# Approximation Algorithms

## Introduction

- Approach to attacking NP-hard problems
  - NP-hard problems cannot be solved in polynomial time
    - NP-hard problems may have practical value but are solvable in exponential time, at best
      1. May be acceptable for small input size
      2. Possible to isolate special cases that are solvable in polynomial time
      3. Possible to find *near-optimal* solution in polynomial time
    - We may want to get a solution to those problems in polynomial time
    - Some solutions may be of sub-exponential complexity, such as $2^{n/c}$ for $c > 1$, $2^{\sqrt{n}}$, or $n^{\log n}$
    - Availability of sub-exponential complexity solution allows for solution of problems with larger data sets
    - $O(n^4)$ complexity may not be good for large data sets; we'll prefer $O(n)$ or $O(n^2)$ complexity algorithms
  - Non-optimal solution in polynomial time
    - Often good enough for practical purposes
    - Known as approximation algorithms
    - Example: Consider the AC Coupling effect in sensor simulation, typically reduced to

$$y_n = \frac{T}{T+1}\left(x_n - x_{n-1} + y_{n-1}\right)$$

      - Rewrite the equation by substituting $c = \frac{T}{T+1}$ as

$$y_n = c(x_n - x_{n-1} + y_{n-1})$$

      - Limiting the recursion to $k$ steps yields a kernel that can be implemented to perform the effect as

$$y_n = c \cdot x_n - \sum_{i=0}^{k-1} c^{k-i}(1-c) \cdot x_i$$

  - Comparison with heuristics
    - Heuristics find reasonably good solution in a reasonable time
    - Heuristics may be based on isolating important special cases that can be solved in polynomial time
    - Heuristics are not guaranteed to work on every instance of the problem data set
      - Exponential algorithms may still show exponential behavior even when used with heuristics
    - Approximation algorithms give provable solution quality in provable run-time bounds
    - Approximation is optimal up to a small constant factor (like 5%)
  - Relax the meaning of "solve"
    - Remove the requirement that the algorithm must always generate an optimal solution
      - Replace it with the requirement that the algorithm must always generate a feasible solution with value close to the value of an optimal (approximate) solution
      - For NP-hard problems, each optimal solution may not be obtainable in reasonable computation time
    - A second goal is to get an algorithm that *almost always* generates optimal solution; such algorithm is called probabilistically good algorithm

- Performance ratios for approximation algorithms
  - A problem (knapsack, traveling salesperson) is denoted by $P$
  - Let $I$ be an instance of problem $P$

- Cost of optimal solution for instance $I$ is given by $C^*(I)$, $C^*(I) > 0$
- Optimal solution may be defined as maximum or minimum possible cost (maximization or minimization problem)
- Cost of approximate solution $\hat{C}(I)$
    * $\hat{C}(I) < C^*(I)$ if P is a maximization problem
    * $\hat{C}(I) > C^*(I)$ if P is a minimization problem

**Definition 1** *A is an **absolute approximation algorithm** if and only if for every instance $I$ of problem $P$, $|C^*(I) - C(I)| \leq k$ for some constant $k$*

- Approximation ratio $\rho(n)$
    * Given input of size $n$
    * $\hat{C}(I)$ is within a factor $\rho(n)$ of $C^*(I)$ if

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

    * Another measure of approximation is given in literature as

$$\frac{|C^*(I) - \hat{C}(I)|}{C^*(I)} \leq \rho(n)$$

**Definition 2** *A is a **$\rho(n)$-approximate algorithm** if and only if for every instance of size $n$, the algorithm achieves an approximation ratio of $\rho(n)$*

* Applies to both maximization ($0 < \hat{C}(I) \leq C^*(I)$) and minimization ($0 < C^*(I) \leq \hat{C}(I)$) problems because of the maximization factor, and because the costs are positive
* $\rho(n) > 1$

**Definition 3** *An **$\epsilon$-approximation algorithm** is a $\rho(n)$-approximation algorithm for which $\rho(n) \leq \epsilon$ for some constant $\epsilon$*

