Coping with NP-Completeness

- Sometimes you are faced with hard problems — problems for which no efficient solutions exist.
- **Step 1**: Try to show that the problem is \textit{NP}-complete
  - This way, you can avoid wasting a lot of time on a fruitless search for an efficient algorithm
- **Step 2a**: Sometimes, you may be able to say “let us solve a different problem”
  - you may be able leverage some special structure of your problem domain that enables a more efficient solution
- **Step 2b**: Other times, you are stuck with a difficult problem and you need to make the best of it.
  - We discuss different coping strategies in such cases.

Intelligent Exhaustive Search

- Exhaustive search will work for almost any problem
- **Hamiltonian Tour**: Consider an edge \( e \).
  - Either \( e = (u, v) \) is part of the tour, in which case you can complete the tour by finding a path from \( u \) to \( v \) in \( G - e \).
  - Or, \( e \) is not part of the tour, in which case you can find the tour by searching \( G - e \).
  - Either case leads to a recurrence \( T(m) = 2T(m - 1) \), i.e., \( T(m) = O(2^m) \). (Here \( m \) is the number of edge in \( G \).)
- **SAT**: Try all \( 2^n \) possible truth assignments to the \( n \) variables in your formula.
  - The key point is to be intelligent in the way this search is conducted, so that the algorithm is faster than \( 2^n \) in practice.

Backtracking

- Depth-first approach to perform exhaustive search
  - In the above example, first try to find a solution that includes \( e \)
  - Looking down further, the algorithm will make additional choices of edges to include: \( e_1, e_2, ..., e_k \)
  - Only when all paths that include \( e \) fail to be Hamiltonian, we consider the alternative (i.e., Hamiltonian path that doesn't include \( e \))
  - Key goal is to recognize and prune failing paths as quickly as possible.
Backtracking Approach for \( SAT \)

- We can improve the worst-case bound by choosing a variable that occurs most times.
  - If it occurs \( k \) times, then you have the recurrence
    \[
    T(n) = 2T(n-k)
    \]
    whose solution is \( O(2^{n/k}) \).
  - Of course, you won’t be able to repeatedly find a variable that occurs \( k \) times, so this solution is meaningless in practice — it just goes to show the exponential pruning effect of a frequently occurring variable.
- Another strategy: pick a clause with fewest number of variables, and pick those variables in sequence.
- \textbf{Exercise:} Show that the backtracking algorithm solves \( 2\text{SAT} \) in polynomial time.

Backtracking Approach for \( SAT \): Complexity

- There are two cases, based on the variable \( w \) chosen for branching:
  - Case 1: Both \( w \) and \( \overline{w} \) occur in the formula. In this case, both branches are present. Moreover, both \( w \) and \( \overline{w} \) are eliminated from the formula at this point, so we have the recurrence
    \[
    T(n) = 2T(n-2) + O(n)
    \]
  - Case 2: Only one of them is present. In this case, only one of the branches needs exploring, so we have the recurrence
    \[
    T(n) = T(n-1) + O(n)
    \]
  - Clearly, case 1 will dominate, so let us ignore case 2. Case 1 yields a solution of \( O(2^{n/2}) \) or \( O(1.414^n) \), which is much better than \( 2^n \).

Branch and Bound

- Generalization of backtracking to support optimization problems.
- Requires a lower bound on the cost of solutions that may result from a partial solution.
  - If the cost is higher than that of a previously encountered solution, then this subproblem need not be explored further.
- Sometimes, we may rely on estimates of cost rather than strict lower bounds.
Branch and Bound for TSP

- Begin with a vertex $a$ — the goal is to compute a TSP that begins and ends at $a$.
- We begin the search by considering an edge from $a$ to its neighbor $x$, another edge from $x$ to a neighbor of $x$, and so on.
- Partial solutions represent a path from $a$ to some vertex $b$, passing through a set $S \subseteq V$ of vertices.
- Completing a partial solution requires the computation of a low cost path from $b$ to $a$ using only vertices in $V - S$.

