Using Top Trees For Easy Programming of Tree Algorithms

Michal Vajbar

Department of Software Engineering, Faculty of Mathematics And Physics, Charles University in Prague, Malostranské nám. 25, 118 00, Praha 1, Czech Republic
`michal.vajbar@mff.cuni.cz`

Abstract. Top trees are a dynamic self-adjusting data structure that can be used by any tree algorithm. Actually, an arbitrary number of different tree algorithms can use a single structure. In our previous work, we have implemented top trees, but the usage still requires a detail knowledge of the structure which is quite complex. In this paper, we introduce Top Tree Friendly Language (TFL) and Top Tree Query Language (TQL). The TFL is a special programming language which combines declarative and procedural approaches that results in simpler and faster algorithm designing. The query language TQL provides an easy top trees administration. The implementation of top trees, the programming language TFL and the query language TQL together form a complex solution for using top trees.

Keywords: data structure, top trees, programming language, query language

1 Introduction

There exist many algorithms that work over tree graphs. Any of them can use different data structures to represent the same tree. If we need to run several of these algorithms over one forest together each one could use its own representation of the forest. Moreover, if the forest dynamically changes over the time by edge insertions and deletions all these changes have to be reflected into the data structures of all algorithms. This is not efficient. Thus several data structures have been proposed to solve this problem. Any of them is good at some properties, but each has a weak point. It appears that top trees provide the most acceptable trade-off. In this paper, we present a complex solution that allows easy using of top trees. The solution is built on our implementation of the structure [7] and it brings Top Tree Friendly Language (TFL) and Top Tree Query Language (TQL). The TFL is a special programming language which combines declarative and procedural approaches that results in simpler and faster algorithm designing. The query language TQL provides easy top trees administration.

The paper is organized as follows. In Section 2 we introduce top trees. Used implementation of top trees is shortly presented in Section 3. Section 4 introduces the TFL programming language and shows how to use it. In Section 5 we present the query language TQL. Final remarks are made in Section 6.

2 Top Trees

Top trees are a dynamic self-adjusting data structure that was proposed by Alstrup et al. [1]. A top tree R is an ordinary binary tree with a root. It is used to represent a tree graph T with defined information that some tree algorithm works with. The structure R consists of clusters. The edges of the T form basic clusters and two clusters connected through a common vertex create other cluster. So the R records a way of clusters connecting into one root cluster that represents whole T. Each cluster holds information of appropriate part of the T. A manner how the information is computed during joining and splitting of clusters characterizes used algorithm. That is a simplified idea. In the rest of this section, the structure is described more precisely.

2.1 Formal Definition and Properties

The structure is defined over a pair consisting of a tree T and a set ∂T of at most two vertices from T that are called *external boundary vertices*. Let $(T, \partial T)$ is the pair, then any subtree C of T has a set $\partial_{(T,\partial T)} C$ of at most two *boundary vertices* from C. Each of them is either from ∂C or incident to an edge from $T \setminus C$. The subtree in undirected graph means any connected subgraph. The subtree C is called a *cluster* of $(T, \partial T)$ if it has at least one edge and at most two boundary vertices. Then T is also cluster and $\partial_{(T,\partial T)} T = \partial T$. If A is a subtree of C then $\partial_{(C,\partial_{(T,\partial T)}C)} A = \partial_{(T,\partial T)} A$. This means that A is a cluster of $(C, \partial_{(T,\partial T)} C)$ if and only if A is a cluster of $(T, \partial T)$. We will use ∂ as a shortcut for $\partial_{(T,\partial T)}$ and also for all subtrees of T.

Definition: A top tree R over $(T, \partial T)$ is a binary tree such that:

1. The nodes of R are the clusters of $(T, \partial T)$.
2. The leaves of R are the edges of T.
3. Two clusters are called *neighbours* if they intersect in a single vertex. Their union is a *parent cluster* (Fig. 1).
4. The root of R is T itself.

Top tree R represents whole T. If T consists of a single vertex then it has an empty top tree. The edges of T are basic building blocks of the clusters and the vertices represent endpoints of the clusters. That is why the cluster is consisted of at least one edge. The neighbouring clusters are edge-disjunct with one common vertex (Fig. 1).

