University of Missouri

Course Final Report

Computational Optimization Methods

Author:

Truc Le
Sean Lander
Giang Bui
Brett Koonce
Zhaolong Zhong

Supervisor:

Dr. Jianlin Cheng

Columbia – Missouri

December 9, 2013

Contents

List of Figures **ii**

List of Tables **iii**

1 MCMC for Motif Search **1**
- 1.1 Abstract . . . 1
- 1.2 Introduction . . . 1
- 1.3 Gibbs Sampling Methods for Motif Search . . . 3
- 1.4 Results and Visualization . . . 3
- 1.5 Implementation Discussion . . . 4

2 Hill Climbing and Simulated Annealing for TSP **9**
- 2.1 Abstract . . . 9
- 2.2 Introduction . . . 9
- 2.3 Move Set . . . 11
- 2.4 Hill Climbing . . . 11
- 2.5 Simulated Annealing . . . 13
- 2.6 Results and Visualization . . . 13
- 2.7 Conclusion . . . 13

3 Sequence Alignment by Dynamic Programming **20**
- 3.1 Abstract . . . 20
- 3.2 Introduction . . . 20
  - 3.2.1 Dynamic Programming . . . 20
  - 3.2.2 Sequence Alignment . . . 21
- 3.3 Sequence Alignment Algorithm . . . 21
  - 3.3.1 Preliminary . . . 21
  - 3.3.2 Global Alignment . . . 22
  - 3.3.3 Local Alignment . . . 23
- 3.4 Results and Visualization . . . 24

4 Max-Flow/Min-Cut by Linear Programming **30**
- 4.1 Abstract . . . 30
- 4.2 Introduction . . . 30
- 4.3 Maximum Flow . . . 31
- 4.4 Minimum Cut . . . 32
- 4.5 Results and Evaluation . . . 33

5 Implementing a Small SVM **35**
5.1 Abstract 35
5.2 Introduction 35
5.3 Sequential Minimal Optimization 36
5.3.1 KKT condition 36
5.3.2 Optimizing Alphas 37
5.4 Gradient Descent 38
5.5 Results and Evaluation 38
5.6 Visualization 42

List of Figures

1.1 Proposal 1 versus Proposal 2 on a short motif . . . 5
1.2 Proposal 1 versus Proposal 2 on a long motif . . . 6
1.3 Proposal 1 motif result for $k = 13$ and $k = 17$ from dataset M00931 . . . 7
1.4 Proposal 2 motif result for $k = 13$ and $k = 17$ from dataset M00931 . . . 7
1.5 Proposal 1 motif result for $k = 13$ and $k = 17$ from dataset M00971 . . . 8
1.6 Proposal 2 motif result for $k = 13$ and $k = 17$ from dataset M00971 . . . 8

2.1 Candidate move generation . . . 10
2.2 Hill Climbing using Two-Swap on 15 cities-dataset . . . 15
2.3 Hill Climbing using Two-Flip on 15 cities-dataset . . . 16
2.4 Simulated Annealing using Two-Swap on 15 cities-dataset . . . 16
2.5 Simulated Annealing using Two-Flip on 15 cities-dataset . . . 17
2.6 Hill Climbing using Two-Swap on 57 cities-dataset . . . 17
2.7 Hill Climbing using Two-Flip on 57 cities-dataset . . . 18
2.8 Simulated Annealing using Two-Swap on 57 cities-dataset . . . 18
2.9 Simulated Annealing using Two-Flip on 57 cities-dataset . . . 19

3.1 An amino acid scoring matrix: BLOSUM62 . . . 21
3.2 Sub-problems . . . 23
3.3 Global alignment scoring matrix . . . 25
3.4 Local alignment scoring matrix . . . 25

4.1 Max-Flow and Min-Cut result . . . 34

5.1 Optimizing outcome function . . . 42
5.2 Optimizing α values. Red indicates KKT violation . . . 42
5.3 Final outcome function . . . 42
5.4 Final α values . . . 42

List of Tables

2.1 Tour Length Result Summary 15

5.1 Comparison of Various Methods for Training a SVM 41

Preface

Computational Optimization is the process of using computers to find optimal (or reasonably close) solutions to mathematically definable problems. Many small problems can be tackled via brute force, but as the size and complexity of the problem mounts it increasingly becomes the duty of the researcher to guide the computer down promising lines and avoid getting sidetracked by noise or local optima.

In this quest, it is exceedingly valuable to have a map of the variety of techniques available in the field. Towards this end, we attack a number of real world problems with well-known classical and modern techniques. Some perform extremely well and others quickly reach their limitations. The frontiers of science, though, are rarely well defined. Today's obscure, under-performing algorithm could well be the basis of tomorrow's cutting edge research.

Ultimately, the computer is nothing more than another tool: a glorified abacus, perhaps. However, if we can break a problem into tiny, repeatable steps, the machine can solve complicated problems in a provable manner far beyond human abilities. As such, it is the duty of the computer scientist to know not only what the machine can do but moreover what the machine cannot do. Only then can the researcher have any hope of extending its capabilities in the future. This report documents our tiny steps toward this goal.

Columbia – Missouri, December 11, 2013

Truc Le, Sean Lander, Giang Bui, Brett Koonce, Zhaolong Zhong

Chapter 1

Using Markov Chain Monte Carlo for Motif Search

1.1 Abstract

We implemented Markov Chain Monte Carlo (MCMC) to discover sequence motifs (common similar patterns) in DNA sequences. We implemented two methods, the first a greedy version of the algorithm (P1) and a less greedy but more non-deterministic method (P2). We found that the greedy version performed better on shorter sequences but on longer sequences the second method performed significantly better and would recommend using this method going forward.

1.2 Introduction

Markov Chain Monte Carlo is a class of methods in statistics which utilizes Markov chains to create a state space which can then sampled via Monte Carlo (stochastic) methods. If the underlying Markov model is robust, then Monte Carlo methods can be used efficiently to generate approximate real-world distributions without having a proper formal model of the problem domain. Gibbs sampling, which we utilized, further improves upon this method by using a known state space which in turn eliminates bad samples (by its nature, since every sample is drawn from a known sample state space).

We applied this method to the problem of finding DNA sequence motifs - that is to find common sub-sequences within DNA chains. This problem is not a traditional string searching problem, though, in that DNA (by its organic nature) is not perfectly well defined in its sequences. That is to say, a pair of sequences may differ by only a single codon. As such, we have to do a fuzzy search of all the potential sub-sequences within the DNA strand to find the best approximate solution. Finding these sequences is a common problem in bioinformatics, where identifying motifs allow researchers to identify genes.

Algorithm 1.1 Gibbs Sampling Motif Search Proposal 1

Input: Set of sequences $\left\{S^{(i)}, i = 1, \ldots, p\right\}$ and the motif length k
Output: Starting positions for each motif in each sequence $X = \{x_1, \ldots, x_p\}$

```
1: Randomly initialize state $X$
2: repeat
3:     for $i \leftarrow 1, p$ do
4:         $M(c,j) \triangleq \frac{\left|\left\{l \middle| S^{(l)}_{x_l+j-1}=c,\ l \in \{1,\ldots,p\} \setminus \{i\}\right\}\right|+1}{p-1+4}$   for   $c \in \{A,G,T,C\}, \quad j = 1,\ldots,k$
5:                                                        ▷ Laplace smoothing probability
6:         for $j \leftarrow 1, \text{length}\left(S^{(i)}\right) - k + 1$ do
7:             $P(x_i = j|M) = \prod_{l=1}^{k} M(S^{(i)}_{j+l-1}, l)$
8:         end for
9:         $x_i \leftarrow \text{argmax}\,(P(x_i|M))$
10:    end for
11: until convergence
```

Algorithm 1.2 Gibbs Sampling Motif Search Proposal 2

Input: Set of sequences $\left\{S^{(i)}, i = 1, \ldots, p\right\}$ and the motif length k
Output: Starting positions for each motif in each sequence $X = \{x_1, \ldots, x_p\}$

```
1: Randomly initialize state $X$
2: repeat
3:     for $i \leftarrow 1, p$ do
4:         $M(c,j) \triangleq \frac{\left|\left\{l \middle| S^{(l)}_{x_l+j-1}=c,\ l \in \{1,\ldots,p\} \setminus \{i\}\right\}\right|+1}{p-1+4}$   for   $c \in \{A,G,T,C\}, \quad j = 1,\ldots,k$
5:                                                        ▷ Laplace smoothing probability
6:         for $j \leftarrow 1, \text{length}\left(S^{(i)}\right) - k + 1$ do
7:             $P(x_i = j|M) = \prod_{l=1}^{k} M(S^{(i)}_{j+l-1}, l)$
8:         end for
9:         Sample new $x_i$ according to $P(x_i|M)$, update current state $X$
10:    end for
11: until convergence or maximum number of iterations hit
```

1.3 Gibbs Sampling Methods for Motif Search

Having identified our problem and the method by which we were going to attack it, we next set out to implement the motif algorithm. The basic idea is simple enough, to find the minimal Hamming distance (simple deviation from the motif string) and most common motif in the same pass. Gibbs sampling helps tremendously here by allowing us to use the sample input to build our state space. From the sample, we build out a state space of every possible k-length motif for each position. From here our approach split into two proposals.

In the first method we tested, we built a target profile (randomly in the first pass) and then successively improved the profile each round by setting it to the best possible profile in that generation and then repeating this process until we had reached maximum improvement. In the second method we tested, we sampled the new state according to the probability distribution computed at each position in each generation and selected the mode for our new motif in the next generation. While this does not guarantee the information gain of P1, the best position still generally has the highest chance of being selected while other positions can also be picked.

Loosely speaking, Proposal 1.1 (P1) is a classical greedy algorithm, whereas Proposal 1.2 (P2) converges much more slowly (but is proved more robust for reasons we will go into shortly). Notice that P1 and P2 only differ at line 9.

To evaluate the result, we use *information* - which is the opposite to entropy. The information is a real-value quantity taken value from 0 to 2 where 0 indicates no information and 2 indicates a perfect match across p sequences. Here we consider average information (i.e. the sum of information values for all k-length motifs divided by k). We plotted the information gain of each step against the number of iterations to produce the graphs in the following section.

We wrote our initial implementation in Matlab but switched to C++ in order to improve the performance of our code. One thing we added to the above was implementing Laplace smoothing on the motif space before starting P1/P2 to eliminate the possibility of zero-probability zones. We indeed tested and realized that smoothing the profile matrix always yields a better a result compared to non-smoothing one.

Since both P1 and P2 are stochastic, a common technique is to initialize many starting states (i.e. run many Gibbs samplers) so that the expected outcome reaches the optimal result. When working with less initializations and a shorter target motif ($k = 6$), P1 performed significantly better than P2, which could not even find a solution in this smaller state space.

1.4 Results and Visualization

We have tested both methods on 12 files (i.e. 12 sets of sequences) with different configurations (e.g. motif length, number of chains). Due to space limits, we only show some of the results, but the whole dataset is available on the server and they can be reproduced easily by running our code. Other results behave similarly. We used the web logo tool `http://weblogo.berkeley.edu/logo.cgi` to visualize the output of our test runs. For example, Fig.1.3 is the P1 results with $k = 13$ and $k = 17$, respectively. Likewise, Fig.1.4 is the P2 results with $k = 13$ and $k = 17$, respectively. Both of them run 50 chains. Fig.1.5 and 1.6 are respectively P1 and P2's results with $k = 13$ and $k = 17$ using 6 chains.

In Fig.1.1, we plot the information versus iterations for each proposal. We run 6 chains (Gibbs samplers) and use a short motif of length 6. As we can see that P1 reaches its optimum very fast and converges while P2 oscillates a lot and does not converge. However, when using a larger number of initializations (50) and larger motif lengths (13 and 17), respectively, we can see more clearly the difference between the two approaches. As can be seen in Fig.1.2, P1 quickly hits its maximum potential, whereas P2 continues to slowly improve as it gets deeper into the search process. Even though this requires more cycles, we feel this method is a better choice because it has less danger of getting stuck on local maxima (a common problem with greedy algorithms). The P1 approach could be improved with traditional solutions such as random restarts, but this technique does not help too much because the number of initializations needed grows exponentially with the number of sequences as well as the length of each sequence. As far as for finding motifs in general, we found that both these methods were relatively stable between different sequence lengths. However, they did not output the same sequences in the same test data.

Recall the last paragraph from the previous section and based on Fig.1.3, 1.4, 1.5, 1.6, it seems that P1 outperforms P2 on short motifs and P2 works better than P1 on long motifs. One possible explanation for this phenomenon is as follows. For long motifs (e.g. $k = 13$ or $k = 17$), graph of the information versus state space has only a few peaks. Hence, a greedy algorithm like P1 easily gets stuck in the local optimum and cannot escape from it, whereas P2 has a chance to *step back* and go forward to find an improved optimum. Nevertheless, for short motif search (e.g. $k = 6$), the information versus state space graph has many peaks and these peaks are almost at very similar height. Thus, the greedy strategy of P1 helps it reach the maxima quickly and stably. P2, on the other hand, keeps jumping across these valleys due to its stochasticity.

It is interesting to note that these two methods did not identify the same motifs even though the two samples above are from the same DNA sequence. We would like to test our algorithm on a known good sample before declaring authoritatively which method is better. Having said that, we believe that proposal 2, the less greedy search algorithm, would be our preferred candidate because it has less chance of getting caught in local maxima. As such, we would recommend it going forward, especially in bioinformatics, where longer sequences and motif search patterns are likely to be encountered.

1.5 Implementation Discussion

Throughout the implementation process, we recommend these implementation tricks. Firstly, we do not need to build the profile matrix from scratch. Instead, we can update it from the last iteration because they only differ in two sequences. What we need to do is to update the change of matrix entries caused by these two sequences. The same technique can also be applied in the evaluation process, where we need to give a score to a state.

The second issue is that to avoid too many floating point multiplications when dealing with calculating probability for each position, we suggest to take the log of probability to reduce from product to sum of terms, which is significantly more efficient. Moreover, since the only concern is the frequency, not really probability, we can use integer operations to speed up so that floating point operations do not appear in proposal 1 and only occur during re-sample step (line 9) in Proposal 2.

