Algorithm II

8. Intractability III

WU Xiaokun 吴晓堃

xkun.wu [at] gmail

Coping with NP-completeness

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

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.

- 1. Solve *arbitrary instances* of the problem.
- 2. Solve problem to *optimality*.
- 3. Solve problem in *polynomial time*.

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.

- 1. Solve *arbitrary instances* of the problem.
- 2. Solve problem to *optimality*.
- 3. Solve problem in *polynomial time*.

Coping strategies.

- 1. Design algorithms for *special cases* of the problem.
- 2. Design *approximation algorithms* or *heuristics*.
- 3. Design algorithms that may take *exponential time*.

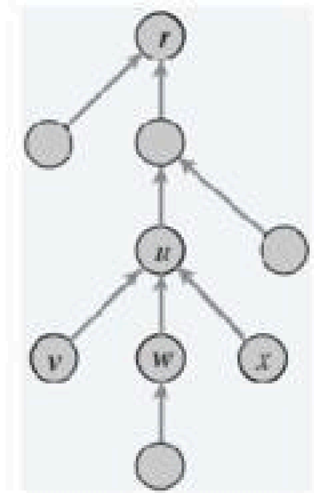
Special cases: trees

Independent set on trees

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

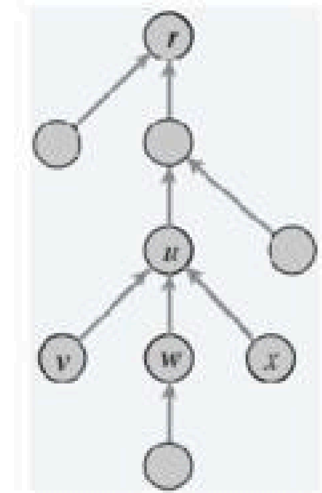
Fact. A tree has at least one leaf node (degree = 1).

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



Independent set on trees

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



Fact. A tree has at least one leaf node (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

IS on trees: greedy

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

1. $S = \emptyset$;
2. WHILE (F has at least 1 edge)
   1. Let v be a leaf node and let (u, v) be the lone edge incident to v;
   2. $S = S \cup \{v\}$;
   3. $F = F - \{u, v\}$;
3. RETURN $S \cup \text{nodes remaining in } F$;

IS on trees: greedy

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

1. $S = \emptyset$;
2. WHILE (F has at least 1 edge)
   1. Let v be a leaf node and let (u, v) be the lone edge incident to v;
   2. $S = S \cup \{v\}$;
   3. $F = F - \{u, v\}$;
3. RETURN $S \cup$ nodes remaining in F;

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

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

Demo: greedy for IS on trees

Quiz: greedy for IS

How might the greedy algorithm fail if the graph is not a tree/forest?

A. Might get stuck.
B. Might take exponential time.
C. Might produce a suboptimal independent set.
D. Any of the above.

Quiz: greedy for IS

How might the greedy algorithm fail if the graph is not a tree/forest?

A. Might get stuck.
B. Might take exponential time.
C. Might produce a suboptimal independent set.
D. Any of the above.

A. the algorithm relies on leave nodes.

Weighted independent set on trees

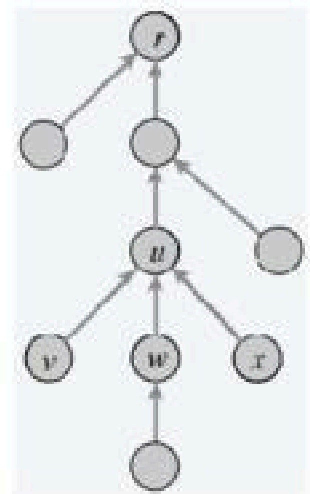
Weighted independent set on trees. Given a tree and node weights $w_v \geq 0$, find an independent set S that maximizes $\sum_{v \in S} w_v$.

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.

- hint: when w_v is huge.



≡

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 $\sum_{v \in S} w_v$.

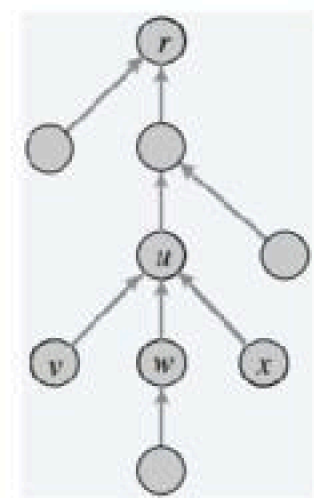
Dynamic-programming solution. Root tree at some node, say r.

- $OPT_{in}(u)$ = max-weight IS in subtree rooted at u, *including* u.
- $OPT_{out}(u)$ = max-weight IS in subtree rooted at u, *excluding* u.
- Goal: $\max\{OPT_{in}(r), OPT_{out}(r)\}$.

Bellman equation.

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

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



Quiz: DP for Weighted IS

In which order to solve the subproblems?

A. Preorder.
B. Postorder.
C. Level order.
D. Any of the above.

Quiz: DP for Weighted IS

In which order to solve the subproblems?

