St. Francis Xavier University
Department of Computer Science

CSCI 550: Approximation Algorithms
Lecture 8: The Primal-Dual Method

Fall 2021

1 Two Sides, Same Coin

Thus far in the course, we've seen a number of formulations of decision problems as integer programs, linear programs, semidefinite programs, and vector programs. We've also seen methods of solving such programs to obtain approximate solutions to the associated problems, although some of those methods weren't particularly efficient. In this lecture, we will take a look at how to use linear programs in a slightly different way: instead of solving the program directly, we will take *some* solution from a closely related program and use it to find an approximate solution to our original program.

If what was just described in the last paragraph sounds familiar, that's because we were introduced to this method back in our introductory lecture. This approach is known as the *primal-dual method*, and it's named for the two linear programs we use to obtain our solution: the *primal* program, which is the original program for our problem, and the *dual* program, which is (in a very loose sense) akin to the "inverse" of the primal program. We recall the definition of the dual program here.

Definition 1 (Dual of linear program)**.** Given a linear program L, its dual is a linear program where

- every decision variable in L becomes a constraint of the dual;
- every constraint of L becomes a decision variable in the dual; and
- the objective function of L is reversed in the dual (i.e., maximization becomes minimization and vice versa).

Because we construct the dual program directly from the formulation of the primal program, we know that there exists some kind of relationship between the two programs; namely, the number of decision variables in the dual program is equal to the number of constraints in the primal program, and vice versa. This suggests that the decision variables of one program are *complementary* to the constraints of the other program.

Furthermore, if we know which value a decision variable takes on, then we can make a claim about the value of the complementary constraint. If a decision variable in the dual program is greater than zero (also known as *slack*), then the corresponding constraint of the primal program must be an equality (also known as *tight*), and vice versa. This property is known as *complementary slackness.*

Specifically, with the notion of primal complementary slackness—that is, if $x_j > 0$, then $\sum_j y_i = c_j$—we can define the general primal-dual method that we will use in this lecture. The method relies on the fact that, if some current solution is not feasible for the primal program, then there must exist some adjustment that we can make to the dual program that gets us closer to a primal feasible solution.

Algorithm 1: Primal-dual method

```
$y \leftarrow 0$
while there does not exist an integral solution satisfying primal complementary slackness do
    get direction of increase for dual and adjust appropriately
return feasible integral solution $x$ satisfying primal complementary slackness
```

1.1 Hitting Sets

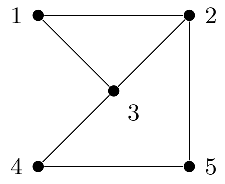
The first problem to which we will apply the primal-dual method is known as the *hitting set* problem. Given a ground set of elements and a collection of subsets, we want to choose a subset of elements from the ground set that both "hits" (or intersects) each subset in the collection and minimizes the total cost of the elements we choose.

<u>Hitting-Set</u>
Given: a ground set of elements $E = \{e_1, \ldots, e_n\}$, subsets $T_1, \ldots, T_p \subseteq E$, and a nonnegative cost $c_e \geq 0$ for each element $e \in E$
Determine: a minimum-cost subset $A \subseteq E$ such that $A \cap T_i \neq \emptyset$ for all $1 \leq i \leq p$

If you think back to our introductory lecture, you may notice that the hitting set problem is quite similar to our familiar set cover problem. In fact, these two problems are equivalent! We can establish the equivalence as follows: observe that each element e_i of the ground set E in an instance of the hitting set problem corresponds to some subset S_i in the set cover problem, while each subset T_j of the hitting set problem instance corresponds to some ground set element e_j in the set cover problem.

We could alternatively relate the two problems in the following way: in the set cover problem, we must find a collection of *subsets* that covers some given set of *elements*, while in the hitting set problem, we must find a set of *elements* that covers some given collection of *subsets*.

Example 2. Consider the following graph:



Furthermore, suppose that we are given a subset of edges $\{(1,2),(1,3),(2,3),(2,5),(3,4)\}$. In this case, a hitting set for this subset contains the vertices $\{2,3,4\}$.

Note that, as a consequence of the connection between the hitting set problem and the set cover problem, and knowing that the set cover problem is NP-complete, we know that the hitting set problem is also NP-complete.

