IEOR 8100: "Matchings" **Fall 2012**

Lecture 21: Distributed Greedy Maximum Weight Matching

Instructor: Cliff Stein *Lecturer: Jelena Marašević*

Even though most distributed algorithms seem very simple at a first glance, the analysis used to prove their correctness and give bounds on the running time can be very challenging. To show the basic ideas used in analyzing distributed matching algorithms, an example of a simple greedy algorithm for the maximum weight matching will be considered. We will start by showing the approximation ratio of a greedy algorithm that chooses (globally or locally) heaviest edge that can be added to the current matching in a centralized manner, then show how to make this algorithm distributed, show that it is correct, and analyze its running time. Finally, we will move to the case of a self-stabilizing algorithm, prove its correctness, and give some bounds on the running time.

In all examples, we will observe an undirected, weighted graph $G = (V, E)$, $|V| = n$, $|E| = m$, with a given positive weight function $w : E \to \mathbb{R}_+ \backslash \{0\}$. We will assume that all the weights on edges are unique, even though all the results could be applied to a more general case in which weights are not necessarily unique. This can be done by associating a unique ID with each edge and breaking ties in favor of edge with the higher ID. Terms graph and network will be used interchangeably when referred to G.

21.1 Centralized Greedy Algorithm

Consider the following greedy algorithm, run in a centralized manner:

GREEDY-CENTRALIZED-GLOBAL(G)
$M = \emptyset$
while($E \neq \emptyset$)
    select the heaviest edge e from E
    $M = M \cup \{e\}$
    delete e and all the edges incident to it from E
return M

In [1], David Avis showed that this strategy gives a $\frac{1}{2}$-approximation ratio, as the following lemma states.

Lemma 21.1 *For a given graph $G = (V, E)$, let M^* be a maximum weight matching, and M be the matching obtained by running* GREEDY-CENTRALIZED-GLOBAL *algorithm. Then $w(M) \geq \frac{1}{2} w(M^*)$.*

Proof: Every time the algorithm selects an edge $e \in E$ and adds it to the current matching, it removes that edge together with all the edges incident to it from the graph, with e being the heaviest among all the edges that get removed. Observe e and all the edges incident to it. There can be three possible cases:

1. $e \in M \cap M^*$

2. There is exactly one edge $e_1 \in M^*$ incident to e. As e is the heaviest: $w(e) > w(e_1) > \frac{1}{2}w(e_1)$
3. There are exactly two edges e_1 and e_2 from M^* incident to e. As $w(e) > w(e_1)$ and $w(e) > w(e_2)$, it follows that $w(e) > \frac{1}{2}(w(e_1) + w(e_2))$.

Summing over all the edges from M, we get:

$$\begin{aligned} w(M) &= w(M \cap M^*) + w(M \backslash M^*) \\ &= w(M \cap M^*) + \sum_{e \in M \backslash M^*} w(e) \\ &> w(M \cap M^*) + \frac{1}{2} \sum_{e \in M^* \backslash M} w(e) \\ &> \frac{1}{2}(w(M \cap M^*) + w(M^* \backslash M)) \\ &= \frac{1}{2} w(M^*) \end{aligned}$$

■

Proof of the previous lemma shows even more than asserted: to achieve $\frac{1}{2}$-approximation ratio it is not necessary to choose the globally heaviest edge; it is enough to select the one that is locally heaviest. Robert Preis used this idea in [2] to develop an algorithm that runs in $O(m)$ time (as compared to $O(m \log n)$ time the above algorithm would have due to edge sorting). We won't get into details of this algorithm, but only state the (high-level) pseudocode for it:

```
GREEDY-CENTRALIZED(G)
M = ∅
while(E ≠ ∅)
    select some locally heaviest edge e from E
    M = M ∪ {e}
    delete e and all the edges incident to it from E
return M
```

21.2 Distributed Greedy Algorithm

Imagine the situation in which we don't want to run an algorithm for the entire graph from one machine. Instead, we associate a computing device with each node in the network, letting each node know only about its neighboring nodes and edges leading to those nodes. Two neighboring nodes can learn about each other by either reading from designated memory registers that both can access (shared memory model), or by exchanging messages (send/receive or message passing model). The goal is to run the same algorithm at each node and obtain (an approximate) maximum weight matching. Jaap-Henk Hoepman showed in [3] how this can be done by using a greedy algorithm. Before delving into the details of this algorithm, we will first introduce basic notions about distributed algorithms in the following section.

