CO150 - Graphs and Algorithms

Prelude

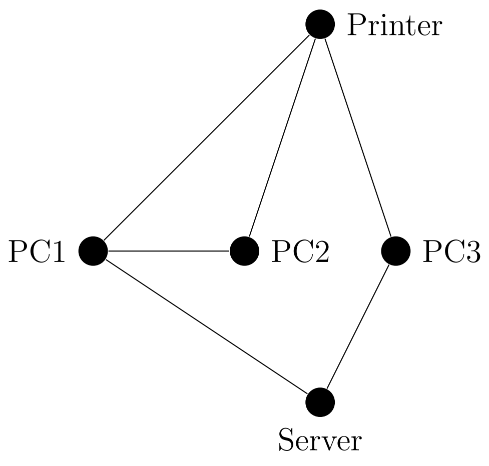
The content discussed here is part of CO150 - Graphs and Algorithms (Computing MEng); taught by Iain Phillips, in Imperial College London during the academic year 2018/19. The notes are written for my personal use, and have no guarantee of being correct (although I hope it is, for my own sake). This should be used in conjunction with the notes.

14th January 2019

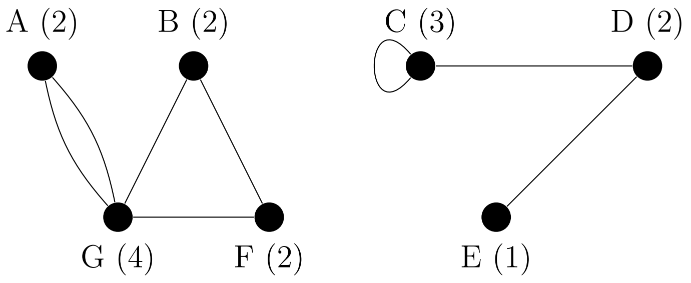
Introduction to the structure of the course;

Part I:	Graphs
Part II:	Graph Algorithms
Part III:	Algorithm Analysis
Part IV:	Introduction to Complexity

An example graph with a real life application;



Note how all the PCs are directly connected to the printer, but PC2 can only reach the server through PC1. On the other hand, we can create a more general graph to display some features that may be less common;



Note that this isn't actually two graphs; it's **disconnected components**. Between A, and G, there are two **parallel arcs / edges**, and C has **loop** with itself. I will continue to refer to this graph as the "example", for the remainder of this section, since it displays properties which we may want to analyse later.

We can say that the left subgraph is robust, as it will remain connected against a single failure. However, the right subgraph isn't robust, as a failure between C, and D, or between E, and D would cause one of the nodes to become disconnected. We can then remedy this by adding a connection between C, and E.

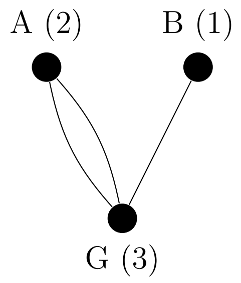
In the graph drawn above, the degrees are also specified - which is the number of arcs connected to it. Note that the degree of C is 3, as we count loops twice for consistency reasons. The sum of the degrees is 16, which is double the number of arcs (8). This is because each arc is counted twice (where it starts, and where it ends), therefore the sum of the degrees is always even. From that, we can then infer that the number of odd nodes (C, and E in our case) must be even. This is trivial to prove with arithmetic.

Subgraphs

We can say that G_1 is a subgraph of G_2 if both of the following criteria apply;

- $\text{nodes}(G_1) \subseteq \text{nodes}(G_2)$
- $\text{arcs}(G_1) \subseteq \text{arcs}(G_2)$

A full (induced) subgraph occurs when we have a set of nodes, X, such that $X \subseteq \text{nodes}(G)$. Every connection between the nodes in X, that was present in G, exists in $G[X]$. Then G' is a full subgraph of G, if $G' = G[X]$ for some X. For example, let $X = \{A, B, G\}$, from the example graph, then we have the following induced subgraph;



If we have some subgraph G', and $\text{nodes}(G') = \text{nodes}(G)$, then it G' spans G.

