Data Literacy

William John Holden

Contents

Preface

This book started with an idle conversation with Mr. Steddum, where I suggested some analytical topics that all Warrants should know. He told me that there is some interest in adding "data literacy" to the curriculum. I wrote an outline of proposed topics and then realized that this outline would work well as a book. For the remainder of my WOILE I worked on the text you have here.

This first edition is only four chapters:

1. An introduction to data types and collections using Excel, JavaScript, and R. We also introduce the term *function* as a foundational concept.
2. A brief discussion of database operations and data processing using SQLite, JavaScript, R, and Go.
3. An explanation of the four moments, with an repeated emphasis of the term *expected value* ($E(x)$), using Excel, R, and JavaScript.
4. An introduction to graph theory terms, concepts, and search algorithms with Neo4j, Go, and Python.

This book contains a lot of code. The reader is *not* expected to be fluent in *any* of the seven programming languages used. These programs are provided as an interactive learning opportunity. Source code in this text is provided along with links to online "playgrounds" where one can edit and run the code using a web browser. This gives the reader an opportunity to work directly with the material, testing ideas and learning new technologies. Another reason for teaching with code is that mathematical notation often hides complexity that one cannot ignore when solving practical problems.

This first edition contains only four chapters. I would like to add four more to cover linear modeling, hypothesis testing (p-values, t-test, χ^2-test), supervised learning (artificial neural networks and decision trees), and unsupervised learning (principal component analysis and clustering). Check https://github.com/wjholden/Data-Literacy for future revisions.

Chapter 1

Introduction

1.1 Parameters and statistics

Statistics are the foundation of most data mining, machine learning (ML), and artificial intelligence (AI) methods today. A *statistic* is an estimate of a *parameter*, which is a characteristic of an entire *population*. Statistics are calculated from taking *samples* (subsets) from the population.

For example, suppose we wanted to find the height of the tallest mountain in the world. We might sample $n = 100$ mountains at random from an almanac. Suppose the tallest mountain in our sample is Mount Fuji. Mount Fuji, the tallest mountain in Japan, is 3776 meters tall. We can conclude that the tallest mountain in the world is *at least* 3776 meters tall.

Our estimate is unfortunately quite low. Mount Everest in Nepal, the *highest* mountain in the world, stands 8849 meters above sea level. Mauna Kea in Hawai'i, the *tallest* mountain in the world, stands 4207 meters above sea level and another 6004 meters below. Our estimates of population parameters, *statistics*, generally improve with larger sample sizes, and many statistical methods provide a *margin of error* quantifying sampling error.

One might use statistics to create a *model* to explain a population, based upon sampling data. Models can be useful both for describing the population and also for forming predictions.

1.2 Levels of measurement

There are four distinct *levels of measurement* that a value may fit [1]. *Nominal* data is simply names or categories, with no concept of order or distance. A movie might be animated or live-action: these are simple categories or order.

Another example might be the film's genre (children, comedy, action, romance, documentary, etc).

Ordinal data has ordering but not distance. Ordinal data might be represented as ordered categories or as numerals, though these numerals do not provide meaningful addition and subtraction. The ratings of a film (G, PG, PG-13, R, and so on) form a ranking, but addition is meaningless (does G + PG-13 = R?) and our concept of distance is weak at best. Another example of ordinal might be the rankings the films receive at an awards ceremony, where one film is the winner and another is the runner-up.

Interval data is numerical data with a concept of distance but not multiplication. The year when a film was produced is an example of interval data. If two films were produced in 2000 and 2010, then it makes sense to say one was made ten years later, but we would not say that the latter film is $\frac{2010}{2000} = 1.005$ times the first.

Ratio data is numerical data with both distance and multiplication. The gross earnings of a film is an example of ratio data. If the 2000 film earned one million dollars and the 2010 film earned two million dollars, then it makes sense to say the second film earned double the first.

Name	Operations	Type
Nominal	$=, \neq$	Categories
Ordinal	$<, >$	Ordered categories
Interval	$+, -$	Numbers with distance
Ratio	$\times, \div$	Numbers with meaningful zero

Interval data might be initially confusing to distinguish from ratio data. One indication is the absence of a meaningful zero. Does zero degrees Celsius or zero degrees Fahrenheit mean the absence of temperature? No. These temperature measurements are simply points along a *scale*. Twenty degrees Celsius is not "twice" ten degrees Celsius; multiplication is not defined on interval data.

Grid coordinates are another example of interval data. One can calculate the distance between two grid coordinates, but we would not say that coordinate 1111 is "half" of coordinate 2222.

Women's pant sizes in the United States, with the confusing size "00," is yet another example of interval data.

Data might be represented in numerical formats when some operations do not make sense. Suppose a political scientist encoded voter's political party as "1", "2", "3", and "4". Is "2" an intermediate value between "1" and "3", or are these actually nominal data where the only arithmetic operations are $=$ and $\neq$? AI methods may form incorrect assumptions about data that domain experts can easily prevent.

1.3 Discretization

Measurements with arbitrarily many decimal digits of precision are *continuous*, whereas measurements with finite steps in between (including categories) are *discrete*. For example, when driving along a road, the house numbers (150 2nd Street, 152 2nd Street, 154 2nd Street…) are discrete; there is no intermediate value between 150 and 151. On the other hand, the grid coordinates associated with each address are continuous; one could (theoretically) specify grid coordinates to the nanometer.

