DNAg: A DNA Self Assembly Generator

Priyanka Shukla, Sonam Jain
DA-IICT, Gandhinagar
200901102, 200901118@daiict.ac.in
Supervisor
Prof. Manish.K.Gupta

***Abstract*—In this project, our main aim was to design a suitable software that can be used as a tool for teaching purpose as well to generate DNA self assembly structures and DNA sequences which could be further used for experimental purposes. Problem statement addresses the issue of constructing DNA sequences using the computational model. DNA Computation means "computation using DNA". DNA has extremely dense information storage and parallel computing power, this makes it a unique computational element to study. The tool DNAg focuses on checking the Grammar Rules satisfying the Languages (Context Free Language and Regular Language) and designing DNA codewords to generate thermodynamically stable DNA sequences that would be used to generate this computational model through the DNA Self Assembly.**

***Index Terms*—Codebook, Constraints, Minimum Free Energy, Production Rules, Regular Language, Context Free Grammar, Self-Assembly.**

I. INTRODUCTION

Self Assembly is a process in which components obtain the most stable structure possible. DNA computing was the initiative of Prof. Leonard Adleman from the University of Southern California in the year 1994. Further, Erik Winfree, a PhD. student at the California Institute of Technology, in the year 1996, using the knowledge of DNA computing showed that concept of Self Assembly can be used at nano-scale computation [1]. DNA self assembly is primarily a dynamic self assembly, where the components demonstrate organized local behavior. While talking about DNA self assembly we try to illustrate the interaction between strand at nano level to generate products which are stable in equilibrium and have minimum free energy.

During DNA computation we direct these organized behaviors according to the grammar rules of the language defining a particular computational model. The Software DNAg: The DNA Self Assembly Generator, generates most stable structures of the DNA self assembly formed by particular set of grammar rules. Based on some simple concept of "Models of computation" we checked the input language and grammar. Further, to represent the Language we require stable DNA codewords. So, our second task was to construct the stable Single strand DNA structure (sequence of nucleotides $\{A,C,G,T\}$). To do so our focus was to generate large set of DNA codewords with certain combinatorial constraints like GC constraint, Hamming Distance, Reverse Hamming Distance and Thermodynamic stability. Those sets of codewords are expected to give a stable structure thus not including any secondary structure formation [1],[3],[4].

This objective of taking up this topic was to address both the issues of generating DNA self assembly using DNA computation and creating a platform which answers to the doubts of a layman about this topic and understanding of the various concepts that lead to the results. Also to provide him with one common module which could help him construct a codebook; check, understand and operate on grammar production rules and then interconnect these at the nano-level by converting this into stable free energy models. Also the tool provides with an online web page called 'Referential Study' which has links to the research papers, previous works of people, websites to be accessed for proper definitions and study for each topic and keyword used. The tool was aimed to extend a user friendly interface and at the same time provide with reliable results which could be used for experimental purposes.

In this report, we present the problem statement in the following manner. Section II deals with codebook, firstly telling what are the combinatorial constraints followed in generating a codebook i.e. GC constraint, thermodynamic stability, Hamming Distance based constraints. These constraints can be implemented in two Greedy algorithm and stochastic local search algorithm, which are explained under the next subsection- 'Approaches to generate codebook'. The next section about DNA Self Assembly and DNA Complexes explains the basic definitions of the grammar rules, which are check for in the next section using CYK and Tree Structure approach. Once the back-end understanding and modelling of the problem is done, the way in which it is implemented, dependent on other tools, available for access and prerequisites for its use, are explained in the next two sections. With this we come to the conclusion summarizing the project, the extent of the project objective achieved and scope for improvements in the future.

II. CODEBOOK

The codebook generation aims at creating a set DNA words which are single strand structures binding (hybridizing) with their complements in the expected manner. If the combinatorial constraints are not taken care of, even in controlled environment single strand DNA's may undergo non-selective and

unwanted hybridization or forming loops onto itself (forming secondary structures). These unexpected results lead to loss of data and makes our system inefficient in handling large amounts of data.
This motivates us to firstly have a deep study of the combinatorial constraints. And secondly, to design the algorithm which best implements these constraints. We here under our study consider two approaches for generating a codebook, Greedy Algorithm and Stochastic Local Search .

