Graphs and Network Flows
IE411

Final Review

Dr. Ted Ralphs
What was this class be about?

- Modeling of Network Flow Problems
- Mathematical Structure of Network Flow Models
- Solution and Analysis of Network Flow Models
- Algorithm Design and Analysis
What were the goals for the course?

After this course, you should be able to:

• Recognize when an optimization problem has an underlying network flow structure and model it as such.

• Understand the underlying mathematical foundations of network flow models.

• Assess the computational complexity of a given network flow model.

• Understand the range of algorithms available for solving network flow problems and be able to choose an appropriate algorithm.

• Implement an algorithm to solve a network flow problems.

• Analyze a given algorithm and determine whether it is efficient.
Take Home Messages

- Network flow problems are ubiquitous and underlie many complex models in the real world.

- Network flow structure can be analyzed efficiently and this fact is important to understand when analyzing more complex models.

- Efficient algorithms for more complex models can be derived by decompositions that exploit network flow structure.

- The details of how an algorithm is implemented are extremely important when it comes to achieving efficiency.

- Learning how to efficiently implement network flow algorithms is a good way to understand more general principles of algorithms design and analysis.
General Topical Coverage

• Graphs and data structures for representing them
• Computational complexity and asymptotic analysis
• The graph search paradigm
• Problem classes
  – Connected components
  – Topological ordering
  – Shortest path problem
  – Maximum flow problem
  – Minimum cost network flow problem
  – Minimum weight spanning tree problem
  – Matching problem
  – Assignment problem
  – Multicommodity flow problems
Graphs and Data structures

- The essence of a graph is a ground set of elements $\mathcal{N}$ and a set $\mathcal{A}$ of pairs of those elements (ordered or unordered) that represent relationships among the elements.

- Data structures
  - Node-Arc Incidence Matrix
  - Node-Node Adjacency Matrix
  - Adjacency List
  - Forward Star (Reverse Star)
Computational Complexity: What is the objective?

- Complexity analysis is aimed at answering two types of questions.
  - How hard is a given problem?
  - How efficient is a given algorithm for a given problem?

- The usual measure of efficiency is *running time*, usually defined as the number of elementary operations required (more on this later).

- The running time will differ by instance, algorithm, and computing platform.

- How should we measure the performance so that we can select the “best” algorithm from among several?
What do We Measure?

Three methods of analysis:

- **Empirical analysis**
  - Try to determine how algorithms behave in practice

- **Average-case analysis**
  - Try to determine the expected number of steps an algorithm will take analytically.

- **Worst-case analysis**
  - Provide an upper bound on the number of steps an algorithm can take on any instance.
Asymptotic Analysis

• So far, we have determined that our measure of *running time* will be a function of instance size (a positive integer).

• Determining the exact function is still problematic at best.

• We will only really be interested in approximately how quickly the function grows “*in the limit*”.

• To determine this, we will use *asymptotic analysis*.

• Order relations

\[ f(n) \in O(g(n)) \iff \exists c \in \mathbb{R}_+, n_0 \in \mathbb{Z}_+ \text{ s.t. } f(n) \leq cg(n) \forall n \geq n_0. \]

• In this case, we say *f is order g* or *f is ‘big O’ of g*.

• Using this relation, we can divide functions into classes that are all *of the same order*. 
Running Time and Complexity

- **Running time** is a measure of the efficiency of an algorithm.

- **Computational complexity** is a measure of the difficulty of a problem.

- The computational complexity of a problem is the running time of the best possible algorithm.

- In most cases, we cannot prove that the best known algorithm is also the best possible algorithm.

- We can therefore only provide an upper bound on the computational complexity in most cases.

- That is why complexity is usually expressed using “big O” notation.

- A case in which we know the exact complexity is comparison-based sorting, but this is unusual.
Search Algorithms

- *Search algorithms* are fundamental techniques applied to solve a wide range of optimization problems.

- Search algorithms attempt to find all the nodes in a network satisfying a particular property.

- **Examples**
  - Find nodes that are reachable by directed paths from a source node.
  - Find nodes that can reach a specific node along directed paths
  - Identify the connected components of a network
  - Identify directed cycles in network
Basic Search Algorithm

This is the basic search algorithm.

