Algorithm HWIV

B06702064 會計五 林聖硯

No collaborator

Problem1

參考資料: **https://walkccc.me/CLRS/Chap23/Problems/23-4/** (https://walkccc.me/CLRS/Chap23/Problems/23-4/), **https://stackoverflow.com/questions/60887403/mst-reverse-delete-algorithm** (https://stackoverflow.com/questions/60887403/mst-reverse-delete-algorithm)

Algorithm a.

This is just the reverse-delete algorithm that the professor (Iris Hui-Ru Jiang) taught in class. To prove this algorithm can product MST, we need to show that we won't remove any edge in the correct MST. Consider any edge removed by the algorithm, say e. It can be observed that e must lie on some cycle (otherwise removing it would disconnect the graph) and since we remove edges in nonincreasing order, the weight of every edge on the cycle must be less than or equal to e. From the result of exercise 23.1-5, there's a MST on G with edge e removed. Thus, this alogrithm can correctly produce MST.

Implementation:

Sorting the edged costs $O(ElgE)$. (line 1)

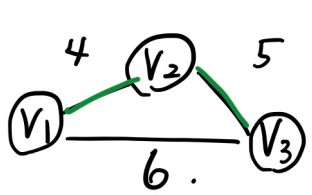
In order to check for the connectivity of $T-e$, we need to run DFS to find the existence of path, which costs $O(V+E)$.

Thus, the total time complexity is $O(ElgE)+O(E(V+E))=O(E^2)$

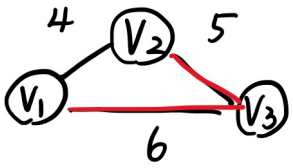
Algorithm b.

This doesn' producce a MST.

Counter example:



— MST



If the algo picks (v_2, v_3), then (v_1, v_3), it won't output the correct answer.

Implementation:
To make sure that we won't form a cycle, we can use disjoint sets data structure and corresponding FIND-SET operation to check. In the begging of the algorithm, we make $|V|$ calls of MAKESET operation. Then, from line 2 to line 4, we at most make $|E|$ calls in both FIND-SET and UNION operations, which costs $O(E\alpha(V))$. Thus, the total time complexity is $O(V) + O(E\alpha(V)) = O(E\alpha(V))$
Algorithm c.
This algorithm produces MST. From the result of exercise 23.1-5, we know that if we revome maximum-weight edge on some cycle of connected graph, the MST still exists in the graph without the maximum-weight edge. In addition, if we remove an edge from every cycle in this algorithm, the resulting graph must not contain any cycle, and thus it must be a tree.
Implementation:
We can use DFS to find both the cycle and the max weight edge at the same time (just use a variable to record the max weight edge DFS passed) and also use the alogrithm in subproblem (b) to solve this problem. Moreover, there will be at most one cycle in each loop, which means that there will be at most $|V|$ edge in the tree. Thus, we can run DFS in $O(V)$ time. In conclusion, the total time complexity is $O(EV)$

Problem2

參考資料: https://walkccc.me/CLRS/Chap24/24.2/ (https://walkccc.me/CLRS/Chap24/24.2/), https://www.geeksforgeeks.org/find-longest-path-directed-acyclic-graph/ (https://www.geeksforgeeks.org/find-longest-path-directed-acyclic-graph/)

If we have a PERT chart $G = (V, E)$ with weights on the vertices and we have some ways to transform it back to a PERT chart $G' = (V', E')$ with weights on edges. In addition, if the way satisfy $|V'| \leq 2|V|$ and $|E'| \leq |V| + |E|$, we can negate the edge weights and run the same algorithm(DAG-SHORTEST-PATHS) to find a longest path in DAG.
Thus, I propose way to transform $G = (V, E)$ to $G' = (V', E')$. Leave the vertices in V and add one new source vertex s to V', where $V' = V \bigcup s$. All edges of E are in E' and E' includes an edge (s, v) for each $v \in V$ that has in-degree 0 in G. Thus, the only vertex with in-degree 0 in G' is the new sources s. We then turned the weight of vertex v in G into the weight of edge $(u, v) \in E'$. Therefore, we have the weight of each entering edge in G' that is the weight of the vertex it enters G.

