ITAT 2016 Proceedings, CEUR Workshop Proceedings Vol. 1649, pp. 212–217
http://ceur-ws.org/Vol-1649, Series ISSN 1613-0073, © 2016 A. Jursa

ITAT

Fast algorithm for finding maximum clique in scale-free networks

Andrej Jursa

Department of Applied Informatics, Faculty of Mathematics, Physics and Informatics, Comenius University in Bratislava
andrej.jursa@fmph.uniba.sk

Abstract: The maximum clique problem in graph theory concerns finding the largest subset of vertices in which all vertices are connected with an edge. Computation of such subset is a well-known NP-hard problem and there exist many algorithms to solve it. For our purposes we created an algorithm specially targeted for solving this problem in scale-free networks, where many significant search improvements may be introduced. We use the general purpose algorithm developed by Östergård in 2002 as subroutine for our new algorithm. We improved the search process by initial heuristic. Firstly we compute preliminary clique size and then reduce the graph by k-core decomposition of this size. Subsequently, we employ greedy algorithm for coloring of chosen vertex as well as its neighbors. Our algorithm is able to solve maximum clique problem for arbitrary graphs, but together with these and some other, less significant pruning techniques, the overall algorithm performs exceptionally well on scale-free networks, which was tested on many real graphs as well as randomly generated graphs.

1 Introduction

For our study of functional brain networks we needed an algorithm to solve maximum clique problem in these graphs to compare networks of three classes of participants. We realize that using a general purpose algorithm for this task may lead to very long computation times. According to the scale-free structure of functional brain networks we can implement several pruning techniques to make the search process faster. Functional brain networks we have data for are simple undirected graphs. Off course there exists randomized or heuristic algorithms like [3] or [5] which can give solution quickly, but for our study we needed algorithm which can give us exact solution in shortest possible time. Our algorithm we have created is based on exact algorithm created by Östergård and described in [4].

Undirected graph is combinatorial structure which we can denote as $G = (V,E)$, where V is set of vertices and E is set of edges or pairs of vertices. We can say that two vertices are adjacent, if there is edge between them. The number of edges which are connecting vertex v with other vertices is denoted as degree k of vertex v (we are denoting this as $deg(v) = k$). For vertex v we can get his open set of neighbors $N(v) = \{u | u \in V \wedge (u,v) \in E\}$ or close set of neighbors $N[v] = N(v) \cup \{v\}$.

Simple graph does not have loops from - to the same vertices, does not have multiple edges between the same pair of vertices and does not have weighted edges. Our algorithm is targeted to solve maximum clique problem for simple undirected graphs. For simple undirected graph we can compute density of that graph as $D = \frac{2|E|}{|V|(|V|-1)}$.

A clique in simple undirected graph is an subset of vertices $V' \subseteq V$ so that $\forall v \in V', \forall u \in V' : u \neq v \implies (v,u) \in E$. The maximum clique is largest clique in undirected simple graph. The size of maximum clique is also called clique number of graph G and can be denoted as $\omega(G)$.

Scale-free networks are simple undirected graphs whose degree distribution asymptotically follows power law. There is no specific scale in degree distribution. Fraction $P(k)$ of vertices having degree equal to k scales for large values of k as $P(k) \sim k^{-\gamma}$, where γ is typically in range $2 < \gamma < 3$. For given graph G we can compute degree distribution. It can be done by counting how many vertices in G have particular degree. We can visualize this distribution for scale-free networks in a log-log plot (it can be seen on figure 1 in part (a)).

The k-core decomposition of the graph G forms an induced subgraph H, where all vertices have the degree at least k. When we compute k-core decomposition from the scale-free network, we can remove large amount of vertices with the degree lower than k. In the diagram in (b) part of figure 1 the red area represents all vertices removed from scale-free network due to k-core decomposition. By computing new k-core decomposition each time we know new size of clique, we can reduce search space for next iteration of searching.

