HW#5

Dong Boyuan
1547489

Problem 1

1(a) procedure Shortest_path(s, t, t)

```
** s, t are points containing their coordinate and path (which is a queue)
** s.x←i  s.y←j  t.x←i'  t.y←j'  for s(i,j)  t(i', j')
** t is the size of chess board  t×t — also the # of points
** Q is a queue store points.
dx ← [2, 2, -2, -2, 1, 1, -1, -1]
dy ← [1, -1, 1, -1, 2, -2, 2, -2]
s.path.enqueue((s.x, s.y))
Q.enqueue(s)
for (i from 1 to t)
    for (j from 1 to t)
        visit[i][j] ← false
visit[s.x][s.y] ← true
while (Q ≠ Φ) then
    t ← Q.dequeue()
    if(t.x = t.x and t.y = t.y) then
        return t.path
    for(k from 1 to 8)
        x ← t.x + dx[k]
        y ← t.y + dy[k]
        if(IsInside(x, y, t) and !visit[x][y]) then
            v.x ← x
            v.y ← y
            path ← t.path
            v.path ← path.enqueue((x, y))
            visit[x][y] ← true
            Q.enqueue(v)
```

procedure IsInside(x, y, n)

```
return  x>0 and x<=n and y>0 and y<=n
```

Dong Boyuan
1547489

Procedure PrintPath (Q)

```
while ( Q ≠ ∅ ) then
    t ← Q.dequeue ( )
    print ( t )
```

For this case, we use BFS to solve the problem. Each point (or position) have 8 possible positions that the knight may move to. So we traverse them one by one. if the possible position is in the txt chessboard and hasn't been visited before then add them into the path Queue and keep traversing its next step. Until we reach the finish position of (i', j'), we get the shortest path that stored in t.path (is a queue).

This is the shortest steps path → has been proved in BFS.

In the worst case, we need to traversing all txt positions :

- We need to Initilize visit[x][y] take $O(t^2)$ time
- We need to dequeue t^2 points which take $O(t^2)$ time
- In each iteration we will take a constant time (since we only have 8 possible points each time)

Overall, the runtime is $O(t^2)$

(b) In this case, we still use BFS to find the shortest path. Since there will be 12 different possible operations to these bottles (① Fill bottle 1 ; ② Fill bottle 2 ; ③ Fill bottle 3; ④ Empty bottle 1 ; ⑤ Empty bottle 2 ; ⑥ Empty bottle 3 ; ⑦ pour bottle 1 to bottle 2 (1→2); ⑧ Pour bottle 1 to bottle 3 (1→3) ; ⑨ Pour bottle 2 to bottle 1 (2→1) ; ⑩ Pour bottle 2 to bottle 3 (2→3) ; ⑪ Pour bottle 3 to bottle 1 (3→1) ; ⑫ Pour bottle 3 to bottle 2 (3→2))

Traverse all possible operations until we get to the end with (a, b, c)

Then we would get the shortest sequence from (0,0,0) → to (a, b, c) for (bottle 1 (k), bottle 2 (l), bottle 3(m)) k, l, m are all integers

procedure FindBottle (a, b, c, k, l, m)

** Each Node will have x, y, z (the water in bottle1 = x, bottle2 = y, bottle3 = z) and the path (a queue) to reach this Node