Let C is a cluster. A vertex v is an *internal vertex* of C when $v \in C$ and $v \notin \partial C$. If C has two boundary vertices a and b then C is called a *path cluster* and the path $a \ldots b$ is called the *cluster path* of C. If C has one boundary vertex a then C is called a *point cluster*.

Top tree nodes contain pointers to their sons and parents. Each node represents a cluster and there is a set of at most two boundary vertices associated with the cluster. The leaf nodes represent the edges. The non-leaf node holds information how it is joined from its sons (Fig. 1). According to this information it is possible to construct any cluster.

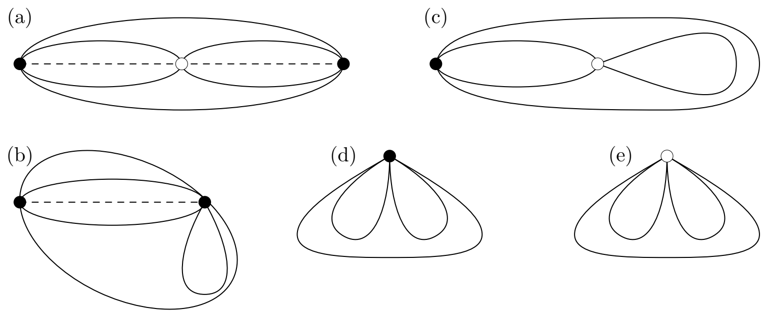

Fig. 1. The cases of joining two neighbouring clusters into the parent cluster (assumed from [1]). The • are the boundary vertices of the parent cluster. The ∘ are the boundary vertices of children clusters that did not become the boundary vertices of the parent. The dashed line presents the cluster path of the parent cluster. Moreover, there exist symmetric variants for (b) and (c).

2.2 Supported Operations

When we work with a forest (a set of trees) then each tree of the forest is represented by one top tree. Given vertices v and w, the whole forest is controlled by following operations:

$link(v, w)$ – If v and w occur in different trees then a new edge (v, w) is created.
$cut(v, w)$ – If the graph contains the edge (v, w) then the edge is removed.
$expose(v, w)$ – If v and w occur in the same tree then the top tree is rebuilt so that v and w become the external boundaries and the path $v \dots w$ becomes the root path (Fig. 1a, 1b). It is possible to call expose with one vertex to create the root cluster with only one external boundary (Fig. 1c, 1d).
$select(R)$ – A special operation for nonlocal algorithms. The R is a top tree. This operation is described in the Section 2.4.

A number of clusters can be changed by the operations above. These changes are executed by *local operations*:

$create() = e$ – Creates a top tree with only one cluster e consisted of one edge.
$join(A, B) = C$ – A and B are the neighbouring root clusters of top trees R_A and R_B. This operation creates a new cluster $C = A \cup B$ which represents a root cluster of the top tree R_C consisted of R_A and R_B.
$split(C) = (A, B)$ – The C is the root cluster of a top tree R_C. The *split* removes C and divides the top tree R_C into top trees R_A and R_B. The clusters A and B are created from the C as the root clusters of the new top trees.
$destroy()$ – This operation removes the cluster represented by one edge.

The operations *link*, *cut* and *expose* invoke a sequence of the local operations. Each local operation always changes just one cluster. The root cluster may represent only a part of the tree T during the rebuilding, but in the end it represents whole T again.

2.3 Usage of Top Trees

Algorithms working over top trees associate information that they need with clusters or vertices. The local operations describe how to deal with the information during the changes. Then to describe the algorithm it is enough to define the manner of the local operations.

To keep the structure in a consistent state it is possible to change information only in the root cluster or in the external boundary vertex. So the top tree must be rebuilt by the *expose* operation when we want to change information saved in the cluster specified by a path $v \dots w$ or by a vertex v. Then the path becomes the root path and specified vertices become external boundary vertices. The structure does not allow to change any information outside of the root cluster.

