

COMP90038

Algorithms and Complexity

Lecture 22: NP Problems and Approximation Algorithms
(with thanks to Harald Søndergaard & Michael Kirley)

Andres Munoz-Acosta

munoz.m@unimelb.edu.au

Peter Hall Building G.83

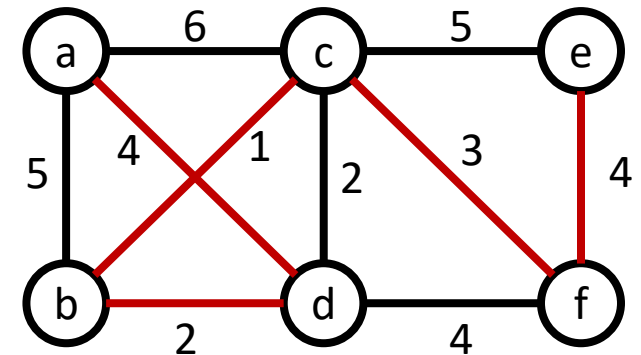
Recap

- We continued discussing **greedy algorithms**:
 - A problem solving strategy that takes the **locally best** choice among all feasible ones. Such choice is **irrevocable**.
 - Usually, **locally best** choices do not yield **global best** results.
 - In some exceptions a greedy algorithm is **correct and fast**.
 - Also, a greedy algorithm can provide good **approximations**.
- We applied this idea to graphs and data compression:
 - Prim's and Dijkstra Algorithms
 - Huffman Algorithms and Trees for variable length encoding.

Prim's Algorithm

- Starting from different nodes produces a different sequence.
 - However, the tree will have the same edges.
 - Unless there are edges with the same weights, as tie breaking would influence which one to take.
- The following example has only one tree. Tie breaking was done alphabetically.

START	SEQUENCE	EDGES
a	a-d-b-c-f-e	(a,d)(b,d)(b,c)(c,f)(e,f)
b	b-c-d-f-a-e	(b,c)(b,d)(c,f)(a,d)(e,f)
c	c-d-b-f-a-e	(c,d)(b,d)(c,f)(a,d)(e,f)

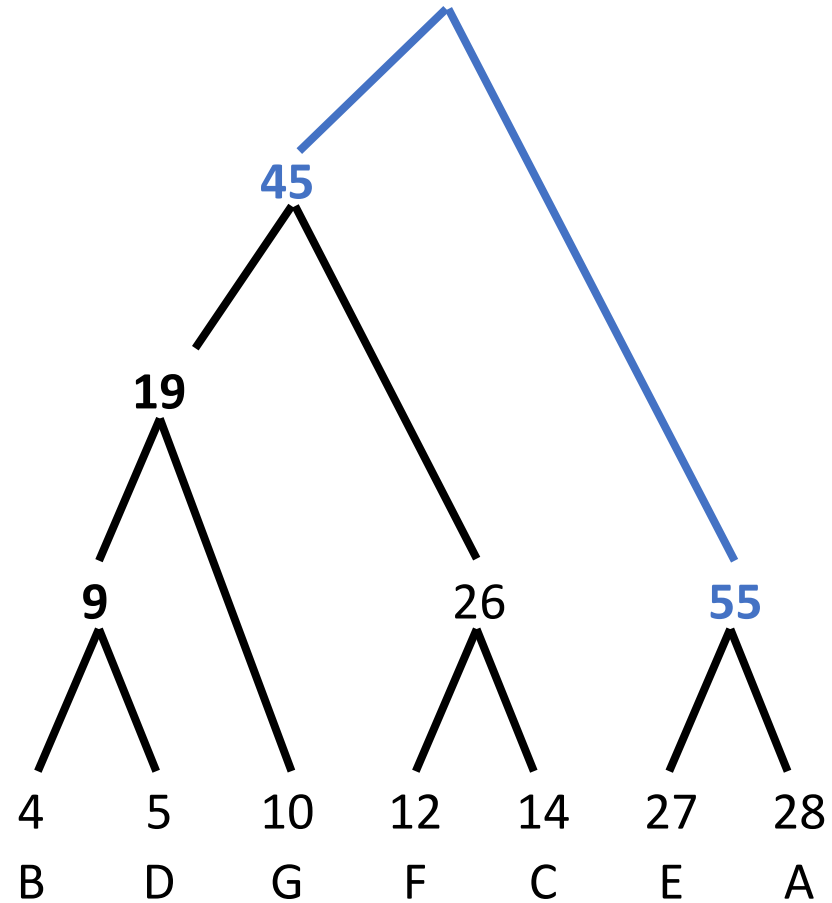


Variable-Length Encoding

- Variable-Length encoding assigns shorter codes to common characters.
 - In English, the most common character is **E**, hence, we could assign **0** to it.
 - However, no other character code can start with **0**.
- That is, no character's code should be a prefix of some other character's code (unless we somehow put separators between characters, which would take up space).
- The table shows the occurrences and some sensible codes for the alphabet {A,B,C,D,E,F,G}
 - This table was generated using **Huffman's algorithm** – another example of a **greedy method**.