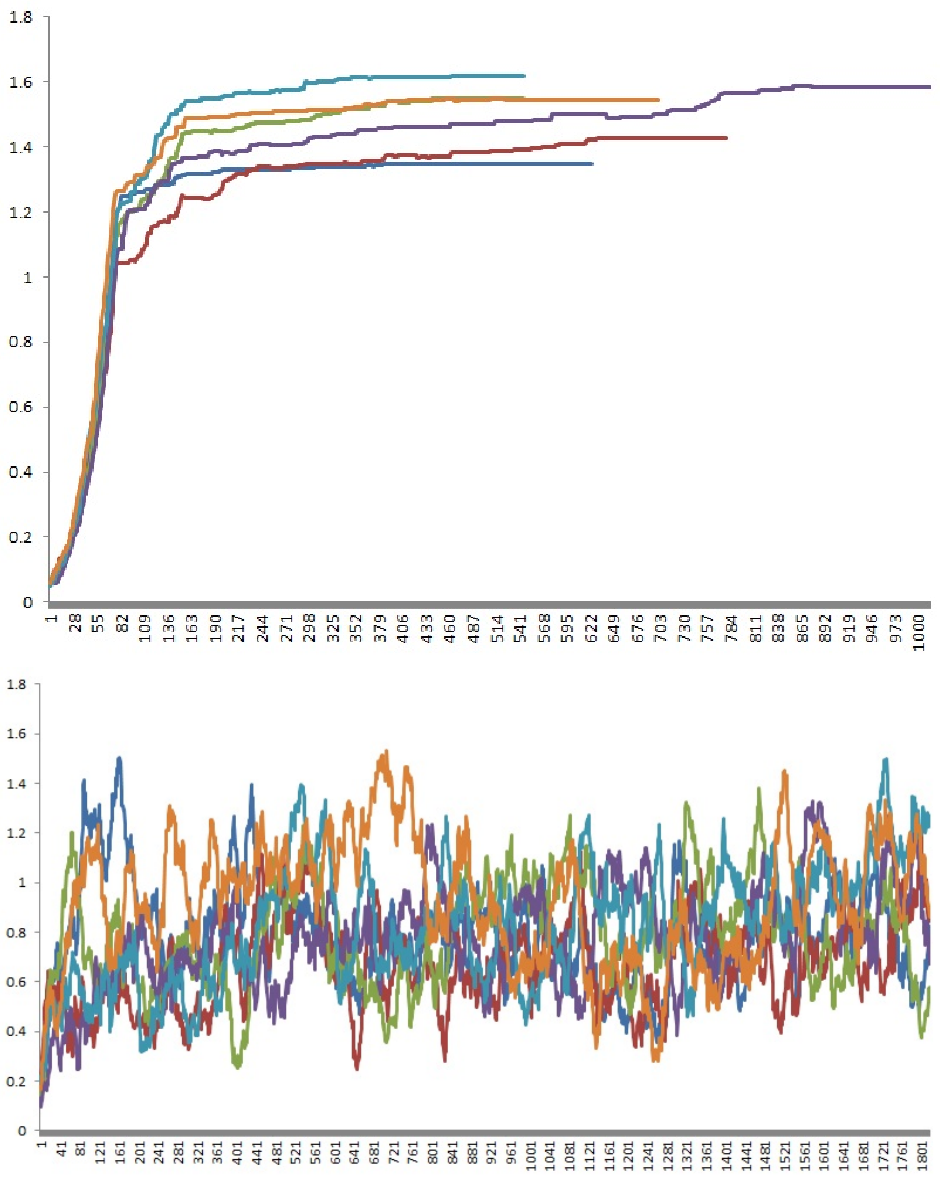


Figure 1.1: Proposal 1 versus Proposal 2 on a short motif

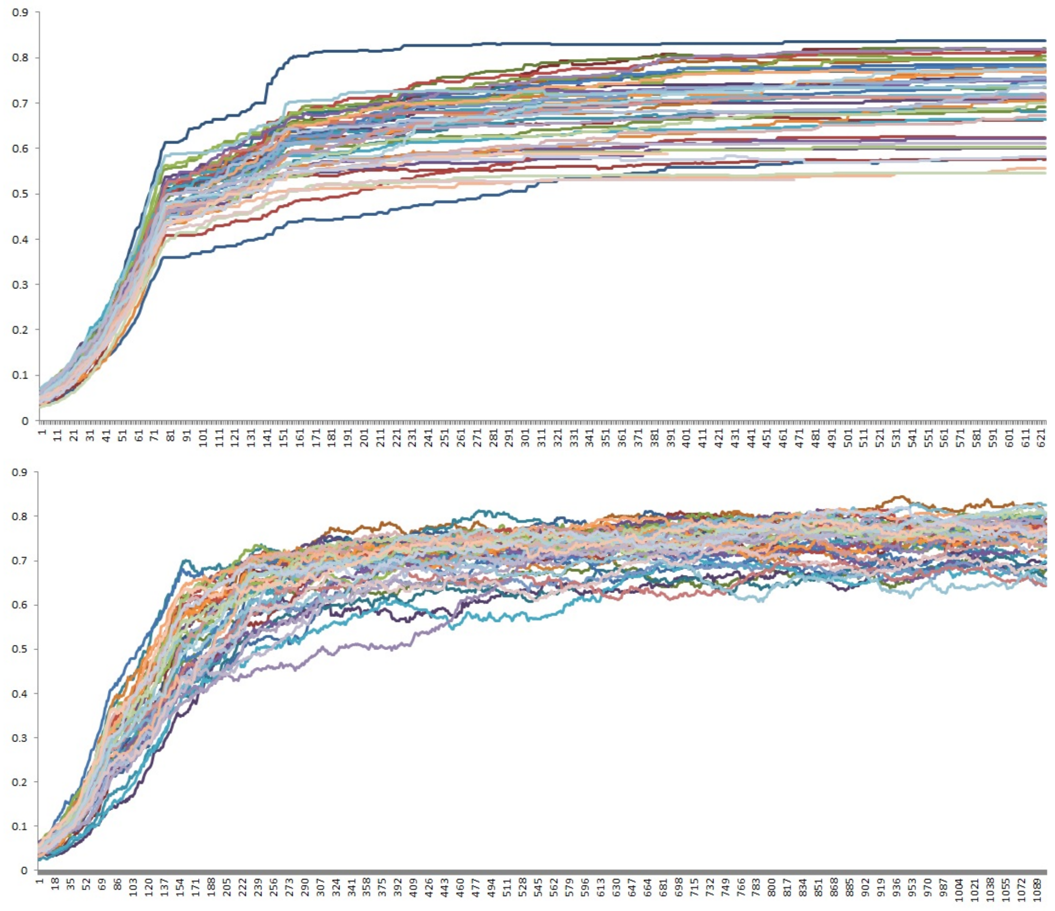


Figure 1.2: Proposal 1 versus Proposal 2 on a long motif

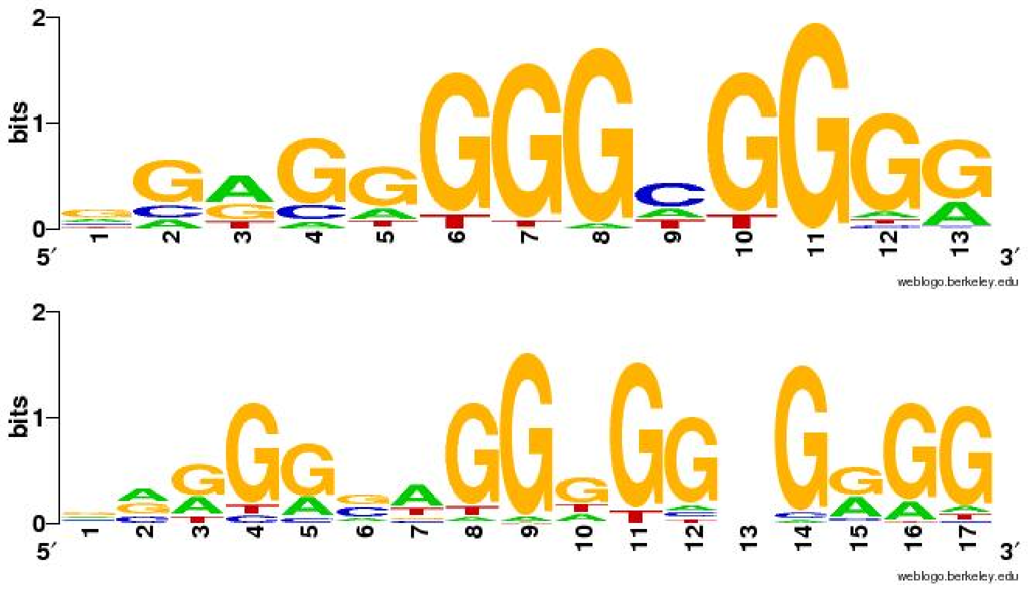


Figure 1.3: Proposal 1 motif result for $k = 13$ and $k = 17$ from dataset M00931

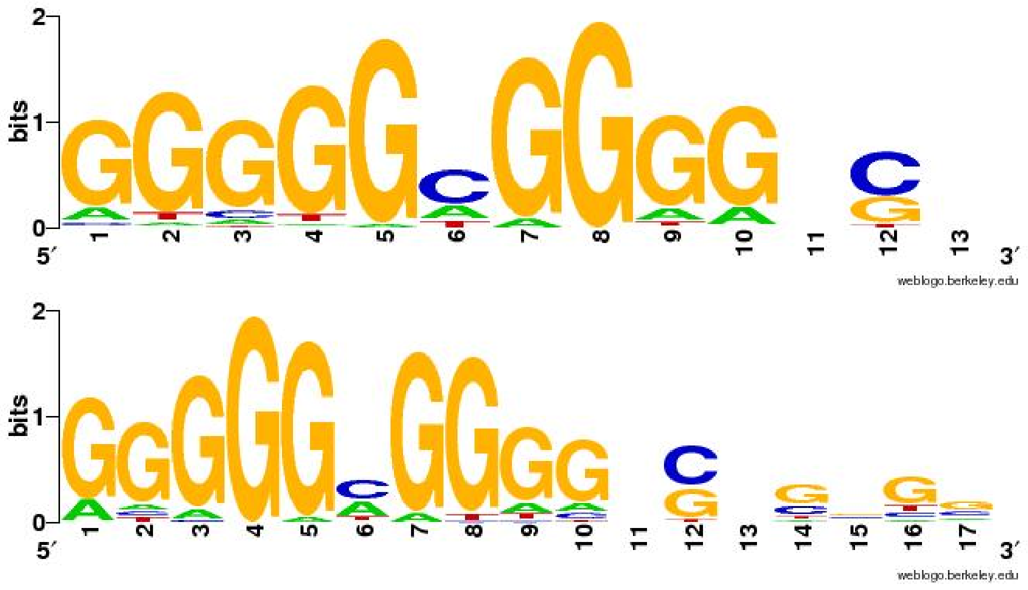


Figure 1.4: Proposal 2 motif result for $k = 13$ and $k = 17$ from dataset M00931

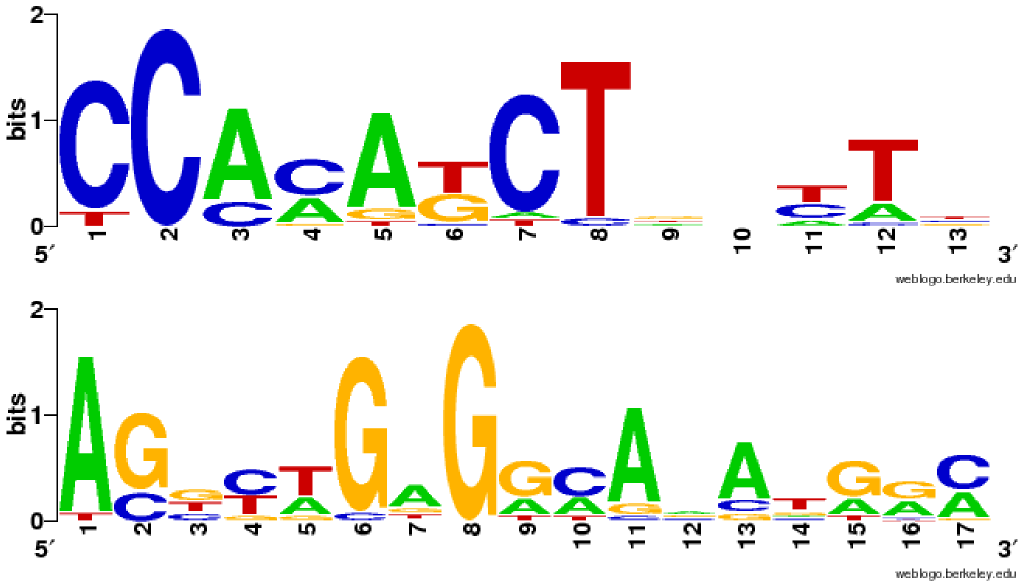


Figure 1.5: Proposal 1 motif result for $k = 13$ and $k = 17$ from dataset M00971

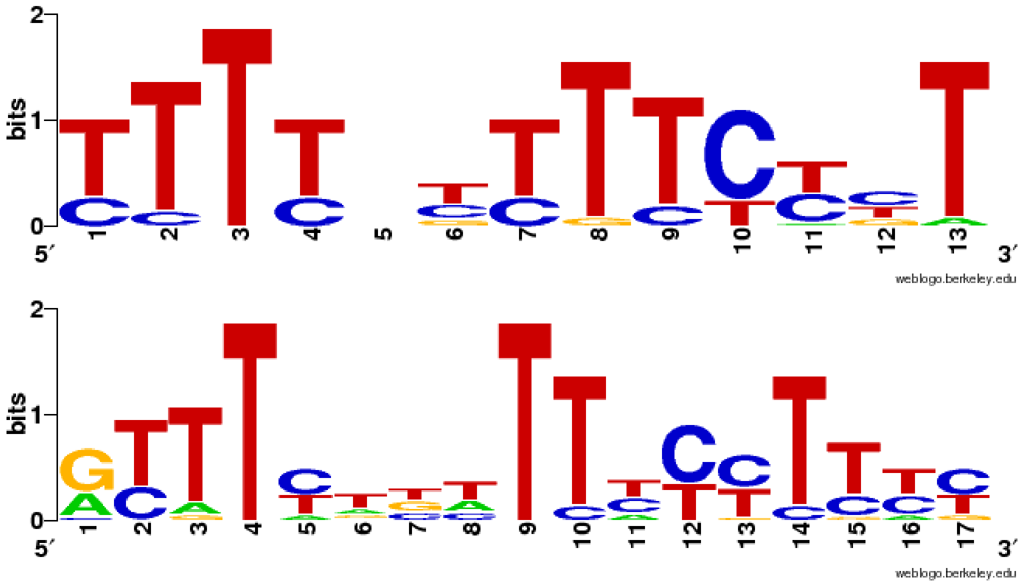


Figure 1.6: Proposal 2 motif result for $k = 13$ and $k = 17$ from dataset M00971

Chapter 2

Using Hill Climbing and Simulated Annealing for Travelling Salesman Problem

2.1 Abstract

We experimented with solving the *Traveling Salesman Problem* (TSP) using two methods: *Hill Climbing* and *Simulated Annealing.* We improved Hill Climbing slightly by adding random restarts and testing two different methods of generating candidate moves: swaps and flips. We also employed these candidate methods with Simulated Annealing as well as tweaking the cooling rate. Both methods produced results close to optimum rapidly, though Simulated Annealing ultimately did slightly better due to its stochastic nature as opposed to Hill Climbing's greedy algorithm. In addition, the flip method (reversing strings) of generating candidate moves produced slightly better results.

2.2 Introduction

The TSP is a well-known classical problem in computer science. Given a set of cities and the distances between every (or some) pairs of them, a salesman needs to start from a particular city, visit all other cities and return the initial city at the end by the shortest possible tour. While this concept seems relatively mundane, it has a number of real world applications in such as simplifying the construction of microchips or (when modified slightly) figuring out the shortest distance between two sequences of DNA for gene recognition.

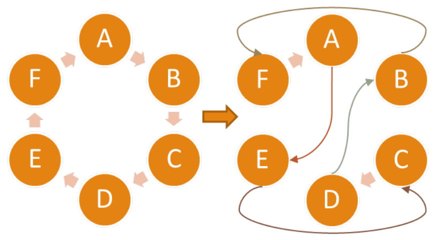
The TSP is better known in mathematics as a form of the Hamiltonian cycle. From a computer science perspective, it is a NP-hard problem: the only known method of producing provably optimal solutions is by enumerating every possible path and selecting the shortest one. This brute force approach has factorial complexity and can only be applied for very small datasets: as the number of cities increases the problem quickly becomes intractable. As such, simpler methods of attacking the problem are desired. We tested two of them for our project.

