Running dynamic programming algorithms on a DNA computer

Eric B. Baum
eric@research.nj.nec.com

Dan Boneh
dabo@cs.princeton.edu

NEC Research Institute
Princeton, NJ 08544

Abstract

In this paper we show that DNA computers are especially useful for running algorithms which are based on dynamic programming. This class of algorithms takes advantage of the large memory capacity of a DNA computer. We present algorithms for solving certain instances of the knapsack problem using a dynamic programming approach. Unlike other algorithms[1, 12] for DNA computers, which are brute force, dynamic programming is the same algorithm one would use to solve (smaller) problems on a conventional computer.

1 Introduction

In a recent seminal paper [1], Adleman introduced the idea of computing using DNA molecules. Adleman's techniques were then generalized by Lipton [12] who showed that formula-SAT can be solved on a DNA computer. These algorithms essentially use a brute force approach to solve hard combinatorial problems. This approach is interesting due to the massive parallelism available in DNA computers. However, the brute force approach is limited by the number of DNA molecules in solution. Breaking the DES cryptosystem [3] is the only "real world" application yet suggested for this approach.

In this paper we propose a new class of algorithms which can be implemented on a DNA computer, namely some algorithms based on dynamic programming. Graph reachability and knapsack are classical problems solvable in this way. These problems are solvable by conventional computers in polynomial time (or pseudo-polynomial time for knapsack), but only so long as they are small enough to fit in memory. DNA computers using dynamic programming could solve substantially larger instances because of their large memory capacity than either conventional computers or previous brute force algorithms on DNA computers.

Throughout the paper we assume an error free model. This means that all molecular biology experiments work perfectly without any errors. There are at least two hurdles to real world DNA computing: finding "killer applications" where they can substantially exceed the capabilities of standard computers, and overcoming errors and other real world departures from the ideal. As a working hypothesis we regard these two hurdles as independent. One may reasonably hope that errors that arise during the operation of a DNA computer can be dealt with. For example, fault tolerant techniques enable error-free computations on conventional machines in the presence of faulty processors. We note that several researchers [5, 10] proposed several techniques for making DNA computers resistant to errors.

2 Dynamic programming

The basic idea of the dynamic programming technique is the following: given a combinatorial problem, we iteratively solve harder and harder problems until the required problem is solved. Thus, to solve the original problem we first solve many easier sub-problems. Classical problems that can be solved using this approach include the shortest path in a graph problem, finding a minimal cost triangulation, the knapsack problem and many others [6].

The reason dynamic programming algorithms are suitable for DNA computers is that the sub-problems can be solved in parallel. Thus, a DNA computer might be useful for solving large instances of the problems mentioned above. Note that this application of a DNA computer is radically different than the one suggested by Adleman [1]. The algorithm we run on the DNA computer is not a brute force algorithm. It is exactly the same algorithm which one would use to solve the problem on a conventional machine. We hope that this application of DNA computers can be used to illustrate their advantages (and disadvantages) with respect to conventional machines.

Currently we only know how to use a DNA computer for dynamic programming algorithms of a very special form. The generic problem which we can solve is that of *graph reachability*: given a directed graph $G = (V, E)$ and two vertices S and T test if there is a path from S to T. The graph reachability problem can be solved using a simple dynamic programming algorithm known as breadth-first-search or BFS for short [6].

As was stated above, our objective is to solve the graph reachability problem for very large graphs. The graph will be described by a set of m functions $f_i : V \to V$ for $i = 1, \ldots, m$. Given a vertex $v \in V$, the neighbors of v are $\{f_1(v), \ldots, f_m(v)\}$. Thus, each vertex can have at most m neighbors. We denote by $C(f_i)$ some fixed circuit computing the function f_i. Using this notation, an instance of the graph reachability problem is $(S, T, m, C(f_1), \ldots, C(f_m))$. Using this formulation it is not difficult to see that the graph reachability problem is PSPACE complete [8].

