Hippograph

The Language for High Performance Parsing of Graphs

Ben Lewinter
bsl2121
Manager

Irina Mateescu
im2441
Language Guru

Harry Smith
hs3061
System Architect

Yasunari Watanabe
yw3239
Test Expert

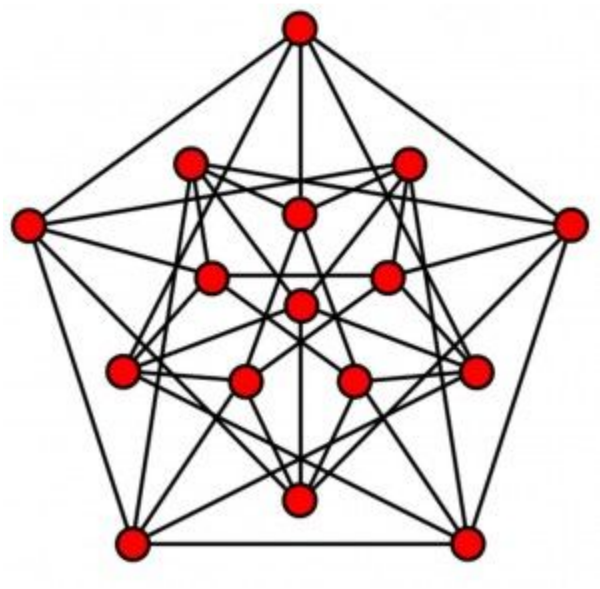
Motivation

A Language for Graphs

Graph theory is an important field in computer science, with wide ranging applications

We thought there should be a language that made experimenting with and utilizing graphs easier!

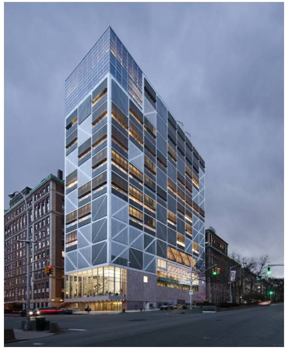
giraph from Fall 2017 was a major inspiration for us, but we had some ideas for what could be added...



Goals

1. **Unified graph type** - generic graph type that can handle any type of edge
2. **Customizable node names** - giving the user greater control over their graphs
3. **Cypher-like query capabilities** - especially helpful when using graph to store large amounts of data
4. **Anonymous functions** - for passing in user-defined graph operations
5. **Search Strategy Type** - specifying traversal method in graph iteration

Workflow and Team Processes



The end result

scanner.mll	70 lines
parser.mly	160
ast.ml	178
sast.ml	109
semant.ml	446
codegen.ml	823
graph.c	1,152
hippograph.ml	29

Plus

197 Test Scripts

156 Git Commits

2 Pies of Pizza

Language Overview

The Basics

- Operators:
  - `+ - * / ; = . > < => <= == and or not`
- Control Flow:
  - `While (true) {make_graphs();}`
  - `For (int i = 0; i <= 10; i = i + 1)`
  - `If (you_dont_mind()) { do_it(); } else { dont_bother(); }`
    - `The ELSE clause is optional!`
- Primitive Types:
  - `int, bool, string`
- Comments:
  - `(* don't run me! *)`

Function Flavors

The Standard:

```
return_type func_name(type1 arg1; type2 arg2; … ) {
                        …
}
```

The Condensed:

```
fun<type1:type2: … :typek, ret_typ> f = ret_type (type1 … )( expr )
```

The Condensed Function

- Allow declarations of functions within the bodies of other functions
  - Stored in variables, which effectively provide the names of anonymous functions
  - Fall in and out of scope with the function!

- Implemented as expressions which resolve to a `FUN` type

- Passing functions as first class data: WIP.

What about graphs?

