

# CS 467/567: Approximation algorithms and other ways to cope with NP-completeness

Stefan D. Bruda

Winter 2020



- Some times we do not really need to solve the original problem
  - A less general variant might do (and might be easy)
  - Example: 2-SAT versus the full-blown SAT
  - Example: most problems on graphs become easy when the graph is a tree



- Some times we do not really need to solve the original problem
  - A less general variant might do (and might be easy)
  - Example: 2-SAT versus the full-blown SAT
  - Example: most problems on graphs become easy when the graph is a tree
- Some other times we can work with a less than perfect solution
  - A “good enough” solution will do instead
  - Pertinent to optimization problems
  - **(1 + ε)-approximation algorithm A:**

$$\frac{|opt(x) - A(x)|}{opt(x)} \leq \varepsilon$$

- $\mathcal{NP}$ -complete problems can be
  - **Fully approximable:** have  $(1 + \varepsilon)$ -approximation algorithms for arbitrarily small  $\varepsilon$
  - **Partly approximable:**  $(1 + \varepsilon)$ -approximation algorithms exist for some  $\varepsilon$  but not all the way to 0
  - **Inapproximable:** no  $(1 + \varepsilon)$ -approximation algorithm exists (unless  $\mathcal{P} = \mathcal{NP}$ )



- Some times we do not really need to solve the original problem
  - A less general variant might do (and might be easy)
  - Example: 2-SAT versus the full-blown SAT
  - Example: most problems on graphs become easy when the graph is a tree
- Some other times we can work with a less than perfect solution
  - A “good enough” solution will do instead
  - Pertinent to optimization problems
  - **(1 + ε)-approximation algorithm A:**

$$\frac{|opt(x) - A(x)|}{opt(x)} \leq \varepsilon$$

- $\mathcal{NP}$ -complete problems can be
  - **Fully approximable:** have (1 + ε)-approximation algorithms for arbitrarily small ε
  - **Partly approximable:** (1 + ε)-approximation algorithms exist for some ε but not all the way to 0
  - **Inapproximable:** no (1 + ε)-approximation algorithm exists (unless  $\mathcal{P} = \mathcal{NP}$ )
- Some other times solving the original problem is a must
  - Use algorithms with exponential running time in general but that often do much better



- An algorithm that for any input of size  $n$  produces a solution  $C$  instead of the optimal solution  $C^*$  has an **approximation ratio**  $\rho(n)$  if

$$\max \left\{ \frac{|C|}{|C^*|}, \frac{|C^*|}{|C|} \right\} \leq \rho(n)$$

- Such an algorithm is called a  **$\rho(n)$ -approximation algorithm**
- **Approximation scheme**: An algorithm that is a  $(1 + \varepsilon)$ -approximation algorithm for any  $\varepsilon > 0$ 
  - **Polynomial-time approximation scheme**: An approximation scheme whose running time is polynomial in the size of the input for any fixed  $\varepsilon > 0$
  - **Fully polynomial-time approximation scheme**: An approximation scheme whose running time is polynomial in both the size of the input and  $\varepsilon$



- Given a graph  $G$  find the minimal vertex cover

**Algorithm APPROX-VERTEX-COVER** ( $G = (V, E)$ ):

- $C \leftarrow \emptyset, E' \leftarrow E$
- while**  $E' \neq \emptyset$  **do**
  - pick some  $(u, v) \in E'$
  - $C \leftarrow C \cup \{u, v\}$
  - remove from  $E'$  every edge incident to either  $u$  or  $v$
- return**  $C$



# VERTEX COVER

- Given a graph  $G$  find the minimal vertex cover

**Algorithm** APPROX-VERTEX-COVER ( $G = (V, E)$ ):

- $C \leftarrow \emptyset, E' \leftarrow E$
- while**  $E' \neq \emptyset$  **do**
  - pick some  $(u, v) \in E'$
  - $C \leftarrow C \cup \{u, v\}$
  - remove from  $E'$  every edge incident to either  $u$  or  $v$
- return**  $C$

## Theorem

APPROX-VERTEX-COVER is a polynomial time 2-approximation algorithm

- Need to prove that the algorithm (a) runs in polynomial time, (b) return a vertex cover, and (c) the returned cover is not worse than twice the optimal one
- This algorithm is the best approximation algorithm known for the vertex cover problem
- There exist a relatively recent proof that no  $(1 + \varepsilon)$ -approximation algorithm exists for this problem for any  $\varepsilon < 1/6$



