Approximation Algorithms

- Main issues
  - Finding a good lower or upper bound for the optimal solution value
  - Relating your algorithm's solution value to this bound
  - Examples: Scheduling and Vertex Cover
- Basic Setting
  - Working with an optimization problem, not a decision problem, where you are trying to minimize or maximize some value
  - Examples:
    - Vertex cover
      - Input: Graph $G = (V, E)$
      - Task: Find a vertex cover of minimum size
      - Value to be minimized: vertex cover size
    - Independent Set
      - Input: Graph $G = (V, E)$
      - Task: Find an independent set of maximum size
      - Value to be maximized: independent set size
  - The problems are **NP**-hard
    - The decision version of all these problems is **NP**-complete
    - Thus, we probably cannot find a polynomial time algorithm that solves the problem optimally for all input instances.
  - Notation
    - Let Π be the problem under consideration
    - Let I be an input instance of Π
    - let OPT denote the optimal algorithm
    - Let A denote the algorithm under consideration
    - For any algorithm A and any input instance I, let $A(I)$ denote the value of A's solution for input instance I
  - Absolute approximation of c (c-absolute-approximation algorithm)
    - $\exists c$ such that $\forall I A(I) \leq OPT(I) + c$ for minimization problem Π
      - A is a 2-absolute-approximation algorithm for vertex cover if A can always find a vertex cover at most 2 nodes larger than the optimal size.
    - $\exists c$ such that $\forall I A(I) \geq OPT(I) - c$ for maximization problem Π
      - A is a 2-absolute-approximation algorithm for independent set if A can always find an independent set at most 2 nodes smaller than the optimal size.
    - Very few **NP**-hard problems have polynomial-time absolute approximation algorithms

- Relative approximation (c-approximation algorithm)
    - $\exists c$ such that $\forall I A(I) \leq c \times OPT(I)$ for minimization problem Π
        - For example, A is a 2-approximation algorithm for vertex cover if A can always find a vertex cover of at most twice the optimal size.
    - $\exists c$ such that $\forall I A(I) \geq (1/c) \times OPT(I)$ for maximization problem Π
        - For example, A is a 2-approximation algorithm for independent set if A can always find an independent set of at least half the optimal size.
    - Most of our focus is finding good relative approximation algorithms; unless explicitly stated otherwise, when I talk about approximation algorithms, I mean relative approximation algorithms

- Example: Multiprocessor scheduling to minimize makespan
  - Input
    - Set of n jobs with processing times x_n
    - Number of machines m
  - Task
    - Schedule the jobs on the m machines with the goal of minimizing the makespan (maximum completion time of any job) of the schedule.
  - Initial thoughts about this problem
    - Suppose $m = 1$: is this problem hard?
    - We know this problem is hard for $m \geq 2$ because of a reduction from the partition problem.

- Graham's list scheduling algorithm (Greedy)
  - Take the items in an arbitrary order
  - Place each item on the currently least loaded machine
  - This is a greedy algorithm

- Proof of $(2 - 1/m)$-approximation factor
  - Greedy's cost
    - Let job j be the last job to complete by Greedy
    - Let h be the load of the machine j is placed onto before job j is placed
    - Greedy's cost is $h + x_j$
  - Bounds on $OPT(I)$
    - $OPT(I) \geq \max_j x_j$
    - $OPT(I) \geq \frac{1}{m} \sum_{i=1}^{n} x_i$
  - Relating the two costs
    - $h \leq \frac{1}{m}((\sum_{i=1}^{n} x_i) - x_j)$
        - This follows because greedy places j onto the least loaded machine available

    - In the worst case, the least loaded machine has height the average cost of the remaining $n-1$ jobs (if job j is the last job and the other jobs can be evenly partitioned)
  - This gives us $GREEDY(I) = h + x_j$
  $\leq \frac{1}{m}((\sum_{i=1}^{n} x_i) - x_j) + x_j$
  $= \frac{1}{m}(\sum_{i=1}^{n} x_i) + \frac{m-1}{m} x_j$
  $\leq OPT(I) + \frac{m-1}{m} OPT(I)$
  $= (2 - \frac{1}{m}) OPT(I)$

