EECS 376 Discussion 10

Sec 27: Th 5:30-6:30 DOW 1017

IA: Eric Khiu

Slide deck available at course drive/Discussion/Slides/Eric Khiu

Midterm Student Feedback (MSF)

- We have Hafiz from engineering teaching consultant joining us today!
- He will collect feedback during the last 10 minutes of the class
- Thank you in advance for your feedback!

Agenda

- Search to Decision
- Approximation Algorithms

Search to Decision

Course notes

Starter: Decision vs Search

- Consider the following language:

$$L = \{A \text{ is an array of } n \text{ integers that contains } m\}$$

where m is a magic integer

- Suppose I have a decider D that decides L, what does the output of $D(A[1, \ldots, n])$ tells me? (m is hard-coded in D)
- What about $D(A[1, \ldots, n-1])$?

Discuss: Suppose I know that m is in A (but I still don't know what m is), how can I use D to determine the *index* of m?

```
findIndex(A):
    for idx = 1, …, n do
        if D(A[i]) accepts then return idx
```

Search to Decision

- **Informal proposition:** A search version of any NP-complete problem has an efficient algorithm iff the decision version does

- **Corollary:** If we have access to an efficient decider for an NP-complete language, we can construct an efficient algorithm to solve corresponding search version of the language

- This efficient algorithm is known as a search to decision reduction

Search + Optimization

- Sometimes, on top of searching for *a* solution, we are interested in the *best* solution (optimization problem) from a set of possible solutions
- **Best solution:** The one that has the highest/ lowest *value*
- **Maximization:** 0-1 Knapsack
  - Solution space: set of subsets of items whose weight does not exceed the capacity
  - Value: Total value of the subset
  - Goal: Find the subset with the highest value
- **Minimization:** Minimum spanning trees
  - Solution space: set of spanning trees
  - Value: Tree weight
  - Goal: Find the spanning tree with the lowest weight

But wait a minute…

- What if I don’t know the optimal value?
  - Still search to decision!
  - Use the same decider for decision problem to find it!

Example: Knapsack Max Value

- Recall the knapsack language (decision problem)

$$\text{KNAPSACK} := \left\{ (W[1\ldots n], V[1\ldots n], C, K) : \exists S \subseteq \{1,\ldots,n\} \text{ s.t. } \sum_{i\in S} W[i] \le C \text{ and } \sum_{i \in S} V[i] \ge K \right\},$$

  Note: Assume all number $W[i], V[i], C$, and K are **non-negative integers** for simplicity.

- Suppose there exists an efficient algorithm D that decides $KNAPSACK$
- Given a knapsack instance (W, V, C), describe an efficient algorithm that uses D to determine the maximum value K^* of a set of items whose total capacity is at most C.
  - Hint: What is the upper bound for K^*?
  - Sum of values of all items! $K^* \le \sum_{i=1}^{n} V[i]$

Example: Knapsack Max Value

- Given a knapsack instance (W, V, C), describe an efficient algorithm that uses D to determine the maximum value K^* of a set of items whose total capacity is at most C. Know: $K^* \leq \sum_{i=1}^{n} V[i]$

```
findMaxVal(W, V, C):
    K* ← −∞
    T ← Σ_{i=1}^{n} V[i]
    for k = 0, …, T do
        if D(W, V, C, k) accepts then K* ← k
        else break
    return K*
```

Discuss: What is wrong with this?

- **Correctness analysis:** The optimal K^* must be in the range of 0 to $\sum_{i=1}^{n} V[i]$, and the algorithm will find the largest value in the range for which D accepts

It is not efficient!

- Recall that the input size of an integer is the number of bits used to represent it
- If we have an array of size n, we often say the input size is $O(n)$
- In fact, if b_{max} is the max number of bits used to represent the element with largest value in A, then the input size is $O(b_{max} \cdot n)$- but we often take b_{max} as a constant
- But it matters here!
  - Let b_w and b_v be the max number of bits of the elements with largest value in W and V
  - Input size of (W, V, C) = $O(nb_w) + O(nb_v) + O(\log C)$
  - Computing $T = \sum_{i=1}^{n} V[i]$ takes $O(n)$
  - Upper bound of $V[i]$: $2^{b_v} - 1$ $\Rightarrow$ Value of T: $O\left(n \cdot (2^{b_v} - 1)\right) = O(n \cdot 2^{b_v})$
  - Linear search over $0, \dots, T$: $O(n \cdot 2^{b_v})$ $\Rightarrow$ Total runtime = $O(n) + O(n \cdot 2^{b_v})$ $\Rightarrow$ Not efficient!

