now be written as follows:

$$OPT(i, j) = \begin{cases} True & \text{if } j = 0 \\ False & \text{if } i = 0 \wedge j > 0 \\ OPT(i-1, j) \vee OPT(i-1, j - A[i]) & \text{otherwise} \end{cases}$$

From the recurrence relation above, it is clear that our topological order of subproblems should be in order of increasing i, since each subproblem always depends on subproblems of strictly smaller i's. So we build our bottom-up dynamic programming table starting at $i = 0, \dots, n$. Once our table is built, we check if $OPT(n, \lfloor b/2 \rfloor)$ is true. Otherwise, do a linear scan of the table row $OPT(n, j)$ for all $j = 0, \dots, b$ to check for the closest entry $OPT(n, j')$ that has a true value to $\lfloor b/2 \rfloor$. We return the value of $j' - \lfloor b/2 \rfloor$ as the residue. To reconstruct the partition of the two sets, we can store parent pointers in a separate table with a pointer from $OPT(i, j)$ to whichever is true between $OPT(i-1, j)$ and $OPT(i-1, j - A[i])$. Note that some entries in the table may have two outgoing pointers in this case. Now we simply follow parent pointers back from the optimal $OPT(n, j')$ to construct one of the sets for our partition. The other half of the partition is just the elements that were not taken by this path.

Since we are constructing a bottom-up table with dimensions $(n+1) \times (b+1)$ including base cases, our algorithm has $O((n+1) \cdot (b+1)) = O(nb)$ subproblems. Each subproblem is just OR-ing the results of two table look-ups, which is a constant $O(1)$ time operation overall. Thus, our algorithm takes $O(nb) \cdot O(1) = O(nb)$ time, and $O(nb)$ space to store the look-up tables. Note that the linear scan to return the result for the original problem is $O(b)$, but is dominated by the time to construct the table.

This running time is polynomial if b can fit within a word size of the machine we run the algorithm on, so it is only polynomial in the value of b, but is exponential in b if the representation of b requires more than a single machine word to store. Thus, the dynamic programming solution has a pseudo-polynomial running time.

Karmarkar-Karp Algorithm

The Karmarkar-Karp algorithm is critical to our investigation since it is used as both a deterministic alternative to the randomized heuristic algorithms, and is also used as a subroutine to calculate the residue of the prepartitioned heuristic algorithms. Therefore, it is important that we find an efficient way to implement it. A naive first approach to implementation could be storing the elements in a sorted array, each time taking the two largest elements at the end and reinserting their difference into the sorted array. Unfortunately, this is not as efficient as this algorithm can be, since there is no easy way to insert into a sorted array that does not involve reallocating a new array. This naive approach would result in a running time of $O(n^2)$, since we are effectively creating a new array for every new difference.

However, we can exploit a max-heap data structure to more efficiently maintain the order of the elements in the sequence. Rather than store the elements of the input sequence in a sorted array, we insert each of the given input elements into a max-heap, H. By the correctness of the max-heap data structure, we are always able to take the top two elements of highest value by invoking the series of operations `find_max()`, `delete_max()`, `find_max()`, and `delete_max()` operations in this order. We then reinsert the difference back into H using the `insert(x)` operation and can begin the next iteration of taking the next two elements of highest value again. We can continue this until one element remains and return that as the final residue. The final element in H must be the residue, since it is the final resulting difference from the entire input sequence - the definition of the Karmarkar-Karp algorithm's residue.

Assuming that all the values in A are small enough that arithmetic operations take one time step, the running time of this algorithm is $O(n \log n)$. This is because initializing the heap takes $O(n)$ time based on the input sequence assuming we are able to mutate it, and each `delete_max()` and `heap_insert(x)` operation takes $O(\log n)$ time, and the `find_max()` operation takes $O(1)$ time. We repeat these operations n times since at every stage we remove two elements and insert one into H. Therefore, the running time of the Karmarkar-Karp algorithm using a max-heap data structure takes $O(n \log n)$ time.

