COMS20010 — Problem sheet 2

This problem sheet covers week 2, focusing on greedy algorithms and graphs.

1 Greedy Algorithms

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.

1. Here we will consider the set cover problem. Given a set of elements $U = \{1, 2, \ldots, n\}$ (called the *universe*), and a collection of subsets of U, $S = \{A, B, C, \ldots\} \in \mathcal{P}(U)$, we want to find a subset of elements of S whose union contains the entirety of U. For example, suppose $U = \{1, 2, 3\}$ and $S = \{\{1, 2\}, \{1\}, \{3\}\}$. Then $\{\{1, 2\}, \{3\}\}$ is the smallest solution to the set cover problem. Note that $\{\{1, 2\}, \{3\}, \{1\}\}$ is also a solution, but is of larger size.
   (a) [⋆⋆] Consider the instance of the set cover problem given by

$$U = \{1, 2, 3, 4, 5\}, \qquad S = \big\{\{1, 2, 3\}, \{3, 4, 5\}, \{5\}, \{2, 3, 4\}, \{1, 2\}\big\}.$$

   What is a smallest solution to this instance? What is the largest solution?

   > **Solution:** $\{\{1, 2, 3\}, \{3, 4, 5\}\}$ is a smallest solution (although there are several others), and S itself is the largest solution.

   (b) [⋆⋆] Consider this greedy algorithm to solve the set cover problem.

Algorithm: GreedySetCover($U, S \in \mathcal{P}(U)$)

```
1 begin
2     Initialise Sol, covered ← ∅.
3     while covered ≠ U do
4         Let I be an element of S maximising |I \ covered|.
5         If I ⊆ covered, return No solution.
6         Set Sol ← Sol ∪ {I} and covered ← covered ∪ I.
7     Return Sol.
```

Run this algorithm using U and S from part (a), showing the value of `Sol` at each iteration.

Solution: Let S_i be the value of `Sol` at the start of the i'th iteration of the while loop. Then depending on how ties are handled, we might have e.g. $S_1 = \emptyset$, $S_2 = \{1, 2, 3\}$ $S_3 = \{\{1, 2, 3\}, \{3, 4, 5\}\}$, and the algorithm returns S_3 and halts before the fourth iteration.

(c) [⋆⋆] Give an example instance of set cover (i.e. values for U and S) for which no solution exists. How does GreedySetCover go wrong on such an instance? How can it be modified to efficiently deal with this?

Solution: One example is $U = [5]$, $S = \{\{3\}\}$. In this case, the while loop will add $\{3\}$ to `Sol` in the first iteration, then take $I = \{3\}$ a second time and return `No solution`.

(d) [⋆⋆⋆] Give a loop invariant which would allow us to prove by induction that GreedySetCover terminates and outputs a valid set cover (i.e. that it is a valid algorithm for the problem).

Solution: There are many such loop invariants. One of them is that at the start of the i'th iteration of lines 3–6, `Sol` is a subset of S, `covered` is the union of all sets in `Sol`, and $|\texttt{covered}| \geq i - 1$. (At $i = 0$ the invariant is maintained trivially, and we can see it's maintained from lines 3–6, so it's valid by induction. Moreover, we have $\texttt{covered} \subseteq U$ and hence $|\texttt{covered}| \leq n$, so the loop must terminate within n iterations. When it does, we have $\texttt{covered} = U$ and so S is a set cover.)

(e) [⋆⋆⋆] Suppose we are instead interested in the size of a *minimum* set cover, i.e. that we wish to ensure `Sol` is as small as possible. Show that GreedySetCover does not always give a minimum set cover, by giving an instance (U, S) on which it gives an answer larger than the minimum. Extend your answer to show that there are arbitrarily large instances on which GreedySetCover fails.

Solution: There are many possible answers here — this is only one of them. Let $U = [6]$ and $S = \{\{1, 2, 3\}, \{4, 5, 6\}, \{2, 3, 4, 5\}\}$. Any cover has to include $\{1, 2, 3\}$ (in order to cover 1) and $\{4, 5, 6\}$ (in order to cover 6), so the minimum cover size is at least two. Conversely, $\{\{1, 2, 3\}, \{4, 5, 6\}\}$ is a set cover, so the minimum cover size is exactly two, and the only minimum cover is $\{\{1, 2, 3\}, \{4, 5, 6\}\}$. However, GreedySetCover will choose $\{2, 3, 4, 5\}$ first, resulting in a cover of size three.

To come up with counterexamples like this for general greedy algorithms, one useful trick is to start out with the thing you want to be optimal (here the cover $\{\{1, 2, 3\}, \{4, 5, 6\}\}$), then add more choices to the input to fool the greedy algorithm into choosing suboptimally (here $\{2, 3, 4, 5\}$).

We can extend this example to arbitrarily large instances by taking a bunch of disjoint copies.

Let $t \geq 1$, and let

$$U = [6t], \qquad S = S_1 \cup S_2 \text{ where}$$
$$S_1 = \{\{1,2,3\}, \{7,8,9\}, \ldots, \{6t-5, 6t-4, 6t-3\}\},$$
$$S_2 = \{\{4,5,6\}, \{10,11,12\}, \ldots, \{6t-2, 6t-1, 6t\}\},$$
$$S_3 = \{\{2,3,4,5\}, \{8,9,10,11\}, \ldots, \{6t-4, 6t-3, 6t-2, 6t-1\}\}.$$

Then exactly as before, every set cover must include every set in S_1 (to cover $1, 7, 13, \ldots$) and every set in S_2 (to cover $6, 12, 18, \ldots$), and the only minimum cover is $S_1 \cup S_2$ which has size $2t$. But GreedySetCover will include every set in S_3 first, resulting in a cover of size $3t$. By choosing t large, we can find counterexamples as large as we like.