Illustration of Branch-and-Bound for TSP

Lower bound on costs of partial TSP solutions

- To complete the path from $b$ to $a$, we must incur at least the following costs:
  - Cost of going from $b$ to a vertex in $V - S$, i.e., the minimum weight edge from $b$ to a vertex in $V - S$.
  - Cost of going from a $V - S$ vertex to $a$, i.e., the minimum weight edge from $a$ to a vertex in $V - S$.
  - Minimal cost path in $V - S$ that visits all $v \in V - S$.
  - Note: Lower bound is the cost of MST for $V - S$.
- By adding the above three cost components, we arrive at a lower bound on solutions derivable from a partial solution.

Approximation Algorithms

- Relax optimality requirement: permit approximate solutions.
  - Solutions that are within a certain distance from optimum.
- Not heuristics: Approximate algorithms guarantee that solutions are within a certain distance from optimal.
  - Differs from heuristics that can sometimes return very bad solutions.
- How to define “distance from optimal?”
  - Additive: Optimal solution $S_O$ and the Solution $S_A$ returned by approximation algorithm differ only by a constant.
  - Quality of approximation is extremely good, but unfortunately, most problems don’t admit such approximations.
  - Factor: $S_O$ and $S_A$ are related by a factor.
  - Most known approximation algorithms fall into this category.
Approximation Factors

**Constant:** $S_A \leq kS_O$ for some fixed constant $k$.
- **Examples:** Vertex cover, Facility location, ...

**Logarithmic:** $S_A \leq O(\log^k n) \cdot S_O$.
- **Examples:** Set cover, dominating set, ...

**Polynomial:** $S_A \leq O(n^k) \cdot S_O$.
- **Examples:** Max Clique, Independent set, graph coloring, ...

**PTAS:** $S_A \leq (1 + \epsilon) \cdot S_O$ for any $\epsilon > 0$.
(“Polynomial-time approximation scheme”)

**FPTAS:** PTAS with runtime $O(\epsilon^{-k})$ for some $k$. (“Fully PTAS”)
- **Examples:** Knapsack, Bin-packing, Euclidean TSP, ...

---

**Bin Packing**

**Problem**
Pack objects of different weight into bins that have a fixed capacity in such a way that minimizes bins used.

- Obvious similarity to Knapsack
- Bin-packing is $NP$-hard
- Very good (and often very simple) approximation algorithms exist

---

**First-fit Algorithm**

A simple, greedy algorithm

**FirstFit**(x[1..n])

for $i = 1$ to $n$ do
- Put $x[i]$ into the first open bin large enough to hold it

**Theorem**
All open bins, except possibly one, are more than half-full

**Proof:** Suppose that there are two bins $b$ and $b'$ that are less than half-full. Then, items in $b'$ would have fitted into $b$, and so the FF algorithm would never have opened the bin $b'$ — a contradiction.

**Theorem**
First-fit is optimal within a factor of 2: specifically, $S_A < 2S_O + 1$. 

---

**Best-Fit Algorithm**

- Another simple, greedy algorithm
- Instead of using the first bin that will can hold $x[i]$, use the open bin whose remaining capacity is closest to $x[i]$
  - Prefers to keep bins close to full.
- Factor-2 optimality can established easily.
Other algorithms for Bin-packing

- **First-fit decreasing** strategy first sorts the items so that \( x[i] \geq x[i+1] \) and then runs first-fit.
- **Best-fit decreasing** strategy first sorts the items so that \( x[i] \geq x[i+1] \) and then runs best-fit.
- Both FFD and BFD achieve approximation factors of \( \frac{11}{9}S_O + \frac{6}{9} \).
- Due to the additive term, bin-packing cannot have a PTAS unless \( P = NP \).
- But \( S_A = (1+\epsilon)S_O + 1 \) is easy to achieve for any \( \epsilon > 0 \)