The BFS algorithm will solve the graph reachability problem in time $O(|E|)$ on a conventional machine. Let d be the diameter of the graph G, i.e. $d = \max\{\text{distance}(u, v) \mid u, v \in V\}$. We give an algorithm by which a DNA computer can solve the graph reachability problem in time $O(d)$. As we shall see, in many applications the diameter of the graph is small even though its size is too large to fit in a conventional computer. Such applications are ideal for a DNA computer.

3 Solving graph reachability using DNA

In this section we show how to run the BFS algorithm on a DNA computer. We first need to recall a basic result from [4].

Theorem 3.1 ([4]) *Let $f : \{0,1\}^n \to \{0,1\}^m$ be a function computable by a circuit of size s. Let T be a solution of DNA strands representing*[1] *a set $S \subseteq \{0,1\}^n$. Then in s "biological-steps" it is possible to construct a solution of DNA strands representing all pairs $\{(x, f(x)) \mid x \in S\}$.*

Let $(S, T, m, C(f_1), \ldots, C(f_m))$ be an instance of graph reachability. The algorithm for solving graph reachability using DNA can be described as follows:

1. Start with a solution containing strands representing the vertex S.

[1] When we say that a solution of DNA strands represents a set S of binary strings we mean that for every $x \in S$ there is at least one strand in the solution representing x. The encoding of binary strings as DNA strands is explained in [4].

2. Iterate the following procedure d times where d is the diameter of the graph:

   (a) Let D be the current solution. Apply PCR to create m copies of the solution D. Let $D_1, \ldots, D_m$ be m separate vats containing the replicas of D.

   (b) Use Theorem 3.1 to compute the function f_i in D_i. We can arrange things so that the strands in solution D_i now look like:

   $$3' \; - \; v \; - \; B_1 \; - \; f_i(v) \; - \; B_2 \; - \; 5'$$

   where B_1 is the restriction site for some enzyme and B_2 is some fixed DNA strand.

   (c) In each D_i apply the restriction enzyme which cuts the strand at B_1. Then extract from D_i all strands containing the string B_2. Denote the set of extracted strands by D'_i.

   (d) Pour all of $D'_1, \ldots, D'_m$ into one vat denoted by D'. Note that D' now represents the set of all neighbors of vertices in D.

   (e) Test if the target vertex T appears in the solution D'. If it does stop. Otherwise set $D \leftarrow D'$ and go to step 2.

Note that in step 2d we pour the contents of m vats into one vat. Since we assume that the volume of a vat is constant (e.g. 1 liter) this step requires that we only pour one m'th fraction out of each of the D'_i. In what follows we analyze the number of DNA strands that need be used so that no information will be lost with high probability.

Unfortunately there is a technical problem which causes this algorithm to fail on certain instances of graph reachability. At every step of the algorithm above, the number of strands representing a vertex V in the graph is proportional to the numbers of paths from S to V. This is fine, as long as the number of paths from S to any vertex V is roughly the same for all vertices V. However, if some vertices have many more paths leading to them then those vertices will dominate the solution. Reachable vertices with only a few paths leading to them might be lost. Thus our algorithm can only work for graphs which are balanced in this sense.

Definition 3.1 *Let G be a directed graph with a distinguished vertex S. For an integer $r > 0$ denote by $P_r(V)$ the number of paths from S to V of length exactly r. Let*

$$M(r) = \max_V \{P_r(V)\} \quad \textit{and} \quad N(r) = \min_{V : P_r(V) > 0} \{P_r(V)\}$$

The graph G is called ϵ-balanced if for any $r > 0$ we have

$$N(r)/M(r) > \epsilon$$

We now analyze the number of DNA molecules needed to make this algorithm work. Our final figure will depend on $1/\epsilon$ which shows that the algorithm will not work on badly balanced graphs.

Let d be the diameter of the graph G. Let W_r be the number of vertices V in G for which there is a directed path from S to V of length exactly r. Define W to be

$$W = \max\{W_r \quad \text{where} \quad 0 < r \leq d\}$$

One can think of W as the size of the widest level in the graph. Note that throughout the algorithm the number of different types of DNA molecules in the solution is bounded by W. However, the total number of DNA molecules is always fixed. This follows since at each step we duplicate each strand m times and then pour out all but an m'th fraction of the solution.

