Personal Details

Name: *Alfred Ajay Aureate Rajakumar*

Country: *India*

University/College:
Department of Electrical Engineering,
Indian Institute of Technology - Madras (IIT-M), Chennai, Tamil Nadu.
University's homepage: http://www.iitm.ac.in/

Years completed: *4*

Current program: *I'm currently in my 4th year (pre-final) of Dual Degree (B.Tech in Electrical Engineering and M.Tech in Microelectronics and VLSI Design).*

Email: alfredajayaureate@gmail.com, ee10b052@smail.iitm.ac.in *or* ee10b052@ee.iitm.ac.in

Phone: *+919600561403, +918939177976*

Other contact channels:
Google+ account: https://plus.google.com/112686277522492098187/posts *(or just search for - Alfred Ajay Aureate Rajakumar)*
Skype ccount: alfredaureate

Personal website: *NA*

Project Description

Project title: *Preprocessing for Steiner Tree Problems*

Detailed description:

Introduction:

Given an undirected graph (or a network) $G = (V, E)$ and a node set $T \subseteq V$, a Steiner tree for T in G is a subset (or a subnetwork) $S \subseteq E$ of the edges such that $(V(S), S)$ contains a path from s to t for all $s, t \in T$, where $V(S)$ denotes the set of nodes incident to an edge in S. In other words, a Steiner tree is an edge set S that spans T. The Steiner tree problem is to find a minimal Steiner tree with respect to some given edge costs $c(e)$, $e \in E$. This is a classical NP-hard problem with many important applications in network design in general and VLSI design in particular.

Preprocessing (or reduction):

Preprocessing is a very important algorithmic tool in solving combinatorial and integer programming problems of large scale. The idea, in general, is to detect unnecessary information in the problem description and to reduce the size of the problem by logical implications. For the
Steiner tree problem, many reduction methods are discussed in the literature and have been shown to be very effective for solving large instances.

Here, preprocessing is done using various reduction tests, that suggests removal of certain edges, vertices, etc. based on suitable constraints or bounds. Firstly, I would like to implement efficient realizations of some classical tests, and then implement some new tests, filling some of the gaps the classical tests had left.

Actually each test is specially effective on a certain type of instances, having less (or even no) effect on some other ones, so it is important to have a large collection of tests at one's disposal. Finally, I will integrate these tests into a packet, using some non-trivial techniques described below. It should be emphasized that the most impressive achievements of reductions are mainly due to the interaction of different tests, achieving results which are incomparable to those each single test could achieve on the same instance on its own.

There are two major classes of reduction tests:

1) *The alternative-based tests use the existence of alternative solutions. For example in case of exclusion tests, it is shown that for any solution containing a certain part of the graph (e.g. a vertex or an edge) there is an alternative solution of no greater cost without this part; the inclusion tests use the converse argument. Some of these tests could be realized in time O(e + n log n), where e:=|E|, n:=|V|*
2) *The bound-based tests use a lower bound for the value of an optimal solution under the assumption that a certain part of the graph is contained (in case of exclusion tests) or is not contained (in case of inclusion tests) in the solution; these tests are successful if such a lower bound exceeds a known upper bound.*

Definitions and notations:

i) *A bottle neck of a path P is a longest edge in P .*
ii) *The bottle neck distance b(vi, vj) or bij between two vertices vi and vj in G is the minimum bottle neck length taken over all paths between vi and vj in G.*
iii) *The restricted bottleneck distance rb(vi , vj) or rbij between vi and vj in G is the bottleneck distance between vi and vj in G − (vi , vj).*
iv) *An elementary path is a path in which only the endpoints may be terminals. Actually, any path between two vertices can be broken at inner terminals into one or more elementary paths.*
v) *The Steiner distance along a path P between vi and vj is the length of a longest elementary path in P.*

vi) *The bottleneck Steiner distance (sometimes also called "special distance") s(vi, vj) or sij between vi and vj in G is the minimum Steiner distance taken over all paths between vi and vj in G.*

vii) *The restricted bottleneck Steiner distance rs(vi, vj) or rsij between vi and vj in G is the bottleneck Steiner distance between vi and vj in G – (vi, vj).*

