Lecture 24: Review Session & AMA

Rafael Oliveira

University of Waterloo
Cheriton School of Computer Science

rafael.oliveira.teaching@gmail.com

December 5, 2023

Overview

- Review Session
  - Divide-and-Conquer
  - Greedy
  - Dynamic Programming
  - Graph Algorithms
  - Max-Flow Min-Cut
  - Reductions
  - Intractability
- Ask me Anything
- Acknowledgements

Divide-and-Conquer

- Structure of divide-and-conquer:
  1. **Divide:** given instance I, construct smaller instances $I_1, \ldots, I_a$ (*subproblems*)

     Ideally want $|I_j|$ small compared to $|I|$ (say constant fraction)

Divide-and-Conquer

- Structure of divide-and-conquer:
  1. **Divide:** given instance I, construct smaller instances $I_1, \ldots, I_a$ (*subprojects*)

     Ideally want $|I_j|$ small compared to $|I|$ (say constant fraction)
  2. **Conquer:** recursively solve instances $I_1, \ldots, I_a$, obtaining solutions $S_1, \ldots, S_a$

Divide-and-Conquer

- Structure of divide-and-conquer:
  1. **Divide:** given instance I, construct smaller instances $I_1, \ldots, I_a$ (*subprojects*)

     Ideally want $|I_j|$ small compared to $|I|$ (say constant fraction)
  2. **Conquer:** recursively solve instances $I_1, \ldots, I_a$, obtaining solutions $S_1, \ldots, S_a$
  3. **Combine:** solutions $S_1, \ldots, S_a \mapsto$ solution S to instance I

Divide-and-Conquer

- Structure of divide-and-conquer:
  1. **Divide:** given instance I, construct smaller instances $I_1, \ldots, I_a$ (*subproblems*)

     Ideally want $|I_j|$ small compared to $|I|$ (say constant fraction)
  2. **Conquer:** recursively solve instances $I_1, \ldots, I_a$, obtaining solutions $S_1, \ldots, S_a$
  3. **Combine:** solutions $S_1, \ldots, S_a \mapsto$ solution S to instance I
- "Recursion for running time:"

$$T(I) = T(I_1) + \cdots + T(I_a) + \text{ time to combine}$$

- Review Session
  - Divide-and-Conquer
  - Greedy
  - Dynamic Programming
  - Graph Algorithms
  - Max-Flow Min-Cut
  - Reductions
  - Intractability
- Ask me Anything
- Acknowledgements

Greedy Algorithms

- Greedy strategy based on following principles:
  1. choose a "progress measure"
  2. preprocess input accordingly
  3. make next decision based on what is *best* given *current* partial solution
  4. **Main idea:** must show that the greedy solution is always *no worse* than any other optimal solution!

     Usually can prove this by begin able to "transform" any optimal solution into the greedy one without losing anything.

     Exchange Argument

  5. *Optimal Substructure*: a problem has optimal substructure if any optimal solution contains optimal solutions to subproblems.

Dynamic Programming

- Sometimes, when trying a divide and conquer approach, we are only able to divide in a way which makes us perform "exhaustive search"

Looks like it is going to be a bad divide and conquer

Dynamic Programming

- Sometimes, when trying a divide and conquer approach, we are only able to divide in a way which makes us perform "exhaustive search"

  Looks like it is going to be a bad divide and conquer

- However, in several situations, it turns out that a *small set* of particular subproblems appear *several times* in our recurrence

Dynamic Programming

- Sometimes, when trying a divide and conquer approach, we are only able to divide in a way which makes us perform "exhaustive search"

  Looks like it is going to be a bad divide and conquer

- However, in several situations, it turns out that a *small set* of particular subproblems appear *several times* in our recurrence
- Instead of recomputing the subproblems, we can:
  1. solve them once
  2. save them to memory
  3. and if we need them again, we already precomputed them! (savings)

Dynamic Programming

- Sometimes, when trying a divide and conquer approach, we are only able to divide in a way which makes us perform "exhaustive search"

