Baseball Elimination using Ford-Fulkerson and Edmond's-Karp algorithm

Jasper Mishra

CSE 417 Winter 2013

8th March , 2013

Introduction

Given a scenario in a game of multiple teams where each one has certain wins midway through the game. Suppose we want to find the team to be eliminated in order for other teams to proceed without having to play extra games. In other words, if a team has certain amount of wins such that even if it plays and wins all of its remaining games, it wouldn't have its win greater than or equal to any other team in the series, then it needs to be eliminated to conserve time and resources. The goal of this project is to devise an algorithm that finds this condition to hold true for the eliminating team, using either Ford-Fulkerson or Edmond's-Karp algorithm

Description

Theory

[1] Suppose we have a set S of teams, and for each $x \in S$, its current number of wins is w_x. Also for two teams x, $y \in S$ they still have to play g_{xy} games against each other. Finally we are given a specific team z. The algorithm then evaluates as follows:

Suppose that team z has already been eliminated. Then there exists a proof of this fact of the following form:

- *z can finish with at most m wins*
- *There is a set of teams* $T \subseteq S$ *so that*

$$\sum_{x \in T} w_x + \sum_{x,y \in T} g_{xy} > m|T|$$

(And hence one of the teams in T must end with strictly more than m wins)

As implied above, we need to find an algorithm that finds if the sum of the wins and sum of the games to be played for all teams excluding *z* should be greater than $m|T|$ since at least one team (which is of course, not *z*) must end up with wins that is greater than *z*'s wins in order for *z* to be eliminated.

We solve the above problem using the traditional min-flow/max-cut network flow problem.

First, let $S' = S - \{z\}$, and let $g^* = \sum_{x,y \in S} g_{xy}$ the total number of games left between all pairs of teams in S'. We include nodes s and *t,* a node v_x for each team $x \in S'$, and a node u_{xy} for each pair of teams $x, y \in S'$ with a nonzero number of games left to play against each other. We have the following edges.

- Edges *(s, u_{xy}) (wins emanate from* s);
- Edges *(u_{xy}, v_x) and (u_{xy},* v_y) *(only x or y can win a game that they play against each other) and*
- Edges *(v_x, t) (wins are absorbed at t).*

We assign *(s, u_{xy})* a capacity of g_{xy} since we want g_{xy} wins to flow from *s* to u_{xy}. We give a capcity of $m - w_x$ to the edge *(v_x, t)* since we want to ensure team *x* cannot win more than $m - w_x$ games. Finally, an edge from u_{xy} should at least have a capacity of g_{xy} so that it has the ability to transport all the flows from u_{xy} to v_x

Now, if there is a flow of value g*, then it is possible for the outcomes of all remaining games to yield a situation where no team has more than m wins; and hence,'if team z wins all its remaining games, it can still achieve at least a tie for first place. Conversely, if there are outcomes for the remaining games in which z achieves at least a tie, we can use these outcomes to define a flow of value g*. This means that the maximum flow through the network should be strictly less than g* in order for team *z* to be eliminated. Hence, we have the following conclusion:

Team z has been eliminated if and only if the maximum flow in G has value strictly less than g. Thus we can test in polynomial time whether z has been eliminated.*

Implementation

There are two ways to implement the above mentioned algorithm. The first is Ford-Fulkerson and the other is Edmond's-Karp. Both find the value of *g*--maximum-flow/min-cut in the network. The difference being only that Ford-Fulkerson uses Depth First Search whereas Edmond's-Karp uses Bread-First Search. Moreover, Edmond's-Karp fits well for irrational edge capacities whereas Ford-Fulkerson only works with integers [3].

Following is the core implementation. Using DFS(..) in the computeFlow(..) [2] We find min-cut using Depth First Search hence turning it into Ford-Fulkerson algorithm. Using BFS(..) employs Breadth First Search, hence turning computeFlow(..) into Edmonds-Karp algorithm

```
# Compute max flow using Depth First Search. The max flow algorithm is then
Ford Fulkerson Algorithm

def DFS(v1,v2, path):
    if v1 == v2:
        return path
    for edge in adjacency_list[v1]:
        residual = edge.w - optFlow[edge]
        if residual>0 and (edge, residual) not in path:
            foundPath = DFS(edge.v2, v2, path + (edge, resCapacity))
            if foundPath is not NULL
               return foundPath


# Computer max flow using Breadth First Search. The algorithm then becomes
Edmond's Karp Algorithm
def BFS(source, sink):
    queue = [source]
    paths = {source:[]}
    while queue:
        v1 = queue.pop(0)
        for edge in adjacency_list[v1]:
            resCapacity = edge.w - optFlow[edge]
            if resCapacity > 0 and (not (edge.v2 in paths)):
                paths[edge.v2] = paths[v1] + [(edge, resCapacity)]
                if edge.v2 == sink:
                    return paths[edge.v2]
                queue.append(edge.v2)
    return None


def computeFlow(source, sink):
    if fordFulkerson == 1:
        augmentedPath = DFS(source, sink, [])
    else:
        augmentedPath = BFS(source, sink)
    while augmentedPath:
        maxCapacity = sys.maxsize
        for component in augmentedPath:
            edge, resCapacity = component
```

```
            if (resCapacity < maxCapacity):
                maxCapacity = resCapacity
        for component in augmentedPath:
            edge, res = component
            ptFlow[edge] = optFlow[edge] + maxCapacity
            optFlow[edge.res] = optFlow[edge.res] - maxCapacity
    augmentedPath = DFS(source, sink, [])
minCut = 0
for edge in adjacency_list[source]:
    minCut = minCut + optFlow[edge]
return minCut
```

