AOA Final

Name: Shiyin Lin
UFID: 49619012

1 Dynamic programming

1.1 Pseudo-code

algorithm 1 Knapsack

Input: *int* item, *Array* weight, *Array* time, *Array* value, *int* W, *int* T
Output: *int* maxValue, *Array* select

```
function KNAPSACK(int, Array, Array, Array, int, int)
    Init dp Array // n * w * t
    for i = 1; i <item + 1; i + + do
        for j = 1; j <W + 1; j + + do
            for k = 1; k <T + 1; k + + do
                if j − weight[i − 1] ≥ and k − time[i − 1] ≥ 0 then
                    nochange = dp[i - 1][j][k]
                    change = dp[i-1][j-weight[i-1]][k-time[i-1]] + value[i-1]
                    if nochange >change then
                        select[i] = 0
                        [i][j][k] = nochange
                    else
                        select[i] = 1
                        dp[i][j][k] = change
                    end if
                else:
                    dp[i][j][k] = dp[i-1][j][k]
                    select[i] = 0
                end if
            end for
        end for
    end for
    return dp[-1][-1][-1], select
end function
```

1.2 Proof of Correctness

Proof of correctness of the algorithm by induction
First, dp[0][j][k] = 0, which means that when there are no items, the maximum sum of values that can be obtained is 0, regardless of the pack limit and the time. Second, suppose we have correctly computed the maximum sum of values for any of the i - 1 items at any of the backpack capacities and at any of the times, i.e., for all dp[i - 1].
Then according to the state transfer equation

$$dp[i][j][k] = max(dp[i-1][j][k], dp[i-1][j-weight[i-1]][k-time[i-1]]+value[i-1]$$

We can correctly derive the values from dp[1] to all values in dp[i].
Thus we prove the correctness of the algorithm.

1.3 Algorithm's running time

In the knapsack problem, we facilitate a three-dimensional dp array, let the number of items be n, the capacity of the knapsack be w, and the available time be t. The time complexity of the code is O(nwt)

2 Network flow

2.1 Pseudo-code

@name 2 DINIC_MIN_CUT

Input: *vertex* S, T, *Array* g, *Array* cap
Output: *int* min-cut

```
function BFS( )
    dic, q = dict, deque[S]
    dic[S] = 0
    while q do
        node = q.popleft()
        for next in g[node] do
            if dic[next]==-1 and cap[node, next]>0 then
                dic[next] = dic[node]+1
                if next == -1 then
                    return dic
                end if
                q.append(next)
            end if
        end for
    end while
    return dic
end function

function DFS(node, flow, dic)
    origin = flow
    for next in g[node] do
        if dic[next]==dic[node]+1 and cap[node, next] >0 then
            a = DFS(next, min(origin, cap[node, next, dic])
            if a then
                cap[node, next] -= a
                cap[next, node] += a
                if cap[next, node] >0 then
                    g[next].append(node)
                end if
                origin -= a
                if origin == 0 then
                    return flow
                end if
            end if
        end if
    end for
    return flow - origin
end function
```

```
function DINIC( )
    ans = 0
    while ans <inf do
        dic = bfs()
        if dic[-1] == -1 then
            break
        end if
        ans += dfs(src, inf)
    end while
    return ans
end function
```

2.2 Proof of Correctness

Let f_i denote residual graph after iteration $i(G_{f_0} = G)$
depth(G_{f_0} = length of the shortest path from s to t
depth(G_{fi+1}) >depth(G_{f_i})
Support that depth($G_{f_{i+1}}$) $\leq depth(\mathrm{G}_{f_i})$
The $G_{f_{i+1}}$ contains an s-t path of length $\leq depth(\mathrm{G}_{f_i})$
This path corresponds to an augmenting path for the flow

$$f' = f_{i+1} - f_i in G_{f_i}$$

But since the augmenting path has length depth(G_{f_i}) it is also an augmenting path in the level graph
Contradiction
So Dinic Algorithm can obtain the max flow. By max-flow min-cut theorem we can know that max-flow is the same as min-cut, so we get min-cut.

2.3 Algorithm's running time

Building a Level graph with BFS takes O(E) time. Obtaining a blocked flow requires O(VE) time. The outer loop takes O(V) time to complete. We create a new Level graph and detect the blocked flow in each iteration. It can be seen that the number of layers increases by at least one in each repetition. As a result, the outer loop takes O(V) time to complete. As a result, the time complexity is $O(EV^2)$. The complexity of the Ford-Fulkerson method O(Ef), where f is the maximum flow of the network, is substantially lower when the maximum flow is large.

3 Complexity

To prove that the problem is NP-complete, we first need to prove the following two points:
1. The problem itself is NP
2. All other problems in NP class can be polynomial-time reducible to that.

NP
Given a certificate that defines the set of numbers as S and divides the partition defined as $A_1, A_2..A_n$, We can complete the certifications according to the following:
For each element x in $A_1, A_2...A_n$, verify that all of them belong to S.
Let S_1 be the sum of A_1, and S_2 to S_n similarly be the subset sum of all subsets of A_2 to A_n
Verify that $S_1, S_2..S_n$ are the same.
It can be easily proved in linear time, so we can proof that problem is NP
NP-Hard
Assuming that we divide the array into two subsets(A, A').
Assume a array T with sum equal to t. Then the remaining elements in S (assume O) will have subsum $o = s - t$. Assuming that the original array is equal to $T' = T \cup (s - 2t)$ with sum equal to t', we can conclude the following:

$$\begin{aligned} o &= s–t \\ o–t &= s–2t \\ t' &= t + (s–2t) \\ &= s–t \\ &= o \end{aligned}$$

The original array S is partitioned into two subsets and the sum of each subset is $(s - t)$ satisfying the assumption.
Denote the partition containing s - 2t as A'. Let A = A'-s-2t. Then the sum of the elements in A is.

$$A = s–t–s–2t = t$$

So the problem can be reduced to Subset-Sum problem.

The problem satisfies the above two points, so the problem is NP-complete