Of course, we could have made this example neater by e.g. including only a single set in S_3, but this trick of taking disjoint copies is a good way of "blowing up" counterexamples to most greedy algorithms, and this is the most straightforward way of applying the trick. Notice how the ratio between the answer returned by GreedySetCover and the correct answer is the same in the small counterexample and the large one — this is generally true when we construct counterexamples this way.

This question is set at roughly the difficulty of a "medium"-level counterexample-finding question on the exam. (See the unit page for what this means.)

(f) [★★★★] Now show that the set cover returned by GreedySetCover can be $\omega(1)$ times larger than the minimum set cover, i.e. larger by a factor that grows arbitrarily large as the input size increases. (**Hint:** One way of doing this has a minimum set cover with 2 sets, and chooses U with $|U| = 2 \cdot 3^t$ for any $t \geq 1$.)

Solution: Again, there are many possible answers here, and this is only one of them. Let $t \geq 1$, and let $U = \{(i,j) \colon i \in \{1,2\}, j \in [3^t]\}$. (I'm defining U this way to make the notation nicer, but you could just as well take $U = [2 \cdot 3^t]$.) Let $S = \{C_1, C_2, X_1, \ldots, X_{t-1}\}$, where

$$C_1 = \{(1,1), (1,2) \ldots, (1,3^t)\},$$
$$C_2 = \{(2,1), (2,2) \ldots, (2,3^t)\},$$
$$X_k = \{(i,j) \colon i \in [2], 3^{t-k} < j \leq 3^{t-k+1}\} \text{ for all } k \in [t-1].$$

There is a minimum cover of size 2, since $\{C_1, C_2\}$ is a cover. However, C_1 and C_2 only cover 3^t elements each, while X_1 covers $4 \cdot 3^{t-1}$, so the algorithm will first choose X_1. Of the remaining uncovered elements, C_1 and C_2 only cover 3^{t-1} elements each, while X_2 covers $4 \cdot 3^{t-2}$, so the algorithm will next choose X_2. It is not hard to prove by induction that for all $k \in [t-1]$, the k'th set the algorithm chooses is X_k, and that this leaves 3^{t-k} uncovered elements in each of C_1 and C_2 while not covering any elements in $X_{k+1} \cup \cdots \cup X_{t-1}$. The algorithm will then have to pick C_1 and C_2 anyway, in order to cover $(1,1)$ and $(2,1)$, so the total cover size is t.

Rephrasing things in terms of $|U|$, we have shown that the set cover returned by GreedySetCover can be too large by a factor of $(t+1)/2 = (\log_3(|U|/2)+1)/2 \in \Theta(\log |U|)$. Proving this isn't part of the question, but this turns out to be best possible — the algorithm does indeed always return something within $\Theta(\log |U|)$ of a minimum set cover. In fact, with a lot more work it's possible to show that the greedy algorithm works to within a factor $(1 + o(1)) \ln |U|$, and it's **impossible** to do better than this with any algorithm unless P = NP — a topic we will discuss much later in the course.

This question is set at roughly the difficulty of a "long"-level counterexample-finding question on the exam. (See the unit page for what this means.)

2. [⋆⋆] You are trying to check whether a log file contains a specific sequence of events, some of which may be duplicates. Since the log file contains records for the whole system, the events may not occur consecutively, but you know they will occur in order. Formally, you are given a *key sequence* of events $a_1, \ldots, a_m$ and a *log sequence* of events $b_1, \ldots, b_n$ with $n \geq m$, and you are able to check whether two events are equal in $O(1)$ time. Give a greedy algorithm to check whether $b_1, \ldots, b_n$ contains a *key subsequence* — indices $i_1 < i_2 < \cdots < i_m \in [n]$ such that $b_{i_j} = a_j$ for all $j \in [m]$ — and to output a key subsequence if one exists. Your algorithm should run in $O(n)$ time; prove it works.

Solution: Informally, we iterate through $b_1, \ldots, b_n$, looking for a_1. When we find it, we add it to the output and keep going, looking for a_2. When we find it, we add that to the output and keep going, looking for a_3, and so on up to a_m. If we succeed in finding $a_1, \ldots, a_m$ in $b_1, \ldots, b_n$ this way then we output their indices, and otherwise we output that no key subsequence exists. Formally, the algorithm reads as follows.

Algorithm: CHECKSUBSEQUENCE($[a_1, \ldots, a_m]$, $[b_1, \ldots, b_n]$)

```
1 begin
2     Initialise nexti ← 1 and i ← [ ].
3     foreach j ∈ [1, ..., n] do
4         if b[j] = a[nexti] then
5             Set I[nexti] = j.
6             if nexti = m then
7                 Return I.
8             Set nexti ← nexti + 1.
9     Return "key subsequence does not occur".
```

You could prove via a loop invariant that CHECKSUBSEQUENCE will output the following sequence $I_1, \ldots, I_m$, if it exists:

$$I_1 = \min\{j : a_1 = b_j\},$$
$$I_{k+1} = \min\{j > I_k : a_{k+1} = b_j\}.$$

It is clear that if $I_1, \ldots, I_m$ exists then it is a key subsequence, so it remains to prove that if a key subsequence $i_1, \ldots, i_m$ exists then so does $I_1, \ldots, I_m$. We will do so using an "exchange" argument to turn $i_1, \ldots, i_n$ into $I_1, \ldots, I_m$ (although a "greedy stays ahead" argument would also work).