Adjacency Matrix

For the entry in the matrix $a_{i,j}$, it represents the number of arcs thast connect i to j. In an undirected graph, this matrix is symmetric (such that $a^\top = a$). In our example, we're doing the rows, and columns, alphabetically. It's also important to note that we count each loop twice in a diagonal entry. We can determine the degree of a node by taking the sum of its respective row (or column), and find the number of arcs by taking the sum of all the values in the matrix, and then halving it.

$$\begin{array}{c} A \\ B \\ C \\ D \\ E \\ F \\ G \end{array}\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 2 & 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Adjacency Lists

You'll notice that in our graph, we have a lot of 0s, which makes it less efficient to store as an adjacency matrix; especially if we don't require random access to the degrees. We tend to use n to represent the number of nodes (vertices), and m to represent the number of arcs (edges). You'll note that the size of this is $\leq n + 2m$ (as we have n nodes on the left, and each arc is counted twice, except for loops). Therefore, we can say a graph is sparse if $2m \ll n^2$. Since certain algorithms we work with only look at the arcs incident to a given node, a linked list will be better for sparse graphs.

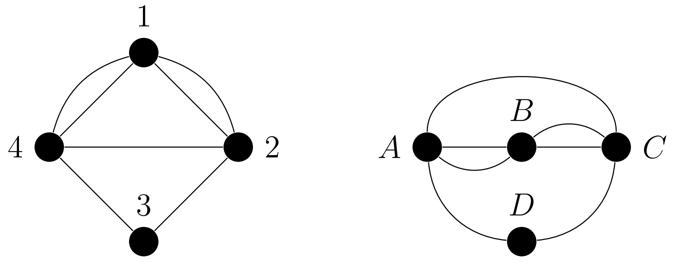
A	→ G, G
B	→ F, G
C	→ C, D
D	→ C, E
E	→ D
F	→ B, G
G	→ A, A, B, F

Big-Oh Notation

I'm too lazy to write out the example, but the idea is that we ignore constant factors, and only consider the most significant term; for example, we could summarise some algorithm that takes $3n^4 + 2n - 4631$ to run as $O(n^4)$. This has significant advantages, since it allows us to abstract away from the implementation / hardware specifics, and instead focus on the factors which determine growth.

Isomorphism

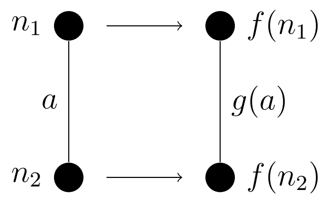
In general, an isomorphism is a bijection that preserves connections. While the two graphs drawn below appear fairly different, they are isomorphic. Mapping from the left, to the right, we know that $3 \mapsto D$, simply because they are the only nodes with degree 2. It's also evident that $1 \mapsto B$, as it's the only node which has two sets of parallel arcs coming out of it. However, it doesn't matter which of 4, or 2, maps to A, or C. Therefore, we can say $4 \mapsto A, 2 \mapsto C$, or $4 \mapsto C, 2 \mapsto A$.



While we're able to check this fairly easily by simply looking at the graph, a computer would have to rearrange the LHS' adjacency matrix to the RHS' (or vice versa).

Given two graphs, G, G', an isomorphism from G to G' consists of two bijections (one-to-one mapping), as well as an additional restriction;

- f : nodes$(G) \mapsto$ nodes(G')
- g : arcs$(G) \mapsto$ arcs(G')
- if $a \in$ arcs(G), with endpoints n_1, n_2, then the endpoints of $g(a)$ are $f(n_1), f(n_2)$ (see the diagram below for a visual example).



In order to confirm whether two graphs are isomorphic, the easiest approach is to first check the obvious; whether the number of arcs, nodes, and loops are the same, as well as the degrees of the nodes. If any of these are different, then the graphs cannot be isomorphic. However if they pass all the tests, then we can attempt to find a bijection on the nodes.

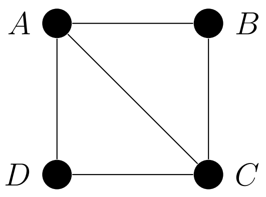
17th January 2019

Complexity

Generally, the process of determining whether two graphs are isomorphic is computationally expensive, hence it has a high complexity. A naive approach would be to check all the permutations, which would then lead to a time complexity of $O(n!)$, which is worse than even exponential ($O(2^n)$).

