2013

An empirical study of 3-vertex connectivity algorithms

Zhigang Jiang
Universty of Windsor

AN EMPIRICAL STUDY OF 3-VERTEX CONNECTIVITY ALGORITHMS

by

ZHIGANG JIANG

A Thesis

Submitted to the Faculty of Graduate Studies

through Computer Science

in Partial Fulfillment of the Requirements for

the Degree of Master of Science at the

University of Windsor

Windsor, Ontario, Canada

2013

AN EMPIRICAL STUDY OF 3-VERTEX CONNECTIVITY ALGORITHMS

by

ZHIGANG JIANG

APPROVED BY:

J. Wu

Electrical and Computer Engineering

D. Wu

Computer Science

Y. H Tsin, Advisor

Computer Science

September 11, 2013

Declaration of originality

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

Abstract

Graph connectivity is one of the most basic properties of graph. Owing to this reason, it is fundamental to the studies of many important applications such as network reliability, cluster analysis, graph optimization, quantum physics, bioinformatics and social networks. Triconnectivity is a topic in graph connectivity which has been used in graph drawing, graph decomposition in geometry constraint solver, and social network studies. Hopcroft and Tarjan (1973) proposed the first linear-time algorithm for this problem. Although elegant, this algorithm is very complex and contains many minor but crucial errors which make it very difficult to understand and implement correctly. Gutwenger and Mutzel (2001) published a list of errors, outlining how to fix them and implemented the corrected algorithm.Recently, Tsin (2012) proposed a new linear-time algorithm which is based on a new graph transformation technique. Tsin's algorithm is conceptually very simple and performs one less pass over the given graph than Hopcroft et al. These make the algorithm much easier to implement. In this thesis , we implemented Tsin's algorithm and compare its performance with Gutwenger and Mutzel's implementation of the algorithm of Hopcroft and Tarjan by carrying out an empirical study.

Dedication

I would like to dedicate this thesis to my girlfriend, my parents and my aunt's family.

Acknowledgements

I would like to express my gratitude to my supervisor Dr. Yung H. Tsin for his invaluable assitance, patience and guidance. The most important is that he teaches me to be an independent thinker. Without his support and help, I would not have been able to write this thesis.

Besides my supervisor, I would like to thank the rest of my thesis committee: Dr. Dan Wu, Dr. Jonathan Wu, and Dr. Jianguo Lu for their engagement, insightful comments.

Contents

Declaration of originality iii

Abstract iv

Dedication v

Acknowledgements vi

List of Figures x

1 Introduction 1

1.1 Graph 1

1.2 Depth-first search 4

1.2.1 Adjacency list 6

1.2.2 Palm tree 7

1.3 Some definitions related to DFS 8

1.4 Graph Connectivity 9

1.4.1 Applications 9

1.4.2 Some definitions 10

2 3-vertex connectivity 12

2.1 Triconnected graph 12

2.2 Split graph 13

2.3 Split components and Triconnected components 14

2.4 Previous work and two algorithms for empirical study 15

3 Two Algorithms **17**

3.1 Hopcroft and Tarjan's Algorithm 17

3.1.1 Key idea 17

3.1.2 Finding Separation Pair 18

3.1.3 Finding Split Components 19

3.2 Tsin's Algorithm 20

3.2.1 Millipede 22

3.2.2 Two Transformations 23

3.2.3 The Algorithm 24

4 Implementation **26**

4.1 Modifying Gutwenger and Mutzel's code for Hopcroft and Tarjan's algorithm . . . 26

4.2 Randomly generate the biconnected input graph 29

4.3 Creating adjacency list 30

4.4 Determining ancestor or descendant relationship 31

4.5 Representation of the millipede 32

4.6 Handling incoming frond 33

4.6.1 How to compute an initial value for f' 39

4.6.2 Time to compute f' 41

4.7 Performing the coalesce operation 42

4.8 Finding separation pair 43

4.9 Creating triple bonds 44

4.9.1 Determining if frond $u \hookrightarrow w$ exists 44

4.9.2 Determining if frond $w \hookrightarrow lowpt1(u)$ exists 45

5 Comparison **47**

5.1 Platform 47

5.2 Data Set 48

5.3 Results 49

5.3.1 Dense graph comparison 49
5.3.2 Sparse graph comparison 75

6 Conclusion **95**

Bibliography **96**

Vita Auctoris **99**

List of Figures

1.1 Examples for directed graph, undirected graph and multigraph. 3

1.2 $G = (V, E)$ 5

1.3 An adjacency-list structure for the graph G in Figure 1.2 6

1.4 The palm tree of graph G. The solid edge denoted tree-edge and the dotted edge denoted frond edge 7

1.5 2-vertex connected component of graph G 11

1.6 Two and three edge-connected component of graph G 11

2.1 Two special conditions for separation pair 13

2.2 Graph G and its split graphs 14

2.3 Graph G and its split components 15

3.1 Type-1 and Type-2 separation pair of Hopcroft and Tarjan's algorithm 18

3.2 separation pair (graph taken from Tsin (2012)) 21

3.3 Example of super graph (graph from Tsin (2012)) 22

3.4 Example of millipede (graph from Tsin (2012)) 23

4.1 Example of stack overflow occurs before modifying Gutwegner and Mutzel's code 28

4.2 Example of ancestor and descendant relationship 31

4.3 Incoming frond of vertex u when w is a first descendant of u 33

4.4 Incoming frond of vertex u when w is not a first descedant of u 34

4.5 Handling incoming frond of vertex u when w is a first descendant of u 35

4.6 Handling incoming frond of vertex u when w is not a first descendant of u but is a first descendant of v. 36

4.7 Handling incoming frond of vertex u when w is not a first descendant of u and is not a first descendant of v. 37
4.8 Example of $path(v)$ and $fork[v]$ 40
4.9 Compute f' 41
4.10 Example of different incoming fronds of vertex u 42
4.11 Example of the existence of frond $u \hookrightarrow w$. 44
4.12 Example of the existence of frond $u \hookrightarrow lowpt1(u)$. 45

5.1 Total execution time (dense graphs with $0 < k \leq 0.1$) 51
5.2 Total execution time (dense graphs with $0.1 < k \leq 0.2$) 52
5.3 Total execution time (dense graphs with $0.2 < k \leq 0.3$) 53
5.4 Total execution time (dense graphs with $0.3 < k \leq 0.4$) 54
5.5 Total execution time (dense graphs with $0.4 < k \leq 0.5$) 55
5.6 Total execution time (dense graphs with $0.5 < k \leq 0.6$) 56
5.7 Total execution time (dense graphs with $0.6 < k \leq 0.7$) 57
5.8 Total execution time (dense graphs with $0.7 < k \leq 0.8$) 58
5.9 Time required to create the adjacency-lists (dense graphs with $0 < k \leq 0.1$) 59
5.10 Time required to create the adjacency-lists (dense graphs with $0.1 < k \leq 0.2$) . . . 60
5.11 Time required to create the adjacency-lists (dense graphs with $0.2 < k \leq 0.3$) . . . 61
5.12 Time required to create the adjacency-lists (dense graphs with $0.3 < k \leq 0.4$) . . . 62
5.13 Time required to create the adjacency-lists (dense graphs with $0.4 < k \leq 0.5$) . . . 63
5.14 Time required to create the adjacency-lists (dense graphs with $0.5 < k \leq 0.6$) . . . 64
5.15 Time required to create the adjacency-lists (dense graphs with $0.6 < k \leq 0.7$) . . . 65
5.16 Time required to create the adjacency-lists (dense graphs with $0.7 < k \leq 0.8$) . . . 66
5.17 Time required to find split components (dense graphs with $0 < k \leq 0.1$) 67
5.18 Time required to find split components (dense graphs with $0.1 < k \leq 0.2$) 68
5.19 Time required to find split components (dense graphs with $0.2 < k \leq 0.3$) 69
5.20 Time required to find split components (dense graphs with $0.3 < k \leq 0.4$) 70

5.21 Time required to find split components (dense graphs with $0.4 < k \leq 0.5$) 71

5.22 Time required to find split components (dense graphs with $0.5 < k \leq 0.6$) 72

5.23 Time required to find split components (dense graphs with $0.6 < k \leq 0.7$) 73

5.24 Time required to find split components (dense graphs with $0.7 < k \leq 0.8$) 74

5.25 Total execution time (sparse graphs with $1 \leq \frac{|E|}{|V|} < 1.1$) 76

5.26 Total execution time (sparse graphs with $1.1 \leq \frac{|E|}{|V|} < 1.3$) 77

5.27 Total execution time (sparse graphs with $1.3 \leq \frac{|E|}{|V|} < 2$) 78

5.28 Total execution time (sparse graphs with $2 \leq \frac{|E|}{|V|} < 5$) 79

5.29 Total execution time (sparse graphs with $5 \leq \frac{|E|}{|V|} < 10$) 80

5.30 Total execution time (sparse graphs with $10 \leq \frac{|E|}{|V|} < 100$) 81

5.31 Time required to create the adjacency-lists (sparse graphs with $1 \leq \frac{|E|}{|V|} < 1.1$) . . . 82

5.32 Time required to create the adjacency-lists (sparse graphs with $1.1 \leq \frac{|E|}{|V|} < 1.3$) . . 83

5.33 Time required to create the adjacency-lists (sparse graphs with $1.3 \leq \frac{|E|}{|V|} < 2$) . . . 84

5.34 Time required to create the adjacency-lists (sparse graphs with $2 \leq \frac{|E|}{|V|} < 5$) 85

5.35 Time required to create the adjacency-lists (sparse graphs with $5 \leq \frac{|E|}{|V|} < 10$) . . . 86

5.36 Time required to create the adjacency-lists (sparse graphs with $10 \leq \frac{|E|}{|V|} < 100$) . . 87

5.37 Time required to find split components (sparse graphs with $1 \leq \frac{|E|}{|V|} < 1.1$) 88

5.38 Time required to find split components (sparse graphs with $1.1 \leq \frac{|E|}{|V|} < 1.3$) 89

5.39 Time required to find split components (sparse graphs with $1.3 \leq \frac{|E|}{|V|} < 2$) 90

5.40 Time required to find split components (sparse graphs with $2 \leq \frac{|E|}{|V|} < 5$) 91

5.41 Time required to find split components (sparse graphs with $5 \leq \frac{|E|}{|V|} < 10$) 92

5.42 Time required to find split components (sparse graphs with $10 \leq \frac{|E|}{|V|} < 100$) 93

Chapter 1

Introduction

1.1 Graph

A *graph*, denoted by $G = (V, E)$, consists of a set of vertices V and a set of edges E such that every edge in E is associated with two vertices in V. The graph is an *undirected graph* if the edges are associated with unordered pairs of vertices, represented by (v, w). The graph is a *directed graph* if the edges are associated with ordered pairs of vertices represented by $w \rightarrow v$, where w is called the *tail* and v is the *head* of the edge. An edge e associated with an unordered pair (v, w) in an undirected graph is denoted by $e = (v, w)$. An edge e associated with an ordered pair $(v \rightarrow w)$ in a directed graph is denoted by $e = (v \rightarrow w)$.

The followings are some graph related definitions.

End-point

Let $e = (v, w)$ be an edge in an undirected graph. The vertices v and w are called the *end-points* of edge e.

Incident

Let $e = (v, w)$ be an edge. Edge e is *incident* to its end-points v and w.

Adjacent

Let (v, w) be an edge in a graph G. Vertices v and w are *adjacent* in G and are *neighbor* of each other.

Degree

In an undirected graph, the *degree* of a vertex v in a graph G, denoted by $deg_G(v)$, is the number of edges incident to v.

In a directed graph, the *indegree* of a vertex v, denoted by $indeg_G(v)$, is the number of edges with v as their head and the *outdegree* of a vertex v, denoted by $outdeg_G(v)$, is the number of edges with v as their tail.

Path

A *path* P in a graph G, denoted by P: $v \xrightarrow{*} w$, is a sequence of vertices and edges leading from v to w. A path is *simple* if all of its vertices are distinct. A *path* P: $v \xrightarrow{*} w$ is a cycle if all of its edges are distinct and the only vertex to occur twice in P is v, which occurs exactly twice.

Multigraph

If two or more edges having the same end-vertices in a graph G, then G is a *multigraph*. The *undirected version* of a directed graph is the graph formed by converting each edge of the directed graph into an undirected edge.

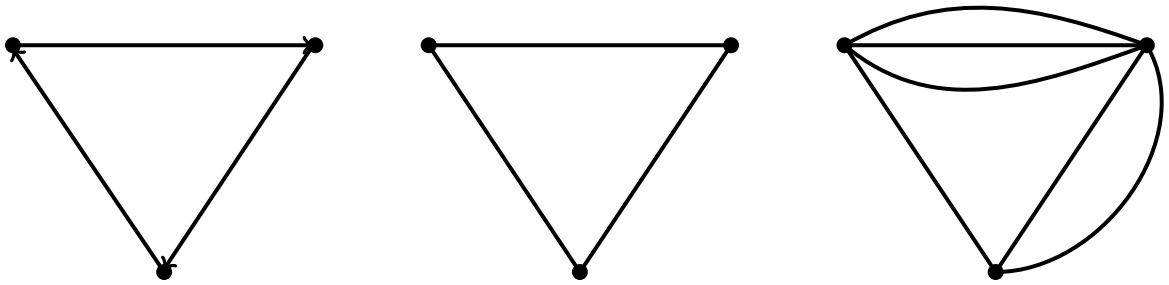


Figure 1.1: Examples for directed graph, undirected graph and multigraph.

Connected graph

A graph is *connected* if every pair of vertices in it is *connected* by a path.

Subgraph

If $G = (V, E)$ and $G' = (V', E')$ are two graphs such that $V' \subseteq V$ and $E' \subseteq E$, then G' is a *subgraph* of G.

Rooted tree

A *rooted tree* T is a directed graph that has exactly one vertex which is the head of no edges (called the *root*) and that all vertices except the root are the head of exactly one edge.

Leaf, Parent and Child

In a rooted tree, a *leaf* is a vertex with outdegree 0. Vertex v is the *parent* of vertex w and w is a *child* of v if $v \rightarrow w$ is a tree-edge in the rooted tree.

Ancestor and Descendant

In a rooted tree, if there exists a path from v to w, denoted by $v \xrightarrow{*} w$, then v is an *ancestor* of w and w is a *descendant* of v.

Spanning tree

If G is a directed graph, a rooted tree T is a *spanning tree* of G if T is a subgraph of G and contains all the vertices of G.

1.2 Depth-first search

Depth-first search (*DFS*), developed by Tarjan and coauthors, is a fundamental technique of efficient algorithm design for graphs (Tarjan (1972)). It has been widely used in solving a variety of graph-theoretic problems including the connectivity problems. It was used to determine the biconnected components of an undirected graph and the strong connected components of a directed graph. This technique had also been used in an efficient algorithm for planarity testing.

In this thesis, we study algorithms that use depth-first search to find the triconnected components of a graph. We shall thus briefly explain the basic idea underlying depth-first search. Let $G = (V, E)$ be a graph to be explored by depth-first search. Initially all vertices are marked as 'unvisited' and all edges are marked as 'unexplored'.

1. Starting from a vertex v, called the *root*, which could be any vertex in G.
2. An unexploded edge incident to v is arbitrarily chosen and mark the edge as exploded. Let w be the other end-point of the edge,
   - if vertex w is unvisited, then mark w as visited and then continue depth-first search

from w.

- if vertex w is visited, then repeat this step.

3. When there is no unexplored edges incident to vertex v, the search backtracks to the vertex u leading to vertex v and then continue depth-first search from u.

4. When the depth-first search backtracks to the root and there is no unexplored edges incident to the root, the search terminates.

Since each vertex is only visited once and each edge is examined twice (once from each endpoint), The time complexity of depth-first search is thus $O(|V| + |E|)$, where V is the set of vertices and E is the set of edges.

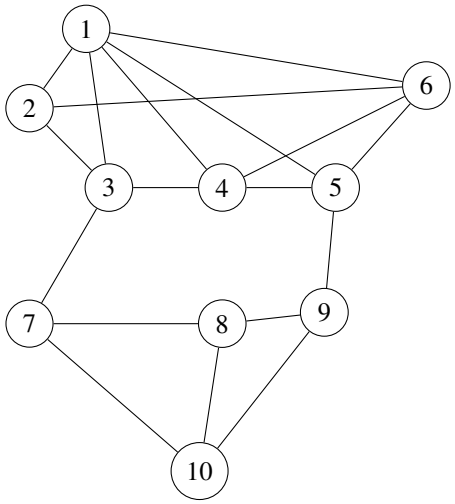


Figure 1.2: $G = (V, E)$

1.2.1 Adjacency list

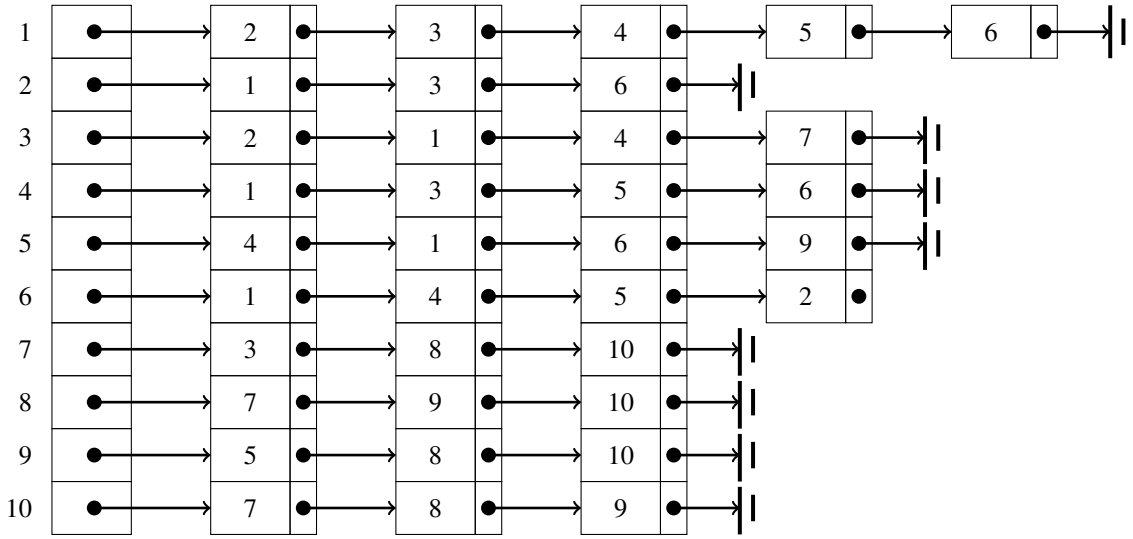


Figure 1.3: An adjacency-list structure for the graph G in Figure 1.2

An *adjacency-lists* structure is a representation of a graph. Each linked-list in it describes the set of the neighbors of the vertex. Specifically, let $G = (V, E)$ be a graph. For each vertex v in V, the linked-list of vertex v contains all of the vertices w in graph G such that $(v, w) \in E$. If G is a undirected graph, each edge (v, w) in graph G is represented twice, one is in the list of vertex v, and another one is in the list of vertex w. These lists together comprises an *adjacency list* data structure for graph G. Figure 1.3 shows an adjacency list of the graph in Figure 1.2.

Using the adjacency-lists representation, depth-first search can be done in linear time.

1.2.2 Palm tree

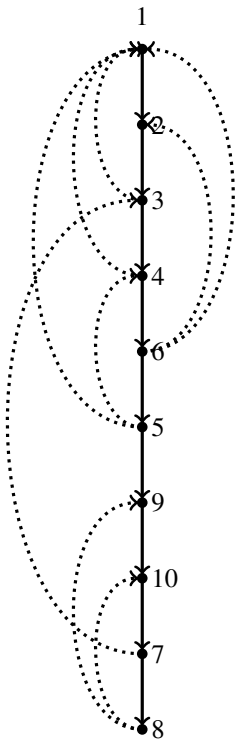


Figure 1.4: The palm tree of graph G. The solid edge denoted tree-edge and the dotted edge denoted frond edge

Let P_G be a directed graph, consisting of two disjoint sets of edges, denoted by $v \rightarrow w$ and $v \hookrightarrow w$ (called tree-edge and frond edge, respectively). Suppose P_G satisfies the following properties:

1. The subgraph T consisting of the tree edges is a spanning tree of P_G;
2. if $v \hookrightarrow w$, then $w \overset{*}{\rightarrow} v$. That is, each edge not in the spanning tree T of P_G connects a vertex with one of its ancestor in T.

Then P_G is called a *palm tree*.

Performing a depth-first search over a graph G transforms the graph into a palm tree. Let $v, w \in V$ and $e = (v, w) \in E$. Edge e is transformed into the tree-edge $v \rightarrow w$ if the depth-first search traverses edge e from v to w; edge e is transformed into the frond edge $v \hookrightarrow w$ if vertex w is

already visited when edge e is examined for the first time at vertex v.

The palm of graph G of Figure 1.2 is shown in Figure 1.4.

1.3 Some definitions related to DFS

DFS number

A depth-first search assigns an unique number to every vertex in the palm tree P_G, hence the graph G, called the *depth-first search number* of the vertex. The depth-first search number of vertex v is denoted by $dfs(v)$. The depth-first search number of vertex v is k if v is the k-*th* vertex visited by the DFS for the first time. Therefore, the *depth-first search number* of the root of spanning tree of P_G is 1.

Subtree

Let T be the spanning tree of the palm tree P_G. The *subtree* of T rooted at vertex w, denoted by T_w, is the maximal subgraph of T which is a rooted tree rooted at w.

Incoming frond edge and Outgoing frond edge

A frond edge $v \hookrightarrow w$ is an *incoming frond edge* of w and an *outgoing frond edge* of v.

Lowpt1(w) and *Lowpt2*(w), $\forall\ w \in V$

$lowpt1(w) = min\ (\{dfs(w)\} \cup \{dfs(u) \mid \exists (w \hookrightarrow u)\} \cup \{lowpt1(u) \mid u\ is\ a\ child\ of\ w\})$

$lowpt1(w)$ is the vertex with the smallest dfs number in the palm tree that is reachable from vertex w by traversing zero or more tree-edges followed by exactly one frond.

$lowpt2\ (w) = min\ (\{dfs(w)\}\ \cup\ ((\{dfs(u) \mid \exists (w \hookrightarrow u)\}\ \cup\ \{lowpt1(u) \mid u\ is\ a\ child\ of\ w\}\ \cup$

$\{lowpt2(u) \mid u \text{ is a child of } w\}) - \{lowpt1(w)\}))$

lowpt2 (w) is the vertex with the second smallest dfs number in the palm tree that is reachable from vertex w by traversing zero or more tree-edges followed by exactly one frond.

First child and First descendant

For each vertex w, the *first child* is a vertex v which is the first child of w satisfying $lowpt1(v) = lowpt1(w)$ during the depth-first search. The *first frond* of w is the first frond $w \hookrightarrow v$ encountered during the depth-first search that satisfies $dfs(v) = lowpt1(w)$. A *first descendant* of w is the first child of w or a first descendant of the first chid of w.

1.4 Graph Connectivity

1.4.1 Applications

Graph connectivity (*k-vertex-connectivity and k-edge-connectivity*) is one of the most basic properties of graph. Owing to this reason, it is fundamental to the studies of many important applications such as network reliability, circuit and chip design, network flow, cluster analysis, graph optimization, quantum physics, and bioinformatics.

The most direct applications arise in operation research for scheduling problems (Boffey (1992)) and performance analysis of telecommunication systems and transportation networks (Jungnickel (2008), Novak & Gibbons (2009)). The application of graph connectivity also arise in irreducibility analysis of Feynman diagrams in quantum physics and chemistry (Nakanishi (1971)); circuit lay-out problems (Ellis-Monaghan & Gutwin (2003)); planarity testing (Knauer (1975)); Flow edge-monitor optimization problem (Chin et al. (2009)); Graph Drawing (Gutwenger & Mutzel (2000)); and clustering algorithm (Hartuv & Shamir (2000)).

1.4.2 Some definitions

k-vertex(*k*-edge) connectivity

A connected undirected graph is *k-vertex-connected* (*k-edge-connected*) if removing less than k vertices (edges) cannot disconnect it.

k-vertex(*k*-edge) connected component

A *k-vertex-connected* (*k-edge-connected*) *component* of a graph is a maximal k-vertex-connected (k-edge-connected) subgraph.

1-vertex(edge)-connected graph

A *1-vertex-connected* (*1-edge-connected*, respectively) graph is simply a connected graph.

2-edge-connected graph

A *2-edge-connected graph* is also called a *bridge-connected* graph. A *bridge* of a graph G is an edge whose removal results in disconnecting G. A graph G is bridge-connected if there is no bridge in it.

2-vertex-connected graph

A *2-vertex-connected graph* is also called a *biconnected* graph.

Let $G = (V, E)$ be a connected, undirected graph with $|V| \geq 2$. A *cut vertex* is a vertex whose removal results in disconnecting the graph G. G is biconnected if there is no cut-vertex in it.

Tarjan presented the first linear-time algorithm for both biconnectivity and 2-edge-connectivity (Tarjan (1972)).

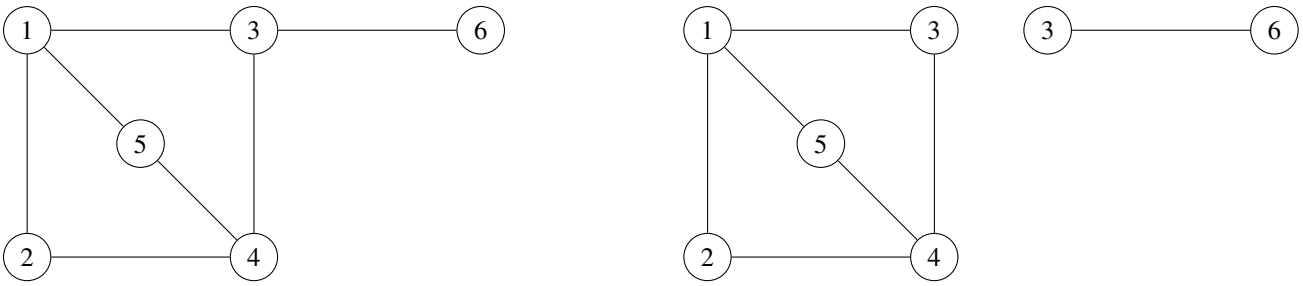


Figure 1.5: 2-vertex connected component of graph *G*

3-edge-connected Graph

Let *G* = (*V, E*) be a bridge-connected undirected graph. A pair of edges in *G* is a *cut-pair* if their removal results in disconnecting *G*. *G* is *3-edge connected* if there is no cut-pair in it.

The first linear-time algorithm for 3-edge-connectivity was reported by Galil & Italiano (1991). The algorithm reduces the problem to 3-vertex-connectivity which makes it very complicated.

Two simpler linear-time algorithms were reported by Taoka et al. (1992) and Nagamochi & Ibaraki (1992).

The conceptually simplest and fastest (in terms of actual run-time) linear-time algorithms were reported by Tsin (2007) and Tsin (2009).

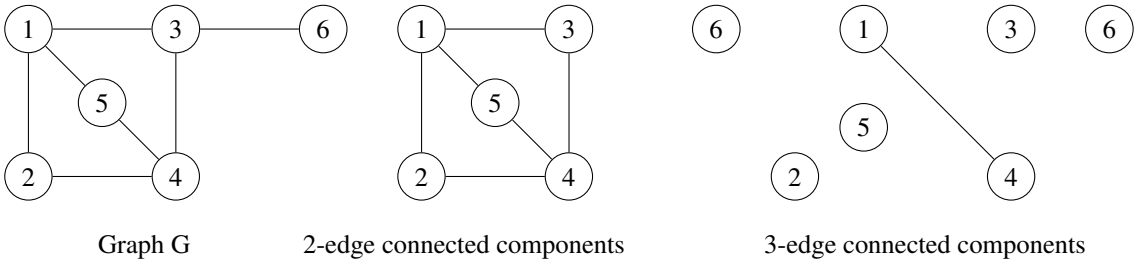


Figure 1.6: Two and three edge-connected component of graph *G*

Chapter 2

3-vertex connectivity

2.1 Triconnected graph

Equivalence relation

A *relation* in a set X is a set of ordered pairs from X. Let R be a relation in a set X. Then, R is *reflexive* if $(\forall x \in X)\ (x,\ x) \in R$; R is *symmetric* if $(x,\ y) \in R \Rightarrow (y,\ x) \in R$; R is *transitive* if $(x,\ y) \in R \wedge (y,\ z) \in R \Rightarrow (x,\ z) \in R$. A relation R in a set X is an *equivalence relation* if R is *reflexive*, *symmetric*, and *transitive*.

Equivalence class

Let R be an equivalence relation in a set X. For each $x \in X$, the *equivalnece class* of x with respect to R is the set $[x]_R = \{y \,|\, y \in X \wedge (x,\ y) \in R\}$.

In other words, the equivalence class of x with respect to R consists of all the elements of X which form an ordered pair with x in R, and in each of such ordered pairs, x is the first coordinate.

Triconnected graph

Let $G = (V, E)$ be a biconnected graph and $a, b \in V$. Let $\sim_{(a,b)}$ be a relation in E such that $\forall e, e' \in E, e \sim_{(a,b)} e'$ if and only if there exists a path containing both e and e' but not a or b except as a terminating vertex. The relation $\sim_{(a,b)}$ is an equivalence relation in E. Therefore, the edge set E can be partitioned into equivalence classes $E_1, E_2, E_3, \ldots\ldots, E_k$ such that any two edges on the common path not containing any vertex of $\{a, b\}$ as an internal vertex are in same equivalence class. $E_i, 1 \leq i \leq k$, are the equivalence classes respect to $\sim_{(a,b)}$. The vertex pair $\{a, b\}$ is a *separation-pair* if there are at least two equivlence classes (i.e $k > 1$), unless:

1. $k = 2$ and $\exists\ i \in \{1, 2\}$, $E_i = \{(a, b)\}$, or
2. $k = 3$ and $E_i = \{(a, b)\}$, $1 \leq i \leq 3$.

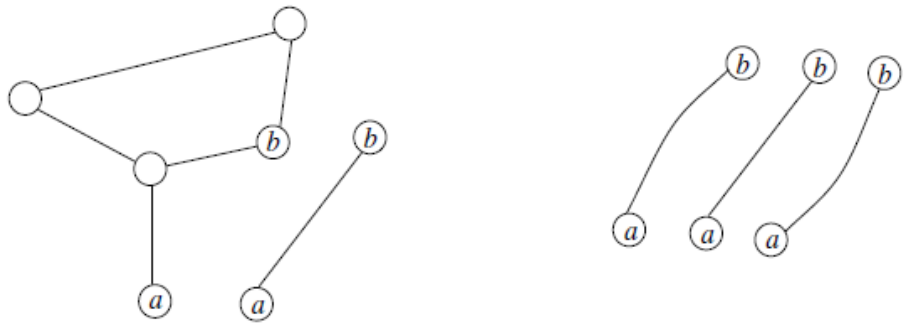


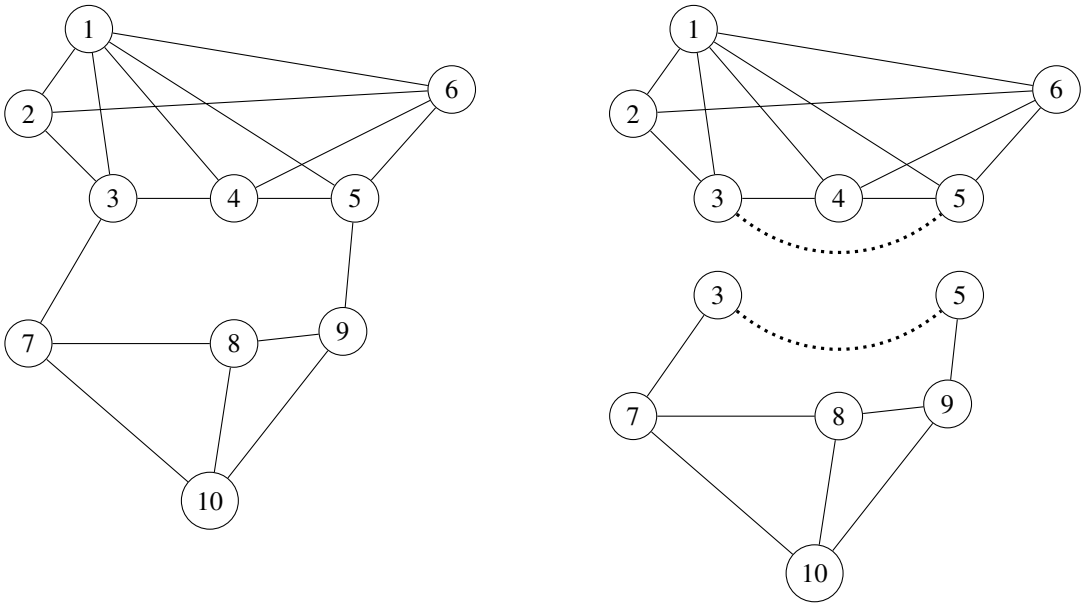
Figure 2.1: Two special conditions for separation pair

A biconnected graph is *3-vertex-connected* if there is no separation pair in it. A 3-vertex-connected graph is also called a *triconnected* graph.

