CS 466/666
Topic 7: NP-Completeness and Dealing with it
Lectures: October 23 and 27

We have proved that SAT is NP-Complete. Indeed, if you look at the proof, you see that only a special case of SAT is used, namely the product (and) of sums (or) of literals (variables and their negations). This $\Pi\ \Sigma$ (literals) form is called **conjunctive normal form.** (Similarly, the $\Sigma\ \Pi$ (literals) form is called **conjunctive normal form**) One can convert an arbitrary Boolean formula to either of these forms, but it may result in a dramatic (exponential) increase in size. The first question we ask, though, is how much more we can restrict the form of a Boolean expression and still have its Sat problem NP-complete. Curiously, 3 literals per clause is as hard as the general problem. We can reduce general CNF-SAT to CNF-3SAT (which we will simply call 3-SAT). Consider an arbitrary CNF expression $\Pi\ C_i$, where $C_i = (a_{i1} \vee a_{i2} \vee \ldots\ a_{ik})$ and a_{ij} denotes a literal. The reduction is quite straightforward and causes only a linear increase in size.
Write a clause $C = (a_1 \vee a_2 \vee \ldots\ a_k)$ as $(a_1 \vee a_2 \vee b_1)\ \Pi_{i=1,..k-4}\ (\neg b_i \vee a_{i+2} \vee b_{i+1})\ (\neg b_{k-3} \vee a_{k-1} \vee a_k)$. Here the b's are new variables not used anywhere else in the construction. If C is made true by assigning a_j true, then the new form is made true by making a_j true, all b_i that occur earlier than the clause where a_j are made true and all that come later than this clause are made false.
So how about just 2 literals per clause? Things become dramatically easier:
2-Sat is easily solved in polynomial time. Simply make an arbitrary assignment of one variable. This satisfies some clauses and forces the assignment of other variables if certain other clauses are to be satisfied. The process of forced assignment and clause elimination continues until we either eliminated some clauses and no remaining variables are forced, or we have a contradiction (the same variable is forced to be true in one clause and false in another). In the former case, we keep the assignments and repeat the process until no clauses are remain unsatisfied. In the latter case, our tentative assignment does not work, so we must make the opposite one … which goes through a similar forcing process and leads either to the conclusion that the expression is not satisfiable or finds a satisfying assignment. It is not hard to eliminate a single variable in O(n), but indeed then entire process can be done in linear time.

Input: CNF expression of n clauses each contains 2 literals
(A literal is a Boolean variable or its negative)

Output: A satisfying assignment of variables or, essentially, a proof if these are none.

Note: There are most 2n variables

Data Structures:
For each variable x_i create
doubly linked list of clauses containing x_i (without $\neg$) and
doubly linked list of clauses containing $\neg\ x_i$.

Each clause representation has 2 sets of flags to give status of each variable (T, F,?) and the clauses (T,?)

Algorithm:

Choose an arbitrary variable x_i from a clause still unsatisfied. Simultaneously (e.g. you could alternate steps) run 2 processes one makes tentative assignment x_i =T, the other makes x_i =F, (Data structures marked independently by the two)

All clauses made true by the assignment are flagged as such. For all clauses in which the negation of the assigned value occurs, the other literal of that clause is forced to hold. Continue with these forced assignments (successively) until either:

i) All assignments have been made; hence every clause is satisfied or has no assignment to either term.

or

ii) We discover some variable is forced to both T & F. Hence a contradiction to original assignment.

Case (i): Halt the other process and run through it again undoing all its markings. (This takes same amount of time as making assignment). Assignments made by the properly terminating process are "made permanent". This leaves us with a subset of clauses & variables, all these remaining clauses in original form. The time taken is the same for each process; the number of clauses removed is proportional to time taken, as a clause is inspected at most twice by properly termination process. Reapply procedure to remaining clauses starting with some remaining variables (dual) assignment.

Case (ii) Terminating assignment leads to contradiction. Let other assignment process continue to completion. Satisfaction if it finds one.

Coping with NP-completeness

Given a problem: Look for a solution by mapping to a known problem or type of problem.

Don't see a fast solution; perhaps it is NP hard.