A. Preorder.
B. Postorder.
C. Level order.
D. Any of the above.

B. the algorithm relies on leave nodesensures a node is processed after all of its descendants.

Weighted IS on trees: DP

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

1. Root the tree T at any node r;
2. $S = \emptyset$;
3. FOREACH (node u of T in postorder/topological order)
   1. IF (u is a leaf node)
      1. $M_{in}[u] = w_u$; $M_{out}[u] = 0$;
   2. ELSE
      1. $M_{in}[u] = w_u + \sum_{v \in children(u)} M_{out}[v]$;
      2. $M_{out}[u] = \sum_{v \in children(u)} \max\{M_{in}[v], M_{out}[v]\}$;
4. RETURN $\max\{M_{in}[r], M_{out}[r]\}$;

Weighted IS on trees: DP

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

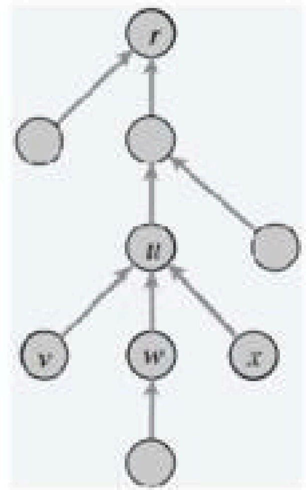
1. Root the tree T at any node r;
2. $S = \emptyset$;
3. FOREACH (node u of T in postorder/topological order)
   1. IF (u is a leaf node)
      1. $M_{in}[u] = w_u$; $M_{out}[u] = 0$;
   2. ELSE
      1. $M_{in}[u] = w_u + \sum_{v \in children(u)} M_{out}[v]$;
      2. $M_{out}[u] = \sum_{v \in children(u)} \max\{M_{in}[v], M_{out}[v]\}$;
4. RETURN $\max\{M_{in}[r], M_{out}[r]\}$;

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

Note: can also find independent set itself (not just value)

NP-hard problems on trees: intuition

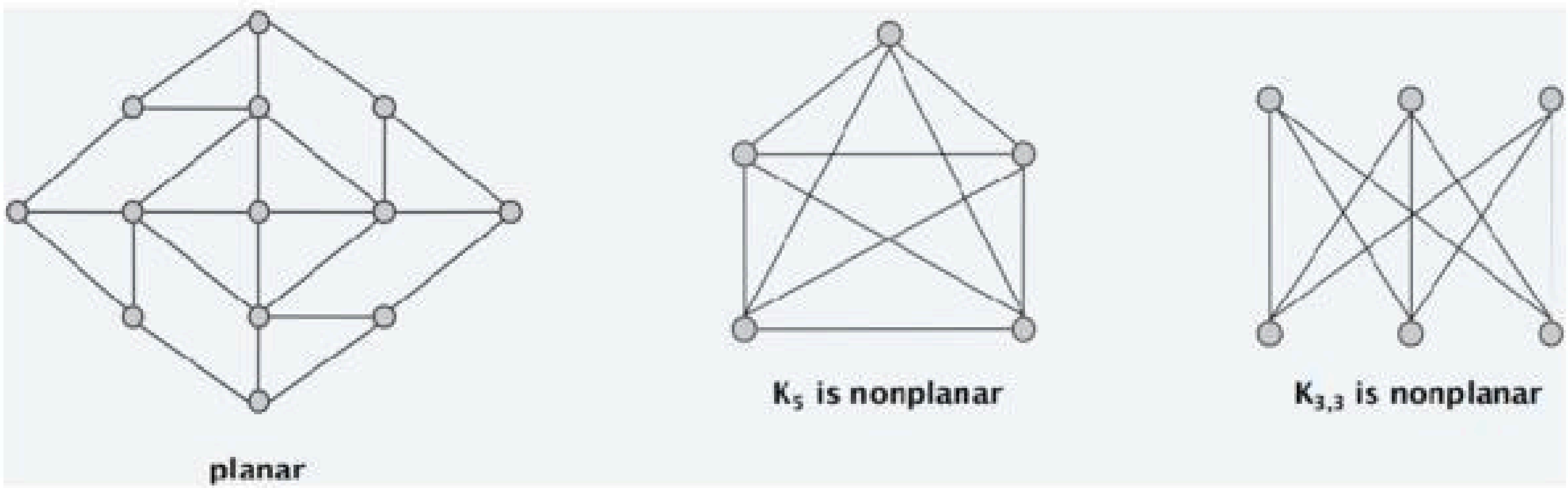
Independent set on trees. Tractable because we can find a node that *breaks the communication* among the subproblems in different subtrees.



Special cases: planarity

Planarity

Def. A graph is **planar** if it can be embedded in the plane in such a way that no two edges cross.



Applications. VLSI circuit design, computer graphics, etc.

Planarity testing

Theorem. [Hopcroft-Tarjan 1974] There exists an $O(n)$ time algorithm to determine whether a graph is planar.

