B Filtering models

Filtering models, an important class of DNA algorithms, operate by filtering out of the solution molecules that are not part of the desired result. A chemical solution can be treated mathematically as a finite *bag* or *multi-set* of molecules, and filtering operations can be treated as operations to produce multi-sets from multi-sets. Typically, for a problem of size n, strings of size $\mathcal{O}(n)$ are required. The chemical solution should contain enough strings to include many copies all possible answers. Therefore, for an exponential problem, we will have $\mathcal{O}(k^n)$ strings. Filtering is essentially a brute-force method of solving problems.

B.1 Adleman: HPP

B.1.a Review of HPP

Leonard Adleman's solution of the Hamiltonian Path Problem was the first successful demonstration of DNA computing. The *Hamiltonian Path Problem* (HPP) is to determine, for a given directed graph $G = (V, E)$ and two of its vertices $v_{\text{in}}, v_{\text{out}} \in V$, whether there is a *Hamiltonian path* from v_{in} to v_{out}, that is, a path that goes through each vertex exactly once. HPP is an NP-complete problem, but we will see that for Adleman's algorithm the *number of algorithm steps* is linear in problem size.

Adleman (the "A," by the way, of "RSA.") gave a laboratory demonstration of the procedure in 1994 for $n = 7$, which is a very small instance of HPP. (We will use this instance, shown in Fig. IV.7, as an example.) Later his group applied similar techniques to solving a 20-variable 3-SAT problem, which has more than a million potential solutions (see p. 218).[4]

B.1.b Problem Representation

The heart of Adleman's algorithm is a clever way to encode candidate paths in DNA. Vertices are represented by single-stranded 20mers, that is, sequences of 20nt (nucleotides). They were generated at random and assigned to the vertices, but with the restriction that none of them were too similar or complementary. Each vertex code can be considered a catenation of two 10mers: $v_i = a_i b_i$ (i.e., a_i is the $5'$ 10mer and b_i is the $3'$ 10mer). Edges are also

[4] `https://en.wikipedia.org/wiki/Adleman,_Leonard` (accessed 2012-11-04).

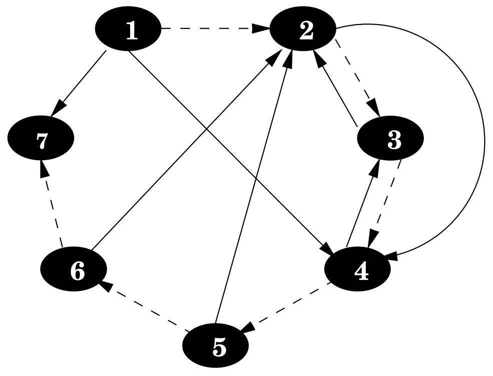

Figure IV.7: HPP solved by Adleman. The HP is indicated by the dotted edges. [source: Amos, Fig. 5.2]

represented by 20mers. The edge from vertex i to vertex j is represented by

$$e_{i\to j} = b_i a_j, \text{ where } v_i = a_i b_i, \text{ and } v_j = a_j b_j.$$

Paths are represented by using complements of the vertex 20mers to stitch together the edge 20mers. (Of course, using the complements of the edges to stitch together the vertices works as well.) For example, a path $2 \to 3 \to 4$ is represented:

$$\begin{array}{c}
\overbrace{b_2\ \ a_3}^{e_{2\to 3}}\ \ \overbrace{b_3\ \ a_4}^{e_{3\to 4}} \\ \hline
\underbrace{\overline{a_3}\ \ \overline{b_3}}_{\overline{v_3}}
\end{array}$$

The edges from v_{in} and to v_{out} have special representations as 30mers:

$$\begin{aligned}
e_{\text{in}\to j} &= v_{\text{in}} a_j, \text{ where } v_j = a_j b_j, \\
e_{i\to\text{out}} &= b_i v_{\text{out}}, \text{ where } v_i = a_i b_i.
\end{aligned}$$

Note that the special representation of the initial and terminal edges results in blunt ends for complete paths.

Therefore, for the $n = 7$ problems, candidate solutions were 140bp in length: There are $n - 1$ edges, but the first and last edges are 30mers. Hence $2 \times 30 + (n - 3) \times 20 = 140$. Ligation is used to remove the nicks in the backbone.

