Finding a matching is a common task in computing that arises in physical and operational problems. If you have a set of possible connections between two groups of objects, you would like to know if there is a way to match each element from the first group with exactly one element of the second group. It is like arranging marriages between two groups or assigning a group of tasks to a group of people, where some connections are possible and some are not.

It might be impossible to find a matching that uses all the people or objects, given the connections that are possible. A maximal matching tries to maximize the number of connections in the matching. A perfect matching is one in which you use all possible objects. Sometimes weights are attached to connections (in this case a maximal matching optimizes the sum of the weights), but we're not going to do that here. We will use interchangeably the terms objects and nodes for the objects, assignments, or people mentioned above. Similarly we will use the terms edges and connections interchangeably.

Figure 1 shows two groups of nodes: we will call the left four nodes the boys and the right four nodes the girl nodes. The black lines are the possible connections, and the blue lines select some black lines that give a 1-1 correspondence between the two sets, namely, a matching, what we would like to find. Figure 1 is an example of a bipartite graph. This means

- it is a set of nodes and edges,
- the nodes are of two types, and
- there are no edges between nodes of the same type.

Figure 2 shows an incidence matrix for Figure 1. The rows represent girl nodes; the columns, boy nodes. There is a 1 in position \((i, j)\) if there is an edge between girl \(i\) and boy \(j\). The circled elements show the edges that appear in blue in the perfect matching in Figure 1. In matrix form a perfect matching is a path of 1s through the matrix with a single 1 in each row and each column, just as in Figure 2.

We can pose several problems about matchings between two groups of objects:

1. Is there a matching?
2. Find a matching.
3. Count the number of matchings.
4. Find all the matchings.

We will show you how to do the first two problems.

A natural approach might be to imitate the data motion of Gaussian elimination. Examine the matrix in Figure 2 and exchange rows and columns to put a 1 in position \((1, 1)\). In the case presented here, there already is a 1 in position \((1, 1)\). Row 1 and column 1 are then "frozen" and we examine the remaining rows and columns to try to put a 1 in position \((2, 2)\). If we are able to continue in this way all the way down to position \((n, n)\), we can read off the path by reversing the sequence of row and column exchanges.

However, in many cases we won't make it all the way; we'll get stuck at some row \(k\) because there are no available 1's in column \(k\). At this point we could backtrack to the first \(j < k\) with more than one choice below position \((j, j)\). If we get stuck again, we must backtrack to some row \(i < j\).

Eventually, of course, the simple backtrack approach will find a path if one exists, but it might take many, many backtracks. A more efficient method of backtracking was devised by J.E. Hopcroft and R.M. Karp.

Figure 2. A sample bipartite graph. The blue lines show a perfect matching.
proving that until a maximal path is found, augmenting paths are always available. In fact, Hopcroft and Karp even determine lower bounds on the number of augmenting paths available. Their result says that for a path of length $s$, at most $2\sqrt{s} + 2$ remakes of the graph will be necessary.

Applications

How is this useful, you may ask. The most obvious applications are scheduling and logistics problems, where this first arose. Two examples from physics are the dimer covering problem and the more general monomer/dimer problem that ask for the number of matchings. Here's how they are related to matchings.

The dimer covering problem (in 2D) asks for the number of ways to cover a two-dimensional grid with dimers. Figure 3a shows a $4 \times 4$ grid in 2D. Figure 3b shows one covering with dimers. How many ways are there to do this? We construct an incidence matrix by having the rows represent red squares and the columns represent black squares. There is a 1 in position $(i, j)$ if red square number $i$ has black square number $j$ as a neighbor. Otherwise, there is a 0. We assume periodic boundary conditions, that is, at the end of a row (column) we assume the beginning of the row (column) wraps around. The matrix corresponding to Figure 3 is given in Figure 4.

There are four 1's in every row and column because each square has a neighbor on the north, south, east, and west side. In 3D, there would be six 1's in each row and column because in addition to north, south, east, and west there are also up and down. So, the dimer covering problem translates into finding the number of paths through these incidence matrices. This, of course, is an instance of problem 3, but all current approaches to doing it are based on statistical sampling of paths, that is, based on many instances of match finding.

Is there a matching?