Problems on planar graphs

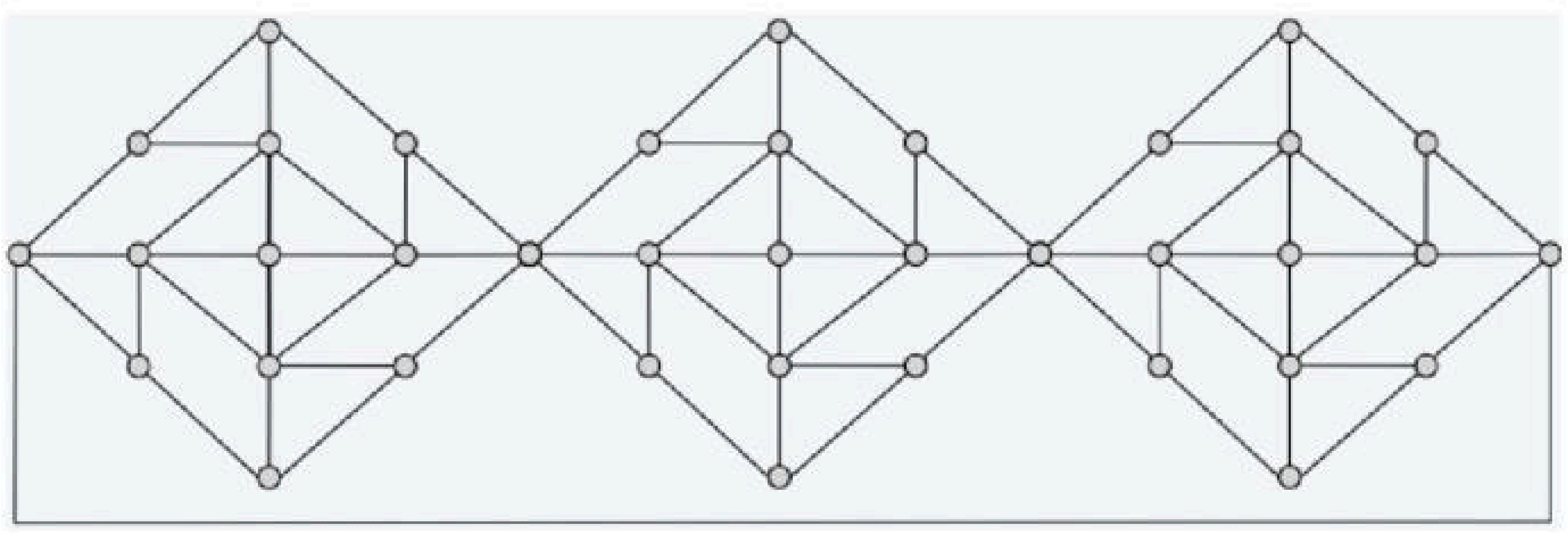
Fact 0. Many graph problems can be solved faster in planar graphs.
Ex. Shortest paths, max flow, MST, matchings, etc.

Fact 1. Some NP-complete problems become tractable in planar graphs.
Ex. MAX-CUT, ISING, CLIQUE, GRAPH-ISOMORPHISM, 4-COLOR, etc.

Fact 2. Other NP-complete problems become easier in planar graphs.
Ex. INDEPENDENT-SET, VERTEX-COVER, TSP, STEINER-TREE, etc.

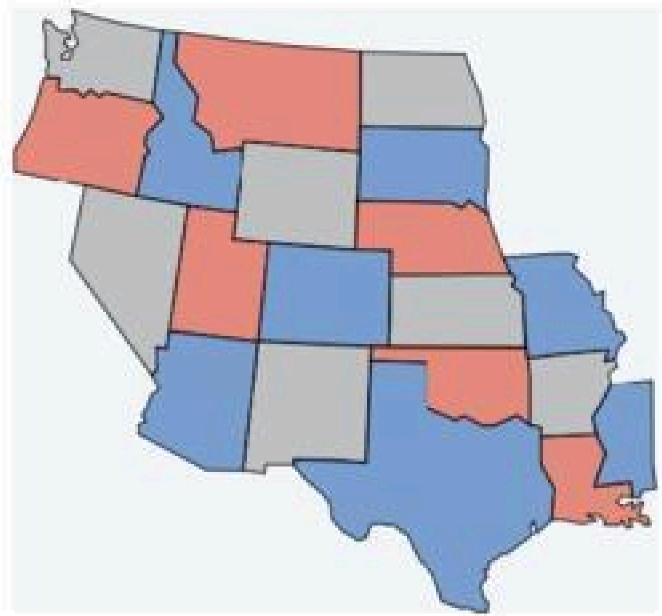
Planar graph 3-colorability

PLANAR-3-COLOR. Given a planar graph, can it be colored using 3 colors so that no two adjacent nodes have the same color?



Planar map 3-colorability

PLANAR-MAP-3-COLOR. Given a planar map, can it be colored using 3 colors so that no two adjacent regions have the same color?



Planar map 3-colorability

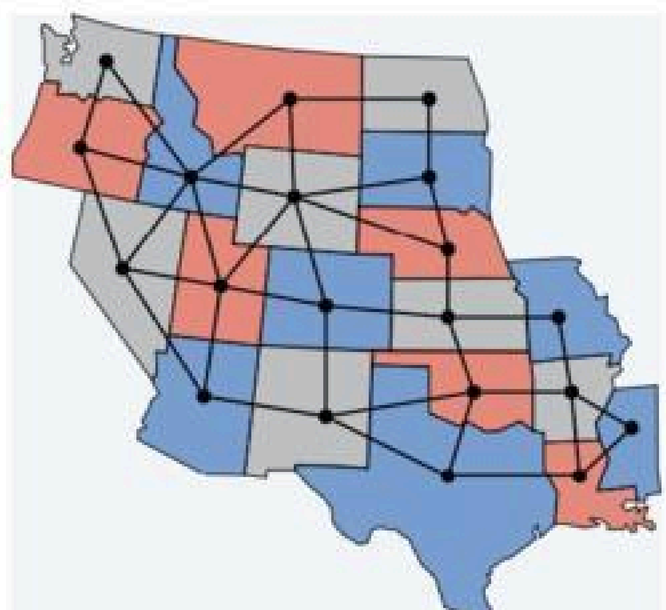
PLANAR-MAP-3-COLOR. Given a planar map, can it be colored using 3 colors so that no two adjacent regions have the same color?