2.4 Types of Algorithms

Tree algorithms can be classified as local or nonlocal. The *local algorithms* deal with local properties of the tree. The local property is defined in the following way: if the property holds for an edge or a vertex in the whole tree T then it holds for the same edge or the vertex in any subtree of T. An example can be the searching for the heaviest edge or the finding out the distance between two vertices. The local properties can be computed easily in bottom-up manner in the top tree. The *nonlocal algorithms* are different. They deal with nonlocal properties: this kind of property can be held by one edge (or a vertex) in the whole tree and by another one in any subtree. For example the searching for the center or the median of the tree. Evidently, the using of top trees for nonlocal algorithms is more difficult than for local algorithms.

Top trees have to support a fourth operation called *select* to enable nonlocal algorithms running. This operation is very complex so only basic idea is showed here. The detailed description can be found in [6]. The *select* gets a top tree R and it picks one of the children clusters of the root cluster. It is important to realize that the R represents the whole tree T. The operation works recursively and it is similar to the binary searching. One child of the root cluster is selected and this child becomes the root for next iteration. The top tree is rebuilt to prepare the new root after the selection (details in [6]). So the new root still represents the whole tree T. The finding finishes when a basic cluster that represents one edge is found. This cluster is the intersection of all selected clusters.

3 Used Implementation

We have developed an implementation of top trees in our previous work [7]. Our implementation results from Tarjan and Werneck's work [5, 8]. It combines tree decomposition used in Sleator and Tarjan's *ST-trees* [4] with compression and raking used in Frederickson's *topology trees* [2]. These principles ensure the representation of a tree graph T and bring the great idea of clustering. For more details see [7].

In our implementation, all four operations are supported in $O(\log n)$ amortized time where n is a number of vertices. This time analysis is proved in [5] for *link*, *cut* and *expose*. For *select* it is proved in [7].

The data structure and both languages TFL and TQL were developed in Java language. Source codes with examples and documentation are available on author's homepage[1].

4 Top Tree Friendly Language (TFL)

In our previous work [7], we have implemented top trees, but the usage still requires a detail knowledge of the structure which is quite complex. So we have decided to develop a declarative programming language that makes the structure available to users with only a basic knowledge.

The declarativity saves users from technical details of the implementation and it allows to focus on the designing of a tree algorithm. So hence it came the name of the language - Top Tree Friendly Language (for short *TFL*).

We did not try to develop the almighty programming language. The aim was a language that simplifies the designing of algorithms. The TFL allows to write a simple and short source code quickly and then easily verify its functionality with the TQL (Sec. 5).

4.1 Basics of TFL

The source code of the language consists of several blocks that form two sections. In the first section, there is declared a name of the algorithm and information stored in vertices and clusters. A description of the designed algorithm occurs in the second section of the script. A line or block comment can be used anywhere.

In the declarative section, it is necessary to keep the order of three obligatory blocks - firstly the algorithm name in the *algorithm* block, then the declaration of the vertex in the *vertex* block and finally the declaration of the cluster in the *cluster* block. An example can be seen in Fig. 2.

The declaration of the vertex and the cluster contains a specification of variables to store information. The variables are described by a data type. All supported types are depicted in the section 4.3. In the *cluster* block, there it is possible to use a special array. The array holds information of the specified data type for two keys - the boundary vertices of the relevant cluster. The *cluster* is the only block where the arrays can be declared. This construction resulted from the need of some algorithms that require to hold different information for both boundary vertices in each cluster.

In the second section, there is described a behaviour of the algorithm. Two levels of the blocks are used here whereas the only one level occurs in the declarative section. The root blocks form the first level. They are *var*, *create*, *destroy*, *join*, *split* and *selectQuestion*. These blocks are not obligatory, but their occurrence has to correspond to the mentioned order. They describe the algorithm

[1] http://siret.ms.mff.cuni.cz/vajbar

```
/* The declaration of the algorithm name and saved information: */
algorithm { algorithmName } // 1. the name of the algorithm
vertex { // 2. the declaration of vertex information
  type1 variable1, variable2;
  type2 variable3; }
cluster { // 3. the declaration of cluster information
  type3 variable4;
  array(type2) variable5, variable6; }

/* The description of the algorithm: */
...
```

Fig. 2. The structure of the TFL source code.

during the local operations as their names imply. In the root blocks, there can be used only the blocks of the second level called auxiliary blocks. The *var* and the *selectQuestion* are exceptions without the local blocks.

