From: ISMB-94 Proceedings. Copyright © 1994, AAAI (www.aaai.org). All rights reserved.

A Restriction Mapping Engine Using Constraint Logic Programming

Trevor I. Dix **Chut N. Yee**[†]

Department of Computer Science
Monash University
Clayton, 3168, AUSTRALIA
trevor@cs.monash.edu.au and cyee@cs.monash.edu.au

Abstract

Restriction mapping generally requires the application of information from various digestions by restriction enzymes to find solution sets. We use both the predicate calculus and constraint solving capabilities of CLP(R) to develop an engine for restriction mapping. Many of the techniques employed by biologists to manually find solutions are supported by the engine in a consistent manner. We provide generalized pipeline and cross-multiply operators for combining sub-maps. Our approach encourages the building of maps iteratively. We show how other techniques can be readily incorporated.

Introduction

Restriction site mapping (RSM) is a very common and important procedure in analysis of DNA molecules. The underlying problem is to construct a map of a DNA molecule. The usual initial step is to break the molecule into fragments which can be investigated individually. Maps of fragments, or indeed sub-fragments, can then be found and combined to obtain a map of the entire molecule. Fragments are obtained by *cleaving* the DNA using a *restriction enzyme*. The RSM problem is the piecing together of all the fragments to yield a map of the restriction sites for the original molecule.

The DNA molecule is prepared in such a way that it can be inserted into another small circular DNA molecule called a *vector*. The insertion is done at a known restriction site. The resulting circular DNA is then completely digested by a restriction enzyme, which cuts the DNA into fragments at all sites with a specific, short subsequence of nucleotide bases. The lengths of the digested fragments are measured.

Digestions are performed for a number of restriction enzymes, and also for pairs of enzymes. These are called single-digests (SD) and double-digests (DD) respectively. In a DD, the DNA is cut at all the restriction sites for both enzymes.

[†] Partially supported by a Monash University FCIT grant and Australian Research Council grant A49330684.

Restriction map construction from digestion data is a combinatorial problem that is well suited for computer application. However, large RSM problems are computationally intractable; the search space grows exponentially with the number of fragments and experimental error (Goldstein and Waterman 1987). Experience (Ho *et al.* 1990) shows that even for problems of relatively small number of fragments the number of consistent solutions is often too big to be useful. To compensate, the biologist usually resorts to data from either biological sources or other supplementary experimental techniques to prune down the number of solutions. Some of these techniques include: sub-experiments, partial digestion, hybridization and end-labeling.

Most RSM programs treat the constraints imposed by fragments in a purely local fashion where the bounds on a fragment only affect placement of adjacent fragments. The main developments in this regard are reflected in the work of Stefik (1978), Pearson (1982), Fitch *et al.* (1983), Zehetner and Lehrach (1986), Zehetner *et al.* (1987) and Krawczak (1988). While progress has been made in solving the standard RSM problem using strict constraints (Allison and Yee 1988) (Dix and Ho-Stuart 1992), we see little attempt to address these diverse and complicated additional techniques.

Another aspect of RSM that is not being adequately addressed is that often not all the digestion data are available at once. The mapping process is an incremental one where the next experiment is decided on by analysis and reasoning about the currently available experimental data. This aspect of restriction mapping is another obvious area where computer tools would prove invaluable.

The RSM problem is a complex synthesis of constraint satisfaction and information processing. Ideally we would like to have a system that integrates these elements. Moreover, the rapid advancements in biological sciences demand that such a system have the flexibility to readily accommodate new techniques and information. It is our judgement that such a system calls for expressive powers beyond the conventional programming framework.

In this paper, we will describe our use of constraint logic programming to implement an integrated restriction site mapping engine. We have chosen this programming framework because of its declarative power and flexibility. In particular, CLP(R) (Jaffar and Lassez 1987) (Jaffar *et al.* 1990) has a general constraint solver in the real number domain that manages the constraint satisfaction problem of RSM automatically.

Our experience demonstrates that the expressiveness of CLP(R) is unmatched in this area. Our engine provides an incremental pipeline generator that is a full generalization of the separate, pipeline and simultaneous permutation schemes first proposed in (Ho *et al.* 1990); a complete implementation of sub-experiments; and consistent and homogeneous treatment of vector sites and fragments using a database of known vectors.

