Chapter 5

The Basis

> All your bases are belong to us.
>
> *Zero Wing*, Sega Mega Drive version, 1991, misquoted

5.1 Coordinate systems

5.1.1 René Descartes' idea

In 1618, the French mathematician René Descartes had an idea that forever transformed the way mathematicians viewed geometry.

In deference to his father's wishes, he studied law in college. But something snapped during this time:

> I entirely abandoned the study of letters. Resolving to seek no knowledge other than that of which could be found in myself or else in the great book of the world, I spent the rest of my youth traveling, visiting courts and armies, mixing with people of diverse temperaments and ranks, gathering various experiences, testing myself in the situations which fortune offered me, and at all times reflecting upon whatever came my way so as to derive some profit from it.

After tiring of the Paris social scene, he joined the army of Prince Maurice of Nassau one year, then joined the opposing army of the duke of Bavaria the next year, although he never saw combat.

He had a practice of lying in bed in the morning, thinking about mathematics. He found the prevailing approach to geometry—the approach taken since the ancient Greeks—needlessly cumbersome.

His great idea about geometry came to him, according to one story, while lying in bed and watching a fly on the ceiling of his room, near a corner of the room. Descartes realized that the location of the fly could be described in terms of two numbers: its distance from the two walls it was near. Significantly, Descartes realized that this was true even if the two walls were not perpendicular. He further realized that geometrical analysis could thereby be reduced to algebra.

5.1.2 Coordinate representation

The two numbers characterizing the fly's location are what we now call *coordinates*. In vector analysis, a *coordinate system* for a vector space $\mathcal{V}$ is specified by generators $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$ of $\mathcal{V}$. Every vector $\boldsymbol{v}$ in $\mathcal{V}$ can be written as a linear combination

$$\boldsymbol{v} = \alpha_1\, \boldsymbol{a}_1 + \cdots + \alpha_n\, \boldsymbol{a}_n$$

We can therefore represent $\boldsymbol{v}$ by the vector $[\alpha_1, \ldots, \alpha_n]$ of coefficients. In this context, the coefficients are called *coordinates*, and the vector $[\alpha_1, \ldots, \alpha_n]$ is called the *coordinate representation* of $\boldsymbol{v}$ in terms of $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$.

But assigning coordinates to points is not enough. In order to avoid confusion, we must ensure that each point is assigned coordinates in exactly one way. To ensure this, we must use care in selecting the generators $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$. We address *existence and uniqueness of representation* in Section 5.7.1.

Example 5.1.1: The vector $\boldsymbol{v} = [1, 3, 5, 3]$ is equal to $1\,[1, 1, 0, 0] + 2\,[0, 1, 1, 0] + 3\,[0, 0, 1, 1]$, so the coordinate representation of $\boldsymbol{v}$ in terms of the vectors $[1, 1, 0, 0], [0, 1, 1, 0], [0, 0, 1, 1]$ is $[1, 2, 3]$.

Example 5.1.2: What is the coordinate representation of the vector $[6, 3, 2, 5]$ in terms of the vectors $[2, 2, 2, 3], [1, 0, -1, 0], [0, 1, 0, 1]$? Since

$$[6, 3, 2, 5] = 2\,[2, 2, 2, 3] + 2\,[1, 0, -1, 0] - 1\,[0, 1, 0, 1],$$

the coordinate representation is $[2, 2, -1]$.

Example 5.1.3: Now we do an example with vectors over $GF(2)$. What is the coordinate representation of the vector [0,0,0,1] in terms of the vectors [1,1,0,1], [0,1,0,1], and [1,1,0,0]? Since

$$[0, 0, 0, 1] = 1\,[1, 1, 0, 1] + 0\,[0, 1, 0, 1] + 1\,[1, 1, 0, 0]$$

the coordinate representation of $[0, 0, 0, 1]$ is $[1, 0, 1]$.

5.1.3 Coordinate representation and matrix-vector multiplication

Why put the coordinates in a vector? This actually makes a lot of sense in view of the linear-combinations definitions of matrix-vector and vector-matrix multiplication. Suppose the coordinate axes are $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$. We form a matrix $A = \left[\begin{array}{c|c|c} & & \\ \boldsymbol{a}_1 & \cdots & \boldsymbol{a}_n \\ & & \end{array}\right]$ whose columns are the generators.