Implementation Details

This section provides some background information on some considerations and reasoning behind decisions made during our investigation. We first discuss the two different solution forms used, then provide a brief discussion on the randomized heuristic algorithms we use to compare against the deterministic Karmarkar-Karp algorithm. Finally we list some miscellaneous ideas that help to improve our solution.

Solution Forms

In our implementation, we use two different solution forms to represent the partition of two sets. The first is the standard representation where the solution S is simply a sequence of +1 and -1 values, the same length as the input sequence. If the entry $S[i] = +1$ for some index $i = 1, \ldots, n$, then the input element $A[i]$ is in the first set. Otherwise, if $S[i] = -1$, then we say that the element $A[i]$ is in the second set. The sum of products $\sum_i^n S[i] \cdot A[i]$ forms the residue of this solution. A random solution is simply assigning each index $i = [1, n]$ a value in $\{-1, +1\}$, and a neighbor of S is simply any solution sequence that differs by at least one and at most two sign flips of any two randomly chosen elements in S.

The second way to represent a solution is called *prepartitioning*. Here, the solution is a sequence $P = \{p_1, p_2, \ldots, p_n\}$, where $p_i \in \{1, \ldots, n\}$. We interpret this solution in the following way: if $P[i] = P[j]$ for some indices $i, j = 1, \ldots, n$, then $A[i]$ and $A[j]$ must both lie in the same subset. We use the same algorithm in the problem set description to calculate the residue from a solution of the prepartitioning form, which involves the use of the Karmarkar-Karp algorithm. This enables us to compare residue values against algorithms which use the standard form.

Randomized Heuristic Algorithms

In addition to investigating the results from the Karmarkar-Karp algorithm, we also study the behavior of three randomized heuristic algorithms - Repeated random, hill climbing, and simulated annealing. We provide implementations for all three randomized heuristic algorithms in both the standard and prepartitioning forms. We do not include the pseudocode for these algorithms in the write-up as they are relatively simple, and full details of the implementation can be reviewed within our code implementation provided. One thing to note is that the cooling schedule we have used in our investigation is the one suggested by the problem set write-up, namely $T(\text{iter}) = 10^{10}(0.8)^{\lfloor \text{iter}/300 \rfloor}$, where our value of "iter" is 25000.

As an extension to this investigation, we have also implemented a variant of the BubbleSearch heuristic algorithm, where our implementation is simply a variant of Karmarkar-Karp based on a max-heap. Our implementation greedily takes the top two elements at every iteration (with some probability). We chose the probability of passing on the top element of the heap to be 0.2 and switching to Karmarkar-Karp when there are less than ten elements left. Once the two elements are chosen, we re-insert all discarded elements from this round back into the max-heap, along with the difference of the two chosen elements, ready for the next iteration until there is only one element left. We have also included the results in the sections below and compare it with the other seven algorithms used in this investigation. Since the probability of passing on too many elements and reaching an empty heap is relatively small (5^{-10}), we have decided to throw an error in the unlikely case that this happens. An alternative solution could be to re-insert all the discarded elements back into the heap and start that iteration again.

Other Miscellaneous Details

Although this write-up does not go into detail about pseudorandom number generators, we chose to use the Mersenne Twister pseudorandom number generator over the more well-known `std::rand()` generator. Based on some experimentation outside the scope of this investigation, we found that `std::rand()` produced a larger skew towards lower numbers compared to the Mersenne Twister implementation. Furthermore, reading the documentation for `std::rand()` also revealed its other disadvantages - low randomness of lower-order bits, a short period, and the low value of `RAND_MAX` (the range of values at which it generates random numbers). We have therefore chosen to use the Mersenne Twister generator for our implementation.