Restriction Site Mapping Problem

Figure 1 illustrates a fragment of DNA to be mapped that has been inserted in a vector. The site for enzymes *a*, *b* and *c* are indicated by sites numbered 1 through 7. The fragment has been inserted at the vector's *a* site.

We denote a digestion by Digest[*E*], where *E* is the cutting enzyme (or enzymes). For example, Digest[*a*] is a SD with enzyme *a* and Digest[*ab*] is a DD with enzymes *a* and *b*. Fragments are denoted by the enzyme(s) and a fragment number, as in a_1, b_2, ac_1, etc.

Figure 2 shows an opened map for figure 1. The map is opened at site 1; sites 1, 2 and 3 are duplicated to represent wrap-around. The map for Digest[*a*] is duplicated to obtain a planar diagram. Digest fragments are represented by lines joining the sites. A double line means two fragments, a SD and a DD fragment, join the same sites, for example, site 6 and 1 are joined by fragments from Digest[*a*] and Digest[*ab*].

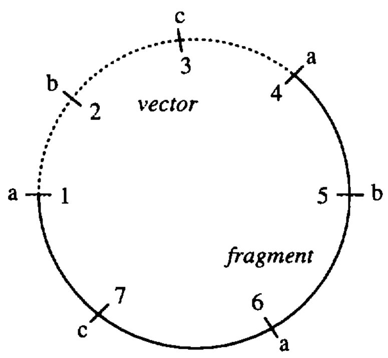

Figure 1: Map for three enzymes

RSM attempts to reconstruct the original map from the measured lengths of the digest fragments. Experimental error in the length of a fragment is represented by a lower and an upper bound. The bound is usually a percentage error on the fragment length.

Placing a fragment between two sites is equivalent to restricting, or constraining, the distance between the sites. If a fragment is placed between sites s_i and s_j, for j > i, the constraint on the sites are:

$$L \le s_j - s_i \le H$$

where H and L are the upper and lower bounds of the fragment concerned. When i > j, the fragment spans the circular boundary, and the constraint becomes:

$$L \le s_j - s_i + C \le H$$

where *C* is a variable that denotes the length of the circular map (Dix and Ho-Stuart 1992).

For a map to be consistent, the constraints imposed by all the fragments must be satisfiable, or mutually consistent. There may be more than one consistent map that falls within the error bounds. We wish to find all such consistent maps.

Most RSM programs only treat constraints on fragments locally. However, each constraint can have a global influence in the system of inequalities (Allison and Yee 1988). Generally, the strict treatment of constraints will reduce the search space and execution time.

Allison and Yee (1988) and Bellon (1988) used logic programming for the RSM problem. Our experience is that more than 60% of logic programming code for RSM dealt with ensuring constraints were satisfied. Yap (1993) demonstrated the use of CLP(R) for two single-digests and one double-digest. The benefit of CLP(R) of over standard logic programming languages is the built-in linear programming solver that can determine the satisfiability of a set of inequalities in the real domain. Moreover, constraints are applied automatically and as soon as possible by the system.

Dix and Ho-Stuart (1992) developed a phased separation theory for a circular restriction map. Our engine handles more general situations, for example,