Actually, if T = V, then bij = sij and rsij = rbij. In the context of alternative-based reductions, the notion of bottle neck Steiner distances (also called special distances) is often helpful.

viii) *In a (Steiner) tree T, non-terminals of degree at least 3 (in T) and terminals are considered as key nodes.*

ix) *A key path in T is a path in which (only) the endpoints are key nodes.*

x) *The unique path in T between two nodes vi and vj is called the fundamental path between vi and vj.*

xi) *A tree bottleneck between two nodes vi and vj in T is a longest subpath on the fundamental path between vi and vj in which only the endpoints may be key nodes.*

xii) *For each terminal z, one can define a neighborhood N(z) as the set of vertices which are not closer to any other terminal. If v ∈ N (z), we call z the base of v (written base(v)). we call N (z) the Voronoi region of z.*

I will implement modules to get the above mentioned parameters for a graph, tree, path, edge or a vertex, for those that aren't there in OGDF library, like a method for finding the bottleneck steiner distance between two vertices, etc.

Consider two terminals zi and zj as neighbors if there is an edge (vk, vl) with vk ∈ N (zi) and vl ∈ N (zj). Given G and T, the Voronoi regions can be computed in time O(e + n log n). Using them, a minimum spanning tree for the corresponding distance network DG (R) (also denoted by TD (R)) can be computed in the same time.

<u>Reduction tests:</u>

A separate module for each of these tests would be implemented, with classes having different parameters to costs, steiner distances, etc. for an edge or for a vertex.

1) Degree-Test I:

The following tests summarized under the name degree-test I are easy to check:

(i) A nonterminal node of degree one can be removed.

(ii) If a nonterminal node v is of degree two, with the two incident edges [u,v] and [v,w], then those two edges can be replaced by an edge connecting u and w, [u,w] of cost c[u,w] = c[u,v] + c[v,w].

(iii) An edge incident to a terminal node of degree one is always in an optimal solution.

(iv) *If an edge e is of minimal cost among the edges incident to a terminal node, and* *the other end node is also a terminal, then e is choosable in any optimal solution.*

2) Special-Distance-Test

This test computes for each pair of nodes a number (called the special distance) which can be exploited to remove some edges. The special distance can be computed by a modified shortest path algorithm. Given the values s(u,v) for all u, v ε V, there is an easy and very effective test for deleting edges. An optimal solution S of a Steiner tree problem ST(G, T, c) cannot contain any edge [u, v] Є E with s(u,v) < c(u,v). Concerning implementation, it should be noted that certain special cases of this test can be implemented more efficiently. However, one can also resort to a well-performing approximation of the special-distance-test that runs in O(|V | log |V| + |E| + (|T|^2))*

3) Bottleneck Degree m Test:

This might create many new edges, but, in general, most of these can be eliminated by the special-distance-test. The running time for this test is O((2^e)g), where g denotes the time for computing a minimal spanning tree. Due to the exponential behavior, we perform this test
only for m < 4.

4) Terminal-Distance-Test

In this test, we consider a connected subgraph H = (W, F) of G with some vertices commmom to T and W, but not all. Let e and f be a shortest and a second shortest edge of the cut induced by W. Edge e = [u,v] with u Є W and v Є V¥W is part of some optimal solution of ST(G, T, c) and can thus be contracted, if c(f) >= du + c(e) + dv with du = min { d (t, u) | t is common to both T and W} and dv = min { d (t, v) | t Є T ¥ W }.

Initially, I would like to mention a rough pseudocode of how the above mentioned tests would be implemented:
Degree-Test I --> Special-Distance-Test --> Degree-Test I --> Terminal-Distance-Test -->Special-Distance-Test --> Degree-Test I --> Special-Distance-Test --> Degree-Test I -->Return
Note that the bottleneck degree m test is not included in my default strategy. For some difficult instances, however, it pays to use the bottleneck degree m test and iterate all four tests as along as there is some reduction possible.