It can be useful to combine continuous measurements into discrete categories. An example might be one's birth date and birth year. No one knows their birth *instant* with subsecond precision. Rather, the year, year and month, or year, month, and day are almost almost always enough information. We even combine years into groups when discussing generations and peer groups. Combining a range of birth years into generational categories is an example of *discretization*.

1.4 Missing values

In practice, *data sets* often have missing values. Different programming languages have substantially different syntax and semantics for representing and handling missing values.

As a small exercise, open Microsoft Excel and enter the values 1, 2, 3, and 5 into cells A1, A2, A3, and A5. Leave cell A4 blank. In cell A6, enter the formula `=PRODUCT(A1:A5)`. The result is $30 = 1 \cdot 2 \cdot 3 \cdot 5$. Excel did *not* treat the missing value as a zero.

Now change cell A4 to `=NA()`. `NA` means "value not available", an explicit indication that a value is not given. The product in cell A6 should update to `#N/A`, which explicitly tells us that there is a problem in the calculation.

Now change cell A4 to `=1/0`. Both cells A4 and A6 should both say `#DIV/0!`, a fault telling us that a division by zero has made further calculation impossible.

Error values propagate from source data through intermediate calculations to final results. If we enter a formula into A7 referencing A6, such as `=SQRT(A6)`, then we will find the same faults in A7 that we see in A6.

Structured Query Language (SQL) databases use the symbol `NULL` to denote missing values. One might build the database *schema* (the structure of the database) to explicitly forbid `NULL` values. For example, `CREATE TABLE Run (Name TEXT NOT NULL, Time INTEGER NOT NULL, Distance REAL NOT NULL)` defines a table *schema* where each of the three columns must be specified. Many programming languages (including C, Java, and JavaScript) also use the term `null` for variables that do not reference any specific value.

Many programming languages support a `NaN` ("not a number") value in error conditions. One might encounter `NaN` when dividing by zero, subtracting infini-

ties, and parsing non-numeric words as numbers. Comparisons with `NaN` can be confusing, such as `NaN == NaN` returning *false*.

Some programming languages will automatically *initialize* variables with some zero value. Other languages give some `Undefined` value to uninitialized variables. Still other languages raise an error if no explicit value is assigned to a variable.

1.5 Strong/weak and static/dynamic typing

Values come in many forms: categorical and numerical, ordered and unordered, discrete and continuous, defined and missing. *Types* can be used to constrain variables to allowable values and applicable operations.

For example, suppose a database indicates how many cars a person owns. It makes no sense to own a fractional or negative car, so we might find an existing type (in this case, whole numbers) or define some new type to model the domain.

Some programming languages offer *dynamic* types that implicitly change the type (*cast*) of values to operate correctly. Go to https://jsfiddle.net or press F12 to open the developer console in most modern browsers. Enter the following into the JavaScript console:

```
>> "5" * 5
<- 25
```

Characters inside quotation marks (`"5"`) are called *strings* and are ordinarily used for text, but JavaScript automatically parses `"5" * 5` as the product of two numerical values and returns `25`.

JavaScript is notoriously inconsistent.

```
>> "5" + 5
<- "55"
```

The resulting string, `"55"`, is the *concatenation* of two strings – perhaps not what one expects.

Many languages and environments seek to automatically parse values. Microsoft Excel and the Python programming language are also dynamic. Other languages, such as Java and Go, are more strict with values and do not automatically change values, especially when the conversion might be "lossy" (where information might be lost, such as approximating the exact value of π as 3.14, or rounding 3.14 to 3, or even changing 3.0 to 3). These languages have both *strong* and *static* typing: the programmer must specify the type of each variable, and lossy type conversions require an explicit cast.

Excel does provide some basic functionality to set number *formats*, but this feature might not stop one from confusing one type of data for another. Excel uses *weak* typing that does prevent one from using unexpected values. Data

analysts can benefit greatly by using the appropriate types for the values in their problem.

1.6 Tables, lists, and data frames

Tables of data are structured in *columns* and *rows*, where the rows represent the *individuals* or *observations* in the data set and the columns represent the *features*. For example, a table of employee names might have two columns (the given and surnames) and ten rows, where each row represents one of the ten employees.

In computer science, the terms *list* and *array* both refer to single-column tables, but with different internal memory representation. The distinction is usually unimportant to data analysts.

Scientific languages, such as Julia and R, often use the term *data frame* (or *dataframe*) as their method for representing tables of data. Data frames often provide rich syntax for row-wise and column-wise operations. By contrast, in an object-oriented language, such as Java and JavaScript, the idiomatic representation of a table is likely an array of objects.

1.7 Vectors and matrices

We now quickly mention the terms *vector* and *matrix* here to disambiguate them from other terms already defined.

Arrays, lists, and columns containing numeric data may sometimes be represented with *vectors*. Likewise, tables and data frames might be represented with *matrices*.

A vector is a quantity with both magnitude and direction, often consisting of two or more elements.

$$\mathbf{x} = (x_1, x_2, x_3, \cdots, x_n)$$

The above vector $\mathbf{x}$ has three components and length $\sqrt{x_1^2 + x_2^2 + x_3^2 + \cdots + x_n^2}$.

A matrix is a collection of vectors used for linear transformations. For example, the three-component *identity matrix*

$$I_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

has the property

$$
\begin{aligned}
I_3\mathbf{x} &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \\
&= \begin{pmatrix} 1 \cdot x_1 + 0 \cdot x_2 + 0 \cdot x_3 \\ 0 \cdot x_1 + 1 \cdot x_2 + 0 \cdot x_3 \\ 0 \cdot x_1 + 0 \cdot x_2 + 1 \cdot x_3 \end{pmatrix} \\
&= \mathbf{x}.
\end{aligned}
$$

Vectors and matrices form the foundations of *linear algebra*, a rich and powerful branch of mathematics that produces many of the results needed for modern statistics, ML, and AI methods.

