# EE3980 Algorithms

Homework 6. Power Ranking

By 105061212 王家駿

2019/04/11

## 1. Introduction

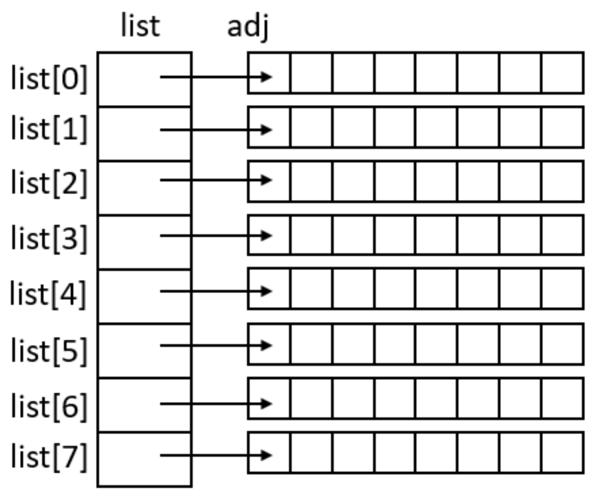
In this homework, we record each player's name and the result of each match. Then we use these data to create a ranking, where a player ranked high means the player wins the contests against players whose ranks are lower. In the end, we construct a list to represent the ranking of every players.

## 2. Implementation

In the program, we first read in the input data, and store them in the data structure. We could view each player as a vertex and each game as an edge, so we could view the whole input data as a directed graph, where the tail of edge is the winner and loser for the head. Then, we use topological sort which includes DFS (Depth First Search) algorithm on the graph to construct the ranking list.

### 2.1. Data structure

In order to store all the input data and use the DFS algorithm, we use the adjacency list for storing the players' names and the result of matches to represent the directed graph. The structure of the list is like this (take 8 players for example):



Each list[i] represents one vertex (player), in which contains the player's name and an array recording the successors' index.

To store the names, we mention that all the input players' names are two Chinese characters. Since the space of a Chinese character is three bytes in UTF-8, we allocate strings with six bytes for the players' names.

The array *adj* stores the data of the successors, which means if the player wins the match, the loser's index would be added at the top of the winner's array. Since for V players, a player can have at most V-1 matches with different opponents, we allocate V-1 spaces for each array *adj*, and initialize them with the same value. Thus, we represent the directed graph by adjacency list, with the two-dimensional array like structure.

Besides, we also construct two arrays *visit* and *result* for the DFS algorithm we would use. *visit* stores the information of whether the vertex was visited, 0 for not been visited and 1 for been visited. *result* stores the output data after topological sort.

## 2.2. Topological sort

After the adjacency list was constructed and all data were stored well. We use the topological sort to find the ranking.

Topological sort uses the method of DFS. We keep find the vertex's successor until the finding goes to the end, and then find another successor (if any). The DFS acts like this way until all the vertices have been found. To prevent a vertex from being found twice, we use the *visit* array to check if the vertex had been found before.

When the visiting of the vertex is completed, we record the vertex into the end of the *result* array. That is because in the DFS algorithm, the visiting of a vertex is completed only when all its successors had been visited. Thus, we could ensure that the losers would be written into the *result* list before the vertex itself, and get the ranking of the visited vertices.

```
Algorithm top_sort(v) {
    visited[v] := 1;
    for each vertex w adjacent to v do {
        if (visited[w] = 0) then top_sort(w);
    }
    add v to the head of list;
7.  }
```

After all the vertices were visited, all of them would be added to the list. So, the ranking would include all players. Thus, we need a function call to ensure that every vertex had been visited.

```
1.  Algorithm topsort_call(G,list)
2.  {
3.      // initialize
4.      for v := 1 to V do visited[v] := 0;
5.      list := NULL;
6.  
7.      // top_sort for every vertices
8.      for v := 1 to V do
9.          if (visited[v] = 0) then top_sort(v);
10. }
```

For the time complexity, in the top_sort function, the assignment at line 2 and the addition to the list at line 6 take constant time. So, we are only interested in the recursion part. For the loop at line 3, it would execute at most k times, where k is the number of vertices adjacent to the vertex.

Now we examine the topsort_call function. For the loop at line 8, it would run V times and lead to V topological sorts. However, if the vertex had been visited, the topological sort would not be operated. Thus, we could ensure that each vertex would been visited one time, which cause the time complexity of $O(V)$.

Since every vertex were visited, all edges were visited, too, and each edge would been visited once. Thus, from the conclusion we got in the last two sections, the summation of k for all vertices is E. So, we know that the total time complexity is $O(V+E)$ , which is linear, since the vertices and edges are visited once.

For the space complexity, it might be $O(n^2)$ to store the input data because we construct the list with the size V*V. Then, we need additional space of V*2, contributed by the two arrays *visit* and *result*.