- We can write the statement "$\boldsymbol{u}$ is the coordinate representation of $\boldsymbol{v}$ in $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$" as the matrix-vector equation
$$A\boldsymbol{u} = \boldsymbol{v}$$
- Therefore, to go from a coordinate representation $\boldsymbol{u}$ to the vector being represented, we multiply A times $\boldsymbol{u}$.
- Moreover, to go from a vector $\boldsymbol{v}$ to its coordinate representation, we can solve the matrix-vector equation $A\boldsymbol{x} = \boldsymbol{v}$. Because the columns of A are generators for $\mathcal{V}$ and $\boldsymbol{v}$ belongs to $\mathcal{V}$, the equation must have at least one solution.

We will often use matrix-vector multiplication in the context of coordinate representations.

5.2 First look at lossy compression

In this section, I describe one application of coordinate representation. Suppose we need to store many 2000×1000 grayscale images. Each such image can be represented by a D-vector where $D = \{0, 1, \ldots, 19999\} \times \{0, 1, \ldots, 999\}$. However, we want to store the images more compactly. We consider three strategies.

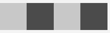
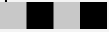
5.2.1 Strategy 1: Replace vector with closest sparse vector

If an image vector has few nonzeroes, it can be stored compactly—but this will happen only rarely. We therefore consider a strategy that replaces an image with a different image, one that is sparse but that we hope will be perceptually similar. Such a compression method is said to be *lossy* since information in the original image is lost.

Consider replacing the vector with the closest k-sparse vector. This strategy raises a Question:

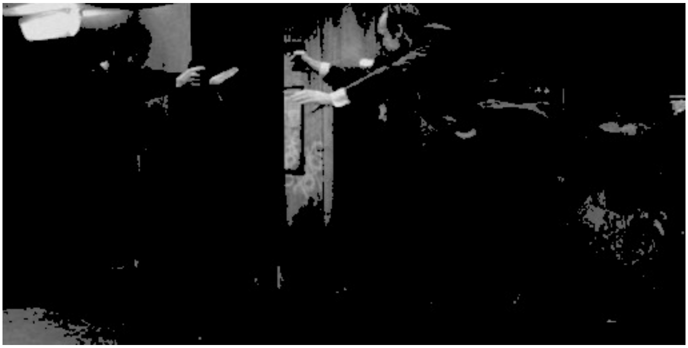
> **Question 5.2.1:** Given a vector $\boldsymbol{v}$ and a positive integer k, what is the k-sparse vector closest to $\boldsymbol{v}$?

We are not yet in a position to say what "closest" means because we have not defined a distance between vectors. The distance between vectors over $\mathbb{R}$ is the subject of Chapter 8, where we will will discover that the closest k-sparse vector is obtained from $\boldsymbol{v}$ by simply replacing all but the k largest-magnitude entries by zeroes. The resulting vector will be k-sparse—and therefore, for, say, $k = 200,000$, can be represented more compactly. But is this a good way to compress an image?

Example 5.2.2: The image  consists of a single row of four pixels, with intensities 200, 75, 200, 75. The image is thus represented by four numbers. The closest 2-sparse image, which has intensities 200, 0, 200, 0, is .

Here is a realistic image:

and here is the result of suppressing all but 10% of the entries:



The result is far from the original image since so many of the pixel intensities have been set to zero. This approach to compression won't work well.

5.2.2 Strategy 2: Represent image vector by its coordinate representation

Here is another strategy, one that will incur no loss of fidelity to the original image.

- Before trying to compress any images, select a collection of vectors $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$.
- Next, for each image vector, find and store its coordinate representation $\boldsymbol{u}$ in terms of $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$.[1]

[1] You could do this by solving a matrix-vector equation, as mentioned in Section 5.1.3.

- To recover the original image from the coordinate representation, compute the corresponding linear combination.[2]

Example 5.2.3: We let $\boldsymbol{a}_1 =$ (an image with one row of pixels with intensities 255, 0, 255, 0) and $\boldsymbol{a}_2 =$ (an image with one row of pixels with intensities 0, 255, 0, 255).

Now suppose we want to represent the image (with intensities 200, 75, 200, 75) in terms of $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$.

$$= \frac{200}{255}\,\boldsymbol{a}_1 + \frac{75}{255}\,\boldsymbol{a}_2$$

Thus this image is represented in compressed form by the coordinate representation $[\frac{200}{255}, \frac{75}{255}]$.

