Algorithms Notes

Alexis Beingessner
Simon Pratt

Contents

1 Two Fibonaccis 3

2 Integer Multiplication 5

3 Solving Recurrences 7
- 3.1 The Master Theorem 7
- 3.2 Recursion Tree 7

4 Heaps 9
- 4.1 How to Implement a Heap 9
- 4.2 Building a Heap 10
- 4.3 Heapsort 10

5 Selection 11
- 5.1 Analysis 12
- 5.2 What is special about 5? 12

6 Union-Find 13
- 6.1 Approach 1: Linked Lists 13
- 6.2 Approach 2: Better Linked Lists 13
- 6.3 Approach 3: Trees 13
- 6.4 Approach 4: Path Compression 14

7 Graphs 15
- 7.1 Representation 16
  - 7.1.1 Adjacency List 16
  - 7.1.2 Adjacency Matrix 17

8 Depth-First Search 19

9 Minimum Spanning Tree 21
- 9.1 Kruskal's Algorithm 21
- 9.2 Prim's Algorithm 21

10 Shortest Path 23
- 10.1 Single Source 23
  - 10.1.1 Dijkstra's Algorithm 23
  - 10.1.2 Analysis of Dijkstra's Algorithm 23

10.2 All Sources . . . 23
10.2.1 Floyd-Warshall Algorithm . . . 23
10.3 Arbitrary Weights . . . 23
10.3.1 Bellman-Ford Algorithm . . . 23

11 Dynamic Programming 25
11.1 Matrix Chain Multiplication . . . 25
11.2 Longest Common Subsequence . . . 25
11.3 Optimal Triangulation of a Convex Polygon . . . 26
11.4 All-Pairs Shortest Path (Floyd-Warshall) . . . 26
11.5 Knapsack . . . 26
11.6 String Edit Distance . . . 27

12 Complexity Classes 29
12.1 Optimization Problems . . . 29
12.2 Decision Problems . . . 29
12.3 P . . . 29
12.4 NP . . . 29
12.5 $P \subseteq NP$ . . . 29
12.6 P = NP? . . . 29

13 NP-Complete 31
13.1 CIRCUIT-SAT . . . 31
13.2 Reduction . . . 31
13.3 SAT . . . 31
13.4 3CNF-SAT . . . 31
13.5 CLIQUE . . . 32
13.6 VERTEX-COVER . . . 32
13.7 HAM-CYCLE . . . 32
13.8 TSP . . . 33
13.9 SUBSET-SUM . . . 33
13.100/1-Integer Programming . . . 33
13.113-COLOR . . . 33

A Approximations 35
A.1 VERTEX-COVER . . . 35
A.2 Bin Packing . . . 35

B Acknowledgements 37

C License 39

Chapter 1

Two Fibonaccis

```
$fib(n)$ :
  if $n = 0$ or $n = 1$ then
    return $n$
  else
    return $fib(n-1) + fib(n-2)$
  end if
```

We can state a recurrence for this algorithm:

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + O(1) \\ &\geq fib(n-1) + fib(n-2) \\ &= fib(n) \end{aligned}$$

```
$fib2(n)$ :
  if $n = 0$ then
    return $0$
  end if
  create array $f[0..n]$
  $f[0] \leftarrow 0, f[1] \leftarrow 1$
  for $i \leftarrow 2$ to $n$ do
    $f[i] \leftarrow f[i-1] + f[i-2]$
  end for
  return $f[n]$
```

Addition of two numbers in the preceding algorithm takes constant time until the values exceed the maximum value that can be stored in a word. After which, we need to consider how values of arbitrary length are added.

Chapter 2

Integer Multiplication

Let us consider an integer X which is composed of X_L which are the leftmost bits of X, and X_R which are the rightmost bits of X.

$$X = X_L | X_R$$

We can multiply integers X, Y as follows:

$$\begin{aligned} XY &= (2^{n/2}X_L + X_R)(2^{n/2}Y_L + Y_R) \\ &= 2^n X_L Y_L + 2^{n/2} X_L Y_R + 2^{n/2} X_R Y_L + X_R Y_R \\ &= 2^n X_L Y_L + 2^{n/2}(X_L Y_R + X_R Y_L) + X_R Y_R \end{aligned}$$

Which gives the recurrence

$$\begin{aligned} T(n) &= 4T(n/2) + O(n) \\ &\leq 4T(n/2) + cn \\ &\leq 4(4T(n/4) + cn/2) + cn \\ &\leq 4(4(4T(n/8) + cn/4) + cn/2) + cn \\ &\leq 64T(n/8) + cn(1 + 2 + 4) \\ &\dots \\ &\leq 4^i T(n/2^i) + cn(1 + 2 + \dots + 2^{i-1}) \end{aligned}$$

Where i is the number of times we can divide n by 2, or $log_2 n$.

$$\begin{aligned} T(n) &\leq 4^{log_2 n} T(n/2^{log_2 n}) + cn(1 + 2 + \dots + 2^{log_2 n - 1}) \\ &\leq n^{log_2 4} T(n/n^{log_2 2}) + cn \sum_{i=0}^{log_2 n - 1} 2^i \\ &\leq n^2 T(1) + cn 2^{log_2 n} \\ &\leq n^2 T(1) + cn n^{log_2 2} \\ &\leq n^2 T(1) + cn^2 \\ &\leq n^2 (T(1) + c) \\ &\leq n^2 (O(1) + c) \\ &\leq O(n^2) \end{aligned}$$

Can we do better? Yes.

We need: $X_L Y_L, X_R Y_R$, and $X_L Y_R + X_R Y_L$

Observe:

$$\begin{aligned} &(X_L + X_R)(Y_L + Y_R) - X_L Y_L - X_R Y_R \\ &= X_L Y_L + X_R Y_L + X_L Y_R + X_R Y_R - X_L Y_L - X_R Y_R \\ &= X_R Y_L + X_L Y_R \end{aligned}$$

Since we must compute $X_L Y_L$ and $X_R Y_R$ anyway, this saves us an entire multiplication. Reducing our recurrence from $T(n) = 4T(n/2) + O(n)$ to $T(n) = 3T(n/2) + O(n)$.