21.2.1 Distributed Algorithms–Modeling and Performance Measures

When modeling a distributed system, we assume that there is a computing element associated with each node of the network. This computing element is often referred to as a process. Each process maintains

some set of variables that describe its state. These variables are initialized by values from some known set before the algorithm is run. Each process can look into its own state and into states of all of its neighbors, and based on this information transition to a new state (i.e., change values of the variables that describe its state). Every node implements the same algorithm (the same set of rules): the algorithm describes how the state of the process is updated, given the current state and the states of all the neighbors. The transition from one state to another is often called a step. There are two models of communication between processes:

- **shared memory model**, in which a process can read its state and states of its neighbors from some designated memory registers. Process can also write to the memory register that describes its own state. Sometimes it is also assumed that there exists some set of registers that all the processes are allowed to both read from and write to.
- **send/receive (or message passing) model**, in which processes learn about each other by exchanging messages. For all the processes that can communicate, there is a channel (link) through which messages are sent. In some models, the channel is uncertain and messages can get lost with a given probability.

Processes can either be assumed to run simultaneously and with the same speed, or to require an arbitrary time to complete their actions. There is also a third model that represents a compromise between these two cases. More specifically, there are three basic timing models [4]:

- **synchronous model,** in which all the nodes can communicate in a synchronous manner–this means that they can transition from one state to another at the same speed, and that all the transitions occur at the same time. As all the steps are taken simultaneously, it is said that the execution proceeds in synchronous rounds. The time complexity is measured in the number of rounds it takes the algorithm to converge.
- **asynchronous model,** a more general model, in which processes can take steps in an arbitrary order, at arbitrary relative speeds. The system is more difficult to analyze, as events can happen in any order. The time complexity is usually measured in the number of steps processes take (in the worst case scenario) until the algorithm converges.
- **partially synchronous (timing-based) model,** which places some restrictions on the relative timing of events, but there is less synchronism than in the synchronous model. Even though these models are the most realistic, they are also the most difficult to program.

In the following section, we will consider an asynchronous, send/receive model. Later, for the analysis of the self-stabilizing algorithm, we will adopt a shared memory asynchronous model.

21.2.2 Distributed Greedy Weighted Matching

Before looking into the pseudocode and the formal analysis of the algorithm, let us first gain some intuition on how the algorithm works. Each process sitting on its corresponding node can see who are the neighboring nodes, and what are the costs (weights) of edges that lead to those nodes. It can also communicate with all the neighboring nodes by sending and receiving messages. How can a node tell if some edge is locally heaviest? Well, suppose that for an edge $e = (u, v)$ both u and v see this edge as the heaviest one of all the edges they can see. Then, e is heavier than all the edges incident to it, that is, locally heaviest edge. Now, if u and v choose each other as a matching pair, and notify all their remaining neighbors about it, their neighbors can stop looking at the edges incident to (u, v), which corresponds to removing these edges from the graph. Intuitively, by running this procedure on all the nodes over and over again, until all of them find their matching pair (or become lonely–i.e., see no neighbors), we should eventually get the same matching as we would for some run of the Greedy-Centralized algorithm.

To formalize the statement of the algorithm described above, let us introduce some notation. We will denote by $\Gamma(v)$ the set of neighbors of the node v in the original graph G. As every time a v's neighbor u gets matched it cannot be considered as a matching pair for v anymore, we want to drop an edge (u, v) from the graph when this happens, and observe only those neighbors that v can still match to. We will assume that when u gets matched to another node it sends a drop message to v (and all other non-matching neighbors), and denote the set of neighbors who are possible matches for v by $N(v)$. Looking at $N(v)$, v will choose the heaviest edge it can see and send a request to the neighbor adjacent to that edge. We will call this neighbor a *candidate* and denote it by $c(v)$. If v receives a request message from its candidate neighbor, it knows that the edge $(v, c(v))$ is locally heaviest and chooses $c(v)$ as its matching pair. We will allow every node to store the list R of the unmatched nodes it received a request message from, and update it every time a request or drop message arrives. The pseudocode for the algorithm run at the node v is stated below:

```
GREEDY-DISTRIBUTED(G, v)
R = ∅
N = Γ(v)
c = candidate(v, N)
if c ≠ null
        send < req > to c
while(N ≠ ∅)
    receive a message m from neighbor u
    if m =< req >
        R = R ∪ {u}
    if m =< drop >
        N = N\{u}
        R = R\{u}
        if u = c
            c = candidate(v, N)
            if c ≠ null
                send < req > to c
    if c ≠ null ∧ c ∈ R
        forall w ∈ N\{c}
            send < drop > to w
        N = ∅
```

The *candidate* in the algorithm above is determined as:

$$candidate(v, N) = u \in N : (\forall u' \in N :: w(u, v) \geq w(u', v)),$$

that is, as the node in the set of the remaining neighbors that is adjacent to the heaviest edge that v can see.