Alternative-based reductions:
5) PTm (paths with many terminals) and related tests

Every edge (vi,vj) with c(vi,vj) > b(vi,vj) can be removed from G.
The PTm test is one of the most effective classical exclusion tests, but it is too time consuming for large instances in its original form. I will implement a modified

version of the PTm test.

For two terminals zi and zj , one readily observes that the bottleneck Steiner distance b(zi, zj) can be computed by determining a bottleneck on the fundamental path between zi and zj in the spanning tree TD(T), which can be constructed in time O(e + n log n). Each such bottleneck can be trivially computed in time O(r), leading to a total time O(mr) for m inquiries (m ∈ O(min{e, (r^2)})). Observing that one actually has a static-tree variant of the bottleneck problem, I'll use a strategy based on depth-first search to achieve time O(r^2) for all inquiries.

I do not need to precompute the b-values, because very often not all the combinations have to be checked; for example if the test condition turns out to be already satisfied during the computation (or, of course, if vi or vj were a terminal).

It is observed that if B' be the length of the longest edge in TD(T). Every edge (vi,vj) with c(vi,vj) > B' can be removed from the network.

Notice that using this test, one can eliminate some edges which could not be eliminated by the PTm test (even in its original form).
It must be mentioned that this observation has a greater impact than one would assume, because in some cases the reduction process is blocked in face of many alternatives with equal weights and can be reactivated only with a measure like this.

6) NTDk (non-terminals of degree k) test:

A non-terminal vi has degree at most 2 in at least one Steiner minimal tree if for each set A, |A | ≥ 3, of vertices adjacent to vi holds: The sum of the lengths of the edges between vi and vertices in is not less than the weight of a minimum spanning tree for the network (A, A×A,b).

If this condition is satisfied, one can remove vi and incident edges, introducing for each two vertices vj and vk adjacent to vi an edge (vj, vk) with length cij + cik (and keeping only the shortest edge between each two vertices).

The special cases with k (degree of vi) in {1,2} can be implemented with total time O(n) (for examination of all non-terminals). For k ∈ {3,...,7} we use the rb-values instead of the exact bottleneck Steiner distances. Because addition of new edges can be a non-trivial matter and the needed rb-values are already available, it is a good idea to check for each new edge if it could be eliminated using the PTm test, in this case it need not be inserted in the first place.

7) NV (Nearest Vertex) and related tests:

Let zi be a terminal with degree at least 2, and let (zi,vi') and (zi,vi'') be the shortest and second shortest edges incident with zi. The edge (zi,vi') belongs to at least one Steiner minimal tree, if there is a terminal zj, zj not equal to zi, with c(zi,vi'')>=c(zi,vi') + d(vi',zj).

The original version of the test NV requires the computation of distances, which is too time-consuming for large instances. But one can accelerate this test without making it less powerful, using an observation given below. For this purpose, we use Voronoi regions, saving some extra information while computing the regions. Let distance(zi) be the length of a shortest path from zi to another terminal zj over the edge (zi,vi'), computed as follows: Each time an edge (vi,vj) with vi ∈ N (zi), vj ∈ N (zj), zj not equal to zi is visited, it is checked whether vi is a successor of vi' in the shortest paths tree with root zi (simply done through marking the successors of vi'). In such a case distance(zi) is updated to min{distance(zi), d(zi,vi) + c(vi,vj) + d(vj,zj)}.
It is observed that the condition of the test NV is satisfied if and only if:
c(zi,vi'')>=c(zi,vi') + d(vi',base(vi')), if vi' is not an element of N(zi), and
c(zi,vi'')>=distance(zi), if vi' ∈ N (zi).

Using this observation, I'll perform the test NV for all terminals in time O(e + n log n). Note that in inclusion tests, each included edge is contracted into a terminal.
The Voronoi regions can also be used to perform a related inclusion test, which is called SL (Short Links)
It is also observed that, if say, zi be a terminal, and (v1,v1') and (v2,v2') the shortest and second shortest edges which leave the Voronoi region of zi (v1, v2 ∈ N(zi), v1', v2' are not elements of N(zi); these edges are called links). The edge (v1,v1') belongs to at least one Steiner minimal tree, if c(v2,v2')>=d(zi,v1) + c(v1,v1') + d(v1',zj), where zj = base(v1').

