

algorithms for NP-hard problems

Coping with NP-completeness

Q. Suppose I need to solve an **NP**-hard problem. What should I do?

A. Sacrifice one of three desired features.

- i. Solve **arbitrary instances** of the problem.
- ii. Solve problem to **optimality**.
- iii. Solve problem in **polynomial time**.

Coping strategies.

- i. Design algorithms **for special cases** of the problem.
- ii. Design **approximation algorithms** or **heuristics**.
- iii. Design algorithms that may take **exponential time**.

using greedy,
dynamic programming,
divide-and-conquer, and
network flow algorithms!



Independent set on trees

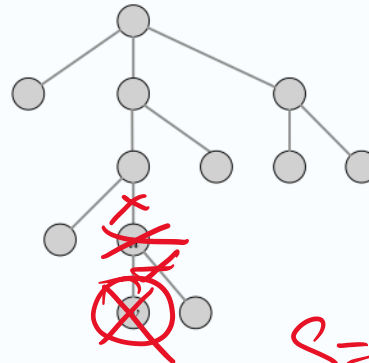
Independent set on trees. Given a **tree**, find a max-cardinality subset of nodes such that no two are adjacent.

Fact. A tree has at least one node that is a leaf (degree = 1).

Key observation. If node v is a leaf, there exists a max-cardinality independent set containing v .

Pf. [exchange argument]

- Consider a max-cardinality independent set S .
- If $v \in S$, we're done.
- Otherwise, let (u, v) denote the lone edge incident to v .
 - if $u \notin S$ and $v \notin S$, then $S \cup \{v\}$ is independent $\Rightarrow S$ not maximum
 - if $u \in S$ and $v \notin S$, then $S \cup \{v\} - \{u\}$ is independent ■



4

Independent set on trees: greedy algorithm

Theorem. The greedy algorithm finds a max-cardinality independent set in forests (and hence trees).



Pf. Correctness follows from the previous key observation. ■

INDEPENDENT-SET-IN-A-FOREST(F)

$S \leftarrow \emptyset$.

WHILE (F has at least 1 edge)

Let v be a leaf node and let (u, v) be the lone edge incident to v .

$S \leftarrow S \cup \{v\}$.

$F \leftarrow F - \{u, v\}$. ← delete both u and v (including all incident edges)

RETURN $S \cup \{ \text{nodes remaining in } F \}$.

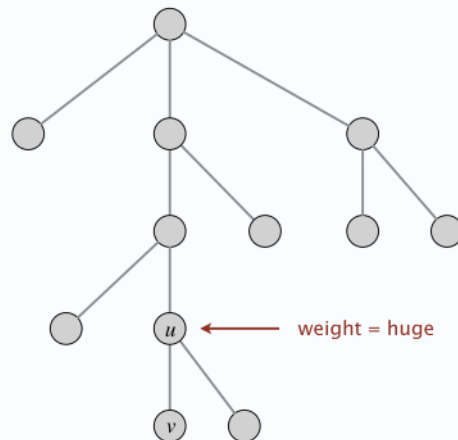
Remark. Can implement in $O(n)$ time by maintaining nodes of degree 1.

5

Weighted independent set on trees

Weighted independent set on trees. Given a tree and node weights $w_v \geq 0$, find an independent set S that maximizes $\sum_{v \in S} w_v$.

Greedy algorithm can fail spectacularly.



7

Weighted independent set on trees

Weighted independent set on trees. Given a tree and node weights $w_v \geq 0$, find an independent set S that maximizes $\sum_{v \in S} w_v$.

Dynamic-programming solution. Root tree at some node, say r .

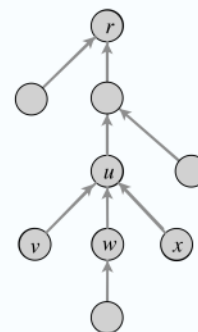
- $OPT_{in}(u)$ = max-weight IS in subtree rooted at u , containing u .
- $OPT_{out}(u)$ = max-weight IS in subtree rooted at u , not containing u .
- Goal: $\max \{ OPT_{in}(r), OPT_{out}(r) \}$.

Bellman equation.

$$OPT_{in}(u) = w_u + \sum_{v \in \text{children}(u)} OPT_{out}(v)$$

$$OPT_{out}(u) = \sum_{v \in \text{children}(u)} \max \{ OPT_{in}(v), OPT_{out}(v) \}$$

overlapping
subproblems



$\text{children}(u) = \{ v, w, x \}$

8

Weighted independent set on trees: dynamic-programming algorithm

Theorem. The DP algorithm computes max weight of an independent set in a tree in $O(n)$ time.

can also find independent set itself
(not just value)

WEIGHTED-INDEPENDENT-SET-IN-A-TREE (T)

Root the tree T at any node r .

$S \leftarrow \emptyset$.

