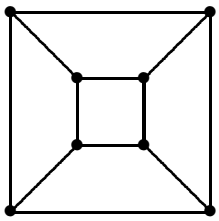
DFS Problems

1. Consider this fact from graph theory:
Theorem. *A bipartite graph with a perfect matching and minimum degree δ has at least δ perfect matchings.*
We use this terminology: A *matching* is a set of edges, no 2 having a common vertex. A matching is *perfect* if every vertex is on a matched edge. As an example of the theorem, it says that the cube has ≥ 3 perfect matchings:



Prove the theorem. *Hint*: An *alternating cycle* is a cycle whose edges are alternately matched and unmatched. If we interchange the matched and unmatched edges along an alternating cycle of even length, we get another perfect matching. Your proof uses path-based dfs to find the desired cycles.

2. **Easy Version**: A *Hamiltonian path* is a simple path of length $n-1$, i.e., it contains every vertex.
Example: The tournament of Handout#6 has the Hamiltonian path a, b, c, d, e.
Any tournament has a Hamiltonian path. We'll prove this by showing the algorithm below finds a Hamiltonian path if its input is a tournament.

The algorithm uses array $I[x], x \in V$ to represent the Hamiltonian path. Vertex x will have $I[x] = i$ if x is the ith vertex of the Hamiltonian path. The algorithm uses a depth-first search path P. P is empty when the algorithm starts, and it may become empty later on too. Whenever P becomes emtpy, if the graph still has vertices, we can choose an arbitrary vertex of V to become the start of the next P.

> $next\_num = n$ (the number of vertices);
> repeat until $next\_num = 0$:
> > grow the dfs path P until its endpoint x has all of its out-neighbors belonging to P;
> > assign $I[x] = next\_num$, decrease $next\_num$ by 1, delete x from P and G;

To prove this algorithm is correct we need only show that if $I[x] > 1$ and $I[y] = I[x] - 1$ then (y, x) is an edge of the graph. To do this consider the point in the algorithm where $I[x]$ is assigned.

(i) Show that every vertex z that is in the current graph but not in P has an edge (z, x).

(ii) Given (i), explain why the algorithm is correct.

Harder Version: A *Hamiltonian path* in a digraph is a simple path of length $n-1$, i.e., it contains every vertex. It's NP-complete to find a Hamiltonian path in a digraph.

Note that the tournament of Handout#6 has the Hamiltonian path a, b, c, d, e. In fact Rédei's Theorem states that every tournament has a Hamiltonian path.

(*Remark.* This is somewhat surprising, e.g., if you ever organize a round-robin tournament, you know even before the first game is played that you'll be able to rank the players as $1, 2, 3, \ldots, n$ so that

$$1 \text{ beat } 2,\ 2 \text{ beat } 3,\ \ldots,\ n-1 \text{ beat } n\ !)$$

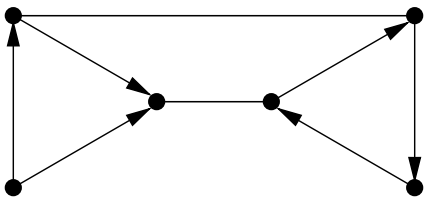
Give an algorithm whose input is a tournament G, and whose output is a Hamiltonian path of G. Your algorithm must be based on dfs and run in $O(m)$ time.

State your algorithm and prove it is correct. Your proof should *not* invoke Rédei's Theorem, i.e., your dfs algorithm actually *proves* Rédei's Theorem. Use less than a page total.

Hint: Use the path-based view of dfs.

3. A *mixed graph* G is one that can have both directed and undirected edges. G is *traversable* if for every ordered pair of vertices u, v, there is a path from u to v that has all its directed edges pointing in the forward direction. (So if G is undirected, G is traversable $\iff$ it's connected; if G is directed, G is traversable $\iff$ it's strongly-connected.) A *bridge* of G is an undirected edge that is a bridge of the undirected version of G (defined on CLR p.89). An *orientation* of G assigns a unique direction to each undirected edge. Robbins' Theorem has this generalization:

Theorem (Boesch & Tindell). *A traversable mixed graph has a strongly-connected orientation $\iff$ it has no bridge.*



Traversable bridgeless graph with a unique strongly connected orientation.

It's obvious that a mixed graph with a bridge has no strong orientation. We wish to prove a traversable bridgeless mixed graph has a strong orientation.

To do this give a high-level path-based dfs algorithm that constructs the desired orientation. Explain why your algorithm is correct (& thus proves Boesch & Tindell's theorem.)

