COMS20010 — Problem sheet 4

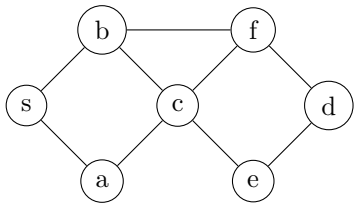
You don't have to finish the problem sheets before the class — focus on understanding the material the problem sheet is based on. If working on the problem sheet is the best way of doing that, for you, then that's what you should do, but don't be afraid to spend the time going back over the quiz and videos instead. (Then come back to the sheet when you need to revise for the exam!) I'll make solutions available shortly after the problem class. As with the Blackboard quizzes, question difficulty is denoted as follows:

- ⋆ You'll need to understand facts from the lecture notes.
- ⋆⋆ You'll need to understand and apply facts and methods from the lecture notes in unfamiliar situations.
- ⋆⋆⋆ You'll need to understand and apply facts and methods from the lecture notes and also move a bit beyond them, e.g. by seeing how to modify an algorithm.
- ⋆⋆⋆⋆ You'll need to understand and apply facts and methods from the lecture notes in a way that requires significant creativity. You should expect to spend at least 5–10 minutes thinking about the question before you see how to answer it, and maybe far longer. At most 10% of marks in the exam will be from questions set at this level.
- ⋆⋆⋆⋆⋆ These questions are harder than anything that will appear on the exam, and are intended for the strongest students to test themselves. It's worth trying them if you're interested in doing an algorithms-based project next year — whether you manage them or not, if you enjoy thinking about them then it would be a good fit.

If you think there is an error in the sheet, please post to the unit Team — if you're right then it's much better for everyone to know about the problem, and if you're wrong then there are probably other people confused about the same thing.

This problem sheet covers week 4, focusing on graph representations, depth- and breadth-first search, and Dijkstra's algorithm.

1. We will begin by considering the following unweighted graph.



(a) [⋆] Write down the adjacency list representation and adjacency matrix representation of this graph.

Solution: The adjacency list representation is:

$$\boxed{\text{s}} \to b, a$$
$$\boxed{\text{a}} \to s, c$$
$$\boxed{\text{b}} \to s, c, f$$
$$\boxed{\text{c}} \to a, b, f, e$$
$$\boxed{\text{d}} \to e, f$$
$$\boxed{\text{e}} \to c, d$$
$$\boxed{\text{f}} \to b, c, d$$

The adjencency matrix representation is:

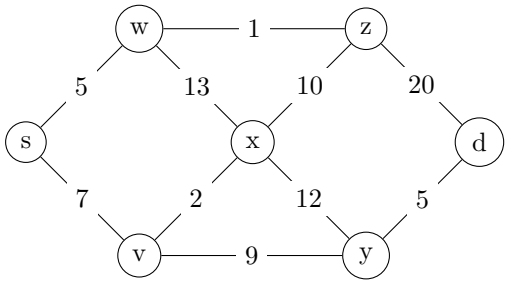
$$\begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ s \end{array} \begin{array}{c} \begin{array}{ccccccc} a & b & c & d & e & f & s \end{array} \\ \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

(b) [⋆⋆] Run the BFS algorithm on this unweighted graph to find the distance from s to d. Show your working by showing the order the algorithm visits each vertex as well as the distances it finds to each vertex.

Solution: BFS works by starting at a vertex and then adding all adjacent vertices to a queue. We then take the first vertex in the queue and look for a new set of adjacent vertices to add, repeating the process until we have reached our destination.

In this case, the algorithm starts at s, and first visits vertices a and b, noting that $d(s,a) = d(s,b) = 1$. Next, we visit vertices c and f, noting that $d(s,c) = d(s,f) = 2$. Next, we visit vertices e and d, noting thar $d(s,e) = d(s,d) = 3$. Then, since d is the destination vertex, the algorithm terminates and returns $d(s,d) = 3$.

