SHANGHAI JIAO TONG UNIVERSITY

Algorithm Analysis HW:2

Author: *519030910100 Huangji Wang F1903004*

Course: *Fall 2021, AI2615: Algorithm*

Date: *November 15, 2021*

Problem 1 (20 points)

Here is a proposal to find the length of the shortest cycle in an unweighted undirected graph:

DFS the graph, when there is a back edge (v; u), it forms a cycle from u to v, and the length is level[v] - level[u] + 1, where the level of a vertex is its distance in the DFS tree from the root. This suggests the following algorithm:

1. Do a DFS and keep tracking the level.
2. Each time we find a back edge, compute the cycle length, and update the smallest length.

Please justify the correctness of the algorithm, prove it or provide a counterexample.

This algorithm is not perfectly true. Counterexample:

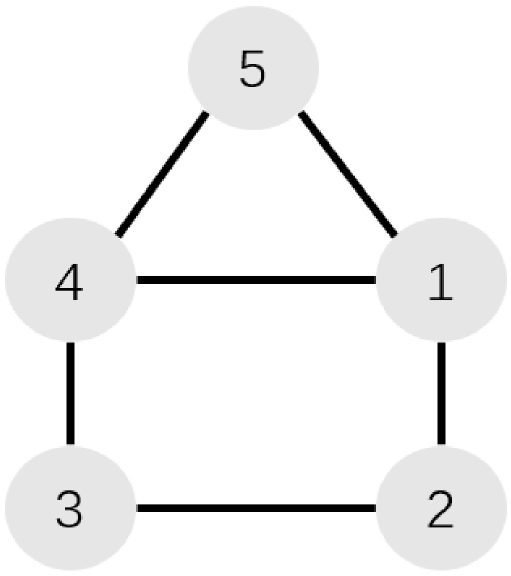


Figure 1: Counterexample

We can always find one path like this picture above. If we do the DFS from the initial vertex $v(1)$, we will form the path as:

- initialize ans = $+\infty$
- (1)
- (1) -> (2)
- (1) -> (2) -> (3)
- (1) -> (2) -> (3) -> (4)
- (1) -> (2) -> (3) -> (4) -> (1) **ans = min(ans, level[4] - level[1] + 1) = 4**
- (1) -> (2) -> (3) -> (4)
- (1) -> (2) -> (3) -> (4) -> (5)
- (1) -> (2) -> (3) -> (4) -> (5) -> (1) **ans = min(ans, level[5] - level[1] + 1) = 4**
- (1) -> (2) -> (3) -> (4) -> (5)
- (1) -> (2) -> (3) -> (4)
- (1) -> (2) -> (3)
- (1) -> (2)
- (1)
- **ans = 4**

However, if we look at the graph, we can easily find that the smallest cycle is $v(5)- > v(1)- > v(4)$ and $ans = 3$. Thus, the algorithm may miss some important cycles, leading to the wrong answer.

My suggestion on how to improve this algorithm is to do DFS without visited arrays. Though it will cost much time than we think, it can find every cycle in the graph and it cannot miss any cycle.

The best way to handle with this problem is to use **union-find disjoint sets** which is very useful to find cycles in (unweighted) undirected graph.

Reference: **ACMOJ 1299 Cycle (Click for detail information)**

Problem 2 (20 points)

Given a directed graph $G = (V; E)$ on which each edge $(u; v) \in E$ has a weight $p(u; v)$ in range $[0; 1]$, that represents the reliability. We can view each edge as a channel, and $p(u; v)$ is the probability that the channel from u to v will not fail. We assume all these probabilities are independent. Give an efficient algorithm to find the most reliable path from two given vertices s and t. Hint: it makes a path failed if any channel on the path fails, and we want to find a path with minimized failure probability.

We want the reliability as more as possible. Notice that all the channels are connected by series, which means if there exists one failure, the whole channel will fail and go wrong. Thus, we hope to find the largest reliability for the channel to **survive** from the failure. It can be regarded as the **largest path** from s to v. Then we can use Dijkstra Algorithm to find the best single-source **'greatest'** path from s to t. The main idea is a little different from normal Dijkstra Algorithm because in this problem, the weight is a real number in range $[0; 1]$. Thus, when we update the total reliability, we make it as large as possible. The codes in the next page are presented below to show my meanings.