This test can also be performed for all terminals in total time O(e + n log n). The classical test SE (Short Edges) is a more powerful inclusion test. It is observed that even this test can be implemented with time O(e + n log n). But although this test is more effective than NV and SL in a single application, the difference almost vanishes when the reduction tests are iterated. Therefore, I only propose to use the much simpler, empirically faster tests NV and SL in our actual implementations.

8) Path substitution (PS)

It is another alternative-based reduction test that is more general than the previous tests in two ways: The test PS examines several edges along a path, instead of examining elementary graph objects (like single edges and vertices). If the test is successful, some of these edges can be deleted at once. The other more general aspect is a consequence of the first: Searching for alternatives for a path, it is not suffcient anymore to find one alternative, because the edges of the path can be involved in many different ways in a Steiner tree. As a consequence, such a test can only be effcient if it has strong requirements as conditions.

The basic idea is to start with a single edge as the path and then try to find *alternative paths for the vertices adjacent to those on the path. If this is not possible for exactly one adjacent vertex, the path is extended by the edge to this vertex and the search for alternative paths is restarted. Such successive extensions could finally lead to the desired situation. I describe here the observation that leads to the formal specification of the test in a simpliÿed way: I give only the description for deleting one edge of the path and define it only for the special case that the starting vertex v0 has degree 3. The extensions to deleting many edges on the path and to vertices with degree 2 or 4 are more or less straightforward.*

It is observed that if a path P (v0,...,vl) with degree(v0) = 3 and vi ∈ V ¥ T for all i ∈ {0,...,l}. We denote by vi1, vi2,... the vertices adjacent to each vi on P which are not contained in P. Let d0(vi,vj) be the length of a shortest path between vi and vj that does not contain (v0,v1); and dP (vi,vj) (for vi and vj in P) the length of the subpath of P between vi and vj .
The edge (v0,v1) can be deleted if for all i ∈ {1,...,l} there are functions fi and gi such that:
(I) for all vik adjacent to vi and for k0 = fi (k): dP (v0 , vi)>=d0 (v0k0 ; vik),
(II) for all v0k0 adjacent to v0 and for k = gi (k0): dP (v0 , vi)>=d0 (v0k0, vik), c(v0, v0k0)>=c(vi , vik).

Bound-based reductions:
Since one cannot expect to solve all instances of an NP-hard problem like the Steiner problem only through reduction tests with a (low-order) polynomial worst-case time (like the tests in the mentioned above), the computation of (sharp) lower bounds is a common part of the standard algorithms for the exact solution of such a problem. The information gained during such computations can be used to reduce the instance further. Besides, by using fast heuristics for generating bounds small worst-case running times can be guaranteed even for this kind of tests

9) Reduction using Voronoi regions:
The Voronoi regions can be used to determine a lower bound for the value of an optimal solution under some additional assumptions (for example, that the solution contains a certain non-terminal). For any terminal z, if I define radius(z) as the length of a shortest path from z leaving its Voronoi region N (z). These values can be easily determined while computing the Voronoi regions. For convenience, we assume here that the terminals are numbered according to non-decreasing radius-values. For each non-terminal vi , let zi1, zi2 nd zi3 be the three next terminals to vi, then I use the following observation to eliminate a non-terminal.
It is observed that if T is a Steiner minimal tree and assuming that vi is a Steiner node in T. Then d(vi , zi1) + d(vi zi2) + Σ radius(zt) (for t=1 to (r-2)) is a lower bound for the weight of T .
A non-terminal vi can be eliminated if this lower bound exceeds a known upper

bound. This method can be extended for eliminating edges, by the following observation:

It is also observed that if T is a Steiner minimal tree and assuming that T contains an edge(vi,vj). Then c(vi , vj) + d(vi , zi1) + d(vj zj1) + Σ radius(zt) (for t=1 to (r-2)) is a lower bound for the weight of T.