Automorphisms

An automorphism on G is an isomorphism from G to itself. Every graph has at least one automorphism (the identity). Consider the following graph;

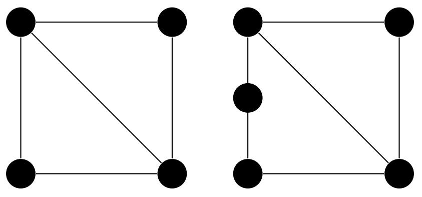


We can do the following method to find the number of automorphisms;

- fix a node, B, it can go to where D is, or stay (2 possibilities)
- take the next node A, it can either stay where it is, or go to where C is (2 possibilities), now fix it
- take the next node C, it can only stay where it is, as it can't go to D since D isn't connected to B, nor does it have a degree of 3 (1 possibility), now fix it
- finally D can only stay where it is (1 possibility)
- multiply all the possibilities, and we have 4 automorphisms

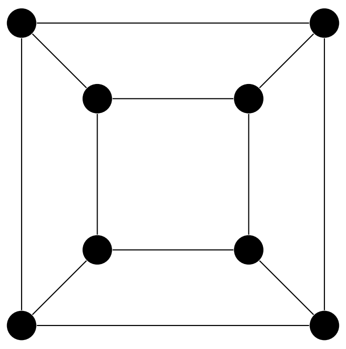
Planar Graphs

We can say a graph is planar if it can be drawn such that no arcs cross. Any non-planar graph contains K_5, or $K_{3,3}$ as a subgraph homeomorphic. We can say that two graphs are **homeomorphic** if they can be obtained by a series of operations such that an arc $x - y$, is replaced by two arcs $x - z$, and $z - y$. For example, the two graphs below are homeomorphic.



There is a linear time algorithm to check whether a graph is planar; however in this case linear time means $O(n + m)$, with the previous definitions.

Any planar graph splits the plane into regions, which are referred to as faces; the graph below splits it into 6 faces (including the outside region). With a graph G that has N nodes, A arcs, and F faces, Euler's formula states $F = A - N + 2$ for any connected planar graph.



Graph Colouring

Any (literal, real-life) map can be convereted into a simple planar graph by letting the countries represent nodes, and joining them if they are neighbours. This newly generated graph is known as the dual graph. We can say some graph G is k-colourable, if the nodes of G can be coloured with no more than k colours, therefore every simple planar graph is 4-colourable.

Bipartite Graphs

We can say a graph is bipartite if we can partition nodes(G), into two sets X, and Y, such that no two nodes of X are joined, and likewise for Y. A graph is biparite $\Leftrightarrow$ it is 2-colourable.

Paths, and Connectedness

A path in a graph is a sequence of adjacent arcs, although normally described by the nodes that we pass through. A path is called **simple** if it doesn't repeat nodes, and a graph is **connected** if there is a path joining any two nodes.

We can define a relation on nodes(G) by $x \sim y \Leftrightarrow$ there is a path from x to y. This is an equivalence relation, as we can prove it's reflexive, symmetric, and transitive.

- $\forall x \in \text{nodes}(G)[x \sim x]$, x is trivially connected to itself, hence it is reflexive
- $\forall x, y \in \text{nodes}(G)[x \sim y \Rightarrow y \sim x]$, as we are working on an undirected graph, this follows trivially
- $\forall x, y, z \in \text{nodes}(G)[x \sim y \land y \sim z \rightarrow x \sim z]$, follows trivially by definition of paths

A cycle (circuit) is a special type of path that finishes where it starts, has at least one arc, and doesn't reuse an arc. A graph which doesn't have cycles is **acyclic**.

21st January 2019

Euler Paths / Circuits

An Euler path is a special type of path where each arc is used exactly once, and an Euler circuit is a cycle which uses each arc exactly once (therefore an EC is an EP which finishes at the start node). A connected graph has an EP $\Leftrightarrow$ there are 0, or 2 odd nodes, and there is an EC $\Leftrightarrow$ there are no odd nodes.

We can justify it by saying that any intermediate node (ones which aren't the start node) have to be entered, and exited the same number of times (otherwise it wouldn't be an intermediate) node. Therefore, if 2 nodes of odd degree, then it follows that we start from one, and end on the other.

Consider the following nodes; n, n' being the start, and end (the odd nodes of the path), and arbitrary intermediate nodes i. Start at n, and keep going until we can go no further (n'). If we've stopped at n, then there must be a spare arc, as we've started, and 'ended' at an odd node. If we stop at some arbitrary $i \neq n'$, then we've still got more arcs, since i is even.

Hamiltonian Path / Circuits

A Hamiltonian path is one that visits each node exactly once, and similarly a Hamiltonian circuit returns to the start node. For this, we will only consider simple graphs, since we won't ever follow a loop, or a parallel arc. In order for there to be a HP, we need a connected graph, and for a HC to exist, each node must have a degree ≥ 2. To determine whether a circuit exists, we can take a brute force approach, since a circuit is really just a permutation on the set of nodes; such that for n nodes, we have $\pi : \{1, ..., n\} \mapsto \{1, ..., n\}$. However, for this to be a circuit, we need $\pi(i)$ to be adjacent to $\pi(i + 1)$, and so on. As we have $n!$ possible circuits, this is far too slow. There exists a dynamic programming approach that reduces this to $O(n^2 2^n)$, but that is still exponential. Compared to EPP, which has $O(n^2)$ time. HCP has been shown to be NP-complete, and are therefore not solvable in polynomial time.