Figure 21.1 depicts an example of one algorithm run.

Correctness and the time complexity. To show that the algorithm is correct, we will show that it simulates the centralized greedy algorithm (GREEDY-CENTRALIZED) we had before. For a run of the algorithm described above, define a *matching event* as the event when a node u sends a $< req >$ message to a node v and v sends a $< req >$ message to u (note that when this happens it is also true that $c(u) = v$ and $c(v) = u$). Order all the matching events in order of their occurrence, letting x_0 be the wake up event and x_1 be the first matching event. Let $e_i = (u_i, v_i)$ denote the edge that gets matched by the event x_i. Define the set of remaining edges E_i inductively as:

$$\begin{aligned} E_0 &= E \\ E_i &= E_{i-1} \backslash \{\text{all edges incident to } u_i \text{ and } v_i\},\ i \geq 1 \end{aligned}$$

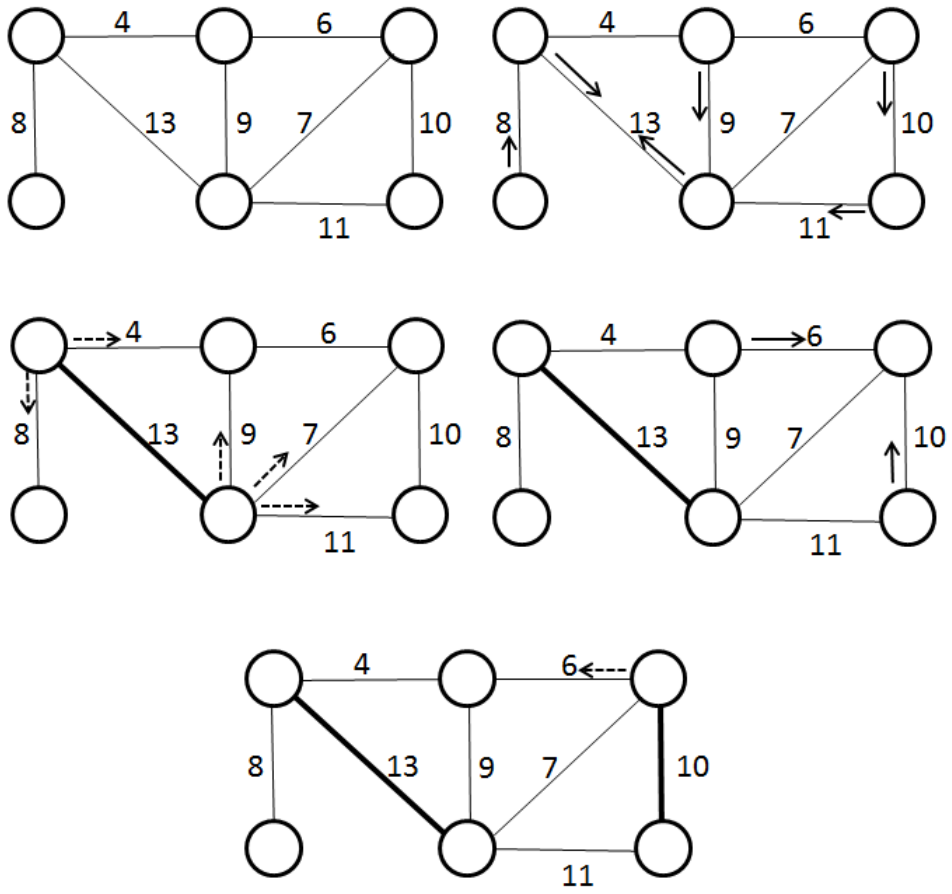

Figure 21.1: An example of a GREEDY-DISTRIBUTED run. Solid arrows represent $< req >$ messages, while dashed arrows represent $< drop >$ messages.

Lemma 21.2 *In* GREEDY-DISTRIBUTED *algorithm, each node (process) sends at most one message over each incident edge.*

Proof: Fix an arbitrary node v. Throughout the execution of the algorithm, v sends one $< req >$ message to its first candidate, and can send another $< req >$ message only if a new candidate is chosen, which can happen only if the current candidate gets removed from $N(v)$. The algorithm never adds new elements to $N(v)$. As $< drop >$ message can be sent only to non-candidate nodes, after which $N(v)$ gets cleared ($N(v) = \emptyset$), it follows that v can send at most one message to each of the neighboring nodes, as claimed. ■

Lemma 21.3 *After x_i and before x_{i+1} (if it occurs), if $(u, v) \in E_i$, then $u \in N(v)$ and $v \in N(u)$.*