- Given a complete graph  $G = (V, E)$  and a cost function  $c : E \rightarrow \mathbb{R}$ , find a Hamiltonian cycle of minimum cost
- Simplifying assumption: cutting intermediate stops never increases the cost, or  $\forall u, v, w \in V : c(u, w) \leq c(u, v) + c(v, w)$

**Algorithm** APPROX-TSP ( $G = (V, E), c$ ):

- 1 Pick  $r \in V$  (the “root” vertex)
- 2 compute the **minimum spanning tree**  $T$  for  $G$  from  $r$
- 3 **return**  $H$ , the list of vertices of  $G$  ordered according to the preorder walk of  $T$

## Theorem

APPROX-TSP is a polynomial time 2-approximation algorithm for TSP with triangle inequality





## Theorem

If  $\mathcal{P} \neq \mathcal{NP}$  then for any  $\varepsilon > 0$  there exists no polynomial-time  $(1 + \varepsilon)$ -approximation algorithm for the traveling salesman problem

- Suppose that we have a  $\rho = (1 + \varepsilon)$ -approximation algorithm  $A$  for some  $\varepsilon \in \mathbb{N}$ ; we then show how to use this algorithm to solve HAMILTONIAN-CYCLE
- Given  $G = (V, E)$  let  $G' = (V, E')$  with  $E' = \{(u, v) \in V \times V : u \neq v\}$ ; let

$$c(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E \\ \rho|V| + 1 & \text{otherwise} \end{cases}$$

- If  $G$  has a Hamiltonian cycle then  $(G', c)$  contains a tour of cost  $|V|$  and so  $A$  will return a tour of cost  $\rho|V|$  or less for  $(G', c)$
- If  $G$  does not have a Hamiltonian cycle then any tour in  $(G', c)$  costs at least  $\rho|V|$  and so  $A$  will return a tour of cost larger than  $\rho|V|$  for  $(G', c)$
- $A$  thus effectively solves HAMILTONIAN-CYCLE in polynomial time □
- **General technique for proving that certain problems do not approximate well!**



- Given a set of integers  $S = \{x_1, x_2, \dots, x_n\}$  and an integer  $t$ , find a subset  $S' \subseteq S$  with  $s = \sum_{x \in S'} x$  such that (a)  $s \leq t$  and (b)  $s$  is maximized
- Exact algorithm (exponential running time): Iterate the from 1 to  $n$ , perform the following for iteration  $i$ :
  - Compute the list  $L_i$  of the sums of all the subsets of  $\{x_1, \dots, x_i\}$  using  $L_{i-1}$ :
    - 1 Add  $x_i$  to all the elements of  $L_{i-1}$  obtaining the list  $L$
    - 2
    - 3 Merge  $L$  and  $L_{i-1}$  thus obtaining  $L_i$
  - Delete from  $L_i$  all the sums that are larger than  $t$



- Given a set of integers  $S = \{x_1, x_2, \dots, x_n\}$  and an integer  $t$ , find a subset  $S' \subseteq S$  with  $s = \sum_{x \in S'} x$  such that (a)  $s \leq t$  and (b)  $s$  is maximized
- Exact algorithm (exponential running time): Iterate the from 1 to  $n$ , perform the following for iteration  $i$ :
  - Compute the list  $L_i$  of the sums of all the subsets of  $\{x_1, \dots, x_i\}$  using  $L_{i-1}$ :
    - Add  $x_i$  to all the elements of  $L_{i-1}$  obtaining the list  $L$
    - 
    - Merge  $L$  and  $L_{i-1}$  thus obtaining  $L_i$
  - Delete from  $L_i$  all the sums that are larger than  $t$
- Approximation algorithm: as above, but **trim** the list  $L_i$  in the (previously empty) Step 2
  - If two values in  $L_i$  are “close enough” to each other then only one is kept
  - Given  $0 < \delta < 1$  for each element  $y$  removed from  $L_i$  there exists an element  $z$  still in  $L_i$  such that  $y/(1 + \delta) \leq z \leq y$