Hill Climbing is one of the classical greedy algorithms. Simply put, we begin with some starting state (in this case, a tour) and then examine the local neighborhood of this state space (generate a slightly different tour and calculate its length). We can either find the most directly uphill direction (steepest gradient) or simply take any direction that will produce improvement (first best/random choice) and take a step in that direction. Then, we repeat this cycle until we can no longer improve our solution. At the end, we are at the top of the hill and declare the optimal solution. If the objective function is convex (it has only one optimum), Hill Climbing is guaranteed to give the global optima. However, in the real world example of the TSP, this condition does not hold so Hill Climbing will only find a local optima.

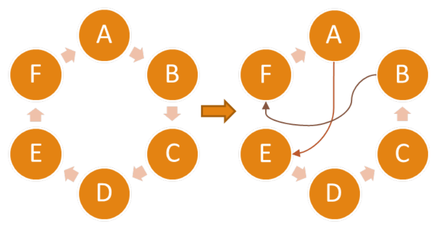
Simulated Annealing is another classical algorithm exploiting stochastic mechanism. To avoid getting stuck into a local optimum, Simulated Annealing uses the Boltzmann model – a cooling schedule to gradually reduce the temperature – to give an opportunity to escape local optima by accepting a worse state with probability depending on both the state value difference and temperature. Unlike Hill Climbing, which always chooses a better state at each step, Simulated Annealing may choose a sub-optimal state with the hope that from this state, it can climb to another optimal state.

In addition to the above methods, we tested two methods of generating candidate moves, Two-Swap and Two-Flip. In the first, we selected two cities at random and swapped their places in the tour. This in turn altered four paths (Fig.2.1a) and produced results quicker but less stable (it produced more change, but this was ultimately less useful). The second method we tested was also to select two cities at random as above but then reverse their order in the tour instead (Fig.2.1b). For example, the tour ABCDEFA, upon selecting cities B and E, would end up as AECDBFA in Two-Swap and AEDCBFA in Two-Flip. The Two-Flip only changes two edges and hence is more stable than Two-Swap.

We also implemented two minor modifications to the above methods. First, we ran many Hill Climbing algorithms with different random initial states and selected the best tour among all results. Likewise, we also experimented with tweaking the cooling rate during the Simulated Annealing process, which affects the algorithm significantly.



(a) Two-Swap Move



(b) Two-Flip Move

Figure 2.1: Candidate move generation

2.3 Move Set

Algorithm 2.1 Generate neighbors

```
function GENERATE-MOVE($(x_1, \dots, x_N), a, b, flip$)
    if $flip = 0$ then
        return TWO-SWAP($(x_1, \dots, x_N), a, b$)
    else
        return TWO-FLIP($(x_1, \dots, x_N), a, b$)
    end if
end function

function TWO-SWAP($X, a, b$)
    $X' \leftarrow X$
    $X'(a) \leftarrow X(b)$
    $X'(b) \leftarrow X(a)$
    return $X'$
end function

function TWO-FLIP($X, a, b$)
    $X' \leftarrow X$
    $N \leftarrow \text{length}(X)$
    $i \leftarrow a$
    while $i \neq b$ do
        $X'(i) \leftarrow X((a + b - i + N) \bmod N + 1)$
        $i \leftarrow i + 1$
        if $i > N$ then
            $i \leftarrow 1$
        end if
    end while
    $X'(b) \leftarrow X(a)$
    return $X'$
end function
```

Algorithm 2.1 describes the two move generation methods we mentioned earlier. Note that in our implementation, we only need to represent a tour as a sequence $(x_1, \dots, x_N)$ because we just simply duplicate x_1 at the end to form a tour. Here we see that Two-Swap is simpler but it introduces more changes and thus less stable. Two-Flip, on the other hand, is a little more complicated but more stable because it only changes two edges.

2.4 Hill Climbing

The Hill Climbing algorithm is described in algorithm 2.3. In step 1, we randomly initialize a tour. Because the distance matrix is fully connected, we can simply generate a random permutation of N cities. Next, in the loop at line 6, we try to explore the neighborhood of the current state by picking randomly K neighbors. The reason for this is because we cannot try all the neighbors (i.e. choose all combinations of two cities to do the swap or

Algorithm 2.2 Calculate tour length

```
function TOUR-LENGTH($(x_1, \dots, x_N)$)
    sum $\leftarrow D(x_N, x_1)$
    for $i \leftarrow 2$ to $N$ do
        sum $\leftarrow$ sum $+ D(x_{i-1}, x_i)$
    end for
    return sum
end function
```

Algorithm 2.3 Hill Climbing for Travelling Salesman Problem

Input:

- N, the number of cities
- $D(i, j)$, the pairwise distance between i^{th} city and j^{th} city, $1 \leq i, j \leq N$
- Parameters:
  - R, the range of interchange, $1 \leq R \leq N$
  - K, the number of neighbors generated at each state, $1 \leq K \leq \binom{N}{2}$
  - $flip \in \{0, 1\}$, 2-swap move $flip = 0$ or 2-flip move $flip = 1$

Output: The best tour $\hat{X} = (\hat{x}_1, \hat{x}_2, \dots, \hat{x}_N)$ with length $\hat{L}$

```
Randomly initialize a tour $X = (x_1, x_2, \dots, x_N)$
$L \leftarrow$ TOUR-LENGTH($X$)
$(\hat{X}, \hat{L}) \leftarrow (X, L)$                                  ▷ Optimal tour
repeat
    $flag \leftarrow true$
    for $k \leftarrow 1$ to $K$ do                                       ▷ Try to explore $K$ neighbors
        $a \leftarrow$ RANDOM($1, N$)                                    ▷ Randomly choose one city
        $b \leftarrow$ RANDOM($a + 1, a + R$) mod $N + 1$
                              ▷ Randomly choose another city within $R$ in current tour
        $X' \leftarrow$ GENERATE-MOVE($X, a, b, flip$)                   ▷ Generate a move, 2-swap or 2-flip
        $L' \leftarrow$ TOUR-LENGTH($X'$)                                ▷ Length of new tour
        if $L' < \hat{L}$ then                                           ▷ Minimize optimal tour
            $(\hat{X}, \hat{L}) \leftarrow (X', L')$
            $flag \leftarrow false$
        end if
    end for
    $(X, L) \leftarrow (\hat{X}, \hat{L})$
until $flag = true$
```

the flip). Line 11 optimizes the new state versus the current optimal state. The algorithm repeats until we cannot find any better tour from the current state's neighbors. From this scheme, the major handicap of the algorithm's success (find the global optimum) is at line 1, the initial random state. Therefore, we need to run the algorithm many times to "climb the hill" from different places.

2.5 Simulated Annealing

Algorithm 2.4 sketches the outline of the Simulated Annealing. Starting at a random tour at line 1, we just randomly pick one neighbor. If the neighbor tour is better, we accept it. Otherwise, we will accept it with the probability $\exp\left(-\frac{\Delta L}{T}\right)$, where $\Delta L = L' - L$. The expression in 19 puts both quality of new tour compared to the current tour and temperature into consideration. The temperature T keeps decreasing exponentially for each iteration as in line 6. The algorithm keeps track of the best tour obtained so far and reports it after exceeding a preset number of iterations. The Simulated Annealing is less sensitive to initial tour than the Hill Climbing, but it requires a careful choice of initial temperature T_0 and the cooling rate r. Therefore, in our implementation, we tried different cooling rates and also different initial tours.

2.6 Results and Visualization

The dataset consists of the 15 cities and 57 cities dataset, which is available at `http://people.sc.fsu.edu/~jburkardt/datasets/cities/cities.html`. We also implemented a graphical program using `FLTK` and `OpenGL` to visualize the result. Fig.2.2 shows our Hill Climbing result on 15 cities-dataset, using the Two-Swap move candidate generator. The upper window on the left and on the right are respectively the current tour and the optimal tour retuned by the algorithm. The lower window displays how the tour length changes during each iteration of the algorithm. The program visualizes the result step by step and Fig.2.2 just the initial step. Thus the left window is the initial tour. We mark changed edges by yellow color and keep other edges' colors unchanged (red). Due to space limits, we only display some of our results, but all the results can be reproduced easily by running our algorithm code to generate the output file and using the visualization program to load the output file and display the result.

Fig.2.4 and 2.5 respectively show the result of the Simulated Annealing with Two-Swap and Two-Flip. Figures from 2.6 to 2.9 show similar results of two algorithms and two move generations on 57 cities-dataset.

2.7 Conclusion

Table 2.1 summaries our results.

Algorithm 2.4 Simulated Annealing for Travelling Salesman Problem

Input:

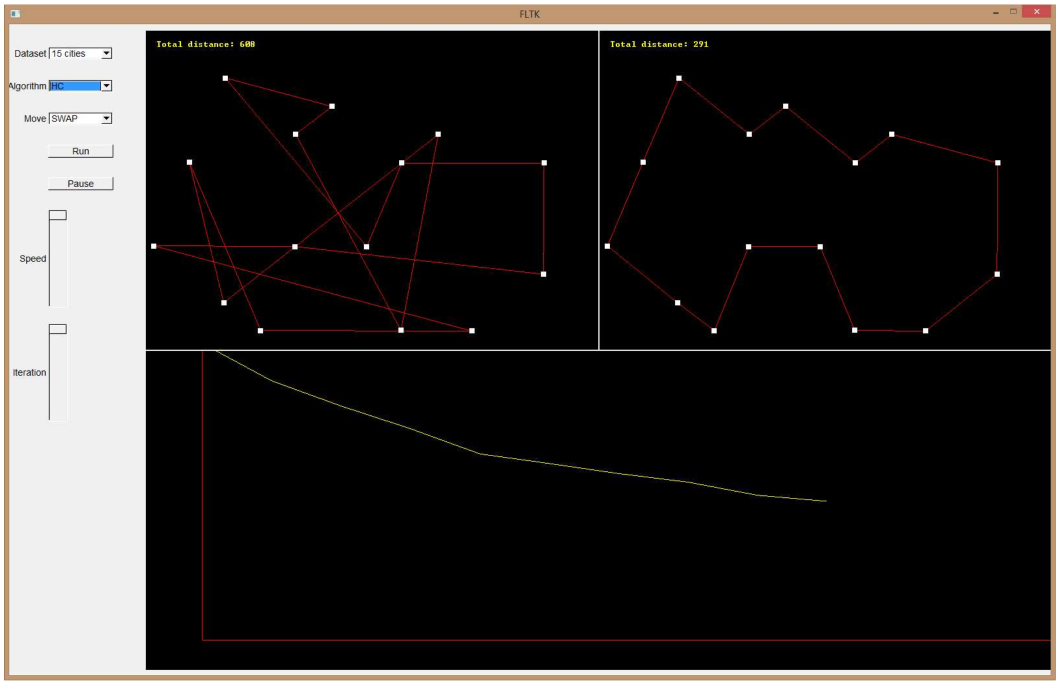
- N, the number of cities
- $D(i, j)$, the pairwise distance between i^{th} city and j^{th} city, $1 \le i, j \le N$
- Parameters:
    - R, the range of interchange, $1 \le R \le N$
    - $flip \in \{0, 1\}$, 2-swap move $flip = 0$ or 2-flip move $flip = 1$
    - T_0, initial temperature
    - r, cooling rate, $r \in (0, 1)$
    - $itermax$, the maximum number of iterations

Output: The best tour $\hat{X} = (\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_N)$ with length $\hat{L}$

```
1: Randomly initialize a tour $X = (x_1, x_2, \ldots, x_N)$
2: $L \leftarrow$ Tour-Length($X$)
3: $(\hat{X}, \hat{L}) \leftarrow (X, L)$                                  ▷ Optimal tour
4: $i \leftarrow 0$                                                        ▷ Current iteration
5: repeat
6:     $T \leftarrow T_0 \times r^i$
7:     $i \leftarrow i + 1$
8:     $a \leftarrow$ Random($1, N$)                                       ▷ Randomly choose one city
9:     $b \leftarrow$ Random($a + 1, a + R$) mod $N + 1$
                                   ▷ Randomly choose another city within $R$ in current tour
10:    $X' \leftarrow$ Generate-Move($X, a, b, flip$)                      ▷ Generate a move, 2-swap or 2-flip
11:    $L' \leftarrow$ Tour-Length($X'$)                                   ▷ Length of new tour
12:    if $L' < L$ then
13:        $(X, L) \leftarrow (X', L')$
14:        if $L < \hat{L}$ then                                           ▷ Minimize optimal tour
15:            $(\hat{X}, \hat{L}) \leftarrow (X, L)$
16:        end if
17:    else
18:        $u \leftarrow$ Random-Uniform($0, 1$)
19:        if $u \le \exp\left(-\frac{L' - L}{T}\right)$ then              ▷ Accept a worse tour with some probability
20:            $(X, L) \leftarrow (X', L')$
21:        end if
22:    end if
23: until $i > itermax$
```

Table 2.1: Tour Length Result Summary

Method		**15 cities**	**57 cities**
Optimal Tour		291	*Unknown*
Hill Climbing	Two-Swap	291	15,157
	Two-Flip	291	13,399
Simulated Annealing	Two-Swap	291	13,690
	Two-Flip	291	13,059

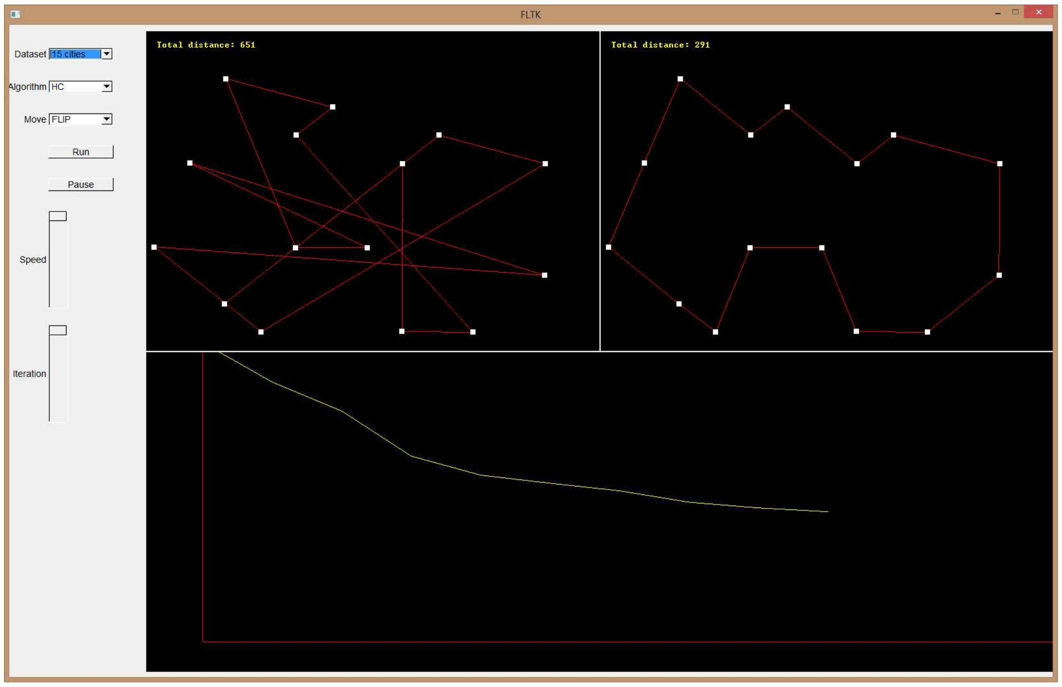


Dataset: 15 cities
Algorithm: HC
Move: SWAP

```
Total distance: 608
```

```
Total distance: 291
```

