Review session 2

Lecture 666

April 7, 2015

Dynamic Programming

1. Find a "smart" recursion for the problem in which the number of distinct subproblems is small; polynomial in the original problem size.
2. Eliminate recursion and find an iterative algorithm to compute the problems bottom up by storing the intermediate values in an appropriate data structure; need to find the right way or order the subproblem evaluation.
3. Estimate the number of subproblems, the time to evaluate each subproblem and the space needed to store the value. Evaluate the total running time.
4. Optimize the resulting algorithm further

Dynamic Programming

1. Find a "smart" recursion for the problem in which the number of distinct subproblems is small; polynomial in the original problem size.
2. Eliminate recursion and find an iterative algorithm to compute the problems bottom up by storing the intermediate values in an appropriate data structure; need to find the right way or order the subproblem evaluation.
3. Estimate the number of subproblems, the time to evaluate each subproblem and the space needed to store the value. Evaluate the total running time.
4. Optimize the resulting algorithm further

Dynamic Programming

1. Find a "smart" recursion for the problem in which the number of distinct subproblems is small; polynomial in the original problem size.
2. Eliminate recursion and find an iterative algorithm to compute the problems bottom up by storing the intermediate values in an appropriate data structure; need to find the right way or order the subproblem evaluation.
3. Estimate the number of subproblems, the time to evaluate each subproblem and the space needed to store the value. Evaluate the total running time.
4. Optimize the resulting algorithm further

Dynamic Programming

1. Find a "smart" recursion for the problem in which the number of distinct subproblems is small; polynomial in the original problem size.
2. Eliminate recursion and find an iterative algorithm to compute the problems bottom up by storing the intermediate values in an appropriate data structure; need to find the right way or order the subproblem evaluation.
3. Estimate the number of subproblems, the time to evaluate each subproblem and the space needed to store the value. Evaluate the total running time.
4. Optimize the resulting algorithm further

Dynamic Programming

1. Find a "smart" recursion for the problem in which the number of distinct subproblems is small; polynomial in the original problem size.
2. Eliminate recursion and find an iterative algorithm to compute the problems bottom up by storing the intermediate values in an appropriate data structure; need to find the right way or order the subproblem evaluation.
3. Estimate the number of subproblems, the time to evaluate each subproblem and the space needed to store the value. Evaluate the total running time.
4. Optimize the resulting algorithm further

Dynamic programming...

1. Longest increasing subsequence.
2. Computing the solution itself (not only its value).
3. Maximum Weight Independent Set in Trees.
4. Dynamic programs can be also solved as problems on DAGs.
5. Edit distance: $O(nm)$ [but linear space!].
6. Floyd-Warshall: $O(n^3)$.
7. Knapsack: $O(nW)$ (pseudo-polynomial).
8. TSP: $O(n^3 2^n)$ time and $O(n^2 2^n)$ space.

Dynamic programming...

1. Longest increasing subsequence.
2. Computing the solution itself (not only its value).
3. Maximum Weight Independent Set in Trees.
4. Dynamic programs can be also solved as problems on DAGs.
5. Edit distance: $O(nm)$ [but linear space!].
6. Floyd-Warshall: $O(n^3)$.
7. Knapsack: $O(nW)$ (pseudo-polynomial).
8. TSP: $O(n^3 2^n)$ time and $O(n^2 2^n)$ space.

Dynamic programming...

1. Longest increasing subsequence.
2. Computing the solution itself (not only its value).
3. Maximum Weight Independent Set in Trees.
4. Dynamic programs can be also solved as problems on DAGs.
5. Edit distance: $O(nm)$ [but linear space!].
6. Floyd-Warshall: $O(n^3)$.
7. Knapsack: $O(nW)$ (pseudo-polynomial).
8. TSP: $O(n^3 2^n)$ time and $O(n^2 2^n)$ space.

Dynamic programming...

1. Longest increasing subsequence.
2. Computing the solution itself (not only its value).
3. Maximum Weight Independent Set in Trees.
4. Dynamic programs can be also solved as problems on DAGs.
5. Edit distance: $O(nm)$ [but linear space!].
6. Floyd-Warshall: $O(n^3)$.
7. Knapsack: $O(nW)$ (pseudo-polynomial).
8. TSP: $O(n^3 2^n)$ time and $O(n^2 2^n)$ space.

Dynamic programming...

