Running dynamic programming algorithms on a DNA computer

Eric B. Baum
baum@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 the knapsack problem and argue that a DNA computer can solve much larger instances of this problem than a conventional computer. Unlike other algorithms[1],[8] 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 [8] 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 [2] 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 connectivity 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.

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 [4].

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 connectivity*: given a graph $G = (V, E)$ and two vertices S and T test if there is a path from S to T. The graph connectivity problem can be solved using a simple dynamic programming algorithm known as breadth-first-search or BFS for short [4].

As was stated above, our objective is to solve the graph connectivity 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 connectivity problem is $(S, T, m, C(f_1), \ldots, C(f_m))$. Using this formulation it is not difficult to see that the graph connectivity problem is PSPACE complete [6].

The BFS algorithm will solve the graph reachbility 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 show that if $|V| < 10^{17}$ then a DNA computer can solve the graph connectivity problem in time $O(d)$. As we shall see, in many applications, the diameter of the graph is small even though its size is large. Such applications are ideal for a DNA computer.

3 Solving graph connectivity 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 [3].

Theorem 3.1 ([3]) *Let $f : \{0,1\}^n \to \{0,1\}^m$ be a function that can be computed by a circuit of size s. Let T be a solution of DNA strands which represents[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 connectivity. The algorithm for solving graph connectivity using DNA can be described as follows:

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

[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.

(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. Thus, each of the D_i's had better contain at least m copies of each strand.

Unfortunately there is a technical problem which causes this algorithm to fail on certain instances of graph connectivity. 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. 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 graph with a distinguished vertex S. For an integer $d > 0$ denote by $P_d(V)$ the number of paths from S to V of length exactly d. Let*

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

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

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

Our algorithm will only work for ϵ-balanced graphs. That value of ϵ depends on the physical apparatus used for the experiment. We estimate that $\epsilon = 10^{-3}$ is a reasonable figure.

It is interesting to note why we are not able as yet to implement more general forms of dynamic programming. 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. In a DNA computer, one has very many "processors", i.e. many DNA molecules. But these have essentially no communication whatever. We do not know how to even pass a single bit from one to another. In applications to dynamic programming, for example, one would like to remove multiple copies of identical DNA molecules arising from multiple paths to some nodes. But we do not have communication from one DNA molecule to another which can enable them to recognize that they are identical[2].

4 Solving the knapsack problem

The dynamic programming approach for solving the knapsack problem [4] can be easily stated as a graph connectivity problem. The diameter of the resulting graph is linear in the size of the knapsack

[2] Reif[9] proposes a solution to this problem. His solution, however, involves using enough DNA molecules to encode all pairs of virtual processors. If one might (optimistically) have 10^{18} DNA molecules, then, his 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.

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[7]. The knapsack problem comes up often in optimization (see [5] and references therein).

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 connectivity algorithm of the previous section. We first need to 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 within the reach of a DNA computer.

5 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 connectivity problem can be solved efficiently on a DNA computer. A useful instance of the graph connectivity 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.

We have not yet been able 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 note that since the knapsack problem is in NP, one could try to use the circuit satisfaction algorithm of [3] 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} difference 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 to solve larger instances of the problem.

References

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

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

[3] D. Boneh, C. Dunworth, R. Lipton, J. Sgall, "On Computational Power of DNA", to appear.

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

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

[6] M. Garey, D. Johnson, "Computers and Interactability", Freeman, 1979.

[7] 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.

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

[9] J. Reif, "Parallel Molecular Computation", SPAA 95.