COMP26122 - Algorithms and Imperative Programming

November 26, 2017

1 The greedy method

An **optimization problem** is a problem that involves searching through a set of configurations to find one that minimizes or maximizes an objective function defined on these configurations. The greedy algorithm design paradigm involves repeatedly making choices that optimize some objective function. In order to solve a given optimization problem, we proceed by a sequence of choices. The sequence of choices starts from some well-understood starting configuration, and then iteratively makes the decision that is best from all of those that are currently possible, in terms of improving the objective function.
A greedy approach does not always lead to an optimal solution, but there are problems possessing the **greedy-choice property** for which it works optimally. This is the property that a global optimal configuration can be reached by a series of **locally optimal choices** i.e. choices that are the best from among the possibilities available at the time, starting from a well-defined configuration.

1.1 The fractional knapsack problem

In the **knapsack problem**, we are given a set of n items, each having a weight and a benefit, and we are interested in choosing the set of items that maximize our total benefit while not going over the weight capacity of the knapsack.
In the "0-1" version of the problem, fractional choices are not allowed, so we are restricted to entirely accepting or rejecting each item.
Given a set S of n items, such that each item i has a positive benefit b_i and a positive weight w_i, and given that the maximum-benefit subset that cannot exceed a given weight W, we can take an amount x_i of each item i such that

$$0 \leq x_i \leq w_i \text{ for each } i \in S \text{ and } \sum_{i \in S} x_i \leq W$$

The total benefit of the items taken is determined by the objective function

$$\sum_{i \in S} b_i (x_i / w_i)$$

1.1.1 Using the Greedy Method to Solve the Fractional Knapsack Problem

In applying the greedy method, the most important decision is to determine the objective function that we wish to optimize. A correct approach is to rank the items by their **value**, which is the ratio of their benefits and weights. The intuition behind this is that benefit-per-weight-unit is a natural measurement of the inherent value that each item possesses. Indeed, this approach leads to an efficient algorithm that finds an optimal solution to the fractional knapsack problem.

Algorithm 1 `FractionalKnapsack`(S, W)

Input: Set S of items, such that each item $i \in S$ has a positive benefit b_i and a positive weight w_i; positive maximum total weight W

Output: Amount x_i of each item $i \in S$ that maximizes the total benefit while not exceeding the maximum total weight W

for each item $i \in S$ **do**
    $x_i \leftarrow 0$
    $v_i \leftarrow {}^{b_i}/_{w_i}$ // **value index** of item i
$w \leftarrow 0$ // total weight
while $w < W$ and $S \neq 0$ **do**
    remove from S an item i with highest value index // greedy choice
    $a \leftarrow \min\{w_i, W - w\}$ // more than $W - w$ causes a weight overflow
    $x_i \leftarrow a$
    $w \leftarrow w + a$

1.1.2 Analysing the Greedy Algorithm for the Fractional Knapsack Problem

The above implementation take $O(n \log n)$ time, where n is the number of items in S. If we use a heap-based priority queue to store the items of S, where the key of each item is its value index, each greedy choice, which removes the item with greatest value index, takes $O(\log n)$ time. Alternatively, we could even sort the items by their benefit-to-weight values ($O(n \log n)$), and then process them in this order($O(n)$).

Theorem 1.1 *Given a collection S of n items, such that each item i has a benefit b_i and weight w_i, we can construct a maximum-benefit subset of S, allowing for fractional amounts, that has a total weight W in $O(n \log n)$ time.*

Proof 1 *For the sake of contradiction, suppose there is an optimal solution better than the one chosen by this greedy algorithm. Then there must be two items i and j such that:*

$$x_i < w_i, \quad x_j > 0, \quad v_i > v_j$$

Let

$$y = \min\{w_i - x_i,\ x_j\}$$

But then we could replace an amount y of item j with an equal amount of item i, increasing the total benefit without changing the total weight, which contradicts the assumption that this non-greedy solution is optimal. Therefore, we can correctly compute optimal amounts for the items by greedily choosing items by increasing benefit-to-weight values.

The proof of this theorem uses an **exchange argument** to show that the greedy method works to solve this problem optimally. The general structure of such an argument is a proof by contradiction, where we assume, for the sake of reaching a contradiction, that there is a better solution than one found by the greedy algorithm. We then argue that there is an exchange that we could make among the components of this solution that would lead to a better solution.