Let G be an ϵ balanced directed graph. We claim that if the initial solution used by the algorithm contains

$$P = \frac{W}{\epsilon}\frac{d}{m}\log^2 W$$

DNA molecules each representing the vertex S then the algorithm will find a path to the vertex T if it exists. This means that throughout the algorithm each vat will contain P DNA molecules. Let V be some vertex for which there is a path of length r from S to V. We show that if P strands are used then vertex V will be present in the solution generated at iteration r of the algorithm. This is sufficient to ensure that the algorithm will work since if a path from S to T exists then vertex T will be found after at most d steps, where d is the diameter of the graph G.

Consider the solution generated at iteration r of the algorithm for some $r > 0$. Recall that there are at most W different types of molecules in the solution. Let k be the number of copies of some molecule in the solution. At step 2d of the algorithm we throw away all but an m'th of the solution. After this step the expected number of copies of the molecule in the solution is k/m. Let $k' = k/m$. Due to random drift the actual number of copies of the molecule would deviate from k' by at most $\sqrt{k'/m}\log W$ with high probability. The number of copies of each molecule should be large enough to ensure that the random drift does not wipe out that molecule.

We now let k be the number of copies of the least frequent molecule in the vat generated at step r of the algorithm. The random drift in the number of copies is at most $\sqrt{rk/m}\log W$ which is bounded by $\sqrt{dk/m}\log W$. Taking k to be $d/m\log^2 W$ ensures that random drift does not kill the least frequent molecule.

Recall that the graph G is ϵ-balanced. This means that the most frequent molecule has at most k/ϵ copies. Since there are at most W different types of molecules in the solution, the total number of molecules is bounded by Wk/ϵ. Substituting the value for k above, we obtain the value P specified above. This proves that having P molecules in each vat generated by the algorithm will ensure that the path to T will be found in d steps if it exists.

4 Communication Issues

In a DNA computer, one has very many "processors", i.e. many DNA molecules. But these have essentially no communication whatever. We know how to compute a function based on the information encoded on each individual string, but in general not how to utilize information from two strings[2]. The balanced graph limitation arises because we have no way to compare strings, and thus to remove duplicate molecules encoding the same node. One can propose several possible methods for removing such duplicate molecules, but it is unclear whether they would be effective.

For example, here is an idea for how to remove duplicate molecules. Devise an encoding whereby if string S codes for a given node on the graph, so does string $\bar{S}$, its Watson-Crick complement. (This is not hard.) One might then arrange to produce the encoding S and $\bar{S}$ in roughly equal numbers, or alternatively to randomly to replace strings by their complements. This could easily be accomplished by including some marker sequence M on all strings created. Extracting on M would get you a tube full of sequences coded in the "positive". PCR and separation and extraction on $\bar{M}$ would get you a tube coded in the complement. You can mix these. Then one might hope that complementary strings might

[2]Reif[13] and Beaver [2] proposed a solution to this problem. This solution, however, involves using enough DNA molecules to encode all pairs of virtual processors. If one might (optimistically) have 10^{18} DNA molecules, then, this proposal would restrict one to 10^9 virtual processors. But since DNA processors are at least 10^9 times slower than electronic ones, this is of little practical interest.

mate, and could then be removed somehow, for example by cutting them with a restriction enzyme acting only on double strings. Since the reaction of combination of S molecules with $\bar{S}$ molecules might be expected to proceed at a rate proportional to the square of their concentrations, one might hope to remove molecules with high concentration, i.e. multiple representation, without removing all the representatives of nodes with low representation. Unfortunately, a rate analysis presented in Appendix A seems to indicate that the time it takes for this reaction to balance the number of copies of each vertex is exponential in the size of the problem. Furthermore, such a procedure seems prone to various kinds of errors.