Reduce NP hard problem to new one – but be we careful

In doing so, make sure your problem with all its special constraints is NP Hard….and not a potentially easier special case (e.g. 2-SAT).

Now you are stuck - try a "heuristic" of some sort, may get (optimal) solution or may not.

One notion is to get an approximation algorithm with a guaranteed approximation ratio.

$\rho(n) \geq \max\left(\frac{c}{c^*}, \frac{c}{c^*}\right)$ where c=our solution cost, c* = optimal (unknown)

The idea of the maximum is that the ratio is greater than 1 whether we have a maximization or minimization problem.

Ideal Situation:

Approximation scheme: taking into account ε, and getting a (1+ ε) ratio.

Polynomial time approximation scheme: a method that for any fixed ε >o, scheme runs in polynomial time (e.g. $n^{1/\varepsilon}$ is ok here)

But really we want
Fully polynomial time approximation scheme: runtime polynomial in both 1/ε and n, e.g. $O((1/\varepsilon)^2 n^3)$.

We will see a variety of problems/solutions.

Generally methods are simple. Bounds are pessimistic in terms of typical behaviour. Once we have an approximate solution, we may be able to find an even better solution and so a better guarantee for the particular case.

Vertex cover: Given undirected graph G=(V,E) and subset of vertices $V' \subseteq V$ such that $\forall$ edge (u,v) $u \varepsilon V'$ $v \varepsilon V'$ or both. Size of vertex cover (# nodes) is issue.

Problem find minimum size Vertex Cover: Given a graph, find smallest set of vertices so every edge incident with at least one of these nodes.

Problem is NP-hard: Reduce Clique problem to it.

```
Approx_VC (G)
C ←Ø
E'←E
While E'≠ Ø
do      { choose arbitrary (u,v) in E'
        C←C  ∪ {u,v}  {ie. take both!!!)
        Remove for E' every edge incident with u or v.
        }

Return C
```

Ok, it's a silly method but..

Theorem: Approx_VC is a polynomial time 2-approximation algorithm.
Proof: With care the method is linear.
Let A denote set of edges chosen
Every vertex cover, including C*, must contain at least one endpoint of some edge in A
Hence $|C^*| \geq |A| = 2|C|$

So we have a 2-approximation. Interesting idea, we don't know C*, but can still argue about it. And note the method can be as bad as the bound states.

Traveling Salesman Problem:

Given complete undirected graph G with nonnegative costs on edges, c(u,v). Find the Hamiltonian cycle of minimum length.

{c(A) will denote cost of our approximation A, c(A) = $\sum_{(u,v)\in A} c(u,v)$

The problems are both NP-hard: To prove Hamiltonian cycle is NP-complete reduce from VC. This is an interesting proof using an innovative construction (see CLRS). Showing TSP is NP-hard is eay. Reduce Hamiltonian cycle to TSP {HC on unweighted graph → make complete graph by connecting edges weight 1 "nonedges" weight 2. Original has HC iff new weighted graph has tour of length n}

Interesting observation: In reduction, give edges weight 0, non edges weight 1. Then HC iff tour of length 0. So no Polytime approximations algorithms unless P=NP. If you feel weights of length 0 are a "cheat", give edges weight 1 and non edges weight 2^n. This makes the encoding still polynomial in the original size, indeed $O(n^3)$ bits where n is the number of nodes in the graph. (n bits for each of the non edges.) Now any deviation from a Hamiltonian circuit gives a path length exponential in n.

In many situations, however, we can add a constraint, the *triangle inequality*: $c(u,w) \leq c(u,v + c(v,w)$

Fact: TSP NP-hard even with triangle inequality {as in original reduction above from HC}

Approx_TSP_2MST
Find a minimum spanning tree of the graph. Tour is given by depth first search order of the tree (and return to start node)

Why is this a 2-Approximation:

1) Let C(T) denote sum of weights of tree edges C(H*) the cost of optimal TSP. C(A) cost of our approximation

Note H* with any edge removed is a tree so:
$C(H^*) \geq C(T)$

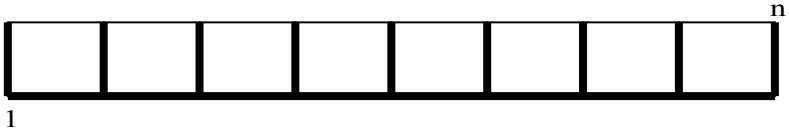
2) Consider a "full walk" of the MST, i.e. follow each edge in both directions
- as we go to a new node

and
- as we return after recursive call.

