PROFILER REPORT

CSP301: PRODUCT 2

SOCIAL NETWORK SIMULATOR

Created By:

Devansh Dalal
2012CS10224

Poojan Nikunjkumar Mehta
2012CS10241

Shubhankar Suman Singh
2012CS10255

PART A

When the program Gprof was run for different sizes and different number of days, The flat profile table for main functions in the program is as shown in following table:

#days-555 effective nodes-500 #

% time	Self seconds(ms)	Total numbers of calls	Call name
33.34	0.01	145446	std::__detail::_Mod<unsigned long, 2147483647ul, 16807ul, 0ul, false>::__calc(unsigned long)
33.34	0.01	248	__gnu_cxx::new_allocator<Course*>::new_allocator()
33.34	0.01	1	Generator::generateFriends(void*)
0.00	0.00	1	Generator::generateCourses(void*)
0	0.00	1	Generator::generateFaculty(void*)
0	0.00	1	Generator::generateStudent(void*)

#days-4000 effective nodes-500 #

% time	Self seconds(ms)	Total numbers of calls	Call name
38.10	0.08	145446	Generator::generateFriends(void*)
14.29	0.02	248	int std::uniform_int_distribution<int>::operator()
4.76	0.21	1	std::vector<GraphNode*, std::allocator<GraphNode*> >::operator[](unsigned
0.00	0.21	1	Generator::generateCourses(void*)
0	0.21	1	Generator::generateFaculty(void*)
0	0.21	1	Generator::generateStudent(void*)

Analysis

- GenerateFriends operates most of time while generateCourses (executed twice every year), generateFaculty (executed once) ,generateStudent (executed once every year) takes time comparably less than generateFriends.
- Also on increasing the num_days,the generateFriends even takes longer % of total time(38 %) time interval keep on increasing.
- Besides our in built functions, std::uniform_int_distribution<int>::operator() and std:: vector size() functions are contributing.

PART B

% time	Self seconds(s)	Total numbers of calls	Call name
94.46	0.17	1	Algorithms::FloydWarshall()
5.56	0.01	1651	std::vector<GraphNode*, std::allocator<GraphNode*>
4.76	0.21	26211	std::_Iter_base<GraphNode**, false>::_S_base(GraphNode**)
0.00	0.00	0	Algorithms::dijkstra()
0.0	0.00	0	Algorithms::bfs()
0.0	0.00	0	Algorithms::dfs()

The table shows the data for the queries where dijkstra, bfs ,dfs etc are not called.

Analysis

- For most of time Floyd-Warshall is taking above 90% of total time while other algorithms are executed only when user asks for corresponding queries. FloydWarshall(O(n3)) is only used for during preprocessing. As the number of queries increases the % of time by FloydWarshall. When we increased the number of nodes to 1.5, FloydWarshall took almost 0.7 s.
- For larger graphs it was observed that FloydWarshall took almost 100% of the time due to its time complexity .
- As we start Gyani it calls the analyser which process the FloyedWarshall(heaviest algorithm as you can see from table) only once at its start. Gyani and analyzer runs parallel with it with ipc(inter process co) between them. This way we have minimized the time needed for preprocessing future request.

Analysis for each query

Since for each query (when run once) FloydWarshall takes most of the time, we ran each query multiple time in order to see which part of the code/ which algorithm takes most of the time (except the Floyd-Warshall).

Query number	Function called the most
1	Algorithms::clique()
2	Algorithms::floyedWarshall() (precomputed),Algorithms::bfs()
3	Algorithms::dijkstra()
4	Algorithms::dijkstra()
5	Algorithms::floyedWarshall()(precomputed)
6	Algorithms::floyedWarshall()(precomputed)