Proof: The claim is true initially. Suppose that $(u, v) \in E_i$, but u got removed from $N(v)$. How could this happen? This happened either because v received a $< req >$ message from its candidate neighbor $c(v)$, or u sent a $< drop >$ message to v. In the first case, v gets matched to a node $c(v)$, whereas in the second case u gets matched to $c(u) \neq v$. In either case, as u or v (or both) get matched, (u, v) must have been dropped from E before the matching event x_{i+1} occurred, which is a contradiction. ■

Lemma 21.4 *For all i, $e_i \in E_{i-1}$.*

Proof: Suppose that $e_i = (u, v)$ got removed in some matching event x_j, $j < i$. This can happen because either (u, v) got matched, or some edge incident to it got matched. If x_j matched (u, v), then both nodes have cleared their neighbor lists in x_j, so neither of them can send a $< req >$ message in x_i, which contradicts the assumption that x_i matches (u, v). Therefore, either u or v got matched (but not to each other), in which case the node that got matched sent a $< drop >$ message over (u, v). Suppose that u got matched (the proof is symmetric for v). Then, v must have gotten removed from $N(u)$ in x_j, so v cannot be its candidate in the matching event x_i. ■

Lemma 21.5 GREEDY-DISTRIBUTED *terminates for every node in the network, with $E_t = \emptyset$ for some t.*

Proof: The algorithm terminates when at each node v the list of neighbors $N(v)$ becomes empty. By the lemma 21.2, each node sends at most one ($< req >$ or $< drop >$) message over each of its adjacent edges. After a node v has received all the requests, it can only remove elements from $N(v)$. Suppose that, for some v, $N(v) \neq \emptyset$. Then $c(v) = u \neq null$, and $v \in N(u)$. If u sends a $< req >$ message to v, then they match to each other and both of these nodes clear their neighbor lists, so $N(u) = N(v) = \emptyset$. Otherwise, if u gets matched to another node, it sends a $< drop >$ message to v, so v removes u from its list. By finiteness of the graph, and lemma 21.2, u must eventually send either a $< req >$ or $< drop >$ message to v. Therefore, eventually, $N(v) = \emptyset$. As v was chosen arbitrarily, this is true for all the nodes in the graph.

To show that $E_t = \emptyset$ for some $t < \infty$, consider the moment when all the nodes have terminated. Then $E_t = \emptyset$ follows as a contrapositive of the lemma 21.3. ■

Lemma 21.6 *Matching edge e_i is a locally heaviest edge in E_{i-1}.*

Proof: Let $e_i = (u, v)$. By lemma 21.4, $e_i \in E_{i-1}$. Suppose that (u, v) is not locally heaviest. Then there is an edge (u, w) (or (v, w)) heavier than (u, v) and all other edges incident to (u, v). But then, by lemma 21.3, $w \in N(u)$ ($w \in N(v)$), and $c(u) = w$ ($c(v) = w$), which is a contradiction, as $c(u) = v$ ($c(v) = u$). ■

Theorem 21.7 *For any graph $G = (V, E)$,* GREEDY-DISTRIBUTED *computes a matching $M(G)$ in time $O(m)$ (where $m = |E|$), such that $w(M) \geq \frac{1}{2} w(M^*)$.*

Proof: To show that GREEDY-DISTRIBUTED algorithm gives $\frac{1}{2}$-approximation, we will observe that for each run of the distributed algorithm there exists a run of GREEDY-CENTRALIZED that produces the same matching, and the result will follow. For the distributed algorithm, let x_i be the ordered sequence of matching events, E_i the set of remaining edges after i^{th} matching event, as defined above.

Now, for the centralized algorithm (GREEDY-CENTRALIZED), let E'_i denote the set of remaining edges after i^{th} edge gets selected by the algorithm, $E'_0 = E_0$. By the lemma 23.6, every edge e_i that gets matched by GREEDY-DISTRIBUTED is locally heaviest at the time it gets matched, so let GREEDY-CENTRALIZED choose the edges in the same order as GREEDY-DISTRIBUTED. Then, $E'_i = E_i$ for each i. By lemma 21.5, for some t $E_t = \emptyset$, so when this happens also $E'_t = \emptyset$, and GREEDY-CENTRALIZED cannot add any additional edges, and the approximation ratio of $\frac{1}{2}$ follows.

The time complexity follows from the lemma 21.2.

■

21.3 Self-Stabilizing Algorithm and Analysis

We will now move to a more general concept of distributed systems–self-stabilizing systems, and see what is the way in which a matching algorithm can be designed for this type of systems. Self-stabilization as an area is quite young–it was first introduced by Edsger W. Dijkstra in 1973, but hasn't received much attention until 1983, when Leslie Lamport pointed out its significance in an invited talk at the ACM Symposium on Principles of Distributed Computing (PODC). The book [5] written by S. Dolev in 2000 summarizes most of the ideas and algorithms designed for self-stabilizing systems. We will focus here on a greedy algorithm for maximum weighted matching, originally proposed by F. Manne and M. Mjelde in [6] and further analyzed by V. Turau and B. Hauck in [7].

