

Algorithms and Networks

Exact Exponential Time Algorithms

Hans Bodlaender and Stefan Kratsch

March 28, 2011

Introduction
Dynamic programming
Branch and reduce
Local Search
Preprocessing
Summary



1

Introduction

- ▶ many interesting/important problems are NP-hard
- ▶ general consensus: $P \neq NP$
⇒ we do not expect to be able to solve NP-hard problems in polynomial time
- ▶ under the widely believed Exponential Time Hypothesis (ETH) one even expects certain problems to require exponential time, e.g., $\mathcal{O}(c^n)$.

ETH: Satisfiability of n -variable 3-CNF formulas (3-SAT) cannot be decided in subexponential worst case time, e.g., $\mathcal{O}(2^{o(n)})$.

Introduction
Dynamic programming
Branch and reduce
Local Search
Preprocessing
Summary



2

Recall: (Big)-O-Notation

- ▶ $f(n) = \mathcal{O}(g(n)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c$
- ▶ $f(n) = o(g(n)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$
- ▶ $f(n) = \mathcal{O}^*(g(n)) \Leftrightarrow$ there is a polynomial $p(n)$ s.t.:

$$\lim_{n \rightarrow \infty} \frac{f(n)}{p(n) \cdot g(n)} = c$$

e.g., $2^n \cdot n^3 = \mathcal{O}^*(2^n)$

\mathcal{O}^* -notation suppresses polynomial factors.

Introduction
Dynamic programming
Branch and reduce
Local Search
Preprocessing
Summary



3

Motivation

- ▶ every problem in NP can be solved by exhaustive search (e.g. trying all possible polynomial length proofs)
- ▶ some NP-complete problems can be solved significantly faster than that, e.g., $\mathcal{O}^*(1.2^n)$ instead of $\mathcal{O}^*(2^n)$
- ▶ giving a 10 times faster machine only allows us to increase the input size by a constant number of bits (if we want the same total time)
- ▶ improving the runtime from $\mathcal{O}^*(2^n)$ to $\mathcal{O}^*(1.41^n)$ allows to solve twice as large instances in the same time
- ▶ for some input sizes (low) exponential time may be faster than polynomial time, e.g., $\mathcal{O}(1.2^n n)$ vs. $\mathcal{O}(n^5)$

Introduction
Dynamic programming
Branch and reduce
Local Search
Preprocessing
Summary



4

Some basic types of problems

with typical runtimes for exhaustive search ($n := |U|$)

subset problems: find a subset of U that has a certain property
 $\mathcal{O}^*(2^n)$

permutation problems: order the elements of U to optimize some criterion
 $\mathcal{O}^*(n!)$

partition problems: partition U into subsets having some property
 $\mathcal{O}^*(n^n)$

Introduction

Dynamic programming

Branch and reduce

Local Search

Preprocessing

Summary



Universiteit Utrecht

5

Optimization vs. Decision vs. Search

note the difference between, e.g.:

- ▶ “is there a solution of value at least k ?” (decision)
- ▶ “find a solution of value at least k if one exists!” (search)
- ▶ “find a maximum value solution!” (optimization)

Take away: NP is defined for languages (i.e. decision) but NP-completeness for a decision problem gives NP-hardness for the optimization version. However, it is often hard to *prove* optimality, so you don't get membership in NP.

recall: NP-complete = NP-hard + membership in NP

Introduction

Dynamic programming

Branch and reduce

Local Search

Preprocessing

Summary



Universiteit Utrecht

6

Outline

Introduction

Dynamic programming

Branch and reduce

Local Search

Preprocessing

Summary

Introduction

Dynamic programming

Branch and reduce

Local Search

Preprocessing

Summary



Universiteit Utrecht

7

Outline

Introduction

Dynamic programming

Branch and reduce

Local Search

Preprocessing

Summary

Introduction

Dynamic programming

Branch and reduce

Local Search

Preprocessing

Summary



Universiteit Utrecht

8

Dynamic programming

- ▶ does the problem get easier with access to solved sub-problems?
- ▶ then solve the sub-problems in order of increasing size
- ▶ runtime is typically dominated by the number of sub-problems, e.g., $\mathcal{O}^*(2^n)$

note: you know this technique from efficient algorithms for problems in P

Introduction
Dynamic programming
Branch and reduce
Local Search
Preprocessing
Summary



Universiteit Utrecht

9

Hamiltonian Path (HP)

Hamiltonian Path

Input: A graph $G = (V, E)$ with n vertices and m edges.

