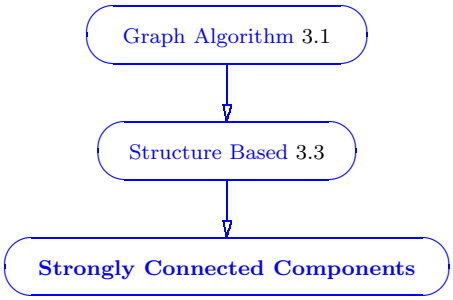
3.3.4 Strongly Connected Components

Section authors: Kaoutar ElMaghraoui, Benjarath Phoophakdee, Fikret Sivrikaya, Mishella Yoshi



Refinement of: Structure Based Graph Algorithm (§3.3), therefore of Graph Algorithm (§3.1).

Prototype:
```
template <class Graph, class ComponentMap,
          class P, class T, class R>
typename property_traits<ComponentMap>::value_type
strong_components(Graph& g, ComponentMap comp, const
bgl_named_params<P, T, R>& params = all defaults)
```

Input: The input to the strongly connected components algorithm is a directed graph.

Output: The strong_components() algorithm computes the strongly connected components of a directed graph using Tarjan's algorithm based on DFS. Essentially, the algorithm computes how many connected components are in the graph, and assigning each component an integer label. The algorithm then records which component each vertex in the graph belongs to by recording the component number in the component property map. The ComponentMap type must be a model of Writable Property Map. The value type should be an integer type, preferably the same as the vertices_size_type of the graph. The key type must be the graph's vertex descriptor type.

Effects: The structure of the graph is not altered by the algorithm.

Asymptotic complexity: Let V = number of vertices. Let E = number of edges.

- Average case (random data): $O(V + E)$
- Complete graph case: $O(V + E)$

Complexity in terms of algorithm running time: The following coefficients were obtained using least squares fitting using Matlab in the case of Adjacency-List implementation and Microsoft Excel in the case of Adjacency-Matrix implementation. The time complexity is in milliseconds.

Average Case :

- Adjacency-List implementation:
  $0.0023V + 0.0012E$
- Adjacency-Matrix implementation:
  $0.0005V^2$

Complete Graph Case :

- Adjacency-List implementation:
  $0.0011V^2 + 8E - 05V + 0.928$

Notice that the time bounds in the case of adjacency-matrix is quadratic. The reason is as follows. As the strongly connected components algorithm is based on DFS, it traverses all the out-edges of each vertex in a graph. This structure of the algorithm yields an $O(V + E)$ time bound for the adjacency-list representation, since the total cost of traversal over the out-edges is E in this case.

However, for the adjacency-matrix representation, where adjacency information is embedded in a $V * V$ matrix, the algorithm needs to go over a whole row in the matrix, of length V, for each vertex. Therefore the total cost of traversal over the out-edges is no more E, but V^2 in adjacency-matrix case.

Additionally, when the graph is complete the running time is also quadratic, in term of V, when using the adjacency-list representation because $E = O(V^2)$ so $O(V + E) = O(V + V^2) = O(V^2)$.

Experimental Data: To experimentally obtain the time complexity in terms of the number of edges and the number of vertices, several graphs have been examined. The graphs have been generated randomly. We have conducted several experiments to test the performance of the algorithm for both complete and sparse graphs. To isolate the impact of the number of vertices and the number of edges, the number of vertices was held constant while varying the number of edges. Similarly, the number of edges was held constant while varying the number of vertices.

The tests were performed using a gcc version 3.2 compiler with the boost library version 1.29.0. For each test, the running time is averaged over 20 runs if the number of vertices is below 500 and averaged over 3 runs otherwise. The results are presented in the tables below:

- Experiment 1: Fixing the edges
  In this experiment, the number of edges was held constant at 1600 while the number of vertices was varied. The number of vertices was multiplied by 2 in each test. The tests have been performed for both the adjacency-list and adjacency-matrix graph representations.