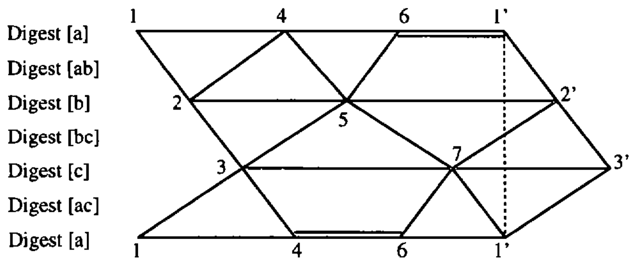

Figure 2: Opened map

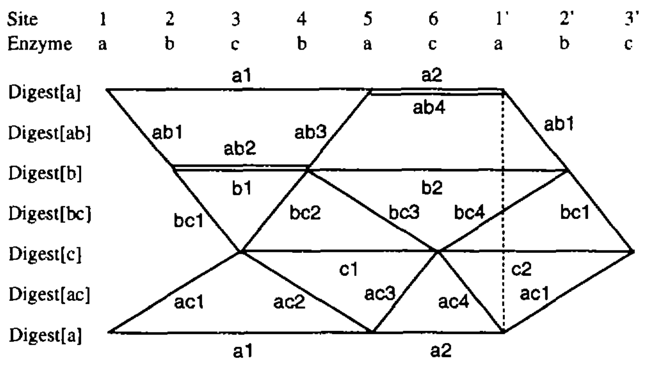

Figure 3: Example generation

circular maps for sub-experiments intersecting with larger circular maps. CLP(R) enforces all constraints correctly without the need to develop a more general theory.

Map Generation

The model that we use for searching for solution maps is based on that of permutations of enzyme sites and digest fragments (Stefik 1982) (Fitch *et al.* 1983) (Dix and Kieronska 1988). The maps are generated by a recursive search process that *extends* the current map by one site at a time. At each step, the site is conjectured to be for one of the enzymes in turn. For each conjecture, we iterate through all possible combinations of digest fragments that could confirm the current site.

Figure 3 shows a map for enzymes *a*, *b* and *c*. The extension process starts from site number 1. Sites 1', 2' and 3' are wrap-around sites for placing the remaining fragments. Table 1 illustrates the placement of the fragments.

The first steps for generating the map are as follows:

1) Site 1 is conjectured to be an *a* site.
2) Site 2 is conjectured to be an *a* site, rejected (say) because we failed to find digest fragments to confirm it. Iterating through the enzymes, site 2 then is conjectured to be a *b* site, and confirmed with fragment ab_1 from Digest[ab] joining sites 1 and 2.
3) Site 3 confirmed as a *c* site, with confirming fragments bc_1 joining sites 2 and 3; and ac_1 joining sites 1 and 3.
4) Etc.

A consistent map is found when all fragments are placed. Each solution is stored. To force CLP(R) to backtrack and find all solutions, the fail predicate is applied. Retract and assert are used to update the fact recording the count of solutions.

Solution Map

A solution map is uniquely defined by the site configuration and the fragment configuration. The site configuration is the order in which the enzymes cut the plasmid. The fragment configuration is the order in which the fragments are placed in each of the digests. The order in which the fragments in different digests are placed relative to each other is implicit in the site ordering.

The map for the example in figure 3 is defined by:

```
Site        : [a,b,c,b,a,c]
Digest[a]   : [a1,a2]
Digest[b]   : [b1,b2]
Digest[c]   : [c1,c2]
Digest[ab]  : [ab1,ab2,ab3,ab4]
Digest[bc]  : [bc1,bc2,bc3,bc4]
Digest[ac]  : [ac1,ac2,ac3,ac4]
```

Mapping Engine

Our engine uses common logic programming techniques to instantiate variables and backtrack on constraint failure. However, CLP(R) uses the test and generate paradigm where appropriate constraints are checked for mutual satisfiability when a constraint is introduced. Satisfiability is ensured by application of the Simplex method. Constraints are tested incrementally. For RSM, this allows maps to be built incrementally, placing one fragment at a time and backtracking to try alternative fragments when a constraint fails.

Table 1: Fragments placed at each site

Site Number	1	2	3	4	5	6	1'	2'	3'
Enzyme	a	b	c	b	a	c	a	b	c
Digest[a]					a_1		a_2		
Digest[b]				b_1				b_2	
Digest[c]						c_1			c_2
Digest[ab]		ab_1		ab_2	ab_3		ab_4		
Digest[bc]			bc_1	bc_2		bc_3		bc_4	
Digest[ac]			ac_1		ac_2	ac_3	ac_4		

The engine is driven by goals, including different sub-map and sub-experiment information for common enzymes and fragments. The nesting of the information in the query determines the order of application of the associated digest information. If the query can be satisfied, a solution map will be available. Currently, CLP(R) only tests constraint satisfiability. Mutual constraints typically reduce the original bounds for each fragment. These final bounds are not directly available as part of the solution.