  Looks like it is going to be a bad divide and conquer

- However, in several situations, it turns out that a *small set* of particular subproblems appear *several times* in our recurrence
- Instead of recomputing the subproblems, we can:
  1. solve them once
  2. save them to memory
  3. and if we need them again, we already precomputed them! (savings)
- **DP template**
  1. identify small set of subproblems
  2. devise proper recursion
  3. show how bottom-up approach correctly compute the subproblems

Graph Search & Connectivity

Undirected graphs:

- **BFS**
  1. Finds shortest paths
  2. Can be used to detect graph is bipartite
  3. Shortest paths encoded in the BFS tree
  4. Non-tree edges in adjacent layers

Graph Search & Connectivity

Undirected graphs:

- **BFS**
  1. Finds shortest paths
  2. Can be used to detect graph is bipartite
  3. Shortest paths encoded in the BFS tree
  4. Non-tree edges in adjacent layers
- **DFS**
  1. Parenthesis lemma: start and finish time intervals are either disjoint, or one contains the other
  2. non-tree edges (in DFS tree) must be back edges
  3. checks for cut vertices or cut edges

Graph Search & Connectivity

Directed graphs:

- BFS still gives you shortest paths from source

Graph Search & Connectivity

Directed graphs:

- BFS still gives you shortest paths from source
- BFS and DFS trees have less structure, but parenthesis lemma still holds for DFS tree

Graph Search & Connectivity

Directed graphs:

- BFS still gives you shortest paths from source
- BFS and DFS trees have less structure, but parenthesis lemma still holds for DFS tree
- DAGs (directed acyclic graphs) (topological sort)

Graph Search & Connectivity

Directed graphs:

- BFS still gives you shortest paths from source
- BFS and DFS trees have less structure, but parenthesis lemma still holds for DFS tree
- DAGs (directed acyclic graphs) (topological sort)
- Any directed graph is a DAG of SCCs (strongly connected components)

Graph Search & Connectivity

Directed graphs:

- BFS still gives you shortest paths from source
- BFS and DFS trees have less structure, but parenthesis lemma still holds for DFS tree
- DAGs (directed acyclic graphs) (topological sort)
- Any directed graph is a DAG of SCCs (strongly connected components)
- Linear time algorithm to find all SCCs!

Minimum Spanning Trees (MSTs)

- Greedy for the rescue!

Minimum Spanning Trees (MSTs)

- Greedy for the rescue!
- Boruvka's algorithm:
  1. pick cheapest edge from a vertex, and contract it
  2. recurse on contracted graph

Minimum Spanning Trees (MSTs)

- Greedy for the rescue!
- Boruvka's algorithm:
  1. pick cheapest edge from a vertex, and contract it
  2. recurse on contracted graph
- Cut property: given any cut $(S, \overline{S})$, there is an MST containing edge with smallest weight across cut.

Minimum Spanning Trees (MSTs)

- Greedy for the rescue!
- Boruvka's algorithm:
  1. pick cheapest edge from a vertex, and contract it
  2. recurse on contracted graph
- Cut property: given any cut $(S, \overline{S})$, there is an MST containing edge with smallest weight across cut.
- Prim's algorithm:
  1. start from arbitrary vertex and grow connected component one vertex at a time

Minimum Spanning Trees (MSTs)

- Greedy for the rescue!
- Boruvka's algorithm:
  1. pick cheapest edge from a vertex, and contract it
  2. recurse on contracted graph
- Cut property: given any cut $(S, \overline{S})$, there is an MST containing edge with smallest weight across cut.
- Prim's algorithm:
  1. start from arbitrary vertex and grow connected component one vertex at a time
- Kruskal's algorithm:
  1. consider edges from cheapest to most expensive, add edge to solution so long as it does not create a cycle
  2. needs UNION-FIND for that last step

Minimum Spanning Trees (MSTs)

- Greedy for the rescue!
- Boruvka's algorithm:
  1. pick cheapest edge from a vertex, and contract it
  2. recurse on contracted graph
- Cut property: given any cut $(S, \overline{S})$, there is an MST containing edge with smallest weight across cut.
- Prim's algorithm:
  1. start from arbitrary vertex and grow connected component one vertex at a time
- Kruskal's algorithm:
  1. consider edges from cheapest to most expensive, add edge to solution so long as it does not create a cycle
  2. needs UNION-FIND for that last step
- All of the above can be assumed to run in $O(m \log n)$ time.