Figure 2.2: Hill Climbing using Two-Swap on 15 cities-dataset

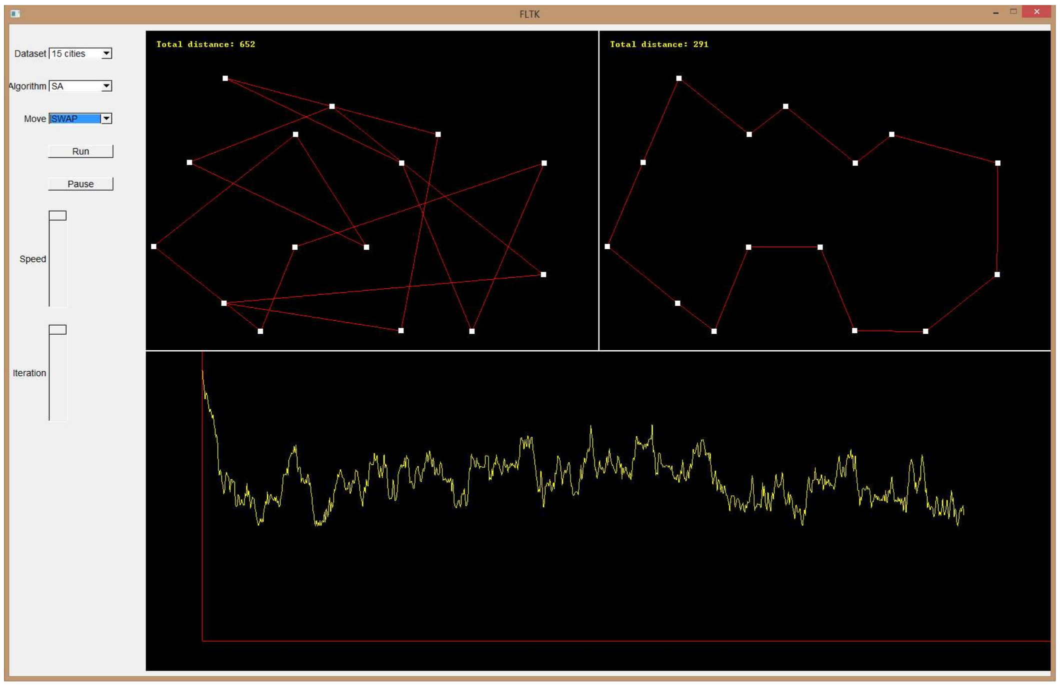


Dataset 15 cities
Algorithm HC
Move FLIP

Total distance: 651

Total distance: 291

Figure 2.3: Hill Climbing using Two-Flip on 15 cities-dataset



Dataset 15 cities
Algorithm SA
Move SWAP

Total distance: 652

Total distance: 291

Figure 2.4: Simulated Annealing using Two-Swap on 15 cities-dataset

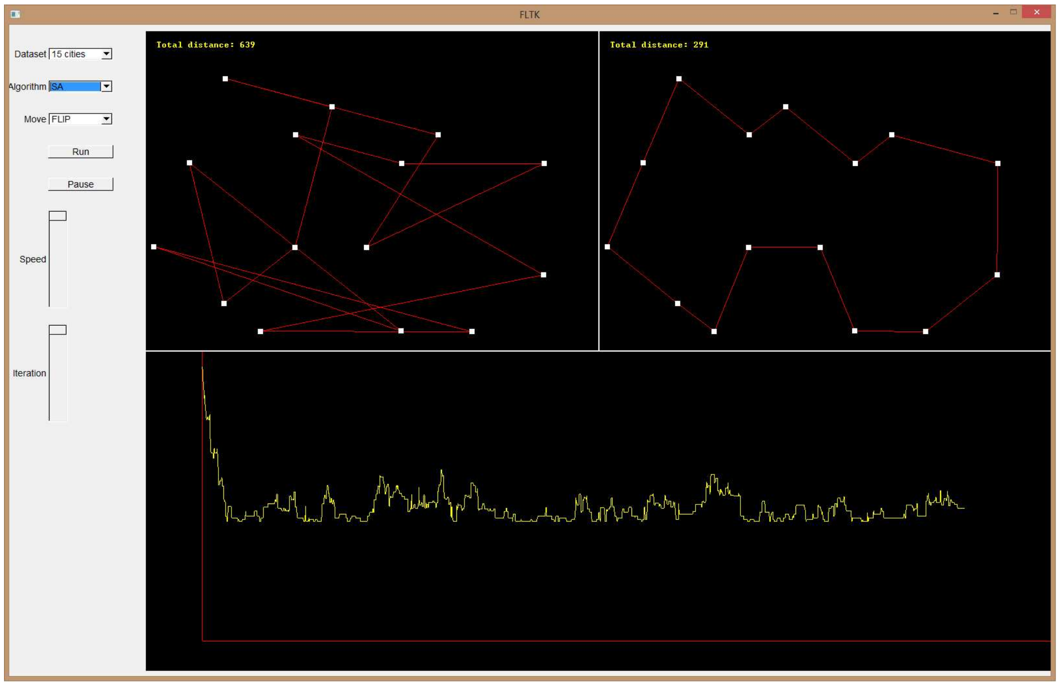


Figure 2.5: Simulated Annealing using Two-Flip on 15 cities-dataset

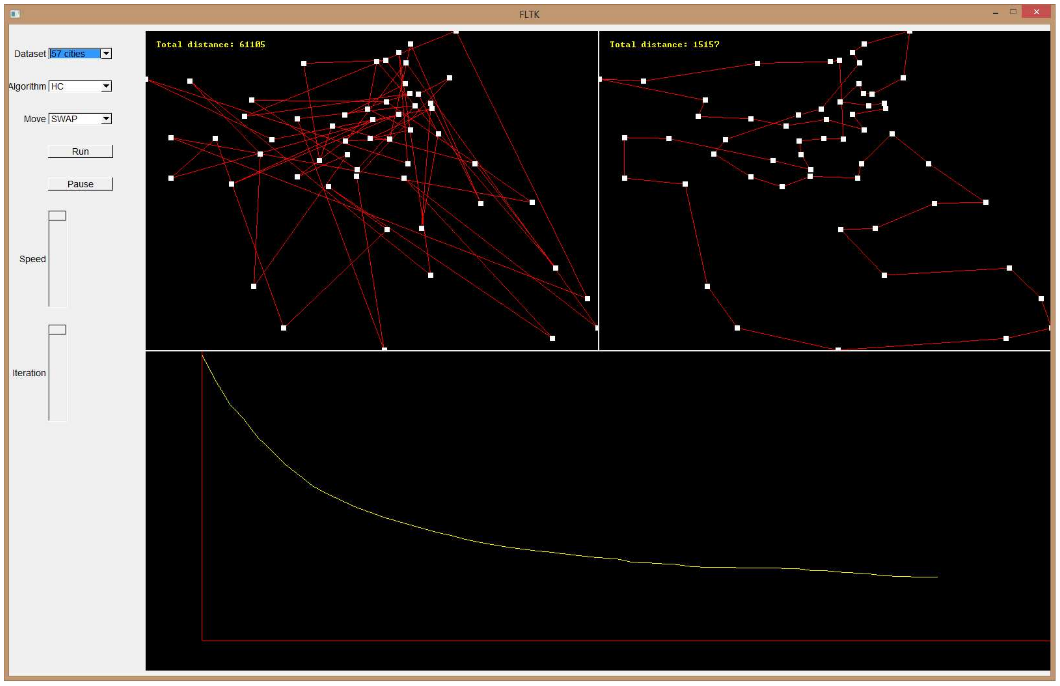
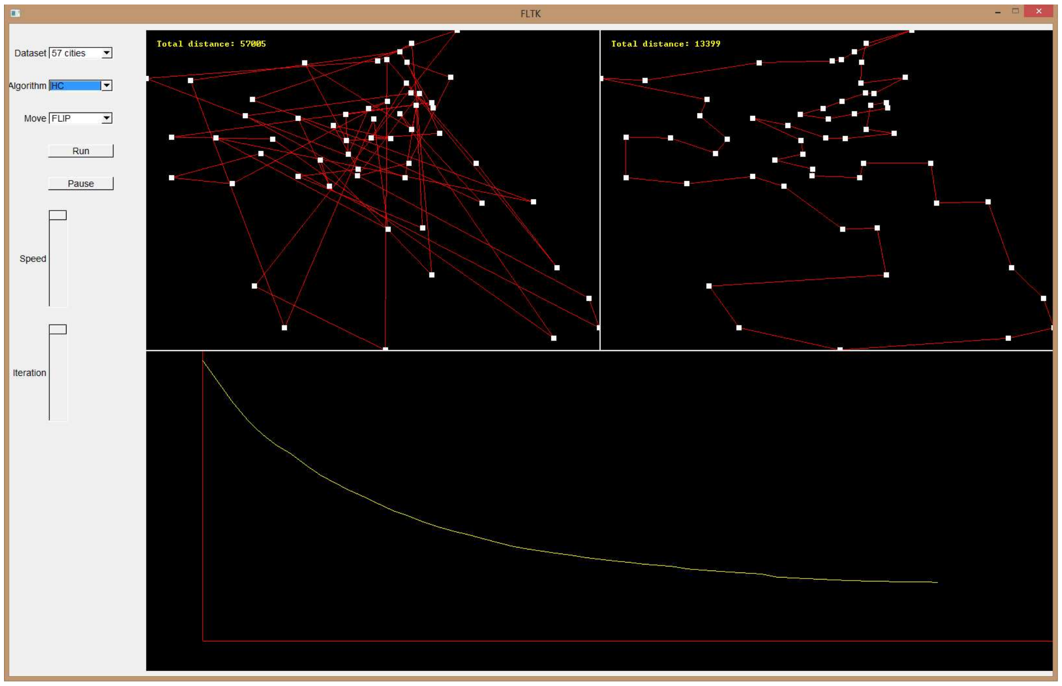
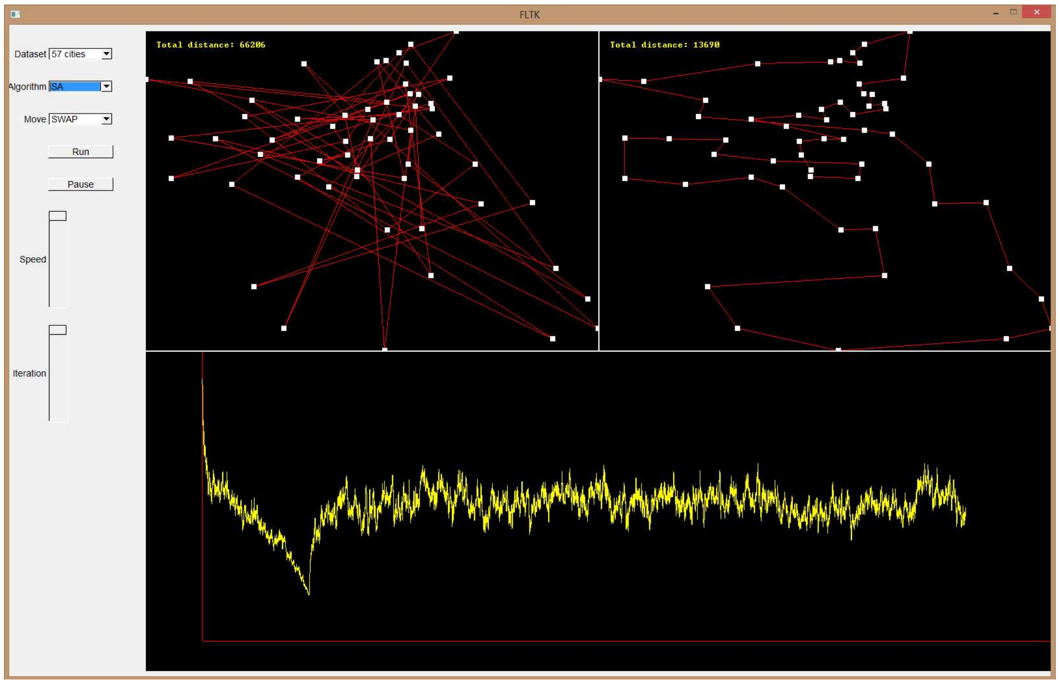


Figure 2.6: Hill Climbing using Two-Swap on 57 cities-dataset



Dataset 57 cities
Algorithm HC
Move FLIP
Total distance: 57005
Total distance: 13399

Figure 2.7: Hill Climbing using Two-Flip on 57 cities-dataset



Dataset 57 cities
Algorithm SA
Move SWAP
Total distance: 66206
Total distance: 13690

Figure 2.8: Simulated Annealing using Two-Swap on 57 cities-dataset

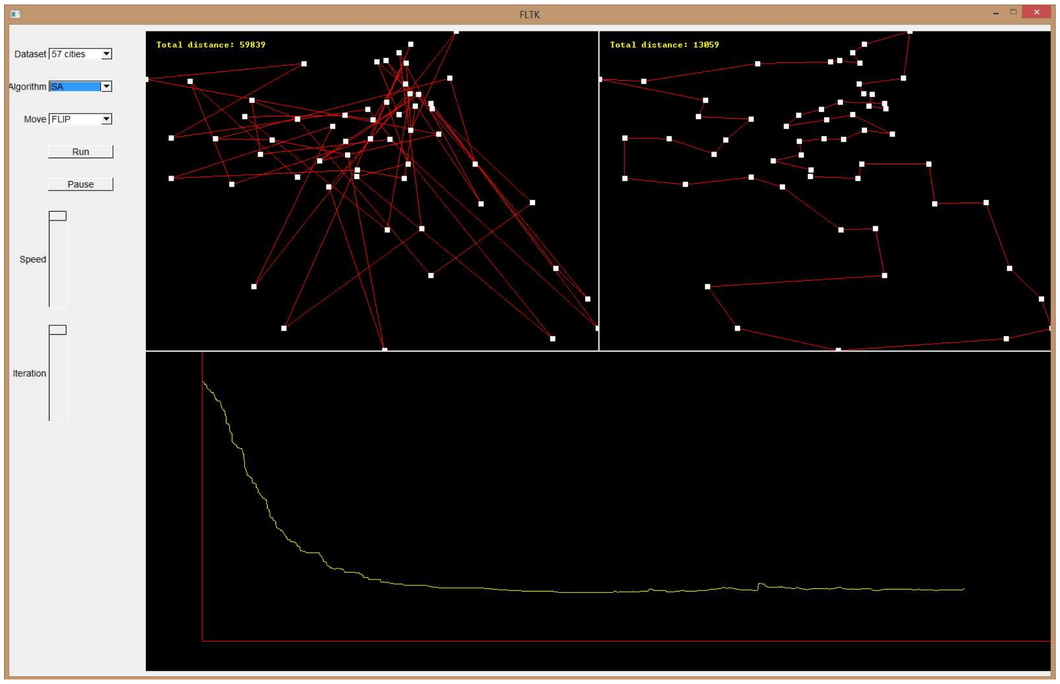


Figure 2.9: Simulated Annealing using Two-Flip on 57 cities-dataset

Chapter 3

Sequence Alignment by Dynamic Programming

3.1 Abstract

We implemented the Needleman-Wunsch algorithm to find the similarity of two protein sequences. We tested two methods, global and local alignment and found good results with both (the methods should differ only in complexity). Finally, we implemented a visualizer for the two aligned sequences as well as a MATLAB representation of the updates of the scoring matrix and demonstrate them with our test data.