We prove by induction on j that for all $0 \leq j \leq m$, we have that $I_1, \ldots, I_j, i_{j+1}, \ldots, i_m$ is a key subsequence. The base case at $j = 0$ is immediate, so suppose that $I_1, \ldots, I_j, i_{j+1}, \ldots, i_m$ is a key subsequence for some $0 \leq j \leq m-1$; then we must prove that $I_1, \ldots, I_{j+1}, i_{j+2}, \ldots, i_m$ is also a key subsequence. Since $I_1, \ldots, I_j, i_{j+1}, \ldots, i_m$ is a key sequence, we already know that

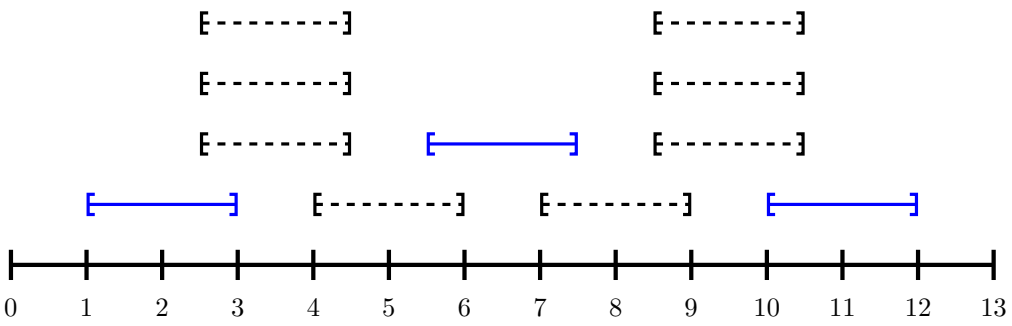
$$I_1 < I_2 < \cdots < I_j < i_{j+1} < i_{j+2} < \cdots < i_m,$$
$$b_{I_k} = a_k \text{ for all } k \in [j],$$
$$b_{i_k} = a_k \text{ for all } k \in \{j+1, \ldots, m\}.$$

It remains only to prove that I_{j+1} exists and $I_{j+1} < i_{j+2}$. Recall that I_{j+1} is defined as the least index greater than I_j satisfying $b_{I_{j+1}} = a_{j+1}$. We know from the above that $i_{j+1} > I_j$ and that $b_{i_{j+1}} = a_{j+1}$, so I_{j+1} exists and is at most i_{j+1}; in particular, this implies $I_{j+1} < i_{j+2} < \cdots < i_m$ as required.

An important note: I've made this answer quite long to try and make all the steps as easy to follow as possible. In an exam situation, I would expect much less detail. In particular, I wouldn't expect you to state the pseudocode or to prove rigorously that the informal algorithm above will output $I_1, \ldots, I_m$.

3. [⋆⋆⋆] In lectures we showed that substituting the greedy heuristic in our interval scheduling algorithm GreedySchedule with "add the compatible request with the earliest starting time" or "add the compatible request which takes least total time" breaks the algorithm. Show that the greedy heuristic "add the compatible request which renders fewest other requests incompatible" would also fail.

Solution: The heuristic will fail on e.g. the following input:



That is, $\mathcal{R} = [(1,3),(4,6),(7,9),(10,12),(2.5,4.5),(2.5,4.5),(2.5,4.5),(5.5,7.5),(8.5,10.5),(8.5,10.5),(8.5,10.5)]$. The modified algorithm will take $(5.5,7.5)$ at the first step, since everything else isn't compatible with at least three requests, and there is no maximum compatible set containing $(5.5,7.5)$. The specific numbers aren't important — what matters is the picture.

4. [⋆⋆⋆⋆] You are consulting for a trucking company that does a large amount of business shipping packages from New York to Boston. The volume is high enough that they have to send a number of trucks each day between the two locations. Trucks have a fixed limit W on the maximum weight they are allowed to carry. Boxes arrive at the New York station one by one, and each package i has a weight $w_i \leq W$. The trucking station is quite small, so at most one truck can be at the station at any time. Company policy requires that boxes are shipped in the order they arrive; otherwise, a customer might get upset upon seeing a box that arrived after his make it to Boston faster. At the moment, the company is using a simple greedy algorithm for packing: they pack boxes in the order they arrive, and whenever the next box does not fit, they send the truck on its way.

   But they wonder if they might be using too many trucks, and they want your opinion on whether the situation can be improved. Perhaps one could decrease the number of trucks needed by sometimes sending off a truck early, allowing the next few trucks to be better packed?

   Prove that this is not the case — that for any given number k of packages, the greedy algorithm they are currently using minimises the number of trucks they need subject to their other constraints. (**Hint:** Use an argument like the one we used for GreedySchedule...)

Solution: Let's denote our trucks by $T_1, \ldots, T_k$, so that our assignments of packages to trucks are functions $f\colon \{1,\ldots,k\} \to \{T_1,\ldots,T_k\}$. As in GreedySchedule, we will show that the output f of the company's greedy algorithm "stays ahead" of any optimal solution g. More specifically, we will show by induction that for all i, either:

(i) $f(i) = T_j$ and $g(i) = T_\ell$ for some $j < \ell$; or

(ii) $f(i) = g(i) = T_j$ for some j, and

$$\sum_{x \leq i\colon f(x)=T_j} w_x \leq \sum_{x \leq i\colon g(x)=T_j} w_x.$$

In other words, when package i is shipped, either f is ahead by a whole truck, or f and g are adding packages to the same truck but f's truck currently contains less weight.