1.2 Task scheduling

The **task scheduling** problem is to schedule all the tasks in a given set T on the fewest machines possible in a **nonconflicting** (i.e. not overlapping in time) way.

As with any greedy strategy, the challenge is to find the right objective function. For example, we might consider the tasks from longest to shortest, assigning them to machines based on the first one available, since the longest tasks seem like they would be the hardest to schedule. Unfortunately, this approach does not necessarily result in an optimal solution.

1.2.1 A better greedy approach to task scheduling

Algorithm 2 `TaskSchedule`(T)

Input: Set T of tasks, such that each task has a start time s_i and a finish time f_i
Output: A nonconflicting schedule of the tasks in T using a minimum number of machines

```
m ← 0                                              // optimal number of machines
while T ≠ 0 do
    remove from T the task i with smallest start time s_i
    if there is a machine j with no task conflicting with task i then
        schedule task i on machine j
    else
        m ← m + 1
        schedule task i on machine m
```

We begin with no machines and we consider the tasks in a greedy fashion, ordered by their start times. For each task i, if we have a machine that can handle task i, then we schedule i on that machine, choosing the first such available machine. Otherwise, we allocate a new machine, schedule i on it, and repeat this greedy selection process until we have considered all the tasks in T.

The intuition behind this approach is that, by processing tasks by their start times, when we process a task for a given start time we will have already processed all the other tasks that would conflict with this starting time.

1.2.2 Analysis of the greedy task scheduling algorithm

Theorem 1.2 *Given a set of n tasks specified by their start and finish times,* `TaskSchedule` *produces, in $O(n \log n)$ time, a schedule for the tasks using a minimum number of machines.*

In **lower-bound** argument, which is another technique for proving greedy algorithms are correct, we argue that any solution to our problem will require a cost of at least some given parameter and we then show that the greedy algorithm achieves this lower bound as an upper bound.

1.3 Text compression and Huffman coding

In a **variable-length encoding scheme**, the codes for various characters are allowed to have different lengths; ideally, the most frequently used characters use the fewest number of bits. In a **prefix code**, no code word in the scheme is a prefix of any other code word in our scheme. A greedy strategy can thus be used for decoding, processing the bits of Y in order, repeatedly matching bits to the first code word they represent.
The savings produced by a variable-length prefix code can be significant, particularly if there is a wide variance in character frequencies. However, the challenge is that to get the maximum compression possible with this approach we want to guarantee that high-frequency characters are assigned to short code words and low-frequency characters are assigned to longer code words.

1.3.1 Huffman coding

This method produces a variable-length prefix code for a string X based on the construction of a proper binary tree T that represents the code. Each edge in T represents a bit in a code word, with each edge to a left child representing a 0 and each edge to a right child representing

a 1. Each external node v is associated with a specific character, and the code word for that character is defined by the sequence of bits associated with the edges in the path from the root of T to v.

Since characters are associated only with external nodes, and no internal node is associated with any code word, such a scheme produces a prefix code. If we start matching bits in X based on the path that they trace out in T, then the external node that we reach will correspond to the character represented by the code word this string of bits is equal to.

1.3.2 The Huffman coding algorithm

Let C be a collection of characters, with each character c in C having a numeric weight $f(c)$ corresponding to its frequency.
Let T be a binary tree, with a numeric weight $f(v)$ assigned to each external node v in T, and a weight $f(v)$ assigned to each internal node v that is the sum of the weights of its external-node descendants. Define the **total path weight** $p(T)$ of T to be the sum of all the weights in T. It is equivalent to summing the product of each external node's weight and its depth, over all the external nodes of T.
The goal is to construct T so that it has minimum total path weight over all binary trees having external nodes associated with the characters in C, using the frequency of each character in C as the weight of its associated external node.