Remember that the different in ratio and interval data was that *multiplication* is only defined for ratio data. Similarly, multiplication is well-defined for vectors and matrices, but not on tables of data. Depending on the problem domain, it may be inappropriate to use matrices and vectors to represent data where such operations are not necessary.

1.8 Data visualization with plots

Plots allow us to visualize data. Good plots help us to quickly intuit patterns in the data that might otherwise be difficult to understand.

(Note: the term *graph* has different definitions in lower and higher mathematics. We will explain the term "graph" in chapter 5. This text uses the term "plot" as the verb and noun for visualizing data with graphics.)

The *bar plot* helps us to compare the count each category in a discrete (or discretized) variable. The *box plot* helps us to see the center and variation of a numerical variable. The *histogram* also helps us to see the center and variation of a numerical variable, often producing the familiar *bell curve* shape, where the height of the curve indicates the count of observations within the range of each "bin." A histogram is essentially a set of bar plots over discretized numerical values.

A *scatter plot* (sometimes called an XY plot) uses x and y axes to show relationships between two variables. One can also color and shape the points to show third and fourth variables. Three-dimensional XYZ plots are sometimes useful, especially in video and interactive presentations.

As a small exercise to experiment with these four plots, go to https://webr.r-wasm.org/latest/ to use the R language in a web browser. R is a programming language for statistics and data visualization.

R includes several built-in data sets. In the *read-evaluate-print loop* (*REPL*), enter

```
> head(mtcars)
```

to view the column names and first six rows of the Motor Trend Cars (`mtcars`) data set. Now enter the following commands to quickly visualize a few columns in the data set.

```
> barplot(mtcars$cyl)
> boxplot(mtcars$mpg)
> hist(mtcars$mpg)
> plot(mtcars$wt, mtcars$mpg)
```

1.9 Linear and logarithmic scales

Scientists use the term *order of magnitude* to compare values only by the power of 10. One would say $a = 1.6 \times 10^3$ is three orders of magnitude smaller than $b = 8.3 \times 10^6$, which is to say $b/a \approx 1000$.

The *scale* of an axis, such as in bar plot, is the spacing between values. A *linear scale* might show marks at 10, 20, 30, 40, and so on. A *logarithmic scale* might show marks at 10, 100, 1000, 10 000, and so on.

Logarithmic scales can be useful for comparing values that differ by more than one order of magnitude. For example, suppose feature of a data set contains categories a, b, c, and d, and the count of each category is

Category	Count
a	10 736
b	1711
c	398
d	319

Return to https://webr.r-wasm.org/latest/ and plot this data with linear and logarithmic scales:

```
> category_counts <- c(10736, 1711, 398, 319)
> category_counts
[1] 10736  1711   398   319
> barplot(category_counts)
> barplot(category_counts, log="y")
```

1.10 Sets, relations, functions, and algorithms

We now introduce a few terms from *discrete mathematics* that are fundamental to all analysis. A *set* is an unordered collection of *distinct* elements. Sets may be finite or infinite in size. Sets are denoted with curly braces, and the empty set has the special symbol $\emptyset = \{\}$. An example of a set might be

$$W = \{\text{Sunday}, \text{Monday}, \text{Tuesday}, \text{Wednesday}, \text{Thursday}, \text{Friday}, \text{Saturday}\}$$

A *relation* is an association between members of sets. Relations can be used to model any relationship between members any two sets, or even members in the same set. An example might be the relation between integers and elements of W with that many letters, i.e. 6 has a relation on Sunday, Monday, and Friday, 7 has a relation on Tuesday, 8 has a relation on Thursday and Saturday, and 9 has a relation on Wednesday. The term "relation" is seldom used outside of discrete mathematics, but there is a *special case* of a relation that occurs in all mathematical disciplines: *functions*.

A *function* is a relation that uniquely relates members of one set (the *domain*) to another set (the *range*). An example of some functions might be:

$$\begin{aligned}
\text{Translate}\,(\text{Friday}, \text{English}, \text{German}) &= \text{Freitag} \\
\text{Length}\,(\text{Wednesday}) &= 9 \\
\text{Distance}\,(\text{Thursday}, \text{Tuesday}) &= -2 \\
\text{DaysOfLength}\,(6) &= \{\text{Sunday}, \text{Monday}, \text{Friday}\} \\
\text{Sunday} &= \text{Next}\,(\text{Saturday}) \\
&= \text{Previous}\,(\text{Monday}) \\
&= \text{Previous}\,(\text{Previous}\,(\text{Tuesday})) \\
&= \text{Previous}\,(\text{Next}\,(\text{Sunday}))
\end{aligned}$$

Each of these functions accepts one or more *parameters* as *arguments* and returns the unique corresponding value (if any) from its range. It may appear that the third function, DaysOfLength, has returned three values, but actually this function has returned a single set which contains three values.

Many programming languages use the term "function" as a synonym for *procedure*, *subroutine*, and *method*. Functions are "pure" if they have no side-effects, such as mutating a value outside of the function.

The mathematical definition of the term *algorithm* is the set of instructions necessary to solve a problem. Long division, a procedure for manually dividing numbers, is an example of an algorithm. The term "algorithm" has recently entered the popular lexicon in relation to AI systems. Here, the instructions of the algorithm are part of a model, which is created from data.