In terms of code style, there were significant similarities between an implementations of the three randomized heuristic algorithms using the two different solution forms, so we have decided to draw on some object-oriented programming principles to create a parent base class called `HeuristicAlgorithms`, and have two child classes `AlgorithmStandard` and `AlgorithmPrepartitioned` inherit all common methods and member variables from the parent class. Doing this enabled us to only implement the algorithms in the parent class, while the child classes handled solution form specific implementations, such as generating random solutions, generating a random neighbor, and calculating the residue of a solution.

We use C++ `long`s rather than `int`s to represent our integers, since each input value is potentially up to a value of 10^{12}, and doing arithmetic operations may result in undefined behavior from signed integer overflow/underflow. To generate our fifty random instances of inputs, we use a Python script to randomly fill fifty files with 100 integers with value between $[1, 10^{12}]$. Our experiment results are then obtained from running each of the eight algorithms (Karmarkar-Karp, Repeated Random - Standard, Hill Climbing - Standard, Simualted Annealing - Standard, Repeated Random - Prepartitioning, Hill Climbing - Prepartitioning, Simualted Annealing - Prepartitioning, and BubbleSearch) on each of the fifty generated input files, with each input's result averaged over 100 trials. We then report both the mean residue and trial time taken for each input file. Each heuristic algorithm is allowed 25000 iterations to find a solution.

Results and Analysis

In this section, we present the results from our experiment and analyze the behavior of each of the algorithms used. We begin with a high-level overview of the results before diving into a deeper discussion on specific metrics. Our detailed discussion sections compare the residual results from each of the algorithms, and then moves on to comparing the running times of each algorithm. Finally we conclude this section by hypothesizing a potential improvement for finding more optimal residue values.

Overall Results and Discussion

We present both Table 1 and Figure 1 to display our overall results. The results shown in this section are averaged over all fifty of our randomly generated inputs. Note that the values of "residue" and "time" for each of the algorithms run on the fifty inputs are already averaged over 100 trials for each input. Note that in Figure 1, both axes are presented in logarithmic scale, due to the wildly different magnitudes of both the residue values and running times of the algorithms compared. The true magnitude of difference can be appreciated with Table 1, which provides the specific values for each metric.

Algorithm	Average Time (ms)	Average Residue
Karmarkar-Karp	0.002273	227491.3
Repeated Random, Standard	15.254717	290363783.0
Hill Climbing, Standard	2.802538	350396193.4
Simulated Annealing, Standard	3.148336	419939441.1
Repeated Random, Prepartitioning	109.643242	218.1
Hill Climbing, Prepartitioning	80.732793	739.18
Simulated Annealing, Prepartitioning	85.133845	301.5
BubbleSearch	0.006913	1625812.2

Table 1: Comparing the overall averaged residue and time results from all fifty inputs.

From Table 1 and Figure 1, we observe that there are three main groups of results. The three randomized heuristic algorithms that use a prepartitioning form solution clearly were able to obtain the lowest residue values, somewhere between 100 and 1000. However, these three algorithms also have the highest running times of approximately 100ms. The deterministic Karmarkar-Karp algorithm has the next lowest average residue value outside of those that use the prepartitioning form of solutions, with a value of approximately $200,000$. In terms of running time, the Karmarkar-Karp algorithm is clearly the most efficient, with an average time of only 0.002ms. The randomized variant of the Karmarkar-Karp algorithm, BubbleSearch appears to do slightly worse at both residue values and running time, although it is much closer to the Karmarkar-Karp results than with any other randomized heuristic algorithm. Finally, the three randomized heuristic algorithms that use the standard form solution appear to do the worse in terms of finding an optimal residue value, with values in the ballpark of 1×10^8. Their running times fall in between the differencing method algorithms and the randomized heuristic algorithms in prepartitioning form.