Program development started with a generalized permutation map generator for an arbitrary number of enzymes and digestions. We then added code for sub-maps and finally sub-experiments. During this process very few changes were made to the permutation generator that forms the core of the RSM engine.

Our CLP(R) code is about 2400 lines, 800 being comments. The permutation core is about 60 lines. Of the remaining lines, about 800 lines deal with the interface. The code runs on a variety of platforms.

Vectors

Traditionally, vector fragments are identified manually prior to the mapping process. Sometimes this approach is unsatisfactory: the vector fragments may not be identifiable within the bounds of experimental error, and removal of these fragments takes constraints imposed by the vector out of the system arbitrarily.

In CLP(R), logical variables and partial instantiation provide a natural way of expressing vector information in a consistent and homogeneous way. In map generation, we require sequences, lists in CLP(R), of fragments and sites. A routine can be written that recursively appends a fragment to the uninstantiated tail of a list, backtracking through alternatives. The same routine can confirm the constraints of a given list. These two modes, generation and confirmation for our application, form the basis of our RSM engine.

For vector fragments, we use the known vector sites to partially define the map. The map generator starts with a partially defined sequence instead of a completely uninstantiated variable and applies the associated constraints to append fragments. If vector fragments cannot be assigned uniquely, each possible start sequence will be tried.

For example, assume the target fragment is inserted into the vector pACYC184 at the EcoRI enzyme site and digestions are performed with enzymes EcoRI, HindII and BamHI. The pACYC184 vector with the fragment attached can be represented by:

```
              Site
 EcoRI     HindII BamHI        EcoRI
   +--------+------+-----------+...target
     1.45kb  0.30kb  2.25kb
              Distance
```

We place the vector at the beginning of the map to make maximum use of the known information. Then the site variable for the map generator will be:

```
Site: ['EcoRI','HindII','BamHI','EcoRI'| _]
```

The square bracket is logic programming syntax for a list. Elements in the lists are separated by commas. The '|' is the symbol that splits the list into a head part and a tail part. Here the tail of the list is unknown and is specified by '_'.

The bounds of the vector fragments are constraints on the distances between the vector sites. They are expressed very simply with the following constraints:

$$V_2 - V_1 = 1.45, \quad V_3 - V_2 = 0.30, \quad V_4 - V_3 = 2.25$$

where V_1, V_2, V_3 and V_4 are variables that represents the location of sites 1, 2, 3 and 4 respectively.

Map Generation in Stages - The Pipeline Scheme

During the RSM process, the biologist will perform sets of digestions, try to map them, then decide whether further experiments are required, and so on. Often not all the digest data are available at once.

Sometimes in constructing a restriction map that involves many enzymes, the biologist will try to divide the problem into smaller sub-problems, each involving a few enzymes, and then try to combine maps from these sub-problems to form the final solution.

For example, say a biologist performs SD and DD with enzymes *a* and *b* – Digest[a], Digest[b] and Digest[ab] – and finds there are many maps satisfying the constraints. To further reduce the solution set, additional digests with enzyme *c* are performed, say Digest[c], Digest[ac] and Digest[bc].

It would be naive to perform the map generation all over again with all the digestion data. Instead, the ability to partially instantiate logical variables gives us a convenient way to use the solution maps obtained for digests *a*, *b* and *ab* as starting points for map generation with the new data. The fragment orders from the *a*, *b* and *ab* digests are already known. We only need permute fragments from the new digests *c*, *ac* and *bc*.

We will illustrate this with an example. Say we found two solutions m1 and m2 over the domain Digest[a], Digest[b] and Digest[ab]:

```
m1 =  Site       : [a,b,a,b]
      Digest[a]  : [a1,a2]
      Digest[b]  : [b1,b2]
      Digest[ab] : [ab1,ab2,ab3,ab4]
m2 =  Site       : [a,b,b,a]
      Digest[a]  : [a1,a2]
      Digest[b]  : [b2,b1]
      Digest[ab] : [ab1,ab3,ab2,ab4]
```

