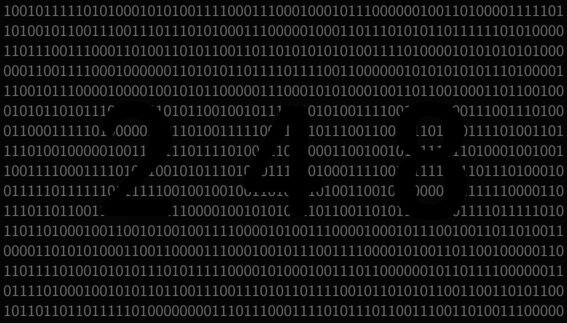


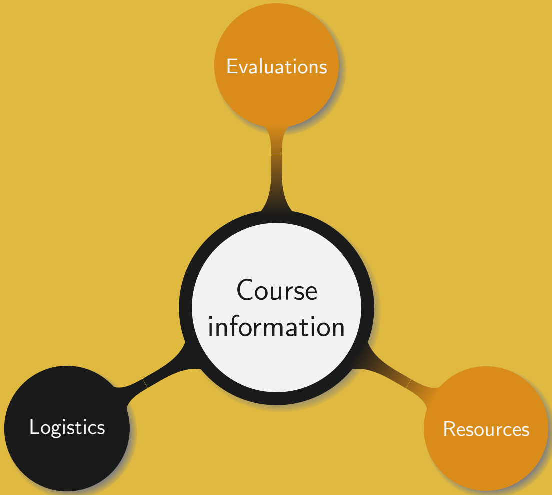
Introduction to Algorithms

Manuel – Fall 2021

Table of contents

0. Course information

Chapter organisation



Basic information

Teaching team:

- Instructor: Manuel (charlem@sjtu.edu.cn)
- Teaching assistants:
  - Haoxiang (iniesta8@sjtu.edu.cn,)
  - Xinmiao (yuxinmiao@sjtu.edu.cn)

Important rules:

- When contacting a TA for an important matter, `CC` the instructor
- Prepend [VE477] to the subject, e.g. `Subject: [VE477] Grades`
- Use SJTU jBox service to share large files (> 2 MB)

Never send large files by email

Course schedule

Course arrangements:

- Lectures:
  - Tuesday 8:00 – 9:40
  - Thursday 8:00 – 9:40
  - Friday 8:00 – 9:40 (weeks 8 and 12)
- Labs:
  - Wednesday 18:20 – 20:20
  - Thursday 18:20 – 20:20
- Manuel's office hours: Appointment (TBD)
- TAs' office hours: TBA

Course objectives

Main goals of this course:

- Become familiar with the most common problems and paradigms
- Understand how to properly analyse and abstract a problem
- Identify or design clear and efficient algorithms to solve a problem

Solve a problem, then assess the solution validity, quality, and efficiency

Course workflow

Learning strategy:

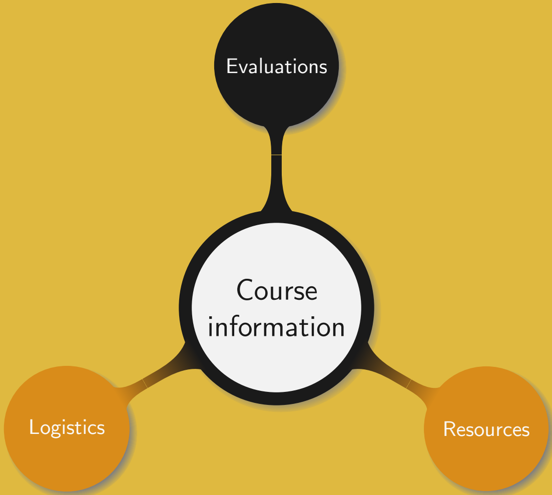
- Course side:
  1. Understand the basic concept of algorithmic
  2. Know the most common problems and their solutions
  3. Get an overview of the wide applications of algorithms
- Personal side:
  1. Read and write code
  2. Relate known strategies to new problems
  3. Perform extra research

Course outcomes

Detailed goals:

- Be able to write clean and clear pseudocode
- Be proficient at using all the basic algorithm paradigms
- Be able to assess the difficulty of a given problem
- Develop critical thinking abilities
- Know when and how to apply dynamic programming
- Have a precise idea of the pros and cons for common data structures
- Know how to efficiently solve common mathematical problems
- Have a basic idea on how to design multi-threaded algorithms
- Be able to efficiently implement the most common algorithms

Chapter organisation



Assignments

Homework:

- Total: 8
- Content: basic concepts, critical thinking, prove results

Labs:

- Total: $8 + 4$
- Content: implement common algorithms, learn Python or OCaml

Project:

- Total: 1, split into three phases
- Content: write a catalog of the most common algorithms

Challenge: TBA

Grading policy

Grade weighting:

- Homework: 15%
- Projects: 25%
- Labs: 10%
- Midterm exam: 25%
- Final exam: 25%

Assignment submissions:

- Bonus: +10% for a work fully written in LaTeX, bounded to 100%
- Penalty: −10% for a work not written in a neat and legible fashion
- Late policy: −10% per day, not accepted after three days

Grades will be curved with the median in the range $[\![B, B+]\!]$

Honor Code

General rules:

- Not allowed:
  - Reuse the code or work from other students
  - Reuse the code or work from the internet
  - Give too many details on how to solve an exercise
- Allowed:
  - Share ideas and understandings on the course
  - Provide general directions on where or how to find information

Honor Code

Documents allowed during the exams:

- The lecture slides with **notes on them** (paper or electronic)
- A mono or bilingual dictionary

Group works:

- Every student in a group is responsible for his group submission
- If a student breaks the Honor Code, the whole group is sent to Honour Council

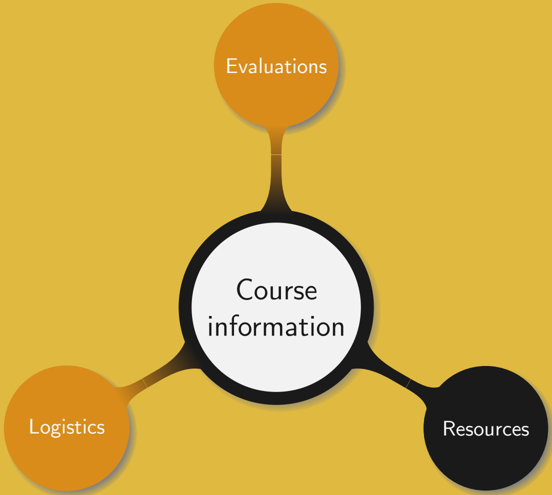
Special circumstances

Contact us as early as possible when:

- Facing special circumstances, e.g. full time work, illness, etc.
- Feeling late in the course
- Feeling to work hard without any result

Any late request will be rejected

Chapter organisation



Canvas

On Canvas platform:

- Course materials:
  - Syllabus
  - Lecture slides
  - Homework
  - Labs
  - Projects
  - Challenges
- Course information:
  - Announcements
  - Notifications
  - Grades
  - Surveys

References

Places to find information:

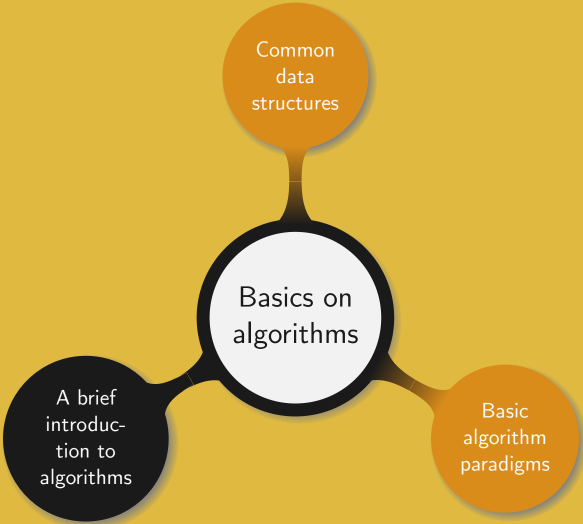
- *Algorithm Design*, J. Kleinberg and E. Tardos
- *Introduction to Algorithms*, H. Cormen, C. Leiserson, R. Rivest, and C. Stein
- *The Art of Computer Programming*, D. Knuth
- Piazza
- Search information online, i.e. $\{\textit{internet} \setminus \{\textit{non-English websites}\}\}$

Key points

- Work regularly, do not wait the last minute
- Respect the Honor Code
- Go beyond what is taught
- Do not learn, understand
- Keep in touch with us
- Advice and suggestions are always much appreciated

1. Basics on algorithms

Chapter organisation



Algorithm

An Algorithm is a recipe telling the computer how to solve a problem

Example. Detail an algorithm to prepare a jam sandwich

Actions: cut, listen, spread, sleep, read, take, eat, dip
Things: knife, guitar, bread, honey, jam jar, sword

Algorithm. (*Sandwich making*)

```
Input   : 1 bread, 1 jamjar, 1 knife
Output: 1 jam sandwich
take the knife and cut 2 slices of bread;
dip the knife into the jamjar;
spread the jam on the bread, using the knife;
assemble the 2 slices together, jam on the inside;
```

A more formal view

An algorithm systematically solves a *well-defined* problem:

- The *Input* is clearly expressed
- The *Output* solves the problem
- The *Algorithm* provides a precise step-by-step procedure starting from the Input and leading to the Output

Algorithms can be described using one of the three following ways:

- English
- Pseudocode
- Programming language

A first example

Algorithm. (*Insertion Sort*)

```
**Input**  : $a_1, \dots, a_n$, $n$ unsorted elements
**Output:** the $a_i, 1 \le i \le n$, in increasing order
for $j \leftarrow 2$ to $n$ do
    $i \leftarrow 1$;
    while $a_j > a_i$ do $i \leftarrow i + 1$;
    $m \leftarrow a_j$;
    for $k \leftarrow 0$ to $j - i - 1$ do $a_{j-k} \leftarrow a_{j-k-1}$;
    $a_i \leftarrow m$
end for
return $(a_1, \dots, a_n)$
```

A first problem

Example. A robot arm solders chips on a board in n contact points. We want to minimize the time to attach a chip to the board, knowing that

- The arm moves at constant speed;
- Once a chip has been attached another one is soldered;

Defining the Input and Output:

- Input: a set S of n points in the plane
- Output: the shortest path visiting all the points in S

A first solution

Algorithm. (*Nearest neighbor*)

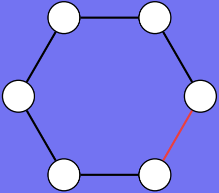
Input : a set $S = \{s_0, \cdots, s_{n-1}\}$ of n points in the plane

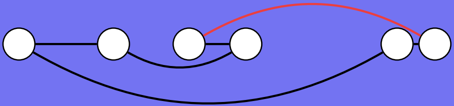
Output: the shortest cycle visiting all the points in S

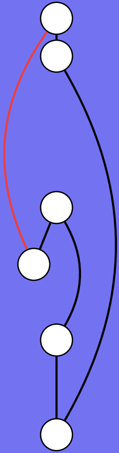
```
$p_0 \leftarrow s_0$;
for $i \leftarrow 1$ to $n - 1$ do
    $p_i \leftarrow$ closest unvisited neighbor to $p_{i-1}$;
    Visit $p_i$;
end for
return $\langle p_0, \ldots, p_{n-1} \rangle$
```

A first failure

How does the nearest neighbor algorithm (1.26) perform in the following three cases?







A second solution

Algorithm. (*Closest pair*)

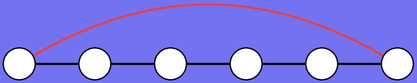
Input : a set S of n points in the plane

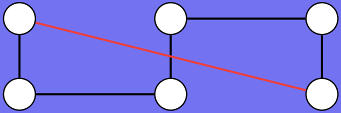
Output: the shortest cycle visiting all the points in S

```
for $i \leftarrow 1$ to $n - 1$ do
    $d \leftarrow \infty$;
    foreach pair of end points $\langle s, t \rangle$ from distinct vertex chains do
        if $dist(s, t) \leq d$ then
            $s_m \leftarrow s$; $t_m \leftarrow t$; $d \leftarrow dist(s, t)$;
        end if
    end foreach
    Connect $s_m$ and $t_m$ by an edge;
end for
return all the points starting from one of the two end points
```

A second failure

Applying the closest pair algorithm (1.28) to the following vertices arrangement yields the two graphs:





Possible strategy to ensure the most optimal path:

- Enumerate all the possible paths
- Select the one that minimizes the total length

For only 20 vertices $20!$ *paths have to be explored*

The first few lessons

A difference:

- Algorithm: always output a correct result
- Heuristic: idea serving as a guide to solve a problem with no guarantee of always providing a good solution

Correctness and efficiency:

- An algorithm working on a set of input does not imply it will work on all instances
- Efficient algorithm totally solving a problem might not exist

Solving problems using algorithms

Common traps when defining the Input and Output:

- Are they precise enough?
- Can all the Input be easily and efficiently generated?
- Could there be any confusion on the expected Output?

Example. For an Output, what does it mean to “find the best route”?

The shortest in distance, the fastest in time, or the one minimizing the number of turns?

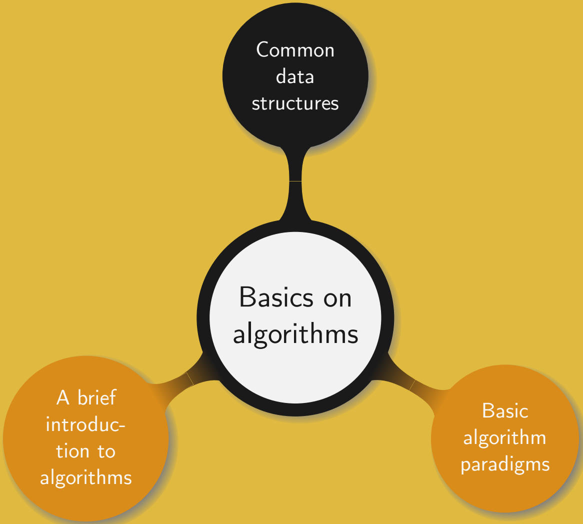
Input and Output must be expressed in simple, precise, and clear terms

Incorrectness

Finding good counter-examples:

- Seek simplicity: make it clear why the algorithm fails
- Think small: algorithms failing for large Input often fail for smaller one
- Test the extremes: study special cases, e.g. inputs equal, tiny, huge...
- Think exhaustively: test whether all the possible cases are covered by the algorithm
- Track weaknesses: check if the underlying idea behind the algorithm has any “unexpected” impact on the output

Chapter organisation


Basics on algorithms

Common data structures

A brief introduction to algorithms

Basic algorithm paradigms

Continuous vs. linked

Data structures can be split into two main categories:

- Continuous: a single piece of memory, e.g. array, matrices, hash tables
- Linked data structures: distinct chunks of memory connected together, e.g. linked list, trees, graph adjacency lists

Choosing an appropriate data structure is of a major importance

Arrays

Each element can be efficiently located using its index:

- Constant access time: each index maps to a memory address
- Space efficiency: no space wasted with links or information on the data
- Memory locality: data is contiguous so *cache* can be used to speed up successive data accesses

The size cannot be easily adjusted during the program's execution

Linked structures

A linked structure is composed of nodes. Each one contains:

- One or more fields on data
- A pointer to at least another node

The most common operations are:

- Search: find an item in the list
- Insert: add an item to the list
- Delete: remove an item from the list

Search can be implemented either iteratively or recursively

Comparison

Linked structure

- Overflow only occurs when memory is full
- Insertion/deletion are simple and fast
- Moving pointers is faster than moving the actual data

Array

- No extra space wasted for the pointer field
- Efficient random access is possible
- Better memory locality and cache performance

Containers

Common data structures allowing the storage and retrieval of data independently of the content:

- Stack:
    - LIFO order
    - Simple to implement and very efficient
- Queue:
    - FIFO order
    - Minimize the maximum waiting time
    - Trickier to implement than stacks

Both can be implemented using either linked lists or arrays

Dictionaries

Data type allowing access by content. Primary operations:

- Search: search a value in a given dictionary
- Insert: add an element to the dictionary
- Delete: remove an element from the dictionary

Most common operations:

- Max/Min: retrieve the largest/smallest element from the dictionary
- Predecessor/Successor: retrieve the element just before/after a given element; before/after are defined with respect to a sorted order

Dictionary using arrays

Let n be the number of elements in the array

Operation	Unsorted array	Sorted array
`search(D,k)`	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
`insert(D,k)`	$\mathcal{O}(1)$	$\mathcal{O}(n)$
`delete(D,k)`	$\mathcal{O}(1)^*$	$\mathcal{O}(n)$
`predecessor(D,k)`	$\mathcal{O}(n)$	$\mathcal{O}(1)$
`successor(D,k)`	$\mathcal{O}(n)$	$\mathcal{O}(1)$
`minimum(D)`	$\mathcal{O}(n)$	$\mathcal{O}(1)$
`maximum(D)`	$\mathcal{O}(n)$	$\mathcal{O}(1)$

* Assuming a pointer to the key k is given how to get $\mathcal{O}(1)$?

Dictionary using linked structures

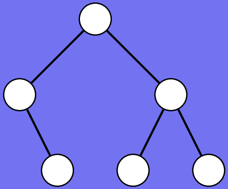
Let n be the number of elements in the list

Operation	Unsorted		Sorted	
	Single	Double	Single	Double
`search(D,k)`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
`insert(D,k)`	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
`delete(D,k)`	$\mathcal{O}(n)^*$	$\mathcal{O}(1)$	$\mathcal{O}(n)^*$	$\mathcal{O}(1)$
`predecessor(D,k)`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)^*$	$\mathcal{O}(1)$
`successor(D,k)`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`minimum(D)`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`maximum(D)`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)^\dagger$	$\mathcal{O}(1)$

[*] Why are singly linked lists slower?

[†] How to achieve $\mathcal{O}(1)$ for singly sorted lists?

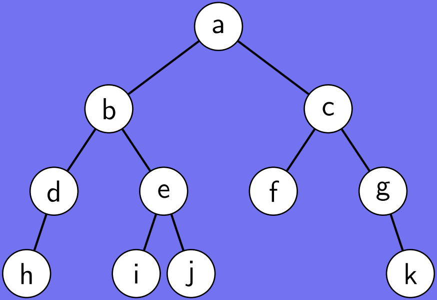
Binary search trees



- Based on doubly linked lists
- First object is the root of the tree
- Second object is a left child if it precedes the root and a right child if it succeeds it
- Third and further object are sorted along the tree following a similar pattern

- The three primary dictionary operations take $\mathcal{O}(h)$, with h the height of the tree
- Binary search trees balance the search time and flexible update
- How to handle deletion?

Binary search tree traversal



- Preorder traversal:
  a, b, d, h, e, i, j, c, f, g, k
- Inorder traversal:
  h, d, b, i, e, j, a, f, c, g, k
- Postorder traversal:
  h, d, i, j, e, b, f, k, g, c, a

How to implement inorder tree traversal?

Priority queue

Primary operations for priority queues:

- Insert: add an element to the queue
- Find min/max: return the last/first element in the queue
- Delete min/max: remove the last/first element in the queue

Operation	Array		Balanced tree
	Unsorted	Sorted	
`insert(Q,x)`	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
`find_min(Q)`	$\mathcal{O}(1)$[*]	$\mathcal{O}(1)$	$\mathcal{O}(1)$[*]
`delete_min(Q)`	$\mathcal{O}(n)$	$\mathcal{O}(1)$[†]	$\mathcal{O}(\log n)$

[*] How to reach $\mathcal{O}(1)$ for an unsorted array and a balanced tree?

[†] How to reach $\mathcal{O}(1)$ when deleting the min in a sorted array?

Hash tables

Practical way to maintain a dictionary where:

- The data is stored in an array
- Each key is hashed and stored at index “the hash of the key”
- Keys with a similar hash are store in a linked list

Good hash function: all indices occur with equiprobability

Example. A common choice is $H(k) = k \bmod m$, with m a prime not too close from a power of 2.

For $n = 2000$, 701 would be a good choice if one desires to have about three keys stored at each index.

Other common data structures

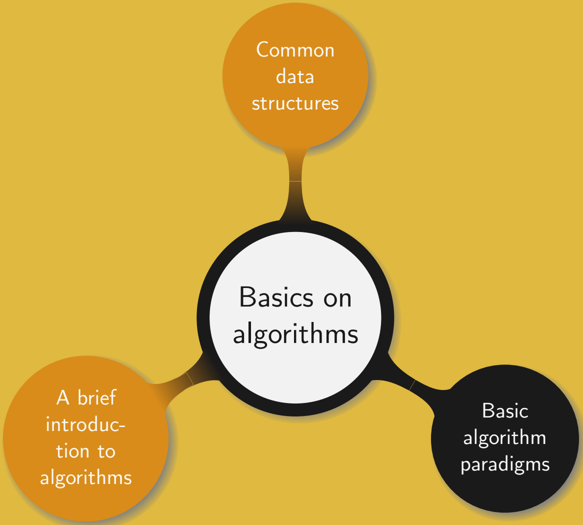
- Strings: array of characters; use suffix trees/arrays for pattern matching
- Geometric element: define regions as polygons using segments and points in an array or a tree
- Graphs: consider the adjacency matrix or an adjacency list; graph algorithms vary depending on the structure
- Sets: bit vector where the element in the set is the index and the value store is 1 or 0 depending whether the element is in the set; dictionaries can be used for fast membership queries

Summary

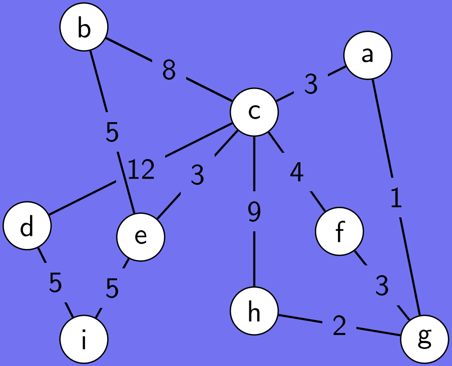
A few points to remember when selecting a data structure:

- Data can be represented in many ways
- No data structure is fast in all aspects
- Choosing the wrong data structure can be disastrous in terms of performance
- Several choices are often possible
- Identifying the best data structure is often not critical
- Always aim for clear, simple, and efficient data structures

Chapter organisation



A first graph problem



Simple problem:

- We have *n* computers connected by wires
- Using different wires implies different costs

We expect to:

- Connect all the computers to the network
- Minimize the cost

Minimum spanning tree

Problem (Minimum Spanning Tree (MST))

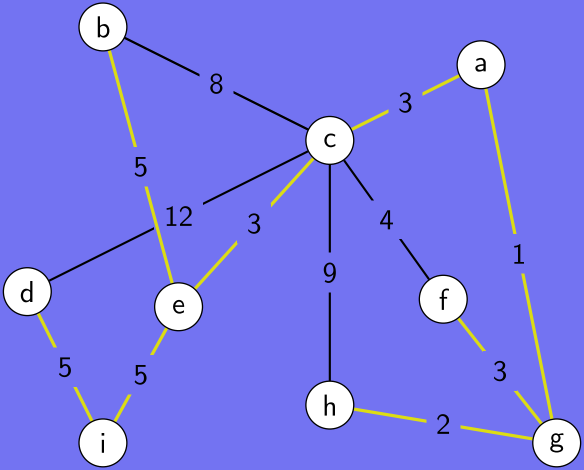
Given a weighted graph G, find a subgraph T such that:

1. All the vertices on G are connected on T,
2. The total weight, defined as the sum of the weight of all the edges in T, is minimized.

The graph T is a *minimum spanning tree* for G.

Remark. Note that T is a tree: if it contained a cycle, at least one edge could be removed, allowing a lower weight while preserving the connected property of T↯.

A greedy approach



Kruskal's algorithm

Algorithm. (*Kruskal*)

```
Input  : A graph $G = \langle V, E \rangle$
Output: A minimum spanning tree $T$ for $G$
Sort the edges $G.E$ by weight;
$T \leftarrow \emptyset$;
for edges $(u, v)$ in $G.E$, in non-decreasing order do
    if adding $(u, v)$ does not create a cycle then
        add edge $(u, v)$ to $T$
    end if
end for
return $T$
```

What needs to be specified?

Correctness of Kruskal's algorithm

Theorem

Assuming the previous notations, Kruskal's algorithm produces a minimum spanning tree for G.

Proof. Let $G = \langle V, E \rangle$ be a graph and let v and w be two vertices connected by an edge. If S is the set of all the vertices with a path to v before e is added, then $w \notin S$, otherwise this would define a cycle. Moreover if there was an edge with smaller weight than e, connecting S and $V - S$, then it would have already been added. Therefore e is the cheapest edge connecting $V - S$ to S, and as such belongs to a minimum spanning tree of G.

Clearly by design the algorithm will not generate any cycle. Moreover as G is connected and all the edges are explored, $V - S$ and S will be linked at some stage. Hence T is connected. □

Back to the algorithm

Issue: how to represent the data such that whether or not adding an edge creates a cycle can be efficiently tested?

For each edge joining two vertices v and w:

- Identify all the connected components of v and w
- If the edge is to be included, merge the two components

Extra notes:

- No edge needs to be removed
- No component needs to be split
- Everything must be done efficiently

Toward a new data structure

Representing data using:

- An array: testing can be done in constant time; merging requires linear time
- A graph: merging is only adding an edge; testing requires a full graph traversal

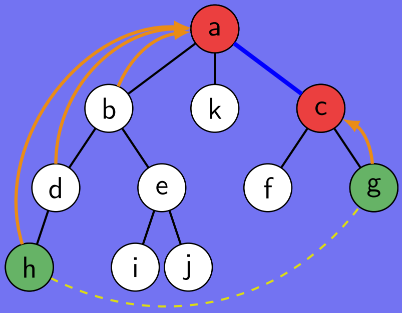
Implement a new data structure containing:

- A pointer to the parent
- The *rank*, or depth, of the sub-tree

The union-find data structure

The two operations are:

- `Find(v)`: find the root of the tree for vertex *v*
- `Union(v,w)`: link the root of the tree containing *v* to the root of the tree containing *w* (or the other way around)



Process:

- We have two sub-trees
- On the graph an edge joins the vertices *h* and *g*
- `Find` on *h* and *g* returns *a* and *c*, respectively
- Update parents for *h*, *d*, *b* and *g*
- `Union` connects *c* to *a*

Handling a union-find structure

Algorithm.

```
Function GenSet($x$):
    $x.parent \leftarrow x$; $x.rank \leftarrow 0$;
end
Function Find($x$):
    if $x.parent \neq x$ then $x.parent \leftarrow$ Find($x.parent$) ;
    return $x.parent$
end
Function Union($x$,$y$):
    $X \leftarrow$ Find($x$); $Y \leftarrow$ Find($y$);
    if $X.rank > Y.rank$ then $Y.parent \leftarrow X$;
    else $X.parent \leftarrow Y$;
    if $X.rank = Y.rank$ then $Y.rank{+}{+}$;
end
```

Revisiting Kruskal's algorithm

Algorithm. (*Kruskal with find-union*)

```
Input  : A graph $G = \langle V, E \rangle$
Output: A minimum spanning tree $T$
Sort the edges $G.E$ by weight;
$T \leftarrow \emptyset$;
for edges $(u, v)$ in $G.E$, in non-decreasing order do
    if Find($u$)$\neq$ Find($v$) then
        add edge $(u, v)$ to $T$;
        Union($u$,$v$)
    end if
end for
return $T$
```

Counting inversions

Problem (Counting inversions)

Given a list of n elements $a_0, \cdots, a_{n-1}$, determine how many pairs $(a_i, a_j)_{\substack{0 \leq i,j \leq n \\ i \neq j}}$ are not in ascending order. Such pairs are called *inversions*.