** a, b, c will be integers bottles end with (bottle1 = a, bottle2 = b, bottle3 = c)

** bottle1.capacity = k, bottle2.capacity = l, bottle3.capacity = m

```
start.x ← 0
start.y ← 0
start.z ← 0
start.path.enqueue({0,0,0})
Q.enqueue(start)
Visited[start] = true
while (Q ≠ ∅) do
    u ← Q.dequeue()
    if (u.x = a and u.y = b and u.z = c) then
        return u.path
① if (u.x < k) then
        new.x ← k
        new.y ← u.y
        new.z ← u.z        path ← u.path
        new.path ← path.enqueue({new.x, new.y, new.z})
        if (Visited[new] = false) then
            Q.enqueue(new)
            Visited[new] ← true
② if (u.y < l) then
        new.x ← u.x
        new.y ← l
        new.z ← u.z        path ← u.path
        new.path ← path.enqueue({new.x, new.y, new.z})
        if (Visited[new] ≠ false) then
            Q.enqueue(new)
            Visited[new] ← true
③ if (u.z < m) then
        new.x ← u.x
        new.y ← u.y
        new.z ← m          path ← u.path
        new.path ← path.enqueue({new.x, new.y, new.z})
        if (Visited[new] = false) then
            Q.enqueue(new)
            Visited[new] ← true
④ if (u.x > 0) then
        new.x ← 0
        new.y ← u.y
        new.z ← u.z        path ← u.path
        new.path ← path.enqueue({new.x, new.y, new.z})
        if (Visited[new] = false) then
            Q.enqueue(new)
            Visited[new] ← true
```

Dong Boyuan
1547489

Dong Boyuan
154748

⑤ If (u.y > 0) then
new.x ← u.x
new.y ← 0
path←u.path > new.z ← u.z
new.path ← path.enqueue ([new.x, new.y, new.z])
If (Visited[new] == false) then
Q.enqueue (new)
Visited [new] ← true

⑥. If (u.z > 0) then
new.x ← u.x
new.y ← u.y.
new.z ← 0
path←u.path > new.path ← path.enqueue ([new.x, new.y, new.z])
If (Visited[new] == false) then
Q.enqueue (new)
Visited [new] ← true

⑦. If (u.x > 0 and u.y < l) then
1->2. new.x ← max(0, u.x+u.y-l)
new.y ← min (u.x+u.y, l)
new.z ← u.z
path←u.path > new.path ← path.enqueue ([new.x, new.y, new.z])
If (Visited[new] == false) then
Q.enqueue (new)
Visited [new] ← true

⑧ If (u.x > 0 and u.z < m) then
1->3. new.x ← max (0, u.x+u.z-m)
new.y ← u.y
new.z ← min (u.x+u.z, m)
path←u.path > new.path ← path.enqueue ([new.x, new.y, new.z])
If (Visited[new] == false) then
Q.enqueue (new)
Visited[new] ← true

⑨ If (u.y > 0 and u.x < k) then
2->1 new.x ← min (u.x+u.y, k)
new.y ← max (0, u.x+u.y-k)
new.z ← u.z
path←u.path > new.path ← path.enqueue
If (Visited[new] == false) then
Q.enqueue (new)
Visited[new] ← true

⑩ If (u.y > 0 and u.z < m) then
2->3. new.x ← u.x
new.y ← max (0, u.y+u.z-m)
new.z ← min (u.y+u.z , m)
path ← u.path
new.path ← ~~u.path~~ path.enqueue ([new.x, new.y, new.z])
If (Visited[new] == false) then
Q.enqueue (new)
Visited[new] ← true.

Dong Boyuan
1547489

```
⑪ If ( u.z > 0 and u.x < k) then
3->1        new.x ← min( u.x+u.z, k)
            new.y ← u.y
            new.z ← max ( 0, u.x+u.z -k)
            path ← u.path
            new.path ← path. enqueue ( { new.x, new.y, new.z } )
            If ( Visited [new] == false ) then
                    Q.enqueue( new)
                    Visited [new] ← true
⑫ If ( u.z>0 and u.y < l ) then
3->2        new.x ← u.x
            new.y ← min( u.y+ u.z, l)
            new.z ← max( 0,  u.y+u.z - l)
            path ← u.path
            new.path ← path. enqueue ( { new.x, new.y, new.z } )
            If ( Visited [new] = false ) then
                    Q. enqueue ( new)
                    Visited [new] ← true
end while
Print (" NOT Find")
return 0

procedure InVisit( Visit, new)
    while ( Visit ≠ ∅) then
        t ← Visit. dequene ()
        If ( t.x = new.x and t.y = new.y and t.z = new.z ) then
                return true
    return false.
```

Runtime:

In the most case,

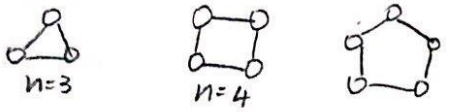
- Each bottle have 4 cases (① Fill ② Empty ③ pour to another bottle ④ pour to another bottle) So, we have total 12 cases.
- We have $k \times l \times m$ possible nodes intotal (k, l, m are all integers)
- So we will dequene $O(k \times l \times m)$ times.