A. Combinatorial Constraints

1) GC Constraint: This constraints follows that the G and C nucleotide content is a fixed percentage of the nucleotide string length . In our code we have assumed it to be 50% as this value provides similar thermodynamic properties to the sequences [1],[3].

2) Hamming Distance Constraint: This constraint follows that for any two distinct words *u1, u2* in the codeword set, the number of positions at which *u1* differs from *u2* should be at least equal to a fixed term hd, where *hd* is fixed parameter. Hamming distance is the number of values of *k* for which, the k^{th} letter of *u1* should differ from the k^{th} letter of *u2* [3],[4].
H(u1,u2) > hd ; where hd is the fixed parameter and H(u1, u2) represents the hamming distance between u1,u2.

3) Reverser Hamming Distance Constraint: This constraint is similar to the Hamming Distance Constraint except that this time the reverse of *u1* and Watson Crick complement of *u2* are compared. This constraint follows that for any two distinct words *u1, u2* in the codeword set, the number of positions at which *rev(u1)* differs from *wc(u2*) should be at least equal to a fixed term *hd*, where *hd* is fixed parameter. Hamming distance is the number of values of *k* for which, the k^{th} letter of rev(u1) should differ from the k^{th} letter of *wc(u2)* [3][4].
HRc(u1, u2)> hd ; where hd is the fixed parameter and HRc() represents the hamming distance between u1, u2; wc(u2) gives the Watson Crick Complement of u2.rev(u1) gives the reverse of the string u1.

4) Thermodynamic Stability: Nussinov Algorithm: The computation using DNA becomes pointless if the data cannot be retrieved back. To prevent this from happening we try to find the DNA strand which does not form DNA secondary structures or pseudo knots within itself. One way of doing this is by calculating the minimum free energy of the sequence, through a dynamic programming Algorithm, known as Nussinov Algorithm. This calculates the value of minimum free energy by generating a matrix which saves the energy, E. Every cell (i, j) in the matrix corresponds to the free energy of the corresponding subsequence. This finally leads us to our result [2].

Algorithm 1 Nussinov Algorithm

procedure NussinovAlgorithm
Input: A,C,G,T string of length n for which the minimum free energy needs to be calculated
Output: The Free energy table and the minimum free energy

```
for i:= 0 to n do
    for i:= 0 to n do
        if E(i,0)= 'A' and E(0,j)='T' then
            a(i,j)← -1
        end if
        if E(i,0)= 'T' and E(0,j)='A' then
            a(i,j)← -1
        end if
        if E(i,0)= 'G' and E(0,j)='C' then
            a(i,j)← -2
        end if
        if E(i,0)= 'C' and E(0,j)='G' then
            a(i,j)← -2
        end if
    end for
end for
Stage two: this is stage two
for k:= 1 to n do
        j← 1
    for i:= k+1 to n do
        for p:= j+1 to i-1 do
            a1[cnt1] ← E(j,p)
            a2[cnt2] ← E(p,i)
            cnt1 ← cnt1+1
            cnt2 ← cnt2+1
        end for
        min1 ← smallest element of a1
        min2 ← smallest element of a2
        b1 ← smaller of min1 and min2
        a1 ← E(j+1,i-1) + a(j,i)
        E(j,i)← smaller of a1 and b1
        j← j+1
    end for
end for
minEnergy ← 0
for i:= 0 to n do
        j← 0
    while j ≤ n do
        if E(i, j) > minEnergy then
            j← j + 1
        end if
        if E(i, j) < minEnergy then
            minEnergy← E(i,j)
        end if
        j=j+1;
    end while
end for
return E and minEnergy
end Nussinov Algorithm
```

The following is the set of required conditions and the

algorithm to compute this energy matrix.