Formulating the hitting set problem as a linear program is straightforward: our objective is clearly to minimize the total cost of the elements we choose, and our constraints are rather natural.

$$\begin{aligned} \text{minimize} \quad & \sum_{e \in E} c_e x_e \\ \text{subject to} \quad & \sum_{e \in T_i} x_e \geq 1, \quad \text{for all } i \in \{1, \ldots, p\} \\ & x_e \geq 0, \quad \text{for all } e \in \{1, \ldots, n\} \end{aligned} \tag{HS-LP}$$

Now, to transform the primal linear program HS-LP to its dual program form, we just need to exchange decision variables and constraints, as well as change our objective from minimization to maximization.

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{p} y_i \\ \text{subject to} \quad & \sum_{i : e \in T_i} y_i \leq c_e, \quad \text{for all } e \in E \\ & y_i \geq 0, \quad \text{for all } i \in \{1, \ldots, p\} \end{aligned} \tag{HS-DualLP}$$

With this dual program formulation, we can apply the primal-dual method. We begin with a solution to the dual program: $y = 0$. Clearly, this solution is feasible for the dual program, since $c_e \geq 0$ for all e. However, the corresponding primal program solution is not feasible, since $A = \emptyset$ at this stage. If the primal program solution is not feasible, then that means A doesn't hit every subset, so there must exist at least one subset T_k where $A \cap T_k = \emptyset$.

This observation implies that, for all elements $e \in T_k$, the corresponding dual constraint has the property $\sum_{i:e\in T_i} y_i < c_e$; that is, the constraint is slack. Since y_k is found only in these constraints, we can increase y_k until the constraint becomes tight, and at the same time we can add a new element e to A corresponding to the cost c_e that y_k has increased by. This new element e will then hit the subset T_k, and we can check once again if the solution is feasible.

This brings us to our primal-dual algorithm for the hitting set problem.

Algorithm 2: Hitting set—primal-dual

```
$y \leftarrow 0$
$A \leftarrow \emptyset$
while $A$ is not feasible do
    find a subset $T_k$ such that $A \cap T_k = \emptyset$
    increase $y_k$ until there exists an element $e \in T_k$ such that $\sum_{i:e\in T_i} y_i = c_e$
    $A \leftarrow A \cup \{e\}$
```

Let's now establish the performance guarantee of our algorithm. Suppose that α denotes the size of the largest subset T_i; that is, for all $1 \leq i \leq p$, $|T_i| \leq \alpha$.

Theorem 3. *Algorithm 2 gives an α-approximation algorithm for the hitting set problem.*

Proof. By the way Algorithm 2 constructs the set A, we have that

$$\begin{aligned}\sum_{e\in A} c_e &= \sum_{e\in A}\sum_{i:e\in T_i} y_i \\ &= \sum_{i=1}^{p} |A \cap T_i| y_i,\end{aligned}$$

since each y_i is counted once for each element $e \in A$ that appears in T_i.

Since each subset T_i contains at most α elements, we have that $|A \cap T_i| \leq \alpha$ for all i, and so

$$\begin{aligned}\sum_{e\in A} c_e &\leq \alpha \cdot \sum_{i=1}^{p} y_i \\ &\leq \alpha \cdot \mathrm{OPT},\end{aligned}$$

since the dual solution $\sum_{i=1}^{p} y_i$ is upper-bounded by the primal solution z^*_{LP}, and additionally, $z^*_{\mathrm{LP}} \leq z^*_{\mathrm{IP}} = \mathrm{OPT}$. □

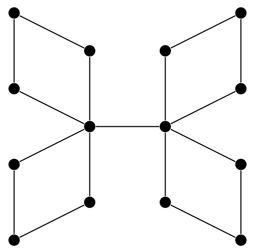
Drawing one final comparison between the hitting set problem and the set cover problem, recall that our primal-dual approximation algorithm for the weighted set cover problem had a performance guarantee of $f = \max_i |\{j \mid e_i \in S_j\}|$, or the maximum frequency of any element e_i. With the hitting set problem, our performance guarantee instead relies on the maximum size of any subset T_i.

1.2 Feedback Vertex Sets

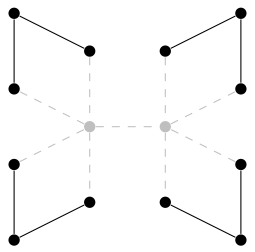
In our next problem, the *feedback vertex set* problem, we consider the problem of finding a subset of vertices in a given graph where, if we remove the vertices in this subset from the graph (and remove the edges adjacent to each removed vertex), then the resultant graph will not contain any cycles.

