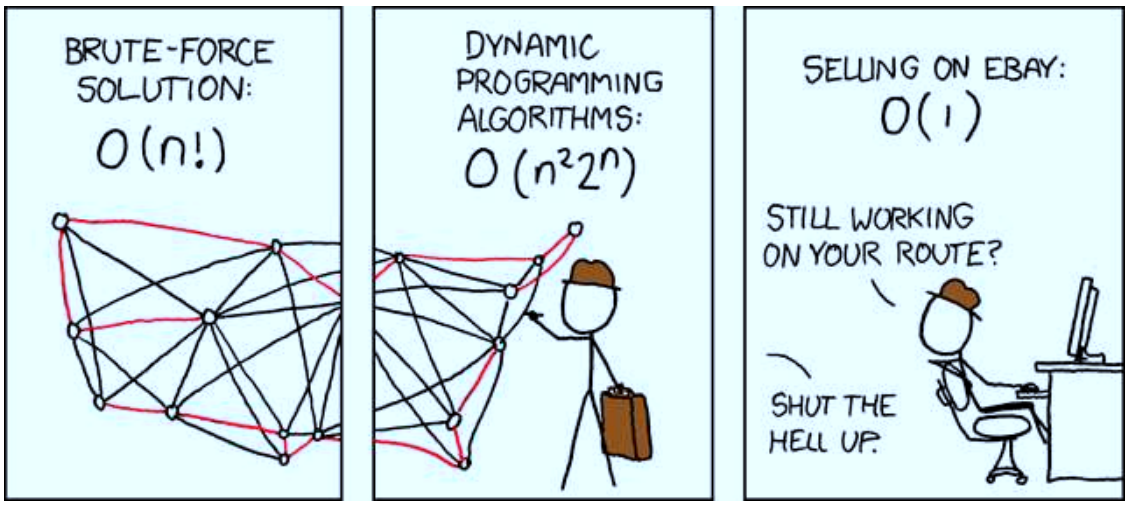
Computer Science and Software Engineering, 2008

CITS3210 Algorithms

Lecture Notes

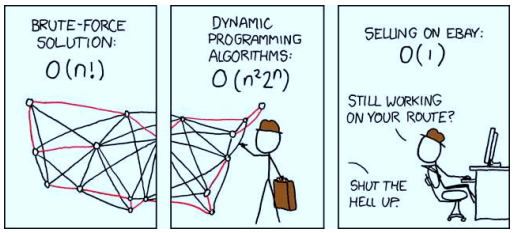


Notes by CSSE, Comics by xkcd.com

Computer Science and Software Engineering, 2011

CITS3210 Algorithms

Introduction



Notes by CSSE, Comics by xkcd.com

Overview

1. Introduction
   (a) What are Algorithms?
   (b) Design of Algorithms.
   (c) Types of Algorithms.
2. Complexity
   (a) Growth rates.
   (b) Asymptotic analysis, O and Θ.
   (c) Average case analysis.
   (d) Recurrence relations.
3. Sorting
   (a) Insertion Sort.
   (b) Merge Sort.
   (c) QuickSort.

What you should already know?

This unit will require the following basic knowledge:

1. Java Programming: *classes, control structures, recursion, testing, etc*
2. Data Structures: *stacks, queues, lists, trees, etc.*
3. Complexity: *definition of "big O", Θ notation, amortized analysis etc.*
4. Some maths: *proof methods, such as proof by induction, some understanding of continuous functions*

What will we be studying?

We will study a collection of algorithms, examining their design, analysis and sometimes even implementation. The topics we will cover will be taken from the following list:

1. Specifying and implementing algorithms.
2. Basic complexity analysis.
3. Sorting Algorithms.
4. Graph algorithms.
5. Network flow algorithms.
6. Computational Geometry.
7. String algorithms.
8. Greedy/Dynamic algorithms.
9. Optimization Algorithms.

What are the outcomes of this unit?

At the end of the unit you will:

1. be able to identify and abstract computational problems.
2. know important algorithmic techniques and a range of useful algorithms.
3. be able to implement algorithms as a solution to any solvable problem.
4. be able to analyse the complexity and correctness of algorithms.
5. be able to design correct and efficient algorithms.

The course will proceed by covering a number of algorithms; as they are covered, the general algorithmic technique involved will be highlighted, and the role of appropriate data structures, and efficient implementation considered.

What are algorithms?

An *algorithm* is a well-defined finite set of rules that specifies a sequential series of elementary operations to be applied to some data called the *input*, producing after a finite amount of time some data called the *output*.

An algorithm solves some computational problem.

Algorithms (along with data structures) are the fundamental "building blocks" from which programs are constructed. Only by fully understanding them is it possible to write very effective programs.

Design and Analysis

An algorithmic solution to a computational problem will usually involve *designing* an algorithm, and then *analysing* its performance.

Design A good algorithm designer must have a thorough background knowledge of algorithmic techniques, but especially substantial creativity and imagination. Often the most obvious way of doing something is inefficient, and a better solution will require thinking "out of the box". In this respect, algorithm design is as much an art as a science.

Analysis A good algorithm analyst must be able to carefully estimate or calculate the resources (time, space or other) that the algorithm will use when running. This requires logic, care and often some mathematical ability.

The aim of this course is to give you sufficient background to understand and appreciate the issues involved in the design and analysis of algorithms.

Design and Analysis

In designing and analysing an algorithm we should consider the following questions:

1. What is the problem we have to solve?
2. Does a solution exist?
3. Can we find a solution (algorithm), and is there more than one solution?
4. Is the algorithm correct?
5. How efficient is the algorithm?

The importance of design

By far the most important thing in a program is the *design of the algorithm*. It is far more significant than the language the program is written in, or the clock speed of the computer.

To demonstrate this, we consider the problem of computing the *Fibonacci numbers*.

The Fibonacci sequence is the sequence of integers starting

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

which is formally defined by

$$F_1 = F_2 = 1 \text{ and } F_n = F_{n-1} + F_{n-2}.$$

Let us devise an algorithm to compute F_n.

The naive solution

The naive solution is to simply write a *recursive* method that directly models the problem.

```
static int fib(int n) {

   return (n<3 ? 1 : fib(n-1) + fib(n-2));

}
```

Is this a *good* algorithm/program in terms of resource usage?

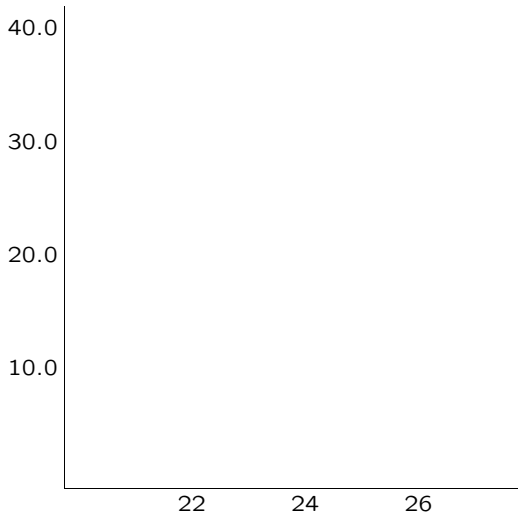
Timing it on a (2005) iMac gives the following results (the time is in seconds and is for a loop calculating F_n 10000 times).

Value	Time	Value	Time
F_{20}	1.65	F_{24}	9.946
F_{21}	2.51	F_{25}	15.95
F_{22}	3.94	F_{26}	25.68
F_{23}	6.29	F_{27}	41.40

How long will it take to compute F_{30}, F_{40} or F_{50}?

Experimental results

Make a plot of the times taken.



Theoretical results

Each method call to `fib()` does roughly the same amount of work (just two comparisons and one addition), so we will have a very rough estimate of the time taken if we count how many method calls are made.

Exercise: *Show the number of method calls made to* `fib()` *is* $2F_n - 1$.

Re-design the algorithm

We can easily re-design the algorithm as an *iterative* algorithm.

```
static int fib(int n) {

  int f_2;        /* F(i+2)   */
  int f_1 = 1;  /* F(i+1) */
  int f_0 = 1;  /* F(i) */

  for (int i = 1; i < n; i++) {
    /* F(i+2) = F(i+1) + F(i) */
    f_2 = f_1 + f_0;

    /* F(i) = F(i+1); F(i+1) = F(i+2) */
    f_0 = f_1;
    f_1 = f_2;
  }

  return f_0;
}
```


An Iterative Algorithm

An iterative algorithm gives the following times:

Value	Time	Value	Time
F_{20}	0.23	F_{10^3}	0.25
F_{21}	0.23	F_{10^4}	0.48
F_{22}	0.23	F_{10^5}	2.20
F_{23}	0.23	F_{10^6}	20.26

Another solution?

The Fibonacci sequence is specified by the *homogeneous recurrence relation*:

$$F(n) = \begin{cases} 1 & \text{if } n = 1,\ 2; \\ F(n-1) + F(n-2) & \text{otherwise.} \end{cases}$$

In general we can define a closed form for these recurrence equations:

$$F(n) = A\alpha^n + B\beta^n$$

where α, β are the roots of

$$x^2 - x - 1 = 0.$$

- You need to be able to derive a recurrence relation that describes an algorithms complexity.

- You need to be able to recognize that linear recurrence relations specify exponential functions.

See CLRS, Chapter 4.

Recurrence Relations

Recurrence relations can be a useful way to specify the complexity of recursive functions.

For example the *linear homogeneous recurrence relation*:

$$F(n) = \begin{cases} 1 & \text{if } n = 1,\ 2; \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

specifies the sequence $1, 1, 2, 3, 5, 8, 13, ...$

In general a linear homogeneous recurrence relation is given as:

$$\begin{aligned} F(1) &= c_1 \\ F(2) &= c_2 \\ &... \\ F(k) &= c_k \\ F(n) &= a_1 F(n-1) + ... + a_k F(n-k) \end{aligned}$$

For example

$$F(n) = \begin{cases} 1 & \text{if } n = 1,\ 2; \\ 2F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

specifies the sequence $1, 1, 3, 7, 17, 41, ...$

Solving the recurrence

All linear homogeneous recurrence relations specify exponential functions. We can find a closed form for the recurrence relation as follows:

Suppose that $F(n) = r^n$.
Then $r^n = a_1 r^{n-1} + ... + a_k r^(n-k)$. We divide both sides of the equation by r^{n-k}.
Then $r^k = a_1 r^{k-1} + ... + a_k$.
To find r we can solve the polynomial equation: $r^k - a_1 r^{k-1} - .... - a_K = 0$.

There are k solutions, $r_1, ..., r_k$ to this equation, and each satisfies the recurrence:

$$F(n) = a_1 F(n-1) + a_2 F(n-1) + ... + a_k F(n-k).$$

We also have to satisfy the rest of the recurrence relation, $F(1) = c_1$ etc. To do this we can use a linear combination of the solutions, r_k^n. That is, we must find $\alpha_1, ..., \alpha_k$ such that

$$F(n) = \alpha_1 r_1^n + ... + \alpha_k r_k^n$$

This can be done by solving linear equations.

Solving the recurrence

The roots of the polynomial are

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{1 \pm \sqrt{5}}{2}$$

and so the solution is

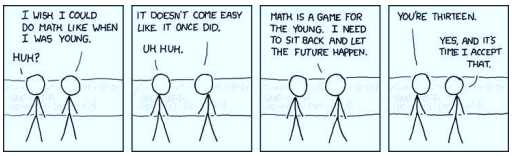
$$U(n) = A\left(\frac{1+\sqrt{5}}{2}\right)^n + B\left(\frac{1-\sqrt{5}}{2}\right)^n$$

If we substitute $n = 1$ and $n = 2$ into the equation we get

$$A = \frac{1}{\sqrt{5}} \qquad B = \frac{-1}{\sqrt{5}}$$

Thus

$$F(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$



What is an algorithm?

We need to be more precise now what we mean by a *problem*, *a solution* and how we shall judge whether or not an algorithm is a *good* solution to the problem.

A *computational problem* consists of a general description of a question to be answered, usually involving some free variables or parameters.

An *instance* of a computational problem is a specific question obtained by assigning values to the parameters of the problem.

An algorithm *solves* a computational problem if when presented with *any* instance of the problem as input, it produces the answer to the question as its output.

A computational problem: Sorting

Instance: A sequence L of comparable objects.

Question: What is the sequence obtained when the elements of L are placed in ascending order?

An instance of **Sorting** is simply a specific list of comparable items, such as

$$L = [25, 15, 11, 30, 101, 16, 21, 2]$$

or

$$L = [\text{“dog”}, \text{“cat”}, \text{“aardvark”}, \text{“possum”}].$$

A computational problem: Travelling Salesman

Instance: A set of "cities" X together with a "distance" $d(x, y)$ between any pair x, $y \in X$.

Question: What is the shortest circular route that starts and ends at a given city and visits all the cities?

An instance of Travelling Salesman is a list of cities, together with the distances between the cities, such as

$$X = \{A, B, C, D, E, F\}$$

$d =$

	A	B	C	D	E	F
A	0	2	4	∞	1	3
B	2	0	6	2	1	4
C	4	6	0	1	2	1
D	∞	2	1	0	6	1
E	1	1	2	6	0	3
F	3	4	1	1	3	0

An algorithm for Sorting

One simple algorithm for **Sorting** is called *Insertion Sort*. The basic principle is that it takes a series of steps such that after the i-th step, the first i objects in the array are sorted. Then the $(i+1)$-th step *inserts* the $(i+1)$-th element into the correct position, so that now the first $i+1$ elements are sorted.

procedure INSERTION-SORT(A)
  for $j \leftarrow 2$ **to** *length*[A]
    do *key* $\leftarrow A[j]$
      $\triangleright$ Insert $A[j]$ into the sorted sequence
      $\triangleright$ $A[1 \ldots j-1]$
      $i = j - 1$
      while $i > 0$ and $A[i] >$ *key*
        do $A[i+1] \leftarrow A[i]$
          $i = i - 1$
      $A[i+1] \leftarrow$ *key*

Pseudo-code

Pseudo-code provides a way of expressing algorithms in a way that is independent of any programming language. It abstracts away other program details such as the type system and declaring variables and arrays. Some points to note are:

- The statement blocks are determined by indentation, rather than { and } delimiters as in Java.
- Control statements, such as **if... then...else** and **while** have similar interpretations to Java.
- The character $\triangleright$ is used to indicate a comment line.

Pseudo-code (contd)

- A statement $v \leftarrow e$ implies that expression e should be evaluated and the resulting value assigned to variable v. Or, in the case of $v_1 \leftarrow v_2 \leftarrow e$, to variables v_1 and v_2.
- All variables should be treated as *local* to their procedures.
- Arrays indexation is denoted by $A[i]$ and arrays are assumed to be indexed from 1 to N (*rather* than 0 to $N-1$, the approach followed by Java).

See CLRS (page 19-20) for more details.

But to return to the insertion sort: *What do we actually mean by a good algorithm?*

Evaluating Algorithms

There are many considerations involved in this question.

- Correctness
  1. Theoretical correctness
  2. Numerical stability
- Efficiency
  1. Complexity
  2. Speed

Correctness of insertion sort

Insertion sort can be shown to be correct by a *proof by induction*.

```
procedure INSERTION-SORT(A)
    for j ← 2 to length[A]
        do key ← A[j]
           ▷ Insert A[j] into the sorted sequence
           ▷ A[1 . . . j − 1]
           i = j − 1
           while i > 0 and A[i] > key
               do A[i + 1] ← A[i]
                  i = i − 1
           A[i + 1] ← key
```

We do the induction over the loop variable j.
The base case of the induction is:
"the first element is sorted",
and the inductive step is:
"given the first j elements are sorted after the j^{th} iteration, the first $j+1$ elements will be sorted after the $j+1^{th}$ iteration.

Proof by Induction

To show insertion sort is correct, let $p(n)$ be the statement "after the n^{th} iteration, the first $n+1$ elements of the array are sorted"

To show $p(0)$ we simply note that a single element is always sorted.

Given $p(i)$ is true for all $i < n$, we must show that $p(n)$ is true:
After the $(n-1)^{th}$ iteration the first n elements of the array are sorted.
The n^{th} iteration takes the $(n+1)^{th}$ element and inserts it after the last element that a) comes before it, and b) is less than it.
Therefore after the n^{th} iteration, the first $n+1$ elements of the array are sorted.

Aside: Proof by Contradiction

Another proof technique you may need is *proof by contradiction*.

Here, if you want to show some property p is true, you assume p is not true, and show this assumption leads to a contradiction (something we know is not true, like $i < i$).

For example, two sorted arrays of integers, L, containing exactly the same elements, must be identical.
Proof by contradiction: Suppose $M \neq N$ are two distinct, sorted arrays containing the same elements. Let i be the least number such that $M[i] \neq N[i]$. Suppose $a = M[i] < N[i]$. Since M and N contain the same elements, and $M[j] = N[j]$ for all $j < i$, we must have $a = N[k]$ for some $k > i$. But then $N[k] < N[i]$ so N is not sorted: contradiction.

Complexity of insertion sort

For simple programs, we can directly calculate the number of basic operations that will be performed:

```
procedure INSERTION-SORT(A)
1  for j ← 2 to length[A]
2      do key ← A[j]
          ▷ Insert A[j] into the sorted sequence A[1 . . . j − 1]
3         i = j − 1
4         while i > 0 and A[i] > key
5             do A[i + 1] ← A[i]
6                i = i − 1
7         A[i + 1] ← key
```

The block containing lines 2-7 will be executed $length[A] - 1$ times, and contains 3 basic operations
In the worst case the block containing lines 5-7 will be executed $j - 1$ times, and contains 2 basic operations.
In the worst case the algorithm will take

$$(N-1).3 + 2(2 + 3 + ... + N) = N^2 + 4N - 5$$

where $length[A] = N$.

Correctness

An algorithm is *correct* if, when it terminates, the output is a correct answer to the given question.

Incorrect algorithms or implementations abound, and there are many costly and embarrassing examples:

- Intel's Pentium division bug—a scientist discovered that the original Pentium chip gave incorrect results on certain divisions. Intel only reluctantly replaced the chips.
- USS Yorktown—after switching their systems to Windows NT, a "division by zero" error crashed every computer on the ship, causing a multi-million dollar warship to drift helplessly for several hours.
- Others...?

Theoretical correctness

It is usually possible to give a mathematical proof that an algorithm in the abstract is correct, but proving that an implementation (that is, actual code) is correct is much more difficult.

This is the province of an area known as *software verification*, which attempts to provide logical tools that allow specification of programs and reasoning about programs to be done in a rigorous fashion.

The alternative to formal software verification is *testing*; although thorough testing is vital for any program, one can never be certain that everything possible has been covered.

Even with vigorous testing, there is always the possibility of hardware error—mission critical software must take this into account.

Types of Algorithm

For all solvable problems, you should (already!) be able to produce a correct algorithm. The *brute force* approach simply requires you to

1. enumerate all possible solutions to the problem, and
2. iterate through them until you find one that works.

This is rarely practical. Other strategies to consider are:

- Divide and conquer - Divide the problem into smaller problems to solve.
- Dynamic programming.
- Greedy algorithms.
- Tree traversals/State space search

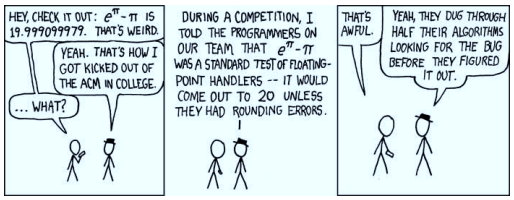
Numerical Stability

You can be fairly certain of exact results from a computer program provided all arithmetic is done with the *integers* $\mathbf{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$ and you guard carefully about any overflow.

However the situation is entirely different when the problem involves real number, because there is necessarily some *round-off* error when real numbers are stored in a computer. A *floating point representation* of a number in base β with precision p is a representation of the form.

$$d.ddddd \times \beta^e$$

where $d.ddddd$ has exactly p digits.



Accumulation of errors

Performing repeated calculations will take the small truncation errors and cause them to accumulate. The resulting error is known as *roundoff error*. If we are careful or lucky, the roundoff error will tend to behave randomly, both positive and negative, and the growth of error will be slow.

Certain calculations however, vastly increase roundoff error and can cause errors to grow catastrophically to the point where they completely swamp the real result.

Two particular operations that can cause numerical instability are

- Subtraction of nearly equal quantities
- Division by numbers that are nearly zero

It is important to be aware of the possibility for roundoff error and to alter your algorithm appropriately.

Efficiency

An algorithm is efficient if it uses as few *resources* as possible. Typically the resources which we are interested in are

- Time, and
- Space (memory)

Other resources are important in practical terms, but are outside the scope of the design and analysis of algorithms.

In many situations there is a trade-off between time and space, in that an algorithm can be made faster if it uses more space or smaller if it takes longer.

Although a thorough analysis of an algorithm should consider both time and space, time is considered more important, and this course will focus on time complexity.

Measuring time

How should we *measure* the time taken by an algorithm?

We can do it experimentally by measuring the number of seconds it takes for a program to run — this is often called *benchmarking* and is often seen in popular magazines. This can be useful, but depends on many factors:

- The machine on which it is running.
- The language in which it is written.
- The skill of the programmer.
- The *instance* on which the program is being run, both in terms of size and which particular instance it is.

So it is not an independent measure of the *algorithm*, but rather a measure of the implementation, the machine and the instance.

Complexity

The complexity of an algorithm is a "device-independent" measure of how much time it consumes. Rather than expressing the time consumed in seconds, we attempt to count how many "elementary operations" the algorithm performs when presented with instances of different sizes.

The result is expressed as a *function*, giving the number of operations in terms of the size of the instance. This measure is not as precise as a benchmark, but much more useful for answering the kind of questions that commonly arise:

- I want to solve a problem twice as big. How long will that take me?
- We can afford to buy a machine twice as fast? What size of problem can we solve in the same time?

The answers to questions like this depend on the *complexity* of the algorithm.

Example

Suppose you run a small business and have a program to keep track of your 1024 customers. The list of customers is changing frequently and you often need to sort it. Your two programmers Alice and Bob both come up with algorithms.

Alice presents an algorithm that will sort n names using $256n \lg n$ comparisons and Bob presents an algorithm that uses n^2 comparisons. (Note: $\lg n \equiv \log_2 n$)

Your current computer system takes 10^{-3} seconds to make one comparison, and so when your boss benchmarks the algorithms he concludes that clearly Bob's algorithm is better.

Size	Alice	Bob
1024	2621	1049

But is he right?

Expansion

Alice however points out that the business is expanding and that using Bob's algorithm could be a mistake. As the business expands, her algorithm becomes more competitive, and soon overtakes Bob's.

Size	Alice	Bob
1024	2621	1049
2048	5767	4194
4096	12583	16777
8192	27263	67109

So Alice's algorithm is much better placed for expansion.

A benchmark only tells you about the situation *today*, whereas a software developer should be thinking about the situation both today and tomorrow!

Hardware improvement

A time-critical application requires you to sort as many items as possible in an hour. How many can you sort?

An hour has 3600 seconds, so we can make 3600000 comparisons. Thus if Alice's algorithm can sort n_A items, and Bob's n_B items, then

$$3600000 = 256 n_A \lg n_A = n_B^2,$$

which has the solution

$$n_A = 1352 \qquad n_B = 1897.$$

But suppose that we replace the machines with ones that are four times as fast. Now each comparison takes $\frac{1}{4} \times 10^{-3}$ seconds so we can make 14400000 comparisons in the same time. Solving

$$14400000 = 256 n_A \lg n_A = n_B^2,$$

yields

$$n_A = 4620 \qquad n_B = 3794.$$

Notice that Alice's algorithm gains much more from the faster machines than Bob's.

Different instances of the same size

So far we have assumed that the algorithm takes the same amount of time on every instance of the same size. But this is almost never true, and so we must decide whether to do *best case*, *worst case* or *average case* analysis.

In best case analysis we consider the time taken by the algorithm to be the time it takes on the *best* input of size n.

In worst case analysis we consider the time taken by the algorithm to be the time it takes on the *worst* input of size n.

In average case analysis we consider the time taken by the algorithm to be the *average* of the times taken on inputs of size n.

Best case analysis has only a limited role, so normally the choice is between a worst case analysis or attempting to do an average case analysis.

Worst case analysis

Most often, algorithms are analysed by their worst case running times — the reasons for this are:

- This is the only "safe" analysis that provides a guaranteed upper bound on the time taken by the algorithm.
- Average case analysis requires making some assumptions about the probability distribution of the inputs.
- Average case analysis is much harder to do.

Big-O notation

Our analysis of insertion sort showed that it took about $n^2 + 4n - 5$ operations, but this is more precise than necessary. As previously discussed, the most important thing about the time taken by an algorithm is its *rate of growth*. The fact that it is $n^2/2$ rather than $2n^2$ or $n^2/10$ is considered irrelevant. This motivates the traditional definition of Big-O notation.

Definition A function $f(n)$ is said to be $O(g(n))$ if there are constants c and N such that

$$f(n) \leq cg(n) \quad \forall n \geq N.$$

Thus by taking $g(n) = n^2$, $c = 1$ and $N = 1$ we conclude that the running time of *Insertion Sort* is $O(n^2)$, and moreover this is the best bound that we can find. (In other words *Insertion Sort* is *not* $O(n)$ or $O(n \lg n)$.)

Big-Theta notation

Big-O notation defines an asymptotic *upper* bound for a function $f(n)$. But sometimes we can define a lower bound as well, allowing a tighter constraint to be defined. In this case we use an alternative notation.

Definition A function $f(n)$ is said to be $\Theta(g(n))$ if there are constants c_1, c_2 and N such that

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall n \geq N.$$

If we say that $f(n) = \Theta(n^2)$ then we are implying that $f(n)$ is *approximately proportional* to n^2 for large values of n.

See CLRS (section 3) for a more detailed description of the O and Θ notation.

Why is big-O notation useful?

In one sense, big-O notation hides or loses a lot of useful information. For example, the functions

$f(n) = n^2$ / 1000
$g(n) = 100\, n^2$
$h(n) = 10^{10}\, n^2$

are all $O(n^2)$ despite being quite different.

However in another sense, the notation contains the *essential* information, in that it completely describes the *asymptotic rate of growth* of the function. In particular it contains enough information to give answers to the questions:

- Which algorithm will ultimately be faster as the input size increases?
- If I buy a machine 10 times as fast, what size problems can I solve in the same time?

An asymptotically better sorting algorithm

```
procedure MERGE-SORT(A, p, r)
    if p < r
        then q ← ⌊(p + r)/2⌋
             MERGE-SORT(A, p, q)
             MERGE-SORT(A, q + 1, r)
             MERGE(A, p, q, r)
```

```
procedure MERGE(A, p, q, r)
    n1 ← q − p + 1; n2 ← r − q
    allocate arrays L[1 . . . n1 + 1] and R[1 . . . n2 + 1]
    for i ← 1 to n1
        do L[i] ← A[p + i − 1]
    for j ← 1 to n2
        do R[j] ← A[q + j]
    L[n1 + 1] ← ∞; R[n2 + 1] ← ∞
    i ← 1; j ← 1
    for k ← p to r
        do if L[i] ≤ R[j]
            then A[k] ← L[i]
                 i ← i + 1
            else A[k] ← R[j]
                 j ← j + 1
```


Merge-sort complexity

The complexity of Merge Sort can be shown to be $\Theta(n \lg n)$.

The Master Theorem

Merge Sort's complexity can be described by the recurrence relation:

$$F(n) = 2F(n/2) + n, \quad \text{where } F(1) = 1.$$

As this variety of recurrence relation appears frequently in divide and conquer algorithms it is useful to have an method to find the asymptotic complexity of these functions.

The Master Theorem: Let $f(n)$ be a function described by the recurrence::

$$f(n) = af(n/b) + cn^d.$$

where $a, b \geq 1$, $d \geq 0$ and $c > 0$ are constants. Then

$$f(n) \text{ is } \left\{ \begin{array}{rl} O(n^d) & \text{if } a < b^d \\ O(n^d lgn) & \text{if } a = b^d \\ O(n^{log_b a}) & \text{if } a > b^d \end{array} \right\}$$

See CLRS, 4.3.

Average case analysis

The major problem with average case analysis is that we must make an assumption about the probability distribution of the inputs. For a problem like **Sorting** there is at least a *theoretically* reasonable choice—assume that every permutation of length n has an equal chance of occurring (already we are assuming that the list has no duplicates).

For example, we can consider each of the 24 permutations when sorting four inputs with insertion sort:

Comparisons	Inputs
3	1234, 2134
4	1243, 1324, 2143, 2314, 3124, 3214
5	1342, 1423, 2341, 2413, 3142, 3241, 4123, 4213
6	1432, 2431, 3412, 3421, 4132, 4231, 4312, 4321

So the weighted average of comparisons is

$$\frac{(3\times 2)+(4\times 6)+(5\times 8)+(6\times 8)}{24} = 4.916$$

(recall that the best case for four inputs is 3, whereas the worst case is 6).

Inversions

Definition An *inversion* in a permutation σ is an ordered pair (i, j) such that

$$i < j \text{ and } \sigma_i > \sigma_j.$$

For example, the permutation $\sigma = 1342$ has two inversions, while $\sigma = 2431$ has four.