Results for E = 1600			
Adjacency-List		Adjacency-Matrix	
V	Time(ms)	V	Time(ms)
40	0	40	0
80	0	80	0
160	0	160	16
320	0	320	62
640	0	640	219
1280	0	1280	860
2560	15	2560	3453
5120	16	5120	13766
10240	31	10240	55031
20480	47	20480	220485
40960	94		
81920	188		
163840	391		
327680	750		
655360	1547		

- Experiment 2: Fixing the vertices
  In this experiment the number of vertices was held constant at 500 while the number of edges was varied. For each graph generated, the number of edges was selected randomly from the range $[1, V^2]$. The random generation of the number of edges is controlled by dividing the range into n intervals where n is the number of tests and selecting E randomly from all these intervals to ensure that we have a good sampling. The results for both adjacency-list and adjacency-matrix are shown in the tables below:

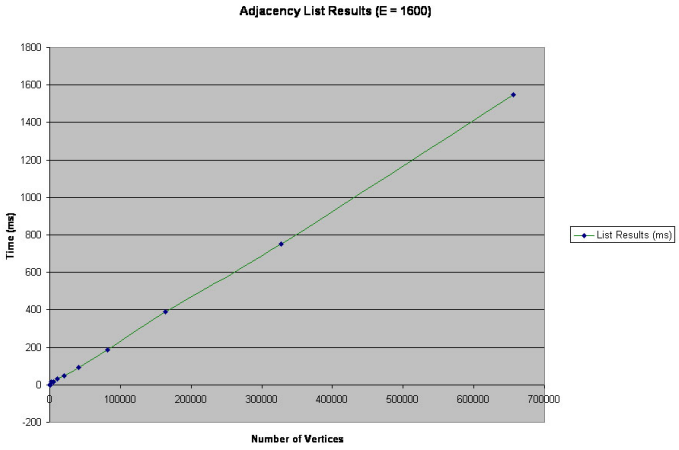
Results for V = 500			
Adjacency-List		Adjacency-Matrix	
E	Time(ms)	E	Time(ms)
43721	62	43721	125
283315	328	283315	125
461222	515	461222	125
654847	734	654847	125
982204	1109	982204	125
1370998	1547	1370998	125
1959484	2219	1959484	125
1233838	1422	1233838	125
2185434	2453	2185434	125
2927853	3297	2927853	125
2974943	3360	2974943	125
3557109	4016	3557109	125
3742877	4219	3742877	125
2561375	2891	2561375	125
3831652	4312	3831652	125

- Experiment 3: Sparse vs. Complete graph performance
  The purpose of this experiment was to test how the algorithm behaves when the graph is sparse and when it is complete or dense. Sparse graphs were constructed by having the number of edges equal to half the number of vertices. Complete graphs were constructed by adding an edge between every pair of vertices. The following tables show results obtained.

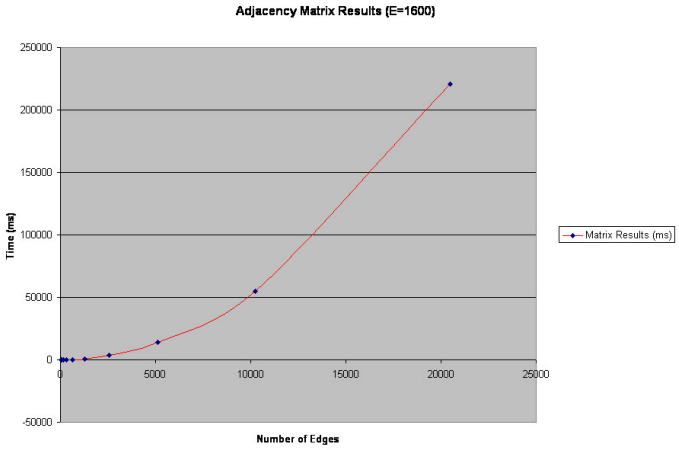
Sparse Graph $E = V/2$		Complete Graph	
V	Time(ms)	V	Time(ms)
50	0	50	0
100	0	100	15
200	0	200	47
400	0	400	187
800	0	800	734
1600	0	1600	2938
3200	0	3200	11750
6400	16		
12800	31		
25600	78		
51200	172		
102400	359		
204800	703		
409600	1437		
819200	2891		

Experimental Plots and Discussion :