Overall It takes $O(k \times l \times m)$ time.

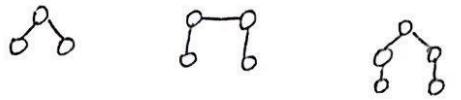
Problem 2

(a). A node is an articulation point iff removing it (and edges through it) disconnect the graph.

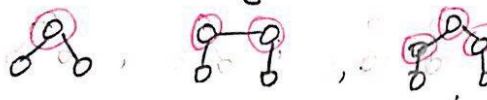
For any $n \geq 3$, there will be a connected graph with n edges where all nodes are not an articulation. This is true only when n nodes connected to a circle with n edges.

eg. 

When we removing 1 edge from the graph, all nodes are still connected with $n-1$ edges.

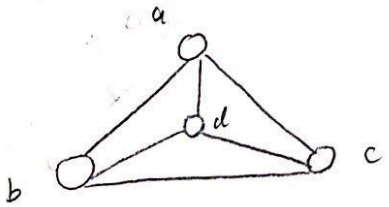


However, when we trying to remove one more edge, there may be a disconnected graph



So when all connected graphs with $1 \leq n-1$ edges must have at least 1 articulation point to make ~~all not all nodes are~~ nodes disconnected

(b). Dong Boyuan 1547489



$n = 4 \quad m = 6$

For node a: e.g.

- $a \to b \to d \to a$
- $a \to c \to d \to a$
- $a \to b \to c \to a$
- $a \to b \to c \to d \to a$
- $a \to c \to b \to d \to a$
- $a \to b \to d \to c \to a$.

are 6 cycles.

And node b, c, d have the same # of cycles as a.

For d: ~~3 cycles~~
- $d \to a \to b \to d$
- $d \to a \to c \to d$
- $d \to b \to c \to d$

$\therefore \quad 4 \times 6 = 24 > 2^n = 2^4 = 16 \quad \therefore$ # of cycles $\in O(2^n)$

(c)

For any undirected graph $G=(V, E)$, if we ~~a~~ have a cycle in this ~~Graph~~ graph G with edges $e_1, \cdots, e_k$ for k connected nodes, $(v_1 \to v_2 \to \cdots v_k)$ we can just put the directed edges of graph G'' like $v_1 \to v_2 \to \cdots v_{k-1}$ but put the last edge $v_{k-1} \leftarrow v_k$ which ~~it~~ can ~~not~~ make $(v_1, \cdots v_k)$ not a cycle.

So for each cycle in graph G we have a non-cycle version in G'', for any undirected graph G, there exists an orientation G'' which is acyclic.

Dong Boyuan

1547489

Problem 3

Dong Boyuan
1547489

(a) Yes

This is actually a greedy ~~algo~~ algorithm.

Optimal substructure: an optimal solution to the ~~with~~ original problem contains within it optimal solutions to subproblems.

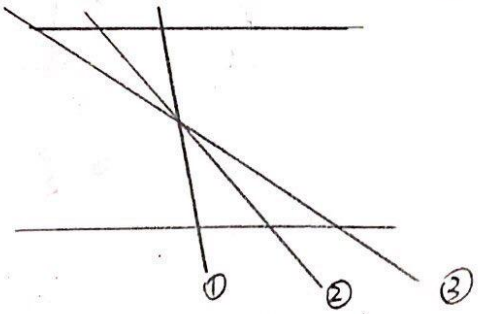
For this case, driving along the only high way ~~after~~ are crossing the Australian outlook, try to go as far as I can before refueling.

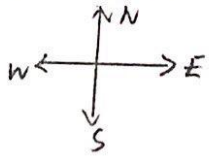
Each time, I ~~chose~~ make the best choice which can be locally minimum drive as far as I can. Then overall, I will have the minimum stops ~~as well~~. by union all local ≤ minimums.

(b). No.