It is straightforward to see that the number of comparisons that a permutation requires to be sorted is equal to the number of inversions in it (check this!) plus a constant, c.
(For sorting four inputs, $c = 3$)

So the average number of comparisons required is equal to the average number of inversions in all the permutations of length n.

Theorem The average number of inversions among all the permutations of length n is $n(n-1)/4$.

Thus *Insertion Sort* takes $O(n^2)$ time on average.

An asymptotically worse algorithm

Quicksort is $\Theta(n^2)$, but it's average complexity is *better* than Merge-sort! (CLRS Chapter 7)

procedure QUICKSORT(A, p, r)
  if $p < r$
    then $q \leftarrow$ PARTITION(A, p, r)
      QUICKSORT$(A, p, q-1)$
      QUICKSORT$(A, q+1, r)$

procedure PARTITION(A, p, r)
  $x \leftarrow A[r]$
  $i \leftarrow p-1$
  for $j \leftarrow p$ **to** $r-1$
    do if $A[j] \le x$
      then $i \leftarrow i+1$
        exchange $A[i] \leftrightarrow A[j]$
  exchange $A[i+1] \leftrightarrow A[r]$
  return $i+1$

Input size

The complexity of an algorithm is a measure of how long it takes as a function of the *size of the input*. For **Sorting** we took the number of items n, as a measure of the size of the input.

This is only true provided that the actual size of the *items* does not grow as their number increases. As long as they are all some constant size K, then the input size is Kn. The actual value of the constant does not matter, as we are only expressing the complexity in big-O notation, which suppresses all constants.

But what is an appropriate input parameter for **Travelling Salesman**? If the instance has n cities, then the input itself has size Kn^2—this is because we need to specify the distance between each pair of cities.

Therefore you must be careful about what parameter most accurately reflects the size of the input.

Travelling Salesman

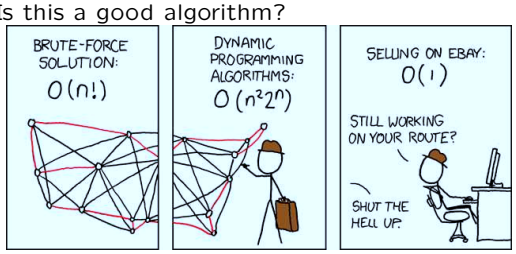
Naive solution: Consider *every* permutation of the n cities, and compute the length of the resulting tour, saving the shortest length.

How long will this take? We count two main operations

- Generating the $n!$ permutations.
- Evaluating each tour at cost of $O(n)$.

If we assume that by clever programming, we can compute each permutation in constant time, then the total time is $O(n.n!)$.

Is this a good algorithm?



Good Algorithms

Theoretical computer scientists use a very broad brush to distinguish between *good* and *bad* algorithms.

An algorithm is *good* if it runs in time that is a *polynomial function* of the size of the input, otherwise it is bad.

Good	$O(1)$	constant time
	$O(n)$	linear
	$O(n \lg n)$	loglinear
	$O(n^2)$	quadratic
	⋮	⋮
	$O(n^k)$	"polynomial"
	⋮	⋮
	⋮	⋮
Bad	2^n	exponential
	c^n	exponential
	$n!$	factorial

A problem for which no polynomial time algorithm can be found is called *intractable*. As far as we know, **Travelling Salesman** is an intractable problem, though no-one has proved this.

Summary

1. An algorithm is a well defined set of rules for solving a computational problem.
2. A well designed algorithm should be efficient for problems of all sizes.
3. Algorithms are generally evaluated with respect to correctness, stability, and efficiency (for space and speed).
4. Theoretical correctness can be established using mathematical proof.
5. Numerical stability is required for algorithms to give accurate answers.

Summary (cont.)

6. Different kinds of algorithms have been defined, including brute-force algorithms, divide and conquer algorithms, greedy algorithms, dynamic algorithms, and tree traversals.
7. The efficiency of an algorithm is a measure of complexity that indicates how long an algorithm will take.
8. Big "O" is a measure of complexity that is the asymptotic worst case upper bound.
9. Θ (big *theta*) is a measure of complexity that is the asymptotic worst case tight bound.
10. Average case analysis attempts to measure how fast an algorithm is for an average (typical) input.

Summary (cont.)

11. *Insertion sort* is a sorting algorithm that runs in time $O(n^2)$.

12. *Merge sort* is a sorting algorithm that runs in time $O(n\lg n)$.

13. *Quicksort* is a sorting algorithm that runs in time $O(n^2)$ but is faster than Merge sort in the average case.

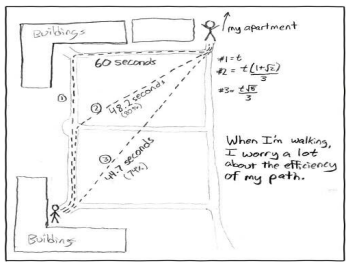
14. Polynomial algorithms (e.g. $O(n)$, $O(n\lg n)$, $O(n^k)$) are regarded as feasible.

15. Exponential algorithms (e.g. $O(2^n)$, $O(n)$) are regarded as infeasible.

Computer Science and Software Engineering, 2011

CITS3210 Algorithms

Graph Algorithms



Notes by CSSE, Comics by xkcd.com

Overview

1. Introduction
   - Terminology and definitions
   - Graph representations
2. Tree Search
   - Breadth first search
   - Depth first search
   - Topological sort
3. Minimum Spanning Trees
   - Kruskal's algorithm
   - Prim's algorithm
   - Implementations
   - Priority first search
4. Shortest Path Algorithms
   - Dijkstra's algorithm
   - Bellman-Ford algorithm
   - Dynamic Programming

What is a graph?

Definition A *graph* G consists of a set $V(G)$ called *vertices* together with a collection $E(G)$ of pairs of vertices. Each pair $\{x, y\} \in E(G)$ is called an *edge* of G.

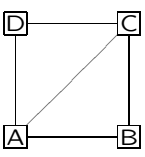
Example If

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

and

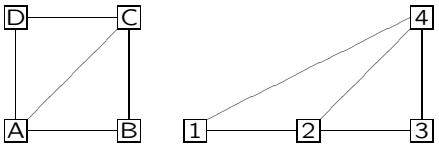
$$E(G) = \{\{A, B\}, \{C, D\}, \{A, D\}, \{B, C\}, \{A, C\}\}$$

then G is a graph with 4 vertices and 5 edges.



Isomorphisms

Consider the following two graphs:



Apart from the "names" of the vertices and the geometric positions it is clear that these two graphs are basically the same — in this situation we say that they are *isomorphic*.

Definition Two graphs G_1 and G_2 are *isomorphic* if there is a one-one mapping $\phi : V(G_1) \to V(G_2)$ such that $\{\phi(x), \phi(y)\} \in E(G_2)$ if and only if $\{x, y\} \in E(G_1)$.

In this case the isomorphism is given by the mapping

$$\phi(A) = 2 \quad \phi(B) = 3 \quad \phi(C) = 4 \quad \phi(D) = 1$$

What are graphs used for?

Graphs are used to model a wide range of commonly occurring situations, enabling questions about a particular problem to be reduced to certain well-studied "standard" graph theory questions.

For examples consider the three graphs G_1, G_2 and G_3 defined as follows:

$V(G_1)$ = all the telephone exchanges in Australia, and $\{x, y\} \in E(G_1)$ if exchanges x and y are physically connected by fibre-optic cable.

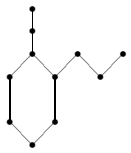
$V(G_2)$ = all the airstrips in the world, and $\{x, y\} \in E(G_2)$ if there is a direct passenger flight from x to y.

$V(G_3)$ = all the people who have ever published a paper in a refereed journal in the world, and $\{x, y\} \in E(G_3)$ if x and y have been joint authors on a paper.

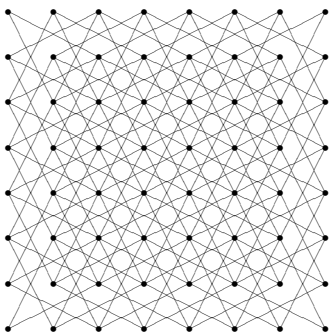
More examples

In computing: A graph can be used to represent processors that are connected via a communication link in a parallel computer system.

In chemistry: The vertices of a graph can be used to represent the carbon atoms in a molecule, and an edge between two vertices represents the bond between the corresponding atoms.

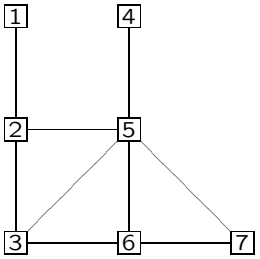


In games: The vertices can be the 64 squares on a chessboard, and the edge that joins two squares can be used to denote the valid movement of a knight from one square to the other.



An example graph

Consider the following graph G_4.



The graph G_4 has 7 vertices and 9 edges.

Basic properties of graphs

Let us consider some of the basic terminology of graphs:

Adjacency If $\{x, y\} \in E(G)$, we say that x and y are *adjacent* to each other, and sometimes write $x \sim y$. The number of vertices adjacent to v is called the *degree* or *valency* of v. The sum of the degrees of the vertices of a graph is even.

Paths A *path* of *length* n in a graph is a sequence of vertices $v_1 \sim v_2 \sim \cdots \sim v_{n+1}$ such that $(v_i, v_{i+1}) \in E(G)$ and vertices $\{v_1, v_2, \ldots, v_{n+1}\}$ are distinct.

Cycles A *cycle* of length n is a sequence of vertices $v_1 \sim v_2 \sim \cdots v_n \sim v_{n+1}$ such that $v_1 = v_{n+1}$, $(v_i, v_{i+1}) \in E(G)$ and therefore only vertices $\{v_1, v_2, \ldots, v_n\}$ are distinct.

Distance The *distance* between two vertices x and y in a graph is the length of the shortest path between them.

Subgraphs

If G is a graph, then a subgraph H is a graph such that

$$V(H) \subseteq V(G)$$

and

$$E(H) \subseteq E(G)$$

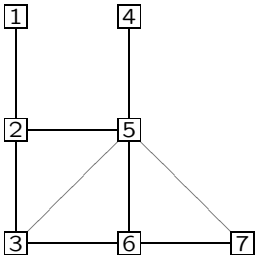
A *spanning* subgraph H has the property that $V(H) = V(G)$ — in other words H has been obtained from G only by removing edges.

An *induced* subgraph H must contain every edge of G whose endpoints lie in $V(H)$ — in other words H has been obtained from G by removing vertices and their adjoining edges.

Counting Exercises

In the graph G_4:

- How many paths are there from 1 to 7?
- How many cycles are there?
- How many spanning subgraphs are there?
- How many induced subgraphs are there?



Connectivity, forests and trees

Connected A graph G is *connected* if there is a path between any two vertices. If the graph is not connected then its connected components are the maximal induced subgraphs that are connected.

Forests A forest is a graph that has no cycles.

Trees A tree is a forest with only one connected component. It is easy to see that a tree with n vertices must have exactly $n - 1$ edges.

The vertices of degree 1 in a tree are called the *leaves* of the tree.

Directed and weighted graphs

There are two important extensions to the basic definition of a graph.

Directed graphs In a directed graph, an edge is an ordered pair of vertices, and hence has a direction. In directed graphs, edges are often called *arcs*.

Directed Tree Each vertex has at most one directed edge leading into it, and there is one vertex (the root) which has a path to every other vertex.

Weighted graphs In a weighted graph, each of the edges is assigned a weight (usually a non-negative integer). More formally we say that a weighted graph is a graph G together with a weight function $w : E(G) \to \mathbf{R}$ (then $w(e)$ represents the weight of the edge e).

Distance in weighted graphs

When talking about weighted graphs, we need to extend the concept of distance.

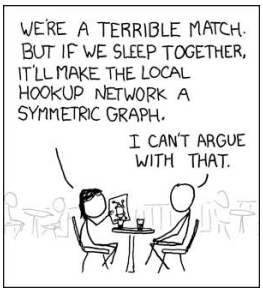
Definition In a weighted graph X a path

$$x = x_0 \sim x_1 \sim \cdots \sim x_n = y$$

has *weight*

$$\sum_{i=0}^{i=n-1} w(x_i, x_{i+1}).$$

The *shortest path* between two vertices x and y is the path of minimum weight.

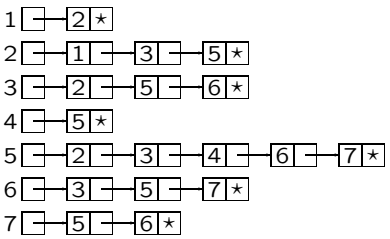


Representation of graphs

There are two main ways to represent a graph — adjacency lists or an adjacency matrix.

Adjacency lists The graph G is represented by an array of $|V(G)|$ linked lists, with each list containing the neighbours of a vertex.

Therefore we would represent G_4 as follows:

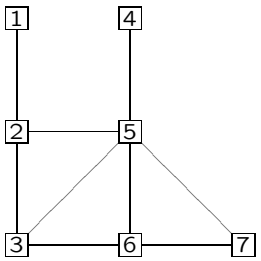


This representation requires two list elements for each edge and therefore the space required is $\Theta(|V(G)| + |E(G)|)$.

Note: In general to avoid writing $|V(G)|$ and $|E(G)|$ we shall simply put $V = |V(G)|$ and $E = |E(G)|$.

For comparison...

...the graph G_4.



Adjacency matrix

The *adjacency matrix* of a graph G is a $V \times V$ matrix A where the rows and columns are indexed by the vertices and such that $A_{ij} = 1$ if and only if vertex i is adjacent to vertex j.

For graph G_4 we have the following

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

The adjacency matrix representation uses $\Theta(V^2)$ space.

For a *sparse* graph E is much less than V^2, and hence we would normally prefer the adjacency list representation.

For a *dense* graph E is close to V^2 and the adjacency matrix representation is preferred.

More on the two representations

For small graphs or those without weighted edges it is often better to use the adjacency matrix representation anyway.

It is also easy and more intuitive to define adjacency matrix representations for directed and weighted graphs.

However your final choice of representation depends precisely what questions you will be asking. Consider how you would answer the following questions in both representations (in particular, how much time it would take).

Is vertex v adjacent to vertex w in an undirected graph?

What is the out-degree of a vertex v in a directed graph?

What is the in-degree of a vertex v in a directed graph?

Recursive Representation

A third representation to consider is a *recursive representation.* In this representation you may not have access to a list of all vertices in the graph. Instead you have access to a single vertex, and from that vertex you can deduce the adjacent vertices.

The following java class is an example of such a representation:

```
abstract class Vertex{

    int data;

    Vertex[] getAdjacentVertices(){}

}
```

This type of data structure is likely to arise if you consider, for example, graphs of all states in a chess game, or communication networks.

Breadth-first search

Searching through a graph is one of the most fundamental of all algorithmic tasks, and therefore we shall examine several techniques for doing so.

Breadth-first search is a simple but extremely important technique for searching a graph. This search technique starts from a given vertex v and constructs a spanning tree for G, called the *breadth-first tree*. It uses a (first-in, first-out) *queue* as its main data structure.

Following CLRS (section 22.2), as the search progresses, we will divide the vertices of the graph into three categories, *black* vertices which are the vertices that have been fully examined and incorporated into the tree, *grey* vertices which are the vertices that have been seen (because they are adjacent to a tree vertex) and placed on the queue, and *white* vertices, which have not yet been examined.

Queues

Recall that a *queue* is a first-in-first-out buffer.

Items are *pushed* (or enqueued) onto the end of the queue, and items can be *popped* (or dequeued) from the front of the queue.

A Queue is commonly implemented using either a block representation, or a linked representation.

We will assume that the push and pop operations can be performed in constant time. You may also assume that we can examine the first element of the queue, and decide if the queue is empty, all in constant time (i.e. $\Theta(1)$).

Breadth-first search initialization

The final breadth-first tree will be stored as an array called π where $\pi[x]$ is the immediate parent of x in the spanning tree. Of course, as v is the root of this tree, $\pi[v]$ will remain undefined (or **nil** in CLRS).

To initialize the search we mark the colour of every vertex as *white* and the queue is empty. Then the first step is to mark the colour of v to be *grey*, put $\pi[v]$ to be undefined.

Breadth-first search repetitive step

Then the following procedure is repeated until the queue, Q, is empty.

```
procedure BFS(v)
    Push v on to the tail of Q
    while Q is not empty
        Pop vertex w from the head of Q
        for each vertex x adjacent to w do
            if colour[x] is white then
                π[x] ← w
                colour[x] ← grey
                Push x on to the tail of Q
            end if
        end for
        colour[w] ← black
    end while
```

At the end of the search, every vertex in the graph will have colour *black* and the parent or predecessor array π will contain the details of the breadth-first search tree.

Queues revisited

Recall that a *queue* is a data structure whereby the element taken off the data structure is the element that has been on the queue for the longest time.

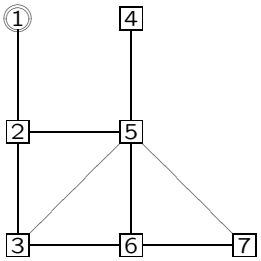
If the maximum length of the queue is known in advance (and is not too great) then a queue can be very efficiently implemented by simply using an array.

An array of n elements is initialized, and two pointers called *head* and *tail* are maintained — the head gives the location of the next element to be removed, while the tail gives the location of the first empty space in the array.

It is trivial to see that both enqueueing and dequeueing operations take $\Theta(1)$ time.

See CLRS (section 10.1) for further details.

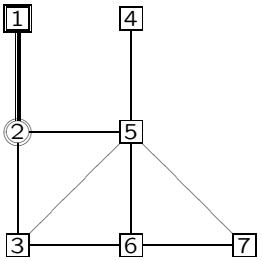
Example of breadth-first search



Head
↓
queue 1

x	*colour[x]*	$\pi[x]$
1	*grey*	**undef**
2	*white*	
3	*white*	
4	*white*	
5	*white*	
6	*white*	
7	*white*	

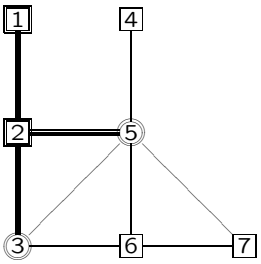
After visiting vertex 1



Head
↓
queue 1 2

x	*colour[x]*	$\pi[x]$
1	*black*	**undef**
2	*grey*	1
3	*white*	
4	*white*	
5	*white*	
6	*white*	
7	*white*	

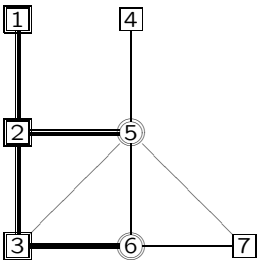
After visiting vertex 2



Head
↓
queue 1 2 3 5

x	*colour[x]*	$\pi[x]$
1	*black*	**undef**
2	*black*	1
3	*grey*	2
4	*white*	
5	*grey*	2
6	*white*	
7	*white*	

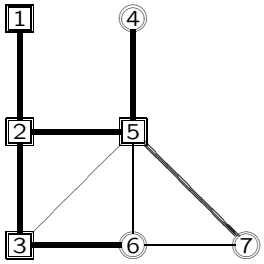
After visiting vertex 3



Head
↓
queue 1 2 3 5 6

x	*colour[x]*	$\pi[x]$
1	*black*	**undef**
2	*black*	1
3	*black*	2
4	*white*	
5	*grey*	2
6	*grey*	3
7	*white*	

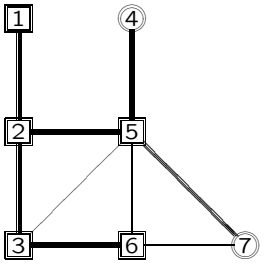
After visiting vertex 5



Head
↓
queue 1 2 3 5 6 4 7

x	*colour[x]*	$\pi[x]$
1	*black*	**undef**
2	*black*	1
3	*black*	2
4	*grey*	5
5	*black*	2
6	*grey*	3
7	*grey*	5

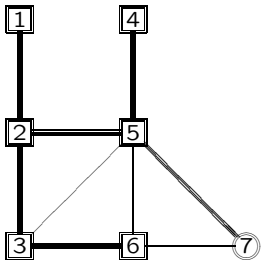
After visiting vertex 6



Head
↓
queue 1 2 3 5 6 4 7

x	*colour[x]*	$\pi[x]$
1	*black*	**undef**
2	*black*	1
3	*black*	2
4	*grey*	5
5	*black*	2
6	*black*	3
7	*grey*	5

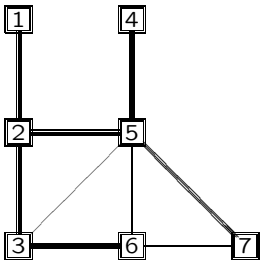
After visiting vertex 4



Head
↓
queue 1 2 3 5 6 4 7

x	*colour[x]*	$\pi[x]$
1	*black*	**undef**
2	*black*	1
3	*black*	2
4	*black*	5
5	*black*	2
6	*black*	3
7	*grey*	5

After visiting vertex 7



Head
↓
queue 1 2 3 5 6 4 7

x	*colour[x]*	$\pi[x]$
1	*black*	**undef**
2	*black*	1
3	*black*	2
4	*black*	5
5	*black*	2
6	*black*	3
7	*black*	5

At termination

At the termination of breadth-first search every vertex in the same connected component as v is a black vertex and the array π contains details of a spanning tree for that component — the breadth-first tree.

Time analysis

During the breadth-first search each vertex is enqueued once and dequeued once. As each enqueueing/dequeueing operation takes constant time, the queue manipulation takes $\Theta(V)$ time. At the time the vertex is dequeued, the adjacency list of that vertex is completely examined. Therefore we take $\Theta(E)$ time examining all the adjacency lists and the total time is $\Theta(V + E)$.

Uses of BFS

Breadth-first search is particularly useful for certain simple tasks such as determining whether a graph is connected, or finding the distance between two vertices.

The vertices of G are examined in order of increasing distance from v — first v, then its neighbours, then the vertices at distance 2 from v and so on. The spanning tree constructed provides a shortest path from any vertex back to v just by following the array π.

Therefore it is simple to modify the breadth-first search to provide an array of distances *dist* where *dist[u]* is the distance of the vertex u from the source vertex v.

Breadth-first search finding distances

To initialize the search we mark the colour of every vertex as *white* and the queue is empty. Then the first step is to mark the colour of v to be *grey*, set $\pi[v]$ to be undefined, set *dist[v]* to be 0, and add v to the queue, Q. Then we repeat the following procedure.

while Q is not empty
  Pop vertex w from the head of Q
  for each vertex x adjacent to w **do**
    if *colour[x]* is *white* **then**
      dist[x] $\leftarrow$ *dist[w]+1*
      $\pi[x] \leftarrow w$
      colour[x] $\leftarrow$ *grey*
      Push x on to the tail of Q
    end if
  end for
  colour[w] $\leftarrow$ *black*
end while

Depth-first search

Depth-first search is another important technique for searching a graph. Similarly to breadth-first search it also computes a spanning tree for the graph, but the tree is very different.

The structure of depth-first search is naturally *recursive* so we will give a recursive description of it. Nevertheless it is useful and important to consider the non-recursive implementation of the search.

The fundamental idea behind depth-first search is to visit the next unvisited vertex, thus extending the current path as far as possible. When the search gets stuck in a "corner" we back up along the path until a new avenue presents itself (this is called *backtracking*).

Basic recursive depth-first search

The following recursive program computes the depth-first search tree for a graph G starting from the source vertex v.

To initialize the search we mark the colour of every vertex as *white*. Then we call the recursive routine DFS(v) where v is the source vertex.

procedure DFS(w)
  colour[w] $\leftarrow$ *grey*
  for each vertex x adjacent to w **do**
    if *colour[x]* is *white* **then**
      $\pi[x] \leftarrow w$
      DFS(x)
    end if
  end for
  colour[w] $\leftarrow$ *black*

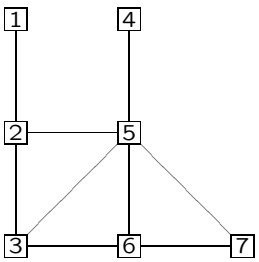
At the end of this depth-first search procedure we have produced a spanning tree containing every vertex in the connected component containing v.

A Non-recursive DFS

All recursive algorithms can be implemented as non-recursive algorithms. A non-recursive DFS requires a *stack* to record the previously visited vertices.

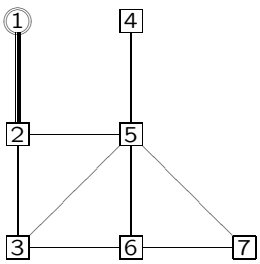
procedure DFS(w)
  initialize stack S
  push w onto S
  while S not empty **do**
    $x \leftarrow$ pop off S
    if *colour[x]*=*white* **then**
      colour[x] $\leftarrow$ *black*
      for each vertex y adjacent to x **do**
        if *colour[y]* is *white* **then**
          push y onto S
          $\pi[y] \leftarrow x$
        end if
      end for
    end if
  end while

Example of depth-first search



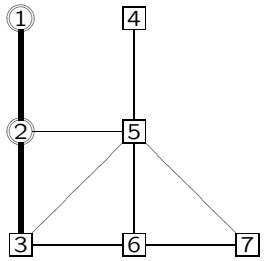
x	*colour[x]*	$\pi[x]$
1	*white*	**undef**
2	*white*	
3	*white*	
4	*white*	
5	*white*	
6	*white*	
7	*white*	

Immediately prior to calling DFS(2)



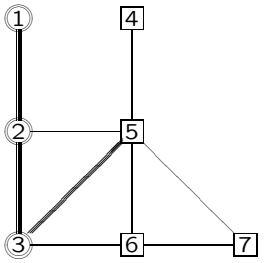
x	*colour[x]*	$\pi[x]$
1	*grey*	**undef**
2	*white*	1
3	*white*	
4	*white*	
5	*white*	
6	*white*	
7	*white*	

Immediately prior to calling DFS(3)



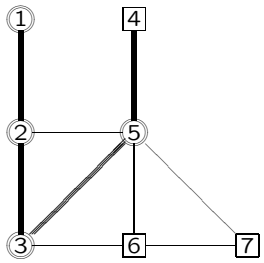
x	*colour[x]*	$\pi[x]$
1	*grey*	**undef**
2	*grey*	1
3	*white*	2
4	*white*	
5	*white*	
6	*white*	
7	*white*	

Immediately prior to calling DFS(5)



x	*colour[x]*	$\pi[x]$
1	*grey*	**undef**
2	*grey*	1
3	*grey*	2
4	*white*	
5	*white*	3
6	*white*	
7	*white*	

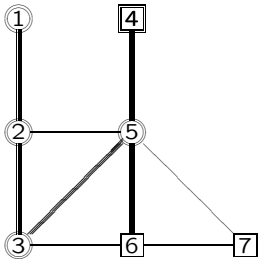
Immediately prior to calling DFS(4)



x	*colour[x]*	$\pi[x]$
1	*grey*	**undef**
2	*grey*	1
3	*grey*	2
4	*white*	5
5	*grey*	3
6	*white*	
7	*white*	

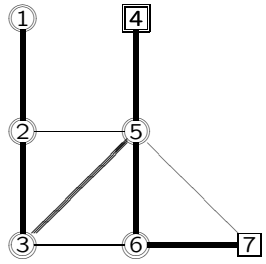
Immediately prior to calling DFS(6)

Now the call to DFS(4) actually finishes without making any more recursive calls so we return to examining the neighbours of vertex 5, the next of which is vertex 6.



x	*colour[x]*	$\pi[x]$
1	*grey*	**undef**
2	*grey*	1
3	*grey*	2
4	*black*	5
5	*grey*	3
6	*white*	5
7	*white*	

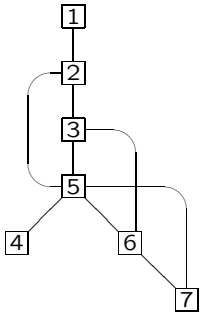
Immediately prior to calling DFS(7)



x	*colour[x]*	$\pi[x]$
1	*grey*	**undef**
2	*grey*	1
3	*grey*	2
4	*black*	5
5	*grey*	3
6	*grey*	5
7	*white*	6

The depth-first search tree

After completion of the search we can draw the depth-first search tree for this graph:



In this picture the slightly thicker straight edges are the **tree edges** (see later) and the remaining edges are the **back edges** — the back edges arise when we examine an edge (u, v) and discover that its endpoint v no longer has the colour *white*

Analysis of DFS

The running time of DFS is easy to analyse as follows.

First we observe that the routine DFS(w) is called exactly once for each vertex w; during the execution of this routine we perform only constant time array accesses, and run through the adjacency list of w once.

Running through the adjacency list of each vertex exactly once takes $\Theta(E)$ time overall, and hence the total time taken is $\Theta(V + E)$.

In fact, we can say more and observe that because every vertex and every edge are examined precisely once in both BFS and DFS, the time taken is $\Theta(V + E)$.

Discovery and finish times

The operation of depth-first search actually gives us more information than simply the depth-first search tree; we can assign two times to each vertex.