Remark. This problem has numerous applications such as

- Voting theory
- Analysis of search engines ranking
- Collaborative filtering

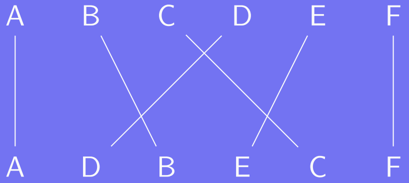
Example

Given 6 movies compare the ranking of two users:

Movie	A	B	C	D	E	F
First user	1	2	3	4	5	6
Second user	1	3	5	2	4	6

Inversions: $(3, 2)$, $(5, 2)$, $(5, 4)$

A simple geometrical view:



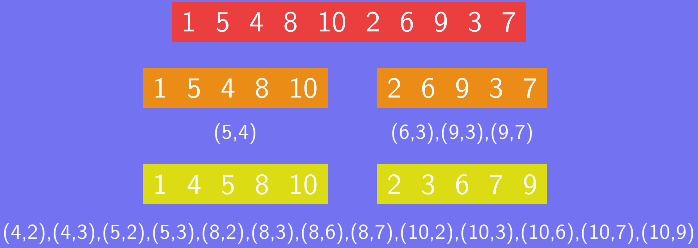
Divide and conquer approach

Strategy for solving the counting inversions problem (1.59):

1. Divide: split the list L into two halves L_1 and L_2
2. Conquer: recursively count inversions in each list
3. Combine: count inversions for the pairs (l_i, l_j) with l_i and l_j belonging to L_1 and L_2 respectively

The sum of the three counts is the total number of inversion in L

Example.



1 5 4 8 10 2 6 9 3 7

1 5 4 8 10

2 6 9 3 7

(5,4)

(6,3),(9,3),(9,7)

1 4 5 8 10

2 3 6 7 9

(4,2),(4,3),(5,2),(5,3),(8,2),(8,3),(8,6),(8,7),(10,2),(10,3),(10,6),(10,7),(10,9)

Merge and count

Algorithm. (*Merge and count*)

Input : Two sorted lists: $L_1 = (l_{1,1}, \cdots, l_{1,n_1})$, $L_2 = (l_{2,1}, \cdots, l_{2,n_2})$
Output: Number of inversions *count*, and L_1 and L_2 merged into L

```
Function MergeCount($L_1, L_2$):
    $count \leftarrow 0$; $L \leftarrow \emptyset$; $i \leftarrow 1$; $j \leftarrow 1$;
    while $i \leq n_1$ and $j \leq n_2$ do
        if $l_{1,i} \leq l_{2,j}$ then
            append $l_{1,i}$ to $L$; $i{+}{+}$;
        else
            append $l_{2,j}$ to $L$; $count \leftarrow count + n_1 - i + 1$; $j{+}{+}$;
        end if
    end while
    if $i > n_1$ then append $l_{2,j}, \cdots, l_{2,n_2}$ to $L$;
    else append $l_{1,i}, \cdots, l_{1,n_1}$ to $L$;
    return $count$ and $L$
end
```

Sort and count

Algorithm. (*Sort and count*)

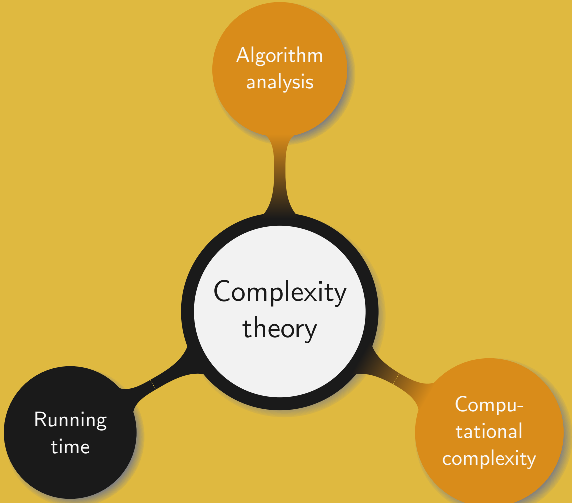
```
Input   : A list $L = (l_1, \cdots, l_n)$
Output: The number of inversions $count$ and $L$ sorted
Function SortCount($L$):
    if $n$=$1$ then return 0 and $L$;
    else
        Split $L$ into $L_1 = (l_1, \cdots, l_{\lceil n/2 \rceil})$ and $L_2 = (l_{\lceil n/2 \rceil + 1}, \cdots, l_n)$;
        $count_1, L_1 \leftarrow$ SortCount($L_1$);
        $count_2, L_2 \leftarrow$ SortCount($L_2$);
        $count, L \leftarrow$MergeCount($L_1, L_2$);
    end if
    $count \leftarrow count_1 + count_2 + count$;
    return $count$ and $L$
end
```

Key points

- How to present pseudocode?
- What are the two main categories of data structure?
- What is a greedy algorithm?
- Describe the divide and conquer strategy
- How is the `Union-Find` data structure working?

2. Complexity theory

Chapter organisation



Preview of the RAM model

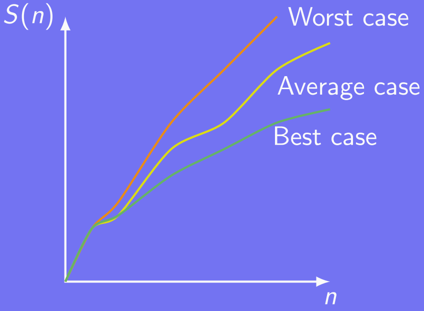
Random Access Machine (RAM) model:

- Each simple operation ($+$, $-$, $\times$, $/$, if,...) takes one step
- Loops are composed of several simple steps, which are repeated a certain number of times
- Each memory access accounts for one step

The runtime of an algorithm is given by the total number of steps necessary to complete it.

The RAM model allows the study of algorithms independently of their implementation or running environment.

Time complexity

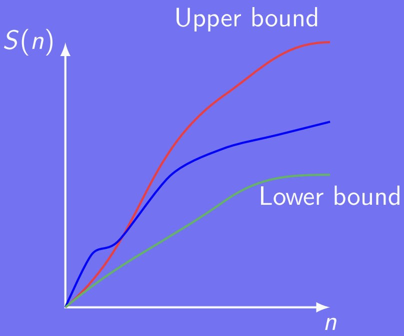


Given a problem with input size n:

- Worst case complexity: maximum number of steps to complete an instance of the algorithm
- Best case complexity: minimum number of steps to complete an instance of the algorithm
- Average case complexity: average number of steps, over all possible instances

The complexity of an algorithm is defined by a numerical function.

Approximation

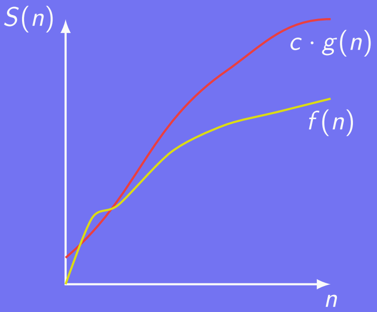


Limitations of the RAM:

- Complexity can display many bumps, e.g. algorithms performing better on input of size a power of 2
- Counting the exact number of operations is too complex, e.g. implementation choices impact the number of RAM instructions

We need notations that simplify the analysis by ignoring the details irrelevant to the comparison of the algorithms.

The $\mathcal{O}$ notation

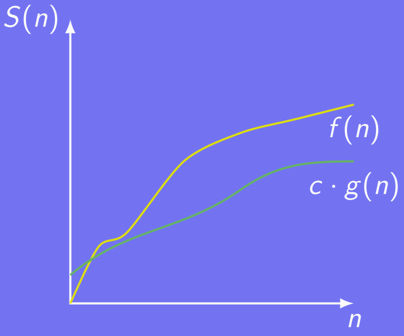


Approximating complexity:

- $c \cdot g(n)$ is an upper bound for $f(n)$
- There exists two constants c and n_0 such that for all n larger than n_0, $f(n) \leq c \cdot g(n)$
- $f(n) = \mathcal{O}(g(n))$

An algorithm in $\mathcal{O}(f)$ has time complexity upper-bounded by f

The Ω notation

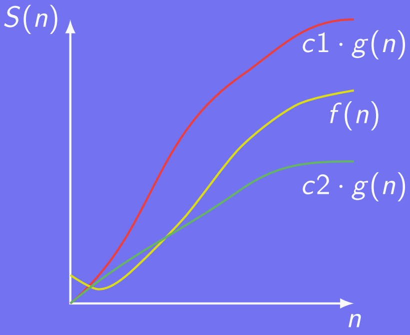


Approximating complexity:

- $c \cdot g(n)$ is a lower bound for $f(n)$
- There exists two constants c and n_0 such that for all n larger than n_0, $f(n) \geq c \cdot g(n)$
- $f(n) = \Omega(g(n))$

An algorithm in $\Omega(f)$ has time complexity lower-bounded by f

The Θ notation



Approximating complexity:

- $c_1 \cdot g(n)$ is an upper bound for $f(n)$ while $c_2 \cdot g(n)$ is a lower bound for $f(n)$
- There exists three constants c_1, c_2, and n_0 such that for all n larger than n_0, $c_2 \cdot g(n) \leq f(n) \leq c_1 \cdot g(n)$
- $f(n) = \Theta(g(n))$

An algorithm in $\Theta(f)$ has time complexity roughly similar to f

Example

Exercise. Prove the following equalities:

- $2^{n+1} = \Theta(2^n)$?
  It suffices to prove that $2^{n+1} = \mathcal{O}(2^n)$ and $2^{n+1} = \Omega(2^n)$.
  First note that $2^{n+1} = 2 \cdot 2^n$, such that $2^{n+1} \leq c \cdot 2^n$, for any $c \geq 2$.
  Then observe that for all $n \geq 2$, $2^{n+1} \geq c \cdot 2^n$, for any $0 < c \leq 2$.

- $(x+y)^2 = \mathcal{O}(x^2+y^2)$?
  We need to find some c such that $(x+y)^2 \leq c(x^2+y^2)$. Without loss of generality we can assume $x \geq y$. Then $2xy \leq 2x^2 \leq 2(x^2+y^2)$ and as $(x+y)^2 = x^2+2xy+y^2$, we conclude that $(x+y)^2 \leq 3(x^2+y^2)$.

Dominance relations

Common complexity functions

- Constant: $\mathcal{O}(1)$
- Logarithmic: $\mathcal{O}(\log n)$
- Linear: $\mathcal{O}(n)$
- Superlinear: $\mathcal{O}(n \log n)$
- Quadratic: $\mathcal{O}(n^2)$
- Cubic: $\mathcal{O}(n^3)$
- Exponential: $\mathcal{O}(c^n)$
- Factorial: $\mathcal{O}(n!)$

More complexity functions

- Inverse Ackerman's function: $\mathcal{O}(\alpha(n))$
- Log Log: $\mathcal{O}(\log \log n)$
- $\mathcal{O}(\log n / \log \log n)$
- $\mathcal{O}(\sqrt{n})$
- $\mathcal{O}(n^{c+\varepsilon})$

$$n! \gg 2^n \gg n^{c+\varepsilon} \gg n^3 \gg n^{2+\varepsilon} \gg n^2 \gg n \log n \gg n^{1+\varepsilon} \gg n$$

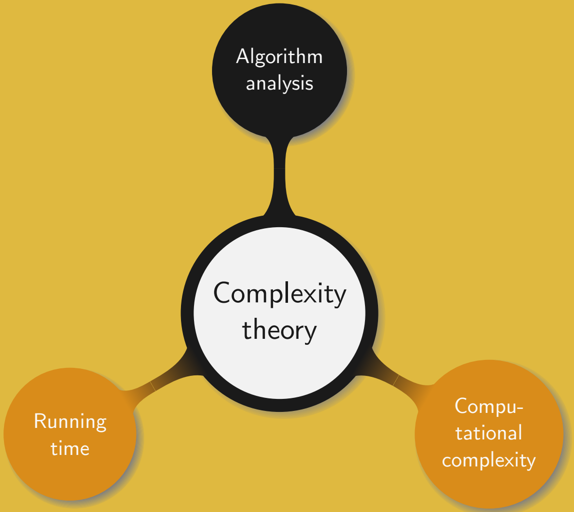
$$\gg \sqrt{n} \gg \log n \gg \frac{\log n}{\log \log n} \gg \log \log n \gg \alpha(n) \gg 1$$

Growth rate

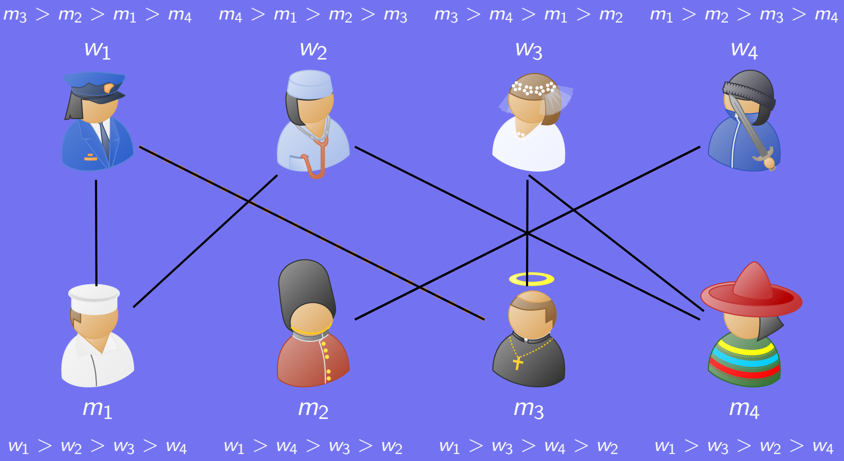
Assuming 10^9 basic operations a second, the time spent depending on the input size (in bits or number of input) and the time complexity is given as follows.

Input size n	$\log n$	n	$n \log n$	n^2	n^3	2^n	$n!$
10	$< 0.01\mu$s	0.01μs	0.03μs	0.1μs	1μs	1μs	3.6ms
20	$< 0.01\mu$s	0.02μs	0.09μs	0.4μs	0.8μs	1ms	77y
30	$< 0.01\mu$s	0.03μs	0.1μs	0.9μs	2.7μs	1s	10^{16}y
40	$< 0.01\mu$s	0.04μs	0.2μs	1.6μs	6.4μs	18min	
50	$< 0.01\mu$s	0.05μs	0.3μs	2.5μs	12.5μs	13d	
100	$< 0.01\mu$s	0.1μs	0.6μs	10μs	1ms	10^{13}y	
1000	0.01μs	1μs	10μs	1ms	1s		
10000	0.01μs	10μs	130μs	100ms	16min		
100000	0.02μs	100μs	1.7ms	10s	11.5d		
1000000	0.02μs	1ms	20ms	17min	32y		
10000000	0.02μs	10ms	200ms	1.2d	32000y		
100000000	0.03μs	100ms	2.6s	116d			
1000000000	0.03μs	1s	30s	32y			

Chapter organisation



Stable marriage problem



$m_3 > m_2 > m_1 > m_4$ $m_4 > m_1 > m_2 > m_3$ $m_3 > m_4 > m_1 > m_2$ $m_1 > m_2 > m_3 > m_4$

w_1 w_2 w_3 w_4

m_1 m_2 m_3 m_4

$w_1 > w_2 > w_3 > w_4$ $w_1 > w_4 > w_3 > w_2$ $w_1 > w_3 > w_4 > w_2$ $w_1 > w_3 > w_2 > w_4$

Formalization

Problem (Stable matching problem)

Given a set of *n* men and a set *n* women, each person ranks all the members of the other set. A match between the men and women where no (*man*, *woman*)-pair, would prefer to be together rather than with their current partner is said to be *stable*.

Visual representation of the problem:

- Each men (women) is represented by a vertex
- No edge connects elements from a same set
- Monogamy is assumed for both men and women
- The number of edges equals the number of vertices in a set
- No blocking pair must exist

Solving the problem

Algorithm. (*Gale-Shapley*)

Input : n men and n women, all initially free

Output : n engaged pairs

```
while there is a free man do
    $m \leftarrow$ select a man; $w \leftarrow$favorite woman $m$ hasn't proposed yet;
    $m$ proposes to $w$;
    if $w$ is free then set $(m, w)$ as an engaged pair;
    else
        $m' \leftarrow$current date of $w$;
        if $w$ prefers $m'$ over $m$ then $m$ remains free;
        else
            set $(m, w)$ as an engaged pair;
            set $m'$ as free;
        end if
    end if
end while
return the $n$ engaged pairs
```

Correctness

Theorem

Given two sets of n men and n women, all initially free, Gale-Shapley algorithm returns a stable matching.

Proof. First we prove that algorithm 2.79 returns a perfect matching, i.e. there is no free man or woman.

Suppose there exists a free man who has already proposed to all the women. As a woman who is engaged once remains engaged, to the same man or a new one, it means that all the n women are engaged. Noting that a woman cannot be engaged to more than one man leads to a contradiction.↯

Lets now assume that the matching is not stable. Then there are two pairs (m, w) and (m', w') such that m prefers w' over w while w' prefers m over m'.

Correctness

By definition, the last proposal of m was to w. If he didn't proposed earlier to w', then he does not prefer w' over w.↯
If he did propose to w', then he was rejected in favor of another man m''. Therefore the final partner of w' is either m' or a man she prefers over m'. Either way w' does not prefer m to m'.↯

It follows that Gale-Shapley algorithm returns a stable matching.

□

Corollary

If each man prefers a different woman then they all end up with their first choice, independently of the women preferences.

Complexity

Theorem

Gale-Shapley algorithm has complexity $\mathcal{O}(n^2)$.

Proof. At each iteration of the `while` loop a man proposes to a woman he has never proposed before. Therefore, for n men, there is a maximum of n^2 proposals.

Since each iteration corresponds to exactly one proposal the `while` loop is applied at most n^2 times.

Hence algorithm 2.79 has complexity $\mathcal{O}(n^2)$. □

Complexity of the Union-Find structure

In order to study the complexity of the Union-Find data structure (1.56) we introduce the following simple results.

Lemma

1. A node that is a root and gets attached to another root will never be a root again.
2. When a node stops being a root its rank remains fixed.
3. As Find travels to the root of the tree the rank of the nodes strictly increases.
4. A node with rank k has at least 2^k nodes in its subtree.
5. Over n elements there are at most $n/2^k$ elements of rank k.

Iterated logarithm

Definition

The *iterated logarithm* function, denoted $\log^*$, is defined by

$$\begin{aligned} \log^* : \quad & \mathbb{R} \longrightarrow \mathbb{N} \\ & x \longmapsto \begin{cases} 0 & \text{if } x \leq 1 \\ 1 + \log^* \log_2 x & \text{if } x > 1 \end{cases} \end{aligned}$$

In a less abstract way the iterated logarithm of n is the number of times the logarithm function has to be applied in order to get a number smaller than 2.

Example.

$$\log^* 4 = 1 + \log^* \log_2 4 = 2$$
$$\log^* 16 = 1 + \log^* 4 = 3$$
$$\log^* 65536 = \log^* 2^{2^{2^2}} = 4$$

Complexity for m Find

Theorem

The cost of one `Find` in the Union-Find data structure is $\mathcal{O}(\log n)$, while m `Find` operations cost $\mathcal{O}\left((m+n)\log^* n\right)$.

Proof. The rank of the root being bounded by the depth of the tree in the Union-Find data structure it is at most $\log n$.

The problem is now to evaluate the effect of m such operations. Since the path is compressed at each `Find` the complexity of any subsequent `Find` reusing a visited path is decreased.

In fact the running time of a `Find` operation is proportional to (i) the length of the path from a node to the root of the tree it belongs to and (ii) m.

Complexity for m Find

We now divide the nodes into "blocks", such that nodes of similar ranks are on a same block. To formerly define a block we introduce the following recursive sequence

$$\begin{cases} T_0 = 1 \\ T_i = 2^{T_{i-1}}, \quad i > 0. \end{cases}$$

The blocks are then defined as the sets

$$B_0 = \{0\}, \; B_1 = \{1\},$$
$$B_i = [\![T_{i-1}, T_i - 1]\!], \; i > 1.$$

Observing the blocks we notice that the maximum number of blocks is $\log^* n$.

Complexity for m Find

Moreover as there is a maximum of $n/2^k$ elements of rank k (lemma 2.83), the number of nodes in the i-th block is at most

$$\begin{aligned}\sum_{j=T_{i-1}}^{T_i-1}\frac{n}{2^j} &= n\sum_{j=T_{i-1}}^{2^{T_{i-1}}-1}\frac{1}{2^j}\\ &\leq \frac{2n}{2^{T_{i-1}}}\\ &= \frac{2n}{T_i}.\end{aligned} \tag{2.1}$$

The idea is to evaluate the cost of the `Find` while traveling from a node to its root. In such a case we traverse nodes which are (i) in the same block (ii) in a different block or (iii) directly connected to the root.

Complexity for m Find

The simplest case is (iii) as only one step is required, implying an $\mathcal{O}(m)$ complexity for the m Find.

Note that (ii) was evaluated earlier when we observed that the maximum number of blocks is $\log^* n$. Therefore the complexity of (ii) is $\mathcal{O}(m \log^* n)$.

Finally for case (i) the path starting at a node x goes through at most $T_i - 1 - T_{i-1}$ ranks, and from (2.1) there are less than $2n/T_i$ such elements x in a block. Thus the total number of times the rank changes in a block is

$$\begin{aligned}(T_i - 1 - T_{i-1})\frac{2n}{T_i} &\leq T_i \frac{2n}{T_i} \\ &= 2n.\end{aligned}$$

Complexity for m Find

Remembering that the maximum number of blocks is $\log^* n$, the `Find` operations generate an overall of at most $\mathcal{O}(n \log^* n)$ internal links.

Hence the time spent on m operations is given by the total time spent on cases (i) to (iii), that is $\mathcal{O}\left((m+n)\log^* n\right)$. □

For long this has been the tightest proven upper bound. However a better result was proven in the 1970's. It bounds the complexity by using the inverse Ackerman's function, which is growing even slower than the $\log^*$ function.

Ackerman's function

Definition (Ackerman's function)

For any two positive integers m and n, Ackerman's function is recursively defined by

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

Example.

$$\begin{aligned} &A(0,0) = 1 \\ &A(1,1) = A(0, A(1,0)) = A(0,2) = 3 \\ &A(2,2) = A(1, A(2,1)) = A(1,5) = 7 \\ &A(3,3) = A(2, A(3,2)) = A(2,29) = 61 \\ &A(4,4) = A(3, A(4,3)) = 2^{2^{2^{65536}}} - 3 \end{aligned}$$

Inverse Ackerman's function

Remark. Ackermann's function is one of the simplest *total computable function* that is not *primitive recursive*.

For $m = n$, one defines the inverse of Ackerman's function $\alpha(x)$. As Ackerman's function is extremely fast-growing its inverse is an extremely-slow growing function which never increases beyond 4 or 5 for any practical value.

Theorem

The amortized time for a sequence of m `GenSet`, `Union`, and `Find` operations, n of which are `GenSet`, can be performed in time $\mathcal{O}(m\alpha(n))$, where $\alpha(n)$ is the inverse Ackerman's function.

Remark. It was recently proven that this upper bound is asymptotically optimal, that is the complexity of the Union-Find structure is $\Omega(\alpha(n))$.

Complexity of Sort and Count

The Sort and Count algorithm (1.63) solves the Counting inversions problem (1.59).

For the merge part:

- Each iteration of the `While` takes constant time
- Elements that are added are never added again
- There is a maximum of n element

The time spent for the merge part is $\mathcal{O}(n)$.

For the sort part the divide and conquer strategy is applied:

- Divide the input into two pieces of equal size
- Recursively solve the two separate problems
- Combine the two results

The time for division and recombination is linear in the size of the input.

Divide and conquer

Let $S(n)$ be the worst-case run time for sorting an instance of size n.

Simple complexity analysis:

- Time spent to divide into two pieces: $\mathcal{O}(n)$
- Time spent to solve each piece: $S(n/2)$
- Combine the results: $\mathcal{O}(n)$

Hence, the running time satisfies the recurrence relation

$$\begin{cases} S(2) & \leq c, \text{ for some constant } c \\ S(n) & \leq 2S(n/2) + cn, \text{ if } n > 2 \end{cases}$$

The Master Theorem

Let A be an algorithm whose running time is described by a recurrence of the form $T(n) = aT(n/b) + f(n)$, with $a \geq 1$, $b > 1$ two constants, and $f(n)$ an asymptotically positive function. This relation corresponds to A dividing the initial problem of size n into a sub-problems of size n/b each. While it takes $T(n/b)$ to recursively solve each of the a sub-problems $f(n)$ corresponds to the time spent splitting them and combining their results.

Remark. For the sake of simplicity we assume n/b to be an integer, and more generally n to be a power of b. Although it is often not true, this simplifies the discussion while having no impact on the asymptotic behavior of the recurrence.