Here's a quick answer to the first problem, that is, deciding if there is any path though the matrix. Suppose you have an incidence matrix $A$. If you multiply $A$ element-wise by random numbers

$$C = A \cdot \text{rand}(n, n)$$

(where $n$ is the dimension of the matrix, $\text{rand}(n, n)$ is a random $n \times n$ matrix, and $\cdot\cdot$ is element-wise multiplication) and then evaluate the determinant, a nonzero determinant means there must be a path and, if there is a path, there will be a nonzero determinant with probability 1.

Figure 2. An incidence matrix for Figure 1. The circled elements form a path through the matrix, a perfect matching.

Figure 3. Example application. Figure 3b is a dimer covering of Figure 3a. How many ways are there to fit dimers over the grid?
positive for $\sigma^+$ positive and negative for $\sigma^-$ negative. Therefore, since the $a_{i,0(j)}$ are all 0 or 1 (an incidence matrix), none of them can be 0 in this particular term. Thus the $a_{i,0(j)}$ form a path through the matrix, that is, a matching.

Conversely, if there is a path, then for some permutation $\sigma,$

$$\prod_{j} r_{i,0(j)}^{a_{i,0(j)}} \neq 0.$$  

With probability 1 it won't cancel with any other term because we would need that

$$\sum_{\sigma} \prod_{j} r_{i,0(j)}^{a_{i,0(j)}} = \sum_{\sigma} \prod_{j} r_{i,0(j)}.$$ 

This won't happen for random $r_{ij}$ unless you are really unlucky. If you think you are really unlucky, do the test twice.

To determine quickly if there is a path going through any particular edge, in the same spirit we can form the matrix $C.$ Let $A$ as above and $\text{inv}$ is the inverse. Look at position $(i,j)$ in the resulting matrix. By Cramer's rule this value will be

$$(-1)^{i+j} \frac{\det(C_{i,j})}{\det(C)},$$ 

and, if this term is nonzero, then there is a path through the $(i,j)$ element of $A.$ So we get the answer to question 2: a matching exists through position $(i,j).$

Find a matching

Actually finding a matching is harder than merely determining that there is one. It is partially a backtracking problem like the eight queens problem. (You may recall a backtracking algorithm for finding connected components in the Fall 1996 Computing Prescriptions column.) Here we present the pseudocode for the Hopcroft-Karp algorithm for finding a matching. If a perfect matching isn't possible, the algorithm will get one of maximal length.

The input matrix is $A.$ We will let the graph $G$ correspond to $A.$ $M$ represents the "current" matching, and it will eventually hold the answer. $M$ begins by being a null matching. The algorithm works on a graph $G'$ related to $G,$ the actual graph. Here $G'$ has the same vertices as $G$ but adds a source (SRC) and
a sink (SNK). The rows of $A$ are considered girl vertices and the columns are boy vertices. The edges of $G'$ are the edges of $G$ directed from a boy to a girl if the edge is in $M$ and from a girl to a boy if the edge is not in $M$. There are edges from SRC to every girl not in $M$ and from every boy not in $M$ to SNK. Figure 5 shows an example of an $M$, $G$, and the corresponding $G'$.

The algorithm starts by finding a quick and dirty partial matching. This is the first time through the "while not done" loop. You start from the source and, because initially all edges go from girls to boys and every boy vertex connects to the sink, your path is SRC, g, b, SNK. You do this as much as possible without repeating vertices. For example, in the matrix $A$ in Figure 5, the circled elements might be selected first as a partial matching after a first pass.

The idea is that at any stage you have a partial matching, $M$. You try to get an augmenting path by reversing the arrows on any edges actually in the partial matching and removing edges that connect SRC and SNK to the vertices in the partial matching. The augmenting path is a path from SRC to SNK which is a set of edges that are alternatively in $M$ and not in $M$.

Figure 6a is an example of an augmenting path. To get the desired new matching (that is, a matching of length greater than the old matching), take the symmetric difference, $\delta$, between the old matching, $M$, and the augmenting path. This will always yield a new matching with more elements than the previous one. Figure 6b is an example of getting the new matching from the old matching and the augmenting path.

So starting from a naive first matching $M_0$, you find an augmenting path $P_0$ and let $M_1 = M_0 \oplus P_0$. Similarly, given $M_i$ find an augmenting path $P_i$ (if possible) and let $M_{i+1} = M_i \oplus P_i$. Continue doing this until there are no more augmenting paths. That's it.

