# Applications of Matchings

Jonathan Turner

February 28, 2013

## 1 Scheduling in Packet Switches

In this section, we'll look at some applications of the matching problem. Our first application arises in the context of packet switching. Many internet routers use a *crossbar switch* to connect inputs to outputs. A crossbar is an array of horizontal and vertical wires, that can be connected using *crosspoints* at the intersection of each pair of wires. This is illustrated on the left hand side of Figure 1.

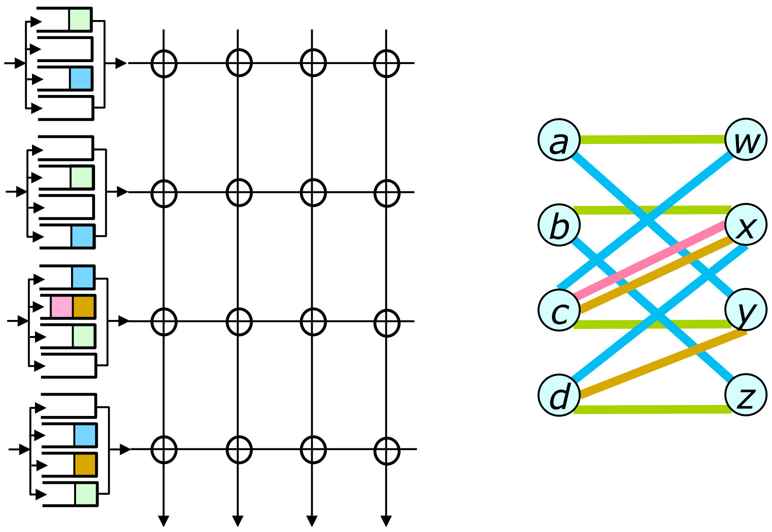

Figure 1: Crossbar and scheduling graph

At each input of the crossbar, there is a set of queues containing packets, with each input having a separate queue for each output. During each time step, the crossbar can transfer a packet from each input and deliver a packet to each output, but each input can supply only one packet at a time, and each output can accept only one packet at a time.

The objective of the *crossbar scheduing problem* is to determine a schedule for transferring a given set of packets in the smallest possible number of time steps. We can formulate this as an *edge coloring problem* in a bipartite graph. An edge coloring is an assignment of *colors* $c_1, c_2, \ldots$ to the edges of a graph in such a way that no vertex is incident to two edges of the same color. The objective of the edge coloring problem is to find an edge coloring with the smallest possible number of colors.

To formulate the crossbar scheduling problem as a graph coloring problem, we construct a bipartite graph with a vertex for every input, a vertex for every output and an edge $(i, j)$ for each packet at input $i$ that is to be delivered to output $j$ (see the right hand side of Figure 1). If the graph can be colored with $C$ colors, we can schedule the transfer of packets through the crossbar in $C$ time steps by mapping the edge colors to time steps. Note that the requirement that no two edges at a vertex have the same color is exactly what we need to avoid scheduling conflicts in the crossbar. A bipartite graph with maximum vertex degree $\Delta$ can be colored with exactly $\Delta$ colors, and we clearly cannot do better than this since a maximum degree vertex requires $\Delta$ distinct colors for its incident edges.

Note that each "color set" in an edge coloring is a matching in the graph. One way to find a minimal edge coloring involves construct a series of matchings. More specifically, we start by making a copy $G'$ of the original graph $G$ and then repeat the following step so long as there are edges remaining in $G'$.

> *Matching step.* Find a matching that includes an edge incident to every vertex of maximum degree in $G'$. Color the edges in the matching with a previously unused color and then remove the edges from $G'$.

Assuming that each step is successful, this will produce a coloring with $\Delta$ colors, as each step reduces the maximum degree by 1. Now, we know how to find a maximum size matching, but that's not what we need in this situation. How can we construct a matching that is guaranteed to include an edge incident to every maximum degree vertex?

One way to do this is based on a variant of the augmenting path algorithm for finding a maximum size matching. Let $p$ be an alternating path with an even number of edges and an unmatched endpoint $r$ of maximum degree. If the other endpoint of $p$ does not have maximum degree then we can "flip" the edges on $p$ to obtain a new matching with the same number of edges as before, but with one more matched vertex of maximum degree. If the other endpoint of $p$ is adjacent to an unmatched vertex, the connecting edge, together with $p$ forms an augmenting path. By flipping the edges on this path, we obtain a new matching with one more edge and at least one more matched vertex of maximum degree.