B.1.c Adleman's Algorithm

algorithm Adlemen:

Step 1 (generation of all paths): Generate multiple representations of all possible paths through the graph. This is done by combining the oligos for the edges with the oligos for the complements of the vertices in a single ligation reaction.

Step 2: Amplify the concentration of paths beginning with v_{in} and ending with v_{out}. This is done by PCR using v_{in} and $\overline{v_{\text{out}}}$ as primers. Remember that denaturation separates the sense and antisense strands. PCR extends the sense strand in the $3'$ direction from v_{in}, and extends the antisense strand in the $3'$ direction from $\overline{v_{\text{out}}}$. At the end of this step we have paths of all sorts from v_{in} to v_{out}.

Step 3: Only paths with the correct length are retained; for $n = 7$ this is 140bp. This operation is accomplished by gel electrophoresis. The band corresponding to 140bp is determined by comparison with a *marker lane*, and the DNA is extracted from this band and amplified by PCR. Gel electrophoresis and PCR are repeated to get a sufficient quantity of the DNA. We now have paths from v_{in} to v_{out}, but they might not be Hamiltonian.

Step 4 (affinity purification): Select for paths that contain all the vertices (and are thus necessarily Hamiltonian). This is done by first selecting all those paths that contain v_1, and then, of those, all that contain v_2, and so forth. To select for v_i, first heat the solution to separate the strands. Then add the $\overline{v_i}$ bound to a magnetic bead. Rehybridize (so the beads are bound to strands containing v_i), and use a magnet to extract all the paths containing v_i. Repeat this process for each vertex.

Step 5 If there any paths left, they are Hamiltonian. Therefore amplify them by PCR and inspect the result by gel electrophoresis to see if there are any

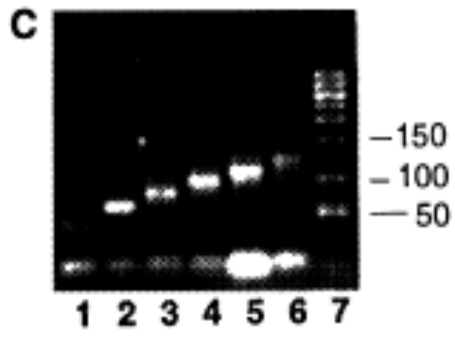

Figure IV.8: Electrophoresis showing solution to HPP problem.

strands of the correct length. If there are, then there is a Hamiltonian path; if there aren't, then there is not. If desired, the precise HP can be determined by a graduated PCR procedure: Run $n - 1$ parallel PCR reactions. In the ith lane, v_{in} is the left primer and v_i is the right primer. This will produce bands with lengths 40, 60, 80, 100, 120, and 140 bp. The lane that has a band at 40 corresponds to the first vertex after v_{in} in the path, the lane with a band at 60 corresponds to the next vertex, etc. This final readout process depends on there being only one Hamiltonian path, and it is error-prone due to its dependence on PCR.
□

B.1.d Discussion

Adleman's algorithm is linear in the number of nodes, since the only iteration is Step 4, which is repeated for each vertex. Step 5 is also linear if the path is read out. Thanks to the massive parallelism of molecular computation, it solves this NP-complete problem in linear time. Adleman's experiment took about a week, but with a more automated approach it could be done in a few hours. On the other hand, the PCR process cannot be significantly shortened.

In addition to time, we need to consider the *molecular resources* required. The number of different oligos required is proportional to n, but the number of strands is much larger, since there must be multiples instances of each possible path. If d is the average degree of the graph, then there are about d^n possible paths (exponential in n). For example, if $d = 10$ and $n = 80$, then the required 10^{80} DNA molecules is more than the estimated number of atoms in the universe. Hartmanis calculated that for $n = 200$ the weight of the DNA would exceed the weight of the earth. So this brute-force approach is still defeated by exponential explosion. Lipton (1995) estimates that Adleman's algorithm is feasible for $n \leq 70$, based on an upper limit of $10^{21} \approx 2^{70}$ DNA strands (Boneh, Dunworth, Lipton & Sgall, 1996), but this is also feasible on conventional computers.