Then cost of full walk = 2C(T)

But over approximations cost is ≤ cost of full walk as we back up we may shorten the path by triangle inequality, hence we have a 2 approx.

Fact: It can be that bad.



All edges indicated have length 1.

All missing edges have length inferred by triangle inequality
Optimal tour clearly length n.

Choose an unfortunate MST (as per heavy lines). Tour starts at 1 then covers bottom then top of each vertical line ending at node n. This has cost 3n/2 – 2. The return to 1 costs n/2, for a total of 2n-2.

Can we do better?

Key idea of 2-MST was to get a "cheap" subset of the edges on which you can do a tour.

New Idea: Eulerian tour (recall Bridges of Könnigsberg). Let G be any connected multigraph (maybe more than one edge (u,v)) in which every node is of even degree. Then it is possible to start at an arbitrary point and take a walk traversing every edge exactly once, returning to the start point. Indeed the algorithm to do this is very easy.

Algorithm Euler-Tour: Start anywhere, at each point go to a new node if possible, otherwise take a new edge to an old node. This must return to start point, but may miss some edges; "splice" them in. E.g. if there is an unused edge at v (and v has been visited) after first visit to v take a formerly unused edge from v, this walk uses formerly unused edges and returns to v. Pick up old tour from that point.

Observe: Our "2 copies" of MST was such a multigraph.

How do we get a cheaper Eulerian graph?

1) Find MST. This is the cheapest connected graph.

2) Consider nodes of <u>odd</u> degree in MST (they can use 1 more edge each).

3) Find Minimum Weighted Matching on nodes of odd degree.
   [Pair these nodes so some of the chosen edges are minimized. This can be done in $O(n^{2.5})$ time, though the method is sophisticated]

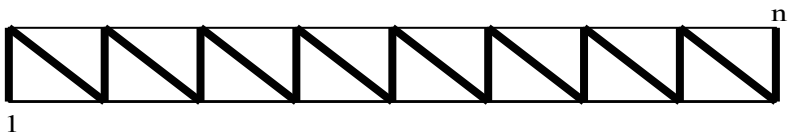
Do Eulerian tour on multigraph of MST plus MWM. Take "shortcuts"?

Cost: 1) $C(MST) \leq C(H_o^*)$
2) Consider the odd degree nodes we used for matching. Let H_o^* denote optimal TSP tour on these nodes and M_o is the matching cost. Consider every second edge of H_o^* (there are 2 choices). M_o is at most the cost of either of these (as they are matching) so $M_o \leq C(H_o^*)/2$

But the TST on a subset of nodes costs at most as much as the TSP on all.
So from 1) and 2) and our Eulerian tour with shortcuts

$$\begin{aligned} C(A) &\leq C(MST) + M_o \\ &\leq C(H^*) + C(H^*)/2 \\ &\leq 3C(H^*)/2 \end{aligned}$$

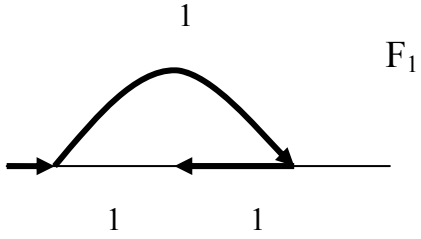
Fact: Our approximation can be this bad



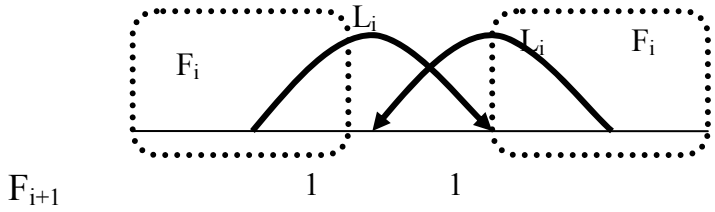
Consider essentially the same "ladder" example, except edges (2i.2i+1) also have length 1
Choose the (unfortunate) MST as the path from node 1 to node n (length n -1). Then the only nodes of odd degree are 1 and n (distance n/2). Our approximation method gives a tour of length 3n/2 - 1 instead of n

Other approaches to TSP
Nearest Neighbour approach to extending path. That is, at each step extend the path to the nearest unvisited node. At the end, we must return to the start node,
This works reasonably in "practice", but can give an approximation ratio, ρ, of (lg n)/3. The bad case occurs when the optimal is to simply go around a circle in unit steps. Unfortunately, we again make bad choices when given several edges of the same length.

