TuSimple
An Easy Graph Language
The Team

Jihao Zhang  Manager
Zicheng Xu   Language Guru
Shen Zhu     System Architect
Ziyi Mu      Language Guru & Architect
Yunzi Chai   Tester
Overview

The TuSimple language is designed to make coding graphs as simple as drawing graphs on paper.

Problem

Graphs has important applications in networking, bioinformatics, software engineering, database and web design, machine learning, and other technical domains.

It's a pain to draw graphs and calculate graph algorithms by hand. It's messy, time consuming and usually results in wrong answers.

It’s also hard to programming graphs with programming languages like C/C++, Java, etc.
Overview

The TuSimple language is designed to make representing and calculating graphs as simple as possible.

Solution
Intuitive syntax to initialize graphs and graph components.

A lot of built-in functions to manipulate complex graphs.

User-friendly built-in containers.

Familiar syntax.
Project status

1922 lines of OCaml code
2486 lines of C code
277 git commits
70 test cases
2083 lines of test code
Architecture

- Source Code (input.tu)
- Scanner
- Parser
- Code generation
- Semantic Checker
- input.ll file
Architecture

input.ll

Assembler

input.s

External Libraries(Set, Hashmap, etc.)

utils.o

Linker

executable file

./compile.sh <code>
int main(){
    node@{int} node1, node2, node3;
    list@{node@{int}} lst; map@{int, node@{int}} mp;
    set@{string} s; graph gp;
    new s; new node1; new node2; new node3; new node4;
    new lst; new gp; new s; new mp;
    node1 -> node2 = 2;
    node2 -> @{node1, node3, node4} = @{2, 4, 8};
    node1.setValue(1);
    lst += @{node1, node2};
    lst++;
    node1 = lst[0];
    s += @{“tusimple”, “is”, “so”, “great”};
}
Language Features

Operators
+ - * / %
+= -= =
== && || !
>= <=
-> --
++

Type and Containers
int
float
string
graph
bool
node@{type}
list@{type}
map@{type, type}
set@{type}

Comments
// this is a comment
/*
so does this
*/
Built-in Functions

Node
- value()
- name()
- length()
- setValue(value)
- iterNode(pos)
- weightIter(pos)

List
- get(pos)
- pop()
- length()
- remove(pos)
- concat(anotherList)
- printList()

Map
- get(key)
- put(key, value)
- size()
- hasKey(key)
- remove(key)

Set
- put(element)
- length()
- contain(element)
- remove(element)

Graph
- bfs(startingNode)
- dfs(startingNode)
- relax()
- expand()
- combine(anotherGraph)
- iterGraph(pos)
- init()
- addNode(node)
- addEdge(node, node, weight)
- printGraph()
Graph depth-first search

```java
node<int> s, a, b, c, d;
graph g;
list<int> lst;
new s; new a; new b; new c; new d; new lst;
s -- {a , b , c} = {1, 1, 1};
d -- {a , b, c} = {1, 1, 1};
lst = g.dfs(s);
lst.printList();
// output: s a d b c
```
Graph relaxation

In shortest path algorithms (Bellman-Ford, Dijkstra's), relaxation is an important operation.

**Edge relaxation.** To relax an edge v→w means to test whether the best known way from s to w is to go from s to v, then take the edge from v to w, and, if so, update our data structures.

**Vertex relaxation.** Relax all the edges pointing from a given vertex.

```java
private void relax(EdgeWeightedDigraph G, int v) {
    for (DirectedEdge e : G.adj(v)) {
        int w = e.to();
        if (distTo[w] > distTo[v] + e.weight()) {
            distTo[w] = distTo[v] + e.weight();
            edgeTo[w] = e;
        }
    }
}
```
Automated tests

We started from MicroC, then added a test for each feature we add.

70 test cases, 26 for should-fail, 44 for should-pass

Use shell script to automate the process

Verifies all the test cases are passed before committing
Automated tests

test-Dijkstra1...SUCCESS
test-arithmetic-add...SUCCESS
test-arithmetic-sub...SUCCESS
test-arithmetic-mul...SUCCESS
test-arithmetic-division...SUCCESS
test-bfs1...SUCCESS
test-bfs2...SUCCESS
test-bfs3...SUCCESS

test-dfs1...SUCCESS

test-for1...SUCCESS

test-for2...SUCCESS

test-for3...SUCCESS

test-fun1...SUCCESS


test-graph1...SUCCESS

test-graph2...SUCCESS

test-if1...SUCCESS

test-if2...SUCCESS

test-int...SUCCESS


test-map1...SUCCESS

test-map2...SUCCESS

test-map3...SUCCESS

test-map4...SUCCESS

test-node1...SUCCESS

test-node2...SUCCESS

test-node3...SUCCESS


test-set1...SUCCESS

test-set2...SUCCESS

test-set3...SUCCESS


test-node5...SUCCESS

test-node6...SUCCESS

test-op1...SUCCESS

test-op2...SUCCESS

test-op3...SUCCESS

test-op4...SUCCESS


test-test1...SUCCESS

test-test2...SUCCESS


test-while1...SUCCESS

test-while2...SUCCESS


test-list1...SUCCESS

test-list2...SUCCESS

test-list3...SUCCESS

test-list4...SUCCESS


test-map1...SUCCESS

test-map2...SUCCESS


test-op-AND1...SUCCESS

test-op-AND2...SUCCESS

test-op-OR1...SUCCESS

test-op-OR2...SUCCESS


test-undefined-fun...SUCCESS
Demo

Breadth-first search (BFS)
Shortest path
Neural network training
Demo

Breadth-first search (BFS) from node 1

BFS result: node1 node2 node3 node5 node4
Demo

Single source shortest path from node 1

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Demo Neural networks training XOR function

single hidden layer with three neurons

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 1)</td>
<td>1</td>
</tr>
<tr>
<td>(0, 0)</td>
<td>0</td>
</tr>
<tr>
<td>(1, 0)</td>
<td>1</td>
</tr>
<tr>
<td>(0, 1)</td>
<td>1</td>
</tr>
</tbody>
</table>
Thank you
Questions?

Special Thanks to Julie, our TA, who continuously support our project.