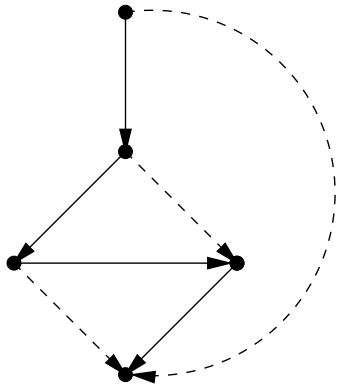
I found it convenient to organize my answer similar to the Handouts – first I gave the Basic Ideas, then the algorithm. I had to prove my basic ideas were correct. But it was then obvious that the algorithm was correct.

You don't have to give a low-level implementation of the algorithm. I can implement my high-level algorithm in $O(n + m)$ time, but that requires an advanced data structure. For that reason the implementation is not required. But be aware that if your algorithm is correct, it likely has a linear time implementation.

Hint. Try to mimic our proof of Robbins' Theorem. You'll discover that some new insights are needed!

4. Let G be a strongly connected digraph. Give an algorithm to find a cycle of odd length, or output a message that none exists. Your algorithm must run in linear time. *Hint*: Base your algorithm on the high-level SC algorithm. Your algorithm will prove a nice TONCAS.

5. In a digraph an edge (x, y) is *transitive* if deleting it does not change the vertices that x can reach.



The transitive edges are drawn dashed.

(i) Consider a dag G. Give a procedure `Transitive`(x) that flags every node in $Adj[x]$ corresponding to a transitive edge (i.e., y gets flagged in $Adj[x]$ if (x, y) is transitive). Assume initially all nodes in $Adj[x]$ are unflagged. Use the pseudocode instruction "flag y", where y is a node in $Adj[x]$.

Base your algorithm on dfs. Your algorithm must run in $O(n + m)$ time. Give pseudocode for the algorithm, and a brief explanation of why the algorithm works correctly. The time bound should be obvious. Use at most 3/4 page.

(ii) **Extra Credit:** Redo part (i), this time assuming G is an arbitrary digraph. The time bound should still be $O(n + m)$.

6. (not inherently DFS) A dag G has vertex set $V = \{1, \ldots, n\}$, and initially the edge set is $E = \emptyset$. We execute a sequence of m instructions

`INSERT`(x, y).

This instruction adds the directed edge (x, y) to E. It is guaranteed that G is always a dag after each `INSERT`. Each `INSERT` outputs a topological numbering of (the current) G.

Assume that the guarantee always holds, i.e., `INSERT` will never create a cycle. Assume for simplicity that $m \geq n$.

We can execute the sequence of m `INSERT`s in total time $O(m^2)$, by simply calling our topological numbering algorithm in each `INSERT`. This problem is to give an algorithm that uses total time $O(nm)$. Your algorithm can do some initialization work before the first `INSERT`, but the time for that gets included in your total time.

Give your algorithm at a high level. My writeup starts by describing the data structures, then gives the high level algorithm, followed by proof of correctness and time bound. Do not write more than 1 page.

7. We're implementing `STRONG` for input graphs G that are guaranteed to be dense. So instead of using an adjacency list representation, we store the graph in an adjacency matrix. Our goal is to implement `STRONG` in time $O(n^2)$.

We'll use the S and B stacks, exactly as in the notes. We'll number the SC's starting at 1 (i.e., $1, 2, \ldots, s$, if there are s SC's). We'll use the I array slightly differently from the notes : As in the notes, $I[v]$ is 0 when v is undiscovered, and $I[v]$ stores the SC number of v when v gets deleted. But unlike the notes, $I[v]$ is set to -1 when v is in the stack S.

Complete the code below by supplying the pseudocode for **a** and **b**. Aside from **a**–**b**, the only changes from the notes are the initialization of c, and the first & last lines of `DFS`.

```
procedure SUCCESSORS G {
empty stacks S and B;
for v ∈ V do I[v] = 0;
c = 0;
for v ∈ V do if I[v] = 0 then DFS(v) }

procedure DFS(v) {
PUSH(v, S); PUSH(TOP(S), B); I[v] = −1; /* add v to the end of P */
for edges (v, w) ∈ E do
    if I[w] = 0 then DFS(w);
/* the following code does the contractions */
    a
if  b  { /* number vertices of the next SC */
    POP(B); increase c by 1;
    do I[POP(S)] = c until I[v] = c; }; }
```

Note that **a** is a block of code (my **a** is about 5 lines long). Use $A[\cdot, \cdot]$ for the adjacency matrix.

