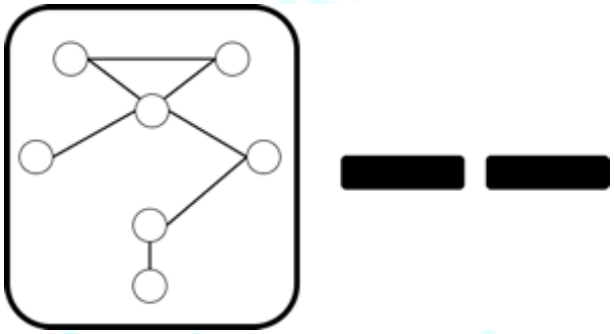
COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

COMS 4115 Programming Language & Translator

Language Reference Manual



TuSimple – An Easy Graph Language

Jihao Zhang(jz2791)

Zicheng Xu(zx2197)

Shen Zhu(sz2609)

Ziyi Mu(zm2263)

Yunzi Chai(yc3228)

TABLE OF CONTENTS

1. INTRODUCTION

This manual describes the TuSimple language.

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. TOKENS

There are five classes of tokens: identifiers, keywords, literals, operators, constants and delimiters . White spaces are ignored.

2.1. COMMENTS

The characters /* introduce a multi-line comment, which terminates with the characters */. Comments do not nest, and they do not occur within a string or character literals.

The characters // introduce a single-line comment.

2.2. IDENTIFIERS

An identifier is a sequence of letters and digits. The first character must be a letter; the underscore _ counts as a letter. Upper and lower case letters are different. Identifiers may have any length.

2.3. KEYWORDS

The following identifiers are reserved for the use as keywords, and may not be used otherwise:

```
int float bool string list set node map graph if else for while continue
break return NULL print TRUE FALSE
```

2.4. LITERALS

String Literals are a sequence of zero or more letters, spaces, digits, other ASCII characters numbers 32 to 126, excluding the single quote (number 39). These strings should be enclosed in single quotes.

Integer literals are a sequence of one or more digits. Only numbers in decimal format are recognized.

Float literals are a sequence of one or more digits and a decimal point with another sequence of one or more digits. Only numbers in decimal format are recognized.

2.5. DELIMITERS

2.5.1 PARENTHESES AND BRACES

Parentheses are used to force evaluation of parts of a program in a specific order. They are also used to enclose arguments for a function.

2.5.2 COMMAS

Commas are used to separate arguments in declaration and function arguments.

2.5.3 SQUARE BRACKETS

Square brackets are used for accessing of array elements.

2.5.4 SEMICOLON

Semicolons are used to terminate a sequence of code.

2.5.5 CURLY BRACES

Curly braces are used to enclose function definitions and for/while loops, as well as node declarations.

2.6 CONSTANTS

maxInt = 2147483647

minInt = -2147483638

3. DATA TYPES

3.1. INTEGERS

Possible value: 32-bit signed Integer (-2147483638 ~ 2147483647)

Example:

```
int i1 = 0;

int i2 = -123

int i3 = 43;
```

3.2. FLOATS

Possible value: A IEEE 754 double-precision (64-bit) numbers

Example:

```
float f1 = 0.356;

float f2 = 3.4e-16;

float f3 = 1;
```

3.3. BOOLEANS

Possible value:

```
bool b1 = true;

bool b2 = false;
```

3.4. STRINGS

Possible value: A sequence of ASCII enclosed by double quotes

Example:

```
string s1 = “I love PLT”;

string s2 = “” ;

string s3 = “Hello world!\n”;
```

3.5. LISTS

A list of homogeneous elements.

List should be declared in the format: list <name of list > (<type of the list element>);

Example:

```
list queue(node);
```

When using the name of the list alone, it actually a pointer points to the first element of the list.

When using with the ‘++’ operator, it means remove the first element from the list.

When using the ‘+=’ operator, it means add the element into the end of the list.

Example:

```
node a;

queue += a; // queue = {a}
```

3.6. SETS

A set of homogeneous elements, in which each element should be unique.