Is there a search that runs in $O(\log(\cdot))$?

- Binary search!
- **Attempt 2:** Perform a binary search over $k = 0, \ldots, T$, calling D with different values of k until we find the highest for which D accepts
- **Take home exercise:** Try to write the algorithm

- **Correctness analysis:** Same as before
- **Runtime analysis:**
  - Input size of (W, V, C) = $O(nb_w) + O(nb_v) + O(\log C)$
  - Value of T: $O\left(n \cdot (2^{b_v} - 1)\right) = O(n \cdot 2^{b_v})$
  - Total runtime = $O(n) + O(\log_2 T) = O(n) + O\left(\log_2 (n \cdot 2^{b_v})\right)$

$$= O(n) + O(\log n) + O(b_v) \Rightarrow \text{Efficient!}$$

Practice: Knapsack Best Subset

- Recall the knapsack language (decision problem)

$$\textsc{Knapsack} := \left\{ (W[1\ldots n], V[1\ldots n], C, K) : \exists S \subseteq \{1,\ldots,n\} \text{ s.t. } \sum_{i\in S} W[i] \le C \text{ and } \sum_{i \in S} V[i] \ge K \right\},$$

  Note: Assume all number $W[i], V[i], C,$ and K are **non-negative integers** for simplicity.

- Suppose there exists an efficient algorithm D that decides $KNAPSACK$
- Suppose K^* is the maximum value obtainable with capacity C
- Given a knapsack instance (W, V, C, K^*), describe an efficient algorithm that uses D to determine the set of items whose total weight is at most C, and whose total value is K^*

  Hint: Recall the intuition from DP: To take or not to take?

Practice: Knapsack Best Subset

- Given a knapsack instance (W, V, C, K^*), describe an efficient algorithm that uses D to determine the set of items whose total weight is at most C, and whose total value is K^*

```
KnapSearch((W, V, C, K), K*):
```

$S \leftarrow \emptyset$

for $i \in \{1, \dots, n\}$ **do**

if $D(W[(i+1), \dots, n], V[(i+1), \dots, n], C - W[i], K^* - V[i])$ **accepts then:**

$S \leftarrow S \cup \{i\}$ // Add an item iff it is possible to obtain K^* with the remaining items (D accepts)

$C \leftarrow C - W[i]$ // Update capacity available for the remaining items

$K \leftarrow K^* - V[i]$ // Update values needed from the remaining items

return S

- **Runtime analysis:** $O(n)$

Practice: Knapsack Best Subset

- **Correctness Analysis:** Consider an optimal knapsack S^* with optimal value K^*,
  - Suppose S^* has the same decision (take/ discard) as the first i items as S
  - Assume S^* has the different decision as S on the $(i+1)^{th}$ item (the other case is trivial)

Item	1	2	…	i	i+1	…
S*	Take	Discard	…	Take	Take	…
S	Take	Discard	…	Take	Discard	…
S'	Take	Discard	…	Take	Discard	…

  - Consider a knapsack S' that follows the first $i+1$ decisions as S
  - By construction of S, it must be that $D(W[(i+1), \dots, n], V[(i+1), \dots, n], C - W[i], K^* - V[i])$ accepts, which means we can still obtain K^* with S' if it follows the first $i+1$ decisions as S
  - Hence, S' is also an optimal solution $\Rightarrow$ first $i+1$ decisions of S is part of an optimal solution

Practice: Knapsack Best Subset

- **Take home exercise:** Why wouldn't this work?

```
KnapSearch((W, V, C, K), K*):
```

$S \leftarrow \emptyset$

for $i \in \{1, \ldots, n\}$ **do**

if $D(W \setminus W[i], V \setminus V[i], C - W[i], K^* - V[i])$ accepts **then:**

$S \leftarrow S \cup \{i\}$

return S

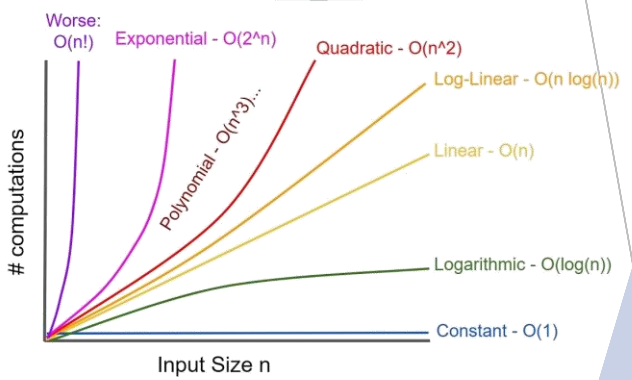
- Note: Here $W \setminus W[i]$ means removing $W[i]$ from W
- Hint: Consider $W = [1,1,1]$, $V = [1,1,1]$, $C = 2$, $K = 2$

