Finite Termination of "Augmenting Path" Algorithms in the Presence of Irrational Problem Data

Brian C. Dean[*] Michel X. Goemans[†] Nicole Immorlica[‡]

June 28, 2006

Abstract

This paper considers two similar graph algorithms that work by repeatedly increasing "flow" along "augmenting paths": the Ford-Fulkerson algorithm for the maximum flow problem and the Gale-Shapley algorithm for the stable allocation problem (a many-to-many generalization of the stable matching problem). Both algorithms clearly terminate when given integral input data. For real-valued input data, it was previously known that the Ford-Fulkerson algorithm runs in polynomial time if augmenting paths are chosen via breadth-first search, but that the algorithm might fail to terminate if augmenting paths are chosen in an arbitrary fashion. However, the performance of the Gale-Shapley algorithm on real-valued data was unresolved. Our main result shows that, in contrast to the Ford-Fulkerson algorithm, the Gale-Shapley algorithm always terminates in finite time on real-valued data. Although the Gale-Shapley algorithm may take exponential time in the worst case, it is a popular algorithm in practice due to its simplicity and the fact that it often runs very quickly (even in sublinear time) for many inputs encountered in practice. We also study the Ford-Fulkerson algorithm when augmenting paths are chosen via depth-first search, a common implementation in practice. We prove that, like breadth-first search, depth-first search also leads to finite termination (although not necessarily in polynomial time).

1 Introduction

The Ford-Fulkerson (FF) algorithm for the s-t maximum flow problem [2] is remarkably simple to describe and implement: as long as we can find an "augmenting path" along which additional flow can be sent from a source node s to a sink node t, send as much flow along the path as possible.

A close relative of the FF algorithm is the Gale-Shapley (GS) algorithm [3] adapted for the stable allocation problem, a many-to-many generalization of the stable matching problem. In this problem, we are assigning, say, a set of jobs to machines, where the jobs have varying processing times and the machines have varying capacities. Each job submits a ranked preference list over machines on which it may be processed, and each machine submits a ranked preference list over jobs that it may process. The stable allocation problem involves finding a feasible (fractional) assignment of jobs to machines in which no job/machine pair has an incentive to deviate from the assigned

[*]Department of Computer Science, Clemson University

[†]Department of Mathematics, M.I.T.

[‡]Microsoft Research

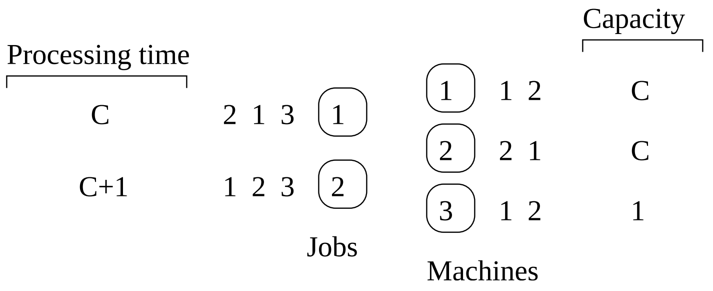

Figure 1: An example instance on which the GS algorithm requires exponential time. Regardless of the proposal order of the jobs, the algorithm will perform at least C proposals before converging to the unique stable assignment.

solution in a sense to be described later. The GS algorithm for the stable allocation problem is a generalization of the original GS algorithm for the simpler stable matching problem (where we are assigning n unit-sized jobs to n unit-sized machines): each job proceeds down its preference list issuing "proposals" to machines, and each machine tentatively accepts the best proposal received thus far. If a machine receives a proposal from a job it prefers more than its current tentative assignment, it accepts the proposal and rejects the job to which it is currently assigned, and this job which then continues issuing proposals to machines further down its preference list. In the stable allocation problem where jobs and machines have non-unit sizes, the GS algorithm operates in an identical fashion, except proposals and rejections now happen in non-unit quantities. When a job proposes to a machine, it proposes all of its unassigned load, and a machine may chose to "fractionally" accept only part of this load and reject the rest, depending on its preference among its current assignments. A sequence of proposals and rejections can be interpreted as an augmenting path.