Output: Does G contain a hamiltonian path, i.e., a path that visits each vertex exactly once?

- ▶ known to be NP-complete
- ▶ naive algorithm: try all $n!$ orderings of the vertices
- ▶ fastest algorithm: $\mathcal{O}^*(1.657^n)$ randomized algorithm for Hamiltonian Circuit by Bjoerklund (2010)

Introduction
Dynamic programming
Branch and reduce
Local Search
Preprocessing
Summary



Universiteit Utrecht

10

Dynamic programming: Hamiltonian Path

Given $G = (V, E)$, does it have a hamiltonian path?

- ▶ key: define appropriate subproblems on subsets of V
- ▶ when is there a hamiltonian path that ends in v ?
- ▶ if there is an edge $\{u, v\}$ and a path through all vertices but v that ends in u !

Subproblem: For any $S \subseteq V$ and $v \in S$, is there a path through all vertices of S that ends in v ?

Introduction
Dynamic programming
Branch and reduce
Local Search
Preprocessing
Summary



Universiteit Utrecht

11

Dynamic programming: Hamiltonian Path

$P(S, v) = 1 \Leftrightarrow$
exists path through all vertices of S that ends in v

- ▶ clearly $P(\{v\}, v) = 1$ for all $v \in V$
- ▶ compute $P(S, v)$ for all $S \subseteq V$ and $v \in S$, ordered by size of S :

$P(S, v) = 1 \Leftrightarrow \exists u \in S : P(S \setminus \{v\}, u) \wedge \{u, v\} \in E$

- ▶ there are 2^n subsets S of V and at most n choices for v
- ▶ each $P(S, v)$ computation takes only polynomial time
- ▶ total time $\mathcal{O}^*(2^n)$

Introduction
Dynamic programming
Branch and reduce
Local Search
Preprocessing
Summary



Universiteit Utrecht

12

Traveling Salesman Problem (TSP)

Traveling Salesman Problem

Input: A graph G with n vertices (cities) and weighted edges (connections).

Output: Find a tour (cycle), visiting each city exactly once, that has the minimum length.

- ▶ NP-complete
- ▶ Dynamic programming algorithm by Held & Karp (1962) runs in time $\mathcal{O}^*(2^n)$
- ▶ Bjoerklund's algorithm gives $\mathcal{O}^*(w^{1.657^n})$ where w is the sum of the weights

Introduction
Dynamic programming
Branch and reduce
Local Search
Preprocessing
Summary



Universiteit Utrecht

13

Dynamic programming: TSP (basic idea)

- ▶ can start at any city, say city 1 (take the vertices to be $1, \dots, n$), visit all other cities and return to 1
- ▶ let $w(i, j)$ denote the weight of edge $\{i, j\}$
- ▶ for each set $S \subseteq \{2, \dots, n\}$ and each $i \in S$ find the length of the shortest tour that starts in 1 visits each vertex of S exactly once and ends in i :

$$OPT(S, i)$$

- ▶ compute the values $OPT(S, i)$ for all sets $S \subseteq \{2, \dots, n\}$ (and each $i \in S$) in order of increasing cardinality
- ▶ (easy) how do you get the length of a shortest tour given that all values $OPT(S, i)$ are available?

Introduction
Dynamic programming
Branch and reduce
Local Search
Preprocessing
Summary



Universiteit Utrecht

14

Outline

Introduction

Dynamic programming

Branch and reduce

Local Search

Preprocessing

Summary

Introduction
Dynamic programming
Branch and reduce
Local Search
Preprocessing
Summary



Universiteit Utrecht

15

Branching algorithms

- ▶ given an input instance we may enumerate all candidate solutions (costly!), e.g., all assignments to a CNF-formula on n variables (there are 2^n)
- ▶ instead, fix only a part of the solutions and try to reduce the number of cases, e.g., ignore assignments that already falsify some clause
- ▶ make recursive calls corresponding to each choice (like one call for each chosen assignment to three variables)

All recursive calls need to be for simpler instances.