(c) [⋆⋆] Now run Dijkstra's algorithm on the following weighted graph to find the distance from s to d. Show your working by showing the order the algorithm visits each vertex as well as the distances it finds to each vertex.



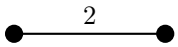
Solution: In Djikstra's algorithm we replace the queue of BFS by a priority queue — this lets us start at the source vertex, and then successively visit the next-nearest vertex to the source.

In this case, we start from s and first visit w, since w is the nearest neighbour of s with $d(s,w) = 5$. Now, from the set $\{s,w\}$, the next nearest place we can go is z, noting $d(s,w) = 5+1 = 5$. Next, we visit vertices v and then x, noting $d(s,v) = 7$ and $d(s,x) = 7+2 = 9$. At this point, we know the distances from s to each of v, w, z and x via the paths swz and svx. The edge zx joins two vertices whose distances we already know, so we can disregard it — we know without checking that the path $swzx$ is no shorter than svx, and that $svxz$ is no shorter than swz. As such, the next edge we follow is $\{v,y\}$ to add vertex y, giving us $d(s,y) = 7+9 = 16$. Finally, we go from y to d, getting a final result of $d(s,d) = 16+5 = 21$.

(d) [⋆] How is Dijkstra's algorithm different or similar to the BFS algorithm?

Solution: Dijkstra's algorithm is equivalent to the BFS algorithm except that it also looks at the weights of edges to determine where to go next — it puts the edges into a priority queue rather than a queue. If we take a weighted graph and convert it to an unweighted graph by replacing each edge of a given weight by a new path with the same length, then both algorithms would visit vertices in the same order too.

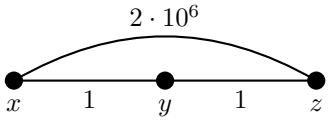
An example of this construction would be to map



to

2. (a) [⋆⋆] John is trying to solve the single-source shortest path problem on a weighted graph with positive weights. He decides that Dijkstra's algorithm is too complicated, and that he should be able to get a "good enough" answer by ignoring the weights and using breadth-first search. Give an example to show that this approach could return a path which is a million times too long.

Solution:



In the graph above, unweighted breadth-first search will return the path xz from x to z with length $2 \cdot 10^6$, while Dijkstra's algorithm will return the path xyz with length 2.

(b) [⋆⋆⋆] Show that the same thing could happen even if John's breadth-first search always picks the edge with the lowest weight.

Solution: Let $x_1 \dots x_n x_1$ be a cycle in which $\{x_1, x_t\}$ has weight 2 and every other edge has weight 1. Then John's modified breadth-first search starting from x_1 will pick only the unit-weight edges, finding the path from x_1 to x_t of length t and missing the path $x_1 x_t$ of length 2. Taking $t = 2 \cdot 10^6$, we obtain the required counterexample.

3. [⋆⋆⋆] Give an algorithm which, given a directed graph $G = (V, E)$ and two disjoint lists of vertices $X, Y \subseteq V$, returns a shortest path from any vertex in X to any vertex in Y in $O(|V| + |E|)$ time. (In other words, the path should start in X, end in Y, and be of length $\min\{d(x, y) \colon x \in X, y \in Y\}$.) You may assume that at least one such path exists. (**Hint:** The "nice" way of doing this involves using an algorithm you already know in a clever way, not coming up with a new one.)

Solution: Form a new directed graph $G' = (V', E')$ by contracting all of X into a new vertex x, and all of Y into a new vertex y, preserving all the attached edges. More formally, for all $v \in V$, let

$$f(v) = \begin{cases} x & \text{if } v \in X, \\ y & \text{if } v \in Y, \\ v & \text{otherwise.} \end{cases}$$

Then we take $V' = V \setminus (X \cup Y) \cup \{x, y\}$, and $E' = \{f(e) \colon e \in E\}$. Any length-$\ell$ path from x to y in G' corresponds to a length-ℓ path from X to Y in G, and vice versa. We then run breadth-first search to find a shortest path from x to y in G', and return the corresponding path in G.