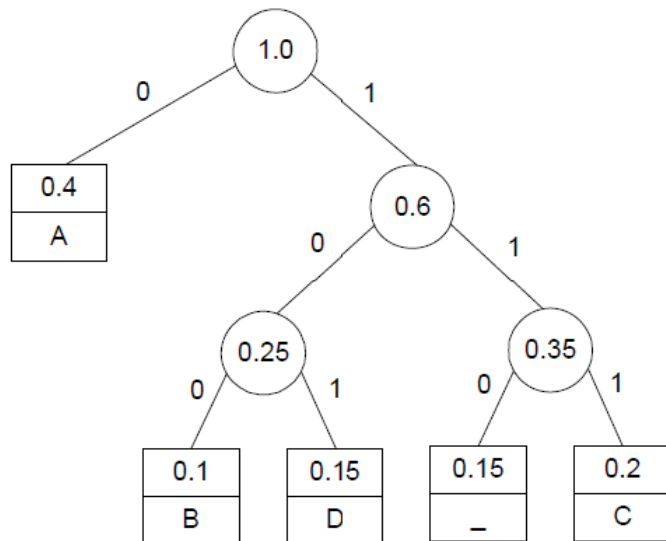
SYMBOL	OCCURRENCE	CODE
A	28	11
B	4	0000
C	14	011
D	5	0001
E	27	10
F	12	010
G	10	001

Huffman Trees (example)



An exercise

- Construct the Huffman code for data in the table, placing in the tree from left to right [A,B,D,C,_]
- Then, encode **ABACABAD** and decode **100010111001010**
- 0100011101000101 / BAD_ADA



SYMBOL	FREQUENCY	CODE
A	0.40	0
B	0.10	100
C	0.20	111
D	0.15	101
_	0.15	110

Concrete Complexity

- So far our concern has been the analysis of algorithms from the running time point of view (best, average, worst cases)
- Our approach has been to determine the **asymptotic** behavior of the running time **as a function of the input size**.
 - For example, the quicksort algorithm is $O(n^2)$ in the worst case, whereas mergesort is $O(n \log n)$.

Abstract Complexity

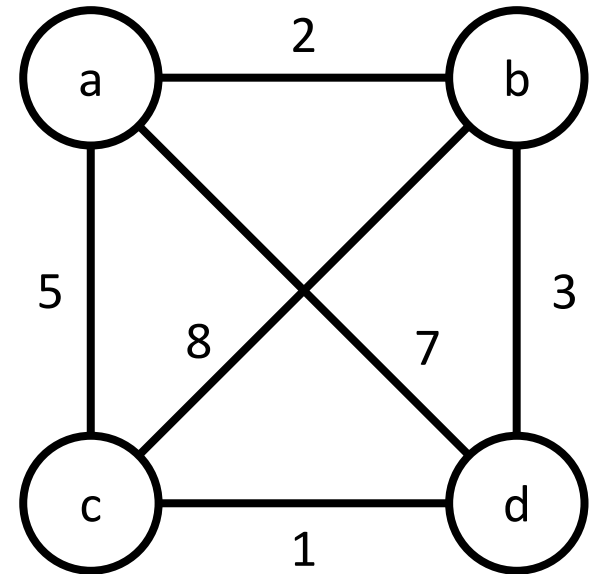
- The field of **complexity theory** focuses on the question:

“What is the inherent difficulty of the **problem**?”

- How do we know that an algorithm is **optimal** (in the asymptotic sense)?

Difficult problems

- Which problems are difficult to solve?
- The Travelling Salesman problem can be solved through brute force for very small instances.
 - One solution is: a-b-d-c-a
- However, it becomes very difficult as the number of nodes and connections increase.
 - However, you can check the solution and determine if it is a good solution or not?



Does $P=NP$?

- The “**P versus NP**” problem comes from **computational complexity theory**
- P means with polynomial time complexity
 - That is, algorithms that have $O(\text{poly}(n))$
 - Sorting is a type of polynomial time problem
- NP means non-deterministic polynomial
 - You can check the answer in polynomial time, but cannot find the answer in polynomial time for large n
 - The TSP problem is an NP problem
- This is the most important question in Computer Science