Briefly explain how your code works, and why it achieves the time bound $O(n^2)$. My brief explanation is about 4 lines.

8. We're given a strongly connected digraph G. We'd like to find a spanning subgraph that is strongly connected and contains the fewest number of edges possible. Since that's NP-hard our goal is a good approximation algorithm.

Professor Dull says, "The pseudocode below is a 3/2-approximation algorithm. It uses the approach of the carving algorithm. Everything's entirely analogous so I won't even bother to prove the 3/2-approximation bound."

```
F denotes the edges of the algorithm's solution
initially F = ∅

repeat until G has 1 vertex:
    grow the dfs path P until its endpoint x has all it's out-neighbors in P
    /* y is an out-neighbor of x if edge (x, y) exists */
    let y be the out-neighbor of x closest to the start of P
    let C be the cycle formed by edge (x, y) & edges of P
    add all edges of C to F
    contract the vertices of C
```

Show Dull is wrong, by giving an example that works as follows. Your graph G has vertices numbered 1 to n, where n can approach infinity. The dfs starts at vertex 1. G has exactly 2 edges directed out of 1: $(1, 2)$ and $(1, x)$, for some vertex x.

• If the dfs starts by traversing $(1, x)$, Dull's algorithm finds a subgraph F that is a Hamiltonian cycle.
• If the dfs starts by traversing $(1, 2)$, Dull's algorithm outputs a subgraph F that shows the approximation ratio of the algorithm is ≥ 2.

For both edges $(1, 2)$ and $(1, x)$, once that edge is chosen there is only 1 possible dfs, i.e., the search makes no more arbitrary choices. Explain your example.

9. This multipart problem discusses issues that come up in the design of a high level path-based dfs algorithm. The problem we discuss may seem artificial. But the algorithm ($\mathcal{A}$ below) has important applications, eg, finding the "components" of a mixed graph. [Mixed graphs are defined in Handout#9 p.3, Ex.3, and "components" are defined in the natural way just like Handout#6.]

Let G be a strongly connected digraph. Say that a simple cycle of length ≥ 3 is *long*. The *long cycle problem* is to repeatedly contract a long cycle until no such cycle remains, i.e., every remaining cycle has length 2. In general the final graph that we wind up with won't be unique – it depends on the specific cycles that get contracted. *Example:*

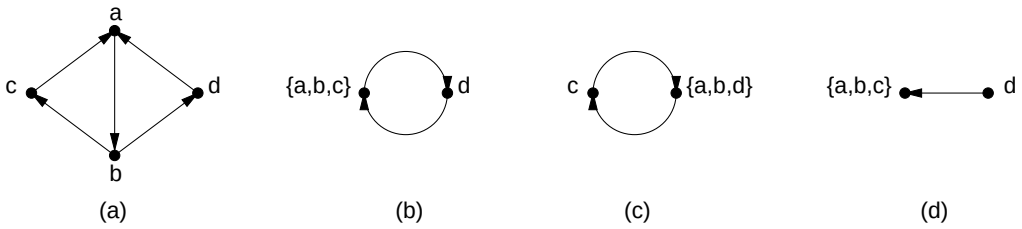


Fig.1 Given graph (a), we can wind up with either (b) or (c).
(Not (d) – (d) is discussed below.)

Algorithm $\mathcal{A}$ below solves the long cycle problem. $\mathcal{A}$ is a modification of the high level algorithm of Handout#7. As in that algorithm, a contract operation discards parallel edges. Algorithm $\mathcal{A}$ uses one new concept: Call the last vertex s of the dfs path P a *pseudosink* if s has exactly 1 outgoing edge, and that edge goes to the vertex preceding s in P.

Example. Suppose in Fig.1(b), the dfs path P consists of the upper edge (from $\{a, b, c\}$ to d). Then d is a pseudosink. If the dfs path consists of the lower edge, then the contracted vertex $\{a, b, c\}$ is a pseudosink.

Algorithm $\mathcal{A}$ /* the contractions done by algorithm $\mathcal{A}$ solve the long cycle problem */
initialize a dfs path P to an arbitrary vertex a;
repeat until P consists of 1 vertex which is a sink:
    grow the dfs path P until a cycle C is found, or P ends in a sink or pseudosink s
        long cycle C: contract the vertices of C
        length 2 cycle C: do nothing for C, just continue growing P
        sink s: delete s from P & G
        pseudosink s: delete s from P; delete the edge of P leading to s from G

/* Now $\mathcal{A}$ halts. The current G has no long cycles, & is a possible solution */

Example cont'd. In Fig.1(b) d gets processed as a pseudosink. When the algorithm halts, the graph has been transformed to Fig.1(d), with $\{a, b, c\}$ a sink.

