Planning & Sampling

Sai Srinadhu Katta & Venkata Sainath Thota

1 Planning

1.1 Forward planning

In this part of lab forward planning is implemented to find the goal. Initially the initial state is given as input. From the initial state all the legal possible actions are applied to the state and added the states into a queue. BFS is applied and the tree is expanded until the given goal in achieved i.e goal becomes a part of the tree.To represent a state we used a integer to know the position of shakey i.e on which box or floor.Using a string of 0's and 1's the status of lights are coded. Using a vector of boxes the position of boxes are coded. Similarly parent for each node and the action which lead to the given node is coded. The snapshot of the state representation is given below.

```
class state{
public:
  string lightson;
  string action;                                    //contains the light status
  vector <int> AT;                                    //contains the shakey ↩
    position and box
  int shakey=0;
  state *parent=NULL;
public:
  state(string s,vector<int>A,int floor,string D){
    lightson=s;
    shakey=floor;
    AT=A;
    action=D;
  }

};
```

Table conating the number of nodes expanded in BFS and length of plan.

file	Number of Nodes	length of plan
1.txt	14	5
3.txt	149	6
6.txt	284	8
7.txt	96	8
9.txt	597	10

1.2 Goal Stack Planning

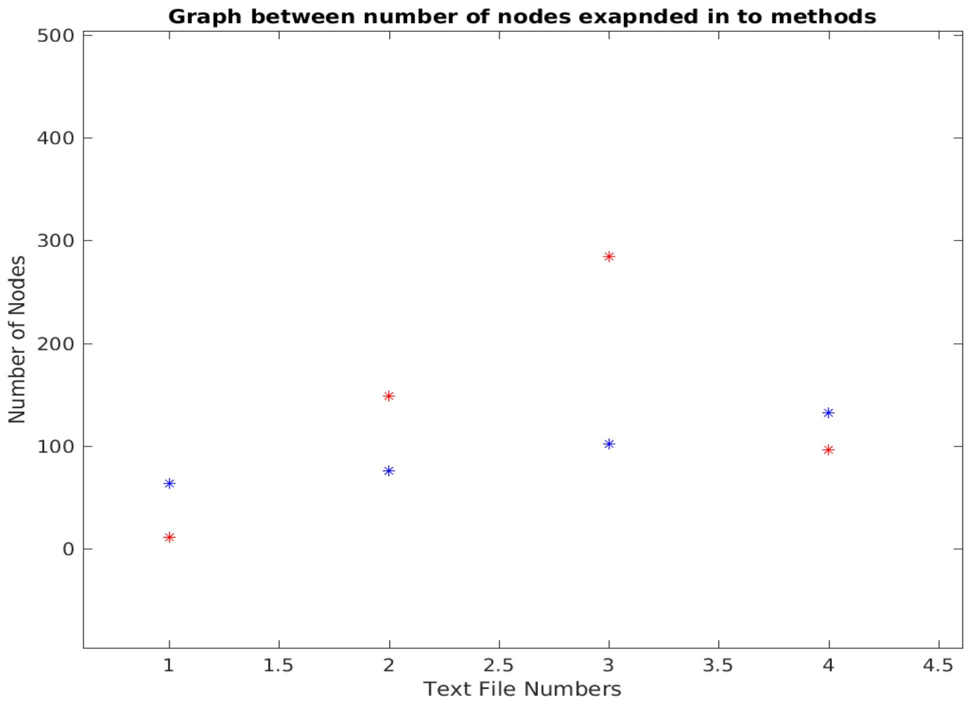
In this part of lab the problem is solved using Goal stack planning. Goal Stack planning is faster than the forward planning and requires much less memory. The number nodes that have to pushed into the stack will be usually less than the number of nodes added into the queue in forward planning. Generally Goal stack planning takes less time to reach the goal than the forward planning. The number of nodes expanded in the goal stack planning and the length of plan is given below in the table.

file	Number of Nodes	length of plan
2.txt	64	9
4.txt	76	11
5.txt	102	15
8.txt	132	19
10.txt	98	14

1.3 Analysis

We can observe from the above tables that the number for nodes that are expanded for a problem is higher for forward planning than the Goal stack planning. Goal Stack planning is faster than the forward planning and requires much less memory but returns suboptimal plan. It is evident from the number of nodes that forward planning requires lot of memory than the Goal stack planning. When we see the length of plan we can observe that lot of unnecessary steps will appear in the Goal stack planning than the Forward planning. The difference in the nodes are given below.