We can solve this new recurrence as follows:

$$
\begin{aligned}
T(n) &= 3T(n/2) + O(n) \\
&\leq 3T(n/2) + cn \\
&\leq 3(3T(n/4) + cn/2) + cn \\
&\leq 3^i T(n/2^i) + cn(1 + 3/2 + ... + (3/2)^{i-1}) \\
&\leq 3^{log_2 n} T(n/2^{log_2 n}) + cn(1 + 2 + ... + (3/2)^{log_2 n-1}) \\
&\leq n^{log_2 3} T(1) + cn \sum_{i=0}^{log_2 n-1} (3/2)^i \\
&\leq n^{log_2 3} T(1) + cn(3/2)^{log_2 n} \\
&\leq n^{log_2 3} T(1) + cnn^{log_2 (3/2)} \\
&\leq n^{log_2 3} T(1) + cnn^{log_2 3 - log_2 2} \\
&\leq n^{log_2 3} T(1) + cnn^{log_2 3 - 1} \\
&\leq n^{log_2 3} T(1) + cnn^{log_2 3} n^{-1} \\
&\leq n^{log_2 3} (T(1) + cn/n) \\
&\leq n^{log_2 3} (T(1) + c) \\
&\leq O(n^{log_2 3})
\end{aligned}
$$

Chapter 3

Solving Recurrences

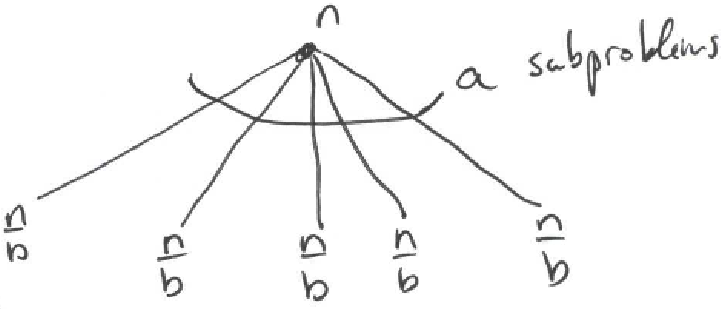
3.1 The Master Theorem

If $T(n) = aT(\frac{n}{b}) + O(n^d)$, then

$$T(n) = \begin{cases} O(n^d) & \text{if} d > log_b a \\ O(n^d logn) & \text{if} d = log_b a \\ O(n^{log_b a}) & \text{if} d < log_b a \end{cases}$$

3.2 Recursion Tree

We can reason about a recurrence of the form: $T(n) = aT(\frac{n}{b}) + f(n)$ where $a \geq 0, b > 0$ with the following recursion tree:



This tree has the following properties:

1. The number of nodes at level i: a^i
2. Work done at each node of level i: $f(\frac{n}{b^i})$
3. Number of levels: $log_b n$
4. Number of leaves: $n^{log_b a}$

We can use this information to solve the recurrence:

$$\begin{aligned} T(n) &= \sum_{i=0}^{log_b n} (\text{\# nodes at level } i)(\text{work done at level } i) \\ &= \sum_{i=0}^{log_b n} a^i f(\frac{n}{b^i}) \end{aligned}$$

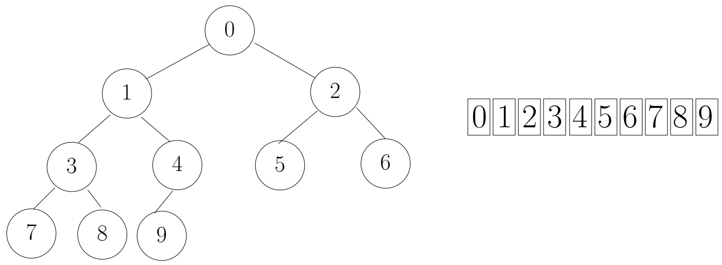
Chapter 4

Heaps

For the purposes of generality, instead of referring to elements that are "greater than" or "less than" others, we will simply say that they are "better than" or "worse than" others. For any particular ordering, the "best" element is desired first. A Heap is a binary tree that satisfies the following properties:

- The root of a heap is better than its two children (the heap property)
- The children of the root are also heaps
- A heap is a complete binary tree (only the last level may not be full, and all elements in the last level are on the left)

A heap differs from most binary trees in that it provides no particular ordering of the elements, but rather guarantees that the best element is at the root. Further, while heaps are usually discussed and defined as binary trees, they need not be implemented as such. A heap can in fact be implemented as an array with no performance reduction. Contrary to standard theoretical convention, we will be using 0-indexed arrays rather than 1-indexed arrays. That is to say, the indexing will reflect how most programming languages perform indexing (0 is the first element, not 1). To represent a heap as an array, we can implement a simple indexing. If an element is at the ith index in the array, its left and right child are at the $2i + 1$th and $2i + 2$th indexes respectively. By placing the root at the 0th index, all others follow. This is depicted below.



Heaps are commonly used to implement priority queues, because they do the minimum amount of work required to keep track of the best element.

4.1 How to Implement a Heap

A heap must support the following operations

- $best()$: returns the best element in the heap
- $pop - best()$: removes the best element in the heap
- $insert(x)$: inserts x into the heap

From the definition of a heap, we know the we can easily implement $best()$ by returning the root of the heap, which should take $O(1)$ time. However the other operations are less obvious.

To $insert(x)$ recall that a heap must be complete, therefore, if a new element is added, it must be added to the left-most available space in the last row of the heap. However, the heap property may now be violated. If the new element x is worse than its parent y, the heap property is satisfied and we may stop. However if it is not satisfied we may swap x with y and recurse on x's new position. This works because we know that y is better than all of x's children, because the heap property was satisfied before x was added. Further, because x is better than y, it is also better than all of y's children. However x may still be better than its new parent, so we must recurse. If x is the new best element, it will eventually reach the root. Because heaps are complete, this operation will take $O(\log n)$ time, as this is the height of the heap.

To $pop - best()$, we may simply remove the root, however this completely destroys the entire heap. Instead, we will swap the root with the bottom-left-most element, y. Now removing the best element leaves us with a still complete tree. However the heap property has likely been violated once more. If y is better than both its children, then we may stop. However, if not, we shall swap y with its best child and recurse on $y's$ new position. Because the element we swap with y is better than both y and the other child, the heap property has been satisfied for this sub-heap. However, the heap property may still be violated for $y's$ new sub-heap, so we must do this again, until y is the root of a valid heap. Once more, this operation requires $O(\log n)$ time, as it must at worse traverse the entire height of the tree.

Therefore, a heap may support $best()$ in $O(1)$ time, $insert(x)$ in $O(\log n)$ time, and $pop - best()$ in $O(\log n)$ time.

4.2 Building a Heap

Now that we can support all the operations that a heap must implement, it would be nice to be able to actually construct one given a list of n elements. A naïve approach is to simply call $insert(x)$ on every element in the list. However, since $insert(x)$ requires $O(\log n)$ time, this will require $O(n \log n)$ time. These seems pretty bad, considering one can find the best element in a list by brute force in $O(n)$ time. Can we achieve a construction time comparable to the brute force time? Instead of building the heap top down with $insert$, we can build it from the bottom up. Remark that a single element is a valid heap. If we were to try to build from the bottom up, we could first take the last $n/2$ elements in the list. All of these elements are their own valid and complete heaps, and we therefore do not need to do anything to them. To add the next $n/4$ elements, we simply perform the procedure we did in $pop-best$ to fix the fact that the new root might be violating the heap property, knowing that all the elements below it are valid heaps. By repeating this process until we reach the first element in the list, we will have created a valid heap on n elements.

