Approximation Algorithms
Approximation Ratios

Optimization Problems

- We have some problem instance $x$ that has many feasible “solutions”.
- We are trying to minimize (or maximize) some cost function $c(S)$ for a “solution” $S$ to $x$. For example,
  - Finding a minimum spanning tree of a graph
  - Finding a smallest vertex cover of a graph
  - Finding a smallest traveling salesperson tour in a graph
Approximation Ratios

- An approximation produces a solution $T$
  - $T$ is a **k-approximation** to the optimal solution $OPT$ if $c(T)/c(OPT) \leq k$ (assuming a min. prob.; a maximization approximation would be the reverse)
  - Put another way, $c(T) \leq k \cdot c(OPT)$
  - We would have $c(T) \geq (1/k) \cdot c(OPT)$
  - In both cases we assume $k > 1$. 
**Special Case of the Traveling Salesperson Problem**

**OPT-TSP:** Given a complete, weighted graph, find a cycle of minimum cost that visits each vertex.

- OPT-TSP is NP-hard
- Special case: edge weights satisfy the triangle inequality (which is common in many applications):
  - \( w(a,b) + w(b,c) \geq w(a,c) \)
Aside: Euler Tour of a Rooted Tree

Input:

Output: 1 5 4 2 4 3 4 5 1
A 2-Approximation for TSP
Special Case

Algorithm $TSP_{Approx}(G)$

- **Input** weighted complete graph $G$, satisfying the triangle inequality
- **Output** a TSP tour $T$ for $G$

1. $M \leftarrow$ a minimum spanning tree for $G$
2. $P \leftarrow$ an Euler tour traversal of $M$, starting at some vertex $s$
3. $T \leftarrow$ empty list

   for each vertex $v$ in $P$ (in traversal order)

      if this is $v$’s first appearance in $P$ then

      $T$.insertLast$(v)$

   $T$.insertLast$(s)$

return $T$
A 2-Approximation for TSP
Special Case - Proof

- The optimal tour is a spanning tour; hence $|M| \leq |OPT|$.
- The Euler tour $P$ visits each edge of $M$ twice; hence $|P| = 2|M|$.
- Each time we shortcut a vertex in the Euler Tour we will not increase the total length, by the triangle inequality $(w(a,b) + w(b,c) \geq w(a,c))$; hence, $|T| \leq |P|$.
- Therefore, $|T| \leq |P| = 2|M| \leq 2|OPT|$.

Output tour $T$  
(at most the cost of $P$)

Euler tour $P$ of MST $M$  
(twice the cost of $M$)

Optimal tour $OPT$  
(at least the cost of MST $M$)
Vertex Cover

- A **vertex cover** of graph $G=(V,E)$ is a subset $W$ of $V$, such that, for every $(a,b)$ in $E$, $a$ is in $W$ or $b$ is in $W$.

- **OPT-VERTEX-COVER**: Given a graph $G$, find a vertex cover of $G$ with smallest size.

- **OPT-VERTEX-COVER** is NP-hard.
A 2-Approximation for Vertex Cover

- Every chosen edge $e$ has both ends in $C$
- But $e$ must be covered by an optimal cover; hence, one end of $e$ must be in OPT
- Thus, there is at most twice as many vertices in $C$ as in OPT.
- That is, $C$ is a 2-approx. of OPT
- Running time: $O(n+m)$

**Algorithm** `VertexCoverApprox(G)`:  
**Input:** A graph $G$  
**Output:** A small vertex cover $C$ for $G$  

1. $C \leftarrow \emptyset$
2. **while** $G$ still has edges **do**
   - select an edge $e = (v, w)$ of $G$
   - add vertices $v$ and $w$ to $C$
   - **for** each edge $f$ incident to $v$ or $w$ **do**
     - remove $f$ from $G$
3. **return** $C$
An Aside: Harmonic Numbers

The \( n \)th harmonic number \( H_n \) is defined as

\[
H_n = \sum_{i=1}^{n} \frac{1}{i}.
\]

**Theorem A.16:** If \( H_n \) is the \( n \)th harmonic number, for \( n > 1 \), then \( \ln n < H_n < \ln n + 1 \).

**Solution:** For the upper bound,

\[
H_n = 1 + \sum_{i=2}^{n} \frac{1}{i} \leq 1 + \int_{x=1}^{n} \frac{dx}{x} = 1 + \ln n.
\]

For the lower bound,

\[
H_n \geq \sum_{i=1}^{n-1} \frac{1}{i} \geq \int_{x=1}^{n} \frac{dx}{x} = \ln n.
\]
Set Cover (Greedy Algorithm)

OPT-SET-COVER: Given a collection of \( m \) sets, find the smallest number of them whose union is the same as the whole collection of \( m \) sets?

