UM-SJTU Joint Institute

Introduction to Algorithms

VE477 Lab6 Report

Name: Taoyue Xia, ID:518370910087

Date: 2021/11/25

0. Dense Graph

For lab6 implementation, I made some small changes to my dense graph class in `lab5` to make it meet the requirement of JOJ.

```python
class Vertex:
    def __init__(self, name, value):
        self.name = name
        self.value = value
        self.adjacent = {}
```

```python
class DenseGraph:
    def __init__(self):
        self.vertex_dict = {}

    def add_edge(self, start: 'Vertex', end: 'Vertex', weight):
        if start.name not in self.vertex_dict.keys():
            self.vertex_dict[start.name] = start
        if end.name not in self.vertex_dict.keys():
            self.vertex_dict[end.name] = end
        if end in start.adjacent.keys():
            self.set_edge_weight(start, end, weight)
            return
        start.adjacent[end] = weight
        end.adjacent[start] = 0

    def remove_vertex(self, v_name):
        if v_name not in self.vertex_dict.keys():
            raise LookupError("No such vertex in graph")
        v = self.vertex_dict[v_name]
        v.adjacent.clear()
        self.vertex_dict.pop(v_name)

    def set_edge_weight(self, start: 'Vertex', end: 'Vertex', weight):
        if start.name not in self.vertex_dict.keys():
            raise LookupError("Start vertex not in graph")
        if end not in start.adjacent.keys():
            raise LookupError("End vertex not in graph")
        start.adjacent[end] = weight

    def remove_edge(self, start: 'Vertex', end: 'Vertex'):
        if start.name not in self.vertex_dict.keys():
            raise LookupError("Start vertex not in graph")
        if end not in start.adjacent.keys():
            raise LookupError("End vertex not in graph")
        start.adjacent.pop(end)
        end.adjacent.pop(start)

    def is_adjacent(self, u: 'Vertex', v: 'Vertex'):
        if u.name not in self.vertex_dict.keys() or v.name not in self.vertex_dict.keys():
            raise LookupError("No such vertex in graph")
        if u.name in self.vertex_dict.keys():
            if v in u.adjacent.keys():
                return True
```

```python
        if v.name in self.vertex_dict.keys():
            if u in v.adjacent.keys():
                return True
        return False

    def get_vertex_value(self, v_name):
        if v_name not in self.vertex_dict.keys():
            raise LookupError("No such vertex in graph")
        return self.vertex_dict[v_name].value

    def add_vertex(self, v_name, value):
        if v_name not in self.vertex_dict.keys():
            v = Vertex(v_name, value)
            self.vertex_dict[v_name] = v

    def get_edge_weight(self, start: 'Vertex', end: 'Vertex'):
        if start.name not in self.vertex_dict.keys():
            raise LookupError("Start vertex not in graph")
        if end not in start.adjacent.keys():
            raise LookupError("End vertex not in graph")
        return start.adjacent[end]

    def set_vertex_value(self, v_name, value):
        if v_name not in self.vertex_dict.keys():
            raise LookupError("No such vertex in graph")
        self.vertex_dict[v_name] = value

    def get_vertex(self, v_name):
        if v_name not in self.vertex_dict.keys():
            return None
        return self.vertex_dict[v_name]
```

1. Breadth First Search Algorithm in OCaml

```ocaml
type vertex = {
  index: int;
  mutable dist: int;
  mutable prev: vertex option;
  mutable adj: vertex list
};;

let rec read_edges num l h =
  if num == 0 then
    List.rev l
  else
    let s = read_line() in
    let vertices = List.map int_of_string (Str.split (Str.regexp " ") s) in
    let idx_u = List.nth vertices 0 in
    let idx_v = List.nth vertices 1 in
    if (Hashtbl.mem h idx_u) && (Hashtbl.mem h idx_v) then
      let u = Hashtbl.find h idx_u in
      let v = Hashtbl.find h idx_v in
      u.adj <- u.adj @ [v];
      v.adj <- v.adj @ [u];
      read_edges (num-1) l h
```

```
    else if (Hashtbl.mem h idx_u) then
      let u = Hashtbl.find h idx_u in
      let v = {index = idx_v; dist = max_int; prev = None; adj = [u]} in
      Hashtbl.add h idx_v v;
      u.adj <- u.adj @ [v];
      read_edges (num-1) (v::l) h
    else if (Hashtbl.mem h idx_v) then
      let v = Hashtbl.find h idx_v in
      let u = {index = idx_u; dist = max_int; prev = None; adj = [v]} in
      Hashtbl.add h idx_u u;
      v.adj <- v.adj @ [u];
      read_edges (num-1) (u::l) h
    else
      let u = {index = idx_u; dist = max_int; prev = None; adj = []} in
      let v = {index = idx_v; dist = max_int; prev = None; adj = []} in
      Hashtbl.add h idx_u u;
      Hashtbl.add h idx_v v;
      u.adj <- v::u.adj;
      v.adj <- u::v.adj;
      read_edges (num - 1) ([v; u] @ l) h
;;


let bfs h start =
  let q = Queue.create () in
  Queue.push start q;
  let rec queue_operate l =
    if Queue.is_empty q then
      List.rev l
    else
      let u = Queue.pop q in
      List.iter (fun x -> if x.dist == max_int then begin x.dist <- u.dist + 1; x.prev <- Some u; Queue.push x q end) u.adj;
      queue_operate (u::l)
  in
  queue_operate []
;;

let rec print_list l =
  match l with
  | [] -> print_string ""
  | head :: [] -> print_int head.index
  | head :: tail -> print_int head.index; print_string " "; print_list tail
;;

let num_edges = read_int();;

let hash = Hashtbl.create num_edges;;

let v_list = read_edges num_edges [] hash;;
let start = Hashtbl.find hash 0;;
start.dist <- 0;;
let l = bfs hash start;;
print_list l;;
```