Nevertheless, Adleman's algorithm illustrates the massive parallelism of molecular computation. Step 1 (generation of all possible paths) took about an hour for $n = 7$. Adleman estimates that about 10^{14} ligation operations were performed, and that it could be scaled up to 10^{20} operations. Therefore, speeds of about 10^{15} to 10^{16} ops/sec (1–10 peta-operations/s) should be achievable, which is, digital supercomputer range. Adlemen also estimates that 2×10^{19} ligation operations were performed per joule of energy. Contemporary supercomputers perform only 10^9 operations per joule, so molecular computation is 10^{10} more energy-efficient. It is near the thermodynamic limit of 34×10^{19} operations per joule. Recall (Ch. II, Sec. **??**) $kT \ln 2 \approx 3 \times 10^{-9}$pJ $= 3 \times 10^{-21}$J, so there can be about 3.3×10^{20} bit changes/J.[5]

A more pervasive problem is the inherent error in the filtering processes (due to incorrect hybridization). Some strands we don't want, get through; and some that we do want, don't. With many filtering stages the errors accumulate to the extent that the algorithms fail. There are some approaches to error-resistant DNA computing, but this is an open problem.

[5]DNA is of course space efficient. One bit of information occupies about 1 cubic nm, whereas contemporary disks store a bit in about 10^{10} cubic nm. That is, DNA is a 10^{10} improvement in density.

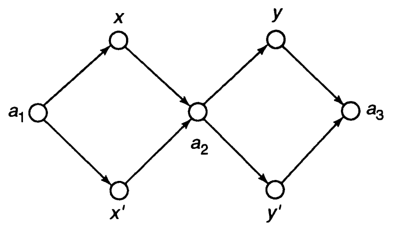

Figure IV.9: Graph G_2 for Lipton's algorithm (with two variables, x and y). [source: Lipton (1995)]

B.2 Lipton: SAT

In this section we will discuss DNA solution of another classic NP-complete problem, *Boolean satisfiability*, in fact the first problem proved to be NP-complete.[6]

B.2.a Review of SAT problem

In the Boolean satisfiability problem (called "SAT"), we are given a Boolean expression of n variables. The problem is to determine if the expression is *satisfiable*, that is, if there is an assignment of Boolean values to the variables that makes the expression true.

Without loss of generality, we can restrict our attention to expressions in *conjunctive normal form*, for every Boolean expression can be put into this form. That is, the expression is a conjunction of *clauses*, each of which is a disjunction of either positive or negated variables, such as this:

$$(x_1 \vee x_2' \vee x_3') \wedge (x_3 \vee x_5' \vee x_6) \wedge (x_3 \vee x_6' \vee x_4) \wedge (x_4 \vee x_5 \vee x_6),$$

For convenience we use primes for negation, for example, $x_2' = \neg x_2$. In the above example, we have $n = 6$ variables $m = 4$ clauses. The (possibly negated) variables are called *literals*.

B.2.b Data representation

To apply DNA computation, we have to find a way to represent potential solutions to the problem as DNA strands. Potential solutions to SAT are

[6]This section is based on Richard J. Lipton (1995), "DNA solution of hard computational problems," *Science* **268**: 542–5.

n-bit binary strings, which can be thought of as paths through a particular graph G_n (see Fig. IV.9). For vertices $a_k, x_k, x'_k, k = 1, \ldots, n$, and a_{n+1}, there are edges from a_k to x_k and x'_k, and from x_k and x'_k to a_{k+1}. Binary strings are represented by paths from a_1 to a_{n+1}. A path that goes through x_k encodes the assignment $x_k = 1$ and a path through x'_k encodes $x_k = 0$. The DNA encoding of these paths is essentially the same as in Adleman's algorithm.

B.2.c Lipton's Algorithm

algorithm Lipton:

Input: Suppose we have an instance (formula) to be solved: $I = C_1 \wedge C_2 \wedge \cdots \wedge C_m$. The algorithm will use a series of "test tubes" (reaction vessels) $T_0, T_1, \ldots, T_m$ and $T_1^i, \overline{T}_1^i, \ldots, T_m^i, \overline{T}_m^i$, for $i = 0, \ldots, n$.

Step 1 (initialization): Create in a test tube T_0 a *library* of all possible n-bit binary strings, encoded as above as paths through the graph.

Step 2 (clause satisfaction): For each clause C_k, $k = 1, \ldots, m$: we will extract from T_{k-1} only those strings that satisfy C_k, and put them in T_k. (These successive filtrations in effect do an AND operation.) The goal is that the DNA in T_k satisfies the first k clauses of the formula. That is, $\forall x \in T_k \; \forall\, 1 \leq j \leq k : C_j(x) = 1$. Here are the details.