To obtain a new map with the new digestions Digest[c], Digest[ab] and Digest[bc], we first expand m1 and m2 into the domain involving the new digests:

```
m1' = Site       : [a,b,a,b]
      Digest[a]  : [a1,a2]
      Digest[b]  : [b1,b2]
      Digest[ab] : [ab1,ab2,ab3,ab4]
      Digest[c]  : _
      Digest[ac] : _
      Digest[bc] : _
m2' = Site       : [a,b,b,a]
      Digest[a]  : [a1,a2]
      Digest[b]  : [b2,b1]
      Digest[ab] : [ab1,ab3,ab2,ab4]
      Digest[c]  : _
      Digest[ac] : _
      Digest[bc] : _
```

We then use m1' as the initial value for the map generator. With the fragment order of Digest[a], Digest[b] and Digest[ab] already known only permutations of the fragments for Digest[c], Digest[ac] and Digest[bc] are required. The same is done for m2' when map generation with m1' is completed.

Performing map generation in stages like this is usually significantly more efficient than permuting all the digests simultaneously. For the above example, the mapping solutions for digests *a*, *b* and *ab* uses *simultaneous* permutation. The subsequent mapping with digests *c*, *ac* and *bc* is *pipelined.*

The above sequence of operations is denoted by

[a,b,ab] → [c,ac,bc]

where [a,b,ab] denotes the simultaneous permutation of fragments for the *a*, *b* and *ab* digests and '→' denotes the pipeline operator. The simultaneous permutation operator [c,ac,bc] causes the digests within the brackets to be permuted against solutions obtained to the left of the pipeline operator.

The two schemes can be combined in any arbitrary order. For example:

[a,b,ab] → [c,ac] → [bc]

denotes a three stage process. The first stage involves solution maps for digests *a*, *b* and *ab*; these are pipelined with permutations of fragments from *c* and *ac* digests, and in turn for *ac*. Notice, not all applications of the operators are useful. For example, [a,b] or [a]→[b] would generate all possible permutations of the *a* and *b* fragments but with no mutual constraints for pruning.

Divide and Conquer - The Cross Multiply Operator

When many fragments result from many digestions, simultaneous generation is infeasible. Solving sub-problems and applying the pipeline operator described in the previous section is an effective way of overcoming this problem. Moreover, it is a natural path to follow – dividing the problem into sub-problems involving smaller numbers of enzymes and digestions, then merging the sub-maps to form the final solution map.

Merging sub-maps involves cross multiplying the solution sets of the sub-maps. The result is a solution set in the larger domain involving the union of all the enzymes and digests in the sub-maps. The sub-maps usually have digests in common. We use × to denote the cross multiply operator. The cross multiply operator only applies fragment ordering constraints for each common digest and site ordering for common enzymes.

Consider the example in the previous section, which involves 6 digestions over 3 enzymes. There are three ways of dividing it into sub-problems involving two enzymes: [a,b,ab], [b,c,bc] and [a,c,ac].

Say we mapped [a,b,ab] and [b,c,bc]. Let {m1, m2} be the solution set for [a,b,ab]:

```
m1 = Site       : [a,b,a,b]
     Digest[a]  : [a1,a2]
     Digest[b]  : [b1,b2]
     Digest[ab] : [ab1,ab2,ab3,ab4]
m2 = Site       : [a,b,b,a]
     Digest[a]  : [a1,a2]
     Digest[b]  : [b2,b1]
     Digest[ab] : [ab1,ab3,ab2,ab4]
```

and {n1} be the solution set for [b,c,bc]:

```
n1 = Site       : [b,c,b,c]
     Digest[b]  : [b1,b2]
     Digest[c]  : [c1,c2]
     Digest[bc] : [bc1,bc2,bc3,bc4]
```

The fragment order for m2 is incompatible with that of n1 – the *b* digest fragment orders are different. Combining m1 with n1 gives:

```
     Site       : [a,b,(a,c),b,c]
     Digest[a]  : [a1,a2]
     Digest[b]  : [b1,b2]
     Digest[c]  : [c1,c2]
     Digest[ab] : [ab1,ab2,ab3,ab4]
     Digest[bc] : [bc1,bc2,bc3,bc4]
```

We also must check that the site configuration of the sub-maps are compatible, by forming an alignment for the common enzymes. For the above example the common enzyme is *b* and the alignment is as follows:

```
m1: a  b  a  b
       |     |
n1:    b  c  b  c
```

The alignment imposes a partial ordering on the sites. In this case, the partial order is [a,b,(a,c),b,c], where atoms within parentheses can appear in any order (and either or both may be valid solutions).