- Longest job first
  - Same as Graham's list scheduling but first sort the jobs into non-decreasing order by size
- Proof of $(4/3 - \frac{1}{3m})$-approximation ratio
  - Let job j be the last job to complete by LJF
  - Without loss of generality, remove all jobs smaller than job j
    - LJF's cost cannot decrease because all jobs removed are scheduled after job j and thus do not affect the completion time of job j by LJF.
    - OPT's cost cannot increase by removing jobs
  - LJF's cost
    - Let h be the load of the machine j is placed onto before job j is placed
    - LJF's cost is $h + x_j$
  - Bound on $OPT(I)$
    - $OPT(I) \geq \frac{1}{m} \sum_{i=1}^{n} x_i$
  - 2 cases based on size of x_j
  - Case 1: $x_j \leq OPT(I)/3$
    - $h \leq \frac{1}{m}((\sum_{i=1}^{n} x_i) - x_j)$
      - This follows because LJF places j onto the least loaded machine available
      - In the worst case, the least loaded machine has height the average cost of the remaining $n-1$ jobs (if job j is the last job and the other jobs can be evenly partitioned)
    - This gives us $LJF(I) = h + x_j$
    $\leq \frac{1}{m}((\sum_{i=1}^{n} x_i) - x_j) + x_j$
    $= \frac{1}{m}(\sum_{i=1}^{n} x_i) + \frac{m-1}{m} x_j$
    $\leq OPT(I) + \frac{m-1}{m} \frac{OPT(I)}{3}$
    $= (4/3 - \frac{1}{3m}) OPT(I)$
  - Case 2: $x_j > OPT(I)/3$
    - Since the smallest job in the instance has size strictly larger than $1/3 OPT(I)$, in the optimal schedule, no machine has more than 2 jobs scheduled on it.
    - In this case, the optimal way to schedule jobs is to do exactly what LJF will do: pair up job $m-i$ with job $m+1+i$

    * Thus, in this case $LJF(I) = OPT(I)$
  - Since in both cases we have shown that $LJF(I) \leq (4/3 - \frac{1}{3m})OPT(I)$, the result follows.

- Example: Vertex Cover
  - Input
    * Graph $G = (V, E)$
  - Task
    * Find a vertex cover of minimum possible size
  - **NP**-hardness result
    * We know this problem is **NP**-hard because of a reduction from the independent set problem.

- Greedy is NOT good for vertex cover
  - Choose a node with highest updated degree breaking ties arbitrarily
  - Update degrees of remaining nodes
  - Can produce solution $\log n$ times as large as optimal

- Maximal matching algorithm
  - Find a maximal matching in the graph
    * A matching is a set of edges such that no two edges in the matching share a node
    * A maximal matching is a matching that cannot be increased by the addition of an edge without violating the condition of no two edges sharing a node
      · A maximum matching is a largest possible maximal matching.
      · Maximum matchings can be found in polynomial time, but the algorithm is fairly sophisticated
    * Maximal matchings can be found efficiently by processing the edges one at a time retaining an edge if and only if it does not share a node with any of the already chosen edges
  - Choose both nodes from all edges in the matching to be in the vertex cover
    * This set of nodes must be a vertex cover
    * Suppose it is not.
    * Then there is some edge (u, v) that does not include a node in the candidate vertex cover
    * In this case, edge (u, v) can be added to the maximal matching contradicting the claim that we had a maximal matching.

- Proof of 2-approximation ratio
  - Let the matching size be M
  - Our algorithm's cost is $2M$

- $OPT(I) \geq M$
  - This follows because each edge in the matching must be covered by at least one node
  - Since no two edges in the matching share a node, at least one of the two nodes from each edge must be chosen.

- Example: Bin Packing
  - Input
    - Set of n objects of sizes s_i
    - Infinite number of bins of size B
  - Task
    - Find a packing of the n objects into the minimum possible number of bins
  - Bin packing is **NP**-complete
    - Can reuse the reduction from Partition to Makespan Scheduling

- Algorithms
  - First Fit Algorithm
    - Arbitrarily order the items
    - Number the bins by the order they are opened (initially no bins are open)
    - When working with item i, place it into the first open bin it fits into. If it does not fit into any open bin, open a new bin and place it into that bin.
  - Best Fit Algorithm
    - Arbitrarily order the items
    - When working with item i, place it into the open bin it fits most tightly into. If it does not fit into any open bin, open a new bin and place it into that bin.
  - First Fit Decreasing Algorithm
    - Sort the items into non-decreasing order by size
    - Number the bins by the order they are opened (initially no bins are open)
    - When working with item i, place it into the first open bin it fits into. If it does not fit into any open bin, open a new bin and place it into that bin.
  - Best Fit Decreasing Algorithm
    - Sort the items into non-decreasing order by size
    - When working with item i, place it into the open bin it fits most tightly into. If it does not fit into any open bin, open a new bin and place it into that bin.
  - Proof that all algorithms are at least 2-approximations
    - There is at most 1 open bin that is more than half empty
      - If not, any of the algorithms would have combined the items in the two at least half-empty bins into 1 bin
    - Thus, an upper bound on the number of bins (ignoring the one possibly half-empty bin) used by any of these algorithms is $2\frac{1}{B}\sum_{i=1}^{n} s_i$

    - Clearly, $OPT(I) \geq \frac{1}{B} \sum_{i=1}^{n} s_i$
    - The result of 2 follows.
  - FF and BF have approximation ratios of 17/10 ignoring a constant additive factor
  - FFD and BFD have approxmation ratios of 11/9 ignoring a constant additive factor
  - The proofs of these results are very long and involved.

