MIT 18.996: Topic in TCS: Internet Research Problems	Spring 2002

Lecture 7 — March 20, 2002

Lecturer: Brian Dean *Scribe: John Kogel, Ben Leong*

Global Load Balancing

In today's lecture, we discuss global load balancing problems and formulation of such problems as instances of network flow problems and stable marriage problems.

7.1 Introduction/Motivation

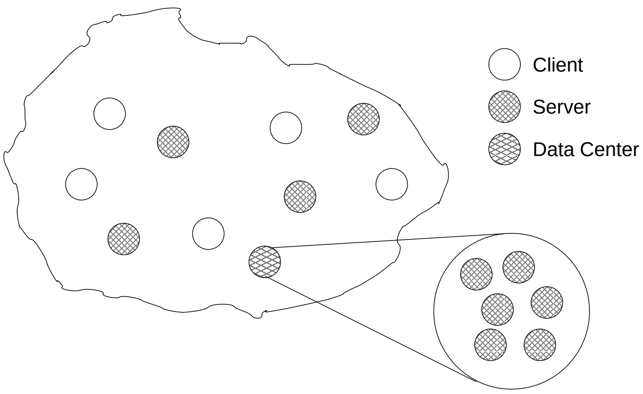


Figure 7.1. Simple model of the Internet.

In our model, the Internet consists of a collection of clients, servers and data centers. A data center is simply a group of servers all situated at the same place. Our goal is to map the clients to the servers (or data centers) given a set of constraints. In our model, a client can be serviced by any server (this is in contrast to the client-cache model present the previous week where data is available only on selected servers. Our goal is to ensure that locality is enforced and clients are served by servers which are located near them.

7.1.1 Objectives

In making assignments, the following are the objectives that we would like to meet:

1. Assign clients to "close" servers. We can define distance according to various measures (i.e., ping latency or AS hops).
2. Map clients to servers with low utilization. As shown in Figure 7.2, we will consider 2 ways of modeling the cost of utilization. In one model, there is no cost until the load on a server

reaches a hard capacity C_{max}, and thereafter the cost is infinite; in the second model, cost increases as a (typically convex) function of utilization.

3. Minimize the bandwidth cost. The cost of bandwidth is typically a concave function with respect to bandwidth utilization as shown in Figure 7.3.

We note that objectives 1 and 2 are desirable from the perspective of the clients/users while objective 3 is desirable from the perspective of the servers/providers. Also, the first two objectives usuallly act in opposition to the third. For the purposes of this lecture, we will focus on objectives 1 and 2.

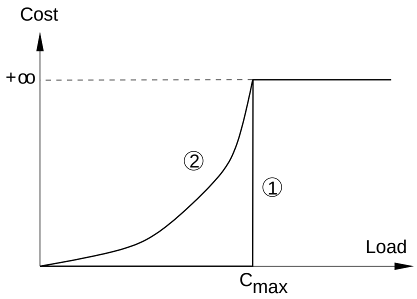


Figure 7.2. Possible cost models.

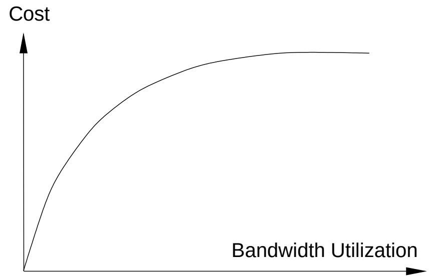


Figure 7.3. Graph of cost versus bandwidth.

7.1.2 What Happens in Practice

We'll focus in this talk on ways to model and solve abstract load balancing problems; however, actual load balancing systems typically involve much more complex machinery than just a core algorithm. In practice, our load balancing system may need to do quite a bit of work just to maintain the problem instance (client information, server information, and locality information between the two) which is periodically fed to the load balancing algorithm. In practice, the number of clients is too large for us to assign them individually to servers, so we usually aggregate clients in some fashion (i.e., by network or by IP address). Servers need to monitored to make sure that if a server crashes, we remove it from the problem instance. Finally, one must somehow determine and maintain a measure of locality (e.g. AS hops) between clients and servers.