- OPT-SET-COVER is NP-hard
- Greedy approach produces an \( O(\log n) \)-approximation algorithm.

**Algorithm SetCoverApprox(\( S \))**:

- **Input**: A collection \( S \) of sets \( S_1, S_2, \ldots, S_m \) whose union is \( U \)
- **Output**: A small set cover \( C \) for \( S \)

\[
\begin{align*}
C &\leftarrow \emptyset \quad \text{// The set cover built so far} \\
E &\leftarrow \emptyset \quad \text{// The elements from \( U \) currently covered by \( C \)} \\
\text{while } E &\neq U \text{ do} \\
&\quad \text{select a set } S_i \text{ that has the maximum number of uncovered elements} \\
&\quad \text{add } S_i \text{ to } C \\
&\quad E \leftarrow E \cup S_i \\
\text{Return } C. \\
\end{align*}
\]
Greedy Set Cover Analysis

Consider the moment in our algorithm when a set $S_j$ is added to $C$, and let $k$ be the number of previously uncovered elements in $S_j$.

We pay a total charge of 1 to add this set to $C$ (meaning we made our cover one subset larger), so we charge each previously uncovered element $i$ of $S_j$ a charge of $c(i) = 1/k$.

Thus, the total size of our cover is equal to the total charges made.

To prove an approximation bound, we will consider the charges made to the elements in each subset $S_j$ that belongs to an optimal cover, $C'$. So, suppose that $S_j$ belongs to $C'$.

Let us write $S_j = \{x_1, x_2, \ldots, x_{n_j}\}$ so that $S_j$’s elements are listed in the order in which they are covered by our algorithm.
Now, consider the iteration in which \( x_1 \) is first covered. At that moment, \( S_j \) has not yet been selected; hence, whichever set is selected must have at least \( n_j \) uncovered elements. Thus, \( x_1 \) is charged at most \( 1/n_j \). So let us consider, then, the moment our algorithm charges an element \( x_l \) of \( S_j \). In the worst case, we will have not yet chosen \( S_j \) (indeed, our algorithm may never choose this \( S_j \)). Whichever set is chosen in this iteration has, in the worst case, at least \( n_j - l + 1 \) uncovered elements; hence, \( x_l \) is charged at most \( 1/(n_j - l + 1) \). Therefore, the total amount charged to all the elements of \( S_j \) is at most

\[
\sum_{l=1}^{n_j} \frac{1}{n_j - l + 1} = \sum_{l=1}^{n_j} \frac{1}{l},
\]

which is the familiar **harmonic number**, \( H_{n_j} \). It is well known (for example, see the Appendix) that \( H_{n_j} \) is \( O(\log n_j) \). Let \( c(S_j) \) denote the total charges given to all the elements of a set \( S_j \) that belongs to the optimal cover \( C' \). Our charging scheme implies that \( c(S_j) \) is \( O(\log n_j) \). Thus, summing over the sets of \( C' \), we obtain

\[
\sum_{S_j \in C'} c(S_j) \leq \sum_{S_j \in C'} b \log n_j
\leq b |C'| \log n,
\]

for some constant \( b \geq 1 \). But, since \( C' \) is a set cover,

\[
\sum_{i \in U} c(i) \leq \sum_{S_j \in C'} c(S_j).
\]

Therefore,

\[
|C| \leq b |C'| \log n.
\]
A problem $L$ has a **polynomial-time approximation scheme (PTAS)** if it has a polynomial-time $(1+\varepsilon)$-approximation algorithm, for any fixed $\varepsilon > 0$ (this value can appear in the running time).