Initialisation :

1) E(i, i) = 0 ; where i = 0 to n
2) E(i, i-1) = 0 ; where i =1 to n
3) a(i, j) = -1 if E(i, j)='A' or'T'
4) a(i, j) = -2 if E(i, j)='G' or 'C'
5) a(i, j) = 0 otherwise

Induction Step:

$$\text{E(i , j)} = \min \begin{cases} E(i+1,j), \\ E(i,j-1), \\ E(i+1,j-1)+a(i,j), \\ \min_{i<k<j} [\ \text{E(i, k) + E(k+1 ,j)}]. \end{cases}$$

0	C	G	C	T	C	C	C	G
C	-2	-2	-2	-2	-2	-2	-2	-4
G	0	-2	-2	-2	-2	-2	-2	-2
C	0	0	0	0	0	0	0	-2
T	0	0	0	0	0	0	0	-2
C	0	0	0	0	0	0	0	-2
C	0	0	0	0	0	0	0	-2
C	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0	0

TABLE I: Matrix Generated by Nussinov Algorithm

According to the Matrix *Mininmum Free Energy* of "*CGCTCCCG*" is: -4

Conclusions that could be derived from this matrix is:

1) The lesser the minimum free energy, the more stable the structure hence forming secondary structure.
2) The longer the string the more tendency of forming pseudo knots.
3) The farther away the complementing nucleotides, the loop more stable.

Now, let us see the ways in which these constraints were implements to generated the codebok from random selections of DNA sequences.

B. Approaches to Generate Codebook

This explains the two approaches we used to generate codebook, the previous works and the improvisations done.

1) Greedy Algorithm: Greedy Algorithm is a design which emphasizes on optimizing substructures. The problem prioritizes in search and bound algorithms at each step. This approach can be referred to as a 'short sighted', in which we generate the permutation of all A,C,G,T strings of the given length giving us 4^n sets of codewords to operate on. Now implementing the definition, we keep reducing the String set, by checking pairs for the strings which do not satisfy the GC Constraint, Hamming and Reverse Hamming Constraints one by one. Hence resulting in O(n^2logn) complexity.
This Algorithm is efficient and results provide large number of Codewords. But the limitation comes in the speed of the computation of 4^n word set at initial stage, which increases in size exponentially as n increases. So even though we obtain large number of satisfying codewords in our codebook, the word size is restricted to 10. Secondly after a few iteration the results reach a stagnancy. This approach is preferable for applications where we need to just use the codebook and not generate one each tiem the code is run. It provides reliable results which could be used and reused.
*** The efficiency of this code may improve with the configuration of the System running the code.*

2) Stochastic Local Search Algorithm: Stochastic Local Search (SLS) Algorithm can be referred to as Randomized Decision making algorithm. The solution is initialized by generating a random DNA word set. Over this set, a pair of words are randomly selected and checked for the constraints, the pair found violating, called a 'conflict', is selected and modified so as the pair is now satisfying the constraints. Now another pair is selected and the steps repeated till either a set of words, all satisfying the given constraints is obtained or the iteration have reached a fixed count. The algorithm uses randomized iterative search strategy [4].
We in our project use the concepts of SLS and modify it to include the thermodynamic stability of the codeword also. The solution is initialized by generating a few random A,C,G,T string of the given word length n, which satisfies the GC Constraint (considering it 50 throughout the project). Now finding its minimum free energy, using the Nussinov Algorithm, for finding the probability of secondary structure formation. Hence proceeding to generate a set of Cyclic words from that codeword and using the approaches of SLS for finding the 'conflicts' violating Hamming Distance and Reverse Hamming Distance Constraints. Removing them we finally obtain the required word set.
This code is extremely fast and result in different results for the same word length. Also the limitation of word Length has been resolved, the code easily runs for strings as long as 120. The use of random string generation, checking the stability and GC of one string rather than an entire word set and then cyclic code generation makes the output more reliable. But this also restricts the number of words that could be generated.
To overcome this bottleneck, one approach that was proposed was to iterate the process till the number of codewords produced does not exceeds the required number. Another approach is to generate the words set of this randomly generated set using cyclic, quasi cyclic, quasi twisted methods, i.e. rotate the substring of variable lengths of the string in a cyclic manner to obtain new word strings. This could be a future improvisation for this given codebook methodology.
***Hence for the given project this algorithm gives best results for string beyond length 10.*

DNA Self Assembly would later use these codebooks to select codewords which would be be used to represent the Grammar rules and computation. The implementation grammar rules would be explained in the next section.

III. Self Assembly rules and DNA Complexes

Self Assembly model have close relationship with Grammar rules. Here we consider two grammars i.e. Regular Grammar and Context-Free grammar [1].

