COMP20007 Design of Algorithms

Week 4 Workshop Solutions

Tutorial

1. Subset-sum problem First, we'll cover some terminology. The *power set* of a set S – often denoted $\mathcal{P}(S)$ – is the set of all subsets of S, *e.g.,* if $S = \{1, 2, 3\}$ then,

$$\mathcal{P}(S) = \big\{\{\}, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}.\big\}$$

If a set has $|S|$ elements then $\mathcal{P}(S)$ has $2^{|S|}$ elements. An exhaustive-search approach to this subset-sum problem will include iterating over all subsets of S, and checking whether their sum is t:

function SubsetSum(S, t)
  for each S' in Powerset(S) **do**
    if $\sum_{i \in S'} i = t$ **then**
      return Yes
  return No

Since $|S| = n$, we know that there are 2^n power sets to iterate through. For each power set we must compute a sum, which will be an $O(n)$ operation. So the runtime of this subset sum problem is $O(n2^n)$.

How can we systematically generate all subsets of the given set S? For each element $x \in S$, we need to generate all the subset of S that happen to contain x, as well as those that do not. This gives us a natural recursive algorithm:

function Powerset(S)
  if $S = \emptyset$ **then**
    return $\{\emptyset\}$
  else
    $x \leftarrow$ some element of S
    $S' \leftarrow S \setminus \{x\}$
    $P \leftarrow$ Powerset(S')
    return $P \cup \{s \cup \{x\} \mid s \in P\}$

2. Partition problem At first it might seem we have to do something more sophisticated than in the subset-sum problem, however this problem reduces nicely to this problem.

First we'll notice that if there are sets A and B which partition S and have equal sum they'll have to satisfy the following two equations:

$$\sum A + \sum B = \sum S \tag{1}$$

$$\sum A = \sum B \tag{2}$$

So we can see that $\sum A = \sum B = \frac{1}{2} \sum S$ would have to hold. First, if $\sum S$ is odd we know the answer is No (given our elements are integers). Second, if such sets A and B exist, any set with sum $\frac{1}{2} \sum S$ will do. So:

function HasPartition(S)
  $sum \leftarrow \sum_{i \in S} i$
  if sum is odd **then**
    return No
  return SubsetSum($S, sum/2$)

3. Graph representations

(a) Adjacency lists:

A → B, C
B → A, C
C → A, B, D
D → C
E →

Adjacency matrix:

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

Sets of vertices and edges:

$G = (V, E)$, where $V = \{A, B, C, D, E\}$, and $E = \{\{A, B\}, \{A, C\}, \{B, C\}, \{C, D\}\}$

Degree is the number of edges connected to a vertex. Node C has the highest degree (3).

(b) Adjacency lists:

A → B, C
B → A
C →
D → A, C
E →

Adjacency matrix:

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

Sets of vertices and edges:

$G = (V, E)$, where $V = \{A, B, C, D, E\}$, and $E = \{(A, B), (A, C), (B, A), (D, A), (D, C)\}$

For a directed graph, degree is separated into *in*-degree and *out*-degree: the number of edges going into or out of each vertex, respectively. Nodes A and C have the highest in-degree (2).

4. Graph representations continued Note that in this question we assume that self-loops are not allowed.

Additionally, we could always check whether a graph is constant time if we know (in constant time) the number of edges. If the number of edges is exactly $\binom{n}{2} = n(n-1)/2$ then the graph is complete, otherwise it is not.

(a) Determining whether a graph is *complete*.

(i) In the adjacency list representation we just want to check that for each node u, every other node is in its adjacency list.

If we can check the size of the list in $O(1)$ time then we just need to go through each vertex and check that the size of the list is $n - 1$, and thus this operation will be linear in the number of nodes: $O(n)$.

If we have to do a scan through the adjacency list then this would take $O(m)$, which for a complete graph will be asymptotically equivalent to $O(n^2)$.

(ii) In the adjacency matrix representation, a complete graph would contain 1's in all position except for the diagonal (since we can not have self loops). So there will be $n^2 - n$ positions in the matrix to lookup, and thus this operation will take $O(n^2)$ time.

(iii) For the graph to be complete the set of edges must contain each pair of vertices. There will be $\binom{n}{2}$ edges if the graph is indeed complete, so we could just check the size of the edge set E. This would be a constant time operation: $O(1)$.

However if we can't just check the size of the set we'll have to look through all edges which will take $O(m)$ time.

(b) Determining whether the graph has an *isolated node*.

We'll have a think about how long it takes to determine if a single node is isolated, and then apply that to each vertex (*i.e.*, n times)

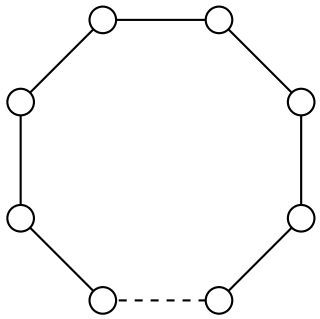
(i) In the adjacency list representation, a node is isolated if its adjacency list is empty. This is $O(1)$ to check per node, so $O(n)$ for the whole graph.

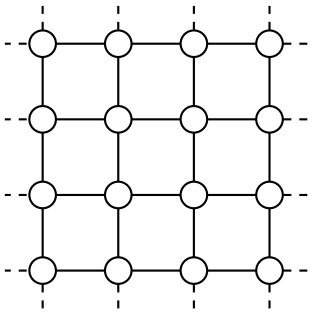
(ii) In the adjacency matrix representation to check if a node is isolated we must look at each entry in that node's row or column and confirm that all entries are 0. This is $O(n)$ per node so $O(n^2)$ in total.

(iii) In the sets of vertices and edges representation we can loop through the set of edges and confirm that a vertex does not appear. This will take $O(m)$ for a single node.

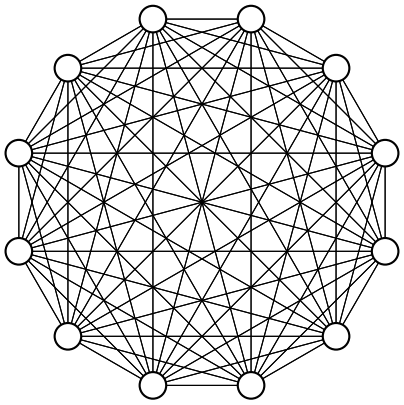
However we can also check a whole graph in a single pass by keeping track of all the nodes at once (in an array or a hash table for instance) and ticking them off as we see them, at the end we can iterate through this array/hash table to check if there were any isolated nodes. So this will take $O(n + m)$ time.

5. Sparse and dense graphs (optional) Graphs with a constant number of edges per vertex (*i.e.*, the degree of the vertices doesn't grow with n) are sparse. Some examples of these are cycles and grid graphs:





Examples of dense graphs are complete graphs:



Real world examples of sparse graphs might arise from a graph representing the internet, and for dense graphs we could consider a network of cities, connected by all the possible aeroplane routes.

Storing sparse graphs using the various graph representations give rise to the following space complexities:

(i) To store an adjacency list we need to store one piece of data per edge. So the space complexity is $O(m)$. Thus in a sparse graph the space complexity is $O(n)$.

(ii) To store an adjacency matrix we need to store n^2 pieces of information, regardless of m. So a sparse graph is still $O(n^2)$ space.

(iii) Sets of vertices and edges just require n items for the vertices and m items for the edges, so $O(n+m)$. In a sparse graph this becomes $O(n+n) = O(n)$.