TAKE ADVANTAGE OF THE COMPUTING POWER OF DNA COMPUTERS

Z. FRANK QIU and MI LU

Department of Electrical Engineering
Texas A&M University
College Station, Texas 77843-3128, U.S.A.
{zhiquan, mlu}@ee.tamu.edu

Abstract. Ever since Adleman [1] solved the Hamilton Path problem using a combinatorial molecular method, many other hard computational problems have been investigated with the proposed DNA computer [3] [25] [9] [12] [19] [22] [24] [27] [29] [30]. However, these computation methods all work toward one destination through a couple of steps based on the initial conditions. If there is a single change on these given conditions, all the procedures need to be gone through no matter how complicate these procedures are and how simple the change is. The new method we are proposing here in the paper will take care of this problem. Only a few extra steps are necessary to take when the initial condition has been changed. This will provide a lot of savings in terms of time and cost.

1 Introduction

Since Adleman [1] and Lipton [25] presented the ideas of solving difficult combinatorial search problems using DNA molecules, there has been a flood of ideas on how DNA can be used for computations [4] [3] [25] [9] [12] [16] [18] [19] [20] [22] [24] [27] [28] [29] [30]. As one liter of water can hold 10^{22} bases of DNA, these methods all take advantage of the massive parallelism available in DNA computers. This also raises the hope to solve the problems intractable for electronic computers. However, the brute force approaches usually take long time to complete due to the long process of the problem and for those existing method, they all need to restart the whole process when the initial condition changes.

In this paper, we propose a new class of algorithms to be implemented on a DNA computer. The algorithms we are going to introduce will not be affected much by the initial condition change. This will give DNA computers great flexibility. Knapsack problems are classical problems solvable by this method. It is unrealistic to solve these problems using conventional electronic computers when the size of them get large due to the NP-complete property of these problems. DNA computers using our method can solve substantially large size problems because of their massive parallelism.

Throughout the paper, we make one assumption that all molecular biological procedures are error free. This is not true for the real world, but there is a large amount of finished research work attacking this error-resisting problem [5] [10] [11] [14] [35] [36]. These research work also showed many fault tolerant techniques. We hope that errors which arise during our DNA computer operations can be dealt with by the given techniques.

The rest of the paper are organized as follows: the next section will explain the methodology. The description of how NP-complete problems are solved will be presented in section 3. Section 4 will explain how we can avoid going through all the procedures when a little change is made to the initial condition. The last section will conclude this paper and point out future work.

2 DNA Computation Model

In this section, we include the fundamental model for basic operations to be used in our DNA computation. All DNA operation models necessary for solving those NP-complete problems in the next section are introduced and explained as follows.

2.1 Operations

1. **reset(S)**: This may also be called initialization. It will generate all the strands for the following operations. These strands can be generated either to represent the same value or to represent different values according to the requirement.
2. **addin(x, S)**: This step adds a value x to all the numbers inside the set **S**.
3. **sub(x, S)**: This operation will subtract a value x from all the numbers inside the set **S**.
4. **divide(S, S1, S2, C)**: This step will separate the set **S** into two different sets based on the criteria C. If no criterion is given, then components in these two sets are randomly picked from set **S** and **S** will be evenly distributed into two sets **S1** and **S2**.
5. **Union(S1, S2, S)**: The operation combines, in parallel, all the components of sets **S1** and **S2** into set **S**.
6. **copy(S, S1)**: This will produce a copy of **S**: **S1**.
7. **select(S, C)**: This operation will select an element of **S** following criteria **C**. If no C is given, then an element is selected randomly.

2.2 Biological Implementation

In this section, we include the fundamental biological operations for our DNA computation model.

1. **reset(S)**: This initialization operation can be accomplished using mature biological DNA operations [1] [4] [6] [13] [15] [30]. It will generate a tube of DNA strands representing the same number, e.g., these strands will consist of exactly the same nucleotides with same order. It may also generate different strands to represent different numbers according to the requirement.