2 Östergård's algorithm

The original algorithm is processing graphs vertices from the highest index to the lowest one. During this process it also maintains array called $c[i]$, where i is the index of already processed vertex. It stores value of *max* in the array $c[i]$ after vertex i and his neighborhood is processed. This algorithm is using subroutine CLIQUE for branching over neighbors of the initial vertex. To get this neighbors it is using sets defined as $S_i = \{v_i, v_{i+1}, v_{i+2}, \ldots, v_{|V|}\}$, where i is an index of starting vertex.

In algorithm 1 one can see pseudo-code of this original algorithm. At line 8 there is condition for number of vertices remaining in set U with respect of current clique *size*. If this condition holds, algorithm is unable to find clique with size higher than *max*, so it can return from subroutine. Also similar test is introduced at line 11. In this

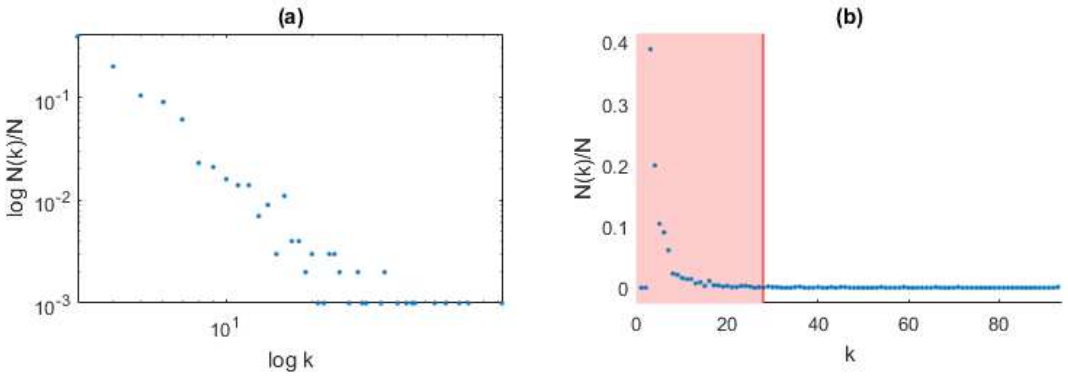

Figure 1: Scale-free network degree distribution in log-log plot (a) and vertices potentially removed by k-core decomposition, where $k = 0.3 \cdot deg_{max}(G)$ (b).

case it is testing if previous search from vertex with index i have founded clique size hight enough that with respect to current clique *size* algorithm can find clique higher than *max*. If this condition does not hold, it will return from subroutine as well.

This algorithm can be used for arbitrary simple undirected graph and it will find maximum clique size. The values of array c behaves following: $c[i] = c[i+1]$ if algorithm does not find new, higher, clique size, otherwise $c[i] = c[i+1]+1$.

3 Our algorithm

Our solution is using original Östergård's algorithm[1] [4] as a subroutine. Our algorithm is following the same principles as the original one by processing vertices from the highest index to the lowest. In the whole process of searching for maximum clique size we have global variable named *max* which contains currently known maximum size of clique found in the process.

Our algorithm is split in two phases. In the first phase, which is called initialization phase, the algorithm is using heuristic function to find preliminary clique size. This process may return clique size close to maximum clique size[2]. The preliminary size returned by heuristic is set into variable *max*. When the initialization phase know this value the algorithm reduces the search space of graph by computing *max*-core decomposition[3]. At the end of this phase it computes new isomorphic graph, where all vertices are indexed again from 1 to $|V_k|$ but also the vertex degree is raising with indexes[4]. The $V_k \subseteq V$ is the set of vertices of the graphs k-core decomposition.