- Example: Traveling Salesperson
  - Input
    - List of n cities
    - Distances $d(i, j)$ between each pair of cities
  - Task
    - Find a tour of the cities of minimum possible total length
  - No polynomial-time c-approximation algorithm exists for this problem unless $\mathbf{P} = \mathbf{NP}$
    - If there is a polynomial-time c-approximation algorithm for this problem, then Hamiltonian Cycle can be solved in polynomial time.
    - Take an arbitrary input instance (graph $G = (V, E)$) of Hamiltonian Cycle and turn it into a TSP problem as follows:
      - For each node in V, create a city
      - For each edge $(i, j) \in E$, set $d(i, j) = 1$
      - For each pair of vertices $(i, j) \notin E$, set $d(i, j) = nc$
      - If G has a Hamiltonian cycle, then the optimal tour in the TSP instance is n
      - If G does not have a Hamiltonian cycle, then the optimal tour in the TSP instance has length at least $n - 1 + nc$
    - Apply our c-approximation algorithm to the TSP instance
      - If our approximation algorithm for TSP returns an answer of at most nc, then we know our original graph had a Hamiltonian cycle.
      - If our approximation algorithm for TSP returns an answer greater than nc, then we know our original graph did not have a Hamiltonian cycle.
      - Thus, we can solve the Hamiltonian cycle problem in polynomial time using our c-approximation algorithm for TSP.

- Example: Metric Traveling Salesperson
  - Input
    - List of n cities
    - Distances $d(i, j)$ between each pair of cities
      - Distances satisfy triangle inequality: $d(i, j) + d(j, k) \leq d(i, k)$
      - Note that the distances in our proof above that unrestricted TSP is hard to approximate do not satisfy this constraint

  - Task
    - Find a tour of the cities of minimum possible total length

- Greedy algorithm
  - Best guarantee is $O(\log n) OPT(I)$

- MST algorithm
  - Find a minimum spanning tree T.
  - Double all the edges in T to create an Eulerian graph.
  - Take an Euler tour of the graph using shortcuts to avoid visiting cities more than once.
    - Shortcuts cannot increase cost because of triangle inequality

- Proof of 2-approximation ratio
  - Let $C(T)$ be the total weight of the minimum spanning tree T
  - $OPT(I) \geq C(T)$
    - Remove an edge from the optimal tour.
    - We now have a path which is one possible spanning tree.
    - The minimum spanning tree must have cost no more than this path.
  - $MST(I) \leq 2C(T)$
    - The Eulerian tour has cost exactly $2C(T)$
    - When taking shortcuts, this cost may decrease but cannot increase because of triangle inequality

- Christofides' matching improvement
  - Find a minimum spanning tree T.
  - Find all the nodes in T with odd degree
  - Find a minimum weight matching M of these nodes
  - Create an Eulerian graph by adding the edges in M to those in T
  - Take an Euler tour of the graph using shortcuts to avoid visiting cities more than once.

- Proof of 3/2-approximation ratio
  - Let $C(T)$ be the total weight of the minimum spanning tree T and let $C(M)$ be the total weight of M
  - $OPT(I) \geq C(T)$
    - Remove an edge from the optimal tour.
    - We now have a path which is one possible spanning tree.
    - The minimum spanning tree must have cost no more than this path.
  - $1/2 OPT(I) \geq C(M)$

    * Consider the optimal tour
    * Take shortcuts to produce a tour that only includes the nodes with odd degree in T
    * This reduced tour has total weight at most $OPT(I)$ because shortcuts cannot increase the weight by the triangle inequality
    * This reduced tour an be used to create two possible matchings of these nodes by selecting every other edge
    * The minimum weight matching M must choose a matching that is less than the minimum of these two matchings, and the result follows.
- $Chr(I) \leq 3/2C(T)$
    * The Eulerian tour has cost exactly $C(T) + C(M) \leq 3/2OPT(I)$
    * When taking shortcuts, this cost may decrease but cannot increase because of triangle inequality