

14 Depth First Search

In this chapter we will see our first major graph algorithm and explore the surprisingly large number of problems that it can be applied to.

14.1 Connectivity

Recall from Chapter 13 that we say two vertices u and v in a graph G are connected if G contains a path from u to v . We say that G itself is connected if every pair of vertices in G is connected (i.e. if G has only one connected component).

One of the most natural questions about a vertex is whether two vertices are connected. For example, in a graph representing a transit network, we may want to determine whether it's possible to get from point A to point B. In a graph representing genealogies, we may want to determine whether two people are related, i.e. whether they are connected through family links that we can trace.

So how can we determine if two vertices a and b are connected? We can try to solve this problem recursively, using the techniques from Chapter 10. If a and b are the same vertex, then they are connected. Otherwise, a is connected to b if and only if, one of a 's neighbors is connected to b . Based on this we could give the following algorithm:

WANDERAROUND(a, b)
If $a = b$:
 Return True
For each vertex $c \in N(a)$:
 If WANDERAROUND(c, b):
 Return True
Return False

If WANDERAROUND returns True, then we know that a and b are connected, and if it returns False, we know that a and b are not connected. This algorithm has a very serious correctness issue, however. Do you see what it is? (Think back to Chapter 7)

The problem is that returning True and returning False are not the only things that this algorithm can do. It can loop forever! Nothing in the algorithm prevents us from wandering back to vertices that we've already visited, so we can wander from a vertex u to a vertex v , and then wander from v right back to u . In fact, if a and b are not connected, but a has at least one neighbor, then WANDERAROUND is guaranteed to get caught in an infinite loop, wandering through the same vertices over and over without ever finding b .

To fix this, we need some way to keep track of which vertices we've already been to so that we don't visit them again. If we continue to follow our standard recursive backtracking model, we could handle this by modifying our recursive function to take a list of all the vertices we've already used on the current path as an additional parameter (i.e. pass along a summary of our previous choices), as shown below:

WANDERBETTER($a, b, S = \emptyset$)
If $a = b$:
 Return True
Add a to S
For each vertex $c \in N(a) \setminus S$:
 If WANDERBETTER(c, b, S):
 Return True
Return False

Here S is the set of all the previous vertices on our path from a . WANDERBETTER always terminates, which is an improvement over WANDERAROUND, but it's runtime is quite bad. The number of subsets of V (i.e. $|2^V|$) is exponential in n . If G is a complete graph, then we can visit the vertices in any order, so we do end up with exponentially many subproblems. Dynamic programming

can't help us here either, since each subproblem has different parameters (in particular a different set S).

A better approach is to use a global array to keep track of all of the vertices that have been visited by any of our recursive calls. We'll assume here and for the rest of the chapter that $V = \{1, \dots, n\}$. If this is not the case, we can assign each vertex a number from 1 to n arbitrarily and use these numbers as the indices into our global arrays.

```

visited ← an array of  $n$  copies of False
DEPTHFIRSTSEARCH( $v$ )
  If not visited[ $v$ ]:
    visited[ $v$ ] ← True
    For each vertex  $w \in N(v)$ :
      DEPTHFIRSTSEARCH( $w$ )

```

Theorem 66. $\text{DEPTHFIRSTSEARCH}(v)$ visits exactly the vertices which are connected to v . (i.e. when the algorithm terminates, $\text{visited}[x]$ is True if x is in v 's connected component, and False otherwise). Furthermore, $\text{DEPTHFIRSTSEARCH}(v)$ runs in time $\Theta(m' + n')$, where n' is the number of vertices in v 's connected component and m' is the number of edges between these vertices.

Proof. To show that $\text{DEPTHFIRSTSEARCH}(v)$ visits exactly the vertices connected to v we need to show that the algorithm visits every vertex connected to v and that every vertex the algorithm visits is connected to v . We can prove both directions by induction:

- First we prove that every vertex connected to v is visited by the algorithm. Let y be a vertex which is connected to v . Then there exists a $v \rightarrow y$ path. Let k be the length of this path. We will prove that $\text{DEPTHFIRSTSEARCH}(v)$ visits y by induction on k .
 - Base Case: If $k = 0$, then $y = v$, so $\text{DEPTHFIRSTSEARCH}(v)$ immediately visits y .
 - Induction Hypothesis: Suppose that for every $0 \leq \ell < k$, $\text{DEPTHFIRSTSEARCH}(v)$ visits all vertices which can be reached from v by a path of length ℓ .
 - Induction Step: Let $v_1, \dots, v_{k+1}$ be the length k path from v to y . Then v_k is connected to v by a path of length $k - 1$. By our IH, this means that $\text{DEPTHFIRSTSEARCH}(v)$ visits v_k . When this happens, the algorithm will recursively call DEPTHFIRSTSEARCH on each neighbor of v_k , including $v_{k+1} = y$, so y is visited.
- Now we prove that every vertex the algorithm visits is connected to v . We will prove by induction on i that the i 'th vertex the algorithm visits is connected to v .
 - Base Case: The first vertex the algorithm visits is v . v is connected to itself.
 - Induction Hypothesis: Suppose that the first $i - 1$ vertices the algorithm visits are all connected to v .
 - Induction Step: Every recursive call the algorithm makes is to a neighbor of a vertex it has already visited. Therefore, if y is the i 'th vertex visited, y is a neighbor of some vertex x which was visited earlier. By our IH, x is connected to v . Thus, since y is connected to x by the edge $\{x, y\}$, y is also connected to v .

Now we analyze the runtime of DEPTHFIRSTSEARCH . Let C be the subgraph of G consisting of the vertices in v 's connected component and the edges between them. As we showed above, $\text{DEPTHFIRSTSEARCH}(v)$ visits exactly the vertices in C . The first time that DEPTHFIRSTSEARCH is called on a vertex u , it performs $\Theta(\deg(u))$ work, outside of the recursive calls, since the algorithm iterates through the neighbors of u , which takes $\Theta(\deg(u))$ time with an adjacency list. Each subsequent call to DEPTHFIRSTSEARCH on the same vertex takes constant time, since DEPTHFIRSTSEARCH will note that the vertex has already been visited and immediately halt.

Thus, each vertex u is responsible for $\deg(u)$ recursive calls and the calls on that vertex take $\Theta(1 + \deg(u))$ total time. Therefore the total time across all recursive calls is $\Theta(\sum_{u \in C} 1 + \deg(u)) = \Theta(n' + \sum_{u \in C} \deg(u)) = \Theta(m' + n')$, by the handshake lemma. $\square$

From Theorem 66, it follows that we can use DEPTHFIRSTSEARCH to check whether two vertices a and b are connected by calling DEPTHFIRSTSEARCH(a) and then checking afterwards if visited[b] is true. Moreover, we can use DEPTHFIRSTSEARCH to find the entire connected component of a given vertex (it's just all the vertices visited by the DFS).

What if we want to find all of the connected components in a graph? In this case we can use multiple calls to depth first search. Every time the depth first search finishes, we restart from a new vertex that we haven't visited yet. The algorithm below implements this idea. We replace the “visited” array with a global array “component” to store an integer representing the component of each vertex (vertices whose number in this array are the same are part of the same component).

<pre> FINDCOMPONENTS($G = (V, E)$) component $\leftarrow$ an array of n 0s numComponents $\leftarrow$ 1 For each $v \in V$: If component[v] = 0: FINDCOMPONENTSHelper(v) numComponents $\leftarrow$ numComponents + 1 Return component FINDCOMPONENTSHelper(v) If component[v] = 0: component[v] $\leftarrow$ numComponents For each vertex $w \in N(v)$: FINDCOMPONENTSHelper(w) </pre>

Theorem 67. FINDCOMPONENTS(G) returns an array “component” such that for every pair of vertices $u, v \in V(G)$, component[u] = component[v] if and only if u and v are in the same connected component of G . Furthermore, FINDCOMPONENTS runs in $O(m + n)$ time.

Proof. We note that FINDCOMPONENTSHelper runs DEPTHFIRSTSEARCH except that instead of marking vertices as visited, it labels them with a component number. By the same arguments we used in proving Theorem 66, the first call that FINDCOMPONENTS makes to FINDCOMPONENTSHelper will find a component and label all of its vertices with component 1. Then the algorithm will find a vertex not part of this first component (i.e. one that still has a component label of 0) and run DFS from it, this time labeling each visited vertex with component 2. This will continue until all vertices are labeled with their component.