Shortest Paths

- Single-source, all weights non-negative: *Dijkstra*
  - Similar to Prim's algorithm (greedy)
  - Start from source, and build shortest paths by adding one vertex at a time
  - efficiently simulates the "water down the pipes" idea
  - Runtime $O((m+n)\log n)$

Shortest Paths

- Single-source, all weights non-negative: *Dijkstra*
  - Similar to Prim's algorithm (greedy)
  - Start from source, and build shortest paths by adding one vertex at a time
  - efficiently simulates the "water down the pipes" idea
  - Runtime $O((m+n)\log n)$
- Single-source, arbitrary weights: *Bellman-Ford*
  - Now cannot do the greedy approach, because of negative weights
  - DP for the rescue!
  - Subproblems $D[v,i] :=$ captures shortest $s \to v$ distance using at most i edges
  - Runtime $O(mn)$

Shortest Paths

- Single-source, all weights non-negative: *Dijkstra*
  - Similar to Prim's algorithm (greedy)
  - Start from source, and build shortest paths by adding one vertex at a time
  - efficiently simulates the "water down the pipes" idea
  - Runtime $O((m+n)\log n)$
- Single-source, arbitrary weights: *Bellman-Ford*
  - Now cannot do the greedy approach, because of negative weights
  - DP for the rescue!
  - Subproblems $D[v,i] :=$ captures shortest $s \to v$ distance using at most i edges
  - Runtime $O(mn)$
- All-pairs shortest-paths (arbitrary weights, no negative cycles)

  Floyd-Warshall

  - Subproblems: $D[u,v,k] :=$ shortest $u \to v$ path using only $\{1,\ldots,k\}$ as intermediate vertices
  - Runtime $O(n^3)$

Max-Flow & Min-Cut

Let $G(V, E, c)$ be an undirected graph, with capacity (weight) function $C : E \to \mathbb{R}_{\geq 0}$, two special vertices $s, t \in V$

- *Flows*: $f : E \to \mathbb{R}_{\geq 0}$ satisfying:
  1. Capacity: $f(e) \leq c(e)$ for all $e \in E$
  2. Conservation: $f_{\mathrm{in}}(u) = f_{\mathrm{out}}(u)$ for all $u \in V \setminus \{s, t\}$
  3. Value: $f_{\mathrm{out}}(s) - f_{\mathrm{in}}(s)$

Max-Flow & Min-Cut

Let $G(V, E, c)$ be an undirected graph, with capacity (weight) function $C : E \to \mathbb{R}_{\geq 0}$, two special vertices $s, t \in V$

- *Flows*: $f : E \to \mathbb{R}_{\geq 0}$ satisfying:
  1. Capacity: $f(e) \leq c(e)$ for all $e \in E$
  2. Conservation: $f_{\mathrm{in}}(u) = f_{\mathrm{out}}(u)$ for all $u \in V \setminus \{s, t\}$
  3. Value: $f_{\mathrm{out}}(s) - f_{\mathrm{in}}(s)$
- **Flow decomposition theorem:** any integral flow $f : E \to \mathbb{N}$ of value r can be decomposed into paths $P_1, \ldots, P_r$ and cycles $C_1, \ldots, C_m$ such that each $e \in E$ appears in exactly $f(e)$ of the paths and cycles.