0/1 Knapsack has a PTAS, with a running time that is $O(n^3/\varepsilon)$. 
Backtracking and Branch-and-Bound
Back to NP-Completeness

Many hard problems are NP-Complete
- But we still need solutions to them, even if they take a long time to generate

Backtracking and Branch-and-Bound are two design techniques that have proven promising for dealing with NP-Complete problems
- And often these find "good enough" solutions in a reasonable amount of time
Backtracking

A backtracking algorithm searches through a large set of possibilities in a systematic way:
- Typically optimized to avoid symmetries in problem instances (think, e.g., TSP)
- Traverses search space in manner such that "easy" solution is found if one exists.

Technique takes advantage of an inherent structure possessed by many NP-Complete problems.
NP-Complete Problem Structure

Recall: acceptance of solution for instance $x$ of NP-complete problem can be verified in polynomial time.

Verification typically involves checking a set of choices (the output of the choice function) to see whether they satisfy a formula or demonstrate a successful configuration.

- E.g., Values assigned to Boolean variables, vertices of graph to include in special set, items to go in knapsack.
Backtracking

- Systematically searches for a solution to problem
- Traverses through possible "search paths" to locate solutions or dead ends
- Configuration at end of such a path consists of a pair \((x, y)\)
  - \(x\) is the remaining problem to be solved
  - \(y\) is the set of choices that have been made to get to this subproblem from the original problem instance
Backtracking

- Start search with the pair $(x, \phi)$, where $x$ is the original problem instance.
- Any time that backtracking algorithm discovers a configuration $(x, y)$ that cannot lead to a valid solution, it cuts off all future searches from this configuration and backtracks to another configuration.
Backtracking Pseudocode

Algorithm Backtrack(x):

Input: A problem instance x for a hard problem
Output: A solution for x or “no solution” if none exists
F ← { (x, ∅) }. \{ F is the “frontier” set of subproblem configurations \}
while F ≠ ∅ do
    select from F he most promising configuration (x, y)
    expand (x, y) by making a small set of additional choices
    let (x₁, y₁), (x₂, y₂), ... (xₖ, yₖ) be the set of new configurations.
    for each new configuration (xᵢ, yᵢ) do
        perform a simple consistency check on (xᵢ, yᵢ).
        if the check returns “solution found” then
            return the solution derived from (xᵢ, yᵢ)
        if the check returns “dead end” then
            discard the configuration (xᵢ, yᵢ) \{ Backtrack \}
        else
            F ← F ∪ { (xᵢ, yᵢ) } \{ (xᵢ, yᵢ) starts a promising search path \}
return “no solution”
Backtracking: Required Details

- Define way of selecting the most promising candidate configuration from frontier set F.
  - If F is a stack, get depth-first search of solution space. F a queue gives a breadth-first search, etc.
- Specify way of expanding a configuration \((x,y)\) into subproblem configurations
  - This process should, in principle, be able to generate all feasible configurations, starting from \((x,\phi)\)
- Describe how to perform a simple consistency check for configuration \((x,y)\) that returns "solution found", "dead end", or "continue"
Example: CNF-SAT

- **Problem:** Given Boolean formula $S$ in CNF, we want to know if $S$ is satisfiable

- **High level algorithm:**
  - Systematically make tentative assignments to variables in $S$
  - Check whether assignments make $S$ evaluate immediately to 0 or 1, or yield a new formula $S'$ for which we can continue making tentative value assignments

- **Configuration in our algorithm is a pair** ($S', y$) where
  - $S'$ is a Boolean formula in CNF
  - $y$ is assignment of values to variables NOT in $S'$ such that making these assignments to $S$ yields the formula $S'$
Example: CNF-SAT (details)

- Given frontier $F$ of configurations, the most promising subproblem $S'$ is the one with the smallest clause.
  - This formula would be the most constrained of all formulas in $F$, so we expect it to hit a dead end most quickly (if it’s going to hit a dead end).