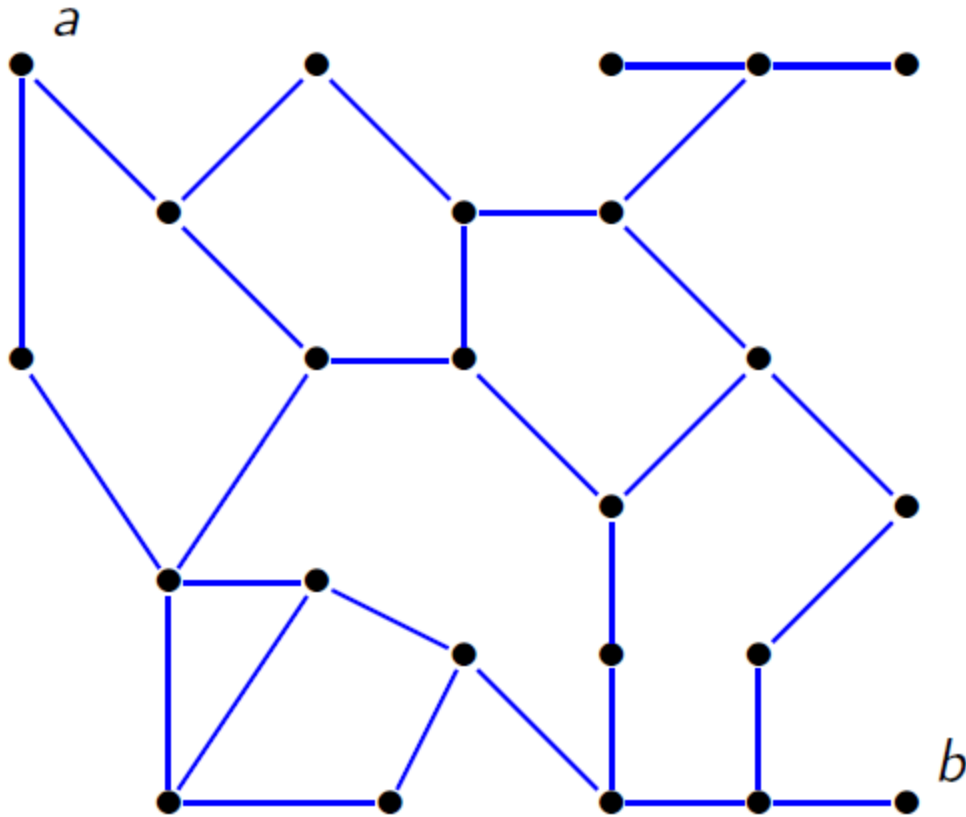
Algorithmic problems

- When we talk about a **problem**, we almost always mean a family of **instances** of a general problem
- An **algorithm** for the problem has to work for all possible instances
- Examples:
 - The **sorting** problem – an instance is a sequence of items.
 - The **graph k-colouring** problem – an instance is a graph.
 - **Equation solving** problems – an instance is a set of, say, linear equations.

Easy and hard problems

- A path in a graph G is **simple** if it visits each node of G at most once.
- Consider these two problems for undirected graphs G :
 - **SPATH**: Given G and two nodes a and b in G , is there a simple path from a to b of length **at most** k ? *simpler*
 - **LPATH**: Given G and two nodes a and b in G , is there a simple path from a to b of length **at least** k ?
- If you had a large graph G , which of the two problems would you rather have to solve?

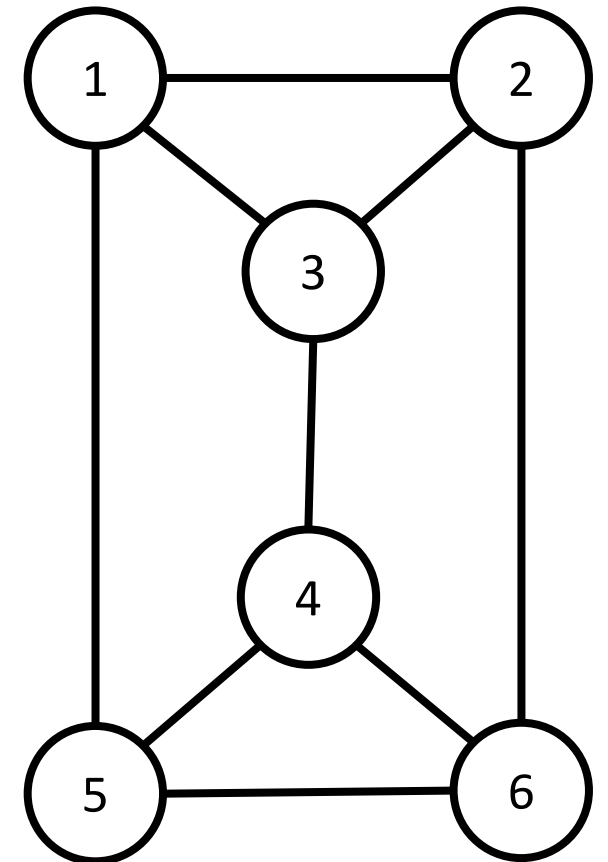
Easy and hard problems



- There are fast algorithms to solve SPATH.
 - For example, we can do a BFS over the graph.
- Nobody knows of a fast algorithm for LPATH.
- It is likely that the LPATH problem cannot be solved in polynomial time.

Easy and hard problems

- Other two related problems:
 - The Eulerian tour problem: In a given graph, is there a path which visits each **edge** of the graph once, returning to the origin?
 - The Hamiltonian tour problem: In a given graph, is there a path which visits each **node** of the graph once, returning to the origin?
- Is the Eulerian tour problem P?
 - We just need to know whether the edge distribution is even.
- Is the Hamiltonian tour P?
 - No. As the nodes increase, runtime becomes exponential.



Easy and hard problems

- Some more examples:
 - **SAT**: Given a propositional formula ψ , is ψ satisfiable?
 - **SUBSET-SUM**: Given a set S of positive integers and a positive integer t , is there a subset of S that adds up to t ?
 - **3COL**: Given a graph G , is it possible to colour the nodes of G using only three colours, so that no edge connects two nodes of the same colour?
- Although these problems are very different they share an interesting property