Consider the following modification of the search, where *time* is a global variable that starts at time 1.

```
procedure DFS(w)
    colour[w] ← grey
    discovery[w] ← time
    time ← time+1
    for each vertex x adjacent to w do
        if colour[x] is white then
            π[x] ← w
            DFS(x)
        end if
    end for
    colour[w] ← black
    finish[w] ← time
    time ← time+1
```

The parenthesis property

This assigns to each vertex a *discovery* time, which is the time at which it is first discovered, and a *finish* time, which is the time at which all its neighbours have been searched and it no longer plays any further role in the search.

The discovery and finish times satisfy a property called the *parenthesis property*.

Imagine writing down an expression consisting entirely of labelled parentheses — at the time of discovering vertex u we open a parenthesis $(_u$ and a the time of finishing with u we close the parenthesis $)_u$.

Then the resulting expression is a well-formed expression with correctly nested parentheses.

For our example depth-first search we get:

$$(_1\ (_2\ (_3\ (_4\ (_5\ 5)\ (_6\ (_7\ 7)\ 6)\ 4)\ 3)\ 2)\ 1)$$

Depth-first search for directed graphs

A depth-first search on an undirected graph produces a classification of the edges of the graph into *tree edges*, or *back edges*. For a directed graph, there are further possibilities. The same depth-first search algorithm can be used to classify the edges into four types:

tree edges If the procedure DFS(u) calls DFS(v) then (u, v) is a tree edge

back edges If the procedure DFS(u) explores the edge (u, v) but finds that v is an already visited ancestor of u, then (u, v) is a back edge

forward edges If the procedure DFS(u) explores the edge (u, v) but finds that v is an already visited descendant of u, then (u, v) is a forward edge

cross edges All other edges are cross-edges

Topological sort

We shall consider a classic simple application of depth-first search.

Definition A *directed acyclic graph (dag)* is a directed graph with no directed cycles.

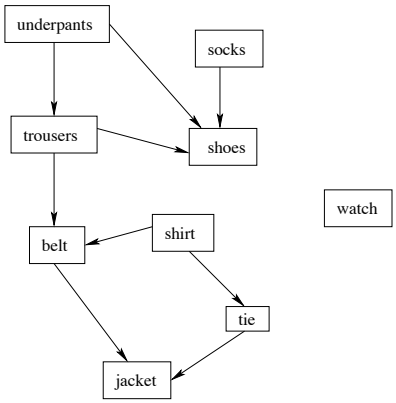
Theorem In a depth-first search of a dag there are no back edges.

Consider now some complicated process in which various jobs must be completed before others are started. We can model this by a graph D where the vertices are the jobs to be completed and there is an edge from job u to job v if job u must be completed before job v is started. Our aim is to find some linear ordering of the jobs such that they can be completed without violating any of the constraints.

This is called finding a *topological sort* of the dag D.

Example of a dag to be topologically sorted

For example, consider this dag describing the stages of getting dressed and the dependency between items of clothing (from CLRS, page 550).



What is the appropriate linear order in which to do these jobs so that all the precedences are satisfied.

Algorithm for TOPOLOGICAL SORT

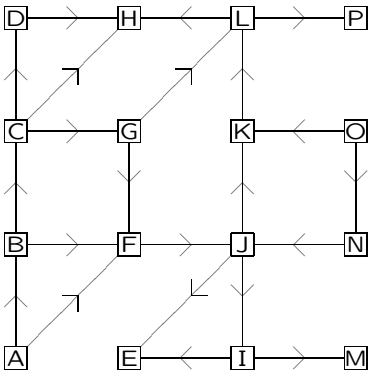
The algorithm for topological sort is an extremely simple application of depth-first search.

Algorithm

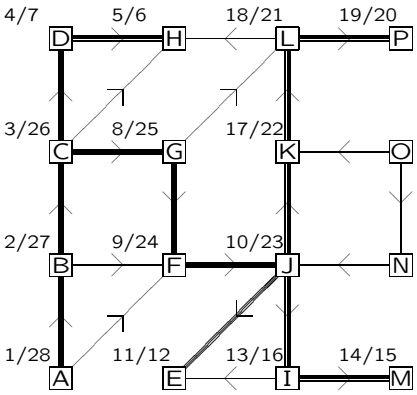
Apply the depth-first search procedure to find the finishing times of each vertex. As each vertex is finished, put it onto the *front* of a linked list.

At the end of the depth-first search the linked list will contain the vertices in topologically sorted order.

Doing the topological sort

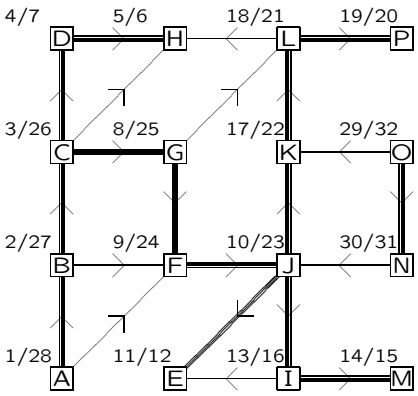


After the first depth-first search



Notice that there is a component that has not been reached by the depth-first search. To complete the search we just repeatedly perform depth-first searches until all vertices have been examined.

After the entire search



As the vertices were placed at the front of a linked list as they became finished the final topological sort is: $O - N - A - B - C - G - F - J - K - L - P - I - M - E - D - H$

A topologically sorted dag has the property that any edges drawn in the above diagram will got from left-to-right.

Analysis and correctness

Time analysis of the algorithm is very easy — to the $\Theta(V+E)$ time for the depth-first search we must add $\Theta(V)$ time for the manipulation of the linked list. Therefore the total time taken is again $\Theta(V+E)$.

Proof of topological sort

Suppose DFS has calculated the finish times of a dag $G=(V,E)$. For any pair of adjacent vertices $u,v \in V$ (implying $(u,v) \in E$) then we just need to show $f[v] < f[u]$ (the destination vertex v must finish first).

For each edge (u,v) explored by DFS of G consider the colour of vertex v.

GREY: v can never be grey since v should therefore be an ancestor of u and so the graph would be cyclic.

Proof (contd)

WHITE: v is a descendant of u so we will set its time now but we are still exploring u so we will set its finished time at some point in the future (and so therefore $f[v] < f[u]$). (refer back to the psuedocode).

BLACK: v has already been visited and so its finish time must have been set earlier, whereas we are still exploring u and so we will set its finish time in the future (and so again $f[v] < f[u]$).

Since for every edge in G there are two possible destination vertex colours and in each case we can show $f[v] < f[u]$, we have shown that this property applies to every connected vertex in G.

See CLRS (theorem 22.11) for a more thorough treatment.

Other uses for DFS

DFS is the standard algorithmic method for solving the following two problems:

Strongly connected components In a directed graph D a strongly connected component is a maximal subset S of the vertices such that for any two vertices u, $v \in S$ there is a directed path from u to v and from v to u.

Depth-first search can be used on a digraph to find strongly connected components in time $\Theta(V+E)$.

Articulation points In a connected, undirected graph, an *articulation point* is a vertex whose removal disconnects the graph.

Depth-first search can be used on a graph to find all the articulation points in time $\Theta(V+E)$.

Minimum spanning tree (MST)

Consider a group of villages in a remote area that are to be connected by telephone lines. There is a certain cost associated with laying the lines between any pair of villages, depending on their distance apart, the terrain and some pairs just cannot be connected.

Our task is to find the minimum possible cost in laying lines that will connect all the villages.

This situation can be modelled by a weighted graph W, in which the weight on each edge is the cost of laying that line. A *minimum spanning tree* in a graph is a subgraph that is (1) a spanning subgraph (2) a tree and (3) has a lower weight than any other spanning tree.

It is clear that finding a MST for W is the solution to this problem.

The greedy method

Definition A *greedy algorithm* is an algorithm in which at each stage a locally optimal choice is made.

A greedy algorithm is therefore one in which no overall strategy is followed, but you simply do whatever looks best at the moment.

For example a mountain climber using the greedy strategy to climb Everest would at every step climb in the steepest direction. From this analogy we get the computational search technique known as *hill-climbing*.

In general greedy methods have limited use, but fortunately, the problem of finding a minimum spanning tree can be solved by a greedy method.

Kruskal's method

Kruskal invented the following very simple method for building a minimum spanning tree. It is based on building a forest of lowest possible weight and continuing to add edges until it becomes a spanning tree.

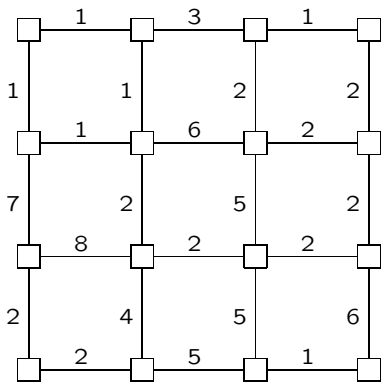
Kruskal's method

Initialize F to be the forest with all the vertices of G but none of the edges.

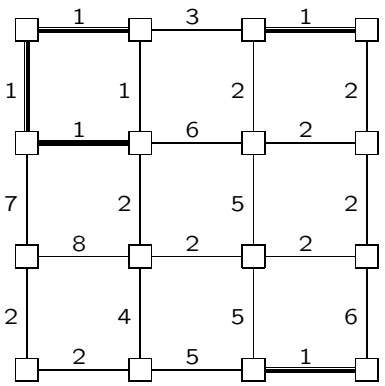
repeat
  Pick an edge e of minimum possible weight
  if $F \cup \{e\}$ is a forest **then**
    $F \leftarrow F \cup \{e\}$
  end if
until F contains $n - 1$ edges

Therefore we just keep on picking the smallest possible edge, and adding it to the forest, providing that we never create a cycle along the way.

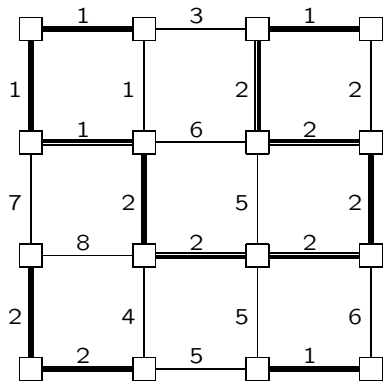
Example



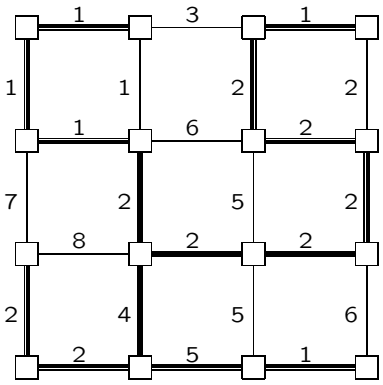
After using edges of weight 1



After using edges of weight 2



The final MST



Prim's algorithm

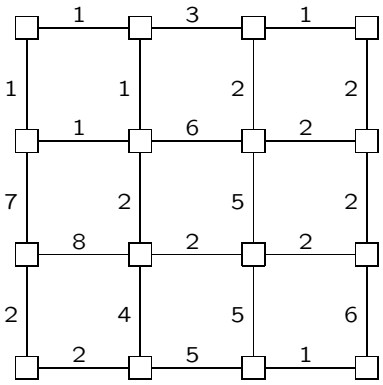
Prim's algorithm is another greedy algorithm for finding a minimum spanning tree.

The idea behind Prim's algorithm is to grow a minimum spanning tree edge-by-edge by always adding the shortest edge that touches a vertex in the current tree.

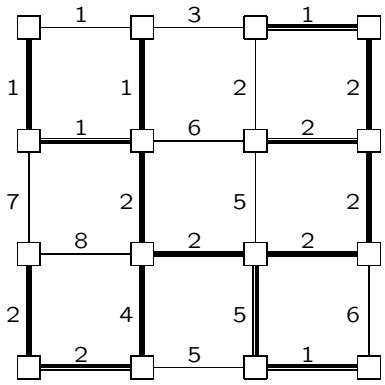
Notice the difference between the algorithms: Kruskal's algorithm always maintains a spanning subgraph which only becomes a tree at the final stage.

On the other hand, Prim's algorithm always maintains a tree which only becomes spanning at the final stage.

Prim's algorithm in action



One solution



Problem solved?

As far as a mathematician is concerned the problem of a minimum spanning tree is well-solved. We have two simple algorithms both of which are guaranteed to find the best solution. (After all, a greedy algorithm must be one of the simplest possible).

In fact, the reason why the greedy algorithm works in this case is well understood — the collection of all the subsets of the edges of a graph that do not contain a cycle forms what is called a (graphic) **matroid** (see CLRS, section 16.4).

Loosely speaking, a greedy algorithm always works on a matroid and never works otherwise.

Implementation issues

In fact the problem is far from solved because we have to decide how to *implement* the two greedy algorithms.

The details of the implementation of the two algorithms are interesting because they use (and illustrate) two important data structures — the *partition* and the *priority queue*.

Implementation of Kruskal

The main problem in the implementation of Kruskal is to decide whether the next edge to be added is allowable — that is, does it create a cycle or not.

Suppose that at some stage in the algorithm the next shortest edge is $\{x, y\}$. Then there are two possibilities:

x and y lie in different trees of F: In this case adding the edge does not create any new cycles, but merges together two of the trees of F

x and y lie in the same tree of F: In this case adding the edge creates a cycle and the edge should not be added to F

Therefore we need data structures that allow us to quickly find the tree to which an element belongs and quickly merge two trees.

Partitions or disjoint sets

The appropriate data structure for this problem is the *partition* (sometimes known under the name disjoint sets). Recall that a partition of a set is a collection of disjoint subsets (called cells) that cover the entire set.

At the beginning of Kruskal's algorithm we have a partition of the vertices into the discrete partition where each cell has size 1. As each edge is added to the minimum spanning tree, the number of cells in the partition decreases by one.

The operations that we need for Kruskal's algorithm are

Union(cell,cell) Creates a new partition by merging two cells

Find(element) Finds the cell containing a given element

Naive partitions

One simple way to represent a partition is simply to choose one element of each cell to be the "leader" of that cell. Then we can simply keep an array π of length n where $\pi(x)$ is the leader of the cell containing x.

Example Consider the partition of 8 elements into 3 cells as follows:

$$\{0,2 \mid 1,3,5 \mid 4,6,7\}$$

We could represent this as an array as follows

x	0	1	2	3	4	5	6	7
$\pi(x)$	0	1	0	1	4	1	4	4

Then certainly the operation **Find** is straightforward — we can decide whether x and y are in the same cell just by comparing $\pi(x)$ with $\pi(y)$.

Thus **Find** has complexity $\Theta(1)$.

Updating the partition

Suppose now that we wish to update the partition by merging the first two cells to obtain the partition

$$\{0,1,2,3,5 \mid 4,6,7\}$$

We could update the data structure by running through the entire array π and updating it as necessary.

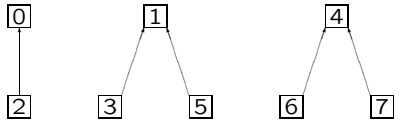
x	0	1	2	3	4	5	6	7
$\pi(x)$	0	0	0	0	4	0	4	4

This takes time $\Theta(n)$, and hence the merging operation is rather slow.

Can we improve the time of the merging operation?

The disjoint sets forest

Consider the following graphical representation of the data structure above, where each element points (upwards) to the "leader" of the cell.

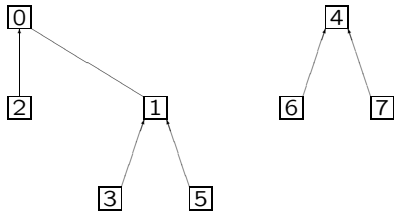


Merging two cells is accomplished by adjusting the pointers so they point to the new leader.

However we can achieve something similar by just adjusting one pointer — suppose we simply change the pointer for the element 1, by making it point to 0 instead of itself.

The new data structure

Just adjusting this one pointer results in the following data structure.



This new improved merging has complexity only $\Theta(1)$. However we have now lost the ability to do the **Find** properly. In order to correctly find the leader of the cell containing an element we have to run through a little loop:

procedure Find(x)
  while $x \neq \pi(x)$
    $x = \pi(x)$

This new find operation may take time $O(n)$ so we seem to have gained nothing.

Union-by-rank heuristic

There are two heuristics that can be applied to the new data structure, that speed things up enormously at the cost of maintaining a little extra data.

Let the *rank* of a root node of a tree be the height of that tree (the maximum distance from a leaf to the root).

The *union-by-rank* heuristic tries to keep the trees *balanced* at all times. When a merging operation needs to be done, the root of the shorter tree is made to point to the root of the taller tree. The resulting tree therefore does not increase its height unless both trees are the same height in which case the height increases by one.

Path compression heuristic

The path compression heuristic is based on the idea that when we perform a **Find(x)** operation we have to follow a path from x to the root of the tree containing x.

After we have done this why do we not simply go back down through this path and make all these elements point *directly* to the root of the tree, rather than in a long chain through each other?

This is reminiscent of our naive algorithm, where we made *every* element point directly to the leader of its cell, but it is much cheaper because we only alter things that we needed to look at anyway.

Complexity of Kruskal

In the worst case, we will perform E operations on the partition data structure which has size V. By the complicated argument in CLRS we see that the total time for these operations if we use both heuristics is $O(E \lg^* V)$. (Note: $\lg^* x$ is the iterated log function, which grows *extremely* slowly with x; see CLRS, page 56)

However we must add to this the time that is needed to sort the edges — because we have to examine the edges in order of length. This time is $O(E \lg E)$ if we use a sorting technique such as mergesort, and hence the overall complexity of Kruskal's algorithm is $O(E \lg E)$.

Implementation of Prim

For Prim's algorithm we repeatedly have to select the next vertex that is *closest* to the tree that we have built so far. Therefore we need some sort of data structure that will enable us to associate a value with each vertex (being the distance to the tree under construction) and rapidly select the vertex with the lowest value.

From our study of Data Structures we know that the appropriate data structure is a *priority queue* and that a priority queue is implemented by using a *heap*.

The priority queue ADT

Recall that a *priority queue* is an abstract data type that stores objects with an associated key. The priority of the object depends on the value of the key.

The operations associated with this data type include

insert(queue,entry,key) Places an entry with its associated key into the data structure

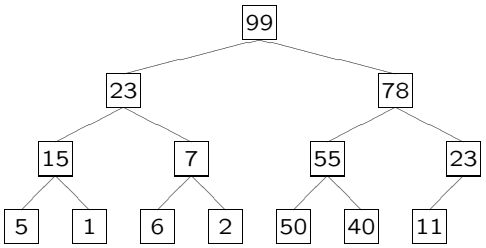
change(queue,entry,newkey) Changes the value of the key associated with a given entry

max(queue) Returns the element with the highest priority

extractmax(queue) Returns the element with the highest priority from the queue and deletes it from the queue

Heaps

A *heap* is a complete binary tree such that the key associated with any node is larger than (or equal to) the key associated with either of its children. This means that the root of the binary tree has the largest key.



We can insert items into a heap, change the key value of an item in the heap, and remove the item at the root from a heap, (always maintaining the heap property) in time $O(\log n)$, where n is the size of the heap.

A heap can be used to implement all priority queue operations in time $O(\log n)$.

Prim's algorithm

It is now easy to see how to implement Prim's algorithm.

We first initialize our priority queue Q to empty. We then select an arbitrary vertex s to grow our minimum spanning tree A and set the key value of s to 0. In addition, we maintain an array, π, where each element $\pi[v]$ contains the vertex that connects v to the spanning tree being grown.

Here we want low key values to represent high priorities, so we will rename our two last priority queue operations to **min(queue)** and **extractmin(queue)**.

Next, we add each vertex $v \neq s$ to Q and set the key value $key[v]$ using the following criteria:

$$key[v] = \begin{cases} weight(v,s) & \text{if } (v,s) \in E \\ \infty & \text{otherwise} \end{cases}$$

Each time an element is added to the priority queue Q, a *heapify* is carried out to maintain the heap property of Q. Since low key values represent high priorities, the heap for Q is so maintained that the key associated with any node is smaller (rather than larger) than the key of any of its children. This means that the root of the binary tree always has the smallest key.

We store the following information in the minimum spanning tree A: $(v, key[v], \pi[v])$. Thus, at the beginning of the algorithm, $A = \{(s, 0, \texttt{undef})\}$.

At each stage of the algorithm:

1. We extract the vertex u that has the highest priority (that is, the lowest key value!). With the binary tree being *heapified*, u is simply the root of the tree.
2. We add $(u, key[u], \pi[u])$ to A and carry out **extractmin(Q)**

3. We then examine the neighbours of u. For each neighbour v, there are two possibilities:

   (a) If v is is already in the spanning tree A being constructed then we do not consider it further.
   (b) If v is currently on the priority queue Q, then we see whether this new edge (u, v) should cause an update in the priority of v. If the value $weight(u, v)$ is lower than the current key value of v, then we change $key[v]$ to $weight(u, v)$ and set $\pi[v] = u$. Note that each time the key value of a vertex in Q is updated, a *heapify* is carried out to maintain the heap property of Q.

At the termination of the algorithm, $Q = \emptyset$ and the spanning tree A contain all the edges, together with their weights, that span the tree.

Complexity of Prim

The complexity of Prim's algorithm is dominated by the heap operations.

Every vertex is extracted from the priority queue at some stage, hence the **extractmin** operations in the worst case take time $O(V \lg V)$.

Also, every edge is examined at some stage in the algorithm and each edge examination potentially causes a **change** operation. Hence in the worst case these operations take time $O(E \lg V)$.

Therefore the total time is

$$O(V \lg V + E \lg V) = O(E \lg V)$$

Which is the better algorithm: Kruskal's or Prim's?

Priority-first search

Let us generalize the ideas behind this implementation of Prim's algorithm.

Consider the following very general graph-searching algorithm. We will later show that by choosing different specifications of the priority we can make this algorithm do very different things. This algorithm will produce a *priority-first search tree*.

The key-values or priorities associated with each vertex are stored in an array called key.

Initially we set $key[v]$ to ∞ for all the vertices $v \in V(G)$ and build a heap with these keys — this can be done in time $O(V)$.

Then we select the source vertex s for the search and perform **change(s,0)** to change the key of s to 0, thus placing s at the top of the priority queue.

The operation of PFS

After initialization the operation of PFS is as follows:

procedure PFS(s)
  change(s,0)
  while $Q \neq \emptyset$
    $u \leftarrow$ **extractmin(**Q**)**
    for each v adjacent to u **do**
      if $v \in Q \wedge$ *PRIORITY* $< key[v]$ **then**
        $\pi[v] \leftarrow u$
        change(Q,v,*PRIORITY*)
      end if
    end for
  end while

It is important to notice how the array π is managed — for every vertex $v \in Q$ with a finite key value, $\pi[v]$ is the vertex *not in* Q that was responsible for the key of v reaching the highest priority it has currently reached.

Complexity of PFS

The complexity of this search is easy to calculate — the main loop is executed V times, and each **extractmin** operation takes $O(\lg V)$ yielding a total time of $O(V \lg V)$ for the extraction operations.

During all V operations of the main loop we examine the adjacency list of each vertex exactly once — hence we make E calls, each of which may cause a **change** to be performed. Hence we do at most $O(E \lg V)$ work on these operations.

Therefore the total is

$$O(V \lg V + E \lg V) = O(E \lg V).$$

Prim's algorithms is PFS

Prim's algorithm can be expressed as a priority-first search by observing that the priority of a vertex is the weight of the shortest edge joining the vertex to the rest of the tree.

This is achieved in the code above by simply replacing the string *PRIORITY* by

$$weight(u, v)$$

At any stage of the algorithm:

• The vertices not in Q form the tree so far.

• For each vertex $v \in Q$, $key[v]$ gives the length of the shortest edge from v to a vertex in the tree, and $\pi[v]$ shows which tree vertex that is.

Shortest paths

Let G be a directed weighted graph. The *shortest path* between two vertices v and w is the path from v to w for which the sum of the weights on the path-edges is lowest. Notice that if we take an unweighted graph to be a special instance of a weighted graph, but with all edge weights equal to 1, then this coincides with the normal definition of shortest path.

The weight of the shortest path from v to w is denoted by $\delta(v, w)$.

Let $s \in V(G)$ be a specified vertex called the *source* vertex.

The *single-source shortest paths* problem is to find the shortest path from s to every other vertex in the graph (as opposed to the *all-pairs shortest paths problem*, where we must find the distance between every pair of vertices).

Dijkstra's algorithm

Dijkstra's algorithm is a famous single-source shortest paths algorithm suitable for the cases when the weights are all non-negative.

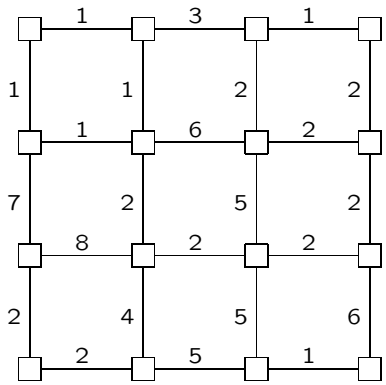
Dijkstra's algorithm can be implemented as a priority-first search by taking the priority of a vertex $v \in Q$ to be the shortest path from s to v that consists entirely of vertices in the priority-first search tree (except of course for v).

This can be implemented as a PFS by replacing PRIORITY with

$$key[u] + weight(u, v)$$

At the end of the search, the array $key[]$ contains the lengths of the shortest paths from the source vertex s.

Dijkstra's algorithm in action



Proof of correctness

It is possible to prove that Dijkstra's algorithm is correct by proving the following claim (assuming $T = V(G) - Q$ is the set of vertices that have already been removed from Q).

> *At the time that a vertex u is removed from Q and placed into T*
> $key[u] = \delta(s, u)$.

This is a proof by contradiction, meaning that we try to prove $key[u] \neq \delta(s, u)$ and if we fail then we will have proved the opposite.

Assuming $u \neq s$ then $T \neq \emptyset$ and there exists a path p from s to u. We can decompose the path into three sections:

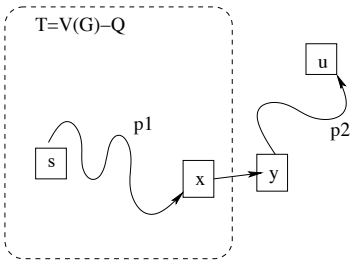
1. A path p_1 from s to vertex x, such that $x \in T$ and the path is of length 0 or more.

2. An edge between x and y, such that $y \in Q$ and $(x, y) \in E(G)$.

3. A path p_2 from y to u of length 0 or more.

Proof (contd)

The decomposed path may be illustrated thus.



Firstly, we know $key[y] = \delta(s, y)$ since the edge (x, y) will have been examined when x was added to T.

Furthermore, we know that y is before u on path p and therefore $\delta(s, y) \leq \delta(s, u)$. This implies $key[y] \leq key[u]$ (inequality A).

Proof (contd)

But we also know that u was chosen from Q before y which implies $key[u] \leq key[y]$ (inequality B) since the priority queue always returns the vertex with the smallest key.

Inequalities A and B can only be satisfied if $key[u] = key[y]$ but this implies

$$key[u] = \delta(s,u) = \delta(s,y) = key[y]$$

But our initial assumption was that $key[u] \neq \delta(s,u)$ giving rise to the contradiction. Hence we have proved that $key[u] = \delta(s,u)$ at the time that u enters T.

Relaxation

Consider the following property of Dijkstra's algorithm.

• At any stage of Dijkstra's algorithm the following inequality holds:

$$\delta(s,v) \leq key[v]$$

This is saying that the *key[]* array always holds a collection of *upper bounds* on the actual values that we are seeking. We can view these values as being our "best estimate" to the value so far, and Dijkstra's algorithm as a procedure for systematically improving our estimates to the correct values.

The fundamental step in Dijkstra's algorithm, where the bounds are altered is when we examine the edge (u,v) and do the following operation

$$key[v] \leftarrow \mathbf{min}(key[v], key[u] + weight(u,v))$$

This is called *relaxing* the edge (u,v).

Relaxation schedules

Consider now an algorithm that is of the following general form:

- Initially an array $d[]$ is initialized to have $d[s] = 0$ for some source vertex s and $d[v] = \infty$ for all other vertices
- A sequence of edge relaxations is performed, possibly altering the values in the array $d[]$.

We observe that the value $d[v]$ is always an upper bound for the value $\delta(s,v)$ because relaxing the edge (u,v) will either leave the upper bound unchanged or replace it by a better estimate from an upper bound on a path from $s \rightarrow u \rightarrow v$.

Dijkstra's algorithm is a particular schedule for performing the edge relaxations that guarantees that the upper bounds converge to the exact values.

Negative edge weights

Dijkstra's algorithm cannot be used when the graph has some negative edge-weights (why not? find an example).

In general, no algorithm for shortest paths can work if the graph contains a cycle of negative total weight (because a path could be made arbitrarily short by going round and round the cycle). Therefore the question of finding shortest paths makes no sense if there is a negative cycle.

However, what if there are some negative edge weights but no negative cycles?

The Bellman-Ford algorithm is a relaxation schedule that can be run on graphs with negative edge weights. It will either *fail* in which case the graph has a negative cycle and the problem is ill-posed, or will finish with the single-source shortest paths in the array $d[]$.