The augmenting path is found using a depth-first search for SNK starting at SRC. Follow a path in the graph as far as you can. Mark the vertices and edges visited so you don't use them again and back up to a spot where there were other choices of paths and follow one of them. Keep doing this until you reach SNK. Then go back to SRC and if there are still paths available, find SNK again. Every path found in this way augments the previous matching $M$. Keep doing this until you have exhausted all the edges out of SRC. You then remake the graph and try to find another augmenting path. You might never make it from SRC to SNK; in that case, there is no augmenting path and you already have a maximal matching.

One subtle but important point may help in understanding the pseudocode: during any set of depth-first searches, the graph does not change except for marking some edges as used. After a graph is completely exhausted and we have found a set of augmenting paths and taken the symmetric difference, we remake the graph by reversing arrows and find a new set of augmenting paths. This remaking of the graph can be thought of as a major step or outer iteration. The Hopcroft-Karp algorithm bounds the number of times we will have to do this. Ultimately we get a graph for which the depth-first search never gets from SRC to SNK. Then we are done.

Example

To begin the algorithm, start with all the arrows in $G'$ going from girls to boys, with edges from SRC to every girl and with edges from every boy to SNK. Try to find a path from SRC to SNK by trying edges that are available out of SRC. Suppose in the example that we picked edge (SRC, g1). We remove (SRC, g1) from $G'$ and mark g1 as having been seen and continue to b1, removing (g1, b1) from $G'$ and then to (b1, SNK). On encountering SNK, back up to SRC and add the path just traversed to the matching. Then try to add more paths out of SRC to SNK and not using vertices used before. These are "disjoint" from the paths added to the matching already.

However, as seen in the example, you may get stuck. This

---

Actually finding a matching is harder than merely determining that there is one. It is partially a backtrack problem like the eight queens problem.

---

Figure 6. (a) Sample of augmenting path $P$ for Figure 5. (b) Result of taking symmetric difference of $M$ and $P$ for Figures 5 and 6a.
We are given an \( n \times n \) matrix \( A \) for which we wish to find a maximal matching. Let \( G \) be the graph corresponding to the matrix \( A \). \( A \) represents the "current" matching, the maximal matching we have found so far. \( \text{STACK} \) is a stack; the operations \( \text{POP} \) and \( \text{PUSH} \) refer to \( \text{STACK} \). \( \text{TOP} \) is the top element (most recently pushed, first to be popped) of \( \text{STACK} \). \( B \) is an array with \( 2N+2 \) elements. \( G' \) is a directed graph on \( 2N+2 \) vertices.

\[
\begin{align*}
\text{Begin:} \\
M & \leftarrow \text{the empty matching} \\
\text{DONE} & \leftarrow \text{false} \\
\text{while (not DONE) do:} \\
\text{Create } G': \text{ Vertices are the vertices of } G, \text{ adjoining a source and} \\
\text{ a sink. Edges are the edges in } G, \text{ directed from boy to girl} \\
\text{ if the edge is in } M, \text{ otherwise directed from girl to boy.} \text{ Also,} \\
\text{ edges are added from source to girls not in } M, \text{ and from boys} \\
\text{ not in } M \text{ to sink. Note that this } G' \text{ remains fixed for each} \\
\text{ iteration of the loop, except for deleting edges.} \\
\end{align*}
\]

\[
\begin{align*}
B[i] & \leftarrow 0 \text{ for all } i \text{ in bounds} \\
\text{STACK} & \leftarrow \text{the empty stack} \\
\text{DONE} & \leftarrow \text{true} \\
\text{PUSH(source)} \\
\text{while STACK is nonempty do:} \\
\text{while some edge from TOP exists in } G' \text{ do:} \\
\text{FIRST} & \leftarrow \text{a vertex such that (TOP, FIRST) in } G' \\
\text{Remove (TOP, FIRST) from } G' \\
\text{if (FIRST = sink) then:} \\
\text{while at least two items in STACK do:} \\
\text{BOY} & \leftarrow \text{POP} \\
\text{GIRL} & \leftarrow \text{POP} \\
\text{Remove any edge including GIRL from } M \\
\text{Add (GIRL, BOY) to } M \\
\text{DONE} & \leftarrow \text{false} \\
\text{end while} \\
\text{end if} \\
\text{if (B[FIRST] = 0) then do:} \\
B[FIRST] & \leftarrow \text{1} \\
\text{PUSH(FIRST)} \\
\text{end if} \\
\text{end while} \\
\text{POP} \\
\text{end while STACK nonempty} \\
\text{end while not DONE} \\
\end{align*}
\]

\[
\text{Return } (M) \\
\text{End.}
\]