For $i = 1$ this is immediate, since $f(1) = T_1$ and $g(1) = T_j$ for some $j \geq 1$. Suppose it holds for some $i \geq 1$. Then by induction we have two cases.

Case 1: $f(i) = T_j$ and $g(i) = T_\ell$ for some $j < \ell$. Then (i) holds for $i+1$ unless $\ell = j+1$ and $f(i+1) = g(i+1) = T_{j+1}$. In this case, we have

$$\sum_{x \leq i+1 \colon f(x) = T_{j+1}} w_x = w_{i+1} \leq \sum_{x \leq i+1 \colon g(x) = T_{j+1}} w_x,$$

so (ii) holds for $i+1$ and we're done.

Case 2: $f(i) = g(i) = T_j$ for some j and $\sum_{x \leq i \colon f(x) = T_j} w_x \leq \sum_{x \leq i \colon g(x) = T_j} w_x$. Since f will add package $i+1$ to truck T_j if possible, and f currently puts as most as much weight in T_j as g does, (i) holds for $i+1$ unless $f(i+1) = g(i+1)$. If $f(i+1) = g(i+1) = T_{j+1}$, then we have

$$\sum_{x \leq i+1 \colon f(x) = T_{j+1}} w_x = w_{i+1} = \sum_{x \leq i+1 \colon g(x) = T_{j+1}} w_x,$$

so (ii) holds for $i+1$ and we're done. Otherwise, we have $f(i+1) = g(i+1) = T_j$, so we have

$$\sum_{x \leq i+1 \colon f(x) = T_j} w_x = w_{i+1} + \sum_{x \leq i \colon f(x) = T_j} w_x \leq w_{i+1} + \sum_{x \leq i \colon g(x) = T_j} w_x = \sum_{x \leq i+1 \colon g(x) = T_j} w_x,$$

so again (ii) holds for $i+1$ and we're done.

5. Consider a variant of the interval scheduling problem where we have multiple "satellites" available, and wish to satisfy **all** our requests while using as few of them as possible. Formally: writing our input as $\mathcal{R} = [(s_1, f_1), \ldots, (s_n, f_n)]$, instead of finding a maximum compatible set of requests, we must partition $\mathcal{R}$ into as few disjoint compatible sets as possible.

   (a) [⋆⋆⋆⋆⋆] Prove that the following greedy algorithm returns the correct answer. (**Hint:** Rather than proving optimality directly, try to find a nice lower bound on the size of a minimum partition, and show the algorithm produces something which matches it.)

Algorithm: GreedyPartition

```
1 begin
2     Sort R according to start time, so that s_1 ≤ ··· ≤ s_n.
3     Initialise A_1, ..., A_n = [ ].
4     for i ∈ {1, ..., n} do
5         Find the least j such that (s_i, f_i) is compatible with A_j.
6         Append (s_i, f_i) to A_j.
7     Return the collection of non-empty lists A_j.
```

Solution: To avoid an annoying notation clash, write $I(a, b)$ for the real open interval $\{x \in \mathbb{R} \colon a < x < b\}$. (Normally we'd just write (a, b), but we're already using (s_i, f_i) to denote a request...)

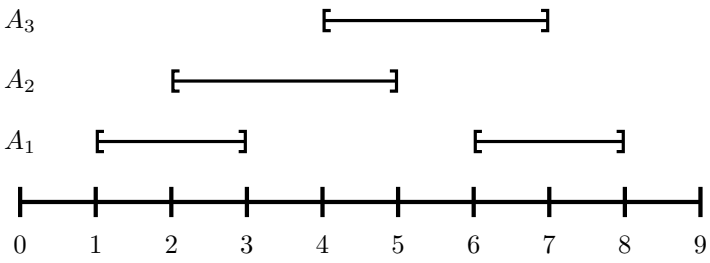
Define the *depth* d of $\mathcal{R}$ to be the maximum size of a multiset $S = \{(S_1, F_1), \ldots, (S_k, F_k)\} \subseteq \mathcal{R}$ such that $I(S_1, F_1) \cap \cdots \cap I(S_k, F_k) \neq \emptyset$. In particular, this means that every pair of requests in S is incompatible, so if we partition $\mathcal{R}$ into compatible sets then every element of S must go into a different set. Thus an optimal partition must contain at least d sets.

We will now show that GreedyPartition outputs a partition with at most d sets. In particular, this implies that the optimal partition has exactly d sets, so GreedyPartition outputs an optimal partition.