2.2 Split graph

Let $\{a, b\}$ be a separation pair, and $E_1, E_2, E_3, \ldots\ldots, E_k$ be the equivalence classes w.r.t. $\sim_{(a,b)}$. Let $E' = \bigcup_{i=1}^{h} E_i$ and $E'' = \bigcup_{i=h+1}^{k} E_i$ such that $\left|E'\right|, \left|E''\right| \geq 2$. Let $G_1 = (V_1, E' \cup \{e\})$, $G_2 = (V_2, E'' \cup \{e\})$ such that V_1 is the set of vertices which are the end-points of edges in E', V_2 is the set

of vertices which are the end-points of edges in $E^{"}$, and e is a new edge (a, b), called a virtual edge. G_1, G_2 are called *split graphs* of G.



Graph G

split graphs

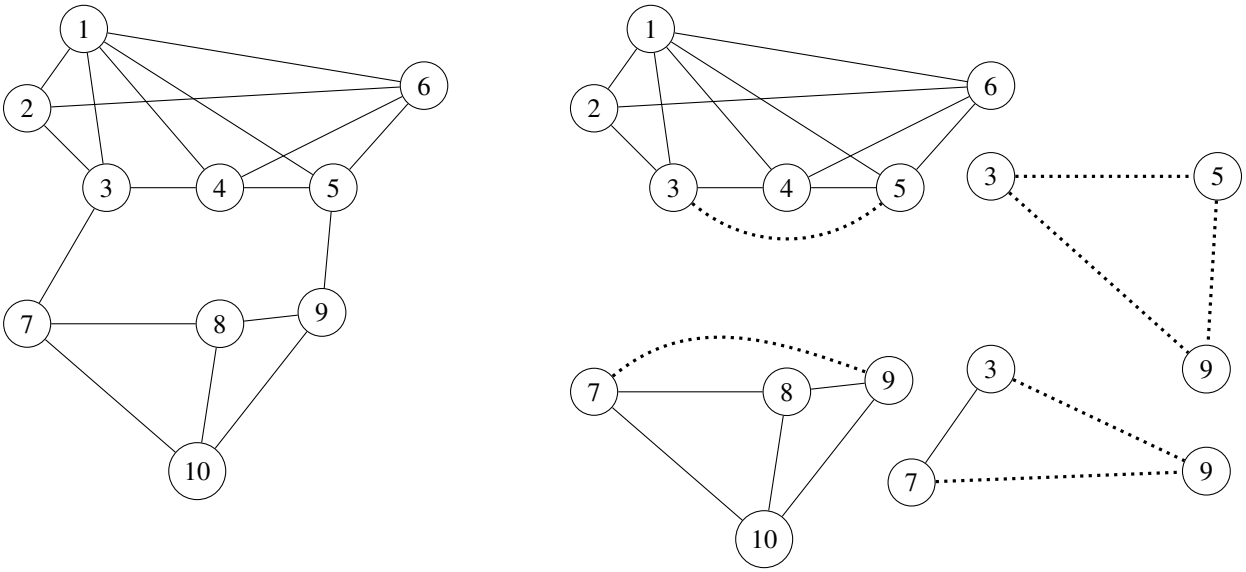
Figure 2.2: Graph G and its split graphs

2.3 Split components and Triconnected components

Let G be an undirected biconnected graph. The graph G is split into two split graphs G_1 and G_2. G_1 (G_2, respectively) is then split into two smaller split graphs, and so on, until no more splitting is possible. The resulting split graphs are the *split components* of G. A split component is one of the followings:

- a triple bond
- a triangle (cycle of length three)
- a triconnected simple graph

Let B_3 bet the set of triple bonds, T_3 be the set of triangles, and G_n be the set of triconnected simple graphs. The triple bonds in B_3 which have common edge are merged into multiple bonds until no more merging is possible. Let the resulting set of multiple bonds be B. Similarly, the triangles in T_3 that have common edge are merged into polygons as much as possible until no more merging is possible. Let the resulting set of polygons be T. Then $B \cup T \cup G_n$ are the *triconnected components* of G.



Graph G. Split components of graph G.

Figure 2.3: Graph G and its split components

2.4 Previous work and two algorithms for empirical study

Hopcroft & Tarjan (1973) presented the first linear-time algorithm. Unfortunately, this algorithm contains quite a number of minor but crucial errors which make it hard to understand and difficult to implement correctly so that the resulting program does run in linear time. Gutwenger & Mutzel (2001) presented a list of such errors and explained how to correct them. However, their explanation for some errors were brief, and no detailed explanation on implementation was given. Nevertheless, they had implemented the corrected algorithm and the code is available at

Gutwenger & Mutzel (2000). Recently, Mallach (2011) revealed further inaccuracies in Hopcroft & Tarjan (1973) and provided more comprehensive description of the algorithm.

Miller & Ramachandran (1992) and Fussell et al. (1989) presented parallel algorithms on the PRAM (Parallel RAM). The parallel algorithms can be converted into sequential algorithms that run in linear-time. However, the resulting algorithms are much more complicated than Hopcroft & Tarjan (1973) and obviously less efficient in terms of actual run-time.

Vo (1983) presented a linear-time algorithm which resembles that of Hopcroft & Tarjan (1973). But no detail on implementation was given.

Saifullah & Üngör (2009) showed that triconnectivity can be reduced to 3-edge-connectivity in linear time, making it possible to use the simpler 3-edge-connectivity algorithms to solve the triconnectivity problem.

Recently, Tsin (2012) presented a new linear-time triconnectivity algorithm that is conceptually simple. The algorithm

- uses a new graph transformation technique in conjunction with the depth-first search technique.
- avoids the time-consuming acceptable adjacency-lists construction required by Hopcroft & Tarjan (1973).
- makes one less pass over the given graph than Hopcroft & Tarjan (1973).

In this thesis, we perform an empirical study on Tsin's algorithm and the algorithm of Hopcroft and Trajan. The study is based on our implementation of Tsin's algorithm and the implementation of Hopcroft & Tarjan (1973) by Gutwenger & Mutzel (2000).

Chapter 3

Two Algorithms

3.1 Hopcroft and Tarjan's Algorithm

3.1.1 Key idea

Let $G = (V, E)$ be a biconnected graph and a, b be two vertices lying on a cycle C in G. Let C be partitioned into two simple paths p_1 and p_2 by a and b. Then $\{a, b\}$ is a separation pair if and only if one of the following cases holds.

- Type-1 Case: $\exists$ a segment S with at least two edges that has only a and b in common with C;
- Type-2 Case: $\nexists$ a segment S containing a vertex v in p_1 and a vertex w in p_2 such that: $v \notin \{a,b\}, w \notin \{a,b\}$, and p_1 and p_2 each contains a vertex besides a and b.

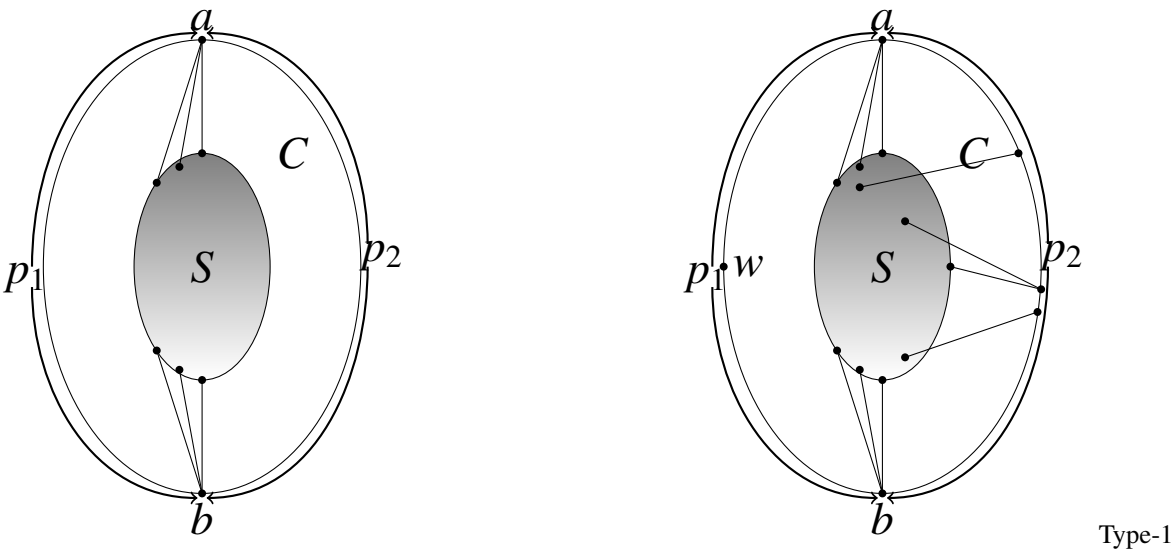


Figure 3.1: Type-1 and Type-2 separation pair of Hopcroft and Tarjan's algorithm

3.1.2 Finding Separation Pair

Given a biconnected graph $G = (V, E)$, the main problem for finding the split components of the given graph is to find the separation pair. The followings are the outline of finding the separation pair.

1. Perform a depth-first search over G to turn G into a palm tree P_G.
2. Create an acceptable adjacency-lists structure for P_G to rearrange the children and outgoing fronds of every vertex in a particular order.
3. Perform a depth-first search again using the acceptable adjacency-lists structure constructed in the previous step. Use a path-finding procedure to partition the edges of P_G into disjoint paths in the following way: each time an edge is traversed, append the edge to the current path; each time a frond is traversed, append the frond to the current path and terminate the current path; then start the current path with a *null* path. As a result, each path consists of a (possible empty) sequence of tree edges followed by a frond. Owing to the way the edges

are ordered in the acceptable adjacency-lists, each path ended in the vertex with the lowest possible dfs number. Furthermore, the first path is a cycle.

After the input graph G, which is a biconnected graph, is converted into a palm tree, if a, b are two vertices with $dfs(a) < dfs(b)$, then $\{a,\ b\}$ satisfies one of the following conditions if and only if $\{a,\ b\}$ is a separation pair.

1. There are two distinct vertices t and s, such that $b \rightarrow t$, $lowpt1(t) = a$, $lowpt2(t) \geq b$, $s \notin \{a, b\}$ and s is not a descendant of t (in this case, $\{a,\ b\}$ is a type-1 separation pair).
2. There is a vertex r such that $a \rightarrow r \overset{*}{\rightarrow} b$, where b is a proper first descendant of r, a is not the root, and every frond $x \hookrightarrow y$ with $r \preceq x \prec b$ has $a \preceq y$; every frond $x \hookrightarrow y$ with $a \prec y \prec b$ and $b \rightarrow w \overset{*}{\rightarrow} x$ has $lowpt1(w) \geq a$ (in this case, $\{a, b\}$ is a type-2 separation pair).
3. ($a,\ b$) is a multiple edge of G and G contains at least four edges.

In Figure 1.4, $\{3,\ 9\}$ is a type-1 separation pair, $\{3,\ 5\}$ and $\{7,\ 9\}$ are type-2 separation pair.

3.1.3 Finding Split Components

Repeatedly use the path-finding procedure to search for a path and use the path to find split components.

1. Maintain a stack of edges (called ESTACK); add edges to this stack as backing up over them during the depth-first search.
2. Maintain a stack of triples ($h,\ a,\ b$) (called TSTACK) such that $\{a,\ b\}$ is a possible type-2 pair and h denotes the largest numbered vertex in the corresponding split component.
3. On finding a separation pair, edges on the ESTACK are popped to form the corresponding split component.

4. Add the corresponding virtual edge to both the split component and the ESTACK.

5. Maintain various pieces of information, such as:

   - *parent*(v): the parent of vertex v in the depth-first search spanning tree.
   - *Degree*(v): the degree of vertex v.
   - *lowpt1*(v): the vertex with the smallest dfs number among the vertices in P_G that can be reached via a tree-path following by one frond.
   - *lowpt2*(v): the vertex with the second smallest dfs number among the vertices in P_G that can be reached via a tree-path following by one frond.

The pseudocode for finding the split components is as follows:

Type-1 split components

1: On backing up over a tree edge $v_i \rightarrow v_{i+1}$ during the path-finding search.
2: **if** lowpt2(v_{i+1}) $\geq v_i \wedge$ lowpt1(v_{i+1}) $\prec v_i \wedge$ (parent(v_i) $\neq$ root $\vee$ v_i is adjacent to a not yet visited tree-edge) **then**
3: $\{v_i, lowpt1(v_{i+1})\}$ is a type-1 separation pair; pop the ESTACK until an edge (x, y) does not satisfy $v_{i+1} \leq x$, $y \leq v_{i+1} + ND(v_{i+1})$ is encountered, where $ND(v_{i+1})$ is the number of descendants of v_{i+1} in the DFS spanning tree.
4: **end if**

Type-2 split components

3.2 Tsin's Algorithm

In the flowing figures (a) and (b), it is obvious that the vertex pair $\{a, b\}$ is a separation pair and the triangle $a\ e_1\ w\ e_2\ b\ e^{'}a$ is a split component, where $e_1 = (a,w), e_2 = (w,b)$ and $e^{'} = (b,a)$ is a virtual edge.

1: Use TSTACK to find separation pairs in the following way:
   On backing up over a tree edge $v_i \rightarrow v_{i+1}$ during the path-finding search
2: **if** $v_i \neq$ root **then**
3: examine the top triple (h_1, a_1, b_1) on TSTACK.
4: **if** $a_1 = v_i$ and $a_1 = parent(b_1)$ **then**
5: pop (h_1, a_1, b_1), discard it and repeat this step.
6: **end if**
7: **if** $a_1 = v_i$ and $a_1 \neq parent(b_1)$ **then**
8: $\{a_1, b_1\}$ is a type-2 separation pair. Repeatedly pop edges from ESTACK to form a split component until an edge (x, y) does not satisfy $a_1 \leq x$, $y \leq h_1$ is encountered. Pop the top entry and repeat this step with the new top entry.
9: **end if**
10: **if** $Degree(v_{i+1}) = 2$ and v_{i+1} has a child x **then**
11: $\{v_i, x\}$ is a type-2 separation pair. Pop the top two entries from ESTACK. Add virtual edge (v_i, x).
12: **end if**
13: **end if**

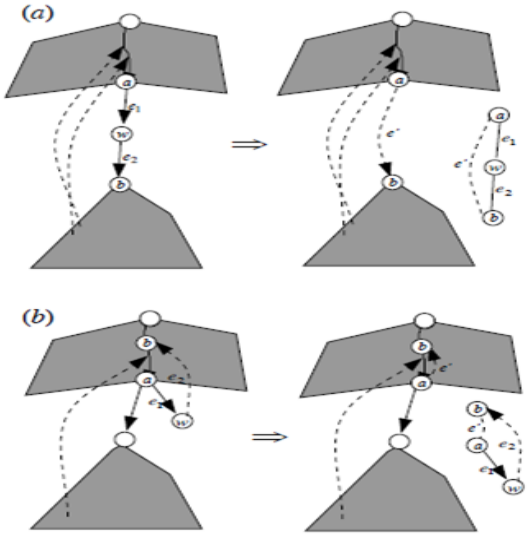


Figure 3.2: separation pair (graph taken from Tsin (2012))

Obviously, not every split component has such simple structure. Tsin (2012) transforms the input graph gradually during a depth-first search so that every split component is transformed into a special structure, called millipede, consisting of two or more superedges (the edges on the $a-b$ path in the following figure (a), (b)) such that no outgoing edge of the superedges or of the internal vertices on the millipede has its head outside the millipede. This condition will be detected when the search backtracks to one of the end-vertices (vertex a in the following figures (a) and (b)). A split component will then be created.

A millipede has a structure similar to the simple structure depicted in Figure 3.2. A *superedge* is an edge representing a set of edges. A supergraph is a graph whose edges are superedges. Tsin (2012) transforms the input graph to a supergraph, keeping the separation pairs intact.

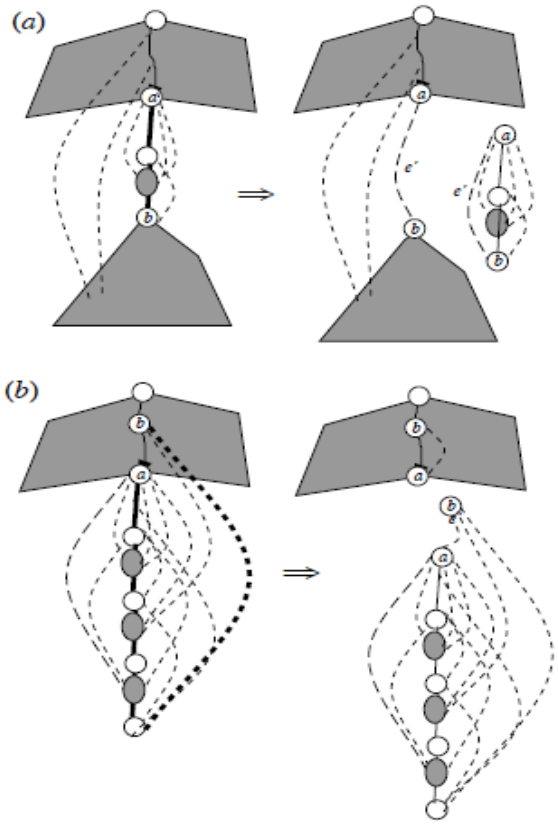


Figure 3.3: Example of super graph (graph from Tsin (2012))

3.2.1 Millipede

Millipede is a special data structure defined in Tsin (2012) for graph transformation. A *millipede*, denoted by $\hat{T}_0 e_1 \hat{T}_1 e_2 \hat{T}_2 \ldots e_k \hat{T}_k$, is a supergraph in which e_i $(1 \leq i \leq k)$ is a superedge associated with a set of edges (tree edge or frond edge) of the palm tree and $\hat{T}_i$ $(0 \leq i \leq k)$ is a tree rooted at u_i with height at most 1. The tree-path $u_0 e_1 u_1 e_2 u_2 \ldots e_k u_k$, where $u_i, 1 \leq i \leq k$, is the root of $\hat{T}_i$ is called the *spine* of the millipede. The edges in the $\hat{T}_i$'s (which are also superedges) are the *leg*s of the millipede. A frond edge, $(x \hookrightarrow y)$, is an outgoing frond edge of a superedge e if the

tail x is inside e. The outgoing frond edges of a millipede consists of the outgoing frond edges of all superedges in the millipede and the outgoing frond edges of all vertices in the millipede. The set of outgoing frond edges of a superedge e and of a vertex u are denoted by $Outfrond(e)$ and $Outfrond(v)$, respectively.

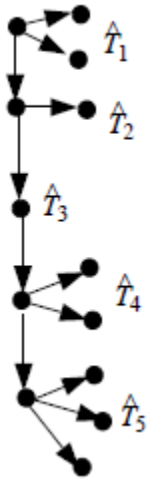


Figure 3.4: Example of millipede (graph from Tsin (2012))

3.2.2 Two Transformations

Tsin (2012) uses two transformations, split and coalesce, to transform the given graph.

Split

Separate a millipede from a supergraph so as to produce a split component.

Coalesce

Applied to a millipede whose spine consists of two superedges after the internal vertex of the spine has been confirmed to be unable to form new separation pair.

As an example, suppose $e_1\hat{T}_1e_2$ is a millipede in which the spine is $u_0e_1u_1e_2u_2$, where $e_1 = (u_0,$

u_1) and $e_2 = (u_1, u_2)$. If a coalesce operation is applied to the millipede, the millipede is replaced by a new superedge $e_1^{'} = (u_0, u_2)$ such that $e_1^{'}$ represents all of the edges in the millipede and $Outfrond(e_1^{'})$ represents all of the outgoing frond edges of the millipede. The coalesce operation can be easily extended to millipedes having more than two superedges on its spine.

3.2.3 The Algorithm

The following is a brief description of Tsin's algorithm:

1. Perform a depth-first search over the input graph G to turn G into a palm tree P_G and create an adjacent-lists structure representing P_G such that the first entry of the linked list of vertex v is the first child or first frond of v.

2. A depth-first search is then performed over P_G based on its adjacency lists created in Step 1.

3. During the depth-first search, whenever the search backtracks from a vertex u to the parent vertex w, the subgraph of G consisting of the edge set of the subtree rooted at u and the outgoing fronds of that subtree has been transformed into a supergraph consisting of a set of split components and a millipede $\hat{P_u}$: $\hat{T}_0 e_1 \hat{T}_1 \ldots e_k \hat{T}_k f$, called the *u-millipede*, where f is an outgoing frond of u_k that reaches the highest vertex (the vertex with the smallest dfs number) in P_G .

4. If u is the first child of w.

   - If there is no outgoing frond of $e_0 \hat{T}_0 e_1$, where $e_0 = (w, u_0)$, with its head being a proper ancestor of w, then $\{w, u_1\}$ is a separation pair and a split operation is applied to P_u to make $e_0 \hat{T}_0 e_1$ a split component.
   - $\hat{P_u}$ then becomes $e_0 \hat{T}_1 e_2 \hat{T}_2 \ldots e_k \hat{T}_k f$, where $e_0 = (w, u_1)$ is a virtual link replacing $e_0 \hat{T}_0 e_1$.
   - If the aforementioned condition applies to e_0 again, then $\{w, u_2\}$ is a separation pair and a split operation is applied to P_u to make $e_0 \hat{T}_1 e_2$ a split component.

- This process is repeated until the aforementioned condition does not apply.
- Let the resulting millipede be $\hat{P}_u$: $e_0\hat{T}_h e_{h+1}\hat{T}_{h+1}\ldots e_k\hat{T}_k f$, where $f = (u_k, z)$ such that $dfs(z) = lowpt1(u_k)$.
- If there is no outgoing frond of $\hat{P}_u$ whose head is an internal vertex of the tree-path $z \overset{*}{\rightarrow} w$ and there is at least one vertex outside $\hat{P}_u$, then $\{w, z\}$ is a separation pair and the entire millipede $\hat{P}_u$ is removed to produce a split component.

5. Otherwise, coalesce the $u - millipede$ into a superedge $e_1 = (u_0, u_k)$ which becomes a leg of the tree rooted at w.

6. If there is a incoming frond, $u \hookrightarrow w$, of w. Coalesce the section of the millipede from w to u into a superedge $e_0 = (w, u_0)$

Chapter 4

Implementation

We implemented Tsin's algorithm using C++. For Hopcroft and Tarajan's algorithm, we use the implementation of Gutwenger & Mutzel (2000) which is also a C++ program. However, to ensure that the empirical study is based on large input sizes, we have to modify the implementation of Gutwenger and Mutzel.

4.1 Modifying Gutwenger and Mutzel's code for Hopcroft and Tarjan's algorithm

The run-time environment of a C++ program consists of three major memory segments: text segment, stack segment and heap segment.

The *text segment* is responsible for storing the compiled code of the C++ program.

The *stack segment* is a region of memory for storing temporary variables that are created in each program function (i.e. main program or subprogram). This segment is LIFO (last in, first Out) as it is a stack. When a variable is declared in a function, this variable is pushed onto the top of the stack. When execution of a function terminates, all variables that were pushed onto the stack

by the function are freed. However, the size of variables can be pushed onto each stack is limited (varies with the operating system).

The *heap segment* is a region of memory for storing dynamically declared variables. To allocate variables in heap segment, the system-defined function malloc() must be used in C and new in C++. To free the memory when it is not required any more, free() and delete are used in C and C++ respectively. The size limitation of this region is not restricted. However, the heap is slightly slower to be read from and written to, because it has to use pointer to be accessed.

Therefore, a large block of memory should be stored in heap segment and the relatively small size of variables are stored in the stack segment.

Gutwenger & Mutzel (2000) uses *recursion* to perform depth-first search, and the path finding search. Since recursion uses the stack segment to store the local variables, and the local variables are pushed onto the stack every time a recursive call to the function is invoked. The recursive function calls continue to require more and more stack memory which does not release until the recursive chain terminates. Stack overflow results when the memory allocation goes beyond what the stack segment is able to provide.

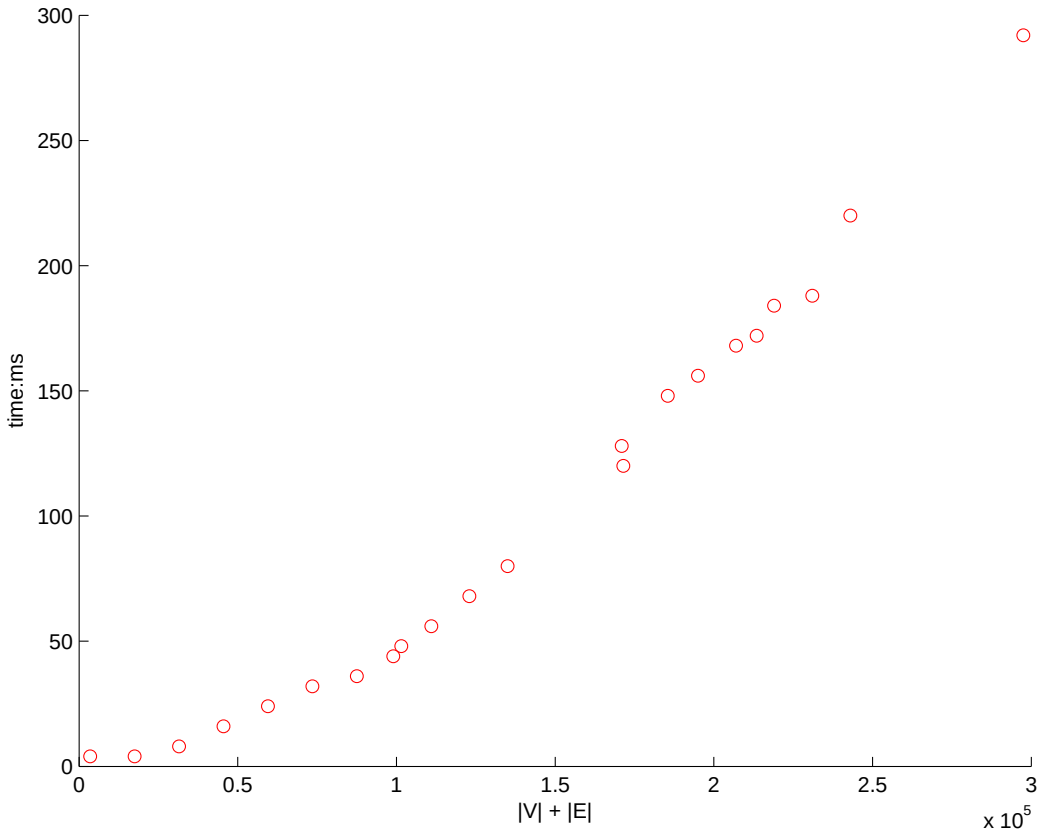


Figure 4.1: Example of stack overflow occurs before modifying Gutwegner and Mutzel's code

Figure 4.1 show that stack overflow occurs when the number of edges is 255,000 and the number of vertex is 42,500 in running Gutwegner and Mutzel's code.

To ensure that Gutwegner and Mutzel's code could handle input graph with millions of vertices and edges, we modified their code to use *iteration* to perform depth-first search and path finding search. This is accomplished by maintaining a self declared stack in the heap segment to record all of the function variables. After the modification, the largest input size that Gutwegner and Mutzel's code could handle is in the range of millions, significantly larger than 42,500.

4.2 Randomly generate the biconnected input graph

In order to get the best results of comparing these two algorithms, the input graphs must be randomly generated. The following describes how to randomly generate the biconnected input graphs:

1. Randomly generate random numbers n and $m, (m > n)$, which are the the number of vertices and edges, respectively, of the graph to be generated.
2. Generate a random number n' such that $n \geq n' > 3$. Then generate a set of n' edges to form a cycle. Set $m' \leftarrow n'$.
3. **If** $n' < n$ **then** generate a random number $b \in \{0, 1\}$.
   - If $b = 0$
     - Randomly choose two vertices v and w from the graph generated thus far.
     - Randomly generate an integer l in the range $[1..(n - n')]$.
     - Generate a path consisting of the following $l + 1$ edges:

       $$(v, v_1), (v_1, v_2) \ldots (v_i, v_{i+1}), \ldots (v_l, w),$$

       where $v_i, 1 \leq i \leq l$, are new vertices
     - Set $n' \leftarrow n' + l$; $m' \leftarrow m' + l + 1$.
   - If $b = 1$
     - Randomly choose two non-adjacent vertices v and w from the generated graph and generate an edge (v, w).
     - Set $m' \leftarrow m' + 1$.

   else if $m' < m$ then

- Randomly choose two non-adjacent vertices v and w from the generated graph and generate an edge (v,w).
- Set $m' \leftarrow m'+1$.

4. Repeat Step 3 until $m' = m$.

4.3 Creating adjacency list

In this and the following subsections, we explain how we implemented Tsin's algorithm based on the description given in Tsin (2012).

First, an adjacency-lists structure for the palm tree P_G created by the first depth-first search is to be created. In this linked-lists structure, the first entry of the linked list of vertex v is the first child or the first frond of v. This is accomplished as follow:

1. Initially all vertices are marked as 'unvisited' and all edges are marked as 'unexplored'. $A[w]$ is used to record all of the adjacent edges of vertex $w, \forall w \in V$.
2. Perform a depth-first search starting from an arbitrary vertex r (which becomes the root of the dfs spanning tree). Let $dfs = 1$, $v = r$ and $dfs(v) = lowpt1(v) = 1$.
3. Choose the next unexplored edge from $A[v]$. Mark this edge as explored, let w be the other end-point of this edge.
   - If w is unvisited, mark w as visited.

     Let $dfs(w) = lowpt1(w) = dfs+1; dfs = dfs+1$. Insert w into the first entry of $A[v]$. Continue the depth-first search from w.
   - If w is visited,
     - if $dfs(w) < lowpt1(v)$, insert w into the first entry of $A[v]$. Let $lowpt1(v) = dfs(w)$;

– otherwise, Insert w into the second entry of $A[v]$.

Repeat step 3.

4. When there is no unexplored edges incident to vertex v, the search backtracks to the vertex u leading to vertex v.

   - If $lowpt1(u) > lowpt1(v)$, let $lowpt1(u) = lowpt1(v)$.
   - Otherwise, switch the first two entries of $A[v]$.

   Continue depth-first search from u.

5. When the depth-first search backtracks to the root and there is no unexplored edges incident to the root, the search terminates.

4.4 Determining ancestor or descendant relationship

In Tsin's algorithm, to ensure that the coalesce transformation is performed efficiency, we need to test the ancestor or descendant relationship in $O(1)$ time. In the following example, vertex u is a descendant of vertex w.

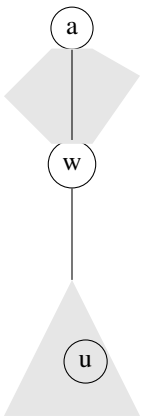


Figure 4.2: Example of ancestor and descendant relationship

To test if there exists an ancestor/descedant relation between two vertices v and w, we use the following criterion:
Vertex w is an ancestor of vertex u if and only if $dfs(w) \leq dfs(u) < dfs(w) + nd(w)$, where $nd(w)$ is the number of descendants of vertex w.

The term $nd(v)$ can be efficiently computed during the depth-first search using the following recursive definition.

$$nd(w) = \begin{cases} 1 & \textit{if } w \textit{ is a leaf;} \\ 1 + \sum_{v \in C(w)} nd(v) & \textit{otherwise.} \end{cases}$$

Note: $C(w)$ is the set of children of w

Knowing $dfs(w), dfs(v)$ and $nd(w)$, $dfs(w) \leq dfs(u) < dfs(w) + nd(w)$ can be evaluated in $O(1)$ time.

4.5 Representation of the millipede

The following data structure is used to represent a millipede $\hat{P}$: $\hat{T}_0 e_1 \hat{T}_1 \dots e_k \hat{T}_k$.

- a linked list $u_0 - u_1 - u_2 - \dots - u_n$, augmented with the following data structure:
  - *Outfrond*(v) $\forall v \in V$, the set of outgoing frond of vertex v.
  - $p(v)$ $\forall v \in V$, the parent vertex of vertex v. In the millipede, for every leg $(u_i \rightarrow v)$ in $\hat{T}_i$, $p(v) = u_i$.
  - $\tilde{e}_v$ $\forall v \in V$, where $\tilde{e}_v = (p(v) \rightarrow v)$. The edges in the superedge $\tilde{e}_v$ are divided into:
    - *Int*$(\tilde{e}_v)$, edges that are not outgoing fronds.
    - *Out*$(\tilde{e}_v)$, edges that are outgoing fronds.

Since we use linked list to represent *Outfrond*(v) $\forall v \in V$, and $\tilde{e}_v$ $\forall v \in V - \{r\}$ (r is the root), therefore, two superedges can be coalesced in $O(1)$ time.

4.6 Handling incoming frond

During the depth-first search, when a vertex u is examined and u has an incoming frond, then a section of the *u-millipede* is to be coalesced. Since the palm tree P_G is being transformed during the depth-first search, when a frond $f = (w \hookrightarrow u)$ is examined at vertex u, the frond could have been transformed to a frond $f' = (w' \hookrightarrow u)$. In the following, we shall illustrate how to determine f' in different situations.

