Implementation of Maximum Flow Algorithms

IT300 Design and Analysis of Algorithms

Laharish S
181IT125

Neil Poonatar
181IT130

Nirmal Khedkar
181IT230

Piyush Ingale
181IT232

Problem Statement

Cricket Elimination Problem:

To determine whether a team x belonging to a set of teams S, can end up being on top of the scoreboard by the end of the tournament. That is whether a team is eliminated of not, given the standings at a particular point of time in the ongoing tournament.

Approach to solve the problem

We focus on a team (say x) at a time. Terminologies include:

w_x: no of games won by team x till now.

g_x: no of games remaining for team x.

g_xy: no of games yet to be played between team x and team y.

m_x = w_x + g_x: this is the maximum number of games a team can win.

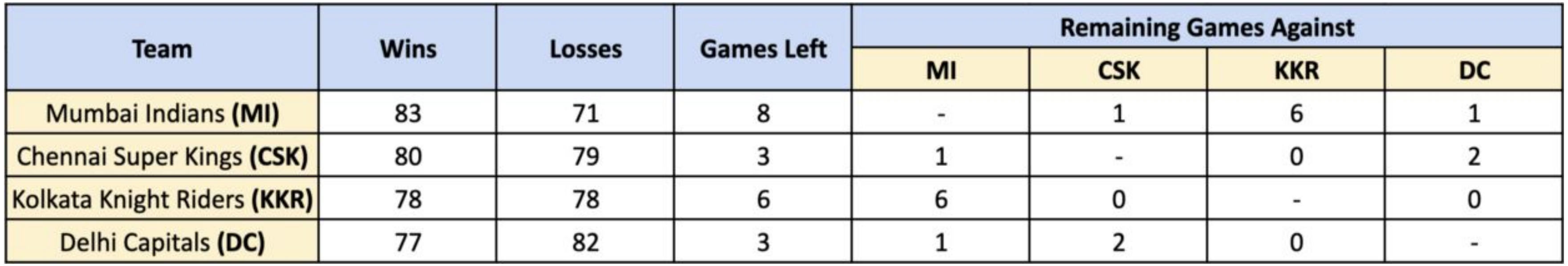
Let's say that the leading team has won q games. We may say that for team x to win the tournament, it should win more that q games.

Ie. m_x >= q

This condition is necessary but not sufficient. As leader will have to lose some games and remaining teams will be benefited from this.

To solve the problem effectively, we need to consider other teams too. The solution is to make a graph such that it simulates the flow of wins and to find the maxflow of that graph.

For example we have current scenario as follows:



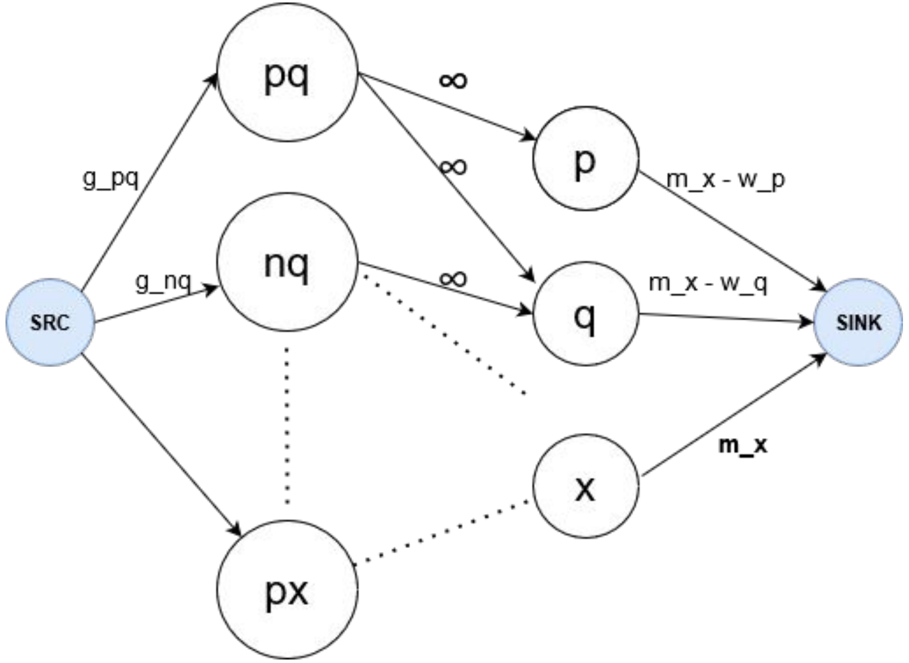
Team	Wins	Losses	Games Left	Remaining Games Against			
				MI	**CSK**	**KKR**	**DC**
Mumbai Indians **(MI)**	83	71	8	-	1	6	1
Chennai Super Kings **(CSK)**	80	79	3	1	-	0	2
Kolkata Knight Riders **(KKR)**	78	78	6	6	0	-	0
Delhi Capitals **(DC)**	77	82	3	1	2	0	-