Polynomial time verifiability

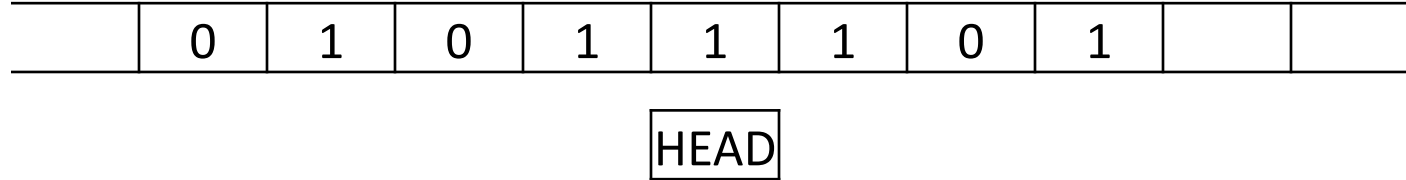
- While most instances of these problems cannot be solved in polynomial time, we can test a solution in polynomial time
- In other words, while they **seem hard to solve**, they allow for **efficient verification**.
- This is called **polynomial-time verifiable**
- To understand this concept we need to talk about **Turing Machines**

Turing Machines

- Turing Machines are an **abstract model of a computer**.
- Despite of their simplicity, they appear to have the same **computational power** than any other computing device
 - That is, any function that can be implemented in C, Java, etc. can be implemented in a Turing Machine
- Moreover, a Turing Machine is able to **simulate** any other Turing Machine.
 - This is known as the **universality** property

Turing Machines

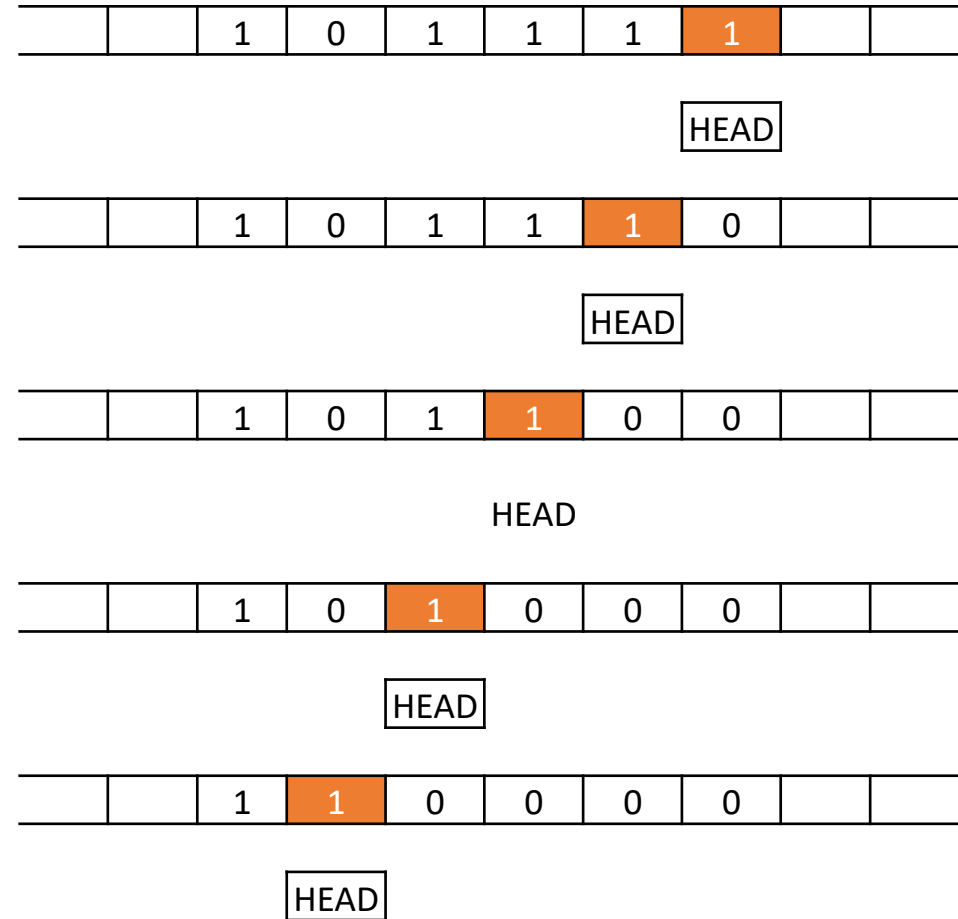
- A Turing machine is represented as an **infinity sized memory space**, and a **read/write head**



- Whether the head reads, writes or moves to left or right depends of a **control sequence**