3.2 Introduction

3.2.1 Dynamic Programming

Dynamic programming is a method by which a larger problem is solved by first solving smaller, partial versions of the problem. By gradually increasing the scope of the initial starting point it builds the solution bottom-up. As such, dynamic programming has some similarity with the *Divide and Conquer* paradigm. However, dynamic programming is different from this approach in that in divide and conquer, the sub-problems are disjoint or separable so they can be solved in parallel without knowing the global state. Whereas, in dynamic programming the sub-problems overlap each other and need to be solved sequentially in a particular order. As such, the two most basic questions in dynamic programming are how to divide the problem into sub-problems and then how to solve them. By breaking the problem up correctly and combining the solutions of these overlapping smaller sub-problems, the solution of the larger problem can be found and guaranteed to be optimal.

In bioinformatics, dynamic programming is commonly used for several tasks such as sequence alignment, protein folding, RNA structure prediction, etc. It is this first case that we investigate in this report.

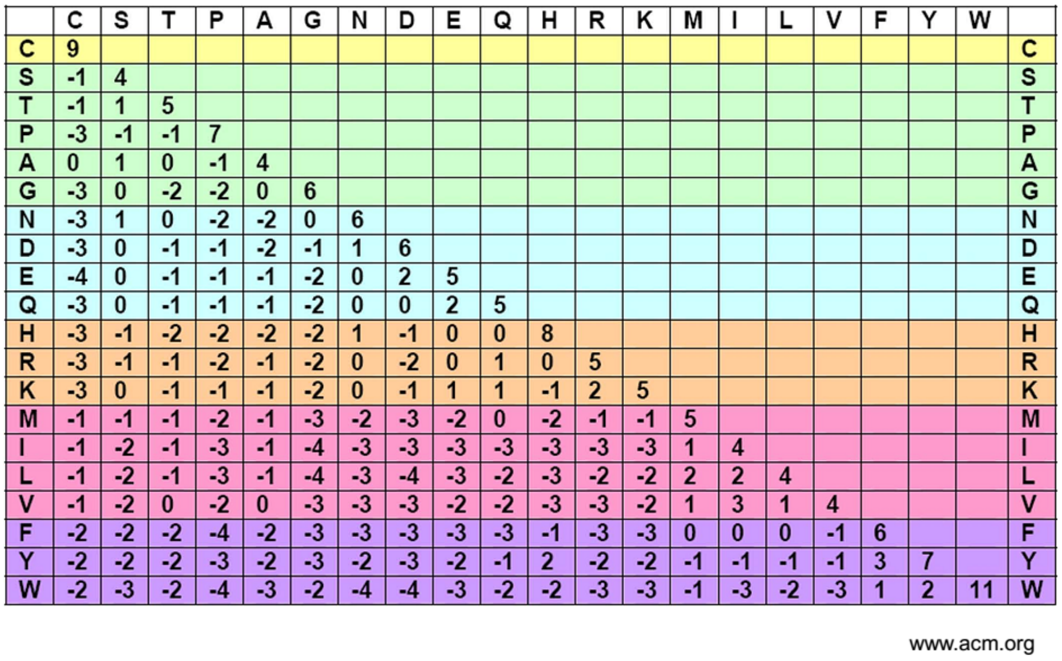
3.2.2 Sequence Alignment

Sequence alignment arises in many fields, like molecular biology, inexact text matching and speech recognition. In molecular biology, it is a way of arranging protein, DNA or RNA to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences.

Aligning protein sequences allows us to compare a set of sequences based on their distance, giving information about their possible relationships, similarities and differences. While a common dynamic programming approach for sequence alignment is based on edit distance, in which two strings are compared by measuring the number of edits needed to reach parity, we use an opposite concept – *similarity* of two sequences, and as such we need to find an optimal alignment that maximizes the similarity score.

3.3 Sequence Alignment Algorithm

3.3.1 Preliminary



	C	S	T	P	A	G	N	D	E	Q	H	R	K	M	I	L	V	F	Y	W	
C	9																				C
S	-1	4																			S
T	-1	1	5																		T
P	-3	-1	-1	7																	P
A	0	1	0	-1	4																A
G	-3	0	-2	-2	0	6															G
N	-3	1	0	-2	-2	0	6														N
D	-3	0	-1	-1	-2	-1	1	6													D
E	-4	0	-1	-1	-1	-2	0	2	5												E
Q	-3	0	-1	-1	-1	-2	0	0	2	5											Q
H	-3	-1	-2	-2	-2	-2	1	-1	0	0	8										H
R	-3	-1	-1	-2	-1	-2	0	-2	0	1	0	5									R
K	-3	0	-1	-1	-1	-2	0	-1	1	1	-1	2	5								K
M	-1	-1	-1	-2	-1	-3	-2	-3	-2	0	-2	-1	-1	5							M
I	-1	-2	-1	-3	-1	-4	-3	-3	-3	-3	-3	-3	-3	1	4						I
L	-1	-2	-1	-3	-1	-4	-3	-4	-3	-2	-3	-2	-2	2	2	4					L
V	-1	-2	0	-2	0	-3	-3	-3	-2	-2	-3	-3	-2	1	3	1	4				V
F	-2	-2	-2	-4	-2	-3	-3	-3	-3	-3	-1	-3	-3	0	0	0	-1	6			F
Y	-2	-2	-2	-3	-2	-3	-2	-3	-2	-1	2	-2	-2	-1	-1	-1	-1	3	7		Y
W	-2	-3	-2	-4	-3	-2	-4	-4	-3	-2	-2	-3	-3	-1	-3	-2	-3	1	2	11	W

Figure 3.1: An amino acid scoring matrix: BLOSUM62

Definition 1. Assume we are working on 1-based index system.

- X and Y are two that sequences need aligning. They have length m and n respectively.
- X_i – the i^{th} character of sequence X $(1 \le i \le m)$

- Y_j – the j^{th} character of sequence Y $(1 \leq j \leq n)$
- $s(X_i, Y_j)$ – the score for aligning character X_i with Y_j
- *gap* – the gap score for aligning any character to a *null*

Definition 1 gives us everything for presenting our algorithm, except the score element $s(X_i, Y_j)$. In normal case, one would simply assign $s(X_i, Y_j)$ to some positive number if $X_i = Y_j$, otherwise set $s(X_i, Y_j)$ to 0 or some negative number for a penalty if needed. Nevertheless, in bioinformatics in general and amino-acids in particular, people prefer to use the BLOSUM (BLOcks SUbstitution Matrix) matrix to represent the score of aligning amino-acids. The matrix construction is based on evolutionary divergence of protein sequences whose details are out of scope and will not be discussed here. Fig.3.1 shows the BLOSUM62 matrix, a miscalculated version of original BLOSUM matrix that surprisingly yields better search performance. Due to symmetry, it suffices to display the lower triangle of the matrix. Therefore, for now, we define $s(X_i, Y_j) = B(X_i, Y_j)$, where B is the BLOSUM62 matrix.

Let $\hat{X}$ and $\hat{Y}$ be an alignment ($\hat{X}$ and $\hat{Y}$ must have the same length). The score for this alignment is calculated as following.

$$\text{Score}(\hat{X}, \hat{Y}) = \sum_i s(\hat{X}_i, \hat{Y}_i) \tag{3.1}$$

In [3.1], we have extended the definition of $s(x, y)$ to allow either x or y to be *null*. Based on applications, we have *global alignment* and *local alignment*. Global alignments, which attempt to align every residue in every sequence, are most useful when the sequences in the query set are similar and of roughly equal size (this does not mean global alignments cannot end in gaps). Local alignments are more useful for dissimilar sequences that are suspected to contain regions of similarity or similar sequence motifs within their larger sequence context.

3.3.2 Global Alignment

Definition 2. Let the scoring matrix S be an $m \times n$ matrix and $S(i, j)$ be the maximum score of aligning the sub-sequences $X_{1..i}$ with $Y_{1..j}$, then $S(i, j)$ can be computed by the following recurrence.

$$S(i,j) = \max \begin{cases} S(i-1, j-1) + s(X_i, Y_j) & \text{if align } X_i \text{ to } Y_j \\ S(i-1, j) + gap(X_i, null) & \text{if align } X_i \text{ to } null \\ S(i, j-1) + gap(null, Y_j) & \text{if align } null \text{ to } Y_j \end{cases} \tag{3.2}$$

To understand how this works, let us have a look at the last characters of $X_{1..i}$ and $Y_{1..j}$. Fig.3.2 demonstrates an example of this scenario. To align these two sub-sequences, there are only three possibilities:

1. X_i is aligned with Y_j, which tells that the score for alignment these two sub-sequences is the score of aligning the sub-sequence $X_{1..(i-1)}$ with $Y_{1..(j-1)}$ plus the score of aligning X_i and Y_j.
2. X_i is aligned with *null*, the score for this case equals the score of aligning $X_{1..(i-1)}$ with $Y_{1..j}$ plus the score of aligning X_i with *null*.

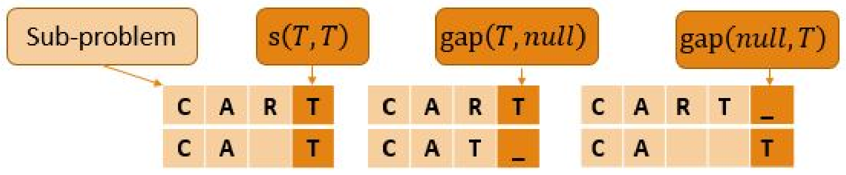


Figure 3.2: Sub-problems

3. *null* is aligned with Y_j, the score for this case equals the score of aligning $X_{1..i}$ with $Y_{1..(j-1)}$ plus the score of aligning *null* with Y_j.

To optimize $S(i,j)$, we simply choose the maximum value among the three cases. The solution of our original problem is $S(m,n)$. In order to obtain the aligned sequences, we need to *trace back* the scoring matrix S. The trace begins that the bottom right corner of S and stops when reaching the top left corner. At each step, we trace from $S(i,j)$ to one of the three directions (diagonal, up, left) depending on which direction optimizes $S(i,j)$ and at the same time construct the aligned sequences accordingly. Pseudo-code of the process is presented in algorithm 3.1.

3.3.3 Local Alignment

The local sequence alignment problem, in fact, can be solved by some modifications of the preceding algorithm. The first change is the definition of the scoring matrix.

Definition 3. Let the scoring matrix S be an $m \times n$ matrix and $S(i,j)$ be the maximum score of *locally* aligning the sub-sequences $X_{1..i}$ with $Y_{1..j}$, then $S(i,j)$ can be computed by the following recurrence.

$$S(i,j) = \max \begin{cases} 0 & \text{No alignment} \\ S(i-1,j-1) + s(X_i,Y_j) & \text{if align } X_i \text{ to } Y_j \\ S(i-1,j) + gap(X_i,null) & \text{if align } X_i \text{ to } null \\ S(i,j-1) + gap(null,Y_j) & \text{if align } null \text{ to } Y_j \end{cases} \tag{3.3}$$

Adding 0 to the maximization of $S(i,j)$ allows $X_{1..i}$ and $Y_{1..j}$ to be the prefix of another local alignment. The second modification is that the solution is no longer the bottom right corner of S, but the largest element of it. Furthermore, when tracing back starting at the position of the maximum value, we should stop whenever reaching element 0. Algorithm 3.2 describes exactly what we use for local alignment.

Note that based on this scoring matrix, we can find as many local optimal alignments as we can. After the best local alignment is detected, the scoring matrix is divided into 9 regions. Excluding the region defined by the best local alignment, the remaining local alignments are among 8 regions. Since we do not want overlapping local alignments, these regions reduce to 4. Intuitively, the best local alignment leave the original two sequences into two disjoint sub-sequences. The second best local alignment must be in these four cases corresponding to 4 regions above. Thus, we can find the maximum element of the scoring matrix in these 4 regions and trace back by applying the same method when detecting the best one. Similarly, we can find as many local alignments as needed. In our implementation, we find the best and the second best alignments.

3.4 Results and Visualization

We tested our algorithms on the following two sequences.

```
MVSQRQRLARKRYKAEHPELFPKPEPTPPKDPEKKKKKKKNSAFKRKRPEPKPGSRKRHPLRVPGMKPGESCFIC
KAMDHIAKLCPEKAEWEKNKICLRCRRRGHRAKNCPEVLDGAKDAMYCYNCGENGHSLTQCPHPLQEGGTKFAEC
FVCNQRGHLSKNCPQNTHGIYPKGGCCKICGGVTHLAKDCPDKGKSGSVAANRPADGWMRIEERPMGQVTKFVSG
DDIEDDFMTDDIHSGDKKKPAKSTEDHVKPKKKEGPKVVNF
QVKKHSKHCRTCNRCVEGFDHHCRWLNNCVGKRNYTAFFLLMIFLLIKGGTAIAIFIRCFVDRRGIEKELQRKLY
VDFPRGVLATICVFLLLLTAYSSAALGQLFFFHVVLIRKTNTHAIKSIIISLRKTYDYILAMKEENEAMELESFD
DSDFSSDESFDFDSPEKPTLMSGFLCKGNQGKKALLAAEKARERIMREKPMGEHNSLKPLPLETKCGPLMNTYKN
MDTEDFGSTSFIAKGRLNESPGRFSSPRRRFSAGSPTVFSSSMMASPHHKYRSSFDLKLTGVSRELETHISRQVL
CSVISKDDSEPSPR
```

Fig.3.3 shows the plot of global alignment scoring matrix. Its values vary from the minimum value (blue part) to the maximum value (red part). The white path records the tracing path from (m, n) to $(0, 0)$. The resulting aligned sequences with the maximum global alignment score of -370 are as following.

```
MVSQRQRLAR--KR-YKA-EHP-E-L---FPKPEPTPPKDPEKKKKKKKNSAFKRK-RP-EPKPGSRKRHPLRV
PGMKP-GESCFICKAMDHI-AKLCPEKAEWEKNKICLRCRRRGHRAKNC--P--EVLDG--A-KD---AM-Y-CY
NCGE-NG-HSLT-QCPH-P-LQEG----GTKFAECFVC-NQ-RGHLSKNCPQNTHGIYPKGGCCKICGGVTHLAK
--DCPDKGKSGSVAANR--PADG-WMRIEER-PMGQVTKFVSG--DDIEDDFMTD-DIH-SG-DKKKPAK-STED
HVKPKKKEGPKVVNF

QVKKHSKHCRTCNRCVEGFDHHCRWLNNCVGKRNYTAFFLLMIFLLIKGGTAIAIFIRCFVDRRGIEKELQRKL
YVDFPRGVLATICVFLLLLTAYSSAALGQLFFFHVVLIRKTNTHAIKSIIISLRKTYDYILAMKEENEAMELESF
DDSDFSSDESFDFDSPEKPTLMSGFLCKGNQGKKALLAAEKARERIMREKPMGEHNSLKPLPLETKCGPLMNTYK
NMDTEDFGSTSFIAKGRLNESPGRFSSPRRRFSAGSPTVFSSSMMASPHHKYRSSFDLKLTGVSRELETHISRQV
LCSVISKDDSEPSPR
```

Similarly, Fig.3.4 demonstrates the plot of local alignment scoring matrix from different views. As expected, this matrix has several peaks corresponding to each local alignment. The highest peak is the best local alignment. In our implementation, we detected the two best local alignments with the scores of 44 and 42 respectively. They are as follows:

```
FAECFVCNQRGHLSKNCPQNTHGIYPKGGCCKIC
FIRCFV-DRRG-IEKELQRKLYVDFPRGVLATIC

RQRLARKRYKAEHPELFPKPEPT
RERIMREKPMGEHNSLKPLPLET
```

We also implemented a program to graphically visualize the two aligned sequences using

OpenGL. Due to space limits, we are not going to demonstrate screen shots here. We instead put it on the server (as well as its source code) with the instructions on its usage.

