R13 Dijkstra's Algorithm

4.1/4.2 https://github.com/GUMI-21/MIT6.006_note

Dijkstra's Algorithm

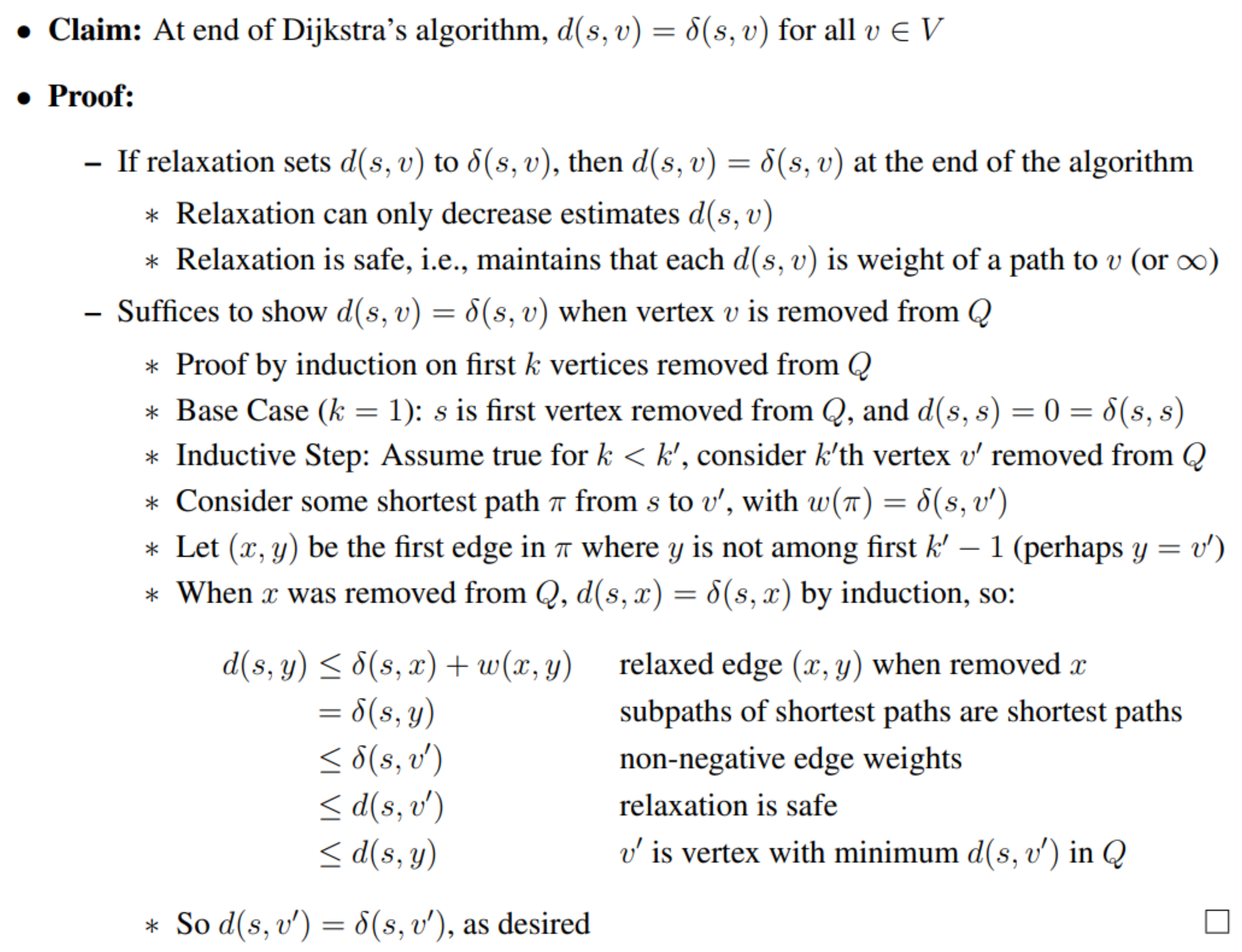
Dijkstra’s running time then depends on how efficiently the priority queue can perform its supported operations. Below is Python code for Dijkstra’s algorithm in terms of priority queue operations.

```python
def dijkstra(Adj, w, s):
    d = [float('inf') for _ in Adj]
    parent = [None for _ in Adj]
    d[s], parent[s] = 0, s
    Q = PriorityQueue()
    V = len(Adj)
    for v in range(V):
        Q.insert(v,d[v])   # insert vertex-estimate pair
    for _ in range(V):
        u = Q.extract_min()  # extract vertex with min estimate
        for v in Adj(u):
            try_to_relax(Adj, w, d, parent, u, v)
            Q.decrease_key(v, d[v])    # update key of vertex
    return d, parent
```

Correctness

The key observation is that shortest path weight estimate of vertex u equals its actual shortest path weight d(s, u) = δ(s, u) when u is removed from the priority queue.