Algorithm 3 `Huffman`(C)

Input: Set C of d characters, each with a given weight $f(c)$
Output: A coding tree T for C with a minimum total path weight

```
Initialise a priority queue Q
for each character c in C do
    Create a single-node binary tree T storing c
    Insert T into Q with key f(c)
while Q.size() > 1 do
    f1 ← Q.minKey()
    T1 ← Q.removeMin()
    f2 ← Q.minKey()
    T2 ← Q.removeMin()
    Create a new binary tree T with left subtree T1 and right subtree T2
    Insert T into Q wiht key f1 + f2
return tree Q.removeMin()
```

1.3.3 Analysis of the Huffman coding algorithm

Each iteration of the while-loop can be implemented in $O(\log d)$ time using a priority queue represented with a heap. Each iteration takes two binary trees out of Q and adds one in, done in $O(\log d)$ time. This process is repeated $d-1$ times before exactly one node is left in Q. Thus, this algorithm runs in $O(d \log d)$ time, assuming we are given the set C of d distinct characters in the string X as input. We can construct C from X in $O(n)$ time.

If T is a binary tree, with minimum total path weight for a set C of characters, with each c in C having a positive weight $f(c)$, then T is proper, that is, each internal node in T has two children.

Given a set C of characters, with a positive weight $f(c)$, defined for each c in C, two characters b and c with the smallest two weights are associated with nodes that have the maximum depth and are siblings in a binary tree T with minimum total path weight for C.

Theorem 1.3 *The Huffman coding algorithm constructs an optimal prefix code for a string of length n with d distinct characters in $O(n + d \log d)$ time.*

1.3.4 How the Huffman coding algorithm uses the greedy method

In order to solve the given optimization code problem using the greedy method, we proceed by a sequence of choices. The sequence starts from a well-understood starting condition, and computes the cost for that initial condition. Finally, we iteratively make additional choices by identifying the decision that achieves the best cost improvement from all of the choices that are currently possible.

This approach does not always lead to an optimal solution, but it does indeed find the optimal prefix code when used according to the approach of the Huffman coding algorithm.
This global optimality for the Huffman coding algorithm is due to the fact that the optimal prefix coding problem possesses the greedy-choice property. In this case, the general condition is defined by a set of disjoint binary trees, each with a given weight for its root, and the greedy choice is to combine two lowest- weight trees into a single tree.

2 Divide-and-Conquer

2.1 Recurrences and the master theorem

The **divide-and-conquer** technique involves solving a particular computational problem by dividing it into one or more subproblems of smaller size, recursively solving each subproblem, and then "merging" or "marrying" the solutions to the subproblem(s) to produce a solution to the original problem.

We can model the divide-and-conquer approach by using a parameter n to denote the size of the original problem, and let $S(n)$ denote this problem. We solve the problem $S(n)$ by solving a collection of k subproblems $S(n_1), S(n_2), \ldots, S(n_k)$, where $n_i < n$ for $i = 1, \ldots, k$, and then merging the solutions to these subproblems.

The general divide-and-conquer technique can be used to build algorithms that have fast running times.To analyze the running time, we often use a **recurrence equation**, that is, we let a function $T(n)$ denote the running time of the algorithm on an input of size n, and we characterize $T(n)$ using an equation that relates $T(n)$ to values of the function T for problem sizes smaller than n.

The Iterative Substitution Method One way to solve a divide-and-conquer recurrence equation is to use the **iterative substitution method**, also known as the "plug-and-chu" method. This comes from the way this method involves "plugging" in the recursive part of an equation for $T(n)$ and then often "chugging" through a considerable amount of algebra in order to get this equation into a form where we can infer a general pattern.
We assume that the problem size n is fairly large and we then substitute the general form of the recurrence for each occurrence of the function T on the right-hand side.
The hope in applying the iterative substitution method is that at some point we will see a pattern that can be converted into a general closed-form equation.

The recursion tree Like the iterative substitution method, this technique uses repeated substitution to solve a recurrence equation, but it differs from the iterative substitution method in that, rather than being an algebraic approach, it is a visual approach.
A tree R is drawn, where each node represents a different substitution of the recurrence equation.

Thus, each node in R has a value of the argument n of the function $T(n)$ associated with it. In addition, we associate an **overhead** with each node v in R, defined as the value of the nonrecursive part of the recurrence equation for v. For divide-and-conquer recurrences, the overhead corresponds to the running time needed to merge the subproblem solutions coming from the children of v. The recurrence equation is then solved by summing the overheads associated with all the nodes of R. This is commonly done by first summing values across the levels of R and then summing up these partial sums for all the levels of R.