1. Longest increasing subsequence.
2. Computing the solution itself (not only its value).
3. Maximum Weight Independent Set in Trees.
4. Dynamic programs can be also solved as problems on DAGs.
5. Edit distance: $O(nm)$ [but linear space!].
6. Floyd-Warshall: $O(n^3)$.
7. Knapsack: $O(nW)$ (pseudo-polynomial).
8. TSP: $O(n^3 2^n)$ time and $O(n^2 2^n)$ space.

Dynamic programming...

1. Longest increasing subsequence.
2. Computing the solution itself (not only its value).
3. Maximum Weight Independent Set in Trees.
4. Dynamic programs can be also solved as problems on DAGs.
5. Edit distance: $O(nm)$ [but linear space!].
6. Floyd-Warshall: $O(n^3)$.
7. Knapsack: $O(nW)$ (pseudo-polynomial).
8. TSP: $O(n^3 2^n)$ time and $O(n^2 2^n)$ space.

Dynamic programming...

1. Longest increasing subsequence.
2. Computing the solution itself (not only its value).
3. Maximum Weight Independent Set in Trees.
4. Dynamic programs can be also solved as problems on DAGs.
5. Edit distance: $O(nm)$ [but linear space!].
6. Floyd-Warshall: $O(n^3)$.
7. Knapsack: $O(nW)$ (pseudo-polynomial).
8. TSP: $O(n^3 2^n)$ time and $O(n^2 2^n)$ space.

Dynamic programming...

1. Longest increasing subsequence.
2. Computing the solution itself (not only its value).
3. Maximum Weight Independent Set in Trees.
4. Dynamic programs can be also solved as problems on DAGs.
5. Edit distance: $O(nm)$ [but linear space!].
6. Floyd-Warshall: $O\left(n^3\right)$.
7. Knapsack: $O(nW)$ (pseudo-polynomial).
8. TSP: $O(n^3 2^n)$ time and $O(n^2 2^n)$ space.

Dynamic programming...

1. Longest increasing subsequence.
2. Computing the solution itself (not only its value).
3. Maximum Weight Independent Set in Trees.
4. Dynamic programs can be also solved as problems on DAGs.
5. Edit distance: $O(nm)$ [but linear space!].
6. Floyd-Warshall: $O(n^3)$.
7. Knapsack: $O(nW)$ (pseudo-polynomial).
8. TSP: $O(n^3 2^n)$ time and $O(n^2 2^n)$ space.

Greedy algorithms...

Greed has its place, but be careful not to be too greedy!

1. Must prove correctness of greedy algorithms.
2. Interval scheduling (so many variants that do not work).
   Proved correctness by showing that one can map the greedy solution to optimal.
3. Interval Partitioning/Coloring.
   Proved the depth of instance was # colors used by greedy.
4. Scheduling to Minimize Lateness.

Greedy algorithms...

Greed has its place, but be careful not to be too greedy!

1. Must prove correctness of greedy algorithms.
2. Interval scheduling (so many variants that do not work).
   Proved correctness by showing that one can map the greedy solution to optimal.
3. Interval Partitioning/Coloring.
   Proved the depth of instance was # colors used by greedy.
4. Scheduling to Minimize Lateness.

Greedy algorithms...

Greed has its place, but be careful not to be too greedy!

1. Must prove correctness of greedy algorithms.
2. Interval scheduling (so many variants that do not work). Proved correctness by showing that one can map the greedy solution to optimal.
3. Interval Partitioning/Coloring. Proved the depth of instance was # colors used by greedy.
4. Scheduling to Minimize Lateness.

Greedy algorithms...

Greed has its place, but be careful not to be too greedy!

1. Must prove correctness of greedy algorithms.
2. Interval scheduling (so many variants that do not work). Proved correctness by showing that one can map the greedy solution to optimal.
3. Interval Partitioning/Coloring. Proved the depth of instance was # colors used by greedy.
4. Scheduling to Minimize Lateness.

Greedy algorithms...

Greed has its place, but be careful not to be too greedy!

1. Must prove correctness of greedy algorithms.
2. Interval scheduling (so many variants that do not work).
   Proved correctness by showing that one can map the greedy solution to optimal.
3. Interval Partitioning/Coloring.
   Proved the depth of instance was # colors used by greedy.
4. Scheduling to Minimize Lateness.