When edge capacities are integral in the FF algorithm, or when processing times and capacities are integral in the GS algorithm, each augmentation pushes at least 1 unit of flow and so the algorithms clearly terminate. However, in the FF algorithm, for certain graphs with real-valued edge capacities, if we choose augmenting paths in a completely arbitrary fashion then we may fail not only to terminate, but also to converge to an optimum flow [2, 5]. The main contribution of this paper is to show that, in contrast to the FF algorithm, the GS algorithm always terminates in finite time for real-valued inputs. This resolves an open question of Baiou and Balinksi [1].

Unfortunately, convergence of the GS algorithm may take exponential time in the worst case (Figure 1), whereas an alternative algorithm has been proposed by Baiou and Balinksi [1] that runs in strongly polynomial time. However, the GS algorithm is perhaps more common in practice due to its simplicity and the fact that for many instances in practice, its running time can be significantly faster than that of the Baiou-Balinski algorithm (in fact, the GS algorithm often runs in sublinear time). It is therefore reassuring to know that termination of the GS algorithm is guaranteed for real-valued inputs.

We also study the FF algorithm when augmenting paths are chosen according to a depth-first search (DFS). While it is well-known that the selection of augmenting paths using breadth-first search (BFS) always leads to finite termination as well as a strongly polynomial running time, the use of DFS is another regrettably common approach for selecting augmenting paths. Even with integral capacities, the use of DFS is to be discouraged since it can lead to exponential worst-case running times (Figure 2); however, our techniques allow us to give a simple proof that at the very

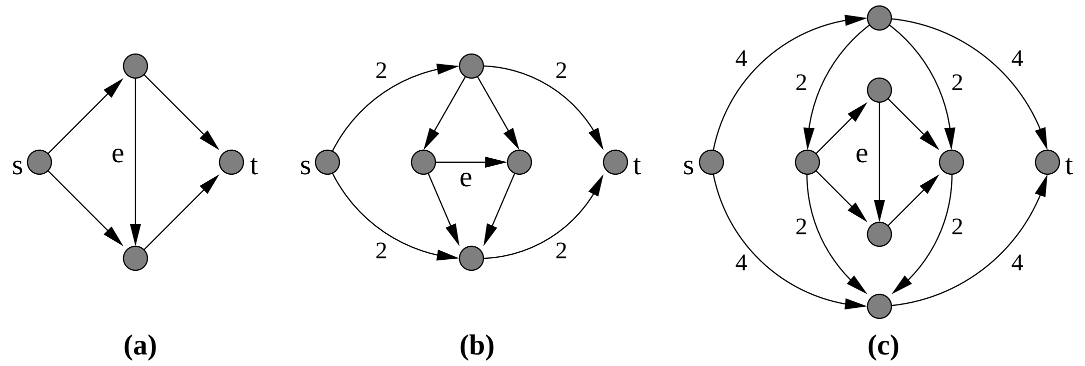

Figure 2: Examples of graphs for which the FF algorithm takes exponential time to run, using DFS to locate augmenting paths. Unlabeled edges have unit capacity. In (a), a DFS that prioritizes edge e will lead to two augmenting paths that each utilize e (in alternating directions). In (b), we recursively expand the graph by replacing e with a copy of the entire graph, leading to four augmenting paths (each of which uses e in alternating directions). The graph in (c) requires eight augmenting paths, and so on. Although this example is not bipartite, one can construct bipartite graphs that behave similarly.

least, the FF algorithm implemented with DFS always terminates even when edge capacities are real-valued. Our results for the FF algorithm are primarily of theoretical interest, since there is little reason to use DFS rather than BFS in practice to locate augmenting paths.

The structure of this paper is as follows. The next section contains our proof of finite termination for the FF algorithm using DFS to locate augmenting paths. Following that, we give a more detailed introduction to both the stable allocation problem and the GS algorithm, and build towards our main result that the GS algorithm always terminates in finite time.

