UM-SJTU Joint Institute

Introduction to Algorithms

VE477 Lab5 Report

Name: Taoyue Xia, ID:518370910087

Date: 2021/11/21

1. Graph Representations

Firstly, define a class `Vertex` to have attributes `name`, `value`, as well as an `adjacent` dictionary which is used in dense graph to store edges to speed up operations.

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

1) Sparse Graph

```


class SparseGraph:
    def __init__(self):
        self.edge_dict = {}
        self.vertex_dict = {}

    def add_edge(self, start: 'Vertex', end: 'Vertex', weight):
        edge = (start, end)
        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
        self.edge_dict[edge] = weight

    def remove_vertex(self, v: 'Vertex'):
        if v.name not in self.vertex_dict.keys():
            raise LookupError("No such vertex in graph")
        self.vertex_dict.pop(v.name)
        for edge in self.edge_dict.keys():
            if v in edge:
                self.edge_dict.pop(edge)

    def set_edge_weight(self, start: 'Vertex', end: 'Vertex', weight):
        if (start, end) in self.edge_dict.keys():
            self.edge_dict[(start, end)] = weight
        elif (end, start) in self.edge_dict.keys():
            self.edge_dict[(end, start)] = weight
        else:
            raise LookupError("No edge linking the two vertices")

    def remove_edge(self, start: 'Vertex', end: 'Vertex'):
        if (start, end) in self.edge_dict.keys():
            self.edge_dict.pop((start, end))
        else:
            raise LookupError("No such edge exists in the graph")

    def is_adjacent(self, u, v):
        if (u, v) in self.edge_dict.keys() or (v, u) in
self.edge_dict.keys():
```

```python
                return True
            else:
                return False

    def get_vertex_value(self, v_name):
        if v_name in self.vertex_dict.keys():
            return self.vertex_dict[v_name].value
        else:
            raise LookupError("No such vertex in graph")

    def add_vertex(self, v_name, value):
        if v_name in self.vertex_dict.keys():
            raise ValueError("Vertex already exists")
        self.vertex_dict[v_name] = Vertex(v_name, value)

    def get_edge_weight(self, start: 'Vertex', end: 'Vertex'):
        if (start, end) in self.edge_dict.keys():
            return self.edge_dict[(start, end)]
        elif (end, start) in self.edge_dict.keys():
            return self.edge_dict[(end, start)]
        else:
            raise LookupError("No such edge in graph")

    def set_vertex_value(self, name, value):
        if name in self.vertex_dict.keys():
            self.vertex_dict[name].value = value

    def get_vertex(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]
```

Since there are not too many edges in a sparse graph, we can use the `edge_dict` to store edge informations and operates on edge efficiently.

2) Dense Graph

```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
        v = self.vertex_dict[start.name]
        if end in v.adjacent.keys():
            raise ValueError("edge already exists in graph, please use
set_edge_weight")
        v.adjacent[end] = weight

    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)

    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
        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 in self.vertex_dict.keys():
            raise ValueError("Name of vertex already exists")
        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():
            raise LookupError("No such vertex")
        return self.vertex_dict[v_name]
```

Since there are a large number of edges in a dense graph, if we store each edge, it would take up too much space, so I add an `adjacent` dictionary to each vertex, to represent an edge for interaction with each other vertex stored in the dictionary.

2. Dijkstra with Fibonacci Heap

The code for fibonacci heap is attached in `lab4`, so I will just show the code for `Dijkstra` using fibonacci heap.

```python
from fibonacci import *


class Edge:
    def __init__(self, nodeA, nodeB, weight):
        self.start = nodeA
        self.end = nodeB
        self.weight = weight


node_dict = {}
edge_num = int(input())

edge_dict = {}
for i in range(edge_num):
    line = input().split()
    if line[0] not in node_dict.keys():
        node_dict[line[0]] = Node(float("inf"))
    if line[1] not in node_dict.keys():
        node_dict[line[1]] = Node(float("inf"))
    weight = int(line[2])
    edge = Edge(node_dict[line[0]], node_dict[line[1]], weight)
    if node_dict[line[0]] not in edge_dict.keys():
        edge_dict[node_dict[line[0]]] = []
        edge_dict[node_dict[line[0]]].append(edge)
    else:
        edge_dict[node_dict[line[0]]].append(edge)


start = input()
node_dict[start].data = 0
end = input()
end_node = node_dict[end]

fib = FibonacciHeap()
for key, node in node_dict.items():
    fib.insert(node)

rev_dict = {}
for key, value in node_dict.items():
    rev_dict[value] = key

v = node_dict[start]

while v is not node_dict[end]:
    for edge in edge_dict[v]:
        u = edge.end
        tmp = v.data + edge.weight
        if tmp < u.data:
            fib.decrease_key(u, tmp)
            u.prev = v
    v = fib.extract_min()

result = []
t = end_node
result.append(rev_dict[t])
while t.prev is not None:
    result.append(rev_dict[t.prev])
    t = t.prev
result.reverse()
print(result)
```

