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

TuSimple

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, sof tware 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 l ike C/C++, Java, etc.

TuSimple

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

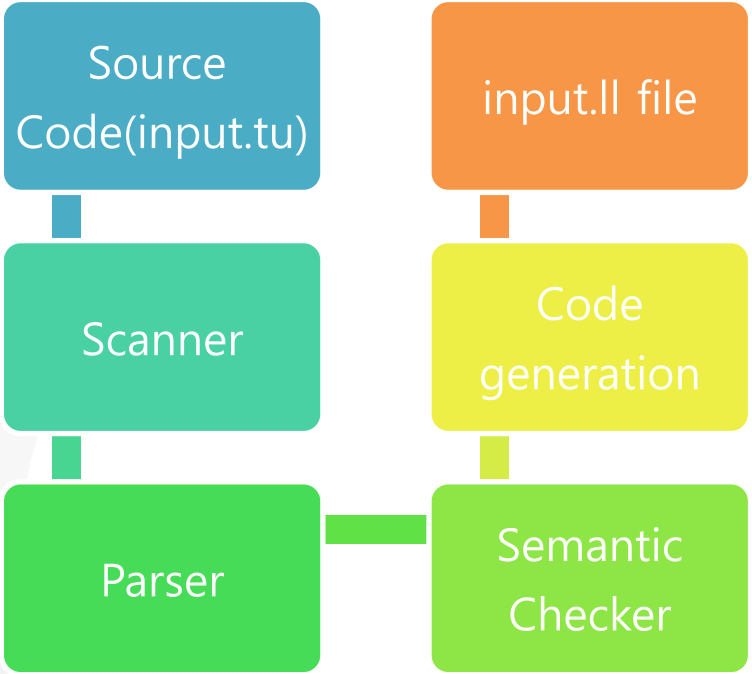
2486 lines of C code

277 git commits

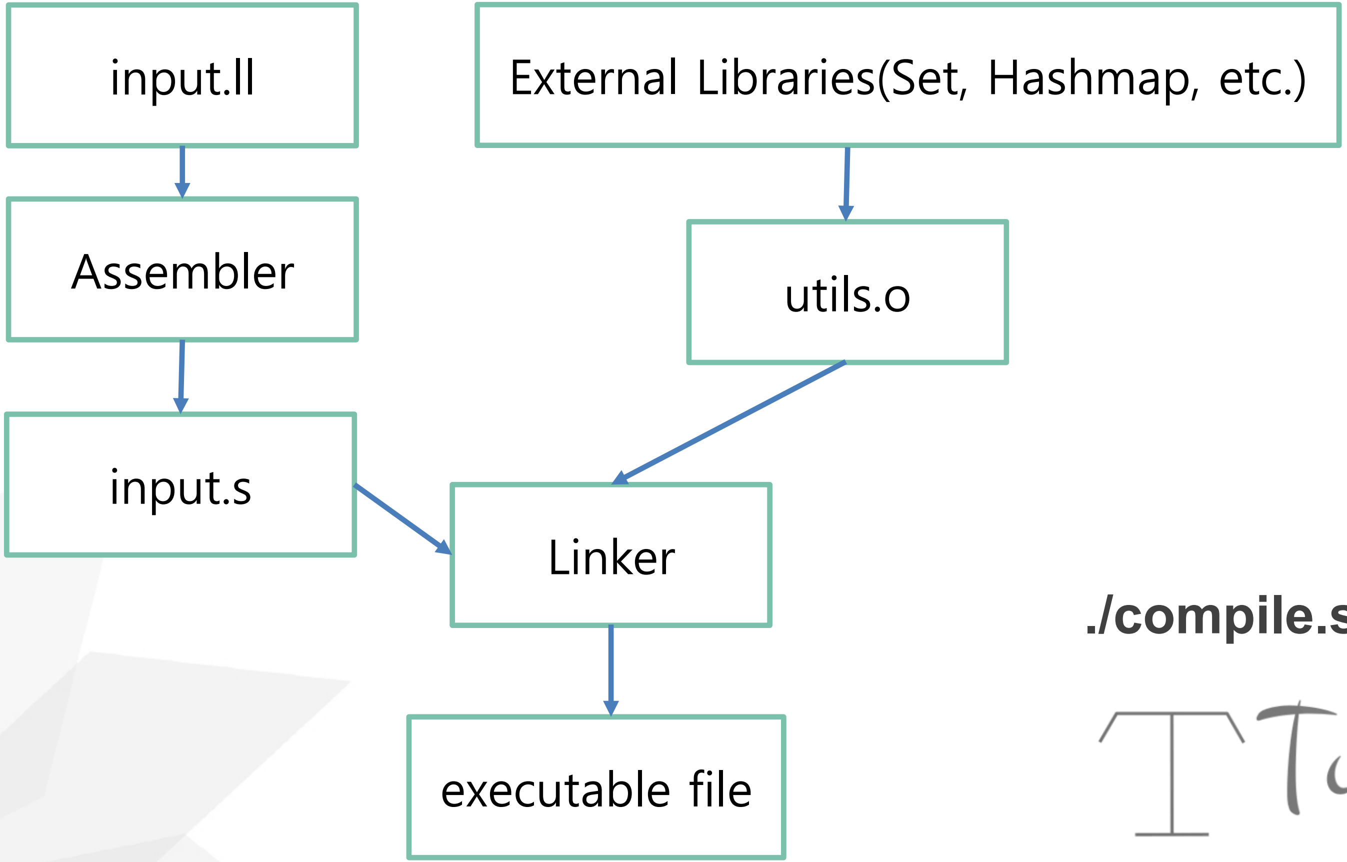
70 test cases

2083 lines of test code

Architecture



Architecture



./compile.sh <code>

A Sneak Peak - Syntax

TuSimple

Declare container types

```
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"};
}
```

Initialize containers and graph components

Connect nodes and initialize edge weights.

Overload operators

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
*/
```

TuSimple

Built-in Functions

Node
value()
name()
length()
setvalue(value)
iterNode(pos)
weightIter(pos)

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

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

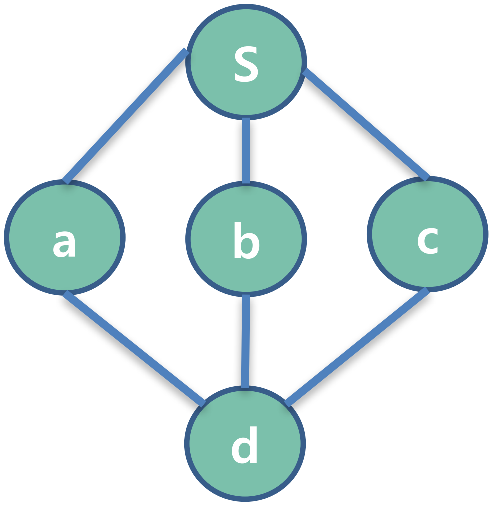
Map
get(key)
put(key, value)
size()
haskey(key)
remove(key)

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

TuSimple

Graph depth-first search

```
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

```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;
        }
    }
}
```

TuSimple

g.relax(v)

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-arith1-add...SUCCESS
test-arith2-sub...SUCCESS
test-arith3-mult...SUCCESS
test-arith4-division...SUCCESS
test-bfs1...SUCCESS
test-bfs2...SUCCESS
test-bool...SUCCESS
test-data1...SUCCESS
test-dfs1...SUCCESS
test-dfs2...SUCCESS
test-for1...SUCCESS
test-for2...SUCCESS
test-for3...SUCCESS
test-fun1...SUCCESS
test-fun2...SUCCESS
test-graph1...SUCCESS
test-graph2...SUCCESS
test-if1...SUCCESS
test-if2...SUCCESS
test-int...SUCCESS
test-list1...SUCCESS
test-list2...SUCCESS
test-list3...SUCCESS
test-list4...SUCCESS
test-list5...SUCCESS
test-map1...SUCCESS
test-map2...SUCCESS
test-map3...SUCCESS
test-map4...SUCCESS
test-node1...SUCCESS
test-node2...SUCCESS
test-node3...SUCCESS
test-node4...SUCCESS
test-node5...SUCCESS
```