FOREACH (node u of T in postorder/topological order)

IF (u is a leaf node)

$$M_{in}[u] = w_u.$$

$$M_{out}[u] = 0.$$

ensures a node is processed
after all of its descendants

ELSE

$$M_{in}[u] = w_u + \sum_{v \in \text{children}(u)} M_{out}[v].$$

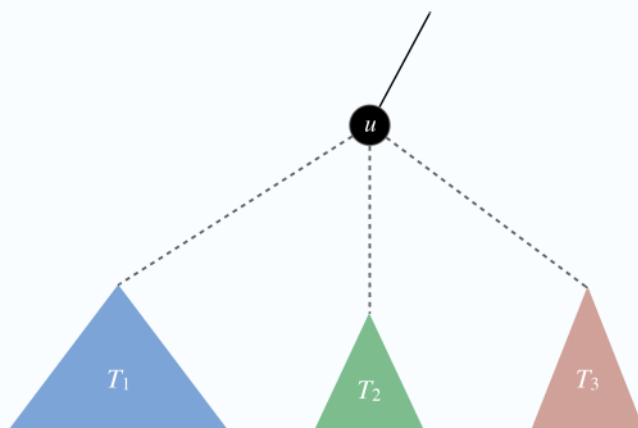
$$M_{out}[u] = \sum_{v \in \text{children}(u)} \max \{ M_{in}[v], M_{out}[v] \}.$$

RETURN $\max \{ M_{in}[r], M_{out}[r] \}.$

10

NP-hard problems on trees: context

Independent set on trees. Tractable because we can find a node that breaks the communication among the subproblems in different subtrees.



11

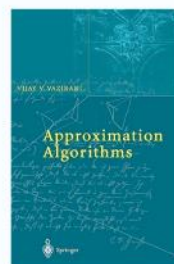
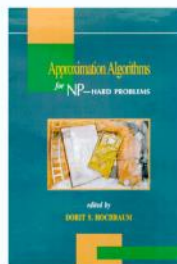
Approximation algorithms

ρ -approximation algorithm.

- Runs in polynomial time.
- Applies to arbitrary instances of the problem.
- Guaranteed to find a solution within ratio ρ of true optimum.

Ex. Given a graph G , can find a vertex cover that uses $\leq 2 \text{OPT}(G)$ vertices in $O(m + n)$ time.

Challenge. Need to prove a solution's value is close to optimum value, without even knowing what optimum value is!

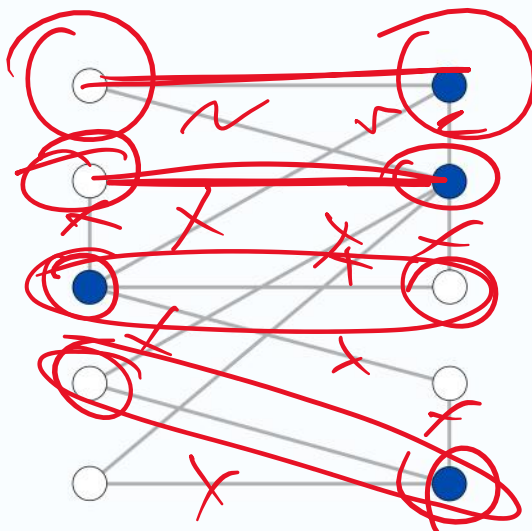


29

Vertex cover

VERTEX-COVER. Given a graph $G = (V, E)$, find a min-size vertex cover.

↑
for each edge $(u, v) \in E$:
either $u \in S$, $v \in S$, or both



● vertex cover of size 4

30

Vertex cover: greedy algorithm

VERTEX-COVER. Given a graph $G = (V, E)$, find a min-size vertex cover.



GREEDY-VERTEX-COVER(G)

$S \leftarrow \emptyset$.

$E' \leftarrow E$.

WHILE ($E' \neq \emptyset$)

Let $(u, v) \in E'$ be an arbitrary edge.

$M \leftarrow M \cup \{(u, v)\}$. $\leftarrow M$ is a matching

$S \leftarrow S \cup \{u\} \cup \{v\}$.

Delete from E' all edges incident to either u or v .

RETURN S .

every vertex cover must take at least one of these; we take both

$|S|$

$$|M| = \frac{|S|}{2}$$

$$|S_{\text{opt}}| \geq |M| = \frac{|S|}{2}$$
$$|S| \leq 2 \cdot |S_{\text{opt}}|$$

Running time. Can be implemented in $O(m + n)$ time.

31

Vertex cover inapproximability

Theorem. [Dinur-Safra 2004] If $\mathbf{P} \neq \mathbf{NP}$, then no ρ -approximation for VERTEX-COVER for any $\rho < 1.3606$.

On the Hardness of Approximating Minimum Vertex Cover

Irit Dinur*

Samuel Safra†

May 26, 2004

Abstract

We prove the Minimum Vertex Cover problem to be NP-hard to approximate to within a factor of 1.3606, extending on previous PCP and hardness of approximation technique. To that end, one needs to develop a new proof framework, and borrow and extend ideas from several fields.