The Master Theorem

Theorem (Master theorem)

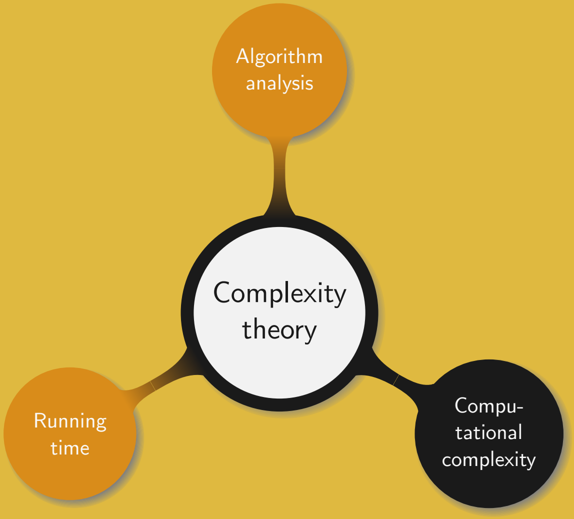
Let $a \geq 1$, $b > 1$, be two constants, $f(n)$ be a function, and $T(n) = aT(n/b) + f(n)$ be a recurrence relation over the positive integers. Then the asymptotic bound on $T(n)$ is given by

$$T(n) = \begin{cases} \Theta(n^{\log_b a} \log n) & \text{if } f(n) = \Theta(n^{\log_b a}); \\ \Theta(n^{\log_b a}) & \text{if } f(n) = \mathcal{O}(n^{\log_b a - \varepsilon}),\ \varepsilon > 0; \\ \Theta(f(n)) & \text{if } f(n) = \Omega(n^{\log_b a + \varepsilon}),\ \varepsilon > 0,\ n \text{ large} \\ & \text{enough, and } af(n/b) \leq cf(n),\ c < 1; \end{cases}$$

Back to Sort and Count:

- By the Master theorem (2.95) sorting takes $\mathcal{O}(n \log n)$
- The merge part has time complexity $\mathcal{O}(n)$
- The overall time complexity of Sort and Count is $\mathcal{O}(n \log n)$

Chapter organisation



Computational problem

So far we have presented a few problems as well as some algorithms to solve them. We have also introduced the basics to study the complexity of those algorithms.

We now formalize these ideas through the following definitions.

Definitions

1. A *computational problem* is a question or set of questions that a computer might be able to solve.
2. The study of the solutions to computational problems composes the field of *Algorithms*.
3. *Computational complexity* attempts to classify algorithms depending on their speed or memory usage.

Taxonomy of computational problems

A *decision problem* is a computational problem admitting exactly one of the two answers, “Yes” or “No”, to the question.

Example. Let n be an integer, is n prime?

A *search problem* is a computational problem where the answer is an arbitrary string.

Example. Let n be an integer, find all the primes less than n.

A *counting problem* is a computational problem where the answer is the number of solutions to a corresponding search problem.

Example. Let n be an integer, count the number of primes less than n.

Taxonomy of computational problems

Am *optimization problem* is a computational problem where the answer is the best solution, with respect to some parameters, to a corresponding search problem.

Example. For *n* a non-prime integer, find the largest prime factor of *n*.

A *function problem* is a computational problem admitting exactly one answer for every input. The answer is more complex than in the case of a decision problem.

Example. Given a list of cities and the distance between each pair, find the shortest route passing through all the cities exactly once and returning to the first visited city.

Relation between computational problems

An obvious observation is that counting and optimization problems are closely related to search problems.

As a less obvious observation one can notice that any optimization problem can be transformed into a decision problem.

Example. Optimization problem: let G be a graph and v_1, v_2 be two vertices in G. Find the shortest path between v_1 and v_2.

A corresponding decision problem: let G be a graph and v_1, v_2 be two vertices in G. Is there a path from v_1 to v_2 that goes through less than 5 edges?

Relation between computational problems

Similarly any function problem can be transformed into a decision problem. If the decision problem can be solved then so is its corresponding function problem. However the computational cost is not always preserved; for instance a function problem might be solved by an exponential time algorithm while its corresponding decision problem can be solved in polynomial time.

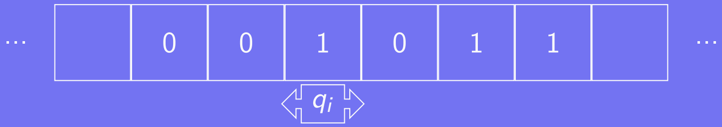
Conversely decision problems can be converted into function problems by computing the characteristic function of the set associated with the decision problem.

Decision problems play a central role in computability and complexity theories. In order to study them in more details we first need to setup a more precise computational model.

Turing machine

> *...an unlimited memory capacity obtained in the form of an infinite tape marked out into squares, on each of which a symbol could be printed. At any moment there is one symbol in the machine; it is called the scanned symbol. The machine can alter the scanned symbol, and its behavior is in part determined by that symbol, but the symbols on the tape elsewhere do not affect the behavior of the machine. However, the tape can be moved back and forth through the machine, this being one of the elementary operations of the machine. Any symbol on the tape may therefore eventually have an innings.*
>
> Turing – *Intelligent Machinery*

Turing machine



A Turing machine can be use to model modern computers:

- It is composed of an infinite tape
- The tape is divided into cells
- Each cell contains a symbol taken from an alphabet or a blank
- The tape is read or written, cell by cell, by a head
- At any time the device is in a state q_i with $i \in \mathbb{N}$

Elementary operation

Let Σ be the alphabet of symbols and Q be the set of the states. Any elementary operation is determined by the symbol read and the state $q_i \in Q$.

Three basic cases can occur:

- The symbol read is replaced by another symbol from $\Sigma \cup \{b\}$, where b represents a blank
- The head moves to the left, right, or stands still
- The machine transitions from state q_i to state q_j

This can be summarized by the following map:

$$M : (\Sigma \cup \{b\}) \times Q \longrightarrow (\Sigma \cup \{b\}) \times \{-1, 0, 1\} \times Q$$

Deterministic computation

A deterministic computation is defined by (i) an initial state q_0 and (ii) a finite sequence of elementary operations.

At the end of the sequence the tape contains an integer x written over a number of cells, all the other cells being set to b (blank). The state of the tape, x, provides the result.

Definition

A function $f : \Sigma^* \longrightarrow \Sigma^*$ is said to be *Turing computable* if there exists a Turing machine M which returns $f(x)$ for any input x.

Remark. Church proved that any computation, in a physical sense, can be performed on a Turing machine. This is however not the case anymore for quantum computers which cannot be modeled by a Turing machine. Its quantum counterpart is called *Quantum Turing machine*.

The RAM model

The RAM consists of:

- A control unit: containing a program and a program register pointing to the instruction to be executed
- An arithmetic unit: executes all the arithmetic operations
- A memory: divided into cells, each containing an integer
- An input unit: an input tape divided into cells and a head which reads the input
- An output unit: an output tape divided into cells and a head which writes the output

A program in the RAM model

The initial configuration of a RAM:

- All the cells are set to 0, besides the first n input cells
- The program register contains 1
- The first n cells contain the value from the n input cells

A computation consists in performing a sequence of configuration based on the program.

A program is composed of operands stored in the memory and onto which instructions can be run (e.g. `LOAD`, `STORE`, `+`, `-`, $\times$, `/`, `READ`, `WRITE`, `JUMP`, `JZERO`, `JGE`, `HALT`, `ACCEPT`, `REJECT`).

RAM vs. Turing machine

The RAM represents a better model for modern computer than a Turing machine. However those two models are equivalent.

Theorem

- For every Turing machine M, there exists a program P for RAM that simulates M.
- For a program P of RAM, there exists a Turing machine with five tapes such that P and M behave the same.

As we have briefly introduced the most common computational models we can now formalize the idea of complexity, and then investigate decision problems.

Formalizing complexity

Definitions

Let Σ be an alphabet and M be a Turing machine.

1. Given an input $x \in \Sigma^*$ for M, the number of operations $t_M(x)$ necessary to perform the computation is called the *length of the computation*.

2. The function
$$\begin{aligned} T_M : \Sigma^* &\longrightarrow \mathbb{N} \\ x &\longmapsto \max_{|x|} t_M(x) \end{aligned}$$
defines the *time complexity* for M.

Remark. Since the maximum of the length of the computation is considered this definition corresponds to the worst case complexity (2.68) with $|x|$, the size of the input x equal to n.

Decision problems

Definitions

1. Let f be a Turing computable function, and M be that Turing machine. If there exists a polynomial P such that for any input x, $T_M(x) \leq P(x)$, then M is called a *deterministic polynomial algorithm*.

2. Let $f : \Sigma^* \to \{0, 1\}$ be a Turing computable function and L be the preimage of $\{1\}$ under f, i.e. $f^{-1}(1)$. Then L is called a *language* and f defines a *decision problem* P. One says that the Turing machine *computes* f, *solves* P, or *decides* L.

The $\mathcal{P}$ and $\mathcal{NP}$ classes

Definitions

1. The set of the decision problems which can be solved by a deterministic polynomial algorithm defines the *class* $\mathcal{P}$.

2. A decision problem Π is computable by a non deterministic polynomial algorithm if and only if there exists a Turing machine M and a polynomial P such that
   1. $x \in L(\Pi)$ if and only if there exists $y \in \Sigma^*$, such that M computes 1 when it has input x in the cells 1 to $|x|$ and y in the cells -1 to $-|y|$.
   2. For all x in $L(\Pi)$ there exists y such that M computes 1 in time less than $P(|x|)$.

   The set of all such decision problems defines the class $\mathcal{NP}$.

Explanations

Informal view:

- Class $\mathcal{P}$: set of decision problems solvable in polynomial time
- Class $\mathcal{NP}$: set of decision problems which (i) can be solved in polynomial time, assuming a *certificate* $y \in \Sigma^*$, is known and (ii) have `True` as answer.
  The certificate is often taken to be the solution and it means that the solution can be verified in polynomial time.
- Class co-$\mathcal{NP}$: set of decision problems which (i) can be solved in polynomial time, assuming a *certificate* $y \in \Sigma^*$, is known and (ii) have `False` as answer.
  The certificate is often taken to be the solution and it means that the solution can be verified in polynomial time.
- Any problem in $\mathcal{P}$ is also in $\mathcal{NP}$ and co-$\mathcal{NP}$.

Complexity reduction

Definitions

1. Let P_1 and P_2 be two decision problems. One says that P_1 can be *reduced in polynomial time* to P_2 and writes $P_1 \propto P_2$, if there exists a function f, computable in polynomial time, such that $x \in L(P_1)$ if and only if $f(x) \in L(P_2)$.
2. A problem Π is $\mathcal{NP}$*-hard* if and only if for all P in $\mathcal{NP}$, P can be reduced in polynomial time to Π.
3. A problem Π is $\mathcal{NP}$*-complete* if and only if Π is (i) in $\mathcal{NP}$ and (ii) $\mathcal{NP}$-hard.

Explanations

Informal view:

- Saying that P_1 can be reduced in polynomial time to P_2 means that P_1 is not any harder than P_2. In fact any instance of P_1 can be transformed, in polynomial time, into an instance of P_2.
- An $\mathcal{NP}$-hard problem Π is at least as hard as the hardest problem in $\mathcal{NP}$. Note that Π does not need to belong to $\mathcal{NP}$, it could for instance be a search problem or an optimization problem.
- An $\mathcal{NP}$-complete problem Π must belong to $\mathcal{NP}$ and is as hard as the hardest problem in $\mathcal{NP}$. In other words if we can solve Π then we can solve any other problem in $\mathcal{NP}$, deriving the solution from the one of Π, in polynomial time.

Problem (Halting problem)

Given the description of a Turing machine M as well as its initial input, decide whether M will halt or run forever.

The problem consists in constructing a Turing machine which would be able to decide whether or not another Turing machine would complete its task. From a simple perspective, if the machine "quickly" completes its task then we know it. Otherwise we have no idea whether it will halt later or never stop.

Turing proved that the halting problem is *undecidable*, i.e. it is impossible to construct an algorithm which always returns the right "yes" or "no" answer. Therefore it does not belong to $\mathcal{NP}$.

Boolean Satisfiability Problem

Problem (Boolean Satisfiability Problem (SAT))

Given a boolean function $(x_1, \cdots, x_n) \mapsto f(x_1, \cdots, x_n)$, where the x_i, $0 \leq i \leq n$ are in $\{0, 1\}$, and f is an expression constructed from the x_i and the boolean symbols $\neg$, $\vee$, and $\wedge$, is there a value for the n-tuple $\langle x_1, \cdots, x_n \rangle$ such that $f(x_1, \cdots, x_n)$ is `True`?

Example. The expression $x_1 \wedge \neg x_2$ is satisfiable since taking $x_1 = \texttt{True}$ and $x_2 = \texttt{False}$ yields $\texttt{True} \wedge \texttt{True} = \texttt{True}$.

On the other hand the expression $x_1 \wedge \neg x_1$ is not satisfiable since it will always evaluate as `False`, independently of the choice of x_1.

Boolean Satisfiability Problem

Theorem (Cook)

SAT is $\mathcal{NP}$-complete.

Sketch of proof. First, SAT is in $\mathcal{NP}$ as any instance can be verified in polynomial time on a Turing machine.

Second, any $\mathcal{NP}$ problem P can be verified in polynomial time on a Turing machine M. Therefore for each input to M, it is possible to construct a boolean formula checking (i) every step of the computation, (ii) if M halts, and (iii) M returns "Yes". Such a boolean expression will be satisfied if and only if those three conditions are met, that is if P is solved.

Finally as it can be proven that the boolean expression can be constructed in polynomial time, SAT is $\mathcal{NP}$-complete. □

SAT and the Halting problem

Theorem

The Halting problem is $\mathcal{NP}$-hard.

Sketch of proof. The basic idea consists in reducing SAT (2.116) to the Halting problem (2.115).

It suffices to transform SAT into a Turing machine that tries all possible assignments of truth values. If a solution is found, then halt and otherwise start an infinite loop.

Since SAT is $\mathcal{NP}$-complete it means that any problem P in $\mathcal{NP}$ can be reduced in polynomial time to the Halting problem. □

True quantified Boolean formula

Problem (True Quantified Boolean Formula (TQBF))

A quantified boolean formula is a formula that can be written in the form

$$Q_1 x_1 Q_2 x_2 \cdots Q_n x_n \phi(x_1, x_2, \cdots, x_n),$$

where each of the Q_i is one of the quantifier $\exists$ or $\forall$.

Calling L the language composed of the `True` quantified boolean formulae, decide L.

Example. The quantified boolean formula

$$\forall x_1 \exists x_2 \forall x_3 \left((x_1 \vee \neg x_2) \wedge (\neg x_1 \wedge x_3) \right)$$

is `False`.

Observe that x_3 must be `True` in order for the formula to evaluate as `True`. However x_3 is a universally quantifiable variable.

True quantified Boolean formula

Theorem

TQBF can be solved in exponential time and polynomial space.

Sketech of proof. The idea consists in exhibiting a polynomial-space algorithm that decides whether any given Quantified Boolean Formula (QBF) is true.

Consider a general input

$$\Psi = Q_1 x_1 Q_2 x_2 \cdots Q_n x_n \phi(x_1, x_2, \cdots, x_n),$$

where ϕ has m clauses. We want to use the same argument as in example 2.119, i.e. if Q_i is $\forall$ then both formulae, with x_i replaced by 0 and 1, must evaluate as `True`; if Q_i is $\exists$ then only one of them is required to be `True`.

True quantified Boolean formula

This is achieved by "pilling off" the quantifiers one by one. Starting with Q_1, one evaluates both

$$\Psi_{1_0} = Q_2 x_2 \cdots Q_n x_n \phi(0, x_2, \cdots, x_n) \text{ and}$$
$$\Psi_{1_1} = Q_2 x_2 \cdots Q_n x_n \phi(1, x_2, \cdots, x_n).$$

Then compute the logical value of $\psi_{1_0} \vee \psi_{1_1}$ or $\psi_{1_0} \wedge \psi_{1_1}$, depending whether Q_1 is $\exists$ or $\forall$.

Following this strategy all the quantifiers are recursively removed and it then suffices to solve a SAT problem.

The computational cost of such a strategy is very high since at each recursive call the problem is transformed into two new linearly smaller subproblems, resulting in a complexity of $\mathcal{O}(2^n)$.

True quantified Boolean formula

We now consider the space necessary to solve TQBF.

As Ψ_i is computed either from $\Psi_{(i+1)_0} \wedge \Psi_{(i+1)_1}$ or $\Psi_{(i+1)_0} \vee \Psi_{(i+1)_1}$, the two instances of $\Psi_{(i+1)}$ can be computed sequentially. Therefore they can use the same memory space.

However at each level of recursion the indices must be saved. This incurs an $\mathcal{O}(\log n)$ overhead, and as a result if computing Ψ_{i+1} requires space S_{i+1} then evaluating Ψ_i requires

$$S_i = S_{i+1} + \mathcal{O}(\log n).$$

Hence Ψ can be reached in $\mathcal{O}(n \log n)$ space, that is in polynomial space. □

PSPACE completeness

Definition

1. Let SPACE denote the set of all the decision problems which can be solved by a Turing machine in $\mathcal{O}(s(n))$ space for some function s of the input size n. Then PSPACE is defined as

$$\text{PSPACE} = \bigcup_k \text{SPACE}(n^k).$$

2. A problem Π is PSPACE-*complete* if and only if (i) Π is in PSPACE and (ii) for all P in PSPACE, P can be reduced in polynomial space to Π.

Theorem

TQBF is PSPACE-complete and in particular it is $\mathcal{NP}$-hard.

Hilbert's tenth problem

> *Given a Diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: To devise a process according to which it can be determined in a finite number of operations whether the equation is solvable in rational integers.*
>
> Hilbert – *List of Hilbert's problem list of 1900*

Problem (Hilbert's tenth problem)

Given a Diophantine equation with any number of unknown quantities and with rational integral numerical coefficients, decide whether the equation is solvable in rational integers.

Beyond Hilbert's tenth problem

After a 21 years long quest Matiyasevich proved that Hilbert's tenth problem was undecidable, that is there is no algorithm which can determine whether any given Diophantine equation is solvable in rational integers.

A side result however states that one can exhibit a Diophantine equation with no solution, but such that it is impossible to prove it.

The idea is to number all the Diophantine equations, $E_1, \cdots, E_n, \cdots$ following some set of axioms such as Peano arithmetic or Zermelo-Fraenkel set theory. The number of axioms and logical operators being finite, one can construct all the possible proof $P_1, \cdots, P_n \cdots$.

At stage 1, consider E_1 and check if:

- There exists a solution for the positive integers less than 1
- P_1 is a proof that E_1 has no solution

Beyond Hilbert's tenth problem

At stage k, consider $E_1, \cdots, E_k$ and check if:

- There exists a solution for the positive integers less than k
- $P_1, \cdots, P_k$ proves one of $E_1, \cdots, E_k$

Then there is at least one Diophantine equation which has neither a solution nor a proof.

This conclusion is drawn from *Gödel incompleteness Theorem* which states that for any axiomatic system built over the Peano arithmetic there is an undecidable problem.

In order to relate *Gödel incompleteness Theorem* to Turing machine we informally define what it means for a set of axioms to be *complete* and *consistent*.

Gödel and Turing

A set of axioms where any statement, built from it, or its negation can be proven in this set of axioms is said to be *complete*. It is *consistent* if it is impossible to prove both a statement and its negation in the set of axioms.

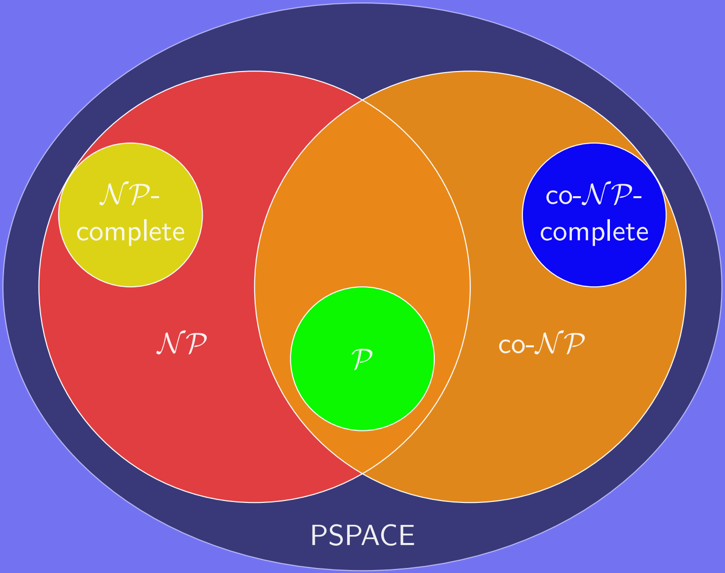
Assume we have a complete and consistent system F, powerful enough to reason about Turing machines. Taking a Turing machine M we can determine whether M halts by enumerating all the possible proofs in F, until one proof states that M halts or runs forever.

The completeness ensures of the correctness of the result while the consistency confirms the truthfulness of the conclusion.

We have proven that given F we can decide the Halting problem (2.115), which is known to be undecidable.↯

Therefore such a complete and consistent system F cannot exist.

Complexity diagram



A few last remarks

Results known to be true:

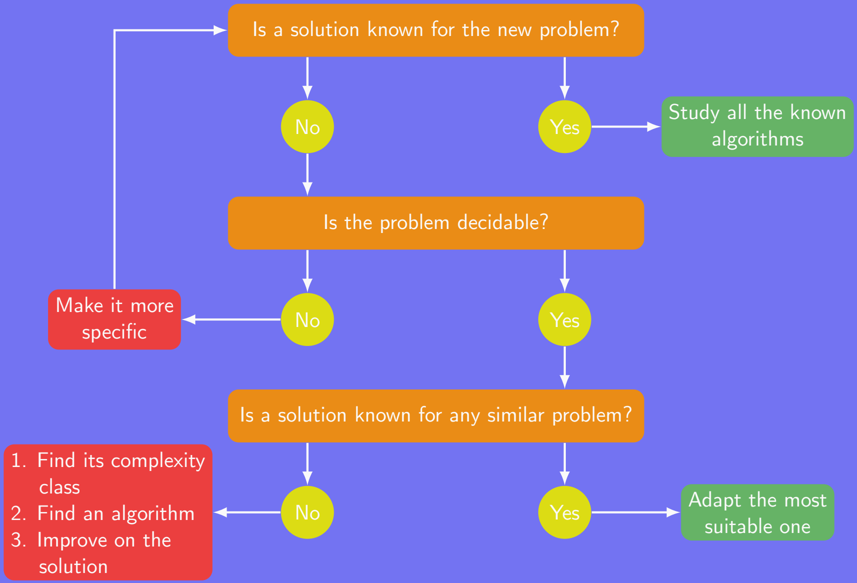
- $\mathcal{P} \subset \mathcal{NP}$
- $\mathcal{NP} \subset \text{PSPACE}$
- PSPACE = NPSPACE (Savitch's theorem)

Results believed to be true:

- $\mathcal{P} \neq \mathcal{NP}$
- $\mathcal{NP} \neq \text{PSPACE}$

More results on computational complexity theory are available at the Complexity Zoo.

Connection to algorithmic



Is a solution known for the new problem?
No
Yes
Study all the known algorithms
Is the problem decidable?
No
Make it more specific
Yes
Is a solution known for any similar problem?
No
1. Find its complexity class
2. Find an algorithm
3. Improve on the solution
Yes
Adapt the most suitable one

Conjunctive Normal Form

We now provide a formal reduction proof as an example of how to proceed to evaluate the complexity class of a given problem. We start with the following definition.

Definition

A boolean formula is said to be in *Conjunctive Normal Form (CNF)* if it is written as the conjunction of disjunctive clauses.

As a reminder the truth table of the conjunction and disjunction of A and B is

A	B	$A \wedge B$	$A \vee B$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

3-SAT

Problem (Satisfiability with 3 literals per clause (3-SAT))

Given a Boolean formula in CNF where each clause contains exactly three literals is there a truth assignment which satisfies all the clauses.

Theorem

3-SAT is $\mathcal{NP}$-complete.

Proof. Clearly 3-SAT is in $\mathcal{NP}$, for it is a particular case of SAT.

To prove that 3-SAT is $\mathcal{NP}$-hard we will show that being able to solve it implies being able to solve SAT, which by Cook theorem (2.117) is known to be $\mathcal{NP}$-hard.

3-SAT

Assuming CNF, we want to transform any instance of SAT into an instance of 3-SAT. Therefore we need to consider the cases where SAT has clauses with (i) one, (ii) two or (iii) more than three literals. Note that the case where there are exactly three clauses is already 3-SAT so no work is necessary.

(ii) The most simple case is when there are two literals, organised in a unique disjunctive clause $x_1 \vee x_2$ denoted (x_1, x_2). It then suffices to consider the pair of (x_1, x_2, u) and $(x_1, x_2, \neg u)$, where u is a newly added literal.

(i) Similarly in the case of a single literal x we convert it into a pair of literals (x, u_1) and $(x, \neg u_1)$, for a new literal u_1. It then suffices to apply the same strategy as above and create the four clauses

$$(x, u_1, u_2),\ (x, u_1, \neg u_2),\ (x, \neg u_1, u_2),\ (x, \neg u_1, \neg u_2).$$

3-SAT

The general idea behind (i) and (ii) is to add new literals which do not impact the satisfiability of the problem.

(iii) The case of a clause with more than three literals $(x_1, \cdots, x_n)$ can be treated by adding some new literals such as to create a chain of clauses where each of them has exactly three literals.

$$(x_1, x_2, u_3), (\neg u_3, x_3, u_4), (\neg u_4, x_4, u_5), \cdots, (\neg u_{n-1}, x_{n-1}, x_n).$$

The question is now to know if this change alters the satisfiability of the original clause.

Assume the original clause evaluates as `True`. Then at least one literal x_i must be `True`. Setting u_j to `True` for all $j \leq i$ and `False` for $j > i$ preserves the satisfiability of the clause.

3-SAT

If now the original clause evaluates as `False` we need to ensure that the altered version cannot be satisfied. If it is originally `False` then all the x_i must be `False`.

Therefore in the altered version the last clause can only be `True` if u_{n-1} is `False`. In turn this implies that u_{n-2} must also be `False`, and by extension all the u_i for $3 \leq i \leq n-2$. However in that case the first clause fails as u_3 is `False` and this means the conjunction of all the clauses evaluates as `False`.

In order to finalize the proof we need to ensure that the proposed transformations are applicable in polynomial time. This is clearly the case since the number of additional literals is polynomial in the number of original literals.