Bellman-Ford algorithm

The initialization step is as described above. Let us suppose that the weights on the edges are given by the function w.

Then consider the following relaxation schedule:

for $i = 1$ **to** $|V(G)| - 1$ **do**
  for each edge $(u, v) \in E(G)$ **do**
    $d[v] \leftarrow \mathbf{min}(d[v], d[u] + w(u, v))$
  end for each
end for

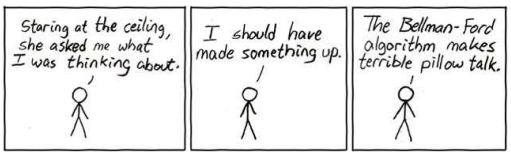
Finally we make a single check to determine if we have a failure:

for each edge $(u, v) \in E(G)$ **do**
  if $d[v] > d[u] + w(u, v)$ **then**
    FAIL
  end if
end for each

Complexity of Bellman-Ford

The complexity is particularly easy to calculate in this case because we know exactly how many relaxations are done — namely $E(V - 1)$, and adding that to the final failure check loop, and the initialization loop we see that Bellman-Ford is $O(EV)$

There remains just one question — how does it work?



Correctness of Bellman-Ford

Let us consider some of the properties of relaxation in a graph with no negative cycles.

Property 1 Consider an edge (u, v) that lies on the shortest path from s to v. If the sequence of relaxations includes relaxing (u, v) at a stage when $d[u] = \delta(s, u)$, then $d[v]$ is set to $\delta(s, v)$ and never changes after that.

Once convinced that Property 1 holds we can show that the algorithm is correct for graphs with no negative cycles, as follows.

Consider any vertex v and let us examine the shortest path from s to v, namely

$$s \sim v_1 \sim v_2 \cdots v_k \sim v$$

Now at the initialization stage $d[s] = 0$ and it always remains the same. After one pass through the main loop the edge (s, v_1) is relaxed and by Property 1, $d[v_1] = \delta(s, v_1)$ and it remains at that value. After the second pass the edge (v_1, v_2) is relaxed and after this relaxation we have $d[v_2] = \delta(s, v_2)$ and it remains at this value.

As the number of edges in the path is at most $|V(G)| - 1$, after all the loops have been performed $d[v] = \delta(s, v)$.

Note that this is an inductive argument where the induction hyptohesis is *"after n iterations, all shortest paths of length n have been found"*.

All-pairs shortest paths

Now we turn our attention to constructing a complete table of shortest distances, which must contain the shortest distance between any pair of vertices.

If the graph has no negative edge weights then we could simply make V runs of Dijkstra's algorithm, at a total cost of $O(VE \lg V)$, whereas if there are negative edge weights then we could make V runs of the Bellman-Ford algorithm at a total cost of $O(V^2E)$.

The two algorithms we shall examine both use the adjacency matrix representation of the graph, hence are most suitable for dense graphs. Recall that for a weighted graph the weighted adjacency matrix A has $weight(i, j)$ as its ij-entry, where $weight(i, j) = \infty$ if i and j are not adjacent.

A dynamic programming method

Dynamic programming is a general algorithmic technique for solving problems that can be characterised by two features:

- The problem is broken down into a collection of smaller subproblems
- The solution is built up from the stored values of the solutions to all of the subproblems

For the all-pairs shortest paths problem we define the simpler problem to be

"What is the length of the shortest path from vertex i to j that uses at most m edges?"

We shall solve this for $m = 1$, then use that solution to solve for $m = 2$, and so on ...

The initial step

We shall let $d_{ij}^{(m)}$ denote the distance from vertex i to vertex j along a path that uses at most m edges, and define $D^{(m)}$ to be the matrix whose ij-entry is the value $d_{ij}^{(m)}$.

As a shortest path between any two vertices can contain at most $V - 1$ edges, the matrix $D^{(V-1)}$ contains the table of all-pairs shortest paths.

Our overall plan therefore is to use $D^{(1)}$ to compute $D^{(2)}$, then use $D^{(2)}$ to compute $D^{(3)}$ and so on.

The case $m = 1$

Now the matrix $D^{(1)}$ is easy to compute — the length of a shortest path using at most one edge from i to j is simply the weight of the edge from i to j. Therefore $D^{(1)}$ is just the adjacency matrix A.

The inductive step

What is the smallest weight of the path from vertex i to vertex j that uses at most m edges? Now a path using at most m edges can either be

(1) A path using less than m edges
(2) A path using exactly m edges, composed of a path using $m - 1$ edges from i to an auxiliary vertex k and the edge (k, j).

We shall take the entry $d_{ij}^{(m)}$ to be the lowest weight path from the above choices.

Therefore we get

$$d_{ij}^{(m)} = \min\left(d_{ij}^{(m-1)}, \min_{1 \le k \le V}\{d_{ik}^{(m-1)} + w(k, j)\}\right)$$

$$= \min_{1 \le k \le V}\{d_{ik}^{(m-1)} + w(k, j)\}$$

Example

Consider the weighted graph with the following weighted adjacency matrix:

$$A = D^{(1)} = \begin{pmatrix} 0 & \infty & 11 & 2 & 6 \\ 1 & 0 & 4 & \infty & \infty \\ 10 & \infty & 0 & \infty & \infty \\ \infty & 2 & 6 & 0 & 3 \\ \infty & \infty & 6 & \infty & 0 \end{pmatrix}$$

Let us see how to compute an entry in $D^{(2)}$, suppose we are interested in the $(1,3)$ entry:

Then we see that

$1 \to 1 \to 3$ has cost $0 + 11 = 11$
$1 \to 2 \to 3$ has cost $\infty + 4 = \infty$
$1 \to 3 \to 3$ has cost $11 + 0 = 11$
$1 \to 4 \to 3$ has cost $2 + 6 = 8$
$1 \to 5 \to 3$ has cost $6 + 6 = 12$

The minimum of all of these is 8, hence the $(1,3)$ entry of $D^{(2)}$ is set to 8.

Computing $D^{(2)}$

$$\begin{pmatrix} 0 & \infty & 11 & 2 & 6 \\ 1 & 0 & 4 & \infty & \infty \\ 10 & \infty & 0 & \infty & \infty \\ \infty & 2 & 6 & 0 & 3 \\ \infty & \infty & 6 & \infty & 0 \end{pmatrix} \begin{pmatrix} 0 & \infty & 11 & 2 & 6 \\ 1 & 0 & 4 & \infty & \infty \\ 10 & \infty & 0 & \infty & \infty \\ \infty & 2 & 6 & 0 & 3 \\ \infty & \infty & 6 & \infty & 0 \end{pmatrix}$$

$$= \begin{pmatrix} 0 & 4 & 8 & 2 & 5 \\ 1 & 0 & 4 & 3 & 7 \\ 10 & \infty & 0 & 12 & 16 \\ 3 & 2 & 6 & 0 & 3 \\ 16 & \infty & 6 & \infty & 0 \end{pmatrix}$$

If we multiply two matrices $AB = C$, then we compute

$$c_{ij} = \sum_{k=1}^{k=V} a_{ik} b_{kj}$$

If we replace the multiplication $a_{ik} b_{kj}$ by addition $a_{ik} + b_{kj}$ and replace summation Σ by the minimum min then we get

$$c_{ij} = \min_{k=1}^{k=V} a_{ik} + b_{kj}$$

which is precisely the operation we are performing to calculate our matrices.

The remaining matrices

Proceeding to compute $D^{(3)}$ from $D^{(2)}$ and A, and then $D^{(4)}$ from $D^{(3)}$ and A we get:

$$D^{(3)} = \begin{pmatrix} 0 & 4 & 8 & 2 & 5 \\ 1 & 0 & 4 & 3 & \boxed{6} \\ 10 & \boxed{14} & 0 & 12 & \boxed{15} \\ 3 & 2 & 6 & 0 & 3 \\ 16 & \infty & 6 & \boxed{18} & 0 \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 4 & 8 & 2 & 5 \\ 1 & 0 & 4 & 3 & 6 \\ 10 & 14 & 0 & 12 & 15 \\ 3 & 2 & 6 & 0 & 3 \\ 16 & \boxed{20} & 6 & 18 & 0 \end{pmatrix}$$

A new matrix "product"

Recall the method for computing $d_{ij}^{(m)}$, the (i,j) entry of the matrix $D^{(m)}$ using the method similar to matrix multiplication.

$d_{ij}^{(m)} \leftarrow \infty$
for $k = 1$ **to** V **do**
  $d_{ij}^{(m)} = \min(d_{ij}^{(m)}, d_{ik}^{(m-1)} + w(k,j))$
end for

Let us use $\star$ to denote this new matrix product.

Then we have

$$D^{(m)} = D^{(m-1)} \star A$$

Hence it is an easy matter to see that we can compute as follows:

$$D^{(2)} = A \star A \qquad D^{(3)} = D^{(2)} \star A \ldots$$

Complexity of this method

The time taken for this method is easily seen to be $\Theta(V^4)$ as it performs V matrix "multiplications" each of which involves a triply nested **for** loop with each variable running from 1 to V.

However we can reduce the complexity of the algorithm by remembering that we do not need to compute *all* the intermediate products $D^{(1)}$, $D^{(2)}$ and so on, but we are only interested in $D^{(V-1)}$. Therefore we can simply compute:

$$D^{(2)} = A \star A$$

$$D^{(4)} = D^{(2)} \star D^{(2)}$$

$$D^{(8)} = D^{(4)} \star D^{(4)}$$

Therefore we only need to do this operation at most $\lg V$ times before we reach the matrix we want. The time required is therefore actually $\Theta(V^3 \lceil \lg V \rceil)$.

Floyd-Warshall

The Floyd-Warshall algorithm uses a different dynamic programming formalism.

For this algorithm we shall define $d_{ij}^{(k)}$ to be the length of the shortest path from i to j whose intermediate vertices all lie in the set $\{1, \ldots, k\}$.

As before, we shall define $D^{(k)}$ to be the matrix whose (i, j) entry is $d_{ij}^{(k)}$.

The initial case

What is the matrix $D^{(0)}$ — the entry $d_{ij}^{(0)}$ is the length of the shortest path from i to j with *no* intermediate vertices. Therefore $D^{(0)}$ is simply the adjacency matrix A.

The inductive step

For the inductive step we assume that we have constructed already the matrix $D^{(k-1)}$ and wish to use it to construct the matrix $D^{(k)}$.

Let us consider all the paths from i to j whose intermediate vertices lie in $\{1, 2, \ldots, k\}$. There are two possibilities for such paths

(1) The path does not use vertex k
(2) The path does use vertex k

The shortest possible length of all the paths in category (1) is given by $d_{ij}^{(k-1)}$ which we already know.

If the path does use vertex k then it must go from vertex i to k and then proceed on to j, and the length of the shortest path in this category is $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.

The overall algorithm

The overall algorithm is then simply a matter of running V times through a loop, with each entry being assigned as the minimum of two possibilities. Therefore the overall complexity of the algorithm is just $O(V^3)$.

$D^{(0)} \leftarrow A$
for $k = 1$ **to** V **do**
  for $i = 1$ **to** V **do**
    for $j = 1$ **to** V **do**
      $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
    end for j
  end for i
end for k

At the end of the procedure we have the matrix $D^{(V)}$ whose (i, j) entry contains the length of the shortest path from i to j, all of whose vertices lie in $\{1, 2, \ldots, V\}$ — in other words, the shortest path in total.

Example

Consider the weighted directed graph with the following adjacency matrix:

$$D^{(0)} = \begin{pmatrix} 0 & \infty & 11 & 2 & 6 \\ 1 & 0 & 4 & \infty & \infty \\ 10 & \infty & 0 & \infty & \infty \\ \infty & 2 & 6 & 0 & 3 \\ \infty & \infty & 6 & \infty & 0 \end{pmatrix}$$

Let us see how to compute $D^{(1)}$

$$D^{(1)} = \begin{pmatrix} 0 & \infty & 11 & 2 & 6 \\ 1 & 0 & 4 & & \\ 10 & \infty & 0 & & \\ \infty & 2 & 6 & 0 & 3 \\ \infty & \infty & 6 & \infty & 0 \end{pmatrix}$$

To find the $(2,4)$ entry of this matrix we have to consider the paths through the vertex 1 — is there a path from $2-1-4$ that has a better value than the current path? If so, then that entry is updated.

The entire sequence of matrices

$$D^{(2)} = \begin{pmatrix} 0 & \infty & 11 & 2 & 6 \\ 1 & 0 & 4 & \boxed{3} & \boxed{7} \\ 10 & \infty & 0 & \boxed{12} & \boxed{16} \\ \boxed{3} & 2 & 6 & 0 & 3 \\ \infty & \infty & 6 & \infty & 0 \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & \infty & 11 & 2 & 6 \\ 1 & 0 & 4 & 3 & 7 \\ 10 & \infty & 0 & 12 & 16 \\ 3 & 2 & 6 & 0 & 3 \\ \boxed{16} & \infty & 6 & \boxed{18} & 0 \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & \boxed{4} & \boxed{8} & 2 & \boxed{5} \\ 1 & 0 & 4 & 3 & \boxed{6} \\ 10 & \boxed{14} & 0 & 12 & \boxed{15} \\ 3 & 2 & 6 & 0 & 3 \\ 16 & \boxed{20} & 6 & 18 & 0 \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 4 & 8 & 2 & 5 \\ 1 & 0 & 4 & 3 & 6 \\ 10 & 14 & 0 & 12 & 15 \\ 3 & 2 & 6 & 0 & 3 \\ 16 & 20 & 6 & 18 & 0 \end{pmatrix}$$

Finding the actual shortest paths

In both of these algorithms we have not addressed the question of actually finding the paths themselves.

For the Floyd-Warshall algorithm this is achieved by constructing a further sequence of arrays $P^{(k)}$ whose (i,j) entry contains a predecessor of j on the path from i to j. As the entries are updated the predecessors will change — if the matrix entry is not changed then the predecessor does not change, but if the entry does change, because the path originally from i to j becomes re-routed through the vertex k, then the predecessor of j becomes the predecessor of j on the path from k to j.

Summary

1. A graph G is fully described by a set of vertices $V(G)$ and a set of edges $E(G)$.

2. Graphs may be directed so that the edges correspond to one directional arcs: $(u,v) \in E(G) \not\Rightarrow (v,u) \in E(G)$

3. Graphs may be weighted when an additional weight value is associated with each edge: $w : E(G) \to \mathbf{R}$.

4. Graphs may be represented as *adjacency list* or *adjacency matrix* data structures.

5. Searching may occur *breadth* first (BFS) or *depth* first (DFS).

6. DFS and BFS create a spanning tree from any graph.

Summary (contd)

7. DFS visits the vertices nearest to the source first. It can be used to determine whether a graph is connected.

8. BFS visits the vertices furtherest to the source first. It can be used to perform a topological sort.

9. Kruskal's and Prim's methods are two greedy algorithms for determining the minimum spanning tree of a graph.

10. Dijkstra's method determines the shortest path between any two vertices in a directed graph so long as all the weights are non-negative.

11. When directed graphs have negative edge weights the Bellman-Ford algorithm may be used (but it will fail if the graph has a negative cycle).

Summary (contd)

12. Dynamic Programming is a general approach for solving problems which can be decomposed into sub-problems and where solutions to sub-problems can be combined to solve the main problem.

13. Dynamic Programming can be used to solve the shortest path problem directly or via the Floyd-Warshall formulation.

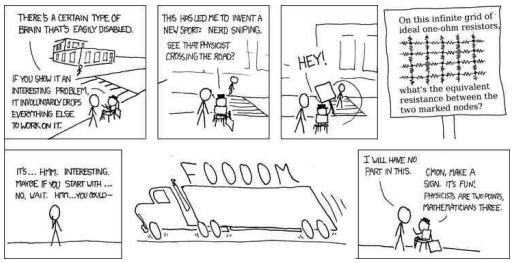
14. The minimum path problem can be used for motion planning of robots through large graphs using a priority first search.

Recommended reading:
CLRS, Chapters 22–25

Computer Science and Software Engineering, 2011

CITS3210 Algorithms

Network Flow

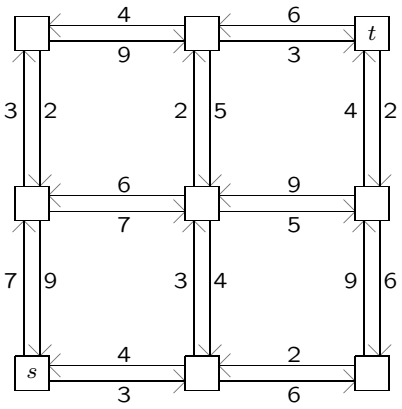


Notes by CSSE, Comics by xkcd.com

Flow networks

In this section we see how our fundamental graph theoretic algorithms can be combined to solve a more complex problem.

A *flow network* is a directed graph in which each directed edge (u, v) has a non-negative *capacity* $c(u, v) \geq 0$. The flow network has two special vertices — a *source* s and a *sink* t.



A flow

A *flow* in a flow network is a function

$$f : V \times V \to \mathbf{R}$$

that satisfies the following properties.

Capacity constraint

For each edge (u, v)

$$f(u, v) \leq c(u, v)$$

Skew symmetry

For each pair of vertices u, v

$$f(u, v) = -f(v, u)$$

Flow conservation

For all vertices $u \in V - \{s, t\}$ we have

$$\sum_{v \in V} f(u, v) = 0$$

The MAX FLOW problem

MAX FLOW
Instance. A flow network G with source s and sink t.
Question. What is the maximum flow from s to t?

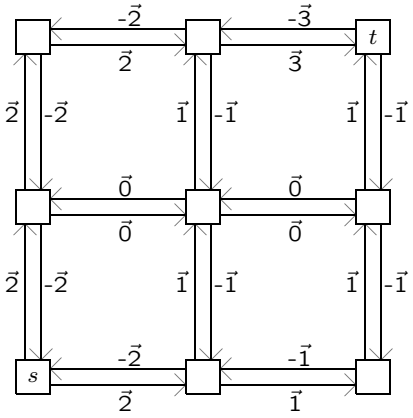
The most convenient mental model for the network flow problem is to think of the edges of the capacity graph as representing *pipelines* of various capacities.

The *source* is to be viewed as a producer of some sort of fluid (maybe an oil well), and the *sink* as a consumer of some sort of fluid (maybe an oil refinery).

The network flow problem is then the problem of deciding how much of the fluid to route along each of the pipelines in order to achieve the *maximum flow* of fluid from the source to the sink.

An example flow

This diagram shows a flow in the above flow network. Each edge is labelled with the amount of flow passing along that edge.



The *value* of a flow f is defined to be the total flow leaving the source vertex

$$|f| = \sum_{v \in V} f(s, v)$$

This flow has value 4.

Interpreting negative flows

The concept of a negative flow often appears to be a little confusing.

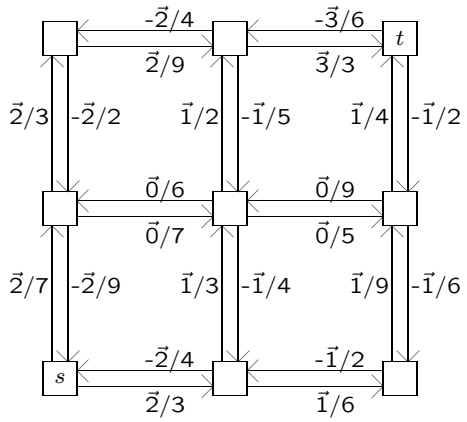
However it is fundamentally no more difficult than the concept of an overdraft at a bank being a negative balance.

If there are 3 units of flow moving *from A to B*, then we can equally view that as being -3 units of flow *from B to A*. This is the concept that is being captured by skew-symmetry.

If we increase the flow from A to B by one unit then the new flow will be 4 units from A to B (same as -4 units from B to A), whereas if we increase the flow from B to A by one unit then the new flow will be -2 units from B to A (same as 2 units from A to B).

The residual network

Consider the same flow, but this time also including the (original) capacities of the edges on the same diagram.

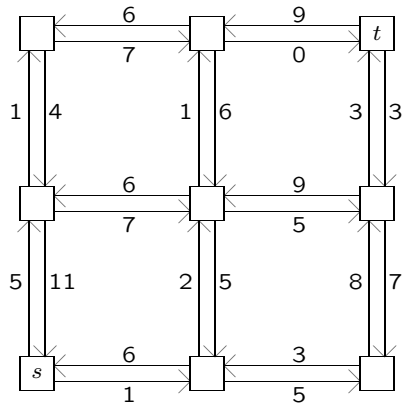


It is clear that some of the pipes have got some *residual capacity* in that they are not being fully used.

The residual network

The *residual network* is the network where we just list the "unused capacities" of the pipes. Given a capacity graph G and a flow f the residual network is called G_f where G_f has the same vertex set as G and capacities $c_f(u, v)$ given by

$$c_f(u, v) = c\,(u, v) - f(u, v)$$

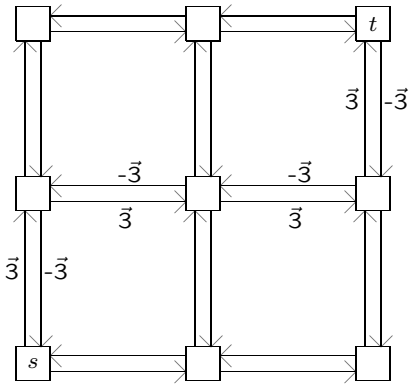


Augmenting flows

If we can find a flow f' in the residual network G_f, then we can form a new flow f^* in the original network G where

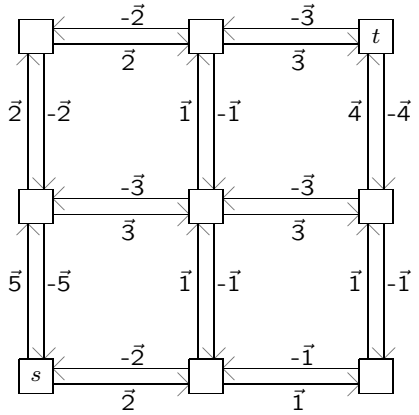
$$f^*(u,v) = f(u,v) + f'(u,v)$$

Such a flow in the residual network is called an *augmenting flow*.

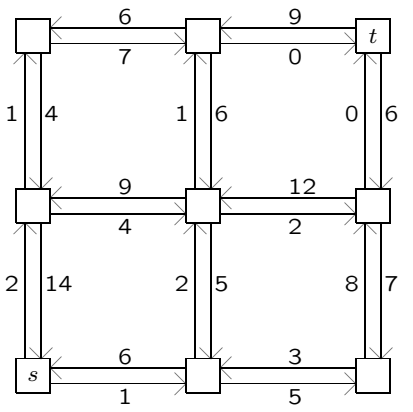


The total flow so far

Adding this flow to the original flow gives us a new flow with a higher value — the new flow has a value of 7.



The new residual network



We see that the pipes into t now have no unused capacity — they are currently *saturated*. Therefore we cannot increase this flow any further by finding an augmenting flow.

The question is: have we found the maximum flow, or did we go wrong at an earlier stage?

Ford-Fulkerson method

The Ford-Fulkerson method is an iterative method for solving the maximum flow problem.

It proceeds by starting with the zero valued flow (where $f(u,v) = 0$ for all u, $v \in V$).

At each stage in the method an *augmenting path* is found — that is, a path from s to t along which we may push some additional flow. Given an augmenting path the *bottleneck capacity* b is the smallest residual capacity of the edges along the path.

We can construct a flow of value b in the residual network by taking flows of b along the edges of the path, and zeros elsewhere.

This process continues until there are no augmenting paths left.

For a given flow network $G = (V, E)$, a source vertex s and a sink vertex t, Ford-Fulkerson method can be summarised as follows:

Ford-Fulkerson(G, s, t)

for each edge $(u, v) \in E$ **do**

$f(u, v) \leftarrow 0$

$f(v, u) \leftarrow 0$

while there exists a path p from s to t in the residual network G_f **do**

$c_f(p) \leftarrow \min\{c_f(u, v) : (u, v) \text{ is in } p\}$

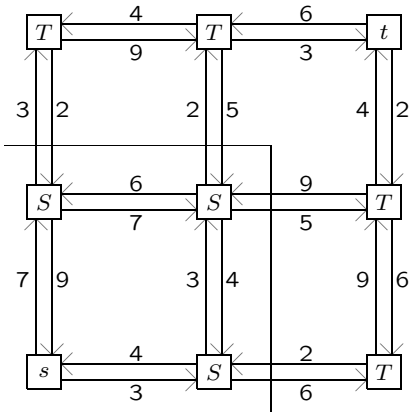
for each edge (u, v) in p **do**

$f(u, v) \leftarrow f(u, v) + c_f(p)$

$f(v, u) \leftarrow -f(u, v)$

Cuts

An s,t-cut is a partition of V into two subsets S and T such that $s \in S$ and $t \in T$.



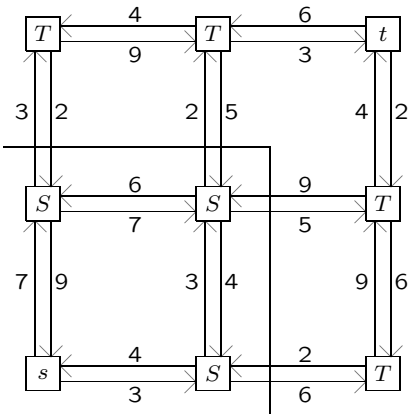
Here the vertices marked S are in S and the ones marked T are in T. The line draws the boundary between S and T.

The capacity of a cut

The *capacity* of a cut is the sum of the capacities of all of the edges that go between S and T.

More formally

$$c(S, T) = \sum_{u \in S, v \in T} c(u, v)$$

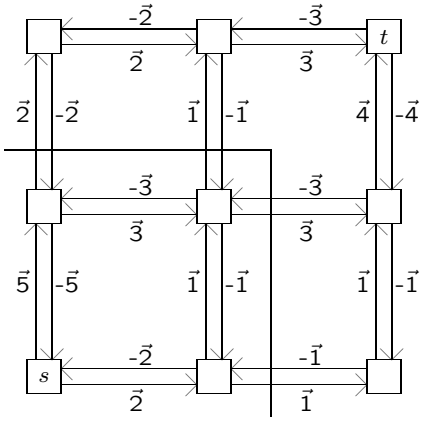


Therefore the capacity of this cut is

$$3 + 2 + 5 + 6 = 16$$

Flow across a cut

Now let us compute the *flow* across this cut when the network is carrying the flow of value 7 that we found earlier.

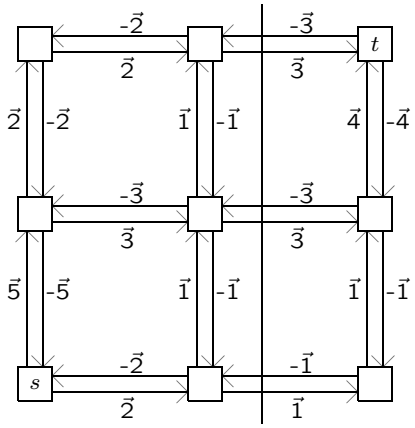


The flow across the cut is

$$2 + 1 + 3 + 1 = 7$$

Flow across another cut

Now let us compute the flow across a different cut.



The flow across this cut is

$$3 + 3 + 1 = 7$$

Flow across all cuts

Theorem. The flow across *every* cut has the same value.

In order to prove this theorem our strategy will be to show that moving a single vertex from one side of a cut to the other does not affect the flow across that cut.

This will then show that any two cuts have the same flow across them because we can shift any number of vertices from one side of the cut to the other without affecting the flow across the cut.

Proof. Suppose S, T is a cut such that $u \in S$. We show that we can move u to T without altering the flow across the cut by considering the value

$$f(S,T) - f(S - \{u\}, T + \{u\})$$

Proof continued

The contribution that vertex u makes to the flow $f(S,T)$ is

$$\sum_{w \in T} f(u,w)$$

whereas the contribution it makes to the flow $f(S - \{u\}, T + \{u\})$ is

$$\sum_{w \in S} f(w,u)$$

Therefore

$$\begin{aligned} & f(S,T) - f(S - \{u\}, T + \{u\}) \\ = & \sum_{w \in T} f(u,w) - \sum_{w \in S} f(w,u) \\ = & \sum_{w \in T} f(u,w) + \sum_{w \in S} f(u,w) \\ = & \sum_{w \in V} f(u,w) \\ = & \ 0 \end{aligned}$$

Minimum cut

For any s, t-cut S, T it is clear that

$$f(S,T) \leq c\,(S,T)$$

Therefore the value of the flow is at most the capacity of the cut. Therefore we can consider the cut with the lowest possible capacity — the *minimum cut*, and it is clear that the capacity of this cut is an upper bound for the maximum flow.

Therefore

$$\textit{max flow} \leq \textit{min cut}$$

Example

For our example, the cut $S = V - \{t\}$, $T = \{t\}$ has capacity 7, so the maximum flow has value at most 7. (As we have already found a flow of value 7 we can be sure that this is indeed the maximum).

Max-flow min-cut theorem

The important *max-flow min-cut theorem* tells us that the inequality of the previous slide is actually an equality.