```
test-node6...SUCCESS
test-ops1...SUCCESS
test-ops2...SUCCESS
test-set1...SUCCESS
test-set2...SUCCESS
test-set3...SUCCESS
test-set4...SUCCESS
test-while1...SUCCESS
test-while2...SUCCESS
fail-add1...SUCCESS
fail-fun1...SUCCESS
fail-fun2...SUCCESS
fail-fun3...SUCCESS
fail-fun4...SUCCESS
fail-list1...SUCCESS
fail-list2...SUCCESS
fail-list3...SUCCESS
fail-list4...SUCCESS
fail-map1...SUCCESS
fail-map2...SUCCESS
fail-node1...SUCCESS
fail-node2...SUCCESS
fail-op-and1...SUCCESS
fail-op-and2...SUCCESS
fail-op-not1...SUCCESS
fail-op-not2...SUCCESS
fail-op-or1...SUCCESS
fail-op-or2...SUCCESS
fail-set1...SUCCESS
fail-set2...SUCCESS
fail-set3...SUCCESS
fail-set4...SUCCESS
fail-set5...SUCCESS
fail-test...SUCCESS
fail-undeclaredfun...SUCCESS
```

Demo

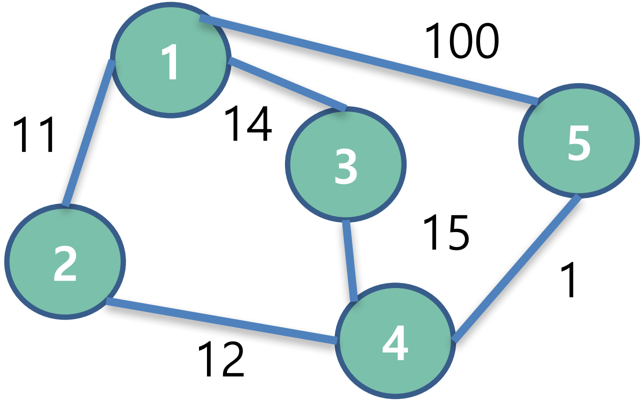
Breadth-first search (BFS)

Shortest path

Neural network training

Demo

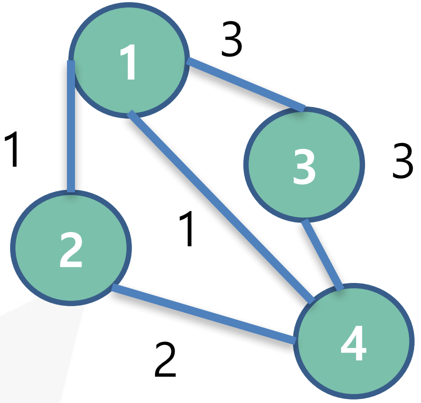
Breadth-first search (BFS) from node 1



BFS result: node1 node2 node3 node5 node4

Demo

Single source shortest path from node 1

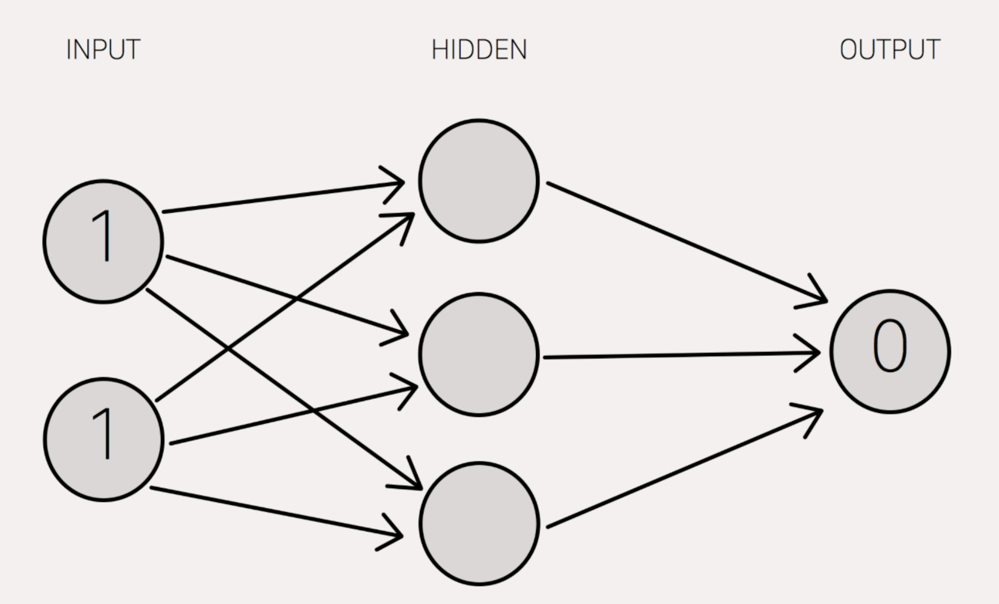


1	**0**
2	1
3	3
4	1

TuSimple

Demo Neural networks training XOR function

single hidden layer with three neurons



Input	Output
(1, 1)	1
(0, 0)	0
(1, 0)	1
(0, 1)	1

TuSimple

Thank you
Questions?

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

TuSimple