Our load balancing algorithm will periodically re-compute assignments from clients to servers, which then need to be published (e.g., using DNS). In terms of timescale, any time a client is reassigned to a new server it takes some time for traffic from that client to migrate fully to the new server, especially for long connections such as streams. So we are looking at a system that will probably be updating its assignments every few seconds or few minutes.

7.2 Network Flows

In this section we describe how to formulate increasingly-sophisticated variants of the client-server assignment problem each as network flow problems.

7.2.1 Finding a Feasible Assignment

We start with the following basic formulation of the problem: we have n clients (which are actually aggregates of individual clients as described above) and m servers. Each client i has an associated $demand(i)$ and each server j has an associated $capacity(j)$. Each client can be mapped to a subset of the servers (presumably those close to it). We can represent this problem instance using a directed bipartite graph as shown in Figure 7.4. We allow the demand for a client to be split between various servers, and we want to determine if a feasible assignment exists which satisfies the client demands and respects the server capacities.

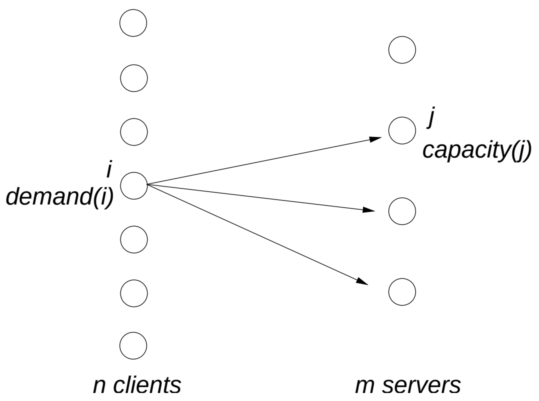


Figure 7.4. Basic model.

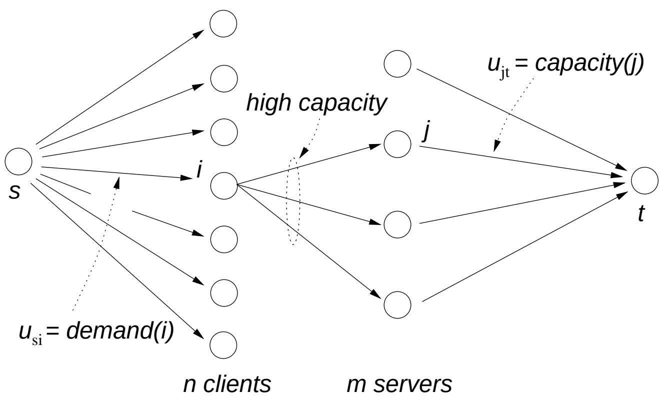


Figure 7.5. As a max-flow problem.

To solve this problem, we transform it into an equivalent max-flow problem by adding a source node, s, and a sink node, t. We also add an edge between s and all the clients and also an edge between all the servers and t. The capacities for the source-client edges are set as $demand(i)$ while that for the server-sink edges are set as $capacity(j)$. A feasible assignment exists (given by the flows on the interior edges) only if the max s-t flow saturates all of the edges leaving the source.

The well-known max-flow problem is defined as follows: Given a directed graph $G = (V, E)$, source and sink nodes s and t, and non-negative capacities u_{ij} for all edges $(i, j) \in E$, find the maximum flow $x \in \mathcal{R}^{|E|}$ one may send from the source to the sink. As a mathematical program,

$$\begin{array}{rcl} \text{Maximize} & \sum_{i:(s,i)\in E} x_{si} & \\ \forall \text{ nodes } i \in V\backslash\{s,t\} : & \sum_j x_{ij} - \sum_j x_{ji} = 0 & \text{(Flow Conservation)} \\ \forall \text{ edges } (i,j) \in E : & 0 \le x_{ij} \le u_{ij} & \text{(Lower and Upper Capacities)} \end{array}$$

The max-flow problem has been extremely well-studied and many efficient solution algorithms exist for its solution.

7.2.2 Minimizing Maximum Utilization

We next consider an extension of our problem in which we wish to minimize the maximum utilization among the servers. To do so, we set the capacity of the edge joining each server j to the sink as $\lambda capacity(j)$, as shown in Figure 7.6. We now want to minimize λ such that there exists a max flow saturating all of the edges leaving the source. This turns out to be a well studied problem called the *parametric max-flow problem*. Interesting note: algorithms exist which can solve this type of parametric max flow problem in essentially the same amount of time as a standard max flow problem (this is a result of Gallo, Grigoriadis, and Tarjan).