Approximation Algorithms

Course notes

Speed vs Accuracy

- Suppose you want to solve a *really hard* classification problem and there are four algorithms available to you:
  A. Runs in $O(n!)$, but the accuracy is guaranteed to be 100%
  B. Runs in $O(2^n)$, but the accuracy is *at least* 90%
  C. Runs in $O(n)$, but the accuracy is *at least* 60%
  D. Runs in $O(1)$, but there is no guarantee on the accuracy



Poll: Which one would you choose if
- this is a real-time spam message detector?
- this is an AI for identifying foes in military applications?

Approximation Algorithms

- **Motivation:** some search problems are very important (TSP, job scheduling, etc.), but if they are NP-hard, then we currently can't solve them efficiently
  - Approximation algorithms get a *close* answer, sacrificing correctness for speed
- We can define how *good* an approximation is in terms of an approximation ratio α
  - Let $val(y)$ be a function that maps the output of a function to some value
  - Let OPT be the value of an optimal solution for some search problem
- An approximate solution y is said to be an α-approximation if

$$\alpha \cdot OPT \leq val(y) \quad \text{for maximization problem}$$

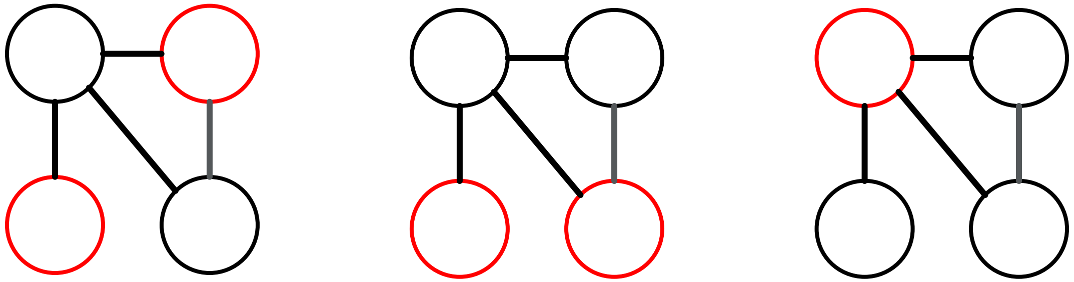
$$val(y) \leq \alpha \cdot OPT \quad \text{for minimization problem}$$

Concept Check

- Suppose algorithm $\mathcal{A}$ is a 2-approximation for a minimization problem. Then, for (all/ some/ no) inputs x we have $val\big(\mathcal{A}(x)\big) = 2 \cdot OPT$.
  - Note: You can assume that 2 is the **tightest value** of α
- Answer: some
  - $val(\mathcal{A}(x)) \leq 2 \cdot OPT$ for all x
  - $\mathcal{A}$ will output a solution at most $2 \cdot OPT$

Example: Independent Set

- An *independent set* of an undirected graph $G = (V, E)$ is a subset $S \subseteq V$ of vertices for which there is no edge between any pair of vertices in S.



- The *maximum independent set* (MIS) problem is: given a graph, find an independent set of maximum size.

Example: Independent Set

- Consider the following algorithm:

> 1. Let $S = \emptyset$ and let $G' = G$.
> 2. While G' still has at least one vertex:
>   a) Choose an arbitrary vertex v of G'.
>   b) Let $S = S \cup \{v\}$.
>   c) Remove v and all its neighbors (including all their incident edges) from G'. (A neighbor of v is any vertex that is connected to v by an edge.)
> 3. Output S.

- Let $U = V \setminus S$ denote the set of all vertices removed in step 2c, **not including** the vertices selected for S, and let Δ be the maximum degree of *all* vertices in G. Prove that $|U| \leq |S| \cdot \Delta$.

Hint: If Δ is the max degree of all vertices, what can you say about the number of vertices added to U for each vertex added to S?