In general dynamic programming algorithms, one might update a node of a graph as some function of its neighbors. For example each node might store a number, and the number at a given node might be updated as the minimum of its neighbors. We do not know how to realize such general forms of dynamic programming because we do not now how to communicate numbers between DNA strands. We now remark that somewhat more general forms of dynamic programming than simple graph reachability can, in fact, be realized. Let n be some node of the graph represented by DNA string s Let s also encode some Boolean vector x. In general, the circuit evaluation procedure allows one to produce a DNA string s' representing a neighboring node n', and also encoding a Boolean vector $f(x')$ for f some circuit. (What we find hard to do is to encode a Boolean vector $g(x, y)$ where y is some vector stored on a different DNA molecule!)

For example, consider the problem of shortest path in a graph. This is ordinarily solved by a Breadth First Search procedure as discussed above, where however at each node one maintains the distance to the source. This can readily be done by adding 1 each time one produces a new DNA molecule. Adding 1 can of course be represented by a simple circuit. When one produces a string representing the goal node T, it will encode the distance traveled. The first time one produces such a string, its distance will be the shortest. Alternatively, if one has produced many, one can search them (using a binary search) to find the shortest distance encoded.

5 Solving the knapsack problem

The dynamic programming approach for solving the knapsack problem [6] can be easily stated as a graph reachability problem. The diameter of the resulting graph is linear in the size of the knapsack problem. In this section we analyze the types of knapsack problems which can be solved on a DNA computer.

The knapsack problem we are interested in is the following: given positive integers $b_1, \ldots, b_n$ and an integer B, does there exist a subset $S \subseteq \{1 \ldots n\}$ such that $\sum_{i \in S} b_i = B$. This problem is known to be NP-complete[9]. The knapsack problem comes up often in optimization (see [7] and references therein).

We note that Lagarias and Odlyzko [11] showed that random instances of knapsack in which the integers b_i are extremely large (larger than 2^{n^2}) can be solved in polynomial time on a conventional machine. Their approach is based on lattice basis reductions. We intend to show that DNA computers can solve certain instances of the knapsack problem even when the integers b_i are smaller than 2^n.

The knapsack problem can easily be posed as a graph problem as follows: Let $M = \sum_{i=1}^{n} b_i$. Consider an $M \times n$ grid where there is an edge from point (x, i) to $(y, i+1)$ if $y = x$ or $y = x + b_i$. The knapsack problem has a solution if and only if there is a path in the above graph from $(0, 1)$ to (B, n).

Note that the amount of memory required to run the BFS algorithm on the knapsack graph on a conventional machine is $O(M)$. Thus, to solve a knapsack problem with a hundred numbers each of size 10^{14} one needs a memory of size 10^{16}. This is far more than is currently attainable by a conventional computer. A standard computer disk might store 10^{10} bits. An optimistic upper bound on the amount of memory available with current technology is 10^{13} bits. On the other hand, a liter of water can easily contain 10^{16} DNA strands.

To solve the knapsack problem using DNA we intend to use the graph reachability algorithm of the previous section. Recall that for the algorithm to work the graph must be balanced. This means that we can only solve instances of knapsack for which the corresponding graph is ϵ-balanced where ϵ is not too small. The exact bound on ϵ depends on the actual physical apparatus used.

We now discuss how to add numbers using DNA. Let T be a solution of DNA strands representing a set of integers $\{\alpha_1, \ldots, \alpha_k\}$. We assume that the numbers are encoded using decimal digits. That is, the strand representing the number α_i simply encodes the decimal digits of α_i. Without loss of generality we assume that all of $\alpha_1, \ldots, \alpha_k$ contain m decimal digits (if this is not the case, one can always pad the numbers with zeros on the left).

We wish to transform the solution T to a solution T' representing all numbers $\{\alpha_1, \alpha_1+X, \ldots, \alpha_k, \alpha_k+X\}$ for some integer X with m decimal digits. This can be done as follows:

1. Let d be the least significant digit of X. Separate the solution T into 10 solutions $T_0, T_1, \ldots, T_9$ such that T_i contains all numbers in T whose least significant digit is d.

2. In each T_i append to all strands in T_i the oligo representing the digit $i + d \bmod 10$.

3. Pour the solutions $T_0, \ldots, T_{9-d}$ into one vat T^0. Pour the remaining solutions into another vat T^1. The strands in T^0 represent those numbers which have not generated a carry. Similarly, the strands in T^1 represent those which have generated a carry.

