CST Part IA: Algorithm, SV 6,7,8
Joe Yan
2017-2-24

1 1997P9Q6

A Fibonacci heap is a collection of rooted trees which obey the min-heap property. This means in each rooted tree, the key in the parent node is greater than the key in child nodes under the assumption that all keys are different.
In detail, a Fibonacci heap structure should have a reference (H.min) to the node with the smallest key in the collection of all roots and an integer (H.n) states the number of roots or trees in the whole Fibonacci heap.
Each nodes has four references to its parent, one node in the child list, right and left siblings. Two auxiliary attributes in each node are needed for maintaining the Fibonacci heap property. They are a boolean value (x.mark) states whether the node recently lost its child and an integer (x.degree) states the number of nodes in the child list. Also each node has its key and value.
All roots of trees should have a null in their parent reference because they do not have parents. All leaves of trees should have a null in their child reference because they do not have any children.
A node has themselves as left and right siblings if it is the only node in that child list or root list.

(a) Let the algorithm initialize a Fibonacci heap called H.

```
H.min = null
H.n = 0
```

(b) Let the algorithm inserting the key, value pair into a Fibonacci heap called H.

```
If key = (H.min.key)
—H.min.value = value (duplicated key, overwrite its value)
—return
x = new node (key,value)
x.p = null
x.child = null
x.mark = false
x.degree = 0
If H.min == null
—x.left = x
—x.right = x
—H.min = x
Else
—x.left = H.min (inserting the node x to the right of the min-node)
—x.right = H.min.right
—H.min.right.left = x
—H.min.right = x
—If H.min.key > key
——H.min = x
H.n = H.n + 1
```

(c) Let the algorithm return a new Fibonacci heap which is the union of the Fibonacci heap A and B.

```
If A.min == null and B.min != null
—return B
```

```
If B.min == null
—return null
tBminL = B.min.left
A.min.left.right = B.min (concatenate two root double linked lists of C and B)
B.min.left = A.min.left
A.min.left = tBminL
tBminL.right= A.min
If A.min.key > B.min.key
—A.min = B.min
A.n = A.n + B.n
```

(d) Let the algorithm return the min node of the Fibonacci tree H.
return H.min (It just say identify?)

The amortized time cost of each of them is O(1).

2 1993P4Q8

The idea of the algorithm is redefining the matrix multiplication for boolean matrix.

$$G_{2ij} = (G_{1ik} \& G_{1kj}) | G_{1ij}$$

$G_{1ik} \& G_{1kj}$ is the suffix notation replacing multiplication with AND and addition with OR.

```
boolean[][] generateNext(boolean[][] G1) {
        int size = G1.length;
        boolean[][] G2 = new boolean[size][size];
        for (int i = 0; i < size; ++i) {
                for (int j = 0; j < size; ++j) {
                        if (G1[i][j] == true) {
                                G2[i][j] = true;
                                continue;
                        }
                        for (int k = 0; k < size; ++k) {
                                if (G1[i][k] && G1[k][j] == true) {
                                        G2[i][j] = true;
                                        break;
                                }
                        }
                }
        }
        return G2;
}
boolean[][] generateTransitiveClosure(boolean[][] G) {
        int size = G.length;
        boolean[][] TC = G;
        boolean[][] nextTC;
        while (true) {
                nextTC = generateNext(TC);
                if (equal(TC, nextTC)) {
                        return TC;
                }
                TC = nextTC;
        }
}
boolean equal(boolean[][] A, boolean[][] B) {
        int size = A.length;
        for (int i = 0; i < size; ++i) {
                for (int j = 0; j < size; ++j) {
                        if (A[i][j] != B[i][j]) {
                                return false;
                        }
                }
        }
        return true;
}
```

The time cost of the function generateNext is n^3 in the worst case.
Using Strassen's method the time cost can be improved to $O(n^{\log_2 7})$
Notice G_1 contains all possible paths with length ≤ 1.
So it follows that G_t contains all possible paths with length $\leq 2^{t-1}$.
Solve $m \leq 2^{t-1}$ gets $t = \lceil log_2(m) + 1 \rceil$
The function generateTransitiveClosure calls $\lceil log_2(m) \rceil$ times of the function generateNext to generate G_n.
Because $m \leq n$ so the function generateTransitiveClosure calls $\lceil log_2(m) \rceil$ times of the function generateNext.
Overall the best time complexity of the function generateTransitiveClosure can be $O(n^{\log_2 7} \times \log m)$

3 2006P5Q1

(a) The algorithm initialises all the vertices by giving them an infinity distance from source and sets the distance of the source node to 0. Then the algorithm construct a priority queue containing all nodes and using the distance as keys.
While the priority queue is not empty, the algorithm keeps popping the node in the priority queue with the smallest distance from the source node and relax it with all its adjacent reachable nodes. After the priority queue is empty, the algorithm terminates and the distance of each node is the shortest distance from the source node.
If a node is not reachable from the source node, it will not be popped until the very last. So when it is popped, the distance of it will still be infinity which means this node is not reachable from the selected source.