1.11 Abstraction and Reification

Take any three-digit decimal (base 10) number, reverse the digits, and their difference will always be divisible by both 9 and 11. For example, $321 - 123 = 198$; $198 \div 9 = 22$ and $198 \div 11 = 18$.

How and why would this strange property hold? The proof is quite easy using algebra. We change the digits of a three-digit number into variables. Some three-digit number $abc = 100a + 10b + c$.

$$\begin{aligned} abc - cba &= (100a + 10b + c) - (100c + 10b + a) \\ &= 100a + 10b + c - 100c - 10b - a \\ &= 99a - 99c \\ &= 99(a - c)\square \end{aligned}$$

By *abstracting* numerals into variables, the claim becomes easy to verify.

Here is another example of abstraction. How does one calculate 30% of 70 without a calculator? First, observe that

$$x\% \text{ of } y = \frac{x \times y}{100}.$$

So, 30% of 70 is $\frac{30 \times 70}{100} = \frac{2100}{100} = 21$.

Abstraction can be a powerful tool for solving problems and developing proofs. In the field of computer networking, countless problems are solved by the pattern, "we have more than one thing, but it is inconvenient to operate more than one of these things, so we built a method to abstractly represent arbitrarily many of these things as super-things."

Reification is the opposite of abstraction: we something specific from something general. For example, suppose we have a language translation function

$$T(x, l_1, l_2)$$

where x is the message to be translated, l_1 is the input language, and l_2 is the output language.

$$T(\text{hello}, \text{English}, \text{French}) = \text{bonjour}$$

From this general function, we can enclose parameters l_1 and l_2 into a specific function.

$$\begin{aligned} T'(x) &= T(x, \text{English}, \text{French}) \\ T'(\text{goodbye}) &= \text{au revoir} \end{aligned}$$

Function T' reifys T into a less general form. Such functions might be called *convenience functions* that are provided as a "quality of life" improvement for the user. An example of a convenience function might be `LOG10(number)` in Excel. Excel also provides `LOG(number,[base])` (where `base` defaults to 10 if omitted), but some users may prefer the explicit syntax `LOG10` to improve clarity.

1.12 Discussion prompts

1. Who owns knowledge management?
2. What are good and bad uses for spreadsheets?
3. What is reproducibility? Why would this be important for scientific inquiry?
4. Like a barplot, a pie chart shows the relative sizes of categorical values. What are some advantages and disadvantages of using pie charts?
5. A manager sends an Excel spreadsheet to their employees, telling them to each enter information and send it back. What are some challenges the manager might experience while merging these spreadsheets?

1.13 Practical exercises

1. Create a small survey using Microsoft Forms (part of Office 365) or Google Forms (part of Google Docs). Compare this experience to the hypothetical manager who gathered information by manually merging spreadsheets.
2. Given a dataset, plot the data and explain why this plot technique is appropriate.
3. Given a noisy and poorly structured dataset, propose a method of restructuring the data.
4. Discretize the values of a dataset and explain the reasoning.
5. Be creative and construct intentionally misleading plots that deliberately distort information presented.

Chapter 2

Data Operations

2.1 Relational algebra

Codd's *relational algebra* is the framework theory describing all modern *database management systems* (DBMS) [2]. The relational algebra can be described with five primitives: *selection* (σ), *projection* (π), the *Cartesian product* ($\times$; also known as the *cross product*), set *union* ($\cup$), and set *difference* ($-$).

Selection takes all or a subset of a table's rows. Projection takes all or a subset of a table's columns. In Structured Query Languages (SQL), selection is specified in the `WHERE` clause and projection is specified in the list of columns immediately after `SELECT`.

A Cartesian product is the multiplication of sets. If $A = \{i, j\}$ and $B = \{x, y, z\}$, then $A \times B = \{(i, x), (i, y), (i, z), (j, x), (j, y), (j, z)\}$. The Cartesian product produces the set of all possible pairwise combinations of elements in each set. These composite values are called *tuples*. Tuples may contain more than two values. If $C = \{c\}$, then

$$A \times B \times C = \{(i, x, c), (i, y, c), (i, z, c), (j, x, c), (j, y, c), (j, z, c)\}.$$

As an exercise, go to https://sqlime.org/#deta:mb9f8wq2mq0b to use a DBMS named SQLite. Enter the following commands to reproduce the above Cartesian product.

```
CREATE TABLE A (a text);
CREATE TABLE B (b text);
CREATE TABLE C (c text);

INSERT INTO A(a) VALUES ('i'), ('j');
INSERT INTO B(b) VALUES ('x'), ('y'), ('z');
```

```
INSERT INTO C(c) VALUES ('c');

SELECT * FROM A CROSS JOIN B CROSS JOIN C;
```

This text views tuples as unordered and "flattened" sets, and therefore Cartesian products are both *commutative* ($R \times S = S \times R$) and *associative* ($R \times (S \times T) = (R \times S) \times T$). Some mathematical texts use a stricter definition for the Cartesian product where the result is a set, which does not "flatten" and therefore provides neither commutivity nor associativity. This text uses the looser definition for compatibility with practical DBMSs, including SQLite. Mathematics is partly discovered and partly invented.