- Node Expressions:

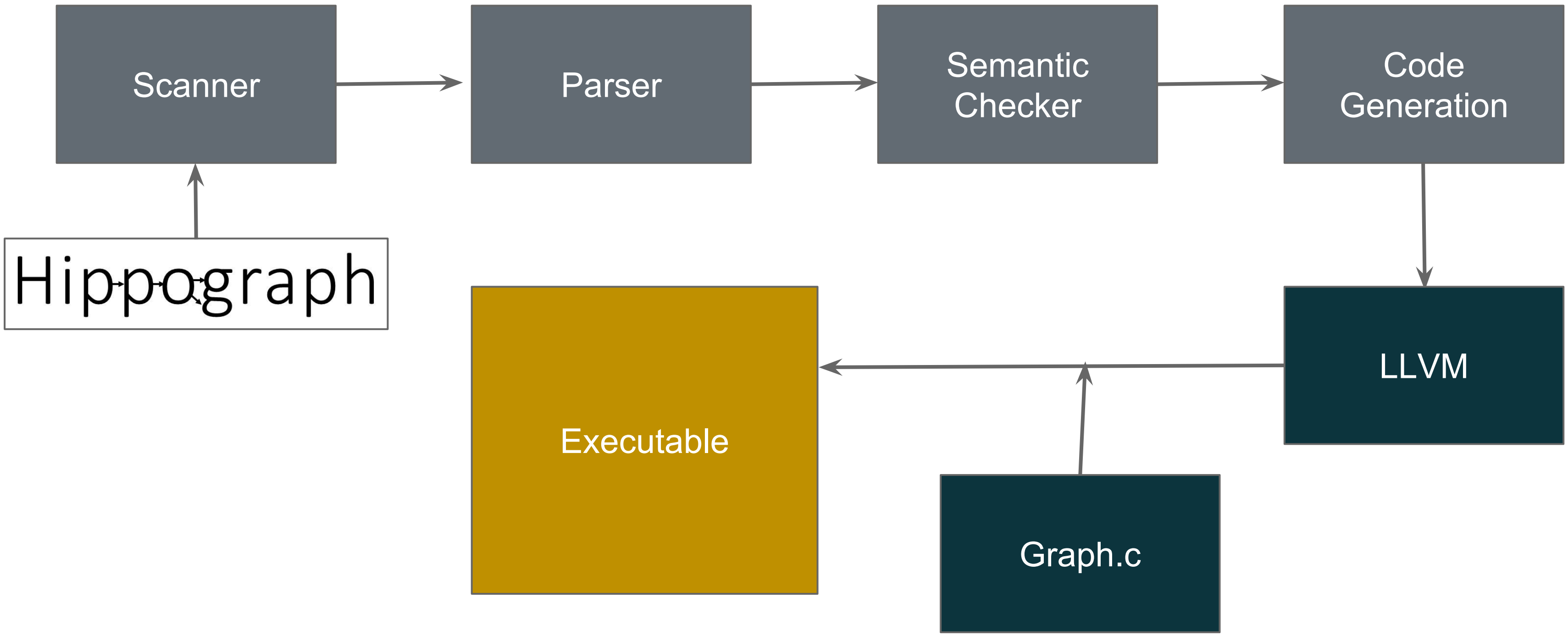
```
Node<t1:t2> = expr_of_t1 : expr_of_t2;
Node<t1:t2> = expr_of_t1;
Node<t1> = expr_of_t1;
```

- Graph Expressions:

```
Graph<int:bool, int> = [1:true <(5)> 3 <(3)- 8:true; 8 -(4)- 1];
Graph<int> = [1 <()> 3 <()- 8; 8 -()- 1];
```

Implementation

Architecture



Graphs

```c
/* constants */

int INTTYPE  = 1;
int STRTYPE  = 2;
int BOOLTYPE = 3;

/* data structures */

typedef union primitive {
  int *i;
  char *s;
} primitive;

typedef struct node node;

typedef struct edge {
  node *src;
  node *dst;
  primitive *w;
  int w_typ;
  struct edge *next;
  int has_val;
} edge;