In terms of comparing between the three standard form algorithms, we observe from Table 1 that the Repeated Random algorithm seems to perform the best out of the three in terms of optimal residue, followed by Hill Climbing, and then Simulated Annealing. However, Repeated Random also appears to have the worse running time out of the three. For the three prepartitioning algorithms, Repeated Random again appears to perform the best, followed by Simulated Annealing, and then Hill Climbing. The running times between these three follow the same pattern as the one observed for the standard form solution. Although this subsection is limited to observations and patterns from our results, we provide a deeper

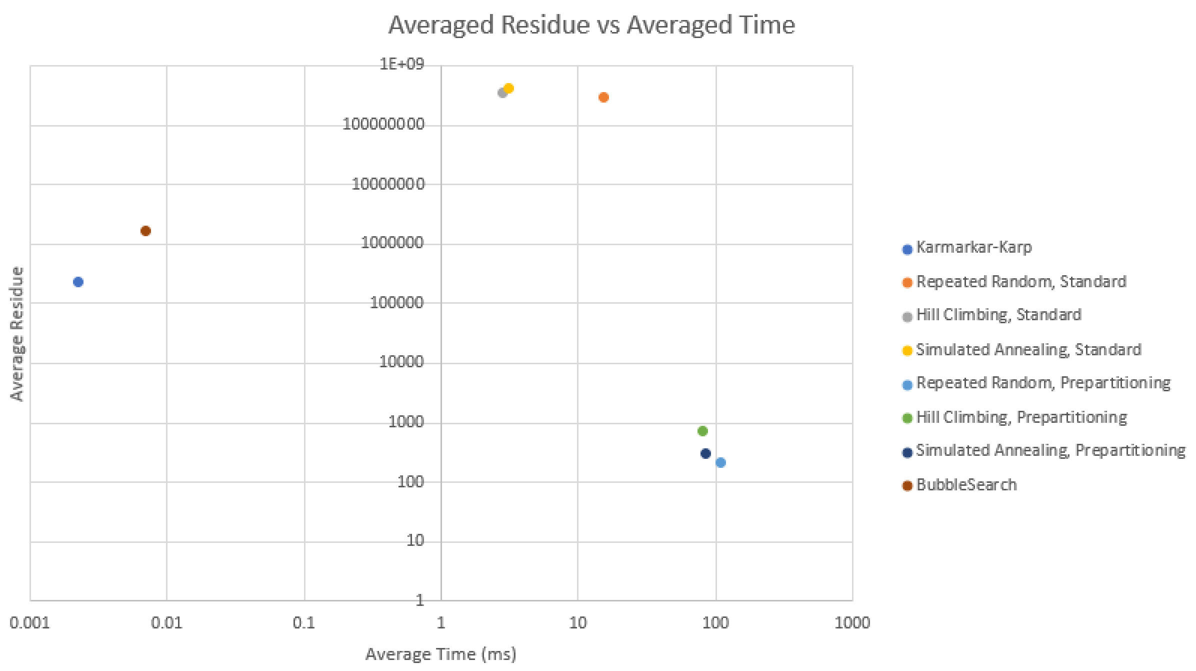

Figure 1: Graphical representation of the overall residue and running times from all fifty inputs.

discussion and suggest several reasons for the behaviors described in the following subsection.

Residue Results and Discussion

Since we run all eight algorithms on each of the fifty input files, we do not provide specific data in tabular format in this section. Rather, we provide a graph to visualize the results obtained. Figure 2 below displays the averaged residue values obtained by each algorithm on every input file. The specific values can be obtained from the `output/` directory of our submission for this problem set.

Again, a logarithmic scale is used for the y-axis, since the values have a significant difference in order of magnitude. Therefore, although the results may appear close visually, white space on the graph represents a very large difference. Again, we generally observe three groups distinct groups, namely the three algorithms in standard form with the highest residues, the two differencing algorithms below them, and finally the three algorithms in prepartitioning form with the lowest residues. It is now slightly more difficult to make intra-group comparisons as it appears to vary with no clear algorithm outperforming the others within the same group. However, on closer inspection of the graph, we can generally observe that Repeated Random has a greater number of lower values for input files compared to the other two algorithms of the same form. Similarly, the Karmarkar-Karp algorithm seems to outperform the BubbleSearch algorithm in our experiments with values roughly around 10^5 versus 10^6 for BubbleSearch.