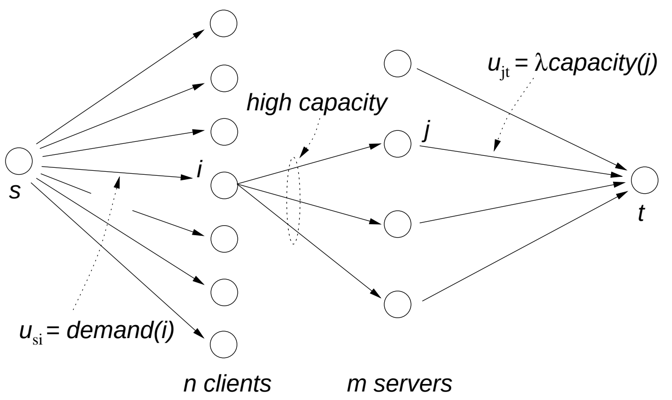


Figure 7.6. Minimizing Max Utilization.

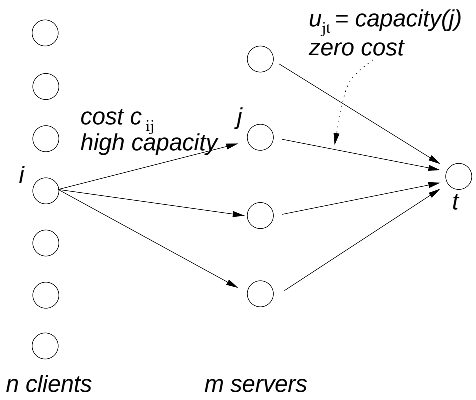


Figure 7.7. Adding Mapping Costs.

7.2.3 Adding a Cost to the Mapping Between Clients and Servers

Next, we add a cost c_{ij} for each connection between the client i and the server j as shown in Figure 7.7 and attempt to find a minimal-cost assignment of the demands. The problem now becomes a min-cost flow problem:

$$\begin{aligned} &\text{Minimize} && \sum_{(i,j)\in E} c_{ij}x_{ij} \\ \forall \text{ nodes } i \in V: &&& \textstyle\sum_j x_{ij} - \sum_j x_{ji} = b_i \\ \forall \text{ edges } (i,j) \in E: &&& 0 \le x_{ij} \le u_{ij} \end{aligned}$$

We assign to each client i, $b_i = demand(i)$, to each server j, $b_j = 0$, and to t, $b_t = -\sum_i demand(i)$. The min-cost flow problem is also well-studied and efficiently solvable.

7.2.4 Replacing Capacity with a Cost Function

All of the previous formulations adhere to our first model of utilization cost (Figure 7.2) in which utilization of a server is "free" up until the server's load reaches some hard capacity limit. We now consider the second model of utilization cost, in which we replace the capacities, $capacity(j)$, with load-dependent cost functions as shown in Figure 7.8. This changes only the objective function of our min-cost flow problem:

$$\begin{aligned} &\text{Minimize} && \sum_{(i,j)\in E} c_{ij}(x_{ij}) \\ \forall \text{ nodes } i \in V: &&& \textstyle\sum_j x_{ij} - \sum_j x_{ji} = b_i \\ \forall \text{ edges } (i,j) \in E: &&& 0 \le x_{ij} \le u_{ij} \end{aligned}$$

The server-sink edges may continue to have upper capacities u_{jt}, although this is no longer necessary as our cost functions can express an upper capacity by jumping to infinity when load reaches a certain level. Also, by adjusting the costs associated with the server-sink edges versus the costs associated with the client-server edges, we can select which objective we wish to emphasize more: mapping with good locality or achieving low utilization.

Assuming the cost functions are all convex (as is typically the case), the entire objective function will be convex, so we are minimizing a convex function over a convex polyhedral set of feasible solutions; such a problem still lies within the realm of "nice" optimization problems. Moreover, if we approximate the convex cost functions as piecewise linear functions, we can reduce this problem back to the simpler previous (linear) min-cost flow problem. To do so, we replace each server-sink link with several parallel links, each representing one linear piece of the piecewise linear cost function as shown in Figure 7.9.