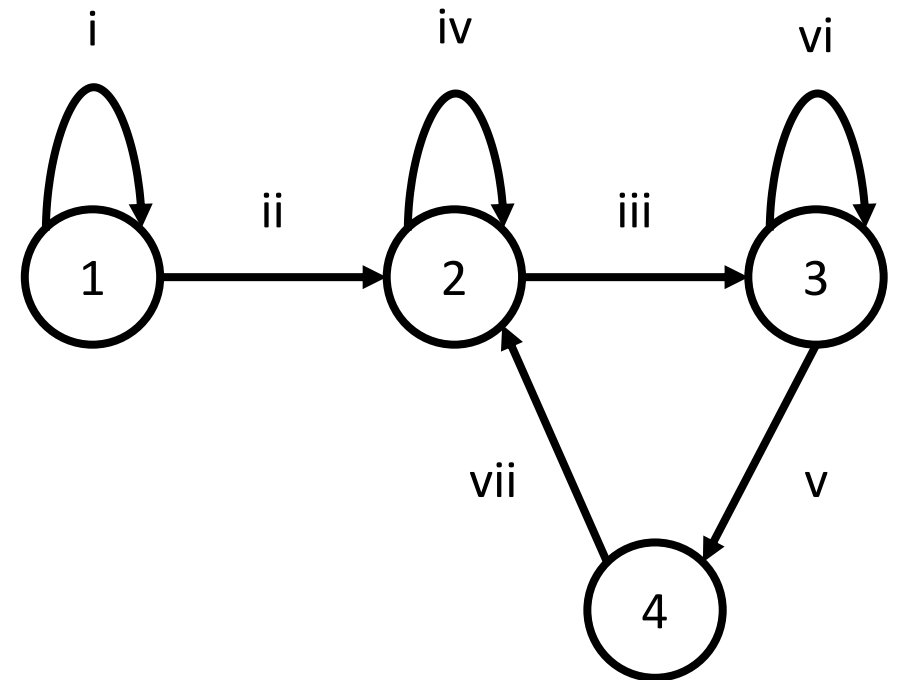
An example

- Let the control sequence be:
 - If read **1**, write **0**, go **LEFT**
 - If read **0**, write **1**, **HALT**
 - If read **_**, write **1**, **HALT**
- The input will be $47_{10} = 101111_2$
- The output is $48_{10} = 11000_2$
 - In other words, this rules add one to a number

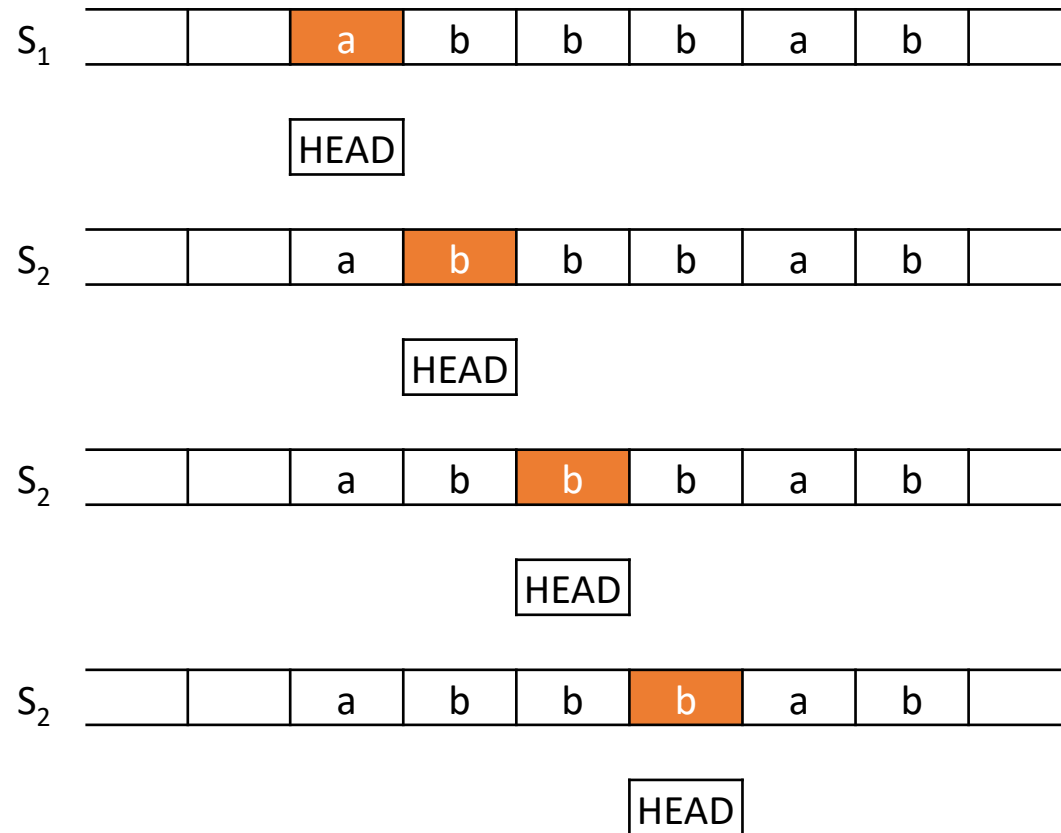


A more complex control sequence

- We will develop an state automaton:
 - If S_1 and **a**, go **RIGHT** stay in S_1
 - If S_1 and **b**, go **RIGHT** go to S_2
 - If S_2 and **a**, write **b** go **LEFT** go to S_3
 - If S_2 and **b**, go **RIGHT** stay in S_2
 - If S_3 and **a** or **_**, go **RIGHT** go to S_4
 - If S_3 and **b**, go **LEFT** stay in S_3
 - If S_4 and **b**, write **a** go **RIGHT** go to S_2