Introduction
Dynamic programming
Branch and reduce
Local Search
Preprocessing
Summary



Universiteit Utrecht

16

Branching tree

- ▶ the recursive calls form a tree, the branching tree
- ▶ typically: work in each call is polynomial time
- ▶ thus, runtime is $\mathcal{O}^*(\# \text{ of calls})$
- ▶ easy to see: if we always branch into at least two calls, then total $\#$ of calls is at most twice the $\#$ of leaves



3-SAT

3-SAT

Input: A formula \mathcal{F} in 3-CNF with n variables.

Output: Is \mathcal{F} satisfiable?

- ▶ NP-complete (2-SAT is in P)
- ▶ trivial algorithm in $\mathcal{O}^*(2^n)$ time by trying all assignments
- ▶ fast algorithms: $\mathcal{O}^*(1.48^n)$ deterministic by Dantsin et al. and $\mathcal{O}^*(1.33^n)$ randomized by Hofmeister et al. (2001)
- ▶ faster than $\mathcal{O}^*(2^n)$ by observing that a 3-clause forbids exactly 1 out of 8 possible assignments to its variables



A slightly better algorithm

3-SAT(\mathcal{F})

1. if no 3-clause then return the result of the poly-time algorithm for 2-SAT
2. else pick a 3-clause, say $(l_1 \vee l_2 \vee l_3)$
3. branch on:
 - 3.1 l_1 true
 - 3.2 l_2 true, l_1 false
 - 3.3 l_3 true, l_1, l_2 false

runtime: $T(n) \leq T(n-1) + T(n-2) + T(n-3)$
 $\Rightarrow T(n) = \mathcal{O}(1.84^n)$



Maximum Independent Set (IS)

Maximum Independent Set

Input: A graph $G = (V, E)$ with $n := |V|$.

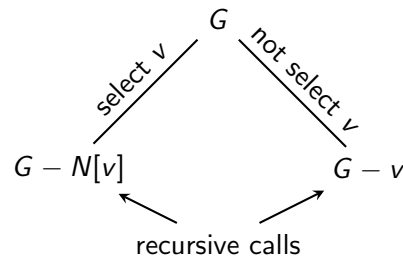
Output: A maximum cardinality independent set, i.e., a set $S \subseteq V$ of pairwise non-adjacent vertices.

- ▶ NP-hard (closely related to Minimum Vertex Cover)
- ▶ solvable in time $\mathcal{O}^*(2^n)$ by trying all subsets S of V
- ▶ fastest algorithm: $\mathcal{O}^*(1.1889^n)$ by Robson (2001)



Simple branching for independent set

Given a graph $G = (V, E)$, focus on any $v \in V$:



afterwards: compare best solution for $G - v$ to one plus best solution for $G - N[v]$ and return the larger one

Observation: If $v \in S$ then $S \cap N(v) = \emptyset$.



Branching algorithm

Algorithm $MIS(G = (V, E))$

1. if maximum degree at most 2 then solve in polynomial time and return max-IS
2. pick a vertex of maximum degree (≥ 3)
3. $S_{\bar{v}} := MIS(G - v)$
4. $S_v := MIS(G - N[v]) \cup \{v\}$
5. return the larger set of S_v and $S_{\bar{v}}$

$T(n) :=$ worst-case time of MIS on any graph on n vertices

$$\begin{aligned} T(n) &\leq T(n-1) + T(n - (\deg(v) + 1)) \\ &\leq T(n-1) + T(n-4) \end{aligned}$$

$$\Rightarrow T(n) = \mathcal{O}^*(1.38^n)$$



How to solve recurrences?