To find a path $p$ we build a single tree rooted at $r$, in much the same way we built a set of trees in the original augmenting path algorithm. We start by selecting a vertex $r$ of maximum degree. We let $state(r) = even$ and for all other vertices $u$, let $state(u) = unreached$. We also let $p(u) = null$ for all vertices $u$. Now, repeat the following step until we find a path that can be used to extend the matching.

Let $e = \{v, w\}$ be a previously unexamined edge with $v$ even.

- If $w$ is not unreached, ignore $e$ and proceed to the next edge.
- If $w$ is unreached and unmatched, then the edge $e$ together with the tree path from $v$ to $r$ is an augmenting path, and we terminate the path search.
- If $w$ is unreached and matched, let $w, x$ be the matching edge incident to $w$. Expand the tree by making $p(w) = v$, $p(x) = w$, $state(w) = odd$ and $state(x) = even$. If $x$ is not a maximum degree vertex, then the tree path from $x$ to $r$ can be flipped to extend the matching, and we terminate the path search.

The algorithm is illustrated in Figure 2.

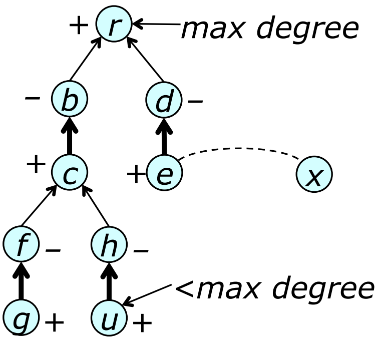

Figure 2: Matching vertices of maximum degree

We claim that for bipartite graphs, the algorithm always returns a path. To understand why, note that the basic step described above maintains the following invariant, as long as it does not terminate.

The number of odd vertices is one less than the number of even vertices and all even vertices have maximum degree.

This is true because every non-terminating step adds one odd and one even vertex, and the algorithm terminates whenever we reach an even vertex that does not have maximum degree. Note that since the even and odd vertices are joined by edges, they are in different subsets of the partition defined by the bipartite graph. Let $X$ be the subset containing the even vertices and $Y$ be the subset containing the odd vertices. Now, suppose there are $k$ even vertices and the maximum degree is $\Delta$. Since no vertex has degree larger than $\Delta$, the number of vertices in $Y$ that are adjacent to an even vertex must be at least $k$. But only $k-1$ of these are currently in the tree, so there must be at least one unreached vertex in $Y$ that has an edge connecting it to an even vertex. Consequently, the algorithm will not terminate without returning a path.

## 2 Max flows with Minimum Flow Requirements

In the last section we saw how we could adapt the augmenting path algorithm for maximum size matchings in bipartite graphs to the problem of finding a matching that includes every vertex of maximum degree. There is another way we can solve this problem using a generalization of the maximum flow problem. In this version of the problem, the input includes *minimum flow requirements* for some edges, and our objective is to find a maximum flow that satisfies all the min flow requirements, in addition to the usual capacity constraints.

Before we look at how we can solve this problem in general, let's consider how we can use it to solve the maximum degree matching problem. The idea is to transform our original bipartite graph into a flow graph as we have done before. However in this case, we are going to add a minimum flow requirement of 1 to each source-edge going to an original vertex of max degree. We'll also add a minimum flow requirement of 1 to each sink-edge coming from an original vertex of max degree. This is illustrated in Figure 3.

In this example, the shaded vertices have max degree in the original graph, so we've added min flow requirements of 1 to their source/sink edges. Assuming we can find a flow that satisfies the min flow requirements, that flow will give us a max degree matching.

Of course, this just raises the question of how one finds flows that satisfy minimum flow requirements. It turns out that there is a general way to

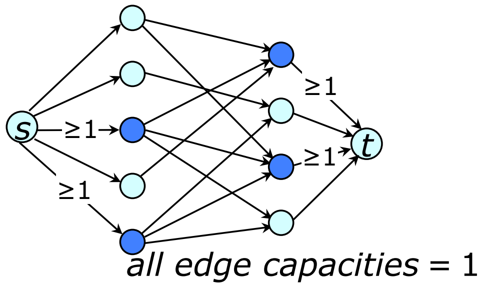

Figure 3: Finding max degree matching using minimum flow requirements

do this that can be applied to any flow graph, not just those constructed from bipartite graphs. This is done by converting our original problem to a related ordinary max flow problem. The basic idea is shown in Figure 4.