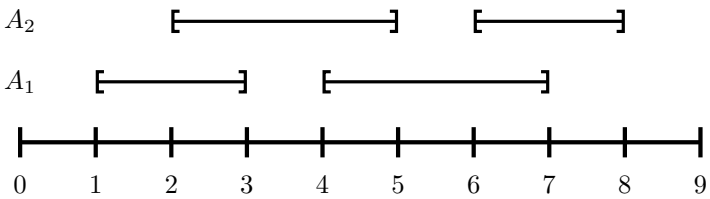
Suppose GreedyPartition assigns (s_i, f_i) to A_j. Then there are intervals $(S_1, F_1) \in A_1, \ldots, (S_{j-1}, F_{j-1}) \in A_j$ which are incompatible with (s_i, f_i). Since $\mathcal{R}$ is in sorted order, we have $S_1, \ldots, S_{j-1} \leq s_i$. Since they are incompatible with (s_i, f_i), it follows that $F_1, \ldots, F_{i-1} > s_i$. So we have $I(S_1, F_1) \cap \cdots \cap I(S_{j-1}, F_{j-1}) \cap I(s_i, f_i) \neq \emptyset$ — for example, the intersection contains the point $s_i + (\min\{F_1, \ldots, F_{j-1}, f_i\} - s_i)/2$. By the definition of d, it follows that $j \leq d$. In particular, GreedyPartition assigns every request to some list $A_1, \ldots, A_d$, so it outputs an optimal partition as required.

(b) Is the sorting step in line 2 necessary?

Solution: The sorting step is necessary — the algorithm may fail without it if e.g. $\mathcal{R} = [(1,3),(2,5),(6,8),(4,7)]$. The greedy algorithm without the sorting step will produce an output of $A_1 = [(1,3),(6,8)]$, $A_2 = [(2,5)]$, $A_3 = [(4,7)]$:



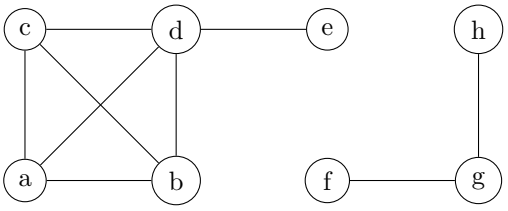
But on adding the sorting step back in we obtain an optimal partition, $A_1 = [(1,3),(4,7)]$, $A_2 = [(2,5),(6,8)]$:



2 Graph Theory

Hint: It can be extremely useful to draw a graph with the property you are trying to consider.

6. In this question we will consider the following graph G.



(a) [⋆⋆] How many components does G have?

Solution: Two, namely the induced subgraphs $G[\{a, b, c, d, e\}]$ and $G[\{f, g, h\}]$.

(b) [⋆⋆] Is $\{f, g, h\}$ a component of G?

Solution: Technically no — this is a type error. Components are graphs, and $\{f, g, h\}$ is just a set of vertices. But the graph $G[\{f, g, h\}]$ induced by $\{f, g, h\}$ is a component.

(c) [⋆⋆] What is the degree of vertex c?

Solution: Three.

(d) [⋆⋆] How many paths are there from vertex a to vertex d? List them.

Solution: Five, namely ad, abd, acd, $abcd$ and $acbd$.

(e) [⋆⋆] How many walks are there from vertex a to vertex d?

Solution: Infinitely many. For example, ad, $adad$, $adadad$ and so on are all walks from vertex a to vertex d.

(f) [⋆⋆] Draw the graph $G[\{a, b, d\}]$ induced by the set of vertices $\{a, b, d\}$.

Solution: Any triangle with vertices labelled a, b and d is fine. (Remember, how you draw a graph doesn't matter as long as all the vertices and edges are present.)

(g) [⋆⋆] Is $G[\{a, b, d\}]$ isomorphic to $G[\{f, g, h\}]$?

Solution: No. Suppose f was an isomorphism from $G[\{a, b, d\}]$ to $G[\{f, g, h\}]$. Then since isomorphisms map edges to edges, and every edge is present in $G[\{a, b, d\}]$, every edge would also have to be present in $G[\{f, g, h\}]$, which isn't true.

(h) [⋆⋆] Is $G[\{c, d, e\}]$ isomorphic to $G[\{f, g, h\}]$?

Solution: Yes. For example, the "relabelling" map $f(c) = f$, $f(d) = g$ and $f(e) = h$ maps the edge $\{c, d\}$ to the edge $\{f, g\}$, the edge $\{d, e\}$ to the edge $\{g, h\}$, and the non-edge $\{c, e\}$ to the non-edge $\{f, h\}$.

(i) [⋆⋆] Does $G[\{a, b, c, d\}]$ contain an Euler walk?

Solution: Recall from lectures that a graph contains an Euler walk if and only if it's connected and either every vertex has even degree or all but two vertices have even degree. $G[\{a, b, c, d\}]$ is connected, but contains four vertices of odd degree, so it doesn't have an Euler walk.

7. (a) [⋆⋆] Let G be a graph containing vertices u and v. Show that any walk from u to v contains a path from u to v. (**Hint:** Try induction based on the length of the walk.)

Solution: We proceed by induction on the length ℓ of the walk. Taking $\ell = 1$ as our base case, we observe that any length-1 walk from u to v contains a single edge, so it must be a path. For

the induction step, suppose that for some $\ell \geq 1$, we have that any walk from u to v of length at most ℓ contains a path from u to v. Let $W = x_1 x_2 \dots x_{\ell+2}$ be a length-$(\ell+1)$ walk from u to v; we must show that it contains a path from u to v.

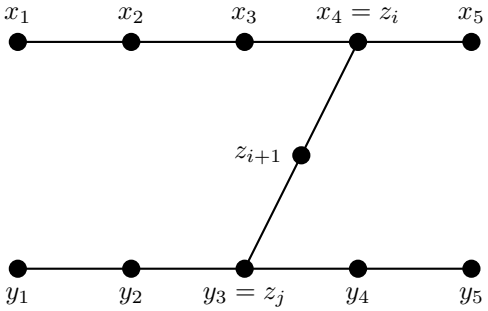
If W is a path, then we are done. If not, W must contain some vertex more than once, say as both x_i and x_j with $i < j$. Then $W^- = x_1 \dots x_i x_{j+1} \dots x_{\ell+2}$ is also a walk, since $x_i = x_j$, and it has length less than $\ell + 1$. By our induction hypothesis applied to W^-, it follows that W^- (and hence W) contains a path.