However, the general Dijkstra algorithm cannot be used to solve the longest length path. Thus, we do the operation on the reliability of edges. $e[i].v = 1 - e[i].v$ and then the longest length path problem can be transferred into the shortest length path.

The main structure of the algorithm goes like:

1. **Initialize**

   - $T = s$,

   - $tdist[s] = 1, tdist[v] \leftarrow \infty$ for all v other than s,

   - $w(s, v) = 1 - w(s, v)$ for all $(s, v) \in E$,

   - $tdist[v] \leftarrow w(s, v)$ for all $(s, v) \in E$.

2. **Explore**

   - Find $v \notin T$ with smallest $tdist[v]$

   - $T \leftarrow T + \{v\}$

3. **Update tdist[u]**

   - $tdist[u] = min\{tdist[u], tdist[v] * w(v, u)\}$ for all $(v, u) \in E$.

   - If $tdist[u]$ is updated, then $pre[u] \leftarrow v$

```
void dijkstra(){
    # The distance d[i] is regarded as the total reliability from s to i.
    # This initialization is different due to real numbers.
    for(int i=1;i<=n;i++)
        dis[i]=INF;
    # Change the longest problem to the shortest problem.
    for(int i=0;i<m;i++)
        e[i].w = 1 - e[i].w
    # Suppose the reliability in s is the largest 1.
    # s is the start point and it has no from vertex.
    dis[s]=1;       from[s] = -1;
    # Put the node s with reliability 1 into the priority queue.
    q.push((node){1,s});
    while(!q.empty()){
        node x=q.top();  q.pop();
        long long u=x.now;
        # To avoid extra visits.
        if(vis[u]) continue;    vis[u]=1;
        for(int i=head[u];i;i=e[i].next){
            long long v=e[i].v;
            # Attention! We update the reliability by multiplication.
            if(dis[v]<dis[u]*e[i].w){
              dis[v]  = dis[u]*e[i].w;
              # Collect its pre_edge so that we can find one full path.
              from[v] = e[i].u;
              q.push((node){dis[v],v});
            }
        }
    }
    int tmp = t;
    while(from[tmp] != -1){
        # Output the edge.
        printf("%d -> ",tmp);
        # Find the whole path with the help of the pre_edge.
        tmp = from[tmp];
    }
    # The last vertex is s.
    printf("%d\n",tmp);         return;
}
```

Problem 3 (20 points)

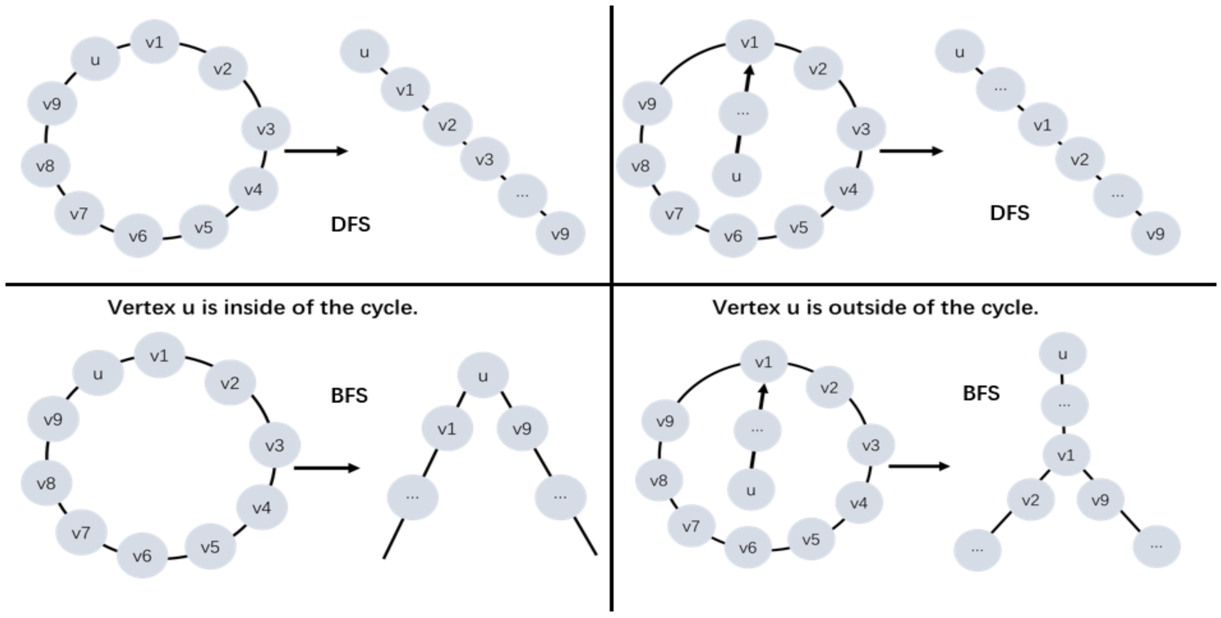
We have a connected undirected graph $G = (V; E)$, and a specific vertex $u \in V$. Suppose we compute a depth-first search tree rooted at u, and obtain a T that includes all nodes of G. Suppose we then compute a breath-first search tree rooted at u, and obtain the same tree T. Prove that $G = T$. (In other words, if T is both a DFS tree and a BFS tree rooted at u, then G cannot contain any edges that do not belong to T.)