**Figure 7. Pseudocode for the Hopcroft Karp algorithm—
finding the maximum matching.**

is the situation in Figure 5. Then look for an augmenting path by remaking \( G' \), marking all the vertices as unseen, turning the arrows of the edges that are in the partial matching connecting SRC and SNK only to the boys and girls that are not in the matching, and starting from SRC again. When you find a path from SRC to SNK, compute the symmetric difference with the matching that you already have, and make the result the new matching. So in Figure 5, on the next pass, you would go from SRC to g3 to b1 to g1 to b3 to SNK—this is an augmenting path to the previous matching. Taking the symmetric difference you get the edges (g3, b1), (g1, b3), and (g2, b2). In practice, if you want take the symmetric difference of (g, b) with the matching, you remove any edge containing g from the matching and add (g, b).

Finding an augmenting path always gets you a matching with at least one more edge than you previously had because,
by turning the arrow around, you are forced to add a (girl, boy) edge at the beginning of the new path that is not in the existing partial matching. By beginning at SRC, you always connect SRC to a girl first and all the existing edges would start with a boy. Likewise at the end of the path you must always have another edge not in the matching already, because to get to SNK you would have to come from a boy, and the only edges that end with boys are those not in the existing matching. So if there are r edges in the existing partial matching that end up in the augmenting path, then there must be connected by r – 1 new edges plus the two new ones at the beginning and the end, so you get (r + 1) new edges in the augmenting path.

Pseudocode

Now let us consider the pseudocode in Figure 7. The outside loop “while not done” keeps iterating as long as there are paths to explore out of SRC. It always begins by PUSH(source), putting source on a stack. If you ever get a perfect matching, then on the next iteration SRC is connected to no vertex. So, when processing the stack on the next line, there is no edge from TOP (SRC) in G'. The source is popped, the stack is empty, done is true, and you end.

Obviously, a perfect matching is impossible if you have either a row or column of 0's in the original matrix. The general case of when perfect matchings are possible would take too long to explain. This is why the pseudocode ends if a perfect matching is not possible. Suppose that you already have a maximal matching. If a row is all 0's, then there will be a girl vertex with no edge coming out of it. SRC will be connected to the vertex but no path comes out. In this case, after SRC is on the stack, and the lone girl vertex is visited and found to be a dead end, the “while some edge from TOP exists” loop will end, that vertex is popped, and we are back to the case of the perfect matching—that is, SRC is popped and the program ends. When you have a column of 0's, there is a vertex that never connects to the rest of the graph. In this case the algorithm tries to connect to this last vertex, but since this is impossible it ends at a dead end, and you pop back to SRC. At this point you have explored all paths out of SRC, so you pop SRC and end, with done equal to true.

Matrix interpretation

It is instructive to see what this looks like from the point of view of a matrix instead of a graph. Figure 8 shows the partial matching of Figure 5 as starred entries. It is “stuck” because there is no other edge (a matrix entry 1) you could add to it. We add an extra column to represent SRC's possible connections with the rows (girls), and we add an extra row to represent SNK's connections with the columns (boys). The augmenting path represented by dotted lines is found as follows.

Go to a row that has not yet been used as part of the matching. You find this by going to the SRC column and finding a 1. (In this case it is only row 3.) Find an unused element (that is, not starred). If there is none, then you must have a row of 0's and there is no matching. We pick the only element (3, 1). Then look in the column of that element (column 1) for an entry that is already in the matching; there is only 1, the (1, 1) element. Then look in that row for an unused element. In this case, the (1, 3) entry is available. Look in column 3 for a starred entry. There is none except for the 1 in the SNK row. This indicates the end of the augmenting path. If there is no new entry in row 1, then this path doesn't work, mark the elements as “visited,” and go back to the last element along the path where there was another choice that has not been visited before. This is the backtracking.

To take the symmetric difference as a matrix, add the elements from the augmenting path that are not in the matching, throw out the elements from the augmenting path that have stars, and keep any starred elements that are not in the augmenting path. Figure 9 is a more complicated example of how the algorithm would find an augmenting path.

Finding an augmenting path always gets you a matching with at least one more edge than you previously had.

Reference


Isabel Beichl is a mathematician in the Computing and Applied Mathematics Laboratory at NIST, Gaithersburg, Md.; e-mail, isabe1@cam.nist.gov. Francis Sullivan is associate editor-in-chief of IEEE CS&E and director of the IDA Center for Computing Sciences, Bowie, Md.; e-mail, fran@super.org.