Because we are doing very little work for the majority of the elements, we end up doing only $O(n)$ work over all, which is optimal, as this is the amount of time required to find the best element.

4.3 Heapsort

Another nice property of a heap is that once one has been implemented, it provides a very simple procedure for sort elements. A simple algorithm to do this to construct a heap on the list and then simply return $best$ and then call $pop - best$ over and over until there are no more elements in the heap. In fact, since our algorithm for building the heap is in-place and takes $O(n)$ time, and our remove method leaves the best element at the end of the array, by simply building a heap on the input array and calling $pop-best$ n times, we will be left with a reverse sorted array in $O(n \log n)$ time. This algorithm is particularly excellent because it requires no extra space, runs deterministically, and is worst-case optimal.

Chapter 5

Selection

Consider the following problem: given an array A of n elements, output the i-th smallest element of A.

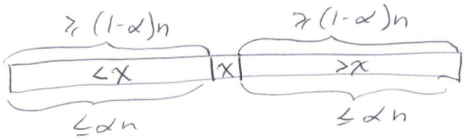
As a simple first solution, we can sort A and then return $A[i]$. Since sorting takes $O(nlogn)$ time and returning $A[i]$ takes $O(1)$ time, this solution takes $O(nlogn) + O(1) = O(nlogn)$ time.

But it should be easy to see that we can do better in specific cases like $i = 1$ or $i = n$. Simply iterate once over the array and store the minimum ($i = 1$) or maximum ($i = n$) value. Since looking at a particular element of the array takes $O(1)$ time, and we look at all n elements, this takes total $n \cdot O(1) = O(n)$ time.

We can also tell that this is optimal, because we know that to determine the i-th element, we need to look at all n elements in the array, so we have a lower bound of $\Omega(n)$ time.

But is this possible in general, for any value of i? Yes.

Suppose in linear time we can find element x such that



$0 < \alpha < 1$ is a constant.

x is somewhere around the middle of the array, and is preceded only by elements smaller than x, and followed only by elements larger than x. We also know that there are $\geq (1-\alpha)n$ and $\leq \alpha n$ elements both before and after x.

We can calculate this x as follows:

1. Split A into groups of 5. There will be $\frac{n}{5}$ of these groups.
2. Compute the median m_j of each group M_j for $1 \leq j \leq \frac{n}{5}$.
3. Compute the median x of $m_1, m_2, ..., m_{n/5}$.

It should be clear that step 1 takes constant time, step 2 takes constant time for each group of constant size and $O(n)$ time total for all $\frac{n}{5}$ groups, and step 3 takes $T(n/5)$ time.

Claim 5.1. This x has the properties we needed above.

Proof. We know that $\frac{1}{2}$ of m_j are smaller than x, and since there are $\frac{n}{5} m_j$s, we know $\frac{n}{10}$ of m_j are $\leq x$.

So for each m_j where $m_j \leq x$

- there are 3 elements that are $\leq m_j$
- so 3 (or more) elements are $\leq x$

□

Now we must put x into its position in the array using partitioning. As a side note, partitioning is used in quicksort.

1. Find x, put it at the end
2. Partition elements around x
3. Put x into its proper position

We now have an x that satisfies the properties we needed, and it is properly located at position q in A. We are left with 3 cases:

1. If $i = q$: x is the i-th element of A.
2. If $i < q$: recurse on the subarray which is $< x$
3. If $i > q$: recurse on the subarray which is $> x$, and $i \leftarrow i - q$

This last step gives a recurrence of $T\left(\frac{7n}{10}\right)$ in the worst case because at least $\frac{3n}{10}$ elements in A are smaller than x.

5.1 Analysis

We now have the following recurrence:

$$T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n)$$

Claim 5.2. $T(n) \leq cn$

Proof.

$$\begin{aligned} T(n) &= dn + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) \\ &\leq dn + c\frac{n}{5} + c\frac{7n}{10} \\ &= dn + \frac{9}{10}cn \\ &= cn\left(\frac{d}{c} + \frac{9}{10}\right) && \leq cn \end{aligned}$$

As long as

$$\begin{aligned} \frac{d}{c} + \frac{9}{10} &\leq 1 \\ \frac{d}{c} &\leq \frac{1}{10} \\ 10d &\leq c \end{aligned}$$

□

5.2 What is special about 5?

First of all, we need an odd number for there to be a median. Secondly, notice:

$$\frac{1}{5} + \frac{7}{10} = \frac{9}{10} < 1$$

Dividing into groups of 3 doesn't work because:

$$T(n) = O(n) + T(n/3) + T(2n/3) = \Theta(nlogn)$$

Dividing into groups of 7 actually does work because:

$$T(n) = O(n) + T(n/7) + T(5n/7) = \Theta(n)$$

Chapter 6

Union-Find

Consider the following problem. Given n disjoint sets of 1 element each, perform n unions and then m queries for what set a given element is in. We will call these two operations $union(A, B)$ and $find(x)$.

6.1 Approach 1: Linked Lists

Our first approach to this problem is to describe our sets as linked lists. We know we can combine linked lists quite quickly, so this seems ideal for *union*. All we must do is have the head of B point to the tail of A, which can be done in constant time. However, linked lists are not particularly well suited for $find$. To resolve this, at every node we shall store a *backpointer* to the linked list the node is part of. This allows us to perform $find$ in constant time. However, now *union* needs some extra work. When we call $union(A, B)$, we will now walk through B and fix all of its back pointers to point to A. This will take time linear in the size of B.

Given n sets, what is the worst possible way to *union* them? Well, since the run time *union* is linear in the size of the second list, adversarial we can do is to *union* every individual element with the current unioned set. For instance, $union(D, union(C, union(A, B)))$.... Clearly this will require $O(\sum_{i=1}^{n-1} i)$, which is $O(n^2)$. If we then perform m $find$s, all of which take $O(1)$ time, we will have performed n *union*s and m $find$s in $O(n^2 + m)$ time.

6.2 Approach 2: Better Linked Lists

Somehow our first approach was naïve, which allowed us to "game" the system to create a very bad result for the *union*s. To get a better result, we will make a slight modification to our *union* algorithm. Instead of blindly attaching B to the end of A, we will attach the smaller set to the larger. This fixes our adversarial approach, but is it actually better?

Consider how frequently we need to change the back pointers on an individual node. At first, it is part of a set of size one, and will have to change its pointer when unioned to a set of size ≥ 1, placing it in a set of size ≥ 2. The next time it will be changed is when it is unioned to a set of size ≥ 2, then ≥ 4 and so on. The last time will be when it is unioned to a set of size $\geq n/2$ after which we can not find another set of large enough size to change it again. Therefore each back pointer needs to be changed $O(\log n)$ times at worst. Since there are n back pointers, this new approach only require $O(n \log n)$ time to perform the unions. Since the $find$ operation is not effected, our approach now performs n *union*s and m $find$s in $O(n \log n + m)$ time.

