

COMP20007 Design of Algorithms

Brute Force Methods

Lars Kulik

Lecture 5

Semester 1, 2023

Brute Force Algorithms



Straightforward problem solving approach, usually based directly on the problem's statement.

Exhaustive search for solutions is a prime example.

- Selection sort $\Theta(n^2)$
- String matching $\Theta(n)$
- Closest pair $\Theta(n^2)$
- Exhaustive search for combinatorial solutions
- Graph traversal $\Theta(mn)$

Example: Selection Sort

We already saw this algorithm:

```
function SELSORT( $A[0..n-1]$ )  
  for  $i \leftarrow 0$  to  $n-2$  do  
     $min \leftarrow i$   
    for  $j \leftarrow i+1$  to  $n-1$  do  
      if  $A[j] < A[min]$  then  
         $min \leftarrow j$   
    swap  $A[i]$  and  $A[min]$ 
```

The complexity is $\Theta(n^2)$.

We shall soon meet better sorting algorithms.

Properties of Sorting Algorithms

A sorting algorithm is

- in-place ^{extra} if it does not require additional memory except, perhaps, for a few units of memory.
- stable if it preserves the relative order of elements that have identical keys.
- input-insensitive if its running time is fairly independent of input properties other than size.

Properties of Selection Sort

While running time is quadratic, selection sort makes only about n exchanges.

So: A good algorithm for sorting small collections of large records.

In-place? ✓

Stable? ✓

Input-insensitive? ✗



Brute Force String Matching



Pattern p : A string of m characters to search **for**.

Text t : A long string of n characters to search **in**.

for $i \leftarrow 0$ to $n - m$ **do**

$j \leftarrow 0$

while $j < m$ and $p[j] = t[i + j]$ **do**

$j \leftarrow j + 1$

if $j = m$ **then**

return i

return -1

$O(mn)$

m : size of pattern

n : size of long string that
searched in.

Analysing Brute Force String Matching

For each of $n - m + 1$ positions in t , we make up to m comparisons.

Assuming n is much larger than m , this means $O(mn)$ comparisons.

However, for random text over a reasonably large alphabet (as in English), the average running time is linear in n .

There are better algorithms, in particular for smaller alphabets such as binary strings or strings of DNA nucleobases.

But for many purposes, the brute-force algorithm is acceptable.

Later we shall see more sophisticated string search.

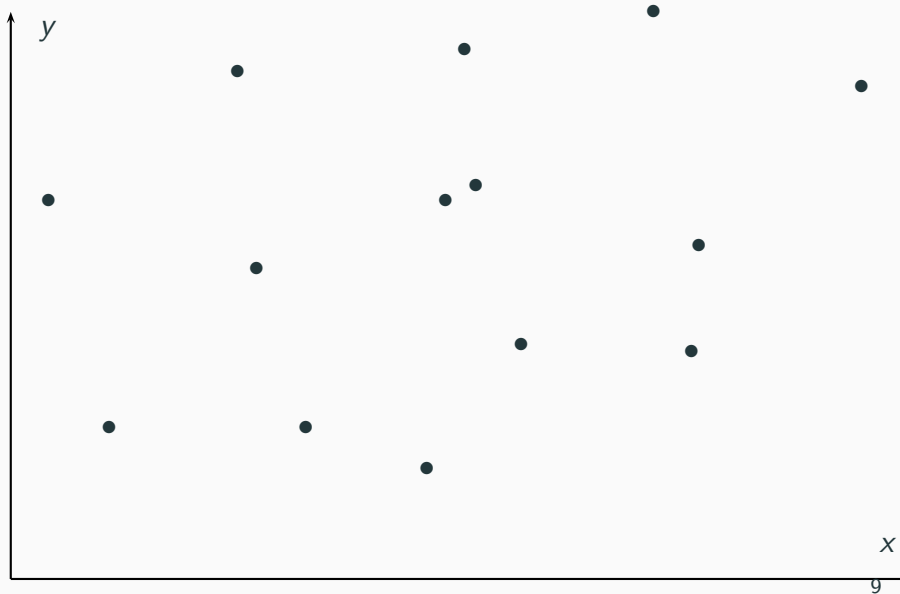
Brute Force Geometric Algorithms: Closest Pair

Problem: Given n points in k -dimensional space, find a pair of points with minimal separating Euclidean distance.