By Theorem 66, each call that FINDCOMPONENTS makes to FINDCOMPONENTSHelper runs in time $\Theta(m' + n')$, where n' is the number of vertices in the component found and m' is the number of edges. Since each vertex and each edge is in exactly one component, the total time for these calls across all components is $\Theta(m + n)$. The work of FINDCOMPONENTS outside of the recursive calls can all be done in $O(n)$ time, so the total runtime is $\Theta(m + n)$. $\square$

So far we have discussed DFS for undirected graphs, but DFS is also useful for directed graphs. For a vertex v in a directed graph $G = (V, E)$, we define $\text{reach}(v) := \{w \in V \mid G \text{ contains a } v \rightarrow w \text{ path}\}$ to be the set of all vertices reachable from v (Note that these vertices may not have paths back to v , so this is not a connected component). We can compute $\text{reach}(v)$ in linear time using the directed version of DFS given below.

<pre> visited $\leftarrow$ an array of n copies of False DIRECTEDDFS(v) If not visited[v]: visited[v] $\leftarrow$ True For each vertex $w \in N_{out}(v)$: DIRECTEDDFS(w) </pre>

Theorem 68. DIRECTEDDFS(v) visits exactly the vertices in $\text{reach}(v)$ in $O(m + n)$ time.

Proof. The proof is basically the same as the one we used for Theorem 66. We can prove that every vertex $w \in \text{reach}(v)$ is visited by induction on the length of the $v \rightarrow w$ path. We can prove that only vertices in $\text{reach}(v)$ are visited by induction on the number of recursive calls (visiting an out-neighbor of a vertex in $\text{reach}(v)$ always gives us another vertex in $\text{reach}(v)$).

Since our DFS visits at most all n vertices and explores at most all m edges, the runtime is $O(m + n)$. $\square$

Finding strongly connected components in a directed graph is a bit more complicated, we will discuss this problem in Section 14.4.

14.2 Finding Cycles

Recall from Chapter 13 that a cyclic graph is one which contains at least one cycle, while an acyclic graph is one which does not contain any cycles. Determining whether or not a graph is acyclic is a common problem which has several applications. For example, we know that a tree is connected and acyclic. We may want to check if a graph has these properties to determine if it is a tree. Directed acyclic graphs are useful for representing dependencies (e.g. between tasks). If we have a set of tasks to complete, some of which depend on other tasks being completed first, we would like to make sure that there are no circular dependencies, i.e. that there is a valid order to complete the tasks in.

We can detect cycles with DFS. We present the algorithm here for directed graphs, but a similar approach also works for undirected graphs. We use a second array “onPath” to keep track of the vertices on the path that we followed to reach the vertex we are currently visiting. Then, if we ever see an edge back to an earlier vertex on the path, we know that G has a cycle.

```

CYCLEDETECTOR( $G = (V, E)$ )
    visited  $\leftarrow$  an array of  $n$  copies of False
    onPath  $\leftarrow$  an array of  $n$  copies of False
    For each  $v \in V$ :
        If CYCLEDETECTORHELPER( $v$ ):
            Return True
    Return False

CYCLEDETECTORHELPER( $v$ )
    If not visited[ $v$ ]:
        visited[ $v$ ]  $\leftarrow$  True
        onPath[ $v$ ]  $\leftarrow$  True
        For each vertex  $w \in N_{out}(v)$ :
            If onPath[ $w$ ]:
                Return True
            Else if CYCLEDETECTORHELPER( $w$ ):
                Return True
        onPath[ $v$ ]  $\leftarrow$  False
    Return False

```

Theorem 69. $\text{CYCLEDETECTOR}(G)$ returns True if and only if G has a cycle. Furthermore, $\text{CYCLEDETECTOR}(G)$ runs in $O(m + n)$ time.

Proof. We need to prove two directions.

- First we show that if G has a cycle then $\text{CYCLEDETECTOR}(G)$ returns True. Let C be a cycle in G . Let v be the first vertex of C that we call $\text{CYCLEDETECTORHELPER}$ on. When we visit v , we add it to our current path, setting $\text{onPath}[v]$ to True. We will only set $\text{onPath}[v]$ back to False after we have fully explored all the out-neighbors of v . By Theorem 68, since $\text{CYCLEDETECTORHELPER}$ is performing a depth-first search, we know that it will visit all of the vertices in $\text{reach}(v)$ before the call to $\text{CYCLEDETECTORHELPER}(v)$ returns (unless it returns True early by finding a different cycle). In particular, if it has not already returned True, $\text{CYCLEDETECTORHELPER}$ will visit the vertex u that precedes v in the cycle.