file	Forward Planning	Goalstack Planning
1_2.txt	14	64
3_4.txt	149	76
5_6.txt	284	102
7_8.txt	96	132
9_10.txt	597	98



Actions required to reach the goal for both the methods for 1.txt is given below.

```
"FORWARD PLANNING"
push(1,1,2)
climbup(1,2)
turnon(2)
climbdown(1)
push(1,2,1)

"GOAL STACK PLANNING"
climbup(1,1)
climbdown(1)
go(1,2)
go(2,1)
push(1,1,2)
climbup(1,2)
turnon(2)
climbdown(1)
push(1,2,1)
```

2 Sampling

In this part of lab, density estimation using Gibbs Sampling is done. We are provided with Adult income dataset [2] as train data and test data. We are already provided with Bayes Net on the train data. Using this Bayes Net, Gibbs Sampler will generate samples, then for each data-point in test data probability with Bayes Net and probability from sample generation will be compared. Mean squared error is used as measure in all the below plots.

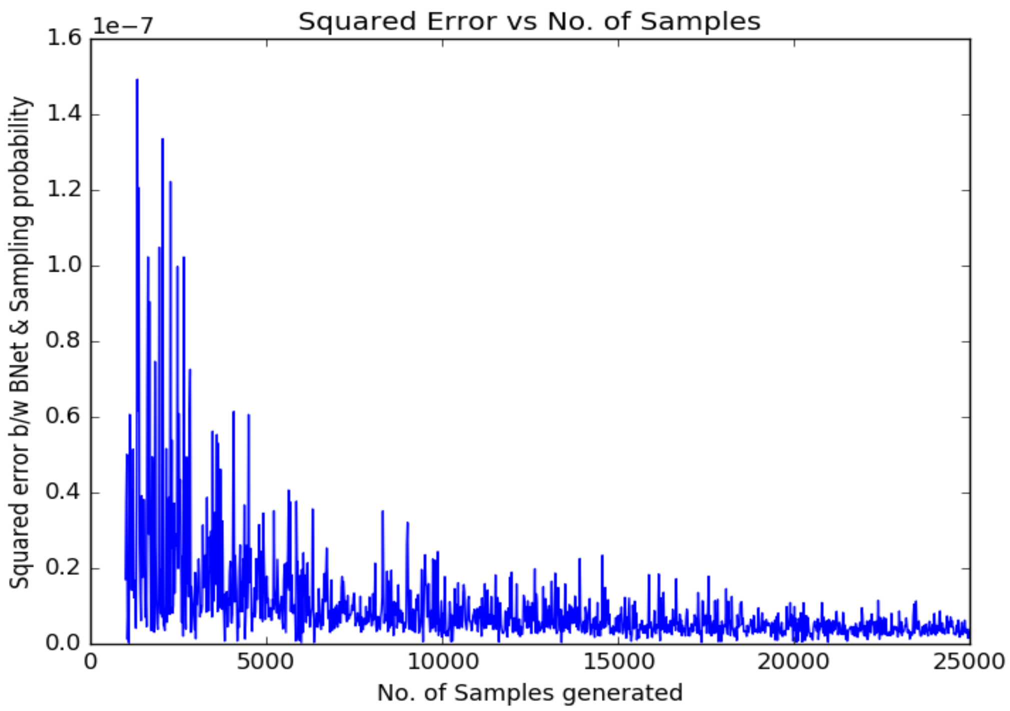
2.1 Gibbs Sampler

```python
def Gibbs_Sampling(marginals, joints, trijoints, bayes_net_prob, test_samples, burn_in = 10, num_samples = 10000, num_features = 14, verbose = True):
    """
      Samples the given number of samples and returns the mean squared error from Bayes Net and Sampling for Test dataset. All about inputs and outputs is present in the code.
    """
    samples = [] #all the samples
    sample_init = sample_intilization(marginals, joints, trijoints, num_features = 14)
    samples.append(sample_init)
    sample = copy.deepcopy(sample_init)

    for i in range(num_samples): #get those many samples
        for j in range(burn_in): #wait till burn_in samples
            k = random.randrange(num_features)
            sample_init = sample_next_feature(marginals, joints, trijoints, sample, k)
            sample = copy.deepcopy(sample_init)

        #generate a sample and add it
        feature_index = random.randrange(num_features)
        sample_init=sample_next_feature(marginals,joints,trijoints,sample,feature_num=feature_index)

        sample = copy.deepcopy(sample_init)
        samples.append(sample)

        if (i%100 == 0 and verbose):
            print "iteration " + str(i) + " done."

    test_prob = Test_Sample_Prob(test_samples, samples)
    error_estimate  = Error_Sampling(bayes_net_prob, test_prob)

    return error_estimate
```