Minimum spanning tree

1. Algorithms can be interpreted as being greedy.
2. Prim: T maintained by algorithm will be a tree. Start with a node in T. In each iteration, pick edge with least attachment cost to T.
3. Reverse delete: Delete edges keeping connectivity. Deleting edges from most expensive to cheapest.
4. Kruskal: Add edges in increasing price. Add edge only if merges two trees in the current forest.
5. Borůvka's: Every vertex pick cheapest edge out of it. Collapse connected components of chosen edges. Repeat till have a single tree.

Minimum spanning tree

1. Algorithms can be interpreted as being greedy.
2. Prim: T maintained by algorithm will be a tree. Start with a node in T. In each iteration, pick edge with least attachment cost to T.
3. Reverse delete: Delete edges keeping connectivity. Deleting edges from most expensive to cheapest.
4. Kruskal: Add edges in increasing price. Add edge only if merges two trees in the current forest.
5. Borůvka's: Every vertex pick cheapest edge out of it. Collapse connected components of chosen edges. Repeat till have a single tree.

Minimum spanning tree

1. Algorithms can be interpreted as being greedy.
2. Prim: T maintained by algorithm will be a tree. Start with a node in T. In each iteration, pick edge with least attachment cost to T.
3. Reverse delete: Delete edges keeping connectivity. Deleting edges from most expensive to cheapest.
4. Kruskal: Add edges in increasing price. Add edge only if merges two trees in the current forest.
5. Borůvka's: Every vertex pick cheapest edge out of it. Collapse connected components of chosen edges. Repeat till have a single tree.

Minimum spanning tree

1. Algorithms can be interpreted as being greedy.
2. Prim: T maintained by algorithm will be a tree. Start with a node in T. In each iteration, pick edge with least attachment cost to T.
3. Reverse delete: Delete edges keeping connectivity. Deleting edges from most expensive to cheapest.
4. Kruskal: Add edges in increasing price. Add edge only if merges two trees in the current forest.
5. Borůvka's: Every vertex pick cheapest edge out of it. Collapse connected components of chosen edges. Repeat till have a single tree.

Minimum spanning tree

1. Algorithms can be interpreted as being greedy.
2. Prim: T maintained by algorithm will be a tree. Start with a node in T. In each iteration, pick edge with least attachment cost to T.
3. Reverse delete: Delete edges keeping connectivity. Deleting edges from most expensive to cheapest.
4. Kruskal: Add edges in increasing price. Add edge only if merges two trees in the current forest.
5. Borůvka's: Every vertex pick cheapest edge out of it. Collapse connected components of chosen edges. Repeat till have a single tree.

Minimum spanning tree

1. Algorithms can be interpreted as being greedy.
2. Prim: T maintained by algorithm will be a tree. Start with a node in T. In each iteration, pick edge with least attachment cost to T.
3. Reverse delete: Delete edges keeping connectivity. Deleting edges from most expensive to cheapest.
4. Kruskal: Add edges in increasing price. Add edge only if merges two trees in the current forest.
5. Borůvka's: Every vertex pick cheapest edge out of it. Collapse connected components of chosen edges. Repeat till have a single tree.

Why MST algorithms work?

Definition

An edge $e = (u, v)$ is a safe edge if there is some partition of V into S and $V \setminus S$ and e is the unique minimum cost edge crossing S (one end in S and the other in $V \setminus S$).

Definition

An edge $e = (u, v)$ is an unsafe edge if there is some cycle C such that e is the unique maximum cost edge in C.

Proposition

If edge costs are distinct then every edge is either safe or unsafe.

Lemma

If e is a safe edge then every minimum spanning tree contains e.

Why MST algorithms work?

Definition

An edge $e = (u, v)$ is a safe edge if there is some partition of V into S and $V \setminus S$ and e is the unique minimum cost edge crossing S (one end in S and the other in $V \setminus S$).

Definition

An edge $e = (u, v)$ is an unsafe edge if there is some cycle C such that e is the unique maximum cost edge in C.

Proposition

If edge costs are distinct then every edge is either safe or unsafe.

Lemma

If e is a safe edge then every minimum spanning tree contains e.

Why MST algorithms work?

Definition

An edge $e = (u, v)$ is a safe edge if there is some partition of V into S and $V \setminus S$ and e is the unique minimum cost edge crossing S (one end in S and the other in $V \setminus S$).

Definition

An edge $e = (u, v)$ is an unsafe edge if there is some cycle C such that e is the unique maximum cost edge in C.

Proposition

If edge costs are distinct then every edge is either safe or unsafe.

Lemma

If e is a safe edge then every minimum spanning tree contains e.

Why MST algorithms work?

