CS222 Homework 6

Algorithm Analysis & Deadline: 2020-10-20 Firday 24:00

Exercises for Algorithm Design and Analysis by Li Jiang, 2020 Autumn Semester

1. An adventure game uses a graph G consisting of N rooms (numbered from 1 to N) to represent the places need to be explored. These rooms can only be linked in one-way, meaning that a person passing through this path can only move from one room to another but cannot return to the room they left/explored. It's worth noting that there is no circle in this graph. People taking part in the game appear randomly in any of the rooms connected via corridors, and we can explore rooms in the same corridor. You should note that the path taken by two players may contain some of the same rooms.

   How many players are the minimum needed to explore all the rooms?

   Describe your design first and write down your algorithm in the form of pseudo-code.

   Solution.

   Problem Def. Given a DAG, find the least paths to cover all the vertices. (Paths can overlap).

   We note the vertices set as V, and the edges set as E. To limit the visit times of vertices, we split every vertice v into 2 vertices v_i, v_o and add an directed edge (v_i, v_o) with capacity $[1, +\infty)$. For every $e = (u, v) \in E$, replace it with a new edge (u_o, v_i) with capacity $[0, +\infty)$. Add two new vertices st, en, and for every v_i add an new edge (st, v_i) with capacity $[0, 1]$ representing whether a player starts at v, and for every v_o add an new edge (v_o, en) with capacity $[0, \infty]$ because more than 1 players can finish at v.

   We have constructed a new graph G' with capacity. Figure 1, 2 shows an example. We need to find

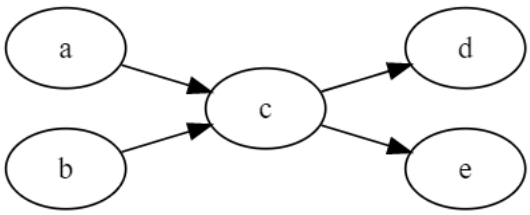

Figure 1: origin graph

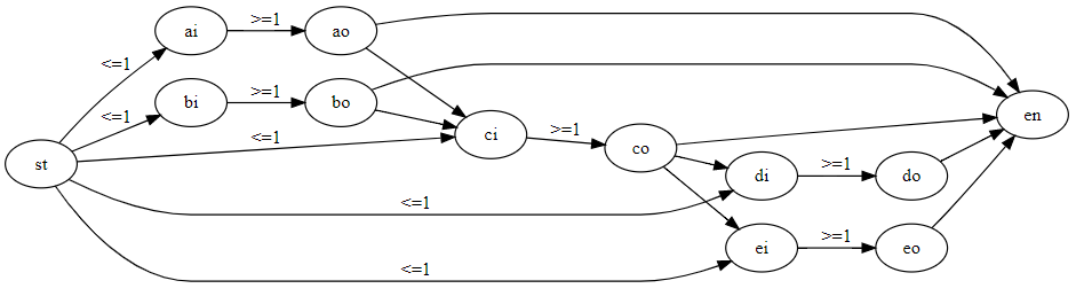

Figure 2: graph for network

   the minimum flow statisfying these capacity limits. For this problem, I have two ideas:

   (1) Binary search the answer.
   Set a demand d_v for every vertice v, $d_v = \sum_{e\ in\ v} f_e - \sum_{e\ out\ v} f_e$. Initially, $d_v = 0$ for all v, except $d_{st} = -c, d_{en} = c$, where c is the answer we need to check. For all $e = (u, v)$ with capacity $[1, +\infty)$,

remove the lower bound limit by $d_u \leftarrow d_u + 1, d_v \leftarrow d_v - 1$. And then add a source s and a sink t, and for every v with $d_v < 0$ add an edge (s, v) with capacity $|d_v|$, and for every v with $d_v > 0$ add an edge (v, t) with capacity $|d_v|$.
Then run a $s \to t$ max flow on the network. If all the edge out of s has a full flow i.e. $f_e = c_e$, then c players are enough; otherwise c players are not enough. We can find the minimum number of players by binary searching c. **Time complexity:** $O(nm \log n)$ (because the flow is at most n and a BFS process is $O(m)$).