2 The FF Algorithm with Irrational Capacities

Consider the use of DFS to locate augmenting paths for the FF algorithm. In this case, every time we visit a node i during an augmenting path search, we recursively search the outgoing edges from i according to some deterministic ordering, and we keep using the same outgoing edge as long as this enables us to find valid augmenting paths to t.

Observation 1. *An augmenting path will never include an edge directed into the source node.*

Theorem 1. *The Ford-Fulkerson algorithm terminates in finite time, even with real-valued edge capacities, if we use DFS to find augmenting paths.*

Proof. Starting from the source node s at the beginning of the algorithm, let i be the first neighboring node chosen by the algorithm to visit. All of our augmenting paths will continue to utilize the edge (s, i) until either (i) the flow along (s, i) reaches the capacity of (s, i), or (ii) no residual augmenting $i \rightsquigarrow t$ path exists that does not include s. We first argue that (i) or (ii) will occur within a finite amount of time. This follows by induction on the number of edges in our problem instance, since as long as the FF algorithm is choosing augmenting paths starting with (s, i), it is essentially performing a recursive maximum flow computation from node i to node t, in which augmenting paths through s are disallowed, and which will be terminated prematurely if it manages to accumulate a total amount of $i \rightsquigarrow t$ flow equal to the capacity of (s, i). Moreover, this recursive max flow computation is taking place on a graph that is smaller by one node, since its augmenting paths never contain s. By induction on the number of nodes in our instance, the recursive max flow computation therefore terminates in finite time.

Suppose now that our recursive max flow computation terminates due to condition (i). In this case, we will never use (s, i) again in any augmenting path, since to do so we would need to augment along the reverse residual edge (i, s), and an augmenting path will never utilize such an edge directed into the source. We can therefore effectively ignore (s, i) and (i, s) henceforth, and this gives us a smaller problem instance in which (by induction) the remainder of the FF algorithm will terminate in finite time. On the other hand, suppose condition (ii) occurs. In this case, we claim that i cannot appear on any future augmenting path, so again we can reduce the size of our instance by one node and claim by induction that the rest of the FF algorithm must terminate finitely. The fact that i cannot be part of any future augmenting path is argued as follows: at the point in time when condition (ii) occurs, let S_i be the set of all nodes that are reachable from i via a residual augmenting path that does not include s. All edges leaving S_i must be saturated except those directed into s (and those edges will never be part of any augmenting path). Therefore, in order for any node in S_i to ever again have an augmenting path to t (that doesn't use s), we would first need to augment along a path that enters S_i (to unsaturate one of the outgoing edges); however, such a path could never leave S_i. $\square$

3 The GS Algorithm with Irrational Data

In this section, we study the performance of the GS algorithm for the stable allocation problem in the presence of real-valued data.

3.1 The Stable Allocation Problem

The stable allocation problem is a many-to-many extension of the well-studied classical stable matching problem. Let $[n] := \{1, 2, \ldots, n\}$ denote a set of n jobs and $[m]$ denote a set of m machines (we will use scheduling terminology and speak of assigning "jobs" to "machines" since the traditional use of "men" and "women" for stable matching problems becomes somewhat awkward once we generalize to the many-to-many case). Job i requires p_i units of processing time, machine j has a capacity of c_j units, and at most $u_{ij} \leq \min(p_i, c_j)$ units of job i can be assigned to machine j. A fractional assignment $x \in \mathbf{R}^{m \times n}$ between jobs and machines is *feasible* if it satisfies

$$\begin{aligned} \sum_{j \in [m]} x_{ij} \leq p_i & \qquad \forall i \in [n] \\ \sum_{i \in [n]} x_{ij} \leq c_j & \qquad \forall j \in [m] \\ 0 \leq x_{ij} \leq u_{ij} & \qquad \forall (i, j) \in [n] \times [m]. \end{aligned} \tag{1}$$

Just as in a stable matching problem, each job i submits a ranked preference list over all machines, and each machine j submits a ranked preference list over all jobs. Our goal is to compute a feasible assignment that is *stable* with respect to these preference lists, defined as follows.