Thus, we will visit u while $\text{onPath}[v]$ is True. At this point, `CYCLEDETECTORHELPER` will see the edge (u, v) and return True.

- Now we will show that if `CYCLEDETECTOR`(G) returns True, then G has a cycle. The key observation is that the vertices for which onPath is true always form a path ending at the vertex we are currently visiting (every time we visit a new vertex we add it to our path, every time we backtrack, we remove the vertex we left from our path). Thus, if we ever encounter an edge from the vertex v that we are visiting to a vertex w with $\text{onPath}[w]$ True, we can form a cycle in G by following the (v, w) edge, then following the path that our recursive calls used to reach v from w .

The runtime analysis for `CYCLEDETECTOR` is similar to the analysis we did when proving Theorem 67. `CYCLEDETECTOR` is simply running DFS from all vertices with a constant amount of additional work per vertex to update the onPath array. Since the visited array is shared across all runs of DFS, the total amount of work done per vertex v is $\Theta(1 + \deg(v))$, so the total time for all the runs of DFS is only $O(m + n)$. $\square$

14.3 Topological Sort

In the previous section we gave the example of task dependencies as an application for checking if a graph is acyclic. When we have a dependency graph, however, often we don't just want to know that it is acyclic, we also want to find an order in which we can perform the tasks that satisfies all of the dependencies.

For example, given a list of packages, we might want to find an order to install them such that when we try to install a package we've already installed the packages it requires. Or given a list of courses we might want to find an order to take them in such that before we take a course we've taken its prerequisites. Both of these are examples of the topological sorting problem.

Formally, we define a topological sort of a directed acyclic graph $G = (V, E)$ as an ordering (permutation) of the vertices of V such that for every edge $(u, v) \in E$, u occurs before v in the ordering. Note that only an acyclic graph can have a topological sort, since no matter how we order a cycle, the vertex we put last will have an edge to one of the earlier vertices.

We can find a topological sort using depth first search by simply modifying our search algorithm to add a vertex to the beginning of our ordering after the algorithm has completely finished exploring the vertex.

```

TOPSORT( $G = (V, E)$ )
   $R \leftarrow$  an empty list
  visited  $\leftarrow$  an array of  $n$  copies of False
  For each  $v \in V$ :
    TOPSORTHELPER( $v$ )
  Return  $R$ 

TOPSORTHELPER( $v$ )
  If not visited[ $v$ ]:
    visited[ $v$ ]  $\leftarrow$  True
    For each vertex  $w \in N_{out}(v)$ :
      TOPSORTHELPER( $w$ )
    Prepend  $v$  to the beginning of  $R$ 

```

Theorem 70. *For every directed acyclic graph $G = (V, E)$, `TOPSORT`(G) returns a topological sort of G .*

Proof. Note that `TOPSORTHELPER`(v) adds v to R the first time it is called. Subsequent calls to `TOPSORTHELPER`(v) do not change R , because v is already marked as visited. Thus, every unique vertex we visit with `TOPSORTHELPER` gets added to R exactly once. Since `TOPSORT` calls `TOPSORTHELPER` on every vertex, this means that when `TOPSORT`(G) returns, R is an ordered list of the vertices of G where each vertex appears exactly once.

It remains to show that this ordering is actually a topological sort, i.e. that for every edge (u, v) , u occurs before v in R . Let $(u, v) \in E$. Since G is acyclic, we know that G does not contain a $v \rightarrow u$ path (because such a path plus the (u, v) edge would form a cycle). Thus, $u \notin \text{reach}(v)$. We consider two cases:

- First suppose that $\text{TOPSORTHELPER}(v)$ is called before $\text{TOPSORTHELPER}(u)$. Since TOPSORTHELPER is just performing DFS with the additional step of adding v to R at the end, by Theorem 68, the recursive calls made by $\text{TOPSORTHELPER}(v)$ will visit only vertices in $\text{reach}(v)$. But as we noted above, $u \notin \text{reach}(v)$. Thus, when the for loop in $\text{TOPSORTHELPER}(v)$ ends, u will still be unvisited, and therefore will not be in R . Thus, v will be prepended to R first, meaning that in the final vertex ordering, u occurs before v .
- Now suppose that $\text{TOPSORTHELPER}(u)$ is called before $\text{TOPSORTHELPER}(v)$. In this case, $\text{TOPSORTHELPER}(u)$ will recursively call $\text{TOPSORTHELPER}(v)$. Thus, the for loop in $\text{TOPSORTHELPER}(u)$ will not end until after the first call to $\text{TOPSORTHELPER}(v)$ has returned and v has been added to R . Thus, u will be prepended to R later than v , and thus u will occur before v in the final ordering.

