Algorithms and Data Structures
Module 4. NP-hard problems

Lecture 18
Algorithms for NP-hard problems. Travelling Salesman Problem.

Time complexity

Let’s recall time complexities of algorithms we studied in this course.

Algorithm	Time complexity	Majorant
Binary search	$O(\log n)$	$O(n)$
Bubble/Insertion/Selection sort	$O(n^2)$	$O(n^2)$
Merge sort	$O(n \log n)$	$O(n^2)$
Graph connectivity components detection	$O(m)$	$O(n^2)$
Kruskal’s (with Union-Find Set data structure)	$O(m \log m) = O(n^2 \log n)$	$O(n^3)$
Prim’s (with binary heap as priority queue)	$O(m\ \log n) = O(n^2 \log n)$	$O(n^3)$
Karatsuba’s integer multiplication	$\Theta\left(n^{\log_2 3}\right)$	$O(n^2)$
Strassen’s matrix multiplication	$O\left(n^{\log_2 7}\right)$	$O(n^3)$
Fast exponentiation	$O(\log n)$	$O(n)$
		(to be continued on the next slide…)

Time complexity

Algorithm	Time complexity	Majorant
		(…continuation)
Dijkstra's algorithm for general case	$O(nm)$	$O(n^3)$
Floyd-Warshall's	$O(n^3)$	$O(n^3)$
Needleman-Wunsch (Levenshtein's edit distance)	$O(nm)$	$O(n^2)$
Longest common subsequence	$O(nm)$	$O(n^2)$
Optimal BST	$O(n^3)$	$O(n^3)$

We see that for all the above algorithms there is a constant *c* such that the algorithm's time complexity is $O(n^c)$.

Such algorithms are called ***polynomial time*** algorithms.

Time complexity

For the problem of calculating Fibonacci numbers we discussed two algorithms:

- A dynamic programming algorithm with polynomial time complexity $O(n)$.
- A recursive algorithm with time complexity $O(\varphi^n)$ for $\varphi = \frac{1+\sqrt{5}}{2}$.

The recursive algorithm is not polynomial time, it is an exponential time algorithm…

Time complexity

Let’s consider two algorithms for a problem with time complexities $O(n)$ and $O(2^n)$.

n	O(n)	$O(2^n)$
50	1.00 sec	1 sec
51	1.02 sec	2 sec
52	1.04 sec	4 sec
60	1.20 sec	17 min
70	1.40 sec	12 days
80	1.60 sec	34 years
90	1.70 sec	~ 35 000 years

Time complexity

That is why polynomial time algorithms are called ***efficient***, whereas exponential time algorithms are considered *inefficient*.

For many problems no efficient algorithms are known… ☹

Moreover, for most of these problems it was proved that if a polynomial time algorithm would be designed for one of these problems, this immediately imply polynomial time algorithms for all such problems.

Such problems are called ***NP-hard***.

Time complexity

There are thousands of NP-hard problems…

One of the most famous NP-hard problems is the Travelling Salesman Problem (TSP).

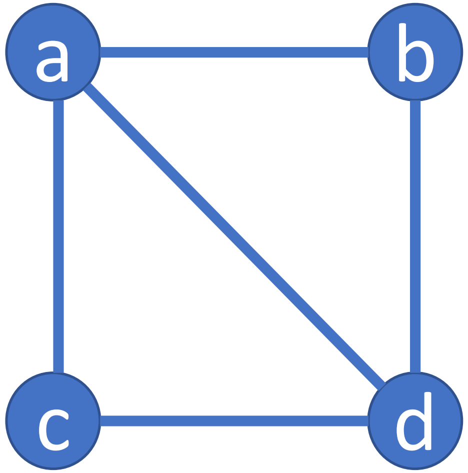
TSP: definitions

Let $G(V, E)$ be a connected graph, $w: E \longrightarrow R_+$ be a weights function.

