CPS296.2 Geometric Optimization **March 27, 2007**

Lecture 20: Euclidean Minimum-Weight Matching

Lecturer: Pankaj K. Agarwal *Scribe: Albert Yu*

Matching is a well-studied problem in graph theory. Given a graph $G = (V, E)$, a **matching** is a subset of edges so that each vertex appears at most once. If every vertex appears exactly once, it is called a **perfect matching**. The cost of a matching is the sum of the weights of its edges.

The minimum weight matching is a perfect matching of the minimum cost that still assigns as many nodes to pairs as possible. In today's lecture we consider minimum weight matching in a graph induced by a set of points.

20.1 Euclidean Minimum Weight Matching

Given a set $P = \{p_1, p_2, \cdots, p_n\}$ of n points in R^d and $Q = \{q_1, q_2, \cdots, q_n\}$, another set of n points in R^d, let $G = (P \cup Q, P \times Q)$: The weight of an edge (p, q) in G is $\|p - q\|$. The Euclidean minimum weight matching is a minimum weight matching in G.

Theorem 1 *Given a bipartite graph $G = (V, E)$, where $|V| = n$ and $|E| = m$, the minimum weight matching in G can be computed in $O(mn \log(n))$ time.*

Theorem 2 *An Euclidean minimum weight matching can be computed in $O(n^3 \log(n))$ time.*

A minimum weight matching can be formulated as an integer LP problem:

Minimize $\sum_{(i,j)} w_{ij} x_{ij}$

subject to:

$\sum_{j=1}^{n} x_{ij} = 1, \forall i \leq n$ (α_i)

$\sum_{j=1}^{n} x_{ij} = 1, \forall j \leq n$ (β_j)

$x_{ij} \in \{0, 1\}$

20.1.1 Matching With Linear Programming

The integer programming representation in the previous section can be converted into a linear programming relaxation by changing $x_{ij} \in \{0, 1\}$ to $0 \leq x_{ij} \leq 1$. If the input graph is bipartite then the linear program

will always return integer-valued solutions and so any LP algorithm will give an optimal solution. This is not always the case for non-bipartite graphs. In either case, an algorithm can be developed by considering at the matching problem's linear program and its dual, as in the example below.

Primal LP				Dual LP	
minimize:	$\sum_{(i,j)} w_{ij} \cdot x_{ij}$			maximize:	$\sum_{i=1}^{n} \alpha_i + \sum_{i=1}^{n} \beta_j$
such that:				such that:	
	$\sum_{j=1}^{n} x_{ij} = 1$	$\forall i \leq n$	(α_i)		$\forall i, j \;\; \alpha_i + \beta_j \leq w_{ij}$
	$\sum_{j=1}^{n} x_{ij} = 1$	$\forall i \leq n$	(β_j)		
	$0 \leq x_{ij} \leq 1$				

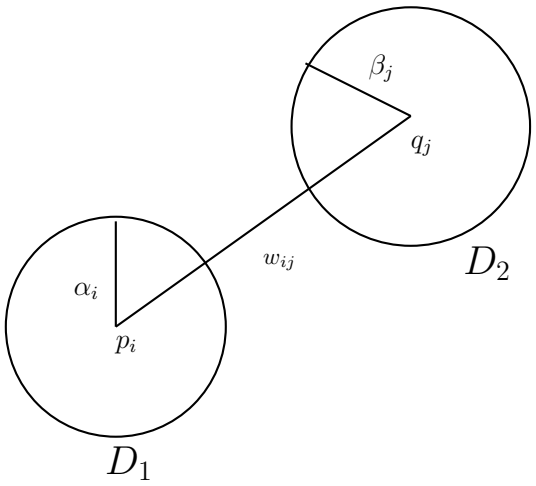

Figure 20.1: $int(D_1) \cap int(D_2) = \emptyset$

The dual variables α and β can be interpreted as the radii of discs centered at p_i and q_j, respectively. The interiors of the two discs do not intersect because $\alpha_i + \beta_j \leq w_{ij}$. LP duality allowes us to verify the optimality of the solution.

20.1.2 Hungarian method

Before we describe the iterative algorithm, we first have to introduce some definitions. Let X be the current matching. An *exposed vertex* is defined as the vertex that is not incident upon any edge of X. An *alternating path* is defined as the path that alternates between an edge of X and an edge not in X. An *augmenting path* is defined as an alternating path between two exposed vertices. Let Π be an augmenting path. $X \oplus \Pi$ is a matching with $|X| + 1$ edges. An *alternating tree* is a rooted tree with path to every node being an alternating path. *Admissible edges* are edges (p_i, q_j) such that $\alpha_i + \beta_j = w_{ij}$.