Regular Grammar is a grammar G=(V,T,S,P) if all production in P are in the form

$$A \rightarrow xB$$
$$A \rightarrow x$$

where A,B $\in Vandx \in T$
Context-Free Grammar is a grammar if production rule p are in the form of

$$A \rightarrow \beta$$
$$A \in Vand\beta \in T;$$

There is no restriction on β

Linear Self assembly can be represented by Regular Grammar and vice versa. Similarly Dendrimer Self assembly can be represented by Context-Free Grammar and vice versa.

IV. Approach for checking the grammar:

A. Cocke Younger Kasami (CYK) Parsing Algorithm:

It employs the concept of dynamic programming and bottom-up parsing. The only limitation of this algorithm is that the input Grammar should be in Chomsky normal form (CNF). A Context-Free Grammar is in Chomsky normal form when every Production rule is *P* is in the form

$$S \rightarrow AB$$
$$S \rightarrow \alpha,$$

where $\alpha \in$ T and (S, A, B)$\in$ V.
The complexity of CYK parsing algorithm is
$\bigcirc$(n) = n^3log(n).

Checking a String using Production rules by :

Production Rules for Grammar:

$$A \rightarrow a|BC$$
$$B \rightarrow b|AC$$
$$C \rightarrow a|b$$
$$S \rightarrow AB$$

Algorithm 2 CYK Algorithm

procedure : CYK algorithm
Input: The Grammar rules G and String S of length *n* characters s_1, $s_2 ... s_n$.
Output: Whether S satisfies G.
Initialisation: The matrix M[|; |] $\rightarrow$ 1, set of nonterminals at each j,i.
Step1:
Convert the grammar in Chomsky normal form.
Step2:
for j:=1 to |n| **do**
forall T[j], add T to the set M[j; j]
end for
for p:=1 to |n| **do**
for q:=1 to |n|-p **do**
for r:=0 to q-1 **do**
forall pairs B$\leftarrow n[r : q + r]; C \leftarrow$ n[q+r+1 : q+p]**do**
for all A $\rightarrow$ BC add A to the set stored in M[q; q +p]
end for
end for
end for
end for
if S $\in$ M[1;|n|] **then**
return true
else
return false
end if
end CYK Algorithm

Input : aabab

CYK Matrix :

[A][C]	[A][C]	[B][C]	[A][C]	[B][C]
[B]	[S][B]	[A]	[S][B]	
[S][A]	[A]	[S][B]		
[B]	[S][B]			
[S][S]				

TABLE II: Matrix Generated by CYK Algorithm

B. The Tree Approach

The second approach which we came across was by traversing a tree. This was supposed to be a recursive design, where the root is a null string, its exact children are the non-terminals and the corresponding derivations from those non-terminals form the children of those respective non-terminals. Once the tree is generated we begin at the root and traverse down to generates multiple permutations till strings of length equal to the input string is not obtained. For backtracking, we assign indexes to all nodes and keep saving the them whenever they are encountered during the course of the string checking. But this turned out to be an inefficient algorithm because of the

large number of unrequired strings formed during process, causing memory overflows. Hence this approach failed and was not brought into the implementation phase.

V. About the Tool

A. Graphical User Interface for DNAg

The graphical user interface (GUI) for DNAg is designed as a teaching tool which works as an interface between the various Input/Output operations that need to be performed by the user and the back- ground computational processing (generating codebook, checking string by production rules and construction of Self Assembled stable DNA structure) and . We have created the java applet in 3 modules.