Theorem: $\equiv_P$

Theorem. PLANAR-3-COLOR $\equiv_P$ PLANAR-MAP-3-COLOR.

Pf sketch.

- Nodes correspond to regions.
- Two nodes are adjacent iff they share a nontrivial border.



PLANAR-3-COLOR $\in$ NP-complete

Theorem. PLANAR-3-COLOR $\in$ NP-complete.

Pf.

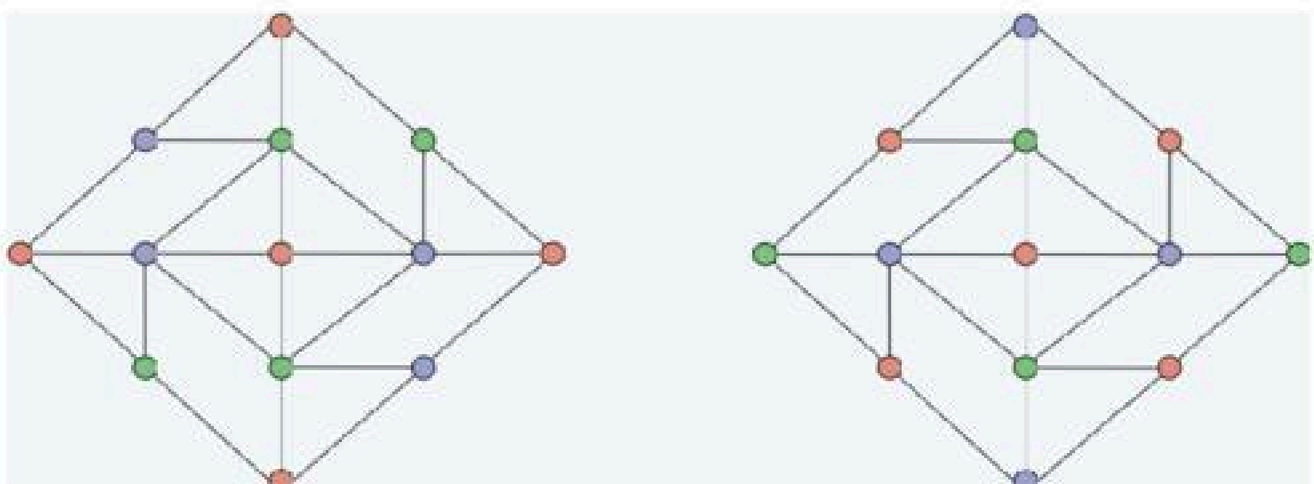
- Easy to see that PLANAR-3-COLOR $\in$ NP.
- We show 3-COLOR $\leq_P$ PLANAR-3-COLOR.
- Given 3-COLOR instance G, we construct an instance of PLANAR-3-COLOR that is 3-colorable iff G is 3-colorable.

PLANAR-3-COLOR $\in$ NP-complete: gadget

Lemma. W is a planar graph such that:

- In any 3-coloring of W, opposite corners have the same color.
- Any assignment of colors to the corners in which opposite corners have the same color extends to a 3-coloring of W.

Pf. The only 3-colorings (modulo permutations) of W are shown below.

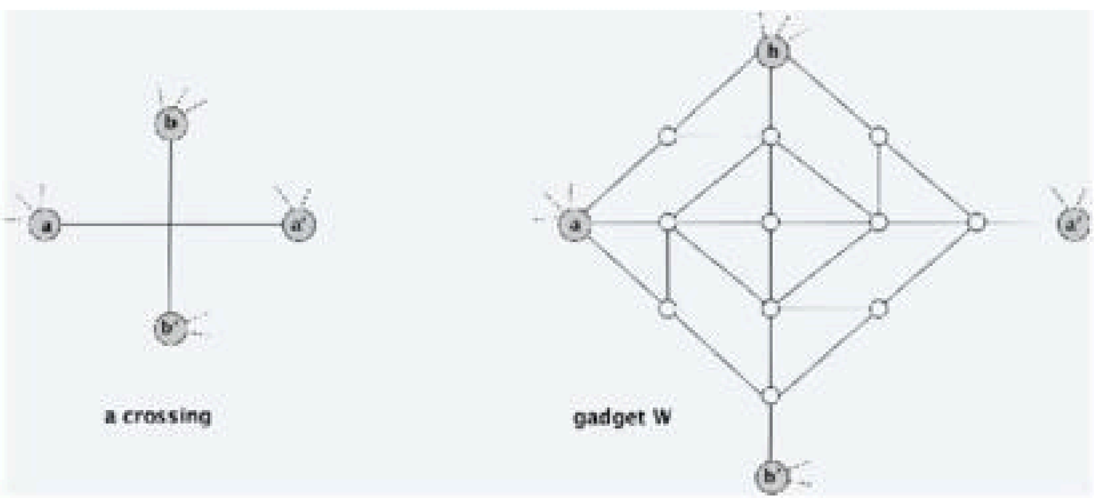


PLANAR-3-COLOR $\in$ NP-complete: lemma

Construction. Given instance G of 3-COLOR, draw G in plane, letting edges cross. Form planar G' by replacing each edge crossing with planar gadget W.