- ▶ given, say, $T(n) = \alpha T(n - \beta) + \gamma T(n - \delta)$
- ▶ assume that $T(n) = c^n$ for some (yet) unknown c :

$$\begin{aligned} c^n &= \alpha c^{n-\beta} + \gamma c^{n-\delta} \\ c^n - \alpha c^{n-\beta} - \gamma c^{n-\delta} &= 0 \end{aligned}$$

dividing by $c^{n-\delta}$ assuming $\delta \geq \beta$:

$$c^\delta - \alpha c^{\delta-\beta} - \gamma = 0$$

- ▶ use e.g. matlab to find the roots (zeros) of this polynomial
- ▶ the largest positive root is the value that we need (it's known to be unique)



Solving recurrences: example

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) \\ c^n &= c^{n-1} + c^{n-2} \end{aligned}$$

$$\Rightarrow c^2 - c - 1 = 0$$

roots (zeros) are: $\frac{1+\sqrt{5}}{2}$ and $\frac{1-\sqrt{5}}{2}$

our result: $T(n) = \mathcal{O}^*\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$

$\frac{1+\sqrt{5}}{2}$ is called the branching number

(1, 2) is the branching vector (one number for each recursive call, indicating how much smaller the subproblems are)



How to improve the algorithm?

current worst case is to branch on a vertex v of degree three:

- ▶ branching on deg 3 gives $T(n) \leq T(n-1) + T(n-4)$
⇒ branching number ~ 1.380
- ▶ branching on $\text{deg} \geq 4$ gives $T(n) \leq T(n-1) + T(n-5)$
⇒ branching number ~ 1.324
- ▶ we would need to find a better rule for vertices of degree 3
- ▶ (without proof) using reduction rules for
 - vertices of degree ≤ 2 and
 - small connected components

one can get a branching for deg-3-vertices with branching number < 1.324

Introduction
Dynamic programming
Branch and reduce
Local Search
Preprocessing
Summary



Universiteit Utrecht

25

Faster Algorithm

Algorithm $MIS''(G = (V, E))$

1. if maximum degree at most 2 then solve in polynomial time and return max-IS
2. pick a vertex of maximum degree (≥ 3)
3. if $\text{deg}(v) = 3$ use the new branching rule for deg 3
4. else branch as before (but have $\text{deg}(v) \geq 4$)

- ▶ branching for degree 3: < 1.324 (possibly significantly less)
- ▶ standard branching for degree ≥ 4 : 1.324
- ▶ in the worst-case we always use the worst branching
⇒ $\mathcal{O}^*(1.324^n)$

Introduction
Dynamic programming
Branch and reduce
Local Search
Preprocessing
Summary



Universiteit Utrecht

26

Outline

Introduction

Dynamic programming

Branch and reduce

Local Search

Preprocessing

Summary

Introduction
Dynamic programming
Branch and reduce
Local Search
Preprocessing
Summary



Universiteit Utrecht

27

Recall: 3-SAT

3-SAT

Input: A formula \mathcal{F} in 3-CNF with n variables.

Output: Is \mathcal{F} satisfiable?

- ▶ NP-complete (2-SAT is in P)
- ▶ trivial algorithm in $\mathcal{O}^*(2^n)$ time by trying all assignments
- ▶ we had $\mathcal{O}^*(1.84^n)$ by carefully branching on 3-clauses

Introduction
Dynamic programming
Branch and reduce
Local Search
Preprocessing
Summary



Universiteit Utrecht

28

Using local search for 3-SAT

- ▶ hamming distance: # of variables in which two assignments differ
- ▶ any assignment (to n variables) has hamming distance at most $n/2$ to “all false” or “all true” assignment
- ▶ how fast can we determine whether there is a solution at distance at most d ?

Introduction
Dynamic programming
Branch and reduce
Local Search
Preprocessing
Summary



Universiteit Utrecht

29

Using local search for 3-SAT

given an assignment to variables of \mathcal{F} , is there an assignment at distance at most d ?

1. if assignment satisfies \mathcal{F} then return YES
2. else pick an unsatisfied clause
3. branch on picking a variable where we deviate from the given assignment (3 choices)
4. at most 3 recursive calls with distance $d - 1$

runtime: $\mathcal{O}^*(3^d)$

Runtime $\mathcal{O}^*(3^{\frac{n}{2}}) = \mathcal{O}^*(1.73^n)$ for 3-SAT by searching for assignment at distance at most $\frac{n}{2}$ from “all false” and “all true”.