Definition

An edge $e = (u, v)$ is a safe edge if there is some partition of V into S and $V \setminus S$ and e is the unique minimum cost edge crossing S (one end in S and the other in $V \setminus S$).

Definition

An edge $e = (u, v)$ is an unsafe edge if there is some cycle C such that e is the unique maximum cost edge in C.

Proposition

If edge costs are distinct then every edge is either safe or unsafe.

Lemma

If e is a safe edge then every minimum spanning tree contains e.

Why MST algorithms work?

Even more

Lemma

Let G be a connected graph with distinct edge costs, then the set of safe edges form a connected graph.

Corollary

Let G be a connected graph with distinct edge costs, then set of safe edges form the unique MST of G.

Lemma

If e is an unsafe edge then no MST of G contains e.

Why MST algorithms work?

Even more

Lemma

Let G be a connected graph with distinct edge costs, then the set of safe edges form a connected graph.

Corollary

Let G be a connected graph with distinct edge costs, then set of safe edges form the unique MST of G.

Lemma

If e is an unsafe edge then no MST of G contains e.

Why MST algorithms work?

Even more

Lemma

Let G be a connected graph with distinct edge costs, then the set of safe edges form a connected graph.

Corollary

Let G be a connected graph with distinct edge costs, then set of safe edges form the unique MST *of G.*

Lemma

If e is an unsafe edge then no MST *of G contains e.*

Data structures for MST

1. Heap.
2. Fibonacci heap.
3. Union-find - path compression and union by rank.
   (Amazing running time - $O(\alpha(m, n))$ per operation,)

Data structures for MST

1. Heap.
2. Fibonacci heap.
3. Union-find - path compression and union by rank. (Amazing running time - $O(\alpha(m,n))$ per operation,)

Data structures for MST

1. Heap.
2. Fibonacci heap.
3. Union-find - path compression and union by rank. (Amazing running time - $O(\alpha(m, n))$ per operation,)

Data structures for MST

1. Heap.
2. Fibonacci heap.
3. Union-find - path compression and union by rank.
   (Amazing running time - $O(\alpha(m, n))$ per operation,)

Randomized algorithms

1. Basic concepts in discrete probability:
   Random variable, probability, expectation, linearity of expectation, independent events, conditional probability, indicator variables.
2. Types of randomized algorithms: Las Vegas and Monte Carlo.
3. Why randomization works - concentration of mass.
4. Proved:

Theorem

Let X_n be the number heads when flipping a coin indepdently n times. Then

$$\Pr\left[X_n \leq \frac{n}{4}\right] \leq 2 \cdot 0.68^{n/4} \text{ and } \Pr\left[X_n \geq \frac{3n}{4}\right] \leq 2 \cdot 0.68^{n/4}$$

Randomized algorithms

1. Basic concepts in discrete probability:
   Random variable, probability, expectation, linearity of expectation, independent events, conditional probability, indicator variables.
2. Types of randomized algorithms: Las Vegas and Monte Carlo.
3. Why randomization works - concentration of mass.
4. Proved:

Theorem

Let X_n be the number heads when flipping a coin indepdently n times. Then

$$\Pr\left[X_n \leq \frac{n}{4}\right] \leq 2 \cdot 0.68^{n/4} \text{ and } \Pr\left[X_n \geq \frac{3n}{4}\right] \leq 2 \cdot 0.68^{n/4}$$

Randomized algorithms

1. Basic concepts in discrete probability:
   Random variable, probability, expectation, linearity of expectation, independent events, conditional probability, indicator variables.
2. Types of randomized algorithms: Las Vegas and Monte Carlo.
3. Why randomization works - concentration of mass.
4. Proved:

Theorem

Let X_n be the number heads when flipping a coin indepdently n times. Then

$$\Pr\left[X_n \leq \frac{n}{4}\right] \leq 2 \cdot 0.68^{n/4} \text{ and } \Pr\left[X_n \geq \frac{3n}{4}\right] \leq 2 \cdot 0.68^{n/4}$$

Randomized algorithms

1. Basic concepts in discrete probability:
   Random variable, probability, expectation, linearity of expectation, independent events, conditional probability, indicator variables.
2. Types of randomized algorithms: Las Vegas and Monte Carlo.
3. Why randomization works - concentration of mass.
4. Proved:

Theorem

Let X_n be the number heads when flipping a coin indepdently n times. Then

$$\Pr\left[X_n \leq \frac{n}{4}\right] \leq 2 \cdot 0.68^{n/4} \text{ and } \Pr\left[X_n \geq \frac{3n}{4}\right] \leq 2 \cdot 0.68^{n/4}$$