Lemma. G is 3-colorable iff G' is 3-colorable.

- In any 3-coloring of W, $a \neq a'$ and $b \neq b'$.
- If $a \neq a'$ and $b \neq b'$ then can extend to a 3-coloring of W.

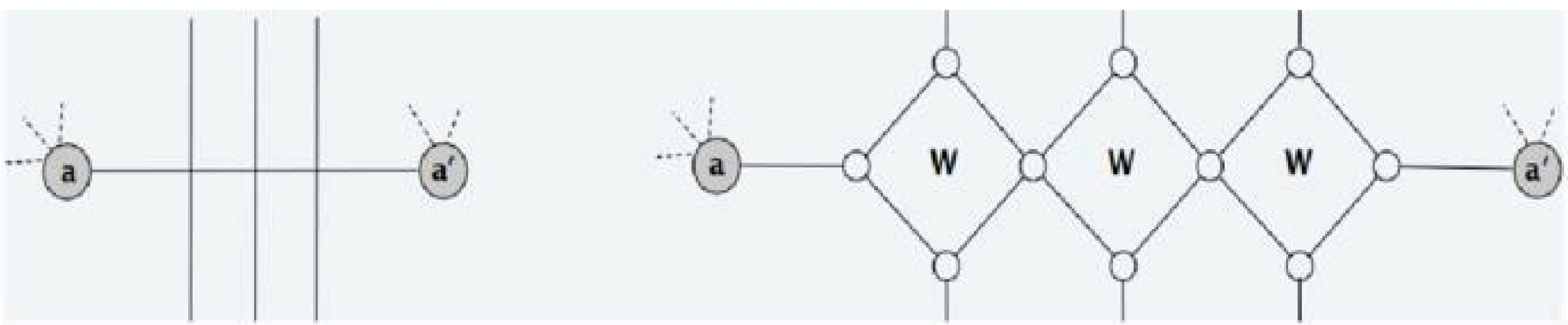


PLANAR-3-COLOR $\in$ NP-complete: lemma

Construction. Given instance G of 3-COLOR, draw G in plane, letting edges cross. Form planar G' by replacing each edge crossing with planar gadget W.

Lemma. G is 3-colorable iff G' is 3-colorable.

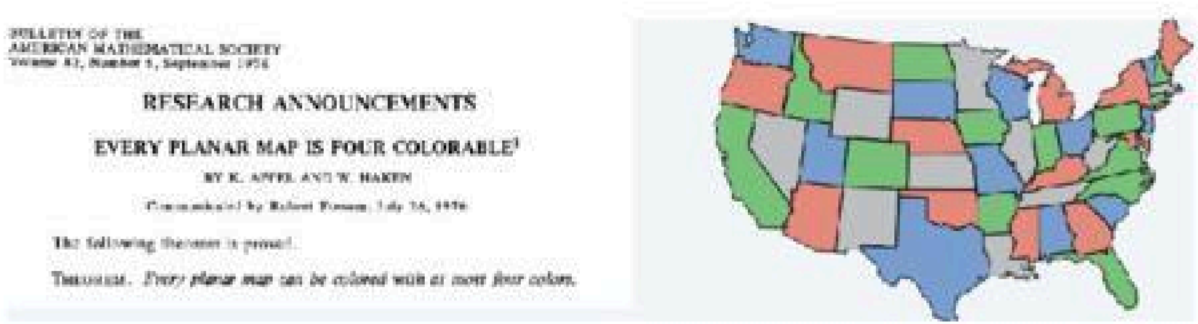
- In any 3-coloring of W, $a \neq a'$ and $b \neq b'$.
- If $a \neq a'$ and $b \neq b'$ then can extend to a 3-coloring of W.



Planar map k-colorability

Theorem. [Appel-Haken 1976] Every planar map is 4-colorable.

- Resolved century-old open problem.
- Used 50 days of computer time to deal with many special cases.
- First major theorem to be proved using computer.



Planar map k-colorability

Theorem. [Appel-Haken 1976] Every planar map is 4-colorable.

- Resolved century-old open problem.
- Used 50 days of computer time to deal with many special cases.
- First major theorem to be proved using computer.


RESEARCH ANNOUNCEMENTS

EVERY PLANAR MAP IS FOUR COLORABLE[1]

Remarks.

- Appel-Haken yields $O(n^4)$ algorithm to 4-color of a planar map.
- Best known: $O(n^2)$ to 4-color; $O(n)$ to 5-color.
- Determining whether 3 colors suffice is NP-complete.

NP-hard: Poly-time special cases

Trees. VERTEX-COVER, INDEPENDENT-SET, LONGEST-PATH, GRAPH-ISOMORPHISM, etc.

Bipartite graphs. VERTEX-COVER, INDEPENDENT-SET, 3-COLOR, EDGE-COLOR, etc.

Planar graphs. MAX-CUT, ISING, CLIQUE, GRAPH-ISOMORPHISM, 4-COLOR, etc.

Bounded treewidth. HAM-CYCLE, INDEPENDENT-SET, GRAPH-ISOMORPHISM, etc.

Small integers. SUBSET-SUM, KNAPSACK, PARTITION, etc.

Approximation algorithms: vertex cover

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 $\leq 2 \cdot OPT(G)$ vertices in $O(m + n)$ time.