Open research problem. Close the gap.

Conjecture. no ρ -approximation for VERTEX-COVER for any $\rho < 2$.

34

Weighted Vertex Cover: 2-Approximation via LP

Given: Graph $G = (V, E)$
weights $w_v \geq 0, \forall v \in V$

Find: Vertex Cover S s.t.
 $w(S) := \sum_{v \in S} w_v$
is minimized.

LP-formulation

$$\min \sum_{v \in V} w_v \cdot x_v$$

$$x_u + x_v \geq 1 \quad \forall (u, v) \in E$$

$$0 \leq x_u \leq 1 \quad \forall u \in V$$

($x_u \in \{0, 1\}$ yield Integer LP (ILP)
that corresponds to VC exactly)

x^* : solution to the LP above

Note: $w(x^*) \leq \min \text{Weighted VC}$

Need to get a VC from x^* .

Rounding to get $x_u \in \{0, 1\}$:

$$c_n \quad \text{if} \quad x^* < \frac{1}{2}$$

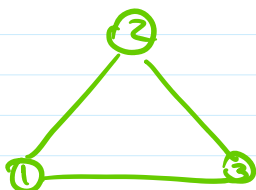
$$x_u = \begin{cases} 0 & \text{if } x_u^* < \frac{1}{2} \\ 1 & \text{if } x_u^* \geq \frac{1}{2} \end{cases}$$

Claim 1: $S = \{u \mid x_u = 1\}$ is a VC.

Proof: \forall edge $(u, v) \in E$, $x_u^* + x_v^* \geq 1$
 $\Rightarrow x_u^* \geq \frac{1}{2}$ or $x_v^* \geq \frac{1}{2}$ (or both)
 $\Rightarrow x_u = 1$ or $x_v = 1$ (or both)
 $\Rightarrow u \in S$ or $v \in S$. \square

Claim 2: $w(S) \leq 2 \cdot w(x^*)$
 $\leq 2 \cdot \text{Min Weight VC}$

Proof: $w(x^*) = \sum_{u \in V} w_u \cdot x_u^*$
 $\geq \sum_{u \in S} w_u \cdot x_u^*$
 $\geq \frac{1}{2} \cdot \sum_{u \in S} w_u$
 $= \frac{1}{2} \cdot w(S)$. \square



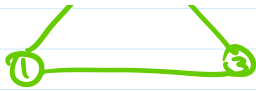
$$x_1 = 1$$

$$x_2 = 1$$

LP

$$x_1^* = \frac{1}{2}$$

$$x_2^* = \frac{1}{2}$$



$$x_2 = 1$$

$$x_3 = 0$$

$$x_2^* = \frac{1}{2}$$

$$x_3^* = \frac{1}{2}$$

Knapsack problem

Knapsack problem.

- Given n objects and a knapsack.
- Item i has value $v_i > 0$ and weighs $w_i > 0$. ← we assume $w_i \leq W$ for each i
- Knapsack has weight limit W .
- Goal: fill knapsack so as to maximize total value.

Ex: $\{3, 4\}$ has value 40.

item	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

original instance ($W = 11$)

Knapsack is NP-complete

SUBSET-SUM. Given a set X , values $u_i \geq 0$, and an integer U , is there a subset $S \subseteq X$ whose elements sum to exactly U ?

KNAPSACK. Given a set X , weights $w_i \geq 0$, values $v_i \geq 0$, a weight limit W , and a target value V , is there a subset $S \subseteq X$ such that:

$$\sum_{i \in S} w_i \leq W$$

$$\sum_{i \in S} v_i \leq V$$

Theorem. SUBSET-SUM \leq_P KNAPSACK.

Pf. Given instance (u_1, \dots, u_n, U) of SUBSET-SUM, create KNAPSACK instance:

$$\begin{array}{ll} v_i = w_i = u_i & \sum_{i \in S} u_i \leq U \\ V = W = U & \sum_{i \in S} u_i \geq U \end{array}$$

37

Knapsack problem: dynamic programming I

Def. $OPT(i, w)$ = max value subset of items $1, \dots, i$ with **weight** limit w .

Case 1. OPT does not select item i .

- OPT selects best of $1, \dots, i-1$ using up to weight limit w .

Case 2. OPT selects item i .

- New weight limit = $w - w_i$.
- OPT selects best of $1, \dots, i-1$ using up to weight limit $w - w_i$.

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max\{OPT(i-1, w), v_i + OPT(i-1, w - w_i)\} & \text{otherwise} \end{cases}$$

Theorem. Computes the optimal value in $O(nW)$ time.

- Not polynomial in input size.
- Polynomial in input size if weights are small integers.

38

Knapsack problem: dynamic programming II

Def. $OPT(i, v)$ = min weight of a knapsack for which we can obtain a solution of value $\geq v$ using a subset of items $1, \dots, i$.