The auxiliary blocks contain the procedural tools of the TQL language that are depicted in the section 4.4. They can be read as the sequences of the commands that are executed if some condition holds (e.g. the cluster represents a path). The blocks are explained in more detail in the following subsection.

4.2 Algorithm Description

The conception of the blocks corresponds to the train of thought during the algorithm designing. Thanks to blocks the process of the designing is divided into the smaller problems which can be solved more simply. That brings a declarative way of the programming, so an user takes care of the designing only. There is no need to take care of the technique how the result is achieved. Moreover, only basic procedural tools are needed to describe the work of the algorithm in the blocks.

The root block *var*. The *var* is a special kind of the root block. There are declared global variables that can be used in any other root block. The syntax of the declaration is the same as in the *vertex* block. Arrays are not allowed there.

The root blocks *create* and *destroy*. These blocks describe the algorithm during the local operations of the corresponding names. It is necessary to distinguish if the cluster represents a path or if it is a point cluster. So there are two auxiliary blocks:

`path` – the current base cluster is a *path cluster*
`point` – the current base cluster is a *point cluster*

A content of the block is executed if the condition above holds. These auxiliary blocks can occur in any order but each at most once.

The root blocks *join* and *split*. The blocks describe the algorithm during the appropriate local operations. There have to be considered the types of the parent and both children clusters. According to these types the behaviour of the algorithm can be described by following auxiliary blocks:

`path_child` – current descendant of the parent cluster is a *path cluster*
`point_child` – current descendant of the parent cluster is a *point cluster*
`path_parent` – parent cluster is a *path cluster*
`point_parent` – parent cluster is a *point cluster*
`path_and_path` – clusters represent the variant Fig.1a
`path_and_point` – clusters represent the variant Fig.1b
`point_and_path` – clusters represent a symmetry to the variant Fig.1b
`point_and_point` – clusters represent the variant Fig.1e
`lpoint_over_rpoint` – clusters represent the variant Fig.1c
`rpoint_over_lpoint` – clusters rep. a symmetry to the variant Fig.1c
`lpoint_and_rpoint` – clusters represent the variant Fig.1d

A content of the block is executed if the condition above holds. These auxiliary blocks form three groups: **_child*, **_parent* and the variants from the figure 1. In the *join* block, the groups have to occur in the mentioned order. In the *split*, the order of **_child* and **_parent* is switched. Within the scope of the groups the blocks can occur in any order.

The mentioned orders correspond to a succession that should be observed during the algorithm designing. When two neighbours are connected by the *join* then typically the data from them are prepared at first (**_child*) and then the data of the parent cluster are computed (**_parent*). If the algorithm needs more information about the clusters then the blocks from third group can be used. The procedure is analogical for the *split*. Firstly the data from the parent cluster are prepared (**_parent*) and then the data for the children are computed (**_child*). Eventually, the blocks from the third group can be used of course.

The root block *selectQuestion*. The last of the root blocks describes the way how the decision during one step of the *select* operation proceeds. The *selectQuestion* can be seen as the fifth local operation. Its content is slightly different from the other root blocks. There are no auxiliary blocks and the syntax follows:

```
selectQuestion {
  /* any code */
  if (condition) {
    /* any code */
    select a;
  }
  else {
    /* any code */
    select b;
  }
}
```

The command `select (a|b);` specifies the cluster that is selected. The *a* (*b*) denotes the left (right) child. The usage of the cluster names can be found in Section 4.5.

4.3 Supported Data Types

The language supports four data types - *integer*, *real*, *string* and *boolean*. It should be sufficient for the most of tree algorithms. The *string* represents the sequences of characters. The values must be surrounded by the quotation marks. The *boolean* type was included to allow working with logical values `true` and `false`.

The integral numbers are represented by the *integer* data type and the real numbers by the *real*. The language does not enable a casting between these two data types. The both numeral types support positive and negative infinity - `IpINF` and `ImINF` for the *integer* and `RpINF` and `RmINF` for the *real*. This differentiation between the types of the infinities results from the using of the auxiliary variables. It is described in the Section 4.5. The work with the infinities abides by the standard IEEE 754 [3].

The variables declared in the vertices and the clusters are initialized by the default values according to the data types. A null string `""` is the default for the *string* and the `false` for the *boolean*. The *integer* is initialized by integral zero `0` and the *real* by decimal zero `0.0`.