Set union, $\cup$, combines two sets. Sets definitionally contain only distinct elements. If $A = \{i, j, k\}$ and $B = \{k, l, m\}$, then

$$A \cup B = \{i, j, k, l, m\}\,.$$

Set difference, $-$, retains the elements of the left set that are not present in the right set.

$$A - B = \{i, j, k\} - \{k, l, m\} = \{i, j\}\,.$$

2.2 Join

The *join* ($\bowtie$) is a combination of the Cartesian product and selection. For example, suppose we have a tables named `Swim`, `Bike`, and `Run`. Each table has a column that uniquely identifies an athlete. To get a triathletes (the athletes who participate in swimming, cycling, and running), we use an *equijoin* to find the product where the names are equal. Return to https://sqlime.org/#deta:36fadcq9apak to demonstrate experiment with the `JOIN` operator.

```
CREATE TABLE IF NOT EXISTS Swim (sn TEXT UNIQUE);
CREATE TABLE IF NOT EXISTS Bike (bn TEXT UNIQUE);
CREATE TABLE IF NOT EXISTS Run (rn TEXT UNIQUE);

INSERT OR IGNORE INTO Swim (sn) VALUES
    ('John'), ('Jane'), ('Luke'), ('Phil');
INSERT OR IGNORE INTO Bike (bn) VALUES
    ('Mary'), ('Alex'), ('Jane'), ('Levi');
INSERT OR IGNORE INTO Run (rn) VALUES
    ('Mike'), ('John'), ('Jane'), ('Sven');

SELECT * FROM Swim, Bike, Run WHERE sn = bn AND sn = rn;
```

There are other syntaxes which achieve the same result using the `ON` and `USING` clauses. As an exercise, try to predict how many rows will return from `SELECT * FROM Swim, Bike, Run` without a `WHERE` clause.

2.3 Grouping and aggregation

DBMSs provide robust *grouping* functions for operating on related rows. Return to https://sqlime.org/#deta:32lpfoo57r8g and create a small table of hypothetical marathon times.

```
CREATE TABLE IF NOT EXISTS Marathon (rn TEXT UNIQUE,
  time INTEGER,
  gender TEXT CHECK( gender IN ('M', 'F') ));

INSERT OR IGNORE INTO Marathon (rn, time, gender) VALUES
  ('Kyle', 2*60*60 + 14*60 + 22, 'M'),
  ('Hank', 2*60*60 + 10*60 + 45, 'M'),
  ('Lily', 2*60*60 + 24*60 + 47, 'F'),
  ('Emma', 2*60*60 + 22*60 + 37, 'F'),
  ('Elle', 2*60*60 + 25*60 + 16, 'F'),
  ('Fred', 2*60*60 + 6*60 + 17, 'M');

SELECT MIN(time) FROM Marathon GROUP BY (gender);
```

`MIN` is one of the *aggregate functions* in SQLite. The `GROUP BY` clause tells the DBMS to split the rows into groups on the `gender` column.

One might be tempted to find the names of our male and female champions with `SELECT rn, MIN(time) FROM Marathon GROUP BY (gender)`. This may work in some DBMSs but there is a subtle bug. It might be obvious that we want the `rn` associated with the `MIN(time)` value, but suppose we change the query to also include `MAX(time)`:

```
SELECT rn, MIN(time), MAX(time) FROM Marathon GROUP BY (gender);
```

Now it is no longer clear which `rn` the query should return. Should the DBMS return the `rn` associated with the `MIN(time)`, the `MAX(time)`, or some other `rn` from the group?

The solution in this particular case is to nest our `MIN(time)` aggregation as a *subquery*.

```
SELECT * FROM Marathon
  WHERE time IN (
    SELECT MIN(time) FROM Marathon GROUP BY (gender));
```

2.4 Filter, map, and reduce

SQL syntax makes it easy to write select, project, and join (SPJ) queries. SQL's grouping and aggregate functions make it possible to perform row-wise and column-wise operations. One can find comparable semantics (with different syntax) in many programming language's *filter*, *map*, and *reduce* functions.

Filter works much like the `WHERE` clause: it takes a subset of the rows, based off of a condition. In JavaScript, we might filter an array with:

```
>> ['cat', 'dog', 'fish', 'bird'].filter(v => v.includes('i'))
<- ['fish', 'bird']
```

Map performs the same function over each element of an input set, creating "mappings" to elements of an output set.

```
>> ['fish', 'bird'].map(v => v.toUpperCase())
<- ['FISH', 'BIRD']
```

Reduce, also known as *fold*, performs some operation on each element of an input set and returns an *accumulator*, which is passed again to the reduce function with the next input value. To take an array's sum, we use an initial accumulator value of 0.

```
>> 15 + 25 + 35
<- 75
>> [15,25,35].reduce((a, v) => a + v, 0)
<- 75
```

For the array's product, we use 1 for the initial accumulator value.

```
>> 15 * 25 * 35
<- 13125
>> [15,25,35].reduce((a, v) => a * v, 1)
<- 13125
```

Both filter and map can be implemented in terms of reduce.

```
>> ['cat', 'dog', 'fish', 'bird'].reduce((a,v) => {
        if (v.includes('i')) {
            a.push(v);
        }
        return a;
    }, [])
<- ['fish', 'bird']
>> ['fish', 'bird'].reduce((a,v) => {
        a.push(v.toUpperCase());
        return a;
    }, [])
<- ['FISH', 'BIRD']
```

Here, we use an empty array (`[]`) instead of a numeric identity as our initial accumulator value.