Here DC is already eliminated as even if DC wins remaining games, it'll have maximum 77+3 = 80 wins and MI already has 83 wins. But we cant tell whether CSK is eliminated or not by simply looking at the table and using the above mentioned intuition

We create a graph having 1st layer of nodes as the remaining games between any two teams.

The capacity of the edges connecting these nodes to source nodes are g_xy.

Second layer of nodes comprises of all the teams in the tournament. And the edges connecting these nodes to the sink node represents the maximum number of games allowed for that particular team such that team x wins maximum games.
We define g* = Σ g_p   (for all p in S)
If the maxflow of the graph is greater than or equal to g*, team is not eliminated.



Ford–Fulkerson algorithm

A greedy algorithm that computes the maximum flow in a flow network.

Published in 1956 by L. R. Ford Jr. and D. R. Fulkerson.

Sometimes called a "method" instead of an "algorithm" as the approach to finding augmenting paths in a residual graph is not fully specified

Idea behind the algorithm: as long as there is a path from the start node to the end node, with available capacity on all edges in the path, we send flow along one of the paths. Then we find another path, and so on.

A path with available capacity is called an augmenting path.

Algorithm

Let G (V,E) be a graph, and for each edge from *u* to *v*, let c(u,v) be the capacity and f(u,v) be the flow
We define the residual network Gf(V,Ef) to be the network with capacity cf (u,v) = c(u,v) − f(u,v) and no flow.

I/P: Given a Network G = (V,E), with flow capacity *c*, a source node *s*, and a sink node *t*
O/P: Compute a flow *f* from *s* to *t* of maximum value

```
1. f (u,v) ← 0 for all edges (u,v)
2. While there is a path p from s to t in Gf, such that cf(u,v) > 0 for all edges (u,v) ∈ p :
    1. Find cf( p ) = min { cf(u,v) : (u,v) ∈ p }
    2. For each edge ( u , v ) ∈ p
        1. f (u,v) ← f(u,v) + cf(p) //Send flow along the path
        2. f (v,u) ← f (v,u) − cf(p) //The flow might be returned later
```

Implementation

Class Edge

- __init__(u, v, capacity)
- __repr__()

Class FlowGraph

- __init__()
- add_node(node)
- get_edges(node)
- add_edge(u, v, capacity=0)
- get_augmenting_path(s, t, path, set_path)
- ford_fulkerson(s, t)

Analysis

The maximum flow will be reached when no more flow augmenting paths can be found in the graph.

However, there is no certainty that this situation will ever be reached, In the case that the algorithm runs forever, the flow might not even converge towards the maximum flow. However, this situation only occurs with irrational flow values.

When the capacities are integers, the runtime of Ford–Fulkerson is bounded by O(Ef) (each augmenting path can be found in O(E) time and increases the flow by an integer amount of at least 1, with the upper bound f.