We hypothesize that a potential reason for Repeated Random outperforming the other two randomized heuristic algorithms (within the same form of solution) could be because the solution space has many local optima. Therefore, a neighbor-based local search like Hill Climbing and Simulated Annealing could have a high probability of being stuck in a local optimum, unlike Repeated Random which is able to "jump" from one solution to another and only keep the best solution seen so far. We did expect that Simulated Annealing would perform better than Hill Climbing where there were many local optima, since there is a non-zero probability of moving to a worse solution to prevent being stuck in a local optimum. This seems

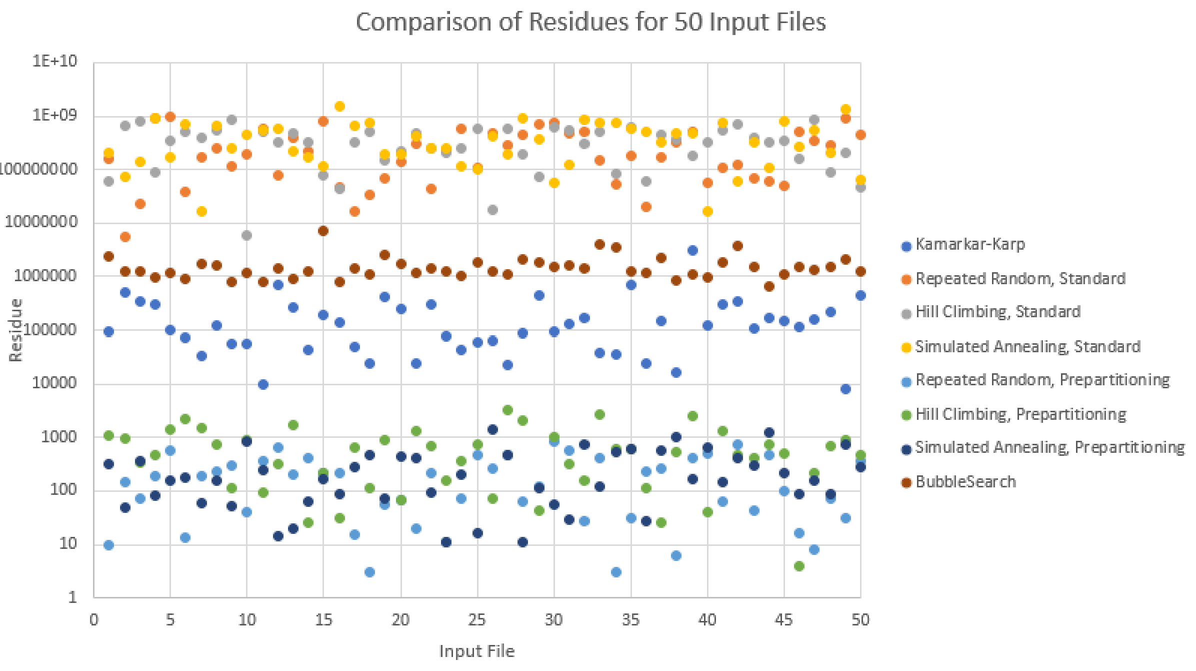

Figure 2: Graphical representation of the residues from all fifty inputs.

to be the case for prepartitioning solutions, with Simulated Annealing having an average residue of 301.5 (from Table 1), which is closer to its corresponding Repeated Random, and less than half of the 739.18 value from Hill Climbing. However the same trend is not observed for the standard solution forms, where Hill Climbing outperforms Simulated Annealing and is closer to Repeated Random. This could potentially be attributed to the number of iterations being insufficient for the ability of Simulated Annealing to escape local optima to take effect, and our expected behavior might be observed if we continued running for more iterations.