Definition 1. *Job i and machine j form a* blocking pair *in an assignment x if $x_{ij} < u_{ij}$ and both i and j are partially assigned to partners they prefer less than each-other (in other words, both i and j would be "happier" if x_{ij} were increased).*

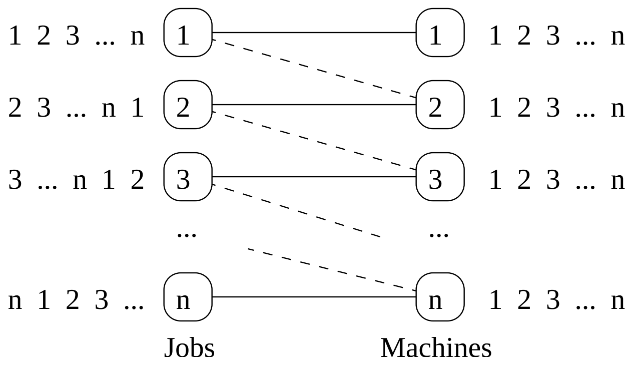

Figure 3: A simple stable allocation instance where the GS algorithm runs much faster than the BB algorithm. Each job has unit processing time and each machine has unit capacity (so this is really an instance of the stable matching problem). Solid lines indicate the unique job-optimal stable assignment. Dashed lines indicate the unique job-optimal stable assignment when machine 1 is removed (job n ends up unassigned).

Definition 2. *A job i is* saturated *if $\sum_j x_{ij} \geq p_i$. A machine j is* saturated *if $\sum_i x_{ij} \geq c_j$.*

Definition 3. *In an assignment x, we say job i is* popular *if there exists a machine j such that $x_{ij} < u_{ij}$ and j prefers i to one of its current assignments. Similarly, machine j is popular if there exists some job i such that $x_{ij} < u_{ij}$ and i prefers j to one of its current assignments.*

Definition 4. *An assignment x is* stable *if (i) it admits no blocking pairs, and (ii) all popular jobs and machines are saturated.*

A feasible stable assignment x is said to be *job-optimal* (*machine-optimal*) if every job (machine) prefers x to any other feasible stable assignment x'.

As a final assumption, we assume that $\sum_i p_i \leq \sum_j c_j$. This comes without loss of generality by introducing a "dummy" machine of large capacity that is ranked last by every job.

3.2 A Stable Allocation Algorithm

Job-optimal and machine-optimal stable assignments always exist, and one can compute either one of them using a strongly-polynomial algorithm of Baiou and Balinski [1]. The Baiou-Balinski (BB) algorithm (say, applied to computing a job-optimal solution) operates by introducing one machine at a time. After each machine is introduced the current assignment is appropriately modified so as to remain job-optimal with respect to the current set of machines. Unfortunately, this might involve a significant amount of "thrashing" in some instances, since the introduction of each new machine can result in significant, or even wholesale change to the current assignment. An example is shown in Figure 3.

An alternative to the BB algorithm is the Gale-Shapley (GS) algorithm, familiar to anyone who has studied stable matchings in the past. The GS algorithm operates as follows. Each job i maintains a sorted list L_i of machines, initially equal to the preference list. Similarly, each machine j maintains an initially empty list R_j of the jobs currently assigned to j. Again, the list is sorted in order of the preference list. The algorithm iteratively performs the following steps until all jobs are saturated.

1. Select an unsaturated job i.

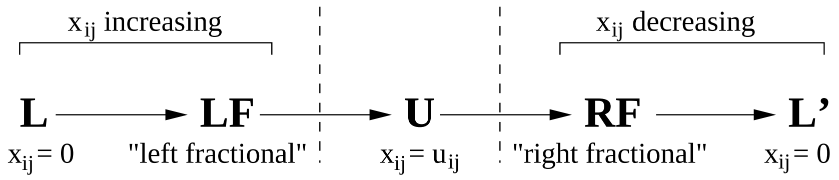

Figure 4: The sequence of labelings through which an edge progresses.

