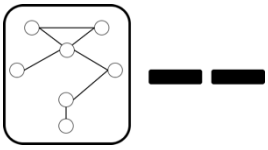
COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

# COMS 4115 Programming Language & Translator

# Project Proposal



TuSimple – An Easy Graph Language


Manager: Jihao Zhang(jz2791)

Language Guru: Zicheng Xu(zx2197)

System Architect: Shen Zhu(sz2609)

Tester 1: Ziyi Mu(zm2263)

Tester 2: Yunzi Chai(yc3228)

# TABLE OF CONTENTS

# 1. INTRODUCTION

The TuSimple language is a programming language which makes coding graphs as simple as drawing graphs on paper. It provides a more intuitive way of creating and manipulating graph. With the help of real-time rendering, construction and manipulation of graphs becomes really easy.

Another design principle is to simplify the expression of graph according to its mathematical definition. In other words, user of TuSimple would be able to implement graph algorithms directly from those pseudo code in textbooks(e.g. Introduction to Algorithms). By eliminating the details, user could be more focus on mathematical thoughts in essence, which would definitely improve the efficiency.

# 2. LANGUAGE FEATURES

## 2.1. SIMPLE GRAPH SYNTAX

Using TuSimple, you can create and modify every part of the graph in a simple sentence. Writing code in TuSimple is like painting on paper, every single operation is fluent and natural. You can pour out your ideas inside your brain without transformation.

## 2.2. GRAPH ALGORITHM

The design principle of TuSimple is to simplify the implementation of graph algorithms. With the help of built-in functions, you can write complicated algorithms in a few line of code. Different from using STL as blackbox, TuSimple show user all the internal details. It could accommodate to different conditions with small modification made by user.

## 2.3. GRAPH RENDERING

We developed this language in order to make graph drawing simpler, so TuSimple is capable of plotting graph. By using the plot function, users could draw their graphs easily. And TuSimple will handle the details of graph drawing and draw the graph using the NetworkX library.

# 3. LANGUAGE DESIGN AND SYNTAX

## 3.1. COMMENTS

| Syntax | Comment Style |
|---|---|
| /*<br>some comments<br>*/ | *Multiline comment* |
| // | *Single line comment* |

## 3.2. KEYWORDS

| Keyword | Definition |
|---|---|
| **int** | *Defines an integer* |
| **float** | *Defines a float* |
| **bool** | *Defines an expression which can only be true or false* |
| **string** | *Defines a sequence of characters* |
| **list** | *Defines a sequence of data in same type* |
| **set** | *Defines a set of data in same type, which can not be duplicated* |
| **node** | *Defines a point in the graph. Each has its own value and can be linked to other nodes.* |
| **map** | *Defines a hash table with provided values* |

| | |
|---|---|
| **graph** | *Defines a set of nodes* |
| **If else** | *Used as if (expression) {/*statements*/} else (expression) {/*statements*/}* |
| **for** | *Uses as for (initialization;termination;increment)* |
| **while** | *Used as while (expression) {/*statements*/}* |
| **continue** | *Used as a jump to next loop* |
| **break** | *Used as a break of current loop* |
| **return** | *Used as the end of function* |
| **NULL** | *Defines the value of non-existing value* |
| **print** | *Print the target value to the console* |

## 3.3. OPERATOR

| Name | Operator | int/float/bool | string | list/set/map | node |
|---|---|---|---|---|---|
| **PLUS** | + | add | Connect two string | Add new element | / |
| **MINUS** | - | substract | / | / | / |
| **MULTIPLY** | * | multiply | / | / | / |
| **DIVIDE** | / | divide | / | / | / |
| **ASSIGN** | = | Set the left variable with the value of right side | Set the left variable with the value of right side | Set the left variable with the value of right side | Set the left variable with the value of right side |
| **EQUAL** | == | Compare the value | Compare the string | Compare the string | Compare the string |
| **AND** | && | Calculate using boolean value | Calculate using boolean value | / | / |
| **OR** | \|\| | Calculate using boolean value | Calculate using boolean value | / | / |
| **NOT** | ! | Calculate using boolean value | Calculate using boolean value | / | / |
| **GT** | > | Compare the value | / | / | / |
| **LT** | < | Compare the value | / | / | / |
| **GE** | >= | Compare the value | / | / | / |

| | | | | | |
|---|---|---|---|---|---|
| **LE** | <= | Compare the value | / | / | / |
| **LINK** | -> | / | / | / | Link current node to another |
| **DI-LINK** | -- | / | / | / | Link both nodes to each other |
| **NEXT** | ++ | Add value by 1 | / | Move to the next element | / |
| **BRACE** | {} | / | / | Take value from set | / |
| **BRACKET** | [] | / | Take character with index | Take value from list | / |
| **PARENTH** | () | / | / | Take value from map | / |