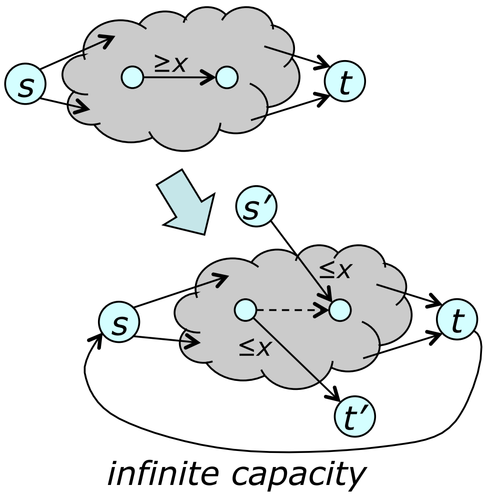

Figure 4: Finding a flow that satisfies min flow requirements

We construct the new graph by adding an infinite capacity edge from the sink to the source, then we add a new source $s'$ and a new sink $t'$ (in the new graph, the original source and sink are ordinary vertices that must satisfy the flow conservation property). Next, we replace every edge $(u, v)$ in the original graph that has a min flow requirement of $x$ with two edges $(s', v)$ and $(u, t')$, both with capacity $x$. This new graph is just an ordinary flow graph, so we can find a maximum flow from $s'$ to $t'$ and if this max flow saturates all the edges leaving $s'$ (and entering $t'$) we can use it to construct a flow in the original graph that satisfies the min flow requirements. For each original edge with a minimum flow requirement, we make the flow equal to the min flow requirement. For each original edge with no min flow requirement, we just make the flow equal to the flow on the corresponding edge in the new graph. Note that this procedure only works if there is some flow that will satisfy the given set of minimum flow requirements. If there is not, then the max flow in the transformed graph will not saturate the edges leaving $s'$.

An example illustrating this method is given in Figure 5. Here, the orig-

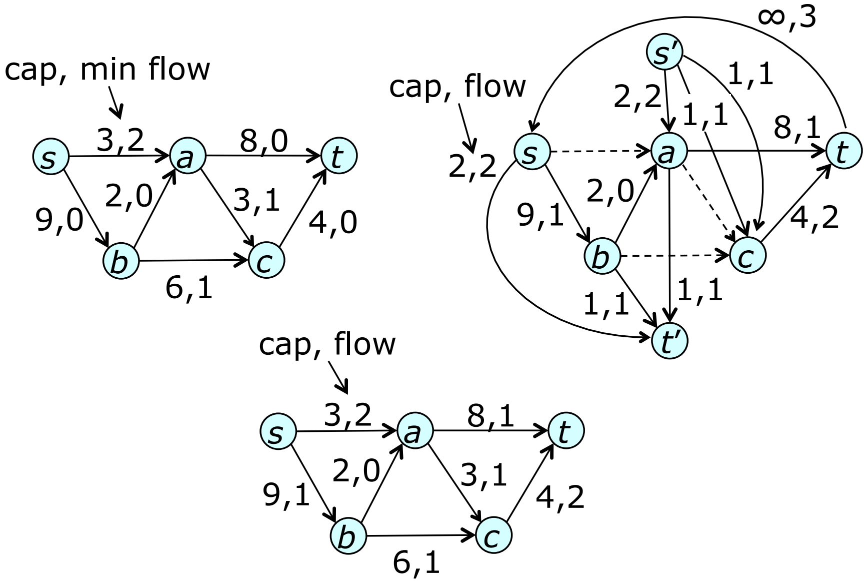

Figure 5: Example of finding a flow satisfying min flow requirements

inal graph with min flow requirements is shown at top left. The transformed graph with a maximum flow is shown at top right, and the resulting flows are shown at the bottom.

Note that while the flow produced in this way satisfies the min flow requirments, it is not a maximum flow in the original graph. To convert it to a maximum flow we can add more flow from $s$ to $t$ taking care not to reduce the flow on any of the edges with min flow requirements. This can be done by adjusting the definition of the residual capacity for these edges and proceeding in the normal way.

There is also another way to obtain a max flow. Let $f_1$ be the flow that satisfies the min flow requirements. For each edge $(u, v)$ in the original graph, reduce the capacity by $f_1(u, v)$. Now, find a max flow in this graph and call the resulting flow $f_2$. By adding $f_1$ and $f_2$ we get a maximum flow in the original graph that satisfies all the min flow requirements.

## 3 Matchings and the Traveling Salesman Problem

Matchings often arise in the context of other problems. In this section we'll look at two ways that matchings can be used to solve the *Traveling Salesman Problem* (TSP). In the traveling salesman problem, we are given a complete graph with edge costs $c(u, v) \geq 0$. Our objective is to find a cycle of minimum cost that includes every vertex exactly once.

