1 Greedy Algorithms

We now turn our attention to another descriptively named set of algorithms: greedy algorithms. The idea is that we can build up a solution to a problem by making simple choices at every step and that each choice can be made to make the current solution as 'good' as possible, without worrying about future steps. This works very well for some problems, as well as being simple and easy to think about. For some problems, greedy search produces the optimal solution, and it turns out that there are a set of problems where greedy choices give a reasonable solution, but not the optimal one. This can be useful if there is no known algorithm that gives the optimal solution in reasonable time, since greedy algorithms are always cheap to run. To get a handle on when greedy choices work well you might want to think about playing Scrabble. You can choose to make the best word you can out of the letters you currently have available, or you can save a handy letter such as an 'E' for the future. If you always make the best word you can then you are making the greedy choice. I prefer to think of them as smash-and-grab algorithms, though. If you are competing for cake with your family then you grab the biggest piece you can get.

The most common applications of greedy algorithms that lead to optimal solutions are in graphs. I don't think that you've seen much about graphs yet in your computer science courses, so we'll have to start from scratch. They are one of the most useful data structures that there is.

1.1 Introduction to Graphs

Suppose that you want to model how a disease spreads. You start by choosing how the disease is transmitted, and then infect a person with the disease (virtually, rather than in reality). After that, anybody that comes within catching distance of the first person has some chance of contracting the disease, and then they can spread it further. To model this in a computer you need some way to depict all of this. Let's say you draw each person as a circle, and then put a line between any person who catches the disease and the person they caught it from. So what you get is a collection of circles (called nodes or vertices), with lines between them linking certain pairs (called edges or links). This is a computer science graph. It is an abstract representation of a set of objects and connections or relationships between them. They could be people and disease spread, program methods and function calls, computers and ethernet links, places and roads, websites and hyperlinks or many, many other things. We represent the objects as nodes and the connections as edges between the nodes. The fact that it is so abstract means that it applies to a massive number of applications, so research into types of graphs and algorithms that run on different types of graph is a very important topic for research.

There are effectively three types of graph. We separate them first into directed and undirected ones. Undirected graphs have no direction information on the edges, so you can travel from A to B as the reverse of traveling from B to A. Directed graphs have arrows on the edges, so that you can only travel one way along each edge. Think of one-way streets as examples of directed graphs. If the graphs are directed that we might have to worry about whether or not they have cycles, which measures whether, if we start from some node A we can possibly get back to A by following edges. Directed graphs without cycles are the third type of graph, and go by the unlovely name of dags, which stands for directed, acyclic graphs. They are of interest because they appear in many problems, such as showing causal relationships and hierarchies.

You might be wondering how to encode the graph inside the computer. Since graphs are represented as a set of vertices connected by edges, we need to work out how to encode that in a data structure. In fact there are three different data structures that are used: a list of edges, each defined by the two nodes it links, an adjacency list, which is a set of linked lists—one for each vertex—that hold the numbers of the vertices that that vertex has edges to, or an adjacency matrix, which is an $n \times n$ array (where n is the number of nodes) where each element of the array is a 1 if there is an edge between the two nodes that index that element, and a 0 otherwise. Slides 2.32 – 2.34 give examples of these three representations. It is important because choosing which of these data structures to use can change the cost of the algorithm considerably. It often depends on how many edges there are in the graph. If the graph is sparse, so that there are only a few edges, then the adjacency matrix is expensive to hold, whereas if the graph is dense with edges then it is very efficient. If the edges have weights on them, associated with some kind of cost, then this information can be added into the adjacency matrix very simply, but requires extra fields for the other data structures. How easy is it to modify the different representations to deal with directed graphs?