```

```c
struct node {
  primitive *label;
  int label_typ;
  primitive *data;
  int data_typ;
  int has_val;
  neighbor_list *neighbor_list;
  node *next;
};

typedef struct node_list {
  node *hd;
} node_list;

typedef struct edge_list {
  edge *hd;
} edge_list;

typedef struct graph {
  node_list *node_list;
  edge_list *edge_list;
} graph;

```

Implemented as adjacency lists

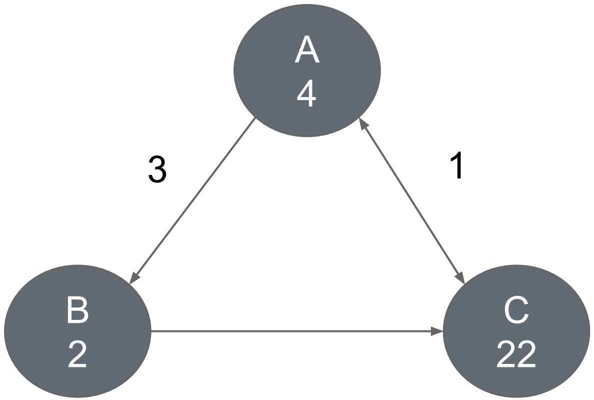
Union *primitive* allowed for flexible typing.

Under the hood, all edges are directed. Non-directional and bidirectional edges are implemented as two one-way edges.

Semantic Checking

```
and check_graph_expr fdecls vars node_list edge_list =
  (* infer node label/data types from first nodes in list if any,
     and check that all items have the same type *)
  let node_label_typ, node_data_typ, s_node_list =
    if node_list = []
    then (Bool, Bool, []) (* bool type, for now *)
    else let err = "type mismatch in graph nodes" in
      let check_node_typ (lt_opt, dt_opt) n =
        match n with
        | (Node(lt, dt), SNodeExpr(_, d)) ->
          (* check matching node label *)
          let lt_opt = (match lt_opt with
          | None -> Some(lt)
          | Some(lt') -> if lt = lt'
                         then lt_opt
                         else raise (Failure err)) in
          (* check matching node data *)
          let dt_opt = (match d with
          | (Bool, SNull) -> dt_opt
          | _ -> match dt_opt with
                 | None -> Some(dt)
                 | Some(dt') -> if dt = dt'
                                then dt_opt
                                else raise (Failure err))
          in (lt_opt, dt_opt)
        | _ -> raise Unsupported_constructor
```

```
graph<string:int, int> = ["A":4 -(3)>
"B":2 -()> "C":22 <(1)> "A"];
```



Testing

```
test-graph-neighbors4...OK
test-graph-neighbors5...OK
test-has-node-bool...OK
test-has-node-int...OK
test-has-node-str...OK
test-helloworld...OK
test-if-else...OK
test-if...OK
test-is-empty...OK
test-print-node...OK
test-printbool...OK
test-printint...OK
test-recursion1...OK
test-recursion2...OK
test-remove-edge1...OK
test-remove-node-bool...OK
test-remove-node-int...OK
test-remove-node-str...OK
test-set-data1...OK
test-set-edge-bool-int...OK
test-set-edge-bool-str...OK
test-set-edge-bool...OK
test-set-edge-int-bool...OK
test-set-edge-int-int...OK
test-set-edge-int-str...OK
test-set-edge-int...OK
test-set-edge-str-bool...OK
test-set-edge-str-str...OK
test-set-node1...OK
test-set-node2...OK
test-vdecls-global...OK
test-vdecls...OK
test-while1...OK
```

```
int main() {
  graph<int, int> g = [1 <(10)> 2; 3];
  int result1 = g.has_node(1);
  print_int(result1);
  int result2 = g.has_node(5);
  print_int(result2);
}
```