Proof.

We consider two situations of the connected undirected graph G.

The **first** is that the G is a connected undirected graph and **it is a tree**. Then any search method that can traverse all the edges from the same start vertex can get the same tree obviously. Thus, DFS and BFS from u can get the same tree T. And it means that the graph is a tree, i.e., $G = T$.

The **second** is that the G is a connected undirected graph and **it is not a tree**. Then there exists at least one cycle consisted of n vertexes like:$v_1 - > v_2 - > ... - > v_n - > v_1$. If we do DFS, then it can form a direct and perfect long path from root to one leaf(Pictures are presented below). However, if we do BFS, it means it will have at least 2 branches and the BFS tree will be different from the DFS tree. Then it yields a contradiction!



DFS
DFS
Vertex u is inside of the cycle.
Vertex u is outside of the cycle.
BFS
BFS

Figure 2: Explanation

Above all, $G = T$. In other words, if T is both a DFS tree and a BFS tree rooted at u, then G cannot contain any edges that do not belong to T.

Problem 4 (20 points)

Given a directed graph $G(V; E)$ where each vertex can be viewed as a port. Consider that you are a salesman, and you plan to travel the graph. Whenever you reach a port v, it earns you a profit of p_v dollars, and it cost you c_{uv} if you travel from u to v. For any directed cycle in the graph, we can define a profit-to-cost ratio to be

$$r(C) = \frac{\Sigma_{(u,v)\in C} P_v}{\Sigma_{(u,v)\in C} c_{uv}}$$

As a salesman, you want to design an algorithm to find the best cycle to travel with the largest profit-to-cost ratio. Let r^* be the maximum profit-to-cost ratio in the graph.

(a) (10 points) If we guess a ratio r, can we determine whether $r^* > r$ or $r^* < r$ efficiently?

(b) (10 points) Based the guessing approach, given a desired accuracy $\epsilon > 0$, design an efficient algorithm to output a good-enough cycle, where $r(C) \geq r^* - \epsilon$.Justify the correctness and analyze the running time in terms of $|V|$, ϵ, and $R = max_{(u;v)\in E}(p_u/c_{uv})$.

We can do the transformation of the equation:

$$r(C) * \Sigma_{(u,v)\in C} c_{uv} = \Sigma_{(u,v)\in C} P_v$$

and thus we can get that for each edge (u, v), we have the loss factor $l_{u,v} = r(C) * c_{uv} - P_v$.

(a) Notice that $\Sigma_{(u,v)\in C} c_{uv}$ is always strictly positive. To determine whether $r^* > r$ or $r^* < r$, we just need to consider the conditions of $\Sigma_{(u,v)\in C} l_{u,v}$.

1. **Zero Weight Cycle**

   In zero weight cycle, we find that each item should be 0 and at last the r is meaningless, which means $r = r^*$ in this condition because r^* and r are all close to infinity .