What problems might you wish to solve on a graph? One obvious thing is to search for a particular vertex in the graph. Another is a development of that, which is to search for the shortest path from vertex i to vertex j. There are two ways to do search on a graph, known as *depth-first* and *breadth-first*. The first algorithm is pretty much the classical one, where classical here means from the classics – it is described in the Greek myth of Theseus and the Minotaur, where Theseus is able to find the Minotaur at the centre of the labyrinth and retrace his steps by using a ball of string given to him by his lover, Ariadne. For the ball of string we keep a stack of places we have visited, pushing them onto the stack as we explore and popping them back off as we return. It will be more efficient to mark each vertex as visited or not as we go, rather than looking through the stack each time to see if we have already been there. This is equivalent to checking if there is any string there. Using just this we can perform an exploration of an undirected graph starting from some node `v` using (where `!` means 'not'):

```
function explore(v):
        visited[v] = true
        for each edge e connected to v:
                w = other vertex on e
                if !visited[w]:
                        explore(w)
```

Obviously, if the graph is not fully connected—so that there are some parts of the graph that are not reachable from `v`—then this will not explore the whole graph. We can make it do so, which completes the depth-first search algorithm, by calling `explore` on each vertex `v`:

```
function depthFirstSearch(graph G):
        for v in G:
                visited[v] = false
        for v in G:
                if !visited[v]:
                        explore(v)
```

This is a linear time algorithm, since it explores each vertex and then looks at each edge from both ends (once when looking at vertex `v` and once when looking at vertex `w`). So the cost is $\mathcal{O}(|V| + |E|)$, where $|V|$ means the number of vertices and $|E|$ the number of edges. The same algorithm works fine on directed graphs, although you obviously have to check that you are travelling the right way along each edge. **You should make a choice of graph representation and then implement this algorithm. Then modify it to deal with directed graphs. Design some tests for your implementation.**

Generally, graphs break down into a set of *connected components*, i.e., a set of subgraphs that are each connected. Keeping track of which component each vertex of the graph belongs to is a trivial extension of depth-first search. **You should work out how to do it and implement it.**

The other way to do search is *breadth-first*. The difference is simple. In depth-first search we started at a node and then recursively followed a path through the graph until we reached a place where there were no new paths to explore. We then back-tracked through the algorithm exploring new paths until we ended up back at the very start with no new paths to explore. The name *depth-first* describes this because we turn the graph into a search tree and with the current node as the root and follow it down through the levels. In breath-first search we look at all of the nodes at level 1 in the tree, then all of the ones at level 2, and so on. At each step we are building a list of the nodes one level lower down in the tree. **See if you can use this to build the algorithm before you read on, but be warned that it isn't easy.**

```
function breadthFirstSearch(graph G, vertex root):
        for all v in G:
                distance[v] = infinity
        distance[root] = 0
        queue = root
        while size(queue)>0:
                v = queue[0]
                remove v from queue
                for each edge e connected to v:
                        w = other vertex on e
                        if distance[w] == infinity:
                                add w to queue
                                distance[w] = distance[v] + 1
```

There are two obvious things about this algorithm. Firstly, it is not recursive, and secondly, it computes the distances between nodes, not just marking whether or not they have been visited. It is still $\mathcal{O}(|V|+|E|)$, which you should be able to see for yourself. If the graph is not fully connected then it ignores parts of the graph that cannot be reached from `root`.

When we are looking at computing distances on graphs we normally assume that there are weights on the edges that tell us the costs. These are not including in the breadth-first search, which just gives a cost of 1 to each move. It might seem trivial to include this by modifying the `+1` in the last line of the algorithm, but that doesn't allow for the possibility that there is a shorter path somewhere else in the graph. However, there is a greedy algorithm (Dijkstra's algorithm) that solves this problem, which we will look at next. It is important enough to get its own subsection, since it is our first greedy algorithm:

1.2 Dijkstra's Algorithm for Shortest Paths

Way back at the start of this section, before we got bogged down with graphs, we talked about greedy rules. The idea is to find a rule that picks the best option now and hope that this will be enough to lead to a universally good algorithm.

In order to find a shortest path, a suitable greedy rule could be: add the minimum edge that connects to the current path and does not form a cycle, since a cycle would be redundant. It is easy to construct examples where this rule fails, (**can you find one?** There is one in example 7.4.1 in the textbook). Dijkstra's algorithm uses this greedy rule, but only amongst those nodes that are one edge away from elements in the current shortest path. This gets rid of the problem in the example, and makes it a good greedy rule. You should try to persuade yourself that there are no more problems with it.