For $k = 0, \ldots, m-1$ do the following steps:

Precondition: The strings in T_k satisfy clauses $C_1, \ldots, C_k$.

Let $\ell = |C_{k+1}|$ (the number of literals in C_{k+1}), and suppose C_{k+1} has the form $v_1 \vee \cdots \vee v_\ell$, where the v_i are literals (positive or negative variables). Our goal is to find all strings that satisfy at least one of these literals. To accomplish this we will use an *extraction operation* $E(T, i, a)$ that extracts from test tube T all (or most) of the strings whose ith bit is a.

Let $\overline{T}_k^0 = T_k$. Do the following for literals $i = 1, \ldots, \ell$.

Positive literal: Suppose $v_i = x_j$ (some positive literal). Let $T_k^i = E(\overline{T}_k^{i-1}, j, 1)$ and let $a = 1$ (used below). These are the paths that satisfy this positive literal, since they have 1 in position j.

Negative literal: Suppose $v_i = x'_j$ (some negative literal). Let $T_k^i = E(\overline{T}_k^{i-1}, j, 0)$ and let $a = 0$. These are the paths that satisfy this negative literal, since they have 0 in position j.

In either case, T_k^i are the strings that satisfy literal i of the clause. Let $\overline{T}_k^i = E(\overline{T}_k^{i-1}, j, \neg a)$ be the remaining strings (which do not satisfy this literal). Continue the process above until all the literals in the clause are processed. At the end, for each $i = 1, \ldots, \ell$, T_k^i will contain the strings that satisfy literal i of clause k.

Combine $T_k^1, \ldots, T_k^\ell$ into T_{k+1}. (Combining the test tubes effectively does OR.) These will be the strings that satisfy at least one of the literals in clause $k + 1$.

Postcondition: The strings in T_{k+1} satisfy clauses $C_1, \ldots, C_{k+1}$.

Continue the above for $k = 1, \ldots, m$.

Step 3 (detection): At this point, the strings in T_m (if any) are those that satisfy $C_1, \ldots, C_m$, so do a *detect* operation (for example, with PCR and gel electrophoresis) to see if there are any strings left. If there are, the formula is satisfiable; if there aren't, then it is not.
$\square$

If the number of literals per clause is fixed (as in the 3-SAT problem), then performance is linear in m. The main problem with this algorithm is the effect of errors, but imperfections in extraction are not fatal, so long as there are enough copies of the desired sequence. In 2002, Adelman's group solved a 20-variable 3-SAT problem with 24 clauses, finding the unique satisfying string.[7] In this case the number of possible solutions is $2^{20} \approx 10^6$. Since the degree of the specialized graph used for this problem is small, the number of possible paths is not excessive (as it might be in the Hamiltonian Path Problem). They stated, "This computational problem may be the largest yet solved by nonelectronic means," and they conjectured that their method might be extended to 30 variables.

[7]Ravinderjit S. Braich, Nickolas Chelyapov, Cliff Johnson, Paul W. K. Rothemund, Leonard Adleman, "Solution of a 20-Variable 3-SAT Problem on a DNA Computer," *Science* 296 (19 Apr. 2002), 499–502.

B.3 Test tube programming language

Filtering algorithms use a small set of basic DNA operations, which can be extended to a *Test Tube Programming Language (TTPL)*, such as was developed in the mid 90s by Lipton and Adleman (Adleman, 1995).

B.3.a Basic Operations

DNA algorithms operate on "test tubes," which are multi-sets of strings over $\Sigma = \{\text{A, C, T, G}\}$. There are four basic operations (all implementable):

Extract (or separate): There are two complementary *extraction* (or *separation*) operations. Given a test tube t and a string w, $+(t, w)$ returns all strings in t that have w as a subsequence:

$$+(t, w) \overset{\text{def}}{=} \{s \in t \mid \exists u, v \in \Sigma^* : s = uwv\}.$$

Likewise, $-(t, w)$ returns a test tube of all the remaining strings:

$$-(t, w) \overset{\text{def}}{=} t \;-\; +(t, w) \qquad \text{(multi-set difference)}.$$

Merge: The *merge* operation combines several test tubes into one test tube:

$$\cup(t_1, t_2, \ldots, t_n) \overset{\text{def}}{=} t_1 \cup t_2 \cup \cdots \cup t_n.$$