2. **Negative Weight Cycle**

   In this cycle, the r we find should satisfy that $\Sigma_{(u,v)\in C} l_{u,v}$<0. Then r should be the worst condition for r^* and in the loss factor graph there should exist at least one negative weight cycle. We can say this is a lower bound of r^*.

3. **Strictly Positive Weight Cycle**

In this cycle, the r we find should satisfy that $\Sigma_{(u,v)\in C} l_{u,v}$. Then r should be the best condition for r^* and it has $\sigma_{(u,v)\in C} l_{u,v} > 0$. We know that r^* is the best maximum profit-to-cost ratio in the graph. Thus, r can be the upper bound of r^* if we choose the cycle with strictly positive weight.

And thus we can provide a efficient boundary to determine whether $r^* > r$ or $r^* < r$. The key method is to find the accurate r^* and prove that it has the real cycle in the graph. We try to use loss factor to directly display the degree of similarity between two graphs. It means, if the graph that is full of loss factor has zero weight cycle, we have the answer! If not, we should adjust our prediction of r^* and continue the judgement.

(b) Actually, the best way to find the best ϵ condition in the given bound is to use the **Divide and Conquer** algorithm. Our goal is to find the r^* and get the cycle based on our r^*. Thus we firstly do binary search to "learn" from the graph and adjust our r^*. The key problem is to judge zero weight cycle, negative weight cycle and strictly positive cycle. The main codes are presented below. Actually, the provided R is the upper bound of r^*.

The main structure of the function goes like:

1. **Initialize**

   - l_pointer = 0, r_pointer = R, mid_pointer = (l+r)/2, r^*;

2. **Binary search**

   - Set $r^* = mid\_pointer$ and build the loss graph $l(u,v) = r^* cost(u,v) - p(v)$

   - Check whether the loss graph is of zero_weight or negative_weight.

3. **Update l_pointer and r_pointer**

   - If the graph is of zero_weight or $\epsilon \leq mid - r^* \leq \epsilon$, end the function.

   - If the graph is of negative_weight, l_pointer = r^*. Otherwise, r_pointer = r^*.

   - Repeat Binary search function.

```
# The prev array is used to show a cycle.
# The parameter ans_end is the last vertex in the cycle we detected.
int prev[maxn], ans_end;
double binary_search(){
    double l = 0, r = R, mid = (l+r)/2, r_star;
    do{
        r_star = mid;
        for(int u=1;u<=|V|;u++)
            for(int v=1;v<=|V|;v++)
                # update each value for given r_star
                w(u,v) = r_star * cost(u,v) - p(v)
        # zero_weight function is used to judge whether it has zero weight
                                                              cycle.
        if(Zero_weight())
            return make_pair(r_star, ans_end);
        # Negative_weight: r_star is small, we should make it larger.
        if(Negative_weight())
            l = r_star;
        else
            r = r_star;
        mid = (l+r)/2;
    }while(abs(mid-r_star)>epsilon);
}
bool Negative_weight(){
    # This is the same as the programming problem 1344
    # I just copy the codes and slightly modify it.
    # This graph is stored as adjlinked( adjmatrix is almost the same)
    # We judge the negative cycle by Bellman-ford Algorithm.
    memset(dis, 0x3f3f3f3f, sizeof(dis));
  for(int i=1;i<n;i++)
    for(int j=0;j<m;j++)
      if(dis[edge[j].to] > dis[edge[j].from] + edge[j].value){
        dis[edge[j].to] = dis[edge[j].from] + edge[j].value;
        pre[edge[j].to] = edge[j].from;
      }
  bool flag = false;
  for(int j=0;j<m;j++)
    if(dis[edge[j].to] > dis[edge[j].from] + edge[j].value){
      flag = true;
      dis[edge[j].to] = dis[edge[j].from] + edge[j].value;
      pre[edge[j].to] = edge[j].from;
      # ans_end is updated in the last method.
      ans_end = edge[j].to;
      break;
    }
    return flag;
}
```

For Zero_weight judgement, I firstly build a new graph $G' = (V, E')$, where all the edges should satisfy that $\forall cost'(u,v) \in E', cost'(u,v) = cost(u,v) - 1$. And it means that if we find all the negative cycle, we can just calculate the weight and length of the negative cycle. For example, if the cycle goes like:

$$v_1 - > v_2 - > v_3 - > ... - > v_n - > u - > v_1$$

We can find the cycle and calculate its length is $length = n + 1$ and if its total weight $cost'(v_n, u) + cost'(u, v_1) + \Sigma_i cost'(v_i, v_i + 1) = -(n+1)$ then we find the zero_weight cycle because we added $(n+1)$ value and when it's not added, the last is 0.