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.

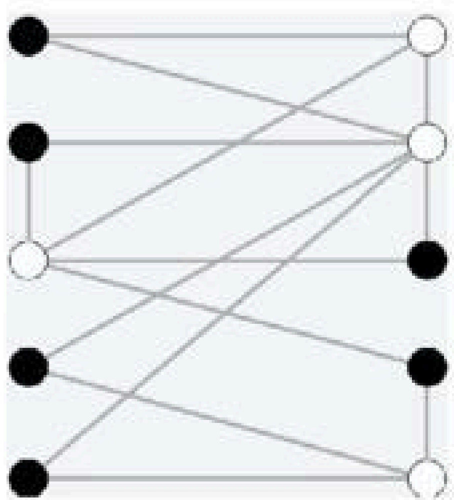
Ex. Given a graph G, can find a vertex cover that uses $\leq 2 \cdot OPT(G)$ vertices in $O(m+n)$ time.

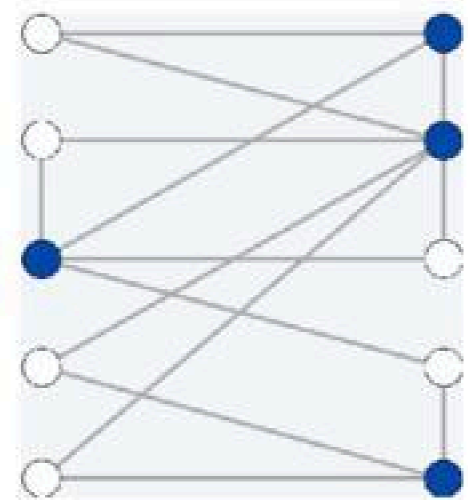
Challenge. Need to prove a solution's value is close to optimum value, without even knowing what optimum value is!

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: greedy

GREEDY-VERTEX-COVER(G)

1. $S = \emptyset$; $E' = E$;
2. WHILE ($E' \neq \emptyset$)
    1. Let $(u, v) \in E'$ be an arbitrary edge;
    2. $M = M \cup \{(u, v)\}$;
    3. $S = S \cup \{u\} \cup \{v\}$;
    4. Delete from E' all edges incident to either u or v;
3. RETURN S;

Vertex cover: greedy

GREEDY-VERTEX-COVER(G)

1. $S = \emptyset; E' = E;$
2. WHILE ($E' \neq \emptyset$)
   1. Let $(u, v) \in E'$ be an arbitrary edge;
   2. $M = M \cup \{(u, v)\};$
   3. $S = S \cup \{u\} \cup \{v\};$
   4. Delete from E' all edges incident to either u or v;
3. RETURN S;

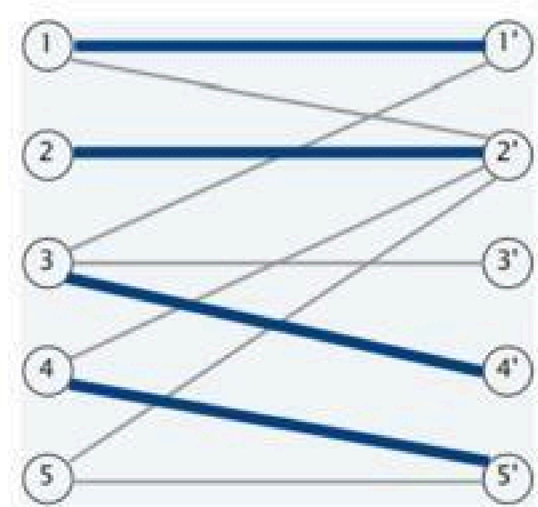
Running time. Can be implemented in $O(m + n)$ time.

Demo: Greedy Vertex-Cover

Quiz: Vertex cover

Given a graph G, let M be any matching and let S be any vertex cover. Which of the following must be true?

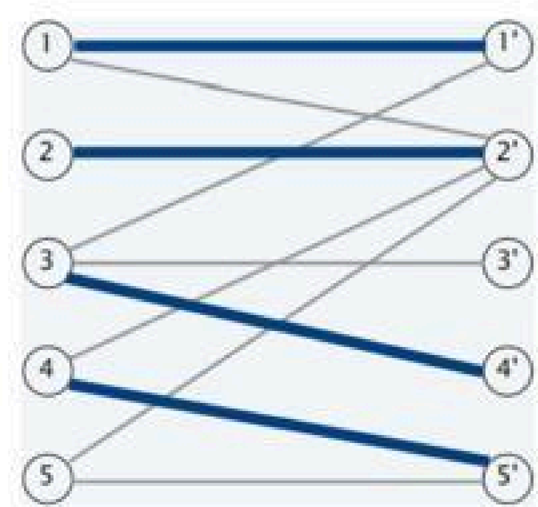
A. $|M| \leq |S|$
B. $|S| \leq |M|$
C. $|S| = |M|$
D. None of the above.



Quiz: Vertex cover

Given a graph G, let M be any matching and let S be any vertex cover. Which of the following must be true?

A. $|M| \leq |S|$
B. $|S| \leq |M|$
C. $|S| = |M|$
D. None of the above.



A. if two nodes not matched, then they are not covered and conected, contra to cover; when covering nodes are matched to each other, strictly less.

Pf. Each vertex can cover at most one edge in any matching.