2. **addin(x, S)**: There are some existing arithmetic operations for DNA computers that have been developed [30] [19]. We are going to use the method introduced by [30]. This operation will add a number to all the strands in the tube using the method first introduced by [8]. After the addition is finished, all the new strands inside the tube will represent the sum of the value represented by the original strand and the number we add in no matter what was in the tube before the operation. Readers may refer to [30] if any detailed information about how to perform this addition is needed.
3. **sub(x, S)**: The operation can be accomplished similarly as the **addin** operation shown above. It will simply subtract the value x from all the strands inside the tube.
4. **divide(S, S1, S2, C)**: The necessary operation for this step of DNA computing is to separate one tube of strands into two tubes. Each resultant tube will have approximately half of the strands of the original tube. The criteria C can be containing or not containing a certain segment, e.g. ATTCG, and we may use the metal bead method to extract them [1] [23].
5. **union(S1, S2, S)**: This operation will simply pour two tubes of strands into one.
6. **copy(S, S1)**: We need to make copies of DNA strands of the original tube and double the number of strands we have for this copy operation. The best and easiest method for this will be PCR (Polymerase Chain Reaction). Because PCR is counted as non-stable by [31] [34] [33] [32], we will try to use this operation as few times as possible.
7. **select(S, C)**: This procedure will actually extract out the strand we are looking for. So, it will extract strands from tube **S** following certain criteria C. We may use existing methods introduced in [2] [21] [25].

3 NP-complete Problem Solving

3.1 One Simplified NP-complete Problem

We will show how to solve a simplified knapsack problem [7], one of the NP-complete problems which is unsolvable by currently electronic computers.

Problem: Given an integer K and n items of different sizes such that the ith item has an integer size k_i, find a subset of the items whose sizes sum to exactly K, or determine that no such subset exists [26] [17].

In solving the knapsack problem using DNA computers, we intend to use the methodology presented in the previous section. It can be accomplished as follows:

a-1 **reset(S)**: We will generate a large amount of strands in a tube **S** and all strands in the tube will represent 0, i.e. they are exactly the same. This will assume that all the potential "bags" are empty because each strand is counted as a bag to hold items.

a-2 **divide(S, S1, S2)**: This will give us two almost identical sets that are close to the exact copies of **S** when we have a large amount of strands in the original set **S**.

a-3 **addin(x_i, S1)**: We will add integer x_i which represents the size of the ith item to set **S1**.

a-4 **union(S1, S2, S)**: Now we have a mixed set with about half of the potential "bags" containing the ith item while others do not.

a-5 Repeat the above steps 2, 3 and 4 until all items have been added in.

a-6 **select(S, C)**: The criteria **C** we are using here is whether number **K** exists or not. If we find an integer **K** in the tube, that means we have a subset of the items with sizes sum exactly equals to **K**. If there is no such kind of strand in the tube, then the answer will be no.

The number of steps the algorithm requires is 3n+2, where n is the total number of items. Unlike another possible solution for this kind of knapsack problem introduced by Baum [7], we do not have any restriction like balanced size of items required by Baum. That means, we can solve all simplified version of knapsack problems within polynomial time as long as the total size of these n items can be represented by the method [30] we use.

3.2 An Advanced Problem

In the previous section, we gave the DNA algorithm on one of the NP-complete problems: simplified knapsack problem. Here, we are going to advance the method and try to solve the complete version of the knapsack problem, a more computation intense problem.

Problem: The input is a set X such that each element $x \in X$ has an associated size k(x) and value v(x). The problem is to determine whether there is a subset $B \in X$ whose total size is $\leq K$ and whose total value is $\geq V$.