Problem3

參考資料: https://walkccc.me/CLRS/Chap24/Problems/24-2/ (https://walkccc.me/CLRS/Chap24/Problems/24-2/)

(a)
Suppose that box $x = (x_1, \ldots, x_d)$ nests with box $y = (y_1, \ldots, y_d)$. (i.e. $x_{\pi(1)} < y_1, \ldots, x_{\pi(d)} < y_d$) In addition, box y nests with box $z = (z_1, \ldots, z_d)$ (i.e. $y_{\sigma(1)} < z_1, \ldots, y_{\sigma(d)} < z_d$). Therefore, $x_{\pi(\sigma(1))} < z_1, \ldots, x_{\pi(\sigma(d))} < z_d$. Thus, we can conclude that nesting relation is transitive.

(b)

To determine whether or box x nests inside box y. We can sort both sequences and compare along each array to check that whether all elements in box x are less than some element in box y. The sorting operation costs $O(dlgd)$ and the comparison operation costs $O(d)$. Thus, the total time complexity is $O(dlgd)$.

(c)

We can created a directed graph with vertices $B_1, \ldots, B_n$ as follows to efficiently solve this problem. For each pair of boxes B_i, B_j, if B_i nests in B_j, we create an edge from B_i vertex to B_j vertex, vice versa.

The complete algorithm is as follows.

1. Sort all sequences, which costs $O(nd \lg d)$
2. Compair all pairs of boxes using the algortihm in (b), which costs $O(C_2^n d) = O(n^2 d)$
3. The resulted graph is acyclic. Thus, we use the DAG-Longest-Path algorithm in problem2 to find the longest path, which costs $O(V + E) = O(n^2)$

The whole graph creation process costs $O(nd \lg d) + O(n^2 d) + O(n^2)$ = $O(nd \max(\lg d, n))$

Problem4

參考資料: https://sites.math.rutgers.edu/~ajl213/CLRS/Ch25.pdf (https://sites.math.rutgers.edu/~ajl213/CLRS/Ch25.pdf)

To be sure we get the same results from this algorithm, we need to check what happens to $d_{ij}^{(k-1)}$, $d_{ik}^{(k-1)}$ and $d_{kj}^{(k-1)}$.

In the original Floyd-Warshall algorithm, when updating $d_{ij}^{(k)}$, we only use the results from $D^{(k-1)}$ (which apparently shows in the original recurrence formula), thus d_{ij}^{k-1} will be unchanged. On the other hand, the other two terms won't be changed since any shortest path from i to k, which includes k, necessarily includes it only once (since

there are no negative-weight cycles, and it is thus the length of a shortest path using only vertices 1 through $k-1$. Thus, updating in place is correct and we only need $\Theta(n^2)$ since no new space will be created during the iteration.

Problem5

參考資料: https://walkccc.me/CLRS/Chap25/Problems/25-1/ (https://walkccc.me/CLRS/Chap25/Problems/25-1/)

(a)

Assuming the we add an edge (v_1, v_2) into the graph. Consider every pair of vertices (i, j), if there's a path from i to j after adding the new edge, there should be a path from i to j and a path from j to v_2. Thus, we only need constant time to determine whether there exists a new path from i to j with new edge. Therefore, the total time complexity is $O(C_2^V) = O(V^2)$

(b)

Assume that we have two disjoint subgraphs each of size $\frac{|V|}{2}$. In addition, assuming that each subgraph is complete (i.e. there exists edges between every pair of vertex (v_i, v_j) for $i, j \in V$ and $i \neq j$). If we add an edge from a vertex in the first subgraph to the other subgraph, we will need to add edges in the transitive closure from every vertex in the first subgraph to every vertex in the second subgraph, which is $\frac{|V|}{2} \times \frac{|V|}{2}$ edges. Therefore, no matter which algorithm we use, $\Omega(V^2)$ time is required.

(c)

Notice that we can only update the transitive closure which use the edge (u, v). That is, we can only update the vertices which have paths to u but doesn't have paths to v. Assume we insert an edge (u, v), the following algorithm helps us to find update the transitive closure.

```
UPDATE-TRANSITIVE-CLOSURE(u, v)
    for i = 1 to |V|
        if T[i, u] = 1 and T[i, v] = 0
            for j = 1 to |V|
                if T[v, j] = 1
                    T[i, j] = 1
```

Now, I'm going to show this procedure takes $O(V^3)$ time to update the transitive closure for any sequence of n insertions.
Observe some facts:

- There cannot be more than $|V|^2$ edgeds in G, thus $m \leq |V|^2$
- Time complexity for the first two lines over n insertions is $O(nV) = O(V^3)$
- The last three lines execute only $O(V^2)$ times for n insertions since there are executed only when $T[i, v] = 0$. In this case, the last line sets $T[i, v] = 1$. Therefore, the 0 elements in the transitive closure is reduced by at least one times each time the last three lines are executed. Thus, these lines can run at most $|V|^2$ times.