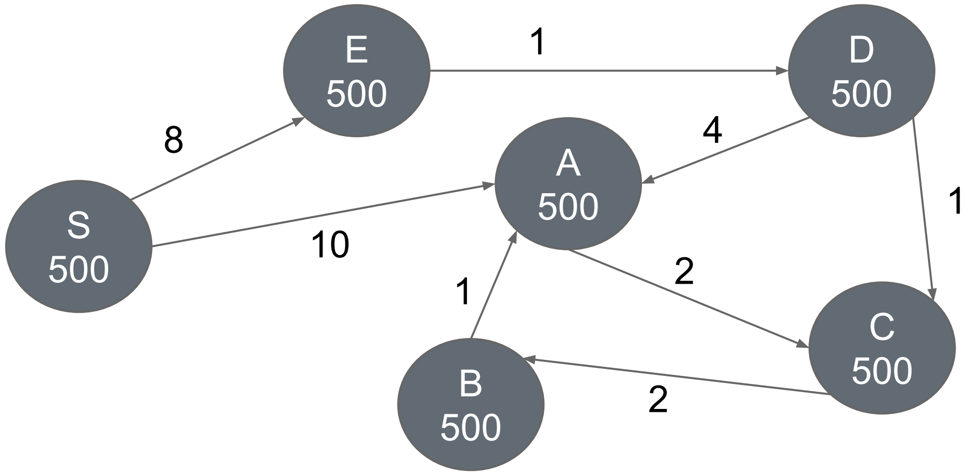
For every new feature implemented, a small test was created to ensure it worked as expected.

Demo

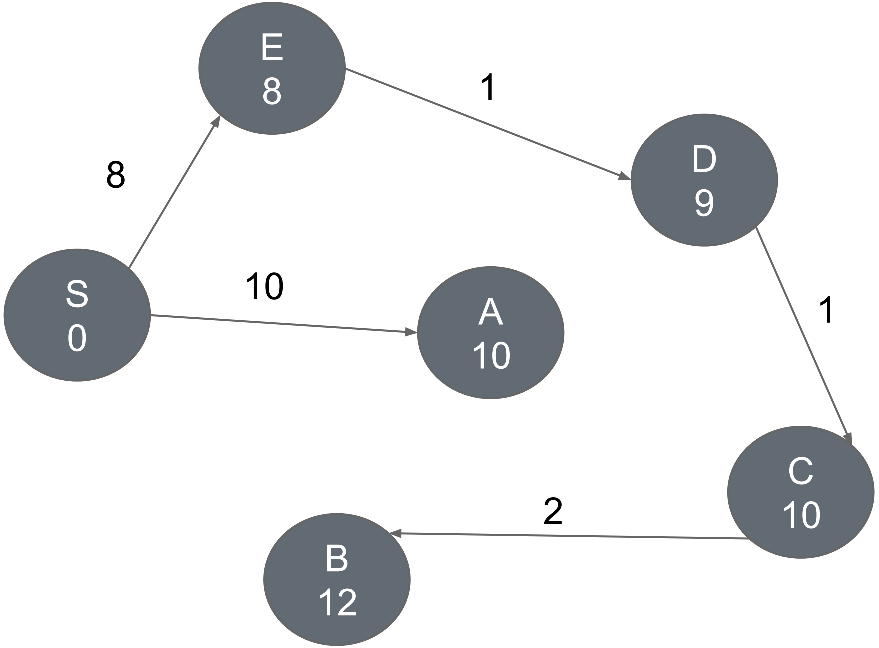
Bellman-Ford Algorithm

Initial Graph

```
graph<string:int, int> g = ["S":500 -(10)> "A":500 -(2)> "C":500 -(2)> "B":500
-(1)> "A"; "S" -(8)>"E":500 -(1)> "D":500 -(1)>"C"; "D" -(4)> "A"];
```