The site ordering can be determined by using each solution of the cross multiplication as a starting point for the map generator. The sequence of operations is represented by:

$$[a,b,ab] \times [b,c,bc] \rightarrow [\,]$$

The pipeline step will ensure that the sub-maps are mutually consistent. The time required for this extra confirming step is small compared to the simultaneous generation from all SD and DD fragments. From experience, it is almost always worthwhile to confirm the solution after every cross multiplication.

Notice again that not all applications of cross multiplication are useful. [a] × [b] produces the product of permutations of the Digest[a] and Digest[b] fragments.

Sub-Experiments

A sub-experiment is a relatively small RSM problem where a SD fragment is isolated and placed in a vector. Its solution map is usually generated independently. Sites for common enzymes are shared within the sub-experiment fragment and the parent map, as shown in figure 4.

Sites e_i and e_{i+m} in the parent map are cut by the SD enzyme. The SD fragment is inserted at site s_j (cut by the same enzyme) and extends to site e_{j+m} in the sub-experiment. In this figure only common enzymes are shown; there could be other digestions for either map that do not contribute mutual constraints. For common enzymes, all sites in the parent map (within s_j and s_{j+m} above) must occur in the sub-experiment map (within sites e_i and e_{i+m}).

There are two mutual constraints for common enzymes:

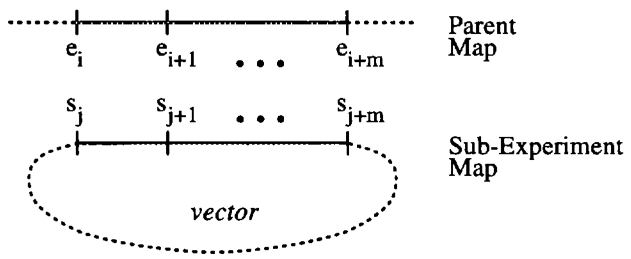

Figure 4: Common sites in sub-experiment

(1) The site configuration for the common enzymes are the same.
(2) The relative positions of the sites are mutually compatible.

This mutual consistency check is denoted by •. For a parent map to succeed, it must be mutually consistent with at least one solution map from each sub-experiment. Since we suppress the generation of symmetric solutions, we also consider the reflection of one of the maps.

For example, let the parent map be M and $S_1 = \{m_1, m_2, ...\}$ be a sub-experiment and its solution set. Similarly for $S_2 = \{n_1, n_2, ...\}$. Finding sub-experiment maps that confirm M is represented by:

$$M \bullet \text{choice}(S_1) \bullet \text{choice}(S_2)$$

where *choice*(S) backtracks through all choices from set S. The choice operator is required because mutual constraint (2) above may propagate to sites beyond the sub-experiment fragment. Confirming a parent map thus involves a backtracking search through the cross product of the solution sets for the sub-experiments. A parent map may have more than one confirming combination of sub-experiments maps.

We could apply the • operation whenever we place a sub-experiment fragment. The constraints from the sub-experiment will limit the search space, however this makes the choice operator a branching factor and maps with multiple confirming sub-experiment maps will be returned as distinct solutions.

We overcame this problem with a compromise operator ○ that only applies mutual constraint (1) above, the site configuration:

$$M \circ \text{first}(S_1) \circ \text{first}(S_2)$$

where *first*(E) finds the first element in E that satisfies the ○ operator. The choice operator becomes unnecessary because constraint (1) is local. We delay applying the • operator until the extension of the parent map is completed.

We found the simple checking for site configuration to be very effective in cutting down the search space and execution time due to the surprisingly small number of site configurations in sub-experiments. For sub-experiments with 8 sites and up to 50 solutions the number of site configurations is usually one and hardly ever more than two.