Trees

A tree is an acyclic connected graph (whether we specify it's rooted, or nonrooted, depends on the author). The root of G is a distinguished node. Assuming a rooted graph, the depth of a node x is the distance along the unique path from the root to x. If x isn't the root node, the parent of x is the node directly before it in the path from the root to x. The depth of tree is the maximum of the depths of all its nodes.

A spanning tree of a graph G, is a tree, T, such that nodes(T) = nodes(G), and arcs$(T) \subseteq$ arcs(G). The spanning trees are not necessarily unique.

Directed Graphs

While we generally cover undirected graphs in this course, it makes sense in some applications for the arcs to be directed. For each $a \in$ arcs(G), it is associated with an **ordered** pair of nodes. In diagrams, these are shown with arrows. In a path for $a_1, ..., a_n$, the source of a_{i+1}, must match the target of a_i. We define the indegree as the number of arcs entering, and likewise the outdegree is the number of args leaving. For any directed graph, the sum of the indegree of all nodes, and the outdegree of all nodes is equal to the number of arcs. We say a directed graph is strongly connected if there exists a path between any two nodes in G.

24th January 2019

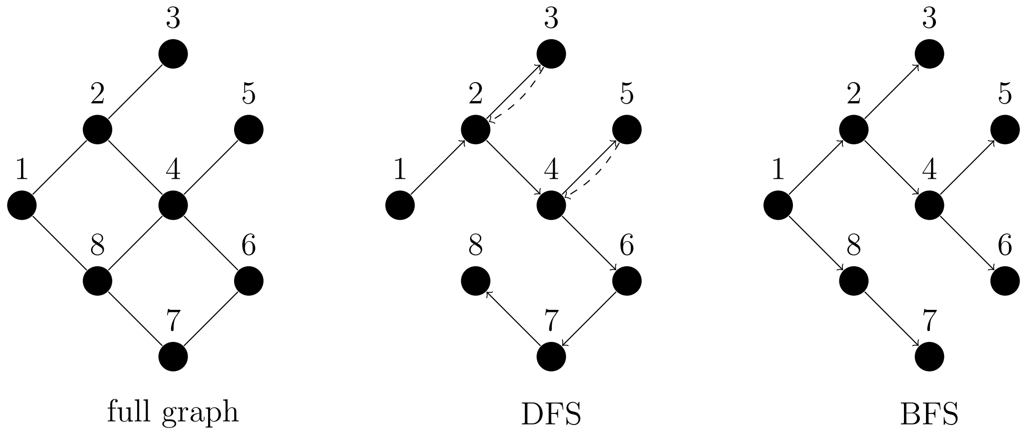
Tree Traversal Algorithms

The two types of traversal covered in this course are depth-first search (DFS), and breadth-first search (BFS). While they are similar in many ways, they also have quite a few differences.

In depth first search, we choose one of the adjacent nodes to the start; from there, we then spawn another depth first search. This is done recursively until there aren't any more (unvisited) adjacent nodes. At this point, the spawned DFS returns back to the parent node, where it checks the next adjacent node (normally ordered by how the adjaceny list / matrix is stored). It does this until we have visited all of the nodes, and then returns back all the way to the start.

In contrast to DFS, breadth-first search goes through all the adjacent nodes, and then goes deeper. This means that we only check the nodes adjacent to the ones adjacent to the start node, after all of the nodes directly adjacent to the start node have been visited.

You'll note that the distance between 1, and 8, is 5 in DFS, but only 1 in BFS. In BFS, the depth of any node is its distance from the start. However, both generate spanning trees on G.



In order to formalise this, let us consider the graph to be traversed as an adjaceny list, a boolean array of nodes (which are visited), and the parent is the parent node in the search tree. The output will be the nodes visited in order.

```
procedure dfs(x):
  visited[x] = true
  print x
  for y in adj[x]:
    if not visited[y]:
      parent[y] = x;
      dfs(y)
      # at this point, control is returned to x
      # we don’t need the parent in this case, but other applications may use it
```

The running time of DFS is $O(n + m)$, therefore it’s linear. However, this implementation may have some overhead due to recursion.

```
procedure bfs(x):
  visited[x] = true
  print[x]
  enqueue(x, Q)
  while not isempty(Q):
    y = front(Q)
    for z in adj[y]:
      if not visited[z]:
        visited[z] = true
        print z
        parent[z] = y
        enqueue(z, Q)
    dequeue(Q)
```

The size of the queue represents the breadth of the front. Once again, the time complexity is $O(n+m)$.