First step of the initialization phase is to compute preliminary clique size by heuristic function. Similar method was used in [2] and [5] but our heuristic function is slightly different. First it needs to determine the threshold value of the degree (line 11 of algorithm 2) and then for each vertex in the graph which have the degree equal or higher than the threshold it runs iterative process to find clique, as it is described on algorithm 2. After several experiments on the scale-free networks we have decides that the good value for this threshold is 0.95 times maximum degree in the graph G. This will enable the heuristic function to run

Algorithm 1 Östergård's original algorithm.

Input: Simple undirected graph G.
Output: Clique number $\omega(G)$ in global variable *max*.

```
1:  procedure CLIQUE(U, size)
2:      if |U| = 0 then
3:          if size > max then
4:              max ← size
5:              found ← true
6:          return
7:      while U ≠ ∅ do
8:          if size + |U| ≤ max then
9:              return        ▷ Nothing better can be found.
10:         i ← min{j | v_j ∈ U}
11:         if size + c[i] ≤ max then
12:             return        ▷ Nothing better can be found.
13:         U ← U \ {v_i}
14:         CLIQUE(U ∩ N(v_i), size + 1)
15:         if found = true then
16:             return        ▷ Higher clique size was found,
    return to main loop.
17: procedure MAIN
18:     max ← 0
19:     for i ← |V| downto 1 do
20:         found ← false
21:         CLIQUE(S_i ∩ N(v_i), 1)
22:         c[i] ← max        ▷ Store clique size for next
    iteration, this is used for pruning.
```

[1] Östergård's subroutine CLIQUE is used in our algorithm.

[2] But there is also chance for returning very small size compared to maximum clique size in graph.

[3] It is k-core decomposition of graph, where $k = max$.

[4] Vertex with index 1 have minimum degree in a graph and vertex with index $|V_k|$ have maximum degree in a graph.

tests for large number of vertices with almost maximum degree in graph but not for all vertices in graph, so the heuristic have good chance to find the preliminary clique size very close to the real maximum clique size.

Heuristic iteration of finding clique is a greedy function which adds vertices with maximum degree to the forming clique and when there is no more vertex to be added the clique is found. Repeating this for more starting vertices this process can find clique size very close to clique number of input graph G. On the other hand the ratio between clique size of graph G and size found by the heuristic can be also close to zero in some cases.

Algorithm 2 Preliminary clique size heuristic function.

Input: Simple undirected graph G.
Output: Preliminary clique size in graph G.

1: **function** CLIQUEHEURISTICSTARTINVERTEX($G, i, lastMax$)
2: $Clique \leftarrow \{v_i\}, Nb \leftarrow N(v_i)$ ▷ Initialize clique and neighbor sets.
3: **while true do**
4: **if** $|Clique| + |Nb| \leq lastMax$ **then** ▷ Higher clique size can not be found.
5: **return** $lastMax$
6: **if** $|Nb| = 0$ **then** ▷ All possible vertices are processed.
7: **return** $|Clique|$
8: $j \leftarrow$ index of vertex $w_j \in Nb$ which have maximum degree
9: $Clique \leftarrow Clique \cup \{v_j\}, Nb \leftarrow Nb \cap N(v_j)$ ▷ Update clique and neighbor sets.
10: **function** CLIQUEHEURISTIC(G)
11: $deg_{threshold} \leftarrow \lfloor 0.95 \cdot \max\{deg(u) | u \in G_S\} \rfloor$
12: $heurMax \leftarrow 0$
13: **for** $i \leftarrow 1$ **to** $|V|$ **do**
14: **if** $deg(v_i) \geq deg_{threshold}$ **then**
15: $max \leftarrow$ CLIQUEHEURISTICSTARTINVERTEX($G, i, heurMax$)
16: **return** $heurMax$

After heuristic search is completed the algorithm knows preliminary size of the clique and it saves it to the variable max. For next phase it is necessary to compute max-core decomposition which removes possibly large amount of vertices and effectively reduces search space[5]. Because algorithm is processing vertices by their indices from higher to lower we run renumbering procedure on new max-core decomposition so that vertices will be sorted by their degree. This sorting can improve search process because it allows to skip several vertices with higher degrees whose neighbor sets are smaller or equal to the current max.