Edmonds–Karp algorithm

A variation of the Ford–Fulkerson algorithm with guaranteed termination and a runtime independent of the maximum flow value.

The algorithm is identical to the Ford–Fulkerson algorithm, except that the search order when finding the augmenting path is defined.

The path found must be a shortest path that has available capacity. This can be found by a breadth-first search, where we apply a weight of 1 to each edge.

The length of the shortest augmenting path in Edmonds-Karp increases monotonically.

Analysis

The running time of Edmonds Karp Algorithm is: O(|V|| E|^2)

This is found by showing that each augmenting path can be found in O(|E|) time, that every time at least one of the E edges becomes saturated (an edge which has the maximum possible flow), that the distance from the saturated edge to the source along the augmenting path must be longer than last time it was saturated, and that the length is at most |V|.

Dinic's algorithm

A strongly polynomial algorithm for computing the maximum flow in a flow network, conceived in 1970 by Yefim A. Dinitz

Runs in O (V^2E) time and is similar to the Edmonds–Karp algorithm, in that it uses shortest augmenting paths.

The concepts of the level graph and blocking flow enable Dinic's algorithm to achieve its performance.

Algorithm

I/P: A network G=((V,E),c,s,t)} .

O/P: An s–t flow f of maximum value.

1. Set f(e)=0 for each e ∈ E
2. Construct GL from Gf of G. If dist (t) = ∞ , stop and output f.
3. Find a blocking flow f′ in GL.
4. Add augment flow f by f′ and go back to step 2.

Implementation

make_auxiliar_network(edges)

get_path(na, edges)

get_residual(n, key, edges)

augment(na, edges, path, mincost)

augment_and_delete(v, first, last, mincost, edges, na)

dinic(edges)

get_corte(na):

read_edges(f=sys.stdin):

run_dinic(file_name)

Analysis

It can be shown that the number of layers in each blocking flow increases by at least 1 each time and thus there are at most |V|-1 blocking flows in the algorithm. For each of them:

- the level graph GL can be constructed by breadth-first search in O(E) time
- a blocking flow in the level graph GL can be found in O(VE) time

With total running time O(E+VE)=O(VE) for each layer. As a consequence, the Therefore running time of Dinic's algorithm is O(V^2E).

Improvement: dynamic trees O(VElogV)

Special cases: Unit capacities, Bipartite Matching (Hopcroft-Karp)

Push Relabel (aka preflow–push algorithm)

Each vertex has:

- Height variable: used to determine whether a vertex can push flow to an adjacent or not (A vertex can push flow only to a smaller height vertex).
- Excess flow is the difference of total flow coming into the vertex minus the total flow going out of the vertex

Each edge has:

- a flow
- and a capacity

Push Relabel Algorithm

Similarities with Ford Fulkerson

- Push-Relabel also works on Residual Graph

Differences with Ford Fulkerson

- Push-relabel algorithm works in a more localized.
- In Ford-Fulkerson, net difference between total outflow and total inflow for every vertex is maintained at 0. Push-Relabel algorithm allows inflow to exceed the outflow before reaching the final flow. In final flow, the net difference is 0 for all except source and sink.
- Time complexity wise more efficient.

Algorithm

```
1. Initialize PreFlow : Initialize Flows and Heights

2. While it is possible to perform a Push() or Relablel() on a vertex

   // Or while there is a vertex that has excess flow

         Do Push() or Relabel()

// At this point all vertices have Excess Flow as 0 (Except source and sink)

3. Return flow.
```

Analysis

The push–relabel algorithm is considered one of the most efficient maximum flow algorithms.

In sum, the algorithm executes O(V^2) relabels, O(VE) saturating pushes and O(V^2E) non-saturating pushes. Therefore, the time complexity of the algorithm is O(V^2E)

Specific variants of the algorithms achieve even lower time complexities

The push–relabel algorithm has been extended to compute minimum cost flows

Thank You