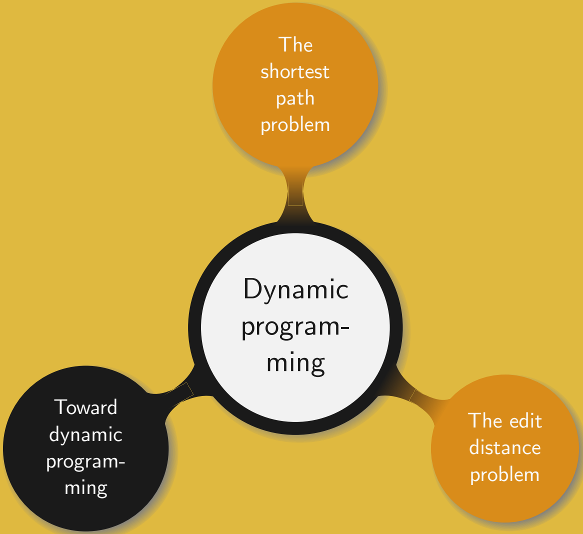
Hence $SAT \ltimes$ 3-SAT and 3-SAT is $\mathcal{NP}$-complete. □

Key points

- How to measure the running time of an algorithm?
- How is the growth rate evolving?
- What is the inverse Ackerman's function?
- Recall the Master Theorem.
- How to relate Turing machines to algorithms?

3. Dynamic programming

Chapter organisation



The shortest path problem
Dynamic programming
Toward dynamic programming
The edit distance problem

Fibonacci Numbers

Fibonacci posed the following problem in his book *Liber Abbaci* (Book of Calculations) in 1202:

> *A certain man puts a pair of rabbits in a place surrounded on all sides by a wall. How many pairs of rabbits can be produced from that pair in a year if it is supposed that every month each pair begets a new pair, which from the second month on becomes productive?*

The solution leads to the sequence of Fibonacci numbers:

- At the beginning of month 1, there is $F_0 = 1$ pair, which is not productive.
- At the beginning of month 2, there is still $F_1 = 1$ pair, which is now productive.
- At the beginning of month 3, there are now $F_2 = 2$ pairs, of which one is productive.

Fibonacci Numbers

- At the beginning of month 4, there are now $F_3 = 3$ pairs and the pair born in month 3 becomes productive.
- As the beginning of month 5, the number of pairs is equal to those of month 4, plus all those that were productive in month 4. These are all pairs that existed in month 2, since all of those will be productive in month 3. Hence, $F_4 = 3 + 2 = 5$.
- In general, at the beginning of every month the number of pairs of rabbits is equal to the number of pairs of the previous month, plus the number of pairs of two months ago, which have since become productive.

Hence,

$$F_n = F_{n-1} + F_{n-2}, \qquad n \geq 2, \qquad F_0 = 1, \qquad F_1 = 1.$$

A very naive approach

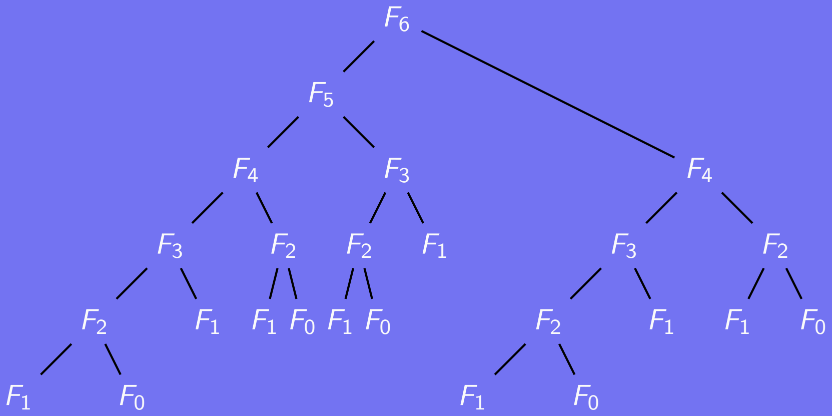
Algorithm.

```
Input  : An integer $n$
Output: $F_n$
Function Fib_vn($n$):
    if $n = 0$ then return 0;
    if $n = 1$ then return 1;
    return Fib_vn($n - 1$)+Fib_vn($n - 2$)
end
```

Since $F_{n+1}/F_n \approx \Phi = \frac{1+\sqrt{5}}{2} > 1.6$, it means that $F_n > 1.6^n$.

Noting that each recursive call has to reach 0 or 1 to be completed it is clear that the complexity of this algorithm is exponential.

A very naive approach



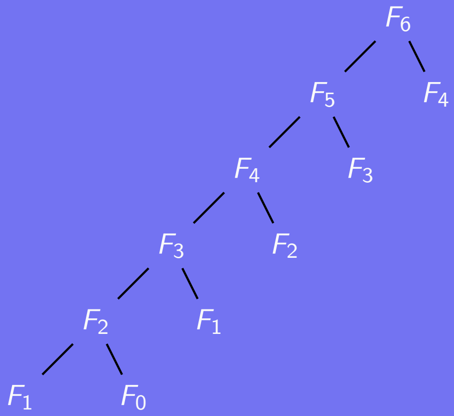
A naive approach

Algorithm.

```
Input   : An integer $n$
Output: $F_n$
Function Fib_n($n$):
    if $F_n = -1$ then $F_n \leftarrow$ Fib_n($n$-1) + Fib_n($n$-2);
    return $F_n$
end
Function Fib_nm($n$):
    $F_0 \leftarrow 0$;
    $F_1 \leftarrow 1$;
    for $i \leftarrow 2$ to $n$ do $F_i \leftarrow -1$;
    return Fib_n($n$)
end
```

Both the time and space complexities are linear in n

A naive approach



A proper approach

Algorithm.

Input : An integer n

Output: F_n

```
Function Fib($n$):
    $F_{old_2} \leftarrow 0$; $F_{old_1} \leftarrow 1$;
    if $n = 0$ then return $0$;
    for $i \leftarrow 2$ to $n - 1$ do
        $F \leftarrow F_{old_1} + F_{old_2}$; $F_{old_2} \leftarrow F_{old_1}$; $F_{old_1} \leftarrow F$;
    end for
    return $F_{old_1} + F_{old_2}$
end
```

The time is linear in n and the storage constant

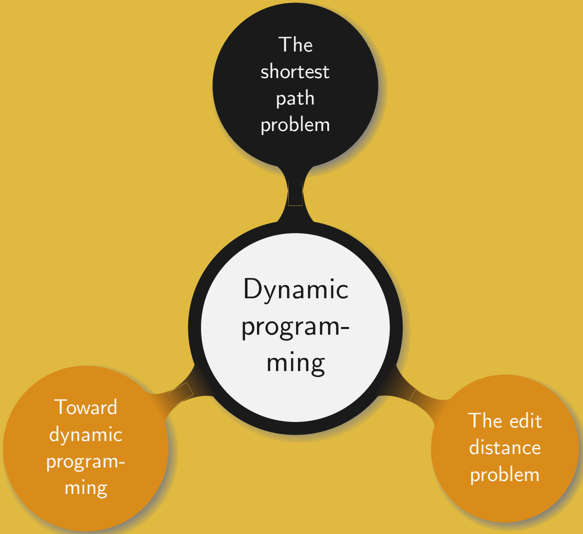
Dynamic programming

Simple idea behind dynamic programming:

- Break a complex problem into simpler subproblems
- Store the result of the overlapping subproblems
- Do not recompute the same information again and again
- Do not waste memory because of recursion

Dynamic programming saves both time and space

Chapter organisation


The shortest path problem
Dynamic programming
Toward dynamic programming
The edit distance problem

Shortest paths in weighted graphs

Problem (Shortest paths in weighted graphs)

Given a connected, simple, weighted graph, and two vertices s and t, find the shortest path that joins s to t.

Two main cases for the graph:

- It only has edges with positive labels
- It has edges with positive and negative labels

We now recall Dijkstra's algorithm to solve the first case and then introduce the Bellman-Ford algorithm which takes advantage of dynamic programming in order to treat the second one.

Dijkstra's Algorithm

Algorithm. (*Dijkstra*)

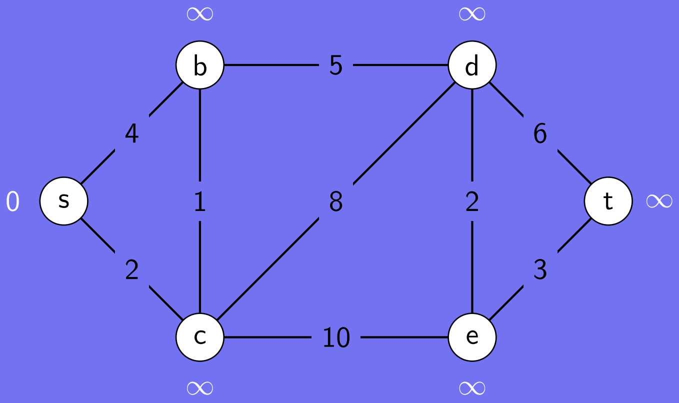
Input : A graph $G = \langle V, E \rangle$ with positive edges, two vertices s and t

Output : The shortest path between s and t

```
$s.dist \leftarrow 0$; $s.prev \leftarrow \mathtt{NULL}$; $S \leftarrow \emptyset$;
foreach vertex $v \in G.V$ do
    if $v \neq s$ then $v.dist \leftarrow \infty$; $v.prev \leftarrow \mathtt{NULL}$;
    add $v$ to $S$;
end foreach
$v \leftarrow s$;
repeat
    foreach neighbor $u$ of $v$ do
        $tmp \leftarrow v.dist + \mathrm{weight}(v, u)$;
        if $tmp < u.dist$ then $u.dist \leftarrow tmp$; $u.prev \leftarrow v$;
    end foreach
    remove $v$ from $S$; $v \leftarrow$ vertex with minimal distance in $S$;
until $v = t$;
return $t, t.prev, \cdots, s$
```

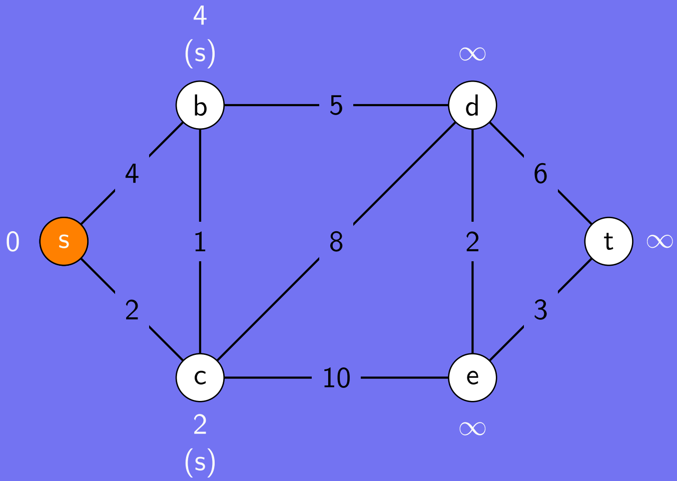
Example

$S = \{s, b, c, d, e, t\}$



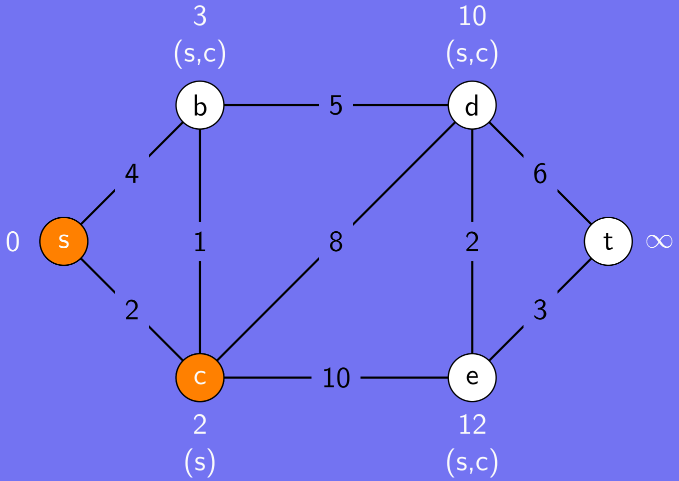
Example

$S = \{b, c, d, e, t\}$



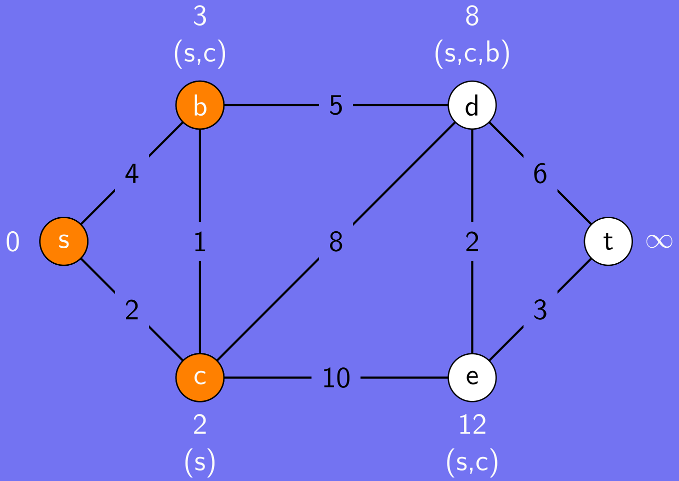
Example

$S = \{b, d, e, t\}$



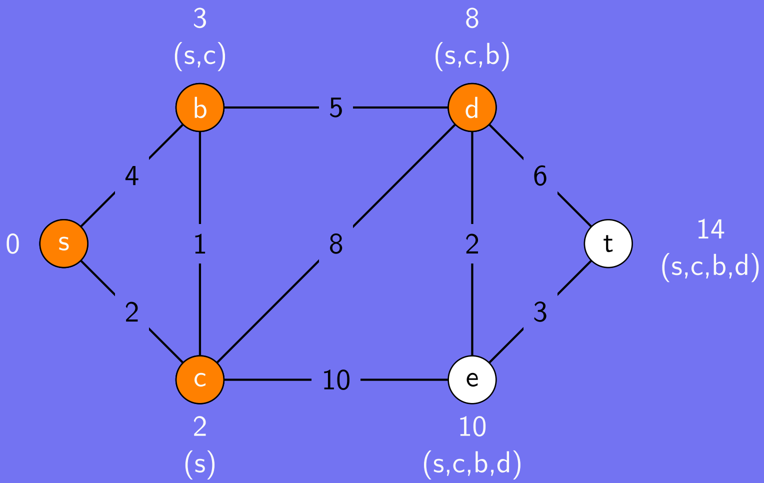
Example

$S = \{d, e, t\}$



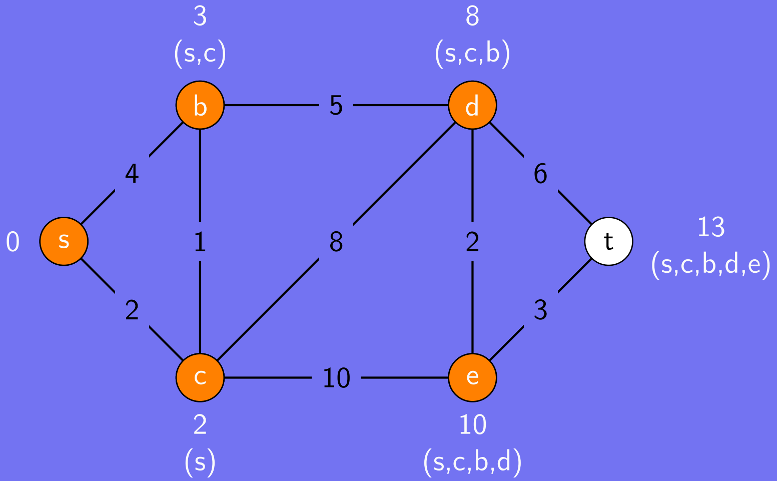
Example

$S = \{e, t\}$



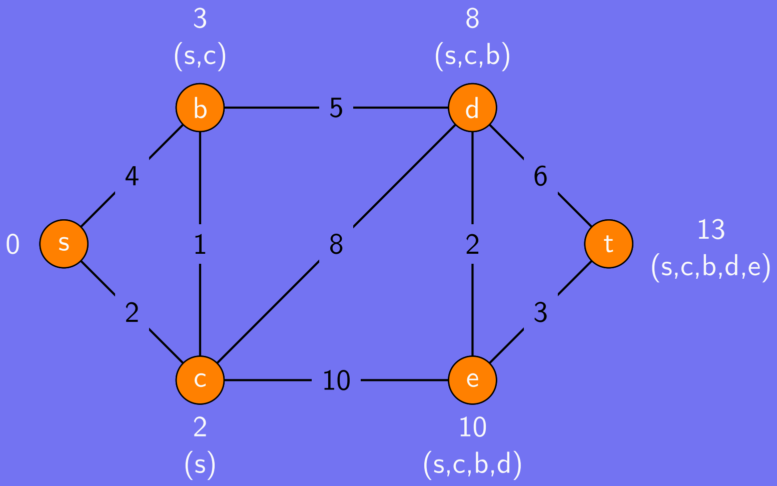
Example

$S = \{t\}$



Example

$S = \{t\}$, $v = t$



Shortest path with negative edges

Proposition

If G is a graph with no negative cycle, then there is a shortest simple path going from a source vertex s to a target vertex t.

Proof. Assume no cycle of negative cost exists. If the path repeated a vertex then removing edges to break this cycle would result in a path of no greater cost and with fewer edges. Therefore a shortest simple path exists. □

Given a graph with n nodes we can find a path of minimum cost which has length at most $n-1$. Let $opt(i,v)$ denote the minimum cost of a path from a vertex v to a vertex t and featuring at most i edges. The goal is to express $opt(i,v)$ into smaller subproblems such as to determine $opt(n-1,s)$ using dynamic programming.

Shortest path with negative edges

Lemma

Let P be an optimal path $opt(i, v)$ in a graph G, and (v, w) be the first edge in P. Then

$$opt(i, v) = \min(opt(i-1, v), opt(i-1, w) + \text{weight}(v, w))$$

Proof. The recursive formula is clear as soon as one notes that two cases can occur.

First if P has at most $i-1$ edges then $opt(i, v) = opt(i-1, v)$. On the other hand if P has i edges and the first one is (v, w), then

$$opt(i, v) = \text{weight}(v, w) + opt(i-1, w).$$

□

Bellman-Ford algorithm

Algorithm. (*Bellman-Ford*)

Input : A directed weighted graph $G = \langle V, E \rangle$ with no negative cycle, two vertices s and t

Output: The shortest path between s and t

```
foreach $v \in G.V$ do
    if $v \neq s$ then $v.dist \leftarrow \infty$; $v.prev = \mathrm{NULL}$;
    else $v.dist \leftarrow 0$;
end foreach
for $i \leftarrow 1$ to $|G.V| - 1$ do
    foreach $(u, v) \in G.E$ do
        $tmp \leftarrow u.dist + \mathrm{weight}(u, v)$;
        if $tmp < v.dist$ then $v.dist \leftarrow tmp$; $v.prev = u$;
    end foreach
end for
return $t, t.prev, \cdots, s$
```

Bellman-Ford algorithm

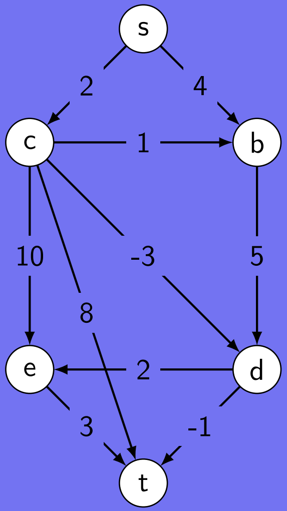
Theorem

Bellman-Ford algorithm is correct and runs in time $\mathcal{O}(mn)$ on a graph composed of n vertices and m edges.

Proof. The correctness of the algorithm follows from the recursive formula introduced in lemma 3.152.

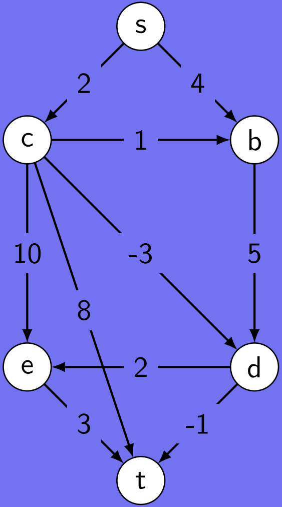
From lines 5 to 9 of Bellman-Ford algorithm (3.153) the loop on all the m edges is applied n times. □

Example



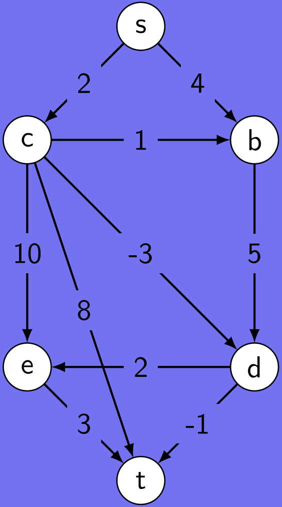
	0	1	2	3	4	5
s	0					
b	∞					
c	∞					
d	∞					
e	∞					
t	∞					

Example



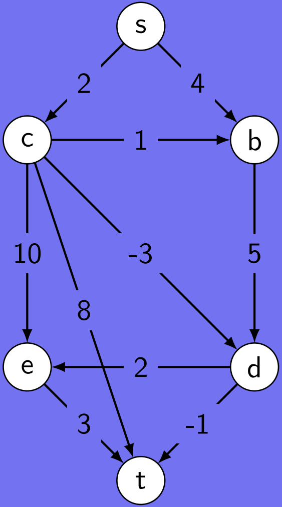
	0	1	2	3	4	5
s	0	0				
b	∞	4				
c	∞	2				
d	∞	∞				
e	∞	∞				
t	∞	∞				

Example



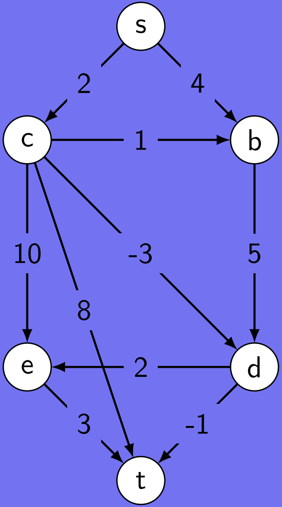
	0	1	2	3	4	5
s	0	0	0			
b	∞	4	3			
c	∞	2	2			
d	∞	∞	-1			
e	∞	∞	12			
t	∞	∞	10			

Example



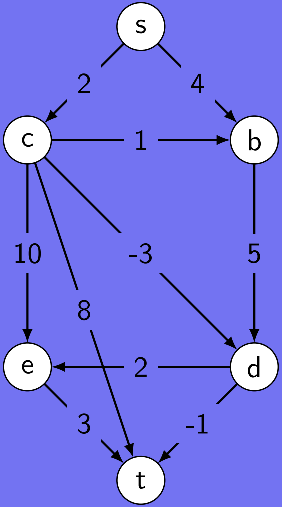
	0	1	2	3	4	5
s	0	0	0	0		
b	∞	4	3	3		
c	∞	2	2	2		
d	∞	∞	-1	-1		
e	∞	∞	12	1		
t	∞	∞	10	-2		

Example



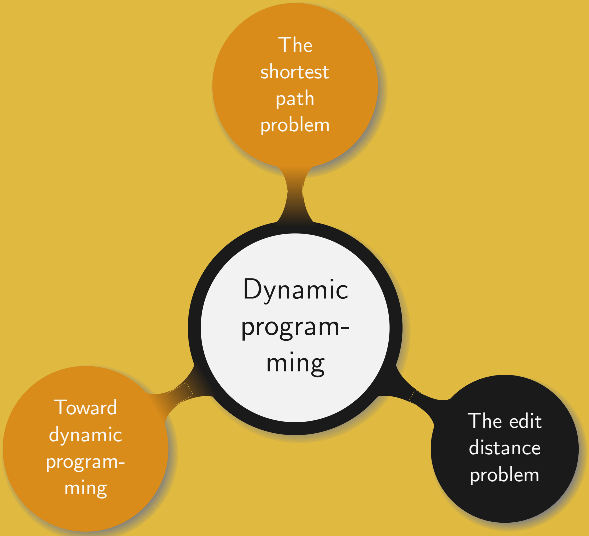
	0	1	2	3	4	5
s	0	0	0	0	0	
b	∞	4	3	3	3	
c	∞	2	2	2	2	
d	∞	∞	-1	-1	-1	
e	∞	∞	12	1	1	
t	∞	∞	10	-2	-2	

Example



	0	1	2	3	4	5
s	0	0	0	0	0	0
b	∞	4	3	3	3	3
c	∞	2	2	2	2	2
d	∞	∞	-1	-1	-1	-1
e	∞	∞	12	1	1	1
t	∞	∞	10	-2	-2	-2

Chapter organisation



Dynamic program-ming

The shortest path problem

Toward dynamic program-ming

The edit distance problem

Intuition of the problem

Given two strings determine whether they match

Comments on the problem:

- Simple to implement
- How to render misspellings?

This is useless in practice where we are more interested in measuring how far two strings are from each others.

Typical applications:

- Spell checker
- Changes in language usage
- DNA sequencing
- Plagiarism

The edit distance problem

Problem (Edit distance)

The number of changes that need to be performed to convert a string into another one defines the *distance* between the two strings. The three types of alterations considered are:

- *Substitution:* a single character is replaced by a different one
- *Insertion:* a single character is added such as to decrease the distance between the two strings
- *Deletion:* a single character is deleted in order to match the other string

Given two strings and assigning distance one to each of these operations determine the *edit distance* between them.

Tackling the problem

Naive idea: search exact places where to add/delete characters

Alternative view:

- What information is needed to decide on what operation to perform?
- What can happen to the last character for each string?

If an optimal solution is known for all the characters but the last, then it becomes simple to find an overall best solution: check the three possibilities, add the cost of each of them to the previous minimal cost and select the best option.

A recursive approach

Given a reference string S and a string T we want to determine their edit distance. At each step, i.e. letter, a decision can be taken upon the previous results:

- If $S_i = T_j$, then consider $dist_{i-1,j-1}$. Otherwise consider $dist_{i-1,j-1}$ and pay a cost 1 for the difference
- If $S_{i-1} = T_j$, then it could be that T has one more character than S. In that case consider $dist_{i-1,j}$ and pay a cost 1 for the insertion of a character in S
- If $S_i = T_{j-1}$, then it could be that T has one less character than S. In that case consider $dist_{i,j-1}$ and pay a cost 1 for the deletion of a character in S

As those three possibilities cover all the cases, taking their minimum yields the edit distance.

Limitations of the recursive approach

Description of the strategy:

- If either of the index is 0 then set *dist* to the other index
- Compute

$$dist_{i,j} = \min(dist_{i-1,j} + 1, dist_{i,j-1} + 1, dist_{i-1,j-1} + (\text{match}(S_i, T_j) \; ? \; 0 \; : \; 1))$$

How fast would be the algorithm?

- At each position in the string, three branches are explored
- Only one branch reduces both indices
- Exponential time $\Omega(3^n)$

How to do better?

- What is the maximum number of pairs?
- The same pairs are recalled many times
- Use a lookup table to decrease the computational cost

Solving edit distance

Algorithm.