4. Iterate this procedure appropriately for each of the m digits.

5. At the end of the process all strands in the solution look like:

$$3' - \alpha_i - S - \alpha_i + X - 5'$$

   where S is some predetermined separator.

6. Use a restriction enzyme to cut all strands at the string S. This will produce the required solution containing all numbers

$$\{\alpha_1, \ldots, \alpha_n, \alpha_1 + X, \ldots, \alpha_n + X\}$$

Suppose we allow 10 extraction operations to be carried out simultaneously. Then the above procedure requires m steps where a step involves 10 extractions in parallel and an append.

To solve the knapsack problem using DNA apply the above procedure for each of the numbers $b_1, \ldots, b_n$ starting with a solution containing only the number 0. If the final solution contains the number B then the answer to the given instance of the knapsack problem is positive. Otherwise the given instance has no solution. It is not difficult to modify the algorithm so that the set S such that $\sum_{i \in S} b_i = B$ is also output by the algorithm.

It is straightforward to generalize this approach to solve other versions of the Knapsack problem. For example, a frequent optimization problem is to find the largest number less than B which can be represented as a sum $\sum_{i \in S} b_i$. The procedure above produces a solution containing all numbers which can be represented as $\sum_{i \in S} b_i$ for any S. By extracting sequentially on each component value it is straightforward to separate this solution into two parts, those less than B and those no smaller than B. By extracting from the first of these sequentially on each component it is straightforward to derive a solution containing only the largest number in it. Sequencing this answers the optimization problem.

The number of steps the algorithm requires is nm. This means that the above example where we are given a hundred numbers each of size 10^{14} can be solved in 1400 "biological-steps". This figure is (hopefully) within the reach of a DNA computer.

6 Conclusions

In this paper we attempted to extend the set of problems to which DNA computers might apply. We noted that DNA computers might be useful for running dynamic programming algorithm since such algorithms require large amounts of memory for solving relatively simple tasks.

As an example we saw that the graph reachability problem can be solved efficiently on a DNA computer. A useful instance of the graph reachability problem is the knapsack problem. Our algorithm was able to solve large instances of the knapsack problem. We estimated that a conventional machine would not be able to solve such large problems using this type of algorithms. We hope that this example will be useful in demonstrating the *raison d'être* of DNA computers.

It is more difficult to implement those dynamic programming algorithms which require communicating information between nodes of a graph, because the DNA molecules in a DNA computer are essentially oblivious of one another, and have no way to communicate. This example may also thus illustrate a limitation of DNA computers. We did suggest some approaches to extending the power of our algorithms.

We note that since the knapsack problem is in NP, one could try to use the circuit satisfaction algorithm of [4] to solve the problem. However, the circuit satisfaction algorithm is a brute force approach. The algorithm will try all possible subsets of numbers $\{b_1, \ldots, b_n\}$ to test if one of them sums up to B. In our example, the input was made up of 100 numbers. This means that the algorithm would have had to test 2^{100} different subsets. Since a liter of water can contain at most 2^{70} DNA strands, the circuit-satisfaction algorithm would not be able to solve this instance of knapsack.

Our algorithm for solving knapsack is not a brute force approach. In fact, it is the exactly the same algorithm that one would use to solve the problem on a conventional machine. The fact that DNA has far more memory than a conventional machine enables a DNA computer, at least in principle, to solve larger instances of the problem.

Acknowledgments: We are grateful to F. Guzman for remarking that our techniques extend to shortest path and to S. Kurtz for the calculation in Appendix A.

References

[1] L. Adleman, "Molecular Computation of Solutions to Combinatorial Problems", Science 266:1021-1024 (Nov. 11) 1994.

[2] D. Beaver, "A Universal Molecular Computer", Penn State University Tech Report CSE-95-001.

[3] D. Boneh, C. Dunworth, R. Lipton, "Breaking DES using a molecular computer", Technical Report CS-TR-489-95, Princeton University, 1995.