The following figure is an example of an incoming frond of vertex u where w is a first descendant of u.

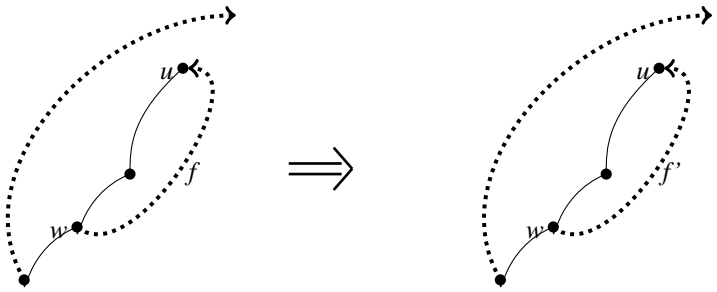


Figure 4.3: Incoming frond of vertex u when w is a first descendant of u

The following figure is an example of an incoming frond of vertex u where w is not a first descendant of u:

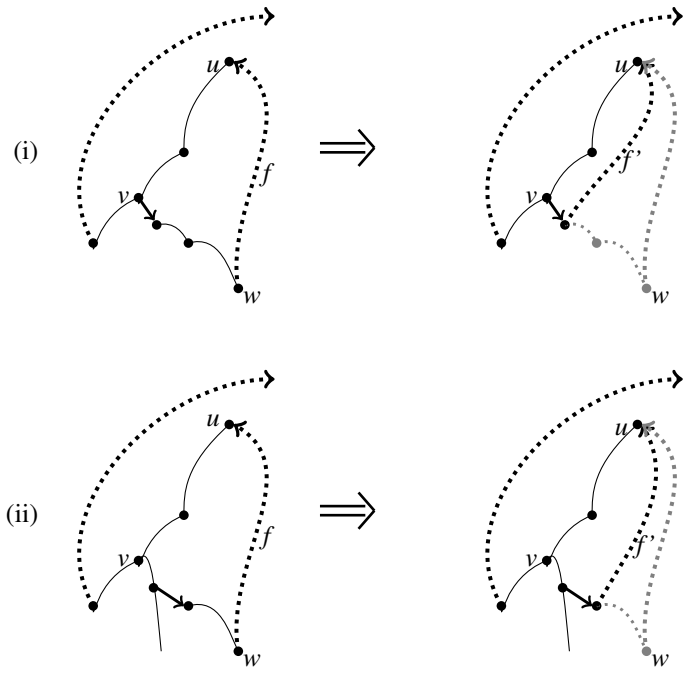


Figure 4.4: Incoming frond of vertex u when w is not a first descedant of u

In case (i), w is a first descendant of vertex v, whereas in case (ii), w is not a first descendant of vertex v.

As was mentioned above, when depth-first search backtracks to vertex u, frond f' instead of f is examined. The following figures illustrate how f is transformed to f' in different situations.

First, consider the situation in which w is a first descendant of u:

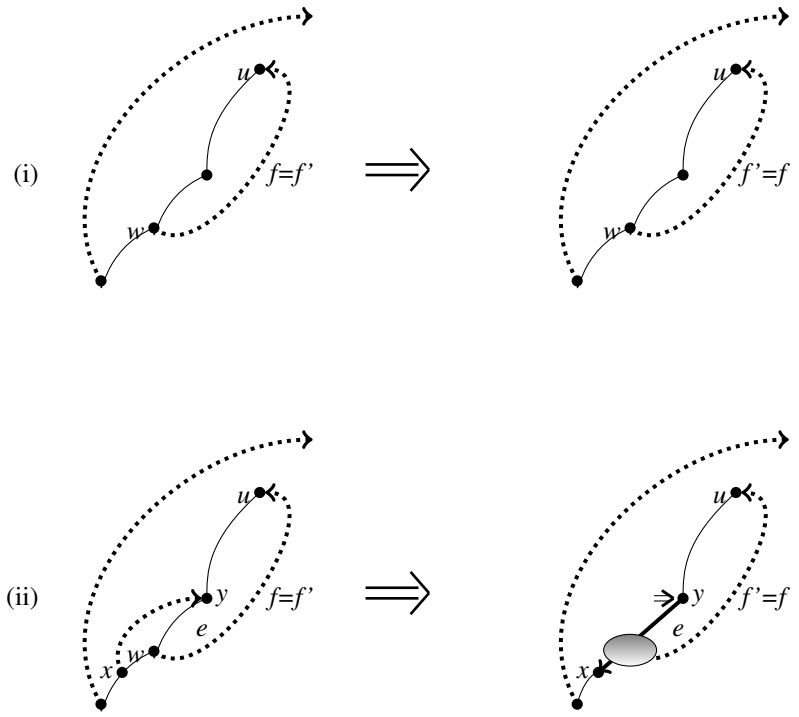


Figure 4.5: Handling incoming frond of vertex u when w is a first descendant of u

In case (i), $f = f'$. This case is trivial.

In case (ii), $x \hookrightarrow y$ is an incoming frond of vertex y. When depth-first search backtracks to vertex y and the frond $x \hookrightarrow y$ is being examined, the millipede whose spine is the tree-path from x to y is coalesced to a superedge e, and f is transformed to f' which is an outgoing frond of e.

Next, consider the situation in which w is not a first descendant of u but is a first descendant of v, where v is a first descendant of u.

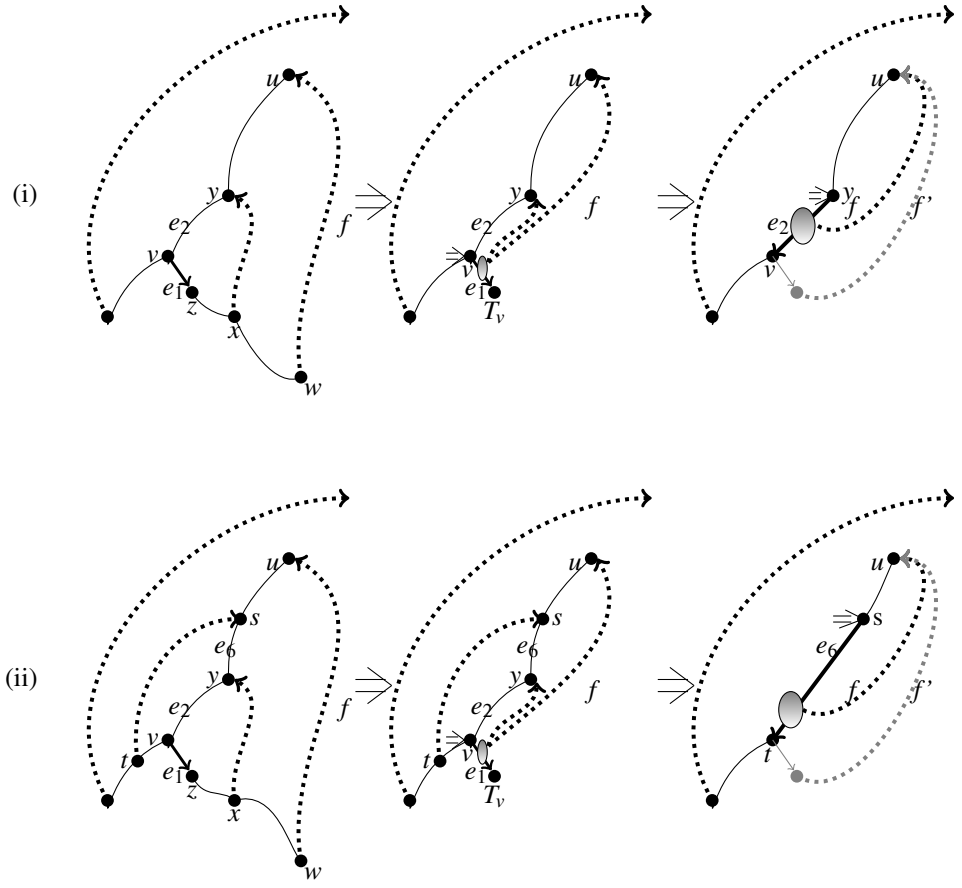


Figure 4.6: Handling incoming frond of vertex u when w is not a first descendant of u but is a first descendant of v.

In case (i), $f = (w \hookrightarrow u)$ is an incoming frond of vertex u, $f_1 = (x \hookrightarrow y)$ and z, x, w are first descendants of vertex v.

1. When depth-first search backtracks to vertex v, the millipede whose spine is the tree-path connecting v and w is coalesced into the superedge e_1 to become a tree $\hat{T}_v$ rooted at vertex v with the height of 1.

2. When depth-first search backtracks to vertex y and the frond $x \hookrightarrow y$ is examined. Since x is located in $\hat{T}_v$, $\hat{T}_v$ is coalesced into the superedge e_2 and f becomes an outgoing frond of e_2.

In case (ii), $f = (w \hookrightarrow u)$ is an incoming frond of vertex u, $f_1 = (x \hookrightarrow y)$, $f_2 = (t \hookrightarrow s)$ and z, x, w are first descendants of vertex v.

1. When depth-first search backtracks to vertex v, as with case (i), the millipede whose spine is the tree-path connecting v and w is coalesced into the superedge e_1 to become a tree $\hat{T}_v$ rooted at vertex v with the height of 1.

2. When depth-first search backtracks to vertex y and the frond $x \hookrightarrow y$ is examined, as with case (i), $\hat{T}_v$ is coalesced into the superedge e_2 and f becomes an outgoing frond of e_2.

3. When depth-first search backtracks to vertex s and the incoming frond $t \hookrightarrow s$ is examined, the millipede whose spine is the tree-path connecting s and t is coalesced into the superedge e_6 and f is transformed into f' which is an outgoing frond of e_6.

It remains to show how to handle incoming fronds $w \hookrightarrow u$ of vertex u where w is not a first descendant of vertex u and is also not a first descendant of vertex v.

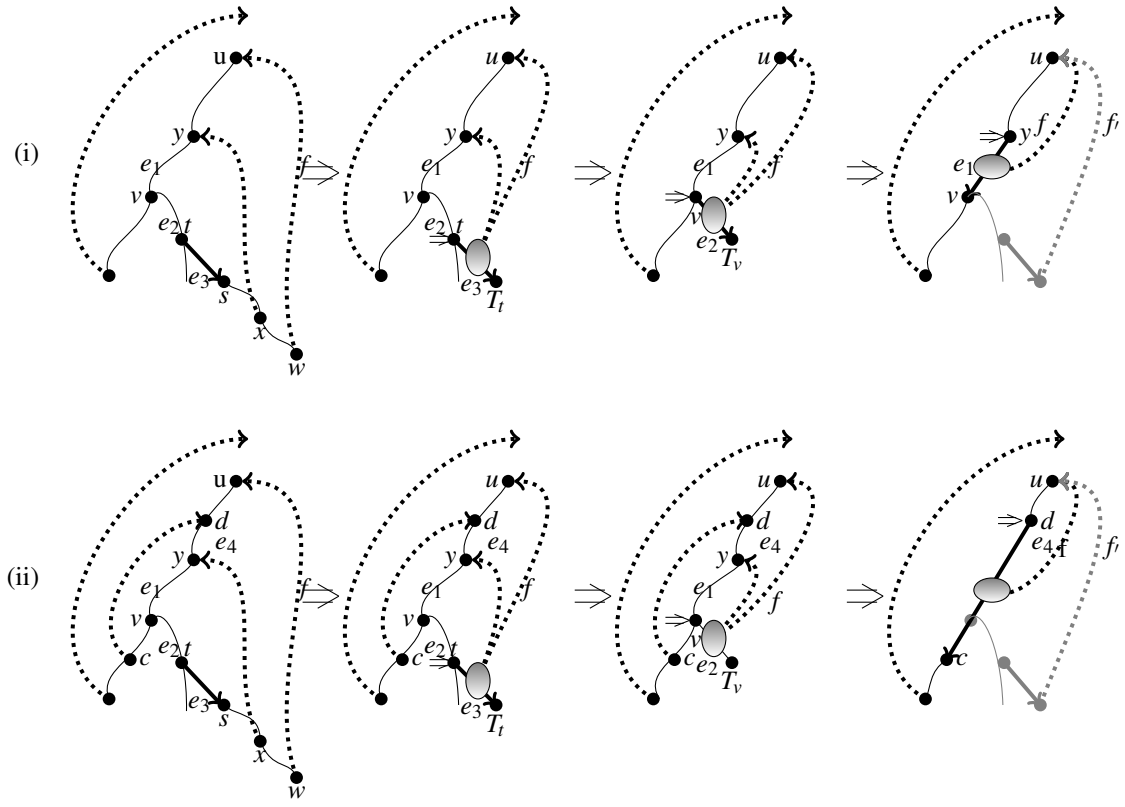


Figure 4.7: Handling incoming frond of vertex u when w is not a first descendant of u and is not a first descendant of v.

In case (i), frond $f = (w \hookrightarrow u)$ is an incoming frond of vertex u, frond $f_1 = (x \hookrightarrow y)$ and w is not a first descendant of u and is also not a first descendant of v.

1. When depth-first search backtracks to vertex t, the millipede whose spine is the tree-path connecting t and w is coalesced into the superedge $e_3 = (t \rightarrow s)$ to become a tree $\hat{T}_t$ of height 1 and rooted at vertex t.

2. When depth-first search backtracks to vertex v, the millipede whose spine is the tree-path connecting v and s is coalesced into the superedge e_2 to become a tree $\hat{T}_v$ of height 1 and rooted at vertex v.

3. When depth-first search backtracks to vertex y and the incoming frond $x \hookrightarrow y$ is examined, since vertex x is located in $\hat{T}_v$ which is a leg of the millipede whose spine is the tree-path connecting vertices y and v, the millipede and $\hat{T}_v$ are coalesced into the superedge e_1 and f becomes an outgoing frond f' of e_1.

In case (ii), frond $f = (w \hookrightarrow u)$ is an incoming frond of vertex u, fronds $f_1 = (x \hookrightarrow y)$, $f_2 = (c \hookrightarrow d)$, and w is not a first descendant of u and is also not a first descendant of v.

1. When depth-first search backtracks to vertex t, as with case (i), the millipede whose spine is the tree-path connecting t and w is coalesced into the superedge $e_3 = (t \rightarrow s)$ to become a tree $\hat{T}_t$ of height 1 and rooted at vertex t.

2. When depth-first search backtracks to vertex v, again as with case (i), the millipede whose spine is the tree-path connecting v and s is coalesced into the superedge e_2 to become a tree $\hat{T}_v$ of height 1 and rooted at vertex v.

3. When depth-first search backtracks to vertex y and the incoming frond $x \hookrightarrow y$ is examined, as with case (i), since vertex x is located in $\hat{T}_v$ which is a leg of the millipede whose spine is the tree-path connecting vertices y and v, the millipede and $\hat{T}_v$ are coalesced into the superedge e_1 and f becomes an outgoing frond f' of e_1.

4. When depth-first search backtracks to vertex d and the incoming frond $c \hookrightarrow d$ is examined, since d is located in e_1, the millipede whose spine is the tree-path connecting vertices d and c is coalesced into the superedge e_4 whereby transforming the frond f to f' which is an outgoing frond of e_4.

4.6.1 How to compute an initial value for *f'*

We observed that when we examined a frond $f = (w \hookrightarrow u)$ at its head u, the frond could have been transformed to a new frond $f' = (w' \hookrightarrow u)$. In order to determine f', we just have to determine w' and to determine w', we have to give it an initial value. This value is determined as follows:

- If $\nexists\ x$ such that x is the first vertex on the tree-path connecting u and w such that $p(x) \neq u$ and x is not the first child of $p(x)$, then $w' = w$.
- If $\exists x$ such that x is the first vertex on the tree-path connecting u and w such that $p(x) \neq u$ and x is not the first child of $p(x)$, then $w' = x$.

The initial value of w' can be determined efficiently as follow:

A number, $path(w)$, is assigned to every vertex w during the depth-first search. If w is the root, then $path(w) = 1$.

Suppose w is not the root. If w is the first child of its parent v, then $path(w) = path(v)$; otherwise, $path(w) = path(v) + 1$. Specifically,

$$path(w) = \begin{cases} path(v) & \textit{if } w \textit{ is the first child of } v; \\ 1 + path(v) & \textit{if } w \textit{ is not the first child of } v. \end{cases}$$

A stack $fork[1..|V|]$ is maintained such that $fork[j]$ records the first vertex u on the current path of the depth-first search with $path(u) = j$.

Stack $fork[j]$ is updated as follows:

- When the depth-first search advances from vertex v to vertex w, and w is not the first child of v, then w is pushed onto $fork$.
- When the depth-first search backtracks for vertex v to vertex w, and v is not the first child of w, then v is popped out of $fork$.

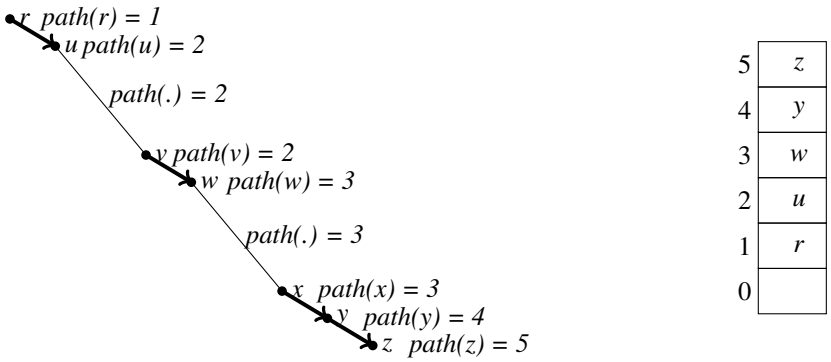


Figure 4.8: Example of $path(v)$ and $fork[v]$

The follow algorithm is for computing the initial value of *f'*:

Algorithm 1 compute f'

1: At vertex w, when frond $f\,(w \hookrightarrow u)$ is examined.
2: **if** $path(w) = path(u) \vee (path(w) = path(u)+1 \wedge u = parent(fork(top)))$ where $fork(top)$ is the vertex at the top of stack $fork$ **then**
3: $f' = f$;
4: **else**
5: **if** $u = parent\,(fork\,(path\,(u)+1))$ **then**
6: $f' = w' \hookrightarrow u$ *where* $w' = fork\,(path\,(u)+2)$;
7: **else**
8: $f' = w' \hookrightarrow u$ *where* $w' = fork\,(path\,(u)+1)$;
9: **end if**
10: **end if**

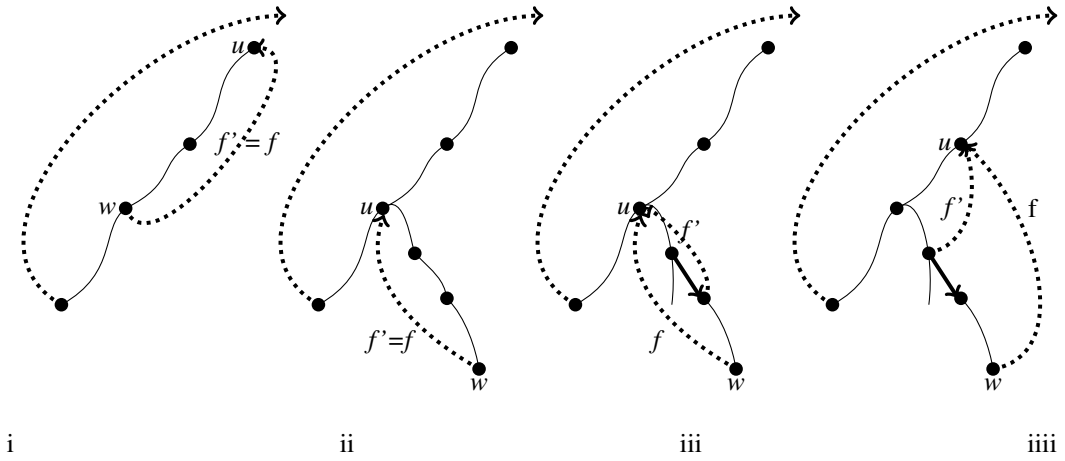


Figure 4.9: Compute f'

4.6.2 Time to compute *f'*

It takes $O(|V|)$ time to calculate $path(v), \forall v \in V$ and $O(|V|)$ time to mainipulate the stack $fork$. Since there are $|E| - |V| + 1$ fronds f and every frond f' is determined in $O(1)$ time, it takes $O(|E| - |V|)$ time to determine all of the fronds f'. The total time spent on determining the initial value of f' for all of the fronds f is thus $O(|E|)$.

4.7 Performing the coalesce operation

Whenever an incoming frond $f' = (w \hookrightarrow u)$ is retrieved from vertex u, a section of the *u-millipede* from vertex u to vertex u_i is to be coalesced into the superedge $e_1 = (u,\ u_i)$, where u_i satisfies one of the following three conditions: (i) $u_i = w$, (ii) f' is an outgoing frond of the superedge $(u_{i-1},\ u_i)$ on the millipede, and (iii) $u_i = parent(w)$.

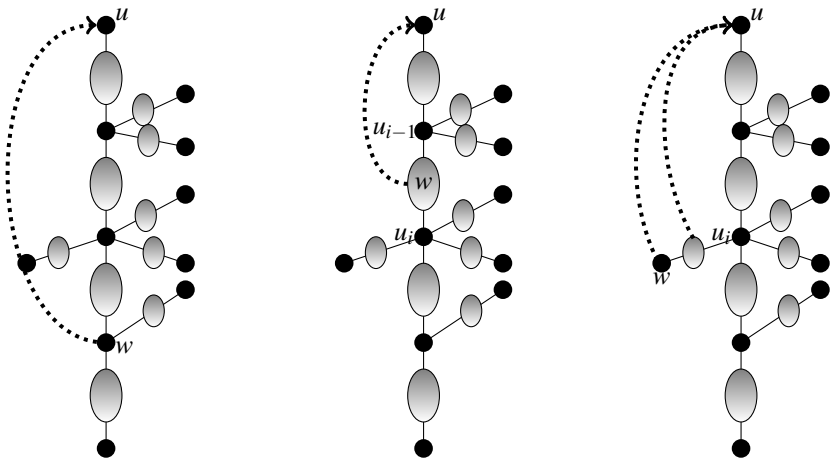


Figure 4.10: Example of different incoming fronds of vertex u

The first condition can clearly be verified in $O(1)$ time. For the second condition, u_{i-1} is an ancestor of vertex w while u_i is not an ancestor of vertex w which can be verified in $O(1)$ time by using the method for testing ancestor/descendant relationship explained earlier. For the third condition, the frond $f' = (w \hookrightarrow u)$ is an outgoing frond of edge $(parent\,(w) \rightarrow w)$ which can be determined in $O(1)$ time.

Coalesceing a section $\hat{T}_0 e_1 \hat{T}_1 \ldots \hat{T}_{h-1} e_h$ of a millipede involves coalesceing $e_i, 1 \leq i \leq h$, and the superedges in $\hat{T}_i, 1 \leq i < h$, into a superedge $e_1^{'} = (u_0,\ u_h)$ and combining $outfrond(u_i), 1 \leq i < h$, and the outgoing fronds of the superedges in $\hat{T}_i, 1 \leq i < h$, to form the set of outgoing fronds for e_1'. Since linked list is used to represent the superedges, and coalescing any two of them takes $O(1)$ time. Therefore, it takes $O\left(h + \sum_{j=1}^{h-1} \left|E_{\hat{T}_j}\right|\right)$ time to coalesce the section of millipede,

possibly including a leg in $\hat{T}_h$.

4.8 Finding separation pair

To find separation pair efficiently, we need to introduce two concepts: *lowpt3*(w) and *lowpt3*$(\tilde{e})$, where $w \in V$ and $\tilde{e}$ is a superedge of a millipede.

$$\forall w \in V,\ lowpt3(w) = min(\{dfs(w)\} \cup \{dfs(u) \mid \exists (w \hookrightarrow u)\} \cup \{lowpt1(u) \mid \exists (w \rightarrow u) \wedge (u\ is\ not\ a\ first\ descendant\ of\ w)\})$$

Specifically, *lowpt3*(w) is the vertex with the smallest dfs number that is reachable from vertex w by traversing a possibly null tree-path that avoids the first child of w following by a frond.

$\forall \tilde{e} = (v \rightarrow w)$, $lowpt3(\tilde{e}) = min(\{dfs(v)\} \cup \{dfs(y) \mid \exists (x \hookrightarrow y) \in Out(\tilde{e})\})$, where $Out(\tilde{e})$ is the set of outgoing fronds of the superedge $\tilde{e}$.

Specifically, for a superedge $\tilde{e}$, *lowpt3*$(\tilde{e})$ is the vertex with the smallest dfs number that is connected to w via an outgoing frond of $\tilde{e}$.

In Section 3.2.3, we shall call the separation pair found in Step 4 a case-1 separation pair and the separation pair found in Step 9 a case-2 separation pair. The condition for finding case-1 separation pair can be converted to:

$$\min\{lowpt3(u_0), lowpt3(\tilde{e_1})\} \geq dfs(w)$$

The condition for finding case-2 separation pair can be converted to:

$$(lowpt2(u_h) \geq dfs(w)) \wedge (p(u_0) \neq r) \vee (|C(w)| > 1)$$

The value of *lowpt1* (w), *lowpt2* (w), *lowpt3* (w) , $\forall w \in V$, can be determined in $O(|E| + |V|)$ time

during the first or second depth-first search.

The value of *lowpt3*($\tilde{e}$) ($\tilde{e}$ is a superedge) is updated when a coalesce operation is performed during the second depth-first search. This happens when an incoming frond or a child which is not the first child of the current vertex is examined. The total number of incoming fronds is $|E|-|V|+1$ and the total number of children is $|V|-1$. Since every edge can be coalesced at most once, the total number of coalesce operation performed is $O(|E|)$ and *lowpt3*($\tilde{e}$) ($\tilde{e}$ is a superedge in P_G) can thus be determined in $O(|E|)$ time.

Using *lowpt3, lowpt1, lowpt2*, each separation pair can be determined in $O(1)$ time. Since there are at most $|E|$ split components, there are at most $|E|$ checkings resulting in finding separation pair and at most $2|V|$ checkings resulting in no separation pair. The total time to find the separation pairs is thus $O(|V|+|E|)$.

4.9 Creating triple bonds

4.9.1 Determining if frond $u \hookrightarrow w$ exists

In the following figure, when the second depth-first search backtracks to vertex w, and $\{u,\ w\}$ is determined as a separation pair, if there is a frond $u \hookrightarrow w$, then a triple bond ($u,\ w$) is to be created.

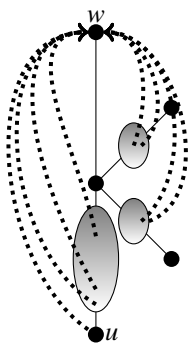


Figure 4.11: Example of the existence of frond $u \hookrightarrow w$.

To efficiently determine if the frond $u \hookrightarrow w$ exists, we maintain a linked list, *Infrondlist*(w) $w \in V$, to store all of the incoming frond of vertex w. During the depth-first search, whenever a frond $(u \hookrightarrow w)$ is encountered at the tail u, the frond is inserted into *Infrondlist*(w).

From the nature of depth-first search, the incoming fronds in *Infrondlist*(w) are stored in descending order of the depth-first search number of their tails. If a frond $(u \hookrightarrow w)$ exists, it must be in *Infrondlist*(w). So, *Infrondlist*(w) is searched. Since the total number of incoming fronds of all vertices is $\sum_{v \in V} indeg(v)$, the total time spent on this step is thus $O(\sum_{v \in V} indeg(v)) = O(|E| - |V|)$.

4.9.2 Determining if frond $w \hookrightarrow lowpt1(u)$ exists

In the following figure, when the second depth-first search backtracks to vertex w, and $\{w, lowpt1(u)\}$ is determined to be a separation pair, if there exists a frond $w \hookrightarrow lowpt1(u)$, then a triple bond $(w, lowpt1(u))$ is to be created.

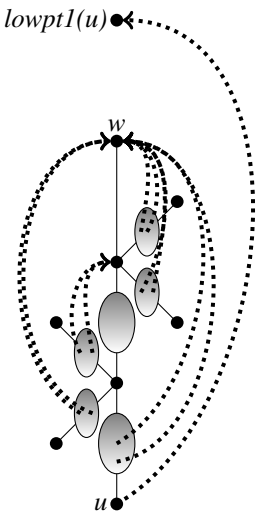


Figure 4.12: Example of the existence of frond $u \hookrightarrow lowpt1(u)$.

To efficiently determine if the frond $w \hookrightarrow lowpt1(u)$ exists, we maintain a stack $fstk[v]$, $v \in V$,

The stack is updated as follow: during the second depth-first search, when a frond $u \hookrightarrow v$ is encountered at vertex u, if the top entry of stack $fstk[v]$ is not u, push u onto $fstk[v]$; otherwise, a virtual edge $u \hookrightarrow v$ was created earlier, a triple bond (u, v) is then created.

On creating a virtual edge $w \hookrightarrow lowpt1(u)$, if the top entry of stack $fstk[lowpt1(u)]$ is not w, vertex w is pushed onto $fstk[lowpt1(u)]$; otherwise, a virtual edge $w \hookrightarrow lowpt1(u)$ was created earlier, then a triple bond (u, $lowpt1(u)$) is created. When the adjacency list of vertex w is completely processed, vertex w is popped out of every $fstk[v]$ for which a frond $w \hookrightarrow v$ exists.

For each vertex v, the number of incoming fronds is at most $indeg(v)$. Since there are at most $3|E| - 6$ split component (from Hopcroft & Tarjan (1973)), there are at most $O(|E|)$ virtual edges created. The total time for this step is thus at most $O(\sum_{v \in V} indeg(v)) + O(|E|) = O(|E| - |V|) + O(|E|) = O(|E|)$.

Chapter 5

Comparison

5.1 Platform

The platform we used for the experiments is as below:

- Hardware:
  - Model: Dell Precision WorkStation T7400
  - Processor: Intel(R) Xeon(R) CPU E5430 @ 2.66GHz 6144KB L2 cache
  - Memory: 3GB
- Software:
  - Operating System: Debian GNU/Linux 5.0.2
  - Programming Language: C++

5.2 Data Set

Let $G = (V, E)$ be an undirected graph. G is a *dense* graph if $|E| = O(|V|^2)$; G is a *sparse* graph if $|E| = O(|V|)$. Since $|E| = O(|V|^2)$ implies that $|E| = k|V|^2$, for some $k > 0$; $|E| = O(|V|)$ implies that $|E| = k|V|$, for some $k > 0$, and $|V|^2 > |V|$ for $|V| > 1$, therefore, dense graphs contains a lot of edges while sparse graphs contains relatively few edges. Clearly, there is a grey area in which it is hard to say if a graph is sparse or dense. This happens when $k|V| = k'$ is a small constant, then $|E| = k|V|^2$ becomes $|E| = k'|V|$ and G could be considered as a sparse graph.

For simple undirected graphs (graphs without self-loop and edges having the same end-vertices), another way of measuring whether a graph is dense or sparse is through the concept *density* (Coleman & Moré (1983)):

$$density(G) = \frac{2|E|}{|V|(|V|-1)}$$

Since for simple undirected graphs, $0 \leq |E| \leq \frac{1}{2}|V|(|V|-1)$, therefore, $0 \leq density(G) \leq 1$. When $density(G)$ is small, the graph G is a sparse graph whereas when $density(G)$ is large, the graph is a dense graph.

In our experiment, we use both dense graphs and sparse graphs to compare the execution time of the two algorithms. As was mentioned before, the graphs are randomly generated.

For dense graph, since $|E| = k|V|^2$ for some constant $k > 0$, we generate eight sets of dense graphs based on the value of $k(= \frac{|E|}{|V|^2})$:

$$0 < k \leq 0.1,$$

$$0.1 < k \leq 0.2,$$

$$0.2 < k \leq 0.3,$$

$$0.3 < k \leq 0.4,$$

$$0.4 < k \leq 0.5,$$

$0.5 < k \leq 0.6$,

$0.6 < k \leq 0.7$,

$0.7 < k \leq 0.8$.

For the sparse graph, since $|E| = k|V|$ for some constant $k > 0$, we also generate six sets of sparse graphs based on the value of $k(= \frac{|E|}{|V|})$.

$1 \leq \frac{|E|}{|V|} < 1.1$;

$1.1 \leq \frac{|E|}{|V|} < 1.3$;

$1.3 \leq \frac{|E|}{|V|} < 2$;

$2 \leq \frac{|E|}{|V|} < 5$;

$5 \leq \frac{|E|}{|V|} < 10$;

$10 \leq \frac{|E|}{|V|} < 100$.

Furthermore, as both algorithms consist of two parts: the first part generates a suitable adjacency-lists structure and the second part generates the split components, we also generated two sets of experiments to see how each these two parts influences the total execution time. One set compares the time needed to construct the acceptable adjacency-lists structure by Hopcroft et al. with the time needed to construct the simple adjacency-lists structure by Tsin's algorithm. The other set compares the time needed to generate the split components.

5.3 Results

5.3.1 Dense graph comparison

Figures 5.1 to 5.24 display the result of the experiment that compares the execution time of Hopcroft and Tarjan's algorithm with that of Tsin's algorithm. Specifically, Figures 5.1 to 5.8

compare the *total* execution time of the two algorithms. Figures 5.9 to 5.16 compare the execution time required by the two algorithms in creating their adjacency-lists structure. The remaining figures compare the execution time of two algorithms spent on generating the split components.