The main structure to find the zero_weight graph goes like:

1. **Initialize**

   - $dis[i] = INF$ for all $v \in V$.

   - $w(s,v) = w(s,v) - 1$ for all $(s,v) \in E$,

2. **Explore for n times**

   - $tdist[v] = min\{tdist[v], tdist[u] + w(u,v)\}$ for all $(u,v) \in E$.

   - If $tdist[u]$ is updated, then $pre[v] \leftarrow u$

3. **Record the precursor node for the extra exploration**

   - $tdist[v] = min\{tdist[v], tdist[u] + w(u,v)\}$ for all $(u,v) \in E$.

   - If $tdist[u]$ is updated, then $pre[u] \leftarrow v$ and there must be **at least one negative cycle**.

4. **Calculate weight of all cycles and check zero_weight**

   - Find a vertex that is not visited and explore its precursor node until visited or end.

   - Calculate the weight if it's a cycle. End if it's not a cycle.

   - If $weight == length$, we find a correct zero_weight cycle, end the function.

   - Otherwise, repeat until all the nodes are visited.

```
# the vis array is used to help find the negative cycle.
bool vis[maxn];
bool Zero_weight(){
    memset(dis, 0x3f3f3f3f, sizeof(dis));
    # Subtract 1 for each edge.
    for(int i=0;i<m;i++)    edge[i].value -= 1;
  for(int i=1;i<n;i++)
    for(int j=0;j<m;j++)
      if(dis[edge[j].to] > dis[edge[j].from] + edge[j].value){
        dis[edge[j].to] = dis[edge[j].from] + edge[j].value;
        pre[edge[j].to] = edge[j].from;
      }
  bool flag = false;
  # Update this cycle for the last time.
  # We will find the cycle in the next time.
  for(int j=0;j<m;j++)
    if(dis[edge[j].to] > dis[edge[j].from] + edge[j].value){
      flag = true;    pre[edge[j].to] = edge[j].from;
      dis[edge[j].to] = dis[edge[j].from] + edge[j].value;
    }
  # find cycle and calculate the weight.
  for(int i=1;i<=n;i++){
      int weight = 0, cnt = 0, tmp = i;
      if(!vis[tmp]){
          weight += graph[pre[tmp]][tmp];
          vis[tmp] = true;    tmp = pre[tmp];     cnt ++;
      }
      # This is the last edge of the cycle
      weight += graph[pre[tmp]][tmp];     tmp = pre[tmp];     cnt ++;
      # Length = (n+1) while total weight = -(n+1) and the sum is 0
      if(weight + cnt ==0){   flag = true;    break;  }
  }
    return flag;
}
```

Above all, finding the negative weight cycle and zero weight cycle both have the complexity of $O(|V||E|)$ and the number of binary search iteration is $n = \log(R/\epsilon)$. Then the final complexity is $O(|V||E|\log(R/\epsilon))$.

Problem 5 (20 points)

Consider if we want to run Dijkstra on a bounded weight graph $G = (V; E)$ such that each edge weight is integer and in the range from 1 to C, where C is a relatively small constant.

(a) (10 points) Show how to make Dijkstra run in $O(C|V| + |E|)$.

(b) (10 points) Show how to make Dijkstra run in $O(\log C(|V| + |E|))$. Hint: Can we use a binary heap with size C but not $|V|$?

Proof.