6.3 Approach 3: Trees

Having to fix back pointers is still fairly wasteful, what if we didn't have to? Instead of implementing our sets as linked lists, we can instead use trees. Each set will be a node with either the name of the set, or a pointer to its parent. When we perform $union(A, B)$, we will simply replace the name of the shorter set with a pointer to the head of the taller set. Since we are just changing a pointer, this will only take $O(1)$ time. Our $find$ algorithm, however, will now have to walk from the node all the way to the root of the tree it is in to find out what list it is in. By a similar argument from the previous section, the height of our tree will only ever be $\log n$. Therefore this approach can support n *union*s and m $find$s in $O(n + m \log n)$ time.

6.4 Approach 4: Path Compression

Our *union* algorithm is optimal using the tree approach, but it has made our $find$ algorithm suffer. To fix this, we make a simple observation. Since our find algorithm must already walk through several nodes in the tree, once we get to the root we can, without worsening the time, relabel all of their pointers to point directly to the root. This approach is called $pathcompression$. This will make subsequent queries on these elements and their children substantially faster. The analysis of this algorithm is beyond the scope of this class, but evidently it can support n *union*s and m $find$s in $O(n + m\alpha(n))$ time. Where α is the inverse Ackermann function. Although $\alpha(n)$ tends towards infinity as n does, for any "practical" n it is at most 4. It turns out that this is in fact optimal for the union-find problem.

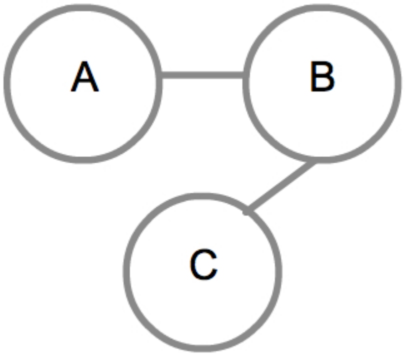
Chapter 7

Graphs

A *graph* is an ordered pair $G = (V, E)$ where V is a set of *vertices* and E is a set of *edges*. An edge is a pair of vertices which are said to be *adjacent*. An edge is said to be *incident* on its component vertices. Usually we consider edges to be unordered, in which case the graph is undirected and an edge $\{A, B\}$ connects A to B and B to A. For example:

$$V = \{A, B, C\}$$
$$E = \{(A, B), (B, C)\}$$

Which can be represented more visually:



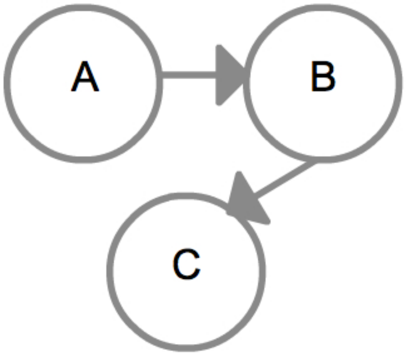
Note that the vertices are represented by labeled circles and the edges are represented by lines connecting vertices to one another.

The number of edges connecting a vertex v is called the degree of v or $deg(v)$. In the above example:

$$deg(A) = deg(C) = 1$$
$$deg(B) = 2$$

Sometimes an edge is directional, meaning the pair of vertices in an edge is ordered. In other words, the edge (A, B) connects A to B, but not B to A. We say such an edge is incoming on B and outgoing on A. A graph whose edges are ordered pairs is called a *directedgraph* or *digraph*.

This is represented visually by an edge with an arrow at one end, indicating the direction:



In a digraph, the number of incoming edges of v is the in-degree or $deg^{-}(v)$. Similarly, the number of outgoing edges is the out-degree or $deg^{+}(v)$.

In the above example:

$$deg^{+}(A) = deg^{+}(B) = 1$$
$$deg^{-}(B) = deg^{-}(C) = 1$$
$$deg^{+}(C) = deg^{-}(A) = 0$$

A *simple graph* is a graph which contains no edges from any vertex v to itself (v, v), called loops. A simple graph also contains no multi-edges which connect more than two vertices.

A *path* is a sequence of edges from vertex u to vertex v.

Two vertices are said to be *connected* if there exists a path between them.

A graph is said to be connected if for any two vertices, there exists a path between them. An adjacent vertex is called a *neighbour*.

A *complete* graph is one in which every vertex is adjacent to every other vertex.

A *subgraph* is a graph consisting of a subset of the vertices and edges in another graph.

A *connected component* is a connected subgraph which does not disconnect any adjacent vertices. It should be easy to see that a connected graph has exactly one connected component.

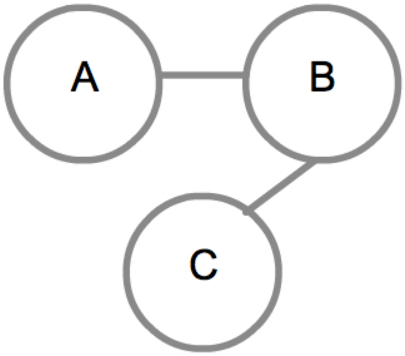
A *cycle* in a graph is when there exists a path from a vertex back to itself without crossing any edge more than once.

A graph is said to be *acyclic* when it does not contain cycles.

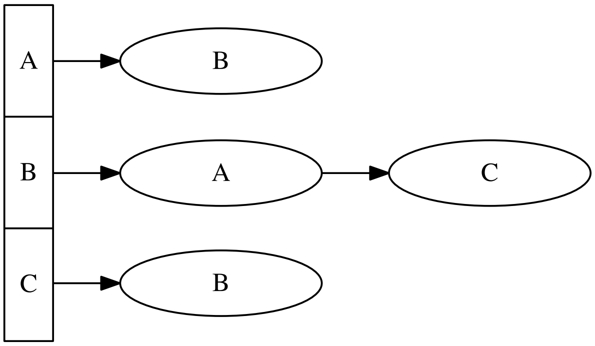
7.1 Representation

7.1.1 Adjacency List

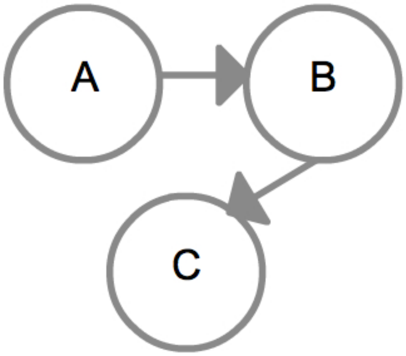
One way to represent a graph is for every vertex, store a list of neighbours. For example:



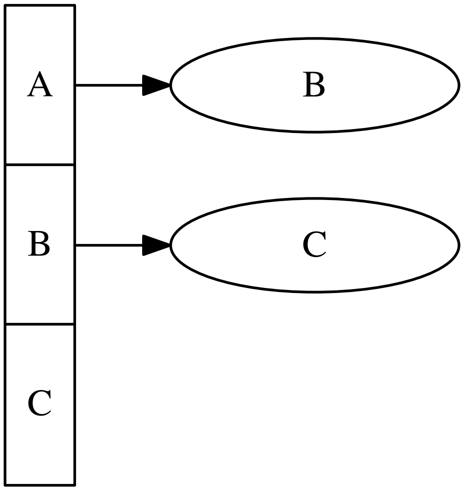
can be represented:



And

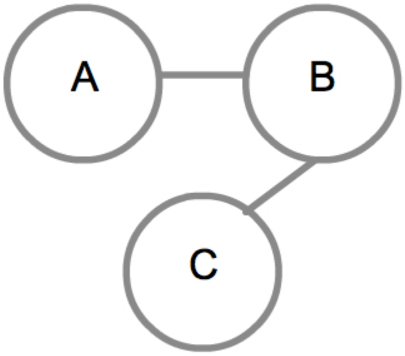


can be represented:



7.1.2 Adjacency Matrix

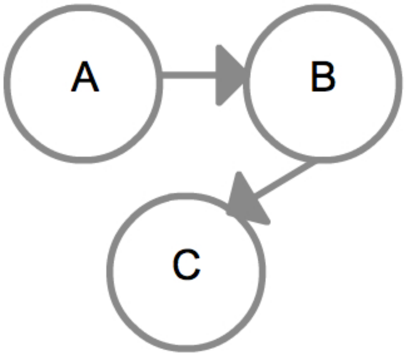
Another representation of a graph is as a square matrix with n rows and columns where the element at row i and column j is 1 if there is an edge between v_i and v_j and 0 otherwise. For example:



can be represented

$$\begin{bmatrix} & a & b & c \\ a & 0 & 1 & 0 \\ b & 1 & 0 & 1 \\ c & 0 & 1 & 0 \end{bmatrix}$$

As for directed graphs, row i column j is 1 if there exists an edge (v_i, v_j). For example:



can be represented

$$\begin{bmatrix} & a & b & c \\ a & 0 & 1 & 0 \\ b & 0 & 0 & 1 \\ c & 0 & 0 & 0 \end{bmatrix}$$

Chapter 8

Depth-First Search

One important operation in a graph is search. A useful kind of search is Depth-First, in which we begin as some start node, marking the node visited, then we recurse on the neighbours one by one until no more unvisited neighbours exist.

During a visit, we label each node with a number before visiting its children and another number after visiting its children. This number (called the clock) starts at 1, and is incremented every time a node receives a label. The number a node v receives before its children are visited is called its pre-number ($pre(v)$), and the number that node receives after its children are visited is called its post-number ($post(v)$).

Theorem 8.1 (Parenthesis Theorem)**.** *For nodes u, v, the interval between $pre(u)$ and $post(u)$ and the same interval for v are either:*

- *Entirely disjoint*
- *The interval of u is completely within the interval of v*
- *The interval of v is completely within the interval of u*

The name comes from the fact that this is just like properly nested parentheses.

Running a DFS on a graph reveals a tree structure where the start node is the root and neighbours are parent and child depending on which was visited first. There are a few interesting classifications of edges in a DFS tree:

1. A *Tree Edge* goes from a parent node to a child node.
2. A *Forward Edge* goes from ancestor to descendant (but not parent to child)
3. A *Back Edge* goes from descendant to ancestor
4. A *Cross Edge* goes to a non-ancestor non-descendant

Theorem 8.2. *A digraph has a cycle if and only if a DFS reveals a back edge.*

Proof. First let us assume there is a back edge. By definition this is a descendant linking back to an ancestor, which has already been visited. This gives us a cycle.

Next let us assume there is a cycle. A DFS will visit each node in the cycle, and as soon as the last edge in the cycle is visited it will link a descendant to ancestor which is the definition of a back edge.

□

Theorem 8.3. *After running a DFS on a directed acyclic graph (DAG), each edge leads to a vertex with a lower post number.*

The proof of this theorem is left up to the reader.

Chapter 9

Minimum Spanning Tree

Given an undirected, connected graph where each edge has positive weight, the *Minimum Spanning Tree* (MST) is a connected subgraph on the same vertex set with minimal weight. Intuitively, the MST is the lightest possible connected subgraph. The MST is not necessarily unique, for proof of this, consider a graph where all edges are of equal weight. Any tree is an MST of such a graph.

9.1 Kruskal's Algorithm

Kruskal's algorithm for computing the MST relies on the simple observation that the smallest edge in a graph G is part of *some* MST of G. Kruskal's algorithm starts by constructing a min-heap containing all m edges in G, and a collection of disjoint sets, each containing one of the n vertices of G. We also construct an empty list T that will hold all the edges of the MST. We then get the minimum edge from the heap, check if the two nodes the edge joins are from the same set, and if not add the edge to T, and *union* the two sets that the edge joins. We repeat this until T contains $n-1$ edges.

Because this algorithm uses structures we already know and understand, analysis will be fairly easy. We require $O(m)+O(n)$ time to construct the initial sets and heap. We also require $O(m \log m)$ to extract the minimums from the heap. Our $n-1$ unions and $n-1$ finds can be done in $(n \log n + n)$ time. Therefore this algorithm takes $O(m \log m + n \log n)$.

9.2 Prim's Algorithm

Prim's algorithm for computing the MST is fairly similar to Kruskal's. However, instead of working with all of the edges and vertices at once, it picks one vertice and builds from that. We start by constructing an empty list A which will hold all the vertices that are part of the MST so far, a list T which will hold all the edges in the MST so far, a min-heap V which will contain all the vertices not in A. At first every node is given a key of infinity. We then pick a random vertex v from V and move it from V to A. Next we look at all the edges of v and set the keys of the corresponding nodes to the weight of these edges. Now we retrieve the minimum node u from V and move it to A, and its key edge to T. Next we look at all the edges on u and if their weight is smaller than the current key of their corresponding edge, set the key to that weight. Then we simply retrieve another node from the heap and repeat. After $n-1$ iterations we will have a complete MST.

This algorithm requires $O(n)$ time to construct the initial heap and initialize the first node, and since we must update at most m keys in the heap of n elements, it requires $O(m \log n)$ time to do this. Therefore the entire algorithm takes $O(n + m \log n)$ time.

Chapter 10

Shortest Path

Given a weighted graph, one interesting problem is to find the path with minimal weight.

10.1 Single Source

This instance of the shortest path problem begins at a specific vertex in the graph, hence “Single Source.”