2. Let j be the first machine on list L_i.
3. Set $x_{ij} = \min(p_i - \sum_{j'} x_{ij'}, u_{ij})$ and add job i to list R_j of machine j making sure to maintain the sorted order. If $x_{ij} = u_{ij}$, then remove j from L_i.
4. While $\sum_{i'} x_{i'j} > c_j$, reject $p = \min(x_{i''j}, c_j - \sum_{i'} x_{i'j})$ processing time from the last job i'' in R_j (i.e., decrement $x_{i''j}$ by p). For any such rejected job i'', remove j from $L_{i''}$.

The GS algorithm can be viewed as an "augmenting path" algorithm just like the FF algorithm. In fact, when we apply the FF algorithm to a bipartite assignment problem, each augmenting path alternates back and forth between the left-hand and right-hand side of our graph, and we can interpret the meaning of such a path as a sequence of "proposals" and "rejections".

In many instances in practice, one expects the GS algorithm to run no slower and potentially much faster than the BB algorithm. For example, if our capacity constraints are reasonably loose such that most jobs receive their first choices (a very common occurrence in practice), the GS algorithm terminates almost immediately, in time sublinear in the input size. The BB algorithm, on the other hand, might take substantially longer due to the need to continually readjust its solution as new machines are introduced. Figure 3 is a good example — the GS algorithm makes one proposal for each job, running in only $O(n)$ total time, and the BB algorithm reassigns each job at least $n-1$ times when the machines are added in the order $n, n-1, \ldots, 1$, for a total running time of $\Omega(n^2)$. While in the GS algorithm each job moves down its preference list proposing first to its must-preferred partner, the BB algorithm moves the opposite direction, potentially assigning a job to many machines as it moves up the list towards the most preferred stable partner. If capacity constraints are fairly loose and most jobs end up with highly-preferred partners, the GS algorithm tends to spend much less work.

3.3 Termination of the Gale-Shapley Algorithm

A technique for proving termination of graph-based algorithms is as follows. First, define a finite set of labels for edges and an ordering on this set of labels. Next, prove that during the course of the algorithm, the labeling of an edge can only advance according to this ordering and that each advancement is guaranteed to occur after a finite amount of time. In the rest of this section, we use this technique to prove termination of the GS algorithm.

3.3.1 Edge Labelings

After every iteration of the GS algorithm, we label the edges of our bipartite assignment graph as follows:

- The set L contains all edges (i, j) at their lower capacities ($x_{ij} = 0$) along which a proposal has never been issued. If $(i, j) \in L$, then job i has yet to reach all the way down to j on its preference list.

- The set LF contains all *left fractional* edges. An edge (i, j) is left fractional if $0 < x_{ij} < u_{ij}$ and if j has never issued a rejection to i. Among its two endpoints i and j, the left endpoint i was therefore the "responsible party" that prevented x_{ij} from growing any larger (i.e., j would have happily accepted more load, but i was reluctant so far to offer it).

- The set U contains all edges (i, j) at their upper capacities ($x_{ij} = u_{ij}$).

- The set RF contains all *right fractional* edges. An edge (i, j) is right fractional if $0 < x_{ij} < u_{ij}$ and if j has rejected i in the past. Among its two endpoints i and j, the right endpoint j is the "responsible party" in this case for having prevented x_{ij} from growing any larger (i might want to send more load to j, but j refuses).

- The set L' contains all edges (i, j) at their lower capacities ($x_{ij} = 0$) along which a proposal has been issued in the past. If $(i, j) \in L'$, then job i has already exhausted its proposals to j and moved on down its preference list. Any assignment from i to j has since been completely rejected, and x_{ij} will never become positive again.

We call $F = LF \cup RF$ the set of all *fractional* edges, since their assignments lie strictly between their lower and upper bounds.

As we see in Figure 4, during the course of the GS algorithm an edge will progress monotonically through the labelings in the order L, LF, U, RF, and L'. It is possible that an edge skips over some of these labels — for example if an edge $(i, j) \in U$ is subject to a massive rejection by j it may become labeled L'. The monotonicity of this progression is significant. If we look at an edge (i, j) as the GS algorithm executes, the value of x_{ij} may increase as (i, j) moves from L to LF to U and then it may decrease if (i, j) further proceeds to RF and L'. By way of contrast, the value of x_{ij} can fluctuate up and down many times in the case of the FF algorithm for a "flow based" assignment problem.