**Algorithm** TRIM ( $L, \delta$ ):

- 1 let  $m$  be the length of  $L$
- 2  $L' \leftarrow \langle y_1 \rangle, last \leftarrow y_1$
- 3 **for**  $i = 1$  **to**  $m$  **do**  
  **if**  $y_i > last \times (1 + \delta)$  **then** (no need to test for  $y_i < last$  since  $L$  is sorted)
  - 1 append  $y_i$  to the end of  $L'$
  - 2  $last \leftarrow y_i$
- 4 **return**  $L'$

## Theorem

*The algorithm just described with  $\delta = \epsilon/2n$  is a fully polynomial approximation scheme for the subset sum problem*



# BACKTRACKING

**Algorithm** BACKTRACKING( $S_0$ : problem)

- 1 OPEN  $\leftarrow \{S_0\}$
  - 2 **while** OPEN  $\neq \emptyset$  **do**
    - 1 choose a sub-problem  $S$  from OPEN and remove it from OPEN
    - 2 choose a way of splitting  $S$  into sub-problems  $S_1, S_2, \dots, S_n$   
[such that a solution for any  $S_i$  is also a solution for  $S$ ]
    - 3 **foreach**  $S_i \in \{S_1, \dots, S_n\}$  **do**
      - 1 **if** TEST( $S_i$ ) **then return** solution for  $S_i$
      - 2 **else** add  $S_i$  to OPEN
  - 3 **return** “no solution”
- Example of algorithm that has exponential running time in general but does much better in most instances
  - Varied strategies of traversing sub-problems (each with advantages and disadvantages)
  - How do we add the sub-problems  $S_1, S_2, \dots, S_n$  back to OPEN?



## Algorithm BACKTRACKING( $S_0$ : problem)

- 1 OPEN  $\leftarrow \{S_0\}$
  - 2 while OPEN  $\neq \emptyset$  do
    - 1 choose a sub-problem  $S$  from OPEN and remove it from OPEN
    - 2 choose a way of splitting  $S$  into sub-problems  $S_1, S_2, \dots, S_n$   
[such that a solution for any  $S_i$  is also a solution for  $S$ ]
    - 3 foreach  $S_i \in \{S_1, \dots, S_n\}$  do
      - 1 if TEST( $S_i$ ) then return solution for  $S_i$
      - 2 else add  $S_i$  to OPEN
  - 3 return "no solution"
- Example of algorithm that has exponential running time in general but does much better in most instances
  - Varied strategies of traversing sub-problems (each with advantages and disadvantages)
  - How do we add the sub-problems  $S_1, S_2, \dots, S_n$  back to OPEN?
    - At the beginning  $\rightarrow$  **depth-first** computation
    - At the end  $\rightarrow$  **breath-first** computation



- Basic backtracking has a straightforward **recursive definition**

**Algorithm** BACKTRACKING( $S$ : problem)

- 1 **if** TEST( $S$ ) **then return** solution for  $S$
  - 2 **else**
    - 1 choose a way of splitting  $S$  into sub-problems  $S_1, S_2, \dots, S_n$
    - 2 combine BACKTRACKING( $S_1$ ),  $\dots$ , BACKTRACKING( $S_n$ ) and return the result
- Depth-first computation (might not be able to find a solution), but
  - Eliminates the need for storing OPEN (substantial savings)
  - Issues specific to every particular problem:
    - How to split into sub-problems
    - How to test for elementary solutions



- Backtracking is especially efficient for decision problems
- For more complex (namely, optimization) problems we can do even better:

**Algorithm** BRANCH-AND-BOUND( $S_0$ : *problem*)

- 1  $A \leftarrow \{S_0\}$ , *bestsofar* =  $\infty$
  - 2 **while** *A* is not empty **do**
    - 1 choose a sub-problem *S* from *A* and remove it from *A*
    - 2 choose a way of branching out *S* into sub-problems  $S_1, S_2, \dots, S_n$
    - 3 **foreach**  $S_i \in \{S_1, \dots, S_n\}$  **do**
      - 1 **if**  $S_i$  is a complete solution **then** update *bestsofar*
      - 2 **else if** LOWERBOUND( $S_i$ ) < *bestsofar* **then** add  $S_i$  to *A*
  - 3 **return** solution associated with *bestsofar*
- Same design issues, plus how to compute LOWERBOUND
  - Other methods include heuristics, local improvements
    - Really the realm of **artificial intelligence**