21.3.1 Background on Self-Stabilizing Algorithms

Self-stabilizing system represents a more general case of a distributed system, in which nodes (processes) can start executing the algorithm from an arbitrary state (no specific initialization is assumed, unlike in standard distributed systems), and must converge to a desired behavior after some bounded time. As the system can start from an arbitrary state, it is able to recover from (any number of) transient faults after finite time. This is the main motivation in designing self-stabilizing algorithms, as the system can handle any dynamic structure of the underlying graph (as long as that structure remains unchanged for sufficiently long time).

As before, each node v has a set of variables that describe its state s_v and can read the states of its neighboring nodes, by either reading some memory registers or by exchanging messages. For the algorithm described here, we will assume the shared memory model, i.e., we will assume that a node accesses its neighbor state variables by reading from designated memory registers. Based on its own state and states of neighboring nodes, a node can determine whether is should update its state or not. We associate a Boolean predicate with each node to determine whether it should perform an action or not–if the predicate evaluates *true*, the node is required to update its state and we say that it is *enabled*, whereas if the predicate evaluates *false*, no action is required for the node (it is not *enabled*). A single update of node's state is called the *move*. Notice here that as action of each node is dependent on the states of neighboring nodes, the relative order in which moves are made determines the behavior of the algorithm. To model how the states of all the nodes in the graph get updated, the algorithms execution is observed in (time) steps. At the beginning of a step, each node determines whether it is enabled or not. After this happens, a *scheduler* determines which nodes will make a step. There are three basic models of the scheduler:

- **central scheduler (sequential model)**, which allows only a single node to make a move in one step,
- **distributed scheduler (general distributed model)**, which can select any subset of the enabled nodes and allow them to make a move, and
- **synchronous scheduler (synchronous model)**, which lets all the enabled nodes make a move.

Notice here that a scheduler is used to model the relative order of moves between enabled nodes and it does not necessarily exist as an individual element of a self-stabilizing system. For central and distributed schedulers, in general, there is no restriction on their scheduling policy. Therefore, it can happen that a node is enabled infinitely often, but it never performs a move. In that case, a scheduler is said to be *unfair*. A scheduler is said to be *fair* if every node that gets enabled at some step after finite number of steps either makes a move or stops being enabled.

A *configuration* C of the graph G is defined as an n-tuple of all the nodes' states: $C = (s_{v_1}, s_{v_2}, ..., s_{v_n})$. Therefore, we can also define a step as a transition from one configuration C_i to another configuration

C_{i+1}. We define an *execution* of the algorithm as a sequence $C_0, C_1, C_2, ...$ of configurations, such that each configuration except for the first one is obtained from the previous one by performing a single step.

For an algorithm to be self-stabilizing, it should exhibit some desired behavior after finite number of steps. We refer to this behavior as *legal* (or *legitimate*) and say that a configuration C in which algorithm shows legal behavior is *legal* or *stable*. It is clear that for the algorithm to operate correctly, the following two properties should be satisfied:

- **closure property**: if the system performs a step from some legal configuration C_i, the following configuration C_{i+1} must also be legal,
- **convergence property**: for every execution $C_0, C_1, C_2, ...$ there exists $i < \infty$ such that C_i is legal.

The time complexity of self-stabilizing algorithms can be measured in the number of rounds (where a round is the minimal sequence of steps in which every node enabled at the beginning of the round either makes a move or becomes disabled), steps, or individual node moves until a legal (stable) configuration is reached.

21.3.2 Greedy Self-Stabilizing Algorithm

In the algorithm presented in this section, we will assume that every node v has two variables m_v and w_v associated with it. In a stable configuration, m_v should contain a pointer to the node u that v gets matched to, and $w_v = w(v, m_v) = w(v, u)$. If a node v doesn't get matched to any node, then $m_v = null$ and $w_v = 0$. As for the distributed algorithm, we will use $\Gamma(v)$ to denote the set of neighbors of node v in the graph G, and $N(v) \subset \Gamma(v)$ to denote the set of "candidate" neighbors that v can get matched to. A node u can be a "candidate" for v if it is in its neighbor list and would achieve the same or a better matching if matched to v, to wit: $N(v) = \{u \in \Gamma(v) : w(u, v) \geq w_u\}$. Note that if u is matched to v, it will also belong to this set.

