Application: Maximum Cardinality Bipartite Matching

A matching $M$ is a set of edges $(u, v)$ such that no two edges share a common vertex. Given a bipartite graph $G = (L, R)$, find a matching in $G$ of maximum cardinality.

Perfect matching: a matching of size $n$, where $n=\#\text{vertices in } L=\#\text{vertices in } R$

Can be solved using our maxflow algorithm (we will see how)

“Bipartite matching reduces to maximum flow”

i.e., if we have an efficient algorithm for maximum flow then we can use it to solve bipartite matching efficiently.
Application: Maximum Cardinality Bipartite Matching

Given a bipartite graph $G = (L, R)$, find a matching in $G$ of maximum cardinality.

Recall: A matching $M$ is a set of edges $(u, v)$ such that no two edges share a common vertex.

Reduction to Max-Flow:

Graph $G$:

- Arthur
- Bill
- Charles
- David
- Angela
- Beth
- Connie
- Doris

Graph $H$:

- Source $s$: connected to all nodes in $L$
- Sink $t$: connected to all nodes in $R$
- Each edge has unit capacity

Perfect matching: a matching of size $n$, where $n = \# \text{vertices in } L = \# \text{vertices in } R$

Size of max flow in $H = \text{cardinality of maximum matching in } G$
Bipartite Matching

**Property:** Cardinality of max matching in $G$ = Size of max flow in $H$
Bipartite Matching

Graph G:

Arthur — Angela
Bill — Beth
Charles — Connie
David — Doris

Graph H:

Property: Cardinality of max matching in G = Size of max flow in H

1. Suppose G has a matching M. Then, consider the flow f in H:
   \[ f(e) = 1, \text{ e in M, } \]
   \[ f(e) = 1, \text{ e = (s, u), u in M, } \]
   \[ f(e) = 1, \text{ e = (v, t), v in M } \]
   \[ f(e) = 0, \text{ ow } \]

   Thus, size(f) = cardinality(M)
**Bipartite Matching**

**Property:** Cardinality of max matching in $G$ = Size of max flow in $H$

1. Suppose $G$ has a matching $M$. Then, consider the flow $f$ in $H$:
   
   $f(e) = 1$, $e$ in $M$, $f(e) = 1$, $e = (s, u)$, $u$ in $M$, $f(e) = 1$, $e = (v, t)$, $v$ in $M$  
   
   $f(e) = 0$, ow

   Thus, $\text{size}(f) = \text{cardinality}(M)$

2. Suppose $H$ has a max flow of size $s$. Then $H$ has an integral max flow $f$ of size $s$.

   Let $Y = (u, v)$ be the edges s.t. $u$ in $L$, $v$ in $R$, and $f(e) = 1$. Then, no $u$ or $v$ can be adjacent to $>1$ nodes in $Y$ (otherwise, capacity constraints violated). Thus $Y$ is a matching

   $\text{Cardinality}(Y) = \text{Flow through cut (L,R)} = \text{size}(f)$
Bipartite Matching

Graph G:

Arthur ——— Angela
——— Bill ——— Beth
——— Charles ——— Connie
David ——— Doris

L ——— R

Graph H:

Algorithm:

Property: Cardinality of max matching in G = Size of max flow in H
**Bipartite Matching**

**Property:** Cardinality of max matching in $G = \text{Size of max flow in } H$

**Algorithm:** To find a maximum cardinality matching, construct $H$, and use Ford-Fulkerson
Bipartite Matching

Graph G:

Arthur
Bill
Charles
David

Angela
Beth
Connie
Doris

Graph H:

Property: Cardinality of max matching in G = Size of max flow in H

Algorithm: To find a maximum cardinality matching, construct H, and use Ford-Fulkerson

Running Time: \(O(mn)\)

- Each FF iteration increases flow value by at least 1
- Max flow size at most n
Practical Tips for Algorithm Design

1. Problem abstraction makes a huge difference
   May make harder problems easier

2. Most interesting problems are NP Hard!
   a. Your input size may be small, so it may not matter
   b. Often easier if input has special properties.
   Examples: Independent set in a tree, traveling salesman on the plane, k-center in a metric space
   c. Approximation algorithms may be a good starting point, but you may need heuristics on top of it
Practical Tips for Algorithm Design

3. Which algorithm is fastest depends on your problem
   a. Different algorithms may be faster, depending on the parameters
   b. How fast an algorithm runs may depend on the architecture of your machine
   c. Your performance metric may not always be speed! (may be memory, amount of network communication, etc)

4. Proofs are an useful tool in getting insight
Algorithm Design Paradigms

• Exhaustive Search

• **Greedy Algorithms**: Build a solution incrementally piece by piece

• **Divide and Conquer**: Divide into parts, solve each part, combine results

• **Dynamic Programming**: Divide into subtasks, perform subtask by size. Combine smaller subtasks to larger ones

• **Hill-climbing**: Start with a solution, improve it

• **Randomized Algorithms**: Algorithm can make random choices