Note. Optimal value is the largest value v such that $OPT(n, v) \leq W$.

Case 1. OPT does not select item i .

- OPT selects best of $1, \dots, i-1$ that achieves value $\geq v$.

Case 2. OPT selects item i .

- Consumes weight w_i , need to achieve value $\geq v - v_i$.
- OPT selects best of $1, \dots, i-1$ that achieves value $\geq v - v_i$.

$$OPT(i, v) = \begin{cases} 0 & \text{if } v \leq 0 \\ \infty & \text{if } i = 0 \text{ and } v > 0 \\ \min \{OPT(i-1, v), w_i + OPT(i-1, v - v_i)\} & \text{otherwise} \end{cases}$$

39

Knapsack problem: dynamic programming II

Theorem. Dynamic programming algorithm II computes the optimal value in $O(n^2 v_{\max})$ time, where v_{\max} is the maximum of any value.

Pf.

- The optimal value $V^* \leq n v_{\max}$.
- There is one subproblem for each item and for each value $v \leq V^*$.
- It takes $O(1)$ time per subproblem. ■

Remark 1. Not polynomial in input size!

Remark 2. Polynomial time if values are small integers.

40

Knapsack problem: poly-time approximation scheme

Intuition for approximation algorithm.

- Round all values up to lie in smaller range.
- Run dynamic programming algorithm II on rounded/scaled instance.
- Return optimal items in rounded instance.

item	value	weight
1	934221	1
2	5956342	2
3	17810013	5
4	21217800	6
5	27343199	7

original instance (W = 11)

item	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

rounded instance (W = 11)

41

Knapsack problem: poly-time approximation scheme

Round up all values:

- $0 < \epsilon \leq 1$ = precision parameter.
- v_{\max} = largest value in original instance.
- θ = scaling factor = $\epsilon v_{\max} / 2n$.

$$\bar{v}_i = \left\lceil \frac{v_i}{\theta} \right\rceil \theta, \quad \hat{v}_i = \left\lceil \frac{v_i}{\theta} \right\rceil$$

Observation. Optimal solutions to problem with \bar{v} are equivalent to optimal solutions to problem with \hat{v} .

Intuition. \bar{v} close to v so optimal solution using \bar{v} is nearly optimal; \hat{v} small and integral so dynamic programming algorithm II is fast.

42

Knapsack problem: poly-time approximation scheme

Theorem. If S is solution found by rounding algorithm and S^* is any other feasible solution, then $(1 + \epsilon) \sum_{i \in S} v_i \geq \sum_{i \in S^*} v_i$

Pf. Let S^* be any feasible solution satisfying weight constraint.

$$\begin{aligned}
 \sum_{i \in S^*} v_i &\leq \sum_{i \in S^*} \bar{v}_i && \text{always round up} \\
 &\leq \sum_{i \in S} \bar{v}_i && \text{solve rounded instance optimally} \\
 &\leq \sum_{i \in S} (v_i + \theta) && \text{never round up by more than } \theta \\
 &\leq \sum_{i \in S} v_i + n\theta && |S| \leq n \\
 &= \sum_{i \in S} v_i + \frac{1}{2} \epsilon v_{\max} && \theta = \epsilon v_{\max} / 2n \\
 &\leq (1 + \epsilon) \sum_{i \in S} v_i && v_{\max} \leq 2 \sum_{i \in S} v_i
 \end{aligned}$$

subset containing only the item of largest value

choosing $S^* = \{ \max \}$

$$v_{\max} \leq \sum_{i \in S} v_i + \frac{1}{2} \epsilon v_{\max}$$

$$\text{thus } v_{\max} \leq \sum_{i \in S} v_i + \frac{1}{2} v_{\max}$$

$$v_{\max} \leq 2 \sum_{i \in S} v_i$$

43

Knapsack problem: poly-time approximation scheme

Theorem. For any $\epsilon > 0$, the rounding algorithm computes a feasible solution whose value is within a $(1 + \epsilon)$ factor of the optimum in $O(n^3 / \epsilon)$ time.

Pf.

- We have already proved the accuracy bound.
- Dynamic program II running time is $O(n^2 \hat{v}_{\max})$, where

$$\hat{v}_{\max} = \left\lceil \frac{v_{\max}}{\theta} \right\rceil = \left\lceil \frac{2n}{\epsilon} \right\rceil$$

44

Set Cover : Greedy Heuristic

Set Cover : Greedy Heuristic

Given: Set U , $|U|=n$
subsets $S_1, \dots, S_m \subseteq U$.

Find: a collection of subsets
 $S_{i_1}, S_{i_2}, \dots, S_{i_k}$
s.t. $S_{i_1} \cup S_{i_2} \cup \dots \cup S_{i_k} = U$
& their number k is min.