In either case, u occurs before v in R . Thus, every edge of G has its endpoints appear in the correct order in R , so R is a topological sort. □

14.4 Strongly Connected Components

Recall that two vertices u and v in a directed graph G are strongly connected if G has both a $u \rightarrow v$ path and a $v \rightarrow u$ path. A strongly connected component is a maximal set of vertices that are all strongly connected to each other.

We said in section 14.1 that while DFS can compute the connected component of a vertex in an undirected graph, it does not compute the strongly connected component of a vertex in a directed graph. For example if G consists of just a long path, then running DIRECTEDDFS from the first vertex s in the path will visit all of the vertices in the path, but since none of the other vertices has a path back to s , the strongly connected component of s is just s itself.

In order to actually find the strongly connected component of v , we need to know not just which vertices v can reach, but also which vertices can reach v . The way to compute this is to use the REVERSAL of G , which is defined as the graph obtained by flipping the direction of all of the edges in G (i.e. for every edge $(u, v) \in E(G)$, $\text{reverse}(G)$ instead has the edge (v, u)). We note that the reach of v in G' is exactly the set of vertices which have paths to v in G . This gives us the following algorithm for computing the strongly connected component of a vertex:

```

STRONGLYCONNECTEDCOMPONENT( $v$ )
    visited  $\leftarrow$  an array of  $n$  copies of False
    DIRECTEDDFS( $v$ )
    reachV  $\leftarrow$  visited
    Reverse all of the edges of  $G$ 
    visited  $\leftarrow$  an array of  $n$  copies of False
    DIRECTEDDFS( $v$ )
    Return  $\{u \in V \mid \text{reachV}[u] = \text{True}\} \cap \{u \in V \mid \text{visited}[u] = \text{True}\}$ 

```

We note that this algorithm runs in time $O(m + n)$ since we are making two calls to DFS, each of which takes $O(m + n)$ time, and the other operations of the algorithm can each be done in $O(n)$ time.

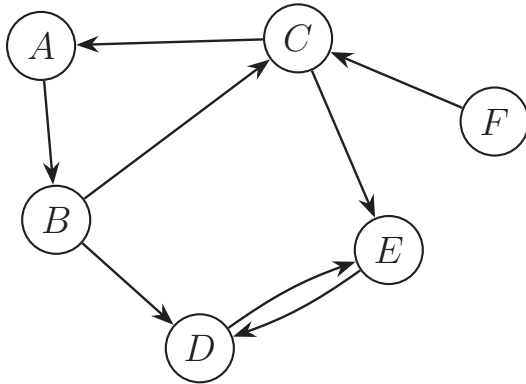
The meta-graph of a directed graph G is a graph G' where the vertices are strongly connected components of G . There is an edge (C_1, C_2) between two components C_1 and C_2 in the meta-graph if and only if there is an edge from some vertex in C_1 to some vertex in C_2 in G . The meta-graph of G is always a DAG (since if it had a cycle, the strongly connected components in the cycle could

all be merged into a single strongly connected component). We can compute the meta-graph in $O(n(m+n))$ time by repeating `STRONGLYCONNECTEDCOMPONENT` once from each vertex.

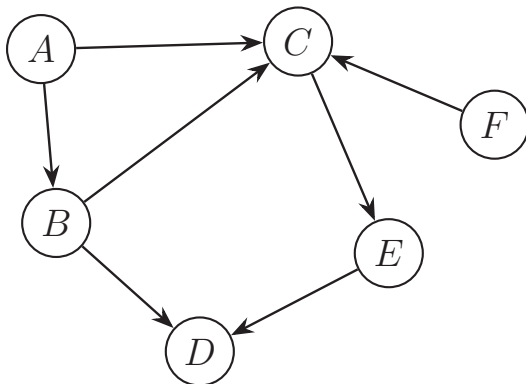
With a bit more effort it is possible to compute all strongly connected components with a single depth first search that does some extra bookkeeping, giving a runtime of $O(m+n)$ for computing the meta-graph. Tarjan's algorithm, developed by Robert Tarjan in 1972, was the first algorithm to do this. You may use this algorithm as a subroutine in algorithms that you write, but we will not describe or analyze it here. See Section 6.6 of Erickson if you are interested in the details.