The process of computing k-core decomposition is very simple iterative deletion of vertices which degree is lower than k and it is described on the algorithm 3. In this algorithm $V(G)$ denotes the set of vertices V belonging to the graph G.

Algorithm 3 k-core decomposition of graph G.

Input: Simple undirected graph G, desired value k.
Output: k-core decomposition of graph G.

1: **function** COMPUTEKCORE(G, k)
2: $G_{k\text{-}core} \leftarrow G$
3: **repeat**
4: $G_{tmp} \leftarrow G_{k\text{-}core}$
5: **for all** $v \in G_{k\text{-}core}$ **do**
6: **if** $deg(v) < k$ **then**
7: $V(G_{k\text{-}core}) \leftarrow V(G_{k\text{-}core}) \setminus \{v\}$
8: **until** $|V(G_{tmp})| \neq |V(G_{k\text{-}core})|$ ▷ If there is no change in graph then the k-core decomposition is finished.
9: **return** $G_{k\text{-}core}$

After completion of the initialization phase the main search phase follows. The algorithm works with two nested loops, first is iterating over last k-core decomposition (line 7 in algorithm 4) and starting the second loop. Here is the algorithm processing vertices from higher index to lower (inside last computed k-core decomposition, line 8 in algorithm 4). For each vertex v_i is testing number of colors needed to color vertices from $N[v_i]$ which forms induced subgraph. If this number of colors is higher than currently know max, there can be clique with higher size than max.

To obtain neighbors set of vertex v_i new metric is introduced. We call it internal degree and it is defined in equation 1.

$$deg_v(u) = |\{w | w \in N[v] \wedge (u,w) \in E\}| \quad (1)$$

In the search process our algorithm will processes each vertex only once. This is guaranteed by checking the array c (line 10) from the Östergård's algorithm. If this array is already set for the given index, then we know that the vertex with this index was processed before.

In the main loop, where the algorithm is iterating through all unprocessed vertices, it is running greedy function for coloring[6]. Result of this coloring function is suboptimal number of minimum colors needed to color vertex and it neighborhood, which can be performed quickly. It gives us a good information whether it is promising to find better clique size when the algorithm starts branching from this vertex. The process of coloring vertex v and it neighborhood $N_v = \{v\} \cup \{u | u \in N(v) \wedge deg(u) \geq max \wedge \arg u > \arg v\}$ is described in algorithm 5.

[5] In some cases heuristic can find maximum clique size and max-core decomposition can be empty.

[6] This function is similar to one mentioned in [2] but in our implementation this is performed in the process of selecting vertices for branching not before the whole process.

Algorithm 4 Algorithm for finding maximum clique in scale-free networks.

Input: Simple undirected graph G.
Output: Clique number $\omega(G)$ in global variable *max*.

```
1: function SORTGRAPHBYDEGREEASCENDING(G)
2:     return new G_I isomorphic graph, where vertex with maximum degree have maximum index
3: procedure MAIN
4:     max ← CLIQUEHEURISTIC(G)
5:     G_max-core ← SORTGRAPHBYDEGREEASCENDING(COMPUTEKCORE(G, max))
6:     lastMax ← max
7:     while true do
8:         for i ← maximum index in G_max-core downto minimum index in G_max-core do
9:             found ← false
10:            if c[i] is already set then ▷ Vertex v_i has been already processed.
11:                continue
12:            if GETMINCOLORS(G_max-core, i, max) > max then
13:                N_{v_i} ← {u | u ∈ S_i ∩ N(v_i) ∧ deg_{v_i}(u) ≥ max}
14:                CLIQUE(N_{v_i}, 1)
15:            c[i] ← max
16:            if found = true then
17:                return
18:        if lastMax = max then ▷ All vertices are processed and there is no better solution.
19:            break
20:        else
21:            lastMax ← max
22:            G_max-core ← COMPUTEKCORE(G, max)
```

Algorithm 5 Greedy algorithm used to find minimum colors needed to color vertex i and it's neighbors.