In conclusion, the total running time over n insertions is $O(V^3)$.

Problem 6

參考資料: http://alogtm.blogspot.com/2012/02/100-escape-problem.html (http://alogtm.blogspot.com/2012/02/100-escape-problem.html), https://walkccc.me/CLRS/Chap26/Problems/26-1/ (https://walkccc.me/CLRS/Chap26/Problems/26-1/)

(a)
In order to handle the additional vertex capacity constraints, we can reduce a flow network $G = (V, E)$ with vertex and edge capacities to a flow network $G' = (V', E')$ with only edge capacities as follows.
Split each vertex v with capacity c into two vertices v' and

v''. Then we construct an edge between v' and v'' with **edge** capacity c.
In this way, we can make sure that the vertex capacity constraints in $G = (V, E)$ can be included in G' since all edges going to v are now going to v' and the edges coming out of v are now coming out of v''. Therefore, any flow going through vertex v with capacity c in G must go through the edge (v', v'') in G'. Thus, the maximum flow in a network with edge and capacities can be reduced to an ordinary maximum-flow problem on a flow network of comparable size.
The running time of the reduced opeartion is $O(V + E)$ since the flow network $G' = (V', E')$ have $|V'| = 2|V|$ vertices and $|E'| = |V| + |E|$ edges.
(b)
To solve the escape problem, we can reduce it to a maximum-flow problem with edge and vertex capacity constraints. We can construct a flow network G' as follows.

1. Set each grid point to be a vertex with capacity 1.
2. Create a bi-directional edge with capacity 1 between two adjacent grid points.
3. Create a source vertex s and create edges from t to each starting point with capacity 1.
4. Create a sink vertex t and create edges from t to each boundary point with capacity 1.

After reducing the problem to a maximum-flow problem with vertex and edge capacity constraints. We can then reduce the reduced problem into a **maximum-flow problem with only edge capacity** using method mentioned in (a). Now, we can apply the Ford-Fulkerson algorithm to find the augmenting paths, which should be the escape paths in the given grid since each vertex has

unit capacity and they thus contain disjoint vertices. Besides, all the augmenting paths have disjoint edges since the edges on flow graph has unit capacity. In conculsion, a solution to this maximum-flow problem with edge capacity contrained indeed correspond to a solution of the escape problem.
Now, we can analyze the running time of the algorithm.

1. Construct the flow network from $n \times n$ grid costs $O(n^2)$
2. Reducing the flow network with edge and vertex capacities to the one with only edge capacity costs $O(V + E)$ time.
3. The Ford-Fulkerson algorithm costs $O(CE)$ time to solve this problem, where C is the number of while loop execution.
4. Converting the solution of the maximum-flow problem to the corresponding escape problem costs $O(V + E)$ time.

Since $|E| = O(V) = O(n^2)$ and $C \leq 4n - 4$ (maximum number of boundary points). The total time complexity is $O(n^3)$.

Problem7

參考資料: **https://www.iitg.ac.in/deepkesh/CS301/assignment-2/2sat.pdf** (https://www.iitg.ac.in/deepkesh/CS301/assignment-2/2sat.pdf)
In order to show that 2-CNF-SAT $\in$ P, we can construct a directed graph as follows.

1. Let ϕ be an instance of 2-CNF-SAT in which each clause has exactly 2 literals.
2. Each vertex in G_ϕ are the variables of ϕ ant their negations. That is, there are exactly $2n$ vertices in G_ϕ
3. Then, for each clause $(x \wedge b)$ in ϕ, where x and y are

literals, create a directed edge from $\neg x$ to y and from $\neg y$ to x. These edges mean that if x is not true, then y must be true and vice-versa. That is, there exists a directed edge (x, y) in G iff there exists a clause $(\neg x \wedge y)$

The edges in G_ϕ capture the logical implications ($\rightarrow$) of ϕ and thus have the following tow property.

- Symmetry property: if there is an edge (x, y) in G_ϕ, then there is an edge $\neg y, \neg x$ in G_ϕ.
- Transivity property: if $x \rightarrow y$ and $y \rightarrow z$, then $x \rightarrow z$. by the transitivity property of implications.

Now we have the following claim.

ϕ is unsatistiable iff there exists a variable such that there exist a path from x to $\neg x$ and a path from $\neg x$ to x in G_ϕ

Proof by contradiction.