Figures 5.1, 5.9 and 5.17 display the result of running the two algorithms on 358 dense graphs with $0 < k \leq 0.1$ and $61 \leq |V| + |E| \leq 9,717,594$.

Figures 5.2, 5.10 and 5.18 display the result of running the two algorithms on 399 dense graphs with $0.1 < k \leq 0.2$ and $595 \leq |V| + |E| \leq 10,325,787$.

Figures 5.3, 5.11 and 5.19 display the result of running the two algorithms on 354 dense graphs with $0.2 < k \leq 0.3$ and $234 \leq |V| + |E| \leq 10,494,231$.

Figures 5.4, 5.12 and 5.20 display the result of running the two algorithms on 300 dense graphs with $0.3 < k \leq 0.4$ and $1,923 \leq |V| + |E| \leq 10,354,049$.

Figures 5.5, 5.13 and 5.21 display the result of running the two algorithms on 326 dense graphs with $0.4 < k \leq 0.5$ and $2,205 \leq |V| + |E| \leq 10,434,846$.

Figures 5.6, 5.14 and 5.22 display the result of running the two algorithms on 297 dense graphs with $0.5 < k \leq 0.6$ and $966 \leq |V| + |E| \leq 10,499,902$.

Figures 5.7, 5.15 and 5.23 display the result of running the two algorithms on 270 dense graphs with $0.6 < k \leq 0.7$ and $825 \leq |V| + |E| \leq 10,240,0357$.

Figures 5.8, 5.16 and 5.24 display the result of running the two algorithms on dense graphs with $0.7 < k \leq 0.8$. For Hopcroft and Tarjan's algorithm, 85 dense graphs were used with $34,865 \leq |V| + |E| \leq 10,789,432$; for Tsin's algorithm, 126 dense graphs were used with $34,865 \leq |V| + |E| \leq 220,22,942$.

Notice that $|V| + |E|$ is the *input size* of the graph G.

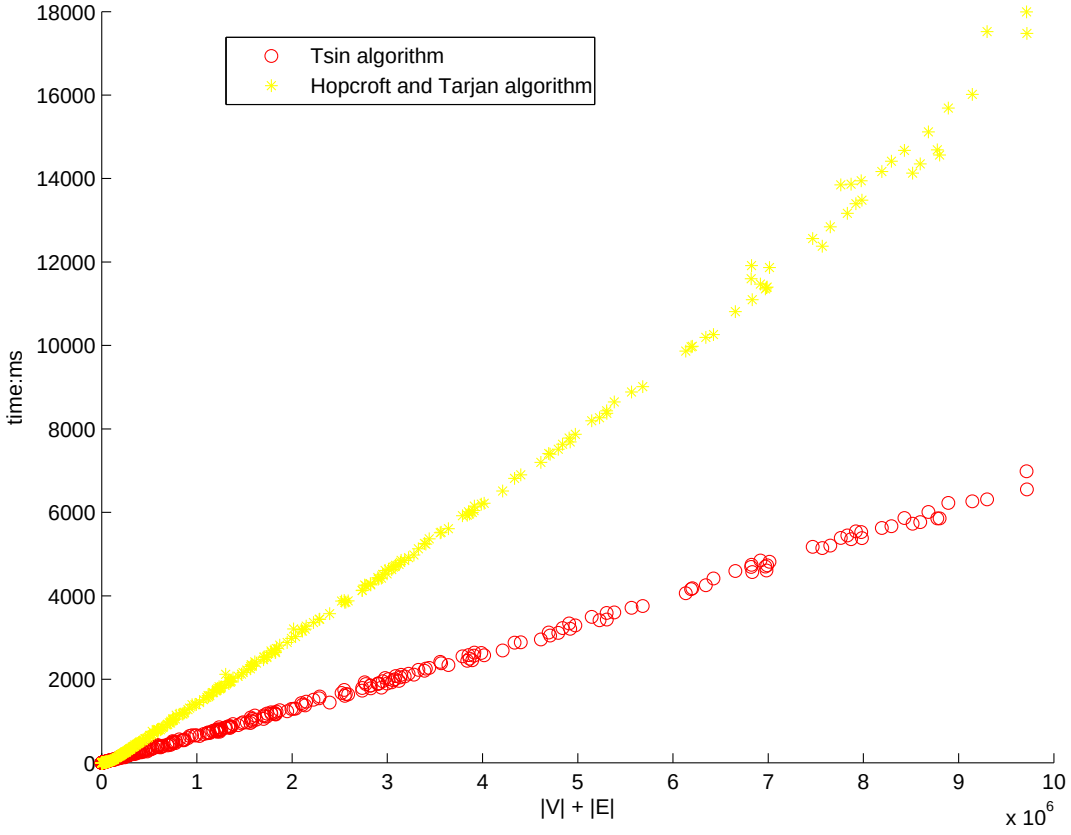


Figure 5.1: Total execution time (dense graphs with $0 < k \leq 0.1$)

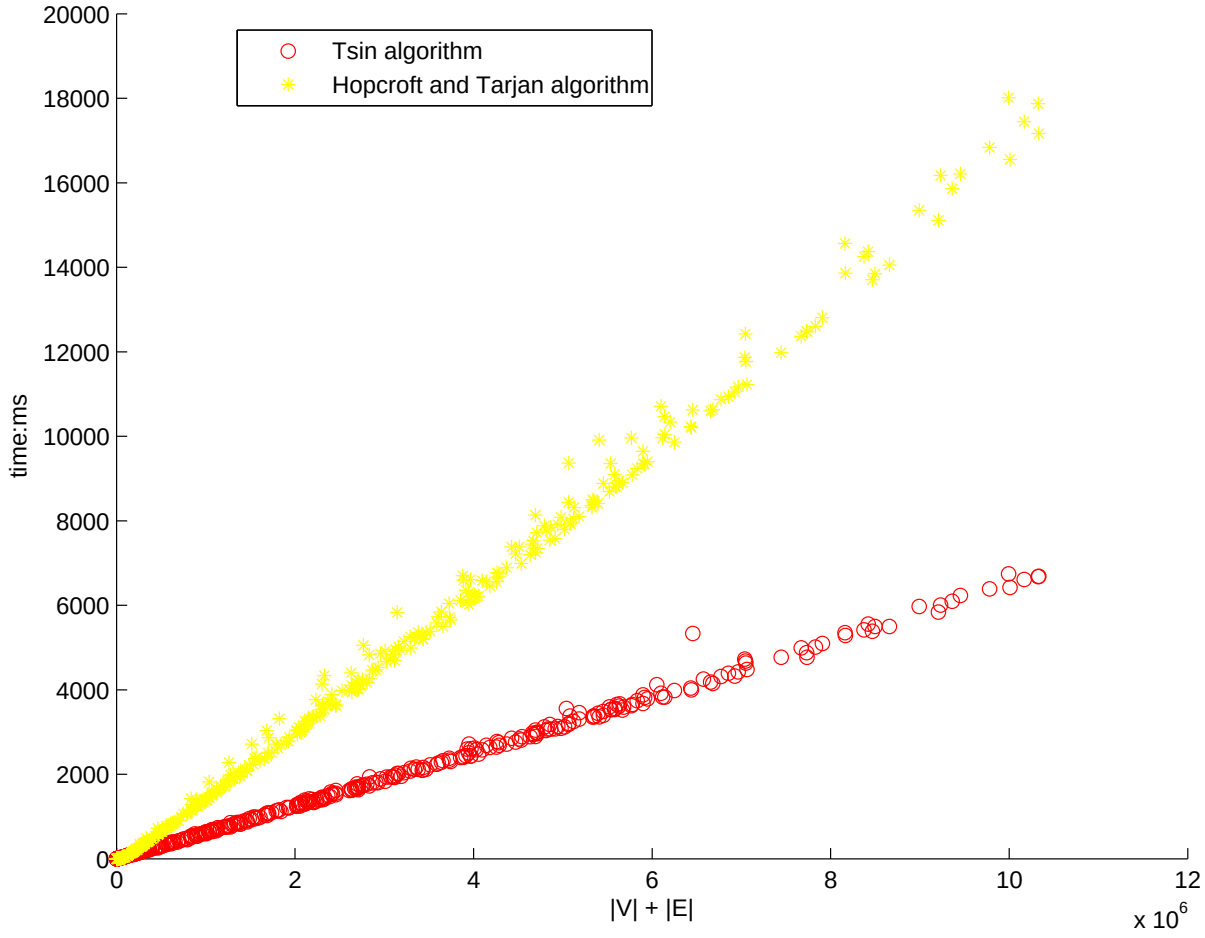


Figure 5.2: Total execution time (dense graphs with $0.1 < k \leq 0.2$)

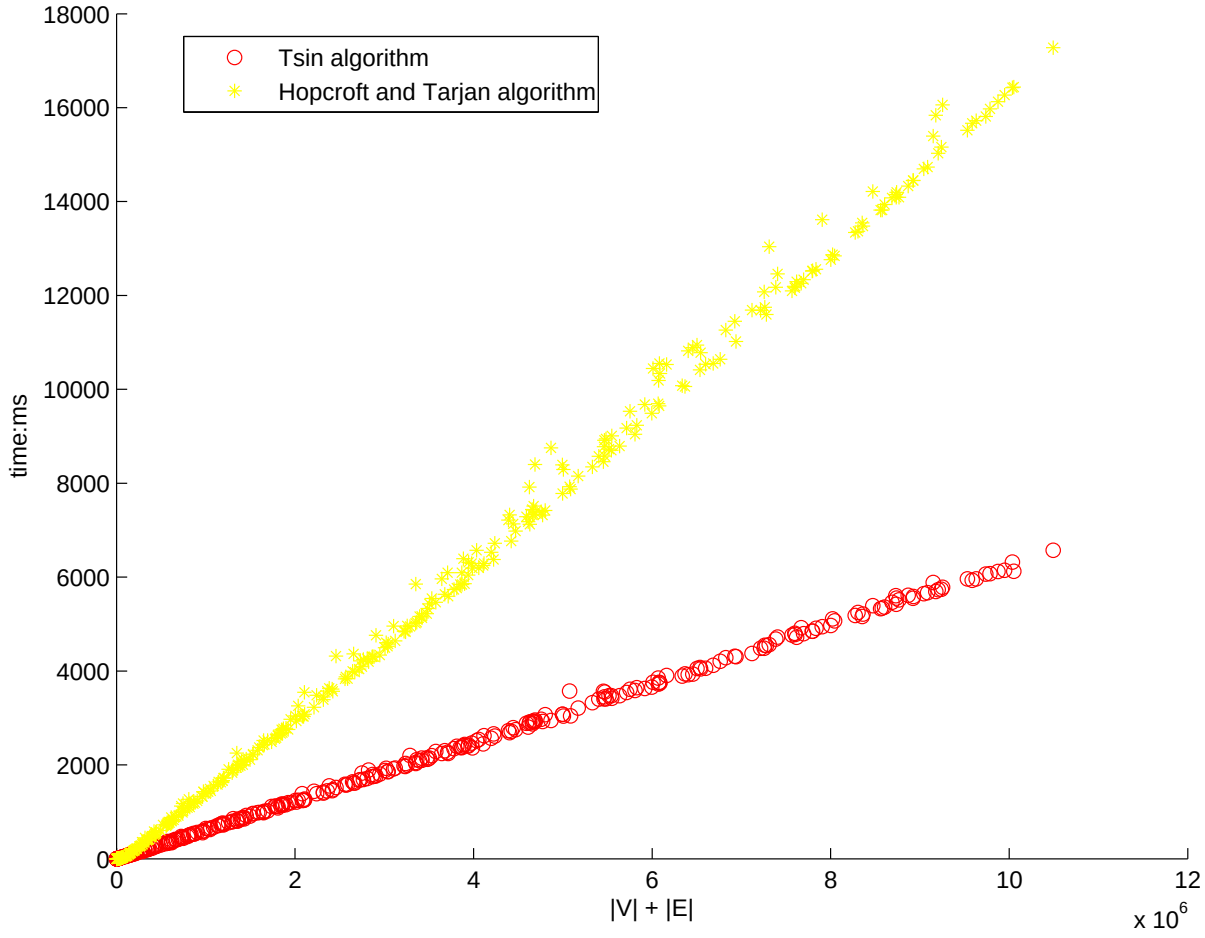


Figure 5.3: Total execution time (dense graphs with $0.2 < k \leq 0.3$)

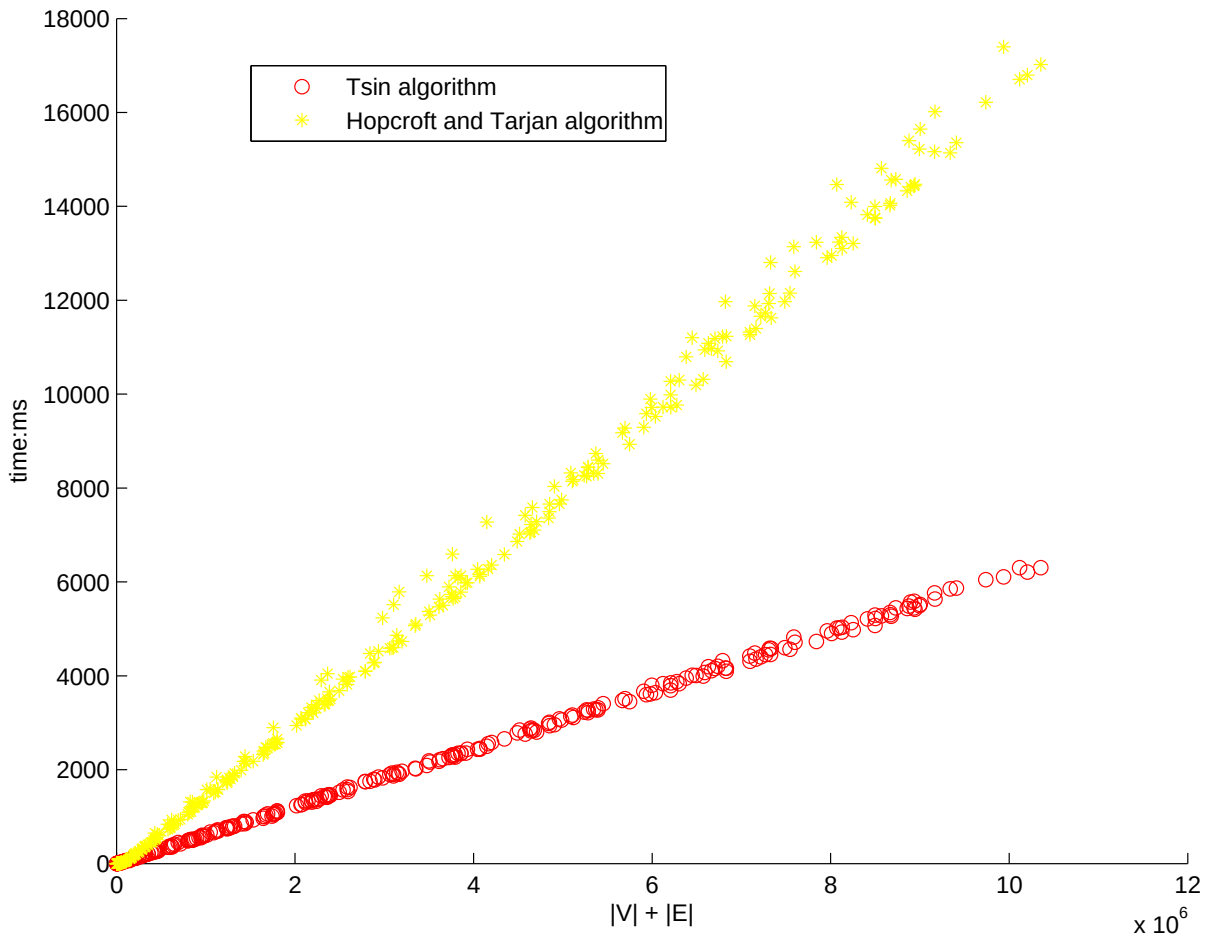


Figure 5.4: Total execution time (dense graphs with $0.3 < k \leq 0.4$)

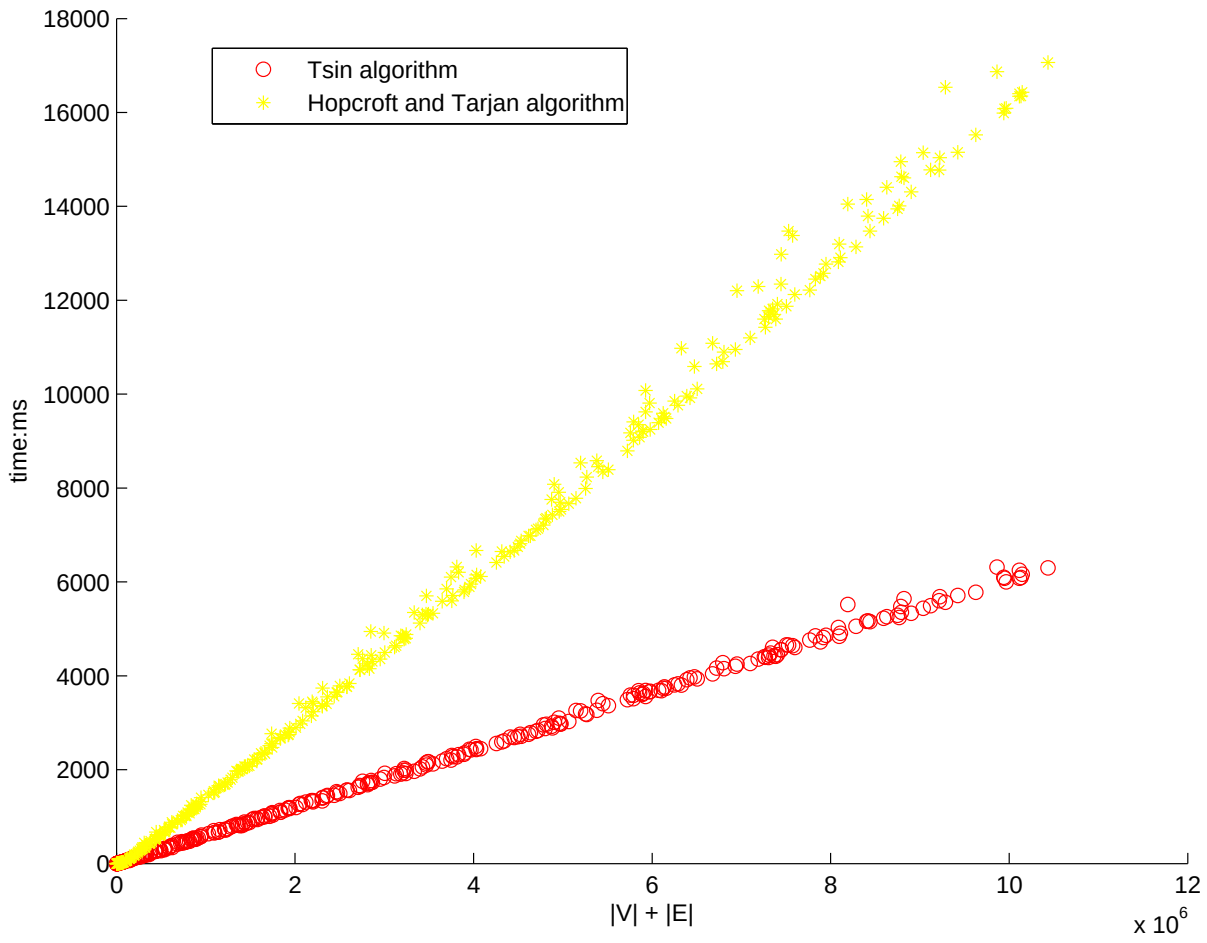


Figure 5.5: Total execution time (dense graphs with $0.4 < k \leq 0.5$)

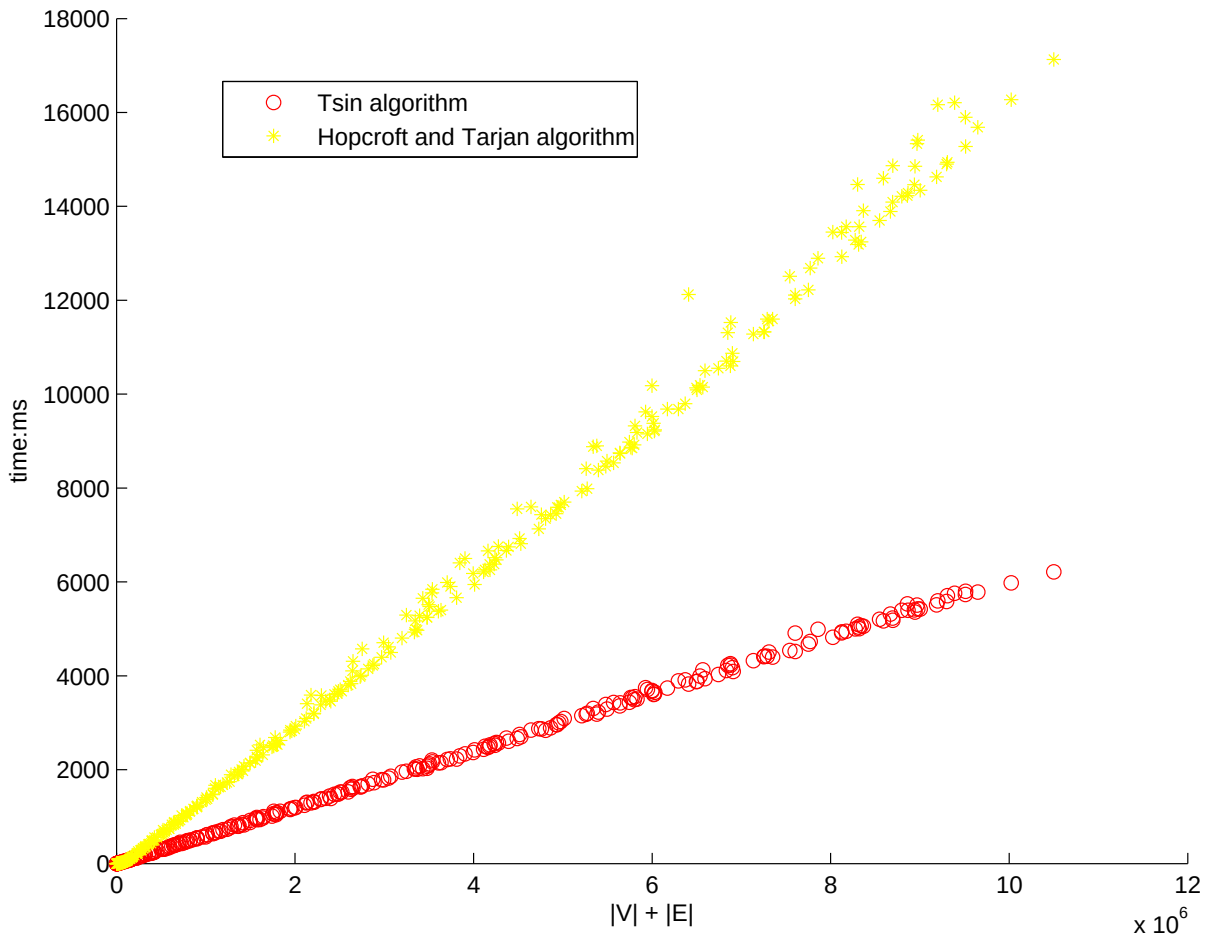


Figure 5.6: Total execution time (dense graphs with $0.5 < k \leq 0.6$)

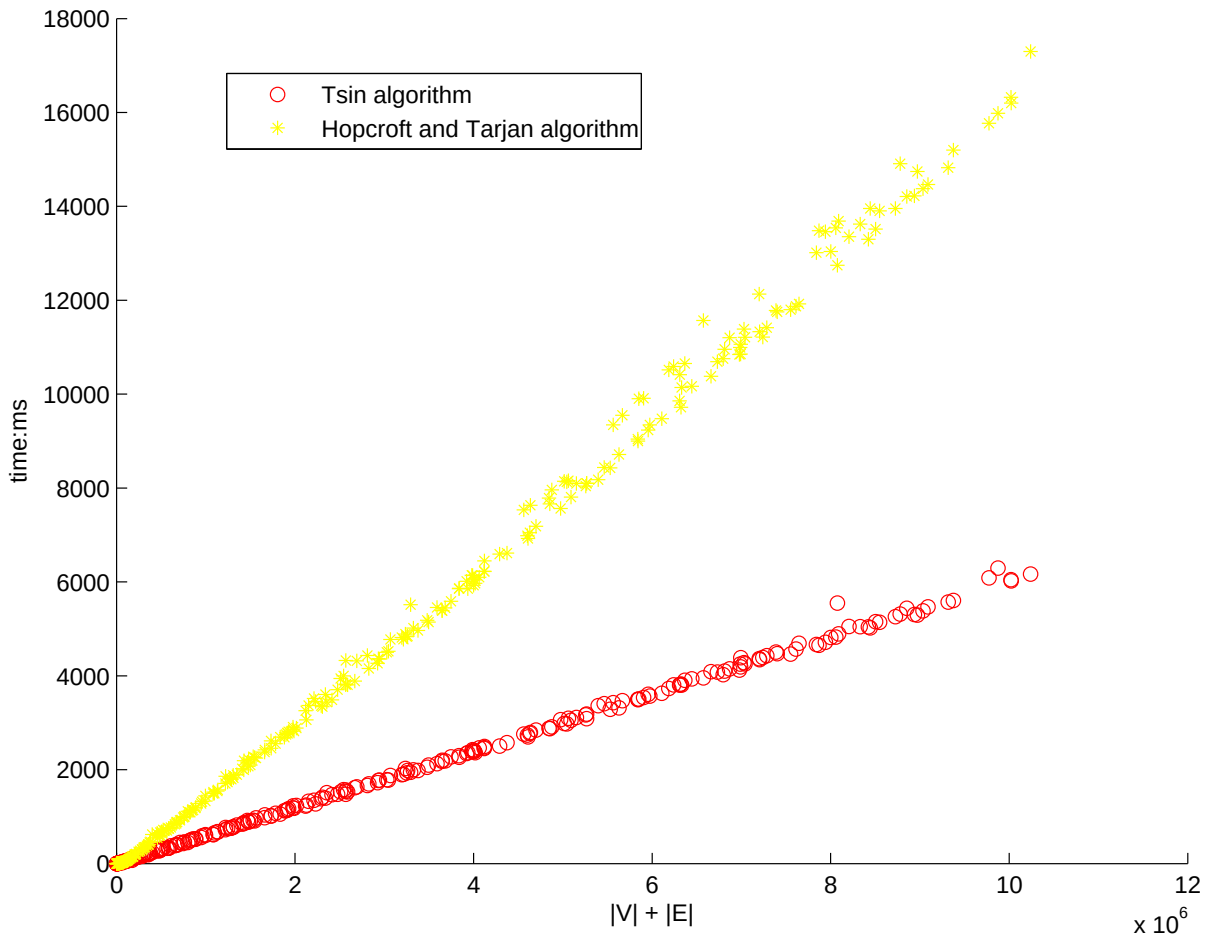


Figure 5.7: Total execution time (dense graphs with $0.6 < k \leq 0.7$)

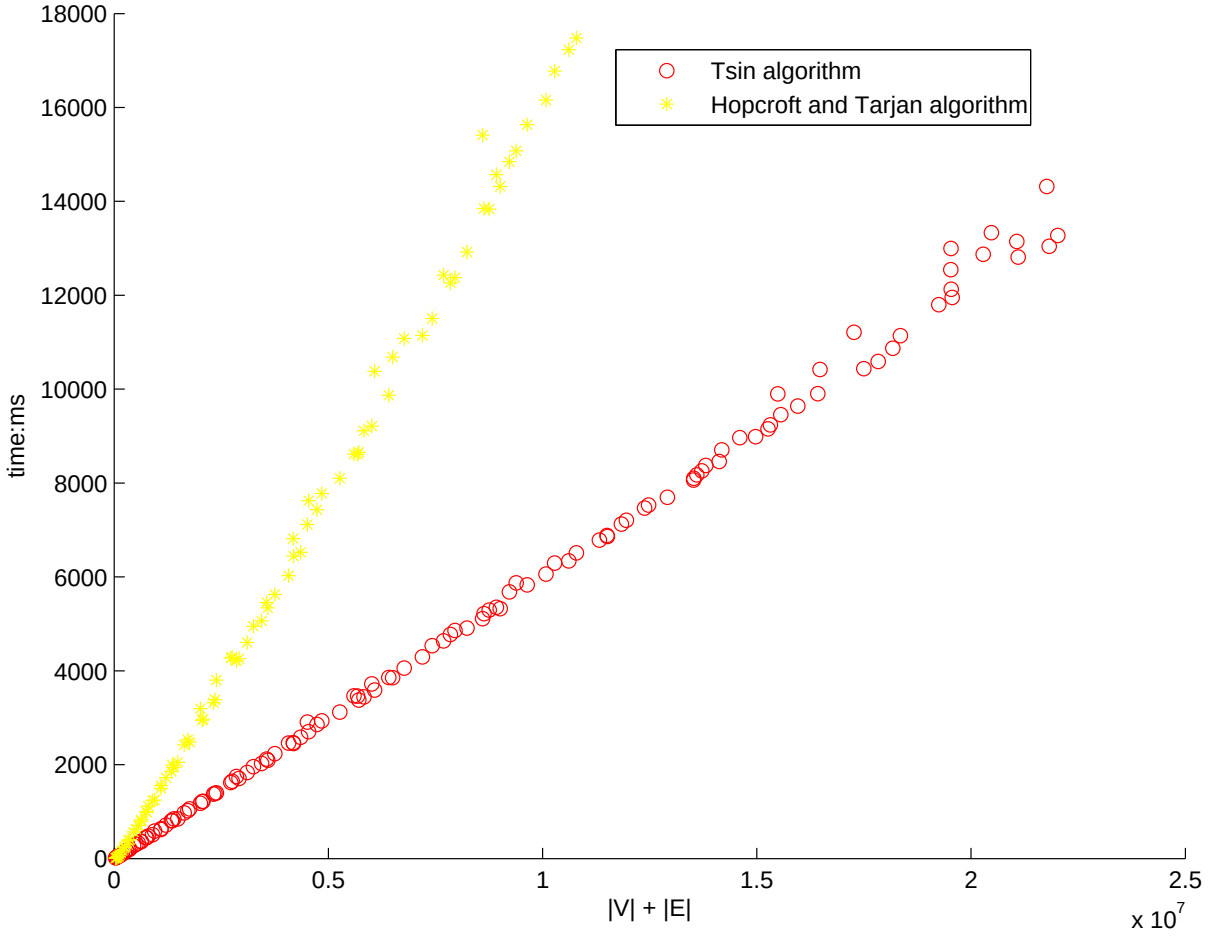


Figure 5.8: Total execution time (dense graphs with $0.7 < k \leq 0.8$)

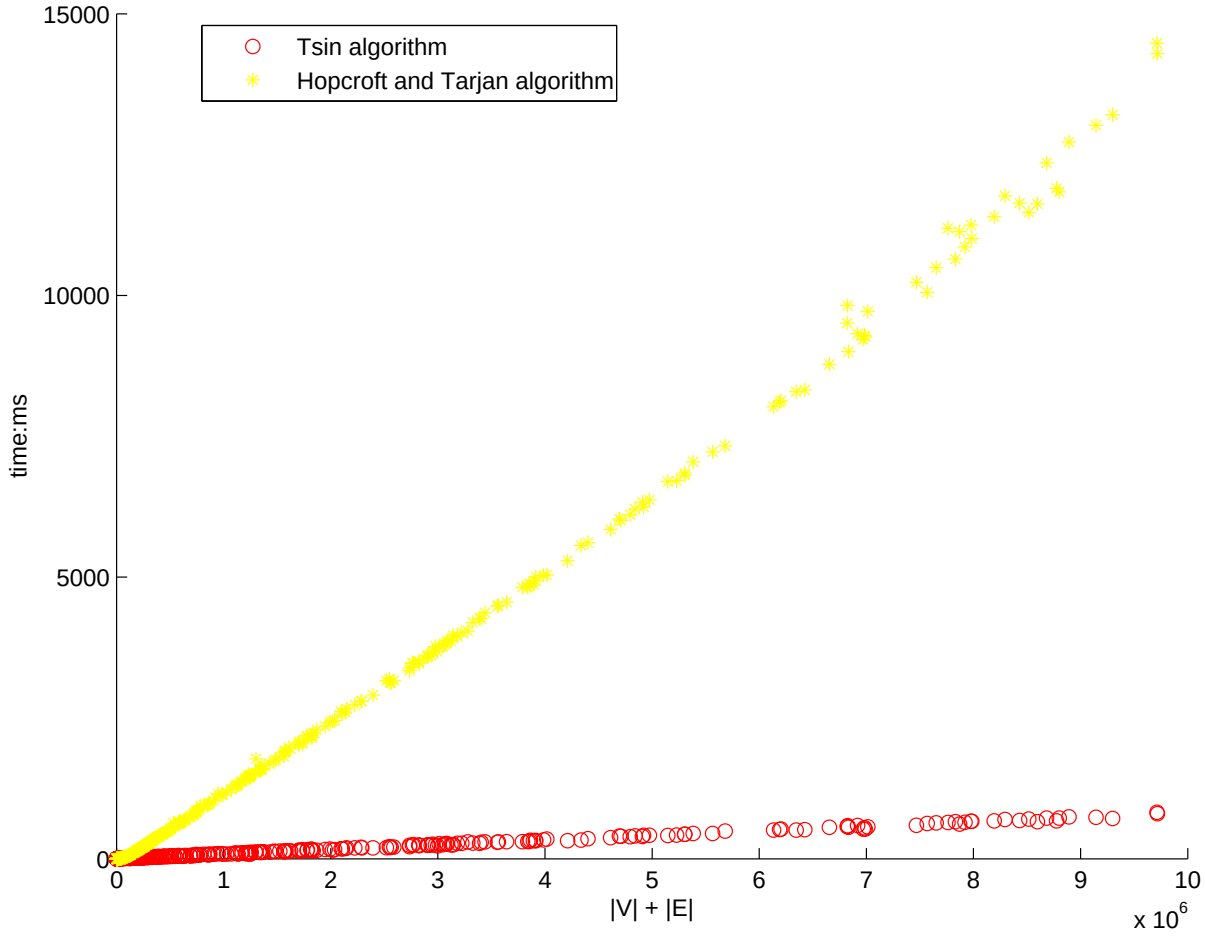


Figure 5.9: Time required to create the adjacency-lists (dense graphs with $0 < k \leq 0.1$)