**Require:** Graph $G = (N, A)$

1: $Q \leftarrow \{s\}$
2: while $Q \neq \emptyset$ do
3: let $v$ be any element of $Q$
4: remove $v$ from $Q$
5: mark $v$
6: for $v' \in A(v)$ do
7: if $v'$ is not marked then
8: $Q \leftarrow Q \cup \{v'\}$
9: end if
10: end for
11: end while
**Topological Ordering**

- In a directed graph, the arcs can be thought of as representing *precedence constraints*.

- In other words, an arc \((i, j)\) represents the constraint that node \(i\) must come before node \(j\).

- Given a graph \(G = (N, A)\) with the nodes labeled with distinct numbers \(1\) through \(n\), let \(\text{order}(i)\) be the label of node \(i\).

- Then, this labeling is a *topological ordering* of the nodes if for every arc \((i, j) \in A\), \(\text{order}(i) < \text{order}(j)\).

- Can all graphs be topologically ordered?
Topological Ordering

The following algorithm will detect presence of a directed cycle or produce a topological ordering of the nodes.

**Require:** Directed acyclic graph $G = (N, A)$

**Ensure:** The array `order` is a topological ordering of $N$.

```
count ← 1
while \{v ∈ N : I(v) = 0\} ≠ \emptyset do
    let $v$ be any vertex with $I(v) = 0$
    order[$v$] ← count
    count ← count + 1
    delete $v$ and all outgoing arcs from $G$
end while
if $V = \emptyset$ then
    return success
else
    report failure
end if
```
The shortest path problem underlies virtually all network flow problems.

- **Variants**
  - Single Source
    * Acyclic
    * Non-negative arc lengths
    * Arbitrary arc lengths
  - All Pairs

**Definition 1.** Given a directed network $G = (N, A)$ with an arc length $c_{ij}$ associated with each arc $(i, j) \in A$ and a distinguished node $s$, the shortest path problem is to determine a shortest length directed path from node $s$ to every node $i \in N - \{s\}$.
Shortest Path Algorithms

- Label Setting (Chapter 4)
  * one label becomes permanent during each iteration
  * acyclic with arbitrary arc lengths OR non-negative arc lengths
- Label Correcting (Chapter 5)
  * all labels are temporary until last iteration
  * more general graphs including negative arc lengths
- Both are iterative; they differ in label update procedure and convergence procedure.
Optimality Conditions

Theorem 1. [5.1] For every node \( j \in N \), let \( d(j) \) denote the length of some directed path from the source node to node \( j \). Then, the numbers \( d(j) \) represent the shortest path distances if and only if they satisfy the following for all \( (i, j) \in A \):

\[
d(j) \leq d(i) + c_{ij}.
\]

Theorem 2. For every pair of nodes \( [i, j] \in N \times N \), let \( d[i, j] \) represent the length of some directed path from node \( i \) to node \( j \) satisfying \( d[i, i] = 0 \ \forall i \in N \) and \( d[i, j] \leq c_{ij} \ \forall (i, j) \in A \). These distances represent shortest path distances if and only if they satisfy

\[
d[i, j] \leq d[i, k] + d[k, j] \ \forall i, j, k \in N.
\]
**Dijkstra’s Algorithm**

**Require:** An acyclic network $G = (N, A)$ and a vector of arc lengths $c \in \mathbb{Z}^A_+$

**Ensure:** $d(i)$ is the length of a shortest path from node $s$ to node $i$ and $\text{pred}(i)$ is the immediate predecessor of $i$ in an associated shortest paths tree.