Correctness is immediate from the correctness of breadth-first search. Forming G', running breadth-first search on G', and finding the corresponding path in G all take $O(|V| + |E|)$ time.

4. You are trying to plan a train journey within the UK as cheaply as possible. Unfortunately, ticket pricing in the UK does not make sense — often the cheapest way to go from point A to point B is to go via points C, D and E.

   (a) [⋆⋆] Explain how to use Dijkstra's algorithm to find the cheapest journey from A to B, given a list of possible journeys and prices. You may ignore arrival and departure times — if your shortest route is $A \to B \to C$, then on arrival at B you will simply wait for the next train from B to C.

   Solution: From the list of possible journeys, we can extract a list of possible stations V. We form a weighted graph G on V, joining station u to station v if there is a journey from u to v in our list, and weighting that edge with the cost of that journey. Thus the total length of a path in G is equal to the total cost of the corresponding journey, and we can find the cheapest journey from x to y by running Dijkstra's algorithm on G and x.

   (b) [⋆⋆⋆] On top of the strange ticket pricing, several train companies are deeply unreliable and may leave you stranded at a station (or even on a track) for hours. You have had particularly bad experiences with one particular company, let's call them Sirius Rail, and wish to give them as little money as possible. Given a list of the journeys they run, adapt your answer to part a) to first minimise the amount of money you pay to Sirius Rail, then minimise the total cost of the route subject to that.

   Solution: Let's measure the cost of a journey in pennies, so that the edges of G are weighted by integers. Go through the entire list of journeys to find the most expensive one, say with cost C. Thus every possible journey has cost at most $|V|C$. Now, form a new graph G' by reweighting every journey with Sirius Rail, so that if its original weight was c, its new weight is $2|V|C \cdot c$. Then the length of a path in G' is

   $$2|V|C \cdot [\text{total cost with Sirius}] + [\text{total cost not with Sirius}].$$

   Consider two paths P and Q in G'. If they spend different amounts with Sirius Rail, say with P spending more than Q, then P must spend at least 1 penny more than Q. This implies

   $$\text{length}(P) - \text{length}(Q) \geq 2|V|C \cdot 1 - |V|C > 0,$$

   so P is longer than Q. If they spend the same amount with Sirius Rail, but P spends more in total than Q, then P will be longer in G'. Thus the shortest path from x to y in G' corresponds to the journey which costs as little as possible subject to giving as little money to Sirius Rail as possible. We can therefore solve the problem by running Dijkstra's algorithm on G'.

   This technique of reducing one problem to another will become very important later on in the unit.

5. (a) [⋆⋆] Give an example in which breadth-first search as described in lectures, with inputs $G = (V, E)$ and $v \in V$, requires $\Omega(|V|^2)$ space.

   Solution: Let G be a complete graph, in which every vertex is joined to every other vertex, and write $V = \{v_1, \ldots, v_n\}$ Then the algorithm will progress as follows, up to renumbering the vertices:

- $(v_1, v_2), \ldots, (v_1, v_n)$ are added to the queue.
- (v_1, v_2) is removed from the queue, and $(v_2, v_3), \ldots, (v_2, v_n)$ are added.
- (v_1, v_3) is removed from the queue, and $(v_3, v_2), (v_3, v_4), \ldots, (v_3, v_n)$ are added.
- And so on down to (v_1, v_n), each time removing one edge from the queue but adding $n - 2$.

By this point in the algorithm, the queue will contain at least $(n-1)(n-2) \in \Omega(n^2)$ vertices, so the algorithm requires $\Omega(n^2)$ space.

(b) [⋆⋆⋆] Suggest a way of reducing the space requirement to $O(|V|)$.