Input : Two strings S and T

Output: The edit distance between S and T

```
for $i \leftarrow 0$ to $|S|$ do $dist_{i,0} \leftarrow i$;
for $i \leftarrow 1$ to $|T|$ do $dist_{0,i} \leftarrow i$;
for $i \leftarrow 1$ to $|S|$ do
    for $j \leftarrow 1$ to $|T|$ do
        $tmp_0 \leftarrow dist_{i-1,j-1} + (\text{match}(S_i, T_j) \ ?\ 0 \ :\ 1)$;
        $tmp_1 \leftarrow dist_{i,j-1} + 1$;            /* skip a letter in $S$ */
        $tmp_2 \leftarrow dist_{i-1,j} + 1$;           /* skip a letter in $T$ */
        $dist_{i,j} \leftarrow \min(tmp_0, tmp_1, tmp_2)$;
    end for
end for
return $dist_{i,j}$
```

Example

		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	**6**

Solving edit distance

Theorem

Let S and T be two strings of length n and m, respectively. Algorithm 3.162 solves edit distance in time $\mathcal{O}(nm)$.

Proof. Algorithm 3.162 is simply a table look up version of the recursive algorithm described on slide 3.160. As all the cases are considered in the initial recursive approach they are also all covered in Algorithm 3.162.

Looking at the description of the algorithm it is clear that most of the work is performed in the nested for loops. This results in the creation of a $n \times m$ table. All the other operations are only reading from this lookup table. Therefore the complexity is $\mathcal{O}(nm)$. □

A few more remarks

Dynamic programming:

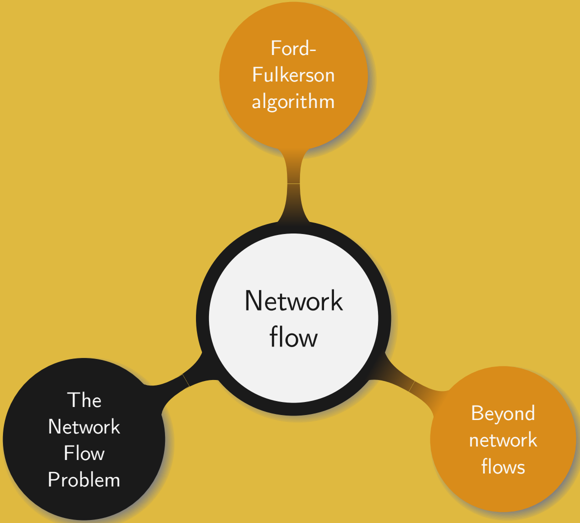
- Simple idea dramatically improving space-time complexity
- Cover all the possibilities at the subproblems level
- Never recompute anything that is already known
- Efficient as long as the number of subproblems remains polynomial

Key points

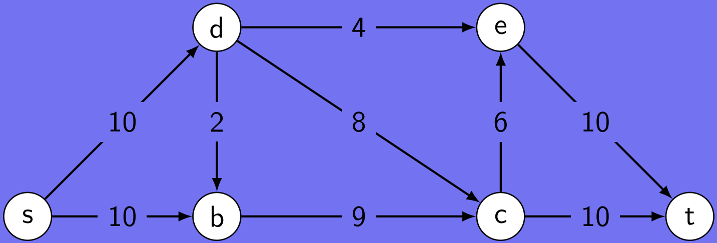
- Explain dynamic programming in three words
- When to use Dijkstra and Bellman-Ford algorithms?
- Describe the edit distance problem?
- Why is recursion not a good strategy to solve edit distance?

4. Network flow

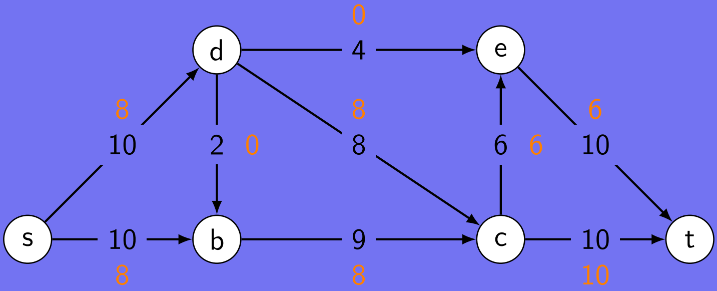
Chapter organisation



A toy example

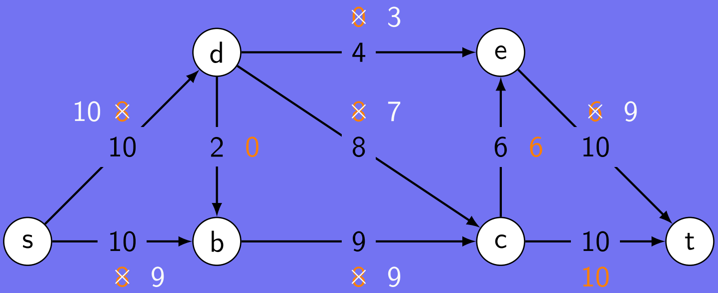


A toy example



Total flow: 16

A toy example



Total flow: 19

Informal approach

Let $G = \langle V, E \rangle$ be a directed graph. We consider each node of G as a switch and each edge as carrying some traffic. This is for instance the case in the context of a highway system: nodes are the interchanges (hubs) and the edges the highway itself. It could also be illustrated using a fluid network where the edges are the pipes and the nodes the junctures where the pipes are plugged together.

To each edge one associates a number called *capacity*, which represents how much traffic an edge can handle. In the maximum network flow problem the goal is to arrange the traffic such as optimizing the available capacity.

Before being able to solve this problem we need to formalize the idea of flow.

Formalization

Definition

Let $G = \langle V, E \rangle$ be a weighted directed graph with a *source* node s and a *sink* node t; G is called a *flow network*.
Given a function $c : E \to \mathbb{R}^+$, called *capacity*, a *flow* is a function $f : E \to \mathbb{R}^+$, satisfying the following properties.

i *Capacity constraint:* the flow of an edge can never exceed its capacity, i.e. $\forall (u, v) \in E,\ \ f(u, v) \leq c(u, v)$.

ii *Flow conservation:* at each node the entering flow equals the exiting flow, i.e.

$$\forall u \in V \setminus \{s, t\}\,,\ \sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v).$$

Remark. Unless specified otherwise, we restrict our attention to the case of integer or at least rational capacities and flows.

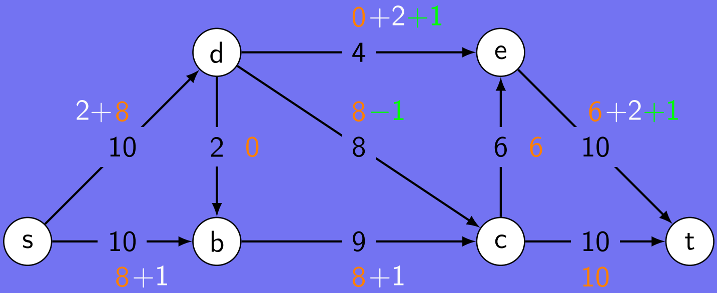
Formalization

Problem (Maximum Network Flow)

Given a flow network arrange the flow such as to maximize the available capacity.

Remark. Assume a network flow is divided into two sets A and B of empty intersection and s is in A while t is in B. Intuitively any flow travelling from s to t must cross from A to B. This suggests that the capacity of the network flow can never exceed the capacity of the *cuts* A and B. The minimum capacity of any such division is called the *minimum cut*. We will prove that it is equal to the maximum flow value.

Back to the toy example



Improving the original path:

- Add $+2$ along $s \to d \to e \to t$; total flow: 18
- Add $+1$ along $s \to b \to c$: c is a bottleneck; total flow: 18
- Reallocate 1 from $d \to c \to t$ to $d \to e \to t$; total flow: 19

Informal strategy

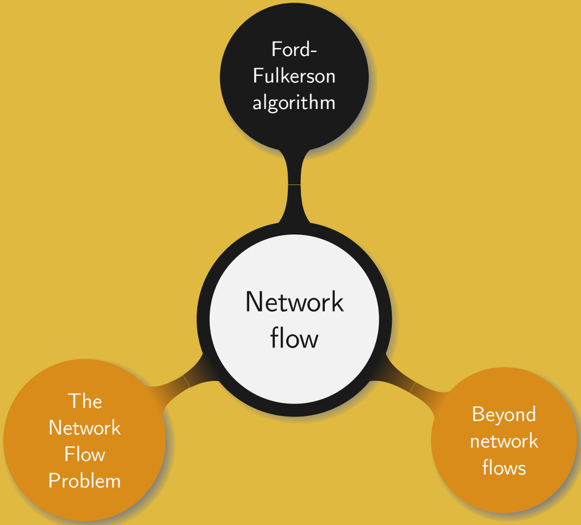
Finding the maximum flow can be achieved as follows:

1. Start with a null flow
2. Find paths to increase the flow:
   - Path where the capacity has not been reached
   - Reallocate flow to a different path and increase the capacity of the current one
3. Stop when no more flow can be injected

Goals:

- Determine all the paths that allow an increase of the flow
- Make sure the search for such paths stops at some stage

Chapter organisation



Residual graph

Definition (Residual graph)

Let G be a graph and f be a flow network. The *residual graph* G_f of G with respect to f is the graph whose vertices are the vertices of G; Regarding its edges:

- Each edge e of G with capacity $c(e)$ and flow $f(e) < c(e)$ is added to G_f with capacity $c(e) - f(e)$; it is a *forward edge*.
- For each edge $e = (u, v)$ of G with flow $f(e) > 0$, a *backward edge* $e' = (v, u)$ with capacity $c(e') = f(e)$ is added; it is a *backward edge*.

For a path P, the maximum amount by which the flow can be increased on each edge along P is called the *residual capacity*.

Remark. Given a graph G its residual graph G_f has at most twice the number of edges as G.

Augmenting paths

Algorithm. (*Augment*)

Input : a flow f and a simple path P
Output: a new flow

```
Function Augment($f$, $P$):
    $b \leftarrow$ minimum residual capacity on $P$ with respect to $f$;
    foreach edge $e \in P$ do
        if $e$ is a forward edge then $f(e) \leftarrow f(e) + b$;
        else $f(e) \leftarrow f(e) - b$;
    end foreach
    return $f$
end
```

Augmenting path

Lemma

The Augment algorithm (4.178) returns a flow in the original graph G.

Proof. We must ensure that the output of the algorithm matches definition 4.172, that is satisfies the capacity constraint and the flow conservation properties.

Consider the edges from the residual graph G_f whose capacity differ from the ones of G. In the case of e being such a forward edge with capacity $c(e) - f(e)$, we have

$$0 \leq f(e) \leq f(e) + b \leq f(e) + (c(e) - f(e)) = c(e).$$

If e is a backward edge then its residual capacity is $f(e)$ and

$$c(e) \geq f(e) \geq f(e) - b \geq f(e) - f(e) = 0.$$

Hence the capacity constraint holds whether e is a forward or backward edge.

Augmenting path

Intuitively the algorithm will return a flow since f itself is a flow. A more formal prove involves looking at the edges depending on whether they are backward or forward edges and compare the entering flow to the exiting one. □

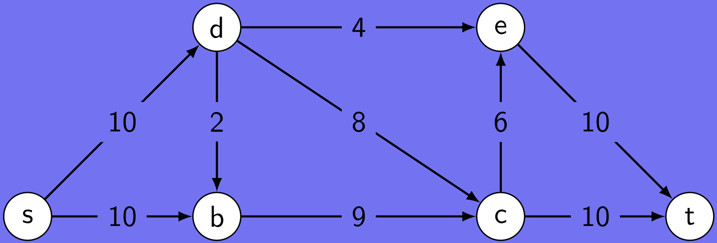
Algorithm. (*Ford-Fulkerson*)

Input : A graph G
Output : A maximum flow in G

```
foreach edge $e$ in $G$ do
    $f(e) \leftarrow 0$;
end foreach
while there is a path $P$ in the residual graph $G_f$ do
    $f \leftarrow \texttt{Augment}(f, P)$;
    update the residual graph $G_f$;
end while
return $f$
```

Exercise

Apply Ford-Fulkerson algorithm (4.180) to our initial toy example.



Cuts

Definitions

Let $G = \langle V, E\rangle$ be a flow network and s and t be the source and sink vertices, respectively.

1. A *cut* is a partition of V into two connected subsets S and $T = V \setminus S$, with $s \in S$ and $t \in T$.
2. Given a flow f the *net flow* across the cut (S, T) is defined as

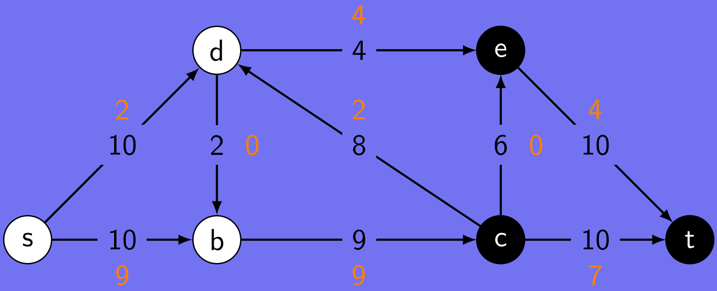
$$f(S, T) = \sum_{u\in S}\sum_{v\in T} f((u, v)) - \sum_{u\in S}\sum_{v\in T} f((v, u)).$$

3. The *capacity* of the cut (S, T) is defined as

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

4. A *minimum cut* is a cut of minimum capacity over all the cuts of the network.

Example



- Cut: $S = \{s, d, b\}$ and $T = \{c, e, t\}$
- Net flow: $4 + 9 - 2 = 11$
- Capacity of the cut: $9 + 4 = 13$

Net flow across cuts

Definition

Let $G = \langle V, E \rangle$ be a flow network, s be the source, and f be a flow on G. We denote the *value* of the flow f by

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

We will now prove that given a flow f the net flow across any cut is the same.

Lemma

Let f be a flow in a flow network $G = \langle V, E \rangle$ with source s and sink t. For any cut (S, T) of G, the net flow across (S, T) is $f(S, T) = |f|$.

Net flow across cuts

Proof. From the flow conservation (4.172) for any node $u \in V \setminus \{s, t\}$

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

Adding it to the value of the flow we get

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) + \sum_{u \in S \setminus \{s\}} \left(\sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u) \right).$$

Rewriting the right-hand sum yields

$$|f| = \sum_{v \in V} \sum_{u \in S} f(u, v) - \sum_{v \in V} \sum_{u \in S} f(v, u).$$

Net flow across cuts

Noting that $S \cup T = V$ and $S \cap T = \emptyset$, the sum over the elements of V can be split over S and T to obtain

$$\begin{aligned}
|f| &= \sum_{v \in S} \sum_{u \in S} f(u, v) + \sum_{v \in T} \sum_{u \in S} f(u, v) - \sum_{v \in S} \sum_{u \in S} f(v, u) - \sum_{v \in T} \sum_{u \in S} f(v, u) \\
&= \sum_{v \in T} \sum_{u \in S} f(u, v) - \sum_{v \in T} \sum_{u \in S} f(v, u) + \left(\sum_{v \in S} \sum_{u \in S} f(u, v) - \sum_{v \in S} \sum_{u \in S} f(v, u) \right) \\
&= \sum_{v \in T} \sum_{u \in S} f(u, v) - \sum_{v \in T} \sum_{u \in S} f(v, u) \\
&= f(S, T)
\end{aligned}$$

Hence the net flow across the cut (S, T) is the value of the flow f. □

Upper bound on a flow

Lemma

Given a flow network G and a flow f, f is upper bounded by the capacity of any cut in G.

Proof. Using lemma 4.184 and the capacity constraint (4.172) we have

$$\begin{aligned}|f| = f(S,T) &= \sum_{v\in T}\sum_{u\in S} f(u,v) - \sum_{v\in T}\sum_{u\in S} f(v,u)\\ &\le \sum_{v\in T}\sum_{u\in S} f(u,v) \le \sum_{v\in T}\sum_{u\in S} c(u,v)\\ &= c(S,T)\end{aligned}$$

□

Max-flow Min-cut Theorem

From the previous lemma (4.187) we can conclude that the maximum flow is upper bounded by the capacity of a minimum cut. We now prove that they are in fact equal.

Theorem (Max-flow Min-cut Theorem)

Let f be a flow in a flow network $G = \langle V, E \rangle$, with source s and sink t. Then following conditions are equivalent.

i. The flow f is a maximum flow in G.

ii. The residual network G_f contains no augmenting path.

iii. The value of the flow $|f|$ is equal to the capacity of some cut (S, T) of G.

Max-flow Min-cut Theorem

Proof. $(i) \Rightarrow (ii)$: we suppose that f is a maximum flow but G_f has an augmenting path. This generates a flow with value strictly larger than $|f|$, which contradicts lemma 4.187.↯

$(ii) \Rightarrow (iii)$: we suppose that G_f has no augmenting path, i.e. has no path from s to t. Let $S = \{v \in V \ / \text{ there is a path from } s \text{ to } v\}$, and $T = V \setminus S$. Clearly (S, T) is a cut, since $s \in S$ and $t \notin S$ for otherwise there would be a path from s to t.

Let $u \in S$ and $v \in T$ be a pair of vertices.
If $e = (u, v)$ is in E then $f(e) = c(e)$, otherwise e would be in E_f and v would be in S.↯
If $e = (v, u)$ is in E then $f(e) = 0$, otherwise $c_f((u, v)) = f(e)$ would be positive, (u, v) would be in E_f, and v would be in S.↯

Max-flow Min-cut Theorem

For the last case, note that if neither (u, v) nor (v, u) belongs to E it means that $f((u, v)) = f((v, u)) = 0$.

Thus we have

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

Therefore, by lemma 4.184, $|f| = c(S, T)$.

$(iii) \Rightarrow (i)$: combining lemma 4.187 with $|f| = c(S, T)$ yields (i). □

Finding paths

While the Max-flow Min-cut theorem (4.188) provides a proof of accuracy for Ford-Fulkerson algorithm (4.180), the question of its efficiency remains unanswered.

In fact the algorithm still features an unclear request: "while there is a path P in the residual graph G_f". So the question boils down to knowing "how to determine all the paths and what is the cost".

A simple implementation of the algorithm would complete as the value of the flow keeps increasing by at least one unit while never exceeding the maximum flow f^*. Therefore in the worst case the complexity is $\mathcal{O}(|E||f^*|)$.

On the other hand if edges have a large capacity this strategy is not optimal and finding a "good path" becomes of a major importance. In practice this can be done using the breadth-first search algorithm which will allow us to determine the shortest path.

Algorithm. (*Breadth-First Search (BFS)*)

Input : a graph G and a starting vertex s

Output: none – only access all the vertices accessible from s

```
foreach vertex $v$ in $G$ do $v.dist \leftarrow \infty$; $v.parent \leftarrow \texttt{NULL}$;
$Q \leftarrow \emptyset$; $s.dist \leftarrow 0$; append $s$ to $Q$;
while $Q \neq \emptyset$ do
    $v \leftarrow$ first element in $Q$; remove $v$ from $Q$;
    foreach vertex $u$ adjacent to $v$ do
        if $u.dist = \infty$ then
            $u.dist \leftarrow v.dist + 1$; $u.parent \leftarrow v$; append $u$ to $Q$;
        end if
    end foreach
end while
```

Finding paths

Theorem

Given a graph $G = \langle V, E \rangle$, the complexity of Breadth-First search is $\mathcal{O}(|E|)$.

Proof. The result is straight forward as the whole adjacency list is to be scanned.

□

We now prove that BFS correctly computes the shortest path distance between s and any vertex v.

Shortest distance

Lemma

Let $G = \langle V, E \rangle$ be a graph and s be a source vertex in G. Upon running BFS on G, for any vertex v, the computed value $v.dist$ is larger than the shortest-path distance between s and v.

Proof. Let δ_v be the shortest distance between s and v. We will prove the result by induction on the number of "append" operations, the hypothesis being $v.dist \geq \delta_v$.

Base case: the first element to be appended to the list is s for which $\delta_s = s.dist = 0$; for all other vertices $\delta_v \leq v.dist = \infty$.

Shortest distance

Induction step: let v and u be two vertices where u is discovered from v. Noting that $u.dist = v.dist + 1$ we apply the induction hypothesis to v and get

$$\begin{aligned} u.dist &\geq \delta_v + 1 \\ &\geq \delta_u \end{aligned}$$

Noting that a vertex in never appended more than once, $u.dist$ will not be updated. Therefore the induction principle applies and for any vertex v in V, $v.dist$ is larger than the shortest-path distance between s and v. □

We now want to prove that (i) nodes in Q are ordered with respect to their distance, and (ii) the first and last have a distance difference of at most one.

Shortest distance

Lemma

Let $G = \langle V, E\rangle$ be a graph and s be a source vertex in G. Upon running BFS on G, Q contains the vertices $\{v_1, v_2, \cdots, v_r\}$ where v_1 and v_r are the head and tail, respectively. Then $v_r.dist \leq v_1.dist + 1$ and for $1 \leq i \leq r - 1$, $v_i.dist \leq v_{i+1}.dist$.

Proof. We prove the result by induction on the number of operations on Q, that is we take into account both "append" and "remove".

Base case: clearly the result holds when Q only contains s.

Induction step: two cases must be considered, when (i) removing and (ii) appending an element.

(i) When removing v_1 two cases can arise: either Q becomes empty and in this case the result holds vacuously, or the second element, v_2 becomes the head.

Shortest distance

If v_2 is the new head then by the induction hypothesis $v_1.dist$ was less that $v_2.dist$. But as by induction $v_r.dist$ was less than $v_1.dist + 1$, we conclude that $v_r.dist \leq v_2.dist + 1$. All the remaining equalities being unaffected by the change of head, the result holds when removing an element.

(ii) When exploring the vertices adjacent to some vertex v, v has already been removed from Q. Let u be a neighboring vertex from v. Then u is appended to Q and can be renamed as v_{r+1}.
By the induction hypothesis $v.dist \leq v_1.dist$. Thus $v_{r+1}.dist$ is the same as $v.dist + 1$ which is less than $v_1.dist + 1$. Furthermore by the induction hypothesis $v_r.dist \leq v.dist + 1$, such that $v_r.dist$ is less than $v.dist + 1 = v_{r+1}.dist$. All the remaining equalities are unaffected.

Therefore by the induction principle lemma 4.196 holds. □

Shortest distance

Theorem

Let $G = \langle V, E \rangle$ be a graph and s be a source vertex in G. Then BFS discovers all the vertices v in V reachable from s and upon termination for all of them $v.dist$ is the shortest distance δ_v.

Proof. Let $v \in V$ be a vertex such that $v.dist \neq \delta_v$. Then by lemma 4.194 $v.dist > \delta_v$. Clearly v must be reachable from s otherwise δ_v would be $\infty \geq v.dist$.

Let u be the vertex immediately preceding v on the shortest path to v. Then $\delta_v = \delta_u + 1$ and $u.dist = \delta_u$. Hence

$$v.dist > \delta_v = \delta_u + 1 = u.dist + 1 \tag{4.1}$$

The vertex v can be in three states: (i) neither in Q nor visited, (ii) in Q, and (iii) not in Q but visited.

Shortest distance

(i) When v is visited $v.dist$ is set to $u.dist + 1$, which contradicts equation (4.1).↯

(ii) If v is in Q then it means it was added during the exploration of the neighbors of a vertex w. Either $u = w$ or w was removed from Q earlier than u. So $v.dist = w.dist + 1$ and $w.dist \leq u.dist$. Therefore $v.dist = w.dist + 1 \leq u.dist + 1$, contradicting (4.1).↯

(iii) If v has already been removed from Q then clearly $v.dist < u.dist$, which once again contradicts (4.1).↯

Hence for all v in V, $v.dist = \delta_v$. Moreover all the vertices from V must be discovered otherwise if such a vertex v existed, then δ_v would be smaller than $v.dist = \infty$. □

The Edmonds-Karp algorithm

In order to evaluate the complexity of the Ford-Fulkerson algorithm (4.180), the question of how to find an augmenting path had to be answered.

We now prove that a shortest path in the residual network can be discovered using BFS and then used as an augmenting path. The resulting algorithm is called the Edmonds-Karp algorithm.

Lemma

Let $G = \langle V, E \rangle$ be a flow network with source s and sink t. If the Edmonds-Karp algorithm is run on G then for any vertex v in $V \setminus \{s, t\}$, the shortest-path distance $\delta_{f,v}$ in the residual network G_f increases monotonically with each flow augmentation.

Shortest path as augmenting path

Proof. Suppose that there exists a vertex $v \in V \setminus \{s, t\}$ such that a flow augmentation causes the shortest-path distance between s and v to decrease.

Let f and f' be the flows just before and after the augmentation that decreases the shortest-path distance, respectively. Then for v we have $\delta_{f',v} < \delta_{f,v}$. If u is the vertex visited just before v in $G_{f'}$ then $\delta_{f',u} = \delta_{f',v} - 1$.

From the choice of v, the shortest distance between s and u did not decrease on the flow augmentation. Therefore $\delta_{f,u} \leq \delta_{f',u}$ and (u, v) cannot belong to E_f. In fact if this was the case then we would have

$$\begin{aligned} \delta_{f,v} &\leq \delta_{f,u} + 1 \\ &\leq \delta_{f',u} + 1 \\ &= \delta_{f',v}. \end{aligned}$$

This contradicts the assumption $\delta_{f',v} < \delta_{f,v}$.↯

Shortest path as augmenting path

Since (u, v) does not belong to E_f but then belongs to $E_{f'}$ it means that the flow has been increased from v to u. However as the Edmonds-Karps applies BFS to augment the flow along the shortest path we conclude that the shortest path from s to u has (v, u) as its last edge. Therefore,

$$\begin{aligned} \delta_{f,v} &= \delta_{f,u} - 1 \\ &\leq \delta_{f',u} - 1 \\ &= \delta_{f',v} - 2. \end{aligned}$$

Again this contradicts the assumption $\delta_{f',v} < \delta_{f,v}$. ↯

Hence there exists no vertex $v \in V \setminus \{s, t\}$ such that a flow augmentation causes the shortest-path distance between s and v to decrease. □

Edmonds-Karp complexity

Theorem

Let $G = \langle V, E \rangle$ be a flow network with source s and sink t. If the Edmonds-Karp algorithm is run on G then it returns a maximum flow in time $\mathcal{O}(|V||E|^2)$.

Proof. Given a path P in the residual network G_f, we call an edge e such that $c_f(P) = c_f(e)$ a *critical edge*. We immediately notice that (i) such an edge disappears from the residual network as soon as an augmentation is performed along P, and (ii) at least one edge on any augmenting path is critical.