Randomized algorithms

1. Basic concepts in discrete probability:
   Random variable, probability, expectation, linearity of expectation, independent events, conditional probability, indicator variables.
2. Types of randomized algorithms: Las Vegas and Monte Carlo.
3. Why randomization works - concentration of mass.
4. Proved:

Theorem

Let X_n be the number heads when flipping a coin indepdently n times. Then

$$\Pr\left[X_n \leq \frac{n}{4}\right] \leq 2 \cdot 0.68^{n/4} \text{ and } \Pr\left[X_n \geq \frac{3n}{4}\right] \leq 2 \cdot 0.68^{n/4}$$

Randomized algorithms

1. Proved **QuickSort** has $O(n \log n)$ expected running time.
2. Proved **QuickSort** has $O(n \log n)$ running time with high probability.
3. Proved **QuickSelect** has $O(n)$ expected running time.
4. Hashing.
    1. Why randomization is a must.
    2. **2-universal** hash functions families.
    3. Showed/proved a 2-universal hash family.
       Guess two random numbers α and β. Hash function is $h(x) = (\alpha x + \beta) \mod p$.

Randomized algorithms

1. Proved **QuickSort** has $O(n \log n)$ expected running time.
2. Proved **QuickSort** has $O(n \log n)$ running time with high probability.
3. Proved **QuickSelect** has $O(n)$ expected running time.
4. Hashing.
   1. Why randomization is a must.
   2. **2-universal** hash functions families.
   3. Showed/proved a 2-universal hash family. Guess two random numbers α and β. Hash function is $h(x) = (\alpha x + \beta) \mod p$.

Randomized algorithms

1. Proved **QuickSort** has $O(n \log n)$ expected running time.
2. Proved **QuickSort** has $O(n \log n)$ running time with high probability.
3. Proved **QuickSelect** has $O(n)$ expected running time.
4. Hashing.
    1. Why randomization is a must.
    2. 2-**universal** hash functions families.
    3. Showed/proved a 2-universal hash family.
       Guess two random numbers α and β. Hash function is $h(x) = (\alpha x + \beta) \mod p$.

Randomized algorithms

1. Proved **QuickSort** has $O(n \log n)$ expected running time.
2. Proved **QuickSort** has $O(n \log n)$ running time with high probability.
3. Proved **QuickSelect** has $O(n)$ expected running time.
4. Hashing.
    1. Why randomization is a must.
    2. 2-**universal** hash functions families.
    3. Showed/proved a 2-universal hash family.
       Guess two random numbers α and β. Hash function is $h(x) = (\alpha x + \beta) \mod p$.

Randomized algorithms

1. Proved **QuickSort** has $O(n \log n)$ expected running time.
2. Proved **QuickSort** has $O(n \log n)$ running time with high probability.
3. Proved **QuickSelect** has $O(n)$ expected running time.
4. Hashing.
   1. Why randomization is a must.
   2. 2-**universal** hash functions families.
   3. Showed/proved a 2-universal hash family.
      Guess two random numbers α and β. Hash function is $h(x) = (\alpha x + \beta) \mod p$.

Randomized algorithms

1. Proved **QuickSort** has $O(n \log n)$ expected running time.
2. Proved **QuickSort** has $O(n \log n)$ running time with high probability.
3. Proved **QuickSelect** has $O(n)$ expected running time.
4. Hashing.
    1. Why randomization is a must.
    2. **2-universal** hash functions families.
    3. Showed/proved a 2-universal hash family.
       Guess two random numbers α and β. Hash function is $h(x) = (\alpha x + \beta) \mod p$.

Randomized algorithms

1. Proved **QuickSort** has $O(n \log n)$ expected running time.
2. Proved **QuickSort** has $O(n \log n)$ running time with high probability.
3. Proved **QuickSelect** has $O(n)$ expected running time.
4. Hashing.
   1. Why randomization is a must.
   2. **2-universal** hash functions families.
   3. Showed/proved a 2-universal hash family.
      Guess two random numbers α and β. Hash function is $h(x) = (\alpha x + \beta) \mod p$.

Randomized algorithms

1. Proved **QuickSort** has $O(n \log n)$ expected running time.
2. Proved **QuickSort** has $O(n \log n)$ running time with high probability.
3. Proved **QuickSelect** has $O(n)$ expected running time.
4. Hashing.
    1. Why randomization is a must.
    2. **2-universal** hash functions families.
    3. Showed/proved a 2-universal hash family.
       Guess two random numbers α and β. Hash function is $h(x) = (\alpha x + \beta) \mod p$.