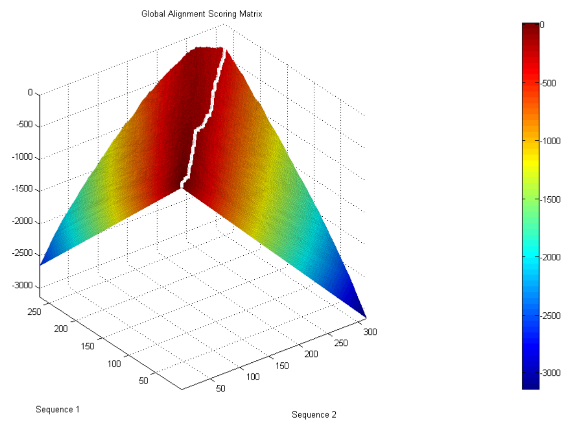
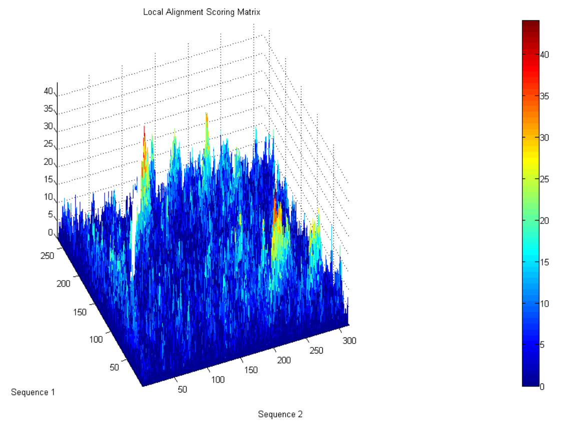
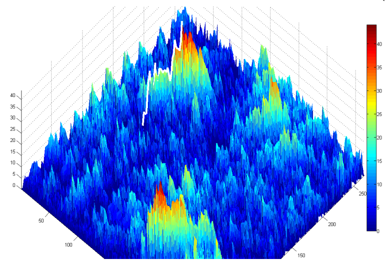


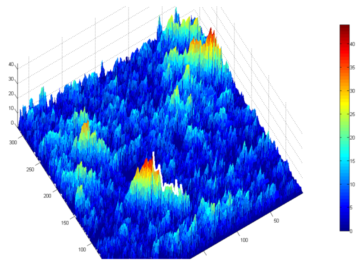
Figure 3.3: Global alignment scoring matrix



(a) View 1



(b) View 2



(c) View 3

Figure 3.4: Local alignment scoring matrix

Algorithm 3.1 Dynamic Programming for *Global* Sequence Alignment

Input:

- X, Y – two sequences
- $B(i, j)$ – the *Blosum* matrix containing the score of amino-acids i and j alignment, $i,\ j \in \{A, R, N, D, C, Q, E, G, H, I, L, K, M, F, P, S, T, W, Y, V\}$
- *gapscore* – gap score (-10 in this project)

Output: The best *global* alignment score $\hat{s}$ and the aligned sequences $\hat{X}$, $\hat{Y}$

```
 1: $m \leftarrow \textsc{Length}(X)$
 2: $n \leftarrow \textsc{Length}(Y)$
                                                        ▷ Initialization
 3: Initialize $S[0..m, 0..n]$,  $T[0..m, 0..n]$          ▷ Score matrix and Trace matrix
 4: for $i \leftarrow 0$ to $m$ do
 5:     $S[i, 0] \leftarrow i * gapscore$
 6:     $T[i, 0] \leftarrow 2$
 7: end for
 8: for $j \leftarrow 0$ to $n$ do
 9:     $S[0, j] \leftarrow j * gapscore$
10:     $T[0, j] \leftarrow 3$
11: end for
                                                        ▷ Needleman-Wunsch algorithm
12: for $i \leftarrow 1$ to $m$ do
13:     for $j \leftarrow 1$ to $n$ do
14:         $s_1 \leftarrow S[i-1, j-1] + B[X[i], Y[j]]$     ▷ $X_i$ aligned with $Y_j$
15:         $s_2 \leftarrow S[i-1, j] + gapscore$            ▷ $X_i$ aligned with gap
16:         $s_3 \leftarrow S[i, j-1] + gapscore$            ▷ gap aligned with $Y_j$
17:         $S[i, j] \leftarrow \max_{k \in \{1,2,3\}} s_k$          ▷ Choose optimal alignment
18:         $T[i, j] \leftarrow \arg\max_{k \in \{1,2,3\}} s_k$      ▷ Save the trace
19:     end for
20: end for
21: $\hat{s} \leftarrow S[m, n]$                                ▷ Best alignment score
                                                        ▷ Begin to trace back
22: $\hat{X} \leftarrow$ empty string
23: $\hat{Y} \leftarrow$ empty string
24: $(i, j) \leftarrow (m, n)$      ▷ Start at the bottom right corner of the scoring matrix
```

Algorithm 3.1 Dynamic Programming for *Global* Sequence Alignment (continued)

```
25: while $i \geq 0$ and $j \geq 0$ do
26:     if $T[i, j] = 1$ then                          ▷ $X[i]$ aligned with $Y_j$
27:         $\hat{X} \leftarrow X[i] + \hat{X}$
28:         $\hat{Y} \leftarrow Y[j] + \hat{Y}$
29:         $i \leftarrow i - 1$
30:         $j \leftarrow j - 1$
31:     else if $T[i, j] = 2$ then                     ▷ $X_i$ aligned with gap
32:         $\hat{X} \leftarrow X[i] + \hat{X}$
33:         $\hat{Y} \leftarrow \text{gap} + \hat{Y}$
34:         $i \leftarrow i - 1$
35:     else                                           ▷ gap aligned with $Y_j$
36:         $\hat{X} \leftarrow \text{gap} + \hat{X}$
37:         $\hat{Y} \leftarrow Y[j] + \hat{Y}$
38:         $j \leftarrow j - 1$
39:     end if
40: end while
41: return $\hat{s}$, $\hat{X}$, $\hat{Y}$
```

Algorithm 3.2 Dynamic Programming for *Local* Sequence Alignment

Input:

- X, Y – two sequences
- $B(i, j)$ – the *Blosum* matrix containing the score of amino-acids i and j alignment, $i,\ j \in \{A, R, N, D, C, Q, E, G, H, I, L, K, M, F, P, S, T, W, Y, V\}$
- *gapscore* – gap score (-10 in this project)

Output: The best *local* alignment score $\hat{s}$ and the aligned sequences $\hat{X}$, $\hat{Y}$

```
1:  $m \leftarrow$ LENGTH($X$)
2:  $n \leftarrow$ LENGTH($Y$)
                                                  ▷ Initialization
3:  Initialize $S[0..m, 0..n]$,   $T[0..m, 0..n]$        ▷ Score matrix and Trace matrix
4:  $\hat{s} \leftarrow 0$
5:  $(u, v) \leftarrow (0, 0)$
6:  for $i \leftarrow 0$ to $m$ do
7:      $S[i, 0] \leftarrow 0$
8:  end for
9:  for $j \leftarrow 0$ to $n$ do
10:     $S[0, j] \leftarrow 0$
11: end for
                                                  ▷ Smith-Waterman algorithm
12: for $i \leftarrow 1$ to $m$ do
13:     for $j \leftarrow 1$ to $n$ do
14:         $s_0 \leftarrow 0$                             ▷ No alignment, just keep it as prefix
15:         $s_1 \leftarrow S[i-1, j-1] + B[X[i], Y[j]]$  ▷ $X_i$ aligned with $Y_j$
16:         $s_2 \leftarrow S[i-1, j] + gapscore$          ▷ $X_i$ aligned with gap
17:         $s_3 \leftarrow S[i, j-1] + gapscore$          ▷ gap aligned with $Y_j$
18:         $S[i, j] \leftarrow \max_{k \in \{0,1,2,3\}} s_k$        ▷ Choose optimal alignment
19:         $T[i, j] \leftarrow \arg\max_{k \in \{0,1,2,3\}} s_k$    ▷ Save the trace
20:         if $\hat{x} < S[i, j]$ then                    ▷ Locate the maximum value
21:             $\hat{x} \leftarrow S[i, j]$
22:             $(u, v) \leftarrow (i, j)$
23:         end if
24:     end for
25: end for
                                                  ▷ Begin to trace back
26: $\hat{X} \leftarrow$ empty string
27: $\hat{Y} \leftarrow$ empty string
```

Algorithm 3.2 Dynamic Programming for *Local* Sequence Alignment (continued)

```
28: while $u \geq 0$ and $v \geq 0$ and $S[u,v] \neq 0$ do
29:     if $T[u,v] = 1$ then                              ▷ $X_u$ aligned with $Y_v$
30:         $\hat{X} \leftarrow X[u] + \hat{X}$
31:         $\hat{Y} \leftarrow Y[v] + \hat{Y}$
32:         $u \leftarrow u - 1$
33:         $v \leftarrow v - 1$
34:     else if $T[u,v] = 2$ then                         ▷ $X_u$ aligned with gap
35:         $\hat{X} \leftarrow X[u] + \hat{X}$
36:         $\hat{Y} \leftarrow \text{gap} + \hat{Y}$
37:         $u \leftarrow u - 1$
38:     else if $T[u,v] = 3$ then                         ▷ gap aligned with $Y_v$
39:         $\hat{X} \leftarrow \text{gap} + \hat{X}$
40:         $\hat{Y} \leftarrow Y[v] + \hat{Y}$
41:         $v \leftarrow v - 1$
42:     end if
43: end while
44: return $\hat{s}$, $\hat{X}$, $\hat{Y}$
```

Chapter 4

Maximum Flow/Minimum Cut by Linear Programming

4.1 Abstract

We experimented with solving a pair of graph-related problems, maximum flow and minimum cut, via linear/integer programming. By modeling the two problems as a series of linear constraints, we then utilized an open-source simplex solving package (Lips IDE) to find the optimal solution. We also compared the results obtained from linear/integer programming with known polynomial-time algorithm (Edmonds-Karp) for solving maximum flow problem.

4.2 Introduction

Graph/network problems are a rich intersection of computer science and mathematics. Beginning with Euler's Königsberg bridge problem in 1735, analysis of the simplification of nodes (points on a graph) and edges (connections between points) has had ramifications in a number of other fields such as abstract algebra and topology. In computer science, the various networks that form the foundation of the internet lend themselves naturally to graph theory. As such, graph problems have a number of real-world applications such as routing algorithms for data networks or failure mode for telephone services. In addition, modeling the social networks of people on online sites such as Facebook are another popular area of modern analysis.

The maximum flow problem is common: given start/end (source/sink) nodes on a network and knowledge of the capacity of each edge between them, figure out the set of flows to produce the maximum flow at the end point. This basic problem can be transformed into a number of other applications, such as *minimum path cover* and *maximum matching in bipartite graphs.*

The minimum cut problem, on the other hand, involves finding the cut (set of edges to be removed from the graph) that not only separates the source and sink but also has the least possible combined weight of the edges removed. A popular recent application of minimum cut problem is *graph-cut based optimization*, which solves the minimization of a wide range of functions in image processing, computer graphics and computer vision.

Due to the duality theorem, these two problems are related: a maximum flow solution can be converted to a minimum cut solution. This is accomplished by grouping the vertices reachable from s in the residual graph $G_f = (V, E)$, where $e_f(u, v) = c(u, v) - f(u, v)$, as S and the remaining as T. As a result, the set of cut edges contains the edges across S and T.

Linear programming (or more precisely linear optimization) is a mathematical method of determining the optimal state of a set of variables given a linear objective function subject to a set of linear equalities/inequalities as constraints. Loosely, the constraints form a convex polygon in the solution space and then we attempt to optimize the objective function in this space. The simplex method, first proposed by Dantzig, walks along the edges of the convex polygon until it finds a maximum state. If the initial set of constraints is done properly, this method can rapidly produce a solution. As such, the design of objective function (which will be optimized) and constraints are the most important part of linear programming, by which a complex problem is modelled in a standard form which can be efficiently solved by many numerical algorithms.

4.3 Maximum Flow

Linear Programming of Max-Flow problem

$$\boxed{\begin{array}{llcll}
\max\limits_{\boldsymbol{f}} & \sum\limits_{(s,v)\in E} f(s,v) & & & \\
\text{subject to} & & & & \\
 & f(u,v) & \geq & 0, & \forall (u,v) \in E \\
 & f(u,v) & \leq & c(u,v), & \forall (u,v) \in E \\
 & \sum\limits_{(u,v)\in E} f(u,v) & = & \sum\limits_{(v,u)\in E} f(v,u), & \forall v \in V \setminus \{s,t\}
\end{array}}$$

(4.1a) (4.1b) (4.1c) (4.1d)

Given a graph (V, E), the source s and the sink t, the capacity $c(u, v)$ for each edge $(u, v) \in E$, we need to determine the flow on each edge. Thus, we introduce the variables $f(u, v)$ for all edges $(u, v) \in E$. The goal of Max-flow problem is to maximize the total flow going through the network. It can be shown that this flow is the sum of all flows going out of the source s (or the sum of all flows going into the sink t, i.e. $\sum\limits_{(s,v)\in E} f(s, v) = \sum\limits_{(v,t)\in E} f(v, t)$). This goal can be mathematically expressed as the objective function in (4.1a). Due to basic properties of flow, the flow $\boldsymbol{f}$ must satisfy the following constraints.

1. The flow along an edge is greater than/equal to zero (4.1b)
2. The flow along an edge is less than the capacity of the edge (4.1c)
3. The total flow into any vertex other than s and t is equal to the total flow out of it (flow conservation and hence vertices do not have reservoirs to store flow) (4.1d)

With these constraints, (4.1a), (4.1b), (4.1c) and (4.1d) form a linear programming problem. However, the resulting maximum flow may not be integer (i.e. there may exist some edge (u, v) such that $f(u, v) \notin \mathbb{Z}$), which is abnormal in many real applications. To enforce the flow integral, we add another constraint as in (4.2e). This modification results in the integer programming as following.

Integer Programming of Max-Flow problem

$$\max_{\boldsymbol{f}} \quad \sum_{(s,v)\in E} f(s,v) \tag{4.2a}$$

subject to

$$\begin{aligned}
f(u,v) &\geq 0, && \forall (u,v)\in E && (4.2b)\\
f(u,v) &\leq c(u,v), && \forall (u,v)\in E && (4.2c)\\
\sum_{(u,v)\in E} f(u,v) &= \sum_{(v,u)\in E} f(v,u), && \forall v\in V\setminus\{s,t\} && (4.2d)\\
f(u,v) &\in \mathbb{Z}^+, && \forall (u,v)\in E && (4.2e)
\end{aligned}$$

4.4 Minimum Cut

In graph theory, given an undirected graph $G = (V, E)$, the capacity $c_{u,v}$ for each edge $(u, v) \in E$, two vertices $s, t \in V$, a cut (S, T) is a partition of V into two disjoint sets S and T (i.e. $S \cap T = \varnothing$) such that $s \in S$ and $t \in T$. The cost of a cut (S, T) is defined as

$$\text{cost}(S,T) = \sum_{i\in S}\sum_{j\in T} c_{i,j} \tag{4.3}$$

Then, the minimum cut problem is to find the cut with the smallest cost. To model a cut (S, T), we first introduce binary variables $d_{i,j}$ for all edges $(i, j) \in E$. We want $d_{i,j} = 1$ if edge (i, j) is cut, otherwise $d_{i,j} = 0$; Thus the objective of the minimum cut problem can be mathematically expressed as function in 4.4a. We also introduce binary variables p_i for each vertex $i \in V$ that encodes the set which vertex i^{th} belongs to (i.e. $p_i = 0$ if $i \in S$, $p_i = 1$ otherwise). The constraints 4.4b and 4.4c are definitions of the two sets of binary variables we have just mentioned.