Introduction
Dynamic programming
Branch and reduce
Local Search
Preprocessing
Summary



Universiteit Utrecht

30

Local search for 3-SAT

- ▶ exploring the “balls” of radius $\frac{n}{2}$ around the all true and the all one false assignments we got runtime $\mathcal{O}^*(3^{\frac{n}{2}})$
- ▶ faster algorithms use:
 - smaller balls
 - a more clever set of points (assignments) to explore around (*covering codes*)
 - randomized choice of starting points
 - randomized exploration of the balls (*random walks*)

Introduction
Dynamic programming
Branch and reduce
Local Search
Preprocessing
Summary



Universiteit Utrecht

31

Outline

Introduction
Dynamic programming
Branch and reduce
Local Search
Preprocessing
Summary

Introduction
Dynamic programming
Branch and reduce
Local Search
Preprocessing
Summary



Universiteit Utrecht

32

Subset Sum

Subset Sum

Input: n integers $a_1, \dots, a_n \in \mathbb{N}$ and integer $S \in \mathbb{N}$.

Output: Does some subset of the numbers sum to S ?

- ▶ NP-complete
- ▶ trivial algorithm: try all subsets: $\mathcal{O}^*(2^n)$ time
- ▶ in P when the numbers are encoded in unary (dynamic programming)

unary encoding: value a_i is encoded as $\overbrace{1 \dots 1}^{a_i}$
equivalent: each number has value at most n^c



Subset Sum

- ▶ split input into $a_1, \dots, a_{n/2}$ and $a_{n/2+1}, \dots, a_n$
- ▶ for both sets: compute all $2^{n/2}$ possible sums
⇒ sets S_1 and S_2
- ▶ make two sorted lists L_1 and L_2 containing the possible sums for both sets (sorting takes $\mathcal{O}(2^{n/2} \log(2^{n/2})) = \mathcal{O}^*(2^{n/2})$)
- ▶ try to find a combination of $s_1 \in L_1$ and $s_2 \in L_2$ with $S = s_1 + s_2$:
 - traverse L_1 from small to big
 - traverse L_2 from big to small

runtime: $\mathcal{O}^*(2^{n/2})$



Outline

Introduction

Dynamic programming

Branch and reduce

Local Search

Preprocessing

Summary



Dynamic programming

- ▶ define appropriate subproblems, typically as many as $\mathcal{O}^*(c^n)$
- ▶ solve the subproblems in order of increasing size (base case should be trivial)
- ▶ typically polynomial runtime to solve a single subproblem by using solutions to all smaller ones

Examples: Hamiltonian Path, Traveling Salesman Problem, (Chromatic Number)



Branch and reduce

- ▶ instead of exhaustive search, only exhaustively consider all possibilities for some part of the solution (e.g. variables)
- ▶ discard all choices that are infeasible or dominated by better ones
- ▶ make recursive calls for the remaining ones
- ▶ use reduction rules to simplify each instance using the already made choices
- ▶ get a branching tree corresponding to the recursive calls
- ▶ its size (and therefore the runtime) can be bounded using recurrences (e.g. $T(n) \leq T(n-1) + T(n-2)$)

Examples: 3-SAT, Independent Set, (Dominating Set), (Set Cover)



Local Search and Preprocessing

Local Search :

- ▶ step-by-step modify infeasible solutions towards feasible ones
- ▶ possibly many starting points
- ▶ also randomized exploration and/or randomized starting points

Preprocessing :

- ▶ known from polynomial time algorithms (e.g. sorted vs. unsorted inputs)
- ▶ preprocessing exponentially large intermediate results can give significant time savings



Inclusion-Exclusion (very high level description)

- ▶ solving decision and counting problems by counting both feasible and infeasible solutions
- ▶ over-counting is handled by inclusion-exclusion formula
- ▶ see for example: Bjoerklund et al. "Set Partitioning via Inclusion-Exclusion." (SIAM Journal 2009)

(have not covered this; it's a pointer for interested students)



Bonus slide

Fibonacci numbers (recursive):

$$f_0 = 0$$

$$f_1 = 1$$

$$f_n = f_{n-1} + f_{n-2}$$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, ...

Fibonacci numbers (closed form):

$$f_n = \frac{1}{\sqrt{5}} \cdot \left[\left(\frac{1 + \sqrt{5}}{2} \right)^n - \left(\frac{1 - \sqrt{5}}{2} \right)^n \right]$$