(There are many different ways of phrasing this argument, but the basic idea here is to repeatedly delete redundant segments from your walk until you're left with a path.)

(b) [⋆⋆⋆] Let G be a connected graph. Show that any two longest paths in a connected graph must have at least one vertex in common. (Here "two longest paths" means that both paths have the same length, and no other path in the graph is longer.)

Solution: Let G be a connected graph, and let $P_x = x_1 \dots x_t$ and $P_y = y_1 \dots y_t$ be two longest paths in G. Suppose for a contradiction that P_x and P_y do not intersect. Since G is connected, there is a path $Q = z_1 \dots x_r$ from x_t to y_1. We will find a path in $P_x Q P_y$ with length greater than t. (Note that $P_x Q P_y$ may not itself be a path, as Q might intersect P_x and P_y.)

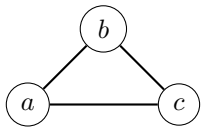
Let $i = \max\{k \colon z_k \in \{x_1, \dots, x_t\}\}$ be the index at which Q last intersects P_x. Choose I such that $x_I = z_i$. Let $j = \min\{k > i \colon z_k \in \{y_1, \dots, y_t\}$ be the first index after x at which Q intersects P_x, and choose J such that $y_J = z_j$. All vertices in $\{x_1, \dots, x_t\}$, $\{y_1, \dots, y_t\}$ and $\{z_{i+1}, \dots, z_{j-1}\}$ are distinct by construction, so we now have a situation similar to the one shown below with $t = 5$.



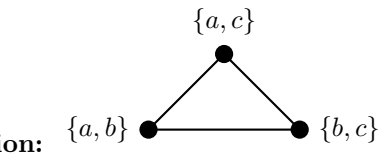
Let R_1 be whichever path is longer out of $x_1 x_2 \dots x_i$ or $x_t x_{t-1} \dots x_i$ (or the first if there is a tie). Let R_2 be whichever path is longer out of $y_1 y_2 \dots y_i$ or $y_t y_{t-1} \dots y_i$ (or the first if there is a tie). Then $R_1 z_{i+1} \dots z_{j-1} R_2$ is a path. The lengths of R_1 and R_2 are both at least $t/2$, and there is at least one edge joining them even if $j = i + 1$, so the total length of this path is at least $t + 1$. But P_x and P_y have length t and were meant to be longest paths, so this is a contradiction and they must intersect.

8. Let G be a graph. To construct the *line graph of* G, $L(G)$, we define the vertices to be the edges of G, and say that two vertices of $L(G)$ (i.e. two edges of G) are connected by the edge in $L(G)$ if they have a non-empty intersection. For example, if G has edges $\{1, 2\}$ and $\{3, 4\}$, then the vertex set of $L(G)$ will be $\{\{1, 2\}, \{3, 4\}\}$ and the edge set of $L(G)$ will be empty.

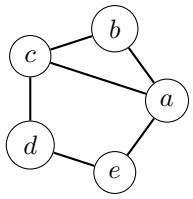
(a) [⋆⋆] For each of these graphs G, draw its line graph $L(G)$.

i. 

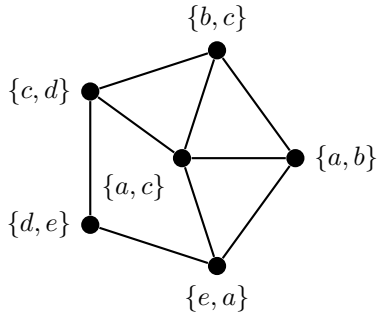
Solution:



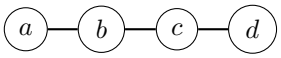
ii.



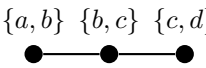
Solution:



iii.

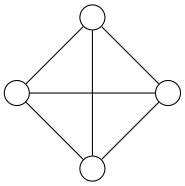


Solution:



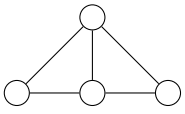
(b) [⋆⋆⋆] For each of the following graphs H, give a graph G such that $L(G)$ is isomorphic to H.

i.



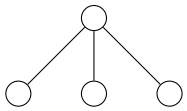
Solution: Many graphs have this line graph, but one example has vertex set $[5]$ and edge set $\{\{1,2\},\{1,3\},\{1,4\},\{1,5\}\}$.

ii.



Solution: The only graph with this line graph (up to isomorphism) is the graph with vertex set $[4]$ and edge set $\{\{1,2\},\{1,3\},\{2,3\},\{1,4\}\}$, i.e. a triangle with an edge hanging off.

(c) [⋆⋆⋆] The claw graph is shown below. Show that if a graph H is a line graph, i.e. if $H = L(G)$ for some graph G, then H doesn't contain any induced subgraph isomorphic to the claw. (**Hint:** First show that the claw itself is not isomorphic to any line graph.)



