Exact Exponential Algorithms for NP-Hard Optimization Problems

An Introduction

Martin Zachariasen

Department of Computer Science
University of Copenhagen

May 21, 2013

Outline

1. NP-Hard Optimization Problems
   - Exact and Heuristic Search Algorithms

Outline

1. NP-Hard Optimization Problems
   - Exact and Heuristic Search Algorithms
2. Exact Exponential Algorithms
   - Preliminaries
   - Dynamic Programming
   - Branch and Reduce

Outline

1. NP-Hard Optimization Problems
   - Exact and Heuristic Search Algorithms
2. Exact Exponential Algorithms
   - Preliminaries
   - Dynamic Programming
   - Branch and Reduce
3. Literature and Exercises

Outline

1. NP-Hard Optimization Problems
   - Exact and Heuristic Search Algorithms
2. Exact Exponential Algorithms
   - Preliminaries
   - Dynamic Programming
   - Branch and Reduce
3. Literature and Exercises

Travelling Salesman Problem (TSP)

Given a set $\{c_1, c_2, \ldots, c_n\}$ of cities and distances $d(c_i, c_j)$ between every pair c_i, c_j of cities, find a shortest tour through all the cities, i.e., a permutation π such that

$$\sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)}) \; + \; d(c_{\pi(n)}, c_{\pi(1)})$$

is minimized.

Equivalently:

Given an edge-weighted graph $G = (V, E)$, find an Hamiltonian cycle with minimum total weight in G.

Exact and Heuristic Search Algorithms

Exact algorithm: Systematically search through the whole search space

Heuristic algorithm: Only visit (hopefully) promising parts of the search space

Exact and Heuristic Search Algorithms

Exact algorithm: Systematically search through the whole search space

Heuristic algorithm: Only visit (hopefully) promising parts of the search space

Exact algorithm (systematic search) is often better suited when ...

- proofs of insolubility or optimality are required,
- time constraints are not critical,
- problem-specific knowledge can be exploited.

Heuristic algorithm (incomplete search) is often better suited when ...

- reasonably good solutions are required within a short time,
- parallel processing is used,
- problem-specific knowledge is rather limited.

Evaluation and Comparison of Search Algorithms

Two interesting parameters:

Evaluation and Comparison of Search Algorithms

Two interesting parameters:

1. Solution quality: How good are the solutions obtained? (Only really relevant for heuristic algorithms.)

Evaluation and Comparison of Search Algorithms

Two interesting parameters:

1. Solution quality: How good are the solutions obtained? (Only really relevant for heuristic algorithms.)
2. Running time: What is the running time of the algorithm as a function of problem instance size?

Evaluation and Comparison of Search Algorithms

Two interesting parameters:

1. Solution quality: How good are the solutions obtained? (Only really relevant for heuristic algorithms.)
2. Running time: What is the running time of the algorithm as a function of problem instance size?

Three possible approaches:

Evaluation and Comparison of Search Algorithms

Two interesting parameters:

1. Solution quality: How good are the solutions obtained? (Only really relevant for heuristic algorithms.)
2. Running time: What is the running time of the algorithm as a function of problem instance size?

Three possible approaches:

1. Worst-case analysis: Give a bound on the worst possible performance (quality and/or running time) of the algorithm.

Evaluation and Comparison of Search Algorithms

Two interesting parameters:

1. Solution quality: How good are the solutions obtained? (Only really relevant for heuristic algorithms.)
2. Running time: What is the running time of the algorithm as a function of problem instance size?

Three possible approaches:

1. Worst-case analysis: Give a bound on the worst possible performance (quality and/or running time) of the algorithm.
2. Average analysis: Assume that the instances (or their respective parameters) are distributed according to a probability function. Compute the average behaviour of the algorithm.

Evaluation and Comparison of Search Algorithms

Two interesting parameters:

1. Solution quality: How good are the solutions obtained? (Only really relevant for heuristic algorithms.)
2. Running time: What is the running time of the algorithm as a function of problem instance size?