(b) Claiming that when the node is popped from the priority queue the distance of the node is shortest distance from the source.
Proof:
Consider such shortest path from s to v. (Otherwise v is not reachable from the source.)
$s \rightsquigarrow u \rightarrow w \rightsquigarrow v$ All nodes in $s \rightsquigarrow u$ is already popped. All nodes in $w \rightsquigarrow v$ is not popped. And node v is going to be popped next. We need to prove $v.d = \delta(s, v)$.
Prove by contradiction: assuming there is a node v such that $\delta(s, v) < v.d$.
By the node v is popped earlier than the node w.
$v.d \leq w.d$
By when u is popped, the relaxation of u and w gives the upper bound of w.d.
$w.d \leq u.d + weight(u, w)$
By the node u is correctly popped i.e. $u.d = \delta(s, u)$.
$u.d + weight(u, w) = \delta(s, u) + weight(u, w)$
By $s \rightsquigarrow u \rightarrow w$ is a part of the path $s \rightsquigarrow u \rightarrow w \rightsquigarrow v$.
$\delta(s, u) + weight(u, w) \leq \delta(s, v)$

Overall $\delta(s, v) < \delta(s, v)$ which is a contradiction.
By this contradiction, $\delta(s, v) \geq v.d$.
Because of all edges have positive distance and the definition of the shortest path $v.d \geq \delta(s, v)$.
Finally, $v.d = \delta(s, v)$, done.
I guess another proof that using the convergence lemma is by maths induction.
Proof:
We need to prove that when the element is popped from the priority queue, its distance is the shortest distance from the source.
Base case: After the graph is initialised, the source vertex has distance 0 which is the shortest distance from source to source and all other vertices have infinity distance. So the first element gets popped holds the property.
Induction case: Assume the next vertex get popped is v.
First case, if v is not relaxed by any vertex then v has distance infinity and also it has smallest distance in the priority queue. That means it is not reachable from the source. So infinity is the shortest distance as definition.
Second case, if v is relaxed by another vertex u and $s \rightsquigarrow u \to v$ is the shortest path from the source to v. As u is popped earlier, when u gets popped, the distance of u is the smallest distance from source by the strong induction assumption. Then the algorithm relaxes all vertices adjacent to u including v. By convergence lemma, the distance v is the shortest distance after the relaxation. So the property holds for v by the strong induction assumption that all vertices got popped earlier has the distance which equals to the shortest distance from the source.
Overall by strong maths induction, all vertices get popped from the priority queue have the shortest distance from the source as we want.

(c) The Dijkstra's algorithm works because of the assertion that when the algorithm pops the node from the priority queue, the distance of the node is the shortest distance from the source the the node which is proved in (b).
Consider a graph with vertices A B C. $w(A, C) = 1$, $w(A, B) = 2$, $w(B, C) = -3$.
Following the instruction, the algorithm will pop A, C then B which gives $\delta(A, C) = 1$,$\delta(A, B) = 2$. This is absolutely wrong because $\delta(A, C) = -1$.
By this counter example we can see the assertion can be broken with negative weight because it is possible to add a negative weighted edge to a path after popping the target vertices and make the path shorter.

(d) No. By counter example that $weight(A, B) = -2$ and $weight(B, A) = 1$.
There is no shortest path from A to B so Dijkstra's algorithm even does not work at all.

(e) No. Consider the same example as (c). Although we added the same weight to each edge but each path contains different numbers of edges so the shortest path in the modified graph may be different from the previous graph.

4 1997P3Q6

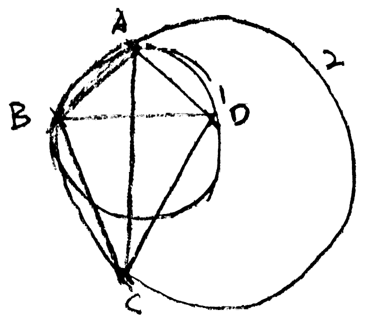
(a) Kruskal's algorithm finds the minimum spanning tree in a graph by connecting unconnected trees (represented as sets) in a forest. At the beginning, the collection of sets contains all separated vertices as singletons. The algorithm runs a loop for each edges in a non-decreasing order (sorting is needed) which checks whether the edge connects two vertices in different sets. If it does then add the edge to the minimum spanning tree. The algorithm terminates with a single minimum spanning tree. Otherwise there is no possible minimum spanning trees in the graph.