Any time an edge changes its label, we say the edge is *promoted*. If we can prove that an iteration of the GS algorithm promotes an edge, then this indicates significant progress towards termination, since each edge can be promoted at most 4 times.

Observation 2. *If $(i, j) \in L$ and i proposes to j, then (i, j) will be promoted.*

Observation 3. *If $(i, j) \in L \cup LF \cup U$ and j issues a rejection to i, then (i, j) will be promoted.*

3.3.2 Properties of Edge Labelings

The structure of the GS algorithm becomes somewhat clearer when we look at properties of edge labelings that hold after each of its iterations.

Observation 4. *For every job i, $\delta_{LF}(i) \leq 1$. That is, i can have at most one outgoing left fractional edge (we use $\delta_S(i)$ to denote the degree of node i restricted to the subset S of edges).*

Recall in the GS algorithm, each job i maintains a pointer to the "current" machine j to which it is issuing proposals. Once j starts to reject i or x_{ij} reaches u_{ij}, this pointer advances down

i's preference list. Therefore, among all the machines to which i has ever proposed, there is only at most one that may still be accepting proposals from i, and this corresponds to the LF edge emanating from i. We will have $(i, j) \in U \cup RF \cup L'$ for all other machines j to which i has proposed in the past.

Observation 5. *For every machine j, $\delta_{RF}(j) \leq 1$. That is, j can have at most one incoming right fractional edge.*

To see this, suppose j somehow had two incoming edges $(i, j) \in RF$ and $(i', j) \in RF$ where j prefers i to i'. This contradicts the behavior of the GS algorithm, since j would never have rejected i when it had the opportunity to reject i' instead.

Lemma 1. *Let G' be any connected subgraph of our assignment graph consisting only of fractional edges (LF and RF edges). Among the fractional edges in G' there can be at most one cycle.*

Proof. Let A denote the nodes of G' on the left hand side (the jobs) and B the nodes of G' on the right hand side (the machines). If $n = |A| + |B|$ denotes the total number of nodes in G' and m is the number of fractional edges, then $m = \sum_{i \in A} \delta_{LF}(i) + \sum_{j \in B} \delta_{RF}(j) \leq |A| + |B| = n$. So either $m = n - 1$, in which case the fractional edges in G' form a tree, or $m = n$, in which case the fractional edges in G' form a tree plus one additional edge that creates a unique cycle. $\square$

Observation 6. *Once a machine becomes saturated, it stays saturated forever. (The same is not true for jobs)*

Observation 7. *If $(i, j) \in RF \cup L'$, then machine j is saturated.*

Observation 8. *Let C be a cycle of fractional edges after some non-terminal iteration of the GS algorithm. Then (i) the edges along C alternate between being labeled LF and RF, and (ii) C cannot contain every node in our bipartite assignment graph.*

The alternating labels are a direct consequence of observations 4 and 5. As a result of each machine j in C having an incoming RF edge, we know that each machine j in C must be saturated. Therefore, it cannot be the case that C contains every machine, due to our assumption that $\sum_i p_i \leq \sum_j c_j$ (if every machine were saturated, then $\sum_i p_i$ units of total load would be assigned and the algorithm would have terminated)

Another observation we could make, but that is not crucial to our ensuing discussion, is that *if* the GS algorithm terminates, then it does so with no fractional cycles, since we could perform a rotation (see [4]) around such a cycle to make the assignment more job-optimal, and we know that the GS algorithm produces a job-optimal assignment. So we could strengthen the preceding observation to say that even in its terminal iteration, the GS algorithm cannot produce a fractional cycle containing all nodes.