Set should be declared in the format: set <name of set > (<type of the set element>);

Example:

```
set visited(bool)
```

3.7. NODES

A node in a graph. Node can be declared separately or consecutively.

Example:

```
node a,b,c,d;
```

3.8. MAPS

A hashmap of nodes and its values.

Map should be declared in the format: map <name of map > (<type of the key, type of the value>);

Example:

```
map distant(node,int);
```

3.9. GRAPHS

A graph contains all nodes and edges.

Example:

```
graph g;
```

4. OPERATORS

Name	Operator	Int & float	Bool	string	list/set/map	node
PLUS	+	Add two values Example: 1 + 2 (evaluates 3) 3.15 + 5.20 (evaluates 8.35)	/	Connect two string Example: “I” + “love PLT” (evaluates “I love PLT”)	Add new element Example: List l Node n a + n (evaluates a list with the node n)	/
MINUS	-	Substract Example: 1 - 2 (evaluates -1) 3.15 - 5.20 (evaluates -2.05)	/	/	/	/
MULTIPLY	*	Multiply Example: 1 * 2 (evaluates 2) 3.15 * 1.0 (evaluates 3.15)	/	/	/	/

DIVIDE	/	Divide Example: 1 / 2 (evaluates 0) 3.15 / 1.0 (evaluates 3.15)	/	/	/	/

Add into	+=	Add the number to the exsisting variable.
		Example:
		int a = 0;
		a += 100;
		(a evaluates 100)
			/	Append to the exsisting string.
				S += str is equal to the expression s = s + str;
					Add the element in to the list/set/map	/

Minus from	-=	Likewise, minus number from a exsisting number. The same as C.	/	/	Delete the right hand side element from the list/set/map	//
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. Assigns the address.	Set the left variable with the value of right side	Set the node with the value of right side

EQUAL	==	Compare the value	Compare the value	Compare the string.	Compare the address.	Compare the address.
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 and return a directed edge.

						a->b is a directed edge from node a to node b, and can be assigned a value of the edge. a->b = 10
DI-LINK	--	/		/	/	Link both nodes to each other and return a undirected edge. a -- b is a undirected edge from node a to node b, and can be assigned a value of the edge. a--b = 10
NEXT	++	Add value by 1		/	For list, this operator moves to the next element and remove the first element out of the lsit.	/
BRACE	{}	/		/	Batch assignment.	/

					For example: a -> b -> a = {7,1}; means the edge from a to b's value is 7 and the edge from b to a values 1. a -> {c, d} = {1,3}; means the edge from a to c and d 's value is 1 and 3 respective.	
BRACKET	[]	/		Return character in the index. For example: stirng s = 'ILOVEPLT' s[1] = 'L'	Return the pointer to the respective element in list.	/
PARENTH	()	/		/	Return the pointer to the respective element in map.	/

5. 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*
print	print(string s)	*Print the target string*
int_to_string	int_to_string(3)	*type casting for int*
string_to_int	string_to_int('3')	*type casting for string*

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) a.fill(T2) a.fill(T1, T2)	*Assign the nodes/values in every pair with given one*
delete	a.del(T1) a.del(T2) a.del(T3) 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*

init_tag	a.init_tag(map m)	*Iterator for tagging each node in the given map, return the node which is currently tagging*
reduce	a.reduce(node n)	*Iterator for updating the value of connected nodes, select the smaller value, return the node which are currently updated*
expand	a.expand(node n)	*Iterator for updating the value of connected nodes, select the bigger value, return the node which are currently updated*
combine	a.combine(target condition, set s)	*Iterator for selecting target nodes to join a target set, return the result set*
component	a.component()	*Return the list of connected components of graph a*
bfs	a.bfs(node n, map m)	*Execute the BFS to the graph, record the result in map m, return the access sequence of nodes*
dfs	a.dfs(node n, map m)	*Execute the DFS to the graph, record the result in map m, return the access sequence of nodes*
find	a.find(node n) a.find(value b)	*Return the list of nodes matching the given condition*
find_path	a.find_path(node n1, node n2, target condition)	*Return the list of paths matching the given condition*
assign	a.assign(ch sign, value v)	*Update the value of every edge with given calculator*
reverse	a.reverse(graph b)	*Return the list of reverse edges in the graph a*