On the other hand, the image (intensities 255, 200, 150, 90) cannot be written as a linear combination of $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$, and so has no coordinate representation in terms of these vectors.

As the previous example suggests, for this strategy to work reliably, we need to ensure that every possible $2,000 \times 1,000$ image vector can be represented as a linear combination of $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$. This comes down to asking whether $\mathbb{R}^D = \text{Span }\{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n\}$.

Formulated in greater generality, this is a Fundamental Question:

Question 5.2.4: For a given vector space $\mathcal{V}$, how can we tell if $\mathcal{V} = \text{Span }\{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n\}$?

Furthermore, the strategy will only be useful in compression if the number n of vectors used in linear combinations is much smaller than the number of pixels. Is it possible to select such vectors? What is the minimum number of vectors whose span equals $\mathbb{R}^D$?

Formulated in greater generality, this is another Fundamental Question:

Question 5.2.5: For a given vector space $\mathcal{V}$, what is the minimum number of vectors whose span equals $\mathcal{V}$?

It will turn out that our second strategy for image compression will fail: the minimum number of vectors required to span the set of all possible $2,000 \times 1,000$ images is not small enough to achieve any compression at all.

Strategy 3: A hybrid approach

The successful strategy will combine both of the previous two strategies: coordinate representation and closest k-sparse vector:

Step 1: Select vectors $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$.

[2] You could do this by matrix-vector multiplication, as mentioned in Section 5.1.3.

Step 2: For each image you want to compress, take the corresponding vector $\boldsymbol{v}$ and find its coordinate representation $\boldsymbol{u}$ in terms of $\boldsymbol{a}_1, \ldots, \boldsymbol{a}_n$.[3]

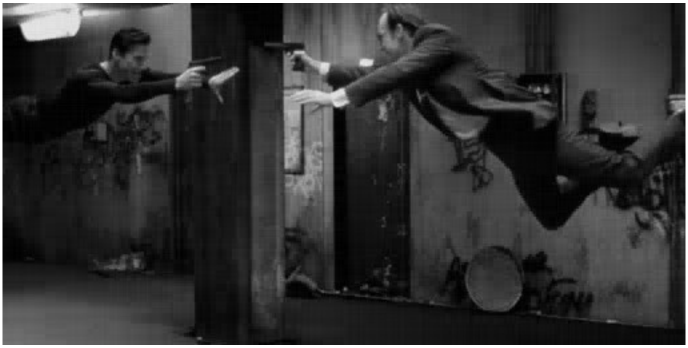
Step 3: Next, replace $\boldsymbol{u}$ with the closest k-sparse vector $\tilde{\boldsymbol{u}}$, and store $\tilde{\boldsymbol{u}}$.

Step 4: To recover an image from $\tilde{\boldsymbol{u}}$, calculate the corresponding linear combination of $\boldsymbol{a}_1, \ldots \boldsymbol{a}_n$.[4]

How well does this method work? It all depends on which vectors we select in Step 1. We need this collection of vectors to have two properties:

- *Step 2 should always succeed.* It should be possible to express any vector $\boldsymbol{v}$ in terms of the vectors in the collection.
- *Step 3 should not distort the image much.* The image whose coordinate representation is $\tilde{\boldsymbol{u}}$ should not differ much from the original image, the image whose coordinate representation is $\boldsymbol{u}$.

How well does this strategy work? Following a well-known approach for selecting the vectors in Step 1 (described in detail in Chapter 10), we get the following nice result using only 10% of the numbers:



5.3 Two greedy algorithms for finding a set of generators

In this section, we consider two algorithms to address Question 5.2.5:

> *For a given vector space $\mathcal{V}$, what is the minimum number of vectors whose span equals $\mathcal{V}$?*

It will turn out that the ideas we discover will eventually help us answer many other questions, including Question 5.2.4.

[3] You could do this by solving a matrix-vector equation, as mentioned in Section 5.1.3.

[4] You could do this by matrix-vector multiplication, as mentioned in Section 5.1.3.

5.3.1 Grow algorithm

How can we obtain a minimum number of vectors? Two natural approaches come to mind, the *Grow* algorithm and the *Shrink* algorithm. Here we present the *Grow* algorithm.

```
def GROW($\mathcal{V}$)
  $B = \emptyset$
  repeat while possible:
        find a vector $\boldsymbol{v}$ in $\mathcal{V}$ that is not in Span $B$, and put it in $B$.
```

The algorithm stops when there is no vector to add, at which time B spans all of $\mathcal{V}$. Thus, if the algorithm stops, it will have found a generating set. The question is: is it bigger than necessary?