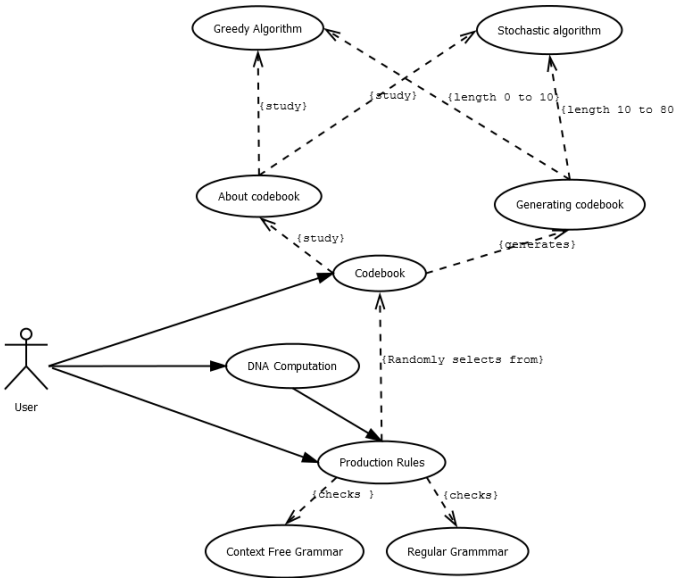

Fig. 1: Use Case Diagram for DNAg : This figure shows the structural flow of the functionalities of DNAg. A user has direct access to three modules, i.e.: production rules, codebook and DNA computation; which further extend the respective functionalities.The connectivity shown by solid lines represent direct access to the functionality, dotted lines show a flow to access it.

1. Codebook generation

The codebook generation can be done using two method, which are explained below, Fig.3 shows the options presented to the user for the same.

1) Greedy Algorithm: Input: User has to give word size i.e. user can enter any number from 1 to 10.

Output: The user will get the codebook of nucleotides A,C,G,T.

Back-ground process: applying combinatorial constraints i.e. GC constraints, Hamming Distance and Thermodynamic Stability on generated permutation.

2) Stochastic Algorithm: Input: User has to give word size i.e. user can enter any number from 10 to 80.

Output: The user will get the codebook of nucleotides A,C,G,T.

Back-ground process: generates a random string satisfying

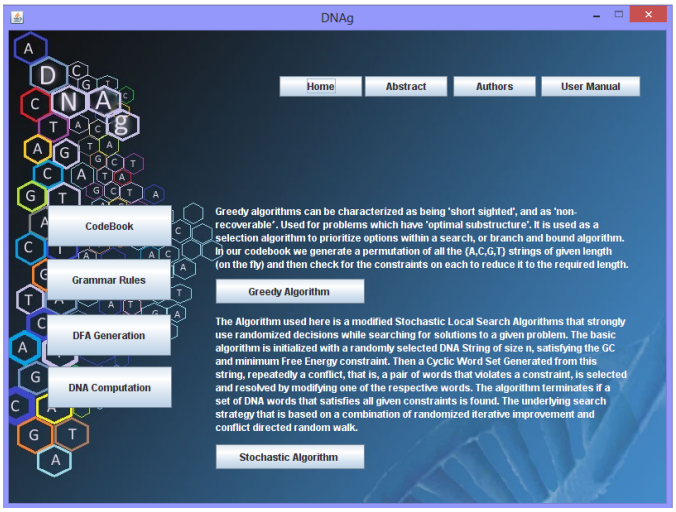

Fig. 2: Screen shot for the DNAg page which shows how the two options to generate a codebook and the details about each option available for the user to read and understand the difference between which codebook to be generated and preferred when, and the Back-End working behind each.

thermodynamic stability values of minimum free energy and GC constraint, this string is used to generate cyclic codes and the resulting set is checked for hamming distance and reverse Hamming Distance constraints.

Fig. 3: Screen Shot:Codebook Output for wordlength 6: Generated using greedy algorithm, on entering word length of 6. Greedy algorithm accepts word length from 1 to 10, and stochastic from 10 to 80. Choosing the option depends on the user's requirements.

2. Checking String by production rules

Input: set of production rules of Grammar and a string. The Grammar should be either Context-Free Grammar or Regular Grammar.

Output: Whether the input string is the member of Grammar or not.

Back-Ground process: Converting grammar into Chomsky normal form then applying CYK algorithm to the resulting production rules.