Algorithm: The advanced algorithm based on the one for the simplified knapsack problem is shown as follows:

b-1 **reset(S)**: We will generate a lot of "empty" strands in the set. Here each strands in the tube will be treated as two parts while both are zeros initially. The first part X will be used to represent the size of the items inside the tube and the second part U is for the value of these items. At the very beginning, because the bags are empty, so both the size and the value are zeros. Each strand will be like $5' - X - U - 3'$. We make X and V large enough to hold the total item sizes and total item values so no matter how we operate on them, X and U will not intervene with each other.

b-2 **divide(S, S1, S2)**: This will give us two almost identical sets that are close to the exact copies of S when we have a large amount of strands in the original set **S**.

b-3 **addin(x_i, S1)**: We will operate on the first set S1. The integer that represents the size of the item:k_i and the integer that represents the value of the item:v_i will be added into different part of all strands in the set. x_i is added to X and v_i is added to U. Set S2 will be untouched. The technique that adding different numbers to their expected locations are based on the locators L_i and R_i shown in [30] b-4 **union(S1, S2, S)**: Now we have a mixed set with about half of the potential "bags" containing the *i*th item while others do not.

b-5 Repeat the above steps 2, 3 and 4 until all items have been added in. Because we are not looking for a particular number but for the numbers smaller than K while the associate value U is larger than V, we can not easily pick one strand out. So we go through one extra step before the final result extraction.

b-6 **divide(S, S1, S2, C)**: The criteria here is $X \leq K$ or $X > K$. Let's assume that S1 contains all strands with $X \leq K$ while S2 holds the rest.

b-7 **select(S1, C)**: We are going to extract the answer from S1 as S1 is the set that contains those "bags" with items less than full, i.e., $X \leq K$. As the value of each strand is represented by a certain number of digits, we only need to go through these digits one by one and find the answer larger than V.

4 Problem Reconsideration

In the previous section, we introduced new algorithms for solving NP complete problems: Knapsack problems. Here we are going to show that the advantage of our algorithm, i.e., unlike other existing algorithms [1] [2] [3] [12] [18] [21] [25] that need to restart the whole computation process when there are changes on the initial condition, our algorithm will only need a few extra operations and the new problem will be solved. This will greatly save time and cost for our DNA computer because usually DNA computing needs a lot of expensive materials and takes very long time, e.g., months, to complete.

We first work on the simplified knapsack problem. The initial condition is an integer K and n items of different sizes. After the procedures we showed in section 3.1, we will obtain a bag with size K and have m items inside where $m<n$. Suppose we want to make a minor change at the initial condition. Let the change be: instead of having n items at the beginning, we lost one of the items. So totally we have n-1 items. If the item is not contributing to the "bag", then nothing will change. If the item is in the "bag" of size K, then we need to generate a totally different new solution. Instead of going through all the steps above, we just add a few new steps to the existing algorithm and it is much easier to obtain the result.

The following are the extra steps we need to add:

A-1 **divide(S, S1, S2, C)**: Seperate the set S into two sets where S1 contains strands with item Y and S2 contains strands without Y. Y is the item we do not want to count.

A-2 **sub(S1, Y)**: S1 is the set left over after the extracting of the previous result.

A-3 **union(S1, S2, S)**: Now S will have no strand containing item Y. So Y has been removed.

A-4 **select(S, C)**: This is the exact same procedure as step a-6 shown above in section 3.1. Still, condition C is regarding whether we have a strand representing number K or not. If at least one such strand exists, then we have a solution. Otherwise, there is no combination of K with these n-1 items.

If more than one item have been removed from the initial list, we need to repeat the above extra steps A-2 and A-4 a few times.

For the complete knapsack problem, similar procedures can be used after the initial condition is changed. If the same modification is performed as the example of simplified knapsack problem showed above: one item is removed from the initial list, the following operations are necessary in order to obtain the solution.

B-1 **union(S1, S2, S)**: This will put the remaining potential answers together.
B-2 **divide(S, S1, S2, C)**: It will sperate the set S into two sets following the criteria C so that set S1 will contain item i which should be removed from the bag and S2 do not have item i in it.
B-3 **sub(y_i, S1)**: Subtract item i from set S1. The detailed operation is that y_i is subtracted from X and the corresponding value v_i is subtracted from U.