3. Bellman-Ford in OCaml

The code for Bellman-Ford implemented in OCaml is shown below:

```
type node = {
  name: string;
  mutable distance: float;
  mutable prev: string option;
};;

type edge = {
  start: node;
  dest: node;
  weight: float;
};;

(*initialize edge list and vertex hashtable from input*)
let rec read_edges num edge_list h =
  if num == 0 then
    List.rev edge_list
  else
    let edge_name = read_line() in
    let parse = Str.split (Str.regexp " ") edge_name in
    let start = List.nth parse 0 in
    let dest = List.nth parse 1 in
    let weight = float_of_string (List.nth parse 2) in
    if (Hashtbl.mem h start) && (Hashtbl.mem h dest) then
      read_edges (num - 1) ({start = Hashtbl.find h start; dest = Hashtbl.find h dest; weight = weight} :: edge_list) h
    else if (Hashtbl.mem h start) then
      let nodeB = {name = dest; distance = infinity; prev = None} in
      Hashtbl.add h dest nodeB;
      read_edges (num - 1) ({start = Hashtbl.find h start; dest = nodeB; weight = weight} :: edge_list) h
    else if (Hashtbl.mem h dest) then
      let nodeA = {name = start; distance = infinity; prev = None} in
      Hashtbl.add h start nodeA;
      read_edges (num - 1) ({start = nodeA; dest = Hashtbl.find h dest; weight = weight} :: edge_list) h
```

```ocaml
    else
      let nodeA = {name = start; distance = infinity; prev = None} in
      let nodeB = {name = dest; distance = infinity; prev = None} in
      let edge = {start = nodeA; dest = nodeB; weight = weight} in
      Hashtbl.add h start nodeA;
      Hashtbl.add h dest nodeB;
      read_edges (num-1) (edge::edge_list) h
;;

(*realize the loop for edges*)
let rec loop_e  edge_list l h =
  match l with
  | [] -> edge_list
  | head::tail -> let u = head.start in
                  let v = head.dest in
                  let tmp = u.distance +. head.weight in
                  if tmp < v.distance then
                    begin
                      v.distance <- tmp;
                      v.prev <- Some u.name;
                    end;
                  loop_e edge_list tail h
;;

(*realize the loop for vertices*)
let rec loop_v vnum l h =
  if vnum == 0 then
    []
  else
    begin
    let el = loop_e l l h in
    loop_v (vnum-1) el h;
    end
;;

(*a helper function to turn option type to string*)
let op_to_str data =
  match data with
  | None -> ""
  | Some str -> str
;;

(*Final path generation*)
let rec find_path dest result h =
  if dest.prev == None then
    (dest.name :: result)
  else
    find_path (Hashtbl.find h (op_to_str dest.prev)) (dest.name :: result) h
;;

(*Main function*)
let rec print_list l =
  match l with
  | [] -> print_string "]"
  | head :: [] -> print_string "'"; print_string head; print_string "']";
  | head :: tail ->  print_string "'"; print_string head; print_string "', "; print_list tail
```

```ocaml
;;

let edge_num = read_int();;
let h = Hashtbl.create edge_num;;

let edge_list = read_edges edge_num [] h;;
let start = read_line();;
let start_node = Hashtbl.find h start;;
start_node.distance <- 0.0;;
let dest = read_line();;
let end_node = Hashtbl.find h dest;;
let v_num = Hashtbl.length h;;

loop_v v_num edge_list h;;
let result = find_path end_node [] h;;
print_string "[";;
print_list result;;
```

4. Comparison of Dijkstra and Bellman-Ford

i. Complexity

Since **Dijkstra** is implemented with fibonacci heap, the complexity of this algorithm will be reduced from $\mathcal{O}((V+E)\log V)$ to $\mathcal{O}(E+V\log V)$.

Bellman-Ford's complexity is always $\mathcal{O}(VE)$.

ii. Running Time

With the example input on JOJ, for **Dijkstra** algorithm, its needs:

```
Running time: 128.972 ms
```

to get the final answer.

For **Bellman-Ford**, it needs:

```
Running time: 9.783 ms
```

to finish the same task.

Bellman-Ford is faster because `OCaml`'s running efficiency is far faster than `Python`. If they are implemented in the same language, **Dijkstra** with **Fibonacci Heap** should be faster.