However, we do not know the cut (S, T) beforehand, we want to explore the relationship between $\boldsymbol{d}$ and $\boldsymbol{p}$. We observe that if vertex i^{th} and vertex j^{th} are in different sets, e.g. $i \in T$ and $j \in S$ (i.e. $p_i = 1$ and $p_j = 0$), edge (i, j) must be cut (i.e. $d_{i,j} = 1$). To enforce this relationship, we add another constraint as in 4.4d. There are two cases to discuss here.

- If i and j are in the same set, we have $-p_i + p_j = 0$, hence $d_{i,j}$ can be either 0 or 1. However, if $d_{i,j} = 1$, the cost for this cut is not minimized because we add a positive constant $c_{i,j}$ to the objective function. Thus, the minimizer will force $d_{i,j} = 0$ or, in other words, preserve the edges between same set's vertices.
- What happens if $i \in S$ and $j \in T$? In this case, $-p_i + p_j = 1$, so $d_{i,j}$ can also be either 0 or 1. With similar argument as above, the minimizer will keep $d_{i,j} = 0$ and it seems to violate the cut definition. However, we will show that this is not the case. Recall that the graph G is undirected, so if the edge $(i, j) \in E$, so does the edge (j, i). In

this case, $d_{i,j} = 0$ for the constraint on edge (i, j) but $d_{j,i} = 1$ for the constraint on edge (j, i). The combined result tells that edge (i, j) is still cut. Moreover, the cost to cut this edge is counted once as $d_{i,j} = 0$ and $d_{j,i} = 1$. Therefore, the overall designs of the constraint (4.4d) and objective function (4.4a) are enough to cover all cases.

Since we know that $s \in S$ and $t \in T$ (i.e. $p_s = 0$ and $p_t = 1$), it is straightforward to introduce the last constraint (4.4e).

Integer Programming of Min-Cut problem

$$\boxed{\begin{array}{lrcll}
\min\limits_{\boldsymbol{d},\boldsymbol{p}} & \displaystyle\sum_{(i,j)\in E} c_{i,j}\ d_{i,j} & & & \\
\text{subject to} & & & & \\
 & d_{i,j} & \in & \{0,\ 1\}, & \forall (i,j) \in E \\
 & p_i & \in & \{0,\ 1\}, & \forall i \in V \\
 & d_{i,j} - p_i + p_j & \geq & 0, & \forall (i,j) \in E \\
 & p_t - p_s & = & 1 &
\end{array}}$$

(4.4a) (4.4b) (4.4c) (4.4d) (4.4e)

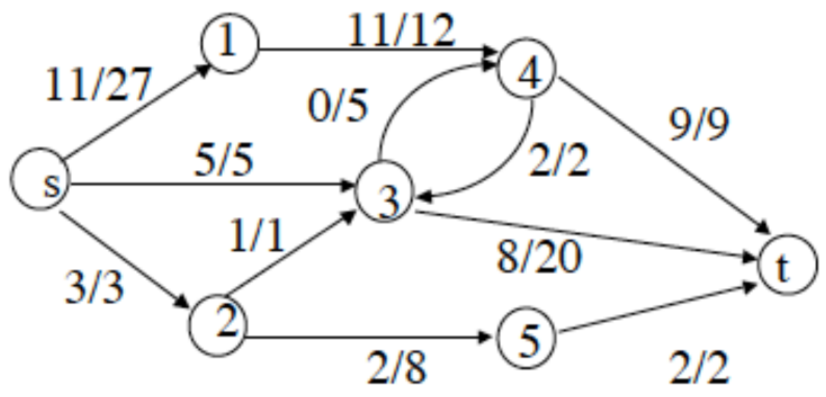
4.5 Results and Evaluation

We are using LiPS software to solve our problems. All we need to do is generate lpx scripts that are supported by the software. To handle the problem with big number of nodes, we are writing a MATLAB programs, maxflow.m and mincut.m, that automatically generate lpx scripts from standard graph. You can get these MATLAB code in Appendix session. The standard input text file describes the structure of the graph. The first row contains four numbers that represent the number of vertices $|V|$, the number of edges number $|E|$, source s and sink t of the graph. In the next $|E|$ rows, each row describes an edge between two vertices with its capacity.

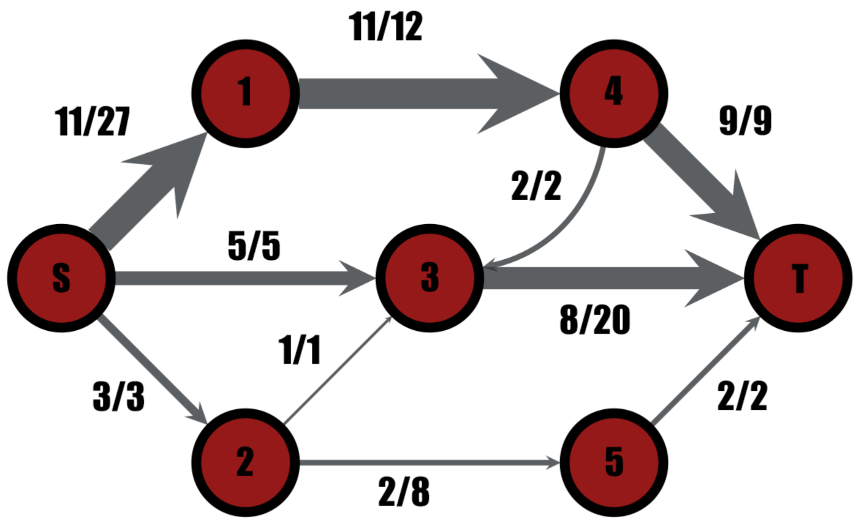
In order to validate and visualize our results we utilized a python open source package, GraphTool (available at: `http://graph-tool.skewed.de`). We converted our sample graph to the file format used by the program and then wrote a simple solver utilizing the Edmonds-Karp algorithm. Next, we utilized GraphTool's builtin utilities to visualize our graph.

We tested on the graph in Fig.4.1a. For the max flow algorithm, GraphTool produced the visualization in Fig.4.1b, with line widths sized relative to the flow in the final graph, with the same result as our linear programming approach.

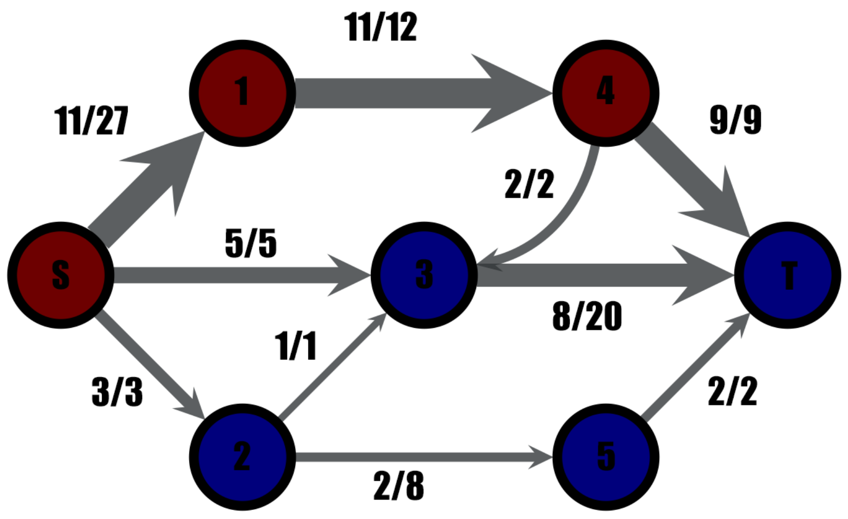
Likewise, for minimum cut, GraphTool produced the same end result, with blue and red corresponding to the source set and sink set (Fig.4.1c).



(a) Original Graph



(b) Maximum Flow result



(c) Minimum Cut result

Figure 4.1: Max-Flow and Min-Cut result

Chapter 5

Implementing a Small Support Vector Machine

5.1 Abstract

We implement a small Support Vector Machine (SVM) utilizing Sequential Minimal Optimization (SMO) and Gradient Descent to classify/predict the incidence of diabetes from a known dataset. We implement a second SVM utilizing gradient descent in Matlab as well as another utilizing open-source package SVM-Light as a benchmark. Finally, we compare and contrast our results on a known data set and demonstrate a visualization of the SMO process by MATLAB.

5.2 Introduction

Support Vector Machines, invented by Vladimir N. Vapnik, are a popular modern technique utilized in machine learning. In its basic form, a SVM attempts to find a way of splitting two data sets via a line drawn in one of the dimensions of the data set. However, real world data sets rarely can be so cleanly divided. As such, the next improvement to the SVM method was the introduction of non-linear kernels, which map the state space to higher dimensions (where technically they are split there linearly). The soft margin technique (proposed by Vapnik in 1995) is the next major improvement to the algorithm which, by assigning a small penalty to misclassified data, allows the SVM to assign a score to each possible solution. Afterwards, the next improvement is simply to find the smallest possible score and by extension optimal state to the classification vector.

Basically, the task is to

$$\min_{\boldsymbol{w}\in\mathbb{F},b\in\mathbb{R},\boldsymbol{\xi}\in\mathbb{R}^m} \quad \frac{1}{2}\|\boldsymbol{w}\| + C\sum_{i=1}^{m}\xi_i \tag{5.1a}$$

$$\text{subject to} \quad y^{(i)}\left(\langle \boldsymbol{w}, \boldsymbol{\phi}(\boldsymbol{x^{(i)}})\rangle + b\right) \geq 1-\xi_i, \qquad \forall i=1,\dots,m \tag{5.1b}$$

$$\xi_i \geq 0, \qquad \forall i=1,\dots,m \tag{5.1c}$$

where C is a parameter for balancing how large the margins are and how many data points allowed to be misclassified or within the margin.

By introducing a Lagrange function, we transform the problem (5.1) to the dual problem of maximizing the following constrained quadratic optimization

$$\max_{\boldsymbol{\alpha}\in\mathbb{R}^m} \quad W(\boldsymbol{\alpha}) = \sum_{i=1}^{m}\alpha_i - \frac{1}{2}\sum_{i,j=1}^{m}\alpha_i\alpha_j y^{(i)}y^{(j)}K(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}) \tag{5.2a}$$

$$\text{subject to} \quad \sum_{i=1}^{m}\alpha_i y^{(i)} = 0 \tag{5.2b}$$

$$0 \le \alpha_i \le C, \qquad \forall i = 1,\dots,m \tag{5.2c}$$

where $K(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)}) = \langle \boldsymbol{\phi}(\boldsymbol{x}^{(i)}), \boldsymbol{\phi}(\boldsymbol{x}^{(j)})\rangle$ is the (potentially nonlinear) kernel. From now, for shorthand, we use $K_{i,j} = K(\boldsymbol{x}^{(i)},\boldsymbol{x}^{(j)})$

The decision function given a new data point $\boldsymbol{z}$ is

$$f(\boldsymbol{z}) = \sum_{i=1}^{m}\alpha_i y^{(i)}K(\boldsymbol{\phi}(\boldsymbol{x}^{(i)}),\boldsymbol{\phi}(\boldsymbol{z})) + b \tag{5.3}$$

We investigated three different methods of finding the optimal solution of above constrained quadratic problem. We surveyed Gradient Descent and Quadratic Programming as well as Sequential Minimum Optimization, a significant modern improvement. Gradient Descent is a classical optimization technique dating back to Newton whereby we estimate the rate of change in the local neighborhood of the current state and then use this to choose a new state for investigation. It is very simple and easy to implement. Usually, it makes good progress when it is far away from the optimum but it becomes slower when it is close to the optimal solution. Quadratic Programming is a significant improvement to this approach but it does not scale well for large datasets as its complexity is $O(m^3)$, where m is the number of data points. In the following section, we are going to present SMO and Gradient Descent.

5.3 Sequential Minimal Optimization

5.3.1 KKT condition

The SMO algorithm gives an efficient way of solving the dual problem of the (regularized) SVM optimization problem. The KKT condition can be check for convergence to the optimal point. For this problem, the KKT conditions are

$$\alpha_i = 0 \rightarrow y^{(i)}(\langle \boldsymbol{w}, \boldsymbol{\phi}(\boldsymbol{x}^{(i)})\rangle + b) \ge 1 \tag{5.4a}$$

$$\alpha_i = C \rightarrow y^{(i)}(\langle \boldsymbol{w}, \boldsymbol{\phi}(\boldsymbol{x}^{(i)})\rangle + b) \le 1 \tag{5.4b}$$

$$0 < \alpha_i < C \rightarrow y^{(i)}(\langle \boldsymbol{w}, \boldsymbol{\phi}(\boldsymbol{x}^{(i)})\rangle + b) = 1 \tag{5.4c}$$

Replacing (5.3), we can rewrite the KKT conditions as

$$\alpha_i = 0 \rightarrow y^{(i)}f(\boldsymbol{x}^{(i)}) \ge 1 \tag{5.5a}$$

$$\alpha_i = C \rightarrow y^{(i)}f(\boldsymbol{x}^{(i)}) \le 1 \tag{5.5b}$$

$$0 < \alpha_i < C \rightarrow y^{(i)}f(\boldsymbol{x}^{(i)}) = 1 \tag{5.5c}$$

5.3.2 Optimizing Alphas

The quadratic problem contains multiple dimensional variables. The idea of SMO algorithm is to break the problem into multiple sub problems that can be solved analytically. From the constraint (5.2b), it is impossible to change one α_i without violating this constraint. By allowing only two alphas changed at a time, we transform it to only one dimensional quadratic problem due to the constraint (5.2b).