Example: Independent Set

- Consider the following algorithm:

> 1. Let $S = \emptyset$ and let $G' = G$.
> 2. While G' still has at least one vertex:
>   a) Choose an arbitrary vertex v of G'.
>   b) Let $S = S \cup \{v\}$.
>   c) Remove v and all its neighbors (including all their incident edges) from G'. (A neighbor of v is any vertex that is connected to v by an edge.)
> 3. Output S.

- Let $U = V \setminus S$ denote the set of all vertices removed in step 2c, **not including** the vertices selected for S, and let Δ be the maximum degree of *all* vertices in G. Prove that $|U| \leq |S| \cdot \Delta$.
  - If Δ is the max degree of all vertices, then at most Δ vertices are added to U for each vertex added to S
  - Since the algorithms adds $|S|$ vertices to S, we have $|U| \leq |S| \cdot \Delta$

Example: Independent Set

- Consider the following algorithm:

> 1. Let $S = \emptyset$ and let $G' = G$.
> 2. While G' still has at least one vertex:
>   a) Choose an arbitrary vertex v of G'.
>   b) Let $S = S \cup \{v\}$.
>   c) Remove v and all its neighbors (including all their incident edges) from G'. (A neighbor of v is any vertex that is connected to v by an edge.)
> 3. Output S.

- Using the fact that $|U| \leq |S| \cdot \Delta$, prove that the algorithm is a $1/(\Delta + 1)$ approximation for MIS. (WTS: $\alpha \cdot OPT \leq val(y)$)

  Hint: $V = U \cup S$ and $U \cap S = \emptyset$

Example: Independent Set

- Consider the following algorithm:

> 1. Let $S = \emptyset$ and let $G' = G$.
> 2. While G' still has at least one vertex:
>   a) Choose an arbitrary vertex v of G'.
>   b) Let $S = S \cup \{v\}$.
>   c) Remove v and all its neighbors (including all their incident edges) from G'. (A neighbor of v is any vertex that is connected to v by an edge.)
> 3. Output S.

- Using the fact that $|U| \leq |S| \cdot \Delta$, prove that the algorithm is a $1/(\Delta + 1)$ approximation for MIS. (WTS: $\alpha \cdot OPT \leq val(y)$)
  - Let T^* be a maximum independent set. Since $T^* \subseteq V$,

$$|T^*| \leq |V| = |U| + |S| \leq |S| \cdot \Delta + |S|$$

$$\frac{1}{\Delta + 1}|T^*| \leq |S|$$

Extra Slides

SAT Search to Decision Reduction

$\text{SAT} = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$

- Assume that SAT ∈ P, then it has an efficient decider D.
- Search objective: find an assignment for each variable $x_1, \ldots, x_n$ in ϕ

1. If $D(\phi)$ returns false, output $\perp$ (the formula is unsatisfiable).
2. For each variable x_i $(1 \leq i \leq n)$ in ϕ, do the following:
   (a) Set x_i to false ($x_i = F$). Let us denote the resulting formula (with x_i set to false) as $\phi_{x_i=F}$. Run $D(\phi_{x_i=F})$.
      i. If $D(\phi_{x_i=F})$ accepts, continue to the next iteration of the algorithm (for x_{i+1}).
      ii. If $D(\phi_{x_i=F})$ rejects, set x_i to true and continue to the next iteration of the algorithm for x_{i+1}.

SAT Search to Decision Reduction

1. If $D(\phi)$ returns false, output $\perp$ (the formula is unsatisfiable).
2. For each variable x_i $(1 \le i \le n)$ in ϕ, do the following:
   (a) Set x_i to false $(x_i = F)$. Let us denote the resulting formula (with x_i set to false) as $\phi_{x_i=F}$. Run $D(\phi_{x_i=F})$.
      i. If $D(\phi_{x_i=F})$ accepts, continue to the next iteration of the algorithm (for x_{i+1}).
      ii. If $D(\phi_{x_i=F})$ rejects, set x_i to true and continue to the next iteration of the algorithm for x_{i+1}.