$S := \emptyset$

$\bar{S} := N$

$d(i) \leftarrow \infty \forall i \in N$

$d(s) \leftarrow 0$ and $\text{pred}(s) \leftarrow 0$

**while** $|S| < n$ **do**

let $i \in \bar{S}$ be the node for which $d(i) = \min\{d(j) : j \in \bar{S}\}$

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

$\bar{S} \leftarrow \bar{S} \setminus \{i\}$

**for** $(i, j) \in A(i)$ **do**

if $d(j) > d(i) + c_{ij}$ then

$d(j) \leftarrow d(i) + c_{ij}$ and $\text{pred}(j) \leftarrow i$

end if

end for

end while
General Label-Correcting Algorithms

Maintain a distance label $d(j)$ for all nodes $j \in N$

– If $d(j)$ is infinite, the algorithm has not found a path joining the source node to node $j$.
– If $d(j)$ is finite, it is the distance from the source node to that node along some path (upper bound).
– No label is permanent until the algorithm terminates.
All-Pairs Label-Correcting Algorithm

Require: A network $G = (N, A)$ and a vector of arc lengths $c \in \mathbb{Z}^A$
Ensure: $d[i, j]$ is the length of a shortest path from node $i$ to node $j$ for pairs $i$ and $j$.

\begin{align*}
& d[i, j] \leftarrow \infty \text{ for all } [i, j] \in N \times N \\
& d[i, j] \leftarrow 0 \text{ for all } i \in N \\
& \text{for } (i, j) \in A \text{ do} \\
& \quad d[i, j] \leftarrow c_{ij} \\
& \quad \text{while } \exists (i, j, k) \text{ satisfying } d[i, j] > d[i, k] + d[k, j] \text{ do} \\
& \quad \quad d[i, j] := d[i, k] + d[k, j] \\
& \quad \text{end while} \\
& \text{end for}
\end{align*}
Floyd-Warshall Algorithm

**Require:** A network $G = (N, A)$ and a vector of arc lengths $c \in \mathbb{Z}^A$

**Ensure:** $d[i, j]$ is the length of a shortest path from node $i$ to node $j$ for pairs $i$ and $j$.

for $(i, j) \in N \times N$ do
  $d[i, j] \leftarrow \infty$ and $pred[i, j] \leftarrow 0$
end for

for $i \in N$ do
  $d[i, i] \leftarrow 0$
end for

for $(i, j) \in A$ do
  $d[i, j] \leftarrow c_{ij}$ and $pred[i, j] := i$
end for

for $k = 1$ to $n$ do
  for $[i, j] \in N \times N$ do
    if $d[i, j] > d[i, k] + d[k, j]$ then
      $d[i, j] \leftarrow d[i, k] + d[k, j]$
      $pred[i, j] \leftarrow pred[k, j]$
    end if
  end for
end for
Maximum Flow Problem

Given a network $G = (N, A)$ with a non-negative capacity $u_{ij}$ associated with each arc $(i, j) \in A$ and two nodes $s$ and $t$, find the maximum flow from $s$ to $t$ that satisfies the arc capacities.

Maximize $v$

subject to

\[
\sum_{j : (s, j) \in A} x_{sj} - \sum_{j : (j, s) \in A} x_{js} = v \quad (2)
\]

\[
\sum_{j : (i, j) \in A} x_{ij} - \sum_{j : (j, i) \in A} x_{ji} = 0 \quad \forall i \in N \setminus \{s, t\} \quad (3)
\]

\[
\sum_{j : (t, j) \in A} x_{tj} - \sum_{j : (j, t) \in A} x_{jt} = -v \quad (4)
\]

\[
x_{ij} \leq u_{ij} \quad \forall (i, j) \in A \quad (5)
\]

\[
x_{ij} \geq 0 \quad \forall (i, j) \in A \quad (6)
\]
Residual Network

- Suppose that an arc \((i, j)\) with capacity \(u_{ij}\) carries \(x_{ij}\) units of flow.
- Then, we can send up to \(u_{ij} - x_{ij}\) additional units of flow.
- We can also send up to \(x_{ij}\) units of flow backwards, canceling the existing flow and decreasing the flow cost.
- The \textit{residual network} \(G(x^0)\) is defined with respect to a given flow \(x^0\) and consists of arcs with positive residual capacity.
- Note that if for some pair of nodes \(i\) and \(j\), \(G\) already contains both \((i, j)\) and \((j, i)\), the residual network may contain parallel arcs with different residual capacities.
Cuts

– A cut is a partition of the node set $N$ into two parts $S$ and $\bar{S} = N \setminus S$.
– An $s − t$ cut is defined with respect to two distinguished nodes $s$ and $t$ and is a cut $[S, \bar{S}]$ such that $s \in S$ and $t \in \bar{S}$.
– A forward arc with respect to a cut is an arc $(i, j)$ with $i \in S$ and $j \in \bar{S}$.
– A backward arc with respect to a cut is an arc $(i, j)$ with $i \in \bar{S}$ and $j \in S$.