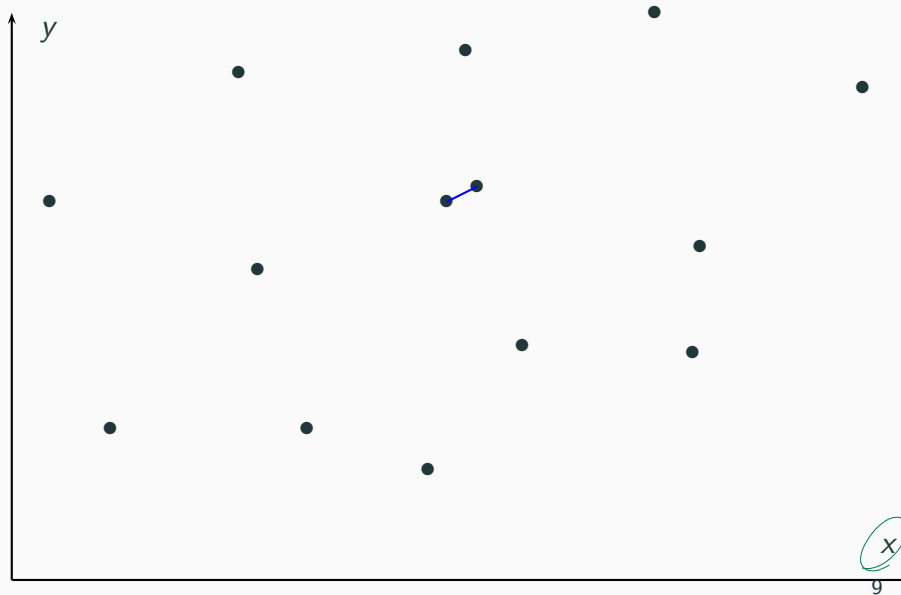
The brute force approach considers each pair in turn (except that once it has found the distance from x to y , it does not need to consider the distance from y to x).

For simplicity, we look at the 2-dimensional case, the points being $(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})$.

The Closest Pair Problem (Two-Dimensional Case)



The Closest Pair Problem (Two-Dimensional Case)



Brute Force Geometric Algorithms: Closest Pair

$min \leftarrow \infty$

for $i \leftarrow 0$ to $n - 2$ **do**

for $j \leftarrow i + 1$ to $n - 1$ **do**

$d \leftarrow \text{sqrt}((x_i - x_j)^2 + (y_i - y_j)^2)$

if $d < min$ **then**

$min \leftarrow d$

$p_1 \leftarrow i$

$p_2 \leftarrow j$

return p_1, p_2

Analysing the Closest Pair Algorithm

It is not hard to see that the algorithm is $\Theta(n^2)$.

Note, however, that we can speed up the algorithm considerably, by utilising the monotonicity of the square root function.

How?

Does this contradict the $\Theta(n^2)$ claim?



Later we shall see how a clever divide-and-conquer approach leads to a $\Theta(n \log n)$ algorithm for this problem.

Brute Force Summary

Simple, easy to program, widely applicable.

Standard approach for small tasks.

Reasonable algorithms for some problems.

But: Generally inefficient—does not scale well.

Use brute force for prototyping, or when it is known that input remains small.