To make an implementation we then need two things: a loop invariant and a data structure to hold the queue of possible edges. The loop invariant says that for every node we have visited we have the shortest path, and for every node that we have not visited yet we have the shortest path that only goes through the parts of the graph we have visited. So how about the data structure? We want a queue of possible nodes that can be added, sorted into order of edge weight, so that we can pick the one that is the cheapest to travel down. This is known as a *priority queue* and we will discuss it after we look at the algorithm, which you might see that it is pretty much breadth-first search, except that we use the priority queue in place of the normal queue:

```
function Dijkstra(graph G, vertex root):
        for all v in G:
                distance[v] = infinity
                previous[v] = none
        distance[root] = 0
        previous[root] = 0
        queue = makePriorityQueue(G)
        while size(queue)>0:
                v = front(queue)
                remove v from queue
                for each edge e connected to v:
                        w = other vertex on e
                        if distance[w] > distance[v] + weight[v,w]:
                                distance[w] = distance[v] + weight[v,w]
                                previous[w] = v
                                modifyPriorityQueue(w)
```

There are three function calls in this code that we don't know about yet, all related to the priority queue. The first one makes the thing, the second removes the minimum from it (`front`), and the third modifies it when a vertex is added into the graph. The `modifyPriorityQueue(w)` function call is meant to reduce the key value for vertex `w` since it is now in the graph. It turns out that you can make the priority queue by using a variety of different data structures. You should think about how you would do it using an array (which simply holds the values for each vertex of the graph ordered by vertex order). Looking up a particular value and modifying it for the `modifyPriorityQueue` function are constant time operations, since we know exactly which element to look at. However, finding `front(queue)` involves a linear search through the array. Since the algorithm does the finding by removing the first element $|V|$ times and the modifying $|V| + |E|$ times, this leads to an $\mathcal{O}(|V|^2)$ algorithm. How about using a binary heap ($\mathcal{O}((|V| + |E|) \log |V|)$)? Which of these data structures you prefer depends upon the relative sizes of $|V|$ and $|E|$. There is another implementation of it that is even faster ($\mathcal{O}(|V| \log |V| + |E|)$) that is known as a Fibonacci Heap. Look it up if you are interested (see my `del.icio.us` links), but I'm pretty sure that you won't want to implement it!

Note that Dijkstra's algorithm does not deal with negative edges (can you see why?). There are algorithms that do, but we aren't going to talk about them in this course.

1.3 Kruskal's Algorithm for the Minimum Spanning Tree

The next graph algorithm that we are going to make finds the shortest path through a graph that visits all of the nodes. This is obviously going to be a tree, since we will not need any cycles (they are clearly a waste since they produce two possible paths between a pair of nodes). There is an example on slides 2.48 and 2.49. This might seem an odd thing to want to do, but it might be useful if we have a set of computers that need to be networked: we might want to connect them together so as to use the smallest amount of cable. Or given an internet package that contains an update for the servers, we might want to find the shortest way to route it to all other computers. If we know the costs of all possible paths in the graph then we can compute this using this greedy rule: add the cheapest edge that doesn't produce a cycle.

So we start with an empty graph, pick the cheapest edge, and add it in, and then iterate this, checking that we don't generate any cycles before adding each edge. Problem 5.1 asks you to work through one way to show that this algorithm will generate a *minimum* spanning tree (italics are to highlight the fact that it is the minimum part that is interesting – the rest is fairly obvious). A different way to understand it is by showing that it satisfies what is known as the *cut property*. A cut is a partition of the vertices of the graph into two parts, and the cut property says that if we cut the graph into sets of nodes that aren't currently connected to each other then we can always add the cheapest path that crosses a cut, since there can't be any other edges across it yet (since it is a cut). That's it.

Turning this into an program is a little more complicated, since we need to find the cheapest edge, which requires a sort. However, it boils down to the following (**As usual, have a go yourself first**):

```
function kruskal(V,E):
        sort E by weight
        for each vertex v in V:
                set[v] = v
        tree = {}
        count = 0
        while count < |E|:
                if (set[E[count].start] != set[E[count].end]):
                        add E[count] to tree
                        merge set[E[count].start] and set[E[count].end]
```

You should implement this algorithm carefully, there are still a few choices to make. Some careful thought should persuade you that the cost is $\mathcal{O}(|E| \log |E|)$.