4.4 Constructions of Programming Language

By the procedural tools we try to cover usual needs of the algorithm designing. We have implemented a lot of tree algorithms over the top trees and it has revealed that there are only two essential commands. The first is the assignment of a value into a variable and the second is *if-else* construction. It seems to be very little, but it is not. There is no need to support more complicated constructions like *for*-loops or *while*-loops and the need of *switch*-block can be substituted by the *if-else* very easily.

The syntax of the *if-else* is the same as in other programming languages. The construction can contain any number of *elseif* parts and the *else* part is not necessary:

```
if (condition1) {
  /* any code */
} elseif (condition2) {
  /* any code */
} else {
  /* any code */
}
```

The language TFL supports basic arithmetic operations `+`, `-`, `*`, `/`. The division is integral for the *integer*. Numbers can be compared by `==`, `!=`, `<`, `>`, `<=` and `>=`. The *integer* and the *real* type cannot be combined anywhere. There is an operator `&` for *string* concatenation. The language also offers the logical AND `&&` and the OR `||`. The priorities of the operators are ordinary and the order can be specified by parentheses `(, )`. The summary of operators with their priorities:

Priority	Operator type	Operator
1	unary minus	`-`
2	multiplication, division	`* /`
3	enumeration, subtraction, concatenation	`+ - &`
4	comparison	`== != < > <= >=`
5	logical operators	`&& ||`
6	assignments	`= += -= *= /= &=`

4.5 Using of Variables

There are two kinds of variables in the TFL. The first kind includes the variables declared in the *vertex* and the *cluster* block. The second kind are auxiliary variables. The names have to match the regular expression `[a-zA-Z][a-zA-Z0-9_]*`. To make accessing to clusters and vertices easier the labels were set up:

`c`	current cluster
`a`	left child of the current cluster
`b`	right child of the current cluster
`child`	mark for `a` and `b` in **_child* blocks
`left`	left boundary vertex
`right`	right boundary vertex
`common`	common vertex of the children
`border`	the only boundary vertex of the point cluster

The usage of the names employs dot notation similarly to the object-oriented programming languages:

`variable`	global (declared in *var*) or auxiliary variable
`cluster.variable`	auxiliary variable or declared in *cluster*
`cluster.array[vertex]`	value attached to vertex in cluster array
`cluster.vertex.variable`	auxiliary or declared variable of the vertex
`vertex.variable`	abbreviation for `c.vertex.variable`

Auxiliary variables. In the root block (excepting the *var*), the auxiliary variables can be used. They do not need to be declared. The TFL learns the type from the first value that is assigned to the variable. Therefore each auxiliary variable must be initialized by some value. This is the reason for using one type of the infinity for the *integer* and another one for the *real*. The creation of auxiliary arrays is not allowed.

4.6 Example of Usage

The usage of the language is very easy as can be seen from the following example - the finding out the length of a path. The path is specified by two vertices and the technique was described by Alstrup et al. [1]. Each cluster holds its length. The algorithm needs to implement only the *join*. If a point cluster is created then its length is zero. The length of a path cluster is computed as the sum of the children lengths. The produced source code is very short and simple:

```
/****** The length of the path ******/

/*** Declaration of stored information ***/
algorithm {lengthOfWay}    // name of the algorithm
vertex { integer name; }   // we store name of the vertex in vertices
cluster { integer length; } // we store length of the cluster in clusters

/*** Description of the algorithm ***/
join {
  path_child { // if a or b is the path, then we remember its length
    child.l = child.length;
  } point_child { // if a or b is the point, then its length is zero
    child.l = 0;
  } path_parent { // path cluster length is the sum of its children
    c.length = a.l + b.l;
  } point_parent { // point cluster length is zero
    c.length = 0;
  }
}
```

5 Top Tree Query Language (TQL)

The TQL is a simple query language that was developed to control top trees. The language allows the adding of the vertices, joining and splitting of the edges or working with information stored in the top trees. There are supported the same data types as in the TFL.

To enable an easy manipulation with nodes of the forest one statement stored in the nodes must be unique. The user can choose that unique identifier which is preferred. This identifier cannot be the *boolean*, because it can determine only two values.