Solution: Consider the pseudocode described in lectures. We are currently adding edges (v_i, v_j) to the queue up to $d(v_j)$ times, but we only actually do anything with them the first time we see each endpoint v_j — after that, we will have $\texttt{L}[j] \neq \infty$, so lines 8–9 won't evaluate and we just throw the edge away. This means we can save space without affecting correctness by making sure that if some edge (v_i, v_j) is already in the queue, then we don't add any more edges (v_k, v_j).

To do this, we could e.g. add a new possible value for $\text{L}[j]$, modifying line 4 and lines 7–8 of the pseudocode from lectures as shown below.

Algorithm: BFS

Input : Graph $G = (V, E)$, vertex $v \in V$.
Output: $d(v, y)$ for all $y \in V$ and "a way of finding shortest paths".

```
1  Number the vertices of G as v = v_1, ..., v_n.
2  Let L[i] ← ∞ for all i ∈ [n].
3  Let L[1] ← 0, pred[1] ← None.
4  Let queue be a queue containing all tuples (v, v_j) with {v, v_j} ∈ E.
5  Let L[j] ← Seen for all j ∈ [n] with {v, v_j} ∈ E.
6  while queue is not empty do
7      Remove front tuple (v_i, v_j) from queue.
8      if L[j] = Seen then
9          foreach {v_j, v_k} ∈ E with L[j] = ∞ do
10             Add (v_j, v_k) to queue.
11             Set L[k] ← Seen.
12         Set L[j] ← L[i] + 1, pred[j] = i.
13 Return L and pred.
```

6. Let $G = (V, E)$ be a connected graph, let $x \in V$, and run depth-first search on G starting from x. Let T be the graph on V whose edges are the edges of G traversed by depth-first search — that is, edges $\{u, v\}$ such that at some point in the algorithm `helper`(v) is called from `helper`(u) while v is unexplored.

(a) [⋆⋆⋆] Prove that T is a tree.

Solution: We proved in lectures that every vertex in x's component has a path to x via edges of T. Since G is connected, x's component is all of G, so it follows that T is connected. We prove that T is acyclic by induction, showing that it stays acyclic throughout the evaluation of depth-first search. Initially T contains no edges, so this is immediate. Moreover, the only way a vertex u can be non-isolated in T is if `helper`(u) has been called, so the only non-isolated

vertices in T have already been marked as explored. Since the edges added to T in evaluating `helper`(u) are precisely the edges in G from u to unexplored vertices, it follows that no cycles in T are added during the evaluation of `helper`(u).

(b) [⋆⋆⋆] Considering T to be rooted at x, prove that if $\{u, v\} \in E$, then either u is an ancestor of v in T or vice versa. (In other words, T is a DFS tree for G.)

Solution: Again we work by induction, this time considering the subtree of explored vertices — in other words, writing X_i for the set of the first i vertices explored, we will show that $T[X_i]$ is a DFS tree for $G[X_i]$. Initially, only one vertex has been explored (x itself), and $T[x]$ contains no edges, so the result is vacuously true. Suppose $T[X_i]$ is a DFS tree for $G[X_i]$ for some $1 \leq i < |V|$, and let u_{i+1} be the $(i+1)$'st vertex we explore. Then by induction, it's enough to show that for any edge $\{u_{i+1}, v\}$ with $v \in X_i$, v is an ancestor of u_{i+1} (since u_{i+1} cannot be an ancestor of v).

Since $v \in X_i$, we know that v has been explored, so `helper`(v) has been called. Since $u_{i+1} \notin X_i$, we know that u_{i+1} has not been explored despite being adjacent to v. The only way this can happen is if the evaluation of `helper`(v) has not yet completed. Thus the path in T from x to u_{i+1} described in lectures must pass through v, and so y is an ancestor of u_{i+1} as required.

7. [⋆⋆] Let $G = (V, E)$ be a connected graph, and suppose that T is **both** a DFS tree **and** a BFS tree for G. Prove that this implies G is a tree (so that $G = T$).