Note that this algorithm is not very restrictive: we ordinarily have lots of choices of which vector to add.

Example 5.3.1: We use the Grow algorithm to select a set of generators for $\mathbb{R}^3$. In Section 3.2.3, we defined the *standard* generators for $\mathbb{R}^n$. In the first iteration of the Grow algorithm we add to our set B the vector $[1, 0, 0]$ It should be apparent that $[0, 1, 0]$ is not in Span $\{[1, 0, 0]\}$. In the second iteration, we therefore add this vector to B. Likewise, in the third iteration we add $[0, 0, 1]$ to B. We can see that any vector $\boldsymbol{v} = (\alpha_1, \alpha_2, \alpha_3) \in \mathbb{R}^3$ is in Span $(\boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3)$ since we can form the linear combination

$$\boldsymbol{v} = \alpha_1 \boldsymbol{e}_1 + \alpha_2 \boldsymbol{e}_2 + \alpha_3 \boldsymbol{e}_3$$

Therefore there is no vector $\boldsymbol{v} \in \mathbb{R}^3$ to add to B, and the algorithm stops.

5.3.2 Shrink algorithm

In our continuing effort to find a minimum set of vectors that span a given vector space $\mathcal{V}$, we now present the *Shrink* algorithm.

```
def SHRINK($\mathcal{V}$)
  $B$ = some finite set of vectors that spans $\mathcal{V}$
  repeat while possible:
        find a vector $\boldsymbol{v}$ in $B$ such that Span $(B - \{v\}) = \mathcal{V}$, and remove $\boldsymbol{v}$ from $B$.
```

The algorithm stops when there is no vector whose removal would leave a spanning set. At every point during the algorithm, B spans $\mathcal{V}$, so it spans $\mathcal{V}$ at the end. Thus the algorithm certainly finds a generating set. The question is, again: is it bigger than necessary?

Example 5.3.2: Consider a simple example where B initially consists of the following vectors:

$$\begin{aligned} \boldsymbol{v}_1 &= [1,0,0] \\ \boldsymbol{v}_2 &= [0,1,0] \\ \boldsymbol{v}_3 &= [1,2,0] \\ \boldsymbol{v}_4 &= [3,1,0] \end{aligned}$$

In the first iteration, since $\boldsymbol{v}_4 = 3\boldsymbol{v}_1 + \boldsymbol{v}_2$, we can remove $\boldsymbol{v}_4$ from B in the first iteration without changing Span B. After this iteration, $B = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3\}$. In the second iteration, since $\boldsymbol{v}_3 = \boldsymbol{v}_1 + 2\boldsymbol{v}_2$, we remove $\boldsymbol{v}_3$ from B, resulting in $B = \{\boldsymbol{v}_1, \boldsymbol{v}_2\}$. Finally, note that Span $B = \mathbb{R}^3$ and that neither $\boldsymbol{v}_1$ nor $\boldsymbol{v}_2$ alone could generate $\mathbb{R}^3$. Therefore the Shrink algorithm stops.

Note: These are not algorithms that you can go and implement. They are abstract algorithms, algorithmic thought experiments:

- We don't specify how the input—a vector space—is specified.
- We don't specify how each step is carried out.
- We don't specify which vector to choose in each iteration.

In fact we later exploit the last property—the freedom to choose which vector to add or remove—in our proofs.

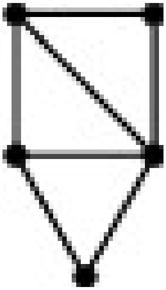
5.3.3 When greed fails

Before analyzing the Grow and Shrink algorithms for finding minimum generating set, I want to look at how similar algorithms perform on a different problem, a problem on graphs.

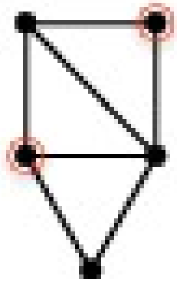
Dominating set A *dominating set* is a set of nodes such that every node in the graph is either in the set or is a neighbor (via a single edge) of some node in the set. The goal of the *minimum-dominating-set problem* is to find a dominating set of minimum size.

I like to think of a dominating set as a set of guards posted at intersections. Each intersection must be guarded by a guard at that intersection or a neighboring intersection.