10.1.1 Dijkstra’s Algorithm

Given an acyclic graph $G = (V, E)$ in which every edge has a positive weight and a single vertex s from which to start the path, we begin by labeling all vertices $u \in E | weight(u) = \infty$, then label $weight(s) = 0$. We also create a min-heap of the vertices using weight as key.

We pop-best a vertex v from our heap. For each edge e incident to v and some other vertex u, we check if $weight(v) + weight(e)$ is less than $weight(u)$, in which case we set $weight(u) = weight(v) + weight(e)$ and decrease the key of u in our heap. We continue this process until no vertices remain in the heap.

It should be easy to see that if we have a particular target vertex, we can halt the algorithm as soon as our target is the root of the heap, because we have found the shortest path to it.

10.1.2 Analysis of Dijkstra’s Algorithm

Intuitively, this algorithm has an upper bound of $O(|V|)$ times however long it takes us to extract the minimum vertex from our heap, plus $O(|E|)$ times however long it takes us to decrease the key, since we have to check every vertex and we may have to check every edge. Using a standard heap, this gives us $O(|V|log|V| + |E|log|V|)$.

10.2 All Sources

This instance of the shortest path problem is concerned with several sources, in which case Dijkstra’s Algorithm does not suffice.

10.2.1 Floyd-Warshall Algorithm

The Floyd-Warshall Algorithm is discussed in detail in the chapter on Dynamic Programming.

10.3 Arbitrary Weights

A keen reader will have noticed that both Dijkstra’s and Floyd Warshall Algorithms assume positive edge weights. Neither of which are generally useful when there may be edges of negative weight.

10.3.1 Bellman-Ford Algorithm

The Bellman-Ford Algorithm can be applied with edges of arbitrary weight. Note that even Bellman-Ford doesn’t deal with cycles of negative weight, since these can be used to make any path have an arbitrarily small total weight.

The Bellman-Ford Algorithm is beyond the scope of these notes, for more information please see Wikipedia’s entry on the Bellman-Ford Algorithm, or CLRS.

Chapter 11

Dynamic Programming

So far we have considered two major strategies in algorithms design: greedy, in which we repeatedly take the local optimum choice; and divide and conquer, in which we divide the greater problem into similar subproblems and recurse.

There may be problems for which these strategies are suboptimal. In which case, we have a third strategy which may be of use: dynamic programming. This strategy divides the problem into sub problems, but rather than recursing on each sub problem individually, we identify easy to compute base cases from which we can build towards the solution to the larger problem, storing the results of previous computations to use in later computations. This technique differs from a similar technique called "memoization" which computes recursively top-down, whereas dynamic programming begins at the base case(s) and works up.

Dynamic programming has these important steps:

1. Determine structure of optimal solution
2. Set up recurrences for optimal solution
3. Solve recurrences bottom-up
4. Construct the optimal solution

And the final step is most often neglected since we can typically add some small amount of information to the process so we can trivially reconstruct the optimal solution.

11.1 Matrix Chain Multiplication

Assuming knowledge of matrices and how they are multiplied.

The problem is to find the way to multiply n matrices $A_1, ..., A_n$ with dimensions $p_0, ..., p_n$ (e.g. A_i has dimensions $p_{i-1} \times p_i$) using the least number of calculations. Notice that multiplying $A_i A_{i+1}$ takes $p_{i-1}p_ip_{i+1}$ calculations.

We start by determining the structure of the optimal solution. Observe that the optimal solution will necessarily involve splitting $A_1A_2...A_n$ into two subproblems at some optimally chosen A_kA_{k+1} so we multiply $A_1...A_k$ then $A_k...A_n$ and then multiply the results together. If we define the function $m(i,j)$ to be the minimum cost of multiplying $A_i...A_j$, then the cost of the optimal solution is $m(1,n) = m(1,k) + m(k+1,n) + p_0p_kp_n$.

We then define the recurrence $m(i,i) = 0$ and $m(i,j) = min\{m(i,k) + m(k+1,j) + p_{i-1}p_kp_j\}$ for all k such that $i < k \leq j$.

We then compute all $m(i,i)$ for $1 \leq i \leq n$, then all $m(i,i+1)$, $m(i,i+2)$, ... until we have calculated $m(1,n)$.

The time complexity of this solution is

$$\sum_{l=2}^{n} \sum_{i=1}^{n-l+1} \sum_{k=i}^{i+l+2} O(1) = O\big(n^3\big)$$

11.2 Longest Common Subsequence

For a string $S = (s_1, s_2, \ldots, s_n)$, a subsequence of S is string S' such that every element of S' is in S, and if an element $a \in S'$ comes before $b \in S'$, this also holds for S. The longest common subsequence of two strings $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_m)$, is the longest string $C = (c_1, c_2, \ldots, c_k)$, such that C is a subsequence of A and B.

Let the substring $(s_i, s_{i+1}, \ldots, s_j)$ be denoted as $S(i,j)$. Further, let $LCS(i,j)$ to be the length of the LCS of $A(0,i)$ and $B(0,j)$. Assume that we have computed $C(1,k)$, and that it is unique. Then if we remove c_k

and everything after it from B and A, $C(1, k-1)$ is now the LCS of our modified A and B.

In general $LCS(i, j) =$

$$\begin{cases} \emptyset & \text{if } i = 0 \text{ or } j = 0 \\ LCS(i-1, j) + 1 & \text{if } a_i = b_j \\ & \text{and } LCS(i-1, j) > LCS(i, j-1) \\ LCS(i, j-1) + 1 & \text{if } a_i = b_j \\ & \text{and } LCS(i-1, j) < LCS(i, j-1) \end{cases}$$

To compute this, we compute $LCS(0, 0)$, $LCS(0, 1)$,...,$LCS(0, m)$; then $LCS(1, 0)$,...,$LCS(1, m)$; ...;$LCS(n, 0)$,...,$LCS(n, m)$. At which point we have our solution, which is $LCS(n, m)$.

11.3 Optimal Triangulation of a Convex Polygon

First some definitions. A *polygon* is a list of vertices $(v_1, ..., v_n)$ such that for any v_i, there exists an edge (v_i, v_{i+1}) and also there exists an edge (v_1, v_n). A polygon is said to be *convex* if any line passing through the polygon crosses the edges of the polygon at most twice. A *chord* is an edge between two non-adjacent vertices in a polygon. A *triangulation* is a set of chords which divide a polygon into triangles.

The problem is to build a triangulation of a given convex polygon which minimizes total edge length. We define the function $w(a, b, c)$ to be the weight of the triangle (v_a, v_b, v_c), which in this case will be the length of the edges (v_a, v_b), (v_b, v_c), and (v_c, v_a). We also define the function $t(a, b)$ to be the optimal triangulation of points $(v_a, ..., v_b)$. We would like to solve $t(1, n)$.