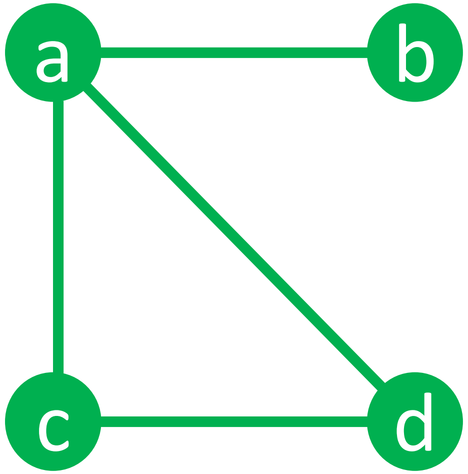
Definitions

- Cycle Z (path P) is called a ***Hamiltonian cycle (Hamiltonian path)*** on G iff Z (P) contains each vertex of G exactly once.
- $G(V, E)$ is called a *Hamiltonian* (*semi-Hamiltonian*) graph iff there is a Hamiltonian cycle (path) on G.
- The weight of Z (or P) is defined as $w(Z) = \sum_{e \in Z} w(e)$.

TSP: definitions

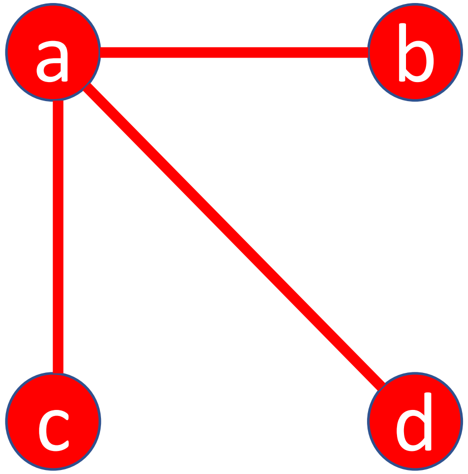


Hamiltonian graph



Semi-hamiltonian graph

Nonhamiltonian graph



TSP: definitions

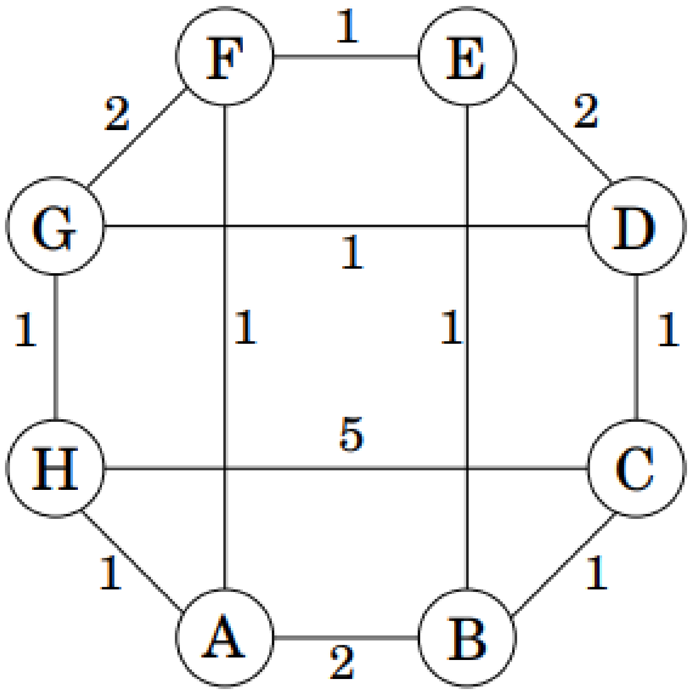
- Decision problem: is the given graph $G(V, E)$ Hamiltonian?

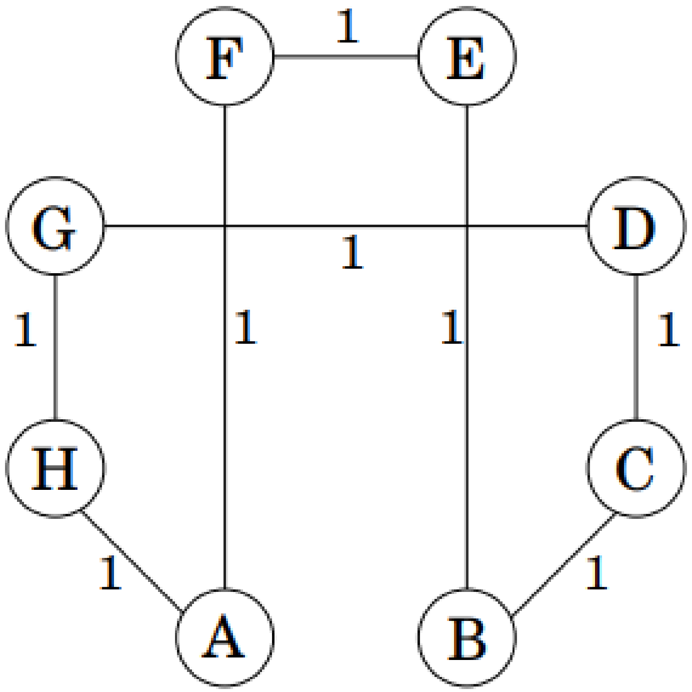
- Search problem: build a Hamiltonian cycle on the given graph $G(V, E)$ (return ‘NULL’ if $G(V, E)$ is not Hamiltonian).