The algorithm consists of n phases, each of which increases the size of matching by one.

Algorithm Ideas:

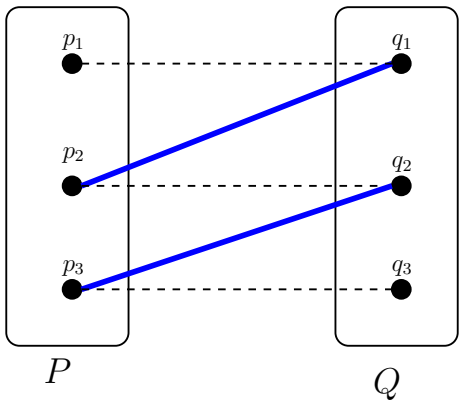

Figure 20.2: In this graph, p_1 and q_3 are the exposed vertices. X includes edges $\overline{p_2q_1}$ and $\overline{p_3q_2}$. An augmenting path π is $\overline{p_1q_1}$, $\overline{q_1p_2}$, $\overline{p_2q_2}$, $\overline{q_2p_3}$, and $\overline{p_3q_3}$.

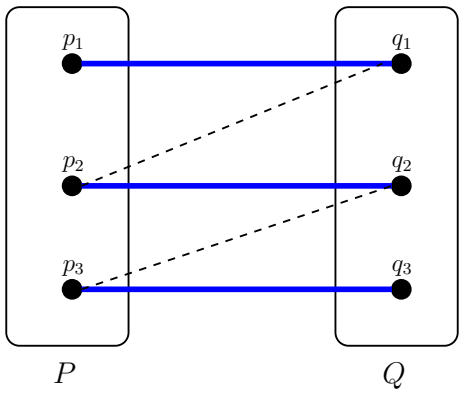

Figure 20.3: The matching $X \oplus \Pi$ includes edges $\overline{p_1q_1}$, $\overline{p_2q_2}$, and $\overline{p_3, q_3}$. The number of edges in $X \oplus \Pi$ is 3 which is 1 more than that of the original matching X.

In each phase, we want to find an augmenting path among the admissible edges. If no augmenting path can be found, we adjust the dual variables such that admissible edges remain admissible and one more edge becomes admissible. When an augmenting path is found, X is updated to $X \oplus \Pi$, thereby increasing the size of matching by one, and the algorithm moves to the next phase.

The matching keeps growing as more augmenting paths are found. When a perfect matching is reached, there is no augmenting path to find because no exposed vertices are left.

Algorithm Description:

$S \subseteq Q$: set of vertices of Q in alternating trees.

$T \subseteq P$: set of vertices of P in alternating trees.

Let E be a set of exposed vertices in Q. Initially we set $S = E$ and $T = \emptyset$. Let

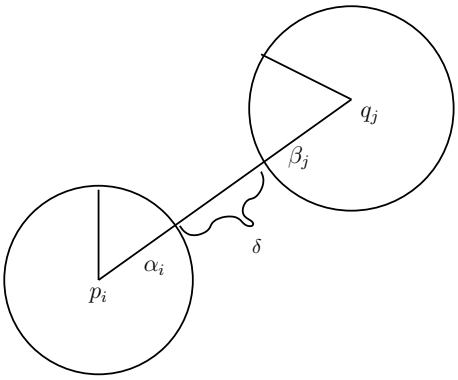

Figure 20.4: We want to minimize δ

$$\delta = \min_{\substack{p_i \in P \setminus T \\ q_j \in S}} w_{ij} - \alpha_i - \beta_j.$$

There are two cases.

Case A: δ = 0

Since δ = 0, there exist an admissible edge (p_i, q_j) where $p_i \in P \setminus T$ and $q_j \in S$ by definition. If p_i is exposed, then we've found an augmenting path. Stop and update X. If not, then let $(p_i, q_k) \in X$, add p_i to T and q_k to S. Store a pointer from p_i to q_j.