Consider this graph:



A dominating set is indicated here:



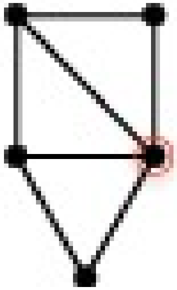
You might consider finding a dominating set using a Grow algorithm:

> *Grow Algorithm for Dominating Set:*
> Initialize B to be empty; while B is not a dominating set, add a node v to B

or a Shrink algorithm:

> *Shrink Algorithm for Dominating Set:*
> Initialize B to contain all nodes; while there is a node v such that $B - \{v\}$ is a dominating set, remove v from B

but either of these algorithms could, by unfortunate choices, end up selecting the dominating set shown above, whereas there is a smaller dominating set:

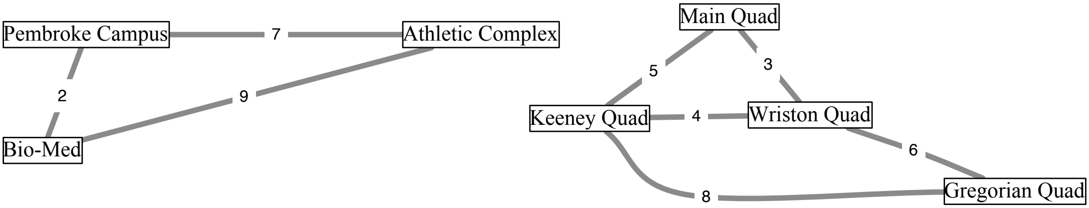


Grow and Shrink algorithms are called *greedy* algorithms because in each step the algorithm makes a choice without giving thought to the future. This example illustrates that greedy algorithms are not reliably good at finding the best solutions.

The Grow and Shrink algorithms for finding a smallest generating set for a vector space are remarkable: as we will see, they do in fact find the smallest solution.

5.4 Minimum Spanning Forest and $GF(2)$

I will illustrate the Grow and Shrink algorithms using a graph problem: *Minimum Spanning Forest*. Imagine you must replace the hot-water delivery network for the Brown University campus. You are given a graph with weights on edges:



where there is a node for each campus area. An edge represents a possible hot-water pipe between different areas, and the edge's weight represents the cost of installing that pipe. Your goal is to select a set of pipes to install so every pair of areas that are connected in the graph are connected by the installed pipes, and to do so at minimum cost.

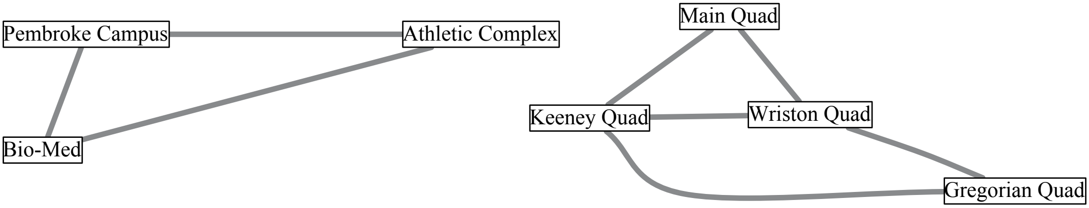
5.4.1 Definitions

Definition 5.4.1: For a graph G, a sequence of edges

$$[\{x_1, x_2\}, \{x_2, x_3\}, \{x_3, x_4\}, \ldots, \{x_{k-1}, x_k\}]$$

is called an x_1-to-x_k *path* (or a *path from x_1 to x_k*).

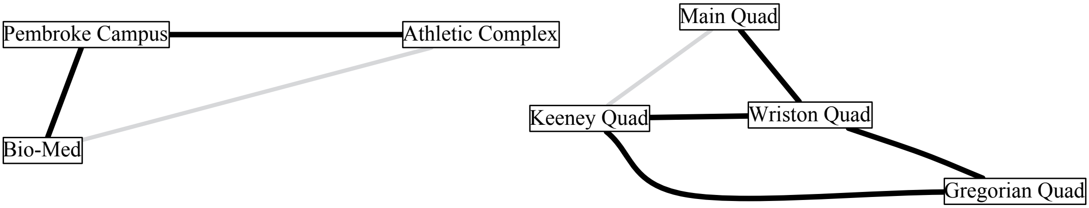
In this graph



there is a path from "Main Quad" to "Gregorian Quad" but no path from "Main Quad" to "Athletic Complex".