Three possible approaches:

1. Worst-case analysis: Give a bound on the worst possible performance (quality and/or running time) of the algorithm.
2. Average analysis: Assume that the instances (or their respective parameters) are distributed according to a probability function. Compute the average behaviour of the algorithm.
3. Empirical/experimental analysis: Implement the algorithm and investigate how it performs on a suitable set of problem instances.

Outline

Big-O Notation for Exponential Algorithms

In this lecture we use a slightly modified definition of O-notation, where polynomially bounded factors are suppressed.

Examples:

$$n^3\, 2^n = O(2^n)$$
$$n^7 \log n\, 1.01^n = O(1.01^n)$$

Big-O Notation for Exponential Algorithms

In this lecture we use a slightly modified definition of O-notation, where polynomially bounded factors are suppressed.

Examples:

$$n^3\,2^n = O(2^n)$$
$$n^7 \log n\, 1.01^n = O(1.01^n)$$

Also, we assume that the objective value of a solution can be computed in polynomial time

$\Longrightarrow$ for search algorithms the O-bound is the number of visited/evaluated solutions in the worst case

Size of Problem Instance (n)

In this talk we simplify input size to a single parameter n:

- For graph problems n is the number of vertices (e.g. TSP)
- For set problems n is the number of elements (e.g. Knapsack)
- For Boolean formula problems n is the number of variables (e.g. SAT)

This definition of size makes it easier to measure the improvement over trivial brute-force algorithms.

Brute-Force Algorithms: Three Classes of Problems

Brute-force algorithms for three classes of problems which cover a wide range of optimization problems:

- Subset problems: $O(2^n)$ where n is the number of elements in the ground set
- Permutation problems: $O(n!) = O(2^{n \log n})$ where n is the size of the permutation vector
- Partition problems: $O(n^n)$ where n is the number of elements in the ground set

Examples of problems from each class?

Outline

1. NP-Hard Optimization Problems
   - Exact and Heuristic Search Algorithms
2. Exact Exponential Algorithms
   - Preliminaries
   - Dynamic Programming
   - Branch and Reduce
3. Literature and Exercises

Dynamic Programming for TSP

For subset $S \subseteq \{c_2, \dots, c_n\}$, $|S| \geq 1$, and $c_i \in S$, define:

$$\begin{aligned} \mathrm{OPT}[S, c_i] = {} & \text{minimum length of tour that starts in } c_1, \\ & \text{visits all cities in } S \text{ and ends in } c_i \end{aligned}$$

Dynamic Programming for TSP

For subset $S \subseteq \{c_2, \ldots, c_n\}$, $|S| \geq 1$, and $c_i \in S$, define:

$$\begin{aligned} \text{OPT}[S, c_i] = &\text{ minimum length of tour that starts in } c_1, \\ &\text{ visits all cities in } S \text{ and ends in } c_i \end{aligned}$$

$|S| = 1 \;:\; \text{OPT}[S, c_i] = d(c_1, c_i)$

Dynamic Programming for TSP

For subset $S \subseteq \{c_2, \ldots, c_n\}$, $|S| \geq 1$, and $c_i \in S$, define:

$$\begin{aligned} \mathrm{OPT}[S, c_i] = \ & \text{minimum length of tour that starts in } c_1, \\ & \text{visits all cities in } S \text{ and ends in } c_i \end{aligned}$$

$$|S| = 1 \ : \ \mathrm{OPT}[S, c_i] = d(c_1, c_i)$$

$$|S| > 1 \ : \ \mathrm{OPT}[S, c_i] = \min\{\mathrm{OPT}[S \setminus \{c_i\}, c_j] + d(c_j, c_i) \ : \ c_j \in S \setminus \{c_i\}\}$$

Dynamic Programming for TSP

For subset $S \subseteq \{c_2, \ldots, c_n\}$, $|S| \geq 1$, and $c_i \in S$, define:

$$\begin{aligned} \mathrm{OPT}[S, c_i] = {} & \text{minimum length of tour that starts in } c_1, \\ & \text{visits all cities in } S \text{ and ends in } c_i \end{aligned}$$

$$|S| = 1 \;:\; \mathrm{OPT}[S, c_i] = d(c_1, c_i)$$

$$|S| > 1 \;:\; \mathrm{OPT}[S, c_i] = \min\{\mathrm{OPT}[S \setminus \{c_i\}, c_j] + d(c_j, c_i) \;:\; c_j \in S \setminus \{c_i\}\}$$