Theorem

If f is a flow in the flow network G with source s and sink t, then the following three conditions are equivalent.

1. f is a maximum flow in G
2. The residual network G_f contains no augmenting paths from s to t
3. $|f| = c(S,T)$ for some s,t-cut S, T

The max-flow min-cut theorem is an instance of duality that is used in linear optimization

Justification

Condition (1) implies Condition (2)

If f is a maximum flow then clearly we cannot find any augmenting paths.

Condition (2) implies Condition (3)

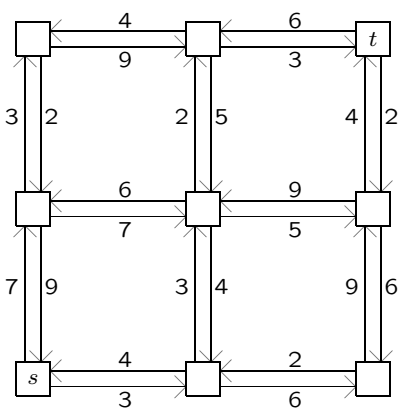
If the residual network contains no paths from s to t, then let S be all the vertices reachable from s, and let T be the remaining vertices. There are no edges in G_f from S to T and hence all of these edges are saturated and $f(S,T) = c(S,T)$.

Condition (3) implies Condition (1)

Obvious.

Finding the minimum cut

Let us find the minimum cut in our previous example.



Ford-Fulkerson is correct

The main significance of the max-flow min-cut theorem is that it tells us that if our current flow is not the maximum flow, then we are *guaranteed* that there will be an augmenting path.

This means that the Ford-Fulkerson method is always guaranteed to find a maximum flow, regardless of how we choose augmenting paths.

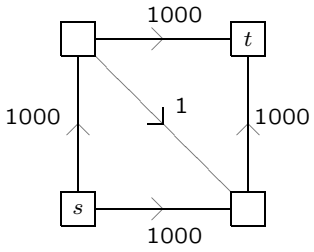
However it is possible to make unfortunate choices of augmenting paths in such a way that the algorithm may take an inordinate amount of time to finish, and indeed examples can be constructed so that if a bad augmenting path is chosen the algorithm will never finish.

Complexity of Ford-Fulkerson

If the capacities of the edges are all integers, then at each step the algorithm augments the flow by at least 1 unit. Finding each augmenting path can be done in time $O(E)$ (where E is the number of edges of the original network that have non-zero capacity).

Therefore the complexity is $O(E|f^*|)$ where $|f^*|$ is the value of the maximum flow.

It is easy to construct examples where it would actually take this long.



Improving the performance

Edmonds and Karp suggested two heuristics for improving the performance of the algorithm by choosing better augmenting paths.

Their first heuristic seems natural enough:

- Always augment by a path of maximum bottleneck capacity

The path of maximum bottleneck capacity can be found by a priority-first search — similar to Dijkstra's algorithm except that the priority is based on the bottleneck capacity of the path so far, rather than distances.

This can be implemented in time $O(E \ln |f^*|)$, but we will not consider the derivation of this bound.

The second heuristic

Edmonds and Karp's second heuristic produces an asymptotic complexity which is independent of the edge capacities.

- Always augment by a path with the fewest number of edges

Suppose we perform a breadth-first search on the residual network G to find the shortest path p from s to t. After augmenting the flow along p by the bottleneck capacity, consider the new residual network G'. The shortest path from s to t in G' must be at least as long as that in G, because any new edges in G' that are not in G must point back along the path p, and hence cannot contribute to a shorter path.

This shows that the lengths of the shortest paths found at each stage is always constant or increasing.

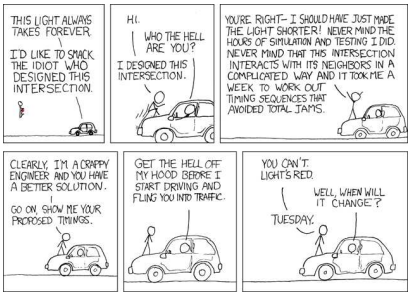
Analysis

We can view the Edmonds-Karp heuristic as operating in several "stages" where each stage deals with *all* the augmenting paths of a given length.

How many augmenting paths of a given length can there be?

There can be at most E augmenting paths of a given length, and if performed efficiently, each augmentation can be done in time E.

As there are at most V stages of the algorithm we get a total time of $O(VE^2)$.



Applications of network flow

One interesting application of network flow is to solve the *bipartite matching problem*.

A *matching* in a graph G is a set of edges that do not share any vertices.

The *maximum matching* problem is to find the largest possible matching in a graph.

Although the problem can be solved in polynomial time, the algorithm required to solve it in the general case (when G can be any graph) is horrendously complicated.

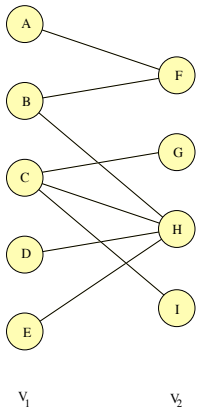
The situation is much simpler for a bipartite graph, where network flow can directly be used.

Matchings for bipartite graphs can be considered to be simple versions of the *stable marriage problem*.

Bipartite graph

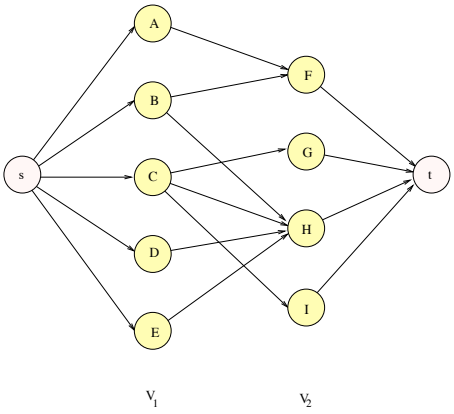
A bipartite graph is an undirected graph $G = (V, E)$ in which V can be partitioned into two sets V_1 and V_2 such that $(u, v) \in E$ implies either $u \in V_1$ and $v \in V_2$ or $u \in V_2$ and $v \in V_1$. That is, all edges go between the two sets V_1 and V_2.

Example of a bipartite graph



(Vertices in V_1 may represent a number of machines and vertices in V_2 may represent a number of tasks.)

Finding a maximum bipartite matching using Ford-Fulkerson method



(Assume all edge capacities are 1).

Recommended reading: CLRS, Chapter 26

The stable marriage problem

The *stable marriage problem* is not a network flow problem, but it is a matching problem. The scenario is as follows:

We are given two sets, V_M and V_F (male and female) of the same size. Also, every man $v \in V_m$ ranks every woman $u \in V_F$ and every woman $u \in V_F$ ranks every man in V_M.

We will write $u <_v u'$ if v would rather marry u than u'.

The stable marriage problem is to find a matching $E \subset V_M \times V_F$ such that if (v, u) and (w, z) are in E, then either $u <_v z$ or $w <_z v$.

Solution

The Gale-Shapley algorithm is a solution to the stable marriage problem that involves a number of rounds. Heuristically, each round every "unengaged" man proposes to his most preferred woman that he has not already proposed to. If this woman is unengaged or engaged to someone she prefers less than her new suitor, she breaks off her current engagement and accepts the new proposal.

These iterations continue until everybody is engaged. It can be shown that at this point we have a solution to the stable marriage problem.

Pseudo-Code

Let P be a listing of everybody's preferences.

procedure StableMarriage(V_M, V_F, P)
  $E \leftarrow \emptyset$
  while $|E| < n$
    for each $v \in V_M$ where $\forall u,\ (v,u) \notin E$
      $u \leftarrow v$'s next preference
      if $(w,u) \in E$ and $v <_u w$
        $E \leftarrow E - \{(w,u)\} \cup \{(v,u)\}$
      else if $\forall w,\ (w,u) \notin E$
        $E \leftarrow E \cup \{(v,u)\}$
  return E

Summary

1. A flow network is a directed, weighted graph with a *source* and a *sink*
2. A flow assigns a real value to each edge in the flow network and satisfies the capacity constraint, skew symmetry and flow conservation.
3. The Ford Fulkerson method solves the maximum flow problem by iteratively search for augmenting paths in the residual flow network.
4. A cut is a set of edges that divides the source form the sink.

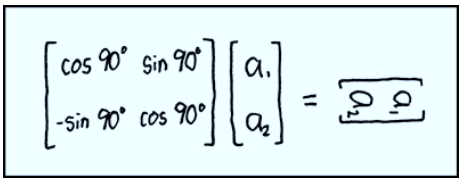
Summary cont.

5. the Max-flow min-cut theorem states that the maximum flow for the source to the sink is equal to the minimum flow across all cuts.
6. Edmonds' and Karp's Heuristics improve the performance of the Ford-Fulkerson method.
7. Flow networks can be used to solve some simple matching problems in Bipartite graphs.
8. The Gale-Shapely algorithm can be used to solve the stable marriage matching problem.

Computer Science and Software Engineering, 2011

CITS3210 Algorithms

Computational Geometry

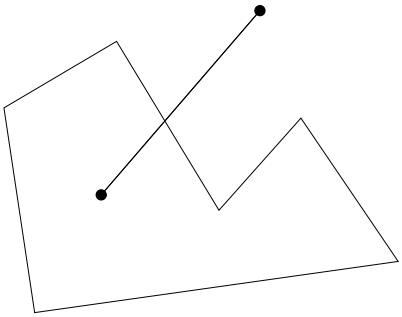


Notes by CSSE, Comics by xkcd.com

Computational Geometry

Computational Geometry is the study of algorithms for solving geometric problems. This has applications in many areas such as graphics, robotic, molecular modelling, forestry, statistics, meteorology,... basically any field that stores data as a set of points in space.

In this topic we will restrict our attention to geometric problems in the 2 dimensional plane.

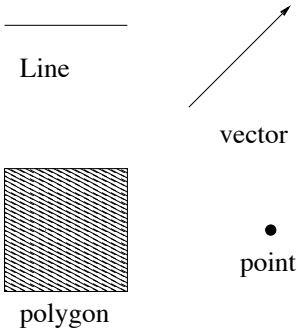


Geometric Objects

The types of objects we will use are:

- Points: a point (x, y) is represented using two Cartesian coordinates: $x, y \in R$. The origin $(0, 0)$ is point.
- Lines: line $\overline{p} = (p1, p2)$ is represented by two points $p1 \neq p2$ which represent the two end points of a line. Sometimes we may represent a line using only one point, with the understanding that second point is the origin.
- Vectors: A vector $\vec{v} = (v_1, v_2)$ is a direction. A vector v and a point (x, y) may represent the (directed) line segment $(x, y), (x + v_1, y + v_2)$.
- Polygons: A polygon is a closed shape made up of straight edges $(e_1, e_2, ..., e_k)$ where e_i is a line for $i = 1, ..., k$ and the start point of e_1 is the end point of e_k.

Geometric Objects

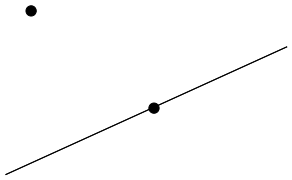


Geometric Problems

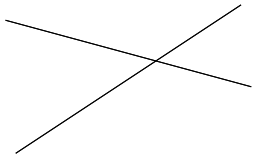
Given these basic structures we may be interested in:

- whether a point is on a line?
- how close is a point to a line?
- whether two lines intersect?
- where do two lines intersect?
- whether a point is inside a polygon?
- what is the area of a polygon?
- what is the smallest polygon that contains a number of points?

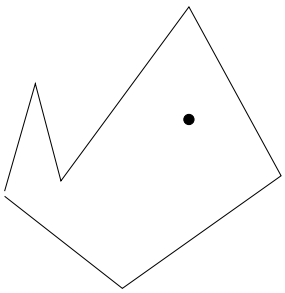
Is this point on a line?



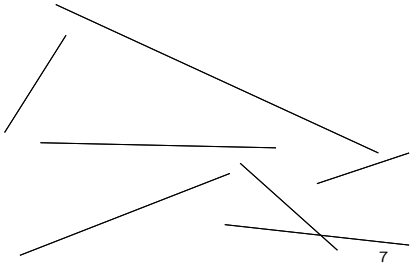
Do these two lines intersect?



Is this point inside the polygon?



Do any of these lines intersect?



The problem with geometry

The difficulty with geometric algorithms is that the "Human" approach is so different to the algorithmic approach. As humans we are able to represent the geometric objects in a 2 dimensional plane, and a single glance at that plane is enough to determine whether or not a point is on a line, or in a polygon.

However there is no algorithmic variation of a "glance", so we are required to look for a mathematical approach to solving these problems.

It turns out that surprisingly little mathematics is required for the algorithms we will be using, although some basic linear algebra is assumed. (You should know how to add and subtract points in the plane, perform scalar multiplication, calculate Euclidean distances and solve systems of linear equations).

The Cross-Product

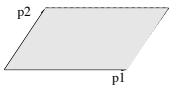
In fact the geometric mathematics in the problems we have mentioned can generally be reduced to the question:

> *With respect to point A, is point B to the left or right of point C?*

Suppose $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ are two vectors (i.e. lines that start at the origin and end at the given point). The cross product of these two vectors is the (3D) vector that is perpendicular to both vectors, and has magnitude

$$p_1 \times p_2 = x_1 y_2 - x_2 y_1$$

which in turn is equal to the *signed* area of the parallelogram with edges p_1 and p_2.



Simplifying the cross product

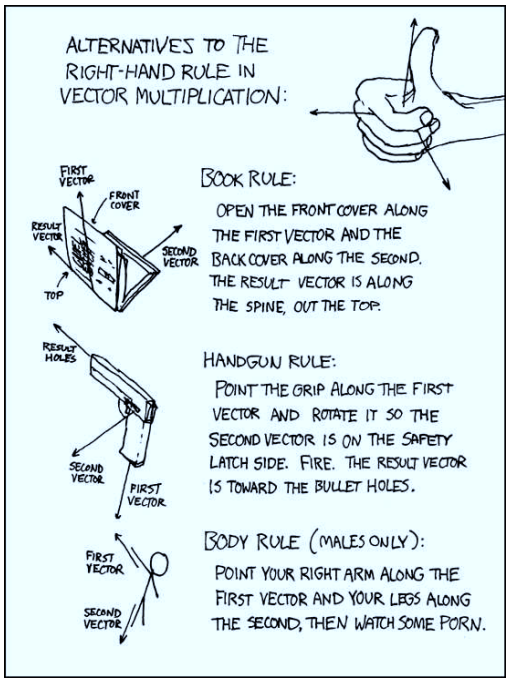
The mathematical definition and properties of the cross product are all very interesting, but the only thing we need to worry about is the sign of $p1 \times p_2$: if it is positive, then p_1 is to the right of p_2; if it is 0, p_1 and p_2 are on the same line; and if it is negative p_1 is to the left of p_2 (all with respect to the origin).

To this end, let's define a function for the direction of an angle

$$\text{Dir}(p_0,p_1,\ p_2) = (p_1 - p_0) \times (p_2 - p_0)$$

The only thing you need to be careful of is that you get the order of the points right.

The right hand rule

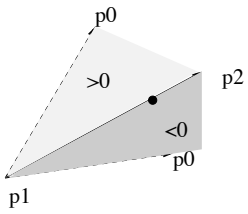


(Which version of the rule you use isn't important as long as it is applied consistently).

Using the Cross-Product

The first question we can answer is whether a point, p_0, is on a line, (p_1, p_2).

procedure OnLine(p_0, p_1, p_2)
  if Dir(p_0, p_1, p_2) = 0
    if $p_1.x \leq p_0.x \leq p_2.x$
      return true
    else if $p_2.x \leq p_0.x \leq p_1.x$
      return true
    else return false
  else return false



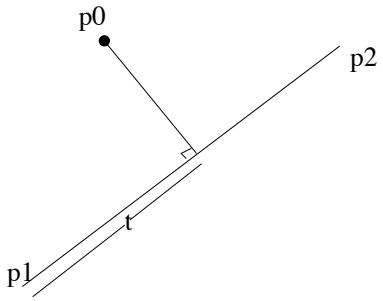
You should always check the specification carefully to see if the end-points of a segment should be included. There are similar boundary conditions for most of the following algorithms.

Closest point on a line

A more general version of the problem above is what is the closest point on a line segment to a given point. This can be solved in a number of ways, including binary search, or using trigonometry. The following solution makes use of some very basic calculus to determine the closest point on the line (p_1, p_2) to the point p_0.

procedure ClosestPoint(p_0, p_1, p_2)
  $a \leftarrow p_2.x - p_1.x$, $b \leftarrow p_2.y - p_1.y$
  $t \leftarrow (a(p_0.x - p_1.x) + b(p_0.y - p_1.y))/(a^2 + b^2)$
  if $t > 1$
    return $p2$
  else if $t < 0$
    return $p1$
  else
    return $((p_1.x + ta), (p_1.y + tb)$

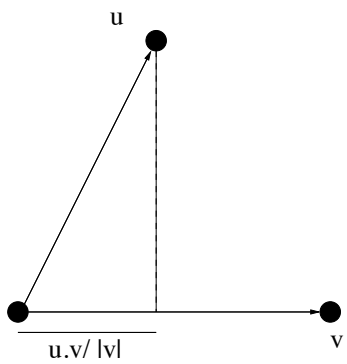
Diagram



Dot products

A related concept is the *dot product* of two vectors $v = (x_1, y_1)$ and $u = (x_2, y_2)$, which is calculated as $x_1x_2 + y_1y_2$.

The dot product is the length of the vector v when it is projected orthogonally onto the vector u multiplied by the length of u (or vice versa).



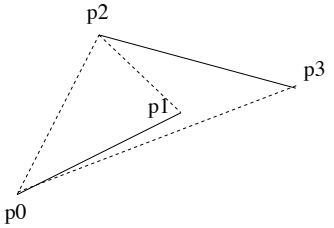
It is useful to know, but be careful not to confuse it with the cross product.

Intersecting Lines

The next question we can answer is whether two lines, (p_0, p_1) and (p_2, p_3) intersect. (To simplify the code we will suppose intersect means the lines cross properly, rather than touch at an end-point, and the endpoints of each line are sorted lexicographically).

procedure Crosses(p_0, p_1, p_2, p_3)
  $d \leftarrow$ Dir$(p_0, p_1, p_2)\times$ Dir(p_0, p_1, p_3)
  if $d < 0$
    return true
  else return false

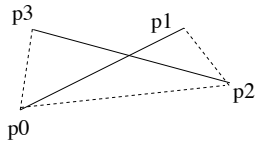
Crosses is true if both p_2 and p_3 are not both on the same side of (p_0, p_1). However this is not enough to decide whether the lines intersect.



Intersecting Line Segments

Crosses($\overline{p}, \overline{q}$) actually returns if the line segment $\overline{q}$ intersects the infinite extension of $\overline{p}$. We can reuse this method the see if the two segments intersect.

procedure Intersects(p_0, p_1, p_2, p_3)
  if Crosses(p_0, p_1, p_2, p_3)
    if Crosses(p_2, p_3, p_0, p_1)
      return true
    else return false
  else return false



Any Segments Intersect

The methods we have seen so far barely deserve to be called algorithms. They are simply achieved through basic arithmetic and comparisons.

However, as with all things computer, once we know how to do one thing well we want to know if we can do it well many times all at once....

Now suppose we are given a set of lines, and we would like to know whether any two lines in the set intersect.

Here we adopt something akin to a human approach, where we scan all of the points making up to line segments, inspecting each to see if there is an intersection.

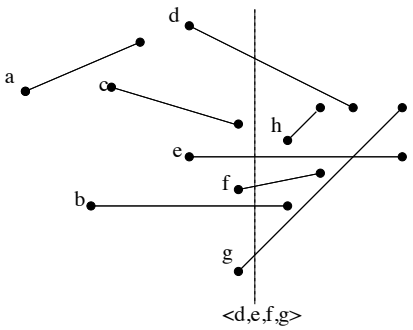
The Sweep Line

To process all the line segments efficiently we imagine a *sweep line* passing moving from left to right.

Every time it the start or the end of a line segment it triggers an *event*, where we must check to see if an intersection has occured.

As the sweep line progresses from left to right, we are required to maintain an ordered list of the line segments that cross the sweep line, ordered from top to bottom (with respect to their first point).

The sweep line



Events

There are two types of events we must consider: when we encounter the first point of a line segment, and when we encounter the second point of a line segment:

- When we encounter the first point of a line segment, we insert the line segment into our ordered list of line segments. When we do this we should check to see if the new line segment intersects the line segment directly above it or below it in the list.
- When we encouter the end point of a line segment, we remove the segment from the ordered list. This will cause the line segment above and below the removed segment to become adjacent, so we must check if they intersect.

Pseudo-code

Suppose that $S = \{s_1, s_2, ...s_n\}$ is a set line segments, where $s_i = (p_i, q_i)$, and let T be an ordered list.

Sort the set of points $\{p_i, q_i \mid s_i \in S\}$
for each point r in the sorted list
  do if $r = p_i$
    then INSERT(T,s_i)
    if ABOVE(T,s_i) exists and intersects s_i
      then return true
    if BELOW(T,s_i) exists and intersects s_i
      then return true
  else if $r = q_i$
    then if ABOVE(T,s_i) and BELOW(T,s_i)
      then return true
    DELETE(T,s_i)
return false

Correctness

To see the correctness of this algorithm we need to show the following three statements:

1. If two line segments s_i and s_j intersect, there is some event point x, such that at x s_i is next to s_j in the list (say s_i and s_j are neighbours).
2. In between event points, the only way new neighbours may arise is if already neigbouring line segments intersect.
3. At an event point, the only way new neighbours may arise is by adding a line segment (creating up to two new pairs of neighbours), or by removing a line segment (creating up to one new pair of neighbours).

Other considerations?

From the previous statements we are able to conclude the algorithm is correct. However:

- What if lines start and end at the same sweep line. Does the proof still work?
- What simplifying assumption does this algorithm make?
- How would you change this algorithm if you actually needed to return all intersections?

Complexity

Assuming there are N line segments there are $2N$ event-points to deal with.

- These points must be sorted (using, say heapsort). This takes $O(N \lg N)$ time.
- Then, for each point we need to either:
  1. insert the corresponding segment into an ordered list (finding the segments above and below) $O(lgN)$, and calculate whether the segments intersect $O(1)$; or
  2. find the segments above and below ($O(\lg N)$), calculate whether they intersect ($O(1)$), and delete the segment from the ordered list ($O(\lg N)$).

This gives overall performance of $O(N \lg N)$.

Note that the insertions, deletions and find operations on the ordered list require an efficient implementation, such as a Red-Black tree (`java.util.TreeMap`) or an AVL tree.

Convex Hull

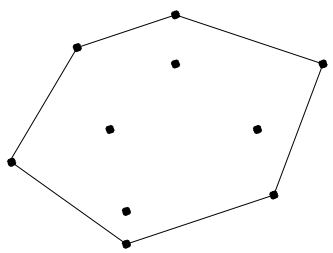
The problem of the convex hull is to find the smallest convex polygon that contains a set of points.

> A shape is *convex* if any line segment between two points inside the shape remains inside the shape.

The convex hull algorithm we will examine is known as Graham's scan. Like the all-segments intersection algorithm, it is based on a sweep, but this time it is a roational sweep, rather than a linear sweep.

The algorithm is based on the fact that you can walk clockwise round the perimeter of a convex hull, and every point of the set will *always* on your right.

The Convex Hull



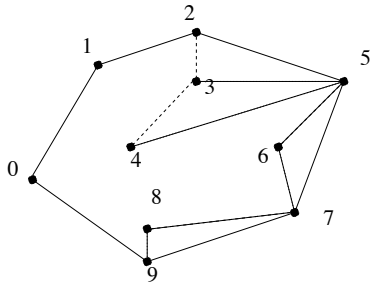
Graham's Scan

Graham's scan starts with the (bottom) left-most point of the set, and processes the points in order from left to right (with resepct to that point). The each point is examined and if there is known point to its left (with respect to any point examined so far) it is a candidate for the convex hull. These condidate points are kept on a stack. As every new point is examined, either:

1. there is only one point already in the stack in which case the new point is added.
2. it is to the right of the line segment through the top two points on the stack, in which case it is pushed on to the stack; or
3. it is to the left of the line segment through the top two points on the stack, in which case the top element of the stack is popped off, and we repeat the test.

After every point has been processed, the remaining points on the stack form the convex hull.

Graham's Scan



Pseudo-code

Let P be a set of points $\{p_1, ..., p_n\}$ and S be a stack.

procedure Graham-Scan(P)
  Find the left-most point p_0 in P
  Sort the set of points $P - \{p_0\}$
    according to their angle around p_0
  for each point p in the sorted list
    do if $|S| = 1$
      then PUSH(S,p)
    else $q_1 \leftarrow$ POP(S), $q_0 \leftarrow$ POP(S)
      while DIR(p_0,p_1, p) ¡0
        do $q_1 \leftarrow q_0$, $q_0 \leftarrow$ POP(S)
      PUSH(S, q_0), PUSH(S,q_1), PUSH(S,q)
  RETURN S.

Correctness (sketch)

The algorithm returns a stack of points. If we list these points in order (wrapping back to the start vertex) we get the edges of a polygon.

We now must show two things:

1. The polygon is convex: This follows from the fact that the algorithm ensures that each corner of the polygon turns right (for a clockwise direction).

2. The polygon contains every point in P: Every point p is added to S, and points are only removed if we find an edge p_1p_2 such that the triangle $p_0p_1p_2$ contains p. As p_0, p_1, p_2 will then appear in the stack, p will be contained in the polygon.

Complexity

We can find the left-most point in time $O(n)$.

We can sort the points according to their angle around p_0 in time $O(n \lg n)$. Note we do not have to calculate the angle to do this. We can just do a normal sort, but instead of using $<$ for comparison, we can use the DIR function.

The algorithm then has two nest loops each potentially going through n iterations. However, we may note that the inner loop is popping elements of the stack. As each element is added to the stack exactly once, this operation cannot be performed more than n times.

Therefore the total complexity is $O(n \lg n)$

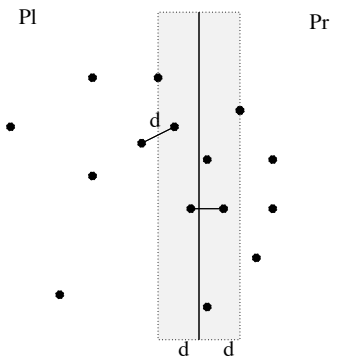
Closest pair of points

Finding the closest pair of points in a set of points is another useful algorithm that demonstrates a useful technique for computational geometry - *divide and conquer*.

The problem assumes we are given a set of points P, and we are required to return the two points tat are closest to one another with respect to Euclidean distance.

The idea of the algorithm is to split the set of points P into two halves, P_L and P_R. We can then recursively solve the problem for the two halves and then combine the solution (take the minimum pairing). But what is there is one point in P_L and one point in P_R that are closer to one another? We need a way to quickly check if there are any closer pairs crossing the partition.

Closest pair of points



Checking Points Across Partitions

If we have solved the Closest Pair of Points problem for P_L and P_R then we know the minimum distance betwwen any pair of points in eitehr partition. Let this distance be δ. Therefore, we only need to check if points within a δ-width strip on either side of the divide are closer.

Furthermore, we know all the points in either side of the 2δ-width strip must be at least a distance of δ from one another. We can use this fact to show that each point in the strip only needs to be compared to the 5 subsequent points (ordered from top to bottom).

Pseudo-code

Let $P = \{p_1, ..., p_n\}$ be a set of points. We will just give the method for finding the closest distance. We will assume that P is sorted by x-coordinate, and also that we have access to a copy of P, Q, that is sorted by y-coordinate (with an inverse index).

procedure ClosestPair(P)
  Split P into P_L (the $n/2$ leftmost points)
    and P_R (the other points)
  $\delta = \min\{\text{ClosestPair}(P_L),\ \text{ClosestPair}(P_R)\}$
  For each point p in P
    if p^x is within δ of $p^x_{n/2}$
      then add p to A
  For each point q_i in A in order of q_i^y
    For $j = 1, ..., 5$
      $\delta = \min\{\delta, DIST(q_i, q_{i+j}\}$
  RETURN δ

Analysis

The divide and conquer strategy is easily seen to be valid, however the algorithm descibed above uses a number of optimizations, such as presorting and examining a relatively small set of pairs of points. See CLRS chapter 33 for a justification of these optimizations.

The complexity of $O(n \lg n)$ can be shown using the recurrence:

$$T(n) = 2T(n/2) + O(n)$$

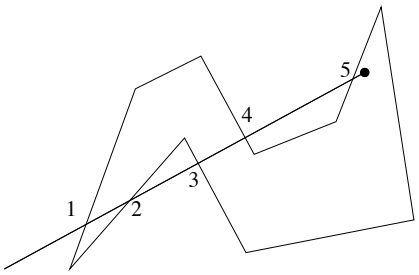
(see for example merge-sort). What optimizations do we require to ensure that the divide and merge can be performed in time $O(n)$?

Point inside a polygon

Suppose we are given a polygon (as a set of points $P = (p_0, p_1, ...p_n)$ where $p_0 = p_n$) and we are required to determine whether a point q is inside the polygon or not.