[4] D. Boneh, C. Dunworth, R. Lipton, J. Sgall, "On Computational Power of DNA", Princeton CS Tech-Report CS-TR-499-95, `ftp://ftp.cs.princeton.edu/reports/1995/499.ps.Z`

[5] D. Boneh, R. Lipton, "Making DNA Computers Error Resistant", Princeton CS Tech-Report CS-TR-491-95, `ftp://ftp.cs.princeton.edu/reports/1995/491.ps.Z`

[6] T. Corman, C. Leiserson, R. Rivest, "Introduction to Algorithms", McGraw Hill, 1991.

[7] G. Dantzig, "Discrete-variable extremum problems", Operations Research 5, pp. 266–277.

[8] M. Garey, D. Johnson, "Computers and Intractability", Freeman, 1979.

[9] R. M. Karp, "Reducibility among combinatorial problems", in R. E. Miller and J. W. Thatcher (eds), *Complexity of Computer Computations*, Plenum Press, NY, pp 85–103, 1972.

[10] R. Karp, C. Kenyon, O. Waarts, "Error-resilient DNA computations", In Proceedings of Symposium on Discrete Algorithms, SODA 1996.

[11] J. Lagarias, A. Odlyzko, "Solving low density subset sum problems", J. Assoc. Comp. Mach., Vol. 32, 1985, pp. 229–246

[12] R. Lipton, "Using DNA to solve NP-Complete Problems", Science 268:542-545 (Apr. 28) 1995.

[13] J. Reif, "Parallel Molecular Computation", Proceedings of SPAA 95.

[14] W. D. Smith, "DNA Computers in Vitro and Vivo", to appear, DNA Based Computers, eds. R. J. Lipton and E. B. Baum, American Mathematical Society, Providence RI.

7 Appendix A

One idea for removing molecules duplicating representation of a vertex is to set it up so that they can stick to each other (or two of them stick to a killer molecule), so that they can then be removed. This was discussed in §4. The hope was that since the sticking rate would be proportional to the square of the concentration, multiply represented molecules would rapidly be removed. Unfortunately, the time this reaction takes to balance the number of copies of each vertex is too large. The following analysis of reaction times was presented to us by S. Kurtz (private communication).

Let y be the concentration of a molecule. Then we have

$$dy/dt = -y^2$$

because the reaction rate is proportional to the square of the concentration, as molecules must collide to react. We have absorbed a rate constant into the time units. Solving this equation we find

$$y = 1/(t + c)$$

for c a constant. Let's say we have two molecule types. Let A represent the concentration of the first type and B the second. Then

$$A = 1/(t + 1/c_A) \qquad B = 1/(t + 1/c_B)$$

where c_A and c_B are respectively the initial concentrations of A and B. Assume $c_A >> c_B$. Now

$$A/B = (c_A c_B t + c_A)/(c_A c_B t + c_B)$$

We ask for the time at which A/B will be as small as α. Solving we find

$$t = (c_A - \alpha c_B)/(c_A c_B(\alpha - 1))$$

Since $c_A >> \alpha c_B$ we can approximate

$$t \sim 1/(c_B(\alpha - 1))$$

This is the amount of time the reaction takes to balance the concentrations of all vertices. Note that this time depends only on the concentration of the rarest molecules. Since the graph contains an exponential number of vertices the concentration of the rarest vertex must be one over exponential. Consequently, the time t is exponential in the size of the problem showing that the reaction time may be too big.

It is interesting to note that the above anlysis neglected to take the reaction rate constant into account. In reality we obtain a differential equation of the form

$$dy/dt = -\beta y^2$$

where β is some reaction rate constant. Above we normalized β to 1. The solution is now

$$y = 1/(\beta t + c)$$

giving t as $t \sim 1/(c_B(\alpha - 1)\beta)$. It is possible that β is sufficiently large so as to kill the exponential in the expression for t for *reasonable size problems*. For instance, suppose one tries to use our algorithm on a graph with 10^{18} nodes. Then, if time is measured in seconds and β is 10^{15} then the reaction time is reasonable after all. The value of β can only be determined through experimentation.