- Optimization problem (=TSP): build a shortest Hamiltonian cycle on the given graph $G(V, E)$ (return ‘NULL’ if $G(V, E)$ is not Hamiltonian).

TSP: definitions

A graph and its optimal Hamiltonian cycle:





http://algorithmics.lsi.upc.edu/docs/Dasgupta-Papadimitriou-Vazirani.pdf

TSP: solving

Theorem 1: TSP is NP-hard.

TSP: solving

Possible options for solving any NP-hard problem (e.g. TSP):

- Exactly but inefficiently:
  - ✓exhaustive search (brute-force, backtracking)
  - ✓smart search (branch-and-bound)
- Exactly, efficiently, but not universally:
  - ✓efficiently solvable special cases.
- Efficiently but inexactly:
  - ✓approximate algorithms,
  - ✓heuristics

TSP: solving

Definition: TSP is called ***metric*** (MTSP) iff the weight function $w\colon E \longrightarrow R_+$ is metric.

MTSP is an important special case of TSP.

An important special case of MTSP is Euclidean TSP (ETSP): vertices are points in R^n and w is Euclidean distance.

TSP: solving

Theorem 2: MTSP is also NP-hard.

Theorem 3: Even ETSP is NP-hard.

TSP: brute force

Brute-force (exhaustive search) approach:

- Exact
- Universal
- Easily adaptable
- Very time-consuming; prohibitive time complexity even for small ($n{\sim}100$) instances.

Principal idea:

1) Generate all feasible solutions.
2) For each feasible solutions calculate its cost (weight).
3) Select the best (minimum/maximum weight) feasible solution.

TSP: brute force

For TSP, feasible solutions are Hamiltonian cycles (paths).

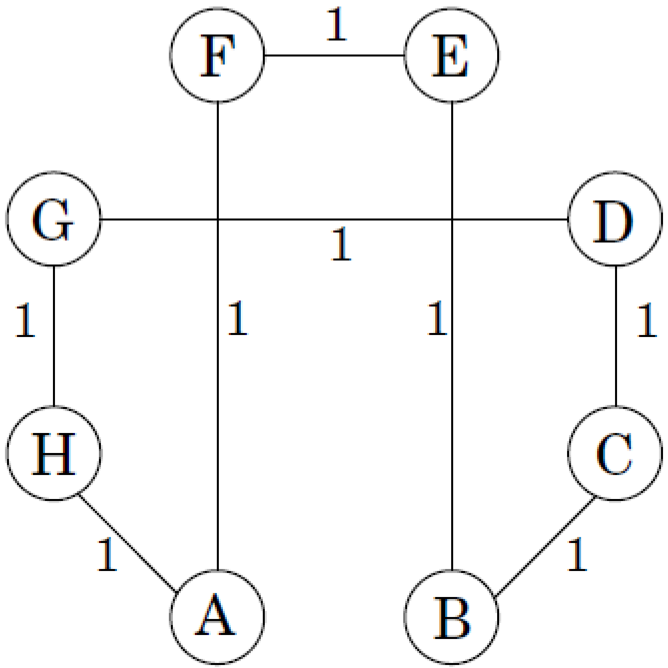
Possible representations of a Hamiltonian cycle (path):

- Vertex permutation: list the vertices in the order the cycle/path passes them.
- Edge sequence: list the edges in the order the cycle/path passes them.

Representing a Hamiltonian cycle/path as a vertex permutation is a bit easier, since we just need to check that all neighbors in the permutation are neighbors (adjacent vertices) in the graph (plus, for cycle: the last vertex is adjacent to the first one). For edge sequence representation checking validity is more complicated.

