Michael McQuade

Josh McMahan

Ivan Guerrero

Analysis & Design of Algorithms

Project Evaluation Problem – Final Project

For the final project, we chose to do the project evaluation problem because we found it to be the most interesting after hearing about how it was assigned to our professor at a real job. We thought this would give us good experience in trying to solve a real-world problem using the techniques we learned in this class.

This problem can be mapped to several known problems in Computer Science. For instance, one approach is to map the problem to the Stable Marriage problem. In the stable marriage problem, you begin with two sets of elements that are equal in size. The goal is to find an ideal match to the second set from the first set for every element in the set. Once you have a map consisting of all the individual elements from set A mapped to individual elements in set B, you can check to see if the “marriage is stable.” Hence, the stable marriage problem.

A pairing, or marriage, is considered stable when there exists no pair which would be better matched with each other instead of the partner they were matched during the mapping process. By this, it means that the ideal match for an element in A is not the element that it is matched to in B, but some other element, and likewise that same element in B has an ideal match of that same element from A, but is matched to some other element in A.

In the case of our project evaluation problem, each project could be considered as a bride, and each expert as a groom. There is a known stable matching algorithm which can be used to match the pairs.

Another problem that is similar to the problem we faced with the project evaluation problem is the maximum flow problem. Maximum flow problems have been around since the mid 1950s and can be demonstrated with many real world issues.

The idea is that you have a source and a sink, and a network of nodes which connects the source to the sink. Each node in the network has a capacity. The source is trying to flow through the network to the sink. This should be done in a way that moves as much as possible across the network at one time, without overflowing the network. By overflowing the network, we mean to say that no node is ever exceeding its capacity. The simplest network for a maximum flow problem would be a straight line where the source flows directly to the sink or through one node. The node will always be at full capacity and thus there is no algorithm to consider because the flow is already maximum.

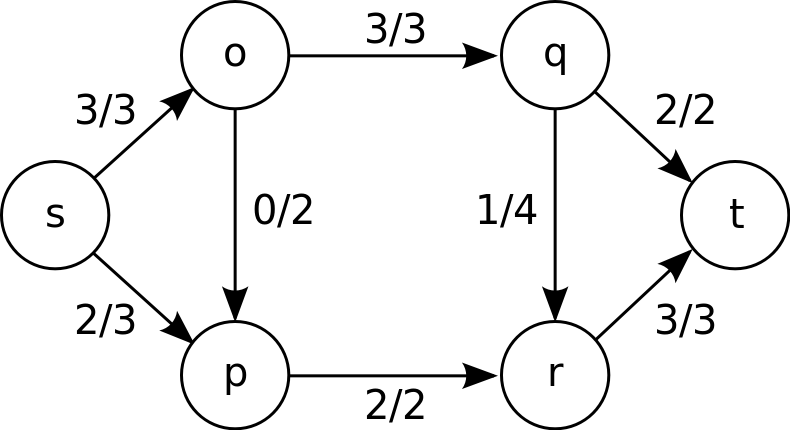


Figure 1 - An example of maximum flow, where s is the source and t is the sink. Image from https://en.wikipedia.org/wiki/Maximum\_flow\_problem

An easy way to consider this in a real-world example is a railway system. The source is the starting location and the sink is the ending location, and nodes represent railroad stations in-between the starting and ending locations. You have many trains that need to arrive at the same destination, how can you send the trains in a way that there aren’t large delays or overflow? In this case overflow is very bad, because it would mean two trains are trying to arrive at the same station at the same time.

One solution to the maximum flow problem is the Ford-Fulkerson method. Basically, the algorithm is that traffic should be sent through the network using the first path found from the source to the sink. Then, after that path has no more capacity, we send traffic through the next path available, until there are no more paths from the source to the sink. This method is considered a greedy algorithm, as it is only concerns itself with what is locally optimal at any given time, not considering other paths while it still has a path available from source to sink.

One way to connect the maximum flow problem to the project evaluation problem that we chose is to consider incoming projects as the source, and experts as nodes. Find a path through the nodes that distributes all possible projects. If a project reaches the sink, then it can be considered that there was no expert available for this project. This type of approach is a greedy approach as if experts are in nodes that would not be traversed by the algorithm and they reach the sink a false report of no experts available is possible. For example, see figure 1 edge from q to r.

An application of the Ford-Fulkerson method could be used in our problem as well. Since no restrictions were imposed on how the project must be distributed, a greedy approach would be to simply give all the projects to one expert until that expert could no longer accept any more projects, and then continue to the next expert. We didn’t choose this method because our situation is modeled after a real world scenario. Leaving an expert without any work while piling more work onto someone else is suboptimal and would definitely result in declined performance.

However, in our project, there are very few variables given on how projects should be matched to experts. Because of this, neither of these two problems necessarily needed to be applied, but were referenced for guidance.

The conditions that we have chosen to impose are:

* All projects are distributed evenly to eligible experts until everyone has one project
* Experts and projects have an area of expertise, and only one area of expertise
* Experts are not eligible for a project if they do not share the projects area of expertise
* Experts have a capacity, or maximum amount of projects assignable, after which they are not considered for additional projects
* After all eligible experts have one project, projects are distributed based on the ratio of their capacity after receiving the project and total capacity

The algorithm that we used to implement these conditions is as follows:

Experts[area] = [*expert1*, *expert2*, . . . , *expertn*]

function distributeProject(project){

Score= 1

ExpertArea = Experts[area]

for expert in ExpertArea:

if expert’s current capacity === 0

continue

if expert’s assigned projects === 0{

candidate = expert

break

} else {

if Ratio of next capacity and total capacity <= Score {

Score = Ratio of next capacity and total capacity

candidate = expert

}

}

return candidate

}

The time complexity of this algorithm is . Even though we have an array inside of an array, we are only ever dealing with a single array at a time. That is, we are only ever working with the array for the experts that share the same area as the project we aim to distribute. Thus, we don’t have any multiplications done to our time complexity. The time complexity is always after the initial condition of everyone having a project assigned is fulfilled. Before that, it breaks from the loop when an expert with no assigned projects is found.

The space complexity is also . We do not store anything within the algorithm itself, but on continuous runs of the distribute project function our array containing the experts grows as the projects are being added.

Our sample run can be found here:

<https://www.youtube.com/watch?v=7oQGTyEVh_Y>

This website hosts a live version of the project:

<https://giraffesyo.io/project-evaluation-problem>

There are two versions of the source code. The first is a web application (linked above) which we made to visualize the algorithm. The second is a simple version which we made to demonstrate the algorithm without the huge overhead of looking through our web application’s code.

To run the web application locally, you need to have the yarn package manager installed. Run ‘yarn install’ and from the react-app directory then you may run ‘yarn start’ from the same location to launch the web app.

To run the simple version you can either run it using Chrome Developer Tools or Node.js using the command ‘node simple.js’