Property 1. [6.1] The value of any feasible flow is less than or equal to the capacity of any cut in the network.
Generic Augmenting Path Algorithm

- An *augmenting path* is a directed path from the source to the sink in the *residual* network.
- The *residual capacity* of an augmenting path is the minimum residual capacity of any arc in the path, which we denote by $\delta$.
  * By definition, $\delta > 0$.
  * When the network contains an augmenting path, we can send additional flow from the source to the sink.

**Theorem 3. [6.4]** A flow $x^*$ is a maximum flow if and only if the residual network $G(x^*)$ contains no augmenting path.
**Generic Augmenting Path Algorithm**

**Require:** A network \( G = (N, A) \) and a vector of capacities \( u \in \mathbb{Z}^A \)

**Ensure:** \( x \) represents the maximum flow from node \( s \) to node \( t \)

\[
x \leftarrow 0
\]

\[
\textbf{while} \ G(x) \text{ contains a directed path from } s \text{ to } t \text{ do}
\]

\[
\text{identify an augmenting path } P \text{ from } s \text{ to } t
\]

\[
\delta \leftarrow \min \{ r_{ij} : (i, j) \in P \}
\]

\[
\text{augment the flow along } P \text{ by } \delta \text{ units and update } G(x) \text{ accordingly.}
\]

**end while**
Identifying an Augmenting Path

- Use search technique to find a directed path in $G(x)$ from $s$ to $t$
  * At any step, partition nodes into labeled and unlabeled
  * Iteratively select a labeled node and scan its arc adjacency list in $G(x)$ to reach and label additional nodes
  * When sink becomes labeled, augment flow, erase labels and repeat
  * Terminate when all labeled nodes have been scanned and sink remains unlabeled
Distance Labels

A *distance function* \( d : N \to Z^+ \cup \{0\} \) with respect to the residual capacity \( r_{ij} \) is *valid* with respect to a flow \( x \) if it satisfies:

\[
\begin{align*}
d(t) &= 0 \\
d(i) &\leq d(j) + 1 \quad \forall (i, j) \in G(x)
\end{align*}
\]

**Property 2. [7.1]** *If the distance labels are valid, \( d(i) \) is a lower bound on the length of the shortest (directed) path from node \( i \) to node \( t \) in the residual network.*

**Property 3. [7.2]** *If \( d(s) \geq n \), then the residual network contains no directed path from \( s \) to \( t \).*

Distance labels are *exact* if \( d(i) \) equals the length of the shortest path from \( i \) to \( t \) in \( G(x) \) for all \( i \in N \).
Shortest Augmenting Path Algorithm

- Always augments flow along a shortest path from $s$ to $t$ in $G(x)$
- Proceeds by augmenting flows along admissible paths
- Constructs an admissible path incrementally – adding one arc at a time
- Maintains a partial admissible path and iteratively performs *advance* or *retreat* operations from current node
- Repeats operations until partial admissible path reaches sink node
Basic Idea of Preflow-Push Algorithm

– Select an active node $i$
– Try to remove the excess $e(i)$ by pushing flow on admissible arcs (push flow to neighbors of $i$ that are closer to $t$ as measured by $d$)
– If active node $i$ has no admissible arcs, increase its distance label
– Terminate when there are no active nodes
Generic Preflow-Push Algorithm

algorithm preflow-push
begin
    preprocess
    while the network contains an active node do
        select an active node \( i \)
        push/relabel\( (i) \)
end
Specific Implementations

By specifying different rules for selecting active nodes for push/relabel, we can derive different algorithms, each with different worst-case complexity.