Solution: We first show that the claw graph itself is not isomorphic to any line graph. Suppose that G is a graph with $L(G)$ isomorphic to the claw graph. Let e_1 be the edge of G with degree 3 in $L(G)$, and let e_2, e_3 and e_4 be the other edges of G. Then e_2, e_3 and e_4 must all intersect e_1 (since they are joined to it by edges in $L(G)$), but cannot intersect each other (since they are not joined to each other by edges in $L(G)$). This is impossible — e.g. by the pigeonhole principle, the three sets $e_2 \cap e_1$, $e_3 \cap e_1$ and $e_4 \cap e_1$ are all contained in e_1, and $|e_1| = 2$, so some two must overlap. We have shown that the claw graph is not isomorphic to any line graph.

Now let H be a graph with an induced subgraph isomorphic to a claw, say $H[F]$ for some $F \subseteq V(H)$, and suppose that $H = L(G)$ for some G. Then consider the subgraph G^- of G formed by deleting all edges except those in F. We will have $L(G^-) = H[F]$, and so $H[F]$ is a line graph — which we have already shown is impossible.

We have shown that all line graphs are *claw-free*. Claw-free graphs are important because a lot of problems have efficient algorithms for claw-free graphs, but not for general graphs.

9. [⋆⋆⋆] A *closed* Euler walk is an Euler walk from a vertex to itself. Suppose G is a graph with exactly two connected components C_1 and C_2, each of which has more than one vertex. Suppose the graphs induced by C_1 and C_2 each have a closed Euler walk. What is the least number of edges we can add to G to give it a **closed** Euler walk?

Solution: The answer is three edges unless both C_1 and C_2 are cliques (i.e. every vertex is adjacent to every other vertex), in which case it becomes four edges.

We first show that four edges always suffice. Recall from lectures that a graph has a closed Euler walk if and only if it is connected and each of its vertices has even degree; hence every vertex of G has even degree. Let $a, b \in V(C_1)$ and $c, d \in V(C_2)$ be distinct; then we form H from G by adding the edges $\{a, c\}$, $\{a, d\}$, $\{b, c\}$ and $\{b, d\}$. For all $v \in V(H)$, we have

$$d_H(v) = \begin{cases} d_G(v) & \text{if } v \notin \{a, b, c, d\}, \\ d_G(v) + 2 & \text{otherwise,} \end{cases}$$

so every vertex of H has even degree. Moreover, the edge $\{b, c\}$ connects C_1 to C_2, so H is connected. Thus H has a closed Euler walk.

If in addition C_1 (say) is not a clique, then we can take a and b to be non-adjacent, and instead add the edges $\{a, b\}$, $\{b, c\}$ and $\{a, c\}$, which works for the same reasons.

We now show that three or four edges are necessary. Suppose H is formed by adding edges to G and that H has an Euler walk. This implies that H is connected, so $|E(H)| \geq |E(G)| + 1$; let $\{u, v\}$ be an edge in $E(H) \setminus E(G)$. Then $d_H(u) > d_G(u)$ and $d_H(v) > d_G(v)$. Since H, C_1 and C_2 all have closed Euler walks, all vertex degrees in H and G are even, so we have $d_H(u) \geq d_G(u) + 2$ and $d_H(v) \geq d_G(v) + 2$. Since G cannot have more than one edge between u and v, it follows there must be at least two more edges $\{u, x\}$ and $\{v, y\}$ between C_1 and C_2, so $|E(H) \setminus E(G)| \geq 3$. If in addition C_1 and C_2 are cliques, then we cannot have $x = y$, and so we have added one edge incident to each of x and y; since their degrees in G are even, it follows that their degrees in H will will be odd unless at least one more edge is present. Since H contains a closed Euler walk, all its vertices must have even degree, so $|E(H) \setminus E(G)| \geq 4$ as required.

10. The *complement* of a graph $G = (V, E)$ is the graph $G^c = (V, \overline{E})$, where $\overline{E} = \{\{u, v\} \colon u, v \in V, u \neq v\} \setminus E$.

A graph is *self-complementary* if it is isomorphic to its complement. Show that:

(a) [$\star$] a four-vertex path and a five-vertex cycle are both self-complementary;

Solution: A four-vertex path is isomorphic to the graph G with $V(G) = [4]$ and $E(G) = \{\{1,2\},\{2,3\},\{3,4\}\}$. Its complement G^c has edge set $\{\{3,1\},\{1,4\},\{4,2\}\}$, again a four-vertex path. An isomorphism is given by $1 \mapsto 3$, $2 \mapsto 1$, $3 \mapsto 4$ and $4 \mapsto 2$.

A five-vertex cycle is isomorphic to the graph G with $V(G) = [5]$ and $E(G) = \{\{1,2\},\{2,3\},\{3,4\},\{4,5\},\{5,1\}\}$. Its complement G^c has edge set $\{\{1,3\},\{3,5\},\{5,2\},\{2,4\},\{4,1\}\}$, again a five-vertex cycle. An isomorphism is given by $1 \mapsto 1$, $2 \mapsto 3$, $3 \mapsto 5$, $4 \mapsto 2$, $5 \mapsto 4$. Both of these should be obvious if you draw pictures.

(b) [$\star\star\star$] every self-complementary graph is connected;

Solution: Suppose that G is disconnected, and let its components be $G_1, \ldots, G_r$. Then no edges between $V(G_1), \ldots, V(G_r)$ are present in G, so all edges between $V(G_1), \ldots, V(G_r)$ must be present in G^c. Then G^c is connected; indeed, let u and v be two vertices of G^c. If they lie in two different components of G, then they are joined by an edge in G^c. Otherwise, they lie in the same component, and have a common neighbour in some other component. We have shown that if G is disconnected, then G^c is connected. Since $(G^c)^c = G$, it also follows that if G^c is disconnected then G is connected.