(2) Find the minimum flow straightly.
The network structure is almost the same as the one above, but we set $d_{st} = d_{en} = 0$ this time. Instead, we add an additional edge (en, st) with capacity $[0, +\infty)$. We run a $s \to t$ max flow on this network to check whether there exists a circulation statisfying the demand limits. Then we get a feasible flow. And next, we remove the vertice s and t (and all edges linked to s, t), and remove the edge (en, st). And then we run a $en \to st$ max flow on this network. Now, $\sum_{e\ out\ st} f_e$ is the minimum number of players needed.
Briefly proof:

- After running $s \to t$ max flow, we get a feasible circulation.
- After running $en \to st$ max flow, the circulation is still feasible, because we remove all the edge associated with demand d_v and this process only sends back the redundancy.
- The second process sends back the redundancy as much as possible, so the remain is minimum.

Algorithm 1 shows the pseudo-code for idea (1).

Algorithm 1 Pseudo Code for Problem 1

Require: $G = (V, E)$
Ensure: *num* the number of minimum players needed

```
function FindTheLeastPlayers(G = (V, E))
    V' ← ∅, E' ← ∅
    V' ← V' ∪ {st, en, s, t}
    for v ∈ V do
        V' ← V' ∪ {v_i, v_o} // split vertice and add edge
        E' ← E' ∪ {(v_i, v_o, c = ∞)}
        E' ← E' ∪ {(st, v_i, c = 1), (v_o, en, c = ∞)}
        E' ← E' ∪ {(vi, t, c = 1), (s, v_o, c = 1)} // deal with demand
    end for
    L ← 1, R ← |V|
    while L ≤ R do
        C = ⌊(L + R)/2⌋
        G' = (V', E' ∪ {(s, st, c = C), (en, t, c = C)})
        flow ← 0
        while FindAugmentPath(G', s, t) do
            flow ← flow+Augment(G', s, t)
        end while
        if flow == |V|+C then // means all the demands are satisfied, hence the circulation is valid
            R ← C − 1 // [C, R] is valid, then search in [L, C − 1]
        else
            L ← C + 1 // [L, C] is invalid, then search in [C + 1, R]
        end if
    end while
    if L ≤ |V| then
        return L
    else
        throw No Feasible Solution Error
    end if
end function
```

And Algorithm 2 shows the pseudo code of max flow algorithm (Dinic) here.

Algorithm 2 Dinic algorithm

```
1: function findAugmentedPath(G = (V, E), s, t)
2:     Q ← an empty queue, and add s into Q, and mark s as visited.
3:     while Q is not empty do
4:         Take out the head u of Q.
5:         for e = (u, v) out of u do
6:             if c_e > 0 and v is not visited then
7:                 Set the precursor of v as u.
8:                 if v == t then
9:                     return True
10:                end if
11:                Add v into Q, and mark v as visited.
12:            end if
13:        end for
14:    end while
15:    return False
16: end function
17:
18: function Augment(G = (V, E), s, t)
19:    flow ← +∞
20:    path ← an empty stack
21:    while t != s do
22:        u ← the precursor of t
23:        flow = min{flow, c_(u,t)}
24:        Add edge (u, t) into path
25:        t ← u
26:    end while
27:    for edge e in path do
28:        c_e ← c_e − flow
29:        c_re ← c_re + flow // re means the residual edge of e.
30:    end for
31:    return flow
32: end function
```

2. Suppose there are $M \times N$ rooms, each of which holds a different number of treasures. Please select several rooms so that the selected rooms have no common sides (i.e., the selected rooms cannot be adjacent), and the selected rooms' treasures add up to the greatest value. Describe your design first and write down your algorithm in the form of pseudo-code.