**FIFO**: examine active nodes in FIFO order ($O(n^3)$)

**Highest-Label**: always push from an active node with highest value of distance label ($O(n^2 \sqrt{m})$)

**Excess Scaling**: push flow from node with sufficiently large excess to node with sufficiently small excess ($O(nm + n^2 \log U)$)
Summary of Maximum Flow Algorithms

Labeling Algorithm \( O(nmU) \)
Capacity Scaling Algorithm \( O(nm \log U) \)
Generic Preflow-Push Algorithm \( O(n^2m) \)
FIFO Preflow-Push Algorithm \( O(n^3) \)
Highest-Label Preflow-Push Algorithm \( O(n^2 \sqrt{m}) \)
Excess Scaling Algorithm \( O(nm + n^2 \log U) \)
Minimum Cost Network Flow Problem

- Let $G = (N, A)$ be a directed network with a cost $c_{ij}$ and a capacity $u_{ij}$ associated with every arc $(i, j) \in A$.
- Associated with each node $i \in N$ is a number $b(i)$; we refer to node $i$ as a supply node if $b(i) > 0$ and as a demand node if $b(i) < 0$.
- We let $x_{ij}$ denote the amount of flow sent on arc $(i, j)$.
- The objective of the minimum cost flow problem is to determine a least cost shipment of a commodity through a network in order to satisfy demands at certain nodes from available supplies at other nodes.
- Combines aspects of maximum flow and shortest path problems.
Optimality Conditions

- **Negative Cycle Optimality Conditions**
  A feasible solution $x^*$ is an optimal solution of the MCFP if and only if the residual network $G(x^*)$ contains no negative cost directed cycle.

- **Reduced Cost Optimality Conditions**
  A feasible solution $x^*$ is an optimal solution of the MCFP if and only if some set of node potentials $\pi$ satisfy
  
  $c^{\pi}_{ij} = c_{ij} - \pi(i) + \pi(j) \geq 0 \quad \forall (i, j) \in G(x^*).$

- **Complementary Slackness Optimality Conditions**: A feasible solution $x^*$ is an optimal solution of the MCFP if and only if for some set of node potentials $\pi$, the reduced costs and the flow values satisfy, for every $(i, j) \in A$,

  (a) If $c^{\pi}_{ij} > 0$, then $x^*_{ij} = 0$.

  (b) If $0 < x^*_{ij} < u_{ij}$, then $c^{\pi}_{ij} = 0$.

  (c) If $c^{\pi}_{ij} < 0$, then $x^*_{ij} = u_{ij}$.
Cycle-Canceling Algorithm

- Maintains a feasible solution; attempts to improve objective function value
- Establishes a feasible flow to start
- Iteratively finds a negative cost directed cycle in residual networks and augments flow along cycle
- Terminates when residual network contains no negative cost directed cycle
**Generic Cycle-Canceling Algorithm**

**algorithm** cycle-canceling (Klein, 1967)

**begin**

establish a feasible flow \( x \) in the network

**while** \( G(x) \) contains a negative cycle **do**

identify a negative cycle \( W \)

\[ \delta = \min \{ r_{ij} : (i, j) \in W \} \]

augment \( \delta \) units in cycle \( W \) and update \( G(x) \)

**end**
Relationship of Node Potentials and Flows

– Given an optimal flow $x^\ast$, we can obtain optimal node potentials by solving a shortest path problem (with possibly negative arc lengths).
– Given optimal node potentials $\pi$, we can obtain an optimal flow $x^\ast$ by solving a maximum flow problem.
– In linear programming terms, the flow is the primal solution and the node potentials are the corresponding dual solution.
Successive Shortest Paths Algorithm

- Start with a pseudo flow $x := 0$ and node potentials $\pi := 0$ and perform the following loop until feasibility is achieved.
  * Identify a node $s$ with an excess and a node $t$ with a deficit.
  * Determine shortest path distances $d$ from $s$ to all other nodes in the residual network with respect to the reduced costs $c_{ij}^\pi$.
  * Send flow along a shortest path from $s$ to $t$.
  * Update $\pi \rightarrow \pi - d$.
- From the previous slide, we know that sending flow along a shortest path and updating node potentials will maintain optimality conditions.
- The algorithm strictly decreases the excess at some node in each iteration.
- The number of iterations is at most $nU$, where $U$ is the largest supply.
**Primal-Dual Algorithm**