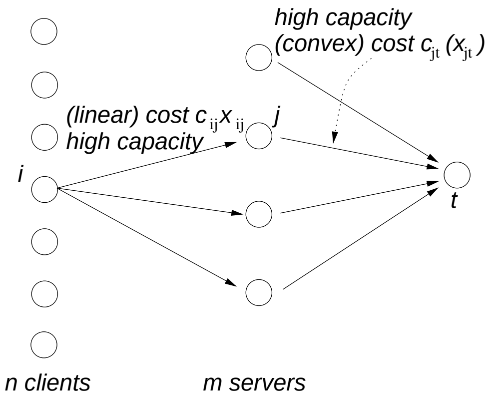


Figure 7.8. Replacing capacities with costs.

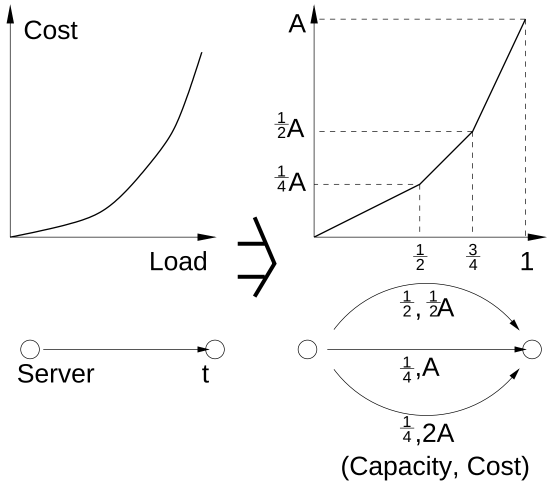


Figure 7.9. Piecewise linear approximation.

There are no known good techniques for solving such problems with non-convex cost functions; moreover, it was shown in a previous lecture that such problems with concave cost functions are NP-Hard and also very difficult to approximate.

7.3 Advanced Topics for Network Flows

We now proceed to discuss some interesting research topics related to the above network flow problems.

7.3.1 Unsplittable Flows

For the unsplittable flow problem, we assume for a given instance of our basic assignment problem with costs (Figure 7.7) that a feasible fractional assignment exists. The question that we would like to ask is whether an assignment exists for which the demand for each client cannot be split between servers and is hence satisfied by only one server. The approach that we will take to solve this problem is to take the fractional solution and "round" it to give us an unsplittable solution with no worse cost, allowing the capacity constraints to be slightly violated in the process (we shall refer to the excess capacity as "congestion"). Congestion may be defined two different ways: (i) the additive amount we increase capacity, or (ii) the multiplicative factor by which we must scale capacity. The problem then becomes that of find the unsplit assignment with minimum congestion, i.e. that minimizes the maximum congestion experienced over all edges, assuming that a feasible fractional assignment initially exists.

There is a lower bound for this problem which states that in the worst case, one experiences congestion of at least $d_{max} = \max_i demand(i)$. This is 'additive' congestion, as opposed to 'multiplicative' congestion. The example given in Figure 7.10 satisfies this bound. In this example, we've reversed the location of the clients and servers, as is often done with unsplitable flow problems, since we now want to route flow on disjoint paths from the source (which was formerly the sink) to each of the clients. Here, the total demand is equal to total capacity, and a feasible fractional assignment exists. However, because the demand of each client is $\frac{m}{m+1}$ but each server has only capacity 1, each server can satisfy only one client integrally without experiencing congestion. Since there is one client more than the number of servers, one of the servers will have to service 2 clients in an unsplit solution, and the edge linking the source to that server will be congested by an additive amount which gets arbitrarily close to d_{max} as m gets large.

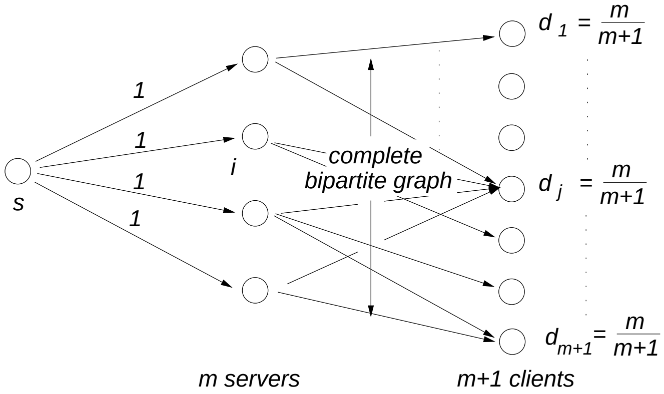


Figure 7.10. Example which satisfies lower bound.