Set Cover

**Problem**
Given a collection \( S_1, \ldots, S_m \) of subsets of \( B \), find a minimum collection \( S_{i_1}, \ldots, S_{i_k} \) such that \( \bigcup_{j=1}^{k} S_{i_j} = B \)

**Greedy Set Cover Algorithm**

\[
GSC(S, B)
cover = \emptyset; \; covered = \emptyset
\]

while covered \( \neq B \) do

Let new be the set in \( S - cover \) containing the maximum number of elements of \( B - covered \) add new to cover; covered = covered \( \cup \) new

return cover

Analysis of Greedy Set Cover

**Theorem**

Greedy set cover is approximate with a factor of \( \ln n \), where \( n = |B| \)

**Proof:**

- Let \( k \) be the size of optimal cover, and \( n_t \) be the number of elements left uncovered after \( t \) steps of \( GSC \)
- These \( n_t \) elements are covered by \( k \) sets in optimal cover \( \Rightarrow \) each of these \( k \) sets must cover at least \( n_t/k \) uncovered elements.
- Thus, \( GSC \) will find at least one set that covers \( n_t/k \) elements.
- This yields the recurrence for bounding uncovered elements:
  \[
  U(t+1) = n_t - n_t/k = n_t(1 - 1/k) = U(t)(1 - 1/k)
  \]
- The solution to recurrence is \( n(1 - 1/k)^t < ne^{-t/k} \)
- Thus, after \( t = k \ln n \) steps, less than 1 (i.e., no) elements uncovered
- Thus, \( GSC \) computes a cover at most \( \ln n \) times the optimal cover.

Vertex Cover

- Note that a vertex cover is a set cover for \((S, E)\), where \( S = \{ (v, u) | v \in V \text{ and } (v, u) \in E \} | v \in V \} \)
- \( S \) contains a set for each vertex; this set lists all edges incident on \( v \)
- Thus \( GSC \) is an approximate algorithm for vertex cover.
- But \( \ln n \) is not a factor to be thrilled about — can we do better?
- Actually, we can do much better! That too with a very simple algorithm.
**Vertex Cover**

Consider any edge \((u, v)\).

- Either \(u\) or \(v\) must belong to any vertex cover.
- If we accept \(S_A = 2S_O\), then we can avoid the guesswork by simply picking both vertices!

**Approximate Vertex Cover Algorithm**

\[
AVC(G = (V, E))
\]

\[
C = \emptyset
\]

\[
\text{while } G \text{ is not empty}
\]

\[
\text{pick any } (u, v) \in E
\]

\[
C = C \cup \{u, v\}
\]

\[
G = G - \{u, v\}
\]

\[
\text{return } C
\]

**Analysis of k-Cluster**

Let \(x\) be the farthest point from \(\mu_1, \ldots, \mu_k\), and let \(r\) the distance to its closest center. Then, we can say:

- Cluster diameter of \(C_1, \ldots, C_k\) is at most \(2r\)
- The distance between any 2 points in \(\{x, \mu_1, \ldots, \mu_k\}\) is at least \(r\).
  - This follows from:
    - how \(\mu_i\)'s was chosen to be the farthest point from \(\mu_j\) for \(j < i\),
    - this distance to \(\mu_i\) must decrease with \(i\), and
    - when \(i = k + 1\), this distance is \(r\)
- Thus, any k-Cluster must have a diameter of at least \(r\)
  - With \(k\) circles, at least two of \(k + 1\) points must be within one of them.
  - This circle's diameter must hence be \(r\) or greater
- Thus, AC is approximate within a factor of 2.

**k-Cluster**

**Problem**

Given \(X = \{x_1, \ldots, x_n\}\) and distances between \(x_i\), partition \(X\) into \(k\) clusters in a way that minimizes maximum cluster diameter.