Transform minimum cost flow problem into a problem with a single excess node and a single deficit node

- Introduce a source node $s$ and a sink node $t$
- For each node $i$ with $b(i) > 0$, add a zero cost arc $(s, i)$ with capacity $b(i)$
- For each node $i$ with $b(i) < 0$, add a zero cost arc $(i, t)$ with capacity $-b(i)$
- Set $b(s) = \sum_{i \in N : b(i) > 0} b(i)$, $b(t) = -b(s)$ and $b(i) = 0 \ \forall i \in N$
Primal-Dual Algorithm

algorithm primal-dual
    \[ x := 0 \] and \[ \pi := 0 \]
    \[ e(s) := b(s) \] and \[ e(t) := b(t) \]
    while \[ e(s) > 0 \] do
        determine \( d(\cdot) \) from node \( s \) to all other nodes in \( G(x) \)
        with respect to \( c_{ij}^\pi \)
        update \( \pi := \pi - d \)
        define the admissible network \( G^\circ(x) \)
        establish a maximum flow from node \( s \) to node \( t \) in \( G^\circ(x) \)
        update \[ e(s), e(t) \] and \( G(x) \)
    end
Spanning Tree Solutions

• For any feasible solution $x$:
  
  – an arc $(i, j)$ is a free arc if $0 < x_{ij} < u_{ij}$
  – an arc $(i, j)$ is a restricted arc if $x_{ij} = 0$ or $x_{ij} = u_{ij}$

• We refer to a solution $x$ as a spanning tree solution if every non-tree arc is a restricted arc.

**Theorem 4. [11.2]** *If the objective function of a MCFP is bounded from below over the feasible region, the problem always has an optimal spanning tree solution.*
Optimality Conditions

Theorem 5. [11.3] A spanning tree structure \((T, L, U)\) is an optimal spanning tree structure of the MCFP if it is feasible and for some choice of node potentials \(\pi\), the arc reduced costs \(c_{ij}^\pi = c_{ij} - \pi(i) + \pi(j)\) satisfy

1. \(c_{ij}^\pi = 0 \ \forall (i, j) \in T\)
2. \(c_{ij}^\pi \geq 0 \ \forall (i, j) \in L\)
3. \(c_{ij}^\pi \leq 0 \ \forall (i, j) \in U\)
Network Simplex Algorithm

**algorithm** *network simplex*

determine an initial feasible tree structure \((T, L, U)\)

let \(x\) be flow and \(\pi\) be node potentials associated with \((T, L, U)\)

**while** some non-tree arc violates the optimality conditions **do**

select an entering arc \((k, l)\) violating optimality condition

add arc \((k, l)\) to tree and determine leaving arc \((p, q)\)

perform a tree update and update solutions \(x\) and \(\pi\)

**end while**

**end**
**Strongly Feasible Spanning Trees**

Let \((T, L, U)\) be a spanning tree structure for a MCFP with integral data. A spanning tree \(T\) is *strongly feasible* if every tree arc with zero flow is upward pointing (toward root) and every tree arc with flow equal to capacity is downward pointing (away from root).

- By maintaining a strongly feasible spanning tree solution in each iteration, we can guarantee termination even with degeneracy.
Sensitivity Analysis

• Determine changes in optimal solution resulting from changes in data
  – arc cost
  – supply/demand
  – arc capacity

• Assuming spanning tree structure remains unchanged, we check whether optimality conditions are still satisfied.

• If not, we can try to adjust solution to satisfy new conditions.
Minimum Spanning Trees

- The total cost of a spanning tree is the sum of the costs of the arcs in the tree.

- Given an undirected graph $G = (N, A)$ with $n$ nodes and $m$ arcs and with a length or cost $c_{ij}$ associated with each arc $(i, j) \in A$, the minimum spanning tree problem is to find a spanning tree with the smallest total cost (length).