This algorithm is relatively simple. We take a line segment starting at q and ending at a point we know to be outside the polygon. Then we count the number of times the line segment crosses an edge of the polygon. As every time it crosses an edge, the line segment goes from insdie the polygon to outside, or from outside the polygon to inside. Therefore if there are an odd number of such crossings, the pointis inside, otherwise it is outside.

Ray casting



Pseudo-code

Given a simple polygon $P = (p_0, p_1, ..., p_n)$ where $p_0 = p_n$ and a point q we determine whether q is in P as follows:

procedure Pt-In-Poly(P, q)
  Find a point r outside the polygon
    e.g. $((\min X) - 1, 0)$
  set $c \leftarrow 0$
  for $i = 1$ to n
    do if Intersects(r, q, p_{i-1}, p_i)
      then $c \leftarrow c + 1$
  if c is even **return** false
  else return true.

You should be careful how the *Intersect* methods treats lines touching at a point, or interval.The intersects method provided in these notes requires that the lines properly cross, but it still requires a careful treatment in the rare case that the ray passes through the corner of a polygon appropriate in this context.

This special case, along with correctness and completeness are left as excercises.

Area of a polygon

We will assume that we are given a polygon P that is not self intersecting, and furthermore we shall suppose that the polygon is represented by a sequence of points $\{p_0, p_1, ..., p_n\}$ where (p_i, p_{i+1}) are the edges in clockwise order, and $p_0 = p_n$.

The area of a polygon can be computed by breaking it up into a series of triangles. The area may then be computed by calculated by computing the sum:

$$\frac{1}{2}\Sigma_0^n(x_{i+1}y_i - x_i y_{i+1})$$

The derivation follows from the characterisation of the cross-product as the area of the parallelogram created by the adjacent vectors.

A final note

We can see computational geometry has as much in common with graph algorithms as it does with maths. However, a main point of difference is to number of boundary cases to consider:

Always check:

1. Can you rearrange a sum to reduce round off error?
2. Do you really need to do (real valued) division?
3. Have you considered all boundary cases: points touching, co-linear points, 0-area polygons?
4. What degree of accuracy is required?

Summary

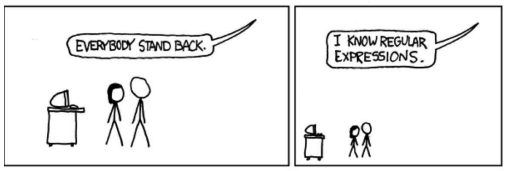
We have examined:

- 2D Geometric objects.
- Cross-products, dot products and the right hand rule.
- Closest point to a line, intersecting line segments.
- Sweep lines:
  - All intersecting segments.
  - Convex hull.
- Divide and Conquer: closest pair of points.
- Ray casting: Point inside a polygon.
- Useful functions: Area of a polygon

Computer Science and Software Engineering, 2011

CITS3210 Algorithms

String and File Processing



Notes by CSSE, Comics by xkcd.com

Overview

In this topic we will look at pattern-matching algorithms for strings. Particularly, we will look at the Rabin-Karp algorithm, the Knuth-Morris-Pratt algorithm, and the Boyer-Moore algorithm.

We will also consider a dynamic programming solution to the *Longest Common Substring* problem.

Finally we will examine some file compression algorithms, including Huffman coding, and the Ziv Lempel algorithms.

Pattern Matching

We consider the following problem. Suppose T is a string of length n over a finite alphabet Σ, and that P is a string of length m over Σ.

The *pattern-matching* problem is to find occurrences of P within T. Analysis of the problem varies according to whether we are searching for all occurrences of P or just the first occurrence of P.

For example, suppose that we have $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ and

$T =$ `abaaabacccaabbaccaababacaababaac`
$P =$ `aab`

Our aim is to find all the substrings of the text that are equal to `aab`.

Matches

String-matching clearly has many important applications — text editing programs being only the most obvious of these. Other applications include searching for patterns in DNA sequences or searching for particular patterns in bit-mapped images.

We can describe a *match* by giving the number of characters s that the pattern must be shifted along the text in order for every character in the shifted pattern match the corresponding text characters. We call this number a *valid shift*.

```
abaaabacccaabbaccaababacaababaac
aab
 aab
  aab
   aab
    aab
```

Here we see that $s = 3$ is a valid shift.

The naive method

The naive pattern matcher simply considers every possible shift s in turn, using a simple loop to check if the shift is valid.

When $s = 0$ we have

```
abaaabacccaabbaccaababacaababaac
aab
```

which fails at the second character of the pattern.

When $s = 1$ we have

```
abaaabacccaabbaccaababacaababaac
 aab
```

which fails at the first character of the pattern.

Eventually this will succeed when $s = 3$.

Analysis of the naive method

In the worst case, we might have to examine each of the m characters of the pattern at every candidate shift.

The number of possible shifts is $n - m + 1$ so the worst case takes

$$m(n - m + 1)$$

comparisons.

The naive string matcher is inefficient because when it checks the shift s it makes no use of any information that might have been found earlier (when checking previous shifts).

For example if we have

```
000000001000001000000010000000
000000001
```

then it is clear that no shift $s \leq 9$ can possibly work.

Rabin-Karp algorithm

The naive algorithm basically consists of two nested loops — the outermost loop runs through all the $n - m + 1$ possible shifts, and for each such shift the innermost loop runs through the m characters seeing if they match.

Rabin and Karp propose a modified algorithm that tries to replace the innermost loop with a single comparison as often as possible.

Consider the following example, with alphabet being decimal digits.

```
1229384912817608213081762831 01
176
```

Suppose now that we have computer words that can store decimal numbers of size less than 1000 in one word (and hence *compare* such numbers in one operation).

Then we can view the entire pattern as a single decimal number and the substrings of the text of length m as single numbers.

Rabin-Karp continued

Thus to try the shift $s = 0$, instead of comparing

$$1 - 7 - 6$$

against

$$1 - 2 - 2$$

character by character, we simply do one operation comparing 176 against 122.

It takes time $O(m)$ to compute the value 176 from the string of characters in the pattern P.

However it is possible to compute *all* the $n - m + 1$ decimal values from the text just in time $O(n)$, because it takes a constant number of operations to get the "next" value from the previous.

To go from 122 to 229 only requires dropping the 1, multiplying by 10 and adding the 9.

Rabin-Karp formalized

Being a bit more formal, let $P[1..m]$ be an array holding the pattern and $T[1..n]$ be an array holding the text.

We define the values

$$p = P[m] + 10P[m-1] + \cdots + 10^{m-1}P[1]$$

$$t_s = T[s+m] + 10T[s+m-1] + \cdots + 10^{m-1}T[s+1]$$

Then clearly the pattern matches the text with shift s if and only if $t_s = p$.

The value t_{s+1} can be calculated from t_s easily by the operation

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1]$$

If the alphabet is not decimal, but in fact has size d, then we can simply regard the values as $d-$ary integers and proceed as before.

But what if the pattern is long?

This algorithm works well, but under the unreasonable restriction that m is sufficiently small that the values p and $\{t_s \mid 0 \le s \le n-m\}$ all fit into a single word.

To make this algorithm practical Rabin and Karp suggested using one-word values *related* to p and t_s and comparing these instead. They suggested using the values

$$p' = p \bmod q$$

and

$$t'_s = t_s \bmod q$$

where q is some large prime number but still sufficiently small that dq fits into one word.

Again it is easy to see that t'_{s+1} can be computed from t'_s in constant time.

The whole algorithm

If $t'_s \neq p'$ then the shift s is definitely not valid, and can thus be rejected with only one comparison. If $t'_s = p'$ then either $t_s = p$ and the shift s is valid, or $t_s \neq p$ and we have a *spurious hit*.

The entire algorithm is thus:

Compute p' and t'_0
for $s \leftarrow 0$ **to** $n-m$ **do**
  if $p' = t'_s$ **then**
    if $T[s+1..s+m] = P[1..m]$ **then**
      output "shift s is valid"
    end if
  end if
  Compute t'_{s+1} from t'_s
end for

The worst case time complexity is the same as for the naive algorithm, but in practice where there are few matches, the algorithm runs quickly.

Example

Suppose we have the following text and pattern

5 4 1 4 2 1 3 5 6 2 1 4 1 4
4 1 4

Suppose we use the modulus $q = 13$, then $p' = 414 \bmod 13 = 11$.

What are the values t'_0, t'_1, etc associated with the text?

$\underbrace{5\ 4\ 1}$ 4 2 1 3 5 6 2 1 4 1 4
$t'_0 = 8$

5 $\underbrace{4\ 1\ 4}$ 2 1 3 5 6 2 1 4 1 4
$t'_1 = 11$

This is a genuine hit, so $s = 1$ is a valid shift.

5 4 $\underbrace{1\ 4\ 2}$ 1 3 5 6 2 1 4 1 4
$t'_2 = 12$

We get one spurious hit in this search:

5 4 1 4 2 1 3 5 6 2 $\underbrace{1\ 4\ 1}$ 4
$t'_{10} = 11$

Finite automata

Recall that a *finite automaton* M is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$ where

- Q is a finite set of *states*
- $q_0 \in Q$ is the *start state*
- $A \subseteq Q$ is a distinguished set of *accepting states*
- Σ is a finite input alphabet
- $\delta : Q \times \Sigma \to Q$ is a function called the *transition function*

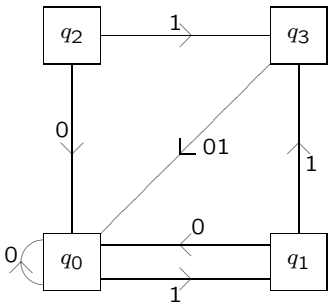
Initially the finite automaton is in the start state q_0. It reads characters from the input string x one at a time, and changes states according to the transition function. Whenever the current state is in A, the set of accepting states, we say that M has accepted the string read so far.

A finite automaton

Consider the following 4-state automaton:

$Q = \{q_0, q_1, q_2, q_3\}$
$A = \{q_3\}$
$\Sigma = \{0, 1\}$
δ is given by the following table

q	0	1
q_0	q_0	q_1
q_1	q_0	q_3
q_2	q_0	q_3
q_3	q_0	q_0



A string matching automaton

We shall devise a string matching automaton such that M accepts any string that ends with the pattern P. Then we can run the text T through the automaton, recording every time the machine enters an accepting state, thereby determining every occurrence of P within T.

To see how we should devise the string matching automaton, let us consider the naive algorithm at some stage of its operation, when trying to find the pattern `abbabaa`.

```
a b b a b b a b a a
a b b a b a a
```

Suppose we are maintaining a counter indicating how many pattern characters have matched so far — this shift s fails at the 6th character. Although the naive algorithm would suggest trying the shift $s + 1$ we should really try the shift $s + 3$ next.

```
a b b a b b a b a a
      a b b a b a a
```


Skipping invalid shifts

The reason that we can immediately eliminate the shift $s + 1$ is that we have already examined the following characters (while trying the shift s)

```
a b b a b b a b a a
```

and it is immediate that the pattern does not start like this, and hence this shift is invalid.

To determine the smallest shift that is consistent with the characters examined so far we need to know the answer to the question:

"What is the longest suffix of this string that is also a prefix of the pattern P?"

In this instance we see that the last 3 characters of this string match the first 3 of the pattern, so the next feasible shift is $s + 6 - 3 = s + 3$.

The states

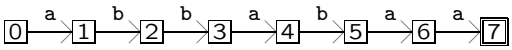
For a pattern P of length m we devise a string matching automaton as follows:

The states will be

$$Q = \{0, 1, \ldots, m\}$$

where the state i corresponds to P_i, the leading substring of P of length i.

The start state $q_0 = 0$ and the only accepting state is m.



This is only a partially specified automaton, but it is clear that it will accept the pattern P.

We will specify the remainder of the automaton so that it is in state i if the last i characters read match the first i characters of the pattern.

The transition function

Now suppose, for example, that the automaton is given the string

`a b b a b b` $\cdots$

The first five characters match the pattern, so the automaton moves from state 0, to 1, to 2, to 3, to 4 and then 5. After receiving the sixth character `b` which does not match the pattern, what state should the automaton enter?

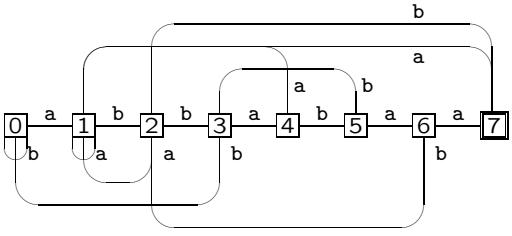
As we observed earlier, the longest suffix of this string that is a prefix of the pattern `abbabaa` has length 3, so we should move to state 3, indicating that only the last 3 characters read match the beginning of the pattern.

The entire automaton

We can express this more formally:

If the machine is in state q and receives a character c, then the next state should be q' where q' is the largest number such that $P_{q'}$ is a suffix of $P_q c$.

Applying this rule we get the following finite state automaton to match the string `abbabaa`.



By convention here all the horizontal edges are pointing to the right, while all the curved line segments are pointing to the left.

Using the automaton

The automaton has the following transition function:

q	a	b
0	1	0
1	1	2
2	1	3
3	4	0
4	1	5
5	6	3
6	7	2
7	1	2

Use it on the following string

`ababbbabbabaabbabaabaaababababbabaabbabbaa`

Character	a	b	a	b	b	b	a	b	b	a	b
Old state	0	1	2	1	2	3	0	1	2	3	4
New state	1	2	1	2	3	0	1	2	3	4	5

Compressing this information:

`ababbbabbabaabbabaabaaababababbabaabbabbaa`
`1212301234567234567211121212345672345341`

Analysis and implementation

Given a pattern P we must first compute the transition function. Once this is computed the time taken to find all occurrences of the pattern in a text of length n is just $O(n)$ — each character is examined precisely once, and no "backing-up" in the text is required. This makes it particularly suitable when the text must be read in from disk or tape and cannot be totally stored in an array.

The time taken to compute the transition function depends on the size of the alphabet, but can be reduced to $O(m|\Sigma|)$, by a clever implementation.

Therefore the total time taken by the program is $O(n + m|\Sigma|)$

Recommended reading: CLRS, Chapter 32, pages 906–922

Regular expressions

Knuth-Morris-Pratt

The Knuth-Morris-Pratt algorithm is a variation on the string matching automaton that works in a very similar fashion, but eliminates the need to compute the entire transition function.

In the string matching automaton, for any state the transition function gives $|\Sigma|$ possible destinations—one for each of the $|\Sigma|$ possible characters that may be read next.

The KMP algorithm replaces this by just two possible destinations — depending only on whether the next character matches the pattern or does not match the pattern.

As we already know that the action for a matching character is to move from state q to $q + 1$, we only need to store the state changes required for a non-matching character. This takes just one array of length m, and we shall see that it can be computed in time $O(m)$.

The prefix function

Let us return to our example where we are matching the pattern `abbabaa`.

Suppose as before that we are matching this against some text and that we detect a mismatch on the sixth character.

```
a b b a b x y z
a b b a b a a
```

In the string-matching automaton we used information about what the actual value of `x` was, and moved to the appropriate state.

In KMP we do exactly the same thing except that we do not use the information about the value of `x` — except that it does not match the pattern. So in this case we simply consider how far along the pattern we could be after reading `abbab` — in this case if we are not at position 5 the next best option is that we are at position 2.

The KMP algorithm

The *prefix function* π then depends entirely on the pattern and is defined as follows: $\pi(q)$ is the largest $k < q$ such that P_k is a suffix of P_q.

The KMP algorithm then proceeds simply:

$q \leftarrow 0$
for i **from** 1 **to** n **do**
    while $q > 0$ and $T[i] \neq P[q+1]$
        $q \leftarrow \pi(q)$
    end while
    if $P[q+1] = T[i]$ **then**
        $q \leftarrow q + 1$
    end if
    if $q = m$ **then**
        output "shift of $i - m$ is valid"
        $q \leftarrow \pi(q)$
    end if
end for

This algorithm has nested loops. Why is it linear rather than quadratic?

Heuristics

Although the KMP algorithm is asymptotically linear, and hence best possible, there are certain heuristics which in some commonly occurring cases allow us to do better.

These heuristics are particularly effective when the alphabet is quite large and the pattern quite long, because they enable us to avoid even looking at many text characters.

The two heuristics are called the *bad character* heuristic and the *good suffix* heuristic.

The algorithm that incorporates these two independent heuristics is called the Boyer-Moore algorithm.

The algorithm without the heuristics

The algorithm before the heuristics are applied is simply a version of the naive algorithm, in which each possible shift $s = 0, 1, \ldots$ is tried in turn.

However when testing a given shift, the characters in the pattern and text are compared *from right to left*. If all the characters match then we have found a valid shift.

If a mismatch is found, then the shift s is not valid, and we try the next possible shift by setting

$$s \leftarrow s + 1$$

and starting the testing loop again.

The two heuristics both operate by providing a number other than 1 by which the current shift can be incremented without missing any matches.

Bad characters

Consider the following situation:

```
o n c e _ w e _ n o t i c e d _ t h a t
          i m b a l a n c e
```

The two last characters `ce` match the text but the `i` in the text is a *bad character*.

Now as soon as we detect the bad character `i` we know immediately that the next shift must be at least 6 places or the `i` will simply not match.

Notice that advancing the shift by 6 places means that 6 text characters are not examined at all.

The bad character heuristic

The bad-character heuristic involves precomputing a function

$$\lambda : \Sigma \to \{0, 1, \ldots, m\}$$

such that for a character c, $\lambda(c)$ is the right-most position in P where c occurs (and 0 if c does not occur in P).

Then if a mismatch is detected when scanning position j of the pattern (remember we are going from right-to-left so j goes from m to 1), the bad character heuristic proposes advancing the shift by the equation:

$$s \leftarrow s + (j - \lambda(T[s + j]))$$

Notice that the bad-character heuristic might occasionally propose altering the shift to the left, so it cannot be used alone.

Good suffixes

Consider the following situation:

```
t h e _ l a t e _ e d i t i o n _ o f
        e d i t e d
```

The characters of the text that *do* match with the pattern are called the *good suffix*. In this case the *good suffix* is `ed`. Any shift of the pattern cannot be valid unless it matches at least the good suffix that we have already found. In this case we must move the pattern at least 4 spaces in order that the `ed` at the beginning of the pattern matches the good suffix.

The good-suffix heuristic

The good-suffix heuristic involves precomputing a function

$$\gamma : \{1, \ldots, m\} \to \{1, \ldots, m\}$$

where $\gamma(j)$ is the smallest positive shift of P so that it matches with all the characters in $P[j + 1..m]$ that it still overlaps.

We notice that this condition can always be vacuously satisfied by taking $\gamma(j)$ to be m, and hence $\gamma(j) > 0$.

Therefore if a mismatch is detected at character j in the pattern, the good-suffix heuristic proposes advancing the shift by

$$s \leftarrow s + \gamma(j)$$

The Boyer-Moore algorithm

The Boyer-Moore algorithm simply involves taking the larger of the two advances in the shift proposed by the two heuristics.

Therefore, if a mismatch is detected at character j of the pattern when examining shift s, we advance the shift according to:

$$s \leftarrow s + \max(\gamma(j), j - \lambda(T[s + j]))$$

The time taken to precompute the two functions γ and λ can be shown to be $O(m)$ and $O(|\Sigma| + m)$ respectively.

Like the naive algorithm the worst case is when the pattern matches every time, and in this case it will take just as much time as the naive algorithm. However this is rarely the case and in practice the Boyer-Moore algorithm performs well.

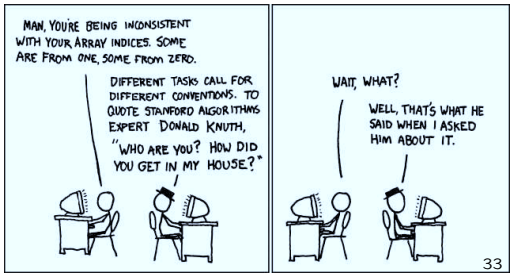
Example

Consider the pattern:

```
o n e _ s h o n e _ t h e _ o n e _ p h o n e
```

What is the last occurrence function λ?

c	$\lambda(c)$	c	$\lambda(c)$	c	$\lambda(c)$	c	$\lambda(c)$
a	0	h	20	o	21	v	0
b	0	i	0	p	19	w	0
c	0	j	0	q	0	x	0
d	0	k	0	r	0	y	0
e	23	l	0	s	5	z	0
f	0	m	0	t	11	-	0
g	0	n	22	u	0	_	18



Example continued

What is $\gamma(22)$? This is the smallest shift of P that will match the 1 character $P[23]$, and this is 6.

```
o n e _ s h o n e _ t h e _ o n e _ p h o n e
          o n e _ s h o n e _ t h e _ o n e
```

The smallest shift that matches $P[22..23]$ is also 6.

```
o n e _ s h o n e _ t h e _ o n e _ p h o n e
          o n e _ s h o n e _ t h e _ o n e
```

so $\gamma(21) = 6$.

The smallest shift that matches $P[21..23]$ is also 6

```
o n e _ s h o n e _ t h e _ o n e _ p h o n e
          o n e _ s h o n e _ t h e _ o n e
```

so $\gamma(20) = 6$.

However the smallest shift that matches $P[20..23]$ is 14

```
o n e _ s h o n e _ t h e _ o n e _ p h o n e
                        o n e _ s h o n e
```

so $\gamma(19) = 14$.

What about $\gamma(18)$? What is the smallest shift that can match the characters `p h o n e`? A shift of 20 will match all those that are still left.

```
o n e _ s h o n e _ t h e _ o n e _ p h o n e
                                    o n e
```

This then shows us that $\gamma(j) = 20$ for all $j \leq 18$, so

$$\gamma(j) = \begin{cases} 6 & 20 \leq j \leq 22 \\ 14 & j = 19 \\ 20 & 1 \leq j \leq 18 \end{cases}$$

Longest Common Subsequence

Consider the following problem

LONGEST COMMON SUBSEQUENCE
Instance: Two sequences X and Y
Question: What is a longest common subsequence of X and Y

Example

If

$$X = \langle A, B, C, B, D, A, B \rangle$$

and

$$Y = \langle B, D, C, A, B, A \rangle$$

then a longest common subsequence is either

$$\langle B, C, B, A \rangle$$

or

$$\langle B, D, A, B \rangle$$

A recursive relationship

As is usual for dynamic programming problems we start by finding an appropriate recursion, whereby the problem can be solved by solving smaller subproblems.

Suppose that

$$X = \langle x_1, x_2, \ldots, x_m \rangle$$

$$Y = \langle y_1, y_2, \ldots, y_n \rangle$$

and that they have a longest common subsequence

$$Z = \langle z_1, z_2, \ldots, z_k \rangle$$

If $x_m = y_n$ then $z_k = x_m = y_n$ and Z_{k-1} is a LCS of X_{m-1} and Y_{n-1}.

Otherwise Z is either a LCS of X_{m-1} and Y or a LCS of X and Y_{n-1}.

(This depends on whether $z_k \neq x_m$ or $z_k \neq y_n$ respectively — at least one of these two possibilities must arise.)

A recursive solution

This can easily be turned into a recursive algorithm as follows.

Given the two sequences X and Y we find the LCS Z as follows:

If $x_m = y_n$ then find the LCS Z' of X_{m-1} and Y_{n-1} and set $Z = Z'x_m$.

If $x_m \neq y_n$ then find the LCS Z_1 of X_{m-1} and Y, and the LCS Z_2 of X and Y_{n-1}, and set Z to be the longer of these two.

It is easy to see that this algorithm requires the computation of the LCS of X_i and Y_j for all values of i and j. We will let $l(i,j)$ denote the length of the longest common subsequence of X_i and Y_j.

Then we have the following relationship on the lengths

$$l(i,j) = \begin{cases} 0 & \text{if } ij = 0 \\ l(i-1,j-1)+1 & \text{if } x_i = y_j \\ \max(l(i-1,j), l(i,j-1)) & \text{if } x_i \neq y_j \end{cases}$$

Memoization

The simplest way to turn a top-down recursive algorithm into a sort of dynamic programming routine is *memoization*. The idea behind this is that the return values of the function calls are simply stored in an array as they are computed.

The function is changed so that its first step is to look up the table and see whether $l(i,j)$ is already known. If so, then it just returns the value immediately, otherwise it computes the value in the normal way.

Alternatively, we can simply accept that we must at some stage compute all the $O(n^2)$ values $l(i,j)$ and try to schedule these computations as efficiently as possible, using a *dynamic programming table*.

The dynamic programming table

We have the choice of memoizing the above algorithm or constructing a bottom-up dynamic programming table.

In this case our table will be an $(m+1) \times (n+1)$ table where the (i,j) entry is the length of the LCS of X_i and Y_j.

Therefore we already know the border entries of this table, and we want to know the value of $l(m,n)$ being the length of the LCS of the original two sequences.

In addition to this however we will retain some additional information in the table - namely each entry will contain either a left-pointing arrow $\leftarrow$, a upward-pointing arrow $\uparrow$ or a diagonal arrow $\nwarrow$.

These arrows will tell us which of the subcases was responsible for the entry getting that value.

Our example

For our worked example we will use the sequences

$$X = \langle 0, 1, 1, 0, 1, 0, 0, 1 \rangle$$

and

$$Y = \langle 1, 1, 0, 1, 1, 0 \rangle$$

Then our initial empty table is:

	j	0	1	2	3	4	5	6
i		y_j	1	1	0	1	1	0
0	x_i							
1	0							
2	1							
3	1							
4	0							
5	1							
5	0							
7	0							
8	1							

The first table

First we fill in the border of the table with the zeros.

	j	0	1	2	3	4	5	6
i		y_j	1	1	0	1	1	0
0	x_i	0	0	0	0	0	0	0
1	0	0						
2	1	0						
3	1	0						
4	0	0						
5	1	0						
5	0	0						
7	0	0						
8	1	0						

Now each entry (i, j) depends on x_i, y_j and the values to the left $(i, j-1)$, above $(i-1, j)$, and above-left $(i-1, j-1)$.

In particular, we proceed as follows:

If $x_i = y_j$ then put the symbol $\nwarrow$ in the square, together with the value $l(i-1, j-1) + 1$.

Otherwise put the greater of the values $l(i-1, j)$ and $l(i, j-1)$ into the square with the appropriate arrow.

The first row

It is easy to compute the first row, starting in the $(1, 1)$ position:

	j	0	1	2	3	4	5	6
i		y_j	1	1	0	1	1	0
0	x_i	0	0	0	0	0	0	0
1	0	0	$\uparrow$ 0	$\uparrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1
2	1	0						
3	1	0						
4	0	0						
5	1	0						
6	0	0						
7	0	0						
8	1	0						

Computation proceeds as described above.

The final array

After filling it in row by row we eventually reach the final array:

	j	0	1	2	3	4	5	6
i		y_j	1	1	0	1	1	0
0	x_i	0	0	0	0	0	0	0
1	0	0	$\uparrow$ 0	$\uparrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1
2	1	0	$\nwarrow$ 1	$\nwarrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\nwarrow$ 2	$\leftarrow$ 2
3	1	0	$\nwarrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2	$\nwarrow$ 2	$\nwarrow$ 3	$\leftarrow$ 3
4	0	0	$\uparrow$ 1	$\uparrow$ 2	$\nwarrow$ 3	$\leftarrow$ 3	$\uparrow$ 3	$\nwarrow$ 4
5	1	0	$\nwarrow$ 1	$\nwarrow$ 2	$\uparrow$ 3	$\nwarrow$ 4	$\nwarrow$ 4	$\uparrow$ 4
6	0	0	$\uparrow$ 1	$\uparrow$ 2	$\nwarrow$ 3	$\uparrow$ 4	$\uparrow$ 4	$\nwarrow$ 5
7	0	0	$\uparrow$ 1	$\uparrow$ 2	$\nwarrow$ 3	$\uparrow$ 4	$\uparrow$ 4	$\nwarrow$ 5
8	1	0	$\uparrow$ 1	$\nwarrow$ 2	$\uparrow$ 3	$\nwarrow$ 4	$\nwarrow$ 5	$\uparrow$ 5

This then tells us that the LCS of $X = X_8$ and $Y = Y_6$ has length 5 — because the entry $l(8, 6) = 5$.

This time we have kept enough information, via the arrows, for us to compute what the LCS of X and Y is.

Finding the LCS

The LCS can be found (in reverse) by tracing the path of the arrows from $l(m,n)$. Each *diagonal* arrow encountered gives us another element of the LCS.

As $l(8,6)$ points to $l(7,6)$ so we know that the LCS is the LCS of X_7 and Y_6.

Now $l(7,6)$ has a diagonal arrow, pointing to $l(6,5)$ so in this case we have found the last entry of the LCS — namely it is $x_7 = y_6 = 0$.

Then $l(6,5)$ points (upwards) to $l(5,5)$, which points diagonally to $l(4,4)$ and hence 1 is the second-last entry of the LCS.