Optimal TSP-tour:

$$\mathrm{OPT} = \min\{\mathrm{OPT}[\{c_2, \ldots, c_n\}, c_i] + d(c_i, c_1),\; i = 2, \ldots, n\}$$

Dynamic Programming for TSP: Running Time

The dynamic programming algorithm (by Bellman, Held and Karp) runs in $O(2^n)$ time.

Observation: For each set S of size k there are k^2 pairs of cities c_i, c_j in S.

Total time to compute OPT:

$$O(\sum_{k=1}^{n-1} \binom{n}{k} k^2) = O(n^2 \sum_{k=1}^{n-1} \binom{n}{k}) = O(n^2 2^n) = O(2^n)$$

Recall that a brute-force algorithm requires $O(n!) = O(2^{n \log n})$ time, so this is a clear improvement.

Outline

1. NP-Hard Optimization Problems
   - Exact and Heuristic Search Algorithms
2. Exact Exponential Algorithms
   - Preliminaries
   - Dynamic Programming
   - Branch and Reduce
3. Literature and Exercises

Branch and Reduce for Independent Set

Maximum Independent Set: Given an undirected graph $G = (V, E)$, find a vertex set $I \subseteq V$ of maximum cardinality such that no pair of vertices in I is adjacent in G.

Branch and Reduce for Independent Set

Maximum Independent Set: Given an undirected graph $G = (V, E)$, find a vertex set $I \subseteq V$ of maximum cardinality such that no pair of vertices in I is adjacent in G.

Key observations:

1. If vertex v is in I, then none of its neighbours are in I
2. If vertex v is not in I, then at least one of its neighbours is

Branch and Reduce for Independent Set

Maximum Independent Set: Given an undirected graph $G = (V, E)$, find a vertex set $I \subseteq V$ of maximum cardinality such that no pair of vertices in I is adjacent in G.

Key observations:

1. If vertex v is in I, then none of its neighbours are in I
2. If vertex v is not in I, then at least one of its neighbours is

That is: For any vertex v and any maximum independent set I, at least one vertex in the closed neighbourhood $N[v]$ is in I. Therefore:

$$\mathrm{OPT}(G) = 1 + \max\{OPT(G \setminus N[y]) \, : \, y \in N[v]\}$$

Branch and Reduce for Independent Set

Maximum Independent Set: Given an undirected graph $G = (V, E)$, find a vertex set $I \subseteq V$ of maximum cardinality such that no pair of vertices in I is adjacent in G.

Key observations:

1. If vertex v is in I, then none of its neighbours are in I
2. If vertex v is not in I, then at least one of its neighbours is

That is: For any vertex v and any maximum independent set I, at least one vertex in the closed neighbourhood $N[v]$ is in I. Therefore:

$$\mathrm{OPT}(G) = 1 + \max\{OPT(G \setminus N[y]) \,:\, y \in N[v]\}$$

Algorithm: Choose v of minimum degree and branch, i.e., solve subproblem $G \setminus N[y]$ for each $y \in N[v]$

Branch and Reduce for Independent Set

Running time of exact algorithm $=$ Number of nodes in search tree T

Branch and Reduce for Independent Set

Running time of exact algorithm = Number of nodes in search tree T

$T(n)$: largest number of nodes in search tree for $n = |V|$
$d(v)$: degree of node v in G
$v_1, v_2, \ldots, v_{d(v)}$: neighbours of v