- This is a combinatorial optimization problem.
Cut Optimality Conditions

Theorem 6. [13.1] A spanning tree $T^*$ is a minimum spanning tree if and only if for every tree arc $(i, j) \in T^*$, $c_{ij} \leq c_{kl}$ for every arc $(k, l)$ contained in the cut formed by deleting arc $(i, j)$ from $T^*$.

Theorem 7. [13.3] A spanning tree $T^*$ is a MST if and only if for every non-tree arc $(k, l)$ of $G$, $c_{ij} \leq c_{kl}$ for every arc $(i, j)$ contained in the path in $T^*$ connecting nodes $k$ and $l$. 
Prim’s and Kruskal’s

- Because spanning trees are the bases of a the graphic matroid, greedy algorithms can be used to find a minimum weight spanning tree.

- Prim’s algorithm
  - Builds up the tree one edge at a time as a single connected component.
  - Adds the cheapest edge crossing the cut between the current component and the remaining nodes.
  - Implementation is similar to Dijkstra’s algorithm.

- Kruskal’s algorithm
  - Builds up multiple components at the same time
  - Adds the cheapest edge that doesn’t form a cycle.
  - Implemented using union-find data structure.
Matching Problems

• MST and Matching Problems are two combinatorial optimization problems that are defined over graphs with a weight associated with each arc.

• A matching in a graph is a set of edges with the property that no two share a node.

• Two well-known matching problems
  – Find a matching that has as many edges as possible.
  – Given weights for each edge, find a matching with the largest total weight.

• Matching algorithms use the concept of augmentations, but detecting and performing augmentations efficiently is more complicated here.
Bipartite Matching and Network Flow

- A graph $G = (N, A)$ is a bipartite graph if we can partition its node set into two subsets $N_1$ and $N_2$ so that for each arc $(i, j) \in A$ either (i) $i \in N_1$ and $j \in N_2$ or (ii) $i \in N_2$ and $j \in N_1$.

- We can reduce bipartite matching problem to maximum flow problem for simple networks and solve efficiently by making use of any algorithm for maximum flow.

**Lemma 1.** The cardinality of the maximum matching in a bipartite graph equals the value of the maximum flow in the corresponding maximum flow network.
Lagrangian Relaxation

• Lagrangian Relaxation is a method of capitalizing on our ability to solve an underlying base model after adding side constraints.

• We remove the complicating constraints and instead assign a price associated with the resource.

• We can think of this price as a penalty for violation of the resource constraint.

• We set the prices, solve the underlying problem and then see if the resulting
Optimality Conditions

• When the constraints to be relaxed are equality constraints, if either

1. For some solution \( x \) to the original problem and some choice of multipliers \( \mu \), we have \( L(\mu) = cx \) or
2. For some choice of multipliers, the solution \( x \) to the Lagrangian subproblem is feasible for the original problem,

then \( x \) is optimal for the original problem.

• When some constraints are inequalities, we must also have complementary slackness, which says that the product of the multiplier and the slack for each constraint must be zero.
Solving the Lagrangian Dual

- Let us consider what $L(\mu)$ looks like as a function.
- We will consider the constrained shortest path problem as an example.
- Conceptually, one way we could compute $L(\mu)$ would be to enumerate all the paths and then taken the one that gave the smallest value.
- For a fixed path, the cost is linear in $\mu$.
- Therefore, $L(\mu)$ as a function is the minimum of a finite number of linear functions.
- This means it is piecewise linear and concave.
- Thus, we need to maximize a concave function.
- This can be done with subgradient optimization.
Multicommodity Flow Problem

• Let $G = (N, A)$ be a directed network with a cost $c_{ij}$ and a capacity $u_{ij}$ associated with every arc $(i, j) \in A$.

• Associated with each node $i \in N$ and each commodity $k \in K$ is a number $b(i, k)$, which is the supply or demand of commodity $k$ at node $i$.

• We let $x_{ij}^k$ denote the amount of flow of commodity $k$ sent on arc $(i, j)$.