The Guess-and-Test method This technique involves first making an educated guess as to what a closed-form solution of the recurrence equation might look like and then justifying that guess, usually by induction. For example, we can use the guess-and-test method as a kind of "binary search" for finding good upper bounds on a given recurrence equation. If the justification of our current guess fails, then it is possible that we need to use a faster-growing function, and if our current guess is justified "too easily", then it is possible that we need to use a slower-growing function. However, using this technique requires us being careful, in each mathematical step we take, in trying to justify that a certain hypothesis holds with respect to our current "guess".

Just because one inductive hypothesis for $T(n)$ does not work, that does not necessarily imply that another one proportional to this one will not work.
The guess-and-test method can be used to establish either an upper or lower bound for the asymptotic complexity of a recurrence equation.

2.1.1 The master theorem

The master theorem is a "cookbook" method for determining the asymptotic characterisation of a wide variety of recurrence equations. Namely, it is used for recurrence equations of the form

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{otherwise} \end{cases}$$

where $d \geq 1$ is an integer constant, $a \geq 1$, $c > 0$, and $b > 1$ are real constants, and $f(n)$ is a function that is positive for $n \geq d$. Such a recurrence equation would arise in the analysis of a divide-and-conquer algorithm that divides a given problem into a subproblems of size at most n/b each, solves each subproblem recursively, and then "merges" the subproblem solutions into a solution to the entire problem. The function $f(n)$, in this equation, denotes the total additional time needed to divide the problem into subproblems and merge the subproblem solutions into a solution to the entire problem.

Theorem 2.1 *Let $f(n)$ and $T(n)$ be defined as above.*

1. *If there is a small constant $\epsilon > 0$, such that $f(n)$ is $O(n^{\log_b a - \epsilon})$, then $T(n)$ is $\Theta(n^{\log_b a}) \rightarrow$ $f(n)$ is polynomially smaller than the function.*
2. *If there is a constant $k \geq 0$, such that $f(n)$ is $O(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n) \rightarrow$ $f(n)$ is asymptotically close to the function.*
3. *If there is a small constant $\epsilon > 0$ and $\delta < 1$, such that $f(n)$ is $\Omega(n^{\log_b a + \epsilon})$ and $af(n/b) \leq \delta f(n)$, for $n \geq d$, then $T(n)$ is $\Theta(f(n)) \rightarrow f(n)$ is polynomially larger than the function.*

2.1.2 High level justification of the master theorem

If we apply the iterative substitution method to the general divide-and-conquer recurrence equation, we get

$$
\begin{aligned}
T(n) &= aT(n/b) + f(n) \\
&= a(aT(n/b^2) + f(n/b)) + f(n) = a^2T(n/b^2) + af(n/b) + f(n) \\
&= a^3T(n/b^3) + a^2T(n/b^2) + af(n/b) + f(n) \\
&\quad \vdots \\
&= a^{\log_b n}T(1) + \sum_{i=0}^{\log_b n - 1} a^i f(n/b^i) \\
&= n^{\log_b a}T(1) + \sum_{i=0}^{\log_b n - 1} a^i f(n/b^i)
\end{aligned}
$$

This equation is where the special function comes from. Given this closed-form characterisation of $T(n)$, we can intuitively see how each of the three cases is derived:

1. Case 1 comes from the situation when $f(n)$ is small and the first term above dominates.
2. Case 2 denotes the situation when each of the terms in the above summation is proportional to the others, so the characterisation of $T(n)$ is $f(n)$ times a logarithmic factor.
3. Case 3 denotes the situation when the first term is smaller than the second and the summation above is a sum of geometrically-decreasing terms that start with $f(n)$; hence, $T(n)$ is itself proportional to $f(n)$.

2.2 Integer multiplication

The problem of multiplying **big integers**, is that integers represented by a large number of bits cannot be handled directly by the arithmetic unit of a single processor. Multiplying big integers has applications to data security, where big integers are used in encryption schemes.

Given two big integers I and J represented with n bits each, we compute $I + J$ and $I - J$ in $O(n)$ time. Efficiently computing the product $I \cdot J$ takes $O(n^2)$ time.