In order to determine the complexity of the Edmonds-Karp algorithm we must figure out how many flow augmentation are performed.

Edmonds-Karp complexity

Let u and v be two vertices and $e = (u, v)$ an edge in E. Since augmenting paths are shortest paths we have $\delta_{f,v} = \delta_{f,u} + 1$.

If e is a critical edge then it will disappear form the residual network as soon as the flow is augmented. It can however reappear later after the flow from u to v is decreased, that is if (v, u) is part of an augmenting path. The question is then to know how many times it can disappear and reappear.

Let f' be the flow in G when (v, u) appears in an augmenting path. Then $\delta_{f',u} = \delta_{f',v} + 1$. Moreover as $\delta_{f,v} \leq \delta_{f',v}$ (4.200) we get

$$\begin{aligned}\delta_{f',u} &= \delta_{f',v} + 1\\ &\geq \delta_{f,v} + 1\\ &= \delta_{f,u} + 2\end{aligned}$$

Edmonds-Karp complexity

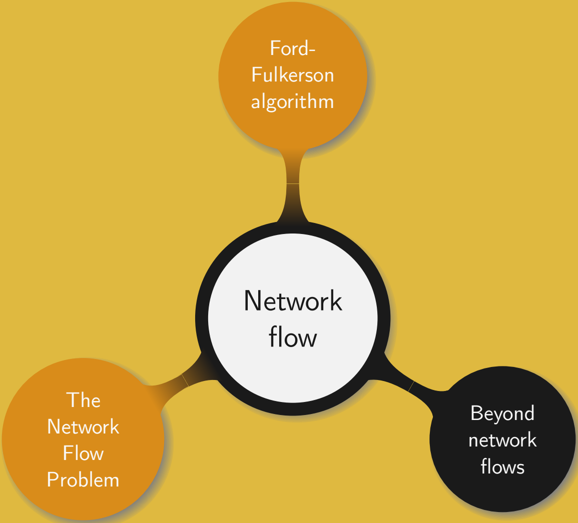
Thus, if e is a critical edge then the next time it becomes critical the distance between the s and u has increased by at least 2, while it originally was at least 1.

Then observing that the path from s to u cannot contain u, or t, we conclude that the distance will be at most $|V| - 2$.

Finally by combining the two bounds we conclude that an edge is critical no more than $|V|/2$ times. And as the number of edges in the residual network is $\mathcal{O}(|E|)$, the total number of critical edges during the execution of the algorithm is $\mathcal{O}(|V||E|)$.

Recalling that an augmenting path has at least one critical edge, we loop $\mathcal{O}(|V||E|)$ times in the Edmonds-Karp algorithm. For each of them, BFS with complexity $\mathcal{O}(|E|)$ (theorem 4.193) is run, leading to a total cost of $\mathcal{O}(|V||E|^2)$. □

Chapter organisation



Network flow

Ford-Fulkerson algorithm

The Network Flow Problem

Beyond network flows

General view on algorithms

When studying computability theory the importance of finding similarities between problems was highlighted (2.130). The idea behind this approach is to solve new problems by deriving appropriate algorithms from known ones.

The difficulty is therefore to view a problem from a different perspective. In practice this is similar to determining polynomial reductions in computability theory (2.113): one wants to efficiently "rephrase " a given problem into an new one for which a solution is known.

We now study such an example as we solve the maximum bipartite matching problem using network flows.

Maximum Bipartite Matching

Definition

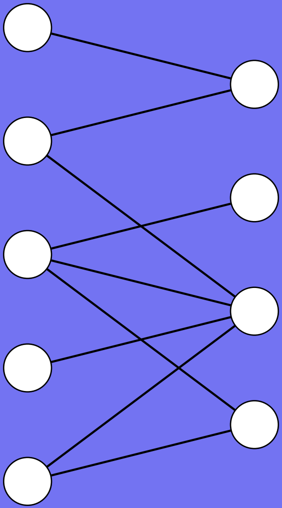
Let $G = \langle V, E \rangle$ be a graph.

1. If $V = L \cup R$, with L and R two disjoint sets, every vertex in V has at least one incident edge, and for any edge (u, v) either $u \in L$ and $v \in R$ or $u \in R$ and $v \in L$, then G is called *bipartite graph*.
2. A *matching* is a subset M of E such that for any vertex $v \in V$ at most one edge in M is incident to v.
3. A *maximum matching* is a matching of maximum cardinality.

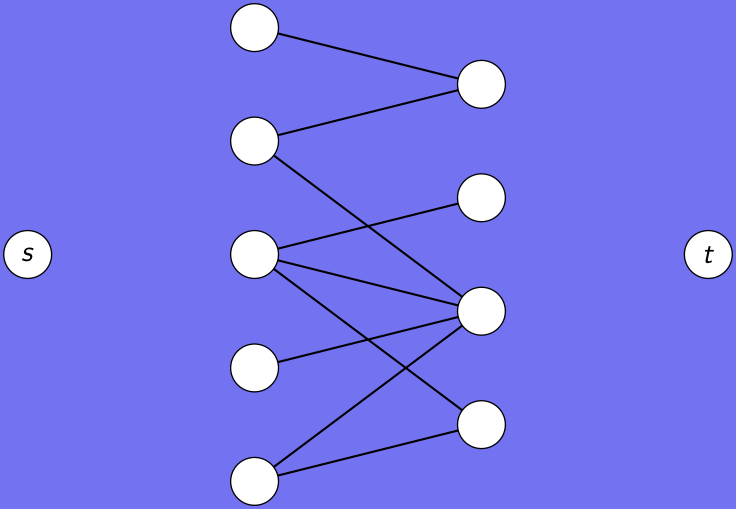
Problem (Maximum Bipartite Matching Problem)

Given a bipartite graph, determine a maximum matching.

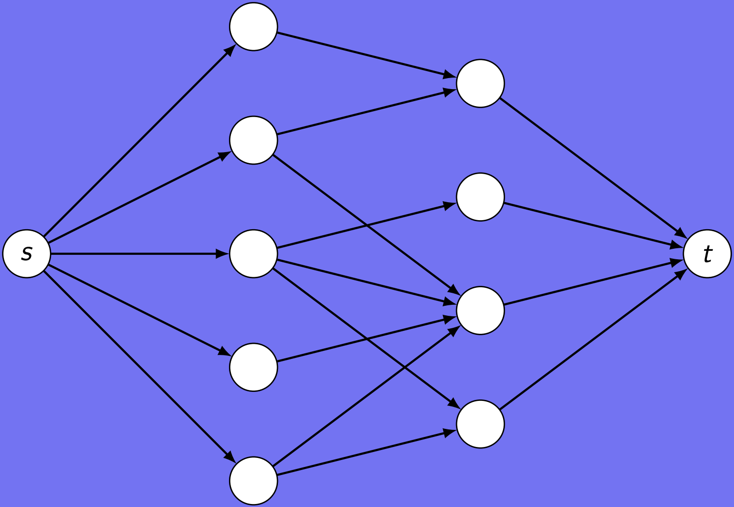
Example



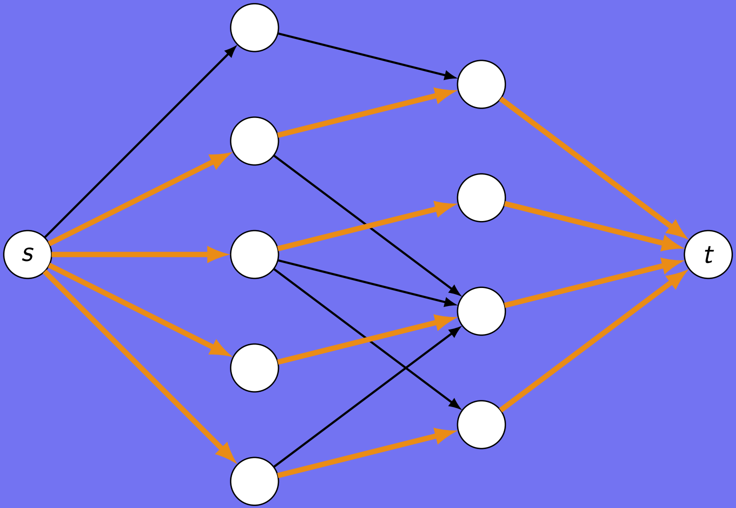
Example



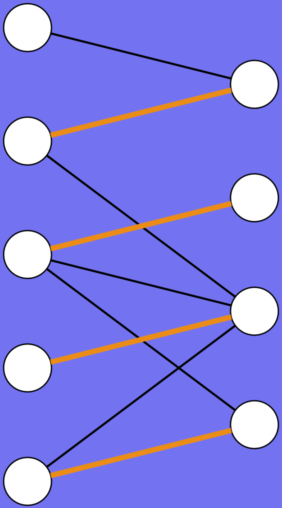
Example



Example



Example



Formalizing the transformation

To be able to apply Ford-Fulkerson method we formalize the transformation of a bipartite graph $G = \langle V, E\rangle$ into a network flow.

First define $G' = \langle V', E'\rangle$, with $V' = V \cup \{s, t\}$, s and t being the source and the sink, respectively. Since G is a bipartite graph V can be partitioned into two sets L and R, and

$$E' = \big\{(s,u) \ / \ u \in L\big\} \cup \big\{(u,v) \ / \ (u,v) \in V\big\} \cup \big\{(v,t) \ / \ v \in R\big\} .$$

It then suffices to assign a unit capacity to each edge in E'.

At this stage an important step, that should not be omitted, is to check the cost of the transformation.

Since each vertex in V has at least one incident edge, $|E|$ is greater or equal to $|V|/2$. Thus the number of edges in G is smaller than the one in G' and $\left|E'\right| = |E| + |V| \leq 3|E|$. Hence $\left|E'\right| = \Theta(|E|)$, meaning that the transformation can be efficiently performed.

Matching and flow

Lemma

Let $G = \langle V, E\rangle$ be a bipartite graph with vertex partition $V = L \cup R$, and $G' = \langle V', E'\rangle$ be its corresponding flow network. There is a matching M in G if and only if there exists a flow f in G'. In particular the value of the flow $|f|$ is equal to the cardinality of the matching $|M|$.

Proof. Let M be a matching in G. For an edge $e = (u, v)$ in E', define a flow f as $f(s, u) = f(u, v) = f(v, t) = 1$ if e is in M and 0 otherwise. Clearly f satisfies both the capacity constraint and the flow conservation properties (4.172).

Moreover as G is a bipartite graph a simple cut can be defined as $(S, T) = (L \cup \{s\}, R \cup \{t\})$. Observing that the net flow across the cut (S, T) is equal to $|M|$, we apply lemma 4.184 and get $|f| = |M|$.

Matching and flow

To prove the converse define a flow f on G' and M to be

$$M = \left\{(u, v) \;/\; u \in L,\; v \in R,\; \text{and } f(u, v) > 0\right\}.$$

By construction each vertex u in L has a single entering edge (s, u) with capacity at most 1. Therefore by the flow conservation property no more than 1 unit can leave on at most one edge. Thus a unit can enter u if and only if there is at most one v in R such that (u, v) is in M. Hence M is a matching.

Consequently for any matched vertex u in L, $f(s, u) = 1$, while $f(u, v) = 0$ for any edge in $E \setminus M$. This means that the net flow across cut the $(L \cup \{s\}, R \cup \{t\})$ is equal to $|M|$, which is exactly $|f|$, by lemma 4.184.

□

Matching and maximum flow

Remark. Over this whole chapter the net flow was implicitly assumed to only take integral values. However in the previous proof (4.211) it was explicitly mentioned that a unit flow cannot be split, which is true only in the case of integer valued flows.

In particular if the capacity function only takes integral values then the Ford-Fulkerson method returns an integer valued flow.

Theorem

Let $G = \langle V, E \rangle$ be a bipartite graph. The cardinality of a maximum matching M is equal to the value of the maximum flow in the flow network G' corresponding to G. This maximum matching is determined in time $\mathcal{O}(|V||E|)$.

Matching and maximum flow

Proof. Suppose M is a maximum matching in G while the corresponding flow in G' is not maximum. Then we can find a flow f' such that $|f'| > |f|$. But as noted in remark 4.213, both f and f' take integral values and there exists a matching M' corresponding to f'. Therefore we get

$$|M| = |f| < |f'| = |M'| . \lightning$$

Similarly if f is a maximum flow in G', then M is a maximum matching in G.

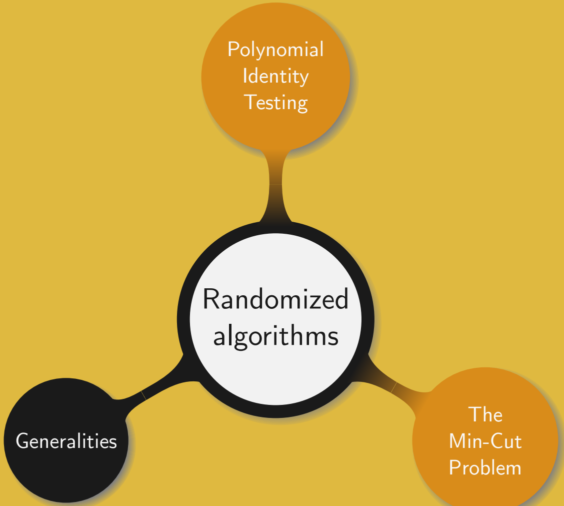
As any matching in a bipartite graph has cardinality at most $\min(|L|, |R|) = \mathcal{O}(|V|)$ the value of the maximum flow in G' is $\mathcal{O}(|V|)$. Therefore by 4.191 and 4.210 a maximum matching can be found in time $\mathcal{O}(|V| |E'|) = \mathcal{O}(|V| |E|)$. □

Key points

- Explain the maximum flow problem
- How fast is the Ford-Fulkerson algorithm?
- How is breadth first search working?
- How to solve the maximum matching problem in a bipartite graph?

5. Randomized algorithms

Chapter organisation



So far...

Along chapter 2, section 3 (2.102) we studied:

- How to model computers
- What an algorithm is
- Various classes of problems

Computation on a deterministic Turing machine depends on:

- The state of the machine
- The symbol read on the tape
- In any situation at most one action is performed

Non-deterministic Turing machine

Computation on a non-deterministic Turing machine:

- The machine has a state and a symbol is read on the tape
- The machine branches into many copies
- The machine transitions into one of the copies

When the machine is run more than once:

- Different paths are chosen
- Computation cannot be exactly reproduced

Remark. A non-deterministic Turing machine can be simulated by a deterministic Turing machine with three tapes.

Deterministic vs. probabilistic

Two main types of algorithms:

- Deterministic: a fixed sequence of steps if followed and the output is completely determined by the input
- Probabilistic: add some randomness to the process
  - Monte Carlo algorithm: returns either `True` or unknown; increasing the number of test decreases the probability of having a "false positive"
  - Las Vegas algorithms: always returns a correct result; the running time is however random; the time complexity cannot be precisely evaluated

Example

Quick sort:

- Worst case: $\mathcal{O}(n^2)$ vs. average case $\mathcal{O}(n \log n)$
- Fixed pivot: if the list to sort is originally structured worst complexity is likely to apply
- Random pivot: even if the list to sort is originally structured choosing a random pivot is equivalent to sorting with a fixed pivot on a randomly ordered list
- Running time: random depending on the choice of the pivot

Example

Primality testing:

- AKS: deterministic in time $\tilde{\mathcal{O}}(n \log^{12} n)$
- Miller-Rabin:
  - Complexity: $\tilde{\mathcal{O}}(k \log^2 n)$, with k the number of witnesses
  - Returns composite or unknown
  - Unknown means prime with probability $1 - 4^{-k}$

Formalization

Definitions

1. A non-deterministic Turing machine which chooses a random transitions according to some probability distribution is called a *probabilistic Turing machine*.

2. A language L is in the Bounded-error Probabilistic Polynomial time complexity class (BPP) if and only if there is a probabilistic Turing machine M such that
   - For any input, M runs in polynomial time;
   - For all x in L, M returns 1 with probability larger than $1/2+\varepsilon$, with $\varepsilon > 0$;
   - For all x not in L, M returns 1 with probability less than $1/2-\varepsilon$, with $\varepsilon > 0$;

Remark. In practice instead of taking $1/2+\varepsilon$, $2/3$ and $1/3$ are often chosen.

Probabilistic or deterministic

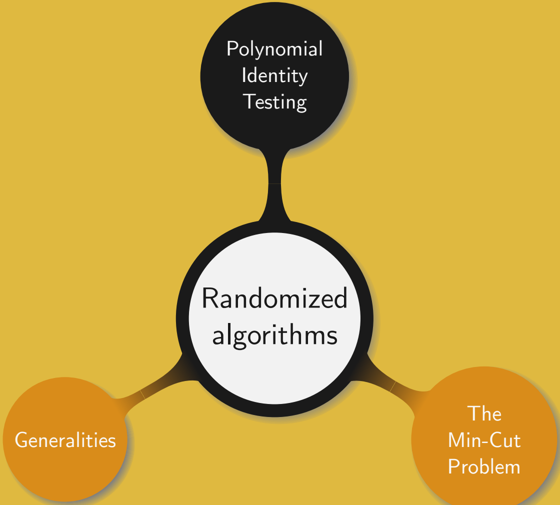
Reasons for using probabilistic algorithms:

- No polynomial time deterministic algorithm is known
- Available polynomial time deterministic algorithms are slower

Open questions:

- Is $\mathcal{P}$ =BPP?
- Are probabilistic algorithm more powerful than deterministic onces?
- What is the relative power of probabilistic and deterministic computations?

Chapter organisation



Polynomial Identity Testing

Randomized algorithms

Generalities

The Min-Cut Problem

The polynomial identity testing problem

Problem (Polynomial Identity Testing (PIT))

Let P be a multivariate polynomial over some field. Decide whether P is identically zero.

Remark. Identically zero means that after expanding P it reduces to zero. In particular this is a different problem from the Evaluate to Zero Everywhere (EZE) problem where one want to decide if an n-multivariate polynomial evaluates to zero for all x_i, $1 \leq i \leq n$.

Univariate case

The univariate case can be easily solved:

- The polynomial is given as a sum of monomials:
    - Reduce the sum
    - Check all the monomials and return `True` if they all equal zero
- The polynomial P of degree d is in a more complex form:
    - Arbitrarily choose $d + 1$ points (e.g. $0, \cdots, d$)
    - Evaluate P at those $d + 1$ points
    - Conclude that $P = 0$ if and only if they all evaluate to zero

What is the issue with multivariate polynomials?

Schwartz-Zippel Lemma

Definitions

1. A *monomial* is an expression of the form $\alpha \prod_{i=1}^{n} x_i^{\beta_i}$, where α is an element from a base field, the x_i are n variables, and the β_i are positive integers.
2. The *total degree of a monomial* is $\sum_i \beta_i$.
3. The *total degree of a polynomial* is the largest total degree among all the monomials composing the polynomial.

Lemma (Schwartz-Zippel)

Let P be a n-multivariate polynomial of total degree d, that is not identically zero, over a field $\mathbb{F}$. For $y_1, \cdots, y_n$, chosen uniformly and independently from a finite set $S \subset \mathbb{F}$,

$$\Pr\left[P(y_1, \cdots, y_n) = 0\right] \leq \frac{d}{|S|}.$$

Schwartz-Zippel Lemma

Proof. We proceed by induction on the number of variables n.

Base case: For $n = 1$ the result is clear as a univariate polynomial of degree d has at most d roots.

Induction step: we assume that the result is true for an $(n-1)$-multivariate polynomial and prove it is also true in the case of n variables.
Let k be the largest power of X_1 in any monomial composing P. Then

$$P(X_1, \cdots, X_n) = \sum_{i=0}^{k} X_1^i Q_i(X_2, \cdots, X_n).$$

By construction, Q_k is not identically zero, its total degree is at most $d-k$, and it has $n-1$ variables. Therefore by the induction hypothesis we get

$$\Pr\left[Q_k(y_2, \cdots, y_n) = 0\right] \leq \frac{d-k}{|S|}.$$

Schwartz-Zippel Lemma

For $y_2, \cdots, y_n$ in $\mathbb{F}$, we call $\mathcal{E}_1$ the event $Q_k(y_2, .., y_n) = 0$. Selecting $y_2, \cdots, y_n$ such that $\mathcal{E}_1$ does not occur, we define $R(X_1)$ to be the polynomial

$$R(X_1) = \sum_{i=0}^{k} X_1^i Q_i(y_2, \cdots, y_n) = P(X_1, y_2, \cdots, y_n).$$

Clearly $R(X_1)$ is not identically zero since $\mathcal{E}_1$ did not occur, meaning that X_1^k has a non zero coefficient. Therefore

$$\Pr\left[R(y_1) = 0 \mid \neg\mathcal{E}_1\right] \leq \frac{k}{|S|}.$$

Let $\mathcal{E}_2$ be the event $R(y_1) = 0$, which can also be stated as $P(y_1, \cdots, y_n) = 0$. In order to prove the lemma it remains to bound $\Pr[\mathcal{E}_2]$.

Schwartz-Zippel Lemma

As we have already bounded $\Pr[\mathcal{E}_2 \mid \neg\mathcal{E}_1]$ and $\Pr[\mathcal{E}_1]$ we can rewrite $\Pr[\mathcal{E}_2]$ as

$$\begin{aligned}\Pr[\mathcal{E}_2] &= \Pr[\mathcal{E}_2 \wedge \mathcal{E}_1] + \Pr[\mathcal{E}_2 \wedge \neg\mathcal{E}_1] \\ &= \Pr[\mathcal{E}_2 \wedge \mathcal{E}_1] + \Pr[\mathcal{E}_2 \mid \neg\mathcal{E}_1]\Pr[\neg\mathcal{E}_1] \\ &\leq \Pr[\mathcal{E}_1] + \Pr[\mathcal{E}_2 \mid \neg\mathcal{E}_1] \\ &\leq \frac{d-k}{|S|} + \frac{k}{|S|} = \frac{d}{|S|}\end{aligned}$$

Hence, by the induction principle Schwartz-Zippel lemma holds. □

Remark. Schwartz-Zippel lemma (5.228) says that when evaluating P at a random point there is a very low probability of finding a root. It however does not mean that a polynomial over $\mathbb{R}$ has finitely many roots. For instance $P(X_1, X_2) = X_1$ has infinitely many roots.

Solving PIT

Algorithm.

```
Input  : $P(X_1, \cdots, X_n)$ of degree $d$ and a field $\mathbb{F}$, $l$ and $k$
Output: not zero or probably zero
Select a subset $S \subset \mathbb{F}$ of size $\geq ld$;
for $i \leftarrow 1$ to $k$ do
    $(y_1, \cdots, y_n) \leftarrow \text{rand}(S)$;
    if $P(y_1, \cdots, y_n) \neq 0$ then return not zero;
end for
return probably zero;
```

Theorem

Let $P(X_1, \cdots, X_n)$ be a polynomial. If $P(X_1, \cdots, X_n)$ is not identically zero, then algorithm 5.232 returns the correct result with probability at least $1 - 1/l^k$.

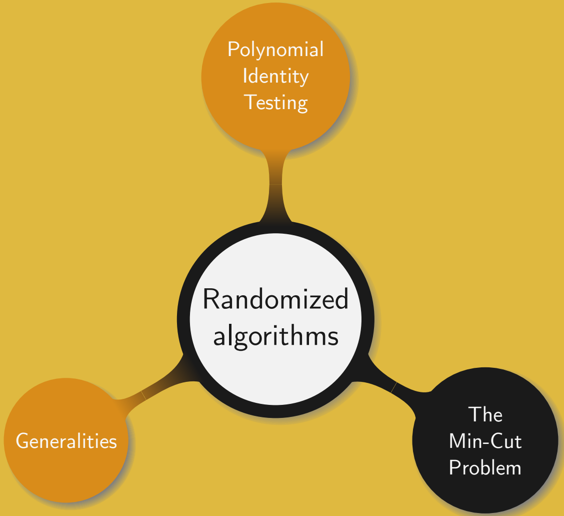
Solving PIT

Proof. For each iteration of the loop the probability of testing a root is at most $1/l$. Each choice of element in S is independent of the previous ones, such that the probability of passing all the tests while $P(X_1, \cdots, X_n)$ is not zero is at most $1/l^k$.

Hence the probability of returning "probably zero" while $P(X_1, \cdots, X_n)$ is not identically zero is at least $1 - 1/l^k$. □

Remark. If the field $\mathbb{F}$ has less elements than the number of roots of $P(X_1, .., X_n)$ then Schwartz-Zippel lemma (5.228) is of no use. It is however possible to overcome this problem by applying algorithm 5.232 to the polynomial $P(X_1, .., X_n)$ over an extension field of $\mathbb{F}$.

Chapter organisation



The Min-Cut Problem

Problem (Min-Cut)

Given a connected multi-graph G, determine the minimum number of edges that must be removed such that G becomes disconnected.

Common applications:

- Split a problem for parallel programming
- Optimize a divide and conquer strategy
- Segment an image into regions of similar color/texture

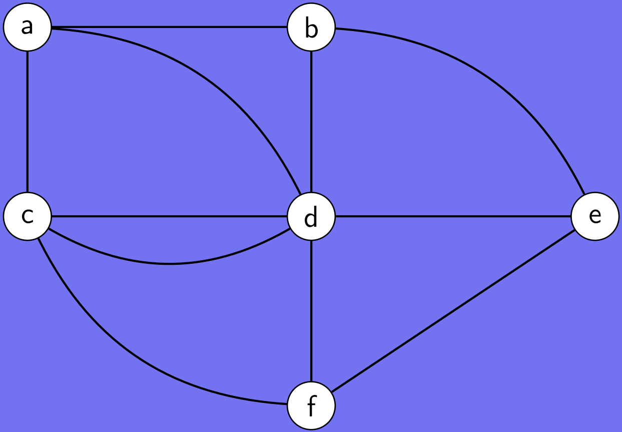
Karger's min-cut algorithm

Algorithm. (*Karger*)

```
Input  : a graph $G = \langle V, E \rangle$
Output : an upper bound on the min-cut
Function Contract($G$, $e$):
    remove all edges between $e.s$ and $e.t$;
    foreach edge $e'$ containing $e.s$ do
        disconnect $e'$ from $e.s$ and reconnect it to $e.t$;
    end foreach
    $S_{e.t} \leftarrow S_{e.t} \cup S_{e.s}$; remove vertex $e.s$ from $V$;
    return $G$;
end
foreach $v \in G.V$ do $S_v = \{v\}$;
while $G$ has more than two vertices do
    $e \leftarrow$rand($G.E$); $G \leftarrow$ Contract($G$, $e$);          /* edge $e = (e.s, e.t)$ */
end while
return $|G.E|$;
```

Exercise

Apply Karger's algorithm to the following graph G and find two different final values for $|G.E|$.