Branch and Reduce for Independent Set

Running time of exact algorithm = Number of nodes in search tree T

$T(n)$: largest number of nodes in search tree for $n = |V|$
$d(v)$: degree of node v in G
$v_1, v_2, \ldots, v_{d(v)}$: neighbours of v

$$
\begin{aligned}
T(n) &\leq 1 + T(n - d(v) - 1) + \sum_{i=1}^{d(v)} T(n - d(v_i) - 1) \\
&\leq 1 + (d(v) + 1)T(n - d(v) - 1) \\
&\leq 1 + sT(n - s) \\
&\leq 1 + s + s^2 + s^3 + \ldots + s^{n/s} \\
&\leq \frac{s^{n/s+1} - 1}{s - 1} \\
&= O(s^{n/s}) = O(3^{n/3}) = O(1.4423^n)
\end{aligned}
$$

Branching Vector and Branching Factor

Consider a branching rule b that divides a problem of size n into r problems of size $n - t_1, n - t_2, \ldots, n - t_r$.

Then $\mathbf{b} = (t_1, t_2, \ldots, t_r)$ is the branching vector of branching rule b.

Branching Vector and Branching Factor

Consider a branching rule b that divides a problem of size n into r problems of size $n - t_1, n - t_2, \dots, n - t_r$.

Then $\mathbf{b} = (t_1, t_2, \dots, t_r)$ is the branching vector of branching rule b.

Linear recurrence

$$T(n) \leq T(n - t_1) + T(n - t_2) + \ldots + T(n - t_r)$$

has solution $T(n) = \alpha^n$ where the branching factor $\alpha = \tau(\mathbf{b})$ is the unique positive real root of

$$x^n - x^{n-t_1} - x^{n-t_1} - \ldots - x^{n-t_r} = 0$$

Properties of Binary Branching Vectors

Binary branching divides the problem into two subproblems in each step.

Two properties of binary branching vectors:

1. $\tau(k, k) \leq \tau(i, j)$ for all $i + j = 2k$

Properties of Binary Branching Vectors

Binary branching divides the problem into two subproblems in each step.

Two properties of binary branching vectors:

1. $\tau(k,k) \leq \tau(i,j)$ for all $i+j=2k$
2. $\tau(i,j) > \tau(i+\epsilon, j-\epsilon)$ for all $0 < i < j$ and $0 < \epsilon < (j-i)/2$

Properties of Binary Branching Vectors

Binary branching divides the problem into two subproblems in each step.

Two properties of binary branching vectors:

1. $\tau(k,k) \leq \tau(i,j)$ for all $i+j=2k$
2. $\tau(i,j) > \tau(i+\epsilon, j-\epsilon)$ for all $0 < i < j$ and $0 < \epsilon < (j-i)/2$

Examples:

$$\begin{aligned}
\tau(3,3) = \sqrt[3]{2} \quad & < \quad 1.2600 \\
\tau(2,4) = \tau(4,2) \quad & < \quad 1.2721 \\
\tau(1,5) = \tau(5,1) \quad & < \quad 1.3248
\end{aligned}$$

Addition of Branching Vectors

Branching vectors can be combined (or "added").

Addition of Branching Vectors

Branching vectors can be combined (or "added").

Example

For a given problem the branching vector is $(2, 2)$. But we also know that whenever we follow the left branch we immediatly do a $(2, 3)$ branch; when we follow the right branch we immediately do a $(1, 4)$ branch.

The combined branching vector is $(2 + 2, 2 + 3, 2 + 1, 2 + 4) = (4, 5, 3, 6)$.

Addition of Branching Vectors

Branching vectors can be combined (or "added").

Example

For a given problem the branching vector is $(2, 2)$. But we also know that whenever we follow the left branch we immediatly do a $(2, 3)$ branch; when we follow the right branch we immediately do a $(1, 4)$ branch.

The combined branching vector is $(2 + 2, 2 + 3, 2 + 1, 2 + 4) = (4, 5, 3, 6)$.

The resulting combined branching vectors may lead to better (or sharper) running time analysis.

Literature and Exercises

Literature

F. V. Fomin, D. Kratsch, *Exact Exponential Algorithms*, Texts in Theoretical Computer Science. An EATCS Series, Springer-Verlag Berlin Heidelberg, 2010. Chapter 1

Exercises

See Absalon.