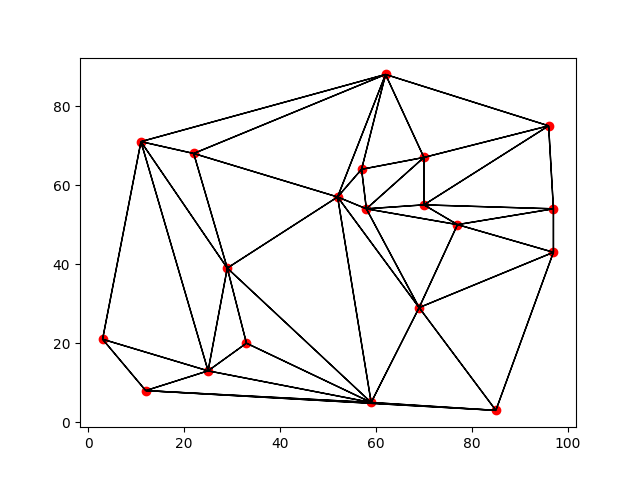
**Constraint Satisfaction Problem on the planar graph generated by a set of random points inside a unit square**

**The problem comprised two steps:**

Graph generation: I generated a set of 20, 30, 40, 50 till 60 points in the unit square. A typical representation of the points in the space can be found by executing:

python PointerGenerationV5.py from the terminal.

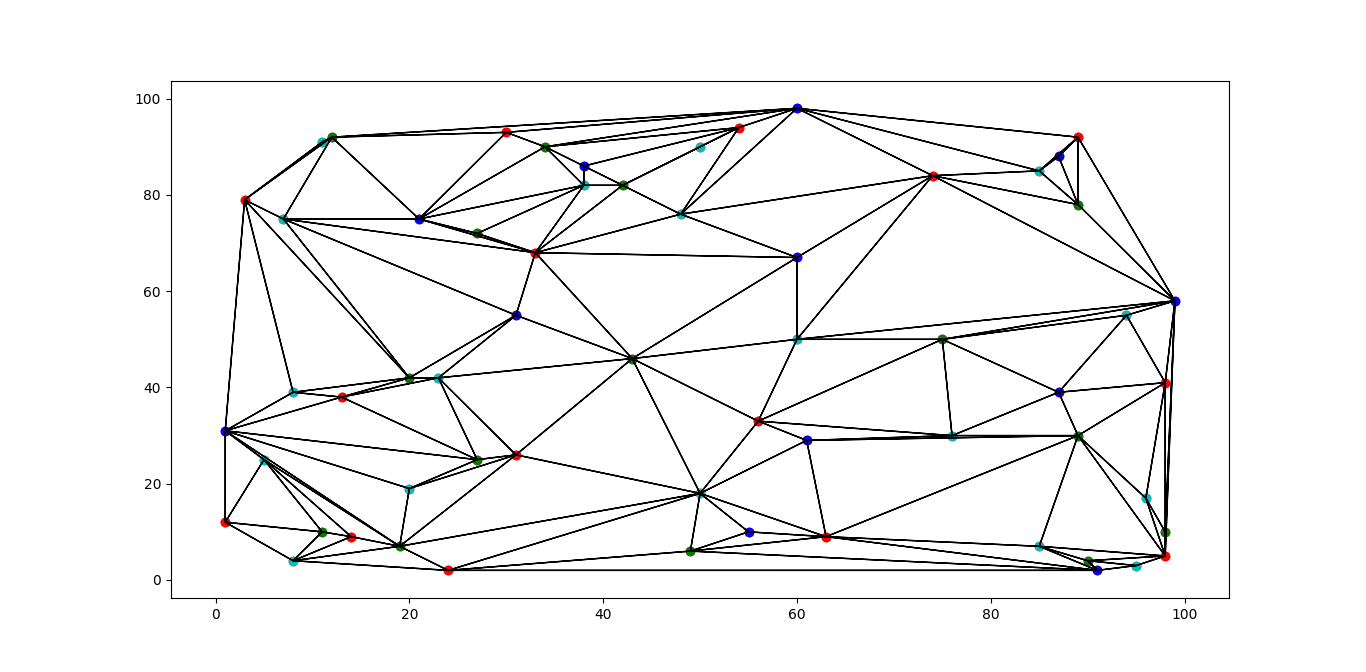
An example graph is provided below. The graph is planar and has 20 points on the unit square.

 The time required to generate the graphs is very negligible for sizes up to 60.