Edge contraction

The process described in the function Contract of Karger's algorithm (5.236) is called *edge contraction*.

As observed in the previous exercise (5.237) running Karger's algorithm can lead to various values. The question is then to figure out how to retrieve the right one with high probability. In order to do this we first evaluate the cost of a contraction.

Lemma

Let $G = \langle V, E \rangle$ be a multigraph. A single contraction in Karger's algorithm takes $\mathcal{O}(|V|)$ time and the whole algorithm $\mathcal{O}(|V|^2)$.

Proof. For an edge $e = (s, t)$, a contraction reattaches all edges incident on s to t. In term of adjacency matrix we replace row t with the sum of rows s and t and column t with the sum of columns s and t. Then both row s and column s are cleared up. This can be performed in $\mathcal{O}(|V|)$ time, meaning Karger's algorithm has complexity $\mathcal{O}(|V|^2)$. □

Probability of finding a min-cut

Lemma

Let $G = \langle V, E\rangle$ be a multigraph, and (S, T) be a min-cut. Then

$$\Pr\left[\text{Karger's algorithm ends with } (S, T)\right] \geq \frac{1}{\binom{|V|}{2}}.$$

Proof. First observe that if an edge (u, v) is contracted then only the cuts containing both u and v are unchanged. Therefore for Karger's algorithm to succeed the minimum cut (S, T) must remain untouched over all the random edge selections. Denoting the number of vertices by n, $n - 2$ contractions are to be performed in order to have only two vertices left. Naming those edges $\{e_1, \cdots, e_{n-2}\}$, the goal is to determine the probability to chose a proper edge at each iteration of the algorithm.

Probability of finding a min-cut

Let k denote the size of a minimum cut in G. Then the minimum degree of any vertex in G is at least k, otherwise a cut of size less than k could be exhibited by disconnecting a vertex of degree less than k. Therefore G has at least $nk/2$ edges.

After a contraction the new graph has one less vertex but the degree of all the vertices remains at least k. So after i steps there are $n - i$ vertices and at least $(n - i)k/2$ edges.

Since any vertex has degree at least k the probability of selecting a "bad edge", assuming none has been chosen before, is $2/(n - i)$.
Hence we can determine the probability of never choosing a "bad edge" during the whole process and thus end with (S, T).

Probability of finding a min-cut

This probability is given by

$$\begin{aligned}
\Pr\left[\text{find }(S,T)\right] &= \Pr\left[e_1,\cdots,e_{n-2}\notin(S,T)\right]\\
&= \Pr\left[e_1\notin(S,T)\right]\prod_{i=1}^{n-3}\Pr\left[e_{i+1}\notin(S,T)\mid e_1,\cdots,e_i\notin(S,T)\right]\\
&\geq \prod_{i=0}^{n-3}\left(1-\frac{2}{n-i}\right)\\
&= \frac{2}{n(n-1)}=\frac{1}{\binom{n}{2}}.
\end{aligned}$$

□

Probability of finding a min-cut

Since the probability of finding a min-cut is least $1/\binom{n}{2}$ it suffices to run the algorithm $l\binom{n}{2}$, for some value l. The probability of a run to succeed is then at least

$$1 - \left(1 - \frac{1}{\binom{n}{2}}\right)^{l\binom{n}{2}} \geq 1 - e^{-l}.$$

Therefore an appropriate choice for l is $c \ln n$, which leads to an error probability of at most $1/n^c$. Hence the total running time of Karger's algorithm is $\mathcal{O}(n^4 \log n)$.

Although this approach works well, it remains slow. The main reason is the random choice of the edge: at the beginning the multigraph features many edges and the probability of selecting an edge from a minimum cut is low. However as the process advances the probability of contracting an edge in the minimum cut grows.

Karger-Stein's algorithm

Algorithm. (*Karger-Stein*)

Input : a graph $G = \langle V, E \rangle$

Output : a mini-cut in G

```
Function FastCut($G$):
    if $|G.V| > 6$ then
        $G_1 \leftarrow G$; $G_2 \leftarrow G$;
        $t \leftarrow 1 + \frac{|G.V|}{\sqrt{2}}$;
        while $|G_1.V| \geq t$ do $e \leftarrow$rand$(G_1.E)$; $G_1 \leftarrow$ Contract($G_1, e$);
        while $|G_2.V| \geq t$ do $e \leftarrow$rand$(G_2.E)$; $G_2 \leftarrow$ Contract($G_2, e$);
        FastCut($G_1$); FastCut($G_2$);
    else
        find the min-cut by enumeration
    end if
    return $|G_1.E| \leq |G_2.E|$ ? $|G_1.E|$ : $|G_2.E|$
end
```

Karger-Stein's algorithm

Theorem

Given a multigraph with n vertices, Karger-Stein's algorithm discovers a minimum cut in time $\mathcal{O}(n^2 \log^3 n)$, with high probability.

Sketch of proof. First note that $6 \ll |V|$ and as such finding a minimum cut by enumeration only impacts the final complexity by a constant factor.

Given a cut, observe that the probability that it survives down to t vertices is at least $\binom{t}{2}/\binom{n}{2}$. Thus for $t = n/\sqrt{2}$ the probability of success is larger than $1/2$.

Since Karger-Stein's algorithm follows a divide and conquer strategy, its complexity can be expressed by a recurrence relation.

Karger-Stein's algorithm

Then recalling that a single edge contraction costs $\mathcal{O}(n^2)$ (lemma 5.238) we get

$$T(n) = 2\left(n^2 + T\left(\frac{n}{\sqrt{2}}\right)\right).$$

Hence by the master theorem (2.95) we conclude that the running time of Karger-Stein's algorithm is $\mathcal{O}(n^2 \log n)$.

We now consider the success probability. First as we start with n vertices and go down to $t = n/\sqrt{2}$, the success probability is $(t/n)^2 \approx 1/2$. Then at the next recursion level the graph shrinks from $n/\sqrt{2}$ to $n/2$ vertices, which means an overall success probability of about $1/4$.

Karger-Stein's algorithm

More generally assume the minimum cut to still be in the graph and let $P(t)$ be the probability that a call to the algorithm with t vertices successfully computes it. Then G_i, $1 \leq i \leq 2$ still contains it with probability larger than a half. Therefore the probability that a recursive call succeeds is $1/2P(t/\sqrt{2})$. And since two recursive calls are performed

$$P(t) = 1 - \left(1 - \frac{1}{2}P\left(\frac{t}{\sqrt{2}}\right)\right)^2 .$$

Solving this recurrence relation yields $P(n) = \Omega(1/\log n)$.[1] This means that Karger-Stein's algorithm needs to be run about $\log^2 n$ times in order to have an error probability of at most $\mathcal{O}(1/n)$ of preserving the minimum cut. This gives a final complexity of $\mathcal{O}(n^2 \log^3 n)$. □

[1] This result is proven in the homework.

Final remarks

Randomized algorithms:

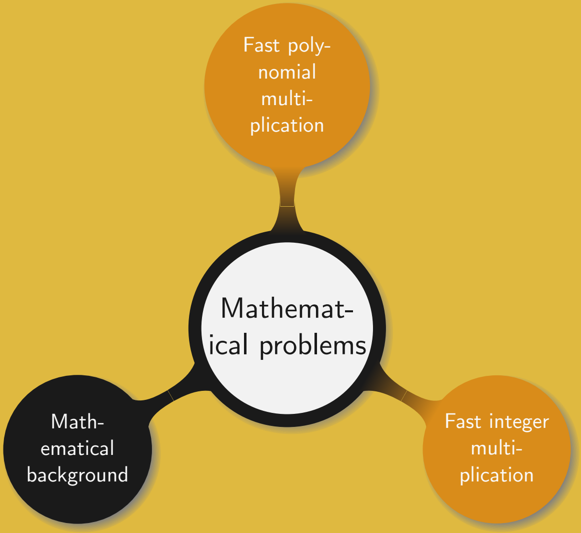
- Bring much flexibility compared to deterministic ones
- Are often faster that deterministic ones
- Introduce imprecision on the output or on the complexity
- Require a good knowledge of probability theory
- Have proof that are often complex, even if the algorithm can be simply expressed

Key points

- What is a randomized algorithm?
- What are the two main types of randomized algorithm?
- How to solve the PIT problem?
- Why using a randomized algorithm for the Min-Cut problem?

6. Mathematical problems

Chapter organisation



Goal

Many applications require large numbers to be multiplied.

Common multiplication algorithms:

- Simple strategy: $\mathcal{O}(n^2)$
- Karatsuba: $\mathcal{O}(n^{\log_2 3})$

Fast Fourier Transform (FFT):

- Fast polynomial and number multiplication
- One of the most important and used algorithms

Composition laws

Definitions

Let S and S' be two sets.

1. An *internal composition law* $(\circ)$ is an map from $S \times S$ into S such that
$$\begin{aligned} S \times S &\longrightarrow S \\ (x, y) &\longmapsto x \circ y. \end{aligned}$$

2. An *external composition law* $(*)$ is an map from $S' \times S$ into S such that
$$\begin{aligned} S' \times S &\longrightarrow S \\ (\alpha, x) &\longmapsto \alpha * x. \end{aligned}$$

Example. For a set S, the intersection $(\bigcap)$ and union $(\bigcup)$ define two internal composition laws for the class of subsets of S.

Definition (Group)

A *group* is a pair $(G, \circ)$ consisting of a set G and an internal composition law that verifies the following properties:

i *Associativity:* $a \circ (b \circ c) = (a \circ b) \circ c$ for all $a, b, c \in G$

ii *Existence of a unit element:* there exists an element $e \in G$ such that $a \circ e = e \circ a = a$ for all $a \in G$

iii *Existence of inverse:* for every $a \in G$ there exists an element $a^{-1} \in G$ such that $a \circ a^{-1} = a^{-1} \circ a = e$

A group is called *abelian* if in addition to the above properties

iv *Commutativity:* $a \circ b = b \circ a$ for all $a, b \in G$.

Rings

Definition (Ring)

A *ring* is a triple $(R, +, \cdot)$ consisting of a set R and two internal composition laws $(+)$ and $(\cdot)$, such that

i. $(R, +)$ is an abelian group

ii. *Multiplicative unit:* there exists an element $1 \in G$ such that

$$a \cdot 1 = 1 \cdot a = a \quad \text{for all } a \in R$$

iii. *Associativity:* for any $a, b, c \in R$,

$$a \cdot (b \cdot c) = (a \cdot b) \cdot c$$

iv. *Distributivity:* for any $a, b, c \in R$,

$$a \cdot (b + c) = (a \cdot b) + (a \cdot c), \quad (b + c) \cdot a = (b \cdot a) + (c \cdot a)$$

A ring is called *commutative* if in addition to the above properties

v. *Commutativity:* $a \cdot b = b \cdot a$ for all $a, b \in R$

Fields

Definition (Field)

Let $(F, +, \cdot)$ be a commutative ring with unit element of addition 0 and unit element of multiplication 1. Then F is a *field* if

i $0 \neq 1$

ii For every $a \in F \setminus \{0\}$ there exists an element a^{-1} such that

$$a \cdot a^{-1} = 1.$$

Remark. Another way of writing this definition is to say that $(F, +, \cdot)$ is a field if $(F, +)$ and $(F \setminus \{0\}, \cdot)$ are abelian groups, $0 \neq 1$, and $\cdot$ distributes over $+$.

Mathematical structures

Example. Let n be an integer, and $\mathbb{Z}/n\mathbb{Z}$ be the set of the integers modulo n

- $(\mathbb{Z}/n\mathbb{Z}, +)$ also denoted $(\mathbb{Z}_n, +)$ is a group
- $(\mathbb{Z}/n\mathbb{Z}, +, \cdot)$ is a ring
- If n is prime then $(\mathbb{Z}/n\mathbb{Z}, +, \cdot)$ is the field $\mathbb{F}_n$
- The invertible elements of $\mathbb{Z}/n\mathbb{Z}$, with respect to '$\cdot$', form a group denoted $\mathrm{U}(\mathbb{Z}/n\mathbb{Z})$ or sometimes $\mathbb{Z}_n^\times$ or $\mathbb{Z}_n^*$
- $(\mathbb{Z}/n\mathbb{Z}[X], +, \cdot)$ is the ring of the polynomials over $\mathbb{Z}/n\mathbb{Z}$
- If n is prime and the polynomial $P(X)$ is irreducible then
  $$(\mathbb{F}_n[X]/\langle P(X)\rangle, +, \cdot)$$
  is a field; this is $\mathbb{F}_{n^{\deg P(x)}}$

Roots of unity

Definitions

Let R be a ring and n be a strictly positive integer.

1. An element a of R is a *zero divisor* if there exists $b \in R \setminus \{0\}$ such that $ab = 0$.
2. If 0 is the only zero divisor in R, then R is an *integral domain*.
3. An element $\omega \in R$ is an *nth root of unity* if $\omega^n = 1$.
4. An element $\omega \in R$ is a *primitive nth root of unity* if $\omega^n = 1$ and for any $1 \leq k \leq n-1$, $\omega^k \neq 1$.

Example.

- In $\mathbb{C}$, $e^{2i\pi/8}$ is a primitive 8th root of unity;
- In $\mathbb{Z}_{17}$, 2 is not a primitive 16th root of unity;

A first result

Lemma

Let R be a ring, l, n be two integers such that $1 < l < n$, and ω be a primitive nth root of unity. Then (i) $\omega^l - 1$ is not a zero divisor in R, and (ii) $\sum_{j=0}^{n-1} \omega^{lj} = 0$.

Proof. First, note that for any $c \in R$ and m in $\mathbb{N}$

$$c^m - 1 = (c-1)(1 + c + c^2 + \cdots + c^{m-1}). \qquad (6.1)$$

(i) Let d be the $\gcd(l, n)$. From the Extended Euclidean Algorithm we know the existence of $s, t \in \mathbb{Z}$ such that $sl + tn = d$. Recalling that $l < n$, we have $1 \leq d < n$, and we can cancel a prime factor r of n such that $d | (n/r)$.

For $c = \omega^d$ and $m = \frac{n}{rd}$ in (6.1) we get

$$\omega^{n/r} - 1 = (\omega^d - 1)(1 + \omega^d + \cdots + \omega^{d\left(\frac{n}{rd} - 1\right)}).$$

A first result

Hence if there exists $b \in R$ such that $b(\omega^d - 1) = 0$, then $b(\omega^{n/r} - 1)$ must also be zero. But as $\omega^{n/r} - 1$ is not a zero divisor, $b = 0$, and neither is $\omega^d - 1$ a zero divisor.

We now set $c = \omega^l$ and $m = s$ in (6.1). In that case we can write $\omega^{sl} = \omega^{sl}\omega^{tn} - 1 = \omega^d$, and we see that $\omega^l - 1$ divides $\omega^d - 1$. Following a similar reasoning as in the previous case we obtain that $\omega^l - 1$ is a not a zero divisor.

(ii) Using (6.1) one more time, for $c = \omega^l$ and $m = n$ we obtain

$$\omega^{ln} - 1 = (\omega^l - 1)(1 + \omega^l + \cdots + \omega^{l(n-1)}).$$

As $\omega^{ln} = 1$, and $\omega^l - 1$ is not a zero divisor, $\sum_{j=0}^{n-1} \omega^{lj} = 0$. □

Discrete Fourier Transform

Definition

Let R be a ring, and $\omega \in R$ be a primitive nth root of unity. We denote a polynomial $P(X)$ of degree less than n in $R[X]$ by its coefficients

$$P(X) = \sum_{i=0}^{n-1} a_i X^i = (a_0, \cdots, a_{n-1}).$$

The linear map

$$\begin{aligned} \mathrm{DFT}_\omega : R^n &\longrightarrow R^n \\ (a_0, \cdots, a_{n-1}) &\longmapsto (P(1), P(\omega), \cdots, P(\omega^{n-1})) \end{aligned}$$

evaluates P at the powers of ω and is called *Discrete Fourier Transform* (DFT).

Discrete Fourier Transform

As DFT_ω is a linear map it is expressed as a matrix transformation

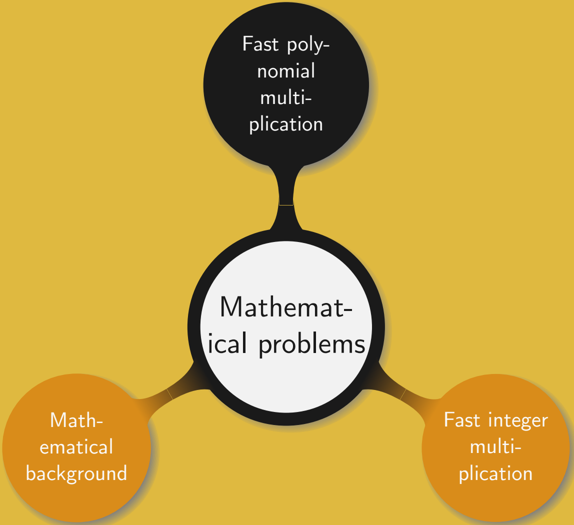
$$\begin{pmatrix} P(1) \\ P(\omega) \\ P(\omega^2) \\ \vdots \\ P(\omega^{n-1}) \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)^2} \end{pmatrix}}_{V_\omega} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix}.$$

Note that if ω is a primitive nth root of unity then so is ω^{-1}. Otherwise there would be a $1 \leq k < n$ such that $(\omega^{-1})^k$ is 1. And as $\omega^n = 1$ we would have $1 = \omega^{n+k}$ and $\omega^k = 1$ ↯.

Then using lemma 6.258 observe that $V_\omega V_{\omega^{-1}} = n\,\mathrm{I}_n$, where I_n is the identity matrix of size $n \times n$. Thus the inverse of DFT_ω is

$$\mathrm{DFT}_\omega^{-1} = \frac{1}{n}\,\mathrm{DFT}_{\omega^{-1}}\,.$$

Chapter organisation



Representing polynomials

Two main cases depending on the structure of the polynomial:

- Dense: use an array where the coefficient of each monomial is stored at index "degree of the monomial"
- Sparse: use a structure composed of two arrays storing the degrees and the corresponding coefficients, respectively

Alternative strategy: over an integral domain a polynomial of degree strictly less than n can be represented using its value at n distinct points

Evaluating polynomials

Let $P(X) = \sum_{i=0}^{n} a_i X^i$ be a polynomial of degree n. Evaluating P costs $\mathcal{O}(n^2)$ if naively computed. However note that $P(X)$ can be rewritten

$$P(X) = a_0 + X(a_1 + X(a_2 + \cdots + X(a_{n-1} + Xa_n))).$$

This remark dramatically decreases the complexity as it drops to $\mathcal{O}(n)$, and yields the following simple algorithm.

Algorithm. (*Horner*)

Input : a polynomial P and x the value to evaluate P at
Output: Px the evaluation of P at x

```
Function Horner($P$, $x$):
    $Px \leftarrow 0$;
    for $i \leftarrow \deg P$ to 0 do  $Px \leftarrow Px \cdot x +$ coeff[$i$] ;
    return $Px$;
end
```

Multiplying polynomials

Two main cases depending on the polynomial representation:

- Dense: usual approach, multiply the various coefficients together; complexity $\mathcal{O}(n^2)$
- Evaluation: evaluate the polynomials in n points, multiply them two by two as
$$PQ(x_i) = P(x_i)Q(x_i).$$
Complexity is $\Omega(n^2)$ since n evaluations are necessary, to which have to be added the cost of the multiplications and of the interpolation to get the final polynomial (usually $\mathcal{O}(n^2)$).

Fast Fourier Transform

Looking back at DFT_ω, it can be viewed as a special multipoint evaluation of a polynomial in the powers $1, \omega, \cdots, \omega^{n-1}$ of a primitive nth root of unity ω. Then its inverse DFT_ω^{-1}, which given n evaluations of a polynomial allows to recover its coefficients, is just the interpolation at the powers of ω.

From the previous discussion on the DFT (6.261), it is clear that knowing how to compute it efficiently means being able to also compute its inverse efficiently.

For the sake of simplicity assume $n = 2^k$, $k \in \mathbb{N}$, and observe that

$$\begin{aligned} P(X) &= \sum_{i=0}^{n-1} a_i X^i \\ &= (a_0 + a_2X^2 + \cdots + a_{n-2}X^{n-2}) + (a_1X + a_3X^3 + \cdots + a_{n-1}X^{n-1}) \\ &= P_1(X^2) + XP_2(X^2) \end{aligned} \tag{6.2}$$

with both P_1 and P_2 of degree less than $(n-2)/2 < n/2$.

Fast Fourier Transform

The structure of equation (6.2) suggests a “divide and conquer” approach in order to determine

$$P(\omega^i) = P_1(\omega^{2i}) + \omega^i P_2(\omega^{2i}), \quad 0 \leq i < n. \tag{6.3}$$

This formulation can be further rewritten by noticing that

$$\begin{aligned} 0 &= \omega^n - 1 \\ &= (\omega^{n/2} - 1)(\omega^{n/2} + 1). \end{aligned}$$

By lemma 6.258 none of the two factors is a zero divisor and as $\omega^{n/2} \neq 1$, ω being a primitive nth root of the unity, we conclude that $\omega^{n/2} = -1$. Hence, for all $0 \leq i < n/2$, $\omega^i = -\omega^{n/2+i}$, and (6.3) can be rewritten

$$\begin{aligned} P(\omega^i) &= P_1(\omega^{2i}) + \omega^i P_2(\omega^{2i}), \quad 0 \leq i < n/2, \\ P(\omega^{n/2+i}) &= P_1(\omega^{2i}) - \omega^i P_2(\omega^{2i}), \quad 0 \leq i < n/2. \end{aligned} \tag{6.4}$$

Fast Fourier Transform

Algorithm. (*Fast Fourier Transform (FFT)*)

Input : a polynomial P of degree $< n$, with n a power of 2, and ω a primitive nth root of unity

Output : $\mathrm{DFT}_\omega(P)$

```
Function FFT($P, \omega$):
    $n \leftarrow \deg P + 1$;
    if $n = 1$ then return $P$;
    $P_1 \leftarrow$ FFT($\sum_{j=0}^{n/2-1} a_{2j} X^{2j}, \omega^2$);
    $P_2 \leftarrow$ FFT($\sum_{j=0}^{n/2-1} a_{2j+1} \omega X^{2j}, \omega^2$);
    for $i \leftarrow 0$ to $n/2$ do
        $P_\omega[i] \leftarrow P_{1,\omega^2}[i] + \omega^i P_{2,\omega^2}[i]$;
        $P_\omega[n/2 + i] \leftarrow P_{1,\omega^2}[i] - \omega^i P_{2,\omega^2}[i]$;
    end for
    return $P_\omega$;
end
```

Theorem

Given a polynomial P over a commutative ring and ω a primitive nth root of unity, the FFT algorithm correctly computes $\mathrm{DFT}_\omega(P)$ in time $\mathcal{O}(n \log n)$.

Proof. The correctness clearly follows from the previous discussion, more particularly from the recurrence relation (6.4).

Let $T(n)$ be the time to compute a DFT. Then in relation (6.4) two smaller DFT are computed, plus $n/2$ multiplications and n additions, that is

$$\begin{aligned} T(n) &\leq 2T(n/2) + n/2 + n \\ &\leq 2T(n/2) + 3n/2. \end{aligned}$$

By the Master theorem (2.95) the time complexity of a DFT is $\mathcal{O}(n \log n)$.

□

Polynomial multiplication

Let R be a ring containing a primitive nth root of unity. Given two polynomials P and Q with degrees less than $n/2$, defined over $R[X]$, we want to determine $S = PQ$. Note that n is still taken to be a power of 2.

As both P and Q are of degrees less than n they can be efficiently evaluated using the FFT algorithm (6.268). Then n multiplications in R are enough to determine S, represented using its evaluation in n points.

Applying DFT_ω^{-1} to the n evaluations of S, is achieved through the calculation of $1/n\,\mathrm{DFT}_{\omega^{-1}}\,S$ (6.261). This computation returns the interpolation of S in n points (6.266), that is it determines the unique polynomial of degree less than n passing through the n points. Hence we obtain a fast strategy to compute the product of two polynomials.

Polynomial multiplication using FFT

Algorithm. (*Fast polynomial multiplication*)

Input : P and Q two polynomials of degree $< n/2$, with n a power of 2, a primitive nth root of unity ω

Output : $S = PQ$

```
Function FPMult($P, Q, \omega$):
    $P_\omega \leftarrow$ FFT($P, \omega$);
    $Q_\omega \leftarrow$ FFT($Q, \omega$);
    $S_\omega \leftarrow P_\omega Q_\omega$;
    $S \leftarrow \frac{1}{n}$FFT($S_\omega, \omega^{-1}$);
    return $S$
end
```

Definition

A commutative ring containing a primitive 2^kth root of unity for any k in $\mathbb{N}^*$ is said to *support the FFT*.

Polynomial multiplication using FFT

Theorem

Let R be a ring supporting the FFT and n be 2^k, with k in $\mathbb{N}^*$. Then for two polynomials P and Q in $R[X]$, with $\deg PQ < n$, the fast polynomial multiplication algorithm computes their product in time $\mathcal{O}(n \log n)$.

Proof. The correctness of the algorithm is clear when considering the previous discussion (6.270).

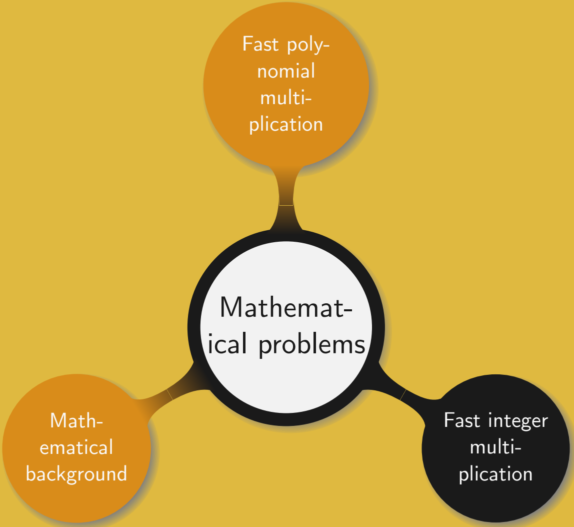
The algorithm computes three DFT, n component-wise products in R, as well as n multiplications by the inverse of n in R. Therefore the overall complexity is dominated by $\mathcal{O}(n \log n)$. □

Rings not supporting the FFT

In order to run the fast polynomial multiplication algorithm (6.271) the underlying ring R must support the FFT. In the case where R does not contain any primitive 2^kth root of unity a "virtual" one can be attached to the ring.