1.4 Prim's Algorithm for the Minimum Spanning Tree

There is another greedy algorithm that does the same job. It adds the minimum weighted edge that has one vertex in the tree and the other not. It is still greedy, but it always guarantees to have a tree at every stage of the algorithm, which Kruskal's algorithm does not until it terminates.

Prim's algorithm is remarkably similar to Dijkstra's algorithm, similar enough that you should be able to work out the details for yourself. It is in section 7.3 of the text if you can't. Slides 2.52 and 2.53 show that ideas of the two algorithms and the different ways that they work.

1.5 Non-Optimal Greedy Algorithms

The rules that are applied in greedy algorithms are simple. While they might not always lead to simple algorithms, the rules themselves are easy to understand and apply. However, they do not always lead to optimal algorithms. That is, for some problems the answer that we find is not necessarily the best one. It might seem odd to want to use an algorithm that doesn't necessarily give the correct answer, but for some problems you might be prepared to take an answer that is good enough, especially if it is a lot cheaper to get than the true best answer. In general, these problems are optimisation problems of some kind: we want the shortest path through the graph, or the timetable with the fewest clashes for students, or the fastest way to get data through a network. These are often search problems in that we want to search through possible solutions to find good ones. In this context, greedy algorithms are often approximation algorithms.

We will look at one example of this now, which is the *travelling salesman problem* (TSP). This problem consists of finding the shortest path through a set of vertices that visits each vertex exactly once and that starts and ends in the same place. It has several applications in fields as diverse as logistics (where it gets its name from) and chip design. The TSP is easy to describe, so is it actually difficult to solve? The difficulty is that the search space of possible solutions is huge. If there are n cities in the map, then we can choose any one of them to be the first city, $n-1$ for the next, and $n-2$ for the one after that, and so on until there is only one city to choose from at the end. So the naïve algorithm that tries all possible routes (the exhaustive search method, so called because you'd be exhausted if you tried it) has time complexity $\mathcal{O}(n!)$. To get some handle on the size of this, $10! = 3,628,800$ and $20! = 2.43 \times 10^{18}$. It turns out that this is an example of a problem where we do not know a good algorithm to solve it. In fact, the TSP is an example of an NP-complete problem; while finding the answer is very difficult, and the best algorithm we know of has exponential time, checking the answer is quite easy: if you give me a route then I can compute the length of it very quickly. You will see more on NP problems later in the course.

We can create a greedy algorithm to find a solution to the TSP by picking a starting node at random and then always choosing the shortest path out of it that takes us to a new city. This is very simple, but it is certainly not guaranteed to produce a good solution. You should be able to see that it produces an algorithm that is quite similar to Kruskal's algorithm. We will see another algorithm for the travelling salesman problem in section 2.2.5.

1.6 Coding: Music to Our Ears

As you are undoubtedly aware, music and other sound is an analogue (continuously varying) signal, but music storage on computers and CDs, etc. is digital. Therefore, to store the data it has to be digitised. In general, samples are taken 44,100 times a second (44.1kHz). For a stereo sound signal, this means that for every hour of music, $2 \times 60 \times 60 \times 44,100 = 317,520,000$ numbers have to be stored. In general these would be real numbers, so they are quantised into one of a finite set (chosen to match human hearing), and then need to be encoded into binary. How large a datafile would that make? To reduce the size, the Huffman code is used. It was chosen for its efficiency in both storage and computational complexity. It is exactly what is used in MP3.

The idea of compression is to replace common characters or strings with shorter versions. This generally works by using a dictionary, which contains a list of the code characters and the strings that they represent. The challenge is to construct the dictionary and the set of code strings. To get the best compression you need to use very short codes for frequent items and longer codes for less common strings, which is imaginatively known as variable length encoding. One problem that you have to think about is how you know when one code word has stopped and the next one starts. This is particularly important since we will be making our code words out of binary digits. Say that `101` is a code and so are `1` and `10`. How did we know if the string `101` is one example of the first code, or one each of the latter two? One solution to this is to include stop bits, such as a double 0 at the end of each code. However, this is a waste of two bits. An nice alternative is known as a prefix code, which demands that no code is a prefix of another. So if `101` was in our codebook, we could not use `1` or `10` as codes, since they are prefixes of `101`. By doing this we don't need the extra characters of stop bits, encoding is just the concatenation of the code strings, and decoding works by starting at the beginning of the code and working until you recognise a dictionary entry, which is completely unambiguous.