Max-Flow & Min-Cut

Let $G(V, E, c)$ be an undirected graph, with capacity (weight) function $C : E \to \mathbb{R}_{\geq 0}$, two special vertices $s, t \in V$

- *Flows*: $f : E \to \mathbb{R}_{\geq 0}$ satisfying:
  1. Capacity: $f(e) \leq c(e)$ for all $e \in E$
  2. Conservation: $f_{\text{in}}(u) = f_{\text{out}}(u)$ for all $u \in V \setminus \{s, t\}$
  3. Value: $f_{\text{out}}(s) - f_{\text{in}}(s)$
- **Flow decomposition theorem:** any integral flow $f : E \to \mathbb{N}$ of value r can be decomposed into paths $P_1, \ldots, P_r$ and cycles $C_1, \ldots, C_m$ such that each $e \in E$ appears in exactly $f(e)$ of the paths and cycles.
- *Cuts*: a cut is a partition of the vertices into two sets $(S, V \setminus S)$.
  - Capacity of a cut: $C_{\text{out}}(S)$ is the total capacity *coming out* of S

Max-Flow & Min-Cut

Let $G(V, E, c)$ be an undirected graph, with capacity (weight) function $C : E \to \mathbb{R}_{\geq 0}$, two special vertices $s, t \in V$

- *Flows*: $f : E \to \mathbb{R}_{\geq 0}$ satisfying:
  1. Capacity: $f(e) \leq c(e)$ for all $e \in E$
  2. Conservation: $f_{\text{in}}(u) = f_{\text{out}}(u)$ for all $u \in V \setminus \{s, t\}$
  3. Value: $f_{\text{out}}(s) - f_{\text{in}}(s)$
- **Flow decomposition theorem:** any integral flow $f : E \to \mathbb{N}$ of value r can be decomposed into paths $P_1, \ldots, P_r$ and cycles $C_1, \ldots, C_m$ such that each $e \in E$ appears in exactly $f(e)$ of the paths and cycles.
- *Cuts*: a cut is a partition of the vertices into two sets $(S, V \setminus S)$.
  - Capacity of a cut: $C_{\text{out}}(S)$ is the total capacity *coming out* of S
- **Max-Flow Min-Cut Theorem:** the value of the maximum flow equals the minimum capacity of a cut

Max-Flow & Min-Cut

Let $G(V, E, c)$ be an undirected graph, with capacity (weight) function $C : E \to \mathbb{R}_{\geq 0}$, two special vertices $s, t \in V$

- *Flows*: $f : E \to \mathbb{R}_{\geq 0}$ satisfying:
  1. Capacity: $f(e) \leq c(e)$ for all $e \in E$
  2. Conservation: $f_{\text{in}}(u) = f_{\text{out}}(u)$ for all $u \in V \setminus \{s, t\}$
  3. Value: $f_{\text{out}}(s) - f_{\text{in}}(s)$
- **Flow decomposition theorem:** any integral flow $f : E \to \mathbb{N}$ of value r can be decomposed into paths $P_1, \ldots, P_r$ and cycles $C_1, \ldots, C_m$ such that each $e \in E$ appears in exactly $f(e)$ of the paths and cycles.
- *Cuts*: a cut is a partition of the vertices into two sets $(S, V \setminus S)$.
  - Capacity of a cut: $C_{\text{out}}(S)$ is the total capacity *coming out* of S
- **Max-Flow Min-Cut Theorem:** the value of the maximum flow equals the minimum capacity of a cut
- **Ford-Fulkerson** algorithm: keep finding $s \to t$ paths in residual graph, when there is none, found a max-flow and a min-cut

Reductions

- How do we prove problem A is "easier than" another problem B?

Reductions

- How do we prove problem A is "easier than" another problem B?
- Intuitive notion is: if we can efficiently solve B, then we can also efficiently solve A

Reductions

- How do we prove problem A is "easier than" another problem B?
- Intuitive notion is: if we can efficiently solve B, then we can also efficiently solve A
- *Turing reductions* (a.k.a. Cook reductions)

  $A \leq_p^T B \Leftrightarrow$ there is a poly-time algorithm M^B with oracle access to B such that M^B solves A.