Network Flow

1. Definitions.
2. Edge flow $\Leftrightarrow$ path flow.
3. Max-flow problem.
4. Cuts and minimum-cut.
5. flow $\leq$ cut capacity.
6. Max-flow Min-cut Theorem.
7. Residual network.
8. Augmenting paths.
9. Ford-Fulkerson Algorithm.
10. Proved correctness of Ford-Fulkerson Algorithm if capacities are integral.

Network Flow

1. Definitions.
2. Edge flow $\Leftrightarrow$ path flow.
3. Max-flow problem.
4. Cuts and minimum-cut.
5. flow $\leq$ cut capacity.
6. Max-flow Min-cut Theorem.
7. Residual network.
8. Augmenting paths.
9. Ford-Fulkerson Algorithm.
10. Proved correctness of Ford-Fulkerson Algorithm if capacities are integral.

Network Flow

1. Definitions.
2. Edge flow $\Leftrightarrow$ path flow.
3. Max-flow problem.
4. Cuts and minimum-cut.
5. flow $\leq$ cut capacity.
6. Max-flow Min-cut Theorem.
7. Residual network.
8. Augmenting paths.
9. Ford-Fulkerson Algorithm.
10. Proved correctness of Ford-Fulkerson Algorithm if capacities are integral.

Network Flow

1. Definitions.
2. Edge flow $\Leftrightarrow$ path flow.
3. Max-flow problem.
4. Cuts and minimum-cut.
5. flow $\leq$ cut capacity.
6. Max-flow Min-cut Theorem.
7. Residual network.
8. Augmenting paths.
9. Ford-Fulkerson Algorithm.
10. Proved correctness of Ford-Fulkerson Algorithm if capacities are integral.

Network Flow

1. Definitions.
2. Edge flow $\Leftrightarrow$ path flow.
3. Max-flow problem.
4. Cuts and minimum-cut.
5. flow $\leq$ cut capacity.
6. Max-flow Min-cut Theorem.
7. Residual network.
8. Augmenting paths.
9. Ford-Fulkerson Algorithm.
10. Proved correctness of Ford-Fulkerson Algorithm if capacities are integral.

Network Flow

1. Definitions.
2. Edge flow $\Leftrightarrow$ path flow.
3. Max-flow problem.
4. Cuts and minimum-cut.
5. flow $\leq$ cut capacity.
6. Max-flow Min-cut Theorem.
7. Residual network.
8. Augmenting paths.
9. Ford-Fulkerson Algorithm.
10. Proved correctness of Ford-Fulkerson Algorithm if capacities are integral.

Network Flow

1. Definitions.
2. Edge flow $\Leftrightarrow$ path flow.
3. Max-flow problem.
4. Cuts and minimum-cut.
5. flow $\leq$ cut capacity.
6. Max-flow Min-cut Theorem.
7. Residual network.
8. Augmenting paths.
9. Ford-Fulkerson Algorithm.
10. Proved correctness of Ford-Fulkerson Algorithm if capacities are integral.

Network Flow

1. Definitions.
2. Edge flow $\Leftrightarrow$ path flow.
3. Max-flow problem.
4. Cuts and minimum-cut.
5. flow $\leq$ cut capacity.
6. Max-flow Min-cut Theorem.
7. Residual network.
8. Augmenting paths.
9. Ford-Fulkerson Algorithm.
10. Proved correctness of Ford-Fulkerson Algorithm if capacities are integral.

Network Flow

1. Definitions.
2. Edge flow $\Leftrightarrow$ path flow.
3. Max-flow problem.
4. Cuts and minimum-cut.
5. flow $\leq$ cut capacity.
6. Max-flow Min-cut Theorem.
7. Residual network.
8. Augmenting paths.
9. Ford-Fulkerson Algorithm.
10. Proved correctness of Ford-Fulkerson Algorithm if capacities are integral.

Network Flow

1. Definitions.
2. Edge flow $\Leftrightarrow$ path flow.
3. Max-flow problem.
4. Cuts and minimum-cut.
5. flow $\leq$ cut capacity.
6. Max-flow Min-cut Theorem.
7. Residual network.
8. Augmenting paths.
9. Ford-Fulkerson Algorithm.
10. Proved correctness of Ford-Fulkerson Algorithm if capacities are integral.