(b) (i) It is based on the structure of the given graph. Let set A and B such that $A \cap B = \emptyset$ $A \cup B = V$ where V is the set of all vertices in the graph. Define a edge connecting one vertices in A and one vertices in B a edge across the cut between set A and B. If we remove any current longest edges as far as each cut has at least one edge across them then Kruskal's algorithm will still give the correct answer. Because the Kruskal's algorithm exams all edges from the shortest to the longest in order. So if we remove all edges longer than L and the Kruskal's algorithm still terminates with only one spanning tree, then this spanning tree is guaranteed to be the minimum one.
Otherwise, we have removed all edges on any one cut and Kruskal's algorithm still terminates but it will leave several disconnected minimum spanning tree due to too many removed edges.

(ii) The question say if the algorithm had given the correct result so i assume the check happens after the algorithm is finished. If the cardinality of the edges in the minimum spanning tree is equal to the number of vertices - 1 then the result must be correct. Otherwise it is not as argued in (i).

(iii) Yes and this is argued in (i).

(iv) Starting from something trivial, if we simply calculate all edges between all vertices which takes $O(n^2)$. Later the sorting will take $O(n^2 \log n)$ which is the first bottleneck of the asymptotic time complexity.
The following is all guessing...
First of all, for any three points which always construct a triangle, the longest side must not be in the minimum spanning tree by the definition of tree that there is no



cyclic paths.
Based on this idea, if there are four points like A,B,C,D in a plane, then we need 3 edges out of 6 to make a minimum spanning tree. By the previous discussion, we know AC which is the longest side must not be in the minimum spanning tree because we know there is at least three edges pass each cut so removing AC will not remove one shortest edge across some cut. So the minimum spanning tree must be in the $\triangle ABD$ and $\triangle CBD$.
As we add the next point E into the graph, we add four edges to the graph and consider again exactly like previous process to remove some edges and make the graph again a triangulation and still the minimum spanning tree is a subset of this graph.
Keeping doing so until the last vertex.
Here I guess this step can be done in $O(n^2)$ in this way or may be better in $O(n \log n)$ by some technic such as divide and conquer but I do not know how....
Then the algorithm still need to sort all rest edges and we need to know the upper bound of the number of edges to see whether this optimise the asymptotic cost of sorting.
Image a infinite triangulation graph, each edge is shared by two triangle so $1.5f = e$.

By Eulers formula $n - e + f = 2$ (guess this works for infinity case?) we have $n = e/3 + 2$ so $e = 3n - 2$ if the triangulation graph is infinity. And if we cut a part of the triangulation graph from the infinity version to get a finite version we remove more edges than vertices. So $e = O(3n) = O(n)$.
So the kruskal's algorithm now costs $O(n/logn)$. Overall the algorithm costs $O(n^2)$ which is slightly better than $O(n^2 \log n)$ by the trivial method.

5 1998P3Q5

(a) First check whether the point on the polygon. If it is then it is in the polygon because we are considering closed polygon.
If the point is not on the polygon then draw a half line which starts from the point. Then the algorithm will check the number of sides the half line intersects with the polygon. If this number is odd then the point is in the polygon. Otherwise the point is out of the polygon. This works because the half line starts from the point and must be out of the polygon finally because the half line goes infinitely far away from the starting point. So the half line must finally leave the polygon no matter how many times it leaves and re-enters the polygon previously. Each time the half line enters or leaves the polygon it will intersect with one side of the polygon. So the correctness holds.
Because this is a closed polygon. If the half line hit a vertices of the polygon then consider it enters and leaves or leaves and enters the polygon. This is naturally solved in the algorithm because a half line cross a vertices also intersects with two segmented line at their end.

(b) Assume the cardinality of the set of points is larger than three otherwise the question will be trivial.
One way to exclude some points which are impossible to be a candidate for the vertices of the convex hull is pick two points which have the largest or the smallest x coordinates.Then pick two points with the largest or the smallest y coordinates from the set exclude two points found previously. Now the algorithm will have four points which can form a quadrangle. The vertices of the convex hull can not be in the quadrangle with these four points. All points inside the closed quadrangle can be excluded. This idea may work very well if all points is uniformly separated in the space. But in some special case this idea will be useless if all points are away from the centre. One case this idea will be useless is that all points are on a circle. Because there can be no points in the quadrangle in this case.
Overall, how good this idea works is by luck.

(c) After (b) we will have a convex quadrangle and a set of points which are out of the quadrangle.
Then we can implement the same idea as the Graham's scan from one vertices of the quadrangle in clockwise or anticlockwise order.