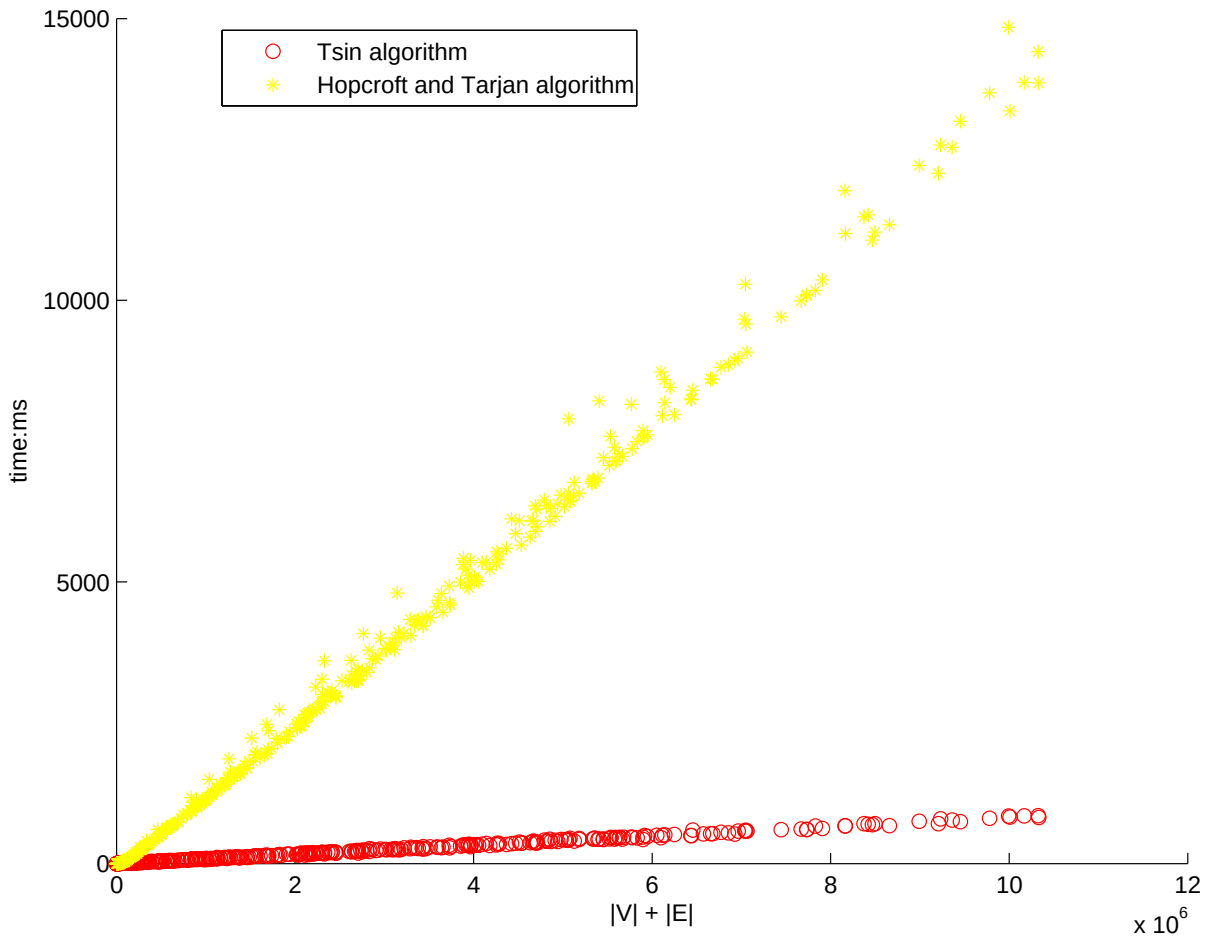


Figure 5.10: Time required to create the adjacency-lists (dense graphs with $0.1 < k \leq 0.2$)

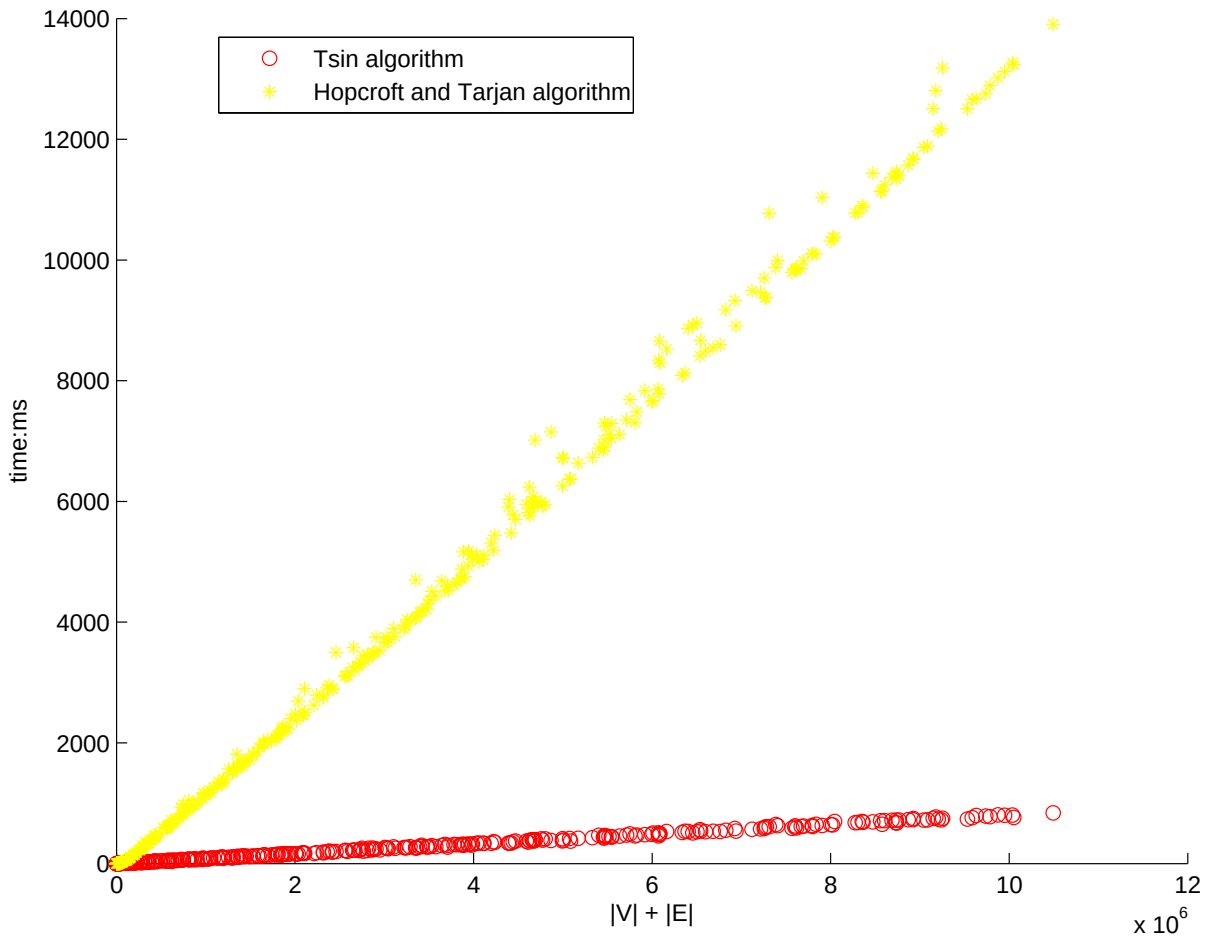


Figure 5.11: Time required to create the adjacency-lists (dense graphs with $0.2 < k \leq 0.3$)

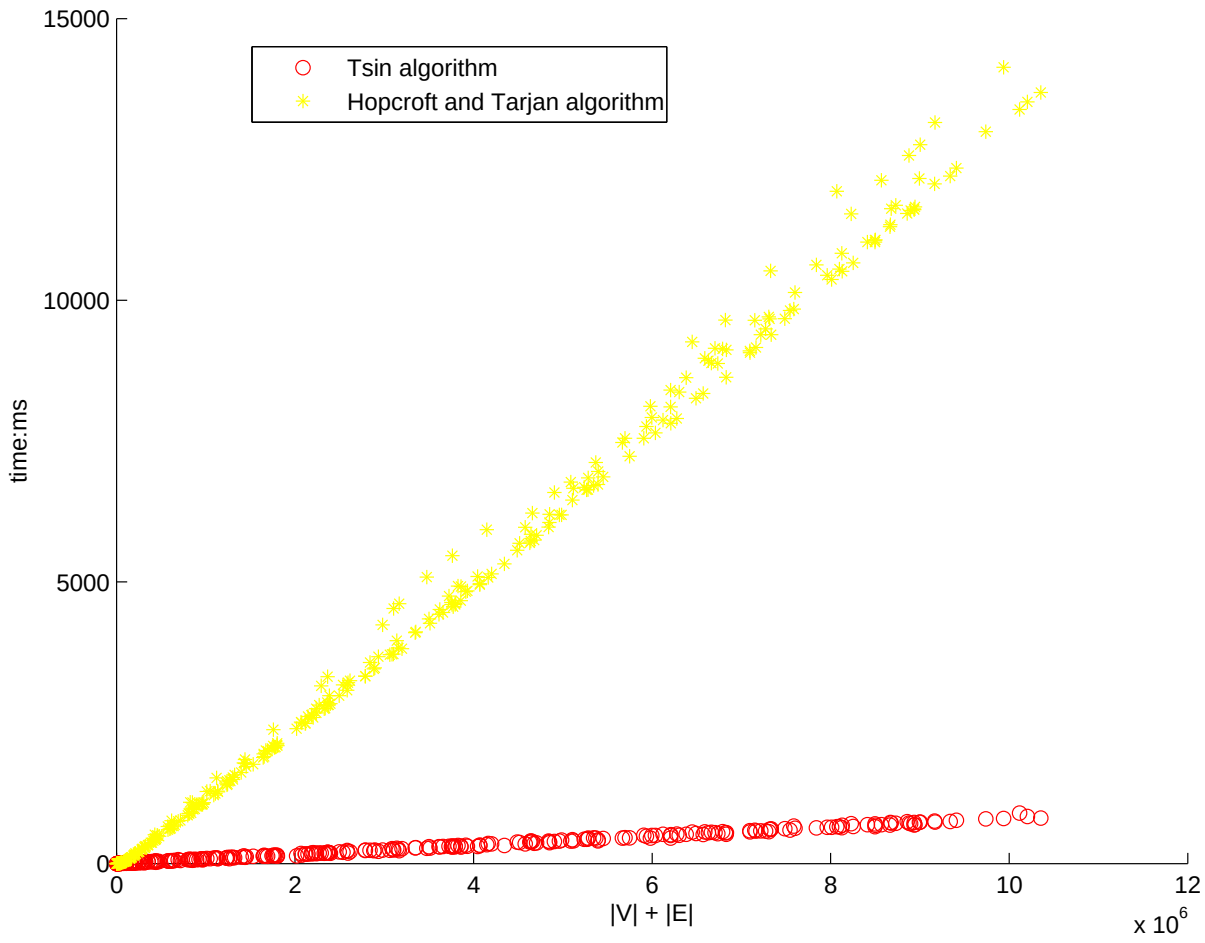


Figure 5.12: Time required to create the adjacency-lists (dense graphs with $0.3 < k \leq 0.4$)

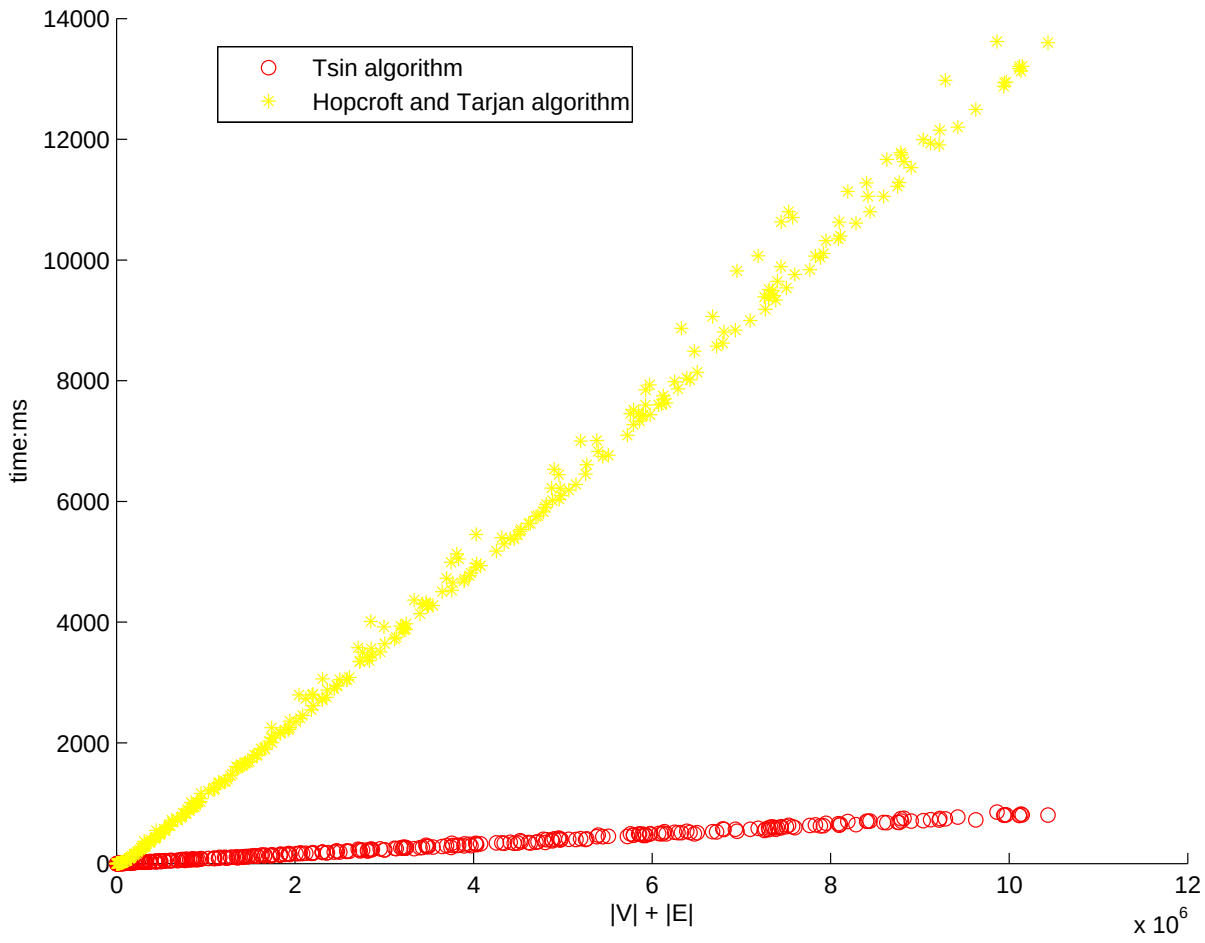


Figure 5.13: Time required to create the adjacency-lists (dense graphs with $0.4 < k \leq 0.5$)

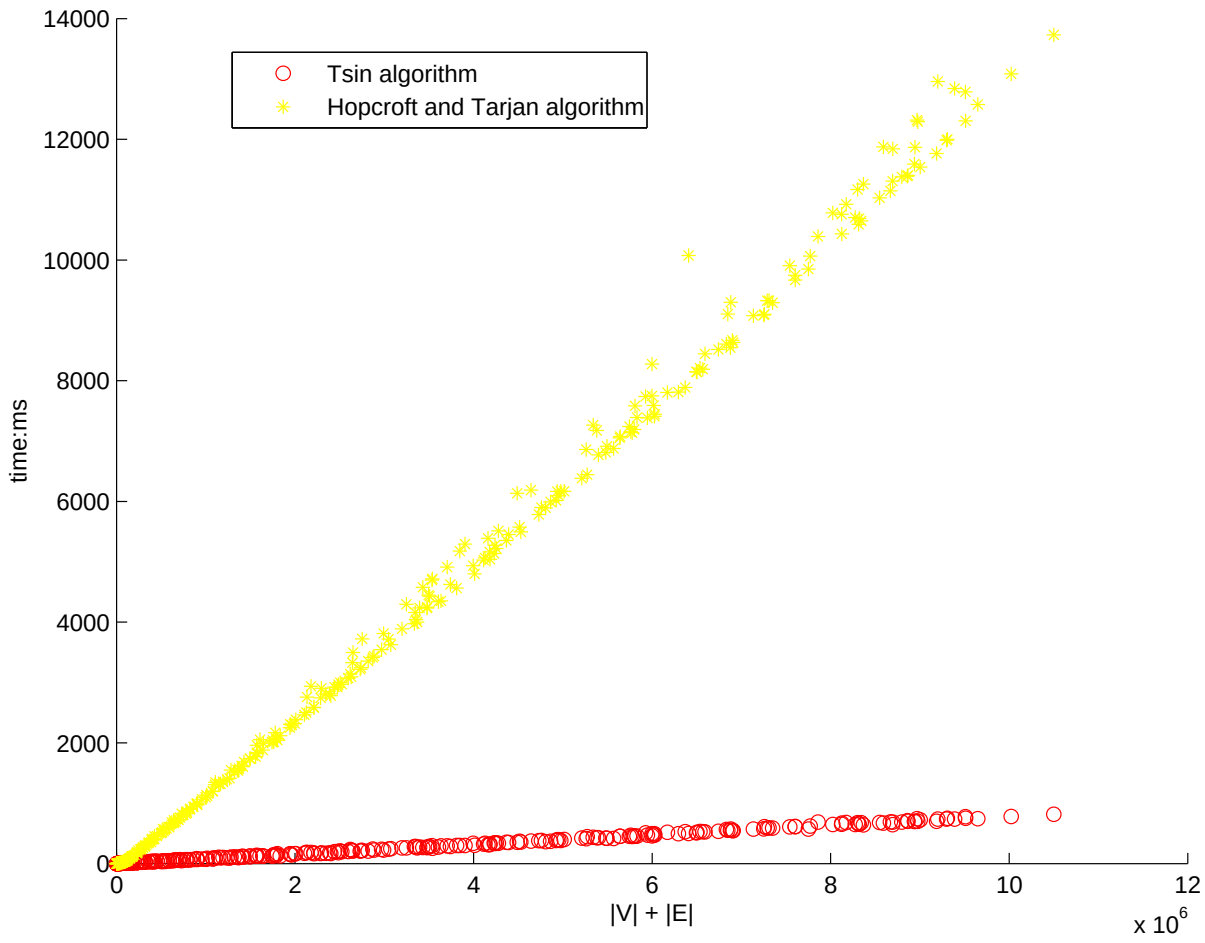


Figure 5.14: Time required to create the adjacency-lists (dense graphs with $0.5 < k \leq 0.6$)

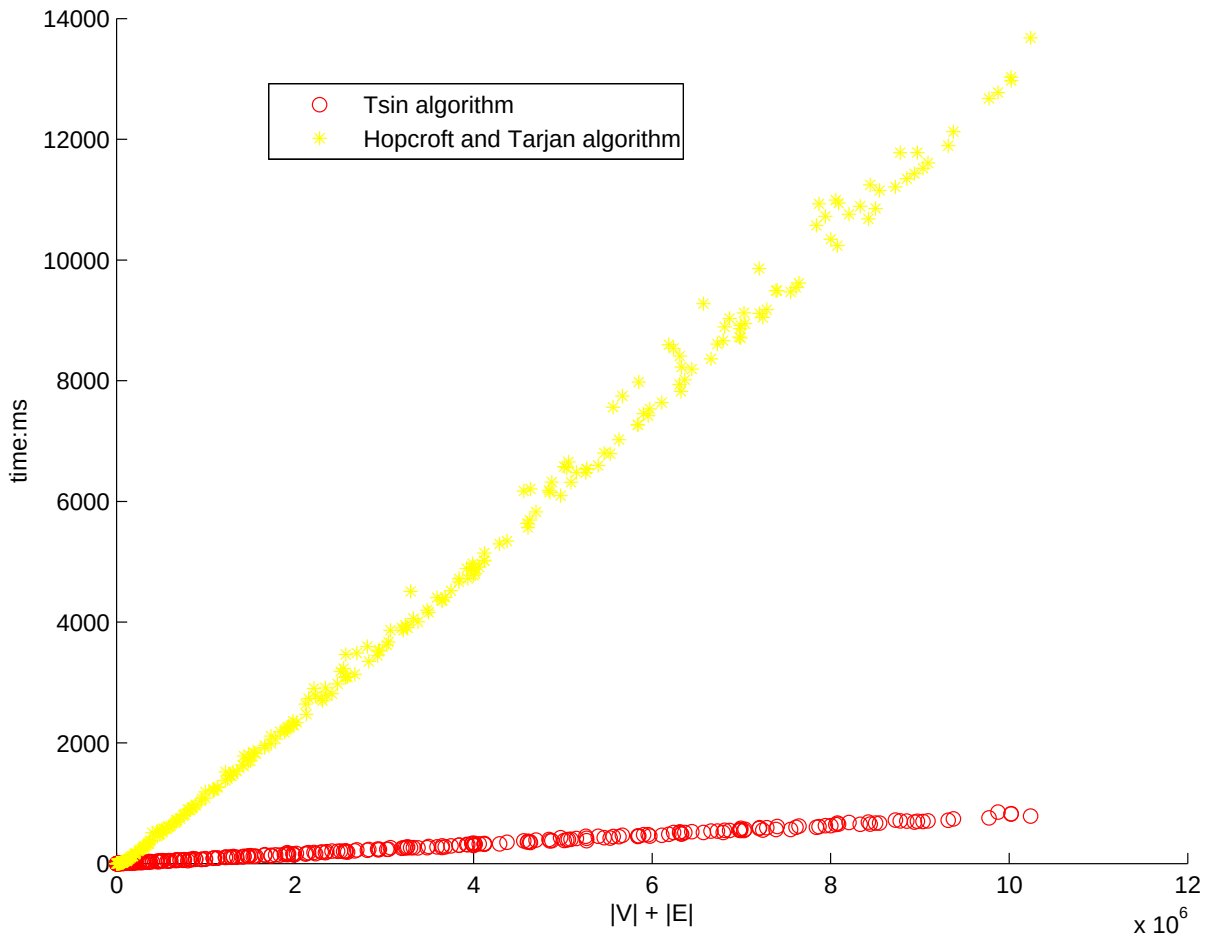


Figure 5.15: Time required to create the adjacency-lists (dense graphs with $0.6 < k \leq 0.7$)

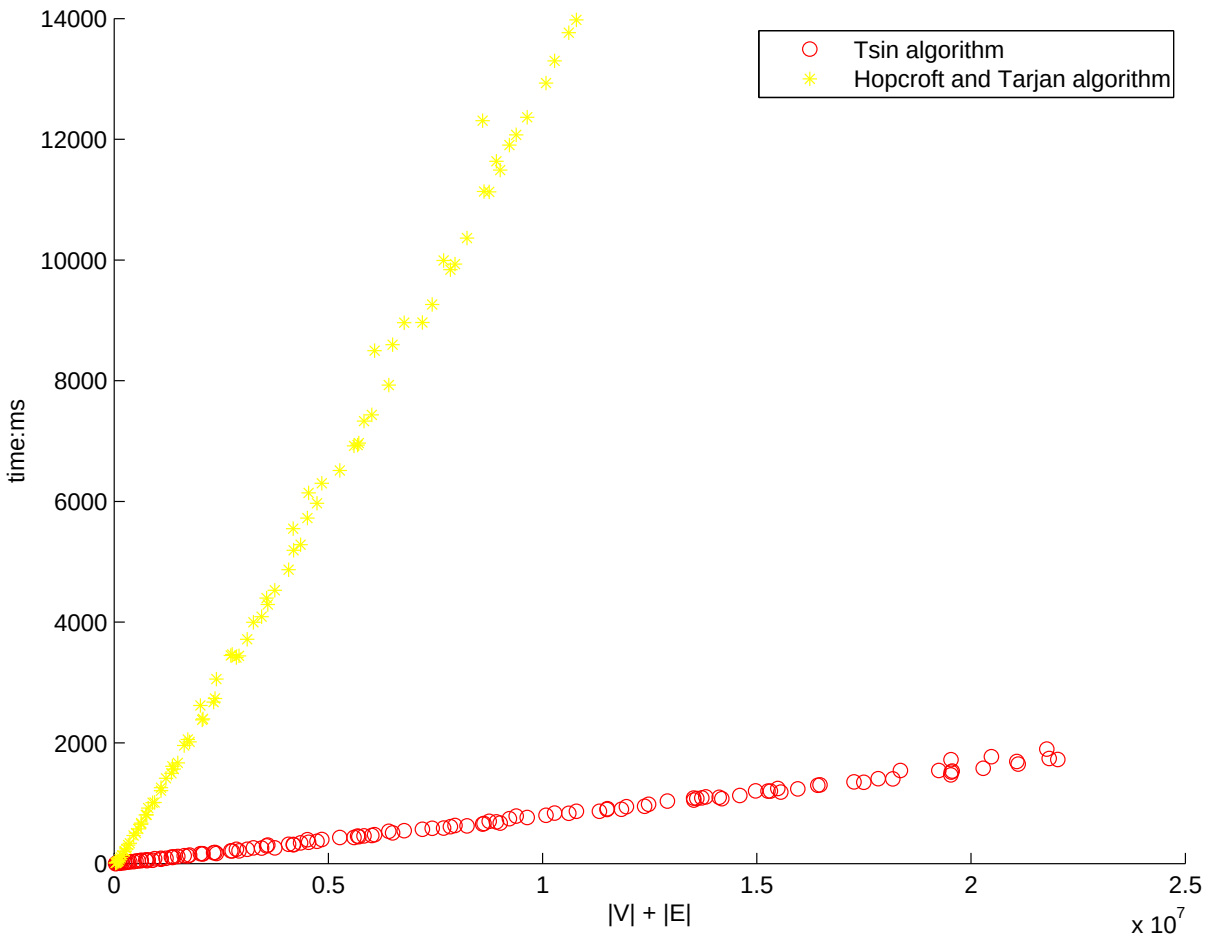


Figure 5.16: Time required to create the adjacency-lists (dense graphs with $0.7 < k \leq 0.8$)

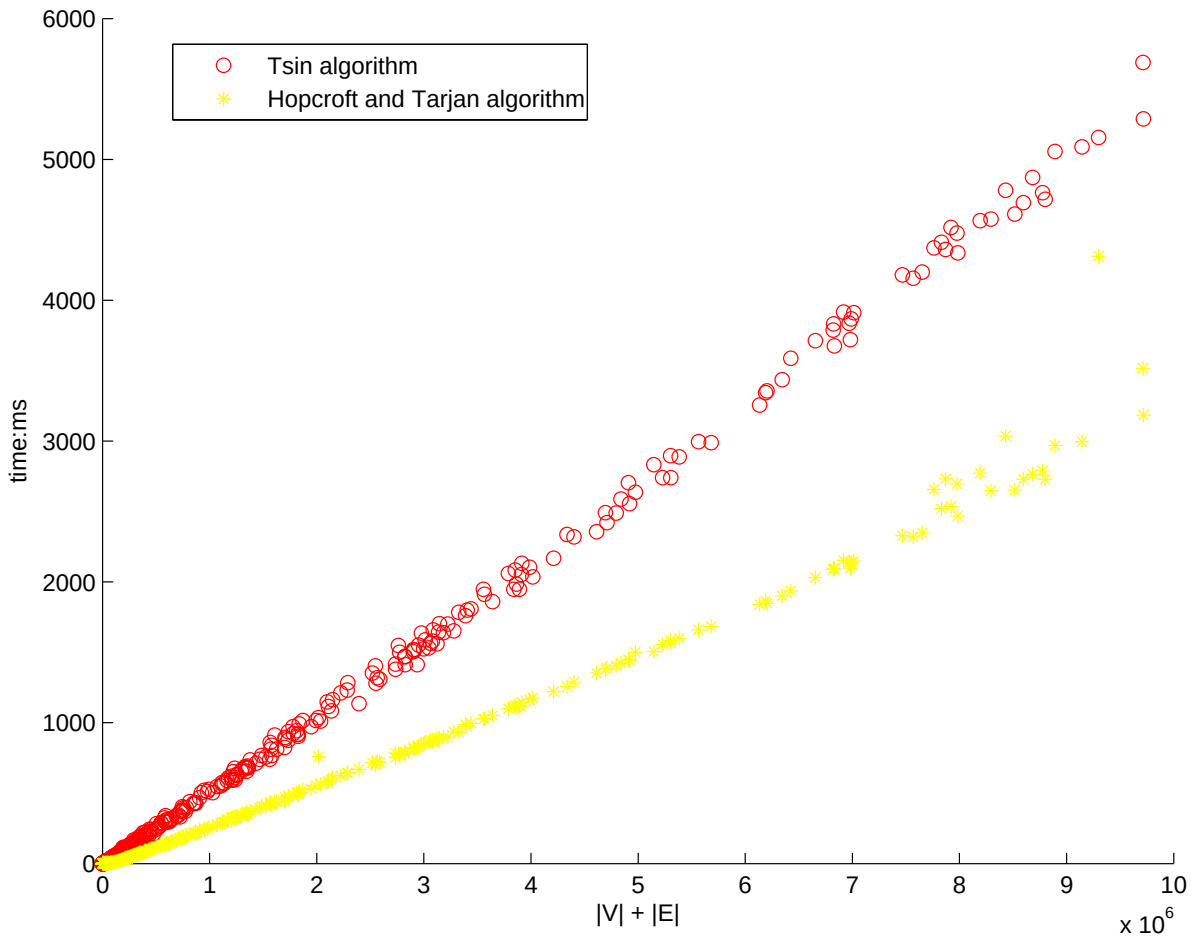


Figure 5.17: Time required to find split components (dense graphs with $0 < k \leq 0.1$)

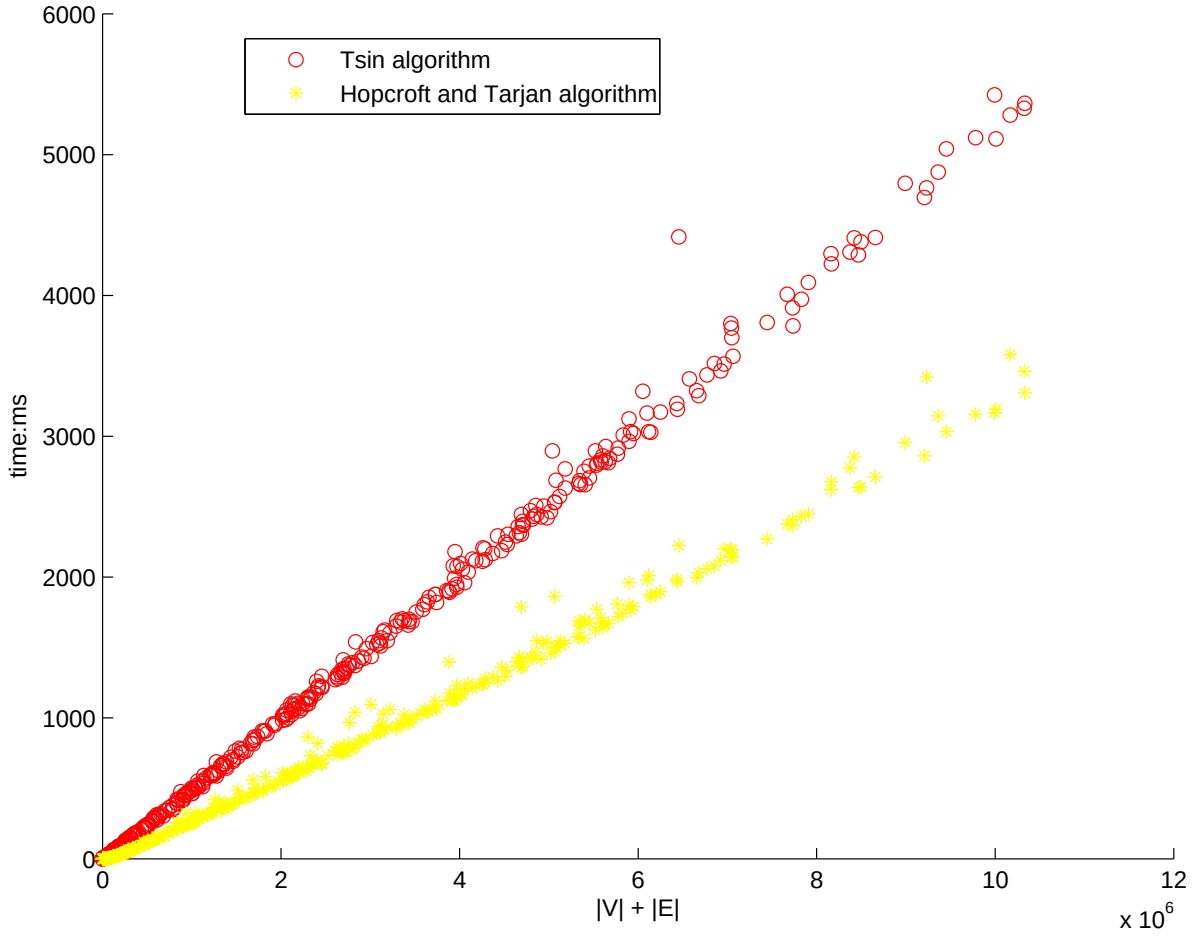


Figure 5.18: Time required to find split components (dense graphs with $0.1 < k \leq 0.2$)