Input: Simple undirected graph G, index of vertex i.
Output: Suboptimal number of minimum colors needed to color v_i and it immediate neighborhood.

```
1: function GETMINCOLORS(G, i, minDegree)
2:     colors ← 1, colorMap[v_i] ← 1
3:     Neighbors ← {u | u ∈ N(v_i) ∧ deg(u) ≥ minDegree ∧ arg u > i}
4:     AllNodes ← Neighbors ∪ {v_i}
5:     for all u ∈ Neighbors do ▷ Iterate over neighbors u of vertex v_i.
6:         N_u ← N(u) ∩ AllNodes ▷ Assemble set of neighbors of u within neighbors of v_i.
7:         minColor ← 1, UsedColors ← {}
8:         for all w ∈ N_u do ▷ Iterate over all neighbors w of vertex u ...
9:             if colorMap[w] ≠ null then ▷ ... and collect all colors already used.
10:                UsedColors ← UsedColors ∪ {colorMap[w]}
11:        for color ← 1 to colors + 1 do
12:            if color ∉ UsedColors then ▷ New unused minimum color is found.
13:                minColor ← color
14:                break
15:        colorMap[u] ← minColor
16:        colors ← max(colors, minColor) ▷ Remember the highest color number used.
17:    return colors
```

4 Results and testing

Our algorithm is able to solve maximum clique problem, as well as Östergård's algorithm, for all sorts of simple undirected graphs. But for all graphs with scale-free property it is our algorithm able to do so much faster. Most useful improvement here is the heuristic function which finds preliminary clique size in initialization phase, with subsequent k-core decomposition throughout whole process of searching. By decomposition the algorithm is able to reduce the graph size by removing vertices which can not be part of a maximum clique. Also degree distribution after each decomposition step is changing from original (asymptotically following power law) to linear looking one. When the final clique is found[7] and last computation of *max*-core decomposition the remaining vertices does not forms scale-free network anymore and they must be processed all to confirm found clique number. It is also possible that the last *max*-core decomposition will return empty graph, this means immediate finish of the algorithm without additional need to confirm found clique number.

In the search phase an additional pruning techniques were introduced. First was computation of minimum colors needed to color starting vertex and his neighborhood. This is greedy function so it can not compute optimal solution, but approximative solution is good to decide if processing of neighbors of the starting vertex is promising to find higher clique size. Second pruning here is neighbors node selection by testing their internal degree (according to equation (1)). Because connections outside neighborhood of the starting vertex does not contribute any vertex to the clique that we look for inside this neighborhood, we may omit these vertices when we deciding if vertex from neighborhood have to be added to the set of neighbors used for branching. This can reduce search space for each starting vertex and speed up the algorithm.

We have tested our algorithm on our FMRi data of functional brain networks as well as on simple undirected graphs from the database which can be found on https://snap.stanford.edu/data/[8] and on some graphs from DIMACS dataset. We also generated some test-

[7] The one representing one of maximum cliques in graph.

[8] Stanford dataset and FMRi functional brain networks represents real world graphs.

ing graphs using Bernoulli graph distribution, Barabáshi-Albert graph distribution [1] and Watts–Strogatz graph distribution [6]. Our implementation is written in Java and all tests were preformed on computer with Inter core i7 4790K@4.0GHz processor with 16GB of RAM. Running process of our algorithm have heap size of 12GB.

Table 1 contains some results from test runs. The graph name is represented by G, D is the density of edges inside graph G. $|V|$ and $|E|$ represents the sizes of vertices and edges set of graph G. Preliminary size of clique is denoted here as s_p while maximum clique size is denoted as clique number of graph $\omega(G)$. There is also ratio between s_p and $\omega(G)$ which represents how close the initialization phase heuristic search was to the real maximum clique in the given graph. Also times are displayed here, the „Time of phase 1" is time needed to compute preliminary clique size[9] and „Time of phase 2" is time needed to complete iterative branching of search phase.