Then we only need to perform steps b-6 and b-7 above to see if we have the answer we expected.

5 Conclusion

In this paper, we attempted to solve a set of problems to which DNA computers can apply. As an example, we demonstrate that simplified knapsack problem can be solved efficiently on our DNA computer. We also extend the algorithm to solve the complete version of knapsack problem. These examples have illustrated the advantage of DNA computers.

We note that knapsack problems are NP complete, whether simplified version or complete version. Our method can solve these problems with different complexities within polynomial time. The biggest advantage of using our method to solve these NP-complete problems comparing with other existing methods is that it not only gives out the correct answer, but also saves a lot of computing time and resouces when there are minor changes to the conditions given. We may also extend our algorithm to what is under investigation: the graph connectivity problem. We may also consider cases when the condition change are huge. The future work will include implementing our algorithms in the biological lab and make it more robust.

References

1. Len Adleman. Molecular computation of solutions to combinatorial problems. *Science*, November 1994.

2. Leonard M. Adleman, Paul W.K. Rothemund, Sam Roweis, and Erik Winfree. On applying molecular computation to the data encryption standard. In *Second Annual Meeting on DNA Based Computers*, pages 28–48, June 1996.
3. Joh-Thomes Amenyo. Mesoscopic computer engineering: Automating dna-based molecular computing via traditional practices of parallel computer architecture design. In *Second Annual Meeting on DNA Based Computers*, pages 217–235, June 1996.
4. Martyn Amos. *DNA Computation.* PhD thesis, University of Warwick, UK, September 1997.
5. Martyn Amos, Alan Gibbons, and David Hodgson. Error-resistant implementation of DNA computations. In *Second Annual Meeting on DNA Based Computers*, pages 87–101, June 1996.
6. Eric B. Baum. DNA sequences useful for computation. In *Second Annual Meeting on DNA Based Computers*, pages 122–127, June 1996.
7. Eric B. Baum and Dan Boneh. Running dynamic programming algorithms on a dna computer. In *Second Annual Meeting on DNA Based Computers*, pages 141–147, June 1996.
8. D. Beaver. Molecular computing. Technical report, Penn State University Technical Report CSE-95-001, 1995.
9. Andrew J. Blumberg. Parallel computation on a dna substrate. In *3rd DIMACS Workshop on DNA Based Computers*, pages 275–289, June 1997.
10. Dan Boneh, Christopher Dunworth, Jeri Sgall, and Richard J. Lipton. Making DNA computers error resistant. In *Second Annual Meeting on DNA Based Computers*, pages 102–110, June 1996.
11. Junghuei Chen and David Wood. A new DNA separation technique with low error rate. In *3rd DIMACS Workshop on DNA Based Computers*, pages 43–58, June 1997.
12. Michael Conrad and Klaus-Peter Zauner. Design for a DNA conformational processor. In *3rd DIMACS Workshop on DNA Based Computers*, pages 290–295, June 1997.
13. R. Deaton, R. C. Murphy, M. Garzon, D. R. Franceschetti, and Jr. S. E. Stevens. Good encodings for DNA-based solutions to combinatorial problems. In *Second Annual Meeting on DNA Based Computers*, pages 131–140, June 1996.
14. Myron Deputat, George Hajduczok, and Erich Schmitt. On error-correcting structures derived from DNA. In *3rd DIMACS Workshop on DNA Based Computers*, pages 223–229, June 1997.
15. Dirk Faulhammer, Richard Lipton, and Laura Landweber. Counting DNA: Estimating the complexity of a test tube of DNA. In *Fourth Internation Meeting on DNA Based Computers*, pages 249–252, June 1998.
16. Y. Gao, M. Garzon, R.C. Murphy, J.A. Rose, R. Deaton, D.R. Franceschetti, and S.E. Stevens Jr. DNA implementattion of nondeterminism. In *3rd DIMACS Workshop on DNA Based Computers*, pages 204–211, June 1997.
17. Michael R. Garey and David S. Johnson. *Computers and Intractability.* W. H. Freeman And Company, 1979.
18. Gre Gloor, Lila Kari, Michelle Gaasenbeek, and Sheng Yu. Towards a DNA solution to the shortest common superstring problem. In *Fourth Internation Meeting on DNA Based Computers*, pages 111–116, June 1998.
19. Vineet Gupta, Srinivasan Parthasarathy, and Mohammed J. Zaki. Arithmetic and logic operation with dna. In *3rd DIMACS Workshop on DNA Based Computers*, pages 212–222, June 1997.