   Note: The value of the treasure is definitely not negative.

   Solution.

   1. Define $V_1 = \{room : (x, y) | x + y \equiv 1 \mod 2\}$, $V_0 = \{room : (x, y) | x + y \equiv 0 \mod 2\}$.
   2. For every room $u \in V_0$, if another room v is adjacent to u, then add an directed edge (u, v) with capacity $+\infty$. We claim that we has constructed a bipartite graph $G = (V = V_0 \cup V_1, E = \{(u, v) | u \in V_0 \wedge u, v \textit{ are adjacent}\})$, because two rooms u, v are adjacent if and only if $(u \in V_0 \wedge v \in V_1) \vee (u \in V_1 \wedge v \in V_0)$.
   3. And then we add a source s and a sink t, and for every $u \in V_0$ add an edge (s, u) with capacity equal to u's treasure, and for every $v \in V_1$ add an edge (v, t) with capacity equal to v's treasure. And we define that (s, u) having flow means we select room u, and (v, t) having flow means we select room v.
   4. If we select two adjacent rooms, we say that there is a contradiction. We assert that there is a contradiction if and only if there is a flow from the source s to the sink t. Briefly proof:

      $\Rightarrow$ A contradiction $\Rightarrow$ we select two adjacent room $u \in V_0, v \in V_1 \Rightarrow$ a flow $s \to u \to v \to t$

      $\Leftarrow$ A flow $s \to u \to v \to t \Rightarrow$ we select $u \in V_0, v \in V_1$, and u, v are adjacent $\Rightarrow$ a contradiction.
   5. We need to give up as minimum as possible treasure $(s, u), (v, t)$ to make the network disconnected, i.e. we need to work out the **minimum cut**. And $max\ treasure = total\ treasure - minimum\ cut = total\ treasure - max\ flow$.

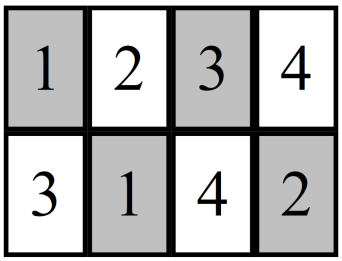

Figure 3: An example for P2

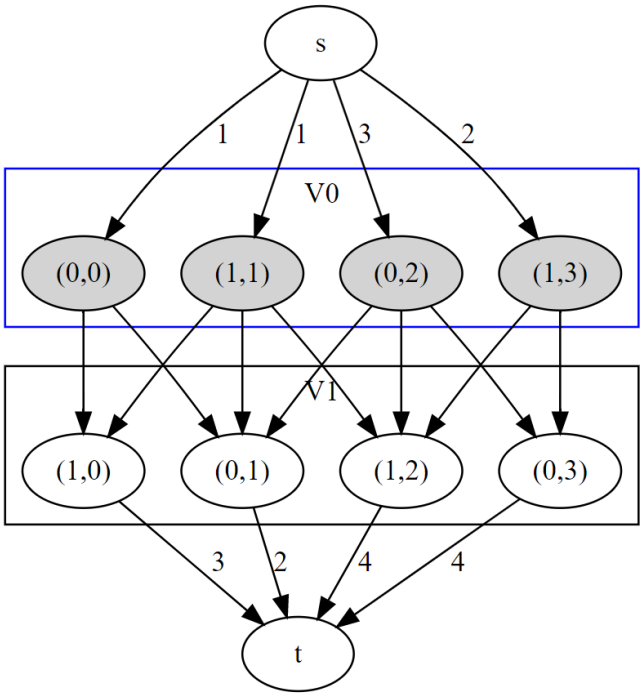

Figure 4: Network for Figure 3

Figure 3 shows an example for this problem and Figure 4 shows an network model constructed for the example.