There are some more sophisticated heuristic improvements to choose which α_i and α_j in order to maximize the objective function as quickly as possible. However, in the scope of this project we utilize a simpler heuristic. We simply pick the first points that violates the KKT conditions in (5.5) and randomly choose the second one from the $m-1$ remaining parameters. Then we attempt to maximize

$$W(\alpha_i, \alpha_j) = \alpha_i + \alpha_j - \frac{1}{2}K_{i,j}\alpha_i^2 - \frac{1}{2}K_{j,j}\alpha_j^2 - sK_{i,j}\alpha_i\alpha_j - y^{(i)}\alpha_i v_i - y^{(j)}\alpha_j v_j + W_{\text{constant}} \tag{5.6a}$$

$$\alpha_i y_i + \alpha_j y_j = k \tag{5.6b}$$

$$0 \leq \alpha_i, \alpha_j \leq C \tag{5.6c}$$

where

$$v_l = \sum_{\substack{k=1 \\ k \neq i,j}}^{m} y^{(k)}\alpha_k^{(old)}K_{k,l} = f^{(\text{old})}(\boldsymbol{x}^{(l)}) + b^{(\text{old})} - y^{(i)}\alpha_i^{(\text{old})}K_{l,i} - y^{(j)}\alpha_j^{(\text{old})}K_{l,j}, \text{ for } l = i, j$$

$$s = y^{(i)}y^{(j)}$$

Substituting (5.6b) into (5.6a) yields

$$\begin{aligned} W(\alpha_j) =& k - s\alpha_j + \alpha_j - \frac{1}{2}K_{i,i}(k - s\alpha_j)^2 - \frac{1}{2}K_{j,j}\alpha_j^2 \\ & - sK_{i,j}(k - s\alpha_j)\alpha_j - y^{(i)}(k - s\alpha_j)v_i - y^{(j)}\alpha_j v_j + W_{\text{const}} \end{aligned} \tag{5.7}$$

Taking the derivative with respect to α_j, equating to 0, we obtain

$$\alpha_j(K_{i,i} + K_{j,j} - 2K_{i,j}) = s(K_{i,i} - K_{i,j})k + y^{(j)}(v_i - v_j) + 1 - s \tag{5.8}$$

Or,

$$\alpha_j(K_{i,i}+K_{j,j}-2K_{i,j}) = \alpha_j^{(\text{old})}(K_{i,i}+K_{j,j}-2K_{i,j}) + y^{(j)}(f(\boldsymbol{x}^{(i)}) - f(\boldsymbol{x}^{(j)}) + y^{(j)} - y^{(i)}) \tag{5.9}$$

$$\alpha_j^{(\text{new})} = \alpha_j^{(\text{old})} - \frac{y^{(j)}(E_i - E_j)}{\eta} \tag{5.10}$$

where $\eta = K_{i,i} + K_{j,j} - 2K_{i,j}$ and $E_l = f(\boldsymbol{x}^{(l)}) - y^{(l)}$

However, due to the constraint in (5.6c), $\alpha_j^{(\text{new})}$ may not be satisfied. Note that the constraints of (5.6c) and (5.6b) infer the bounds $[L, H]$ for $\alpha_j^{(\text{new})}$, where

$$L = \begin{cases} \max(0, \alpha_j - \alpha_i) & \text{if } y^{(i)} \neq y^{(j)} \\ \max(\alpha_j + \alpha_i - C) & \text{if } y^{(i)} = y^{(j)} \end{cases}, \quad H = \begin{cases} \min(C, C + \alpha_j - \alpha_i) & \text{if } y^{(i)} \neq y^{(j)} \\ \min(C, \alpha_i + \alpha_j) & \text{if } y^{(i)} = y^{(j)} \end{cases} \tag{5.11}$$

Thus, we can clip $\alpha_j^{(\text{new})}$ to this $[L, H]$ bounds by setting

$$\alpha_j^{(\text{new})} := \begin{cases} L & \text{if } \alpha_j^{(\text{new})} \leq L \\ H & \text{if } \alpha_j^{(\text{new})} \geq H \\ \alpha_j^{(\text{new})} & \text{if } L < \alpha_j^{(\text{new})} < H \end{cases} \tag{5.12}$$

Then, we update α_i according to the constraint (5.6b), we have

$$\alpha_i^{(\text{new})} = \alpha_i^{(\text{old})} + y^{(i)} y^{(j)} (\alpha_j^{(\text{old})} - \alpha_j^{(\text{new})}) \tag{5.13}$$

Lastly, we need to update the bias b according to the KKT condition (5.5). If $0 < \alpha_i < C$, we can calculate b_1. If $0 < \alpha_j < C$, we can calculate b_2. Otherwise, all b values between b_1 and b_2 make α_i and α_j satisfy the KKT condition. SMO picks the average value in this case.

Algorithm (5.1) sketches the outline of the SMO algorithm. The detailed version is in algorithm (5.2).

5.4 Gradient Descent

Basically, we can solve for (5.2a) by substituting α_1 from (5.2b) to the objective function. We obtain the following objective function.

$$W(\alpha_2, \ldots, \alpha_m) = \sum_{i=2}^{m} (1 - y_1 y_i) \alpha_i - \frac{1}{2} \sum_{i=2}^{m} \sum_{j=2}^{m} \alpha_i \alpha_j y^{(i)} y^{(j)} (K_{i,j} + K_{1,1} - 2K_{1,i}) \tag{5.14}$$

We also calculate the gradient of objective function with respect to each α_i, $i = 2, \ldots, m$

$$\frac{\partial W}{\partial \alpha_i} = (2K_{1,i} - K_{i,i} - K_{1,1}) \alpha_i + 1 - y^{(1)} y^{(i)} - \sum_{\substack{j=2 \\ j \neq i}}^{m} \alpha_j y^{(j)} y^{(i)} (K_{i,j} + K_{1,1} - K_{1,j} - K_{1,i}) \tag{5.15}$$

$$= 1 - y^{(1)} y^{(i)} - \sum_{j=2}^{m} \alpha_j y^{(j)} y^{(i)} (K_{i,j} + K_{1,1} - K_{1,j} - K_{1,i}) \tag{5.16}$$

for $i = 2, \ldots, m$ Algorithm (5.3) describes a standard Gradient Descent algorithm that we applied.

5.5 Results and Evaluation

In order to evaluate our implementation, we are using the diabetic dataset available at `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary/diabetes`. We use 70%

Algorithm 5.1 Sequential Minimal Optimization (short version)

Input:

- C, regularization parameter
- tol, numerical tolerance
- max_passes, the maximum of times to iterate over α's without changing
- $\{(x^{(i)}, y^{(i)}), i = 1, \dots, m\}$, training data

Output: α's

```
1: $\alpha_i \leftarrow 0, \forall i, \quad b \leftarrow 0$                              ▷ Initialization
2: $passes \leftarrow 0$                                                                   ▷ Initialization
3: while $passes < max\_passes$ do
4:     $num\_changed\_alphas \leftarrow 0$
5:     Iterate over $\alpha_i$ to choose the first one violate KKT condition
6:     Randomly choose $j \neq i$
7:     Solve closed form solution for constrained quadratic problem
8:     if $|\alpha_j^{(new)} - \alpha_j^{(old)}| > tol$ then
9:         Update bias $b$
10:        $num\_changed\_alphas \leftarrow num\_changed\_alphas + 1$
11:    end if
12:    if $num\_changed\_alphas = 0$ then
13:        $passes \leftarrow passes + 1$
14:    else
15:        $passes \leftarrow 0$
16:    end if
17: end while
```

Algorithm 5.2 Sequential Minimal Optimization (detailed version)

Input:

- C, regularization parameter
- tol, numerical tolerance
- max_passes, the maximum of times to iterate over α's without changing
- $\{(x^{(i)}, y^{(i)}), i = 1, \dots, m\}$, training data

Output: α's

```
1: $\alpha_i \leftarrow 0, \forall i, \quad b \leftarrow 0$                              ▷ Initialization
2: $passes \leftarrow 0$                                                                   ▷ Initialization
3: while $passes < max\_passes$ do
4:     $num\_changed\_alphas \leftarrow 0$
5:     for $i \leftarrow 1$ to $m$ do
6:         $E_i \leftarrow \sum_{t=1}^{m} \alpha_t y^{(t)} K(x^{(t)}, x^{(i)}) + b - y^{(i)}$     ▷ $E_i = w^T x^{(i)} + b - y^{(i)}$
```

Algorithm 5.2 Sequential Minimal Optimization (detailed version) (continued)

```
 7:         **if** $(y^{(i)}E_i < -tol \land \alpha_i < C) \lor (y^{(i)}E_i > tol \land \alpha_i > 0)$ **then**
 8:             Randomly choose $j \neq i$
 9:             $E_j \leftarrow \sum_{t=1}^{m} \alpha_t y^{(t)} K(x^{(t)}, x^{(j)}) + b - y^{(j)}$        ▷ $E_j = w^T x^{(j)} + b - y^{(j)}$
10:             $(\alpha_i^{(\text{old})}, \alpha_j^{(\text{old})}) \leftarrow (\alpha_i, \alpha_j)$        ▷ Save old $\alpha_i, \alpha_j$
11:             **if** $y^{(i)} \neq y^{(j)}$ **then**
12:                 $(L, H) \leftarrow (\max(0, \alpha_j - \alpha_i), \min(C, C + \alpha_j - \alpha_i))$
13:             **else**
14:                 $(L, H) \leftarrow (\max(0, \alpha_i + \alpha_j - C), \min(C, \alpha_i + \alpha_j))$
15:             **end if**
16:             **If** $L = H$ **then**
17:                 continue to next $i$
18:             $\eta \leftarrow 2K(x^{(i)}, x^{(j)}) - K(x^{(i)}, x^{(i)}) - K(x^{(j)}, x^{(j)})$
19:             **If** $\eta \geq 0$ **then**
20:                 continue to next $i$
21:             $\alpha_j \leftarrow \alpha_j - \frac{y^{(j)}(E_i - E_j)}{\eta}$        ▷ Find optimal $\alpha_j$ by Newton's method
22:             $\alpha_j \leftarrow \begin{cases} H & \text{if } \alpha_j > H \\ \alpha_j & \text{if } L \leq \alpha_j \leq H \\ L & \text{if } \alpha_j < L \end{cases}$        ▷ Clip $\alpha_j$ to $[L, H]$
23:             **If** $|\alpha_j - \alpha_j^{(old)}| < tol$ **then**
24:                 continue to next $i$
25:             $\alpha_i \leftarrow \alpha_i + y^{(i)} y^{(j)} (\alpha_j^{(old)} - \alpha_j)$
26:             $b_1 \leftarrow b - E_i - y^{(i)}(\alpha_i - \alpha_i^{(\text{old})})K(x^{(i)}, x^{(i)}) - y^{(j)}(\alpha_j - \alpha_j^{(\text{old})})K(x^{(i)}, x^{(j)})$
27:             $b_2 \leftarrow b - E_j - y^{(i)}(\alpha_i - \alpha_i^{(\text{old})})K(x^{(i)}, x^{(j)}) - y^{(j)}(\alpha_j - \alpha_j^{(\text{old})})K(x^{(j)}, x^{(j)})$
28:             $b \leftarrow \begin{cases} b_1 & \text{if } 0 < \alpha_i < C \\ b_2 & \text{if } 0 < \alpha_j < C \\ (b_1 + b_2)/2 & \text{otherwise} \end{cases}$
29:             $num\_changed\_alphas \leftarrow num\_changed\_alphas + 1$
30:         **end if**
31:     **end for**
32:     **if** $num\_changed\_alphas = 0$ **then**
33:         $passes \leftarrow passes + 1$
34:     **else**
35:         $passes \leftarrow 0$
36:     **end if**
37: **end while**
```

of the positive diabetes samples and 70% of the negative diabetes samples to form our training dataset, with the remaining data points forming our testing dataset. Table (5.1) summaries our results with SMO and Gradient Descent as well as our reference SVM-Light and Quadratic Programming solutions.

Algorithm 5.3 Gradient Descent for QP

Input:

- C, regularization parameter
- η, learning rate
- tol, numerical tolerance
- $max\_iterator$, the maximum number of iterator

Output: α's

```
1: $\alpha_i \leftarrow rand(0, C), \forall i, \quad b \leftarrow 0$          ▷ Initialization
2: $k \leftarrow 0$          ▷ Initialization
3: **while** $k < max\_iterator$ **do**
4:     Compute gradient of objective function
5:     $\alpha^{(k+1)} \leftarrow \alpha^{(k)} + \eta \nabla W$
6:     Clip all $\alpha_i^{(k+1)}$ to the [0, C]
7:     **if** $||\alpha^{(k+1)} - \alpha^{(k)}|| < tol$ **then**
8:         break;
9:     **end if**
10: **end while**
```

Table 5.1: Comparison of Various Methods for Training a SVM

	GD	SMO	MATLAB GD	SVM-Light	QP
Accuracy on training set	63.45%	77.65%	63.45%	77.65%	77.65%
Accuracy on testing set	68.33%	78.33%	68.33%	78.33%	78.33%
Training time (*seconds*)	95.00	86.00	18.20	0.16	2.00
Number of SVs	10	10	10	9	9

5.6 Visualization

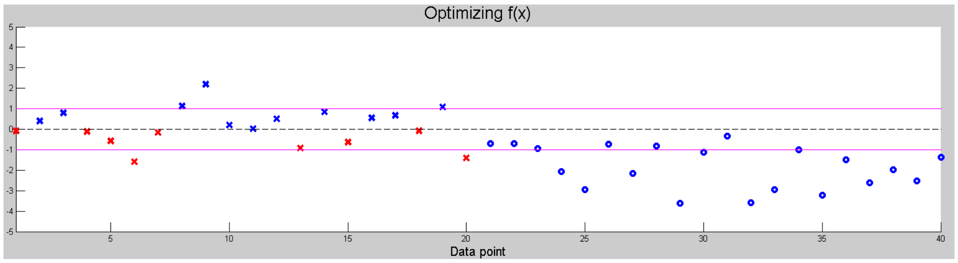


Figure 5.1: Optimizing outcome function

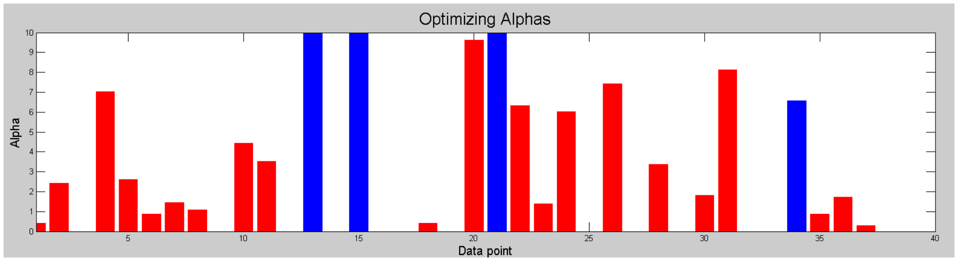


Figure 5.2: Optimizing α values. Red indicates KKT violation

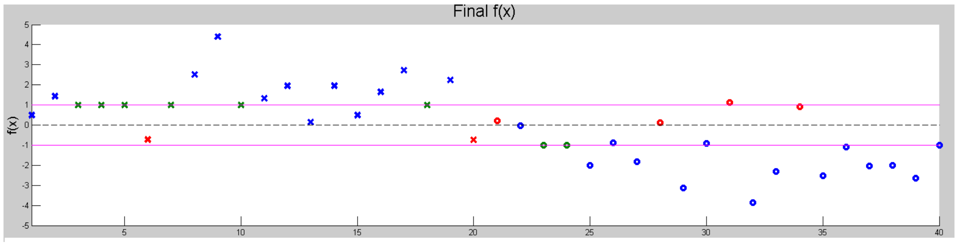


Figure 5.3: Final outcome function

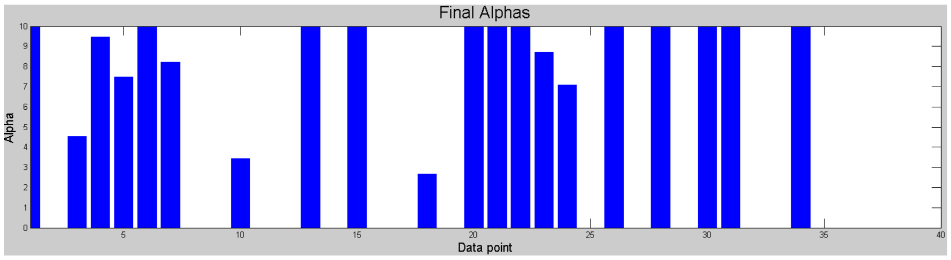


Figure 5.4: Final α values