TSP: brute force

So, we need to generate all $n!$ possible vertex permutations.



In case of *cycle* we need to generate $(n-1)!$

AFEBCDGH

FEBCDGHA

EBCDGHAF

…

For an *undirected* graph: only $\frac{(n-1)!}{2}$:

AHGDCBEF

HGDCBEFA

…

TSP: brute force

Generating permutations [Lectures Notes on Algorithm Analysis and Computational Complexity (Fourth Edition) - Ian Parberry: http://ianparberry.com/books/free/license.html].

Problem: given positive integer n, generate all possible permutations of $1, \ldots, n$.

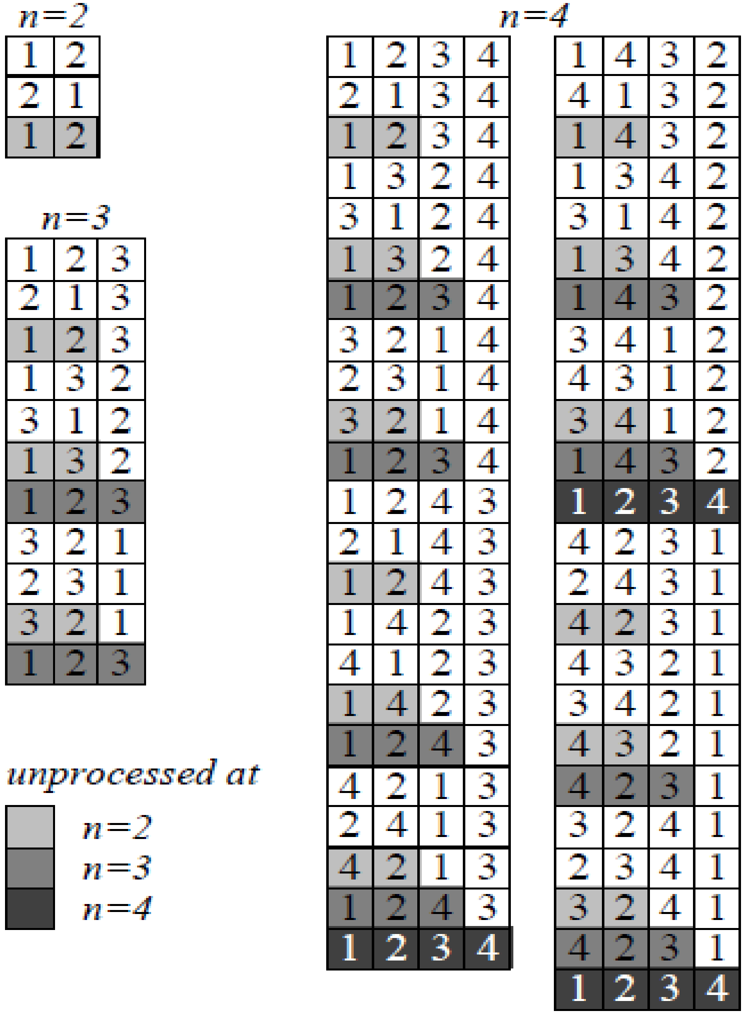
Idea of the generation algorithm:

- Create array `A[1..n]`.
- Initialization: `for each i: A[i] = i`.
- For each *k* successively swap $A[k]$ with $A[i]$ for $i = 1, \ldots, k$.

TSP: brute force

Call: `ProcessPermutations(A,k)`

```
Function ProcessPermutations(A,k)
if k = 1 then Process(A)
else
        ProcessPermutations(A, k-1);
        for i = k-1 downto 1 do
        {
          swap A[k] and A[i];
          ProcessPermutations(A, k-1);
          swap A[k] and A[i];
        }
```



n=2

1	2
2	1
1	2

n=3

1	2	3
2	1	3
1	2	3
1	3	2
3	1	2
1	3	2
1	2	3
3	2	1
2	3	1
3	2	1
1	2	3

n=4

1	2	3	4
2	1	3	4
1	2	3	4
1	3	2	4
3	1	2	4
1	3	2	4
1	2	3	4
3	2	1	4
2	3	1	4
3	2	1	4
1	2	3	4
1	2	4	3
2	1	4	3
1	2	4	3
1	4	2	3
4	1	2	3
1	4	2	3
1	2	4	3
4	2	1	3
2	4	1	3
4	2	1	3
1	2	4	3
1	2	3	4