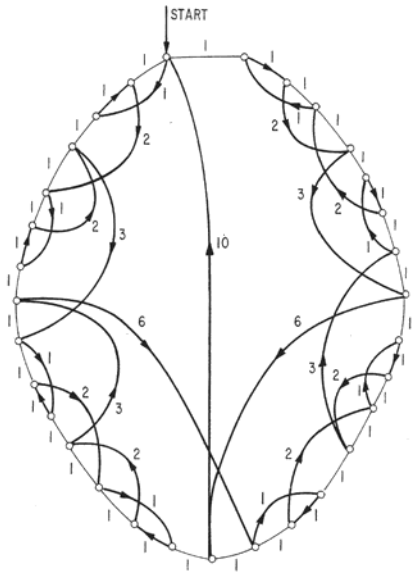
Construction: The base case, F_1:



In general, F_{i+1}:



Choosing the L_i to be the largest permitted by the triangle inequality so that we take these edges, we can force $L_1 = 2$ and $L_i = (6i2^i + 8\ 2^i + (-1)^i - 9)/9$. This gives the desired ratio. It can be shown that the method can produce no worse result.



Another approach is Nearest Insert, that is to start with a subtour and add in the cheapest new node. This gives $\rho \leq 2$.
Often the TSP to be solved is actually Euclidean distance in the plane. In such a case one can start with ant tour (even nearest neighbour) and improve it by detecting whether any edges cross. So if a pair of edges, say AD and BC crossed, them by opposing pairs of side edges. It can be shown that one of the choices will produce a single shorter cycle tour, while the other will give two disjoint cycles.

Subject Sum Problem

Given $S = \{x_1, ..., x_n\}$ and t where x_i, t positive integers
Decision problem: Is there a subset of S with sum t?
Optimization problem: Find subset of S with sum as close as possible to t but no larger.
{Application t is a capacity (e.g. weight) and we are to carry as much as possible without going over.}

Related problem … generalization is knapsack problem where we have values $\{v_i\}$ and weights $\{w_i\}$ and must maximize value while keeping weight below a threshold.

Observe that the "natural" greedy approach of the knapsack problem would be to focus on items with "high unit value", i.e. v_i / w_i. With the subset problem these unit values are all 1. A natural heuristic may be to take some large items and then use the small ones to see how close to full we can come.

In any case:
Theorem: Subset sum (decision problem) is NP-complete.
The proof is by reducing vertex cover to subset sum.

i). Trivially subset sum is NP
ii). We reduce from 3-CNF

Given variables $x_1 ... x_n$
Clauses $C_1 ... C_k$ each with 3 literals.
2 simplifying assumptions that do not change the worst case complexity:

- No clause has variable and negation (clearly we can remove clause and the other variable)
- Each variable in at least 1 clause (otherwise we can ignore it)

Construction

2 numbers in S for each x_i

2 numbers in S for each C_j

We'll use base 10, and n+k digit #'s. Label each digit position by a variable or a clause

- The target, t, has 1 in each variable digit and 4 in each clause digit.
- For each x_i we have v_i and v_i', each has a 1 in digit position x_i and 0 in all other x_j positions.
  - If x_i is in C_j then position C_j is 1 in v_i.
  - If $\neg x_i$ is in C_j then position C_j is 1 in v_i'. All other clause digits are 0 (Note: all v_i, v_i' are unique in S)
- For each clause C_j there are two integers S_j and S_j' in S.
  - All digits are 0 except the one labeled by C_j

    S_j is 1 in C_j

    S_j' is 2 in C_j

These are "slack" variables that can get the digit position, labeled by the clause up to 4.

We refer to the text for the formal details of why the construction works and an example.