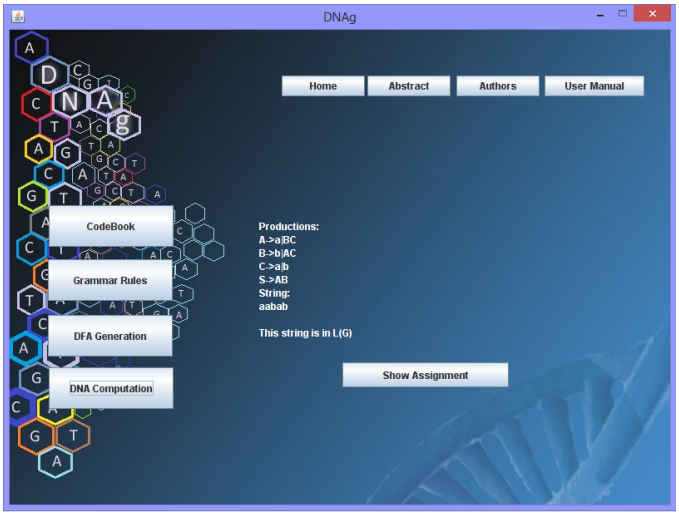

Fig. 4: Screen Shot: Output for Production Rules: A→a|BC, B →b|AC, C→a|b, S→AB and input String: 'aabab'. Here, as the input string satisfies the input Grammar, the button "Show Assignment" appears by which one can see the A,C,G,T code assigned to the input string through the random assignment of codebook

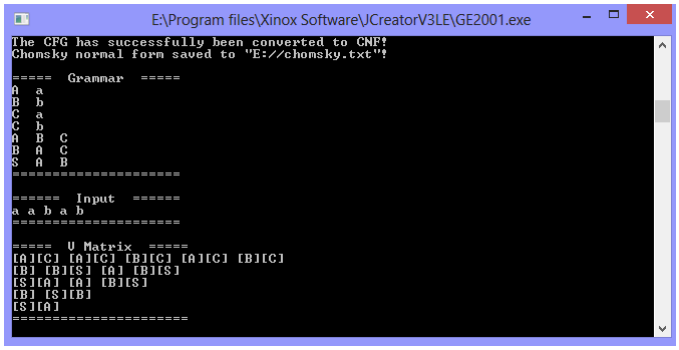

Fig. 5: Screen shots: CYK Matrix formed in the Back-End, according to the value with which the table is filled, S in the variable and the input String satisfies the given grammar rules. This could be back traced to give the productions sequences which result in this sequence.

3. Representation of the self assembly of DNA compute

This Module is actually merge of first two module. It shows how Grammar, Self assembly and Codebook are related.
Input: In order to view the structure the user has to give correct grammar and string.
Output: if grammar is correct the structure get display other wise user have to try again.
Background process: When user gave input the grammar get checked using module 2. The input string and production rules are checked for the number variables or terminals. A cdebook of random wordlength is generated and codewords are selected and assigned to represent the computation. this concatenated result is then passed to the Nupack software[5] and the stable structure is displayed.

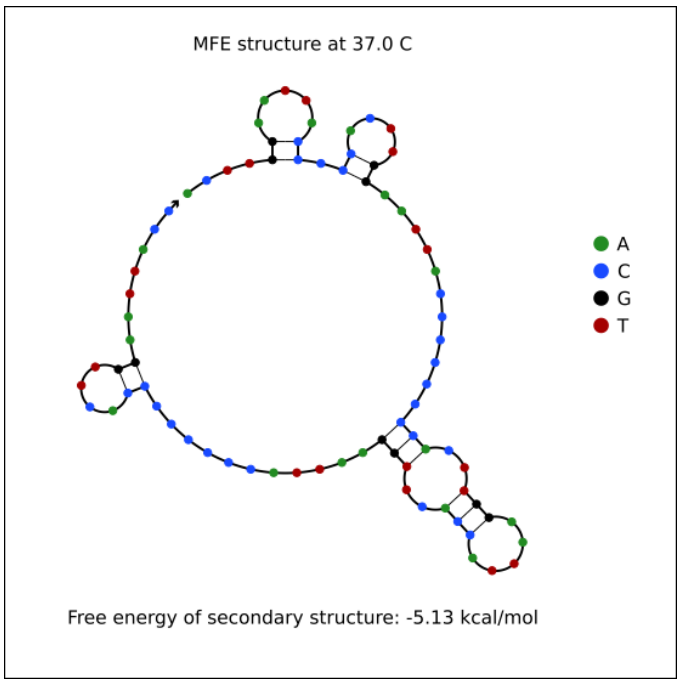

Fig. 6: Screen shots: Final DNA Self Assembly structure, with maximum stabilty and given free energy generated in the figure. Here the string is aabab. By random selection from codebook code for a is "ACTTGGAATTACCCCC" and code for b is "CCCACTTGGAATTACC". The structure has just four pseudo knots and this shows a structure which is stable on average. The stability can be measured according to the -5.13k kcal/mol as shown below in the figure.

B. Dependencies

This tool depends on external software for improvised representations and additional learning functionalities.