3.3.3 Transient Configurations

We are now ready to prove the following:

Theorem 2. *The GS algorithm terminates after a finite number of iterations, even with irrational problem data.*

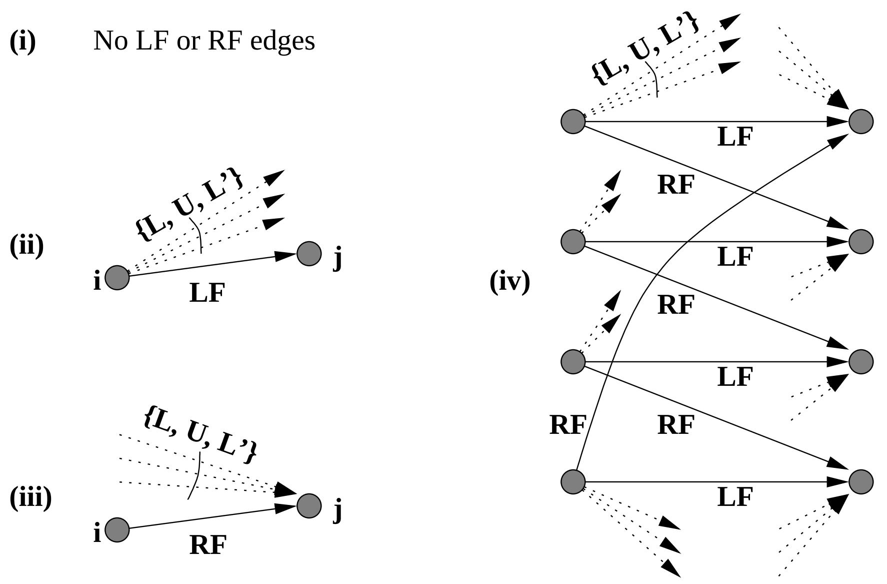

Figure 5: Transient configurations. Solid edges represent fractional edges (LF or RF) and dotted edges represent edges labeled either L, U, or L'.

Our proof of the theorem rests on analyzing certain types of labeled subgraphs, which we call *transient configurations*, that are known to last only a finite number of iterations before they disappear and never return. As long as the GS algorithm has not terminated, we argue that at least one of these configurations must be present, and since there are only finitely many such configurations, this implies that the GS algorithm must eventually terminate after a finite number of iterations.

Definition 5. *Consider the labeled edges in our bipartite assignment graph after some iteration of the GS algorithm. A subset of edges and their associated labels is called a* transient configuration *if after a finite number of iterations, either the GS algorithm must terminate or one of these edges must be promoted.*

For example, consider configuration type (i) listed in Figure 5, where simply every edge in our graph belongs to $L \cup U \cup L'$ (i.e., there are no fractional edges). This is a transient configuration because, if the GS algorithm has not terminated, its next proposal will utilize and promote one of the edges in L.

The remaining transient configuration types we consider are, as shown in Figure 5:

- Type (ii): The set of all edges emanating from a job i, where precisely one of these edges is in LF and the remaining are in $L \cup U \cup L'$.
- Type (iii): The set of all incoming edges to a machine j into which precisely one of these edges is in RF and the remaining are in $L \cup U \cup L'$.
- Type (iv): The set of all edges adjacent to the nodes in a fractional cycle, where edges on the cycle are in $LF \cup RF$ and all other edges are in $L \cup U \cup L'$.

Lemma 2. *After each iteration of the GS algorithm, we will find at least one configuration of type (i), (ii), (iii), or (iv).*

Proof. If there are no fractional edges then we have a configuration of type (i), so let us assume that either LF or RF edges exist. Consider only the subgraph of our assignment containing edges in $LF \cup RF$. Due to our previous observations, we know that each connected component in this subgraph is either a tree or a tree plus one additional edge (forming a unique cycle). If we have a tree component, then at one of its leaf edges we must find a configuration of either type (ii) or (iii). If we have a cycle component, then either it is a "pure" cycle (which is a configuration of type (iv)), or it consists of a cycle with trees branching off it, and again in this case we must find configurations of type (ii) or (iii) at the leaf edges of these trees. □

Lemma 3. *The type (iii) configuration is transient.*

Proof. Consider a machine j whose incoming edges currently form a type (iii) configuration. If subsequent iterations of the GS algorithm are to avoid promoting edges incoming to j, then there can be no proposals to j. The next proposal j receives must come from one of its incoming L edges (the others are in $RF \cup U \cup L'$ and hence are "spent" and will issue no further proposals). However, as we have observed, any proposal along an edge in L will promote the edge. Therefore, the set of edges incoming to j will retain their labeling for exactly as long as j receives no further proposals.