Algorithm 3 pseudocode for P2

Require: An $M \times N$ grid *grid* representing treasures in the rooms.
Ensure: An set *selection* representing selected rooms.

```
1:  function selectRooms(grid)
2:      E ← ∅, V ← {rooms in grid} ∪ {s, t}.
3:      for room = (i, j) in grid do
4:          if i + j ≡ 0 mod 2 then
5:              for room0 = (i0, j0) adjacent to room = (i, j) do
6:                  E ← E ∪ {(room, room0, capacity = +∞)}
7:              end for
8:              E ← E ∪ {(s, room, capacity = room.treasure)}
9:          else
10:             E ← E ∪ {(room, t, capacity = room.treasure)}
11:         end if
12:     end for
13:     G ← (V, E)
14:     while findAugmentedPath(G, s, t) do
15:         Augment(G, s, t)
16:     end while
17:     selection ← ∅
18:     for e = (s, room) or (room, t) do
19:         if c_e > 0 then
20:             selection ← selection ∪ {room} // remained capacity > 0 ⇒ not in the min cut
21:         end if
22:     end for
23:     return selection
24: end function
```

Algorithm 3 shows the pseudocode for Problem2, and the Dinic algorithm used here are showed in Algorithm 2.

3. There are a large number of servers in the data center. Each server has different performance, the program is executed on different servers with different efficiency, and now there is a set of the new programs that need to run. Please assign these programs to the appropriate server based on the historical test results to make this batch of programs run most efficiently. Describe your design first and write down your algorithm in the form of pseudo-code.

Solution.
Problem Def. Given some programs $P = \{p_1, p_2, \cdots, p_n\}$ and some servers $Q = \{q_1, q_2, \cdots, q_m\}$, and a weight matrix $score_{n\times m}$. Find a match (i, m_i) to maximize $score = \sum_{1\le i\le n} score_{i,m_i}$. If $n \neq m$ there may be some programs not being run or some servers not runing program.
We constructed a weighted bipartite graph:

- A vertice set V_0 representing programs and a vertice set V_1 representing servers.
- For program p_i and server q_j, add an directed edge (p_i, q_j) with weight (or called cost) $MaxScore - score_{i,j}$, where $MaxScore := \max score_{i,j} + 1$.
- If $|P| < |Q|$, we add $|Q| - |P|$ pseudo programs. For each pseudo program p_i and each server q_j add an directed edge (p_i, q_j) with weight $MaxScore$.
- If $|Q| < |P|$, we add $|P| - |Q|$ pseudo servers. For each pseudo server q_i and each program p_j, add an directed edge (p_j, q_i) with weight $MaxScore$.

Then we need to find a perfect match with minimum weight (or called cost). Use the algorithm we talked in class. The correctness is proved in class, and hence I do not explain here. The pseudocode is showed in Algorithm 4.

Algorithm 4 pseudocode for P3

Require: Program set $P = \{p_1, p_2, \cdots, p_n\}$, server set $Q = \{q_1, q_2, \cdots, q_m\}$ and score matrix $score_{n\times m}$
Ensure: $Assignment = \{(p_i, q_{match_i}) | p_i \in P\}$

```
function assignServer(P, Q, score)
    V ← P ∪ Q ∪ {s, t}, E ← ∅, Assignment ← ∅, MaxScore ← max score_{i,j} + 1
    for each (p_i, q_j) do
        E ← E ∪ {(p_i, q_j, cost = MaxScore − score_{i,j})}
    end for
    if |P| < |Q| then
        for i ← |P| + 1 to |Q| do
            V ← V ∪ {p_i}
            for each q_j do
                E ← E ∪ {(p_i, q_j, cost = MaxScore)}
            end for
        end for
    end if
    if |Q| < |P| then
        for i ← |Q| + 1 to |P| do
            V ← V ∪ {q_i}
            for each p_j do
                E ← E ∪ {(p_j, q_i, cost = MaxScore)}
            end for
        end for
    end if
    foreach v ∈ V \ {s, t}: p(v) ← 0
    while Assignment is not a perfect matching do
        dis ← shortest distances using costs c^p(x, y) = p(x) + cost(x, y) − p(y)
        path ← shortest alternating path from s to t according to dis
        Update Assignment after augmenting along path.
        foreach v ∈ V \ {s, t}: p(v) ← p(v) + dis(v).
    end while
    return Assignment
end function
```