Vertex cover: 2-approximation

Theorem. Let S^* be a minimum vertex cover. Then, greedy algorithm computes a vertex cover S with $|S| \leq 2|S^*|$ (ie. **2-approximation** algorithm).

Pf.

- S is a vertex cover.
  - (delete edge only after it's already covered)
- M is a matching.
  - (when (u, v) added to M, all edges incident to either u or v are deleted)
- $|S| = 2|M| \leq 2|S^*|$.
  - by design of algorithm, and "weak duality"

Vertex cover: 2-approximation

Theorem. Let S^* be a minimum vertex cover. Then, greedy algorithm computes a vertex cover S with $|S| \leq 2|S^*|$ (ie. **2-approximation** algorithm).
Pf.

- S is a vertex cover.
  - (delete edge only after it's already covered)
- M is a matching.
  - (when (u, v) added to M, all edges incident to either u or v are deleted)
- $|S| = 2|M| \leq 2|S^*|$.
  - by design of algorithm, and "weak duality"

Corollary. Let M^* be a maximum matching. Then, greedy algorithm computes a matching M with $|M| \geq ½|M^*|$.
Pf. $|M| = ½|S| \geq ½|M^*|$.

Vertex cover inapproximability

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

Open research problem. Close the gap $(1.3606, 2)$.

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

Approximation algorithms: knapsack

Knapsack problem

Knapsack problem.

- Given n objects and a knapsack.
- Item i has value $v_i > 0$ and weighs $w_i > 0$.
- Knapsack has *weight limit* W.
- Goal: fill knapsack so as to *maximize total value*.

Knapsack problem

Knapsack problem.

- Given n objects and a knapsack.
- Item i has value $v_i > 0$ and weighs $w_i > 0$.
- 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

Knapsack is NP-complete

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:

$$\sum_{i \in S} w_i \leq W, \sum_{i \in S} v_i \leq V$$

Knapsack is NP-complete

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:

$$\sum_{i \in S} w_i \leq W, \sum_{i \in S} v_i \leq V$$

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?

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

$$v_i = w_i = u_i \quad \sum_{i \in S} u_i \leq U$$

$$V = W = U \quad \sum_{i \in S} u_i \leq U$$

Knapsack problem: DP I

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

Knapsack problem: DP I

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

Case 1. OPT does not select item i.

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

Case 2. OPT selects item i.

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

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

Knapsack problem: DP I

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

Case 1. OPT does not select item i.

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

Case 2. OPT selects item i.

- New weight limit = $w - w_i$.
- OPT selects best of $1, .., i-1$ 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.

Knapsack problem: DP 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 $1, .., i$.

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

Knapsack problem: DP 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 $1, .., i$.

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 $1, .., i-1$ 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 $1, .., i-1$ that achieves value $\geq v - v_i$.

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

Knapsack problem: DP II (cont.)

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 $v \leq v_{max}$.
- It takes $O(1)$ time per subproblem.

Knapsack problem: DP II (cont.)

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 $v \leq v_{max}$.
- It takes $O(1)$ time per subproblem.

Remark 1. Not polynomial in input size! (pseudo-polynomial)
Remark 2. Polynomial time if values are small integers.

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

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

Poly-time approximation scheme

Round up all values:

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

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

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.

Poly-time approximation scheme

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

Pf.

$$
\begin{aligned}
\sum_{i \in S^*} v_i &\le \sum_{i \in S^*} \bar{v}_i && \text{round up} \\
&\le \sum_{i \in S} \bar{v}_i && \text{optimality} \\
&\le \sum_{i \in S} (v_i + \theta) && \text{rounding gap} \\
&\le \sum_{i \in S} v_i + n\theta && |S| \le n \\
&= \sum_{i \in S} v_i + \frac{1}{2}\epsilon v_{max} && \theta = \epsilon v_{max} / 2n \\
&\le (1+\epsilon) \sum_{i \in S} v_i && v_{max} \le 2 \sum_{i \in S} v_i
\end{aligned}
$$

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

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

Exponential algorithms: 3-SAT

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*

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.

3-satisfiability: recursive

A recursive framework. A 3-SAT formula Φ is either empty or the disjunction of a clause $(l_1 \vee l_2 \vee l_3)$ and a 3-SAT formula Φ' with one fewer clause.

$$\begin{aligned}
\Phi &= (l_1 \vee l_2 \vee l_3) \wedge \Phi' \\
&= (l_1 \wedge \Phi') \vee (l_2 \wedge \Phi') \vee (l_3 \wedge \Phi') \\
&= (\Phi' | l_1 = true) \vee (\Phi' | l_2 = true) \vee (\Phi' | l_3 = true)
\end{aligned}$$

Notation. $\Phi | x = true$ is the simplification of Φ by setting x to $true$.

3-satisfiability: recursive

A recursive framework. A 3-SAT formula Φ is either empty or the disjunction of a clause $(l_1 \vee l_2 \vee l_3)$ and a 3-SAT formula Φ' with one fewer clause.

$$\begin{aligned}
\Phi &= (l_1 \vee l_2 \vee l_3) \wedge \Phi' \\
&= (l_1 \wedge \Phi') \vee (l_2 \wedge \Phi') \vee (l_3 \wedge \Phi') \\
&= (\Phi' | l_1 = true) \vee (\Phi' | l_2 = true) \vee (\Phi' | l_3 = true)
\end{aligned}$$

Notation. $\Phi | x = true$ is the simplification of Φ by setting x to $true$.