Runtime and Complexity

Runtime: O(E*f) where E is the number of edges and f is the maximum flow of the network

Using either Ford-Fulkerson or Edmond's-Karp the runtime of the algorithm is *O(Ef)* where *E* is the set of all edges in the algorithm and *f* is the maximum flow through the network. This makes sense, since by adding augmenting path to the flow already established in the graph, the maximum flow will be reached when no more flow augmenting paths can be found in the graph. Hence, every edge is explored to find the augmenting path

Since the capacity of the minimum cut is n*f where f is maximum capacity of the concerned augmented path and n is the number of edges in that path, it follows that there are at most O(nf) iterations. Since each iteration takes *O(E)* time to find a path and augmented flow using either DFS or BFS, the total complexity then is *O(nEf)* or simply *O(Ef)* since the number *n* of edges in each path varies with path.

Test Cases and Observations

Small Suit:

For this example, we used the textbook problem of selecting 4 baseball teams: *New York, Tornoto , Boston, Baltimore* with their respective initial wins given by *90, 88, 87, 79*. Currently, *New York and Toronto have 6* games left between each other, *Baltimore and New York have 1, Toronto and Baltimore have 1*, and *Boston has 3 games each against the rest of the teams*. Looking at the current wins, things don't look good for Boston and we want to find out if it has been eliminated. Hence we construct a graph where:

Input:

Set of vertices *V = {Source, Sink, New York, Toronto, Baltimore, (Toronto, New York), (New York, Baltimore), (Baltimore Toronto)}*

Set of Edged *E = {Source-(Toronto, New York), {Source-(Toronto, New York) , Source- (New York, Baltimore), Source -(Baltimore Toronto), Tornoto-Sink: (m – $w_{Toronto}$), New York –Sink:(m – $w_{NewYork}$), Baltimore –Sink:(m – $w_{Baltimore}$), (Toronto, New York)-Toronto, (Toronto, New York)-New York, (Baltimorase, Toronto)-Toronto, (Baltimore, Toronto)-Baltimore, (New York, Baltimore)-Baltimore, (New York, Baltimore)-New York:}*

Output

Eliminating team Boston needs at least 91

g -> 8*

g --> 7

Boston has been eliminated

Feeding the above graph *G*:=[*V,E*] in our algorithm gives us a result of g*= 8 indicating the total possible flow, and the value g = 7 indicating the actual optimal flow. Since g < g* we find that correctness of our algorithm and conclude that Boston has certainly been eliminated.

Large Suit

Input:

For large datasets, finding actual team elimination stats in real-life tournaments was difficult since elimination criteria depend on several parameters and hence our result would have not coincided with the actual results. To overcome this, we constructed several random teams where if *numTeam* is the maximum size of the number of teams participating, then.

Set *V = {S, T, randomTeam$_1$, randomTeam$_2$, randomTeam$_3$, randomTeam$_4$,……., randomTeam$_{numTeam}$ }*

Set E = {S-(randomTeam$_i$, randomTeam$_k$), S-(randomTeam$_a$, randomTeam$_b$), S-(randomTeam$_i$ / randomTeam$_s$)..... (randomTeam$_i$, randomTeam$_k$)- randomTeam$_i$, (randomTeam$_i$, randomTeam$_k$)- randomTeam$_k$, ……. randomTeam$_i$-T, randomTeam$_a$, randomTeam$_k$- T, ...}

In short, two teams are randomly picked from a list of *numTeams* and assigned a random number of left-over games usually in a specific range. This range applies to all such randomly picked pairs. Finally, the team whose elimination is to be tested is also randomly selected and excluded from *G:*=[*V,E*]

Output

Following are test results of g and g* for various random instances of the problem with *numTeams=30*

$g^* \text{ -> } 135, g \text{ --> } 131$

$g^* \text{ -> } 128, g \text{ --> } 126$

$g^* \text{ -> } 118, g \text{ --> } 110$

$g^* \text{ -> } 122, g \text{ --> } 120$

$g^* \text{ -> } 110, g \text{ --> } 107$

$g^* \text{ -> } 95, g \text{ --> } 93$

Feeding the above graph into our algorithm we see that the algorithm does not converge for inputs larger than in the range 50-75. This is because since the augmenting path algorithm does not apply any heuristics, it explores many such paths in this dense graph and as such the algorithm fails to converge.

There are some alternatives that can be done to "work-around" a possible convergence in the above case of randomly setting up the game scenario:

1) Low range of randomly assigned leftover games, i.e., the difference between the team having the highest number of games left to be played and the team having the lowest should be as low as possible.
2) Low range of randomly assigned wins, i.e., the difference between the teams having the highest number of wins and the team having the lowest should be significantly low

Conclusion

Although the augmented path algorithm, whether Ford-Fulkerson or Edmonds-Karp helps us find the minimum-cut and max-flow of a given flow network, it doesn't guarantee convergence for large dense graphs. This is because the way augmenting paths are picked is not specified and hence arbitrary exploration algorithms that only rely on capacities, such as DFS and BFS are not ideal for larger datasets.

Nevertheless, for our game problem scenario, the algorithm converges and yields proper results that coincide with the actual outcome.

Source Code

Click here to get the python code.

References

[1] Kleinbegr and Tardos *Algorithm Design*

[2] Page 13, Maximum flow Algorithm
http://www.cs.purdue.edu/homes/ayg/CS251/slides/chap10c.pdf

[3] Page 92, Lecture 17, More on Max flow
http://www.cs.cornell.edu/courses/CS6820/2012sp/Handouts/84-99.pdf