• The objective of the multicommodity flow problem is to determine a least cost way to move all commodities through the network in order to satisfy demands at each node subject to shared capacity constraints.
Optimality Conditions

**Theorem 8.** A feasible solution \( y^k_{ij} \) is an optimal solution for the multicommodity flow problem with \( u^k_{ij} = \infty \) if and only if for some set of node potentials \( \pi^k, k \in K \) and some set of arc prices \( w_{ij}, (i, j) \in A \) the reduced costs and the flow values satisfy, for every \( (i, j) \in A \) satisfy

(a) \( w_{ij}(\sum_{k \in K} y^k_{ij} - u_{ij}) = 0 \) for all \( (i, j) \in A \).

(b) If \( c_{ij}^{\pi, k} \geq 0 \) for all \( (i, j) \in A, k \in K \)

(c) \( c_{ij}^{\pi, k} y^k_{ij} = 0 \) for all \( (i, j) \in A, k \in K \)
Deomposition by Commodity

• The role of the arch prices is to assign a cost to flow that is independent of commodity.

• Suppose we have optimal arc prices and solve a minimum cost flow problem with each commodity separately
  – No arc capacities
  – Cost of each arc adjusted for all commodities by $w_{ij}$.

• It turns out that the flows that are optimal for the complete multicommodity flow problem will also be optimal for the decomposed flow problems with the adjusted costs!

• This is easy to show.
Lagrangian Relaxation

- We can calculate optimal arc costs using Lagrangian Relaxation and subgradient optimization.
- The constraints being relaxed are the joint capacity constraints.
- The subgradient update formula becomes

\[ w_{ij}^{q+1} = [w_{ij}^q + \theta_q \left( \sum_{k \in K} y_{ij}^k - u_{ij} \right)]^+ \]
Path Formulation

• Another way to approach solution of multi-commodity flow problems is to use a *path formulation*.

• The concept is to have a variable for each possible path that each commodity can take.

• Let \( P^k \) be the collection of all paths from \( s^k \) to \( t^k \) for commodity \( k \).

• With each path \( P \) in the union of all paths for all commodities, we have a decision variable \( f(P) \) that denotes the flow on path \( P \).

• We then get an (apparently) very simple reformulation of the problem in terms of these new variables.
Path Formulation

Minimize \[ z(x) = \sum_{k \in K} \sum_{P \in \mathcal{P}^k} c^k(P) f(P) \] (7)

subject to \[ \sum_{(i,j) \in P} \delta_{ij}(P) f(P) \leq u_{ij} \quad \forall (i,j) \in A \] (8)

\[ \sum_{P \in \mathcal{P}^k} f(P) = d^k \quad \forall k \in K \] (9)

\[ f(P) \geq 0 \quad \forall k \in K, P \in \mathcal{P}^k \] (10)

where \( \delta_{ij}(P) \) takes value one if arch \((i, j)\) is in path \(P\) and 0 otherwise.
Path Flow Optimality Conditions

**Theorem 9.** The commodity path flows \( f(P) \) are optimal for the path flow formulation of the multicommodity flow problem with \( u^k_{i,j} = \infty \) if and only if for some set of commodity prices \( \sigma^k, k \in K \) and some set of arc prices \( w_{i,j}, (i,j) \in A \) the reduced costs and the flow values satisfy

(a) \( w_{i,j}(\sum_{k \in K, P \in \mathcal{P}^k} \delta_{i,j}(P)f(P) - u_{i,j}) = 0 \) for all \( (i,j) \in A \).

(b) If \( c^\sigma,^k_P \geq 0 \) for all \( P \in \mathcal{P}^k, k \in K \)

(b) If \( c^\sigma,^k_P f(P) = 0 \) for all \( P \in \mathcal{P}^k, k \in K \).
Generating Paths

• For a fixed set of prices, the reduced cost of a path is simply

\[ c_{P}^{\sigma,w} = \sum_{(i,j) \in P} (c_{ij}^{k} + w_{ij}) - \sigma_{k} \]

• This is the cost of the path with respect to “reduced arc prices” and adjusted by the commodity price.

• We can therefore easily determine the path with the smallest reduced cost for each commodity by solving a shortest path problem.

• This leads to a decomposition algorithm very similar to the previous one in which we adjust arc and commodity prices.