UNDIRECTED-FEEDBACK-VERTEX-SET
Given: an undirected graph $G = (V, E)$ and an associated weight $w_i \geq 0$ for each vertex $i \in V$
Determine: a subset of vertices $A \subseteq V$ minimizing the sum of weights $\sum_{i \in A} w_i$ such that, for each cycle C in G, $A \cap C \neq \emptyset$

Example 4. Consider the following graph:



The two vertices in the center of the graph, connected by the single middle edge, constitute a feedback vertex set. If we remove these vertices from the graph along with the adjacent edges, then there will be no more cycles within the graph.



If you look at the definition of the feedback vertex set problem carefully, you might notice that it's quite similar to the hitting set problem we studied in the last section. If we take our "ground set" to be the set of vertices V, and we take the "cost" of each element to be the weight w_i, then the "subsets to hit" in the feedback vertex set problem consist exactly of the set of cycles C in the graph.

Since we can formulate the feedback vertex set problem as an instance of the hitting set problem, we know that this problem is NP-complete. We can, however, make a stronger statement: the undirected version of the feedback vertex set problem is APX-complete, so unless P = NP, we cannot find a polynomial-time approximation scheme for this problem.

Like before, we can formulate the feedback vertex set problem as a linear program:

$$\begin{aligned} \text{minimize} \quad & \sum_{i \in V} w_i x_i \\ \text{subject to} \quad & \sum_{i \in C} x_i \geq 1, \quad \text{for all cycles } C \\ & x_i \geq 0, \quad \text{for all } i \in V \end{aligned} \tag{FVS-LP}$$

One problem arises with this formulation, though: we could potentially have a number of cycles in the graph exponential to the number of vertices, which means we would have an exponential number of "subsets to hit". This means that using Algorithm 2 would have a worst-case exponential runtime! Fortunately, we don't necessarily need to consider *all* cycles in the graph; since the primal-dual method starts with a feasible solution to the dual problem and modifies the solution as appropriate, we only need to find *some* cycle that makes the primal solution not feasible, if it exists, and handle that cycle.

Converting the program FVS-LP to its dual form, we get the following:

$$\begin{aligned} \text{maximize} \quad & \sum_{\text{cycles } C} y_C \\ \text{subject to} \quad & \sum_{\substack{i \in C \\ \text{for all cycles } C}} y_C \le w_i, \quad \text{for all } i \in V \\ & y_C \ge 0, \quad \text{for all cycles } C \end{aligned} \tag{FVS-DualLP}$$

We can now apply a variation of Algorithm 2 to this dual program and get an α-approximation algorithm for the feedback vertex set problem. The biggest modification we need to make to the algorithm is, whenever we add a vertex v to our subset S, we remove v from G and subsequently remove any vertices of degree 1 from G. However, simply applying the algorithm as is may not result in the best approximate solution; if we choose some arbitrary cycle C in our graph and increase the corresponding dual decision variable y_C, the size of the intersection $S \cap C$ could become very large and our subset of vertices S could contain far more vertices than we need to "hit" all cycles of the graph.

To refine our algorithm, and to ensure that our subset S doesn't grow too large or include unnecessary vertices, we can make a couple of observations. First, we will reduce our given graph G to an equivalent graph G' where (i) G' doesn't contain any vertices of degree 1, and (ii) every vertex of degree 2 in G' is adjacent to a vertex of higher degree. To see how we can perform this reduction, consider two vertices of degree 2 i and j in G, where, without loss of generality, we have $w_i \le w_j$. Any cycle that visits vertex i must also visit vertex j, and because vertex i has a weight less than or equal to the weight of vertex j, we can always choose vertex i to "hit" this cycle. Thus, we can "shortcut" vertex j out of G by adding an edge to skip over the vertex.

Given a graph satisfying the previous two conditions, we can make a claim about the length of certain cycles in the graph.

Lemma 5 (Erdős–Pósa property). *In any nonempty graph G' with n vertices in which (i) there are no vertices of degree 1 and (ii) for each vertex of degree 2, every adjacent vertex has higher degree, there exists a cycle in G' with length no greater than $4\log_2(n)$.*

Proof. Perform a breadth-first search through G'. If we do not revisit a previously-visited vertex (and therefore form a cycle), then at every other level we will have increased the number of visited vertices by a factor of two. At iteration i of the search, we will have visited $2^{i/2}$ vertices, Thus, we must find a cycle by iteration $2\log_2(n)$, which corresponds to traversing both "halves" of the cycle and revisiting a previously-visited vertex at the "halfway point" of the cycle. ☐

Therefore, reducing our graph in this way will allow us to refine our algorithm to choose only "unhit" cycles of length at most $4\log_2(n)$, and we therefore get a $4\log_2(n)$-approximation algorithm.