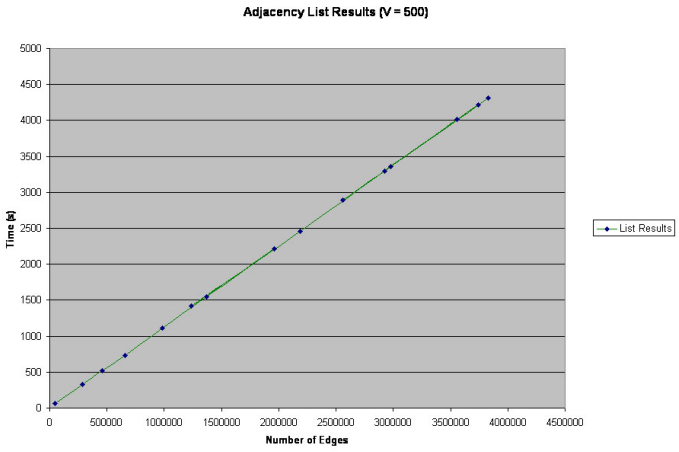
- Plot 1: Experimenting with vertices for the list case
  This plot shows the results of the experiment 1 with the adjacency list representation. The number of vertices is varied while E is held constant at 1600 edges. The running time is linear in terms of the number of vertices.



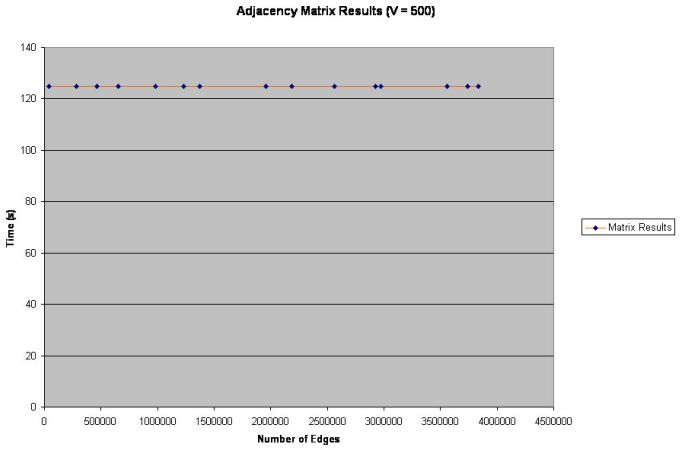
- Plot 2: Experimenting with vertices for the matrix case
  The following plot shows the result of experiment 1 in the case of an adjacency-matrix representation. The running time is shown to be quadratic in the number of vertices. This confirms the theory. The coefficient obtained from the curve fitting is 0.0005.



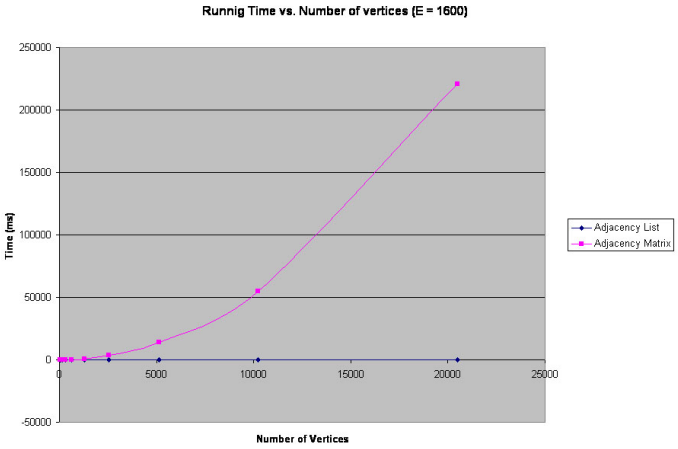
- Plot 3: Experimenting with Edges for the list case
  This plot shows the results of experiment 2 with adjacency list graph representation. The number of edges is varied while the number of vertices is held constant at 500. Again as expected, the running time is linear in terms of E. The coefficient that was obtained from Matlab curve fitting is 0.0012.