Although we had initially hypothesized that the prepartitioning algorithms might outperform the others due to a more “relaxed” method of forming a partition allowing for the algorithm a broader scope to optimize, we were surprised by how much better the prepartitioning algorithms performed relative to the others. To clarify a potential ambiguity, by “relaxed” we mean that the prepartitioning form only enforces restrictions between variables and does not explicitly assign each variable to a specific partition like the standard form. Furthermore, we expected the prepartitioning algorithms to also outperform Karmarkar-Karp and BubbleSearch since prepartitioning allows for some improvement to the partitions before applying Karmarkar-Karp to calculate the residue. This is effectively starting Karmarkar-Karp from an improved initial position. But as before, we were also surprised by the magnitude of difference between the prepartitioning algorithms and the differencing method algorithms.

Timing Results and Discussion

Since we run all eight algorithms on each of the fifty input files, we also do not provide specific data in tabular format in this section. Rather, we provide a graph to visualize the results obtained. Figure 3 below displays the averaged times obtained by each algorithm on every input file. The specific values can be obtained from the `output/` directory of our submission for this problem set. Again, we use a logarithmic scale for the y-axis to minimize the white space on the graph.

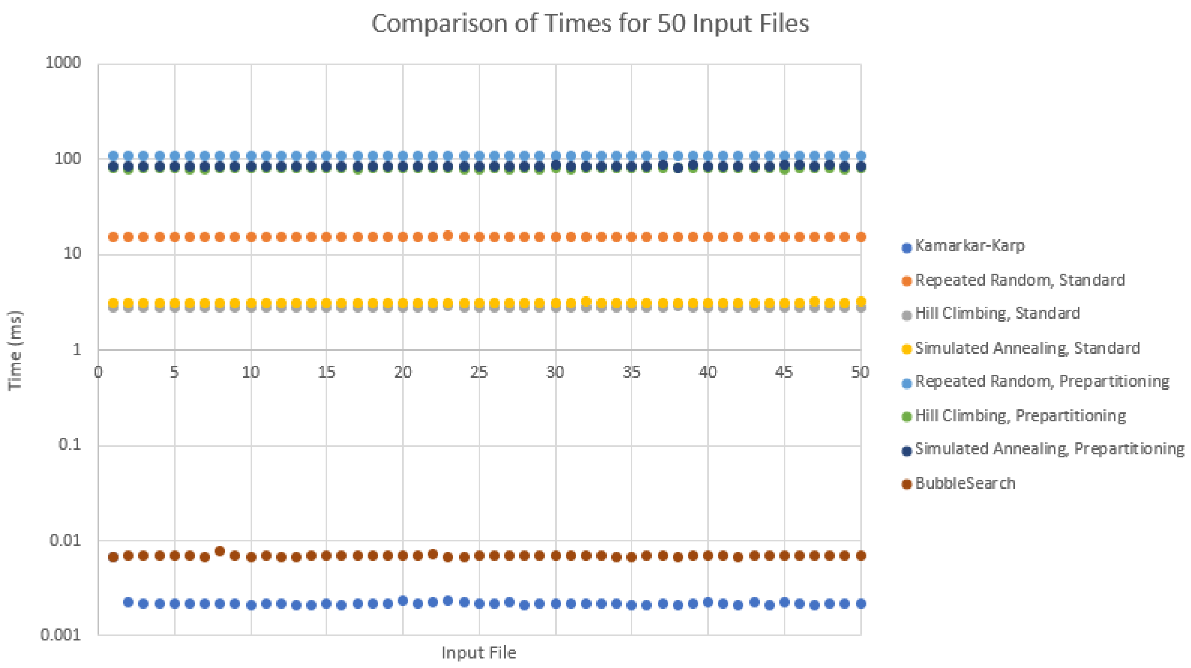

Figure 3: Graphical representation of the times from all fifty inputs.