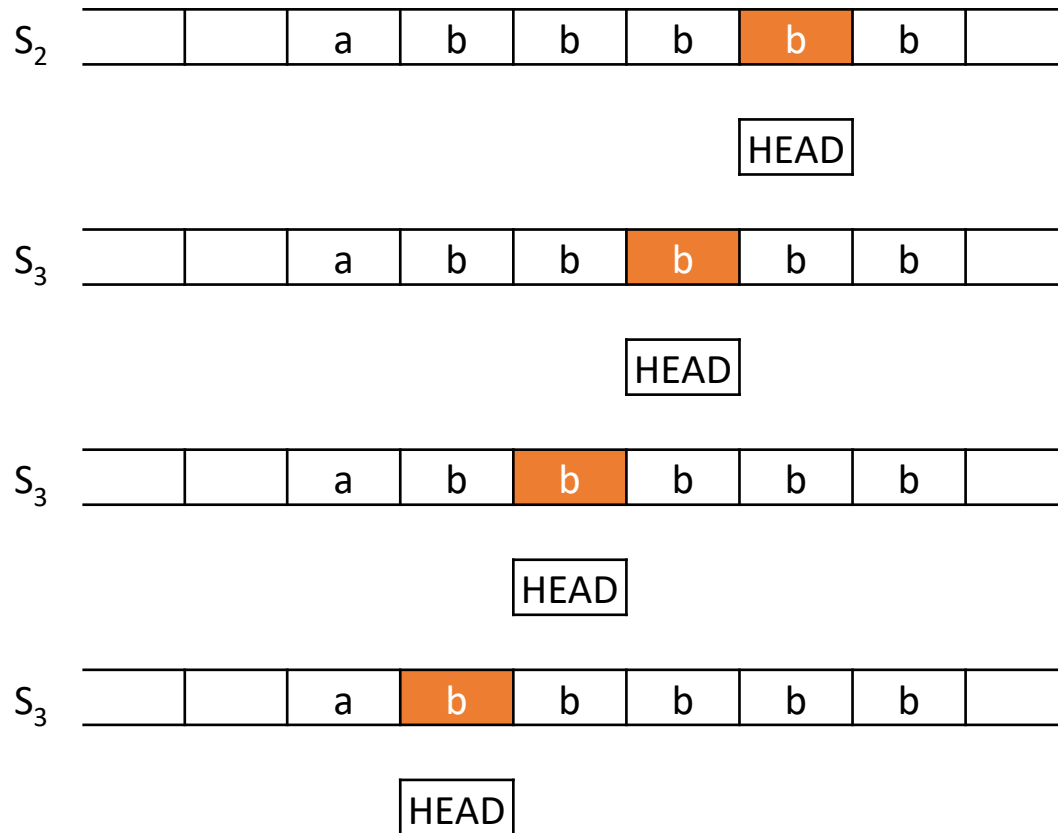
Example



• What would this machine do for the input **abbbab**?

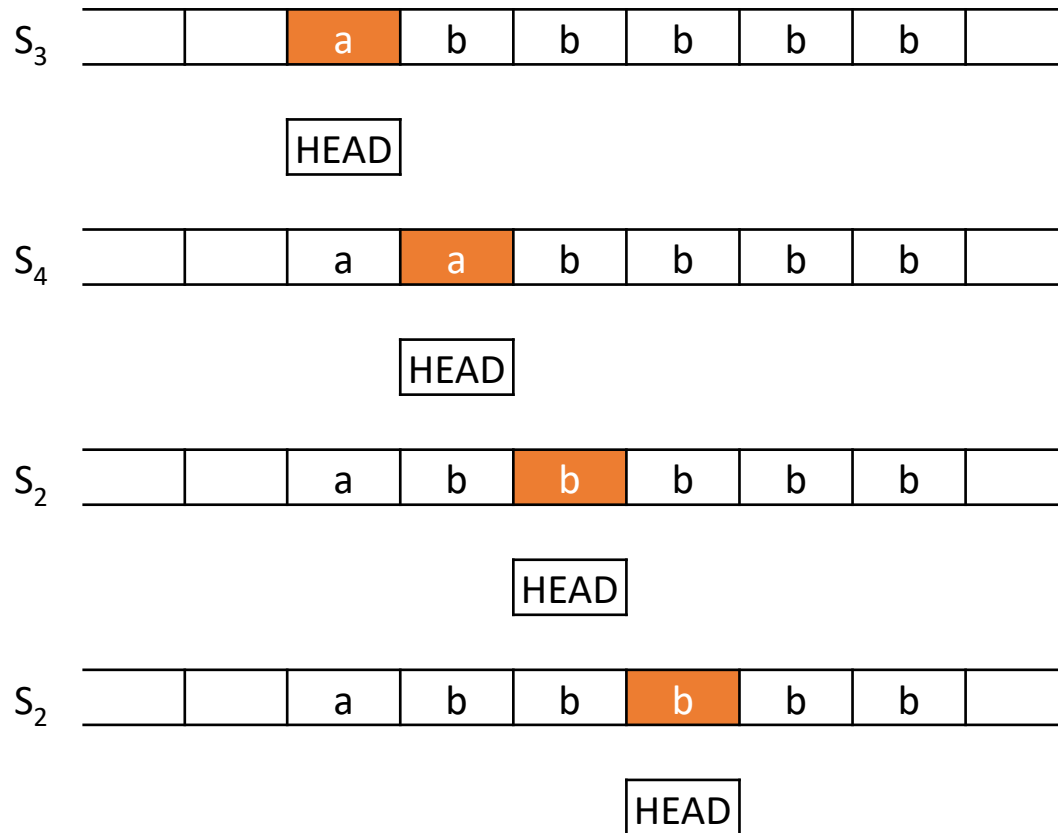
- i. If S_1 and **a**, go **RIGHT** stay in S_1
- ii. If S_1 and **b**, go **RIGHT** go to S_2
- iii. If S_2 and **a**, write **b** go **LEFT** go to S_3
- iv. If S_2 and **b**, go **RIGHT** stay in S_2
- v. If S_3 and **a** or **_**, go **RIGHT** go to S_4
- vi. If S_3 and **b**, go **LEFT** stay in S_3
- vii. If S_4 and **b**, write **a** go **RIGHT** go to S_2