2.5 Vectorized functions and concurrency

A *vectorized function* automatically iterates over array inputs. This design is less common in traditional languages (C, Java, JavaScript) and more common in scientific programming (R, Matlab, Julia). Some examples in the R language, which one can reproduce at https://webr.r-wasm.org/latest/, are:

```
> c(1, 2, 3) + 4
[1] 5 6 7
> c(1, 2, 3) + c(4, 5, 6)
[1] 5 7 9
> sqrt(c(1, 4, 9))
[1] 1 2 3
```

Observe that the pairwise sums in `c(1, 2, 3) + c(4, 5, 6)` are independent. No sum depends on another, and therefore the computing machine can safely perform each operation in *parallel*.

Concurrency is the ability for a computing machine to perform simulataneous operations. Concurrent programming can be challenging because one *process* or *thread* (sometimes called *task* or *routine*) might interfere with another, but performance benefits often justify the additional complexity.

Some problems can be partitioned into *subproblems* which can be solved in parallel. Other problems cannot. Some encryption algorithms intentionally *chain* the output from one block into the next. One cannot calculate block n without first calculating block $n-1$, and $n-2$, and so on. The reduce operation applies to this algorithm design.

Other problems can be effortlessly partitioned into subproblems and solved quickly with a *divide-and-conquer* approach. A trivial example might be finding the minimum value in a large dataset. One can partition the dataset, find the minimum value in each partition, and then find the minimum value among those results. This process can be repeated.

Go to https://go.dev/play/p/IOwH08R_z7Z to experiment with a divide-and-conquer `minimum` function in the Go language.

```
package main

import "fmt"

func min(x, y int) int {
    if x <= y {
        return x
```

```
    }
    return y
}

func minimum(x []int) int {
    fmt.Println(x)
    n := len(x)
    switch n {
    case 1:
        return x[0]
    case 2:
        return min(x[0], x[1])
    default:
        middle := n / 2
        lower := minimum(x[:middle])
        upper := minimum(x[middle:])
        return min(lower, upper)
    }
}

func main() {
    fmt.Println(minimum([]int{610, 144, 34, 21, 2584, 55, 55}))
}
```

Click the "Run" button several times and observe that the output is completely *deterministic*. Now go to https://go.dev/play/p/Vbe7BWrwlku for a slightly modified version of the same program.

```
    default:
        middle := n / 2
        lower := make(chan int)
        upper := make(chan int)
        go func() { lower <- minimum(x[:middle]) }()
        go func() { upper <- minimum(x[middle:]) }()
        return min(<-lower, <-upper)
    }
```

This version constructs two *channels* for communication among concurrent tasks. We use the `go` keyword to create two *Goroutines* (threads in the Go language), which concurrently solve the `minimum` function over subproblems. Finally, we read the results from each channel with `<-lower` and `<-upper` and return. Click the "Run" button several times and observe that the final result is consistent, but the order of operations is not.

The computer industry has recently turned to *Graphical Processing Units* (GPU) as a fast, inexpensive, and energy-efficient method for solving highly parallelizable problems. GPUs were originally designed to draw computer graphics, which extensively use matrix and vector multiplication. These linear transformations

can be performed in parallel and GPU makers designed their products to perform many simple calculations in parallel.

2.6 Consistency, availability, and partition-tolerance (CAP) theorem

Brewer's *CAP theorem* states that a *distributed system* has at most two qualities of *consistency*, *availability*, and *partition-tolerance*. Consider a system of databases with many replicas. The replicas are consistent if they contain perfect copies of the database, and they are available only they are writable. The distributed system is partition-tolerant if all replicas remain identical, but this is impossible if one allows writes that cannot propagate into the other partition.

The CAP theorem has many practical implications on data integrity and should be considered in design methodology. One must anticipate server and network outages that would create a partition in the distributed in the system and then choose the desired behavior. Can we accept lost database writes when we reconcile after a partition is restored? Or should be accept service outages in order to protect the integrity of the database during an interruption?

A partial solution is to weaken our definition of each quality. Perhaps a system reserves certain rows or columns that are only writable by a specific database, guaranteeing that there will be no conflict if this database continues to write to those changes during a partition. A system might establish some form of confidence intervals in certain data, such as the position of a tracked aircraft with error margins, in recognition that imperfect information might still be useful. Finally, a system might use a quorum model (i.e., 3 of 5 available nodes) to preserve partial availability in the majority partition.

2.7 Discussion prompts

1. How does the CAP theorem impact intelligence and fires in relation to the command and control (C2) warfighting function (WfF)?
2. Where should unclassified data be stored and processed?
3. What are some methods to prevent conflicts among concurrent writes in a shared database?
4. What could possibly go wrong when altering database schema?

2.8 Practical exercises

1. Create a custom list in SharePoint that provides multiple views showing grouped and aggregated values.

2. Given a noisy dataset, identify problems in each column that could influence inclusion and exclusion criteria.
3. Implement filter and map in terms of reduce using a programming language which provides reduce.
4. Define an "embarrassingly parallel" problem and provide both examples and counterexamples.

Chapter 3

Measures of Central Tendency

3.1 Least squares method

The canonical definition of the *arithmetic mean* for a set of n numbers x is

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i = \frac{x_1 + x_2 + x_3 + \cdots + x_4}{n}.$$

Where does this definition come from? Let's open a new workbook in Microsoft Excel and find out.

Leave A1 blank. This will become our *estimate of the mean*. In fields B1 through B9, enter integers 1 through 9. In fields C1 through C9, enter formulas `=B1-$A$1`, `=B2-$A$1`, `=B3-$A$1`, and so on (keeping `$A$1` fixed). In fields D1 through D9, enter formulas `=POWER(C1,2)`, `=POWER(C2,2)`, and so on. In field E1, enter formula `=SUM(D1:D9)`. Finally in field F1, enter the formula `=AVERAGE(B1:B9)`.