Definition 5.4.2: A set S of edges is *spanning* for a graph G if, for every edge $\{x, y\}$ of G, there is an x-to-y path consisting of edges of S.

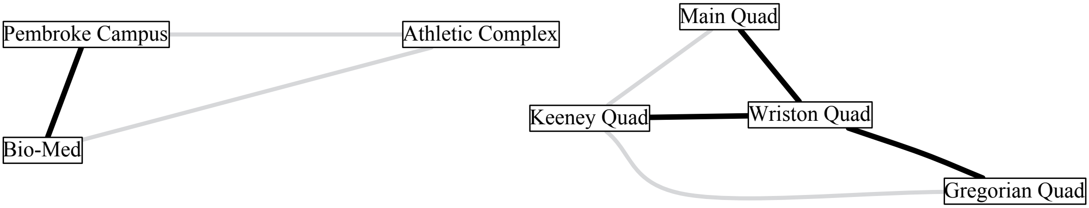
For example, the dark edges in the following diagram are spanning for the graph depicted:



We will soon see a connection between this sense of "spanning" and the sense in which we use the term in linear algebra.

Definition 5.4.3: A *forest* is a set of edges containing no cycles (loops possibly consisting of several edges).

For example, the dark edges in the earlier diagram do *not* form a forest because there are three dark edges that form a cycle. On the other hand, the dark edges in the following diagram *do* form a forest:



A graph-theoretical forest resembles a biological forest, i.e. collection of trees, in that a tree's branches do not diverge and then rejoin to form a cycle.

We will give two algorithms for a computational problem, *Minimum Spanning Forest*,[5] abbreviated MSF.

- *input:* a graph G, and an assignment of real-number *weights* to the edges of G.
- *output:* a minimum-weight set B of edges that is spanning and a forest.

The reason for the term "forest" is that the solution need not contain any cycles (as we will see), so the solution resembles a collection of trees. (A tree's branches do not diverge and then rejoin to form a cycle.)

5.4.2 The Grow algorithm and the Shrink algorithm for *Minimum Spanning Forest*

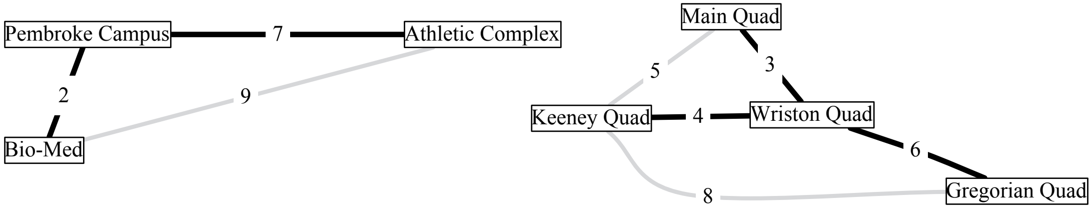
There are many algorithms for *Minimum Spanning Forest* but I will focus on two: a Grow algorithm and a Shrink algorithm. First, the Grow algorithm:

[5] The problem is also called *minimum-weight spanning forest.* The problem *maximum-weight spanning forest* can be solved by the same algorithms by just negating the weights.

```
def Grow($G$)
  $B := \emptyset$
  consider the edges in order, from lowest-weight to highest-weight
  for each edge $e$:
      if $e$'s endpoints are not yet connected via edges in $B$:
         add $e$ to $B$.
```

This algorithm exploits the freedom we have in the Grow algorithm to select which vector to add.

The weights in increasing order are: 2, 3, 4, 5, 6, 7, 8, 9. The solution obtained, which consists of the edges with weights 2, 3, 4, 6, 7, is this:



Here is the Shrink algorithm:

```
def Shrink($G$)
  $B = \{\text{all edges}\}$
  consider the edges in order, from highest-weight to lowest-weight
  for each edge $e$:
      if every pair of nodes are connected via $B - \{e\}$:
         remove $e$ from $B$.
```

This algorithm exploits the freedom in the Shrink algorithm to select which vector to remove. The weights in decreasing order are: 9, 8, 7, 6, 5, 4, 3, 2. The solution consists of the edges with weights 7, 6, 4, 3, and 2.

The Grow algorithm and the Shrink algorithm came up with the same solution, the correct solution.

5.4.3 Formulating *Minimum Spanning Forest* in linear algebra

It is no coincidence that the Grow and Shrink algorithms for minimum spanning forest resemble those for finding a set of generators for a vector space. In this section, we describe how to model a graph using vectors over $GF(2)$.