Proceeding in this way, we find that the LCS is

11010

Notice that if at the very final stage of the algorithm (where we had a free choice) we had chosen to make $l(8,6)$ point to $l(8,5)$ we would have found a different LCS

11011

Finding the LCS

We can trace back the arrows in our final array, in the manner just described, to determine that the LCS is 11010 and see which elements within the two sequences match.

	j	0	1	2	3	4	5	6
i		y_j	1	1	0	1	1	0
0	x_i	0	0	0	0	0	0	0
1	0	0	↑ 0	↑ 0	↖ 1	← 1	← 1	↖ 1
2	1	0	↖ 1	↖ 1	↑ 1	↖ 2	↖ 2	← 2
3	1	0	↖ 1	↖ 2	← 2	↖ 2	↖ 3	← 3
4	0	0	↑ 1	↑ 2	↖ 3	← 3	↑ 3	↖ 4
5	1	0	↖ 1	↖ 2	↑ 3	↖ 4	↖ 4	↑ 4
6	0	0	↑ 1	↑ 2	↖ 3	↑ 4	↑ 4	↖ 5
7	0	0	↑ 1	↑ 2	↖ 3	↑ 4	↑ 4	↖ 5
8	1	0	↑ 1	↖ 2	↑ 3	↖ 4	↖ 5	↑ 5

A match occurs whenever we encounter a diagonal arrow along the reverse path.

See section 15.4 of CLRS for the pseudo-code for this algorithm.

Analysis

The analysis for longest common subsequence is particularly easy.

After initialization we simply fill in mn entries in the table — with each entry costing only a constant number of comparisons. Therefore the cost to produce the table is $\Theta(mn)$

Following the trail back to actually find the LCS takes time at most $O(m+n)$ and therefore the total time taken is $\Theta(mn)$.

Data Compression Algorithms

Data compression algorithms exploit patterns in data files to compress the files. Every compression algorithm should have a corresponding decompression algorithm that can recover (most of) the original data.

Data compression algorihtms are used by programs such as WinZip, pkzip and zip. They are also used in the definition of many data formats such as pdf, jpeg, mpeg and .doc.

Data compression algorithms can either be *lossless* (e.g. for archiving purposes) or *lossy* (e.g. for media files).

We will consider some lossless algorithms below.

Huffman coding

A nice application of a greedy algorithm is found in an approach to data compression called Huffman coding.

Suppose that we have a large amount of text that we wish to store on a computer disk in an efficient way. The simplest way to do this is simply to assign a binary code to each character, and then store the binary codes consecutively in the computer memory.

The ASCII system for example, uses a fixed 8-bit code to represent each character. Storing n characters as ASCII text requires $8n$ bits of memory.

Simplification

Let C be the set of characters we are working with. To simplify things, let us suppose that we are storing only the 10 numeric characters 0, 1, ..., 9. That is, set $C = \{0, 1, \cdots, 9\}$.

A fixed length code to store these 10 characters would require at least 4 bits per character. For example we might use a code like this:

Char	Code
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

However in any non-random piece of text, some characters occur far more frequently than others, and hence it is possible to save space by using a variable length code where the more frequently occurring characters are given shorter codes.

Non-random data

Consider the following data, which is taken from a Postscript file.

Char	Freq
5	1294
9	1525
6	2260
4	2561
2	4442
3	5960
7	6878
8	8865
1	11610
0	70784

Notice that there are many more occurrences of 0 and 1 than the other characters.

A good code

What would happen if we used the following code to store the data rather than the fixed length code?

Char	Code
0	1
1	010
2	01111
3	0011
4	00101
5	011100
6	00100
7	0110
8	000
9	011101

To store the string 0748901 we would get

00000111010010001001000000001

using the fixed length code and

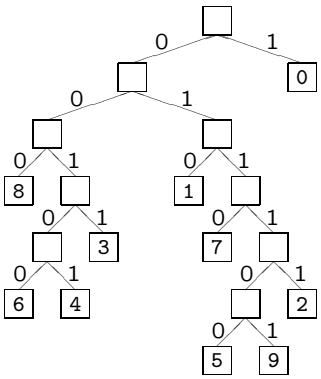
10110001010000111011010

using the variable length code.

Prefix codes

In order to be able to decode the variable length code properly it is necessary that it be a **prefix code** — that is, a code in which no codeword is a prefix of any other codeword.

Decoding such a code is done using a binary tree.



Cost of a tree

Now assign to each leaf of the tree a value, $f(c)$, which is the frequency of occurrence of the character c represented by the leaf.

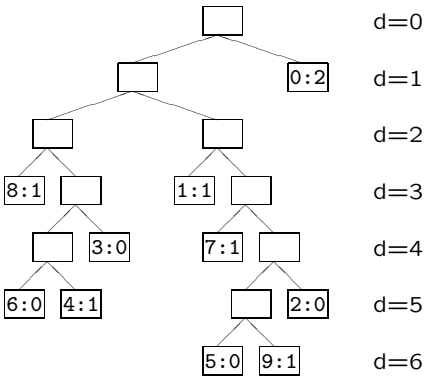
Let $d_T(c)$ be the depth of character c's leaf in the tree T.

Then the number of bits required to encode a file is

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

which we define as the **cost** of the tree T.

For example, the number of bits required to store the string 0748901 can be computed from the tree T:



giving

$B(T) = 2{\times}1{+}1{\times}3{+}1{\times}3{+}1{\times}4{+}1{\times}5{+}1{\times}6 = 23.$

Thus, the cost of the tree T is 23.

Optimal trees

A tree representing an optimal code for a file is always a *full* binary tree — namely, one where every node is either a leaf or has precisely two children.

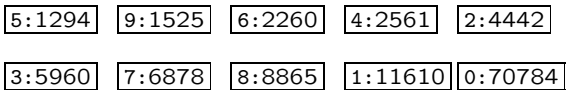
Therefore if we are dealing with an alphabet of s symbols we can be sure that our tree has precisely s leaves and $s - 1$ internal nodes, each with two children.

Huffman invented a greedy algorithm to construct such an optimal tree.

The resulting code is called a **Huffman code** for that file.

Huffman's algorithm

The algorithm starts by creating a forest of s single nodes, each representing one character, and each with an associated value, being the frequency of occurrence of that character. These values are placed into a priority queue (implemented as a linear array).



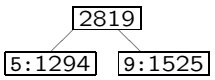
Then repeat the following procedure $s - 1$ times:

Remove from the priority queue the two nodes L and R with the lowest values, and create a internal node of the binary tree whose left child is L and right child R.

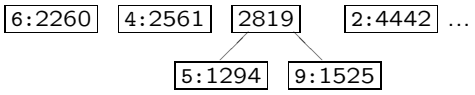
Compute the value of the new node as the sum of the values of L and R and insert this into the priority queue.

The first few steps

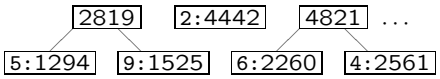
Given the data above, the first two entries off the priority queue are 5 and 9 so we create a new node



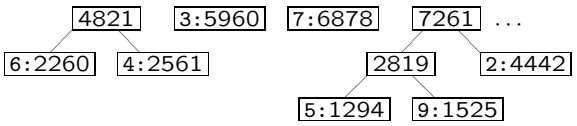
The priority queue is now one element shorter, as shown below:



The next two are 6 and 4 yielding



Now the smallest two nodes are 2 and the internal node with value 2819, hence we now get:



Notice how we are growing sections of the tree from the bottom-up (compare with the tree on slide 16).

See CLRS (page 388) for the pseudo-code corresponding to this algorithm.

Why does it work?

In order to show that Huffman's algorithm works, we must show that there can be no prefix codes that are better than the one produced by Huffman's algorithm.

The proof is divided into two steps:

First it is necessary to demonstrate that the first step (merging the two lowest frequency characters) cannot cause the tree to be non-optimal. This is done by showing that any optimal tree can be reorganised so that these two characters have the same parent node. (see CLRS, Lemma 16.2, page 388)

Secondly we note that after making an optimal first choice, the problem can be reduced to finding a Huffman code for a smaller alphabet. (see CLRS, Lemma 16.3, page 391)

Adaptive Huffman Coding

Huffman coding requires that we have accurate estimates of the probablities of each character occuring.

In general, we can make estimates of the frequencies of characters occuring in English text, but these estimates are not useful when we consider other data formats.

Adaptive Huffman coding calculates character frequencies on the fly and uses these dynamic frequencies to encode characters. This technique can be applied to binary files as well as text files.

Algorithms: Adaptive Huffman Coding

The **Adaptive Huffman Coding** algorithms seek to create a Huffman tree on the fly. A Huffman Coding allows us to encode frequently occurring characters in a lesser number of bits than rarely occurring characters. Adaptive Huffman Coding determines determines the Huffman Tree only from the frequencies of the characters already read.

Recall that prefix codes are defined using a binary tree. It can be shown that a prefix code is optimal if and only if the binary tree has the *sibling property*.

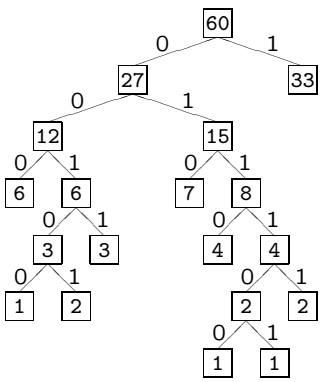
A binary tree recording the frequency of characters has the sibling property iff

1. every node except the root has a sibling.

2. each right-hand sibling (including non-leaf nodes) has at least as high a frequency as its left-hand sibling

(The frequency of non-leaf nodes ins the sum of the frequency of it's children).

Adaptive Huffman Coding



Adaptive Huffman Coding

As characters are read it is possible to efficiently update the frequencies, and modify the binary tree so that the sibling property is preserved. It is also possible to do this in a deterministic way so that a similar process can decompress the code.

See
`http://www.cs.duke.edu/ jsv/Papers/Vit87.jacmACMve`
for more details.

As opposed to the *LZ* algorithms that follow, Huffman methods only encode one character at a time. However, best performance often comes from combining compression algorithms (for example, *gzip* combines *LZ77* and *Adaptive Huffman Coding*).

Ziv-Lempel compression algorithms

The Ziv-Lempel compression algorithms are a family of compression algorithms that can be applied to arbitrary file types.

The Ziv-Lempel algorithms represent recurring strings with abbreviated codes. There are two main types:

- LZ77 variants use a buffer to look for recurring strings in a small section of the file.
- LZW variants dynamically create a dictionary of recurring strings, and assigns a simple code to each such string.

Algorithms: LZ77

The **LZ77** algorithms use a *sliding window*. The sliding window is a buffer consisting of the last m letters encoded ($a_0 ... a_{m-1}$) and the next n letters to be encoded ($b_0 ... b_{n-1}$).

Initially we let $a_0 = a_1 = ... = a_{n-1} = w_0$ and output $\langle 0, 0, w \rangle$ where w_0 is the first letter of the word to be compressed

The algorithm looks for the longest prefix of $b_0 ... b_{n-1}$ appearing in $a_0 ... a_{m-1}$. If the longest prefix found is $b_0 ... b_{k-1} = a_i ... a_{i+k-1}$, then the entire prefix is encoded as the tuple

$$\langle i, k, b_k \rangle$$

where i is the *offset*, k is the *length* and b_k is the *next character*.

LZ77 Example

Suppose that $m = n = 4$ and we would like to compress the word $w = aababacbaa$

Word	Window	Output
$aababacbaa$		$\langle 0, 0, a \rangle$
$aababacbaa$	$\underline{aa}aa\ \underline{aa}ba$	$\langle 0, 2, b \rangle$
$abacbaa$	$aa\underline{ab\ a}\ \underline{aba}c$	$\langle 2, 3, c \rangle$
baa	$a\underline{ba}c\ \underline{ba}a$	$\langle 1, 2, a \rangle$

This outputs

$$\langle 0, 0, a \rangle \langle 0, 2, b \rangle \langle 2, 3, c \rangle \langle 1, 2, a \rangle$$

LZ77 Example cont.

To decompress the code we can reconstruct the sliding window at each step of the algorithm. Eg, given

$$\langle 0, 0, a \rangle \langle 0, 2, b \rangle \langle 2, 3, c \rangle \langle 1, 2, a \rangle$$

Input	Window	Output
$\langle 0, 0, a \rangle$		
$\langle 0, 2, b \rangle$	$\underline{aa}aa\ aab$?	aab
$\langle 2, 3, c \rangle$	$aa\underline{ab\ a}\ abac$	$abac$
$\langle 1, 2, a \rangle$	$a\underline{ba}c\ baa$?	baa

Note the trick with the third triple $\langle 2, 3, c \rangle$ that allows the look-back buffer to overflow into the look ahead buffer. See `http://en.wikipedia.org/wiki/LZ77_and_LZ78` for more information.

Algorithms: LZW

The **LZW** algorithms use a *dynamic dictionary* The dictionary maps words to codes and is initially defined for every byte (0-255). The compression algorithm is as follows:

```
w = null
while(k = next byte)
    if wk in the dictionary
        w = wk
    else
        add wk to dictionary
        output code for w
        w = k
output code for w
```


Algorithms: LZW

The decompression algorithm is as follows:

```
k = next byte
output k
w = k
while(k = next byte)
    if there's no dictionary entry for k
        entry = w + first letter of w
    else
        entry = dictionary entry for k
    output entry
    add w + first letter of entry to dictionary
    w = entry
```


LZW Example

Consider the word $w = aababa$, and a dictionary D where $D[0] = a$, $D[1] = b$ and $D[2] = c$. The compression algorithm proceeds as follows:

Read	Do	Output
a	$w = a$	–
a	$w = a, D[3] = aa$	0
b	$w = b, D[4] = ab$	0
a	$w = a, D[5] = ba$	1
b	$w = ab$	–
a	$w = a, D[6] = aba$	4
c	$w = c, D[7] = ac$	0
b	$w = b, D[8] = cb$	2
a	$w = ba$	–
a	$w = a, D[9] = baa$	5
		0

LZW Example cont.

To decompress the code $\langle 00140250 \rangle$ we initialize the dictionary as before. Then

Read	Do	Output
0	$w = a$	a
0	$w = a, D[3] = aa$	a
1	$w = b, D[4] = ab$	b
4	$w = ab, D[5] = ba$	ab
0	$w = a, D[6] = aba$	a
2	$w = c, D[7] = ac$	c
5	$w = ba, D[8] = cb$	ba
0	$w = a, D[9] = baa$	a

See `http://en.wikipedia.org/wiki/LZ77_and_LZ78`, also.

Summary

1. String matching is the problem of finding all matches for a given pattern, in a given sample of text.

2. The Rabin-Karp algorithm uses prime numbers to find matches in linear time in the expected case.

3. A String matching automata works in linear time, but requires a significant amount of precomputing.

4. The Knuth-Morris-Pratt uses the same principal as a string matching automata, but reduces the amount of precomputation required.

Summary cont.

5. The Boyer-Moore algorithm uses the *bad character* and *good suffix* heuristics to give the best performance in the expected case.

6. The longest common subsequence problem is can be solved using dynamic programming.

7. Dynammic programming can improve the efficiency of divide and conquor algorithms by storing the resul;ts of sub-computations so they can be reused later.

8. Data Compression algorithms use pattern matching to find efficient ways to compress file.

Summary cont.

9. Huffman coding uses a greedy approach to recode the alphabet with a more efficient binary code.

10. Adaptive Huffman coding uses the same approach, but with the overhead of precomputing the code.

11. LZ77 uses pattern matching to express segments of the file in terms of recently occuring segments.

12. LZW uses a hash function to store commonly occuring strings so it can refer to them by their key.

Computer Science and Software Engineering, 2011

CITS3210 Algorithms

Optimization Algorithms

Notes by CSSE, Comics by xkcd.com

Algorithm Design

In this section we will consider some general algorithmic techniques for optimization problems — namely **greedy algorithms**, **dynamic programming** and **Approximation Algorithms**.

A greedy algorithm proceeds by making a single choice at each stage of the computation — at each stage the algorithm chooses the "best" move to make based on purely local information. Previously seen examples include Kruskal's algorithm, Prim's algorithm and Huffman Coding.

Greedy algorithms are usually extremely efficient, but they can only be applied to a small number of problems.

Greedy Algorithms

Consider the following simple computational problem.

ACTIVITY SELECTION
Instance: A set $S = \{t_1, t_2, \ldots, t_n\}$ of "activities" where each activity t_i has an associated start time s_i and finish time f_i.
Question: Select the largest possible number of tasks from S that can be completed without incompatibilities (two activities are *incompatible* iff they overlap).

Example Consider the following set of activities

$\{(6,9), (1,10), (2,4), (1,7), (5,6), (8,11), (9,11)\}$

The following schedules are all allowable

$(1,10)$
$(1,7), (8,11)$
$(2,4), (5,6), (9,11)$

Intervals

There is an obvious relationship between activities and *intervals* on the real line.

An interval of the real line consists of the real numbers lying between two reals called the *endpoints* of the interval.

$$(a,b) = \{x \in \mathbf{R} \mid a < x < b\}$$

If the interval includes its endpoints then it is said to be closed, otherwise open. It can also be open at one endpoint and closed at the other.

$$(a,b) = \{x \in \mathbf{R} \mid a < x < b\}$$

$$[a,b) = \{x \in \mathbf{R} \mid a \leq x < b\}$$

$$(a,b] = \{x \in \mathbf{R} \mid a < x \leq b\}$$

$$[a,b] = \{x \in \mathbf{R} \mid a \leq x \leq b\}$$

For definiteness we will henceforth make the assumption that all the activity intervals are *closed on the left* and *open on the right*.

$$t_i = [s_i, f_i)$$

Problem reduction

To solve this problem we must make some choice of the first interval, then the second interval and so on. Clearly the later choices depend on the earlier ones in that some time-slots are no longer available.

Suppose that we (arbitrarily) select the interval $[1, 7)$ from the collection

$$\{[6, 9), [1, 10), [2, 4), [1, 7), [5, 6), [8, 11), [9, 11)\}.$$

Then all the intervals that overlap with this one can no longer be scheduled, leaving the set

$$\{[8, 11), [9, 11)\}$$

from which we must choose the largest possible set of pairwise disjoint intervals — in this case just one of the remaining intervals.

This is simply a smaller instance of the same problem **ACTIVITY SELECTION**. Therefore an algorithm for the problem can be expressed recursively simply by specifying a rule for choosing one interval.

Greedy approach

It is easy to see that choosing [1,7) was a bad choice, which raises the question of what would be a good choice?

A greedy algorithm simply chooses what is locally the best option at every stage. There are various possible ways to be greedy, including

- Choose the shortest interval
- Choose the interval starting the first
- Choose the interval finishing the first
- Choose the interval that intersects with the fewest others

The greedy approach can be viewed as a very local procedure — making the best choice for the current moment without regard for any possible future consequences of that choice.

Sometimes a greedy approach yields an optimal solution, but frequently it does not.

Activity Selection

Consider the greedy approach of selecting the interval that finishes first from the collection

$$\{[6, 9), [1, 10), [2, 4), [1, 7), [5, 6), [8, 11), [9, 11)\}$$

Then we would choose $[2, 4)$ as the first interval, and after eliminating clashes we are left with the task of finding the largest set of mutually disjoint intervals from the set

$$\{[6, 9), [5, 6), [8, 11), [9, 11)\}.$$

At this stage, we simply apply the algorithm recursively. Therefore being greedy in the same way we select $[5, 6)$ as the next interval, and after eliminating clashes (none in this case) we are left with.

$$\{[6, 9), [8, 11), [9, 11)\}.$$

Continuing in this way gives the ultimate result that the largest possible collection of non-intersecting intervals is

[2,4) then [5,6) then [6,9) then [9,11).

Algorithm

As a precondition the list of tasks must be sorted into ascending order of their finish times to ensure

$$\text{finish}(t_1) \leq \text{finish}(t_2) \leq \text{finish}(t_3) \leq \ldots$$

The pseudo-code will then process the sorted list of tasks t:

procedure GREEDY-ACTIVITY-SEL(t)
  $A \leftarrow \{t_1\}$
  $i \leftarrow 1$
  for $m \leftarrow 2$ **to** length(t) **do**
    if start(t_m) $\geq$ finish(t_i) **then**
      $A \leftarrow A \cup \{t_m\}$
      $i \leftarrow m$
    end if
  end for
  return A

It returns A, a subset of compatible activities.

Does it work?

The greedy algorithm gives us a solution to the activity scheduling problem — but is it actually the best solution, or could we do better by considering the global impact of our choices more carefully.

For the problem **ACTIVITY SELECTION** we can show that the greedy algorithm always finds an optimal solution.

We suppose first that the activities are ordered by finishing time - so that

$$f_1 \leq f_2 \leq \cdots \leq f_n$$

Now consider some optimal solution for the problem consisting of k tasks

$$t_{i_1}, t_{i_2}, \ldots, t_{i_k}$$

Then

$$t_1, t_{i_2}, \ldots, t_{i_k}$$

is also an optimal solution since it will also consist of k tasks.

Intuitive Proof

The formal proof that we can use t_1 as the first task and be certain that we will not change the number of compatible tasks is rather involved and you are referred to the text book (see CLRS, pages 373-375).

However the basic idea is a proof by contradiction. Assume using t_1 results in a sub-optimal solution and therefore we can find a compatible solution with $(k+1)$ tasks. This would only be possible if we can find two tasks t'_1 and t''_1 which occupy the same interval as t_1. But this would imply

$$(s_1 \leq (s'_1 < f'_1) \leq (s''_1 < f''_1) \leq f_1)$$

and hence that $f'_1 < f_1$ but we know that the tasks are sorted in order of ascending finish times, so no task can have a finish time less that that of t_1, leading to a contradiction. Hence using t_1 as the first task **must** lead to an optimal solution with k tasks.

Running time

The running time for this algorithm is dominated by the time taken to sort the n inputs at the start of the algorithm.

Using quicksort this can be accomplished in an average time of $O(n \lg n)$.

As greedy algorithms are so simple, they always have low degree polynomial running times.

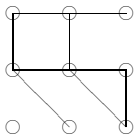
Because they are so quick, we might be tempted to ask why we should not always use greedy algorithms.

Unfortunately, greedy algorithms only work for a certain narrow range of problems — most problems cannot be solved by a greedy algorithm.

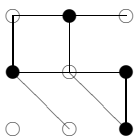
Vertex Cover

A *vertex cover* for a graph G is a set of vertices $V' \subseteq V(G)$ such that every edge has at least one end in V' (the set of vertices *covers* all the edges.

The following graph



has a vertex cover of size 4.

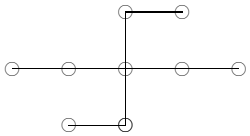


The **VERTEX COVER** problem is to find the smallest vertex cover for a graph G.

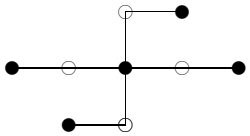
A greedy algorithm

One greedy algorithm is to cover as many edges as possible with each choice, by choosing the vertex of highest degree at each stage and then deleting the covered edges.

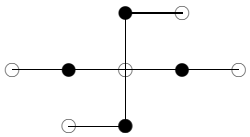
For this graph



the greedy algorithm gives



while the true solution is



Greed is not always good

The previous example shows that it does not always pay to be greedy. Although choosing the vertex of highest degree does cover the greatest number of edges, that choice makes our later choices worse.

In problems where the greedy algorithm works, the earlier choices do not interfere negatively with the later choices.

Unfortunately, most problems are not amenable to the greedy algorithm.

VERTEX COVER is actually a very hard problem, and there is no known algorithm that is essentially better than just enumerating all the possible subsets of vertices. (Technically speaking, it is an example of a problem that is known to be NP-hard.)

Non-deterministic polynomial time

A computational problem is in the class **P**, (the polynomial time problems) if there is a deterministic algorithm that solves the problem and runs in time $O(n^k)$ where k is some integer. These problems are generally considered *feasible*.

A computational problem is in the class **NP**, (the non-deterministic polynomial time problems) if there is a non-deterministic algorithm that that can solve the problem in polynomial time.

That is, an **NP** algorithm requires lucky guesses to work efficiently (i.e. guessing what the optimal vertex cover is).

More NP-problems

Consider the following two problems:

TRAVELLING SALESMAN
Instance: A finite set $C = \{c_1, c_2, \ldots, c_n\}$ of "cities", a "distance" $d(c_i, c_j) \in \mathbf{R}^+$ between each pair of cities.
Question: What is the shortest circular tour visiting each city exactly once?

DOMINATING SET
Instance: A graph G
Question: What is the smallest *dominating set* for G?

(A dominating set of a graph is a set of vertices $V' \subseteq V$ such that every vertex of G has distance at most 1 from some vertex in V'.)

How hard are these problems?

There are no algorithms known for these problems whose time complexity is a polynomial function of the size of the input. This means that the only known algorithms take time that is exponential in the size of the input.

There is a large class of problems, known as NP-hard problems which have the following properties

- There is no polynomial time algorithm known for the problem
- If you could solve one of these problems in polynomial time, then you could solve them all in polynomial time

Both TRAVELLING SALESMAN and DOMINATING SET are NP-hard.

The most important problem in theoretical computer science is whether or not this class of problems can be solved in polynomial time.

The 0-1 Knapsack Problem

Suppose we are given a *knapsack* of a given capacity, and a selection of items, each with a given weight and value. The 0-1 knapsack problem is to select the combination of items with the greatest value that will fit into the knapsack.

Formally, if W is the size of the knapsack and $\{1, ...n\}$ is a set of items where the weight of i is w_i and the value of i is v_i, then the problem is to:
Select $T \subseteq \{1, ..., n\}$ that maximizes $\Sigma_{i \in T} v_i$, given $\Sigma_{i \in T} w_i < W$.

For example W might be the amount of memory on an MP3 player, w_i may be the size of the song i, and v_i may reflect how much you like song i.

The Fractional Knapsack Problem

The fractional knapsack problem is similar, except that rather than choosing which items to take, you are able to choose how much of each item you will take. That is the problem is to find a function $\mathcal{T} : \{1, ..., n\} \rightarrow [0, 1]$ that maximizes $\Sigma_{i \in T} \mathcal{T}(i) v_i$, given $\Sigma_{i \in T} \mathcal{T}(i) w_i < W$.

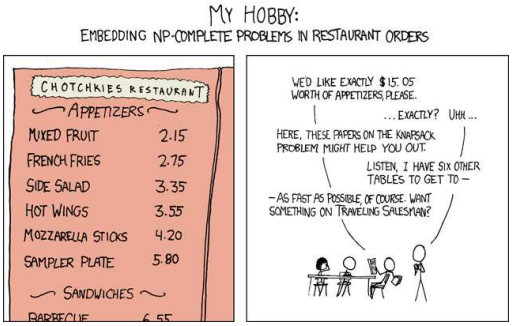
It is easy to see that the fractional knapsack problem can be solved by a greedy algorithm. However the 0-1 knapsack problem is much harder, and has been shown to be NP-complete.

While there is no known "feasible" solution for the 0-1 knapsack problem we will examine a dynamic programming solution that can give reasonable performance.

A dynamic programming solution

The structure of a dynamic programming algorithm is to:

1. define the solution to the problem in terms of solutions to sub-problems;
2. recursively solve the smaller sub-problems, recording the solutions in a table;
3. construct the solution to the original problem from the table.



A recursive solution

Given the 0-1 knapsack problem specified by the pair $(\{w_1, ...w_n\}, \{v_1, ..., v_n\}, W)$, we will consider the solution to the sub-problems specified by the pairs $(\{w_1, ...w_m\}, \{v_1, ..., v_m\}, w)$ where $m < n$ and $w < W$.

Let $V(m, w)$ be the value of the optimal solution to this subproblem. Then for any m and any w, we can see

$V(m, w) = \max\{V(m-1, w), v_m + V(m-1, w-w_m)\}$.

Since $V(0, w) = 0$ for all w this allows us to define a (very inefficient) recursive algorithm.

A dynamic programming solution

Often inefficient recursive algorithms can be made more efficient by using dynamic programming. The structure of a dynamic programming algorithm is to:

1. define as recursive solution to the problem in terms of solutions to sub-problems;
2. recursively solve the smaller sub-problems, recording the solutions in a table;
3. construct the solution to the original problem from the table.

For the 0-1 knapsack problem we will construct a table where the entries are $V(i, j)$ for $i = 0, ..., n$ and $j = 0, ..., W$.

Example

Suppose $W = 5$ and we are given three items where

i	1	2	3
v_i	2	3	4
w_i	1	2	3

The table initially looks like

$i \backslash w$	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						

Pseudo-code

Knapsack$(\{w_1, ...w_n\}, \{v_1, ..., v_n\}, W)$
  for w **from** 1 **to** n **do**
    $V(0, w) \leftarrow 0$
  for i **from** 1 **to** n **do**
    for w **from** 1 **to** n **do**
      if $V(i-1, w) > v_i + V(i-1, w-w_i)$ **do**
        $V(i, w) \leftarrow V(i-1, w)$
      else
        $V(i, w) \leftarrow v_i + V(i-1, w-w_i)$
  return $V(n, W)$

It is clear that the complexity of this algorithm is $O(nW)$. Note that this is **not** a polynomial solution to an NP-complete problem. *Why not*

Example

$i\backslash w$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	<u>0</u>	2	2	2	2	2
2	0	2	<u>3</u>	4	4	4
3	0	2	3	4	6	<u>7</u>

Note that the actual items contributing to the solution (that is, items 2 and 3) can be found by examination of the table. If $T(i, w)$ are the items that produce the solution $V(i, w)$, then

$$\begin{aligned} T(i,w) &= T(i-1,w) \text{ if } V(i,w) = V(i-1,w) \\ &= \{i\} \cup T(i-1, w-w_i) \text{ otherwise.} \end{aligned}$$

Linear Programming

The fractional knapsack problem is an example of a *linear programming problem*. A linear programme is an optimization problem of the form:
Find real numbers: $x_1, ..., x_n$
that maximizes $\Sigma_{i=1}^{n} c_i x_i$
subject to $\Sigma_{i=1}^{n} a_{ij} x_i \leq b_j$ for $j = 1, ..., m$
and $x_i \geq 0$ for $j = 1, ..., n$.
Therefore a linear programme is paramaterized by the the *cost vector*, $(c_1, ..., c_n)$, an $n \times m$ array of constraint coefficients, a_{ij}, and a *bounds vector* $(b_1, ..., b_m)$.