The graph „C125.9" is from DIMACS dataset. It is the smallest one from DIMACS and it does not represent the scale-free network. It is random graph with edge probability of 0.9, with 125 vertices and 6963 edges. It takes 1637 seconds for our algorithm to process this graph. This is clear demonstration that our algorithm is able to find a maximum clique in an arbitrary undirected graph, but the time to process it is not better than the time needed by Östergård's algorithm[10]. Also graphs called „BG_n_p" are random graph, generated by Bernoulli graph distribution, where n is number of vertices and $\frac{p}{100}$ is probability of edge between two vertices. The time needed to solve these graphs is exponentially rising with number of vertices (the probability of edge is the same for both of them).

The graph called „facebook", as well as „as-skitter", „Email-Enron", „CA-AstroPh", „CA-CondMat" and „CA-HepPh" represent scale-free networks from Stanford snap datasets. One may see that the „facebook" graph have higher density than other graphs from dataset and the time needed to compute the maximum clique is around 379 seconds for just 4039 vertices and 88234 edges. On the other hand graph „as-skitter" with has 1696415 vertices and 11095298 edges, and very low density is processed in only 99 seconds. Initialization phases heuristic time for this graph is 28.6 seconds due to the large amount of vertices. We may say that number of vertices and the edges density is the key factor of the algorithm speed. Graph „CA-HepPh" is an example of a graph, where heuristic function finds maximum clique size and resulting k-core decomposition forms an empty graph. We also tested some randomly generated graphs by Barabási-Albert graph distribution. These graphs are „BA_n_k", where n is number of vertices and each new vertex is connected with k edges. These graphs was also solved quite fast and initial heuristic have found preliminary clique sizes very close to graphs clique numbers.

Graphs „awith_*", „awout_*" and „young_*" represents FMRi functional brain networks for adult participants with Alzheimer disease, without this disease and young participants without disease. All are scale-free networks having from 5400 to 8200 vertices. Here we can also see that the key factor is a density of edges and the number of vertices. We can see that some graphs were processed under one second by our algorithm. We also tested the performance of the original Östergård's algorithm on these graphs, which needs tens of minutes or several hours to process them.

Last dataset of graphs are named „WS_n_p_k", these are random graphs generated using Watts–Strogatz graph distribution, where n is number of vertices, $\frac{p}{100}$ is rewiring probability starting from $2k$-regular graphs. One can see that our algorithm is also performing well on small-world networks even it was not designed for this type of graphs. Results of heuristic here is also very close to final graph clique numbers.

For our scale-free brain functional networks or other graphs with scale-free property our algorithm gives us exact results much faster than the Östergård's one. That means we are able to analyze scale-free graphs for our purposes very quickly.

[9] Graph loading and construction as well as post heuristic renumbering of graph and k-core decomposition is not included in this time.

[10] Actually by using additional techniques like k-core decomposition and others which are not included in Östergård's algorithm, the run time of our algorithm for graphs of these types is higher than time needed by original algorithm.

References

[1] Albert, R., and Barabási, A.-L. Statistical mechanics of complex networks. *Reviews of Modern Physics 74* (Jan. 2002), 47–97.

[2] Eblen, J. D., Phillips, C. A., Rogers, G. L., and Langston, M. A. The maximum clique enumeration problem: Algorithms, applications and implementations. In *Proceedings of the 7th International Conference on Bioinformatics Research and Applications* (Berlin, Heidelberg, 2011), ISBRA'11, Springer-Verlag, pp. 306–319.

[3] Nehéz, M. Analysis of the randomized algorithm for clique problems. In *Proceedings of the 15th Conference on Applied Mathematics APLIMAT 2016* (Bratislava, Slovak Republic, Feb. 2016), SUT Publishing, pp. 867–875.