While this modification ensures our subset S doesn't grow too large, it still doesn't address the other underlying problem with our algorithm: at each iteration, the algorithm adds some vertex to its subset to make the solution feasible, but by the end of the algorithm's execution, there's no guarantee that *every* vertex in the subset is truly necessary to maintain feasibility. Leaving these unnecessary vertices in the subset increases the cost of the subset, which in turn reduces the accuracy of our approximation.

Fortunately, we can make a small and easy tweak to our algorithm to remove unnecessary elements from our subset.

Algorithm 3: Feedback vertex set—primal-dual

```
$y \leftarrow 0$
$A \leftarrow \emptyset$
$\ell \leftarrow 0$
repeatedly remove vertices of degree 1 from $G$
while $A$ is not feasible do
    $\ell \leftarrow \ell + 1$
    find a cycle $C$ such that $A \cap C = \emptyset$
    increase $y_k$ until there exists a vertex $v_\ell \in C$ such that $\sum_{i: v_\ell \in C} y_i = w_{v_\ell}$
    $A \leftarrow A \cup \{v_\ell\}$
    remove $v_\ell$ from $G$ and repeatedly remove any new vertices of degree 1 from $G$
for $j$ from $\ell$ down to 1 do
    if $A \setminus \{v_j\}$ remains a feasible solution then
        $A \leftarrow A \setminus \{v_j\}$
```

If A is the solution produced by Algorithm 3 after a total of ℓ iterations, then on iteration j, the algorithm finds some "unhit" cycle C_j where $A \cap C_j = \emptyset$ and adjusts the dual decision variable until some vertex v_j is added to A. By this construction, we know that $C_j \cap \{v_1, \ldots, v_{j-1}\} = \emptyset$.

Before we establish the performance guarantee of Algorithm 3, we require one notion. Given a set E, say that $Z \subseteq E$ is a *minimal augmentation* of $X \subseteq E$ if

1. $X \cup Z$ is a feasible solution; and
2. for any $e \in Z$, $X \cup Z \setminus \{e\}$ is not a feasible solution.

We can then claim that $A \setminus \{v_1, \ldots, v_{j-1}\}$ is a minimal augmentation of $\{v_1, \ldots, v_{j-1}\}$. Clearly, the union of these two subsets is feasible, so the first condition is satisfied. Additionally, since $A \setminus \{v_1, \ldots, v_{j-1}\} \subseteq \{v_j, \ldots, v_\ell\}$, for any $v_t \in \{v_j, \ldots, v_\ell\}$ where $v_t \in A$, since v_t was not removed by the for loop of Algorithm 3, $A \setminus \{v_t\}$ is not feasible and so the second condition is also satisfied.

It follows from this observation that $|A \cap C_j| \leq \max |B \cap C_j|$, where the maximum is taken over all subsets B such that B is a minimum augmentation of $\{v_1, \ldots, v_{j-1}\}$. If we denote by β the maximum number of elements of any "unhit" cycle that could possibly be added by the algorithm under a minimal augmentation, then we get the following result:

Theorem 6. *Algorithm 3 gives a β-approximation algorithm for the feedback vertex set problem.*

Proof. Since $|A \cap C_j| \leq \max |B \cap C_j| \leq \beta$, we have that

$$\begin{aligned}\sum_{i \in A} w_i &= \sum_i |A \cap C_i| \cdot y_i \\ &\leq \beta \cdot \sum_i y_i \\ &\leq \beta \cdot \text{OPT}.\end{aligned}$$

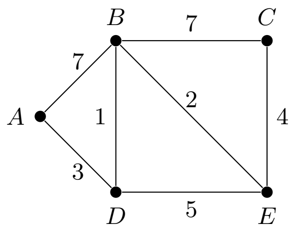
□

1.3 Shortest s-t Paths

Finally, we will consider the problem of finding the shortest path between two specified vertices s and t in an undirected graph G. Each edge of the graph has an associated cost, so by "shortest path", we mean the path of least cost between the two vertices.

Shortest-S-T-Path
Given: an undirected graph $G = (V, E)$, two vertices $s, t \in V$, and an associated cost $c_i \geq 0$ for each edge $e \in E$
Determine: a minimum-cost path from vertex s to vertex t

Example 7. Consider the following graph:



Suppose that we take $s = A$ and $t = C$. While a number of paths exist between s and t, the shortest path has a total cost of $3 + 1 + 2 + 4 = 10$.