6. STATEMENTS

6.1 THE IF STATEMENT

```
// Connect two nodes if they are not connected

bool isConnected;

// Declareation of nodes

node a, b;

if (!isConnected) {

      a -> b = 1;

}
```

6.2 THE WHILE STATEMENT

```
// Add ten nodes to a queue

int count = 0;

list queue(node);

while (count < 10) {

      node tempNode;

      queue += tempNode;

}
```

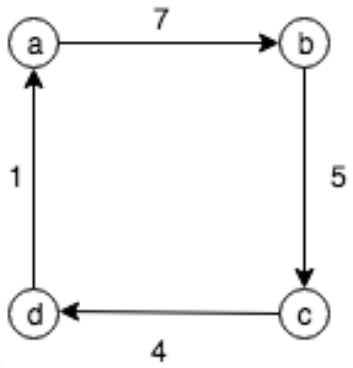

7. SAMPLE PROGRAMS:

7.1. DIJKSTRA'S ALGORITHM

```
// build the map

// declaration of a hashmap between node and integers.

map distant(node,int);
```

```
// declaration of nodes.

node a,b,c,d;

// assign values of an edge between a and b.

a -> b = 7;

b -> c = 5;

c -> d = 4;

d -> a = 1;

// add a,b,c,d four nodes into the hashmap, and then initialize their
value to the default integer 0.

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

// fill all the value of the nodes with the constant max integer.

distant.fill(maxint);

// execute the algorithm

//declaration of a list named queue.

list queue(node);

//declaration of a hashmap named visited between node and bool.


map visited(node,bool);

//declaration of a node n.

node n;

//add node “a” into the list “queue”

queue += a;

// when the list “queue” is not null, do the loop

while (queue!=NULL){

      // check if the fist element of the list
```

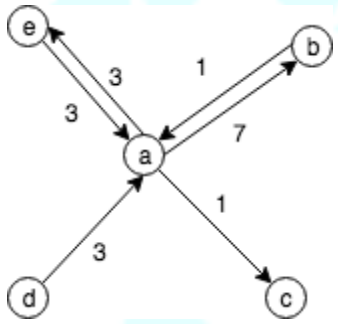
```
      visited[queue] = true;
      for (node i=queue;i!=NULL;i++){
            if (distant[i]==null || distant[i]<distant[queue]+queue->i){
                  distant[i] = distant[queue]+queue->i;
            }
      }
      queue++;
      queue += distant.del(visited).min.node;
}
```



Using the API of graph:

```
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);
```

```
map visited(node,bool);
graph a(distant);
node n;
while ((n=a.min.init_tag(visited))!=NULL){
      a.reduce(n);
}
```



7.2. 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);
distant += {a, b, c, d, e};
```

```
// 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;
}
```

Using the API of graph:

```
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);
distant += {a, b, c, d, e};
graph g(distant);
in flow = 0;
while (g.bfs(start).find(end)!=NULL){
// if we can find end node in the bfs process started from start node
      list gp = g.find_path(start, end, distant, 1);
```

```
// if we can find the path from start node to end node with a step of 1
in distant
      flow += gp.min;
// flow must be added by the minimum value of the whole path
      gp.assign('+', gp.min);
// for every edge in the path, add the value by this minimum value
      g.reverse(gp).assign('-', gp.min);
// for every reverse edge, subtract the value
}
```

7.3. HUNGARIAN

```
node a, b, c, d;
node x, y, z;
a->x = 1; a->y = 1; a->z = 1;
b->y = 1; c->x = 1; d->x = 1;
map distant(node, int);
distant += {a, b, c, d, x, y, z};
map visited(node, bool);
map match(node, node);
graph a(distant);
while ((n=a.init_tag(visited))!=NULL){
// if there exist a node which has not been visited
      node n1;
      match[n] = (n1=a.dfs(n, visited, true)[1]);
// update its match with the second node of dfs sequence
```

```
      match[n1] = n;
// update the second node correspondingly
}
```