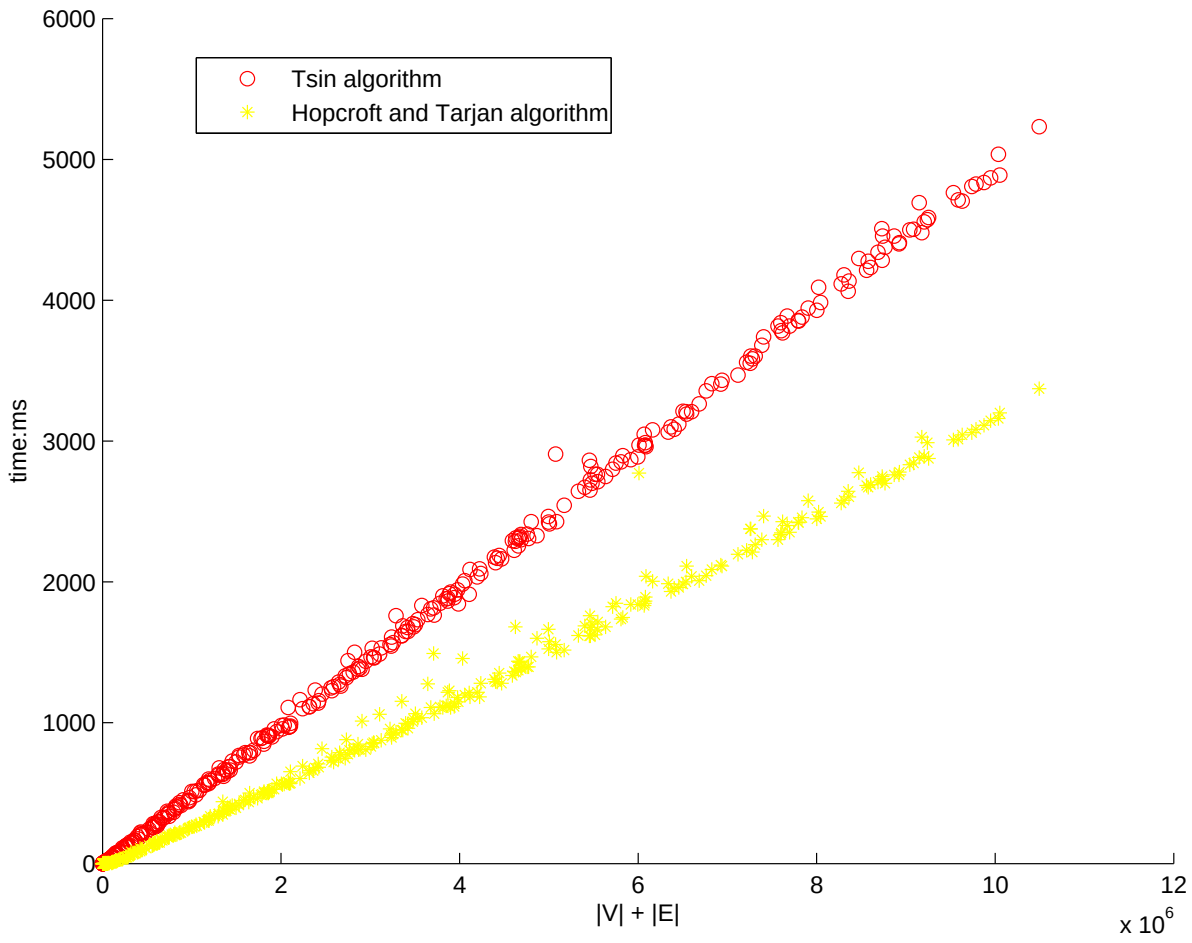


Figure 5.19: Time required to find split components (dense graphs with $0.2 < k \leq 0.3$)

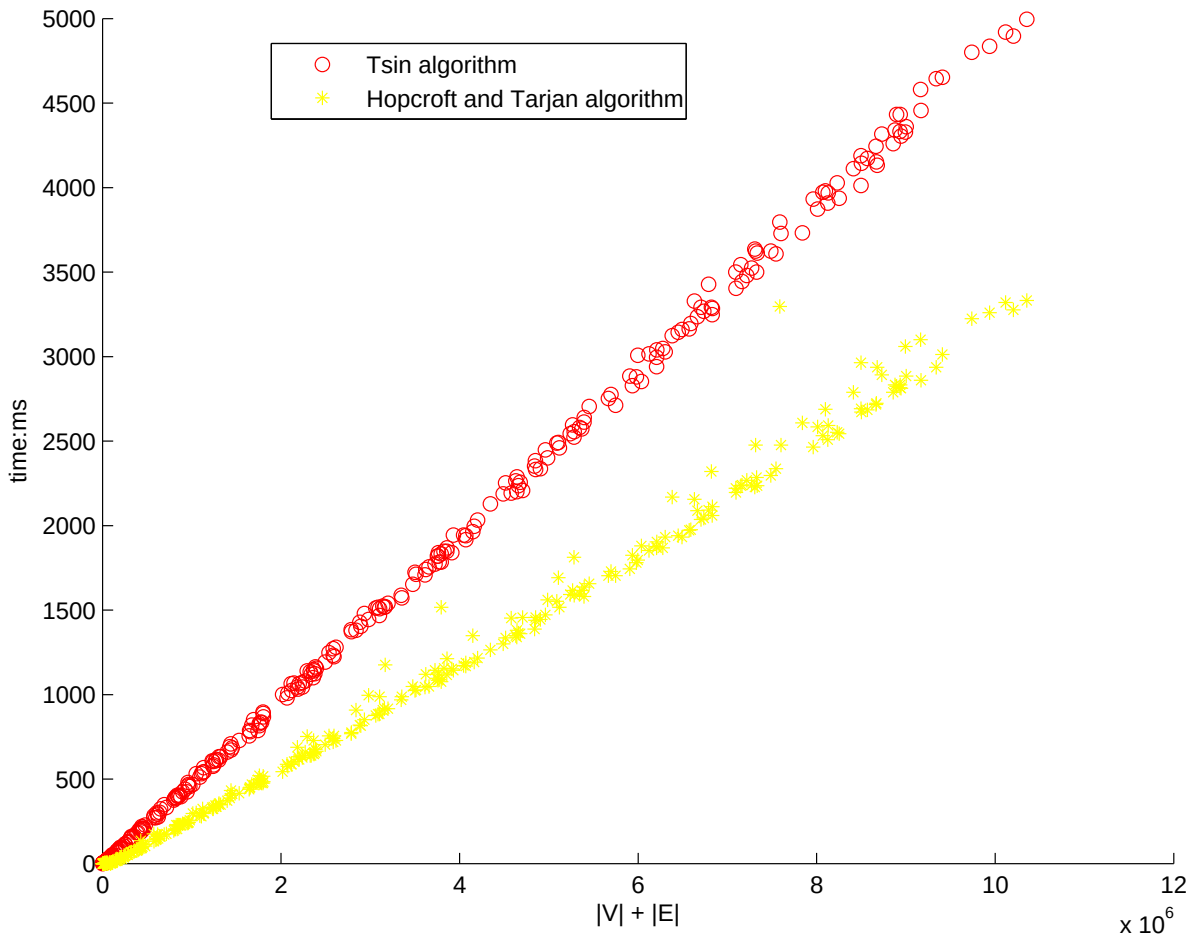


Figure 5.20: Time required to find split components (dense graphs with $0.3 < k \leq 0.4$)

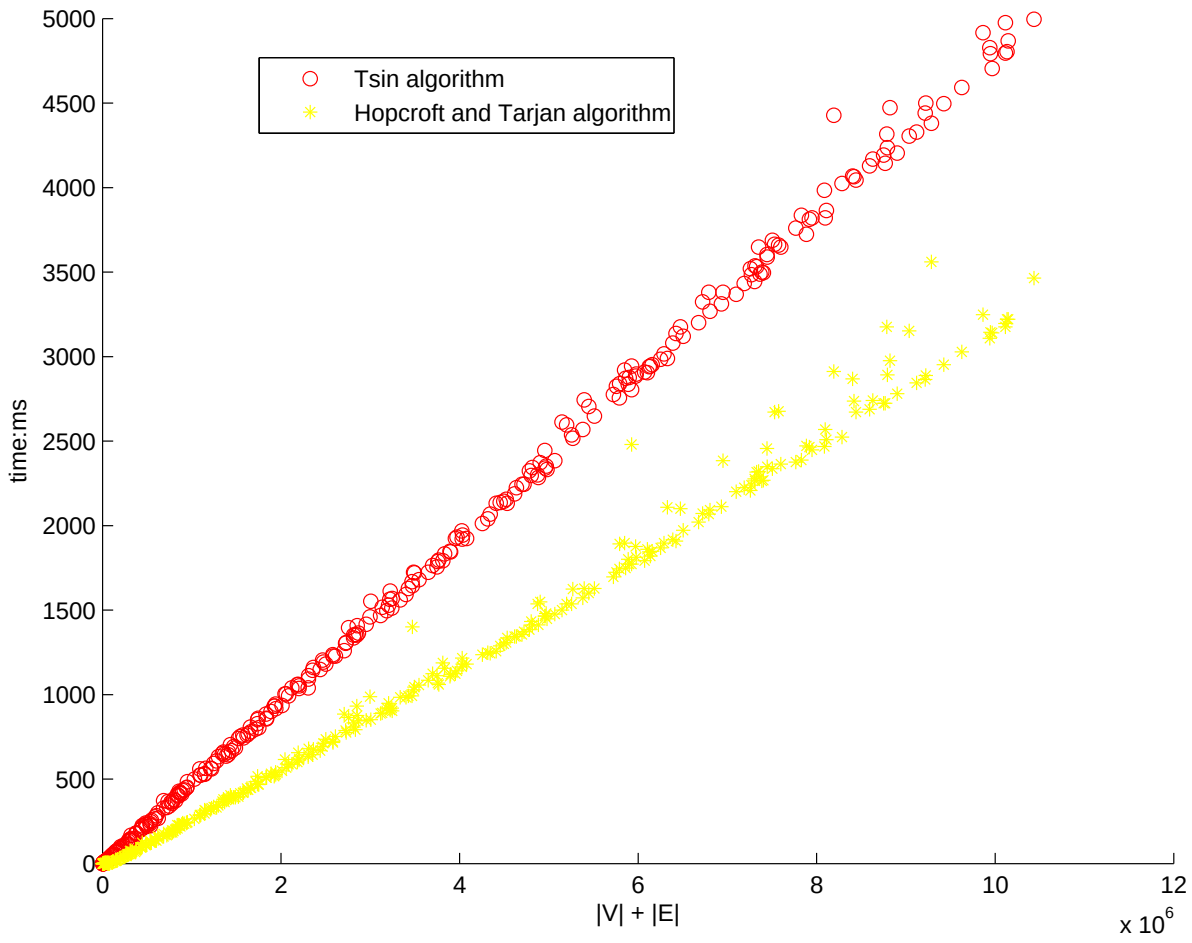


Figure 5.21: Time required to find split components (dense graphs with $0.4 < k \leq 0.5$)

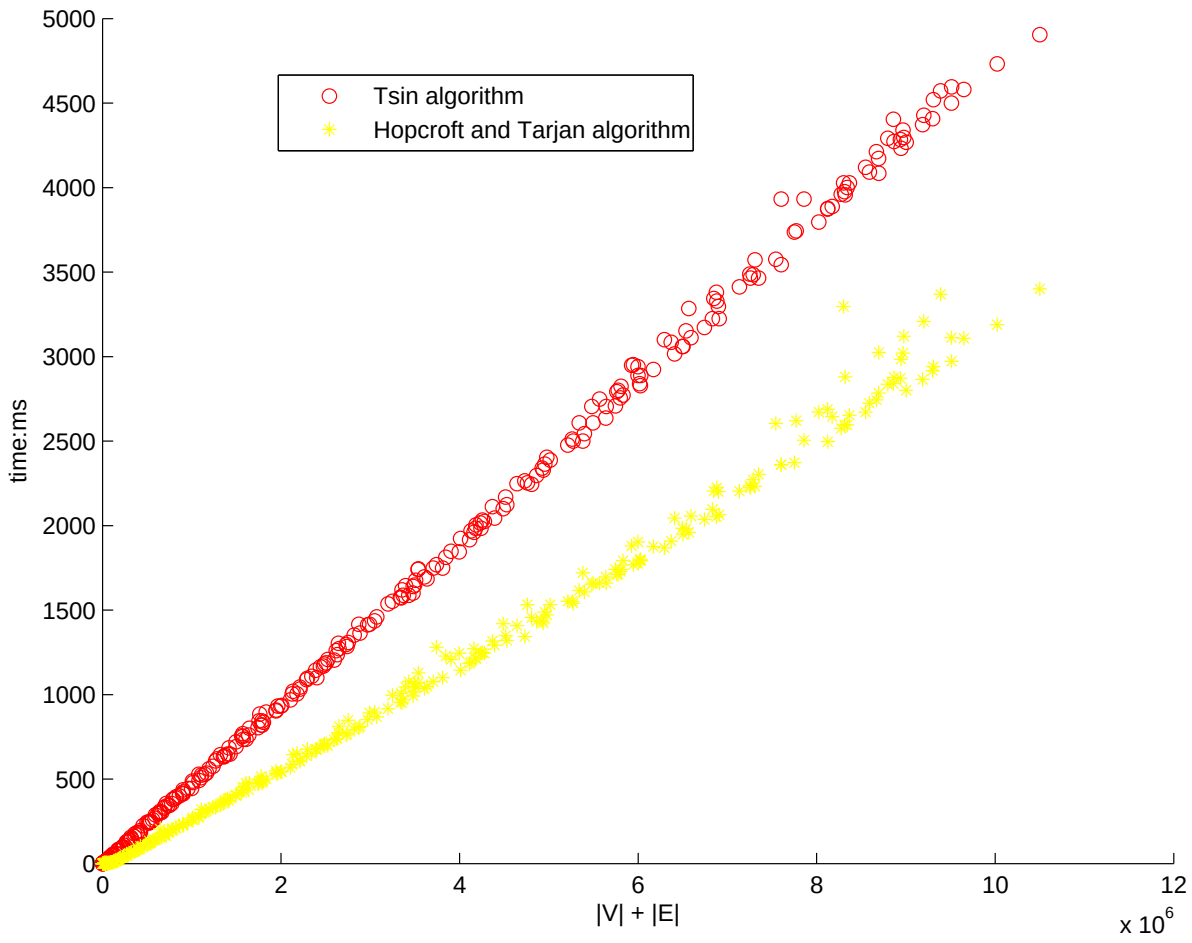


Figure 5.22: Time required to find split components (dense graphs with $0.5 < k \leq 0.6$)

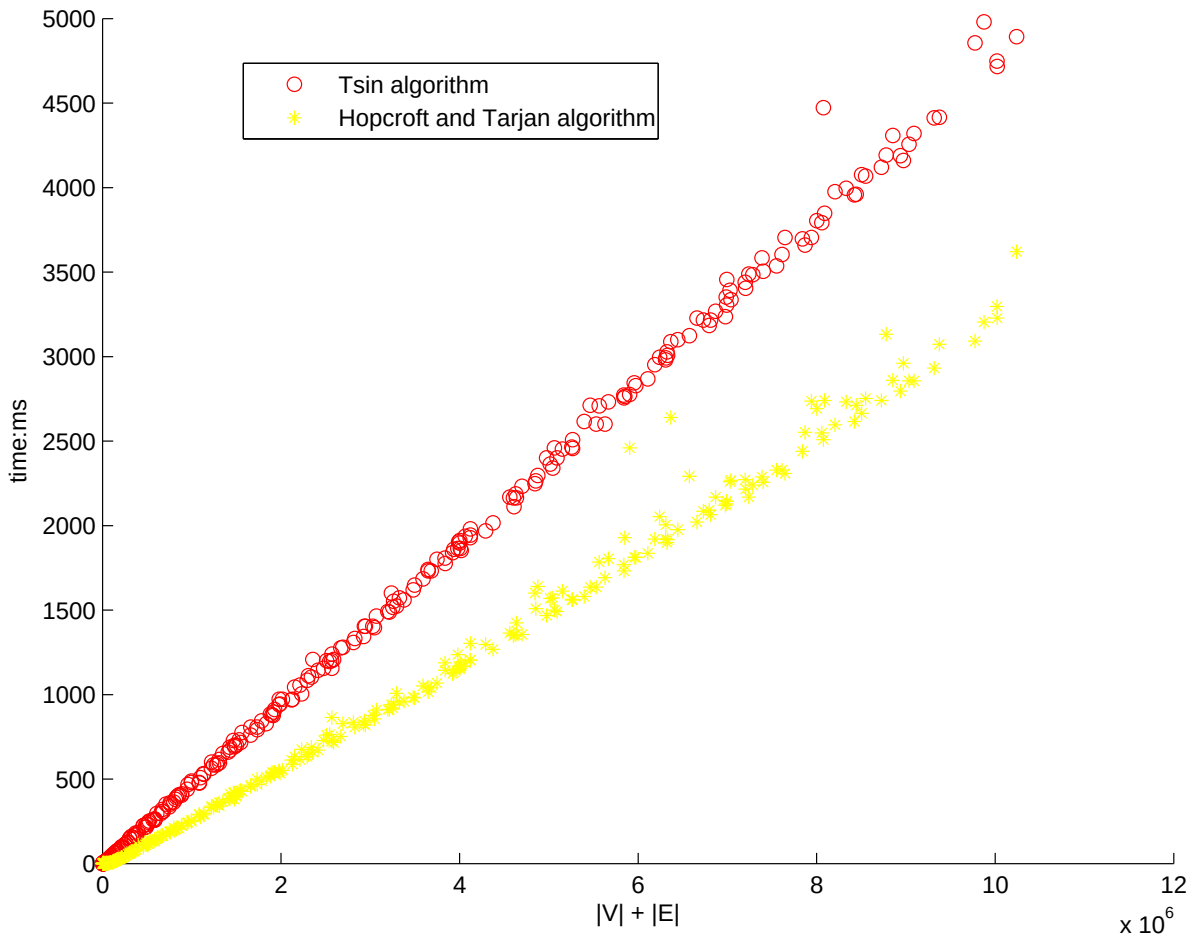


Figure 5.23: Time required to find split components (dense graphs with $0.6 < k \leq 0.7$)

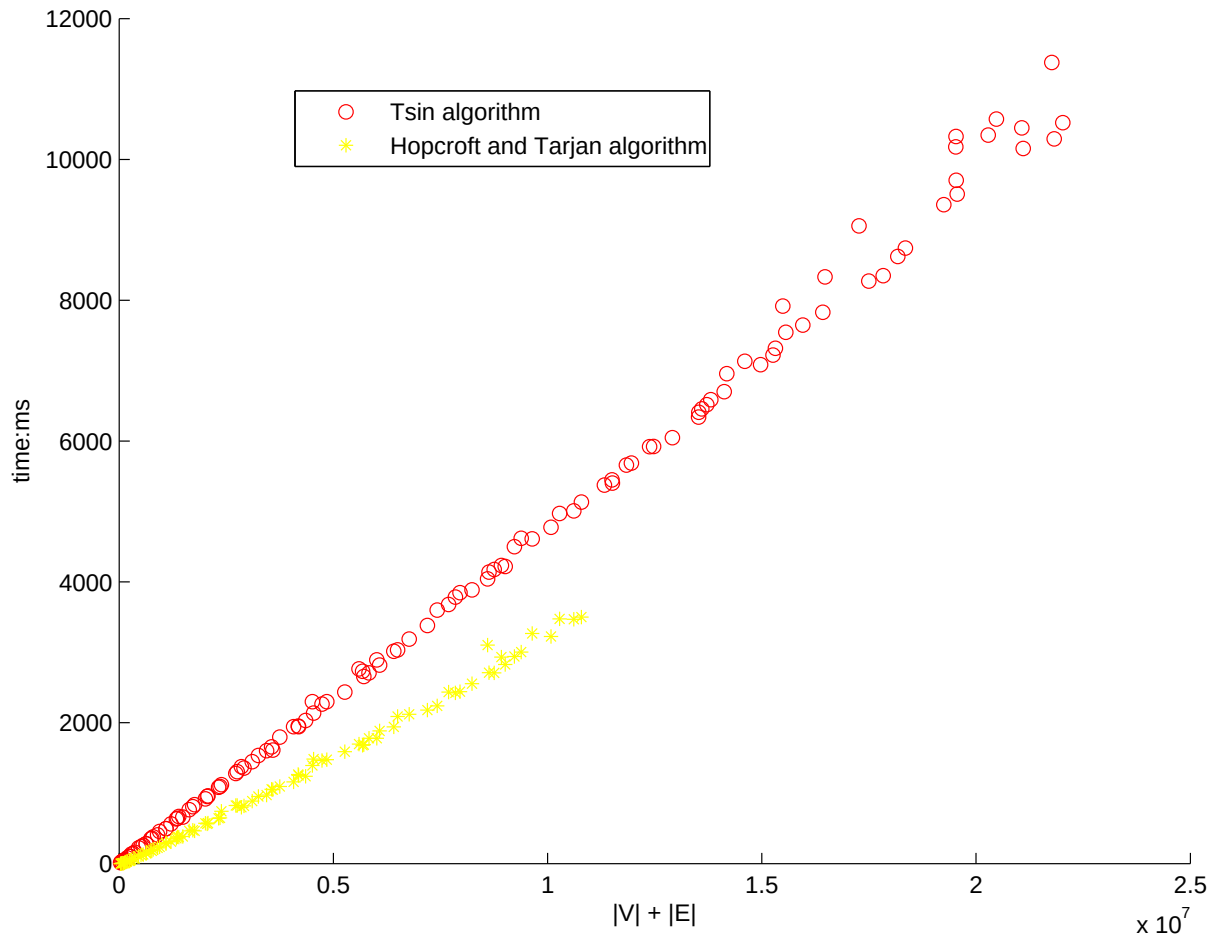


Figure 5.24: Time required to find split components (dense graphs with $0.7 < k \leq 0.8$)

From Figures 5.1 to 5.24, we observe that Tsin's algorithm runs faster than Hopcroft and Tarjan's algorithm for dense graphs. Figures 5.9 to 5.16 show that Hopcroft and Tarjan's algorithm takes much longer to create their adjacency-lists structure. This is because the algorithm needs two depth-first searches and a bucket sort to build the acceptable adjacency-lists structure. By contrast, Tsin's algorithm only needs one depth-first search. However, Tsin's algorithm takes longer to generate the split components. This is shown in Figures 5.17 to 5.24. The reason is perhaps owing to the fact that Tsin's algorithm uses linked lists to maintain the millipedes whereas Hopcroft and Tarjan's algrotihm uses a stack (i.e. an arrays) to keep edges belonging to the same split components together. Finally, it is worth noting that for the case $0.7 < k \leq 0.8$ (i.e. very

dense graphs), when the input size goes beyond 10,700,201, Hopcroft and Tarjan algrothm starts to collapse as it runs out of memory whereas Tsin's algorithm continues to run until $24,622,080$ (Figures 5.8, 5.16 and 5.24).

5.3.2 Sparse graph comparison

Figures 5.25 to 5.42 display the result of the experiment that compares the execution time of Hopcroft and Tarjan's algorithm with that of Tsin's algorithm. Specifically, Figures 5.25 to 5.30 compare the *total* execution time of the two algorithms. Figures 5.31 to 5.36 compare the execution time required by the two algorithms in creating their adjacency-lists structure. The remaining figures compare the execution time of two algorithms spent on generating the split components.

Figures 5.25, 5.31 and 5.37 display the result of running the two algorithms on 471 sparse graphs with $1 \leq \frac{|E|}{|V|} \leq 1.1$ and $1,001 \leq |V|+|E| \leq 9,828,745$.

Figures 5.26, 5.32 and 5.38 display the result of running the two algorithms on 459 sparse graphs with $1.1 \leq \frac{|E|}{|V|} \leq 1.3$ and $1,350 \leq |V|+|E| \leq 10,269,705$.

Figures 5.27, 5.33 and 5.39 display the result of running the two algorithms on 317 sparse graphs with $1.3 \leq \frac{|E|}{|V|} \leq 2$ and $129 \leq |V|+|E| \leq 10,808,947$.

Figures 5.28, 5.34 and 5.40 display the result of running the two algorithms on 195 sparse graphs with $2 \leq \frac{|E|}{|V|} \leq 5$ and $2,944 \leq |V|+|E| \leq 114,35,194$.

Figures 5.29, 5.35 and 5.41 display the result of running the two algorithms on 254 sparse graphs with $5 \leq \frac{|E|}{|V|} \leq 10$ and $3,924 \leq |V|+|E| \leq 10,094,725$.

Figures 5.30, 5.36 and 5.42 display the result of running the two algorithms on sparse graphs with $10 \leq \frac{|E|}{|V|} \leq 100$. For Hopcroft and Tarjan's algorithm, 220 sparse graphs were used with $34,175 \leq |V|+|E| \leq 10,993,919$; for Tsin's algorithm, 239 sparse graphs were used with $34,175 \leq |V|+$

$|E| \leq 11,494,050.$

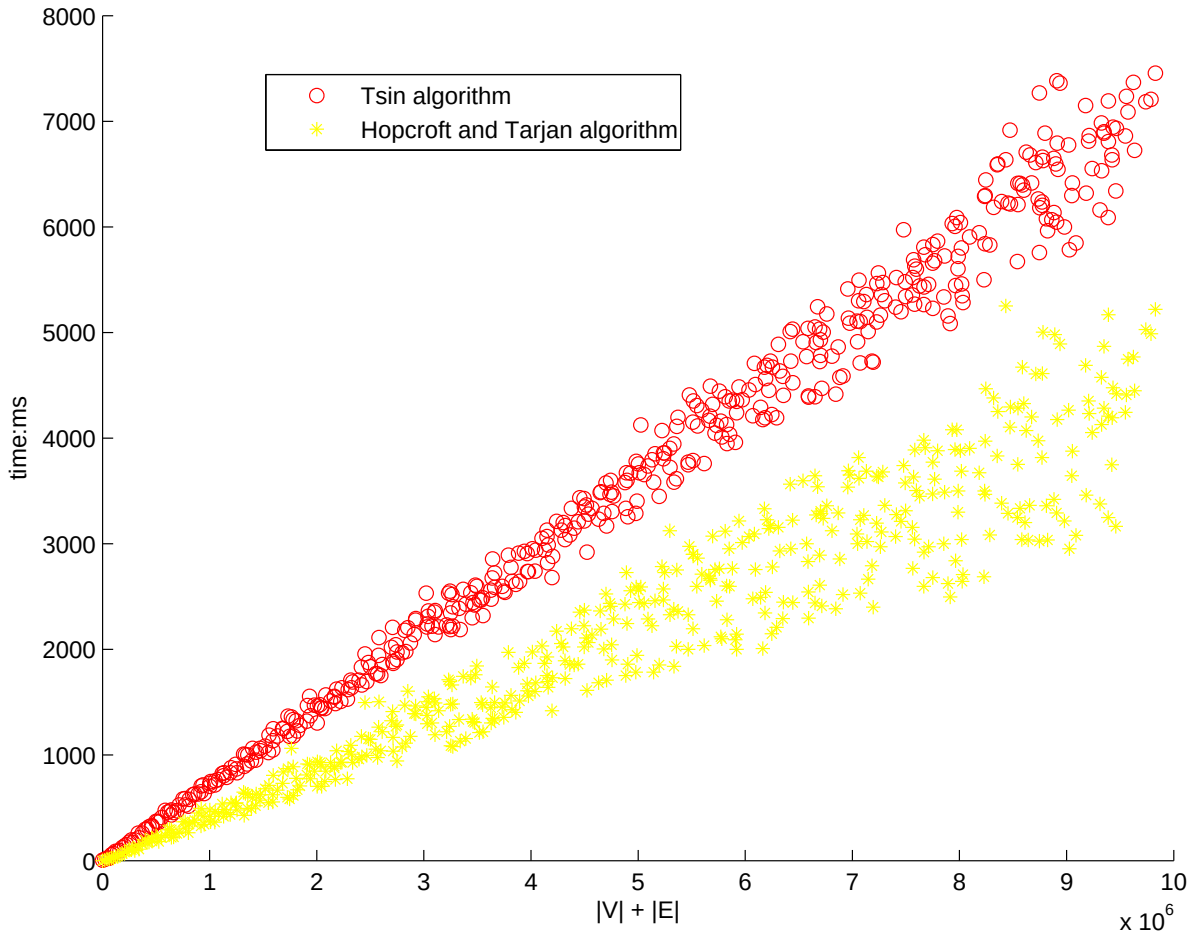


Figure 5.25: Total execution time (sparse graphs with $1 \leq \frac{|E|}{|V|} < 1.1$)

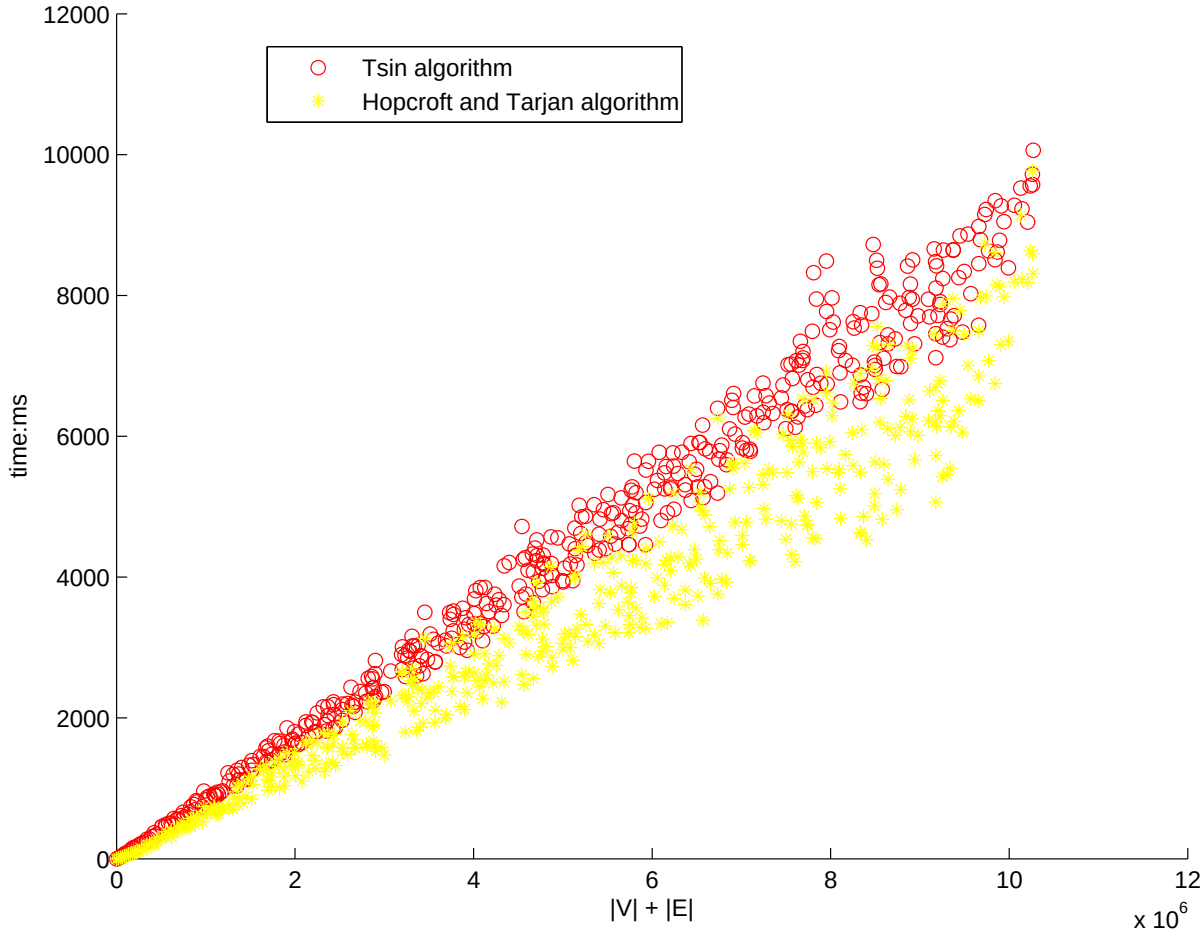


Figure 5.26: Total execution time (sparse graphs with $1.1 \leq \frac{|E|}{|V|} < 1.3$)