[4] Östergård, P. R. J. A fast algorithm for the maximum clique problem. *Discrete Appl. Math. 120*, 1-3 (Aug. 2002), 197–207.

[5] Pattabiraman, B., Patwary, M. M. A., Gebremedhin, A. H., Liao, W., and Choudhary, A. N. Fast algorithms for the maximum clique problem on massive sparse graphs. *CoRR abs/1209.5818* (2012).

[6] Watts, D. J., and Strogatz, S. H. Collective dynamics of 'small-world' networks. *Nature 393*, 6684 (1998), 409–10.

G	$\|V\|$	$\|E\|$	D	Preliminary size (s_p)	Time of phase 1	$\omega(G)$	$s_p/\omega(G)$ ratio	Time of phase 2
C125.9	125	6963	0.8984516129	32	0.016	34	0.94118	1636.768
BG_500_40	500	49820	0.3993587174	9	0.031	11	0.81818	21.386
BG_1000_40	1000	199909	0.4002182182	11	0.169	12	0.91667	2031.105
facebook	4039	88234	0.0108199635	7	0.068	69	0.10145	378.944
as-skitter	1696415	11095298	0.0000077109	37	28.609	67	0.55224	98.769
Email-Enron	36692	183831	0.0002730976	17	0.177	20	0.85000	1.974
CA-AstroPh	18771	198050	0.0011242248	23	0.202	57	0.40351	0.934
CA-CondMat	23133	93439	0.0003492312	4	0.142	26	0.15385	0.297
CA-HepPh	12006	118489	0,0016441731	239	0.097	239	1.00000	0.068
BA_300_20	300	5790	0.1290969900	20	0.003	21	0.95238	0.008
BA_1000_20	1000	19790	0.0396196196	17	0.019	21	0.80952	0.071
BA_10000_20	10000	199790	0.0039961996	20	0.191	21	0.95238	0.430
BA_100000_20	100000	1999790	0.0003999620	20	3.249	21	0.95238	7.786
BA_250000_50	250000	12498725	0.0003999608	51	25.653	51	1.00000	25.013
awith_09	7558	102091	0.0035748773	23	0.078	29	0.79310	0.884
awith_37	6218	73880	0.0038223046	33	0.058	36	0.91667	0.324
awith_02	7529	66494	0.0023463649	28	0.077	41	0.68293	0.302
awith_40	5677	17200	0.0010675720	11	0.018	12	0.91667	0.012
awout_13	7416	139482	0.0050730283	62	0.116	68	0.91176	31.105
awout_14	6849	172969	0.0073757698	80	0.149	94	0.85106	12.116
awout_04	8000	183940	0.0057488436	45	0.152	55	0.81818	4.518
awout_15	5439	20276	0.0013710523	36	0.017	38	0.94737	0.014
young_26	8183	211232	0.0063098303	103	0.192	124	0.83065	335.649
young_17	7553	134565	0.0047182467	84	0.091	94	0.89362	40.639
young_16	7032	131887	0.0053350197	88	0.102	108	0.81481	18.437
young_34	7131	93643	0.0036835396	66	0.069	89	0.74157	0.884
WS_500_25_50	500	25000	0.2004008016	13	0.011	16	0.81250	1.345
WS_500_40_50	500	25000	0.2004008016	9	0.012	11	0.81818	0.892
WS_5000_25_50	5000	250000	0.0200040008	13	0.165	16	0.81250	13.091
WS_5000_40_50	5000	250000	0.0200040008	8	0.172	11	0.72727	8.630
WS_10000_25_50	10000	500000	0.0100010001	14	0.405	16	0.87500	25.030
WS_10000_40_50	10000	500000	0.0100010001	9	0.485	12	0.75000	15.354

Table 1: Results of test runs of our algorithm for different datasets. Times are in seconds.