Example



- What would this machine do for the input **abbbab**?
 - i. If S_1 and **a**, go **RIGHT** stay in S_1
 - ii. If S_1 and **b**, go **RIGHT** go to S_2
 - iii. If S_2 and **a**, write **b** go **LEFT** go to S_3
 - iv. If S_2 and **b**, go **RIGHT** stay in S_2
 - v. If S_3 and **a** or `_`, go **RIGHT** go to S_4
 - vi. If S_3 and **b**, go **LEFT** stay in S_3
 - vii. If S_4 and **b**, write **a** go **RIGHT** go to S_2

Example



- What would this machine do for the input **abbbab**?
 - i. If S₁ and **a**, go **RIGHT** stay in S₁
 - ii. If S₁ and **b**, go **RIGHT** go to S₂
 - iii. If S₂ and **a**, write **b** go **LEFT** go to S₃
 - iv. If S₂ and **b**, go **RIGHT** stay in S₂
 - v. If S₃ and **a** or **_**, go **RIGHT** go to S₄
 - vi. If S₃ and **b**, go **LEFT** stay in S₃
 - vii. If S₄ and **b**, write **a** go **RIGHT** go to S₂
- The machine **sorts** the letters upon completion

Non-deterministic Turing Machines

- From now onwards we will assume that a Turing Machine will be used to implement **decision procedures**
 - That is an algorithm with **YES/NO** answers
- Now, lets assume that one of such machines has a powerful **guessing** capability:
 - If different moves are available, the machine will favour one that leads to a **YES** answer
- Adding this **non-deterministic** capability does not change **what** the machine can compute, but affects its **efficiency**

Non-deterministic Turing Machines

- What a non-deterministic Turing machine can compute in polynomial time corresponds exactly to the class of polynomial-time verifiable problems.
- In other words:
 - **P** is the class of problems solvable in polynomial time by a **deterministic** Turing Machine
 - **NP** is the class of problems solvable in polynomial time by a **non-deterministic** Turing Machine
- Clearly $P \subseteq NP$. Is $P = NP$?

Problem reduction

- The main tool used to determine the class of a problem is **reducibility**
- Consider two problems P and Q
- Suppose that we can transform, **without too much effort**, any instance p of P into an instance q of Q
- Such transformation should be **faithful**. That is we can extract a solution to p from a solution of q

A very simple example

- **Multiplication and squaring:**

- Suppose all we know to do is how to add, subtract, take squares and divide by two.
- Then, we can use this formula to calculate the product of any two numbers:

$$a \times b = \frac{((a + b)^2 - a^2 - b^2)}{2}$$

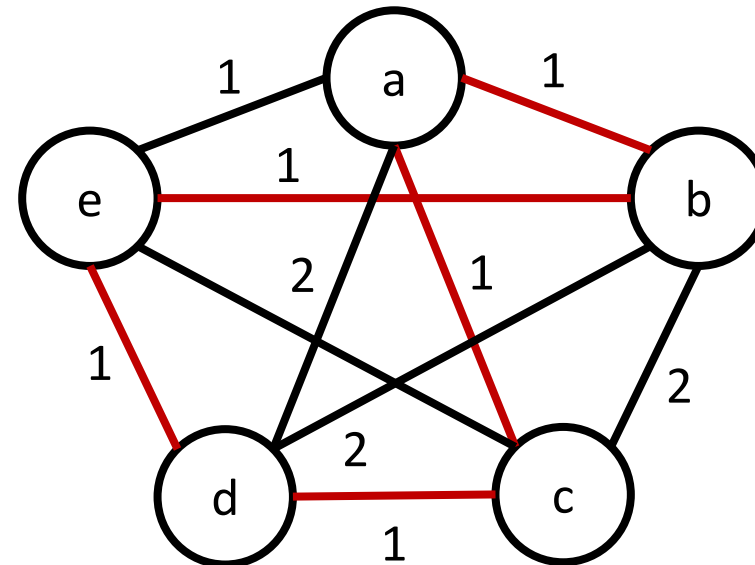
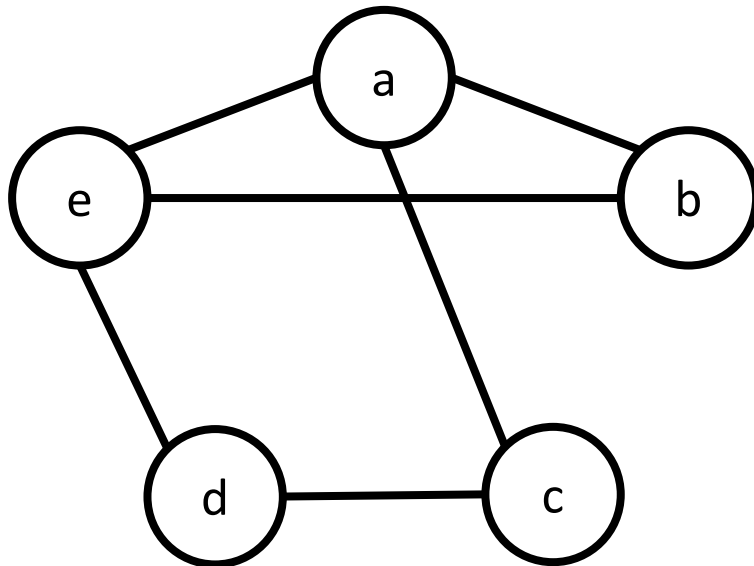
- We can also go the other direction, that is, if we can multiply two numbers, we can calculate the square.