Solution: Suppose for a contradiction that $G \neq T$, so that there is an edge $\{u, v\}$ in $E(G) \setminus E(T)$. Let L_i and L_j be the layers containing u and v, respectively. Since T is a BFS tree, we have $|i-j| \leq 1$ — i.e. L_i and L_j must be equal or adjacent. Since T is a DFS tree, either u is an ancestor of v or v is an ancestor of u. It follows that u must be the parent of v, in which case $\{u, v\} \in E(T)$. This is a contradiction, so we're done.

8. (a) [⋆⋆⋆] Give an algorithm which, given a connected undirected graph $G = (V, E)$ in adjacency list form, decides whether G contains a cycle with an odd number of edges in $O(|E|)$ time. (**Hint:** A BFS tree will be useful.)

Solution: First use breadth-first search to create a BFS tree T for G, rooted at some arbitrary vertex v. Let the layers be $L_0, L_1, \ldots, L_t$, and organise the vertices of G by layer; this takes $O(|E|)$ time. Then iterate over all the edges of G, return `Yes` if G contains an edge with both endpoints in some L_i, and return `No` otherwise. This also takes $O(|E|)$ time.

We now prove correctness. Suppose our algorithm returns `Yes`, so that G contains an edge $\{x, y\}$ internal to some layer L_k of T. Then since T is a tree covering all of V, there is a path $va_1 \ldots a_{k-1}x$ from v to x and a path $vb_1 \ldots b_{k-1}y$ from v to y, where $a_i, b_i \in L_i$ for all $i \in [k-1]$. If these paths are vertex-disjoint except at v, then $va_1 \ldots a_{k-1}xyb_{k-1} \ldots b_1 v$ is a cycle containing $2k+1$ edges and we're done. Suppose they're not disjoint; then they must diverge at some point $\ell = \max\{i \in [k-1] \colon a_i = b_i\}$. Since $a_\ell = b_\ell$, $a_\ell a_{\ell+1} \ldots a_{k-1}xyb_{k-1} \ldots b_{\ell+1}b_\ell$ is a cycle containing $2(k-\ell)+1$ edges and we're done.

Now suppose G contains an odd cycle $C = x_1 \ldots x_{2k+1}x_1$, and write $x_{2k+2} = x_1$ for convenience; we will show our algorithm returns `Yes`, i.e. that there is an edge internal to some layer of T. Suppose for a contradiction that no such edge exists. Let ℓ_i be the index of the layer containing x_i, so that $x_i \in L_{\ell_i}$. Then since T is a BFS tree, and since it contains no edge internal to any layer, we have $\ell_{i+1} \in \{\ell_i + 1, \ell_i - 1\}$ for all $i \in [2k+1]$. Say x_i is an *up-vertex* if $\ell_{i+1} = \ell_i + 1$,

and a *down-vertex* if $\ell_{i+1} = \ell_i - 1$. Then since $2k+1$ is odd, there cannot be an equal number of up-vertices and down-vertices in the cycle; thus x_{2k+2} must be in a different layer to x_1. But x_{2k+2} and x_1 are the same vertex, so this is impossible.

(b) [$\star\star$] How would you adapt your algorithm to work with arbitrary graphs G which may or may not be connected?

Solution: G contains an odd cycle if and only if some component of G contains an odd cycle. We therefore use breadth-first search or depth-first search to find the components of G, apply the algorithm of part a) to each component, and return `Yes` if and only if we find an odd cycle in some component.

9. You are building a package management system, like apt-get for Linux or pip for Python. A basic problem such systems have to solve is resolving dependencies — a user wishes to install several packages at once, and some of these packages require other packages to be installed first. This question will walk you through a way of using DFS trees to build a system able to handle this without further input from the user.

We can view the situation as a directed graph G, where the vertex set P is the set of all available packages. If there is an edge from x to y, this indicates that x *directly depends* on y, and that x cannot be installed unless y is present. *Indirect dependencies* are also possible — we might have (x, y), $(y, z) \in E(P)$, so that x cannot be installed without first installing z, but $(x, z) \notin E(P)$.