Greedy-Set-Cover

$R = U$; $C = \emptyset$

while $R \neq \emptyset$
select set $S_i \notin C$
that maximizes $|S_i \cap R|$;

$C = C \cup \{i\}$;

$R = R - S_i$

endwhile

return C

Claim 1: C returned by the algo
is a Set Cover.

Proof: At the end, $R = \emptyset$ \square

Claim 2: $|C| \leq (\ln V) \cdot \text{OPT} + 1$

Proof: Let $K = \text{OPT}$ (the min size of the Set Cover).

Let S_1, \dots, S_K be the min Set Cover,
i.e., $S_1 \cup \dots \cup S_K = V$. (*)

After each iteration, R loses at least $\frac{1}{K}$ of its size.

For the first iteration, (*) \Rightarrow

$\exists S_i, 1 \leq i \leq K$, s.t. $|S_i| \geq \frac{1}{K} \cdot |V|$.

The greedy algo will select S_j with $|S_j| \geq |S_i|$.

[Same is true for each subsequent iteration. **Prove it!**]

Greedy algo stops after iteration t when

$$|V| \cdot \left(1 - \frac{1}{K}\right)^t$$

$$1 - \frac{1}{K}$$

$$-\frac{t}{K}$$

$$1$$

$$1+x \leq e^x$$

$$1 + x \leq e^x$$

$$\leq |V| \cdot e^{-\frac{t}{\kappa}} < 1$$

$$|V| < e^{\frac{t}{\kappa}} \Leftrightarrow \ln |V| < \frac{t}{\kappa}$$

$$\Leftrightarrow t > \kappa \cdot \ln |V|$$

Hence, $|Q| \leq \kappa \cdot \ln |V| + 1$. □

Exact exponential algorithms

Complexity theory deals with worst-case behavior.

- Instances you want to solve may be “easy.”

“For every polynomial-time algorithm you have, there is an exponential algorithm that I would rather run.” — Alan Perlis



“Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it.”

Alan Perlis

Exact algorithms for 3-satisfiability

Brute force. Given a 3-SAT instance with n variables and m clauses, the brute-force algorithm takes $O((m+n) 2^n)$ time.

Pf.

- There are 2^n possible truth assignments to the n variables.
- We can evaluate a truth assignment in $O(m+n)$ time. ▀

48

Exact algorithms for 3-satisfiability


A recursive framework. A 3-SAT formula Φ is either empty or the disjunction of a clause $(\ell_1 \vee \ell_2 \vee \ell_3)$ and a 3-SAT formula Φ' with one fewer clause.

$$\begin{aligned}\Phi &= (\ell_1 \vee \ell_2 \vee \ell_3) \wedge \Phi' \\ &= (\ell_1 \wedge \Phi') \vee (\ell_2 \wedge \Phi') \vee (\ell_3 \wedge \Phi') \\ &= (\Phi' \mid \ell_1 = \text{true}) \vee (\Phi' \mid \ell_2 = \text{true}) \vee (\Phi' \mid \ell_3 = \text{true})\end{aligned}$$

Notation. $\Phi \mid x = \text{true}$ is the simplification of Φ by setting x to *true*.

Ex.

- $\Phi = (x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (w \vee y \vee \neg z) \wedge (\neg x \vee y \vee z).$
- $\Phi' = (x \vee \neg y \vee z) \wedge (w \vee y \vee \neg z) \wedge (\neg x \vee y \vee z).$
- $(\Phi' \mid x = \text{true}) = (w \vee y \vee \neg z) \wedge (y \vee z).$

 each clause has ≤ 3 literals

49

Exact algorithms for 3-satisfiability

A recursive framework. A 3-SAT formula Φ is either empty or the disjunction of a clause $(\ell_1 \vee \ell_2 \vee \ell_3)$ and a 3-SAT formula Φ' with one fewer clause.

3-SAT (Φ)

```
IF  $\Phi$  is empty RETURN true.  
 $(\ell_1 \vee \ell_2 \vee \ell_3) \wedge \Phi' \leftarrow \Phi$ .  
IF 3-SAT ( $\Phi' \mid \ell_1 = \text{true}$ ) RETURN true.  
IF 3-SAT ( $\Phi' \mid \ell_2 = \text{true}$ ) RETURN true.  
IF 3-SAT ( $\Phi' \mid \ell_3 = \text{true}$ ) RETURN true.  
RETURN false.
```

Theorem. The brute-force 3-SAT algorithm takes $O(\text{poly}(n) 3^n)$ time.

Pf. $T(n) \leq 3T(n-1) + \text{poly}(n)$. ▀

50

Exact algorithms for 3-satisfiability

Key observation. The cases are not mutually exclusive. Every satisfiable assignment containing clause $(\ell_1 \vee \ell_2 \vee \ell_3)$ must fall into one of 3 classes:

- ℓ_1 is true.
- ℓ_1 is false; ℓ_2 is true.
- ℓ_1 is false; ℓ_2 is false; ℓ_3 is true.