It can be shown that in the bipartite case, there exist simple polynomial-time approximation algorithms which achieve an additive congestion of d_{max}. It is an open question as to when we can meet this bound without the bipartite constraint.

7.3.2 Vector (Multi-resource) Load Balancing

The vector load balancing problem is formulated by generalizing the demand and capacity functions from scalars to vectors. An example is given in Figure 7.11. Clearly a problem that allows for fractional assignment can be solved as a linear program by adding more constraints. The interesting question is thus whether there is a feasible integral assignment for a given problem. We know that such a problem is NP-Complete, since it is NP-Complete for scalars (it's the unsplittable flow problem). It might be interesting to know if the problem can be solved more efficiently than with linear programming (pointed out by Bobby). It might also be interesting to see if any results for the unsplittable flow problem generalize to the vector problem.

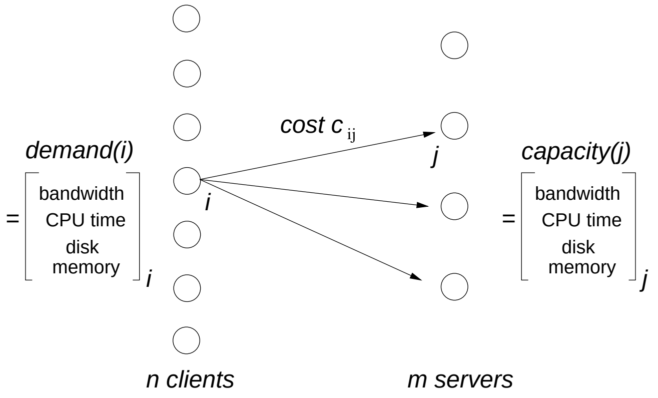


Figure 7.11. An example of a vector load balancing problem

7.3.3 Unknown Demand

Oftentimes, we know the number of requests made by a client, but we do not know beforehand the actual load which will be imposed on the servers by the requests. Thus, we may be interested to figure out how we can do load assignment without full knowledge of the demand.

One simple approach is to derive a scaling factor based on the total system-wide load and requests measured in some historical interval of time: $s = \frac{\text{total load}}{\text{total \# of requests}}$. We then estimate the demand for each client based on the number of requests on the assumption that the ratio s is a constant across all clients. Although this approach seems to provide us with a reasonable approximation, the assumption that the ratio s is a constant across all clients is questionable. In particular, if we consider the aggregation of clients by geographic location, then due to different time zones, it is clear that the nature of the activities of different clients would be different for different time zone (i.e. clients in areas where it is nighttime may have slow dialup access while those in areas where it is daytime may be using high speed Internet access in their offices). The challenge is thus to come up with a scheme that can handle unknown demand without assuming that s is system-wide constant.

The mapping through DNS is primarily responsible for this problem, since the client requests can be identified with IP address of the client's DNS server, but the load induced on servers is identified with the actual IP address of the client (not the client's DNS server).

7.3.4 Dynamic Stability

Clearly, the demands of clients are not constant, but they change in time. Hence, we want an algorithm that will not make changes in assignment if there are only small changes in the demand. We would like to have persistent mappings because this is required in caching and state preservation. For example, it would be very annoying for a user if his shopping cart gets lost frequently because his browser is directed to another server. This is also to make the system immune to dynamic instability and oscillations. One simple approach for this problem is to slightly increase the cost of adjacent links once a stable solution is found for a given link.

7.4 Stable Marriage Techniques

In the second half of this talk, we discuss methods for formulating and solving load balancing problems using “stable marriage” techniques. In the basic stable marriage problem, we have n men and n women, which we can model as nodes in a bipartite graph. Associated with each man is a preference list of all the women and associated with each woman is a preference list of all the men. We then compute a “marriage” between the men and women, which is nothing more than a matching in our bipartite graph.