(a) [$\star$] Explain briefly why it is reasonable to assume that G does not contain a (directed) cycle. We call such graphs *directed acyclic graphs* or *dags*.

Solution: If G contains a directed cycle $x_1 \dots x_t x_1$, then none of the packages $x_1, \dots, x_t$ can ever be installed — to install x_t you must first install x_{t-1}, which requires installing x_{t-2}, and so on all the way back to x_t again. In other words, $x_1, \dots, x_t$ must all be installed simultaneously and should have been bundled as a single package.

(b) [$\star\star\star\star$] Give an algorithm which, given a package x and a dag G in adjacency list form, outputs an **ordered list** of which packages must be installed in order to install x. All the direct and indirect dependencies of any given package should appear before it on the list, and your algorithm should run in $O(|E(G)|)$ time. This is called a *topological order*.

Solution: Build a DFS tree T, rooted at x. The vertices of T will be precisely the set of vertices y such that there's a path from x to y in G — that is, the set of direct and indirect dependencies of x. We require an ordering $V(T)$ such that for all $y, z \in S$, if there's an path from y to z, then y comes after z in the order. By induction, it's enough to show that if there's an **edge** from y to z, then y comes after z in the order. (Indeed, given this, if $x_1 \dots x_t$ is a path, then x_{t-1} must come before x_t, x_{t-2} must come before x_{t-1}, and so on down to x_1.)

We partition T into layers $L_0, \dots, L_a$ in the usual way. We then take our ordering to be $L_a, L_{a-1}, \dots, L_0$, ordering vertices within a layer arbitrarily. By the DFS tree property, if there is an edge from y to z for some $y, z \in S$, then either y be an ancestor of z or vice versa. Since G does not contain any directed cycles, z can't be an ancestor of y, so y must be an ancestor of z — and hence appear after z in the order.

Building the DFS tree takes $O(|E(G)|)$ time, and finding the layers of the tree and forming the desired order each take $O(|V(T)|)$ time. Since T is a tree, $|V(T)| = |E(T)| - 1 \leq |E(G)|$, so the overall running time is $O(|E(G)|)$.

10. (a) [⋆⋆] Let G be a graph with vertex set $[n]$ and adjacency matrix A. Prove by induction that for all $t \geq 1$ and all $i, j \in [n]$, $(A^t)_{i,j}$ is the number of **walks** from i to j of length t.

Solution: We proceed by induction on t. Since $A_{i,j} = 1$ if there's an edge from i to j and $A_{i,j} = 0$ otherwise, the result holds for $t = 1$. Suppose now that it holds for some t. By the formula for matrix multiplication, for all i and j we have

$$(A^{t+1})_{i,j} = (A^t)_{i,j} \cdot A_{i,j} = \sum_{k \in [n]} (A^t)_{i,k} A_{k,j}.$$

Since $A_{k,j} = 1$ if there is an edge from k to j and 0 otherwise, we can rewrite this as

$$(A^{t+1})_{i,j} = \sum_{k \in [n] \colon \{k,j\} \in E(G)} (A^t)_{i,k}.$$

By the induction hypothesis, this implies

$$(A^{t+1})_{i,j} = \sum_{k \in [n] \colon \{k,j\} \in E(G)} \#(\text{walks from } i \text{ to } k)$$

Since every length-$(k+1)$ walk from i to j corresponds to a unique length-k walk from i to some neighbour k of j (by removing the last step), the result follows.

(b) [⋆⋆⋆] Give a recursive algorithm which, given an $n \times n$ matrix A and an integer $t \geq 0$, calculates A^t using $O(\log n)$ matrix multiplications.