3-SAT (Φ)

```
IF  $\Phi$  is empty RETURN true.  
 $(\ell_1 \vee \ell_2 \vee \ell_3) \wedge \Phi' \leftarrow \Phi$ .  
IF 3-SAT( $\Phi' \mid \ell_1 = \text{true}$ ) RETURN true.  
IF 3-SAT( $\Phi' \mid \ell_1 = \text{false}, \ell_2 = \text{true}$ ) RETURN true.  
IF 3-SAT( $\Phi' \mid \ell_1 = \text{false}, \ell_2 = \text{false}, \ell_3 = \text{true}$ ) RETURN true.  
RETURN false.
```

51

Exact algorithms for 3-satisfiability

Theorem. The brute-force algorithm takes $O(1.84^n)$ time.

Pf. $T(n) \leq T(n-1) + T(n-2) + T(n-3) + O(m+n)$. ■

largest root of $r^3 = r^2 + r + 1$

3-SAT (Φ)

IF Φ is empty RETURN *true*.

$(\ell_1 \vee \ell_2 \vee \ell_3) \wedge \Phi' \leftarrow \Phi$.

IF 3-SAT($\Phi' \mid \ell_1 = \text{true}$) RETURN *true*.

IF 3-SAT($\Phi' \mid \ell_1 = \text{false}, \ell_2 = \text{true}$) RETURN *true*.

IF 3-SAT($\Phi' \mid \ell_1 = \text{false}, \ell_2 = \text{false}, \ell_3 = \text{true}$) RETURN *true*.

RETURN *false*.

52

Exact algorithms for 3-satisfiability

Theorem. There exists a $O(1.33334^n)$ deterministic algorithm for 3-SAT.

A Full Derandomization of Schöning's k -SAT Algorithm

Robin A. Moser and Dominik Scheder

Institute for Theoretical Computer Science

Department of Computer Science

ETH Zürich, 8092 Zürich, Switzerland

{robin.moser, dominik.scheder}@inf.ethz.ch

August 25, 2010

Abstract

Schöning [7] presents a simple randomized algorithm for k -SAT with running time $O(a_k^n \text{poly}(n))$ for $a_k = 2(k-1)/k$. We give a deterministic version of this algorithm running in time $O((a_k + \epsilon)^n \text{poly}(n))$, where $\epsilon > 0$ can be made arbitrarily small.

53

Exact algorithms for satisfiability

DPPL algorithm. Highly-effective backtracking procedure.

- Splitting rule: assign truth value to literal; solve both possibilities.
- Unit propagation: clause contains only a single unassigned literal.
- Pure literal elimination: if literal appears only negated or unnegated.

A Computing Procedure for Quantification Theory*

MARTIN DAVIS

Research Polytechnic Institute, Hartford Division, East Windsor Hill, Conn.

AND

HILARY PUTNAM

Princeton University, Princeton, New Jersey

The hope that mathematical methods employed in the investigation of formal logic would lead to purely computational methods for obtaining mathematical theorems goes back to Leibniz and has been revived by Peano around the turn of the century and by Hilbert's school in the 1920's. Hilbert, noting that all of classical mathematics could be formalized within quantification theory, declared that the problem of finding an algorithm for determining whether or not a given formula of quantification theory is valid was the central problem of mathematical logic. And indeed, at one time it seemed as if investigations of this "decision" problem were on the verge of success. However, it was shown by Church and by Turing that such an algorithm can not exist. This result led to considerable pessimism regarding the possibility of using modern digital computers in deciding significant mathematical questions. However, recently there has been a revival of interest in the whole question. Specifically, it has been realized that while no decision procedure exists for quantification theory there are many proof procedures available—that is, uniform procedures which will ultimately locate a proof for any formula of quantification theory which is valid but which will usually involve seeking "forever" in the case of a formula which is not valid—and that some of these proof procedures could well turn out to be feasible for use with modern computing machinery.

A Machine Program for Theorem-Proving†

Martin Davis, George Logemann, and Donald Loveland

Institute of Mathematical Sciences, New York University

The programming of a proof procedure is discussed in connection with trial runs and possible improvements.

In [1] is set forth an algorithm for proving theorems of quantification theory which is an improvement in certain respects over previously available algorithms such as that of [2]. The present paper deals with the programming of the algorithm of [1] for the New York University, Institute of Mathematical Sciences' IBM 704 computer, with some modifications in the algorithm suggested by this work, with the results obtained using the completed algorithm. Familiarity with [1] is assumed throughout.

54

Exact algorithms for satisfiability

Chaff. State-of-the-art SAT solver.

- Solves real-world SAT instances with ~ 10K variable.
- Developed at Princeton by undergrads.