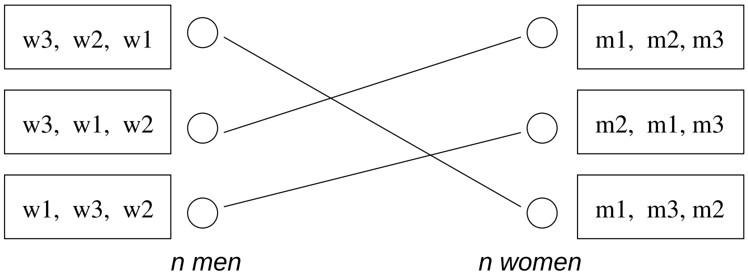


Figure 7.12. A stable marriage.

Given some marriage, we call a man-woman couple (m, w) a *rogue couple* if they are not paired together but both of them prefer each-other to their currently-assigned mates. A marriage is said to be *stable* if it has no rogue couples, and *unstable* otherwise.

7.4.1 Gale-Shapley Algorithm

Stable marriages are typically computed using the well-known *Gale-Shapley* algorithm. The algorithm is actually quite simple: each man proceeds through his preference list, starting from his preferred mate, and he proposes to each of the women on the list in sequence until he is accepted. Each woman simply waits and accepts the best proposal issued to her so far. If she is currently engaged to a man m and then receives a proposal for a more favorable man, she will drop m and accept the new proposal, in which case m will simply continue down his preference list until he is again accepted. The algorithm is actually completely symmetric, in that either the men or the women may do the proposing, but for simplicity we’ll assume here that the men do the proposing. More precisely, the algorithm repeats the following operations until all men and women are paired:

- Find an un-paired man m
- A proposal is issued by m to the next candidate on his list, w
- If m is more preferable to w than her currently-assigned mate (if any), then she accepts m and drops her currently-assigned mate (if any).

This algorithm runs in time $O(n^2)$ in the worst case, since we consider at most once each entry in each man’s preference list, and spend $O(1)$ time per entry.

Theorem 7.1. *At the end of the Gale-Shapley algorithm, everyone is engaged.*

Proof: Note that the number of engaged men at the end will be equal to the number of engaged women at the end. It could not be the case that there remains an un-paired couple (m, w), since at some point m would have proposed to w, and she would have accepted. □

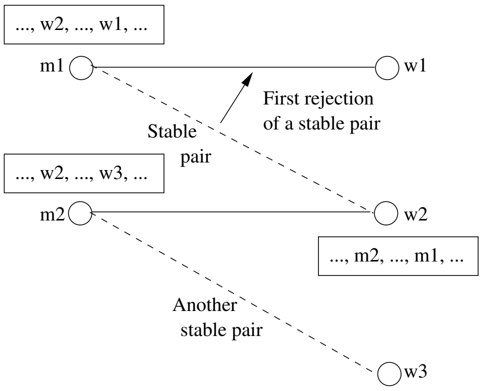
Theorem 7.2. *The solution obtained by the Gale-Shapley Algorithm is a stable matching.*

Proof: Assume that the final matching is not stable, i.e. that there exists parings (m_1, w_1) and (m_2, w_2) but (m_1, w_2) is a rogue couple. Since m_1 prefers w_2 to w_1, m_1 would have proposed to w_2 first, who must have rejected him, so w_2's current mate must be preferred to m_1. This is a contradiction, since m_1 is higher than m_2 in w_2's preference list. Hence, the matching must be stable. □

A *stable pair* (m, w) is a pair which is matched in some stable marriage. If (m, w) is a stable pair, we say that w is a *stable partner* of m.

Theorem 7.3. *The outcome of the Gale-Shapley Algorithm is a "man-optimal" assignment. That is, each man is paired with his most-preferred stable partner.*

Proof: Suppose not. Consider some execution order and consider the first point in time when a stable pair, (m_1, w_2), is rejected.



This rejection can either occur if w_2 is already engaged to a preferred man (m_2), or if m_2 comes along and proposes to w_2 while she's engaged to m_1. Since we're presuming that (m_1, w_2) is a stable pair, there is some stable matching M in which m_1 and w_2 are paired together – let w_3 be the woman with whom m_2 is paired in M, meaning that (m_2, w_3) is also a stable pair. It cannot be the case that w_3 is higher than w_2 on m_2's preference list, as that would imply that m_2 would have already proposed to w_3 and been rejected, and we are assuming that m_1's rejection by w_2 is the *first* rejection of a stable partner so far. Thus w_2 appears before w_3 on m_2's preference list. This means that (m_2, w_2) is a rogue couple in M, which contradicts the stability of M. □

The preceding theorem tells us that remarkably, the order of the proposals during the algorithm is irrelevant. The algorithm always arrives at the same stable matching, even though there is not necessarily a unique stable matching.