There is also a test defined performing the same actions as NTDk when it is successful, using the following observation:

It is observed that if T is a Steiner minimal tree and assuming that vi is a Steiner node whose degree in T is at least three. Then d(vi , zi1) + d(vi , zi2) + d(vi , zi3) + Σ radius(zt) (for t=1 to (r-3)) is a lower bound for the weight of T

Intuitively, one expects that a better lower bound should be achievable through this line of argument, because the paths between the terminals in a Steiner tree not only leave the corresponding Voronoi regions, but also span all terminals. Indeed, I would like to use the following idea:
It is observed that, suppose if I consider the auxiliary network G' = (T, E',d), in which two terminals are adjacent if and only if they are neighbors in the original network; defining:
d'(zi , zj):=min{min{d(zi , vi), d(zj , vj)} + c(vi , vj) | vi ∈ N (zi), vj ∈ N (zj)}. The weight of a minimum spanning tree for G is a lower bound for the weight of any Steiner tree for the original instance (G, T).

This observation can be extended to a test condition, for example, for any non-terminal vi , the weight of such a spanning tree minus the length of its longest edge plus d(vi , zi1) + d(vi , zi2) is a lower bound for the weight of any Steiner minimal tree that contains vi . The resulting test is very fast: The network G can be determined without much extra work while computing the Voronoi regions, and a minimum spanning tree for it can be computed in time O(|E| + |T|log|T|). For computing upper bounds in this context, we use a modified path heuristic with time O(e + n log n). So, all these tests can be performed in time O(e + n log n), this combined test is called as VR (Voronoi Regions).

With a heuristic solution available, all these tests can be easily extended to the case of equality of lower and upper bound. As the intuition suggests, the VR test is most effective for sparse networks with relatively few terminals; in this sense, it is a nice complement to the alternative-based tests, which are often specially successful if the proportion of terminals to all vertices is high.

Integration and implementation of tests:

I would like to implement through a direct control of loops (through parentheses), their termination criteria, switching of parameters, etc. It is pbserved that the (alternative-based) tests are not very sensitive to the order in which they are executed. On the other hand, the ordering has often an impact on the total time for reductions, (usually the fast version of the PTm test is performed first).
Again, for the implementation of the graph, I have chosen a kind of adjacency-list representation of networks (with all edges in a single array), but depending on the context, maybe, I'm thinking to sometimes switch to other auxiliary representations (all linear in the number of edges) for certain operations. For each test, I'm planning to perform all actions in a single pass (and do not, for example, delete an edge and start the test from scratch).

I would like to stress here that all actions following each test can be realized in a time dominated by the worst-case time O(e + n log n) of the fast tests. With the additional postulation that in each loop of the selected tests a constant proportion (say 5%) of vertices and edges must be eliminated and that instances of trivially small size are solved directly (by enumeration), one gets the same asymptotical time bound for the whole reduction process as for the first iteration (O(e + n log n), if one confines oneself to the fast tests).

Another technical aspect is the efficient reconstruction of a solution for the original instance out of a solution for the reduced instance (which often consists of a single terminal). Saving appropriate information during the reduction process, this can be done in time O(e). I'll always perform such a transformation after each run of the program, checking the feasibility and value of the solution in the original instance.

<u>Issues related to implementation:</u>
1) *Precomputing Steiner distances:*
A crucial issue for the implementation of the test is the calculation of bottleneck Steiner distances. An exact calculation of all sij would make the test impractical even for medium-size instances. So I'm using a good approximation of these distances and some appropriate data structures for retrieving them.
2) *Tree bottle necks:*
I'm going to use the tree bottleneck test, because every distance between tree nodes calculated for a minimum spanning tree or a Steiner minimal tree computation can be tested against the tree bottleneck; and in many of the cases where a tree can be ruled out, already an intermediate bottleneck test can rule out this tree, leading to a shortcut in the computation.
3) *For the bound-based tests:*
An efficient method for generating the dual feasible solution needed for the bound-based test is to use something called the DUAL-ASCENT algorithm. I plan to

improve the test by calculating a lower bound and reduced costs for different roots.

