Advanced Algorithms

Winter term 2019/20

Lecture 1. Introduction & Held-Karp-algorithm for TSP

(slides by Joachim Spoerhase, Thomas van Dijk, & Alexander Wolff)

*Steven Chaplick & Alexander Wolff*  
*Chair for Computer Science I*
Advanced Algorithms

Learning goals: At the end of this lecture you will
– have an overview of advanced algorithmic topics (i.e., exact, approximate, geometric, and randomized computations), and advanced data structures,
– be able to analyze (and design algorithms for) new problems via the concepts of the lecture.

Requirements: – Big-Oh notation (Landau); e.g., $O(n \log n)$
– Some Algorithms & Data Structures
  (Balanced) binary search tree, priority queue
– Some Algorithmic Graph Theory
  Breadth-first search, Dijkstra’s algorithm
– Basic Theoretical Computer Science (P vs. NP)

Evaluation: • oral exam at the end of the semester
• 0,3 bonus for 50% on the exercises
What is this course about?

Many important (practical) problems are NP-hard
- Sacrifice optimality for speed
  - Heuristics (sim. Annealing, Tabu-Search)
  - Approximation Algorithms (Christofides-Algorithm)
- Optimal Solutions
  - Exact (exponential) time algorithms
  - Fine-grained analysis (parameterized) algorithms

Also, more on polytime solvable problems
- Geometric algorithms (sweep-line approach)
- More graph algorithms (shortest paths w/ neg. weights)
- Advanced data structures (splay trees)
- Randomized algorithms
Textbooks

F. Fomin & D. Kratsch: Exact Exponential Algorithms, Springer 2010
abbrev: EEA

abbrev: PA

abbrev: CLRS

abbrev: CG: A&A

This Lecture: Chapter 1
Background

- efficient vs. inefficient algorithms

\[ n^2 \sim 2^n \] polynomial vs. super-polynomial algorithms
Motivation: exact exponential algorithms

- can be “fast” for medium-sized instances
  $n^4 > 1.2^n$ for $n \leq 100$
  TSP solvable exactly for $n \leq 2000$ and specialized instances with $n \leq 85900$
  “hidden” constants in polynomial time algorithms: $2^{100} \cdot n > 2^n$ for $n \leq 100$
- theoretical interest
Typical Results

- Idea (simplified): find exact algorithms which are faster than *brute force* (trivial) approaches.
- Typically results for a (hypothetical) NP-hard problem

<table>
<thead>
<tr>
<th>Approach</th>
<th>Runtime in $O$-Notation</th>
<th>$O^*$-Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute-Force</td>
<td>$O(2^n)$</td>
<td>$O^*(2^n)$</td>
</tr>
<tr>
<td>Algorithm A</td>
<td>$O(1.5^n \cdot n)$</td>
<td>$O^*(1.5^n)$</td>
</tr>
<tr>
<td>Algorithm B</td>
<td>$O(1.4^n \cdot n^2)$</td>
<td>$O^*(1.4^n)$</td>
</tr>
</tbody>
</table>

$O(1.4^n \cdot n^2) \subsetneq O(1.5^n \cdot n) \subsetneq O(2^n)$

$\sim$ negligible polynomial factors (exp. dominates)

$f(n) \in O^*(g(n)) \iff \exists$ polynomial $p(n)$ w/ $f(n) \in O(g(n)p(n))$
Better Algorithms vs. Faster Hardware

Suppose an algorithm uses $a^n$ steps.

- For a fixed amount of time $t$, improving hardware by a constant factor $c$ only adds a (relative to $c$) constant to the max. size of solvable instances (in time $t$).
- Whereas reducing the base of the runtime to $b < a$ results in a multiplicative increase!

Why?

Hardware speedup: $a^{n'_0} = c \cdot a^{n_0} \Rightarrow n'_0 = \log_a c + n_0$

Base reduction: $b^{n'_0} = a^{n_0} \Rightarrow n'_0 = n_0 \cdot \log_b a$
Traveling Salesperson Problem (TSP)

Input
Complete directed graph $G = (V, E)$ with $n$ vertices and edge weights $c: E \to \mathbb{Q}_{\geq 0}$

Output
Hamiltonian cycle $(v_{\pi(1)}, \ldots, v_{\pi(n)}, v_{\pi(n+1)} = v_{\pi(1)})$ of $G$, of minimum weight $\sum_{i=1}^{n} c(v_i, v_{i+1})$, permutation $\pi$.

Brute-Force?

- Each tour is a permutation $\pi$ of the vertices.
- Pick a permutation with the smallest weight.

Runtime: $\Theta(n! \cdot n) = n \cdot 2^{\Theta(n \log n)}$
Bellman-Held-Karp-Algorithm

Technique: Dynamic Programming!

Reuse optimal substructures!

Select any starting vertex \( s \in V \).