Exhaustive Search

全周搜索

Problem type:

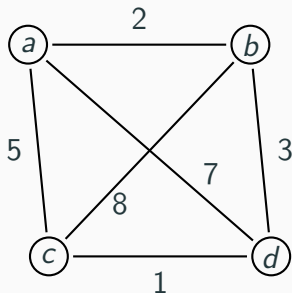
- Combinatorial decision or optimization problems
- Search for an element with a particular property
- Domain grows exponentially, for example all permutations

The brute-force approach—generate and test:

- Systematically construct all possible solutions
- Evaluate each, keeping track of the best so far
- When all potential solutions have been examined, return the best found

Example 1: Travelling Salesperson (TSP)

Find the shortest **tour** (visiting each node exactly once before returning to the start) in a weighted undirected graph.



$$a - b - c - d - a : 18$$

$$a - b - d - c - a : 11$$

$$a - c - b - d - a : 23$$

$$a - c - d - b - a : 11$$

$$a - d - b - c - a : 23$$

$$a - d - c - b - a : 18$$

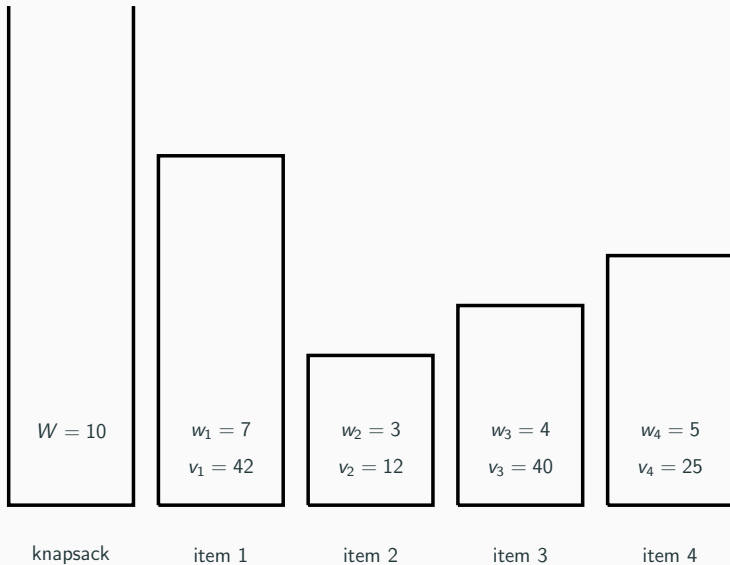
Example 2: Knapsack

Given n items with

- weights: w_1, w_2, \dots, w_n
- values: v_1, v_2, \dots, v_n
- knapsack of capacity W

find the most valuable selection of items that will fit in the knapsack.

Example 2: Knapsack



Example 2: Knapsack

Set	Weight	Value
\emptyset	0	0
{1}	7	42
{2}	3	12
{3}	4	40
{4}	5	25
{1, 2}	10	54
{1, 3}	11	NF
{1, 4}	12	NF

Set	Weight	Value
{2, 3}	7	52
{2, 4}	8	37
{3, 4}	9	65
{1, 2, 3}	14	NF
{1, 2, 4}	15	NF
{1, 3, 4}	16	NF
{2, 3, 4}	12	NF
{1, 2, 3, 4}	19	NF

NF means “not feasible”: exhausts the capacity of the knapsack.

Later we shall consider a better algorithm based on **dynamic programming**.

Comments on Exhaustive Search

Exhaustive search algorithms have acceptable running times only for very small instances.

In many cases there are better alternatives, for example, Eulerian tours, shortest paths, minimum spanning trees, ...

For some problems, it is **known** that there is essentially no better alternative.

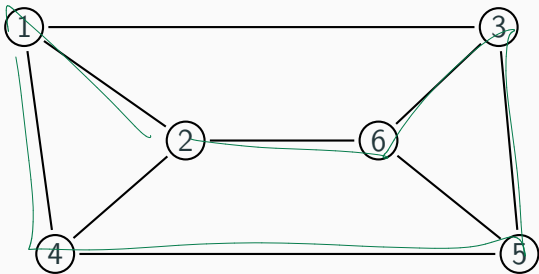
For a large class of important problems, **it appears** that there is no better alternative, but we have no proof either way.