5.1 Adding of Nodes

When a new node is created then some values are saved into the statements declared in the *vertex* block of the TFL. To make it simpler there is a *node* command:

```
node (param1, param2, ...) (param3=value1, param4=value2, ...);
node;
```

So there can be declared the order of some parameters and default values of another parameters. The order of declarative and definition part is arbitrary and it is not necessary to use both of them. The default values have to match the data types and the unique identifier cannot be used. It is possible to call this command at any time. When the *node* is called without the parameters then it displays the current order and the default values settings.

A new node is created in the following manner:

```
unique_name (value1, value2, ...) (param1=value3, param2=value4, ...);
```

The enumeration part must agree with the declaration of the *node*. In the assignment part, the default values can be overwritten and other parameters defined. The parts can occur in any order. Any part can be omitted, but at least one has to be applied.

5.2 Joining and Splitting of Edges

Analogous to the nodes there is a command to specify the order of parameters and the settings of default values for edges that were declared in the *cluster* block of TFL:

```
edge (param1, ...) (param2=val1, param3[L]=val2, param3[R]=val3, ...);
edge;
```

The usage is the same as for the *node*, but moreover there can be used an array. To access the entries of the array there are marks `L` and `R` for the left and the right boundary vertex.

The command for the edge joining enables to specify the position of the new edge with regard to the position of other edges:

```
u {a} -- v {b} (value1, value2, ...) (param1=value3, ...);
```

When edges are organized in circular order around u and v, then the new edge (u, v) is the right successor of the (u, a) and the (v, b). This position specification can be omitted. For the enumeration and the assignment parts there are the same rules as for the nodes creation. A node can be created during the joining - all its parameters have to be set by default values and no position specification can be used.

The removing of the edges is very easy:

```
u ## v;
```

5.3 Reading Information and Another Commands

Sometimes it is important to read the information stored in the vertex v or in the cluster with boundary vertices u, v:

```
info(v);
info(u,v);
```

In the TQL language, there is a lot of embedded functions that were prepared for the most frequently used operations (e.g. value changing, ...). In addition, the language enables to program custom functions as a plug-in in the Java language. The details can be seen in our previous work [6].

6 Final Remarks

We have developed a complex solution that allows using top trees for easy programming of tree algorithms. The solution is built on our implementation of top trees [7] and it is formed by Top Tree Friendly Language (TFL) and Top Tree Query Language (TQL).

The TFL is a declarative programming language that makes the structure available to users with only a basic knowledge of top trees. The declarativity saves users from technical details of the implementation and it allows to focus on the designing of tree algorithms. The TQL is a simple query language that was developed to control top trees. The language allows the adding of the vertices, joining and splitting of the edges or working with information stored in the top trees. In addition, the language TQL can be extended by custom functions.

Both languages form an useful tool that enables to write a simple and short source code quickly and then easily verify its functionality. So the tool makes the algorithm designing easier and more comfortable.

6.1 Related Work

To the best of our knowledge, there exists no similar tool to compare with. Top trees are relatively young data structure. Excepting the work of Alstrup et al. [1] and Tarjan and Werneck's works [5, 8], we did not find any other information about top trees.

References

1. S. Alstrup, J. Holm, K. D. Lichtenberg, and M. Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Trans. Algorithms*, 1(2):243–264, 2005.
2. G. N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal of Computing*, 14(4):781–798, 1985.
3. IEEE-Task-P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic.* IEEE, New York, aug 12 1985. A preliminary draft was published in the January 1980 issue of IEEE Computer, together with several companion articles. Available from the IEEE Service Center, Piscataway, NJ, USA.
4. D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.
5. R. E. Tarjan and R. F. Werneck. Self-adjusting top trees. In *SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 813–822, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.
6. M. Vajbar. Modelling dynamic trees. Master's thesis, Department of Software Engineering, Charles University in Prague, 2008.
7. M. Vajbar and K. Toman. Implementation of self-adjusting top trees. Technical Report No 2008/3, Charles University, Prague, Czech Republic, 2008.
8. R. F. Werneck. *Design and analysis of data structures for dynamic trees.* PhD thesis, Princeton University, Princeton, NJ, USA, 2006. Adviser-Robert E. Tarjan.