* 1-approximation implies $\hat{C}(I) = C^*(I)$, resulting in an optimal solution
* An approximation algorithm with a large $\rho(n)$ may return a solution that is far worse than optimal

- Approximation scheme
    * Tradeoff between computation time and quality of approximation
    * An algorithm may achieve increasingly smaller $\rho(n)$ using more and more computation time
    * Approximation algorithm takes a value $\epsilon > 0$ as an additional input such that for any fixed $\epsilon$, the scheme is a $(1 + \epsilon)$-approximation algorithm

**Definition 4** *$A(\epsilon)$ is an **approximation scheme** if and only if for every given $\epsilon > 0$ and problem instance $I$, $A(\epsilon)$ generates a feasible solution such that $\frac{|C^*(I) - \hat{C}(I)|}{C^*(I)} \leq \epsilon$, assuming $C^*(I) > 0$*

**Definition 5** *An approximation scheme is a **polynomial-time approximation scheme** if and only if for any fixed $\epsilon > 0$, the scheme runs in time polynomial in the size $n$ of input instance*

* Running time of a polynomial-time approximation scheme can increase rapidly as $\epsilon$ decreases
    · Assume running time of a polynomial-time approximation scheme is $O(n^{2/\epsilon})$
    · Ideally, if $\epsilon$ decreases by a constant factor, the run time for desired aproximation should not increase by a constant factor
    · Preferable to have run time polynomial in $1/\epsilon$ as well as in $n$

**Definition 6** *An approximation scheme whose computing time is a polynomial both in problem size and in $1/\epsilon$ is a **fully polynomial-time approximation scheme***

* Example running time for scheme: $O((1/\epsilon)^2 n^3)$

        * Allows a constant-factor decrease in $\epsilon$ with a corresponding constant-factor increase in running-time
    - Absolute approximation algorithm is the most desirable approximation algorithm
        * For most NP-hard problems, fast algorithms of this type exists only if $\mathcal{P} = \text{NP}$
    - Example: Knapsack problem
        * $n = 3$, $m = 100$, $P = \{20, 10, 19\}$, $W = \{65, 20, 35\}$
        * Solution $X = \{1, 1, 1\}$ is not feasible
        * Solution $X = \{1, 0, 1\}$ is optimal, with $\sum p_i x_i = 39$
        * We have $C^*(I) = 39$ for this instance
        * Solution $X = \{1, 1, 0\}$ is suboptimal with $\sum p_i x_i = 30$
            · Candidate for a possible output from an approximation algorithm
            · $\hat{C}(I) = 30$
            · $\rho(n) = \max\left(\frac{30}{39}, \frac{39}{30}\right) = 1.3$ for this instance
            · $\frac{|C^*(I) - \hat{C}(I)|}{C^*(I)} = \frac{39-30}{39} = 0.23$
        * Perform the computation with other feasible solution: $\{0, 1, 1\}$
    - Example: Knapsack problem
        * Assume the objects are in nonincreasing order of $p_i/w_i$
        * If object $i$ fits, set $x_i$ to 1; otherwise set $x_i$ to 0
        * Instance 1
            · $n = 2$, $m = 4$, $P = \{100, 20\}$, $W = \{4, 1\}$
            · Objects considered in the order $1, 2$
            · Solution is: $X = \{1, 0\}$, which is optimal
        * Instance 2
            · $n = 2$, $m = r$, $P = \{2, r\}$, $W = \{1, r\}$
            · When $r > 2$, optimal solution is $X = \{0, 1\}$, and $C^*(I) = r$
            · Solution generated by approximation algorithm is $X = \{1, 0\}$, with $\hat{C}(I) = 2$
            · $\frac{|C^*(I) - \hat{C}(I)|}{C^*(I)} = \frac{r-2}{r}$
            · The approximation algorithm is not an absolute approximation as there is no $k$ such that $\frac{|C^*(I) - \hat{C}(I)|}{C^*(I)} \leq k$ for all instances $I$
            · Also notice that $\frac{|C^*(I) - \hat{C}(I)|}{C^*(I)} = 1 - \frac{2}{r}$; this approaches 1 as $r$ becomes large
            · $\frac{|C^*(I) - \hat{C}(I)|}{C^*(I)} \leq 1$ for every feasible solution to every knapsack instance
            · Since this algorithm always generates a feasible solution, it is a 1-approximate algorithm; it is not $\epsilon$-approximate for any $\epsilon$, $\epsilon < 1$