A nice way to make a prefix code is to use a binary tree. Every time we read a 0 we go left in the tree, and every time we read a 1 we go right, as shown in slide 2.62. When we reach a leaf, the corresponding entry of the dictionary gets printed out. Note that the tree will always be balanced, which makes it efficient. So the problem is now to work out how to construct the tree is such a way that frequent items are high up the tree, and so have short codes. This is the problem that the Huffman code solves, and it does it by using a greedy rule, which says that the symbols with the two smallest frequencies must be at the bottom of the tree. So actually, it is kind-of the opposite of a greedy rule – it eats the smallest things first and only eats the big things at the end!

To make the rule work, we take the two smallest frequencies out of the list of frequencies and then make them both children of a node. The frequency of that node will be the sum of those of its children, so we put that sum back in the list and then iterate the rule. So all we need is a nice data structure to hold the list of frequencies, and since we want them sorted, a priority queue is a pretty obvious option. The algorithm is written out below, but as usual, it would be good to see if you can construct it for yourself first. Compression is a very interesting topic for further study, and one that we have barely scratched the surface of. Section 7.5 of the textbook also talks about the Huffman encoding problem.

```
function Huffman(f):
        H = priority queue of integers
        for i = 1..n:
                insert i into H
        for k = n+1..2n-1:
                i = deletemin(H)
                j = deletemin(H)
                k = new node with children i, j
                f[k] = f[i] + f[j]
                insert k into H
```

1.7 Makes A Change

Suppose that you want to give correct monetary change with the minimum number of coins, for example in a vending machine. Can you write a greedy algorithm to do this correctly? The general problem says that you have a set of coins with values d_i, so the New Zealand version has d_1 =10¢, d_2 =20¢, d_3 =50¢, d_4 =\$1, d_5 =\$2. Obviously, we can't get change less than 10¢ with this, so we won't try.

There is a pretty obvious exhaustive search that is very simple to write, but there is also a greedy solution, which is to start with the largest denomination and work downwards, since this makes the largest progress towards the goal. If the set of coins is sorted into size order then this is a $\mathcal{O}(N)$ algorithm. Will it find the optimal solution?

1.8 Practical Exercise Three: Covering Points

Suppose that you are setting up a wireless network to cover a town. You get a set of people who agree to subscribe to you, and you need to work out how many transmitters you need, and where to site them around the city. To do this, you plot the x and y coordinates of all the houses that you need to cover, and you find out that the transmitters will cover a circle of radius 100 metres. If the town is approximately a square of size 2 km by 2 km, design a greedy algorithm that locates the transmitters to cover your subscribers. Think about whether or not it is optimal before you implement it. Design and implement test cases for the problem. You will hear more about this problem in a very different context and with a different name (the k-means algorithm) in the Artificial Intelligence paper if you do that. Use the random number generator as in the previous problem to create the locations of the houses.

1.9 Practice Questions

Question 5.1: Kruskal's Algorithm Kruskal's algorithm for a minimum spanning tree has the following loop invariant:
The current edge set (`tree`) contains only edges of a minimum spanning tree for each subtree constructed so far.

Demonstrate that this invariant will hold for each iteration of the algorithm, and suggest a suitable precondition and postcondition.

Question 5.2: Breadth-first Search The breadth-first search on graphs has the following loop invariant:
The queue consists of all vertices that have been discovered, but not visited.

Demonstrate that this invariant will hold for each iteration of the algorithm, and suggest a suitable precondition and postcondition.

Question 5.3: Coins Suppose that the coins that you have are worth d_1 =4¢, d_2 =3¢, d_3 =1¢. Would your algorithm from Section 1.7 still be optimal? Can you identify a general rule that says whether or not a set of coins are fair?