4. Given a weighted directed graph $G(V, E)$ and its corresponding weight matrix $W = (w_{ij})_{n\times n}$ and shortest path matrix $D = (d_{ij})_{n\times n}$, where w_{ij} is the weight of edge (v_i, v_j) and d_{ij} is the weight of a shortest path

from pairwise vertex v_i to v_j. Now, assume the weight of a particular edge (v_a, v_b) is decreased from w_{ab} to w'_{ab}. Design an algorithm to update matrix D with respect to this change, whose time complexity should be no larger than $O(n^2)$. Describe your design first and write down your algorithm in the form of pseudo-code.

Solution.
I assume $w_{ij} \geq 0, w'_{ab} \geq 0$.
Consider the change of d_{ij} after decreasing w_{ab} to w'_{ab}: I think that $d'_{ij} = \min\{d_{ij}, d_{ia} + w'_{ab} + d_{bj}\}$ holds. Briefly proof: A new shortest path from i to j either passes by or does not pass by the directed edge (a, b). If not passing by, $d'_{ij} = d_{ij}$ because other edges do not change. If passing by, $d'_{ij} = d'_{ia} + w'_{ab} + d'_{bj} = d_{ia} + w'_{ab} + d_{bj}$, because the shortest path from i to a and the shortest path from b to j do not pass by (a, b) (assuming positive weight).

Algorithm 5 pseudocode for P4

Require: W,D,a,b,w'_{ab}
Ensure: $D' = (d'_{ij})_{n\times n}$
1: **function** UPDATEDISTANCE(W, D, a, b, w'_{ab})
2: **for** $(i, j) \in \{1, 2, \cdots, n\} \times \{1, 2, \cdots, n\}$ **do**
3: $d'_{ij} = \min\{d_{ij}, d_{ia} + w'_{ab} + d_{bj}\}$
4: **end for**
5: **return** $D' = (d'_{ij})_{n\times n}$
6: **end function**

Algorithm 5 shows my pseudocode for this problem, **Time complexity:** $O(n^2)$.

5. Please review the papers to find an algorithmic problem related to the non-obvious network flow or circulation problem, summarize its problem formulation, and transform it into the network flow problem.

Answer.
I reviewed the paper *On effective and efficient in-field TSV repair for stacked 3D ICs. (DAC '13)*, which was mentioned briefly in class. This paper describes a reconfigurable in-field repair solution to tolerate latent TSV (through-silicon-via) defects through the judicious use of spares.

Problem formulation: Given a set of signals $S = \{s_1, s_2, \cdots, s_n\}$ and a set of TSVs $T = \{TSV_1, TSV_2, \cdots, TSV_m\}$, $(n < m)$. The goal is to link every signal in S with a dedicated TSV in T. Considering the physical constraints and design requirements of the actual problem, some links (s, TSV) are invalid. And with the hardware aging, some valid links may become invalid. To check whether a link (s, TSV) is valid, we need to invoke testing procedure which may be expensive. So we need to link every signal to a valid TSV and update the assignment when some links become invalid, and invoke as few as possible testing procedures.

This problem can be transformed into a bipartite graph matching problem. To reduce the invoking of testing procedure, it is necessary to reuse the previous information. So we do not run a random maximum matching for every updating. Instead, when some invalid links are detected, we remove the failed links, and run the alternative path algorithm for the mismatched signals. When we find a new matching, invoke the testing procedure, if faults are detected, repeat the process above.

Remark: You need to upload your .pdf file and write the pseudocode.