Detect: The detect operation determines if any DNA strings remain in a test tube:

$$\text{detect}(t) \overset{\text{def}}{=} \begin{cases} \textbf{true}, & \text{if } t \neq \emptyset \\ \textbf{false}, & \text{otherwise} \end{cases}.$$

Amplify: Given a test tube t, the *amplify* operation produces two copies of it: $t', t'' \leftarrow \text{amplify}(t)$. Amplification is a problematic operation, which depends on the special properties of DNA and RNA, and it may be error prone. Therefore it is useful to consider a *restricted model* of DNA computing that avoids or minimizes the use of amplification.

The following additional operations have been proposed:

Length-separate: This operation produces a test tube containing all the strands less than a specified length:

$$(t, \leq n) \overset{\text{def}}{=} \{s \in t \mid |s| \leq n\}.$$

Position-separate: There are two *position-separation* operations, one that selects for strings that *begin* with a given sequence, and one for sequences that end with it:

$$\begin{aligned} B(t,w) &\stackrel{\text{def}}{=} \{s \in t \mid \exists v \in \Sigma^* : s = wv\}, \\ E(t,w) &\stackrel{\text{def}}{=} \{s \in t \mid \exists u \in \Sigma^* : s = uw\}. \end{aligned}$$

B.3.b Examples

AllC: The following example algorithm detects if there are any sequences that contain only C:

```
procedure [out] = AllC(t, A, T, G)
  t ← −(t, A)
  t ← −(t, T)
  t ← −(t, G)
  out ← detect (t)
end procedure
```

HPP: Adelman's solution of the HPP can be expressed in TTPL:

```
procedure [out] = HPP(t, vin, vout)
  t ← B(t, vin)                  //begin with vin
  t ← E(t, vout)                 //end with vout
  t ← (t, ≤ 140)                 //correct length
  for i=1 to 5 do                //except vin and vout
    t ← +(t, s[i])               //contains vertex i
  end for
  out ← detect(t)                //any HP left?
end procedure
```

SAT: Programming Lipton's solution to Sat requires another primitive operation, which extracts all sequences for which the jth bit is $a \in \mathbf{2}$: $E(t, j, a)$. Recall that these are represented by the sequences containing x_j and x'_j. Therefore:

$$\begin{aligned} E(t,j,1) &= +(t, x_j), \\ E(t,j,0) &= +(t, x'_j). \end{aligned}$$

```
procedure [out] = Sat(t)
  for k = 1 to m do                  // for each clause
    for i = 1 to n do                // for each literal
      if C[k][i] = $x_j$               // i-th literal in clause k
        then t[i] ← E(t,j,1)
        else t[i] ← E(t,j,0)
      end if
    end for
    t ← merge(t[1], t[2], ..., t[n]) // solutions for clauses 1,...,k
  end for
  out ← detect(t)
end procedure
```

B.4 Parallel filtering model

The *parallel filtering model* (PFM) was developed in the mid 90s by Martyn Amos and colleagues to be a means of describing DNA algorithms for any NP problem (as opposed to Ableson's and Lipton's algorthms, which are specialized to particular problems). "Our choice is determined by what we know can be effectively implemented by very precise and complete chemical reactions within the DNA implementation."[8] All PFM algorithms begin with a multi-set of all candidate solutions. The PFM differs from other DNA computation models in that removed strings are discarded and cannot be used in further operations. Therefore this is a "mark and destroy" approach to DNA computation.

B.4.a Basic operations

The basic operations are *remove*, *union*, *copy*, and *select*.

Remove: The operation remove$(U, \{S_1, \ldots, S_n\})$ removes from U any strings that contain any of the substrings S_i. Remove is implemented by two primitive operations, *mark* and *destroy*:

Mark: mark(U, S) marks all strands that have S as a substring. This is done by adding $\overline{S}$ as a primer with polymerase to make it double-stranded.

Destroy: destroy(U) removes all the marked sequences from U. This is done by adding a restriction enzyme that cuts up the double-stranded part. These fragments can be removed by gel electrophoresis, or left in the solution (since they won't affect it). Restriction enzymes are much more reliable than other DNA operations, which is one advantage of the PFM approach.

Union: The operation union$(\{U_1, \ldots, U_n\}, U)$ combines *in parallel* the multi-sets $U_1, \ldots, U_n$ into U.