[$<1/2$ page] (*i*) Professor Dull says, "My algorithm is simpler than $\mathcal{A}$! I treat pseudosinks just like sinks, i.e., replace the last 2 lines of $\mathcal{A}$ by

sink or pseudosink s: delete s from P & G"

Give an example where Dull's algorithm halts with an incorrect answer. Use only 3 vertices. A picture will help you explain the answer.

[1 page] (*ii*) Consider a graph G on n vertices that contains a Hamiltonian cycle, i.e., a cycle passing through every vertex, like this:

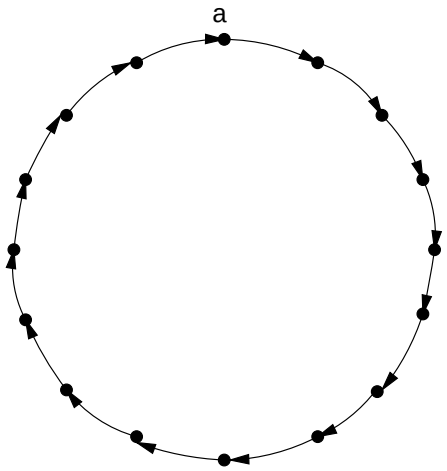


Fig.2 Graph with Hamiltonian cycle.

One way to execute algorithm $\mathcal{A}$ on this graph would be to grow P along the cycle, starting at vertex a. When P contains every vertex, the edge entering a creates a cycle, causing all n vertices to be contracted into 1.

Show how to add edges to the cycle so that algorithm $\mathcal{A}$ can halt with a graph containing $\Theta(n)$ vertices instead of only 1 vertex.

You may use Fig.2 (or something like it) to illustrate your construction. But your construction should work for an infinite number of values of n. [My answer works for all n, but you can skip some n's if it's convenient.] Explain your answer.

[$<1/2$ page] (*iii*) This problem is to justify the last step of algorithm $\mathcal{A}$, that deletes the edge of P leading to the pseudosink s. Let (r, s) be that edge. There are 2 issues to check:

(a) Deleting (r, s) does not destroy a long cycle.

(b) Deleting (r, s) does not cause a long cycle to become unreachable from P.

[(b) is an issue because the algorithm only finds cycles that are reachable from P. So if (b) failed the algorithm would miss a long cycle.]

Explain why (a) is true. Then explain why (b) is true. *Hint for (b)*: Argue by contradiction, focussing on the first time a long cycle becomes unreachable from P.

[<1/2 page] (*iv*) For simplicity we stated algorithm $\mathcal{A}$ to include the case when s is a sink. But actually the line labelled "sink s" can be eliminated. This is because the only time s can be a sink is when s is the first vertex of P. Your task is to explain why this is true.

Hint. Here's a good way to start the proof: Consider some point in the execution of algorihtm $\mathcal{A}$. Call the current graph G. Let s be a sink of G. We want to show that s is the first vertex of P.

Let G' be the current graph with all edges that $\mathcal{A}$ deleted added back in. [Strictly speaking, the deleted edges may not be in G – rather they could be in graphs that got contracted to G. But they correspond to edges in G in an obvious way.] Now analyze G' to reach the desired conclusion. Your analysis will use the fact that the algorithm starts with a graph, say G_0, that is strongly connected.

[<1/2 page] (*v*) Here's the same pseudocode as Handout#7, p.3 but with the lines numbered:

```
1.  procedure DFS(v) {
2.  PUSH(v, S); I[v] = TOP(S); PUSH(I[v], B); /* add v to the end of P */
3.  for edges (v, w) ∈ E do
4.     if I[w] = 0 then DFS(w)
5.     else /* possible contract */ while B[TOP(B)] > I[w] do POP(B);
6.  if B[TOP(B)] = I[v] then { /* number vertices of the next SC */
7.     POP(B); increase c by 1;
8.     while TOP(S) ≥ I[v] do I[POP(S)] = c }; }
```

Line 5 contracts cycles, but it doesn't use use the rule of algorithm $\mathcal{A}$, i.e., only contract long cycles. Modify line 5 so it uses $\mathcal{A}$'s rule, i.e., it contracts a cycle if and only if the cycle is long. You must keep the time $O(m + n)$. (This isn't hard.)