The correctness of this algorithm can be verified by the full mark tested on JOJ.

2. Edmonds-Karp algorithm

It is quite hard for me to implement `class` in `OCaml`, so I just compromised and implement it using `python` with the `DenseGraph` class defined.

```python
from graph import DenseGraph, Vertex


def bfs(g: 'DenseGraph', s: 'Vertex', t: 'Vertex', prev: 'dict'):
    visited = {}
    for vertex in g.vertex_dict.values():
        visited[vertex] = False

    q = [s]
    visited[s] = True
    while len(q) != 0:
        u = q.pop(0)
        for v in u.adjacent.keys():
            if (not visited[v]) and (u.adjacent[v] > 0):
                prev[v] = u
                visited[v] = True
                if v == t:
                    return True
                q.append(v)

    return visited[t]


def edmonds_karp(g: 'DenseGraph', s: 'Vertex', t: 'Vertex'):
    prev = {}
    for vertex in g.vertex_dict.values():
        prev[vertex] = None
    max_flow = 0
    while bfs(g, s, t, prev):
        min_cut = float('inf')
        current = t
        while current != s:
            p = prev[current]
            min_cut = min(min_cut, p.adjacent[current])
            current = prev[current]

        max_flow += min_cut
        # print(max_flow)
        current = t
        while current != s:
            p = prev[current]
            g.set_edge_weight(p, current, p.adjacent[current] - min_cut)
            g.set_edge_weight(current, p, current.adjacent[p] + min_cut)
            current = prev[current]
    return max_flow


if __name__ == '__main__':
    graph = DenseGraph()
    edge_num = int(input())

    for i in range(edge_num):
```

```
        elements = input().split()
        graph.add_vertex(elements[0], 0)
        graph.add_vertex(elements[1], 0)
        start = graph.get_vertex(elements[0])
        end = graph.get_vertex(elements[1])
        capacity = int(elements[2])
        graph.add_edge(start, end, capacity)
    source = graph.get_vertex(input())
    sink = graph.get_vertex(input())

    print(edmonds_karp(graph, source, sink))
```

3. Maximum Bipartite Matching

To make it easy to implement, each time an edge is added by input, the edge's capacity would bee set as 1, and the former vertex will be added into a list `left`, the latter vertex will be added into the list `right`.

After all the vertices and edges are initialized. A source vertex `s` and a sink vertex `t` is created, and `s` will then be linked to all the vertices in `left`, so will `t` be linked to all the vertices in `right`, Also with all the edges' capacity set as 1.

Finally, by running the `edmonds_karp` function defined in `EdmondsKarp.py`, we can get the correct maximum number of matching.

```
from graph import DenseGraph, Vertex
from EdmondsKarp import bfs, edmonds_karp


if __name__ == '__main__':
    graph = DenseGraph()
    v_num = int(input())
    e_num = int(input())

    left = []
    right = []
    for i in range(e_num):
        elements = input().split()
        graph.add_vertex(elements[0], 0)
        graph.add_vertex(elements[1], 0)
        start = graph.get_vertex(elements[0])
        end = graph.get_vertex(elements[1])
        graph.add_edge(start, end, 1)
        if start not in left:
            left.append(start)
        if end not in right:
            right.append(end)

    # source_sink = input().split()
    graph.add_vertex('source', 0)
    graph.add_vertex('sink', 0)
    source = graph.get_vertex('source')
    sink = graph.get_vertex('sink')
    for v in left:
```

```python
        graph.add_edge(source, v, 1)
    for v in right:
        graph.add_edge(v, sink, 1)
    print(edmonds_karp(graph, source, sink))
```

Given the input as (on JOJ):

```
10
12
0 7
0 8
0 9
1 6
1 9
2 5
2 6
2 9
3 6
3 9
4 5
4 6
10 12
```

We can get the correct answer as `4`.

Furthermore, all the other cases pass.

By constructing, since we transform the bipartite graph into a flow graph, with the capacity of all the edges being 1, we can just simply run Edmonds-Karp Algorithm on the graph, and the final returned flow would be the maximum matching number. Therefore, the well-functioning is proved.