Let us assume that n is a power of two. We can therefore divide the bit representations of I and J in half, with one half representing the higher-order bits and the other representing the lower-order bits. In particular, if we split I into I_h and I_l and J into J_h and J_l, then

$$I = I_h 2^{n/2} + I_l$$

$$J = J_h 2^{n/2} + J_l$$

Multiplying a binary number I by a power of two involves shifting left the number I by k bit positions, which takes $O(k)$ time.
We can compute $I \cdot J$ by applying a divide-and-conquer algorithm that divides the bit representations of I and J in half, recursively computes the product as four products of $n/2$ bits each, and then merges the solutions to these subproducts in $O(n)$ time using addition and multiplication by powers of two. We can terminate the recursion when we get down to the multiplication of two 1-bit numbers. This divide-and-conquer algorithm has a running time $\Theta(n^2)$.

2.2.1 A better algorithm

If we can reduce the number of recursive calls, then we will reduce the complexity of the special function used in the master theorem, which is currently the dominating factor in our running time.

When expading $(I_h - I_l) \cdot (J_l - J_h)$, it contains two of the products we want to compute. and two products that can be computed recursively. Thus, $I \cdot J$ can be computed as follows:

$$I \cdot J = I_h J_h 2^n + [(I_h - I_l) \cdot (J_l - J_h) + I_h J_h + I_l J_l] 2^{n/2} + I_l J_l$$

Theorem 2.2 *We can multiply two n-bit integers in $O(n^{\log_2 3})$ time, which is $O(n^{1.585})$.*

This is done applying the master theorem with the special function

$$n^{\log_b a} = n^{\log_2 3}$$

We can actually achieve a running time that is "almost" $O(nlogn)$, by using a more complex divide-and-conquer algorithm called the **fast Fourier transform**.

3 Graphs and Traversals

3.1 Graph terminology and representation

A **graph** G is a set, V, of **vertices** and a collection, E, of pairs of vertices from V, which are called **edges**. Thus, a graph is a way of representing connections or relationships between pairs of objects from some set V.

An edge (u, v) is said to be **directed** from u to v if the pair (u, v) is ordered, with u preceding v. An edge (u, v) is said to be **undirected** if the pair (u, v) is not ordered. Undirected edges are sometimes denoted with set notation, u, v: (u, v) is the same as (v, u). Graphs are typically visualised by drawing the vertices as circles or rectangles and the edges as segments or curves connecting pairs of these circles or rectangles.

3.1.1 Some graph terminology

If all the edges in a graph are undirected, then the graph is an **undirected graph**. Likewise, a **directed graph**, also called a **digraph**, is a graph whose edges are all directed. A graph that has both directed and undirected edges is a **mixed graph**. An undirected or mixed graph can be converted into a directed graph by replacing every undirected edge (u, v) by the pair of directed edges (u, v) and (v, u).

The two vertices joined by an edge are called the **end vertices** of the edge, or the **endpoints** of that edge. If an edge is directed, its first endpoint is its **origin** and the other is the **destination** of the edge.

Two vertices are **adjacent** if they are endpoints of the same edge. An edge is **incident** on a vertex if the vertex is one of the edge's endpoints.
The **outgoing edges** of a vertex are the directed edges whose origin is that vertex.
The **incoming edges** of a vertex are the directed edges whose destination is that vertex.
The **degree** of a vertex v, denoted $degree(v)$, is the number of incident edges of v. The **in-degree** and **out-degree** of a vertex v are the number of the incoming and outgoing edges of v, and are denoted indeg(v) and outdeg(v), respectively.

The definition of a graph groups edges in a **collection**, allowing for two undirected edges to have the same end vertices, and for two directed edges to have the same origin and destination. Such edges are called **parallel edges** or **multiple edges**. Parallel edges may exist in a **flight network**, in which case multiple edges between the same pair of vertices could indicate different flights operating on the same route at different times of the day. An edge (undirected or directed)

is a **self-loop** if its two endpoints coincide. A self-loop may occur in a graph associated with a city map, where it would correspond to a curving street that returns to its starting point.

Simple graphs do not have parallel edges or self-loops, thus its edges are a **set** of vertex pairs.