14.5 Comprehension Questions

1. Consider the following graph:



- (a) What is $\text{reach}(C)$ in this graph?
 - (b) List the strongly connected components of this graph.
 - (c) Is this graph a DAG?
2. True or false? Running `DIRECTEDDFS( $v$ )` on a vertex v in a directed graph will visit all of the vertices in v 's strongly connected component.
 3. True or false? If G is a directed acyclic graph, then all of G 's vertices form a single strongly connected component.
 4. What is the runtime of counting the number of connected components in an undirected graph, using the results from this chapter?
 5. Give a topological sort of the following graph:



14.6 Exercises

1. A *Hamiltonian path* in a (directed or undirected) graph G is a path that visits every vertex of G . Prove that if a directed graph G has a *unique* topological sort, then G has a Hamiltonian path.
2. In Section 14.2 we saw an algorithm to determine whether or not a graph G contains a cycle in $O(m + n)$ time. Now suppose that we are given an undirected graph $G = (V, E)$ and a vertex $v \in V$ and want to determine whether or not G has a cycle containing the vertex v . Give an $O(m + n)$ time algorithm for this problem.
3. Prove that if G is a DAG then the following algorithm correctly finds a topological sort of G . Make sure to justify why the algorithm will always be able to find a vertex of in-degree 0.

```

SOURCEREMOVER( $G$ )
  If  $|V(G)| = 1$ 
    Return the sequence containing the lone vertex of  $G$ 
   $s \leftarrow$  a vertex in  $V(G)$  such that  $\deg_{in}(s) = 0$ 
   $G' \leftarrow G$  with vertex  $s$  removed
   $R \leftarrow$  SOURCEREMOVER( $G'$ )
  Prepend  $s$  to the beginning of  $R$ 
  Return  $R$ 

```

4. Consider the following variant of the job scheduling problem from Section 12.2: We are given a set of n jobs J which our machine needs to run. Each job j has three attributes:
 - $j.time$ is the time that j takes our machine to complete (measured e.g. in minutes)
 - $j.deadline$ is the time by which we are supposed to complete j (measured in minutes since the time when we start running jobs).
 - $j.dependencies$ is a (possibly empty) list of other jobs that must be completed before running j
 - (a) Give an $O(n^2)$ time algorithm to determine whether it is possible to complete the jobs in an order which satisfies all of the dependencies.
 - (b) Give an $O(n^2)$ time algorithm to find an ordering of the jobs which satisfies all of the dependencies and has the minimum possible maximum lateness out of all orderings that satisfy the dependencies (Hint: start by choosing which job to schedule last).
5. Your friend's Go club ([https://en.wikipedia.org/wiki/Go_\(game\)](https://en.wikipedia.org/wiki/Go_(game))) recently held a tournament in which players could freely play games against whoever was available, without any assigned bracket or rounds. Unfortunately, the club forgot to come up with a system for determining who won the event. Knowing that you are an algorithms expert, the club president has come to you for help. Suppose that you are given as input a list of the n go club members and a list of k tournament games, where each game is a triple consisting of which player played the black pieces, which player played the white pieces, and whether white or black won. Do not make any assumptions about which members played each other. Some players may have more games than others, and a given pair of players may have played against each other once, more than once, or not at all.
 - (a) The club president has asked you to rank all the players based on their performance in the tournament. To avoid any complaints about the methodology, the president wants a ranking (i.e. an ordered list of the club members) such that nobody is ranked higher than someone who beat them in the tournament. That is, if player A won against player B in a tournament game, then player A must be ranked higher than player B . Give an efficient algorithm to determine whether there exists such a ranking, given the results from the tournament.

- (b) Give an efficient algorithm to generate a ranking satisfying the condition in part (a), assuming such a ranking exists.
- (c) It turns out that there is no way to rank the members which satisfies the club president's condition, so the president has decided to allow ties in the ranking (e.g. several players may share 2nd place). Nevertheless, the president would like to minimize the number of players who tie for their position. Give an efficient algorithm to compute a rank for each player such that no player has a strictly better ranking than someone who beat them in a tournament game, and the total number of players who share their rank with somebody else is minimized.