Hamiltonian Tours

The Hamiltonian tour problem is this:

In a given N-P hard undirected graph, is there a simple tour (a path that visits each **node** exactly once, except it returns to the starting node)?

Is there a Hamiltonian tour of this graph?



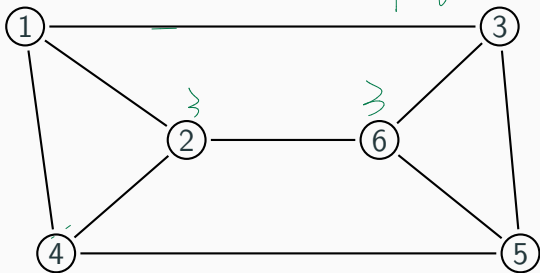
Eulerian Tours

The Eulerian tour problem is this:

In a given ^Pundirected graph, is there a path which visits each edge exactly once?

Is there a Eulerian tour of this graph?

No.



$\# \text{ odd} \geq 2 \Rightarrow \text{no}$

Hard and Easy Problems

Recall that by a **problem** we usually mean a parametric problem: an infinite family of problem “instances”.

The Hamiltonian Tour problem and the Eulerian Tour problem look very similar, but one is hard and the other is easy. We will see more examples of this phenomenon later.

For many **optimization** problems we do not know of solutions that are essentially better than exhaustive search (a whole raft of **NP-complete** problems, including TSP and knapsack).

In those cases we try to find **approximation algorithms** that are fast and close to the optimal solution.

We return to this idea later.

Next Up

Graphs.

COMP20007 Design of Algorithms

Graphs and Graph Concepts

Lars Kulik

Lecture 6

Semester 1, 2023

Graphs Again

One instance of the **exhaustive search** paradigm is **graph traversal**.

After this lecture we shall look at two ways of systematically visiting every node of a graph, namely **depth-first** and **breadth-first** search.

These two methods of graph traversal form the backbone of a surprisingly large number of useful graph algorithms.

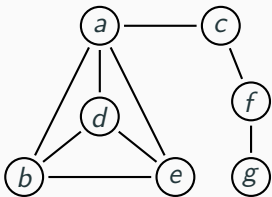
Graph algorithms are useful for a large number of practical problems: network design, flow design, planning, scheduling, route finding, and other logistics applications.

Graphs, Mathematically

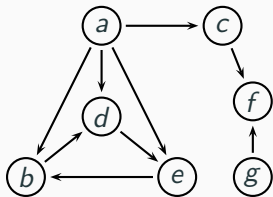
$$G = \langle V, E \rangle$$

V : Set of **nodes** or **vertices**

E : Set of **edges** (a binary relation on V)



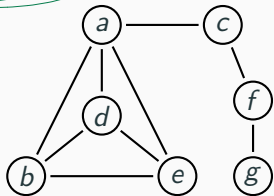
$$\begin{aligned} V &= \{a, b, c, d, e, f, g\} \\ E &= \text{symmetric closure of} \\ &\quad \{(a, b), (a, c), (a, d), \\ &\quad (a, e), (b, d), (b, e), \\ &\quad (c, f), (d, e), (f, g)\} \end{aligned}$$



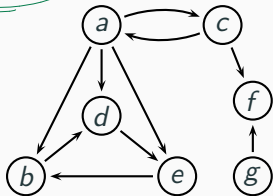
$$\begin{aligned} V &= \{a, b, c, d, e, f, g\} \\ E &= \{(a, b), (a, c), (a, d), \\ &\quad (a, e), (b, d), (c, f), \\ &\quad (d, e), (e, b), (f, g)\} \end{aligned}$$

Graph Concepts

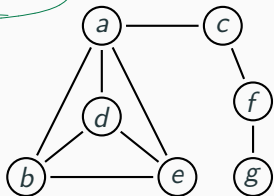
Undirected:



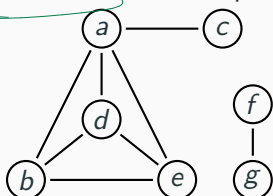
Directed:



Connected:



Not connected, two components:



More Graph Concepts: Degrees of Nodes

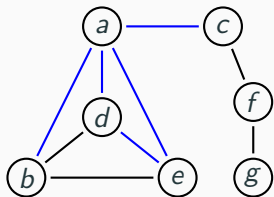
If $(u, v) \in E$ then u and v are **adjacent**, or **neighbours**.