Suppose that are paths from x to $\neg x$ and $\neg x$ to x for some x in G_ϕ and there also exists a satisfying assignment $(x_1, x_2, \ldots, x_n)$ for ϕ.

Casel: $(x_1, x_2, \ldots, x_n)$ be such that x is TRUE.

Let the path from x to $\neg x$ be

$x \rightarrow \cdots \rightarrow A \rightarrow B \rightarrow \cdots \rightarrow \neg x$

From the construction of graph, we know that there is an edge between x to y iff there is a clause $(\neg x \wedge y)$ is ϕ. The edge from x to y represents that if x is TRUE then y must be TRUE. Now, since x is true, all literals in path from x to A (including A) must be TRUE. Similarily, all literals in the path from B to $\neg x$ (including B) must be FALSE since $\neg x$ is FALSE. This leads to the result that the clause $\neg A \wedge B$ is FALSE which contradicts our assumption that there exists a satisfying assignment $(x_1, x_2, \ldots, x_n)$ for ϕ.

CaseII: $(x_1, x_2, \ldots, x_n)$ be such that x is FALSE.

Following a similar analysis in CaseI, we know that this leads to a contradiction.

Therefore, by checking if there exists a variable x in G_ϕ such that the two paths x to $\neg x$ and $\neg x$ to x is in G_ϕ, we can decide whether the corresponding 2CNF-SAT expression is satisfiable or not.
Run time analysis.

- Constructing $G_\phi = (V, E)$ takes $O(n+m)$ time where n and m is the number of variables and the number of cluases in ϕ respectively.
- We can use graph traversal algorithm like BFS or DFS to determine the existence of a path from one node to another, which costs $O(V+E) = O(n+m)$

Thus, the total time complexity of this algorithm is $O(n+m)$, which indicates that 2-CNF-SAT in P.

Problem8

(a) Since we just need to check the color of neighbors, we can just apply BFS as the following steps.

- First, label every vertex in G 0 (which means “not visited”).
- Start from randomly-picked vertex v and label it 1 (color A).
- Apply BFS from source v to visit each vertex in G. Each time a 0-labeled vertex is visited, check the labels of its neighboring vertices. If it has two neighbors which have different labels, return False since G is not 2-colorable. Otherwise, label it with the type that is different from its neighbors.
- If all the vertices are visited without termination, we will have a feasible solution of a 2-coloring problem.

The time complexity is $O(V+E)$ if we use adjacency list to implement a graph.
(b) Decision problem of the graph-cloring problem

P: Given an undirected graph $G = (V, E)$ and a positive integer k, is there a k-coloring for G?
Proof (=>):
If we can determine whether a k-coloring exists for graph G, then we can solved the graph-coloring problem by trying k from 1 to $|V|$. In worst case, we need to solve $|V|$ times for the problem P, which implies that the graph-coloring problem can be solved in polynomial time.
Proof (<=):
If the graph-coloring problem is solvable in polynomial time, we can find the minimum number of colors k^* in polynomial time. By checking whether $k \geq k^*$, P can be solved directly.
(c) To show that 3-COLOR is NP-complete, we need to prove the following two things.

- First, P can be verifited in polynomial time. We can visit all the vertices in G to check if there are two neighboring vertices with the same color and also count the number of colors. Thus, $P \in NP$
- Second, since 3-COLOR is a subset of P and 3-COLOR is NP-complete, L is NP-hard.

Therefore, the decision problem P is NP-comeplete.
(d)
First, since x_i and $\neg x_i$ are both connected to the vertex RED, they cannot be colored c(RED) and thus have to be colored c(TRUE) or c(FALSE). Second, since a variable x_i is connected to $\neg x_i$, exactly one of them is colored c(TRUE) and the other is colored c(FALSE). If a variable x_i is assigned TRUE/FALSE in ϕ, then we color x_i c(TRUE)/c(FALSE) and color its negation $\neg x_i$ c(FALSE)/c(TRUE)0
Therefore, each triangle $(x_i, \neg x_i, (RED)$ are 3-colored, which means that the graph containing just the literal edges which consists of exactly those triangles and thus