Using a similar line of reasoning, we can show that the matching computed by the Gale-Shapley algorithm is also "woman-pessimal", in the every woman ends up assigned to her least-preferred stable partner. Remembering the symmetry of the problem, however, one should note that if the women were to do the proposing, they would receive an optimal assigment while the men would receive a pessimal assignment. Some researchers have attempted to address the question of computing stable marriages which are fair to both parties.

7.4.2 Truncating the Preference Lists

In order to model the load balancing problem using stable marriages, we need to consider a few extensions to the basic stable marriage problem above. First, we consider a class of problems where we truncate the preference lists of the men and/or, since for large load balancing problems, we shall see that maintaining complete preference lists may lead to impractical storage requirements. For the purposes of determining whom to reject, women can consider the men not on their preference lists as being arbitrarily-ordered at the bottom of their preference lists. We now redefine (m, w) as a rogue couple if:

1. Both m and w are on each other's preference lists,
2. m is either unassigned or prefers w to his current mate, and
3. w is either unassigned or prefers m to her current mate

The Gale-Shapley algorithm runs essentially the same on problem instances with truncated preference lists. The outcome will be a stable assignment (i.e. there will be no rogue couples, as defined above), although it is possible that some men and women may end up unassigned. The following theorem then holds.

Theorem 7.4. *Applying the Gale-Shapley Algorithm with truncated preference lists still gives a stable assignment. Furthermore, each man who ends up assigned (unassigned) will be assigned (unassigned) in* every *stable assignment, and each assigned (unassigned) woman will be end up being assigned (unassigned) in* every *stable assignment.*

Proof of this theorem and the remaining theorems is omitted. For a complete treatment of these topics, refer to the book "The Stable Marriage Problem" by Gusfield and Irving.

7.4.3 Different Numbers of Men and Women

Next, we can consider a class of problems where the numbers of men and women are not the same but the men and women all have complete preference lists. The Gale-Shapley algorithm still works in this setting, and still computes a stable matching with some nice properties.

Theorem 7.5. *If there are fewer men than women, and everyone has a complete preference list, the Gale-Shapley algorithm computes a stable matching which is still "man-optimal" and "woman-possible". Moreover, the men will all be assigned, and the women will be partitioned into 2 sets: (i) those who are always assigned in every stable matching, and (ii) those who are never assigned in any stable matching. If there are more men than women, then the women will all be assigned and the men will undergo a partition into 2 sets.*

Theorem 7.6. *In the event that there are different numbers of men and women and that preference lists are incomplete, there will be a partition into 2 sets induced in both the men and in the women. Members of one set will be assigned in every stable matching, and members in the other set will never be assigned in any stable matching.*

7.4.4 Many-to-One Assignments: Adding Capacities

To avoid any awkward situations going forward, we will alter our discussion to talk about assigning medical students to hospitals (this was actually one of the original applications of stable marriage techniques). The difference here is that hospitals can accept more than one medical student; to each hospital H_j we assign an integral capacity C_j.