Case B: $\delta > 0$

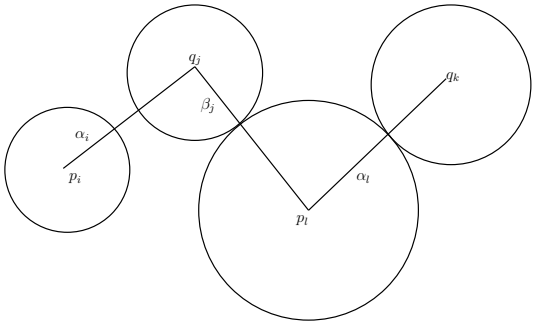

Figure 20.5: case B: $\delta > 0$

If $\delta > 0$, the dual variables are updated as follow:

$\forall q_j \in S, \beta_j = \beta_j + \delta$.

$\forall p_i \in T, \alpha_i = \alpha_i - \delta$.

Note that if there is no augmenting path after adjustment of radii, there is also no augmenting path in the

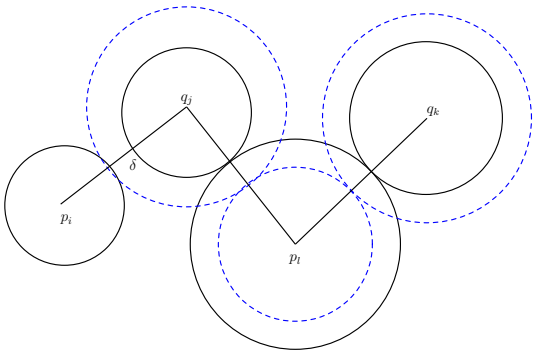

Figure 20.6: α is increased by δ; β is decreased by δ

original graph.

The algorithm repeats the above steps until an augmenting path is found in case A.

Algorithm running time:

Since each phase increases the number of matching by 1, the algorithm has at most n phases. Each phase contains $O(n)$ steps and each step is implemented in $O(n)$ time because β_j and α_i are modified for all $q_j \in S$ and $p_i \in T$ at each occurrence of case B. Therefore, the total running time is $O(n^3)$.

Better implementation

For the sake of effectiveness, it is not desirable to update β_j and α_i at each occurence of case B. Instead, we introduce $\triangle$ which represents the total increase in dual variables so far. More specifically, $\triangle_i = \sum_{j=1}^{i} \delta_i$, where δ_i is the increases in the dual variables at the i^{th} occurrence of case B. We also associate a weight (w) for each point in the graph. The reason of using weights is that they change much less frequently than the dual variables. Let $\widetilde{\alpha_i}$ and $\widetilde{\beta_j}$ be the values of dual variables at the beginning of the phase. Initialize w_i and w_j to $\widetilde{\alpha_i}$ and $\widetilde{\beta_j}$. We modify the two cases as follow:

	Before	After
Case A:		$w_i = \widetilde{\alpha_i} + \triangle$ $w_j = \widetilde{\beta_j} - \triangle$
Case B:	$\forall p_i \in T, \alpha_i = \alpha_i - \delta$ $\forall q_j \in S, \beta_j = \beta_j + \delta$	$\triangle = \triangle + \delta$

The modifications are valid because:

$\forall p_i \in T, \alpha_i = \widetilde{\alpha_i} - \triangle$, which implies $w_i = \widetilde{\alpha_i} = \alpha_i + \triangle$ for all $p_i \in T$.

$\forall q_j \in S, \beta_j = \widetilde{\beta_j} + \triangle$, which implies $w_j = \widetilde{\beta_j} = \beta_j - \triangle$ for all $q_j \in S$.

In summary,

$w_i = \alpha_i + \triangle$ for all $p_i \in T$

$w_i = \alpha_i$ for all $p_i \in P \backslash T$

$w_j = \beta_j - \triangle$ for all $\forall q_j \in S$

$w_j = \beta_j$ for all $\forall q_j \in Q \backslash S$

For all $p_i \in P \backslash T$ and $q_j \in S$,

$$w_{ij} - \alpha_i - \beta_j = w_{ij} - w_i - w_j - \triangle.$$

Therefore,

$$\delta = \min_{\substack{p_i \in P \backslash T \\ q_j \in S}} w_{ij} - w_i - w_j - \triangle.$$

Running time: Instead of modifying all β_j and α_i for all $q_j \in S$ and $p_i \in T$, we adjust the value of $\triangle$ only once at each occurrence of case B. Since the weight of each point is modified only when it is added to either T or S, the number of changes for each weight is 1 at each phase. We can spend $\log n$ time for maintaining priority queue for δ. Thus, each phase can be implemented in $O(n \log(n))$. The total running time is $O(n^2 \log(n))$.