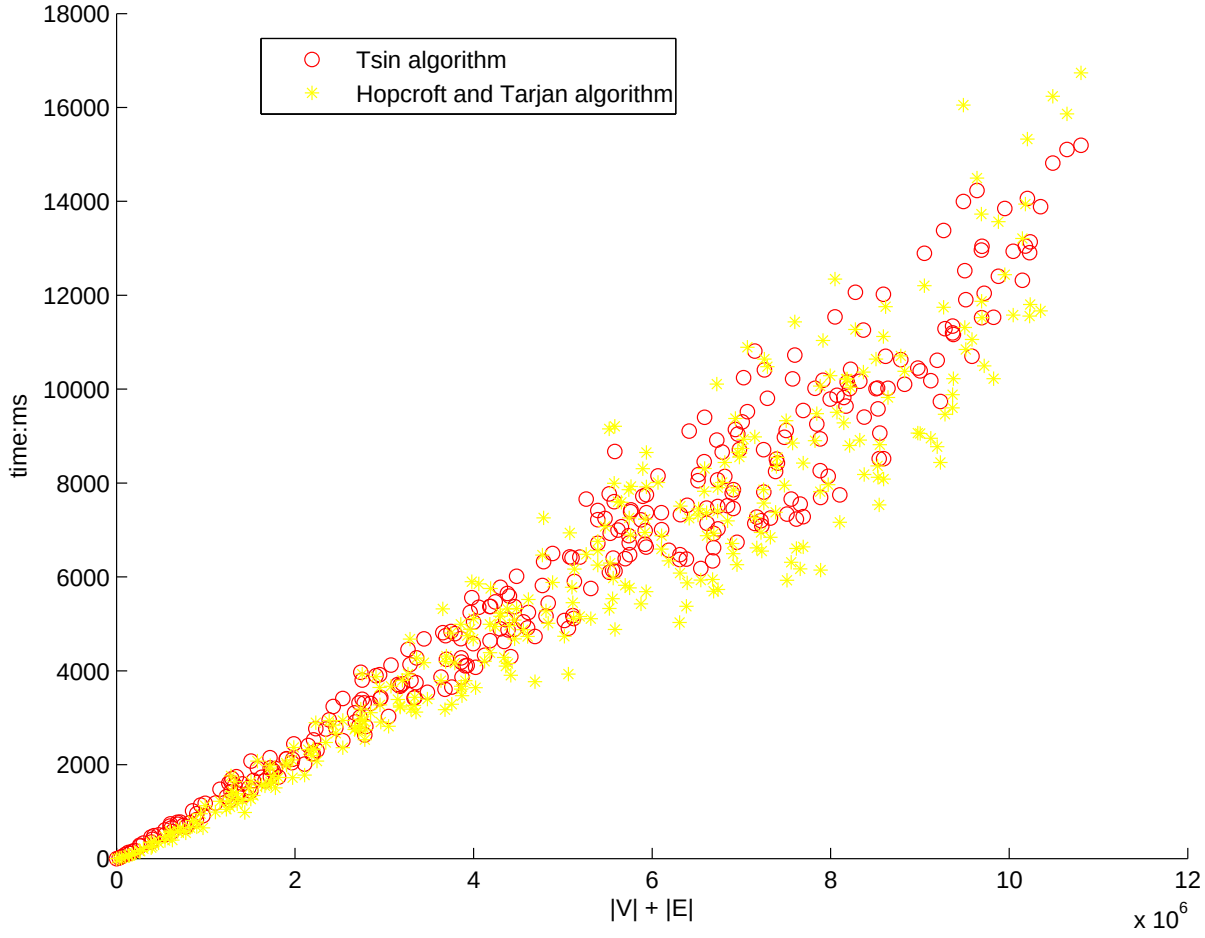


Figure 5.27: Total execution time (sparse graphs with $1.3 \leq \frac{|E|}{|V|} < 2$)

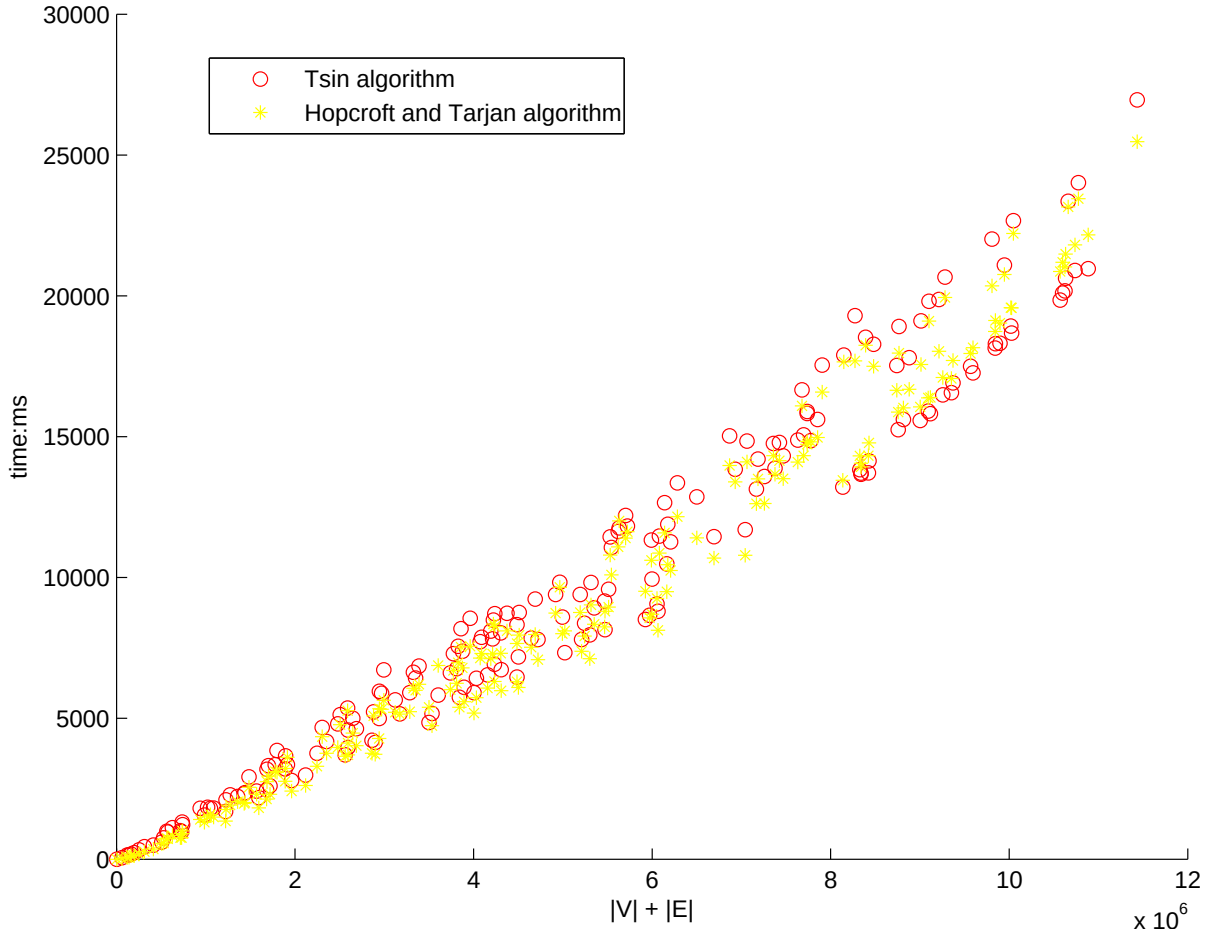


Figure 5.28: Total execution time (sparse graphs with $2 \leq \frac{|E|}{|V|} < 5$)

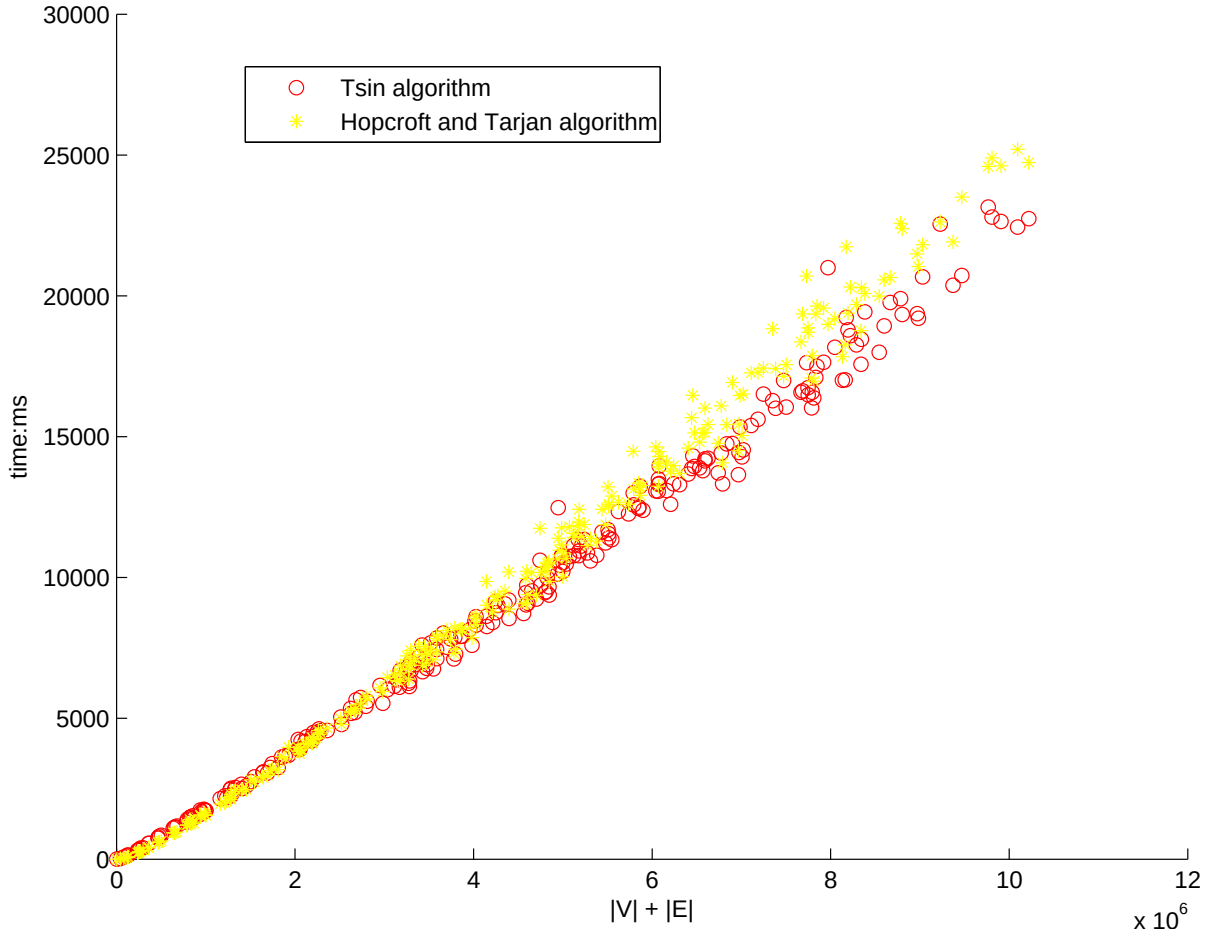


Figure 5.29: Total execution time (sparse graphs with $5 \leq \frac{|E|}{|V|} < 10$)

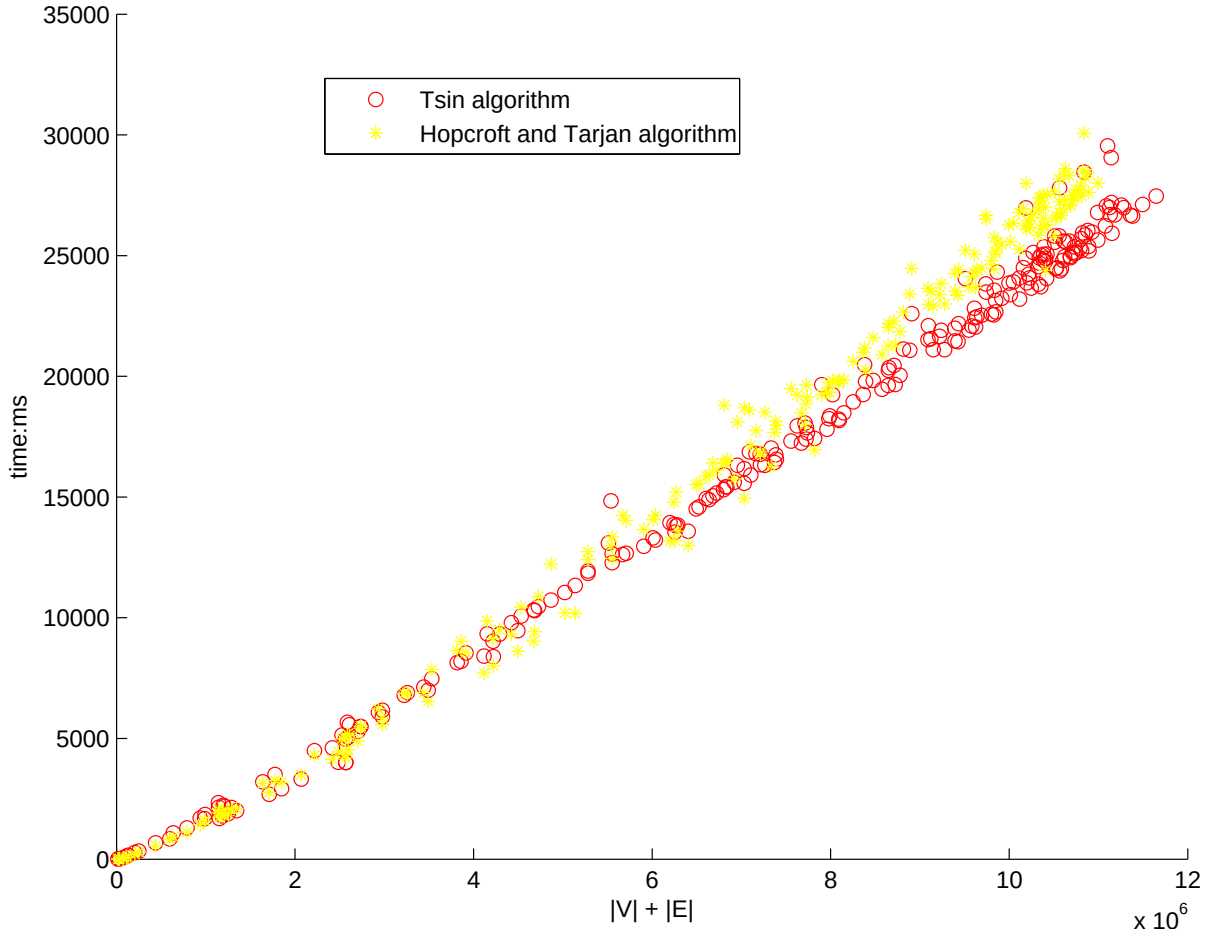


Figure 5.30: Total execution time (sparse graphs with $10 \leq \frac{|E|}{|V|} < 100$)

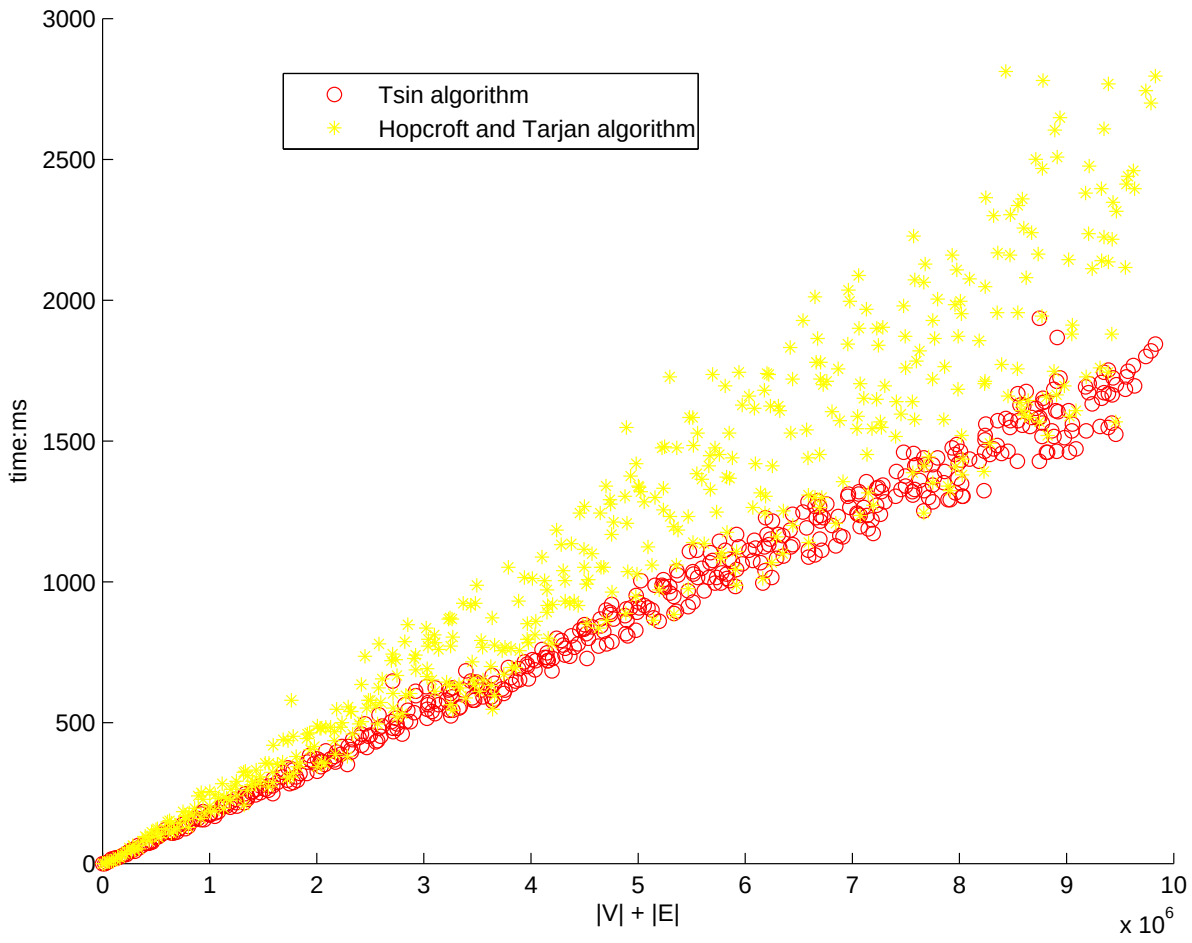


Figure 5.31: Time required to create the adjacency-lists (sparse graphs with $1 \leq \frac{|E|}{|V|} < 1.1$)

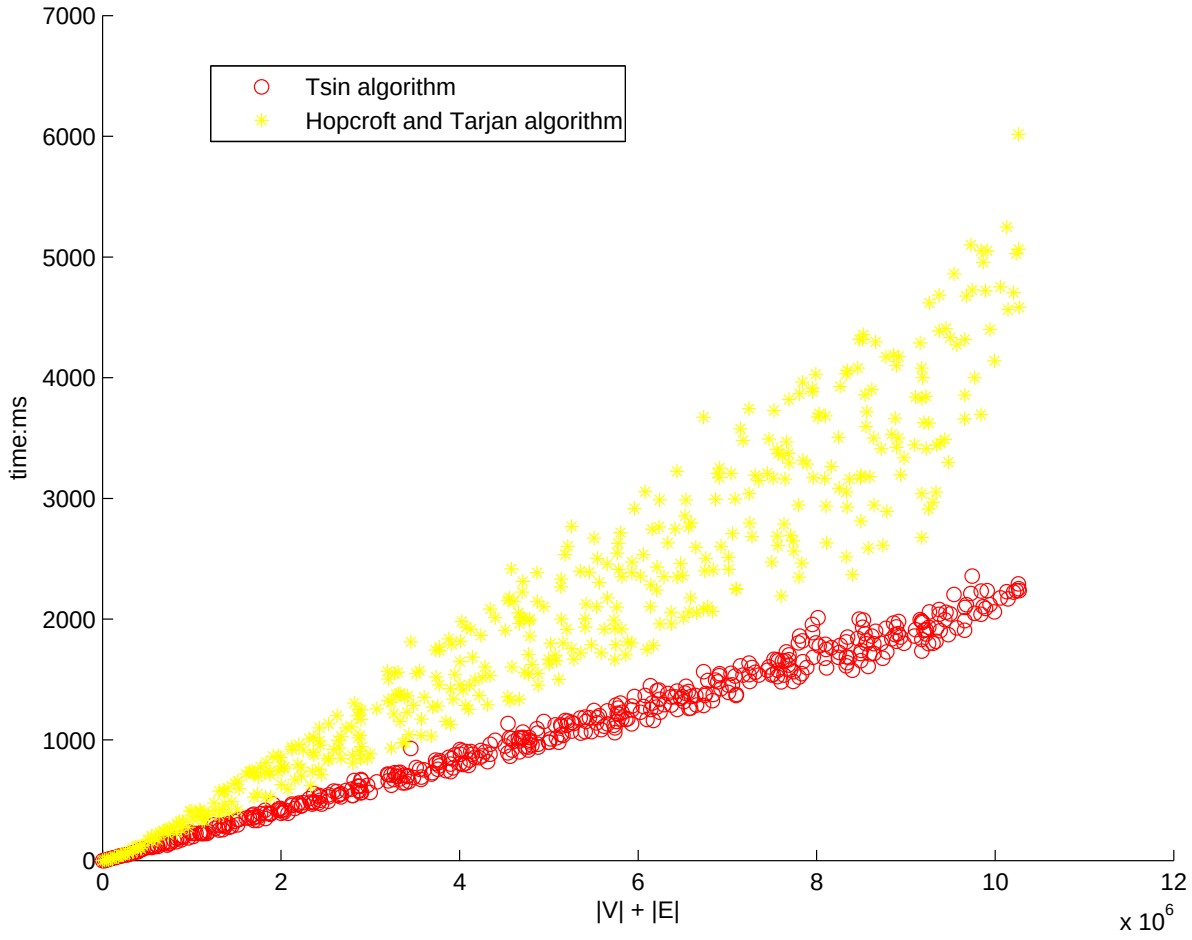


Figure 5.32: Time required to create the adjacency-lists (sparse graphs with $1.1 \leq \frac{|E|}{|V|} < 1.3$)

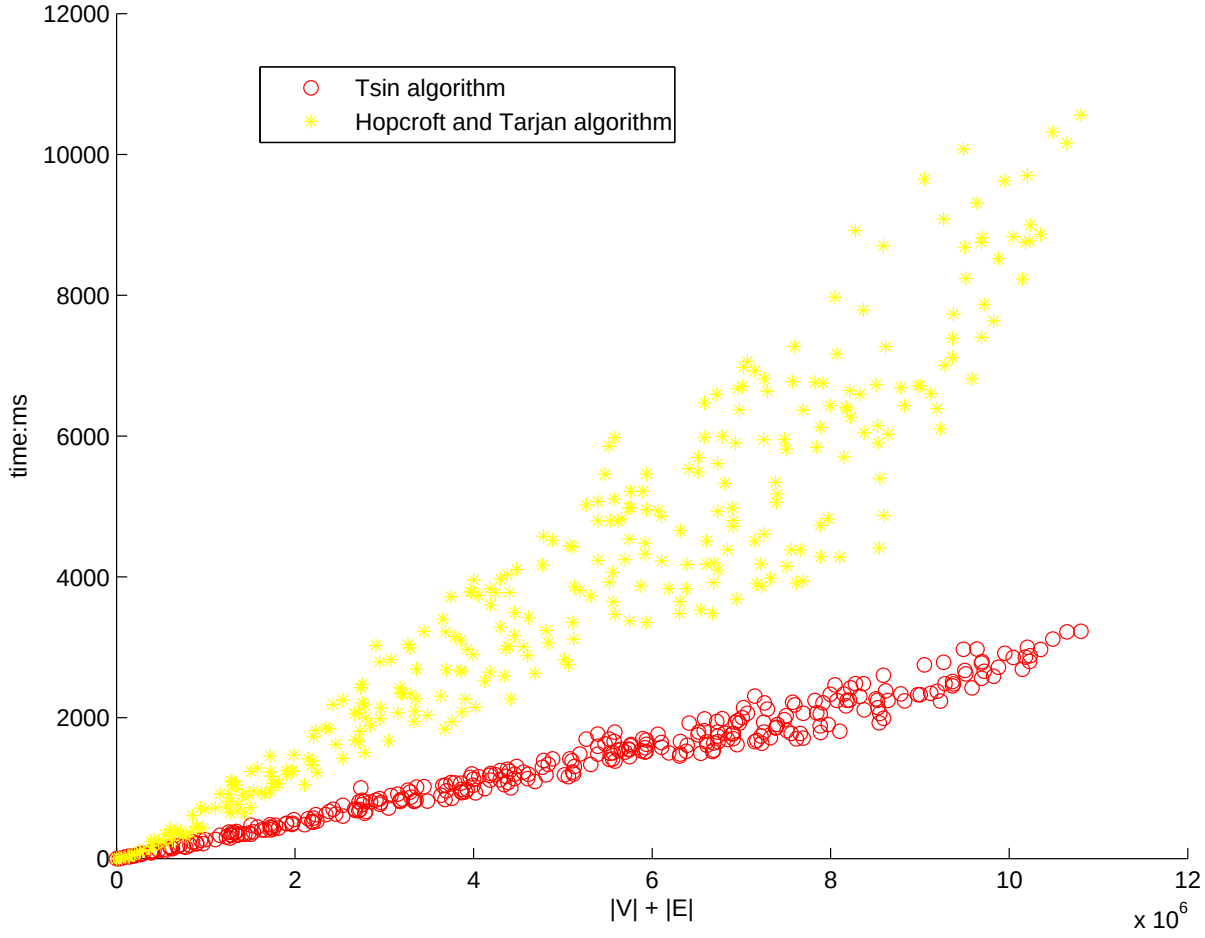


Figure 5.33: Time required to create the adjacency-lists (sparse graphs with $1.3 \leq \frac{|E|}{|V|} < 2$)

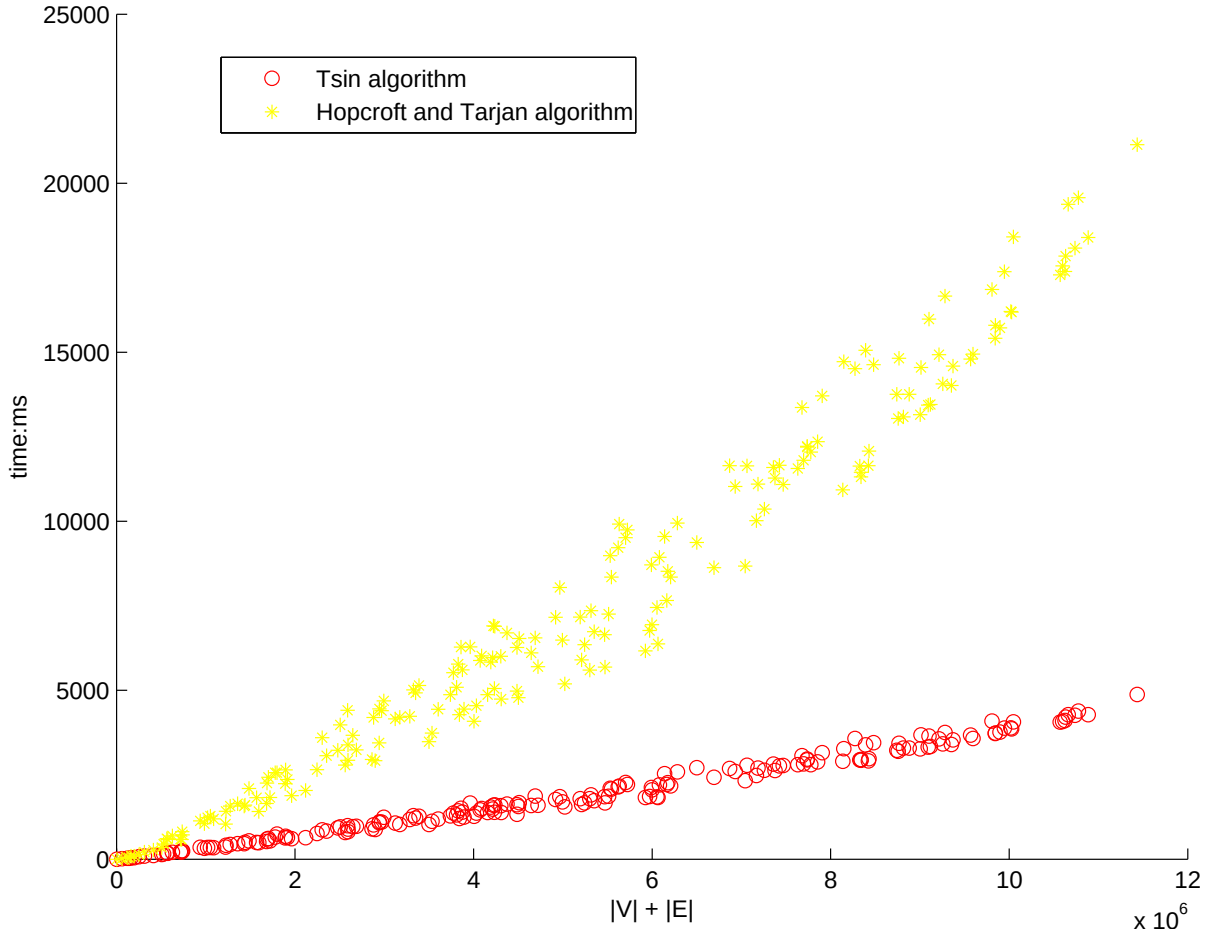


Figure 5.34: Time required to create the adjacency-lists (sparse graphs with $2 \leq \frac{|E|}{|V|} < 5$)

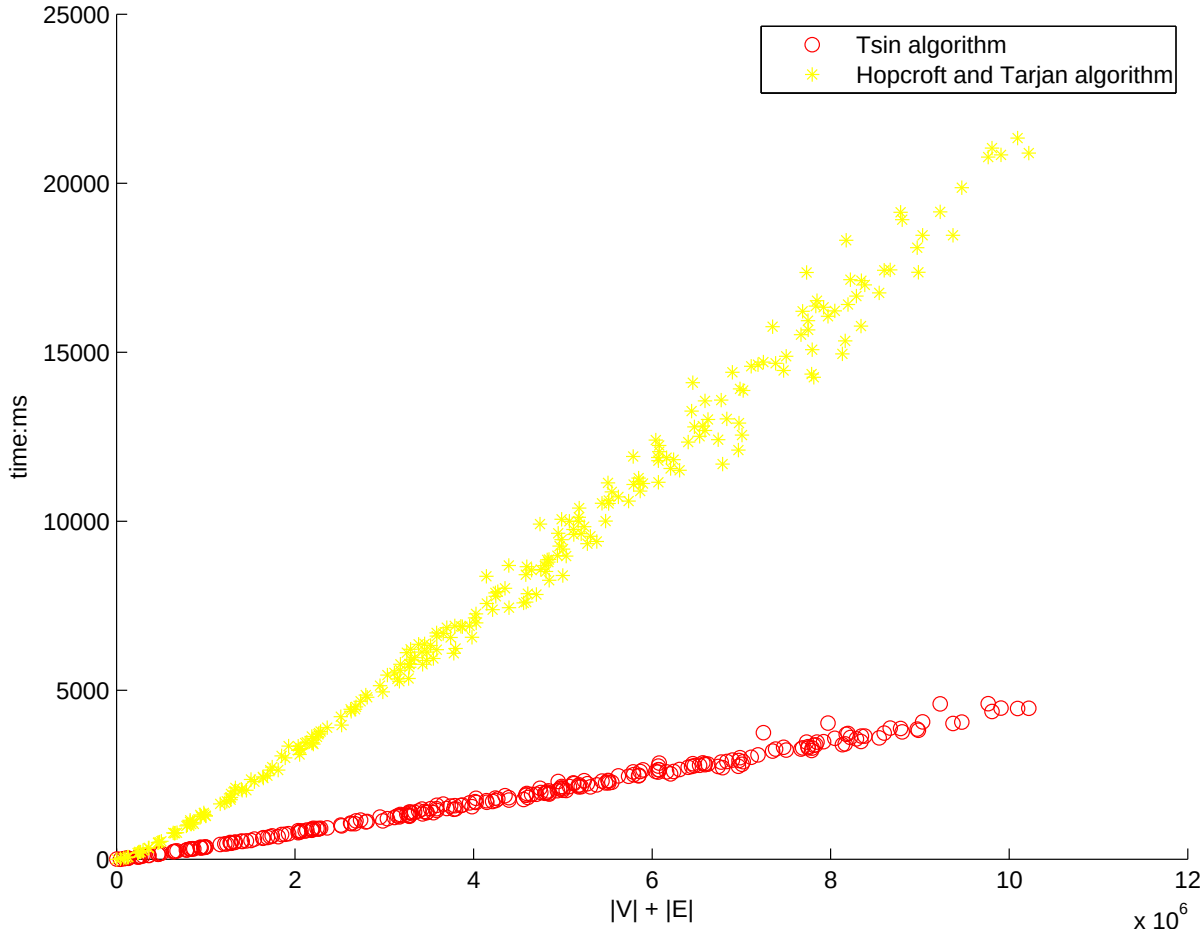


Figure 5.35: Time required to create the adjacency-lists (sparse graphs with $5 \leq \frac{|E|}{|V|} < 10$)