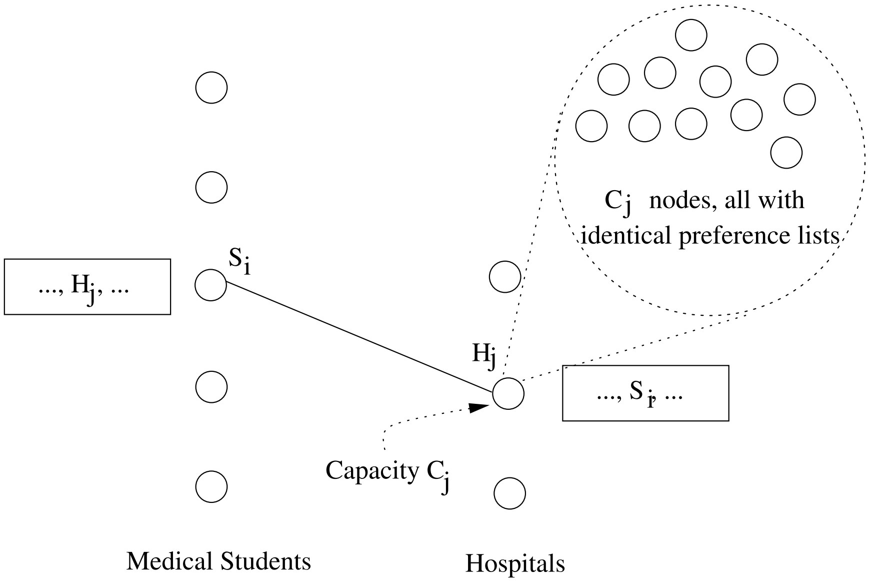


Figure 7.13. Assigning medical students to hospitals (with capacities).

This generalization of the stable marriage problem is easy to accomodate if we simply split every hospital node H_j into C_j nodes with unit capacity, all with equivalent preference lists. This transforms the problem into an uncapacitated stable marriage problem, which we can solve using the Gale-Shapely algorithm as described above.

One should note that it is not necessary to explicitly perform this transformation. All we need to do is maintain, for each hospital, a list of medical students currently assigned to it. If the hospital ever receives a proposal when it has already reached capacity, it simply rejects the least-preferred of these students (including possibly the proposing student). If the algorithm stores these lists of assigned students in heaps, then we will only spend $O(\log n)$ time per proposal (here, n is the number of students), for a total running time which is $O(L \log n)$, where L is the sum of the lengths of all of the medical students' preference lists (since L is an upper bound on the total number of proposals issued).

7.4.5 Dealing with Demands and Capacities

We now take the final steps necessary to model our load balancing problem. Consider the problem of assigning clients to servers, where each server has an integral capacity and each client has an integral demand. The clients rank the servers according to locality, and the servers must choose some method of ranking the clients – one potential way to do this is to have a server S prefer client A to client B if client B will suffer less from a rejection (i.e., client B may have other choices in its preference list which are nearly as good as S). We reduce the problem to an uncapacitated problem by splitting each client with demand D into D unit-sized nodes with equal preference lists, and by (implicitly) splitting each server with capacity C into C unit-sized nodes with equal preference lists. To avoid utilizing excessive space, we typically truncate each client's preference list to contain only its first few preferred servers. We may not need to explicitly generate and store preference lists for the servers at all, for example if we base these on the client preference lists as discussed above.

The Gale-Shapley Algorithm gives us a stable assignment, but it may be "fractional" in the sense that a client may have some units of its demand assigned to different servers. It is also possible that if there is not enough capacity, there may not be a matching which assigns all of the demand. However, due to the stable marriage properties previously discussed, we at least know that if a client has only an α fraction of its demand satisfied by our stable marriage solution, then it will have exactly an α fraction of its demand satisfied in *every* stable matching. Similarly, we utilize the same amount of capacity in each server in every stable marriage solution.

7.4.6 Improving the Running Time

Recall that when we added capacities to the servers, we could implicitly blow up server nodes into clouds of unit nodes without affecting running time much. This is not quite possible with the client nodes, however, so the running time of our algorithm will now be roughly linear in the total demand over all clients, which could be quite inefficient.

To reduce this running time, we make requests in batches. That is, an aggregate client node will propose to a server with a block of several units of demand all at once. If the server can accept all of these, then we continue to treat the client node as an aggregate quantity. If the server accepts only part of the request, then we split the client node into two smaller aggregate nodes, one of which is assigned to the server and the other of which continues to issue aggregate proposals.

The issuance of aggregate proposals by itself does not improve our worst-case running time, so we use the following trick: a client node will only issue an aggregate request if at least some ϵ fraction of its demand remains unassigned. If this is the case, there will be $O(\frac{L}{\epsilon})$ total requests, and one may conclude after a bit of analysis that this yields a total running time of $O(\frac{L}{\epsilon}\log n)$. The algorithm may terminate with some demand unassigned, but at most an ϵ fraction of every client's demand will remain unassigned, which is perhaps a reasonable trade-off for a running time which is nearly linear in the sizes of the clients' preference lists.