- To generate new subproblem from $S'$, locate smallest clause in $S'$, pick a variable $x_i$ that appears in that clause, and create two new subproblems associated with assigning $x_i=0$ and $x_i=1$ respectively.
Example: CNF-SAT (details)

- Processing $S'$ for consistency check for assignment of a variable $x_i$ in $S'$
  - First, reduce any clauses containing $x_i$ based on the value assigned to $x_i$
  - If this reduction results in new single literal clause ($x_j$ or $\neg x_j$) we assign value to $x_j$ to make single literal clause satisfied. Repeat process until no single literal clauses
  - If we discover a contradiction (clauses $x_i$ and $\neg x_i$, or an empty clause) we return ``dead end''
  - If we reduce $S'$ all the way to constant 1, return ``solution found'' along with assignments made to reach this
  - Else, derive new subformula $S''$ such that each clause has at least two literals, along with value assignments that lead from $S$ to $S''$ (this is the reduce operation)
Example: CNF-SAT (details)

- Placing this in backtracking template results in algorithm that has exponential worst case run time, but usually does OK
- In fact, if each clause has no more than two literals, this algorithm runs in polynomial time.
Branch-and-Bound

- Backtracking not designed for optimization problems, where in addition to having some feasibility condition that must be satisfied, we also have a cost $f(x)$ to be optimized
  - We’ll assume minimized
- Can extend backtracking to work with optimization problems (result is Branch-and-Bound)
  - When a solution is found, we continue processing to find the best solution
  - Add a scoring mechanism to always choose most promising configuration to explore in each iteration (best-first search)
In addition to the details required for the backtracking design pattern, we add one more to handle optimization:

For any configuration \((x,y)\) we assume we have a lower bound function, \(lb(x,y)\), that returns a lower bound on the cost of any solution that is derived from this configuration.

- Only requirement for \(lb(x,y)\) is that it must be less than or equal to the cost of any derived solution.
- Of course, a more accurate lower bound function results in a more efficient algorithm.
Algorithm Branch-and-Bound($x$):

**Input**: A problem instance $x$ for a hard optimization (minimization) problem

**Output**: An optimal solution for $x$ or “no solution” if none exists

$F \leftarrow \{(x, \emptyset)\}$. {Frontier set of subproblem configurations}.

$b \leftarrow (+\infty, \emptyset)$. {Cost and configuration of current best solution}.

while $F \neq \emptyset$ do

select from $F$ the most promising configuration $(x, y)$

expand $(x, y)$ yielding new configurations $(x_1, y_1), (x_2, y_2), \ldots (x_k, y_k)$

for each new configuration $(x_i, y_i)$ do

perform a simple consistency check on $(x_i, y_i)$.

if the check returns “solution found” then

if the cost $c$ of the solution for $(x_i, y_i)$ beats $b$ then

$b \leftarrow (c, (x_i, y_i))$

else

discard the configuration $(x_i, y_i)$

if the check returns “dead end” then

discard the configuration $(x_i, y_i)$  {Backtrack}

else

if $lb(x_i, y_i)$ is less than the cost of $b$ then

$F \leftarrow F \cup \{(x_i, y_i)\}$  {$(x_i, y_i)$ starts a promising search path}

else

discard the configuration $(x_i, y_i)$  {A “bound” prune}

return $b$
Branch-and-Bound Alg. for TSP

Problem: optimization version of TSP
- Given weighted graph G, want least cost tour
- For edge e, let c(e) be the weight (cost) of edge e

We compute, for each edge e = (v,w) in G, the minimum cost path that begins at v and ends at w while visiting all other vertices of G along the way.

We generate the path from v to w in G – {e} by augmenting a current path by one vertex in each loop of the branch-and-bound algorithm.
Branch-and-Bound Alg. for TSP

- After having built a partial path P, starting, say, at v, we only consider augmenting P with vertices not in P.
- We can classify a partial path P as a "dead end" if the vertices not in P are disconnected in G – {e}.
- We define the lower bound function to be the total cost of the edges in P plus c(e).
  - Certainly a lower bound for any tour built from e and P.
Branch-and-Bound Alg. for TSP

- After having run the algorithm to completion for one edge of $G$, we can use the best path found so far over all tested edges, rather than restarting the current best solution $b$ at $+\infty$.

- Run time can still be exponential, but we eliminate a considerable amount of redundancy.

- There are other heuristics for approximating TSP. We won’t discuss them here.