Chaff: Engineering an Efficient SAT Solver

Matthew W. Moskewicz

Department of EECS

UC Berkeley

moskewicz@alumni.princeton.edu

Conor F. Madigan

Department of EECS

MIT

cmadigan@mit.edu

Ying Zhao, Lintao Zhang, Sharad Malik

Department of Electrical Engineering

Princeton University

{yingzhao, lintaoz, sharad}@ee.princeton.edu

ABSTRACT

Boolean Satisfiability is probably the most studied of combinatorial optimization/search problems. Significant effort has been devoted to trying to provide practical solutions to this problem for problem instances encountered in a range of applications in Electronic Design Automation (EDA), as well as in Artificial Intelligence (AI). This study has culminated in the

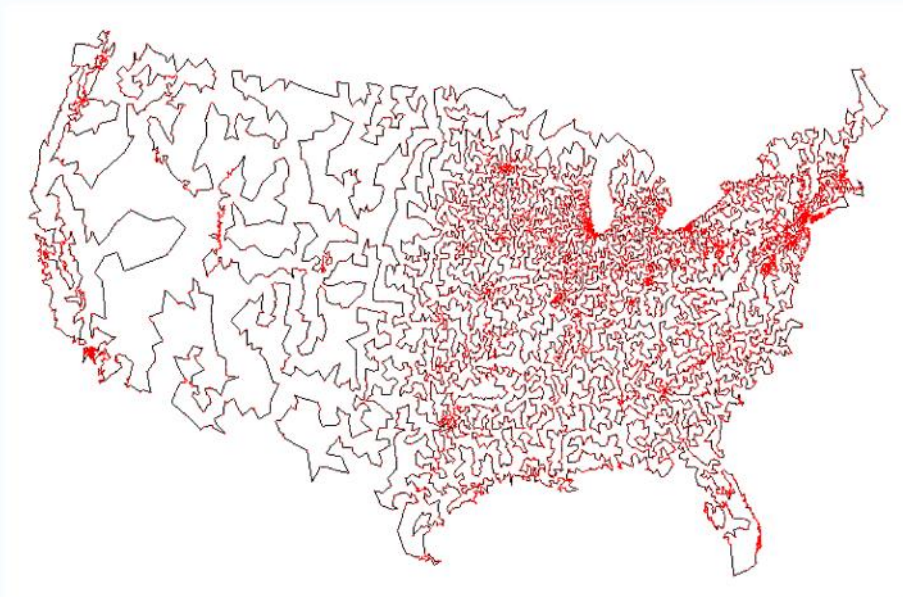
Many publicly available SAT solvers (e.g. GRASP [8], POSIT [5], SATO [13], rel_sat [2], WalkSAT [9]) have been developed, most employing some combination of two main strategies: the Davis-Putnam (DP) backtrack search and heuristic local search. Heuristic local search techniques are not guaranteed to be complete (i.e. they are not guaranteed to find a satisfying assignment if one exists or prove unsatisfiability); as a

55

Traveling salesperson problem

TSP. Given a set of n cities and a pairwise distance function $d(u, v)$, is there a tour of length $\leq D$?

can view as a complete graph

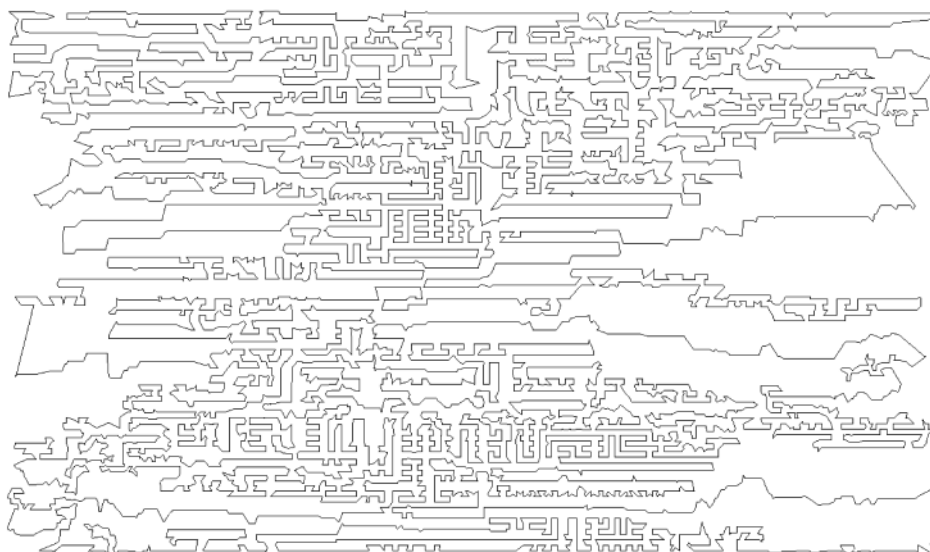


13,509 cities in the United States
<http://www.math.uwaterloo.ca/tsp>

58

Traveling salesperson problem

TSP. Given a set of n cities and a pairwise distance function $d(u, v)$, is there a tour of length $\leq D$?