can be 3-colored.
(e)
From part(d), we know that none of the vertices x, y, z cam be colored c(RED), which means that x, y, z can be colored either c(TRUE) or c(FALSE)
I draw a figure to argue that the widget is 3-colorable iff at least one of x, y, z is colored c(TRUE).
Proof (=>):
Use ~p => ~q. Suppose that none of x, y, z is colored c(TRUE), i.e. x, y, z are all colored c(FALSE). Since x, y are colored c(FALSE), a and b must be colored either c(TRUE) or c(RED). Besides, since a and b sould be colored differently, one of them is colored c(TRUE) while the other is colored c(RED). Thus, c should be colored c(FALSE). Since c and z are both colored c(FALSE), one the d and e is colored C(TRUE) and the other one is colored c(RED), which indicated that the widget is not 3-colorable (contradiction). Therefore, we prove that if the widget is 3-colorable, then at least one of x, y, z is colored c(TRUE).
Proof(<=):
The following table shows 3-coloring of the widget for all possible cases that at least one of x, y and z is colored c(TRUE). Therefore, we proof that if at least one of x, y, z is colored c(TRUE), then the widget is 3-colorable.

x	y	c	a	b	c
c(TRUE)	c(FALSE)	c(FALSE)	c(FALSE)	c(RED)	c(TRU
c(FALSE)	c(TRUE)	c(FALSE)	c(RED)	c(FALSE)	c(TRU
c(FALSE)	c(FALSE)	c(TRUE)	c(RED)	c(TRUE)	c(FAL

(f)
In order to proof 3-COLOR is NP-complete, we need to prove the following two claims.

- First, the 3-COLOR problem is in NP. To verify this, we

can check whether a coloring c for a graph G in polynomial time by using BFS to visit all the vertices and edges in $O(V+E)$ time. When visiting, we can count the number of oclors used and check if there are two neighboring vertices with the same color.

- Seoncd, we select the 3-CNF-SAT problem (a NP-complete problem) and we want to find a function f to that maps any instance of 3-CNF-SAT to an instance of 3-COLOR in polynomial time (i.e. 3-CNF-SAY $\leq_p$ 3-COLOR). To verify this, givena formul ϕ of m clauses and n variables, we can construct a graph $G=(V,E)$ by the steps described in the problem, where $|V|=2n+5m+3$ and $E=3n+10m+3$. Apparently, the reduction takes polynomial time.

Now, we need to prove that ϕ is satisfiable iff $G(V,E)$ is 3-colorable.

Proof(=>):

From subproblem(d), we know that for any truth assignement, G is 3-colorable regardless of the clause edges. Therefore, there's no literal edge that is incident on two same-color vertices. Since ϕ is satisfiable, every clause is True, which means that at least one of three literals in a clause is assigned True. In addition, for a truth assignment, each literal can only be True or False. From subproblem (e), for each clause, if at least one of three literal vertices is colored c(TRUE), then the corresponding widget is 3-colorable. Thus, $G(V,E)$ is 3-colorable.

Proof(<=):

Since $G(V,E)$ is 3-colorable, every widget is 3-colorable. By subproblem (e), if a widget is 3-colorable, then at least one of three literal vertices is colored c(TRUE), which means that the corresponding clause in ϕ is satisfiable because all clauses are True.

Thus, we proved that 3-COLOR problem in NP-complete.

Problem9

(a) The total cost is $3+6+2+3+1+6+1=22$ and the cost of each procedure is as follows.

a.

$S_1=[\quad]$ $S_2=[\quad]$
$S_1=[a\ b\ c]$ $S_2=[\quad]$ cost=3
$S_1=[\quad]$ $S_2=[a\ b\ c]$ cost=6
$S_1=[\quad]$ $S_2=[\quad c]$ cost=2

$S_1=[def]$ $S_2=[\quad c]$ cost 3
$S_1=[def]$ $S_2=[\quad]$ cost 1
$S_1=[\quad]$ $S_2=[def]$ cost 6
$S_1=[\quad]$ $S_2=[\ ef]$ cost 2.

(b) The worst case happens when we first insert n elements in S_1 and then pop n elements in S_2. The largest running time happens at the **first pop operation** since we have to first **dump** all the elements of S_1 into S_2 and then pop one element from S_2, which costs $2n+1$.
(c) Intuitively, since we have to dump the element in S_1 into S_2 (which costs 2 for an element), I will pay \$3 for the insertion operation and \$1 for the removal operation.
Prove:

- Insertion (\$3): \$1 is used for pusing the element onto S_1 and the remaining \$2 credits attached to the element is for the removal operation.
- Removal (\$1): In worst case, we have to dump the element from S_1 into S_2. However, we can use \$2 credit on each element in S_1 for this operation, and the removal operation thus only need \$1.

Therefore for a sequence of n INSERT and DELETE operations, the total amortized cost is $O(n)$ and the average cost per operation is $O(1)$.