Much like we did with the feedback vertex set problem, we can model an instance of the shortest s-t path problem as an instance of the hitting set problem. We take as our "ground set" the set of edges E, and we take the "cost" of each element simply to be the cost of each edge. Then, the "subsets to hit" are the edges that make up an s-t cut; that is, the edges that, if removed, would place vertices s and t in disjoint subsets. We can formalize this notion by taking $T_i = \delta(S_i)$, where $s \in S_i$, $t \notin S_i$, and $\delta(S) = \{(u, v) \in E \mid u \in S, v \notin S\}$.

Lemma 8. *A set of edges E contains an s-t path if and only if E hits every s-t cut.*

Proof. ($\Rightarrow$): Suppose to the contrary that E does not contain an s-t path. Let S_i denote the largest connected component containing vertex s. By our assumption, vertex t is not in S_i. Moreover, E cannot contain any edge from $\delta(S_i)$, or else we could include the other vertex incident to that edge and obtain a larger connected component containing s. This implies that some s-t cut is not hit by E.

($\Leftarrow$): Suppose to the contrary that E does not hit some s-t cut S_i. In this case, E must only contain edges that either join two vertices of S_i or join two vertices of the complement of S_i. Then, any path starting from vertex $s \in S_i$ consisting of edges in E can only bring us to other vertices in S_i, but vertex t is not in S_i. Therefore, E does not contain an s-t path. □

Interestingly, you might remember from a previous class on algorithm analysis that shortest path problems can be solved in polynomial time using Dijkstra's algorithm, the Bellman–Ford algorithm, or any number of other shortest path algorithms. So, why then are we considering this problem, when all of the other problems we've studied in this course are much more difficult? As it turns out, we can draw an interesting parallel between our primal-dual algorithm and Dijkstra's algorithm: they behave in exactly the same way! We can also take the findings we obtain from studying this problem and apply them to more general, harder problems, such as the generalized Steiner tree problem.

Having established this connection between the shortest s-t path problem and the hitting set problem, we can use a modified version of Algorithm 3 to obtain a solution to an instance of the shortest s-t path problem.

The idea behind the modification is that, whenever the subset A constructed by the algorithm is not feasible, the algorithm chooses $\delta(S_k)$ as the next subset to "hit", where S_k denotes the connected component of the subgraph (V, A) containing vertex s. Since the algorithm chose $\delta(S_k)$, we know that $A \cap S_k = \emptyset$, and much of the analysis of the previous section can be applied here.

From our earlier observation that our primal-dual algorithm essentially behaves in the same way as Dijkstra's algorithm, which is known to produce a shortest path between two vertices, we can deduce that our algorithm will give us an optimal answer to any given instance of the problem. This, in turn, means that we obtain a fabled 1-approximation algorithm for the problem!

Theorem 9. *Modifying Algorithm 3 gives a 1-approximation algorithm for the shortest s-t path problem.*

Proof. The goal of the proof is simply to show that $\beta = 1$, where β denotes the performance guarantee of Algorithm 3 established in Theorem 6.

Let A be a solution produced by the algorithm that is not feasible, and let B be a minimal augmentation of A. Suppose that $(s, v_1, \ldots, v_\ell, t)$ is an s-t path in the subgraph $(V, A \cup B)$. Choose the maximal value i such that $v_i \in S_k$ and $v_{i+1} \notin S_k$, where S_k is the connected component containing vertex s.

Since S_k is a connected component, there exists an s-v_i path within S_k, $(s, w_1, \ldots, w_j, v_i)$, where $w_m \in S_k$ for all $1 \leq m \leq j$. Therefore, combining the two paths to obtain $(s, w_1, \ldots, w_j, v_i, v_{i+1}, \ldots, v_\ell, t)$ produces an s-t path. We can then take $B' = \{(v_i, v_{i+1}), \ldots, (v_\ell, t)\}$ to be an augmentation.

Since all edges in B' are adjacent to at least one vertex not in S_k, then none of these edges appear in A, which implies that each edge comes from B; that is, $B' \subseteq B$. Moreover, by our assumption that B was a minimal augmentation, we have that $B' = B$. Therefore,

$$\begin{aligned} |B \cap \delta(S_k)| &= |B' \cap \delta(S_k)| \\ &= |\{v_i, v_{i+1}\}|, \end{aligned}$$

since the first edge in B' is the only one to share a vertex with S_k. Since $|\{v_i, v_{i+1}\}| = 1$, we get that $\beta = 1$ as desired. $\square$

Following a similar approach, we can obtain a 2-approximation algorithm for the generalized Steiner tree problem and many other problems taking graphs as input.