11,849 holes to drill in a programmed logic array
<http://www.math.uwaterloo.ca/tsp>

59

Hamilton cycle reduces to traveling salesperson problem

TSP. Given a set of n cities and a pairwise distance function $d(u, v)$, is there a tour of length $\leq D$?

HAMILTON-CYCLE. Given an undirected graph $G = (V, E)$, does there exist a cycle that visits every node exactly once?

Theorem. $\text{HAMILTON-CYCLE} \leq_p \text{TSP}$.

Pf.

- Given an instance $G = (V, E)$ of HAMILTON-CYCLE, create $n = |V|$ cities with distance function

$$d(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E \\ 2 & \text{if } (u, v) \notin E \end{cases}$$

- TSP instance has tour of length $\leq n$ iff G has a Hamilton cycle. ■

61

Exponential algorithm for TSP: dynamic programming

Theorem. [Held-Karp, Bellman 1962] TSP can be solved in $O(n^2 2^n)$ time.

HAMILTON-CYCLE is a special case

J. Soc. Indust. Appl. Math.
Vol. 10, No. 1, March, 1962
Printed in U.S.A.

A DYNAMIC PROGRAMMING APPROACH TO SEQUENCING PROBLEMS*

MICHAEL HELD† and RICHARD M. KARP‡

INTRODUCTION

Many interesting and important optimization problems require the determination of a best order of performing a given set of operations. This paper is concerned with the solution of three such *sequencing problems*: a scheduling problem involving arbitrary cost functions, the traveling-salesman problem, and an assembly-line balancing problem. Each of these problems has a structure permitting solution by means of recursion schemes of the type associated with dynamic programming. In essence, these recursion schemes permit the problems to be treated in terms of *combinations*, rather than *permutations*, of the operations to be performed. The dynamic programming formulations are given in §1, together with a discussion of various extensions such as the inclusion of precedence constraints. In each case the proposed method of solution is computationally effective for problems in a certain limited range. Approximate solutions to larger problems may be obtained by solving sequences of small derived problems, each having the same structure as the original one. This procedure of successive approximations is developed in detail in §2 with specific reference to the traveling-salesman problem, and §3 summarizes computational experience with an IBM 7090 program using the procedure.

Dynamic Programming Treatment of the Travelling Salesman Problem*

RICHARD BELLMAN

RAND Corporation, Santa Monica, California

Introduction

The well-known travelling salesman problem is the following: "A salesman is required to visit once and only once each of n different cities starting from a base city, and returning to this city. What path minimizes the total distance travelled by the salesman?"

The problem has been treated by a number of different people using a variety of techniques; cf. Dantzig, Fulkerson, Johnson [1], where a combination of ingenuity and linear programming is used, and Miller, Tucker and Zemlin [2], whose experiments using an all-integer program of Gomory did not produce results in cases with ten cities although some success was achieved in cases of simply four cities. The purpose of this note is to show that this problem can easily be formulated in dynamic programming terms [3], and resolved computationally for up to 17 cities. For larger numbers, the method presented below, combined with various simple manipulations, may be used to obtain quick approximate solutions. Results of this nature were independently obtained by M. Held and R. M. Karp, who are in the process of publishing some extensions and computational results.

63

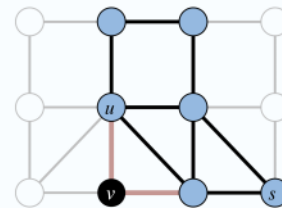
Exponential algorithm for TSP: dynamic programming

Theorem. [Held–Karp, Bellman 1962] TSP can be solved in $O(n^2 2^n)$ time.

Pf. [dynamic programming]

- Subproblems: $c(s, v, X)$ = cost of cheapest path between s and $v \neq s$ that visits every node in X exactly once (and uses only nodes in X).
- Goal: $\min_{v \in V} c(s, v, V) + c(v, s)$
- There are $\leq n 2^n$ subproblems and they satisfy the recurrence:

$$c(s, v, X) = \begin{cases} c(s, v) & \text{if } |X| = 2 \\ \min_{u \in X \setminus \{s, v\}} c(s, u, X \setminus \{v\}) + c(u, v) & \text{if } |X| > 2. \end{cases}$$



- The values $c(s, v, X)$ can be computed in increasing order of the cardinality of X . ▀

64

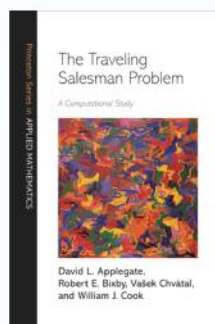
Concorde TSP solver

Concorde TSP solver. [Applegate–Bixby–Chvátal–Cook]

- Linear programming + branch-and-bound + polyhedral combinatorics.
- Greedy heuristics, including Lin–Kernighan.
- MST, Delaunay triangulations, fractional b -matchings, ...

Remarkable fact. Concorde has solved all 110 TSPLIB instances.

largest instance has 85,900 cities!



66