6 2000P5Q1

(a) A graph G can be represented as $G = (V, E)$ where V is a set which contains all vertices and E is a set which contains all edges in the graph. In an undirected graph, E is a set of unordered pairs of vertices which represent edges i.e. (X,Y) is an edge representing either from X to Y or from Y to X. In a directed graph, E is a set of ordered pairs of vertices which represent directed edges i.e. (X,Y) is an edge strictly representing from X to Y instead of from Y to X.
A (undirected) bipartite graph can be represented as $G = (U, V, E)$ where U and V are

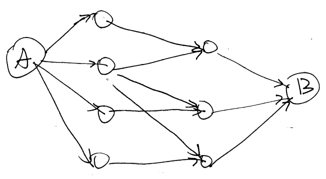
two disjoint sets which contain vertices in the bipartite graph. E is still a set of edges but the element in set E must be a(n) (unordered) pair of vertices where these two vertices are one in set U and one in set V.

(b) For a graph $G = (V, E)$, a matching is a set $M \subseteq E$ and for all edges in the set M do not share any common vertices.
An augmenting path is an alternating path that starts from and end with unmatched vertices.
An alternating path is a path that starts from an unmatched vertex and the adjacent edges can not be the same in concept of whether being matched. i.e. UNMATCHED-MATCHED-UNMATCHED-MATCHED... For a bipartite graph $G = (U, V, E)$, the definitions are the same as the definitions for graph.

(c) Proof by contrapositive: If the matching is not maximal then there will be at least one augmenting path.
The matching M is not maximal means there is at least one edge can be added to set M and set M is still a matching. Then this edge must have two unmatched vertices. This edge is an augmenting path by definition. So there is at least one augmenting path. the contrapositive statement is proved. Done.
Instead of maximal matching, I guess what the question really means is maximum matching. Otherwise the proof looks really short and doing nothing. Maybe I was totally wrong by messing up with definitions... I thought a maximum matching is the largest possible cardinality for the set M which is a matching.
Proof by contrapositive for maximum matching: If the matching is not maximum then there will be at least one augmenting path.
Because the matching is not maximum so there will be at least two vertices to be unmatched by the definition that all edges in the matching do not share any common vertices. Otherwise it is a perfect or near-perfect match which must be a maximum match.
Then we want to prove there is at least one augmenting path in such graph. And by definition, an augmenting path must start from and end at unmatched vertices.
Prove by contradiction, assuming there is no augmenting path.
Use the idea of maths induction:
Without lose of generality, starting from an unmatched vertex as a base case and constructing all possible path from the vertex.
Induction case 1 is that if the previous edge is in the set M then the next edge in the path must be not in the set M by the definition of matching.
Induction case 2 is that if the previous edge is not in the set M then the next edge in the path must be in the set M otherwise we reach a unmatched edge again and so we have another unmatched vertex and we find an augmenting path.
By maths induction and the definition of alternating path, all paths from all free vertices in such graph are alternating path. Because we assumed that there is no augmenting path so all such alternating paths are non-augmenting. i.e. The alternating path starts from an unmatched vertex and ends at an matched vertex.
This is a contradiction with the assumption that this is not a maximum matching. Because we just proved that all such unmatched vertices are not reachable from each other with a alternating path and so augmenting path which means the graph can be separated to several near-perfect or perfect matching which means for each part of the separated graph. There is either no or just one unmatched vertex in each separated part of the graph. So all these separated parts are maximum matched. Putting all these separated parts together back to our origin graph makes it still maximum matched because there is no way to increase the cardinality of set M. So the contradiction is reached and by the contradiction there is at least one augmenting path in the graph.

(d) Because the augmenting path concept exists so Ford-Fulkerson algorithm should be a

good idea for the required algorithm.
At the beginning, the algorithm has a set of vertices which are unmatched. Now by the idea of Ford-Fulkerson algorithm that while there is an augmenting path in the graph, alternating all edges whether matched states on the path and then repeat. Because alternating all edges on an augmenting path will increase the cardinality of set M by one and the matching property will still holds. This algorithm will terminate when there is no more augmenting path in the graph which means the matching is maximum by the proof in (c).
In detail, to find an augmenting path, the algorithm will do a breath first search which restricts to alternating path, starting from a unmatched vertex and end with anther unmatched vertex.



Analysis of cost in a bipartite graph $G(U, V, E)$:
In the step to find a maximal matching we can first modify the graph to make Ford-Fulkerson algorithm easier to run by adding a source vertex A and drain vertex B and given all edges directional capacity 1 as the picture shows.
Then the algorithm will keep alternating the match on an augmenting path if there is one.
Each time the algorithm alternates the match on an augmenting path will increase the number of matched edges by 1. And we know there are at most $Min(|U|, |V|)$ and $|U| + |V| = n$. So the breath first search for an augmenting path will be run at most $n/2$ times.
Each BFS costs $O(n + e)$ to find an augmenting path and alternating the match on the path costs $O(e)$ so each iteration costs $O(n + e + e) = O(e)$ by $e > n/2$ assuming each vertex is contained in at least one edge (otherwise we can just remove it from the graph, there is no difference for the result of the maximum matching).
Overall, it takes $O(ne)$ to compute one possible maximum matching in a bipartite graph.