Theorem 3.1 *If G is a graph with m edges, then*

$$\sum_{v \in G} \deg(v) = 2m$$

Proof 2 *An edge (u, v) is counted twice in the summation: once by its endpoint u and once by its endpoint v. Thus, the total contribution of the edges to the degrees of the vertices is twice the number of edges.*

Theorem 3.2 *If G is a directed graph with m edges, then*

$$\sum_{v \in G} \text{indeg}(v) = \sum_{v \in G} \text{outdeg}(v) = m$$

Proof 3 *In a directed graph, an edge (u, v) contributes one unit to the out-degree of its origin u and one unit to the in-degree of its destination v. Thus, the total contribution of the edges to the out-degrees of the vertices is equal to the number of edges, and similarly for the in-degrees.*

Theorem 3.3 *A simple graph G with n vertices and m edges has $O(n^2)$ edges.*
If G is undirected, then

$$m \leq n(n-1)/2$$

and if G is directed, then

$$m \leq n(n-1)$$

Proof 4 *Suppose that G is undirected. Since no two edges can have the same endpoints and there are no self-loops, the maximum degree of a vertex in G is $n-1$. By Theorem 3.1, $2m \leq n(n-1)$. Suppose that G is directed. Since no two edges can have the same origin and destination, and there are no self-loops, the maximum in-degree of a vertex in G is $n-1$. By Theorem 3.2, $m \leq n(n-1)$.*

A **path** in a graph is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex, such that each edge is incident to its predecessor and successor vertex. A **cycle** is a path with the same start and end vertices. A path is **simple** if each vertex in the path is distinct, and we a cycle is **simple** if each vertex in the cycle is distinct, except for the first and last one. A **directed path** is a path such that all the edges are directed and are traversed along their direction. A **directed cycle** is defined similarly.

A **subgraph** of a graph G is a graph H whose vertices and edges are subsets of the vertices and edges of G. A **spanning subgraph** of G is a subgraph of G that contains all the vertices of the graph G. A graph is **connected** if, for any two vertices, there is a path between them. If a graph G is **not connected**, its maximal connected subgraphs are called the **connected components** of G. A **forest** is a graph without cycles. A **tree** is a connected forest (a connected graph without cycles). Trees have no root, and they are called **free trees**. A spanning tree of a graph is a spanning subgraph that is a free tree.

Theorem 3.4 *Let G be an undirected graph with n vertices and m edges. Then we have the following:*

- *If G is connected, then $m \geq n - 1$*
- *If G is a tree, then $m = n - 1$*
- *If G is a forest, then $m \leq n - 1$*

3.1.2 Operations on graphs

- Return the number n of vertices in G
- Return the number m of edges in G
- Return a set or list containing all n vertices in G
- Return a set or list containing all m edges in G
- Return some vertex v in G
- Return the degree $\deg(v)$ of a given vertex v in G
- Return a set or list containing all the edges incident upon a given vertex v in G
- Return a set or list containing all the vertices adjacent to a given vertex v in G
- Return the two end vertices of an edge e in G; if e is directed, indicate which vertex is the origin of e and which is the destination of e
- Return whether two given vertices v and w are adjacent in G
- Indicate whether a given edge e is directed in G
- Return the in-degree of v $\text{indeg}(v)$
- Return a set or list containing all the incoming (or outgoing) edges incident upon a given vertex v in G
- Return a set or list containing all the vertices adjacent to a given vertex v along incoming (or outgoing) edges in G
- Insert a new directed (or undirected) edge e between two given vertices v and w in G
- Insert a new (isolated) vertex v in G
- Remove a given edge e from G
- Remove a given vertex v and all its incident edges from G

Any edge or vertex can store additional information, including numeric weights, Boolean values, or even pointers to general objects.

3.1.3 Data structures for representing graphs

Deciding which representation to use for a particular graph G boils down to determining how **dense** G is. For instance, if G has close to a quadratic number of edges, then the adjacency matrix is often a good choice for representing G, but if G has close to a linear number of edges, then the adjacency list representation is probably superior.

The adjacency list structure For a graph G, it includes:

- A collection V of n vertices. This could be a set, list, or array, or it could be defined implicitly as simply the integers from 1 to n. If vertices can store data, there also needs to be some way to map each vertex v to the data associated with v.
- A collection E of m edges. This collection could be a set, list, or array, or it could be defined implicitly by the pairs of vertices that are determined by adjacency lists. If edges can store data, there also needs to be some way to map each edge e to the data associated with e.

- For each vertex v in V, we store the adjacency list for v, that represents all the edges incident on v. This is implemented either as a list of references to each vertex w such that (v, w) is an edge in E, or it is implemented as a list of references to each edge e that is incident on v. If G is a directed graph, then the adjacency list for v is typically divided into two parts: one representing the incoming edges for v and one representing the outgoing edges for v.