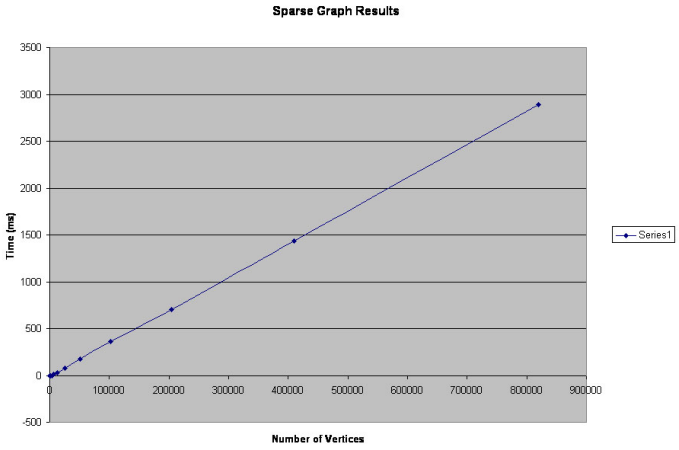
- Plot 4: Experimenting with Edges for the matrix case
  This plot shows the results of experiment 2 with the adjacency-matrix representation. The number of edges should not affect the running time in this case. The results bear this out since the running time remains constant even if we vary the number of edges.

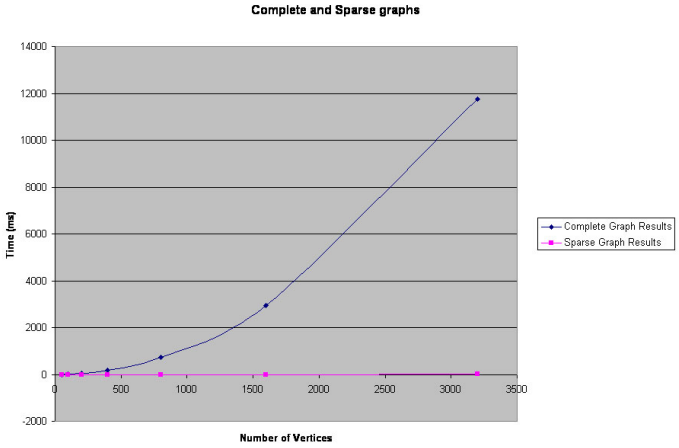


- Plot 5: Adjacency-List vs. Adjacency-Matrix
  This plot shows that the algorithm runs much faster with an adjacency-list representation as the number of vertices increases, while the number of edges, E, is held constant. For this experiment, E is fixed at 1600. Note that the graph gets sparser as we increase the number of vertices with a fixed E.



- Plot 6-7:
  The sparse/complete graph experiments have been done with the adjacency matrix representation. In the sparse graph case the number of edges generated is equal to half the number of vertices in the graph. When the graph is sparse, the running time is linear in terms of V and E. When the graph is dense the running time is quadratic in terms of the number of vertices. The coefficient obtained from curve fitting is 0.0011 which is greater than the coefficient obtained from plot 2 (0.0005). This indicates that it is better to use the adjacency matrix representation when the graph is dense. The results confirm that the algorithm runs much faster when the graph is sparse.


Sparse Graph Results
3500
3000
2500
2000
1500
1000
500
0
-500
Time (ms)
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
Number of Vertices
Series1


Complete and Sparse graphs
14000
12000
10000
8000
6000
4000
2000
0
-2000
Time (ms)
0
500
1000
1500
2000
2500
3000
3500
Number of Vertices
Complete Graph Results
Sparse Graph Results

Pseudocode: Pseudocode of strongly connected components algorithm. The Boost Graph Library uses Tarjan's algorithm based on DFS in implementing strong_components functions. The psuedocode can be found here.

References :

- An animation of an operating strongly connected components algorithm is given here.

  This animation shows the DFS-based Strong Components algorithm of a directed graph G. You can view from the animation a trace of the two phases of the algorithm: phase one where the finish stack is set up to contain the vertices in reverse order of a DFS traversal and phase two where the a DFS traversal is done on the transpose of G based on the contents of finishStack. You can also view how finishStrack is being filled up and emptied as the algorithm progresses.

  As the animation progresses, the vertices are colored according to their state in the DFS traversal. Initially all the vertices are white. When a vertex is discovered it becomes grey and when it is finished it becomes dark.

  You can control the animation by starting it, stoping it, pausing it, or stepping through it at any time. You can also control the speed of the animation through the scroll bar provided at the bottom of the window.

- The BGL reference material for strongly connected components algorithm can be found here.