Now go to Data, What-If Analysis, Goal Seek. In the Goal Seek dialog, enter `Set cell:` to `E1`, `To value:` to `0`, and `By changing cell:` to `A1`. Click OK. The Goal Seek function runs and should produce a value in A1 near to that in F1. (Goal Seek is not foolproof. Enter `10` into A1 to nudge Excel with a hint if you get a ridiculous answer.)

We have used Goal Seek to minimize the *sum of the squared differences* between our values, x_i, and our estimate of the mean, $\bar{x}$. This *least squares method* dates back to Carl Friedrich Gauss and Adrien-Marie Legendre in the 1800s [3] [4].

Squaring the errors makes the values positive, which prevents underestimates from negating overestimates. One might also consider the *absolute value* (`ABS`

in Excel) as an alternative, but there is a second reason for squaring the errors. Squaring the errors penalizes large errors more than small errors. Accepting small errors but avoiding large errors is the bias that gives the least squares method its strength.

3.2 Calculus-based derivation

Readers familiar with calculus may recognize that one can find the arithmetic mean, μ, by finding the zero in the *derivative* for the sum of the squared errors (SSE) function. Let X be a sample of size n.

$$X = \{x_1, x_2, ..., x_n\}.$$

Then the errors of our estimate of the mean, $\bar{x}$, can be found using the error function

$$\text{Err}(\bar{x}) = X - \bar{x} = \{x_1 - \bar{x}, x_2 - \bar{x}, ..., x_n - \bar{x}\},$$

and the sum of the squared errors is

$$\text{SSE}(\bar{x}) = (x_1 - \bar{x})^2 + (x_2 - \bar{x})^2 + ... + (x_n - \bar{x})^2.$$

To minimize SSE, we take the derivative of SSE in respect to $\bar{x}$ and find its zero.

$$\begin{aligned}
0 &= \text{SSE}'(\bar{x}) \\
&= \left((x_1 - \bar{x})^2 + (x_2 - \bar{x})^2 + ... + (x_n - \bar{x})^2\right)' \\
&= \left((x_1^2 - 2x_1\bar{x} + \bar{x}^2) + (x_2^2 - 2x_2\bar{x} + \bar{x}^2) + ... + (x_n^2 - 2x_n\bar{x} + \bar{x}^2)\right)' \\
&= (-2x_1 + 2\bar{x}) + (-2x_2 + 2\bar{x}) + ... + (-2x_n + 2\bar{x}) \\
&= -2x_1 - 2x_2 - ... - 2x_n + n(2\bar{x}) \\
-2n\bar{x} &= -2x_1 - 2x_2 - ... - 2x_n \\
\bar{x} &= \frac{-2x_1 - 2x_2 - ... - 2x_n}{-2n} \\
&= \frac{x_1 + x_2 + ... + x_n}{n}
\end{aligned}$$

The arithmetic mean is found at $\mu = \bar{x} = \frac{x_1 + x_2 + ... + x_n}{n}$. □

A less-general demonstration of the above proof is given below in the Wolfram Language.

```
Wolfram Language 14.0.0 Engine for Microsoft Windows (64-bit)
Copyright 1988-2023 Wolfram Research, Inc.

In[1]:= X := {x1, x2, x3}

In[2]:= Err[mu_] := X - mu

In[3]:= SSE[mu_] := Total[Err[mu]^2]

In[4]:= SSE'[mu]

Out[4]= -2 (-mu + x1) - 2 (-mu + x2) - 2 (-mu + x3)

In[5]:= Solve[SSE'[mu] == 0, mu]

                 x1 + x2 + x3
Out[5]= {{mu -> ------------}}
                       3
```

3.3 Expected values

The term *average* can refer to *mean*, *median*, and *mode*. Mean only applies to interval and ratio data. Median is simply the middle value among ordinal, interval, and ratio data. Mode is the "commonest" (most frequent) value among nominal, ordinal, interval, and ratio data.

Average	Levels of measurement	Symbols
Mean	Interval, ratio	μ, $\bar{x}$
Median	Ordinal, interval, ratio	(None)
Mode	Nominal, ordinal, interval, ratio	(None)

All three of these *measures of central tendency* enable us to find the *expected value* in a data set, $E(x)$. Population means are assigned the symbol μ. Estimates of the population mean (the sample mean) usually use the name of the sample with a bar, such as $\bar{x}$.

Median and mode can be useful even when analyzing interval and ratio data. Consider a classroom of 10 students who are 6 years old and 1 teacher who is 50 years old. If one selects a random person in the room, what is the expected value for their age? In this case, the modal value (6) is likely a better estimate than the mean value (10).

3.4 The Four Moments

The *four moments* describe the *distribution* of values in a data set. The first moment is the mean. The second moment is *variance*, the expected squared difference of values to the mean. The third moment is *skewness*, the expected cubed difference of values to the mean. The fourth moment is *kurtosis*, the expected difference of values to the mean raised to the fourth power.