Another example

- The Hamiltonian cycle (HAM) and the Travelling Salesman (TSP) problems have similarities:
 - Both operate on graphs
 - Both try to find a tour that visits the vertices just once
- The only difference is that the HAM works in unweighted graphs and TSP does in weighted graphs

Reducing HAM to TSP

- We can transform a HAM problem into a TSP problem:
 - By assigning **1** to all the edges in the unweighted graph
 - By creating paths between unconnected edges with weight of **2**
 - If there is a TSP tour of length n , then there is a Hamiltonian cycle.



Problem reduction

- Problem reduction allows us to make a few conclusions:
 - If a reduction from P to Q exist, then the P is **at least as hard** as Q
 - If Q is known to be hard, then we may decide **not to waste more time** trying to find an efficient algorithm for P

Dealing with difficult problems

- **Pseudo-polynomial problems** (SUBSET-SUM and KNAPSACK are in this class): Unless you have really large instance, there is no need to panic. For small enough instances the bad behavior is not yet present.
- **Clever engineering** to push the boundary slowly: SAT solvers.
- **Approximation algorithms**: Settle for less than perfection.
- **Live happily** with intractability: Sometimes the bad instances never turn up in practice.

Approximation Algorithms

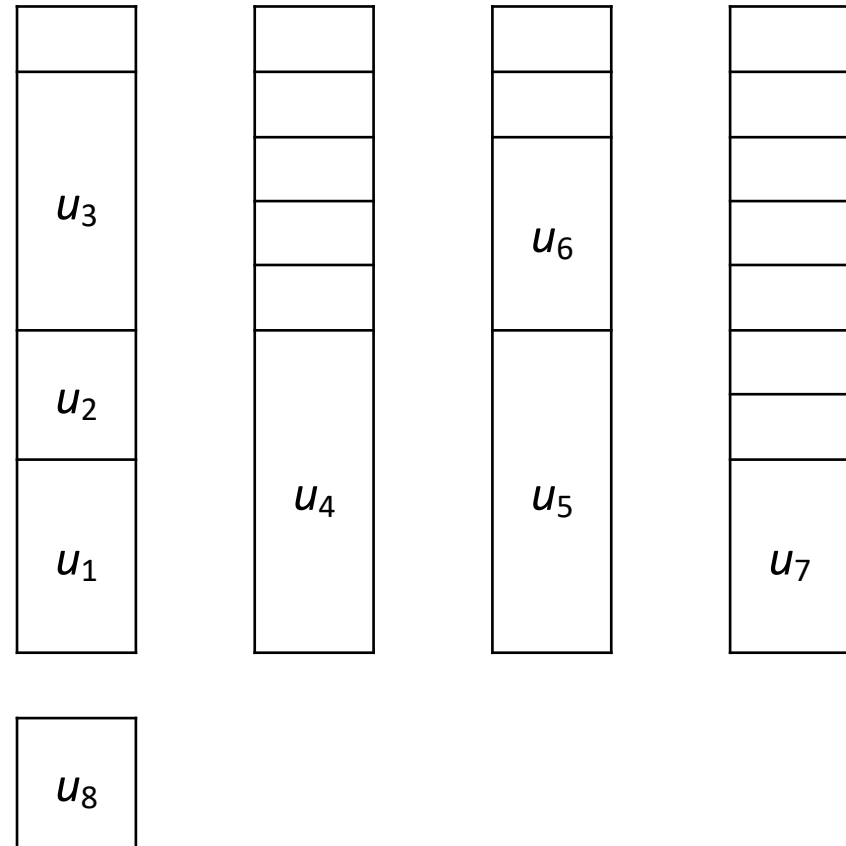
- For intractable optimization problems, it makes sense to look for **approximation algorithms** that are fast and still find solutions that are reasonably close to the optimal.

Example: Bin packing

- **Bin packing** is closely related to the knapsack problem.
- Given a finite set $U = \{u_1, u_2, \dots, u_n\}$ of items and a rational size $s(u) \in [0, 1]$ for each item $u \in U$, partition U into disjoint subsets U_1, U_2, \dots, U_k such that
 - the sum of the sizes of items in U_i is at most 1; and
 - k is as small as possible.
- The bin-packing problem is NP-hard.

Bin packing

- In plain English, Each subset U_i gives the set of items to be placed in a unit-sized “bin”, with the objective of using as few bins as possible.
- There some **heuristics** that can be used.
 - First Fit: Use the first bin that has the necessary capacity



Bin packing

- For First Bin, the number of bins used Fit is never more than **twice** the minimal number required.
 - First Fit behaves worst when we are left with many large items towards the end.
- The variant in which the items are taken in order of decreasing size performs better.
- The added cost (for sorting the items) is not large.
- This variation guarantees that the number of bins used cannot exceed $\frac{11n}{9} + 4$ where n is the optimal solution.

Next week

- We will review the contents of this unit