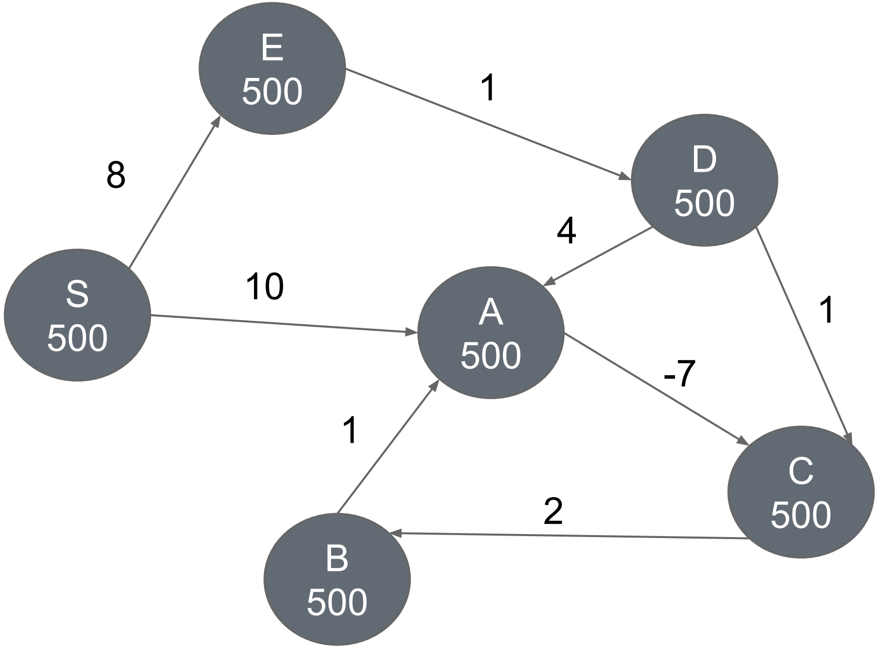


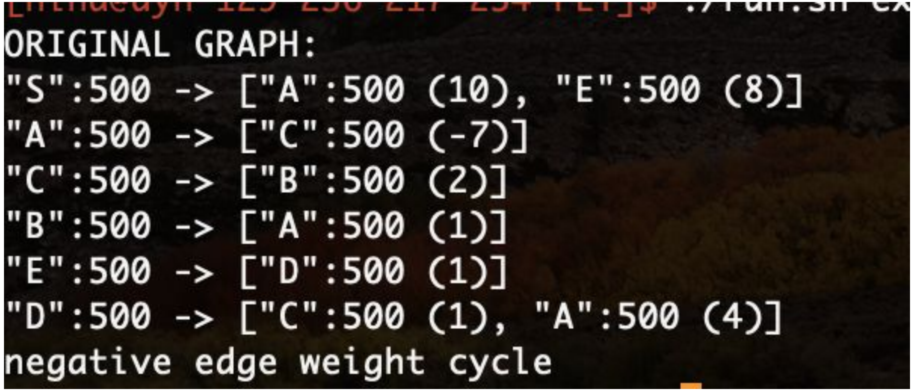
Shortest-path Graph




```
ORIGINAL GRAPH:
"S":500 -> ["A":500 (10), "E":500 (8)]
"A":500 -> ["C":500 (2)]
"C":500 -> ["B":500 (2)]
"B":500 -> ["A":500 (1)]
"E":500 -> ["D":500 (1)]
"D":500 -> ["C":500 (1), "A":500 (4)]
SHORTEST PATH:
"S":0 -> ["A":10 (10), "E":8 (8)]
"A":10 -> []
"C":10 -> ["B":12 (2)]
"B":12 -> []
"E":8 -> ["D":9 (1)]
"D":9 -> ["C":10 (1)]
```

Negative Edge Weight Cycles in Graph





Thank you!

Special thanks to our TA
Jennifer “codejen.ml” Bi!