For a vertex v, the space used by the adjacency list for v is proportional to the degree of v $\rightarrow O(\deg(v))$. The space requirement of the adjacency list structure for a graph G of n vertices and m edges is $O(n + m)$.
Returning the incident edges or adjacent vertices for a vertex v run in $O(\deg(v))$ time.
Determining whether two vertices u and v are adjacent can be performed by inspecting either the adjacency list for u or that of v. By choosing the smaller of the two, we get $O(\min\{\deg(u), \deg(v)\})$ running time for this operation.

The adjacency matrix structure We number the vertices $1, 2, \ldots, n$ and we view the edges as being pairs of such integers. Historically, the adjacency matrix was the first representation used for graphs, with the adjacency matrix being a Boolean $n \times n$ matrix A:

$$A[i, j] = \begin{cases} 1 & \text{if } (i, j) \text{ is an edge of } G \\ 0 & \text{otherwise} \end{cases}$$

Modern instances of an adjacency matrix representation have a graph G with an $n \times n$ array A, such that $A[i, j]$ stores a reference to an edge object e if there is an edge $e = (i, j)$ in G. If there is no edge, then $A[i, j]$ is `null`.

Using an adjacency matrix A, we can determine whether two vertices v and w are adjacent in $O(1)$ time. We can access the vertices v and w to determine their respective indices i and j, and then testing whether the cell $A[i, j]$ is `null` or not. This performance achievement is traded off by an increase in the space usage, which is now $O(n^2)$, and in the running time of some other graph operations as well. For example, listing out the incident edges or adjacent vertices for a vertex v now requires that we examine an entire row or column of the 2D-array A which takes $O(n)$ time.

3.2 Depth-First Search

A **traversal** is a systematic procedure for exploring a graph by examining all of its vertices and edges. A traversal is efficient if it visits all the vertices and edges in linear time.

Traversing a graph using the backtracking technique

4 Shortest paths

In general, a **weighted graph** is a graph that has a numeric label, $w(e)$, associated with each edge, e, called the **weight** of edge e. Edge weights can be integers, rational numbers, or real numbers, which represent a concept such as distance, connection costs, or affinity.

4.1 Single-source shortest paths

Let G be a weighted graph. The **length** (or **weight**) of a path, P, in G, is the sum of the weights of the edges of P. That is, if P consists of edges, $e_0, e_1, \ldots, e_{k-1}$, then the length of P,

denoted $w(P)$, is defined as

$$w(P) = \sum_{i=0}^{k-1} w(e_i)$$

The **distance** from a vertex v to a vertex u in G, denoted $d(v, u)$, is the length of a minimum length path (also called **shortest path**) from v to u, if such a path exists.
It is often used the convention that $d(v, u) = +\infty$ if there is no path at all from v to u in G. Even if there is a path from v to u in G, the distance from v to u may not be defined, however, if there is a cycle in G whose total weight is negative The existence of such paths allows us to build arbitrarily low negative-cost paths, but distances cannot be arbitrarily low negative numbers. Thus, any time we use edge weights to represent distances, we must be careful not to introduce any negative-weight cycles.

4.2 Dijkstra's algorithm

A productive approach for applying the greedy method pattern to the single-source shortest-path problem is to perform a "weighted" breadth-first search starting at v. We can develop an algorithm that iteratively grows a "cloud" of vertices out of v, with the vertices entering the cloud in order of their distances from v. In each iteration, the next vertex chosen is the vertex outside the cloud that is closest to v. The algorithm terminates when no more vertices are outside the cloud, at which point we have a shortest path from v to every other vertex of G.
Applying the greedy method to the single-source shortest-path problem in this way results in an algorithm known as Dijkstra's algorithm. We assume that the input graph G is undirected and simple. We denote the edges of G as unordered vertex pairs (u, z).