The Schönage-Strassen algorithm handles this special case at the cost of a slightly worse complexity. In fact their result states that over any ring the product of two polynomials of degree less than n can be computed in $\mathcal{O}(n \log n \log \log n)$ operations.

Chapter organisation



From polynomials to integers

Let a and b be two N-bit long integers. For the sake of simplicity we assume N to be of the form 2^{2^l} for some integer $l > 0$. Let $a_N, \cdots, a_0$ and $b_N, \cdots, b_0$ denote the binary representations of a and b, respectively. Since N was chosen to be a square it is easy to split a and b into blocks of size $\sqrt{N}$ and write them

$$a = \sum_{i=0}^{\sqrt{N}-1} A_i 2^{i\sqrt{N}}, \text{ and } b = \sum_{i=0}^{\sqrt{N}-1} B_i 2^{i\sqrt{N}}, \quad 0 \leq A_i, B_i \leq 2^{\sqrt{N}} - 1.$$

If we consider the polynomials $A(X) = \sum_{i=0}^{\sqrt{N}-1} A_i X^i$ and $B(X) = \sum_{i=0}^{\sqrt{N}-1} B_i X$ then the product AB evaluated at $2^{\sqrt{N}}$ is exactly the product ab. Therefore integer multiplication can be performed via polynomial multiplication: (i) write the two integers as polynomials, (ii) apply the fast polynomial multiplication on them, and (iii) finally evaluate the product at $2^{\sqrt{N}}$.

Attaching a new element

While the first step is simple to achieve the second one requires more technical considerations. In fact the two polynomials A and B are both of degree less than $2\sqrt{N}$ and as such R must contain a primitive $2\sqrt{N}$th root of unity.

As $\mathbb{Z}$ does not support FFT some extra work is needed in order to attach a new "virtual" element to the ring without altering the final result.

Consider the ring $\mathbb{Z}_{2^{\sqrt{N}}+1}$ and observe that 2 is a primitive $2\sqrt{N}$th root of unity. This is clear as $2^{\sqrt{N}}$ is -1 modulo $2^{\sqrt{N}}+1$. Thus $t = 2\sqrt{N}$ is the smallest power for which 2^t is 1.

An obvious idea is then to perform the computation in the ring $\mathbb{Z}_{2^{\sqrt{N}}+1}[X]$. However as both A and B can feature coefficients as large as $2^{\sqrt{N}} - 1$, the polynomial C resulting from their product can have coefficients up to $\sqrt{N}2^{2\sqrt{N}}$, which is larger than $2^{\sqrt{N}} + 1$.

Expanding the ring

As a result, if coefficients in C happen to be too large they will be reduced modulo $2^{\sqrt{N}}+1$, ruining the whole calculation. A simple solution consists in performing all the computation in a larger ring. At that stage two points must be taken into consideration: (i) the use of a large ring increases the computational cost and (ii) the ring must contain a primitive $2\sqrt{N}$th root of unity.

Note that for any $N \geq 1$, $2^{3\sqrt{N}} > \sqrt{N}2^{2\sqrt{N}}$, while 8 is a primitive $2\sqrt{N}$th root of unity in $\mathbb{Z}_{2^{3\sqrt{N}}+1}$. The latter being a consequence of 2 being a primitive $6\sqrt{N}$th root of unity.

Therefore it suffices to consider A and B as polynomials over the ring $\mathbb{Z}_{2^{3\sqrt{N}}+1}[X]$. Then applying the fast polynomial multiplication algorithm (6.271) and evaluating the product at $2^{\sqrt{N}}$ yields the result.

Fast integer multiplication

Algorithm. (*Fast integer multiplication*)

Input : two N bit integers a and b, a ring $R = \mathbb{Z}_{2^{3\sqrt{N}}+1}$, $\omega = 8$ a primitive $2\sqrt{N}$th root of unity

Output: $c = a \cdot b$

```
$A \leftarrow$ poly($a$);                      /* encode $a$ as a polynomial */
$B \leftarrow$ poly($b$);                      /* encode $b$ as a polynomial */
$C \leftarrow$ FPMult $(A, B, \omega)$;
$c \leftarrow$ Horner $(C, 2^{\sqrt{N}})$;
return c;
```

Fast integer multiplication

Theorem

Given two integers of length N in a ring R, the fast integer multiplication algorithm computes their product in $\mathcal{O}(\sqrt{N}\log\sqrt{N})$ arithmetic operations in R.

Proof. The correctness results from the previous discussion (6.275).

The pre-dominant computation in the algorithm is the fast polynomial multiplication of A and B, which by theorem 6.272 takes $\mathcal{O}(\sqrt{N}\log\sqrt{N})$. □

Remark. With a bit more work it is possible to determine the complexity in term of bit operations, instead of arithmetic operations. In that case the complexity becomes

$$\mathcal{O}(N\log^{2+\log_2 3-1} N).$$

Fast integer multiplication

Other interesting remarks:

- At the bit level, most operations can be performed using "shifts", incurring a linear cost in the length of the integers. This is, in particular the main reason for choosing ω to be a power of 2.
- The integer $2\sqrt{N}$ has inverse $2^{6\sqrt{N}-\log_2\sqrt{N}-1}$ in $\mathbb{Z}_{2^{3\sqrt{N}}+1}$. Observe that $2^{6\sqrt{N}} \equiv 1 \bmod 2^{3\sqrt{N}+1}$, and $2^{\log_2\sqrt{N}} = \sqrt{N}$.
- Although in the algorithm no coefficient reaches $2^{3\sqrt{N}}+1$ all the calculations are performed in the ring $\mathbb{Z}_{2^{3\sqrt{N}}+1}$ and the special case of adding two $3\sqrt{N}$ bits long elements has to be considered when defining addition.
- To date the asymptotically fastest integer multiplication algorithm, due to Fürer, takes $N \log N 2^{\mathcal{O}(\log^* N)}$ bit operations.

Multiplication complexity

Definition

The complexity of multiplying two polynomials of degree less than n is denoted $\mathsf{M}(n)$, while the complexity of multiplying two n-bit integer is denoted $\mathsf{M}_I(n)$.

For all n and m in $\mathbb{N}$,

$$\mathsf{M}(n+m) \geq \mathsf{M}(m) + \mathsf{M}(n) \text{ and } \mathsf{M}_I(n+m) \geq \mathsf{M}_I(n) + \mathsf{M}_I(m).$$

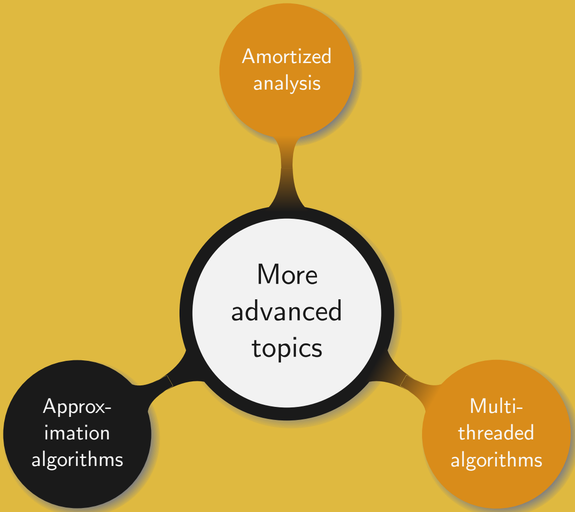
These simpler notations allows an easier complexity study of more advanced algorithms such as fast multi-point evaluation and fast interpolation.

Key points

- Define all the basic mathematical structures.
- What is the function DFT_ω?
- How to represent polynomials?
- Describe the basic idea behind the FFT.
- How to view integers if one wants to run fast multiplication?

7. More advanced topics

Chapter organisation



Solving hard problems

Given an $\mathcal{NP}$-complete problem:

- It is impossible to efficiently find a solution
- In practice it might have many applications

Hope for such problems:

- Inputs are always small
- Specific sub-cases arising in practice can be solved efficiently
- An almost optimal solution is sufficient and can be computed efficiently

Approximation algorithms

Definition

1. For a given problem P, an algorithm returning a near-optimal solution to P is called an *approximation algorithm*.

2. Let the cost of an optimal solution be C^* and the one of an approximation be C. If for any input of size n there exists an *approximation ratio* $\rho(n)$, such that

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n),$$

then the algorithm is said to be a $\rho(n)$*-approximation algorithm*.

Approximation algorithms

Remark.

- The approximation ratio can never be less than 1
- Approximation algorithms are expected to
  - Be polynomial time
  - Feature slowly growing approximation ratio as n increases
- Some approximation algorithms take an input parameter defining the precision of the approximation. The more precise the approximation, the longer the running time.

The vertex-cover problem

Problem (Optimal Vertex Cover)

Let $G = \langle V, E \rangle$ be an undirected graph. A *vertex cover* is a subset $V' \subseteq V$ such that if $(u, v) \in E$ then at least u or v is in V'. Find a vertex cover of minimum size.

Algorithm. (*Approx Vertex Cover*)

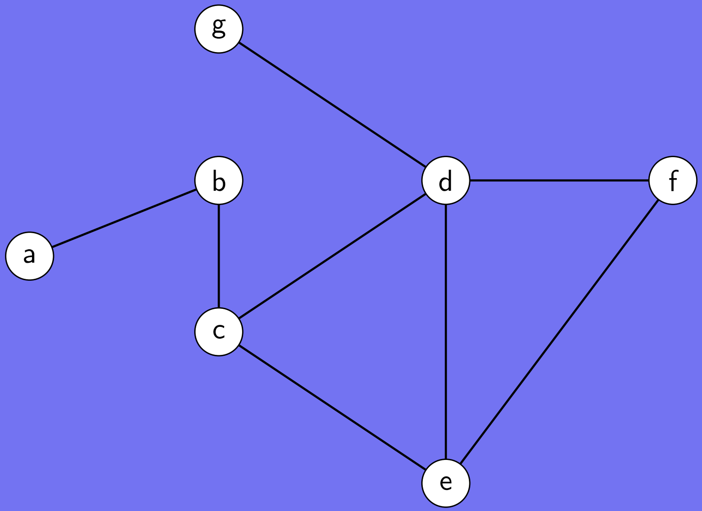
Input : $G = \langle V, E \rangle$ an undirected graph

Output : C a nearly optimal vertex cover

```
$C \leftarrow \emptyset$; $E' \leftarrow G.E$;
while $E' \neq \emptyset$ do
    $e \leftarrow$ an arbitrary edge of $E'$; $C \leftarrow C \cup \{e.u, e.v\}$;
    remove from $E'$ any edge incident to $u$ or $v$;
end while
return $C$;
```

Approximating vertex-cover

Exercise.



Approximating vertex-cover

Theorem

Approx Vertex Cover runs in time $\mathcal{O}(|V|+|E|)$ and is a 2-approximation algorithm.

Proof. Representing E' using an adjacency list leads to $\mathcal{O}(|V|+|E|)$.

First note that C is a vertex cover since the algorithm loops until all the edges from E have been covered by some vertex in C.

Let A denote the set of all the edges selected by the algorithm. Then any cover includes at least one endpoint, and by construction no two edges in A share a common endpoint. Thus there is no two edges in A that are covered by the same vertex in C^*, and we have the lower bound $|C^*| \geq |A|$.

Finally noting that $|C| = 2|A|$ yields $|C| \leq 2|C^*|$. □

Quality of the approximation

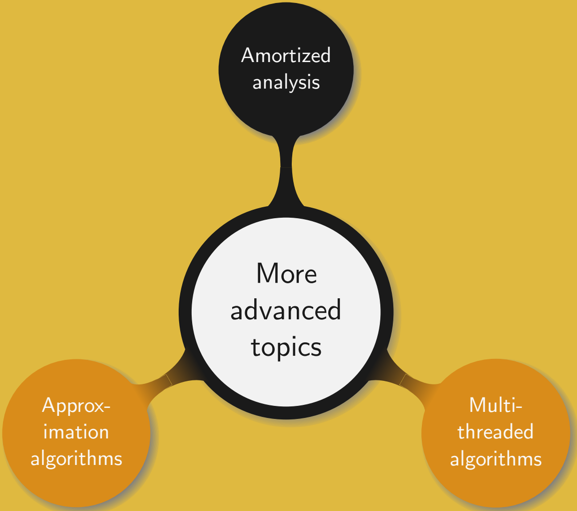
Common methodology for approximation algorithms:

- An optimal solution is unknown
  e.g. the size of the vertex cover is unknown

- Determine a lower bound on an optimal solution
  e.g. in approx vertex cover (7.288) the set of the selected edges forms a maximum matching (4.208), which provides a lower bound on the size of an optimal vertex cover

- Relate the size of the approximation to the lower bound on the optimal solution
  e.g. the approximation ratio is obtained by relating $|C|$ to $|A|$

Chapter organisation



More advanced topics

Amortized analysis

Approximation algorithms

Multithreaded algorithms

Motivating problem

A *dynamic table* is an array which automatically resizes as elements are added or removed. A common strategy consists in doubling the size of the table as soon as an overflow occurs. In such a context we want to determine the cost of n insertions.

We define the cost c_i of the ith operation to be i, if $i-1$ is a power of 2, and 1 otherwise. The cost C_n of n insertions is given by

$$\begin{aligned} C_n &= \sum_{i=1}^{n} c_i \\ &\leq n + \sum_{j=0}^{\lfloor \log(n-1) \rfloor} 2^j \\ &\leq 3n = \Theta(n) \end{aligned}$$

Hence the average cost for each operation is $\Theta(n)/n = \Theta(1)$.

Amortized analysis

Definition (Amortized analysis)

Given a sequence of operations, an *amortized analysis* is a strategy allowing to show that the average cost per operation is small although single operations in the sequence might be expensive.

Remark. Amortize analysis:

- Does not uses probabilities
- Evaluates the average performance of each operation in the worst case
- Provides a more precise and useful evaluation of the difficulty of a problem than average case complexity

Common strategies

Three main approaches to amortized analysis:

- *Aggregate method:* most simple approach where the total running time for the sequence is analysed and divided by the number of operations
- *Accounting method:* charge each type of operation a constant cost such that the extra charge on inexpensive ones can be stored in a "bank" and used to pay expensive subsequent operations
- *Potential method:* evaluate and store the total amount of extra work done over the whole data structure and release it to cover the cost of subsequent operations

Remark. The average cost for each operation in the case of dynamic tables (7.293) was determined using the aggregate method.

Accounting method

Let c_i denote the actual real cost of the ith operation, while $\hat{c}_i$ represents its amortized cost, i being larger than 1. The goal is to always ensure that at any stage the sum of the $\hat{c}_i$ is larger than the sum of the c_i, meaning that some extra credit is available for future operations.

For a sequence of n operations the bank balance never becoming negative can be expressed as

$$\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \geq 0.$$

Example. In the case of dynamic tables (7.293) charge $\hat{c}_i = 3$ units for each insertion: one is used on insertion while the 2 remaining are saved for a future use. In particular when memory is reallocated a unit is use for reassigning the current element while the last one is spent to move an older value.

Potential method

Let D_0 be an initial data structure. At step i, the ith operation, costing c_i, is applied to D_{i-1}. The *potential* associated with the data structure D_i, denoted $\Phi(D_i)$, is a real number and Φ is called the *potential function*. The amortized cost corresponds to the actual cost plus the change in potential induced by the operation. This formalizes as $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$, and we get

$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) \geq \sum_{i=1}^{n} c_i .\end{aligned}$$

Example. For the dynamic tables (7.293) set the potential after the ith insertion to $\Phi(D_i) = 2i - 2^{\lceil \log_2 i \rceil}$. Then whether or not $i - 1$ is a power of 2, the amortized cost is 3.

Notes

Remarks on the three methods:

- The accounting and potential methods are more powerful and refined than the aggregate one; they are equivalent in terms of applicability and precision of the bound provided
- The accounting method charges a different cost for each type of operation
- The potential method focuses on the effect of a particular operation at a specific time, and in particular on the cost of future operations
- Different methods might lead to different bounds, but always upper bound the actual cost

Union-Find data structure

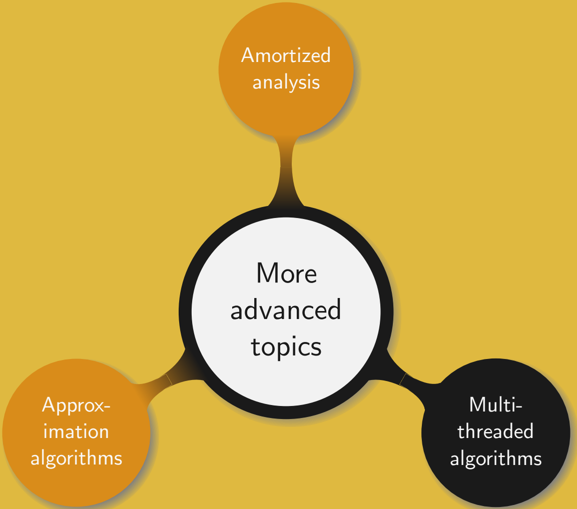
For the Union-Find data structure (1.56), theorem 2.91 can be proven using either the accounting or the potential method.

The potential method suits this example since the goal is to determine how fast the tree is flattening, or in other words how each operation affects the whole data structure. By observing the change in potential after each type of operation it is then possible to determine the amortized cost associated with each of them.

The first and less straight forward step consists in properly setting the potential function. Two cases have to be considered (i) x is a root or has rank 0, and (ii) x is not a root and has rank larger or equal to 1. Defining the potential of x after i operations is most complicated in the latter case.

Once the potential function has been properly defined it can be bounded by $0 \leq \phi_i(x) \leq \alpha(n) \cdot x.rank$, and the amortized cost of the various operations can be determined.

Chapter organisation



More advanced topics

Amortized analysis

Approx-imation algorithms

Multi-threaded algorithms

A new model

Algorithm discussed so far targeted *uniprocessor* computers, while most modern systems feature *multiprocessors* sharing a common memory. The question is then to know how to adapt those algorithms to this new context, and in particular how to efficiently partition the work among several *threads*, each having roughly the same load.

The most simple strategy consists in employing a software layer, called *concurrency platform*, which coordinates, schedules, and manages the resources.

A simple extension to *serial programming* is the addition of the following instructions: `parallel`, `spawn`, and `sync`.

Concurrency instructions

The three new instructions:

- `parallel:` added to loops to indicate that iterations can be computed in parallel
- `spawn:` executes a new parallel process, the current one may then choose to run concurrently or wait for its child
- `sync:` requests the process to wait for all spawned processes to complete

We measure the improvement resulting from using multiprocessors by looking at the ratio of the complexity of the multi-threaded algorithm with the one of the single threaded version. When measuring this ratio, called *parallelism*, an infinite number of processors is assumed.

Sieve of Eratosthenes

	2	3	4	5	6	7	8	9	10	11
12	13	14	15	16	17	18	19	20	21	22
23	24	25	26	27	28	29	30	31	32	33
34	35	36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63	64	65	66
67	68	69	70	71	72	73	74	75	76	77
78	79	80	81	82	83	84	85	86	87	88
89	90	91	92	93	94	95	96	97	98	99
100	101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120	121

Parallel threads

In Eratosthenes sieve, after processing multiples of a prime p up to p^2, new threads can be spawned for all the integers less than p^2 which have not been marked.

In some cases a *race condition* might occurs, leading to a result that is non-deterministic. A simple example is as follows.

Algorithm.

```
Input  : $x$
Output: $x + 2$ is expected
parallel for $i \leftarrow 1$ to $2$ do
    $x \leftarrow x + 1$;
end for
return $x$;
```

Modeling multi-threaded algorithms

Two main strategies can be employed to avoid race conditions: (i) a thread sets a *lock* when reaching a critical part of the code to prevent any other thread to run it at the same time; (ii) use special hardware instructions called *atomic operations* which can run several operations at once.

Definition (Computation DAG)

Multi-threaded algorithms can be represented as Directed Acyclic Graph (DAG), often referred to as *computation dag*.

More specifically each vertex stands for an instruction while the edges organize the dependencies between the various instructions. An edge (u, v) implies that instruction u must be run before v.

A chain composed of one or more instructions and not containing any concurrency instruction is called a *strand*. Two strands connected by a directed path are in *series*, or otherwise in *parallel*.

A computer composed of *n* processors is expected to run *n* concurrent threads. The *scheduler* is the part of the Operating System which decides which thread to run and on which CPU.

Several approaches are to be used depending on the hardware. Modern desktop and laptop computers feature a memory shared among several CPU, which is much easier to handle than having an independent memory for each CPU.

Example. OpenMP allows to easily add parallelism to existing source code without requiring any significant rewrite. It perfectly suites systems where the memory is shared among multiple CPUs.

MPI offers advanced possibilities more specifically targeting systems with a distributed memory. It is very common in the realm of high end computing, especially on clusters. MPI often requires a complete redesign, or even change of algorithm to implement.

Merge Sort

Assuming the existence of a `Merge` function, the following algorithm performs serial Merge Sort on a given list L.

Algorithm. (*Merge Sort*)

```
Input  : $L = a_1, \dots, a_n$
Output : $L$, sorted into elements in non-decreasing order

Function MergeSort($L$):
    if $n > 1$ then
        $m \leftarrow \lfloor n/2 \rfloor$;
        $L_1 \leftarrow a_1, \dots, a_m$; $L_2 \leftarrow a_{m+1}, \dots, a_n$;
        $L \leftarrow$ Merge(MergeSort($L_1$), MergeSort($L_2$))
    end if
    return $L$
end
```

The recursive calls `MergeSort` seem to make this algorithm a good candidate for parallelism.

Merge

Algorithm. (*Merge*)

Input : L_1, L_2, two sorted lists
Output : L a merged list of L_1 and L_2, with elements in increasing order

```
Function Merge($L_1, L_2$):
    $L \leftarrow \emptyset$;
    while $L_1 \neq \emptyset$ and $L_2 \neq \emptyset$ do
        $x \leftarrow \min(L_1, L_2)$;              /* smallest element in $L_1$ and $L_2$ */
        append $x$ to $L$;
        if $L_1 = \emptyset$ or $L_2 = \emptyset$ then
            $L_1 = \emptyset$ ? append $L_2$ to $L$ : append $L_1$ to $L$;
        end if
    end while
    return $L$;
end
```

Algorithm. (*Parallel Merge Sort*)

Input : $L = a_1, \dots, a_n$

Output : L, sorted into elements in non-decreasing order

```
Function P-MergeSort(L):
    if $n > 1$ then
        $m \leftarrow \lfloor n/2 \rfloor$;
        $L_1 \leftarrow a_1, \dots, a_m$; $L_2 \leftarrow a_{m+1}, \dots, a_n$;
        $L_1 \leftarrow$ spawn P-MergeSort ($L_1$);
        $L_2 \leftarrow$ spawn P-MergeSort ($L_2$);
        sync $L \leftarrow$ Merge ($L_1, L_2$);
    end if
    return $L$;
end
```

In fact this parallel version does not improve much on the serial version: although Merge Sort is parallel, Merge remains serial and as such the recurrence relation still features the same $\mathcal{O}(n)$ term. As a result parallelism is only $\Theta(\log n)$.

Parallel Merge

Although at times serial algorithms can easily be made parallel, it is often impossible, and they require to be completely redesigned using a totally different approach. In some other cases algorithms might seem to be intrinsically serial, but can still be adjusted at the cost of some extra work. This is for instance the case of the Merge algorithm.

The idea is to carefully generate four lists that can be merged two by two without altering the ordering. The process is as follows.

1. Find the median element in the longest list: define two sublists $L_{1,L}$ and $L_{1,R}$
2. Determine its corresponding potential location in the second list: define two sublists $L_{2,L}$ and $L_{2,R}$
3. Recursively merge the lists $L_{1,L}$ and $L_{2,L}$, and $L_{1,R}$ and $L_{2,R}$

Parallel Merge

Algorithm. (*Parallel Merge*)

Input : L_1, L_2, two sorted lists, with $L_1.length \geq L_2.length$

Output : L, a merged list of L_1 and L_2, with elements in increasing order

```
Function P-Merge($L_1$, $L_2$):
    if $L_2 = \emptyset$ then $L \leftarrow L_1$ ;
    else
        $m_1 \leftarrow \lfloor L_1.length/2 \rfloor$;
        $L_{1,L} \leftarrow \left\{L_{1_i}\right\}_{1 \leq i < m_1}$; $L_{1,R} \leftarrow \left\{L_{1_i}\right\}_{m_1 < i \leq L_1.length}$;
        $m_2 \leftarrow$ index where $L_{1_{m_1}}$ would be in $L_2$;
        $L_{2,L} \leftarrow \left\{L_{2_i}\right\}_{1 \leq i < m_2}$; $L_{2,R} \leftarrow \left\{L_{2_i}\right\}_{m_2 \leq i \leq L_2.length}$;
        if $L_{1,L}.length < L_{2,L}.length$ then swap($L_{1,L}$, $L_{2,L}$);
        if $L_{1,R}.length < L_{2,R}.length$ then swap($L_{1,R}$, $L_{2,R}$);
        $L_L \leftarrow$ spawn P-Merge($L_{1,L}$, $L_{2,L}$); $L_R \leftarrow$ P-Merge($L_{1,R}$, $L_{2,R}$); sync;
        $L \leftarrow$ concatenate $L_L$, $L_{1_{m_1}}$, $L_R$;
    end if
    return $L$;
end
```

The complexity is $\Theta(n)$ while the parallelism grows to $\Theta(n/\log^2 n)$.

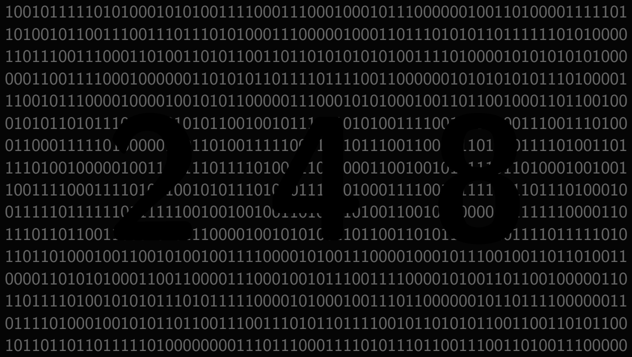
Final notes

Designing efficient algorithms requires:

- A good understanding of the problems
- An advanced knowledge of the underlying mathematics
- Much time spent on a trial and errors approach
- The consideration of all the corner cases

Key points

- When to use approximation algorithms?
- How to evaluate the quality of an approximation algorithm?
- Describe amortized analysis.
- Cite the three main methods to perform amortized analysis.
- What are the instruction `parallel`, `spawn`, and `sync`?



Thank you, enjoy the Winter break!