Let $D = \{\text{Pembroke, Athletic, Bio-Med, MainKeeney, Wriston, Gregorian}\}$ be the set of nodes.

A subset of D is represented by the vector with ones in the corresponding entries and zeroes elsewhere. For example, the subset {`Pembroke, Main, Gregorian`} is represented by the vector whose dictionary is {`Pembroke:one, Main:one, Gregorian:one`}, which we can write as

Pembroke	Athletic	Bio-Med	Main	Keeney	Wriston	Gregorian
1			1			1

Each edge is a two-element subset of D, so it is represented by a vector, namely the vector that has a one at each of the endpoints of e and zeroes elsewhere. For example, the edge connecting Pembroke and Athletic is represented by the vector `{'Pembroke':one, 'Athletic':one}`.

Here are the vectors corresponding to all the edges in our graph:

edge	vector						
	Pem.	Athletic	Bio-Med	Main	Keeney	Wriston	Greg.
{Pem., Athletic}	1	1					
{Pem., Bio-Med}	1		1				
{Athletic, Bio-Med}		1	1				
{Main, Keeney}				1	1		
{Main, Wriston}				1		1	
{Keeney, Wriston}					1	1	
{Keeney, Greg.}					1		1
{Wriston, Greg.}						1	1

The vector representing {`Keeney, Gregorian`},

Pembroke	Athletic	Bio-Med	Main	Keeney	Wriston	Gregorian
				1		1

is the sum, for example, of the vectors representing {`Keeney, Main`}, {`Main, Wriston`}, and {`Wriston, Gregorian`},

Pembroke	Athletic	Bio-Med	Main	Keeney	Wriston	Gregorian
			1	1		
			1		1	
					1	1

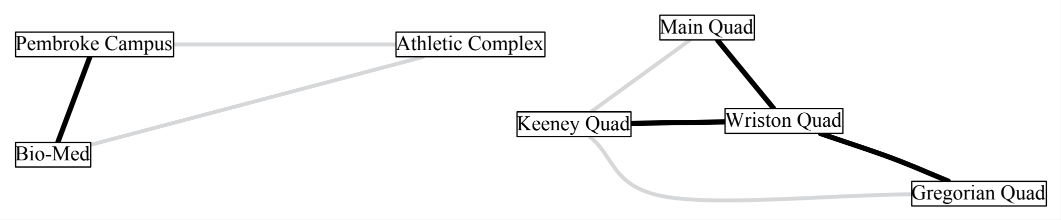
because the 1's in entries `Main` and `Wriston` cancel out, leaving 1's just in entries `Keeney` and `Gregorian`.

In general, a vector with 1's in entries x and y is the sum of vectors corresponding to edges that form an x-to-y path in the graph. Thus, for these vectors, it is easy to tell whether one vector is in the span of some others.

Example 5.4.4: The span of the vectors representing

{`Pembroke, Bio-Med`}, {`Main, Wriston`}, {`Keeney, Wriston`}, {`Wriston, Gregorian`}

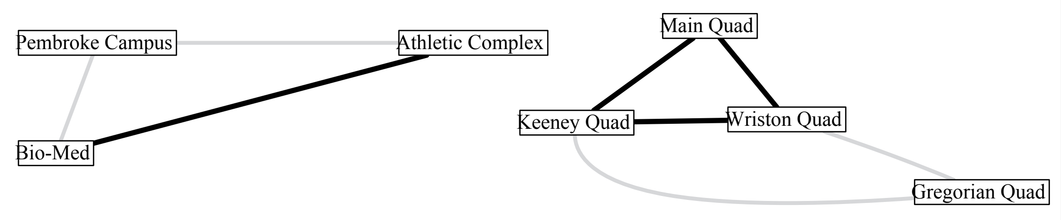
contains the vector corresponding to {`Main, Keeney`} but not the vector corresponding to {`Athletic, Bio-Med`} or the vector corresponding to {`Bio-Med, Main`}.



Example 5.4.5: The span of the vectors representing

{`Athletic, Bio-Med`}, {`Main, Keeney`}, {`Keeney, Wriston`}, {`Main, Wriston`}

does not contain {`Pembroke, Keeney`} or {`Main, Gregorian`} or {`Pembroke, Gregorian`}:



We see that the conditions used in the MSF algorithms to decide whether to add an edge (in the Grow algorithm) or remove an edge (in the Shrink algorithm) are just testing a span condition, exactly as in the vector Grow and Shrink algorithms.