For each \( S \subseteq V - s \) and \( v \in S \), let:

\[ \text{OPT}[S, v] = \text{length of a shortest } s-v\text{-path that visits precisely the vertices of } S \cup \{s\}. \]
Bellmann-Held-Karp-Algorithm

The base case: $S = \{v\}$, is easy: $\text{OPT}[\{v\}, v] = c(s, v)$.

When $|S| \geq 2$, we compute $\text{OPT}[S, v]$ recursively:

$$\text{OPT}[S, v] = \min \{ \text{OPT}[S - v, u] + c(u, v) \mid u \in S - v \}$$

After computing $\text{OPT}[S, v]$ for each $S \subseteq V - s$, the optimal solution is easily obtained as follows:

$$\text{OPT} = \min \{ \text{OPT}[V - s, v] + c(v, s) \mid v \in V - s \}$$
Pseudocode for the dynamic program

Algorithm Bellmann-Held-Karp($G, c$)

**foreach** $v \in V - s$ **do**

\[
\text{OPT}[\{v\}, v] = c(s, v)
\]

**for** $j = 2$ **to** $n - 1$ **do**

**foreach** $S \subseteq V - s$ with $|S| = j$ **do**

\[
\text{OPT}[S, v] = \min \{ \text{OPT}[S - v, u] + c(u, v) | u \in S - v \}
\]

**return** $\min \{ \text{OPT}[V - s, v] + c(v, s) | v \in V - s \}$

Runtime: the innermost loop executes $O(2^n \cdot n)$ iterations where each one takes $O(n)$ time. Thus, in total, we have $O(2^n \cdot n^2) = O^*(2^n)$. 

Space (memory) usage: $\Theta(2^n \cdot n)$

A shortest tour can be produced by backtracking the DP table (as usual).

Compare: $O^*(2^n)$ with $2^{O(n \log n)}$ for Brute-Force

Only use table-values for $j - 1$ to compute $j$, less space?
Maximum Independent Set

Input  Graph \( G = (V, E) \) with \( n \) vertices.

Output Maximum size *independent* set, i.e., a largest set \( U \subseteq V \), such that no pair of vertices in \( U \) are adjacent in \( G \).

Brute Force?
- Try all subsets of \( V \) \( \Rightarrow \) \( O(2^n \cdot n) \) runtime.

Algorithm NaiveMIS(\( G' \))
- if \( V = \emptyset \) then
  - return 0
- \( v \leftarrow \) arbitrary vertex in \( V(G) \)
- return \( \max\{1 + \text{NaiveMIS}(G - N(v) - \{v\}), \text{NaiveMIS}(G - \{v\})\} \)
Observations

**Lemma** Let $U$ be a *maximum* independent set in $G$. Then, for each vertex $v \in V$:

(i) $v \in U \implies N(v) \cap U = \emptyset$

(ii) $v \notin U \implies |N(v) \cap U| \geq 1$

Thus, $N[v] := N(v) \cup \{v\}$ contains some $y \in U$ and no other vertex of $N[y]$ is in $U$.
Smarter Branching-Algorithm

Algorithm MIS($G$)

if $V = \emptyset$ then
   return 0

$v \leftarrow$ vertex of minimum degree in $V(G)$

return $1 + \max\{\text{MIS}(G - N[y]) \mid y \in N[v]\}$

Correctness follows from the previous Lemma.

We will now prove a runtime of $O^*(3^{n/3}) = O^*(1.4423^n)$
Runtime

Execution corresponds to a search tree whose nodes are labeled with the input of the respective recursive call.

Let $B(n)$ be the maximum number of leaves of a search tree for a graph with $n$ vertices.

Search-tree has height $\leq n$, $\Rightarrow$ the algorithm’s runtime is $T(n) \in O^*(nB(n)) = O^*(B(n))$

Let’s consider an example run.

∅  ∅  ∅  ∅  ∅  ∅
Runtime Analysis

For a worst-case \( n \)-vertex graph \( G \) \((n \geq 1)\):

\[
B(n) \leq \sum_{y \in N[v]} B(n - (\text{deg}(y) + 1))
\]

\[
\leq (\text{deg}(v) + 1) \cdot B(n - (\text{deg}(v) + 1))
\]

where \( v \) is a minimum degree vertex of \( G \), and we note that \( B(n') \leq B(n) \) for any \( n' \leq n \).
Runtime Analysis (cont)

We proceed by induction to show $B(n) \leq 3^{n/3}$

Base case: $B(0) = 1 \leq 3^{0/3}$

Hypothesis: for $n \geq 1$, set $s = \deg(v) + 1$ in the above inequality

$$B(n) \leq s \cdot B(n - s) \leq s \cdot 3^{(n-s)/3} = \frac{s}{3^{s/3}} \cdot 3^{n/3} \leq 3^{n/3}$$

$B(n) \in O^*(3^{\sqrt{3}n}) \subset O^*(1.44225^n)$