(a)

The complexity of the origin Dijkstra Algorithm without optimizing methods is $O(|V|^2 + |E|)$. The main reason for the large factor $|V|^2$ is that the cost to find the minimum in the simple array is $|V| - i$ in the $i - th$ round and leads to $|V|^2$. Thus, if we want to simplify the complexity and decline it from $|V|^2$ to $C|V|$, we should optimize the minimum choosing strategy.

The idea on how to make this goal come true is to explore the distance at the total complexity $O(C|V|)$. Notice that the largest distance is at most $C * (|V| - 1)$. Thus, we can build $C * (|V| - 1)$ baskets and iterate them from 0 to $C * (|V| - 1)$. Each basket contains the vertex that can get to the distance. Other methods are totally the same as Dijkstra Algorithm.

The main structure of the new Dijkstra function goes like:

1. **Initialize**

   - $tdist[s] = 1, tdist[v] \leftarrow \infty$ for all v other than s,

   - Set $C(|V| - 1)$ queues and push the start node s in the zero basket.

2. **Explore the baskets until the end**

   - If the basket we visit now is empty, continue to find the next basket.

   - Select one node v in the basket that isn't visited.

   - $tdist[u] = min\{tdist[u], tdist[v] + w(v, u)\}$ for all $(v, u) \in E$.

   - Push the updated $tdist$ into the commensurate basket.

```
void new_dijkstra(){
    # initialize the basket;
    queue<int>basket[C*(|V|-1)];
    # initialize the distance;
    for(int i=1;i<=n;i++)
        dis[i]=214748364700000;
    # initialize the distance array and basket array.
    dis[s]=0;   basket[0].push_back(s);     int idx = 0;
    # end the recursion until the basket is empty.
    while(true){
        # find the first non-empty basket to explore.
        while(basket[idx].empty() && idx < C*(|V|-1)) idx++;
        # end the recursion until the basket is empty.
        if(idx == C*(|V|-1)) break;
        # take the value u in the basket.
        long long u = basket[idx].front();
        basket[idx].pop_front();
        # explore all the edge from the vertex u
        for(int i=head[u];i;i=e[i].next){
            long long v=e[i].v;
            if(dis[v]>dis[u]+e[i].w){
              dis[v]=dis[u]+e[i].w;
              # push the update vertex into the basket.
              basket[dis[v]].push_back(v);
            }
        }
    }
    # This is the result of all the smallest distance.
    for(int i=1;i<=|V|;i++)
        printf("%lld \n", dis[i]);
}
```

And the total complexity cost is $O(C|V| + |E|)$ because we explore and update totally $O(|V| + |E|)$, finding minimum $O(1)$, and creating buckets has the same cost: $O(C|V|)$.

(b)

We find that the choose method costs much time then we think. Thus, if we can optimize the selection method, the result must be good. We try to build one heap that has the capacity of C and it contains C nodes. The nodes have the property:

```
struct node{
    int flag; int dist;
    inline bool operator <(const node &x) const{
        return dist>x.dist;
    }
}
```

We know that there are at most $C|V|$ nodes in the total extension path and therefore if we do the mod operation, we can set these $C|V|$ nodes into C baskets and for each baskets, we can ensure that if we choose nodes in order to expand, the distance we choose must be from large to small.

To quickly find the smallest element in these baskets, we use one heap to optimize the selection method. Put nodes into the heap and the top of the heap contains the shortest distance and the places it goes to. For example, if the top node of the heap is $flag = 0$ and $dist = A$ like the figure presented below, we can update this node and push one new node at the first item of $\{\%c = 0\}$ basket. Thus, the result leads to optimizing $\log|V| \to \log C$ in each iteration in Dijkstra Algorithm (Note that for each iteration there can be at most C elements in the heap as the picture presented below). We know that the original Dijkstra Algorithm has the time complexity of $O((|V| + |E|) \log |V|)$, Therefore, the time complexity of our new method is $O((|V| + |E|) \log C)$

To build the priority of the heap, we should firstly prove that there are at most C buckets in the heap. The proposal can be changed by **Bucket distances in the heap must satisfy that max_dist $-$ min_dist $\leq C - 1$ by the induction method.**—-This idea is provided by Jingxiang Li.