The algorithm is quite simple: every node $v \in V$ looks at its candidate neighbors and determines which one of them, u, is adjacent to the heaviest edge from v (if $N(v) = \emptyset$, it sets $u = null$). Then, if v is not already matched to this node (i.e., if $m_v \neq u$) or if the information that v has about the weight of the edge (v, m_v) is inaccurate, it updates its state variables by setting $m_v = u$ and $w_v = w(u, v)$. The algorithm stabilizes when $(\forall u \in V)(\forall v \in V)\ u = m_v \Leftrightarrow v = m_u$, and, at the same time $w_v = w(v, m_v)$, $\forall v$. The pseudocode for this algorithm is given below.

GREEDY-SS(v)
if $m_v \neq BestMatch(v)$ or $w_v \neq w(v, m_v)$
    $m_v = BestMatch(v)$
    $w_v = w(v, m_v)$

$$BestMatch(v) = \underset{u \in N(v) \cup \{null\}}{\arg\max} w(u, v)$$

Let us stop here for a moment and try to understand what this algorithm tries to achieve. Suppose that we have reached a stable configuration, and let (u, v) be a matched edge, that is, an edge such that $u = m_v$ and $v = m_u$. Look at all the edges incident to (u, v). Every edge $(v, w) \neq (u, v)$ is either incident to another edge (w, x) such that $w(w, x) > w(v, w)$ (so w doesn't belong to $N(v)$), or it is true that $w(u, v) > w(v, w)$ (otherwise v would have chosen w as its matching pair). The same argument can be made for all the edges $(u, w) \neq (u, v)$. Therefore, (u, v) is locally heaviest, so GREEDY-SS choses the same matching as GREEDY-DISTRIBUTED, and the approximation ratio of $\frac{1}{2}$ follows directly! However, to argue the correctness of GREEDY-SS, we must argue that it always reaches a stable configuration, regardless of its initial configuration.

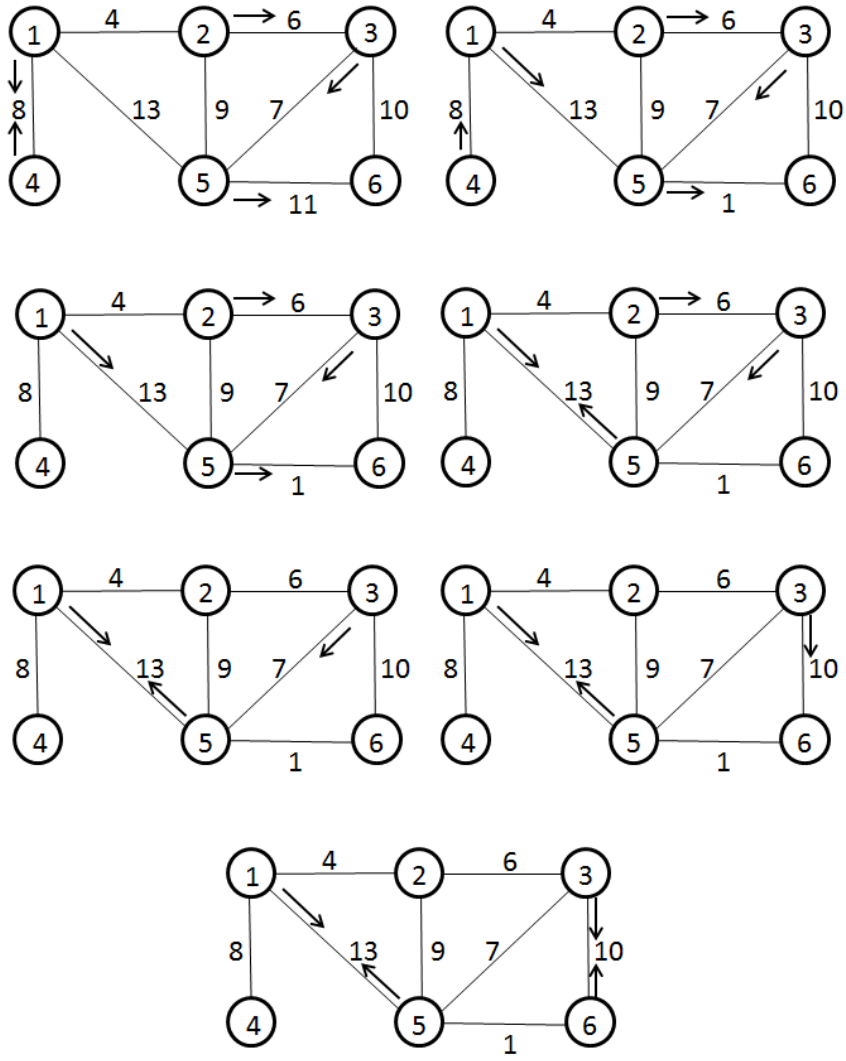

Figure 21.2: An example of a GREEDY-SS run. Arrows point from a node to its chosen matching pair.

