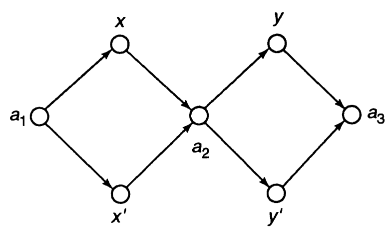

Figure IV.9: Graph G_2 for Lipton's algorithm (with two variables, x and y). [source: Lipton (1995)]

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 a test tube T_0 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.
□

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.