There are several variants of this problem. In the most general case, edge costs are completely arbitrary and may even be asymmetric ($c(u, v) \neq c(v, u)$). A common special case requires that the costs satisfy the so-called *triangle inequality* which states that for any three vertices $u$, $v$ and $w$, $c(u, w) \leq c(u, v) + c(v, w)$. That is, the direct path from $u$ to $w$ cannot be more costly than any indirect path. The Euclidean TSP restricts the costs even further. In this case, the vertices are mapped to points in the plane and the costs are the corresponding Euclidean distances.

The problem is *NP*-hard, so there are no known algorithms that can find optimal solutions in polynomial time. Indeed the general version of the problem cannot even be approximated to within a constant factor by any polynomial time algorithm unless *P*=*NP*. However, if the costs are symmetric and satisfy the triangle inequality, we can produce solutions that are guaranteed to be reasonably close to optimal.

### 3.1 Approximating TSP with triangle inequality

First, notice that if we remove any edge from a TSP tour, the remaining path is a spanning tree. Hence, if $\mathit{TSP}(G)$ is the cost of an minimum cost TSP tour and $\mathit{MST}(G)$ is the cost of a minimum spanning tree, then $\mathit{MST}(G) \leq \mathit{TSP}(G)$. If we double every edge in the MST, we get an Eulerian graph in which every vertex has even degree. An Euler tour of this graph uses every edge exactly once and returns to the starting point. We can convert the Euler tour to a TSP tour by taking shortcuts to avoid repeating any vertex until we complete the cycle. Because the costs satisfy the triangle inequality, the resulting TSP tour cannot be more expensive than the Euler tour. Since the cost of the Euler tour is $2\mathit{MST}(G)$, the cost of the TSP tour is $\leq 2\mathit{TSP}(G)$.

Figure 6 shows an example of this. On the left is an MST with edges doubled. One way to construct an Euler tour is to do a depth-first tree

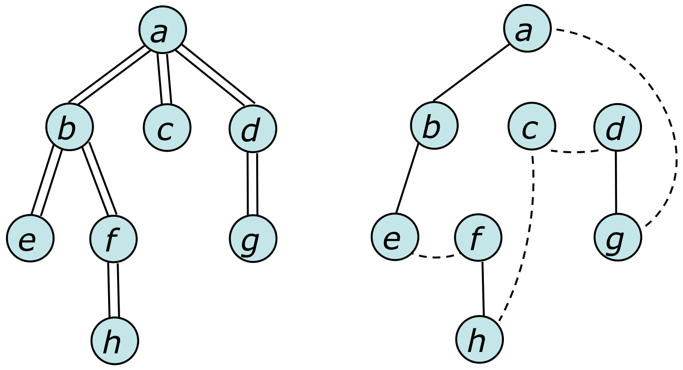

Figure 6: Example of constructing a TSP tour from a spanning tree

traversal starting at $a$ and listing each edge as we go down and up in the tree. In the example, this yields the tour $abebfhfbacadgda$. The right side of the figure shows the correponding TSP tour, where the dashed edges represent the shortcuts.

Now this is good, but we can do better. Recall that we doubled the edges in the MST in order to get an Eulerian graph. What if we could obtain an Eulerian graph without adding so many additional edges? Then, we could convert this Eulerian graph to a TSP tour with potentially lower cost. So, how can we add edges to an MST to make it Eulerian, without doubling the edges in the MST? The answer is to construct a matching.

In this case, we start by taking the vertices that have odd degree in the MST and then define a complete graph $K$ on just these vertices, with costs equal to the original edge costs. Note that any graph has an even number of odd-degree vertices, so $K$ contains a perfect matching. Now if we add the edges in a perfect matching on $K$ to the MST, we get an Eulerian graph in which every vertex has an even number of edges. If we pick a *minimum weight* perfect matching in $K$, we would expect the resulting Eulerian graph to have a significantly lower cost than the doubled MST, but how much better is it?

This question can be answered by looking at any TSP tour and extracting from that tour the vertices that have odd degree in the MST, while retaining the order of those vertices. This results in a cycle, which by the triangle inequality, has a cost no larger than that of the complete TSP tour. Moreover, that cycle can be divided into two perfect matchings on $K$. The lower weight matching has a cost that's no more than half the cost of the optimal TSP tour. So by combining a min weight perfect matching in $K$

with the MST, we get an Eulerian tour with a cost no more than $1.5TSP(G)$. We can convert this to a TSP tour by taking shortcuts, yielding a TSP tour with a cost no more than $1.5TSP(G)$. This method is known Christofides' algorithm.