**Back Tracking with MRV, LCV, Degree Heuristics**:

Here I employed the standard backtracking procedure from Bartak’s paper. I added separate methods to select variables based on their minimum remaining values. Once the variable was selected, I created a method to determine a look ahead type method where I pre-assigned the variable with a value and saw how much it decreased the domain of other variables. The value which decremented the domain of other variables by the least amount was chosen as an assignment. For the starting point, the variable with the most number of constraints was chosen as the starting variable. The different statistics reported are as follows for K = 3 and K = 4 over 10 average runs for each. Note that for both k = 3 and k = 4, there were several instances where coloring was not possible.

|  |  |  |
| --- | --- | --- |
| Number of Nodes | Average Memory | Average Time in seconds |
| 20 | 2.456 MB | 0.674 |
| 30 | 3.67MB | 0.873 |
| 40 | 6.7MB | 1.45 |
| 60 | 9.951MB | 2.502 |



The 4 coloring for one of the 60 nodes case is provided here. It is to be noted here that we cannot always get a 3 or 4 coloring for every planar graph

**Back Tracking with Maintaining Arc Consistency**:

Here the MAC technique was applied in the problem. As stated in the assignment, once a particular node was assigned a value, all other nodes which are neighbors of that node are checked for arc consistency in the form of where is the node which has been already assigned a value and is a neighbor of it and has not been assigned a value. This way the domain of values of the unassigned variable reduces.

OBSERVATION: What I noticed was that the MAC algorithm for this problem was basically similar to the problem 1 because it caused the domain reduction. Mainly because the assignment instructions wanted me to implement the MAC algorithm at each stage of value assignment of a variable rather than at the beginning of problem. This is different from a standard MAC procedure where constraint propagation is done using MAC right before the BT algorithm is implemented. As such the run times of the algorithm were not widely different from the earlier one.

METHOD OF IMPLEMENTATION: I implemented the MAC algorithm by creating a data structure where I could store both the Variables and their respective domain inside a dictionary. The Adjacency List was also created comprising the points and their connections in the Point generation stage. I also kept track of the space of points and their corresponding indices in another dictionary. This helped me to quickly reference the points in space without trading off memory. This also did not take up much space because the all the other computations were stored in lists which require significantly less memory than dictionaries. The only time consuming operation in the entire stage of operation were the deep copying the Adjacency List for the graph coloring.

The results are displayed below for statistical comparison for k = 3 and k = 4

|  |  |  |
| --- | --- | --- |
| Number of Nodes | Average Memory | Average Time in seconds |
| 20 | 1.72 MB | 0.781 |
| 30 | 2.98MB | 1.002 |
| 40 | 4.3MB | 1.435 |
| 60 | 6.21MB | 2.083 |

NB: It should however be kept in mind that I have represented the graph very efficiently with respect to the search method. That’s why it takes more memory and less time. For 60 points, I never ran into memory issues or it never happened that the program did not terminate.

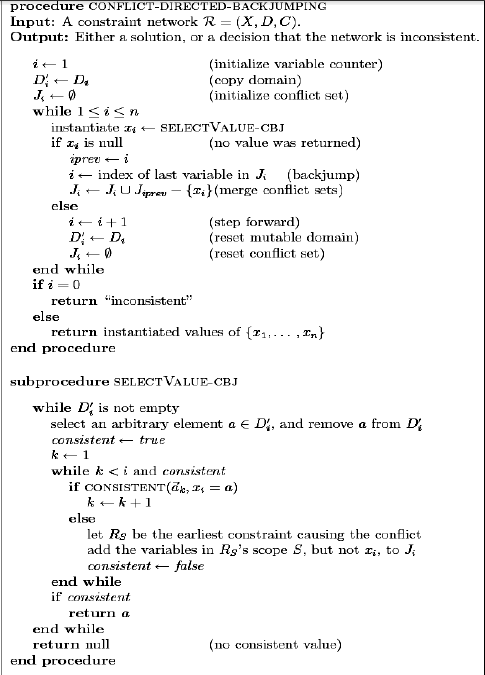
**BT with CONFLICT DIRECTED BACKJUMPING**:

Here I have employed conflict directed back jumping using PROSSER’s methodology.

I am attaching the pseudo code as this the one I have implemented. As the instructions for the CDBJ were not clear this the best I could find. Since the idea of BT and CDBJ are a little different with respect to the backing up part I did not use the idea of BT in this section for the sake of any confusion because in general the BT is implemented has already been implemented.