Results

We now give results of execution with a randomly generated digestion with three enzymes. The enzymes are represented by *b* (BamHI), *e* (EcoRI) and *h* (HindII) respectively. The vector used is *pACYC184*, whose pictorial representation is given earlier. The length of the vector is 4kb, and the length of the DNA is 14kb.

The data was generated by making random cuts from a uniform distribution for each enzyme. The length of the digest fragments are computed from the enzyme sites. The number of sites (including the vector sites) for each enzyme are 6, 7 and 8. Data are generated for the full set of SD's and DD's, a total of 6 digestions, *b*, *e*, *h*, *be*, *bh* and *eh*.

This artificial problem is quite large and difficult for current mapping programs to solve. Even with 0% error, there are 8 solutions that satisfy the constraints. The program was run on a relatively slow Decstation 2100 under Ultrix.

Table 2 presents the results for various numbers of sub-experiments. Fragments for sub-experiments were chosen randomly from amongst the longest fragments. All the sub-experiment maps are generated using the full set of SD's and DD's. An entry in the table with fewer sub-experiments is not necessarily a proper subset of entries with more sub-experiments. There are two stages: first applying the ○ operator when introducing a sub-experiment and second applying the • operator when a solution has been generated that satisfies the weaker constraints. The difference in number of solutions and computation times between the best and worst entries are quite marked, 60 times and 15 times respectively. There

Table 2: Effect of sub-experiments on the number of solutions and computation time

Number of	Error = 2%		Error = 3%			Error = 4%		
Sub-	Solns	Time	Solns		Time	Solns		Time
Experiments	○ and •	(100 sec)	○	•	(100 sec)	○	•	(100 sec)
0	240	23	4608	4608	216	6144	6144	410
1	240	14	2304	2304	133	3072	3072	249
2	64	16	384	256	140	512	384	285
3	32	14	1536	1024	156	2048	1536	363
4	16	12	384	256	99	512	384	154
5	16	6	96	64	32	128	96	53
6	16	3	96	64	17	128	96	29
7	16	3	96	64	17	128	96	28

Table 3: Computation time for various generation rules with 6 digestions

Rule	Stage	Error = 0%		Error = 2%		Error = 4%	
		Solns	Time (sec)	Solns	Time (sec)	Solns	Time (sec)
1. [b,e,h,be,bh,eh]	1	8	40	16	292	96	2920
	Total		40		292		2920
2. [b,h,bh]→[e,be]→[eh]	1	4	8	4	46	4	332
	2	4	23	4	70	12	260
	3	8	9	16	37	96	443
	Total		46		166		1068
3. [b,e,be]x[b,h,bh]→[eh]	1	48	59	48	100	144	299
	2	4	8	4	44	4	344
	3	8	66	16	166	96	1510
	Total		140		325		2180
4. [b,c,be]x[b,h,bh]→[]→[eh]	1	48	58	48	101	144	301
	2	4	8	4	46	4	360
	3	4	25	4	37	12	217
	4	8	9	16	36	96	445
	Total		106		234		1355
5. [e,h,eh]→[b,be,bh]	1	8	47	96	4094	288	34600
	2	8	22	16	660	96	4209
	Total		75		4770		38900

is a general trend in reduction of the number of solutions and time. However, the increases for 2 and 3 sub-experiments illustrate the data dependence of such search problems. Some sub-experiments provide more useful data than others, but this can not be predicted.

Table 3 presents the results of map generation with various computation rules for the 6 digestions from table 2. The rows for each rule represent the stages of computation, with the total execution time, in seconds, in the last. The total time includes various initializations and all sub-experiment generation, so it is greater than the sum of the sub-columns.

The first entry, the simultaneous permutation of all the digests, serves as a reference for others. The difference in time with the various rules are quite drastic, almost two orders of magnitude between the best entry (2) and the worst entry (5). The reference entry lies somewhere in the middle.

The timing for the various operators is highly data dependent. In general, simultaneous permutation should only be used for a small number of digests and fragments. Pipelined mapping in stages usually performs best.