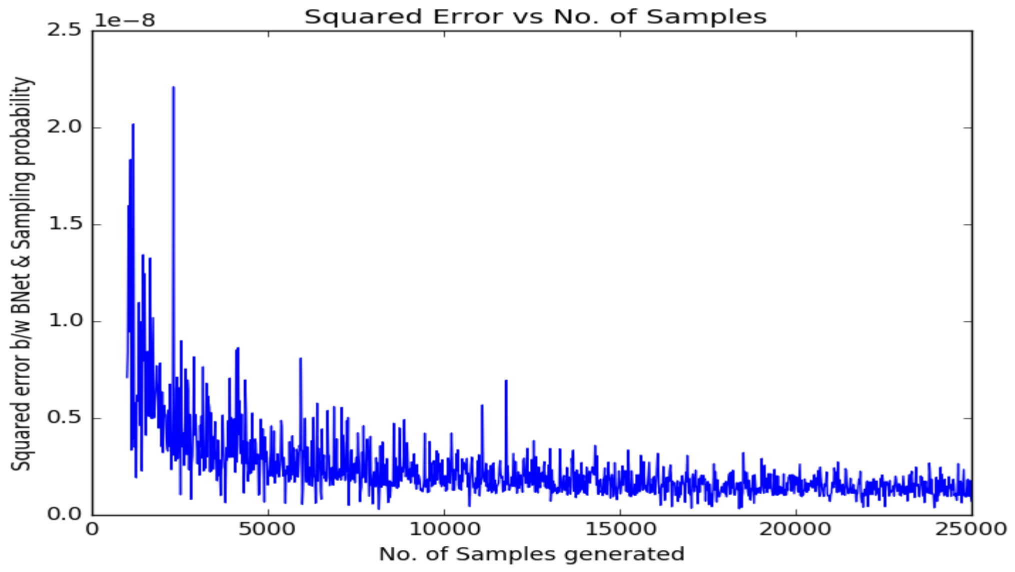
Code Snippet for Gibbs Sampler.

2.2 Plots & Observations

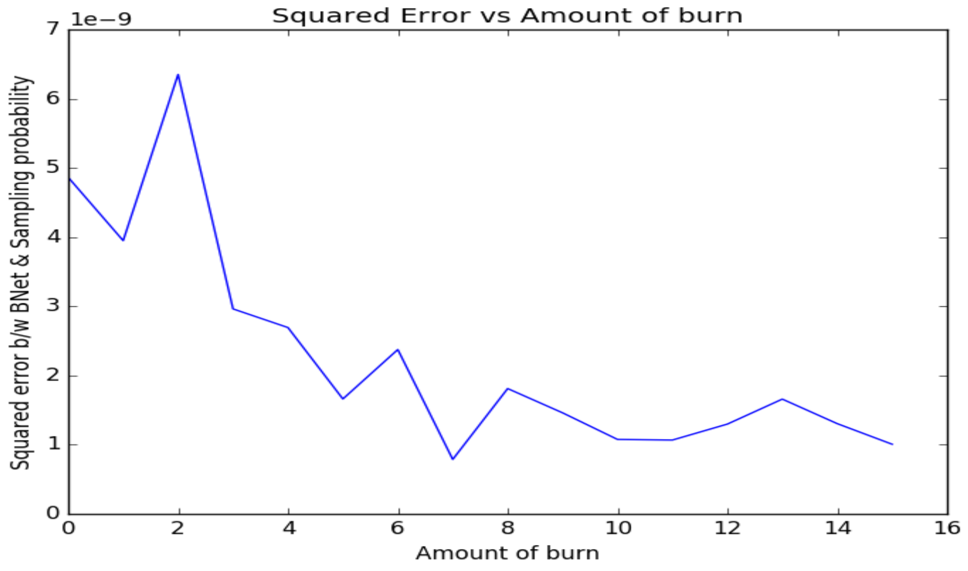
In the first plot burn_in set to 0, which means gibbs sampler's every sample will be considered in our sampled data, which may not be great since as such two samples picked one after another aren't truly independent. I started with 1000 samples and went till 25000 samples with a gap of 20. Some observations are that this plot is clearly more noisy compared to next one where we are using burn of 10 for which reason might be, not so much independence from samples. This has an advantage that this runs much faster compared to one which has burn since effective number of samples picked here is very less compared to one having burn. As the number of samples increases error decreases first and then almost goes to constant(decreases very less) which is intuitively expected.