1) Nupack : Nucleic Acid Package: It is a Software suite, which formulates algorithms to generate secondary strucutres for nucleic acids, showing thermodynamic stability. We use this software to represent the resulting string of the DNA computation in its most stable form[5].

2) Exorciser: Exorciser is a Teaching tool. It implements in details the concepts of our Introductory course "The Models of Computation". Since a grammars and production rules form a main role in our project formulation, we liked the idea to introduce a teaching tool which helps understand these concepts better. Developing one was not in the scope of this project. So we imported the software 'exorcise' to one add to the functionalities of our project and secondly presenting a taste of what the entire project could have incorporated.

**we do not take credit for the work done in the 'exorcise' software; this is just to make the learning experience of the user more interesting and to give them, this one software, with as many functionalities as possible[6].

VI. Software Availability

The Software DNAg can be downloaded from *http://www.guptalab.org/DNAg*. The access to the online user manual and referential study is available through the Software.

A. Prerequisites for using the Software:

Your System should have:

- internet connectivity.
- phantom js and casperjs Installed.
- Environmental variables should be set accordingly.

VII. Conclusion

As a whole the project was successful in generating codebook with variations in wordlength. Checking the correctness of grammar and acceptance of a string by the language. Selecting a set of thermally most stable words, to generate DNA codebook which could hence be used to represent the DNA computational self assembly for regular grammar rules and particular input string.

During the course of this project we faced various problems related to the limitations of an algorithm. This forced us to introduce new approach to solve them and also provide multiple options at small stages to make the code more optimized. Like when greedy algorithm was unable to deliver long length sequences, we had to include stochastic algorithm. Similar CYK algorithm was modified to expand its scope of accepting productions rules in forms other than CNF. Nussinov Algorithm[3] was implemented to find the thermal stability but to generate the most stable structure by including the pseudo knots, we used the software of nupack[5].

In the future options for generating codebook of multiple lengths, increasing the number of sequences in Stochastic-ally generated codebook by arranging them according to their thermal stability could be worked upon. Also the present version of the Software computes best on Regular languages and so to extend it context free, context sensitive grammars shall be an important objective to be served.

Hence the project could be called as one which is versatile, flexible and leads to scope of adding more functionalities in future to make it more efficient and diverse in its field of implementation.

VIII. Acknowledgment

We would like express our gratitude to *Prof. Manish .K.Gupta*, our BTP Mentor and Coordinator, for his constant guidance, support and encouragement through the course of this project. We are also grateful to *Anurag Nigam* and *Pallav Vyas*, as their work in the field of Stochastic Local Search Algorithm provided a new approach to implement. Our tool would have been incomplete without the use of *Nupack (Nucleic Acid Package)* and *Exorciser Softwares* and we would like to acknowledge the team behind these Softwares for making our work accessible for further use in Research and teaching purpose.

IX. Contributions

The project has been combinedly done by the two members of the group in complete interdependence. The modules which were individually formulated were:

Sonam Jain
ID- 200901118

1) Greedy Algorithm
2) CYK Customization
3) Intergrating GUI with backend

Priyanka Shukla
ID- 200901102

1) Stochastic Local Search Algorithm
2) Nussinov Algorithm
3) GUI Design

References

[1] E. Winfree, “Algorithmic Self-Assembly of DNA,” PhD. thesis, California Institute of Technology, Pasadena, CA, 1998.
[2] AmitMarathe, Anne Condon, Robert M. Corn “On Combinatorial DNA Word Design.” Journal of Computational Biology 8(3): pp 201-219, Nov. 1999.
[3] Olgica Milenkovic, NavinKashyap , “DNA Codes that Avoid Secondary Structures,” Proc. 2005 IEEE International Symposium on Information Theory, Adelaide, Australia, Sept. 4-9(2005).
[4] Dan C. Tulpan, Holger H. Hoos and Anne E. Condon,“Stochastic local search algorithms for DNA word design,” Proceeding DNA8 Revised Papers from the 8th International workshop on DNA Based Computers : DNA Computing, Pages 229-241, Springer-VerlagLondon,UK (2003).
[5] http://www.nupack.org/.
[6] http://www.educeth.ch/exorciser.
[7] code.google.com/p/cfgtocnf/source/browser/?r=2.