We observe that the two differencing method algorithms Karmarkar-Karp and BubbleSearch have the lowest running times, between 0.001 and 0.01ms across all input files. This is followed by the standard form solution algorithms with times of at most 20ms. Finally, the prepartitioning algorithms are the slowest, with times around the 100ms mark. We believe that our reported results agree with a theoretical run time analysis, as shown below.

Firstly, we know from earlier sections in this write-up that the asymptotic running time of the Karmarkar-Karp algorithm is $O(n \log n)$ using a max-heap, where n is the number of input elements to our algorithms. In our implementation, we use the C++ standard library's `std::priority_queue`, which we believe would provide good optimizations in terms of practical run times that we would not be able to achieve by implementing our own version. The BubbleSearch algorithm would also be $O(n \log n)$ since it is also based on a max-heap, with an expected constant number of passed on elements per iteration (since the probability of continued passing of elements becomes exponentially small as the number of passed on elements increase).

Next, we consider the time complexity of the standard algorithms. It takes $O(n)$ time to construct the initial random solution, and we run the algorithm for $iter$ number of iterations. For each iteration, we consider all operations other than a random move as constant operations (just comparisons, arithmetic, and assignments). In the case of Hill Climbing and Simulated Annealing, the neighboring solutions only make a constant number of calls to the pseudorandom number generator (at most two values change). However, for Repeated Random, the entire solution sequence could potentially change, which involves n calls to the pseudorandom number generator. Therefore, since our implementation creates a new solution for each of the three algorithms' random move operation, the time complexity of Hill Climbing and Simulated Annealing is $O(iter \cdot n)$, and the time complexity of Repeated Random is $O(iter \cdot 2n) = O(iter \cdot n)$. The constant 2 (from constructing a size n solution with n calls to the pseudorandom number generator) is hidden by the big-oh notation for Repeated Random, but is observed in real time as we run the algorithm as can be seen in Figure 3. The effect of having to invoke the pseudorandom number generator more times could also have a multiplier effect in prolonging the measured running time.

The time complexity analysis of the prepartitioning algorithms are almost similar to that of the standard algorithms, except that within each iteration $iter$ being dominated by the $O(n)$ running time of constructing a solution as in the standard algorithms' case, it is now $O(n \log n)$ from applying the Karmarkar-Karp algorithm. Therefore, the time complexity of all three prepartitioning algorithms becomes $O(iter \cdot n \log n)$. As expected, the extra n calls to the pseudorandom number generator in Repeated Random is dwarfed by the application of the Karmarkar-Karp algorithm, and Figure 3 shows that the three algorithms are much closer to each other relatively, compared to the case of the standard solution form algorithms.

Potential Optimizations

We now discuss a potential optimization to the randomized heuristic algorithms that could improve their performance (the residue value they produce). Since the Karmarkar-Karp algorithm already outperforms the algorithms in standard form, using the solution sequence provided by Karmarkar-Karp algorithm at their start would clearly improve their results (provides an improved upper bound to the residue returned by the three algorithms in standard form). The more interesting question is whether or not it would improve the performance of the prepartitioning algorithms, which already have the best performance in our experiment. We are effectively trying to "warm-start" our heuristic algorithms using the solution provided by the Karmarkar-Karp algorithm. Below, we consider the effects of using this strategy to warm-start the prepartitioning algorithms.

From our observed results, we believe that using Karmarkar-Karp as a starting point would probably not improve the Repeated Random algorithm. This is because the prepartitioning Repeated Random algorithm already outperforms the Karmarkar-Karp algorithm from our results, and jumps to a completely unrelated solution at every iteration, which negates the effect of having a good starting solution. However, if we only had a limited number of iterations for our algorithm to run, using Karmarkar-Karp may improve the results, since it guarantees that the final solution will be at least as good as the Karmarkar-Karp algorithm.