Ex.

$$\begin{aligned}
\Phi &= (x \vee y \vee \neg z) \quad \wedge (x \vee \neg y \vee z) \wedge (w \vee y \vee \neg z) \quad \wedge (\neg x \vee y \vee z) \\
\Phi' &= \qquad\qquad\qquad\quad\; \wedge (x \vee \neg y \vee z) \wedge (w \vee y \vee \neg z) \quad \wedge (\neg x \vee y \vee z) \\
(\Phi' | x = true) &= \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\; \wedge (w \vee y \vee \neg z) \qquad\qquad \wedge (y \vee z)
\end{aligned}$$

3-satisfiability: algorithm

A recursive framework. A 3-SAT formula Φ is either empty or the disjunction of a clause $(l_1 \vee l_2 \vee l_3)$ and a 3-SAT formula Φ' with one fewer clause.

3-satisfiability: algorithm

A recursive framework. A 3-SAT formula Φ is either empty or the disjunction of a clause $(l_1 \vee l_2 \vee l_3)$ and a 3-SAT formula Φ' with one fewer clause.

`3-SAT` (Φ)

1. IF Φ is empty RETURN $true$;
2. $(l_1 \vee l_2 \vee l_3) \wedge \Phi' = \Phi$;
3. IF `3-SAT` $(\Phi' | l_1 = true)$ RETURN true;
4. IF `3-SAT` $(\Phi' | l_2 = true)$ RETURN true;
5. IF `3-SAT` $(\Phi' | l_3 = true)$ RETURN true;
6. RETURN false;

3-satisfiability: algorithm

A recursive framework. A 3-SAT formula Φ is either empty or the disjunction of a clause $(l_1 \vee l_2 \vee l_3)$ and a 3-SAT formula Φ' with one fewer clause.

`3-SAT` (Φ)

1. IF Φ is empty RETURN $true$;
2. $(l_1 \vee l_2 \vee l_3) \wedge \Phi' = \Phi$;
3. IF `3-SAT` $(\Phi' | l_1 = true)$ RETURN true;
4. IF `3-SAT` $(\Phi' | l_2 = true)$ RETURN true;
5. IF `3-SAT` $(\Phi' | l_3 = true)$ RETURN true;
6. RETURN false;

Theorem. The brute-force 3-SAT algorithm takes $O(poly(n) 3^n)$ time.
Pf. $T(n) \leq 3T(n-1) + poly(n)$.

3-satisfiability: algorithm II

Key observation. The cases are not mutually exclusive. Every satisfiable assignment containing clause $(l_1 \vee l_2 \vee l_3)$ 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-satisfiability: algorithm II

Key observation. The cases are not mutually exclusive. Every satisfiable assignment containing clause $(l_1 \vee l_2 \vee l_3)$ 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)

1. IF Φ is empty RETURN $true$;
2. $(l_1 \vee l_2 \vee l_3) \wedge \Phi' = \Phi$;
3. IF `3-SAT` $(\Phi' | l_1 = true)$ RETURN true;
4. IF `3-SAT` $(\Phi' | l_1 = false, l_2 = true)$ RETURN true;
5. IF `3-SAT` $(\Phi' | l_1 = false, l_2 = false, l_3 = true)$ RETURN true;
6. RETURN false;

3-satisfiability: theoretical

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

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

- 1.84? largest root of $r^3 = r^2 + r + 1$

Theorem. [Moser and Scheder 2010] There exists a $O(1.33334^n)$ deterministic algorithm for 3-SAT.

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.

Satisfiability: best known

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

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

Exponential algorithms: TSP

Pokemon Go

Given the locations of n Pokémon, find shortest tour to collect them all.

Traveling salesperson problem

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

13,509 cities in the United States

- http://www.math.uwaterloo.ca/tsp

HAM-CYCLE $\leq_P$ TSP

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

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

Theorem. HAM-CYCLE $\leq_P$ TSP.
Pf.

- Given an instance $G = (V, E)$ of HAM-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.

Exponential algorithm for TSP: DP

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 $\leq n2^n$ subproblems and they satisfy the recurrence:

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

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

.

Exponential algorithm for TSP: DP

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 $\leq n2^n$ subproblems and they satisfy the recurrence:

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

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

.

Remark. 22-city TSP instance takes 1,000 years!

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, etc.

Remarkable fact. Concorde has solved all 110 TSPLIB instances.

- largest instance has 85,900 cities!

Euclidean TSP

Euclidean TSP. Given n points in the plane and a real number L, is there a tour that visit every city exactly once that has distance $\leq L$?

Fact. 3-SAT $\leq_P$ EUCLIDEAN-TSP.

Remark. Not known to be in $\mathcal{NP}$.

Euclidean TSP

Euclidean TSP. Given n points in the plane and a real number L, is there a tour that visit every city exactly once that has distance $\leq L$?

Fact. 3-SAT $\leq_P$ EUCLIDEAN-TSP.

Remark. Not known to be in $\mathcal{NP}$.

Theorem. [Arora 1998, Mitchell 1999] Given n points in the plane, for any constant $\epsilon > 0$: there exists a poly-time algorithm to find a tour whose length is at most $(1 + \epsilon)$ times that of the optimal tour.
Pf recipe. Structure theorem + divide-and-conquer + dynamic programming.