References:
[1] T. Polzin, S. V. Daneshmand: Improved algorithms for the Steiner problem in networks, Discrete Applied Mathematics 112, pp. 263-300, 2001
[2] Chapter 3 of the Dissertation of S. V. Daneshmand: Algorithmic Approaches to the Steiner Problem in Networks
[3] T. Koch, A. Martin: Solving Steiner tree problems in graphs to optimality, Networks 32(3), pp. 207-232, 1998
[4] Tobias Polzin, Siavash Vahdati: Extending Reduction Techniques for the Steiner Tree Problem: A combination of Alternative and Bound-Based Approaches MPI-I-2001-1-007, Dec. 2001
[5] Eduardo Uchoa, Marcus Poggi de Aragao, Celso Ribeiro: Preprocessing Steiner Problems from VLSI Layout, PUC-Rio.Inf.MCC 32/99, Dec. 1999.

What have you done so far?

As part of the GSoC application process, I've done the programming exercise specified for this project topic using OGDF, in which I've implemented three of the special cases of "Bottleneck Degree m Tests", of which the first two are also called "Degree-Test 1".

How do you plan to realize your project?

I would like to describe my plan to implement various preprocessing heuristics that are mentioned above, during the GSoC.

Bottleneck Degree m tests are partially done already as part of the application process.

Week 1 (19th May to 25th May):
I would like to continue with the Bottleneck Degree m test and complete it, with options to implement each of the special case, individually.

Week 2 (26th May to 1st June):
I'll implement the special-distance test and terminals-distance test, again with options to implement each of the special case, individually.

Week 3 (2nd June to 8th June):
Combine these tests into one packet, and implement them, with and without bottleneck degree m test.

Week 4 (9th June to 15th June):
Make snippets for various quantities, mentioned in the definitions part of the above

project description. These are going to be used in the following algorithms.

Week 5 (16th June to 22nd June):
Start with the alternative-based reductions – the PTm and related tests, that are based on the bove mentioned observations.

Mid-term Evaluations:
Week 6 (23th June to 29th June):
Implement NTDk test, and various observations based on it.

Week 7 (30th June to 6th July):
Implement NV test, and various observations based on it.

Week 8 (7th July to 13th July):
Implement SL test and possibly the SE test too.

Week 9 (14th July to 20th July):
Path substitution is implemented and also the observations based on it are also implemented.

Week 10 (21st July to 27th July):
Now, I'll go to the bound-based reductions, starting with the reduction using Voronoi regions (VR). Here, I'll use the snippets that would be made during week 4 for finding Voronoi region.

Week 11 (28th July to 3rd Aug):
Based on observations from the VR, various other tests mentioned above are implemented.

Week 12 (4th Aug to 10th Aug):
I'll find the optimum sequence of the above tests to be performed, so that the instances are significantly reduced, i.e graphs are reduced significantly.

Pencils down:
Week 13 (11th Aug to 17th Aug):

I'll compile all the modules together and provide it as a single/multiple library file (depending on the necessity), with options given to choose between the above reductions techniques. I'll make sure it compiles with the OGDF source code.

Final Deadline:
Week 14 (18th Aug to 22nd Aug):

I'll debug and make sure there is no error in the code, and also make sure that the preprocessing is significantly reducing the instances.

If time permits, I'm planning to implement the algorithms for finding upper bound – namely, the path heurestics and heurestic reductions, which uses the the above mentioned VR (reductions based on Voronoi regions). Otherwise, I would like to contribute to OGDF after GSoC. Also, after GSoC, I would like to work on the reductions based on DUAL-ASCENT algorithms.

Anticipated challenges:

1) I'm not sure if all the observations for each of the reduction tests, mentioned in the reference papers could be implemented during GSoC. I plan to give more importance to being stuck to the schedule and implement the various tests and implement as many observations as possible in a week.
2) I want make sure, that I combine all the reduction tests and compile it with the OGDF source code, in an elegant manner.