**Approximate k-Cluster Algorithm (AC)**

Pick any point \(\mu_1 \in X\) as the first cluster center

\[
\text{for } i = 2 \text{ to } k \text{ do}
\]

Choose \(\mu_i\) to be the farthest point from \(\mu_1, \ldots, \mu_{i-1}\)

Create \(k\) clusters \(C_i = \{x \in X | \mu_i\text{ is the closest center to } x\}\)

**Euclidean TSP**

- Our starting point is once again the MST
- Note that no TSP solution can be smaller than MST
  - Deleting an edge from TSP solution yields a spanning tree
- **Simple algorithm:**
  - Start with the MST
Approximating Euclidean TSP: An Illustration

- Start with the MST
- Make a tour that uses each MST edge twice (forward and backward)
  - This tour is like TSP in ending at the starting node, and differs from TSP by visiting some vertices and edges twice

Avoid revisits by short-circuiting to next unvisited vertex
- By triangle inequality, short-circuit distance can only be less than the distance following MST edges.
- Thus, tour length less than 2xMST, i.e., approximate within a factor 2.

Knapsack

\[ knap01(w, v, n, W) \]
\[ V = \sum_{j=0}^{n} v[j] \]
\[ k[0, v] = 0, \forall 0 \leq v \leq V \]
\[ \text{for } j = 1 \text{ to } n \text{ do} \]
\[ \text{for } v = 1 \text{ to } V \text{ do} \]
\[ \text{if } v[j] > v \text{ then } k[j, v] = k[j-1, v] \]
\[ \text{else } k[j, v] = \min(k[j-1, v], k[j-1, v-v[j]] + w[j]) \]
\[ \text{return maximum } v \text{ such that } k[n, v] \leq W \]

- Computes minimum weight of knapsack for a given value.
- Iterates over all possible items and all possible values: \( O(nV) \)
  - we derive a polynomial time approximate algorithm from this

FPTAS for 0-1 Knapsack

\[ knap01FPTAS(w, v, n, W, \epsilon) \]
\[ v'_i = \left[ \frac{v_i}{\max_{0 \leq j \leq n} v_j} \cdot \frac{n}{\epsilon} \right] \text{, for } 1 \leq i \leq n \]
\[ knap01(w, v', n, W) \]

- Rescaling consists of two steps:
  - Express value of each item relative to the most valuable item
    - If we worked with real values, this step won’t change the optimal solution
    - Multiply relative values by a factor \( n/\epsilon \) to get an integer
  - Floor operation introduces an error \( \leq 1 \) in \( v'_i \) (e.g., \( \lfloor 3.99 \rfloor = 3 \))
  - Error in \( knap01 \) output is error in \( \sum v'_i \), which is at most \( n \cdot 1 \)
  - We scale each \( v'_i \) by \( n/\epsilon \), so relative error is \( n/(n/\epsilon) = \epsilon \)
  - Thus we have achieved the desired approximation.
FPTAS for 0-1 Knapsack: Runtime

\[ \text{Knap01FPTAS}(w, v, n, W, \epsilon) \]

\[ v'_i = \left\lfloor \frac{v_i}{\max_{1 \leq j \leq n} v_j} \cdot \frac{n}{\epsilon} \right\rfloor, \text{ for } 1 \leq i \leq n \]

Note that we are using \textit{Knap01} with rescaled values, so the complexity is \( O(nV') \).

Note: \( V' = \sum_i^n v'_i \leq n \cdot \max_{1 \leq j \leq n} v'_j \)

It is easy to see from definition of \( v'_i \) that \( \max_{1 \leq j \leq n} v'_j = n/\epsilon \).

Substituting this into the above equation yields a complexity of:

\[ O(nV') \leq O(n(n \cdot \max_{1 \leq j \leq n} v'_j)) = O(n(n \cdot (n/\epsilon))) = O(n^3/\epsilon) \]

By varying \( \epsilon \), we can trade off accuracy against runtime.