- Runtime Analysis:
  - D runs in $O(|\phi|^k)$ for some constant k, so step 1 is efficient
  - Step 2 loops n times, which is $\le |\phi|$, within each iteration we assign truth assignments to one variable which is linear worst case, then run D.
    $O(n \cdot (|\phi| + |\phi|^k)) = O(|\phi|^2 + |\phi|^{k+1}) = O(|\phi|^{k+1})$

Metric TSP Approx (Lecture Review)

$$\text{TSP} = \{\langle G, k\rangle \mid G \text{ is an undirected, weighted, complete graph with a tour of weight} \leq k\}$$

- Traveling Salesperson Problem
  - Input is a complete, weighted, undirected graph G
  - The weight of a subgraph is the sum of its edge weights
  - Goal is to find an *optimal tour*, or a Hamiltonian cycle with minimum weight
- This is very difficult to solve, so we impose the triangle inequality constraint:
  - Any three vertices in V satisfy the triangle inequality

$$w((v_1, v_2)) \leq w((v_1, v_3)) + w((v_3, v_2)).$$

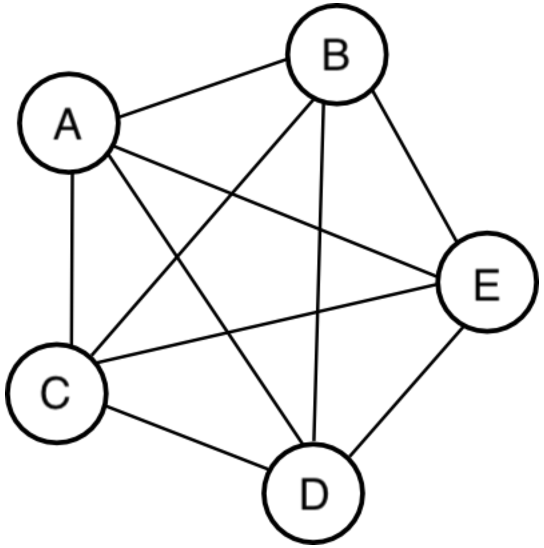
  - This version of TSP is known as *Metric TSP*
  - Even Metric TSP is NP-Complete! So we present a 2-approximation

Metric TSP Approx (Lecture Review)

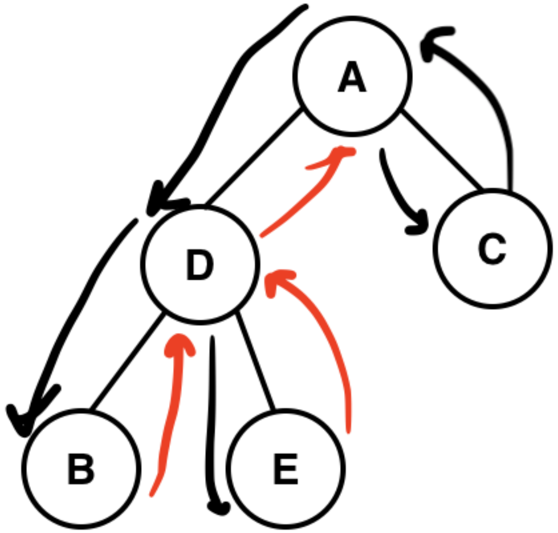
- Recall a minimum spanning tree is an undirected, connected, acyclic graph that contains all vertices in G with as little weight as possible
- The weight of the MST T is $\leq$ the weight of the optimal tour H
  - Proof: assume we have a graph where the weight of the MST is greater than the weight of the optimal tour. Removing an edge in the tour would result in a spanning tree of weight less than the MST, which is a contradiction
- Algorithm
  - Use Kruskal's algorithm to get T, an MST of G
  - Perform a depth-first search on the MST, but skip vertices we've already visited
    - Triangle inequality guarantees that this is better than visiting every edge twice

Example Run of TSP Approximation

- Start with a complete, undirected graph



- Find the MST and do a DFS, skipping repeated edges



Original DFS: A → D → B → D → E → D → A → C → A

Modified: A → D → B → E → C → A

Metric TSP Approx (Lecture Review)

- The weight of the MST T is $\leq$ the weight of the optimal tour H
  - Proof: assume we have a graph where the weight of the MST is greater than the weight of the optimal tour. Removing an edge in the tour would result in a spanning tree of weight less than the MST, which is a contradiction

- Algorithm
  - Use Kruskal's algorithm to get T, an MST of G
  - Perform a depth-first search on the MST, but skip vertices we've already visited
    - Triangle inequality guarantees that this is better than visiting every edge twice

- This gives us a Hamiltonian cycle with weight c
- $c \leq 2w(T)$ because we traverse each edge in T at most twice
- $c \leq 2w(T) \leq 2w(H)$ because $w(T) \leq w(H)$ (proved above)
- This is a 2-approximation of constrained TSP

- Worksheet Problem 8 result: Even *approximating* general TSP with a fixed α bound is NP-complete!