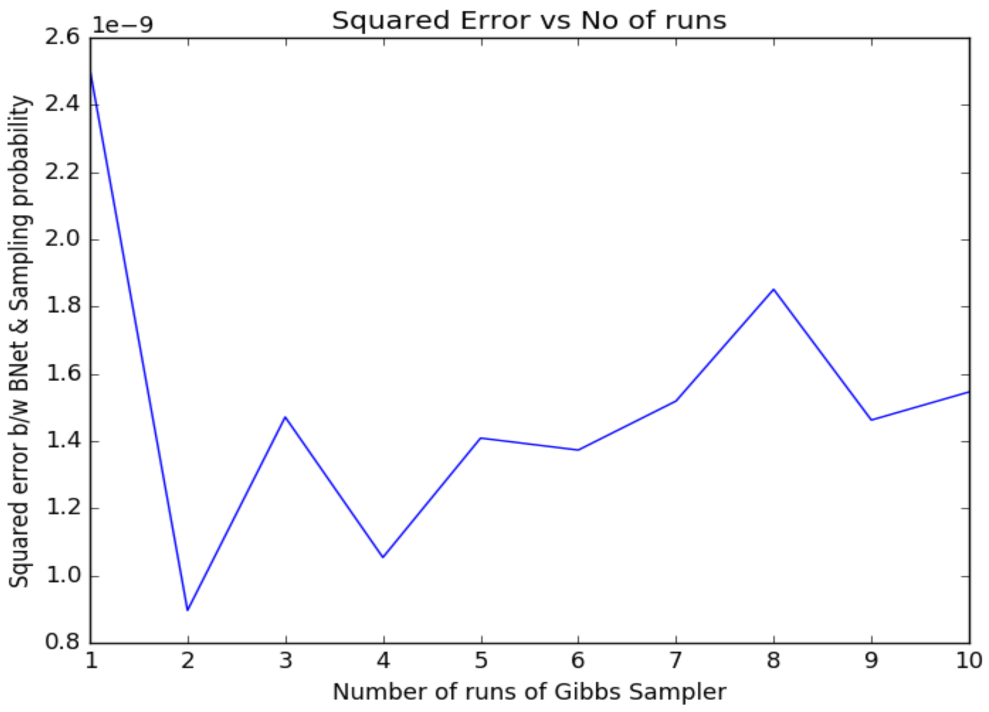
In the second plot burn_in set to 10, which means for gibbs sampler's every sample first burn_in number of samples are rejected and then next sample will be considered in our sampled data, which is better than without burn_in since samples now will be more independent or comes from many parts of joint probability space unlike without burn whose samples come from mostly localized spaces in joint probability distribution. I started with 1000 samples and went till 25000 samples with a gap of 20. Some observations are that this plot is clearly far less noisy compared to first one where we are not using burn for which reason might be, better independence of samples than without burn_in. This has an dis-advantage that this runs much slower compared to one which has no burn since effective number of samples picked here is very huge. As the number of samples increases error decreases first and then almost goes to constant(decreases very less) which is intuitively expected.



In the third plot burn_in is varied from 0 to 15 and 20000 samples are picked for each burn_in. The error measure is of mean squared. More the burn better it is, up's and down's can be due to stochastic behaviour of sampler and they aren't very drastic and so it's not a big concern for us. From the plot around burn_in of 10 seems to be great and so it is taken as burn_in in next plot. This also tells having a non-zero burn_in is usually better.



In the final plot burn_in is fixed to 10 and 20000 samples are picked. The number of runs of sampler are varied from 1 to 10 and error is averaged out based on runs. The error measure is of mean squared. More the runs better it is, up's and down's can be due to stochastic behaviour of sampler and they aren't very drastic and so it's not a big concern for us. More times sampler is ran, errors tend to get averaged out and better estimate of errors comes for us.



The main observation from this is that for a good sampler fix a burn_in (say around 10) and fix number of samples as high as possible (say 20000), re-run the sampler as many times as possible (say around 5-10 times) and average out the error to get better and more accurate estimate of error on test data.

References

[1] LaTeX Templates for Laboratory Reports,
`https://github.com/mgius/calpoly_csc300_templates`

[2] Adult income dataset.
`https://archive.ics.uci.edu/ml/datasets/adult`