  - Oracle access: algorithm M^B can query the oracle on inputs to problem B, and each query is counted as 1 unit of time

Reductions

- How do we prove problem A is "easier than" another problem B?
- Intuitive notion is: if we can efficiently solve B, then we can also efficiently solve A
- *Turing reductions* (a.k.a. Cook reductions)

  $A \leq_p^T B \Leftrightarrow$ there is a poly-time algorithm M^B with oracle access to B such that M^B solves A.

  - Oracle access: algorithm M^B can query the oracle on inputs to problem B, and each query is counted as 1 unit of time
- *Karp reductions* (a.k.a. *polynomial transformations*)

  $A \leq_p B \Leftrightarrow$ there is a poly-time computable function $f : A \to B$ such that

  - x is a YES instance of $A \Leftrightarrow f(x)$ is YES instance of B

NP-completeness

- NP is a class of *decision problems*

NP-completeness

- NP is a class of *decision problems*
- To prove a problem B is NP-complete, need to prove
  1. $B \in \mathrm{NP}$
  2. there is an NP-complete problem A such that

$$A \leq_p B$$

Notation: $A \leq_p B$ is the same as I have used as $A \leq_m B$.
These are polynomial transformations (a.k.a. Karp reductions).

NP-completeness

- NP is a class of *decision problems*
- To prove a problem B is NP-complete, need to prove
  1. $B \in \mathrm{NP}$
  2. there is an NP-complete problem A such that

$$A \leq_p B$$

- Example:
  - $\textsc{HittingSet}$
    - **Input:** collection of sets $S_1, \ldots, S_m \subset [n]$, integer $k \in \mathbb{N}$
    - Is there a collection of k sets S_i which contain all elements of $[n]$?

NP-hardness

- A problem B is NP-hard if there is an NP-complete problem A such that
$$A \leq_p^T B$$
Notation: $A \leq_p^T B$ is the same as I have used as $A \leq_T B$. These are Cook reductions (which I have denoted as Turing reductions).

NP-hardness

- A problem B is NP-hard if there is an NP-complete problem A such that
$$A \leq_p^T B$$
- Note that the class of NP-hard problems can contain problems which are not decision problems

NP-hardness

- A problem B is NP-hard if there is an NP-complete problem A such that
$$A \leq_p^T B$$
- Note that the class of NP-hard problems can contain problems which are not decision problems
- Bonus: a bogus video on super mario bros and NP-hardness

  https://www.youtube.com/watch?v=HhGI-GqAK9c

  Disclaimer: if you are still confused about the complexity part, please do not watch it!

NP-hardness

- A problem B is NP-hard if there is an NP-complete problem A such that
$$A \leq_p^T B$$
- Note that the class of NP-hard problems can contain problems which are not decision problems
- Bonus: a bogus video on super mario bros and NP-hardness

  `https://www.youtube.com/watch?v=HhGI-GqAK9c`

  Disclaimer: if you are still confused about the complexity part, please do not watch it!
- A real proof can be found here

- Review Session
  - Divide-and-Conquer
  - Greedy
  - Dynamic Programming
  - Graph Algorithms
  - Max-Flow Min-Cut
  - Reductions
  - Intractability
- Ask me Anything
- Acknowledgements

AMA

Acknowledgement

Based on

- Entire course :)

References I

Cormen, Thomas and Leiserson, Charles and Rivest, Ronald and Stein, Clifford (2009)
Introduction to Algorithms, third edition.
MIT Press

Dasgupta, Sanjay and Papadimitriou, Christos and Vazirani, Umesh (2006)
Algorithms

Erickson, Jeff (2019)
Algorithms
https://jeffe.cs.illinois.edu/teaching/algorithms/

Kleinberg, Jon and Tardos, Eva (2006)
Algorithm Design.
Addison Wesley