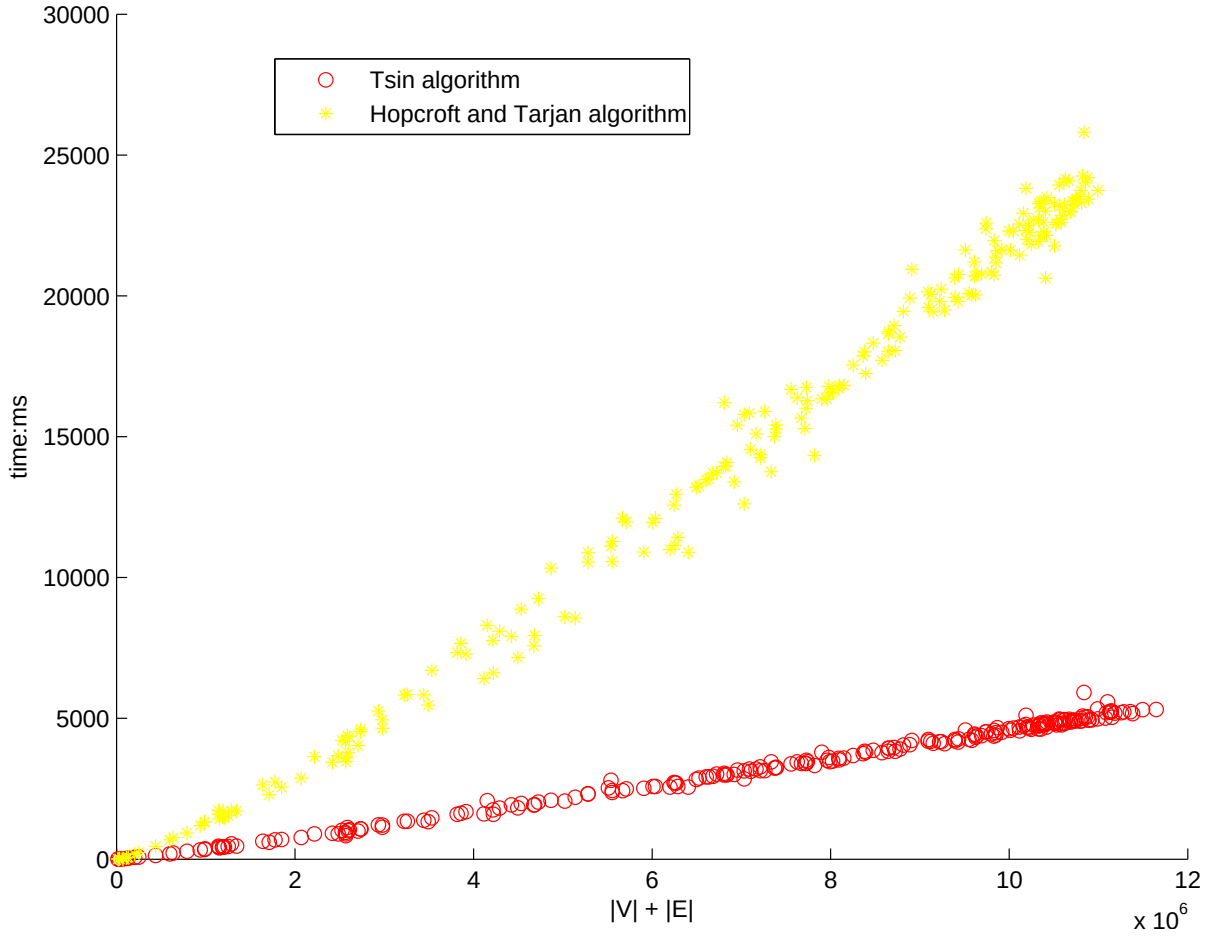


Figure 5.36: Time required to create the adjacency-lists (sparse graphs with $10 \leq \frac{|E|}{|V|} < 100$)

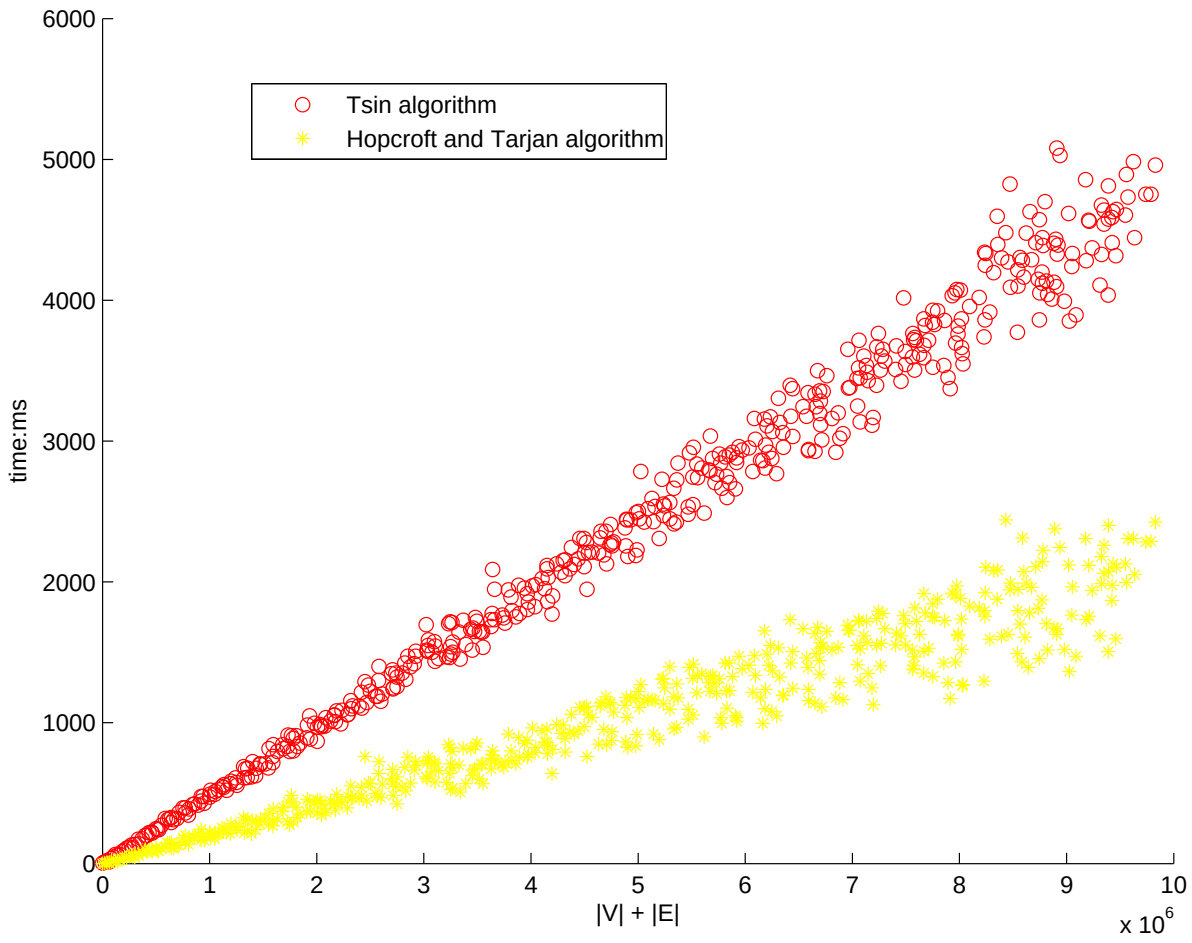


Figure 5.37: Time required to find split components (sparse graphs with $1 \leq \frac{|E|}{|V|} < 1.1$)

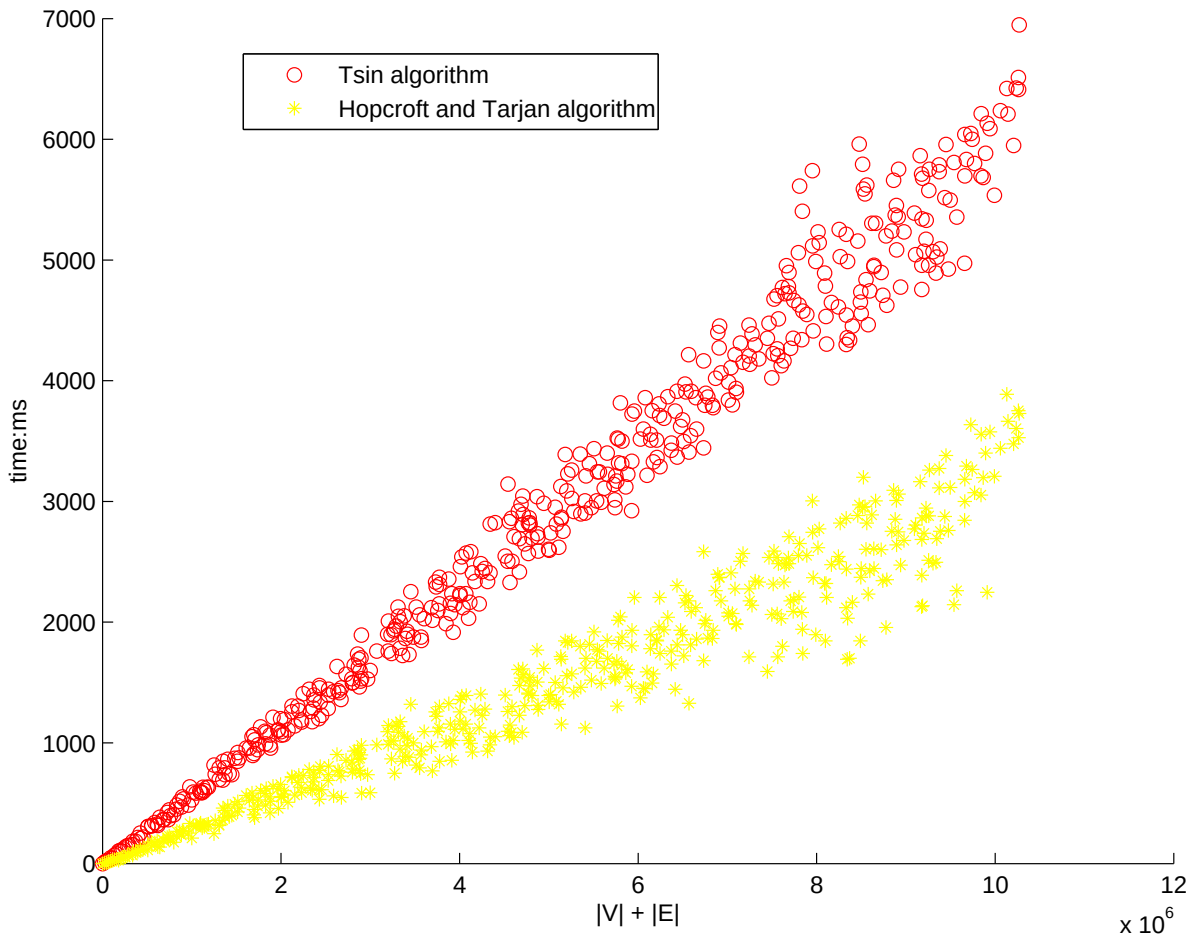


Figure 5.38: Time required to find split components (sparse graphs with $1.1 \leq \frac{|E|}{|V|} < 1.3$)

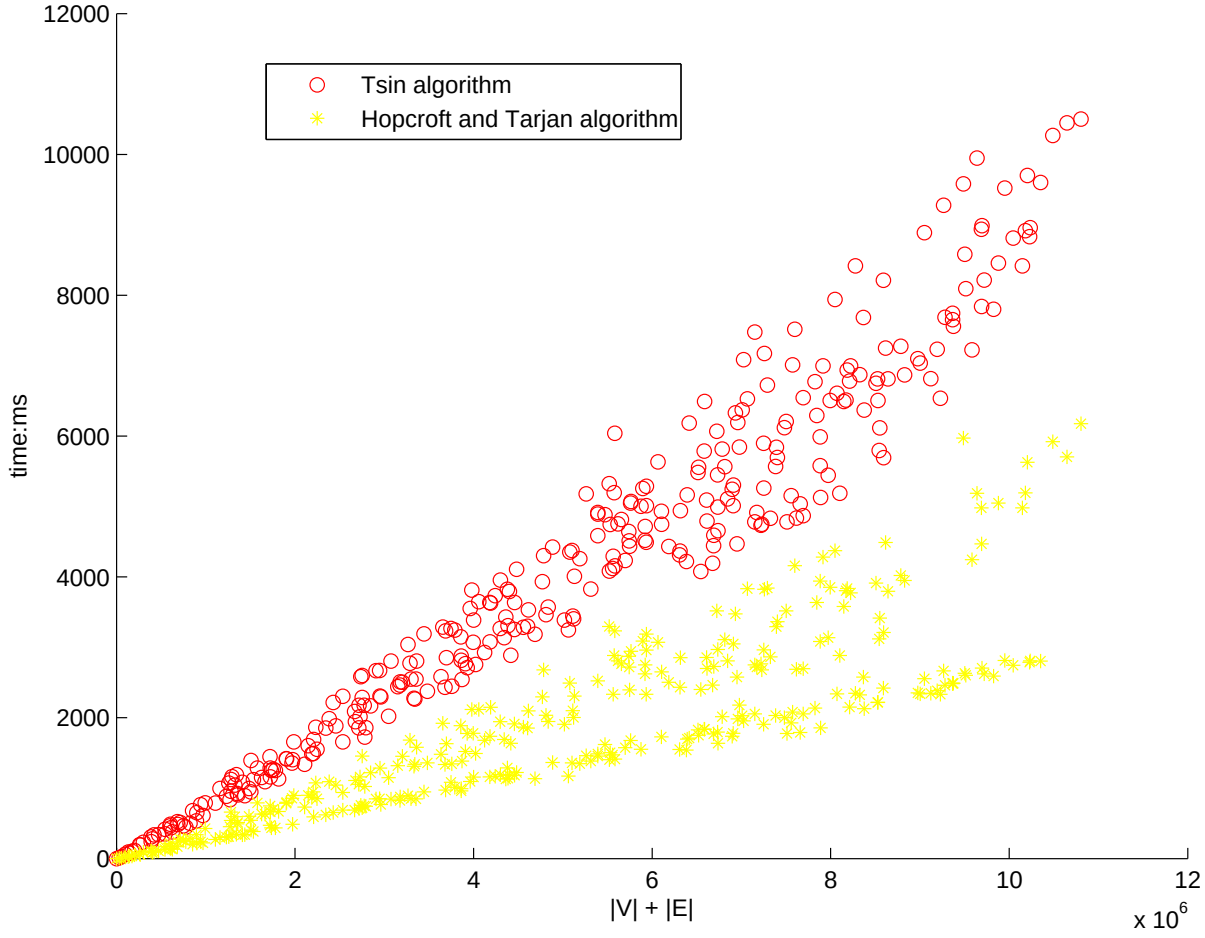


Figure 5.39: Time required to find split components (sparse graphs with $1.3 \leq \frac{|E|}{|V|} < 2$)

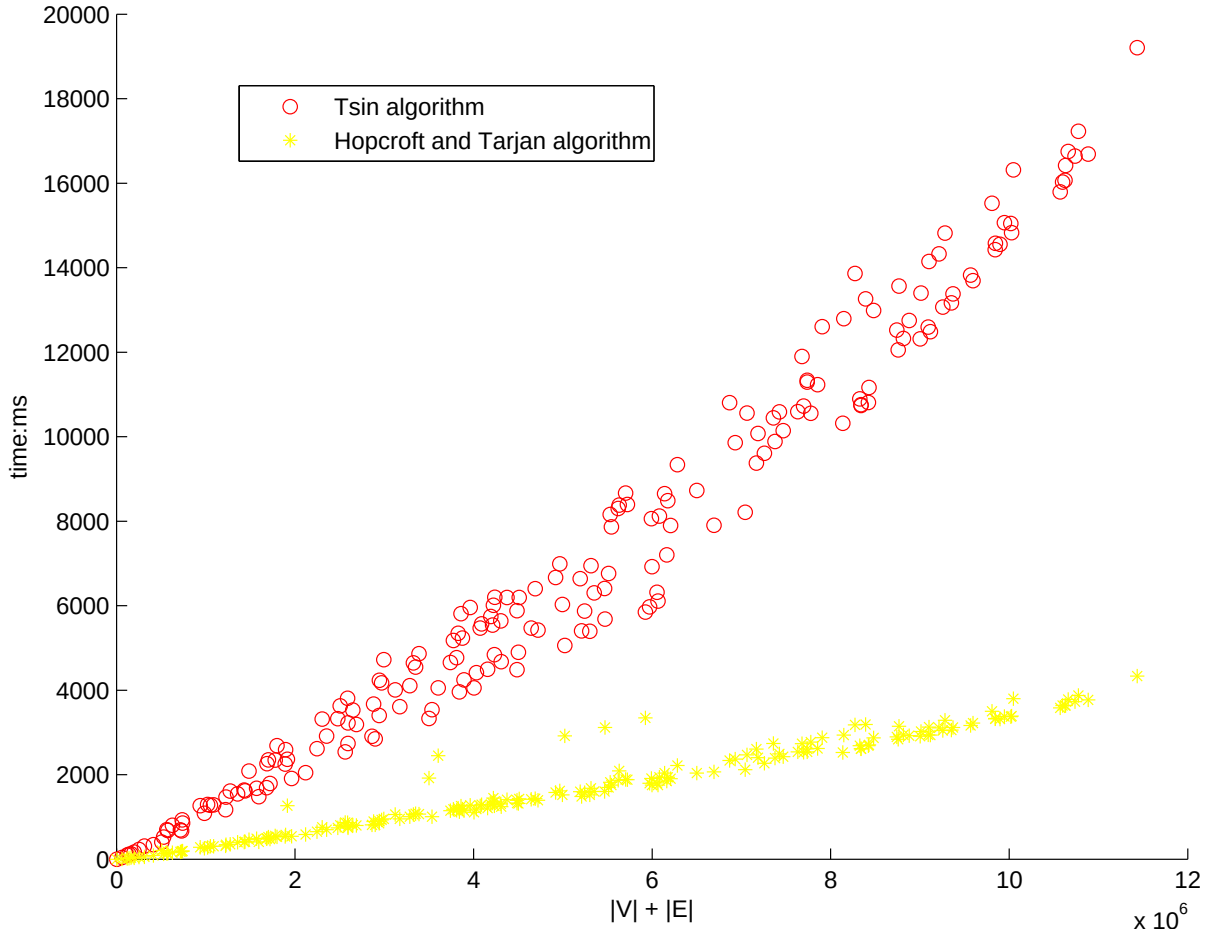


Figure 5.40: Time required to find split components (sparse graphs with $2 \leq \frac{|E|}{|V|} < 5$)

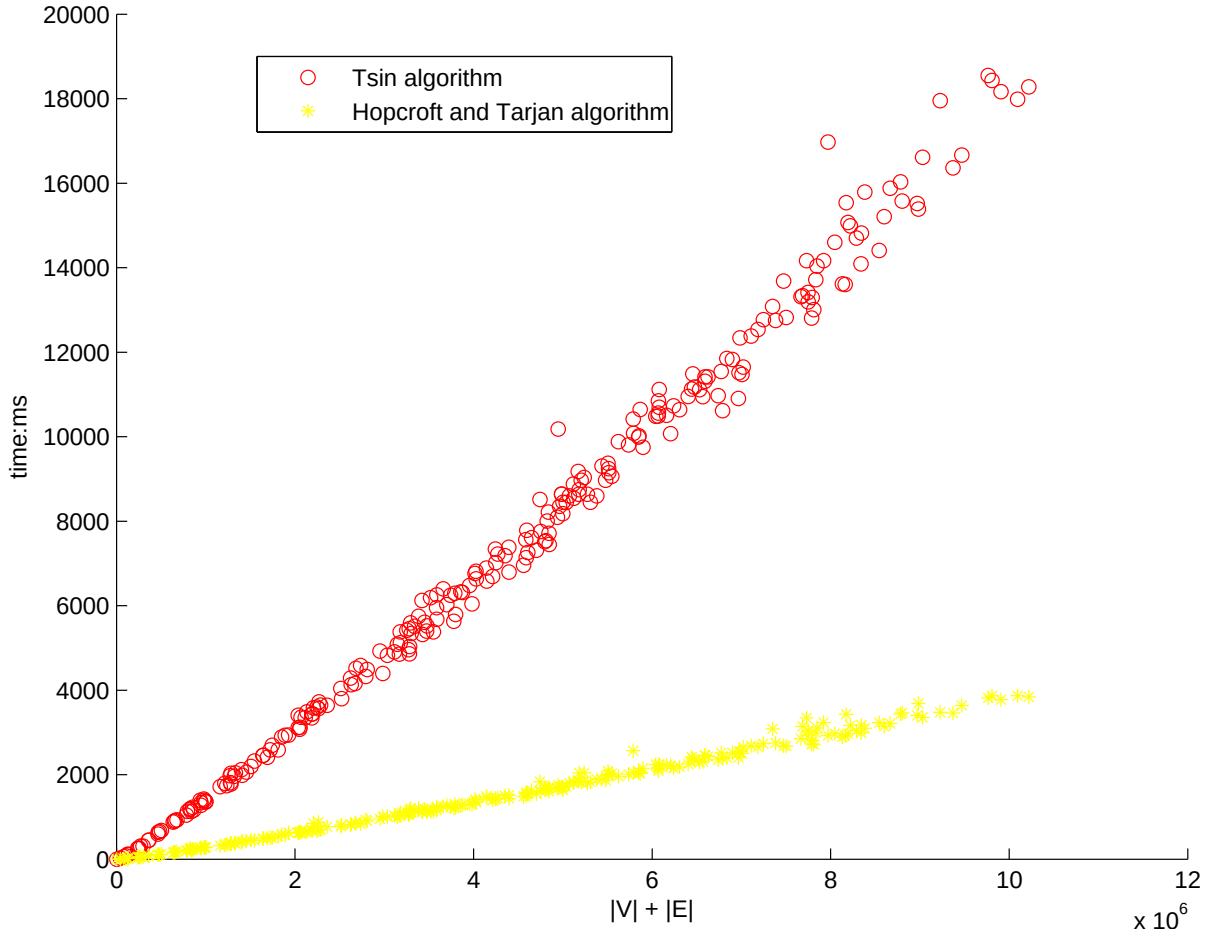


Figure 5.41: Time required to find split components (sparse graphs with $5 \leq \frac{|E|}{|V|} < 10$)

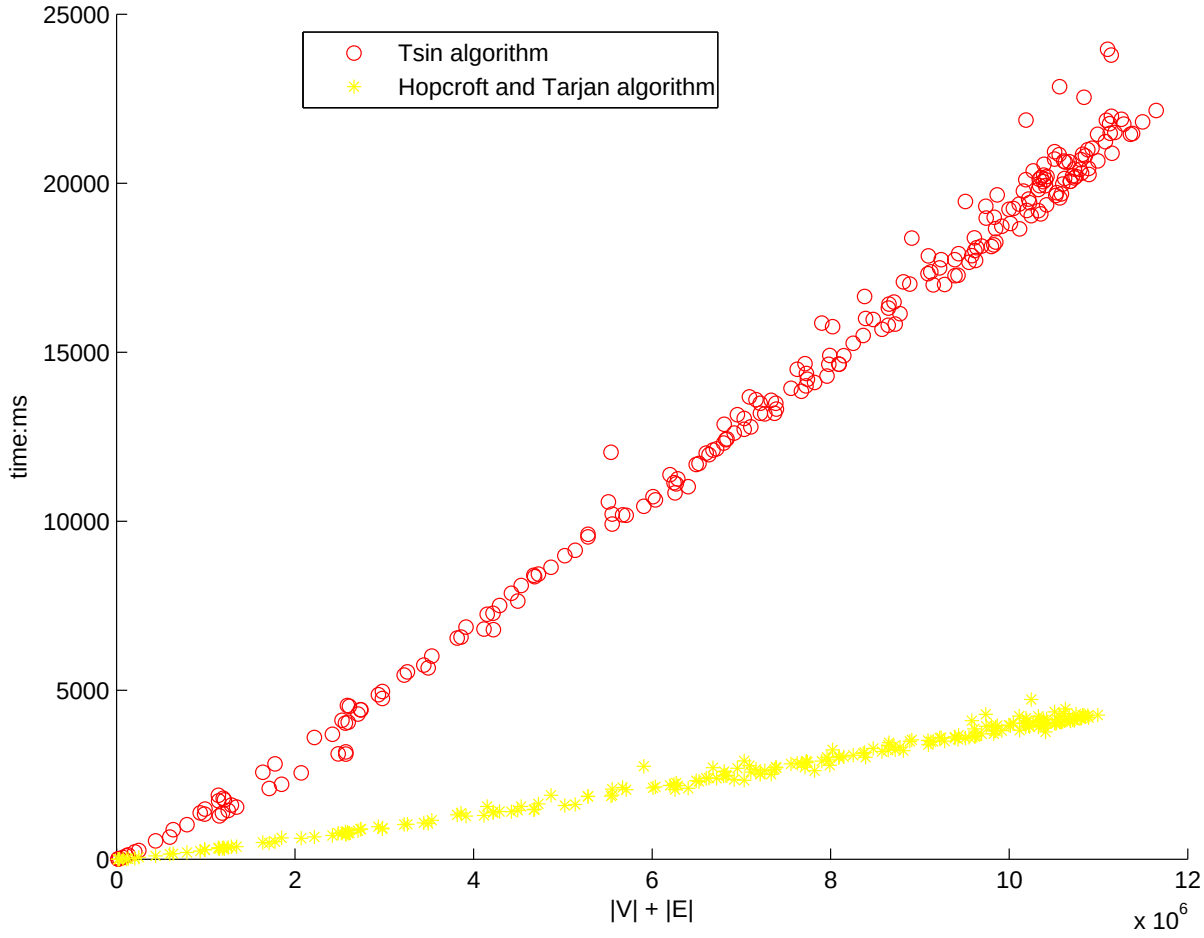


Figure 5.42: Time required to find split components (sparse graphs with $10 \leq \frac{|E|}{|V|} < 100$)

From Figures 5.25 and 5.26, we observe that Tsin's algorithm runs slower than Hopcroft and Tarjan's algorithm when $1 \leq \frac{|E|}{|V|} < 1.3$. When $1.3 \leq \frac{|E|}{|V|} < 5$ (Figures 5.27, and 5.28), the execution times of two algorithms are almost the same, but when $5 \leq \frac{|E|}{|V|} < 100$ (Figures 5.29 and 5.30), Tsin's algorithm runs faster.

As with the case for dense graphs, Figures 5.31 to 5.36 show that Tsin's algorithm uses less time to create the adjacency-lists structure while Figures 5.37 to 5.42 show that Hopcroft and Tarjan's algorithm runs faster in generating the split components.

Finally, it is worth noting that for the case $10 < \frac{|E|}{|V|} \leq 100$, when the input size exceeds 10,993,919,

Hopcroft and Tarjan algrothm starts to collapse as it runs out of memory whereas Tsin's algorithm continues to run until $11,494,050$.

Chapter 6

Conclusion

From Chapter 5, we observe that, for dense graphs, while Tsin's algorithm runs much faster than Hopcroft and Tarjan's algorithm in creating the adjacency-lists structure, it runs slower in generating split components. Overall, Tsin's algorithms runs faster than Hopcroft and Tarjan's algorithm for dense graphs.

For sparse graphs, when $1 < \frac{|E|}{|V|} \leq 1.3$, Tsin's algorithm runs slower than Hopcroft and Tarjan's algorithm; when $1.3 < \frac{|E|}{|V|} \leq 5$, the execution times of two algorithms are almost the same; when $5 < \frac{|E|}{|V|} \leq 100$, Tsin's algorithm runs faster than Hopcroft and Tarjan's algorithm.

In conclusion, for dense graphs, Tsin's algorithm should be used. For sparse graphs, Tsin's algorithm should be used when $5 < \frac{|E|}{|V|} \leq 100$ whereas Hopcroft and Tarjan's algorithm should be used when $1 < \frac{|E|}{|V|} \leq 1.3$. For $1.3 < \frac{|E|}{|V|} \leq 5$, either algorithm can be used.

Bibliography

Boffey, T. (1992), *Graph theory in operations research*, Scholium International.

Chin, F., Chrobak, M. & Yan, L. (2009), Algorithms for placing monitors in a flow network, *in* 'Algorithmic Aspects in Information and Management', Springer, pp. 114–128.

Coleman, T. F. & Moré, J. J. (1983), 'Estimation of sparse jacobian matrices and graph coloring blems', *SIAM journal on Numerical Analysis* **20**(1), 187–209.

Ellis-Monaghan, J. A. & Gutwin, P. (2003), 'Graph theoretical problems in next-generation chip design', *Congressus Numerantium* pp. 143–160.

Fussell, D., Ramachandran, V. & Thurimella, R. (1989), Finding triconnected components by local replacements, *in* 'Automata, Languages and Programming', Springer, pp. 379–393.

Galil, Z. & Italiano, G. F. (1991), 'Reducing edge connectivity to vertex connectivity', *ACM SIGACT News* **22**(1), 57–61.

Gutwenger, C. & Mutzel, P. (2000), 'http://www.ogdf.net/doku.php'.

Gutwenger, C. & Mutzel, P. (2001), A linear time implementation of spqr-trees, *in* 'Graph Drawing', Springer, pp. 77–90.

Hartuv, E. & Shamir, R. (2000), 'A clustering algorithm based on graph connectivity', *Information processing letters* **76**(4), 175–181.

Hopcroft, J. E. & Tarjan, R. E. (1973), 'Dividing a graph into triconnected components', *SIAM Journal on Computing* **2**(3), 135–158.

Jungnickel, D. (2008), *Graphs, networks and algorithms*, Springer.

Knauer, B. (1975), 'A simple planarity criterion', *Journal of the ACM (JACM)* **22**(2), 226–230.

Mallach, S. (2011), On separation pairs and split components of biconnected graphs, Technical report, Institut fur Informatik, Universitat zu Koln, Germany.

Miller, G. L. & Ramachandran, V. (1992), 'A new graph triconnectivity algorithm and its parallelization', *Combinatorica* **12**(1), 53–76.

Nagamochi, H. & Ibaraki, T. (1992), 'A linear time algorithm for computing 3-edge-connected components in a multigraph', *Japan journal of industrial and applied mathematics* **9**(2), 163–180.

Nakanishi, N. (1971), *Graph theory and Feynman integrals*, Gordon and Breach New York.

Novak, L. & Gibbons, A. (2009), *Hybrid graph theory and network analysis*, Cambridge University Press.

Saifullah, A. M. & Üngör, A. (2009), A simple algorithm for triconnectivity of a multigraph, *in* 'Proceedings of the Fifteenth Australasian Symposium on Computing: The Australasian Theory-Volume 94', Australian Computer Society, Inc., pp. 53–62.

Taoka, S., Watanabe, T. & Onaga, K. (1992), 'A linear-time algorithm for computing all 3-edge-connected components of a multigraph', *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences* **75**(3), 410–424.

Tarjan, R. (1972), 'Depth-first search and linear graph algorithms', *SIAM journal on computing* **1**(2), 146–160.

Tsin, Y. H. (2007), 'A simple 3-edge-connected component algorithm', *Theory of Computing Systems* **40**(2), 125–142.

Tsin, Y. H. (2009), 'Yet another optimal algorithm for 3-edge-connectivity', *Journal of Discrete Algorithms* **7**(1), 130–146.

Tsin, Y. H. (2012), Decomposing a multigraph into split components, *in* 'Proceedings of the Eighteenth Australasian Symposium on Computing: The Australasian Theory Symposium (CATS 2012)', pp. 3–12.

Vo, K. P. (1983), 'Finding triconnected components of graphs', *Linear and Multilinear Algebra* **13**(2), 143–165.

Vita Auctoris

Zhigang Jiang was born in 1986 in Hunan, China. He obtained his Bachelor degree of Information and Computation Science from Changsha University, he went to University of Windsor in 2011, and graduated with a Master of Computer Science degree in 2013.