- Proof



- **Claim:** At end of Dijkstra's algorithm, $d(s,v) = \delta(s,v)$ for all $v \in V$
- **Proof:**
  - If relaxation sets $d(s,v)$ to $\delta(s,v)$, then $d(s,v) = \delta(s,v)$ at the end of the algorithm
    - Relaxation can only decrease estimates $d(s,v)$
    - Relaxation is safe, i.e., maintains that each $d(s,v)$ is weight of a path to v (or ∞)
  - Suffices to show $d(s,v) = \delta(s,v)$ when vertex v is removed from Q
    - Proof by induction on first k vertices removed from Q
    - Base Case ($k = 1$): s is first vertex removed from Q, and $d(s,s) = 0 = \delta(s,s)$
    - Inductive Step: Assume true for $k < k'$, consider k'th vertex v' removed from Q
    - Consider some shortest path π from s to v', with $w(\pi) = \delta(s,v')$
    - Let (x,y) be the first edge in π where y is not among first $k'-1$ (perhaps $y = v'$)
    - When x was removed from Q, $d(s,x) = \delta(s,x)$ by induction, so:

$$\begin{aligned} d(s,y) &\le \delta(s,x) + w(x,y) && \text{relaxed edge } (x,y) \text{ when removed } x \\ &= \delta(s,y) && \text{subpaths of shortest paths are shortest paths} \\ &\le \delta(s,v') && \text{non-negative edge weights} \\ &\le d(s,v') && \text{relaxation is safe} \\ &\le d(s,y) && v' \text{ is vertex with minimum } d(s,v') \text{ in } Q \end{aligned}$$

    - So $d(s,v') = \delta(s,v')$, as desired □

when x is removed, y is relaxed => d(s,y)≤δ(s,y)
but from Q we know d(s,v')≤d(s,y)≤δ(s,y), and we know y is before v', so δ(s,y) <= δ(s,v')
so d(s,v') <= δ(s,v'), means d(s,v') = δ(s,v'). *we choose x,y which y is in \pi but not removed from Q.*
When remove v' from Q, there is possible some vertex in Q not removed.
we assume x,y on the shortest path which x is removed but y is not removed first edge.

Priority Queues

Here, a priority queue maintains a set of key-value pairs,
where *vertex v is a value* and *d(s, v) is its key*.

- operations
  insert(val, key): T*i adds a key-value pair to the queue*
  extract_min(): T_e removes and returns a value from the queue whose key is minimum,
  decrease_key(val, new key): T_d which reduces the key of a given value stored in the queue to the provided new key. THEN
  *$T*{Dijkstra} = O(|V| \cdot T_i + |V| \cdot T_e + |E| \cdot T_d).$

- If use directly access array or dictionary. O(1) T*i,T_d O(|V|) T_e* *$T*{Dict} = O(|V|^2 + |E|).*Thisisactuallyquitegood!Ifthegraphisdense,*|E| = \Omega(|V|^2)$
- Use a direct access array to implement the priority queue.

```
class ProirityQueue   # Hash Table Implementation
    def __init__(self):
        self.A = {}

    def insert(self, label, key):   # insert labeled key
        self.A[lable] = key

    def extract_min(self):   # return a label with minimum key
        min_label = None
        for label in self.A
            if (min_label is None) or (self.A[lable] < self.A[min_label1].key)
                min_label = label
            del self.A[min_label]
            return min_label

    def decrease_key(self, label, key):  # decrease key of a given lable
        if (label in self.A) and (key < self.A[label]):
            self.A[label] = key
```

OR, if graph is sparse, |E| = O(|V |).

- *Use binary-heap*
  1.insertion and extract-min in O(log n) time.
  2.each vertex can maintain a pointer to its stored location within the heap, or the heap can maintain a mapping from values (vertices) to locations within the heap.
  Either solution can support finding a given value in the heap in constant time.
  Then, after decreasing the value's key, one can restore the min heap property in logarithmic time by re-heapifying the tree.
  So the three operations in O(log|V|) time.
  => $T_{Heap} = O((|V| + |E|) \log |V|)$.
  For sparse graph, this is $O(|V| \log |V|)$
- *Fibonacci Heaps*
  For graphs in between sparse and dense, there is an even more sophisticated priority queue implementation using a data structure called a Fibonacci Heap.
  which supports amortized O(1) time insertion and decrease-key operations, along with O(log n) minimum extraction.
  $T_{FibHeap} = O(|V| \log |V| + |E|)$.
  Fibonacci Heaps are not actually used very often in practice as it is more complex to

implement, and results in larger constant factor overhead than the other two implementations described above.

When the number of edges in the graph is known to be at most linear (e.g., planar or bounded degree graphs) or at least quadratic (e.g. complete graphs) in the number of vertices, then using a binary heap or dictionary respectively will perform as well asymptotically as a Fibonacci Heap.

- Python code of binary-min-heap Priority queue

```python
class Item:
    def __init__(self, label, key):
        self.label, self.key = label, key

class PriorityQueue:
    def __init__(self):
        self.A = []
        self.label2idx = {}

    def min_heapify_up(self, c):
        if c == 0: return
        p = (c - 1) // 2
        if self.A[p].key > self.A[c].key
            self.A[c], self.A[p] = self.A[p], self.A[c]
            self.label2idx[self.A[c].label] = c
            self.label2idx[self.A[p].label] = p
            self.min_heapify_up(p)
    def min_heapify_down(self, p):
        if p >= len(self.A): return
        l = 2*p+1
        r = 2*p+2
        if l >= len(self.A): l=p
        if r >= len(self.A): r=p
        c = l if self.A[r].key > self.A[l].key else r
        if self.A[p].key > self.A[c].key:
            self.A[c], self.A[p] = self.A[p], self.A[c]
            self.label2idx[self.A[c].label] = c
            self.label2idx[self.A[p].label] = p
            self.min_heapify_down(c)
    def insert(self, label, key):
        self.A.append(Item(label,key))
        idx = len(self.A) - 1
        self.label2idx(self.A[idx].label) = idx
        self.min_heapify_up(idx)
    def extract_min(self):
        self.A[0], self.A[-1] = self.A[-1], self.A[0]
        self.label2idx[self.A[-1].label] = 0
        del self.label2idx[self.A[-1].label]
        min_label = self.A.pop().label
        self.min_heapify_down(0)
        return min_label
    def decrease_key(self, label, key):
        if label in self.label2idx:
            idx = self.labe12idx[label]
            if key < self.A[idx].key:
                self.A[idx].key = key
                self.min_heapify_up(idx)
```