Now suppose G is self-complementary. This means G is isomorphic to G^c, so they are either both connected or both disconnected. We've just shown they can't both be disconnected, so they must both be connected.

(c) [$\star\star\star\star$] if G is self-complementary, then $|V(G)| \equiv 0$ or $1 \bmod 4$;

Solution: If G is self-complementary then since G and G^c are isomorphic we must have $|E(G)| = |E(G^c)|$. But by the definition of the complement, we have $|E(G^c)| = \binom{n}{2} - |E(G)|$. It follows that $|E(G)| = |E(G^c)| = \frac{1}{2}\binom{n}{2}$. In particular, this means that $\frac{1}{2}\binom{n}{2} = n(n-1)/4$ must be an integer, which holds if and only if $n \equiv 0$ or $1 \bmod 4$.

(d) [$\star\star\star\star$] every self-complementary graph on $4k+1$ vertices has a vertex of degree $2k$.

Solution: For all positive integers i, let N_i be the number of vertices of degree i in G, and let N_i^c be the number of vertices of degree i in G^c. By the definition of the complement, we have $N_i = N_{4k-i}^c$; moreover, since G is isomorphic to G^c, we have $N_i = N_i^c$. It follows that $N_i^c = N_{4k-i}^c$, and likewise $N_i = N_{4k-i}$.

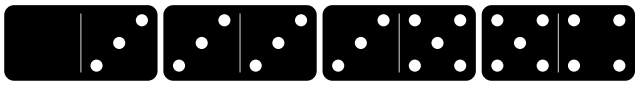
Suppose for a contradiction that G has no vertices of degree $2k$. Then since every vertex has some degree, and there are $4k+1$ vertices in total, we have

$$4k+1 = \sum_{i=0}^{4k} N_i = 2\sum_{i=0}^{2k-1} N_i + N_{2k} = 2\sum_{i=0}^{2k-1} N_i.$$

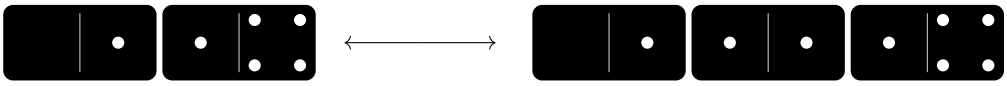
But the left-hand side of this equation is odd and the right-hand side is even, so we have our contradiction.

11. [$\star\star\star$ and a half] A *numbered domino* is a rectangle divided into two halves, with a number on each half. A standard "double six" set of numbered dominoes contains one domino with each possible pair of numbers from zero to six, for a total of 28. Is it possible to lay them all out in a line so that each adjacent pair of dominoes agrees, as shown below for four dominoes? What about a "double k" set,

which contains one domino with each possible pair of numbers from zero to $k \in \mathbb{N}$? (**Hint:** I will never ask a question like this unless there's a way to solve it quickly with pencil and paper.)



Solution: First note that we can ignore all dominoes with the same number on both sides, as we can add or remove these freely from a solution not containing them as shown below.



Consider the complete graph on the $k+1$ vertices $\{0, \ldots, k\}$, in which every vertex is joined to every other vertex. There is a natural bijection between edges and the remaining dominos: the edge $\{i, j\}$ corresponds to the domino with i on one face and j on the other. A sequence of edges is a valid walk in the graph if and only if the corresponding dominoes have matching ends. A valid line of dominoes therefore corresponds to a walk in the graph using every edge exactly once — an Euler walk.

We know from lectures that a graph has an Euler walk if and only if it is connected and either zero or two of its vertices have odd degree. Our graph is certainly connected, and all of its $k+1$ vertices have degree k, so it has an Euler walk if and only if $k = 1$ or k is even. In particular, it is possible to lay out the dominoes from a "double six" set as described.

12. [★★★★★] Give an example of a self-complementary graph (see Question 3) with infinitely many vertices.

Solution: One way of doing this is to take a four-vertex path, a graph which we already know is self-complementary from question 3a), and "blow it up" by replacing each vertex with an infinite set of vertices. For example, for all $i \in \{0, 1, 2, 3\}$, define V_i to be the set of all integers congruent to i modulo 4. Then we define our edge set to be:

- all edges internal to V_1 or V_2;
- all edges between V_0 and V_1;
- all edges between V_1 and V_2;
- all edges between V_2 and V_3.

The complement will then have edge set given by:

- all edges internal to V_0 or V_3;
- all edges between V_2 and V_0;
- all edges between V_0 and V_3;
- all edges between V_3 and V_1.

These two graphs are then isomorphic under any map taking $V_0 \mapsto V_2$, $V_1 \mapsto V_0$, $V_2 \mapsto V_3$ and $V_3 \mapsto V_1$. (Draw a picture!)

Here's a more complicated example. Take the vertex set to be $\mathbb{Z}$, and define the edge set by tossing a coin for every pair of vertices and joining them if it comes up heads — that is, each edge is present independently with probability 1/2. Then it turns out that with probability 1, the resulting graph $G_{\infty,1/2}$ is self-complementary. The **really surprising** thing about this, though, is that you can replace 1/2 with any constant $0 < p < 1$ and it still works — all these infinite graphs are isomorphic to each other with probability 1, even though we'd expect a graph where each edge is present with probability 1/100 to be much "sparser" than one where each edge is present with probability 99/100!