**Vertex cover problem**

- A vertex cover of an undirected graph $G = (V, E)$ is a subset $V'$ of the vertics of the graph which contains at least one of the two endpoints of each edge

$$V' \subseteq V : \forall \{a, b\} \in E, a \in V' \vee b \in V'$$

    - The size of a vertex cover is the number of vertices in it
- Optimization problem of finding a vertex cover of minimum size (*optimal vertex cover*) in a graph
    - Useful in circuit design
    - NP-complete problem
- Decision problem: Given a graph $G$ and a positive integer $k$, is there a vertex cover of size less than or equal to $k$

- 2-approximate algorithm for vertex cover

```
algorithm approx_vertex_cover ( G )
{
    // Input:  Undirected graph G
    // Output: Vertex cover whose size is guaranteed to be no more than twice
    //         the size of optimal vertex cover

    C = NULL                // Current vertex cover
    E' = E[G]               // All edges in the graph
    while ( E' != NULL )
    {
        e = (u,v)           // Select an arbitrary edge from E'
        C = C + e           // Add the selected edge to C (set union)
        Remove from E' every edge incident on either u or v
    }
    return ( C )
}
```

- Example
  - Graph with six vertices $V = \{a, b, c, d, e, f, g\}$ and edges given by $E = \{ab, bc, cd, ce, de, df, dg\}$
  - Approximate vertex cover is given by $\{b, c, d, e, f, g\}$
  - Optimal vertex cover is given by $\{b, d, e\}$
- Run-time of vertex cover approximation algorithm is $O(V + E)$, using adjacency lists to represent $E'$

**Theorem 1** *`approx_vertex_cover` is a polynomial-time 2-approximation algorithm.*

  - Easy to see that `approx_vertex_cover` runs in polynomial time
  - Set of vertices $C$ is a vertex cover since algorithm loops until every edge in $E$ has been covered by some vertex in $C$
  - Proving that $C$ is at most twice the size of optimal cover
    - Let $A$ be the set of edges selected by the algorithm (first statement in loop)
    - To cover the edges in $A$, any vertex cover must include at least one endpoint of each edge in $A$
    - No two edges in $A$ share an endpoint
      - Once an edge is picked up in $A$, all other edges incident on its endpoints are deleted from $E'$
    - No two edges are covered by the same vertex from optimal cover $C^*$ giving us the lower bound as $|C^*| \geq |A|$
    - Each iteration of the loop picks an edge with neither endpoints in $C$, giving an upper bound on the vertex cover as $|C| = 2|A|$
    - From those two observations

$$\begin{aligned} |C| &= 2|A| \\ &\leq 2|C^*| \end{aligned}$$

  - We do not require that we know $|C^*|$ exactly; instead, we rely on the lower bound on the size

## Absolute approximations

- Planar graph coloring
  - Graph coloring

    * Assignment of colors (or labels) to vertices in a graph such that no two adjacent vertices share the same color
- Determine the minimum number of colors needed to color a planar graph $G = (V, E)$
- A graph is 0-colorable iff $V = \emptyset$
- A graph is 1-colorable iff $E = \emptyset$
- A graph is 2-colorable iff it is bipartite
- Determining whether a graph is 3-colorable is $\mathcal{NP}$-hard
- It is known that every planar graph is four colorable
- It is easy to obtain an algorithm with $|C^*(I) - \hat{C}(I)| \leq 1$

```
algorithm acolor ( V, E )
{
    // Determine an approximation to the minimum number of colors
    // Input:  Set of vertices and set of edges
    // Output: Number of colors

    if ( V == NULL )   return 0;
    if ( E == NULL )   return 1;
    if ( G is bipartite ) return 2;
    return ( 4 );
}
```

    * We can determine that a graph is bipartite in time $O(|V| + |E|)$
    * Complexity of `acolor` is $O(|V| + |E|)$