Network Flow

1. Definitions.
2. Edge flow $\Leftrightarrow$ path flow.
3. Max-flow problem.
4. Cuts and minimum-cut.
5. flow $\leq$ cut capacity.
6. Max-flow Min-cut Theorem.
7. Residual network.
8. Augmenting paths.
9. Ford-Fulkerson Algorithm.
10. Proved correctness of Ford-Fulkerson Algorithm if capacities are integral.

Network Flow II

1. Ford-Fulkerson running time is $O(mC)$.
2. Mentioned the strongly polynomial time algorithm by Edmonds-Karp.
3. Computing minimum cut from max-flow.
4. One can convert a flow to an acyclic flow.
5. A flow can be decomposed into paths from the source to the target + cycles.
6. Computing edge-disjoint paths using flow.
7. Computing vertex-disjoint paths using flow.
8. Menger's theorem (# edge to cut = # edge disjoint paths).
9. Multiple sinks/sources.
10. Matching in bipartite graph.
11. Perfect matching.

Network Flow II

1. Ford-Fulkerson running time is $O(mC)$.
2. Mentioned the strongly polynomial time algorithm by Edmonds-Karp.
3. Computing minimum cut from max-flow.
4. One can convert a flow to an acyclic flow.
5. A flow can be decomposed into paths from the source to the target + cycles.
6. Computing edge-disjoint paths using flow.
7. Computing vertex-disjoint paths using flow.
8. Menger's theorem (# edge to cut = # edge disjoint paths).
9. Multiple sinks/sources.
10. Matching in bipartite graph.
11. Perfect matching.

Network Flow II

1. Ford-Fulkerson running time is $O(mC)$.
2. Mentioned the strongly polynomial time algorithm by Edmonds-Karp.
3. Computing minimum cut from max-flow.
4. One can convert a flow to an acyclic flow.
5. A flow can be decomposed into paths from the source to the target + cycles.
6. Computing edge-disjoint paths using flow.
7. Computing vertex-disjoint paths using flow.
8. Menger's theorem (# edge to cut = # edge disjoint paths).
9. Multiple sinks/sources.
10. Matching in bipartite graph.
11. Perfect matching.

Network Flow II

1. Ford-Fulkerson running time is $O(mC)$.
2. Mentioned the strongly polynomial time algorithm by Edmonds-Karp.
3. Computing minimum cut from max-flow.
4. One can convert a flow to an acyclic flow.
5. A flow can be decomposed into paths from the source to the target + cycles.
6. Computing edge-disjoint paths using flow.
7. Computing vertex-disjoint paths using flow.
8. Menger's theorem (# edge to cut = # edge disjoint paths).
9. Multiple sinks/sources.
10. Matching in bipartite graph.
11. Perfect matching.

Network Flow II

1. Ford-Fulkerson running time is $O(mC)$.
2. Mentioned the strongly polynomial time algorithm by Edmonds-Karp.
3. Computing minimum cut from max-flow.
4. One can convert a flow to an acyclic flow.
5. A flow can be decomposed into paths from the source to the target + cycles.
6. Computing edge-disjoint paths using flow.
7. Computing vertex-disjoint paths using flow.
8. Menger's theorem (# edge to cut = # edge disjoint paths).
9. Multiple sinks/sources.
10. Matching in bipartite graph.
11. Perfect matching.

Network Flow II

1. Ford-Fulkerson running time is $O(mC)$.
2. Mentioned the strongly polynomial time algorithm by Edmonds-Karp.
3. Computing minimum cut from max-flow.
4. One can convert a flow to an acyclic flow.
5. A flow can be decomposed into paths from the source to the target + cycles.
6. Computing edge-disjoint paths using flow.
7. Computing vertex-disjoint paths using flow.
8. Menger's theorem (# edge to cut = # edge disjoint paths).
9. Multiple sinks/sources.
10. Matching in bipartite graph.
11. Perfect matching.

Network Flow II

1. Ford-Fulkerson running time is $O(mC)$.
2. Mentioned the strongly polynomial time algorithm by Edmonds-Karp.
3. Computing minimum cut from max-flow.
4. One can convert a flow to an acyclic flow.
5. A flow can be decomposed into paths from the source to the target + cycles.
6. Computing edge-disjoint paths using flow.
7. Computing vertex-disjoint paths using flow.
8. Menger's theorem (# edge to cut = # edge disjoint paths).
9. Multiple sinks/sources.
10. Matching in bipartite graph.
11. Perfect matching.