We argue using induction on the number of jobs and machines in our instance that the time until j must receive a proposal is finite. Let us form a strictly smaller problem instance by removing j and all its incident edges, and by subtracting x_{ij} from p_i for all machines i. Any sequence of iterations in our original instance with no proposals to j corresponds to an analogous sequence of proposals that is a valid GS sequence for the reduced problem instance. Since the reduced instance is strictly smaller, we know by applying Theorem 2 inductively that the GS algorithm terminates finitely on it. Hence, in the original problem instance we must encounter a proposal to j after a finite number of iterations, and this will promote one of the edges incoming to j. □

Lemma 4. *The type (ii) configuration is transient.*

Proof. This proof is similar and essentially symmetric to the preceding proof. Consider a job i whose outgoing edges currently form a type (ii) configuration. We consider two cases, the first of which involves i being saturated. Here, i will not issue any proposals until some of its load is rejected, but any such rejection will result in the promotion of one of i's outgoing edges. We therefore want to know how long the GS algorithm can continue to operate before some machine rejects i, and as before we argue by induction that this amount of time must be finite. This is done by removing i from the instance and subtracting x_{ij} from c_j for each machine j, and (just as before) noting a correspondence between any sequence of proposals for the GS algorithm in the original instance where no machine rejects i, and a valid sequence of proposals for the GS algorithm on the reduced instance.

Next, we consider the case where i is not saturated. Here, we repeat the argument above twice: once to argue that a finite amount of time must elapse before i must propose along the edge $(i, j) \in LF$ (at which point it becomes saturated), and again to argue as above that a finite amount of time must elapse before some machine rejects i. □

Lemma 5. *The type (iv) configuration is transient.*

Proof. Consider a cycle C of fractional edges (alternating between LF and RF) that forms a configuration of type (iv), and recall that C does not include every node in the entire graph. When the jobs in C issue proposals, they will use their outgoing LF edges and hence propose "inside" the cycle rather than to machines outside the cycle. When such proposals occur, machines inside the cycle will issue rejections along their incoming RF edges (also within the cycle). This behavior will finally stop when the cycle "breaks" due to one its edges being promoted (for example, if one of its RF edges joins L'). We can therefore view the remaining iterations of the GS algorithm as consisting of 3 types of proposals: (a) proposals from jobs in C to machines in C, (b) proposals from jobs not in C to machines in C, and (c) proposals from jobs not in C to machines not in C. A proposal of type (b) will promote an edge, so we need to determine how long the GS algorithm can continue to run while only issuing proposals of types (a) and (c). For this purpose, we can consider the cycle and the rest of the instance as two independent entities, and since these are both strictly smaller than the main problem instance, we know by induction that the GS algorithm can spend only a finite amount of time in both cases. □

The proof of Theorem 2 is now complete.

References

[1] M. Baiou and M. Balinski. Erratum: The stable allocation (or ordinal transportation) problem. *Mathematics of Operations Research*, 27(4):662–680, 2002.

[2] L.R. Ford and D.R. Fulkerson. *Flows in Networks.* Princeton University Press, 1962.

[3] D. Gale and L.S. Shapley. College admissions and the stability of marriage. *American Mathematical Monthly*, 69(1):9–14, 1962.

[4] D. Gusfield and R. Irving. *The Stable Marriage Problem: Structure and Algorithms.* MIT Press, 1989.

[5] U. Zwick. The smallest networks on which the ford-fulkerson maximum flow procedure may fail to terminate. *Theoretical Computer Science*, 148:165–170, 1995.