Approximating Subset Sum: We are aiming for a fully polynomial time approximation scheme, so we start with an (exponential) algorithm to find the optimal solution.

If L is a list of positive integers and x be another positive integer. L+x is the list with each value increased by x.

{so L = <5,6,7> x = 2 then L+x = <7,8,9>}

Similarly this notation applies to sets, indeed we will represent our sets as lists in increasing order. Hence we can have the linear time procedure, Merge-Lists(L, L'), that merges list L and L' into increasing order.

A procedure for subset sum is given by

Exact-Subset-Sum (S,t)

n←|S|

L_0← <0>

for i←1 to n do {L_i←Merge_Lists(L_{i-1}, $L_{i-1}+x_i$); remove values >t from L_i}

return the largest value from L_i

The idea is simple; we keep track of every possible set that is not too large. {See text section 35.5 for example}. $S = \{1,4,5\}$, $P_1 = \{0,1\}$, $P_2\{0,1,4,5\}$, $P_3\{0,1,4,5,6,9,10\}$.

The method is clearly exponential in general, but if t is small (polynomial in $|S|$) then it is polynomial.
A *fully polynomial time approximation scheme*: The key idea is to keep the size of L_i down to a reasonable size by "trimming" out values that are close to ones we retain. Hence we don't miss the optimal by too much.

Trim L by δ means remove as many elements as possible, giving L', so that for every y removed, there is still a $z \le y$ in L' so $\frac{y}{1+\delta} \le z \le y$. Note this removal of a y guarantees there is an "acceptable" z that is not too much worse.

Example from text at $\delta = 0.1$

$$L = \langle 10,11,12,15,20,21,22,23,24,29 \rangle$$
$$L' = \langle 10,12,15,20,23,29 \rangle$$

Clearly we can give a δ trim in linear time; call the procedure Trim(L,δ). So, the idea is clear, but if we trim by δ on every round the approximation will get worse. Perhaps by an δ each time so the ratio goes to $(1+\delta)^n$. We will use $\in / 2n$ each time for the desired $\in$-approximation (i.e. $C^* \le C(1+\in)$).

```
Approx_subset_sum(s, t, ∈)
    n ← |s|
    L_0 ← ⟨0⟩
    for i ← 1 to n
        do{L_i ← Merge_Lists(L_{i-1}, L_{i-1} + x_i)
           L_i ← Trim(L_i, ∈/(2n))
           remove from L_i every element > t
          }
return maximum value in L_n
```

See text for example on $L = <104,102,01,101>$ with $t = 308$ and $\in = .20$, so $\delta = .20/4 = .05$

Theorem: Approx-Subset-Sum is a fully polynomial-time approximation scheme for the subset sum problem.

Proof: There are two issues runtime and quality of solution. First consider quality of solution.

When L_i is trimmed we introduce a relative error of at most $\in / 2n$ {over and above previous error}. Hence by induction in number of steps

$$(1+ \in / 2n)^{-n} y^* \le z \le y^*$$

where y* denotes the optimal solution so

$$\frac{y^*}{z} \le \left(1+\frac{\in}{2n}\right)^n$$

$$\Rightarrow e^{\in/2} \le 1+\in/2+(\in/2)^2 < 1+\in$$

$$\left[as \quad e^x + 1 = x + \frac{x^2}{2!} + \frac{x^3}{3!}...\right]$$

Now consider the size of a list:
Ratio of two consecutive values z and z', $z/z' > (1+\in/2n)$
And values range from 1 to t. Hence number of values is at most

$$2+\log_{1+\in/2n} t$$

$$\le 2+\ln t/\ln(1+\in/2n) \text{ (see note below)}$$

$$\le 2+\frac{2n(1+\in/2n)\ln t}{\in} \text{ [Just take} \in < 1]$$

$$\le \frac{4n\ln t}{\in}+2 \text{ which is polynomial}$$

in problem size (at most square) and $\in$.

Note: use

$$\ln(1+x) = x-\frac{x^2}{2}+\frac{x^3}{3}-\frac{x^4}{4}...$$

$$\text{and} \quad \frac{x}{1+x} = x-x^2+x^3-x^4...$$

$$\text{so} \quad \frac{x}{1+x} \le \ln(1+x) \le x$$