4.2.1 Edge relaxation

We define a **label**, $D[u]$, for each vertex u of G, which is used to approximate the distance in G from v to u. $D[u]$ will always store the length of the best path we have found so far from v to u. Initially, $D[v] = 0$ and $D[u] = +\infty$ for each $u \neq v$, and the set C, which is our "cloud" of vertices, is set to be the empty set $\emptyset$. At each iteration of the algorithm, we select a vertex u not in C with smallest $D[u]$ label, and we pull u into C. In the very first iteration v is pulled into C. Once a new vertex u is pulled into C, $D[z]$ of each vertex z that is adjacent to u and is outside of C, is updated to reflect the fact that there may be a new and better way to get to z via u. This update operation is known as a **relaxation** procedure, for it takes an old estimate and checks whether it can be improved to get closer to its true value. In the case of Dijkstra's algorithm, the relaxation is performed for an edge (u, z), such that we have computed a new value of $D[u]$ and wish to see if there is a better value for $D[z]$ using the edge (u, z). The specific edge relaxation operation is as follows:

if $D[u] + w((u, z)) < D[z]$ **then**
$D[u] + w((u, z)) < D[z]$

Note that if the newly discovered path to z is no better than the old way, then we do not change $D[z]$.

4.2.2 The details of Dijkstra's algorithm

Algorithm 4 `DijkstraShortestPaths`(G, v)

Input: A simple undirected weighted graph G with non-negative edge weights; a distinguished vertex v of G

Output: A label, $D[u]$, for each vertex u of G, such that $D[u]$ is the distance from v to u in G

```
D[v] ← 0
for each vertex u ≠ v of G do
    D[u] ← +∞
Let a priority queue, Q, contain all the vertices of G using the D labels as keys
while Q is not empty do
    //pull a new vertex u into the cloud
    u ← Q.removeMin()
    for each vertex z adjacent to u such that z is in Q do
        if D[u] + w((u, z)) < D[z] then
            //perform the relaxation procedure on edge (u, z)
            D[z] ← D[u] + w((u, z))
            Change the key for vertex z in Q to D[z]
return the label D[u] of each vertex u
```

4.2.3 Why it works

"In Dijkstra's algorithm, whenever a vertex u is pulled into the cloud, the label $D[u]$ is equal to $d(v, u)$, the length of a shortest path from v to u."
Suppose that $D[t] > d(v, t)$ for some vertex t in V, and let u be the first vertex the algorithm removed from Q, such that $D[u] > d(v, u)$. There is a shortest path P from v to u. Considering the moment when u is pulled into C, let z be the first vertex of P (when going from v to u) that is not in C at this moment. Let y be the predecessor of z in path P. We know, by our choice of z, that y is already in C at this point. Moreover, $D[y] = d(v, y)$, since u is the first incorrect vertex. When y was pulled into C, we tested (and possibly updated) $D[z]$ so that we had at that point

$$D[z] \leq D[y] + w((y, z)) = d(v, y) + w((y, z))$$

But since z is the next vertex on the shortest path from v to u, this implies that

$$D[z] = d(v, z)$$

But we are now at the moment when we are picking u, not z, to join C. Thus

$$D[u] \leq D[z]$$

A **subpath** of a shortest path is itself a shortest path. Hence, since z is on the shortest path from v to u

$$d(v, z) + d(z, u) = d(v, u)$$

Moreover, $d(z, u) \geq 0$ because there are no negative-weight edges. Therefore

$$D[u] \leq D[z] = d(v, z) \leq d(v, z) + d(z, u) = d(v, u)$$

But this contradicts the definition of u; hence, there can be no such vertex u.

4.2.4 The running time of Dijkstra's Algorithm

We assume that the edge weights can be added and compared in constant time.
Assume that we are representing the graph G using an adjacency list structure. This data structure allows us to step through the vertices adjacent to u during the relaxation step in time

proportional to their number
An efficient implementation of the priority queue Q uses a heap. This allows us to extract the vertex u with smallest D label, by calling the `removeMin` method, in $O(\log n)$ time. As the standard heap structure does not support a removal method for arbitrary elements, we can maintain a pointer with each vertex, v, that supports constant-time access to the node in our heap that is holding v. Given a pointer to this node, we can remove v or update its key and perform the associated up-heap or down-heap bubbling as needed in $O(\log n)$ time.

- Inserting all the vertices in Q with their initial key value can be done in $O(n \log n)$ time by repeated insertions, or in $O(n)$ time using bottom-up heap construction.
- At each iteration of the while loop, we spend $O(\log n)$ time to remove vertex u from Q, and $O(\deg(v) \log n)$ time to perform the relaxation procedure on the edges incident on u.