Programming Experience

Programming languages:

C	–	*Good*
C++	-	*Good*
Python	–	*Good*
Java	–	*Medium*
Matlab	-	*Medium*

Operating systems:

Linux, Windows.

Libraries:

1) *I've used many python-based programming libraries like – SymPy, SciPy, NumPy, etc.*
2) *I worked on an orbit propagator code (a set of libraries) (SGP4) in Matlab, that was used in a student satellite project.*
3) *I also worked on microcontrollers for various applications using C, like getting serial port data from a sensor into a microcontroller and then wirelessly from microcontroller to PC, etc. So, I had to use a lot of inbuilt libraries that are specific to the microcontroller model to implement various functions.*
4) *I developed a previously available code for mathematically simulating ray intersection on pipes of different shapes using visual C++, along with a GUI interface to it (it was for a project related to ultrasonics).*

Open source software experience:

I have used a wide range of open source softwares – from LibreOffice to Mozilla Firefox, etc. One significant usage was as follows:
As part of my, 3 months summer internship last year (at Ittiam Systems Pvt. Ltd.), I had to demonstrate the improvement in ASR's (Automatic Speech Recognizer's) recognition accuracy of real-time speech signals over the beam-formed (a signal processing technique) signals using CMU's Sphinx - open source Java-based ASR. But, later realized that the Java-based version was not meant to be accurate but to be more reliable. So, I had to shift to their C++ version of CMU's Sphinx, which was more accurate and relevant to my project.
So, I wish if I could contribute my part to the open source community, starting from GSoC 2014.
I actually wanted to contribute to CMU's sphinx itself, but since 2012, they haven't participated in GSoC.

Academic Experience

Academic Performance:
My current CGPA (Cumulative Grade Point Average) after 7 semesters is 7.7/10
I've done a basic course on C programming language (theory and lab). I have also done a course on computer simulations where we had to use C and Python for simulating various real-life problems in electrical engineering.

Do you have any experience with graph algorithm or graph drawing (theory or applications)?
I've done a course on graph theory, where I studied about trees, connectivity of graphs, euler tours and hamiltonian cycles, matchings, edge colourings, directed graphs and various algorithms related to each of those topics with their proofs.
I'm currently doing a project on quantum walks (or quantum random walks) - a computational tool in quantum computing related algorithms. It is based on lot of graph algorithms.

Please describe your knowledge in algorithms and data structures:
I'm currently doing a course on data structures and algorithms, where I'm studying about stacks, queues, binary search trees, trees, graphs, etc. for various applications. I am also studying about and working on algorithms like branch and bound, back tracking, breadth-first search, depth-first search, binary sort, topological sort, etc.

Why OGDF?

Have you already used OGDF before GSoC?
No, this is my first time.

Why are you interested in graph algorithms / graph drawing / your particular project?

One reason is that, I would like to implement algorithms used in steiner tree problems to quantum walks, i.e. trying to find the quantum version of those algorithms, so that it would be useful in quantum computing too.

The other reason is that, since I'm doing my master's in VLSI design, where steiner tree problems are crucial, I would like to learn more about those algorithms as part of GSoC 2014.

After GSoC, you envision your involvement with OGDF will be:

I would like to contribute various libraries for OGDF later too, especially implementing some of the latest improvised graph algorithms, say for shortest path, longest path, travelling salesman, steiner tree and for other such NP-hard problems. I really liked the responses that I got from the mentors of OGDF for my queries. So, I would like to work with them and contribute to the open sourse community through OGDF.

Additional Notes

Anything else you want to tell us?

My semester gets over by 10th of May. So, I'll be free for the rest of the summer, i.e. May, June and July. Though my next semester starts in August, since I'll be in my final year, I'll just have two courses and no exams till September. So, I'm quite comfortable with coding the entire summer. In fact, I'm looking forward to it, so that I'll be sure that I'm occupied.