## 3.2 Solving asymmetric TSP with random costs

We noted earlier that for the asymmetric version of the traveling salesman problem, there is no algorithm that can produce solutions that are guaranteed to be within a constant bound of the optimal value unless $P{=}NP$. However, this does not mean that the situation is totally hopeless. If we consider instances of the problem in which edge costs are chosen randomly over a fixed range (say 0 to 1), then there is an algorithm due to Karp, that with high probability will produce TSP tours that are close to optimal.

Karp's algorithm exploits a close relationship between the TSP and the weighted bipartite matching problem. This relationship is illustrated in Figure 7 The left side of the figure shows a TSP tour in a complete graph.

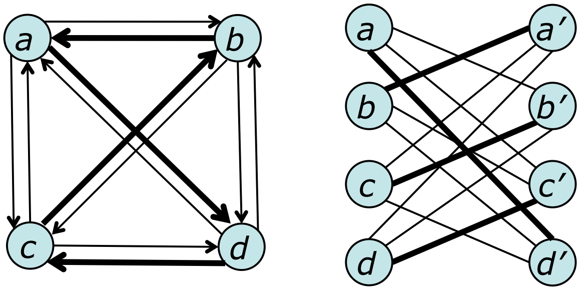

Figure 7: Converting a tour to a matching

In the bipartite graph on the right, we have two vertices $u$ and $u'$ for each original vertex $u$ and an edge $(u, v')$ for each original edge $(u, v)$. The edges in the new graph are assigned weights equal to the edge costs in the original graph. Notice that the edges in the tour correspond to a perfect matching in the new graph, and the total weight of this matching is equal to the cost of the TSP tour. If $MATCH(G)$ is the total weight of a *minimum weight perfect matching* for the bipartite graph derived from a given TSP graph $G$, then this implies that $MATCH(G) \leq TSP(G)$.

Now, not every perfect matching in the graph constructed from a TSP graph corresponds to a TSP tour. In Figue 8, we see an example of a matching that corresponds to a pair of separate cycles in the original TSP graph. In general, any perfect matching in the derived graph will correspond

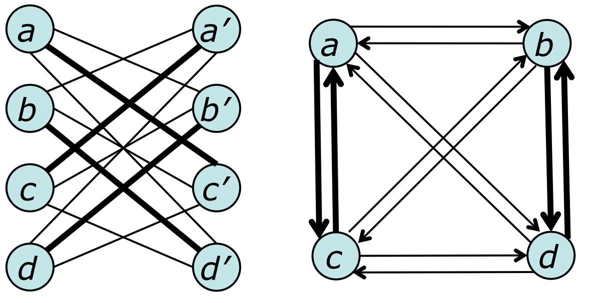

Figure 8: Converting a matching to a set of cycles

to a set of cycles in the TSP graph. If we can convert such a set of cycles into a single cycle, without increasing their cost by too much, we can obtain a near-optimal TSP tour.

Karp's algorithm does this by "patching" pairs of cycles to create a larger cycle. If $(u, v)$ is an edge in one cycle and $(x, y)$ is an edge in another cycle, we can combine the two cycles by replacing these two edges with the edges $(u, y)$ and $(x, v)$ as illustrated in Figure 9. This change increases the overall

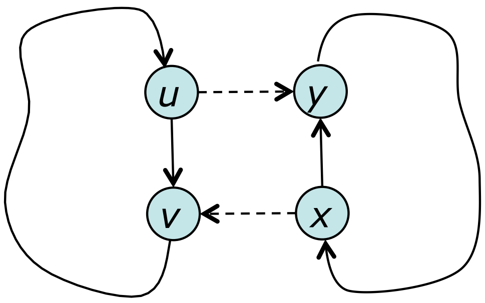

Figure 9: Patching a pair of cycles

cost by

$$c(u, y) + c(x, v) - (c(u, v) - c(x, y))$$

We can now give a complete description of Karp's algorithm. First, we construct a bipartite graph corresponding to the given TSP graph and compute a minimum weight perfect matching in that graph. This can be done use a variant of the min-cost flow method, we've used to find maximum weight matchings. Next, we find the cycles in the TSP tour defined by the matching edges. Finally, we patch pairs of cycles until there is just one cycle left. When patching a pair of cycles, we select the edges $(u, v)$ and $(x, y)$ that produce the smallest increase in cost for that pair.

Karp showed that when the edge costs are chosen randomly that with high probability, the initial set of cycles contains one large cycle containing most of the vertices, and a relatively small number of additional cycles. Consequently, the number of patching operations required to produce a TSP tour is relatively small, and if we always combine a small cycle with the large one, the overall increase in cost is also small.