For the heuristic algorithms that rely on moving to neighbors with only differences in one or two values, it is also not clear if using the Karmarkar-Karp algorithm as a warm-start would improve the final residue returned. This is because they are already susceptible to being trapped in local optima, and if the Karmarkar-Karp leads them to a local optima to start with, it could mean that they would continue to be trapped in that local optima. This risk may not be as high for Simulated Annealing given enough iterations, since it has a probability of escaping local optima. Again, if we are limited in the number of iterations we are allowed to run our heuristic algorithms, it may make sense to use Karmarkar-Karp as a warm-start. But if we are free to run our algorithms for a large number of iterations, starting at a random solution might have better results due to being able to avoid local optima to start with.

Although we do not implement this warm-start method in our implementation, here we propose a method to do so for both the standard and prepartitioning solution forms. For the standard form, if we encounter two original (un-differenced) elements during the course of our iterative differencing, we place the larger element in a different set to the smaller element. When differencing an element, we use additional storage to track the largest and smallest elements that were used to produce this difference. Then when we encounter this differenced element in the course of our iterative differencing, we simply check if it is the first or second element of the two taken from the max-heap at the current stage. If it is the first element, we restrict the second element from being in the same set as the set of the larger element that produced its difference (i.e. the second element will be in the same set as the smaller element that produced the difference in the first element). Otherwise, we restrict the second element from being in the same set as the set of the smaller element that produced its difference. For the prepartitioning solution form, our approach is similar to that of the standard form, except that we treat the elements that form a set together initially as a super-element, with its value being the sum of the elements contained within it. Now we proceed in the same way as we would for the standard partitioning solution form.

In terms of time complexity, using Karmarkar-Karp as a warm-start for any of our randomized heuristic algorithms would not make a significant difference, assuming that the number of iterations is much larger than the n, the length of our input sequence, as is the case for our experiment.

Conclusion

In summary, we have found that the preparitioning form enables the heuristic algorithms to perform much better (finds relatively more optimal partitions) than algorithms based on the differencing method (Karmarkar-Karp and BubbleSearch), and also heuristic algorithms based on the standard solution form. However, our experimentation results have also revealed that this improved performance comes at a cost, namely the time complexity of the prepartition algorithms is higher both in theory and in practice, compared to the other algorithms considered in our experiment. Another trade-off that our experiments have allowed us to analyze is of that between a deterministic solution and a heuristic. We found that a dynamic programming approach can help to find the global optimum partition of the input sequence, at the cost of a pseudo-polynomial running time. If we relax our problem requirements to only find an approximate optimal solution, we have found that we can reduce the running time complexity to strongly-polynomial in n. What is important to note is that all these algorithms improve on the $O(2^n)$ time complexity of brute-force searching the entire solution space.

Another interesting observation was that the representation of the solution form had a significant impact on the quality of results obtained from the same algorithm. The standard form solution was poor enough that even simple deterministic algorithms like the Karmarkar-Karp and BubbleSearch were able to provide better running times and residues. This again is a trade-off. The simple standard solution form is easy to understand for users, compared to the more complicated prepartitioning form. But the results were also much worse. Finally, we hypothesized about using the Karmarkar-Karp algorithm to warm-start the heuristic algorithms and proposed some situations where it would be an improvement, and other situations where it would not offer any real advantage compared to starting at a random solution.

To conclude, we describe several interesting areas for furthering this investigation, as we were not able to fully explore every area due to the time constraints of this problem set. Firstly, we would be keen to also implement a dynamic programming solution for comparison with the heuristic algorithms. We believe it would be very competitive in terms of running time for smaller values of inputs, and what the crossover point is before it becomes prohibitively slow to use would be an interesting study. Next, we wonder if prepartitioning is the best solution form to use for all inputs or did it only perform as well as it did here because of the parameters of our experiment? A natural follow-on from this is whether there are better solution forms out there, and if so, what scenarios do they perform better under? Lastly, we would be interested to know if there are any other heuristic algorithms like parallel search or Tabu search that would offer any improvements (and if they can beat Repeated Random), or if Repeated Random is the true king for heuristic algorithms for the Number Partition problem.