Conclusion

We have used CLP(R) to implement a RSM engine that is sound and complete with respect to the strictest application of constraints. We encorporate a complete implementation of sub-maps and sub-experiments, dealing with vector sites consistently. With the generalized pipeline and cross-multiply operator the RSM engine also allows very many ways of building a map in stages and dividing a problem into smaller sub-problems. The implementation encourages the building of maps iteratively.

Perhaps the most important criteria for a RSM implementation is the capacity to adapt to new experimental techniques effectively and easily. We find CLP(R) is unmatched in this respect. We initially chose CLP(R) as a prototype language. However, as the project progressed we realized that the exponential time complexity of the RSM problem is the determining factor in execution time. The constant factor between different implementation languages is outweighed by flexibility.

We also found the most effective way to reduce exponential explosion is to employ divide and conquer techniques and make full use of information provided by auxiliary sources like sub-experiments. The problem covered in tables 2 and 3 has 6 digestions, 21 sites and a total of 63 fragments. This is far too large for our previous programs that were written in C (Dix and Kieronska 1988) (Ho *et al.* 1991). We were able to solve it within a reasonable time using both divide and conquer and sub-experiment techniques.

Other aspects of RSM like partial digestion, end-labeling and hybridization probes can be readily incorporated into our implementation. For example, a probe can be represented by a site variable s_p. When placing a hybridized fragment between the sites s_i and s_j, we only need to introduce an extra constraint:

$$s_i \le s_p \le s_j$$

We are currently building an integrated graphical user interface and data management system to drive the RSM engine.

References

Allison, L.; and Yee, C.N. 1988. Restriction site mapping is in separation theory. *Computer Applications in Biosciences* 4: 91-101.

Bellon, B. 1988. Construction of restriction maps. *Computer Applications in Biosciences* 4: 111-115.

Dix, T.I.; and Ho-Stuart, C.J. 1992. Constraint checking for circular restriction site mapping. In Proceedings of Twenty-Fifth Annual Hawaii International Conference on Systems Sciences, 635-642.

Dix, T.I.; and Kieronska, D.H. 1988. Errors between sites in restriction site mapping. *Computer Applications in Biosciences* 4: 117-123.

Fitch, W.M.; Smith, T.F.; and Ralph, W.W. 1983. Mapping the order of DNA restriction fragments. *Gene* 22: 19-29.

Goldstein, L.; and Waterman, M.S. 1987. Mapping DNA by stochastic relaxation. *Advances in Applied Mathematics* 8: 194-207.

Ho, S.T.S.; Allison, L.; and Yee, C.N. 1990. Restriction site mapping for three or more enzymes. *Computer Applications in Biosciences* 6: 195-204.

Ho, S.T.S.; Allison, L.; Yee, C.N.; and Dix, T.I. 1991. Constraint checking for restriction site mapping. In Proceedings of Twenty-Fourth Annual Hawaii International Conference on Systems Sciences, 605-614.

Jaffar, J.; and Lassez, J-L. 1987. Constraint logic programming. In Proceedings 14th ACM Symposium on Principles of Programming Languages, 111-119.

Jaffar, J.; Michaylov, S.; Stuckey, P.J.; and Yap, R.H.C. 1992. The CLP(R) language and system. *ACM Transactions on Programming Languages and Systems* 14(3): 339-395.

Krawczak, M. 1988. Algorithms for restriction site mapping of DNA molecules. In Proceedings of National Academy of Science USA 85, 7298-7301.

Stefik, M. 1982. Inferring DNA structures from segmentation data. *Artificial Intelligence* 11: 85-114.

Pearson, W.R. 1982. Automatic construction of restriction site maps. *Nucleic Acids Research* 10: 217-227.

Yap, R.H.C. 1993. A constraint logic programming framework for constructing DNA restriction maps. *Artificial Intelligence in Medicine* 5: 447-464.

Zehetner, G.; and Lehrach, H. 1986. A computer program package for restriction map analysis and manipulation. *Nucleic Acids Research* 14: 335-349.

Zehetner, G.; Frischauf, A.; and Lehrach, H. 1987. Approaches to restriction map determination. In Nucleic Acid and Protein Sequence Analysis, a Practical Approach, 147-164. M.J. Bishop and C.J. Rawlings (Eds.), IRL Press.