Figure 21.2 shows one example of the algorithm run, assuming that one node makes a move at a time (central scheduler–sequential model). The system starts from a configuration that doesn't even correspond to a matching. In the first step, node 1 is selected to make a move. At that point, $N(1) = \{4, 5\}$, $m_1 = 4$, and as $BestMatch(1) = 5$, node 1 updates its state, so as $m_1 = 5$ and $w_1 = 13$. In the following steps, nodes $4, 5, 2, 3, 6$ make their moves, respectively.

Correctness. We start by an observation that follows directly from looking at the algorithm pseudocode: in a stable configuration $m_v \in N(v) \cup \{null\}$ and $w_v = w(v, m_v)$, for every node $v \in V$.

The next step is to show that once the algorithm has converged, what we have obtained is a matching. The following lemma formalizes this claim.

Lemma 21.8 *In a stable configuration $m_v = u \Leftrightarrow m_u = v$ for every edge $(u, v) \in E$.*

Proof: $\Rightarrow$. Assume that the system has reached a stable configuration in which $m_v = u$, but $m_u = w \neq v$.

As all the edge weights are unique, we have the following two possibilities:

i) $w(u,v) > w(u,w)$: in this case, as $w(u,v) = w_v$, it follows that $v \in N(u)$, and therefore u would be enabled, which contradicts the assumption that the configuration was stable.

ii) $w(u,v) < w(u,w)$: in this case $u \notin N(v)$, so u is not the best match for v ($BestMatch(v) \neq u$), so v is enabled, which, again, contradicts the assumption that the system was in a stable configuration.

Therefore, $m_v = u \Rightarrow m_u = v$, and, by the symmetric argument, $m_u = v \Rightarrow m_v = u$, which proves the lemma. ■

The following lemma will allow us to show that GREEDY-SS gives $\frac{1}{2}$-approximation.

Lemma 21.9 *In a stable configuration, for every edge $(u,v) \in E$ we have $w(u,v) \leq \max(w_u, w_v)$.*

Proof: Assume that the system has reached a stable configuration, and that there is an edge (u,v) such that $w(u,v) > \max(w_u, w_v)$. As $w(u,v) > w_u$, it follows that $u \in N(v)$. Since the configuration is stable, there exists a node $x \in N(v)$, such that $w(x,v) \geq w(u,v)$. But as all the edge weights are unique, if $w(x,v) = w(u,v)$, then $x = u$, otherwise $w(x,v) > w(u,v)$. But the former cannot be the case, as we would have $w(u,v) = w_v$, which contradicts that $w(u,v) > w_v$. The latter implies $w_v = w(x,v) > w(u,v)$, which is, again, a contradiction, as we assumed that $w(u,v) > w_v$. ■

In a given stable configuration of graph G, with respect to the algorithm GREEDY-SS, we will call a stable matching, denoted by M, a subset of edges (u,v) from E, such that $m_v = u$ (and $m_u = v$, as given configuration is stable). Now we are ready to prove the approximation ratio.

Theorem 21.10 A*ny stable matching M obtained by* GREEDY-SS *algorithm provides a $\frac{1}{2}$-approximation for the maximum weight matching problem.*

Proof: Let M^* denote the maximum weight matching for the graph G. Fix an arbitrary edge $e = (u,v) \in M^*$. By the previous lemma, $w(e) \leq \max(w_u, w_v) < w_u + w_v$, as all the weights are positive. Summing over all the edges from M^*, we get $w(M^*) < 2w(M)$, which proves the theorem. ■

Convergence and the time complexity. We will now bound the number of moves it takes the algorithm to reach a stable configuration. As soon as we show that this number is finite, we have shown that the algorithm converges, because in a stable configuration no node changes its state variables (look at the pseudocode for GREEDY-SS). The time complexity in the case of an unfair scheduler was shown in [7] to be $O(mn)$ moves. The analysis used for proving this bound is rather involved and won't be covered here. Instead, we will focus on the case of a *fair distributed scheduler*. Notice that this case subsumes the cases of central and synchronous fair schedulers, as the former is obtained by selecting a single enabled node, and the latter is obtained by selecting all the enabled nodes, in each step.

Recall that a round was defined as a minimal sequence of steps after which every node that was enabled at the beginning of the round has either made at least one step, or became disabled. As the scheduler is fair, every node that is enabled infinitely often moves infinitely often, so we can take that each round takes $O(n)$ moves.