Copy: The operation copy$(U, \{U_1, \ldots, U_n\})$ divides multi-set U into n equal multi-sets $U_1, \ldots, U_n$.

Select: The operation select(U) returns a random element of U. If $U = \emptyset$, then it returns $\emptyset$.

Homogeneous DNA can be detected and sequenced by PCR, and nested PCR can be used in non-homogeneous cases (multiple solutions). All of these operations are assumed to be constant-time. Periodic amplification (especially after copy operations) may be necessary to ensure an adequate

[8]Amos, p. 50.

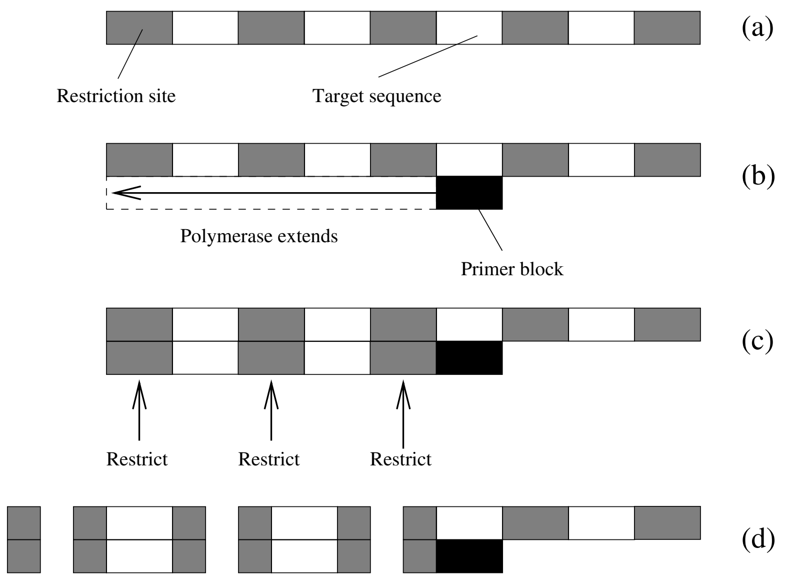

Figure IV.10: *Remove* operation implemented by *mark* and *destroy*. [source: Amos]

number of instances. Amos et al. have done a number of experiments to determine optimum reactions parameters and procedures.

B.4.b Permutations

Amos et al. describe a PFM algorithm for generating all possible permutations of a set of integers.

algorithm Permutations:

Input: "The input set U consists of all strings of the form $p_1 i_1 p_2 i_2 \cdots p_n i_n$ where, for all j, p_j uniquely encodes 'position j' and each i_j is in $\{1, 2, \ldots, n\}$. Thus each string consists of n integers with (possibly) many occurrences of the same integer."[9]

[9]Amos, p. 51.

Iteration:

```
for $j = 1$ to $n - 1$ do
   copy($U, \{U_1, U_2, \ldots, U_n\}$)
   for $i = 1, 2, \ldots, n$ and all $k > j$
      in parallel do remove($U_i, \{p_j i_j \neq p_j i, p_k i\}$)
   // $U_i$ contains $i$ in $j$th position and no other $i$s
   union($\{U_1, U_2, \ldots, U_n\}, U$)
end for
$P_n \leftarrow U$
```

In the preceding, remove$(U_i, \{p_j i_j \neq p_j i, p_k i\})$ means to remove from U_i all strings that have a p_j value not equal to i and all strings containing $p_k i$ for any $k > j$. For example, if $i = 2$ and $j = n - 1$, this remove operation translates to remove$(U_2, \{p_{n-1}1, p_{n-1}3, p_{n-1}4, \ldots, p_{n-1}n, p_n 2\})$. That is, it eliminates all strings except those with 2 in the $n - 1$ position, and eliminates those with 2 in the n position. At the end of iteration j we have:

$$\overbrace{p_1 i_1 p_2 i_2 \cdots p_j i_j}^{\alpha} \underbrace{p_{j+1} i_{j+1} \cdots \ p_n i_n}_{\beta}$$

where α represents a permutation of j integers from $1, \ldots, n$, and none of these integers $i_1, \ldots, i_j$ are in β.

Amos shows how to do a number of NP-complete problems, including 3-vertex-colorability, HPP, subgraph isomorphism, and maximum clique.