Network Flow II

1. Ford-Fulkerson running time is $O(mC)$.
2. Mentioned the strongly polynomial time algorithm by Edmonds-Karp.
3. Computing minimum cut from max-flow.
4. One can convert a flow to an acyclic flow.
5. A flow can be decomposed into paths from the source to the target + cycles.
6. Computing edge-disjoint paths using flow.
7. Computing vertex-disjoint paths using flow.
8. Menger's theorem (# edge to cut = # edge disjoint paths).
9. Multiple sinks/sources.
10. Matching in bipartite graph.
11. Perfect matching.

Network Flow II

1. Ford-Fulkerson running time is $O(mC)$.
2. Mentioned the strongly polynomial time algorithm by Edmonds-Karp.
3. Computing minimum cut from max-flow.
4. One can convert a flow to an acyclic flow.
5. A flow can be decomposed into paths from the source to the target + cycles.
6. Computing edge-disjoint paths using flow.
7. Computing vertex-disjoint paths using flow.
8. Menger's theorem (# edge to cut = # edge disjoint paths).
9. Multiple sinks/sources.
10. Matching in bipartite graph.
11. Perfect matching.

Network Flow II

1. Ford-Fulkerson running time is $O(mC)$.
2. Mentioned the strongly polynomial time algorithm by Edmonds-Karp.
3. Computing minimum cut from max-flow.
4. One can convert a flow to an acyclic flow.
5. A flow can be decomposed into paths from the source to the target + cycles.
6. Computing edge-disjoint paths using flow.
7. Computing vertex-disjoint paths using flow.
8. Menger's theorem (# edge to cut = # edge disjoint paths).
9. Multiple sinks/sources.
10. Matching in bipartite graph.
11. Perfect matching.

Network Flow II

1. Ford-Fulkerson running time is $O(mC)$.
2. Mentioned the strongly polynomial time algorithm by Edmonds-Karp.
3. Computing minimum cut from max-flow.
4. One can convert a flow to an acyclic flow.
5. A flow can be decomposed into paths from the source to the target + cycles.
6. Computing edge-disjoint paths using flow.
7. Computing vertex-disjoint paths using flow.
8. Menger's theorem (# edge to cut = # edge disjoint paths).
9. Multiple sinks/sources.
10. Matching in bipartite graph.
11. Perfect matching.

Network Flow II

1. Ford-Fulkerson running time is $O(mC)$.
2. Mentioned the strongly polynomial time algorithm by Edmonds-Karp.
3. Computing minimum cut from max-flow.
4. One can convert a flow to an acyclic flow.
5. A flow can be decomposed into paths from the source to the target + cycles.
6. Computing edge-disjoint paths using flow.
7. Computing vertex-disjoint paths using flow.
8. Menger's theorem (# edge to cut = # edge disjoint paths).
9. Multiple sinks/sources.
10. Matching in bipartite graph.
11. Perfect matching.

Network Flow III

1. Deciding if a specific team can win the Pennant using network flow.
2. Project scheduling.
3. Mentioned extensions to min-cost flow, and lower bounds on flow.
4. Circulations.
5. Survey design (using lower/upper bounds on flow).

Network Flow III

1. Deciding if a specific team can win the Pennant using network flow.
2. Project scheduling.
3. Mentioned extensions to min-cost flow, and lower bounds on flow.
4. Circulations.
5. Survey design (using lower/upper bounds on flow).

Network Flow III

1. Deciding if a specific team can win the Pennant using network flow.
2. Project scheduling.
3. Mentioned extensions to min-cost flow, and lower bounds on flow.
4. Circulations.
5. Survey design (using lower/upper bounds on flow).

Network Flow III

1. Deciding if a specific team can win the Pennant using network flow.
2. Project scheduling.
3. Mentioned extensions to min-cost flow, and lower bounds on flow.
4. Circulations.
5. Survey design (using lower/upper bounds on flow).

Network Flow III

1. Deciding if a specific team can win the Pennant using network flow.
2. Project scheduling.
3. Mentioned extensions to min-cost flow, and lower bounds on flow.
4. Circulations.
5. Survey design (using lower/upper bounds on flow).

Network Flow III

1. Deciding if a specific team can win the Pennant using network flow.
2. Project scheduling.
3. Mentioned extensions to min-cost flow, and lower bounds on flow.
4. Circulations.
5. Survey design (using lower/upper bounds on flow).

Notes

Notes

Notes

Notes