Time complexity: $O(V+E)$

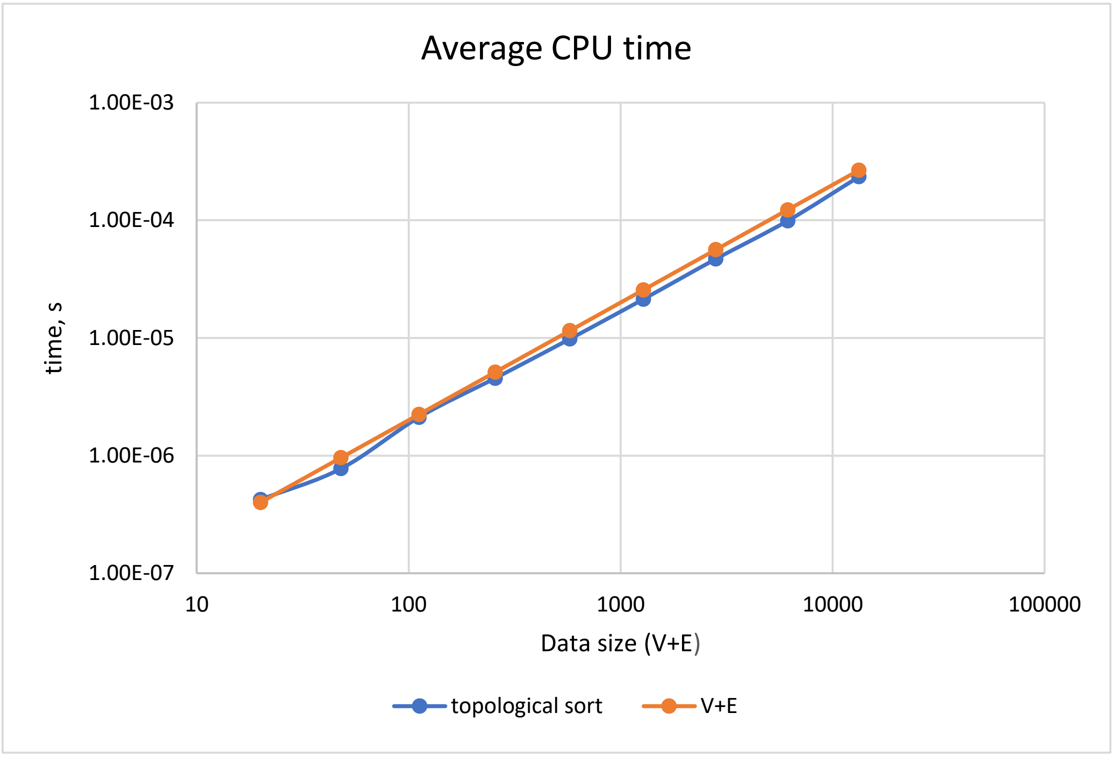
Space complexity: $O(V^2)$ (input data) + $O(V)$ (addition space)

## 3. Executing results

We run the testing data from g1.dat to g9.dat with different input data size for 100,000 times, record the average CPU time used, and compare to our estimation.

| **Number of players (V)** | **Number of games (E)** | **Data size (V+E)** | **Average time** |
|---|---|---|---|
| 8 | 12 | 20 | 422.5ns |
| 16 | 32 | 48 | 775.2ns |
| 32 | 80 | 112 | 2.119 $\mu$ s |
| 64 | 192 | 256 | 4.551 $\mu$ s |
| 128 | 448 | 576 | 9.801 $\mu$ s |

| | | | |
|---|---|---|---|
| 256 | 1024 | 1280 | 21.35 $\mu$ s |
| 512 | 2304 | 2816 | 46.98 $\mu$ s |
| 1024 | 5120 | 6144 | 99.11 $\mu$ s |
| 2048 | 11264 | 13312 | 234.5 $\mu$ s |



## 4. Result analysis and conclusion

From the graph, we could find that the result has a trend of V+E, which is O(n), and it is same as our estimation. That is, the topological sort algorithm with DFS has a linear time complexity.

The least time complexity of the power ranking problem must be $O(V + E)$, because we have to first read the input, which are V+E data, and it cause the time complexity of $O(V + E)$.

For a sparse graph, where the number of edges is close to the number of vertices, the time complexity is $O(V + E) \cong O(2V) = O(V)$. However, it would lead to a waste of space in our implementation since we allocate fixed size arrays for adjacency lists. So, there are lots of space would not be used. The solution could be replacing the arrays by linked lists for the adjacency list, with a pointer pointing to the next element.

While for a dense graph, where the number of edges is close to square of the number of vertices, the time complexity is $O(V + E) \cong O(V + V^2) = O(V^2)$. Since there are many edges, it would lead to a waste of time during the operation of the linked lists, and also a waste of space because the linked list occupies more spaces than array when storing the same amount of data. Thus, it would be more feasible by using arrays, and it’s also easily for coding.

All in all, there’s a trade-off between the array and linked list representation. They might be used with regard to different input data and what we are concerned with, time or space. So, it’s important to choose the data structure and implementation methods under different circumstances.