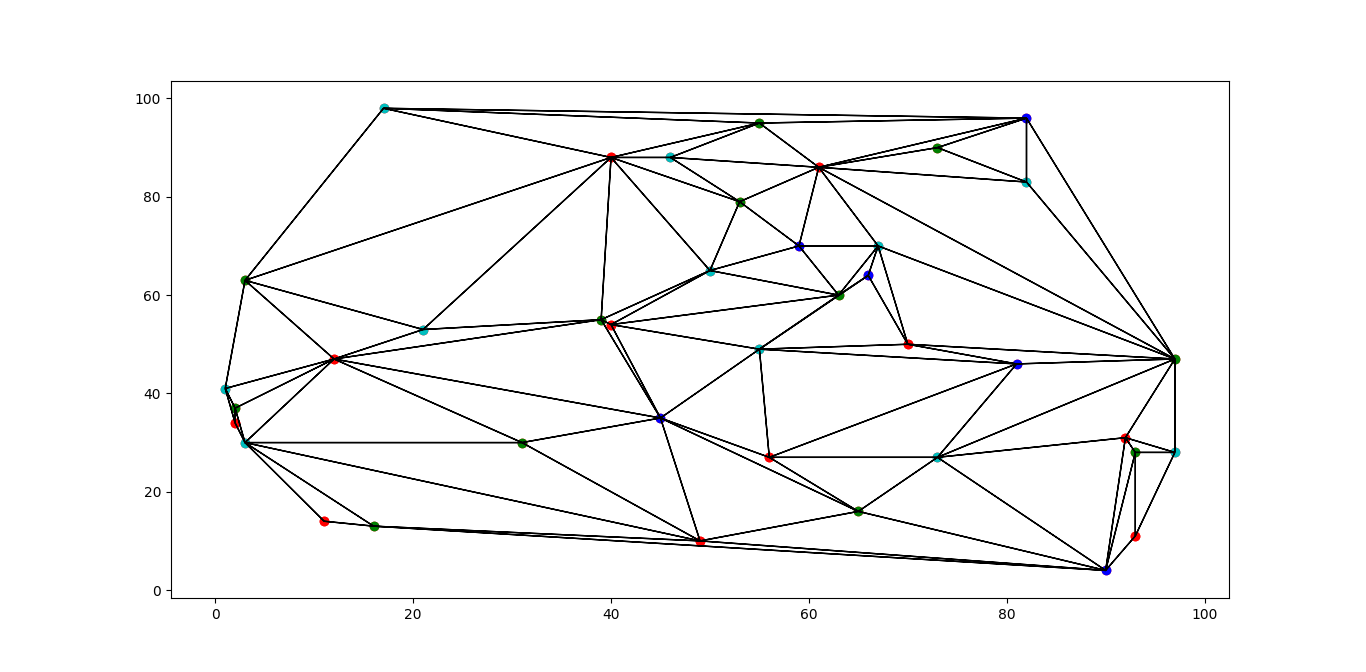
The caveat over here is that CDBJ helped me to jump back to nodes which were earlier in the constraint violation with the current label and jump back directly to it. In the process I optimized on some time lost for the futile search. The main advantage of CDBJ is in the way in which it keeps track of the conflicts even when it backs up the results. **For example in my problem when I used the step by step debugger to see the program execution I noticed that that it had efficiently identified the source of the conflict and backed up in some cases to 5 levels above it whereas BT would only go to the upper level.**

Continued on next page.



The results for the CDBJ are displayed below for the case of k = 3 and k = 4 points.

|  |  |  |
| --- | --- | --- |
| Number of Nodes | Average Memory | Average Time in seconds |
| 20 | 3.486 MB | 0.591 |
| 30 | 5.305MB | 0.734 |
| 40 | 12.647MB | 1.521 |
| 60 | 18.409MB | 1.854 |

The result for the 60 node case was node improved much which I could not interpret why. The 4 coloring for one of the 60 nodes case is provided here. It is to be noted here that we cannot always get a 3 or 4 coloring for every planar graph.

CONCLUSION:

* In solving this problem one of the main challenges was to develop a planar graph for the problem to work on.
* Next I had to think of efficient data structure representations for solving the time constraints when the number of points is large.
* I implemented a doubly linked list named “Adjacent List” and “Sink List”. The former kept track of points which came out of a particular graph while the latter kept track of points which are which terminated in the different vertices within the graph. These lists were tracked easily by maintaining a dictionary where the keys were the points in tuple format and the values were the indices to the Adjacent and Sink List.
* The other data structure I had to use was a constraint list where each constraint was represented as a binary tuple. This meant that these two points were connected in a planar graph.
* The fast membership testing method for constraints which had variables which were only labeled was implemented in linear time previously. I ordered them like the Adjacency list and accessed them if they were present in the set of labeled variables using the dictionary again. This reduced the membership testing from linear to constant time.