20. Masami Hagiya and Masanori Arita. Towards parallel evaluation and learning of boolean mu-formulas with molecules. In *3rd DIMACS Workshop on DNA Based Computers*, pages 105–114, June 1997.
21. Natasa Jonoska and Stephen A. Karl. A molecular computation of the road coloring problem. In *Second Annual Meeting on DNA Based Computers*, pages 148–158, June 1996.
22. Peter Kaplan, David Thaler, and Albert Libchaber. Paralle overlap assembly of paths through a directed graph. In *3rd DIMACS Workshop on DNA Based Computers*, pages 127–141, June 1997.
23. Julia Khodor and David K. Gifford. The efficiency of sequence-specific separation of DNA mixtures for biological computing. In *3rd DIMACS Workshop on DNA Based Computers*, pages 26–34, June 1997.
24. Thomas H. Leete, Matthew D. Schwartz, Robert M. Williams, David H. Wood, Jerome S. Salem, and Harvey Rubin. Massively parallel dna computation: Expansion of symbolic determinants. In *Second Annual Meeting on DNA Based Computers*, pages 49–66, June 1996.
25. Richard Lipton. Using DNA to solve SAT. Unpulished Draft, 1995.
26. Udi Manber. *Introduction To Algorithms*. Addison-Wesley Publishing company, 1989.
27. Nobuhiko Morimoto and Masanori Arita Akira Suyama. Solid phase DNA solution to the hamiltonian path problem. In *3rd DIMACS Workshop on DNA Based Computers*, pages 83–92, June 1997.
28. Mitsunori Ogihara and Animesh Ray. DNA-based parallel computation by 'counting'. In *3rd DIMACS Workshop on DNA Based Computers*, pages 265–274, June 1997.
29. John S. Oliver. Computation with DNA-matrix multiplication. In *Second Annual Meeting on DNA Based Computers*, pages 236–248, June 1996.
30. Z. Frank Qiu and Mi Lu. Arithmetic and logic operations for DNA computers. In *Parallel and Distributed Computing and Networks (PDCN'98)*, pages 481–486. IASTED, December 1998.
31. Sam Roweis, Erik Winfree, Richard Burgoyne, Nickolas Chelyapov, Myron Goodman, Paul Rothemund, and Leonard Adleman. A sticker based architecture for DNA computation. In *Second Annual Meeting on DNA Based Computers*, pages 1–27, June 1996.
32. Erik Winfree. *Algorithmic Self-Assembly of DNA*. PhD thesis, California Institute of Technology, June 1998.
33. Erik Winfree. Proposed techniques. In *Fourth Internation Meeting on DNA Based Computers*, pages 175–188, June 1998.
34. Erik Winfree, Xiaoping Yang, and Nadrian C. Seeman. Universal computation via self-assembly of DNA: Some theory and experiments. In *Second Annual Meeting on DNA Based Computers*, pages 172–190, June 1996.
35. David Harlan Wood. applying error correcting codes to DNA computing. In *Fourth Internation Meeting on DNA Based Computers*, pages 109–110, June 1998.
36. Tatsuo Yoshinobu, Yohei Aoi, Katsuyuki Tanizawa, and Hiroshi Iwasaki. Ligation errors in DNA computing. In *Fourth Internation Meeting on DNA Based Computers*, pages 245–246, June 1998.