(u, v) **connects** u and v , and are u and v **incident** to (u, v) .

The **degree** of node v is the number of edges incident to v .

For directed graphs, we talk about v 's **in-degree** (number of edges going **to** v) and its **out-degree** (number of edges going **from** v).

More Graph Concepts: Paths and Cycles



Path b, a, d, e, a, c shown in blue

A **path** in $\langle V, E \rangle$ is a sequence of nodes v_0, v_1, \dots, v_k from V , so that $(v_i, v_{i+1}) \in E$.

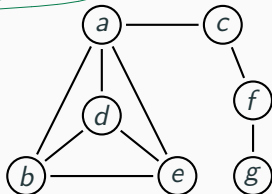
The path v_0, v_1, \dots, v_k has **length** k .

A **simple path** is one that has no repeated nodes.

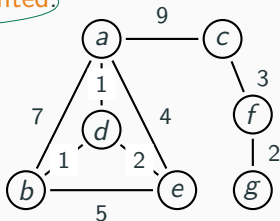
A **cycle** is a simple path, except that $v_0 = v_k$, that is, the last node is the same as the first node.

More Graph Concepts

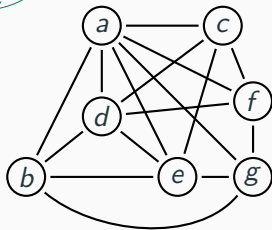
Unweighted:



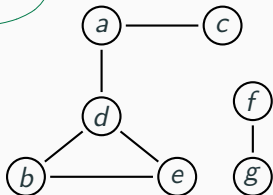
Weighted:



Dense:

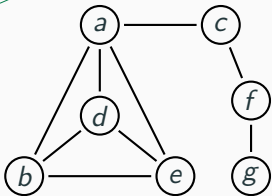


Sparse:

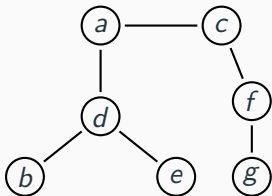


More Graph Concepts

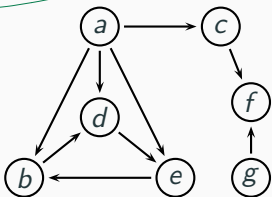
Cyclic:



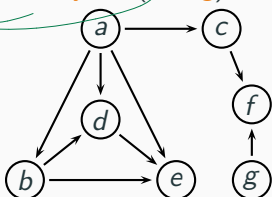
Acyclic (actually, a tree):



Directed cyclic:



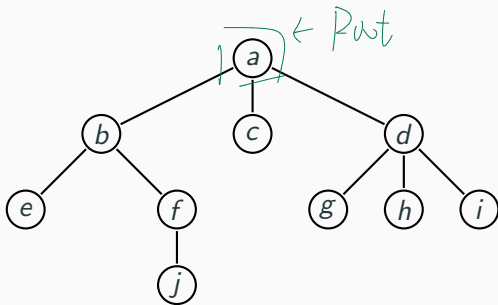
Directed acyclic (a dag):



Rooted Trees

A (free) **tree** is a connected acyclic graph.

A **rooted tree** is a tree with one node identified as special. Every other node is reachable from the root node.



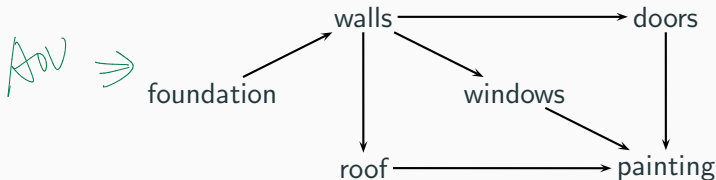
When the root is removed, a set of rooted (sub-)trees remain.