1. Induction Base

   At first, the values are in the interval $[1, C]$ and thus it must satisfy that there are at most C buckets in the heap. max_dist $-$ min_dist $\leq C - 1$

2. Induction Suppose

   We find that there are at most C buckets in the first iteration. Suppose there are at most C buckets in k-th iteration.($k \geq 1$). Our goal is to prove that **there are at most C**

buckets in (k+1)-th iteration.

3. Induction Step

   We know that for the k-th iteration it must satisfy that $\text{max\_dist} - \text{min\_dist} \leq C - 1$. For the (k+1)-th iteration, we choose the smallest dist in the heap. And when the smallest is poped, the smallest value in the heap must satisfy $\text{new\_min\_dist} \geq \text{min\_dist} + 1$ Extend the node in the heap and we can get that the value range of the node extension must be in the interval $[\text{new\_min\_dist}, \text{min\_dist} + C] \in [\text{min\_dist} + 1, \text{min\_dist} + C]$ and it is ovbisouly also true that $\text{new\_max\_dist} - \text{new\_min\_dist} \leq C - 1$.

4. Conclusion

   Thus, there are at most C buckets in each iteration.

Knowing the property, we can analyze the time complexity easily. Choose buckets: $|V|$ times with $\log C$ costs leading to $O(|V| \log C)$. Update edges: $|E|$ times with $\log C$ costs, leading to $O(|E| \log C)$ at most (Actually, this is the upper bound of the edge operation complexity). Above all, the total time complexity is:$O(\log C(|V| + |E|)$.

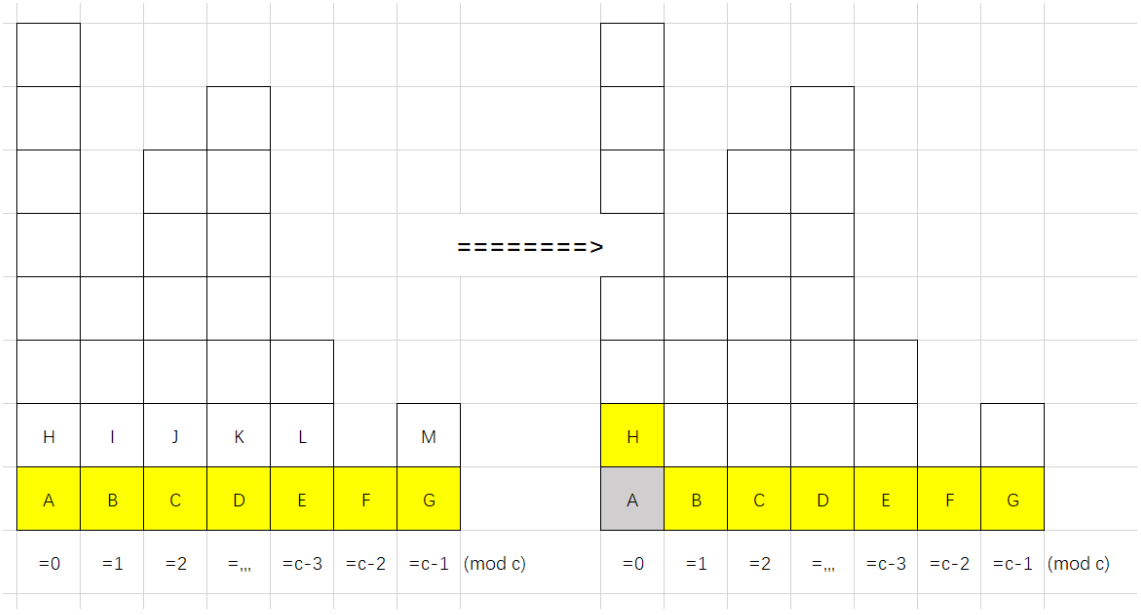


Figure 3: Explanation for the selection method.

Problem 6

How long does it take you to finish the assignment (include thinking and discussing)? Give a score (1,2,3,4,5) to the difficulty. Do you have any collaborators? Write down their names here.

This homework is harder than I think. I costs 8 hours to finish this assignment.

The difficulty score: **5**.

Collaborators: **Jingxiang Li** in Problem 5(b).

Suggestion: can you give one ".tex" format file to us? We can finish our homework easier.