5.5 Linear dependence

5.5.1 The Superfluous-Vector Lemma

To better understand the Grow and Shrink algorithms, we need to understand what makes it possible to omit a vector from a set of generators without changing the span.

Lemma 5.5.1 (Superfluous-Vector Lemma): For any set S and any vector $\boldsymbol{v} \in S$, if $\boldsymbol{v}$ can be written as a linear combination of the other vectors in S then Span $(S - \{\boldsymbol{v}\}) =$ Span S

Proof

Let $S = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$, and suppose

$$\boldsymbol{v}_n = \alpha_1 \, \boldsymbol{v}_1 + \alpha_2 \, \boldsymbol{v}_2 + \cdots + \alpha_{n-1} \, \boldsymbol{v}_{n-1} \tag{5.1}$$

Our goal is to show that every vector in Span S is also in Span $(S - \{\boldsymbol{v}\})$. Every vector $\boldsymbol{v}$

in Span S can be written as

$$\boldsymbol{v} = \beta_1\,\boldsymbol{v}_1 + \cdots \beta_n\,\boldsymbol{v}_n$$

Using Equation 5.1 to substitute for $\boldsymbol{v}_n$, we obtain

$$\begin{aligned}\boldsymbol{v} &= \beta_1\,\boldsymbol{v}_1 + \beta_2\,\boldsymbol{v}_2 + \cdots + \beta_n\,(\alpha_1\,\boldsymbol{v}_1 + \alpha_2\,\boldsymbol{v}_2 + \cdots + \alpha_{n-1}\,\boldsymbol{v}_{n-1}) \\ &= (\beta_1 + \beta_n\alpha_1)\boldsymbol{v}_1 + (\beta_2 + \beta_n\alpha_2)\boldsymbol{v}_2 + \cdots + (\beta_{n-1} + \beta_n\alpha_{n-1})\boldsymbol{v}_{n-1}\end{aligned}$$

which shows that an arbitrary vector in Span S can be written as a linear combination of vectors in $S - \{\boldsymbol{v}_n\}$ and is therefore in Span $(S - \{\boldsymbol{v}_n\})$. □

5.5.2 Defining linear dependence

The concept that connects the Grow algorithm and the Shrink algorithm, shows that each algorithm produces an optimal solution, resolves many other questions, and generally saves the world is... linear dependence.

Definition 5.5.2: Vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n$ are *linearly dependent* if the zero vector can be written as a **nontrivial** linear combination of the vectors:

$$\mathbf{0} = \alpha_1\boldsymbol{v}_1 + \cdots + \alpha_n\boldsymbol{v}_n$$

In this case, we refer to the linear combination as a *linear dependency* in $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n$.

On the other hand, if the *only* linear combination that equals the zero vector is the trivial linear combination, we say $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n$ are linearly *in*dependent.

Remember that a nontrivial linear combination is one in which at least one coefficient is nonzero.

Example 5.5.3: The vectors $[1, 0, 0]$, $[0, 2, 0]$, and $[2, 4, 0]$ are linearly dependent, as shown by the following equation:

$$2\,[1, 0, 0] + 2\,[0, 2, 0] - 1\,[2, 4, 0] = [0, 0, 0]$$

Thus $2\,[1, 0, 0] + 2\,[0, 2, 0] - 1\,[2, 4, 0]$ is a linear dependency in $[1, 0, 0]$, $[0, 2, 0]$, and $[2, 4, 0]$.

Example 5.5.4: The vectors $[1, 0, 0]$, $[0, 2, 0]$, and $[0, 0, 4]$ are linearly independent. This is easy to see because of the particularly simple form of these vectors: each has a nonzero entry in a position in which the others have zeroes. Consider any nontrivial linear combination

$$\alpha_1\,[1, 0, 0] + \alpha_2\,[0, 2, 0] + \alpha_3\,[0, 0, 4]$$

i.e., one in which at least one of the coefficients is nonzero. Suppose α_1 is nonzero. Then the first entry of $\alpha_1\,[1, 0, 0]$ is nonzero. Since the first entry of $\alpha_2\,[0, 2, 0]$ is zero and the first entry of $\alpha_3\,[0, 0, 4]$ is zero, adding these other vectors to $\alpha_1\,[1, 0, 0]$ cannot affect the first entry, so it