1	4	3	2
4	1	3	2
1	4	3	2
1	3	4	2
3	1	4	2
1	3	4	2
1	4	3	2
3	4	1	2
4	3	1	2
3	4	1	2
1	4	3	2
1	2	3	4
4	2	3	1
2	4	3	1
4	2	3	1
4	3	2	1
3	4	2	1
4	3	2	1
4	2	3	1
3	2	4	1
2	3	4	1
3	2	4	1
4	2	3	1
1	2	3	4

unprocessed at

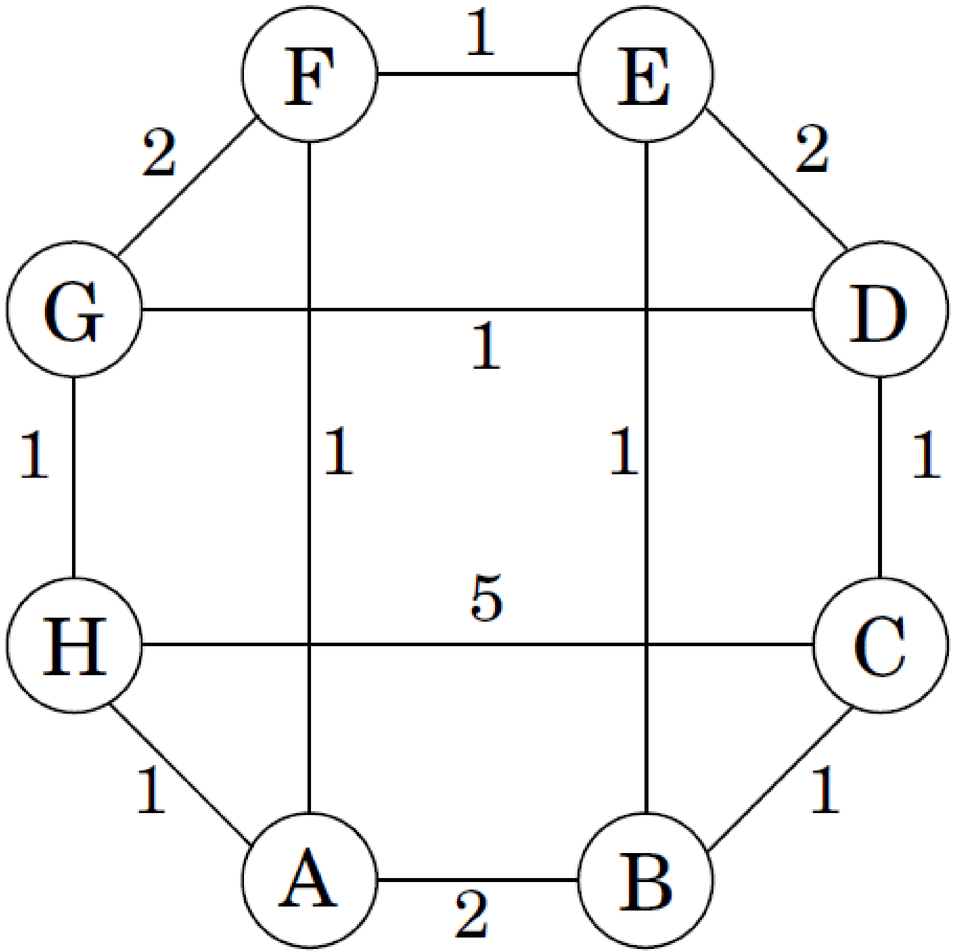
n=2
n=3
n=4

TSP: brute force

What the procedure `Process()` is for?

- Check whether the current permutation represents a feasible solution (Hamiltonian cycle).
- If it does, yield the current feasible solution (Hamiltonian cycle), calculate its weight and compare to the current champion.

TSP: brute force



Example:

- Generate 7! permutations, fix A as the 1st vertex.
- Permutation ‘aBCDEFGH’ is feasible, its weight is 11.
- Permutation ‘aBCDEFHG’ is not feasible because F and H are not adjacent in the graph.
- Permutation ‘aFEBCHGD’ is not feasible (doesn’t represent a Hamiltonian cycle) because D is not adjacent to A.