Lemma 21.11 *After at most one round: $m_v \in \Gamma(v) \cup \{null\}$ and $w_v = w(v, m_v)$, for every $v \in V$.*

Proof: At the beginning of the first round, all the nodes that are enabled either have incorrect value for the edge leading to their matching pair ($w_v \neq w(v, m_v)$), or are not matched to their best match. Nodes

that are not enabled must have $m_v \in \{N(v) \cup \{null\}\} \subset \{\Gamma(v) \cup \{null\}\}$ and $w_v = w(v, m_v)$. As during one round every enabled node makes at least one move, each one of them will set m_v to some value from $\{N(v) \cup \{null\}\} \subset \{\Gamma(v) \cup \{null\}\}$ and w_v to $w(v, m_v)$. No move can cause a node v to choose a matching pair that doesn't belong to its neighbor list, nor to set w_v to a value different than $w(v, m_v)$. ■

Note that it is not necessarily the case that we have $m_v \in N(v) \cup \{null\}$ after the first round, as m_v can match to a node $x \neq v$ after v has made its move, and thus leave $N(v)$.

Lemma 21.12 *After at most two rounds, the heaviest edge $(u, v) \in E$ is in M. Furthermore, once in M, (u, v) never leaves M.*

Proof: From the previous lemma, after the first round every node $v \in V$ has a correct value of w_v and $m_v \in \Gamma \cup \{null\}$. If (u, v) is the heaviest edge in the graph, then it must be $w_u \leq w(u, v)$ and $w_v \leq w(u, v)$, implying that $u \in N(v)$ and $v \in N(u)$. If $m_u \neq v$, then u is enabled, and if $m_v \neq u$, then v is enabled. If these two nodes are not matched to each other, they will get matched to each other after at most one round, as (u, v) is the heaviest edge and no other neighbor can offer a better matching.

Once u and v get matched to each other, neither of them can become enabled again, as $w_u = w_v = w(u, v) > w(e) \, \forall e \in E, \, e \neq (u, v)$. ■

Theorem 21.13 GREEDY-SS *converges after at most $2|M| + 1$ rounds.*

Proof: Let $e_1, e_2, ..., e_{|M|}$, where $e_i = (u_i, v_i)$, represent edges from E sorted in the order as they would get chosen by GREEDY-CENTRALIZED-GLOBAL algorithm from the first section (recall that this algorithm always choses globally heaviest edge that can be added to the matching).

By the previous lemma, we have that after at most 2 rounds $e_1 \in |M|$. As $w_{u_1} = w_{v_1} = w(e_1) > w(e) \, \forall e \in E, \, e \neq e_1$, neither of the nodes different from u_1 and v_1 can have one of these nodes in their neighbor list. This means that after e_1 gets matched, no edge incident to it can be added to the matching. Thus, after e_1 gets matched, e_2 is the globally heaviest edge that can possibly be added to the current matching (if not already in it). After at most one round it must be $u_2 \in N(v_2)$ and $v_2 \in N(u_2)$. If these two nodes are not matched to each other, by the same argument as in lemma 21.12, they get matched by the end of the round. Reapplying the same argument, we get that after $2|M|$ rounds all the nodes $u_1, ..., u_{|M|}, v_1, ..., v_{|M|}$ get matched correctly. As not all the nodes necessarily contribute to the matching $|M|$, after at most one additional round these nodes set their matching pair to be *null*. ■

The bound of $O(|M|n) = O(n^2)$ moves for the time complexity now follows directly from the Theorem 21.13.

References

[1] D. Avis, "A survey of heuristics for the weighted matching problem," *Networks*, vol. 13, no. 4, pp. 475–493, 1983.

[2] R. Preis, "Linear time 1/2 -approximation algorithm for maximum weighted matching in general graphs," in *Proceedings of the 16th annual conference on Theoretical aspects of computer science*, STACS'99, (Berlin, Heidelberg), pp. 259–269, Springer-Verlag, 1999.

[3] J.-H. Hoepman, "Simple distributed weighted matchings," *CoRR*, vol. cs.DC/0410047, 2004.

[4] N. A. Lynch, *Distributed Algorithms.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.

[5] S. Dolev, *Self-stabilization.* Cambridge, MA, USA: MIT Press, 2000.

[6] F. Manne and M. Mjelde, "A self-stabilizing weighted matching algorithm," in *Proceedings of the 9h international conference on Stabilization, safety, and security of distributed systems*, SSS'07, (Berlin, Heidelberg), pp. 383–393, Springer-Verlag, 2007.

[7] V. Turau and B. Hauck, "A new analysis of a self-stabilizing maximum weight matching algorithm with approximation ratio 2," *Theor. Comput. Sci.*, vol. 412, pp. 5527–5540, Sept. 2011.