It is clear the fractional knapsack problem can be presented as a linear programme.

Applications of linear programming

Many natural optimization problems can be expressed as a linear programme.

For example, given a weighted, directed graph, $G = (V, E)$, the length of shortest path from s to t can be described using a linear programme. Using the distance array from the Bellman-Ford algorithm, we have the programme:
Maximize $d[t]$
subject to $d[v] - d[u] \leq w(u, v)$ for $j = 1, ..., m$
and $d[s] = 0$.
Maximum flow problems can also be easily converted into linear programmes.

Solving linear programmes

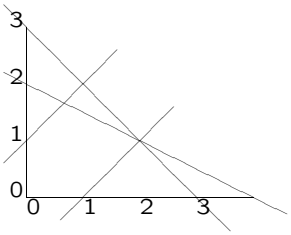
All linear programmes can be solved by the *simplex algorithm*, which requires exponential time, but is generally feasible in practise.

The simplex algorithm is effectively a hill-climbing algorithm that moves incrementally improves the solution until no further improvements can be made.

There are also polynomial *interior point methods* to solve linear programmes.

We won't examine these algorithms. Rather we will simply consider the technique of *converting* problems into linear programmes.

Example



Maximize $x + y$, where $x + 2y \leq 4$
$y - x \leq 1$
$x - y \leq 1$
$x, y \geq 0$

Integer Linear Programming

Adding the constraint that all solutions to a linear programme be integer values, gives an *integer linear programme*.

The 0-1 knapsack problem can be written as an integer linear programme, as can the travelling salesmen problem.

Therefore we should not expect to find a feasible algorithm to to solve the integer linear programming problem.

Approximation Algorithms

An *approximation algorithm* is an algorithm that produces some *feasible* solution but with no guarantee that the solution is optimal.

Therefore an approximation algorithm for the travelling salesman problem would produce some valid circular tour, but it may not be the shortest tour.

An approximation algorithm for the minimum dominating set problem would produce some dominating set for G, but it may not be the smallest possible dominating set.

The performance of an approximation algorithm on a given instance I is measured by the ratio

$$A(I)/OPT(I)$$

where $A(I)$ is the value given by the approximation algorithm and $OPT(I)$ is the true optimum value.

Standard Instances

Both TRAVELLING SALESMAN and DOMINATING SET have been fairly extensively studied, and a number of algorithms for their solution have been proposed.

In each case there are some *standard instances* for would-be solvers to test their code on. A package called TSPLIB provides a variety of standard travelling salesman problems. Some of them have known optimal solutions, while others are currently unsolved and TSPLIB just records the best known solution.

There are problems with around 2000 cities for which the best solution is not known, but this problem has been very heavily studied by brilliant groups of researchers using massive computer power and very sophisticated techniques.

The football pool problem

In many European countries a popular form of lottery is the "football pools", which are based on the results of soccer matches. Each player picks the results of n matches, where the result can be either a Home Win, Away Win or Draw.

By assigning three values as follows

0 for Home Win
1 for Away Win
2 for Draw

we can think of this choice as a word of length n with entries from the alphabet $\{0, 1, 2\}$.

For example

020201

would mean that the player had picked Home Wins for matches 1, 3 and 5, Away Win for match 6 and Draws for matches 2 and 4.

Winning 2nd prize

Now there are a total of 729 possible outcomes for the 6 matches. To guarantee winning the first prize we would need to make 729 different entries to cover every possible outcome.

Suppose however that getting all but one of the predictions correct results in winning second prize. So for example if our entry was 020201 and the actual outcome was 010201 then we would have 5 out of 6 correct and would win second prize.

In trying to generate pools "systems" we want to be able to answer the question

"How many entries do we need to make in order to *guarantee* winning at least second prize?"

A graph domination problem

We can define a graph F_6 as follows:

The vertices of F_6 are the 729 words of length 6 over $\{0, 1, 2\}$.
Two vertices are adjacent if the corresponding words differ in only one coordinate position.

Then we are seeking a minimum dominating set for the graph F_6.

More generally, we can define a series of graphs F_n where the vertices are the 3^n words of length n with entries from $\{0, 1, 2\}$ with the same rule for determining adjacency.

This collection of graphs is called the *football pool graphs* and has been quite extensively studied with regard to the size of the minimum dominating set.

Known records

The following are the best known values for a minimum dominating set for F_n.

n	Number vertices	Best known dom. set
2	9	3
3	27	5
4	81	9
5	243	27
6	729	≤ 73
7	2187	≤ 186
8	6561	≤ 486

Notice that the minimum dominating set for F_4 is *perfect* — each vertex is adjacent to 8 others, so that each vertex of the dominating set dominates 9 vertices. As there are 81 vertices in F_4 this means every vertex is dominated by *exactly one* vertex in the dominating set.

This is usually called a *perfect code*.

A greedy approximation algorithm

There is a natural greedy approximation algorithm for the minimum dominating set problem.

Start by selecting a vertex of maximum degree (so it dominates the greatest number of vertices). Then mark or delete all of the dominated vertices, and select the next vertex that dominates the greatest number of currently undominated vertices. Repeat until all vertices are dominated.

The graph P_5 (a path with 5 vertices) shows that this algorithm does not always find the optimal solution.

Types of Travelling Salesman Instance

Consider a travelling salesman problem defined in the following way. The "cities" are n randomly chosen points $c_i = (x_i, y_i)$ on the Euclidean plane, and the "distances" are defined by the normal Euclidean distance

$$d(c_i, c_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

or the *Manhattan distance*

$$d(c_i, c_j) = |x_i - x_j| + |y_i - y_j|$$

Instance of the travelling salesman problem that arise in this fashion are called *geometric* travelling salesman problems. Here the "distance" between the cities is actually the geometric distance between the corresponding points under some metric.

Properties of geometric instances

All geometric instances have the properties that they are *symmetric* and satisfy the *triangle inequality*.

If

$$d(c_i, c_j) = d(c_j, c_i)$$

for all pairs of cities in an instance of TRAVELLING SALESMAN then we say that the instance is *symmetric*.

If

$$d(c_i, c_k) \leq d(c_i, c_j) + d(c_j, c_k)$$

for all triples of cities in an instance of TRAVELLING SALESMAN then we say that the instance satisfies the *triangle inequality*.

Non-geometric instances

Of course it is easy to define instances that are not geometric.

Let $X = \{A, B, C, D, E, F\}$

Let d be given by

	A	B	C	D	E	F
A	0	2	4	∞	1	3
B	2	0	6	2	1	4
C	4	∞	0	1	2	1
D	∞	2	1	0	9	1
E	1	1	2	6	0	3
F	3	4	1	1	3	0

Many approximation algorithms only work for geometric instances because it is such an important special case, but remember that it is only a special case!

Nearest Neighbour

One example of an approximation algorithm is the following greedy algorithm known as Nearest Neighbour (NN).

- Start at a randomly chosen vertex
- At each stage visit the closest currently unvisited city

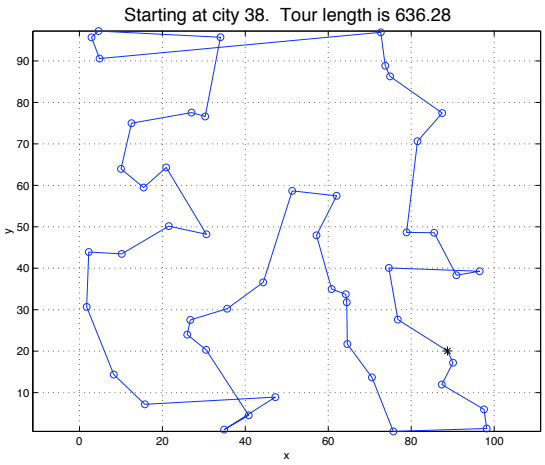
For an n-city instance of TRAVELLING SALESMAN this algorithm takes time $O(n^2)$.

For any instance I, let $NN(I)$ be the length of the tour found by NN and let $OPT(I)$ be the length of the optimal tour. Then $NN(I)/OPT(I)$ is a measure of how good this algorithm is on a given instance.

Unfortunately this is not very good.

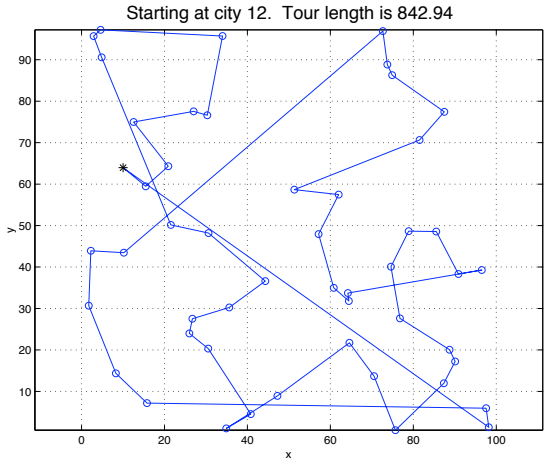
A geometric instance of NN

The best case gave a tour of length 636.28



A geometric instance of NN

The worst case gave a tour of length 842.94



Approximation algorithms

Theorem For any constant $k > 1$ there are instances of TRAVELLING SALESMAN such that $NN(I) \geq k\ OPT(I)$.

Even more seriously this is not just because NN is not sufficiently sophisticated — we cannot expect good behaviour from *any* polynomial time heuristic.

Theorem Suppose A is a polynomial time approximation algorithm for TRAVELLING SALESMAN such that $A(I) \leq k\ OPT(I)$ for some constant k. Then there is a polynomial time algorithm to solve TRAVELLING SALESMAN.

Therefore it seems hopeless to try to find decent approximation algorithms for TRAVELLING SALESMAN.

Minimum spanning tree

Suppose that we have an instance I of TRAVELLING SALESMAN that is symmetric and satisfies the triangle inequality. Then the following algorithm called MST is guaranteed to find a tour that is at most *twice* the optimal length.

- Find a minimum spanning tree for I
- Do a depth-first search on the tree
- Visit the vertices in order of discovery time

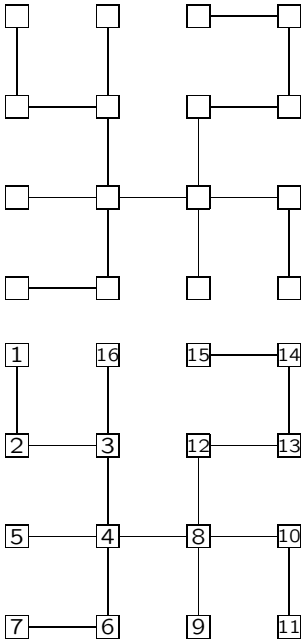
Then

$$MST(I) \leq 2\ OPT(I).$$

In order to see why this works, we first observe that removing one edge from the optimal tour yields a spanning tree for I, and therefore the weight of the minimum spanning tree is less than the length of the shortest tour.

Search the tree ...

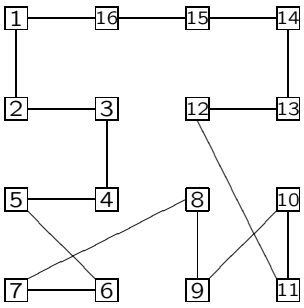
Perform a depth first search on the minimum spanning tree.



... and take shortcuts

If we were to simply follow the path of the depth-first search algorithm — including the backtracking — we would walk along each edge exactly once in each direction, creating a tour that has length exactly twice the weight of the minimum spanning tree, but is illegal because it visits some vertices twice.

The simple solution is to just take "shortcuts" according to the ordering of the vertices.



Coalesced simple paths

The method of coalesced simple paths uses a greedy method to build up a tour edge by edge. At every stage the "partial tour" is a collection of simple paths.

- Sort the edges into increasing weight
- At each stage add the lowest weight edge possible without creating a cycle or a vertex of degree 3.
- Join the ends of the path to form a cycle

This algorithm proceeds very much like Kruskal's algorithm, but the added simplicity means that the complicated **union-find** data structure is unnecessary.

Insertion methods

There is a large class of methods called *insertion methods* which maintain a closed cycle as a partial tour and at each stage of the procedure *insert* an extra vertex into the partial tour.

Suppose that we are intending to insert the new vertex x into the partial tour C (called C because it is a cycle).

In turn we consider each edge $\{u, v\}$ of the partial tour C, and we find the edge such that

$$d(u, x) + d(x, v) - d(u, v)$$

is a minimum.

Then the edge $\{u, v\}$ is deleted, and edges $\{u, x\}$ and $\{x, y\}$ added, hence creating a tour with one additional edge.

Three insertion techniques

Random insertion

At each stage the next vertex x is chosen randomly from the untouched vertices.

Nearest insertion

At each stage the vertex x is chosen to be the one closest to C.

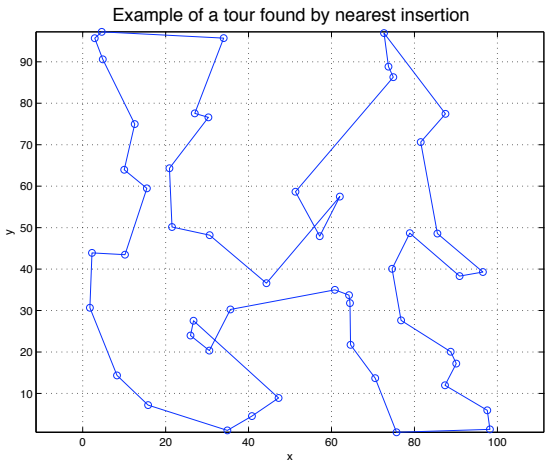
Farthest insertion

At each stage the vertex x is chosen to be the one farthest from C.

(In all three insertion methods the vertex x is chosen first and then it is inserted in the *best* position.)

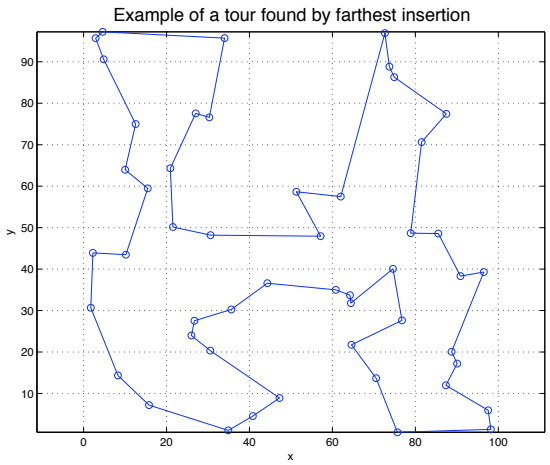
Tour found by nearest insertion

Nearest insertion tours ranged from 631 to 701 on the above example.



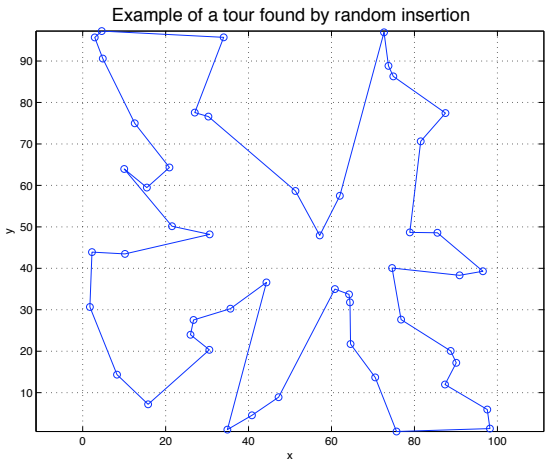
Tour found by farthest insertion

Farthest insertion tours ranged from 594 to 679 on the above example.



Tour found by random insertion

Random insertion tours ranged from 607 to 667 on the above example.



A fourth insertion technique

Cheapest insertion

This method is a bit more expensive than the other methods in that we search through all the edges $\{u, v\}$ in C and all the vertices $x \notin C$ trying to find the vertex and edge which minimizes

$$d(u, x) + d(x, v) - d(u, v)$$

The other three methods can all be programmed in time $O(n^2)$ whereas this method seems to require at least an additional factor of $\lg n$.

Nearest insertion and cheapest insertion can be shown to produce tours of length no greater than twice the optimal tour length by their close relationship to minimum spanning tree algorithms.

Iterative Improvement

One common feature of the tours produced by the greedy heuristics that we have seen is that it is immediately easy to see how they can be improved, just by changing a few edges here and there.

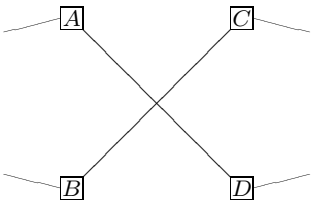
The procedure of *iterative improvement* refers to the process of starting with a feasible solution to a problem and changing it slightly in order to improve it.

An iterative improvement algorithm involves two things

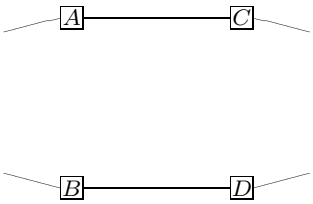
- A rule for changing one feasible solution to another
- A *schedule* for deciding which moves to make

Improving TRAVELLING SALESMAN tours

A basic move for TRAVELLING SALESMAN problems involves deleting two edges in the tour, and replacing them with two non-edges, as follows.



Suppose the tour runs AD, $D \rightsquigarrow C$, CB, $B \rightsquigarrow A$. Then deleting AD and CB, we replace them with AC and DB.



2-OPT

Consider now an iterative improvement algorithm that proceeds by examining every pair of edges, and performing an exchange if the tour can be improved.

This procedure must eventually terminate, and the resulting tour is called *2-optimal*.

There are more complicated "moves" that involve deleting 3 edges and reconnecting the tour, and in general deleting k edges and then reconnecting the tour.

A tour that cannot be improved by a k edge exchange is called k-optimal. In practice it is rare to compute anything beyond 2-optimal or 3-optimal tours.

A state space graph

We can view this process in a more abstract sense as a heuristic search on a huge graph called the *state space graph*.

We define the state space graph $S(I)$ for an instance of TRAVELLING SALESMAN as follows.

> The vertices of $S(I)$ consist of all the feasible tours for the instance I.
> Two feasible tours T_1 and T_2 are neighbours if they can be obtained from each other by the edge exchange process above.

Each vertex T has a cost $c(T)$ associated with it, being the length of the tour T.

To completely solve TRAVELLING SALESMAN requires finding which of the $(n-1)!$ vertices of $S(I)$ has the lowest cost.

Searching the state space graph

In general $S(I)$ is so vast that it is totally impossible to write down the entire graph.

The greedy insertion methods all provide us with a single vertex in $S(I)$ (a single tour), and the iterative improvement heuristics all involve doing a walk in $S(I)$ moving along edges from tour to neighbouring tour attempting to find the lowest cost vertex.

In this type of state space searching we have the concept of a "current" tour T and at each stage of the search we generate a neighbour T' of T and decide whether the search should proceed to T' or not.

Hill-climbing

The simplest heuristic state-space search is known as *hill-climbing*.

The rule for proceeding from one state to another is very easy

- Systematically generate neighbours T' of T and move to the first neighbour of lower cost than T.

This procedure will terminate when T has no neighbours of lower cost — in this case T is a 2-optimal tour.

An obvious variant of this is to always choose the *best* move at each step.

A local optimum

A hill-climb will always finish on a vertex of lower cost than all its neighbours — such a vertex is a *local minimum*.

Unfortunately the state space graph has an enormous number of local minima, each of them possibly tremendously different from the global minimum.

If we mentally picture the state space graph as a kind of "landscape" where costs are represented by heights, then $S(I)$ is a savagely jagged landscape of enormously high dimension.

Hill climbing merely moves directly into the nearest local optimum and cannot proceed from there.

State-space for DOMINATING SET

We can apply similar methods to the graph domination problem provided that we define the state-space graph carefully.

Suppose that we are trying to see whether a graph G has a dominating set of size k. Then the "states" in the state space graph are all the possible subsets of $V(G)$ of size k. The "cost" of each can be taken to be the number of vertices not dominated by the corresponding k-subset. The solution that we are seeking is then a state of cost 0.

Now we must define some concept of "neighbouring states". In this situation a natural way to define a neighbouring state is the state that results from moving one of the k vertices to a different position.

Heuristic search for graph domination

We can now apply the hill-climbing procedure to this state space graph.

In this fashion the search "wanders" around the state-space graph, but again it will inevitably end up in a local minimum from which there is not escape.

Hill climbing is unsatisfactory because it has no mechanism for escaping locally optimum solutions. Ideally we want a heuristic search technique that tries to improve the current solution but has some method for escaping local optima.

Two techniques that have been proposed and extensively investigated in the last decade or so are

Simulated Annealing
Tabu Search

Annealing

Annealing is a physical process used in forming crystalline solids.

At a high temperature the solid is molten, and the molecules are moving fast and randomly. If the mixture is very gradually cooled, then as the temperature drops the mixture becomes more ordered, with molecules beginning to align into a crystalline structure. If the cooling is sufficiently slow, then at freezing point the resulting solid has a perfect regular crystalline structure.

The crystalline structure has the lowest potential energy, so we can regard the process as trying to find the configuration of a group of molecules with a global minimum potential energy.

Annealing is successful because the slow cooling allows the physical system to escape from local minima.

Simulated annealing

Simulated annealing is an attempt to apply these same principles to problems of combinatorial optimization.

For TRAVELLING SALESMAN we regard the optimal tour as the "crystal" for which we are searching and the other tours, being less perfect, as the flawed semi-molten crystals, while for GRAPH DOMINATION we regard the states with cost 0 (that is, genuine dominating sets) as the "crystals".

The overall structure of simulated annealing is:

- Randomly generate a neighbour T' of the vertex T
- If $c(T') \leq c(T)$ then accept the move to T'
- If $c(T') > c(T)$ then *with a certain probability* p accept the move to T'

The probability p of accepting an uphill move is dynamically altered throughout the algorithm.

Uphill moves

Dynamically altering p is usually done by maintaining a *temperature* variable t which is gradually lowered throughout the operation of the algorithm, and applying the following rules.

Suppose that we are currently at a vertex T with a cost $c(T)$. The randomly generated neighbour T' of T has cost $c(T')$ and so if the move is made then the difference will be

$$\Delta c = c(T') - c(T)$$

Then the probability of accepting the move is taken to be

$$p = \exp(-\Delta c/t)$$

If $\Delta c < 0$, then $p > 1$, so this corresponds to accepting all moves to a lower cost neighbour.

Otherwise, if t is high, then $-\Delta c/t$ is very small and $p \approx 1$. If t is small then $-\Delta c/t$ will be large and negative and $p \approx 0$.

Cooling schedule

Therefore at *high* temperatures, almost all moves are accepted, good or bad, whereas as the temperature reduces, fewer bad moves are accepted and the procedure settles down again. When $t \approx 0$ then the procedure reverts to a hill-climb.

The value of the the initial temperature and the way in which it is reduced is called a *cooling schedule*:

- Start with some initial temperature t_0
- Perform N iterations at each temperature
- Reduce the temperature by a constant multiplicative factor $t \leftarrow Kt$

For example the values $t_0 = 1$, $N = 1000$, $K = 0.95$ might be suitable.

Performance of this algorithm is highly problem-specific and cooling schedule-specific.

How good is it?

Simulated annealing has had success in several areas of combinatorial optimization, particularly in problems with continuous variables.

In general it seems to work considerably better than hill-climbing, though it is not clear whether it works much better than *multiple* hill-climbs.

Each of these combinatorial optimization heuristics has their own adherents, and something akin to religious wars can erupt if anyone is rash enough to say "X is better than Y".

Experimentation is fraught with problems also, in that an empirical comparison of techniques depends so heavily on the test problems that almost any desired result can be convincingly produced by careful enough choice.

Nonetheless the literature is liberally dotted with "An empirical comparison of ... and ...".

Tabu search

The word *tabu* (or taboo) means something prohibited or forbidden.

Tabu search is another combinatorial search heuristic that combines some of the features of hill-climbing *and* simulated annealing. However it can only be used in slightly more restricted circumstances.

Tabu search attempts to combat two obvious weaknesses of hill-climbing and simulated annealing — the inability of hill-climbing to escape from local minima, and the early waste of time in simulated annealing where the temperature is very high and the search is proceeding almost randomly with almost no pressure to improve the solution quality.

Tabu search attempts to spend almost all of its time close to local minima, while still having the facility to escape them.

The basic idea

The basic idea of a tabu search is that it always maintains a *tabu list* detailing the last h vertices that it has visited.

- Select the *best* possible neighbour T' of T.
- If T' is not on the tabu list, then move to it and update the tabu list accordingly.

We notice that the tabu search is very aggressive — it always seeks to move in the best possible direction. Without a tabu list this process would always end in a cycle of length 2, with the algorithm flipping between a local minimum and its nearest neighbour.

The tabu list prevents the search from immediately returning to a recently visited tour and (hopefully) forces it to take a different track out of that local minimum.

Practical considerations

The main problem of tabu search is that at each iteration it requires complete enumeration of the neighbourhood of a vertex — this may be prohibitively expensive.

Similarly to choosing a cooling schedule for simulated annealing, a tabu schedule must be chosen for tabu search. It is important to choose the length of the tabu list very carefully — this is again very problem-specific.

On the positive side, tabu search manages to examine many more "close-to-optimum" solutions than simulated annealing.

Another positive feature of tabu search is that provided care is taken to prevent cycling, the search can be left running for as long as resources allow, while the length of a simulated annealing run is usually fixed in advance.

Tabu search for graph domination

The best dominating sets for the football pool graphs were largely constructed by tabu search techniques, together with a mathematical construction that reduces the search to smaller but denser graphs.

There are many practical considerations in implementing a tabu search — firstly it is necessary to be very efficient in evaluating the cost function on the neighbouring states.

There are also many variants on a tabu search — for example, only searching a portion of the neighbourhood of a given state, maybe by concentrating on the moves that are likely to result in an improvement rather than all possible moves.

Genetic algorithms

Genetic algorithms provide an entirely different approach to the problems of combinatorial optimization.

Like simulated annealing, genetic algorithms try to model a physical process that improves "quality" — in this case the physical process is evolution.

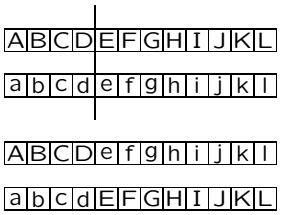
A genetic algorithm proceeds by maintaining a pool containing many feasible solutions, each with its associated fitness.

At each iteration, a new population of solutions is created by breeding and mutation, with the fitter solutions being more likely to procreate.

A glimpse of GAs

Each solution is encoded as a string.

Breeding two strings involves selecting a position at random, breaking the strings into a head and tail at that point, and swapping tails. This operation is referred to as *cross-over*:



Parents are chosen in direct proportion to their fitness so that the fitter strings breed more often.

Mutation involves arbitrarily altering one of the elements of the string.

As usual there are several parameters to fine-tune the algorithm such as population size, mutation frequency and so on.

GAs for combinatorial optimization?

Although GAs have their adherents it may not be easy to adapt them successfully to combinatorial optimization problems such as TRAVELLING SALESMAN and GRAPH DOMINATION.

The problem here seems to be that there is no way one can arbitrarily combine two tours to create a third tour — simply hacking two tours apart and joining the bits together will not work in general.

Similarly, it is hard to come up with a good representation for a candidate dominating set in such a way that arbitrary cross-over does not destroy all its good properties.

The crucial distinction seems to be that hill-climbing, simulated annealing and tabu search are all *local search methods* whereas a genetic algorithm is not.

Recommended reading: CLRS, Chapter 35

Summary

1. Greedy algorithms solve optimization problems by searching the best local direction. They are applied in the Activity selection problem, Huffman coding and some graph algorithms.

2. Vertex cover, travelling salesman and the 0-1 knapsack problem are all instances of NP-complete problems, (i.e. for which no feasible algorithm is known).

3. A dynamic programming solution exists for the 0-1 knapsack problem.

4. Linear programmes are problems of optimizing a linear cost function, subject to linear constraints. They can be applied in many optimization problems, and may be solved by the simplex algorithm.

Summary cont.

5. Heuristic algorithms can be applied to approximate optimal solutions to geometric instances of the travelling salesman problem.

6. Other heuristic methods include hill-climbing, simulated annealing, tabu search and genetic algorithms.

The End?