Applications of Traversal

The algorithms used above also work on non-connected graphs. If we analyse the set of visited nodes, and see that it isn’t the same as the set of all nodes, then it is clear that the graph is not connected. As such, we have an $O(n + m)$ algorithm for detecting non-connected graphs.

We can trivially say that a graph has a cycle if it has $\geq n$ arcs. Alternatively, we can use DFS; if we encounter a node that has already been visited (other than by backtracking), then it has a cycle.

It’s also trivial to modify BFS to find the distance of each node, by having a running counter. Due to how BFS is implemented, we can also extract the shortest path from `y`; as `y, parent[y], parent[parent[y]], ..., start`.

Weihted Graphs

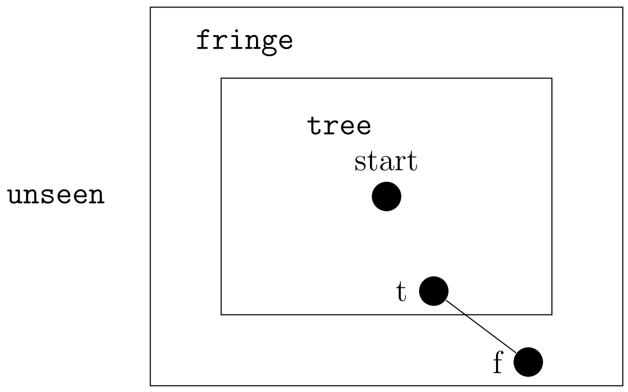
We can associate a cost with each arc on a network. We can define a weighted graph as a graph, G, with a weight function $W : \text{arcs}(G) \mapsto \mathbb{R}^+$. With weights, we’re able to consider the following problems; finding an MST (minimum spanning tree), finding shortest paths, and finding a shortest circuit.

We’re only going to consider simple graphs, as there is no point in taking a loop if we’re trying to minimise cost, nor is there any point in taking the more expensive arc in a parallel arc.

MST Algorithms

We can say that T is a minimum spanning tree for Gm if T is a spanning tree for G, and no other spanning tree for G has a smaller weight. Once again, MSTs do not have to be unique.

The main idea in Prim’s algorithm is to add the shortest arc that will extend the tree. This is an example of a greedy algorithm, which gives a short-term advantage byt may not be the best overall (it is in this case). At any stage in Prim's algorithm, we have three types of nodes; tree nodes (which are already part of the MST), candidate nodes - which are fringe nodes adjacent to a tree node, and the rest are unseen nodes. At the start, all nodes are unseen.



The general idea is to pick a random node (doesn't matter which, as all nodes will be in the MST by definition). This is now the start node, and therefore part of the MST (hence classified as `tree`). Now classify all the nodes adjacent to this node as `fringe`. While the fringe isn't empty, select the arc with minimum length between a tree node t, and a fringe node f. Classify f as `tree`, and add the arc (t, f) to the tree. Reclassify all the unseen nodes adjacent to f as `fringe`.

The while loop is executed roughly n times, as it will be executed for each node. However, in the worst case, when we're selecting a minimum arc, that takes $n + m$. Therefore, we can calculate this to be an $O(n(n+m))$ algorithm. This is a more naive approach, we can improve this by choosing candidate arcs; as we're avoiding redoing work. If we consider a parent function, such that the parent of f (`fringe`) is t (`tree`), such that the arc (t, f) has the least weight. This can be summarised in two parts; first the initialsation, and then the execution of the algorithm;

```
tree[start] = true
for x in adj[start]:
  fringe[x] = true
  parent[x] = start
  weight[x] = $W$(start, x)
while not isempty(fringe):
  select f such that weight[f] is minimum
  fringe[f] = false
  tree[f] = true
  for y in adj[f]:
    if not tree[y]:
      if fringe[y]:
        # updating arcs if we can get a lower weight
        if $W$(f, y) < weight[y]:
          weight[y] = $W$(f, y)
          parent[y] = f
      else:
        # we haven't seen y yet
        # this can probably be shortend with the above
        fringe[y] = true
        weight[y] = $W$(f, y)
        parent[y] = f
```