We start by defining the structure of an optimal solution. Notice that the optimal triangulation contains the triangle (v_1, v_k, v_n) for some k. The cost of this triangulation is $t(1, k) + t(k, n) + w(1, k, n)$.

We then define the recurrence $t(i, i+1) = 0$ for all i, and $t(i, j) = min\{t(i, k) + t(k, j) + w(i, k, j)\}$ for all k such that $i < k < j$.

We then compute all $t(i, i + 1)$, then all $t(i, i + 2)$, $t(i, i + 3)$, ... until we have calculated $t(1, n)$.

11.4 All-Pairs Shortest Path (Floyd-Warshall)

Given a graph $G = (V, E)$ where edges have an associated weight $w(e)$ for $e \in E$, compute the shortest path between u and v for all $u, v \in V$.

We start by defining the structure of an optimal solution. Let us decompose the problem into finding the optimal path between v_i and v_j. All of the vertices on this path are contained in $\{v_1, ..., v_k\}$ except perhaps v_i, v_j. Notice that if $k = 0$ then if there is an edge $(v_i, v_j) \in E$ then the distance is the weight of that edge, and if there is no such edge then the weight is infinite. Otherwise, we take the minimum of either including or excluding v_k on the path. This gives the recurrence:

$$d^k_{i,j} = \begin{cases} weight((v_i, v_j)) & \text{if } k = 0 \text{ and} \\ & (v_i, v_j) \in E \\ \infty & \text{if } k = 0 \text{ and} \\ & (v_i, v_j) \notin E \\ min\{d^{k-1}_{i,j}, d^{k-1}_{i,k} + d^{k-1}_{k,j}\} & \text{if } k > 0 \end{cases}$$

We then compute $d^0_{i,j}$ for all $v_i, v_j \in V$, then all $d^1_{i,j}$, $d^2_{i,j}$, ... until we have computed $d^{n-2}_{i,j}$.

Running time: $O(n^3)$. Analysis is left as an exercise to the reader.

11.5 Knapsack

Given a knapsack (or bag) which can carry W units of weight, and n items where item i has weight w_i and value v_i, what is the most valuable list of items which can fit in the given bag? Notice that this may include multiples of a particular item.

We start by defining the structure of an optimal solution. Let $p(w)$ be the value of the optimal packing of a bag which can carry w units of weight, and let $p_i(w)$ be the same but necessarily including item i.

$$\begin{aligned} p_i(w) &= p(w - w_i) + v_i \\ p(0) &= 0 \\ p(w) &= max\{p(w - w_i) + v_i\} \qquad \text{where } i : w_i \leq w \end{aligned}$$

We then calculate $p(0), p(1), ..., p(W)$.

The running time for this algorithm is $O(nW)$. Notice that W requires $O(logW)$ bits to represent. Since the input to the problem is $O(n + logW)$, and $W = 2^{logW}$, the time complexity is $O\big(n2^{logW}\big)$ so this solution is exponential with respect to the input.

11.6 String Edit Distance

We define the edit distance between two strings as the minimum number of edits necessary to transform one string into another, where edits are insertions, deletions, or replacements of a single character.

The problem is to compute this edit distance, given strings X of length m and Y of length n.

We start by defining the structure of an optimal solution. Let us define $E[i, j]$ as the minimal edit distance between $X[1..i]$ and $Y[1..j]$. We would like to know $E[m, n]$.

It should be easy to see that $E[0, 0] = 0$ and $E[1, 1]$ is either 0 if $X[1] = Y[1]$ or 1 otherwise. $E[0, j] = j$ and $E[i, 0] = i$.

This gives the recurrence:

$$E[i, j] = \begin{cases} 0 & \text{if } i = 0, j = 0 \\ i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ min\{1 + E[i-1, j], 1 + E[i, j-1], 1 + E[i-1, j-1]\} & \text{if } X[i] \neq Y[j] \\ min\{1 + E[i-1, j], 1 + E[i, j-1], E[i-1, j-1]\} & \text{otherwise} \end{cases}$$

Then we calculate $E[i, j]$ from $i = 0, j = 0$ to $i = m, j = n$.

Chapter 12

Complexity Classes

A complexity class is a set of related problems. Before we talk about specific classes, it is important to understand the difference between a decision problem and an optimization problem.

12.1 Optimization Problems

An optimization problem is a problem of the form "find the optimal solution for problem p." Each such problem may have a different defintion of optimal, and there may be many optimal solutions.

12.2 Decision Problems

A decision problem is much more limited in scope. These are problems with strictly yes/no answers, often of the form "is there a solution of size (at least or at most) k to problem p?".

Notice that optimization problems have related decision problems. For any optimization problem we can ask the decision problem "is x the optimal solution to problem p?".

12.3 P

The complexity class P is the set of all decision problems which are solvable in polynomial time. That is to say, all problems for which there exists an algorithm to solve the problem bounded above by $O(n^d)$ where $d \in \mathbb{Z}$.

12.4 NP

The complexity class NP is the set of all decision problems for which a certificate (possible solution) may be verified in polynomial time.

12.5 $P \subseteq NP$

It should be obvious that any problem in P is also in NP, since if we can find the answer in polynomial time without the certificate, we must certainly be able to do so with the certificate.

12.6 P = NP?

It is not obvious if there are problems in NP but not in P or if all problems in NP are also in P. In fact, the Clay Mathematics Institute has listed this as one of the Millenium Prizes for which the award for solving is 1 million USD.

Chapter 13

NP-Complete

The complexity class known as *NP-Complete* is a special subset of NP, such that any problem in *NP* can be translated to a problem in *NP-Complete* in polynomial time.

13.1 CIRCUIT-SAT

Given a boolean circuit, is it possible to provide a set of inputs that cause the output to be *True*?

In 1971, Cook proved that *CIRCUIT-SAT* is *NP-Complete*. The proof is beyond the scope of these notes. Cook showed that all operations of a polynomial-sized Turing machine can be performed in polynomial time using an instance of this problem. In other words, Cook showed that a Turing machine can be implemented using circuits (surprise!!).

13.2 Reduction

Given at least one *NP-Complete* problem, we can prove any other problem L is *NP-Complete* by showing that $L \in NP$ and that given an instance x of a proven *NP-Complete* problem we can translate x to an instance of L in polynomial time, such that by solving our generated instance of L, we can solve x.

The steps to do this are:

1. Show $L \in NP$. We do this by first showing that the certificate for L is polynomial with respect to the input, and that a certificate for L can be verified in polynomial time.
2. Select a known *NP-Complete* L', ideally one that is similar to L.
3. Describe a polynomial time algorithm that maps any instance $x \in L'$ to an instance $f(x) \in L$.
4. Show that we can solve $x \in L'$ if and only if we can solve $f(x) \in L$.

13.3 SAT

Given a boolean formula, are there values of the variables that cause the formula to be *True*?