We should draw the rooted tree as a directed graph, but usually we instead rely on the layout: “parents” sit higher than “children”.

Modelling with Graphs

Graph algorithms are of great importance because so many different problem types can be abstracted to graph problems.

For example, directed graphs are central in scheduling problems:



Modelling with Graphs

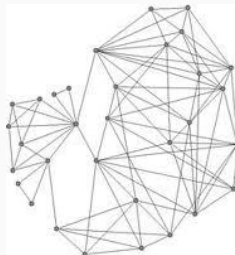
Graphs find use in all sorts of modelling.

Assume you want to invite friends to dinner and you have k tables available.

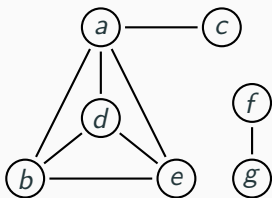
Some guests dislike some of the others; thus we need a seating plan that avoids placing foes at the same table.

The natural model is an undirected graph, with a node for each guest, and an edge between any two guests that don't get along.

This reduces your problem to the “graph k -colouring problem”: find, if possible, a colouring of nodes so that all connected nodes have a different colour.



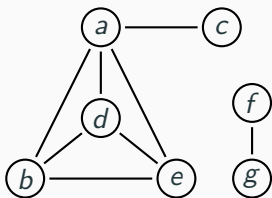
Graph Representations, Undirected Graphs



	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>	<i>g</i>
<i>a</i>	0	1	1	1	1	0	0
<i>b</i>	1	0	0	1	1	0	0
<i>c</i>	1	0	0	0	0	0	0
<i>d</i>	1	1	0	0	1	0	0
<i>e</i>	1	1	0	1	0	0	0
<i>f</i>	0	0	0	0	0	0	1
<i>g</i>	0	0	0	0	0	1	0

The **adjacency matrix** for the graph.

Graph Representations, Undirected Graphs



vector.

$a \rightarrow b \rightarrow c \rightarrow d \rightarrow e$

$b \rightarrow a \rightarrow d \rightarrow e$

$c \rightarrow a$

$d \rightarrow a \rightarrow b \rightarrow e$

$e \rightarrow a \rightarrow b \rightarrow d$

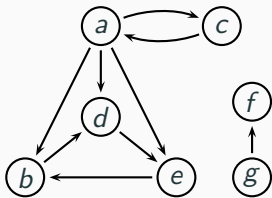
$f \rightarrow g$

$g \rightarrow f$

The adjacency list representation.

(Assuming lists are kept in sorted order.)

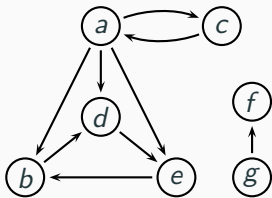
Graph Representations, Directed Graphs



	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>	<i>g</i>
<i>a</i>	0	1	1	1	1	0	0
<i>b</i>	0	0	0	1	0	0	0
<i>c</i>	1	0	0	0	0	0	0
<i>d</i>	0	0	0	0	1	0	0
<i>e</i>	0	1	0	0	0	0	0
<i>f</i>	0	0	0	0	0	0	0
<i>g</i>	0	0	0	0	0	1	0

The **adjacency matrix** for the graph.

Graph Representations, Directed Graphs



<i>a</i>	$\rightarrow b \rightarrow c \rightarrow d \rightarrow e$
<i>b</i>	$\rightarrow d$
<i>c</i>	$\rightarrow a$
<i>d</i>	$\rightarrow e$
<i>e</i>	$\rightarrow b$
<i>f</i>	
<i>g</i>	$\rightarrow f$

The **adjacency list** representation.

Graph Representations

Each representation has advantages and disadvantages.

Think of some!



Graph traversal, in which we get down to the nitty-gritty details of graph algorithms.