## 3.4. BUILT-IN FUNCTIONS

| Public Function | | |
|---|---|---|
| **Name** | **Signature** | **Description** |
| max | min(T a, T b) | *Return the major value* |
| min | max(T a, T b) | *Return the minor value* |

| API of node(node a) | | |
|---|---|---|
| **Name** | **Signature** | **Description** |
| value | a.v | *The values of node* |
| begin | a | *Begin of the linked-node list* |

| API of set(set a(T)) | | |
|---|---|---|
| **Name** | **Signature** | **Description** |
| minimum | a.min | *Return the minimum value* |
| maximum | a.max | *Return the maximum value* |

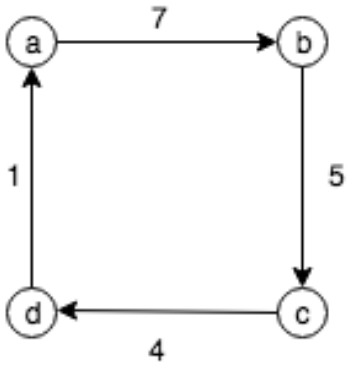
| API of map(map a(T1, T2)) | | |
|---|---|---|
| **Name** | **Signature** | **Description** |
| minimum | a.min | *Take the pair with the minimum value in the digital dimension* |
| maximum | a.max | *Take the pair with the maximum value in the digital dimension* |
| node | a.node | *Take the node values from a map* |
| value | a.v | *Take the digital values from a map* |
| fill | a.fill(T1)<br>a.fill(T2)<br>a.fill(T1, T2) | *Assign the nodes/values in every pair with given one* |
| delete | a.del(T1)<br>a.del(T2)<br>a.del(T3)<br>a.del(T1, T2) | *Delete the pair with target node/value* |
| delete all | a.del_all | *Delete all the data* |

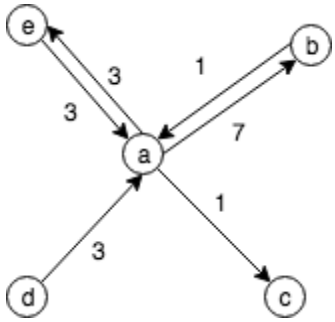
| API of graph(graph a(set b)) | | |
|---|---|---|
| **Name** | **Signature** | **Description** |
| plot | a.plot | *Draw the graph* |

# 4. CODE EXAMPLES:

## 4.1. DIJKSTRA'S ALGORITHM

```
// build the map

map distant(node,int);

node a,b,c,d;

a -> b = 7;

b -> c = 5;

c -> d = 4;

d -> a = 1;

distant += {a,b,c,d};

distant.fill(maxint);

// execute the algorithm

list queue(node);

map visited(node,bool);

node n;

queue += a;

while (queue!=NULL){
        visited[queue] = true;
        for (node i=queue;i!=NULL;i++){
                if (distant[i]==null || distant[i]<distant[queue]+queue->i){
                        distant[i] = distant[queue]+queue->i;
                }
        }
        queue++;
        queue += (distant.del(visited)).min.node;
}
```




## 4.2. MAX FLOW

// build the map

node a,b,c,d;

a -> b -> a = {7,1};

a -> {c, d} = {1,3};

a -- e = 3;

start = a;

end = d;

map distant(node,int);

// execute the algorithm

bool BFS(){

distant.del_all;

```
	list queue(node);
	queue += start;
	while (queue!=NULL){
		for (node i=queue;i!=NULL;i++){
			if (distant[i]==NULL){
				distant[i] = distant[queue]+1;
				queue += i;
			}
		}
		queue++;
	}
	return distant[end];
}
int find(node x, int lim){
	if (x==end) return lim;
	for (node i=x;i!=NULL;i++){
		if (x->i > 0 && distant[i]==distant[x]+1 && int tmp = find(i, min(lim, x->i))){
			x->i -= tmp;
			i->x += tmp;
			return tmp;
		}
	}
	return 0;
}
int ans = 0;
while (BFS()){
```

```
        while (flow=find(start,maxint)) ans += flow;
}
```