Since, I start a direction between south and east, so the total distance I traveling can be really different. according to the direction I choose

For example,





In this case ①, ②, ③ are all directions between south and east, however, the distance is ① < ② < ③ which implies that ② and ③ ~~can use more~~ will need more stops or gases to ~~go~~ travel through.

So for this can, I can chose the shortest distance that connects Morocco and Sudan, and follow along that direction to stop at furthest possible gas station which is still within distance ≤ d from the last stop.

Problem 4

Dong Boyuan
154789

(a) Procedure CheckNest (B_1, B_2, d)

** $B_1 = (l_1, l_2, \cdots l_d)$, $B_2 = (m_1, m_2, \cdots, m_d)$

** d is the dimentional of boxes.

Sort both B_1 and B_2 in a non-decreasing order.

IsNest ← true

for(i from 1 to d)

    if($B_1[i] > B_2[i]$)

        IsNest ← false

return IsNest.

First sort B_1 and B_2 in a non-decreasing order. Then traversing their length one by one, if there exist length $l_i > m_i$ then, the box $B_1(l_1, l_2, \cdots l_d)$ can not be nested into the box $B_2(m_1, \cdots, m_d)$. B_1 can be nested into B_2 only when all length $l_i < m_i$.

(b) procedure DFS (B, n, d)

** $B = \{B_1, B_2, \cdots, B_n\}$. n is the # of box in B, d is the dimention of boxes.

for(i from 1 to n)

    DFS-visit (B[i], d, n, Q_i)

    Q_i.enquene(Q_i)

procedure DFS-visit (B_i, d, n, Q_i)

Q_i.enquene (B_i)

for(j from 1 to n)

    if (CheckNest (B_i, B[j], d) = true) then

        DFS-Visit (B[j], d, n, Q_i)

For this problem, we make use of DFS to find the longest sequence of each B_i in $B = \{B_1, B_2, \cdots, B_n\}$. So that we can tell their relations easily and store them nicely in cupboard.

We will traverse each B_i in B and then for each B_i try to find the longest sequence Q_i, store all Q_i in Q. Then we can tell their relations by getting Q.

Dong Boyuan
1547489

Problem 5

Dong Boyuan
1547489

For this claim in the ~~pp~~ problem:

We get T_i after iteration i and $t_{u,v}(i)$ donate the weight of the heaviest edge on T_i.

Then we will do the next iteration $(i+1)$:

We will find the heavist edge in T_i $e' = (u,v)$

Case I: If $w(e') \leq w(e)$ then we will do nothing. T_{i+1} is the same as T_i

So $t_{u,v}(i+1) = t_{u,v}(i)$

Case II: If $w(e') > w(e)$ the we ~~pp~~ remove the edge e' in T_i and ~~and~~ union the edge e with T_i: $w(e) - w(e') < 0$.

$T_{i+1} = T_i - w(e') + w(e) = T_i + (w(e) - w(e')) < T_i$

So the heavest edge $e' = (u,v)$ has been removed from T_i.

The heavest edge $t_{u,v}(i+1)$ now is less than $t_{u,v}(i) = e' = (u,v)$

$\therefore$ $t_{u,v}(i+1) < t_{u,v}(i)$

So the claim is true.

According to the claim above, we can tell this algorithm return a MST.

Since each time we are removing the heavist edge connected u and v, we are actually reducing its weight, until we get the minimum weight of the edge.

For each subproblem we will get its ~~s~~ minimum weight, then after union all subproblems, we will get the MST.

Runtime:

Runtime of kruskal's algorithm, including sorting m edges and $O(m)$ calls to find(): $O(m\log(m)) + O(m\log(n)) = O(m\log m) = O(m\log(n))$ for $n-1 \leq m \leq \binom{n}{2}$

b in SomeTree foreach call back-edges B, ~~and~~ $b = |B|$

Find the ~~largest~~ heavest edge each time takes $O(\log b)$ and we need foreach b edges which will take $O(b\log b)$ time.

SomeTree would be faster only when $b \in \Omega(n)$, we have at most $b(n-1)$ updates as we will remove at most $b-(n-1)$ edges, then we will get MST with $n-1$ ~~eg~~ edges.