First let us show that $SAT \in NP$ by using the truth values as a certificate, which is clearly polynomial in size because it is necessarily a subset of the input. To verify, we evaluate the formula given the truth values and return the output.

Next, we select *CIRCUIT-SAT* from which to reduce to this problem.

Then given an instance of *CIRCUIT-SAT*, we transform the circuit to a boolean formula such that the circuit is satisfiable if and only if the formula is satisfiable. We can do this easily by mapping gates to boolean operators.

Since the instances of each problem are equivalent, it should be easy to see that one is satisfiable if and only if the other is.

13.4 3CNF-SAT

Is a given boolean formula in *3CNF* form satisfiable? A boolean formula is said to be in *3CNF* form if there are no more than 3 variables in each clause, and within a clause there are only *OR* operators and between clauses there are only *AND* operators.

By the same proof as *SAT* above, $3CNF - SAT \in NP$.

We reduce from *SAT*. Given an instance of *SAT*, we use equivalences to reduce all boolean operators to *AND*, *OR*, and *NOT*. Then we use DeMorgan's law to put the formula into *3CNF* form.

Since the instances of each problem are equivalent, it should be easy to see that one is satisfiable if and only if the other is.

13.5 CLIQUE

Given a simple, undirected graph G, find a *complete* subgraph of G of at least size k.

Let the certificate be the set of vertices on which there is a complete subgraph. Since this is necessarily a subset of the input vertices, it must be of polynomial size. We can verify the set of vertices are complete by checking that every pair of vertices i, j where $i \neq j$ are adjacent, and we can easily check if the set is of size k by counting them.

We reduce from *3CNF-SAT*. Given an instance Φ of *3CNF-SAT* with k clauses, we would like to construct an instance of *CLIQUE*. Our graph has $3k$ vertices, one for each variable in each clause. Clause r has vertices v_1^r, v_2^r, v_3^r. We put an edge between v_i^r and v_j^s if and only if:

- $r \neq s$
- their literals are consistent

Lemma 13.1. *If Φ is satisfiable, then G must contain a clique of size k.*

Proof. Assume Φ is satisfiable, which is to say that Φ has a satisfying truth assignment. We select the vertices corresponding to that truth assignment. Since this truth assignment satisfies Φ, there must exist one selected vertex in each box. Each of these is connected to every vertex in a different box, therefore there exists a clique in G. □

Lemma 13.2. *If G contains a clique of size k, then Φ is satisfiable.*

Proof. Assume G contains a clique of size k. One vertex of the clique must be in each box. Assign all variables corresponding to the vertices of the clique to be true. This must be a satisfying assignment because at least one true variable exists in each clause, and there are no inconsistent truth values. □

Theorem 13.1. *Φ is satisfiable if and only if G contains a clique of size k.*

Proof. This follows from the above lemmas. □

13.6 VERTEX-COVER

Given a graph $G = (V, E)$, does there exist a set of vertices V' of size at most k such that every edge in E is incident to one vertex in V'?

Let the certificate be the set of vertices V'. This is a subset of V, so it is polynomial with respect to the input. We can verify that V' is of size at most k by counting the vertices in V'. We can verify that the vertices in V' form a vertex cover of G by checking that every edge in E is incident to at least one vertex in V' in polynomial time.

We reduce from *Clique*. Let $\overline{G} = (V, \overline{E})$ be the complement of the graph G, where $\overline{E} = \{(u, v) | (u, v) \notin E\}$.

Lemma 13.3. *$\overline{G}$ has a vertex cover of size $|V| - k$ if G has a clique of size k.*

Proof. Assume G has a clique V' of size k. V V' is a vertex cover of G. Let $(u, v) \in \overline{E}$, u and v are not both in V'. Either u or v is in V V'. □

Lemma 13.4. *G has a clique of size k if $\overline{G}$ has a vertex cover of size $|V| - k$.*

Proof. Assume $\overline{G}$ has a vertex cover V' of size $|V| - k$. For all edges $(u, v) \in \overline{E}$, either u is in V' or v is in V'. For all vertices u,v and $u \neq v$ then if neither u nor v is in V' then $(u, v) \in E$. □

Theorem 13.2. *G has a clique of size k if and only if $\overline{G}$ has a vertex cover of size $|V| - k$.*

Proof. This follows from the preceding lemmas. □

13.7 HAM-CYCLE

Given a graph $G = (V, E)$, does there exist a simple cycle of at least size k that contains every vertex in V?

The proof that this is *NP-Complete* is beyond the scope of these notes, and not terribly interesting.

13.8 TSP

Given $G = (V, E)$ which is the complete graph on n vertices and each edge (u, v) has a cost $c(u, v)$, does G have a cycle which visits each vertex (except the start) exactly once with total cost of at most k? We call such a cycle a *TSP-tour*

Let the certificate be a permutation of the vertices which is $O(|V| + 1)$, so polynomial in size. We can verify a certificate by summing the edges between each of the subsequent vertices in the permutation and checking that the sum is at most k.

We reduce from *HAM-CYCLE*. Given a graph $G = (V, E)$ on which we would like to solve *HAM-CYCLE*, we create $G' = (V, E')$ where $E' = \{(u, v) | u, v \in V, u \neq v\}$. And we create a cost function:

$$c(u, v) = \begin{cases} 0 & \text{if } (u, v) \in E \\ 1 & \text{if } (u, v) \notin E \end{cases}$$

It should be easy to see that G has a *HAM-CYCLE* if and only if G' has a *TSP-tour* of $k \leq 0$.

13.9 SUBSET-SUM

13.10 0/1-Integer Programming

Beyond the scope of these notes, but interesting.

13.11 3-COLOR

Given a graph $G = (V, E)$, can we label the vertices of a graph using 3 colors such that no adjacent vertices have the same color?

Beyond the scope of these notes, but interesting.

Appendix A

Approximations

A.1 VERTEX-COVER

Create a copy of the edge list. Choose an arbitrary edge from this copy, and add both vertices on this edge to the set which will become the vertex cover. Then remove all edges in our edge list copy which are incident to either vertex on the chosen edge. Repeat this process until our edge list copy is empty.

Exercise for the reader: prove that this is a 2-approximation.

A.2 Bin Packing

We describe the "First-Fit" approximation. Given an object to pack and a list of open bins, add the object to the first bin into which it will fit. If it fits into no bins, open a new bin and insert the object.

It should be easy to see that no two bins will be less than half full, otherwise the object(s) in the second bin would have been added to the first.

The worst case for this approximation is that every bin will be about half full. In which case it is no worse than twice the optimal, so this is a 2-approximation.

Appendix B

Acknowledgements

The authors would like to thank Michelle Burrows for proof reading and correction.

Appendix C

License

Copyright (c) 2012, Alexis Beingessner, Simon Pratt
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The views and conclusions contained in the software and documentation are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the project.