Solution: The trick is to repeatedly apply the following recurrence relation:

$$A^k = \begin{cases} 1 \text{ if } k = 0, \\ A \cdot A^{k-1} \text{ if } k \text{ is odd}, \\ (A^{k/2})^2 \text{ otherwise.} \end{cases}$$

Observe that after every (at most) two multiplications, the exponent in the power of A we have to calculate is halved; thus only $\Theta(\log n)$ multiplications are needed in total.

(c) [⋆⋆] Using parts a) and b), give an algorithm which lists all pairs of vertices $i, j \in [n]$ which are distance at most k apart, using at most $O(n^3 \log k)$ arithmetic operations.

Solution: We can calculate A^k by performing $\log k$ multiplications of $n \times n$ matrices, which using the naive algorithm will take $O(n^3)$ time each for a total of $O(n^3 \log k)$. We then spend $O(n^2) \subseteq o(n^3 \log k)$ time listing the non-zero entries of A^k to find the set of all vertices which are a length-k walk away from i. We repeat the process to find the set of all vertices which are a length-$k-1$ walk away from i. We then combine the lists into one, removing duplicates in $o(n^3 \log k)$ time, and output the result.

Why does this work? Well, we output the set of all vertices joined to i by a length-k or length-$k-1$ walk. This walk must contain a path, so these vertices all satisfy $d(i,j) \leq k$. Conversely, suppose $d(i,j) \leq k$ — then we must show that there is a length-k or length-$k-1$ walk form i to j, so our algorithm outputs it. We know there is a path $x_1 x_2 \dots x_a$ from i to j for some $a \leq k$. By repeatedly hopping to and fro between x_a and $x - a - 1$, we can turn this path into a walk $x_1 x_2 \dots x_a x_{a-1} x_a x_{a-1} \dots x_a$ from i to j of length either k or $k-1$, depending on the parities of k and a, so we're done.

(d) [⋆] Is the algorithm of part c) ever useful?

Solution: No, at least as described. Remember that breadth-first search from a vertex i runs in $O(n + |E(G)|)$ time, and gives the distance from i to j for all $j \in [n]$. Thus we can get an $O(n^2 + n|E(G)|)$-time algorithm by just running breadth-first search from every vertex. Since $n^2 \geq |E(G)|$, this will be faster and achieve better results.

(e) [$\star\star\star\star$] By calculating A^k more efficiently, the running time for part c) can be cut to $O(n^\omega \log k)$ arithmetic operations, where $\omega \approx 2.373$. (Yes, ω is the standard name for this constant, and yes, this is a very stupid choice given that ω-notation exists.) Will this version of the algorithm ever be useful?

Solution: The part b) algorithm now runs faster than repeated breadth-first search in the RAM model for dense graphs - if $|E(G)| \in \Theta(n^2)$, then we have $n^\omega \log n \in o((n^2 + n|E(G)|) \log k)$.

However, the algorithm is still strictly worse than repeated breadth-first search! The problem is that the RAM model is no longer applicable to the real world when we try to store very large numbers in the RAM. In general, if e.g. $k = n/2$, then there could very easily be $2^{n/2}$ length-k walks between i and j, in which case the matrix multiplications in the algorithm will be working with $\Theta(n)$-bit numbers. On real hardware, this will increase the running time to $\Theta(n^{\omega+1} \log n)$, which is worse than repeated breadth-first search. (The "word-RAM" model accounts for these issues in the RAM model by requiring each word to contain at most $\log n$ bits, which is reasonable on the ground that typical RAM size is 64 and typical input sizes are less than 2^{256} or so, but this is of course more unpleasant to reason about. In practice it is best to simply be very wary about bounds in terms of arithmetic operations when large numbers are involved.)

That said, the core idea is salvageable. With a little bit of subtlety you can avoid the issues associated with large numbers, and even improve the algorithm to output a list of the exact distances between each pair of vertices, while preserving an $O(n^\omega \log n)$ running time. This is known as *Seidel's algorithm*, and the problem is known as *all-pairs shortest path* or *APSP*.