Moment	Name	Definition	Symbol
μ_1	Mean	$E(x)$	μ
μ_2	Variance	$E(x-\mu)^2$	σ^2
μ_3	Skewness	$E(x-\mu)^3$	β_1
μ_4	Kurtosis	$E(x-\mu)^4$	β_2

Variance (σ^2) is calculated from the sum of the squared differences in the random variable (x) and the mean (μ).

$$\begin{aligned}\sigma^2 &= E(x-\mu)^2 \\ &= E(x^2 - 2x\mu + \mu^2) \\ &= E(x^2) - 2E(x)\mu + \mu^2 \\ &= E(x^2) - 2(\mu)\mu + \mu^2 \\ &= E(x^2) - 2\mu^2 + \mu^2 \\ &= E(x^2) - \mu^2.\end{aligned}$$

To calculate the sample variance (s^2) in practice, we use the formula

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2.$$

where n is the number of elements in x. Observe that the calculation for this statistic is similar to the least squares method in section 3.1. s^2 is the average squared distance from the mean within the data set.

The *standard deviation* is the square root of variance,

$$s = \sqrt{s^2}.$$

An *outlier* is a value that is very different from other values in the data set. Let $x = \{15, 75, 79, 10, 7, 54, 4600, 91, 45, 86\}$. One can immediately intuit that the value 4600 is substantially different from all of the other values. Outliers can be defined in terms of the mean and standard deviation, where outliers are any values greater than $\bar{x} + 2s$ or less than $\bar{x} - 2s$. We can use the `mean` and `sd`

functions with `subset` in the R language at https://webr.r-wasm.org/latest/ to confirm that 4600 is an outlier.

```
> x = c(15, 75, 79, 10, 7, 54, 4600, 91, 45, 86)
> subset(x, x <= mean(x) - sd(x) | x >= mean(x) + sd(x))
[1] 4600
```

We can use skewness (β_1) to detect whether the data is imbalanced (skewed) above or below the mean. If skewness is negative then the left tail is longer, if skewness is positive then the right tail is longer, and if skewness is zero then the distribution is equally balanced over the mean. The Excel function for skewness is `SKEW`.

We can use kurtosis (β_2) to detect if a data set contains outliers. The kurtosis of the normal distribution is 3. Karl Pearson defines the *degree of kurtosis* as $\eta = \beta_2 - 3$ [5, p. 181]. Other texts call this *excess kurtosis*. Excel's `KURT` function returns excess kurtosis, so when `KURT` returns 0 then the distribution may fit the normal and contains no outliers.

3.5 Exponential moving averages

An *exponential moving average* (EMA) is a weighted average that creates a bias favoring recent observations. EMAs are used in the financial sector as an implicit means of modeling stock prices with time.

EMA is defined *recursively* and parameterized with a weighting multiplier $0 < p < 1$.

$$\text{EMA}\left(x, i\right) = \begin{cases} px_i + (1-p)\text{EMA}\left(x, i-1\right), & \text{if } i > 1. \\ x_1, & \text{otherwise.} \end{cases}$$

EMA can be easily implemented in terms of the `reduce` operation, as shown below in JavaScript.

```
>> x = [7, 8, 9, 10, 9, 8, 7, 9, 11, 13, 15, 17]
>> p = .25
>> x.reduce((ema,v) => p * v + (1-p) * ema, x[0])
<- 12.661770105361938
```

The *base case* is at x_1 in mathematical notation but `x[0]` in JavaScript. This is a matter of convention; the first element of a list is position 1 in math, but 0 in many programming languages.

3.6 Discussion prompts

1. Is four a lot?

2. First battalion has an average ACFT score of 482 while second battalion has an average ACFT score of 491. Which is better?
3. What do we do when statistics show us something that contradicts our values? For example, suppose we discover that Soldiers of a specific demographic have much lower promotion rates than their peers.
4. Is it more important for an organization to think about variance or the 99th percentile?
5. Given a sample set $x = \{5\}$, what is the estimate of the mean ($\bar{x}$), and what is the sample variance (s_x)? That is, what is the expected value ($E(x)$) of a random sample taken from x, and what is the average difference of variables to the expected value? Use software to verify your answer. In the R language, this would be `mean(c(5))` and `sd(c(5))`.

3.7 Practical exercises

1. Calculate the influence that outliers have on different-sized datasets that contain outliers.
2. Calculate the exponential moving average in a small dataset.
3. Given a dataset and experimental result, identify problems caused by analyzing categorical data represented in a numeric form.
4. Given multiple datasets with identical mean and standard deviation, use kurtosis to identify the dataset with more outliers.
5. Design or implement an algorithm to incrementally calculate standard deviation, where the estimate of the sample standard deviation is updated with each additional value.

Chapter 4

Dimensionality

4.1 Combinatorics

Suppose one has four children, $\{a, b, c, d\}$, and a motorcycle. The motorcycle can carry only one passenger, so there are four possible *combinations* of children that you can transport by motorcycle:

$$4 = |\{\{a\}, \{b\}, \{c\}, \{d\}\}|.$$

$|S| = n$ gives the *cardinality* (the size) n of set S.

The family adds a sidecar to the motorcycle and can now transport two children at once. There are now six ways that one can *choose* two elements from a four-element set:

$$6 = |\{a, b\}, \{a, c\}, \{a, d\}, \{b, c\}, \{b, d\}, \{c, d\}|.$$

Recall from section 1.10 that sets are unordered; $\{a, b\}$ is equal to $\{b, a\}$.

Two common notations for the number of possible subsets we can choose are $\binom{n}{r}$ and nCr. The former is favored in higher mathematics, the latter in secondary schools. $\binom{n}{r}$ is read "n choose r" and nCr is read "n combinations taken r at a time."

The family purchases a small car that can transport three passengers:

$$\binom{4}{3} = 4 = |\{a, b, c\}, \{a, b, d\}, \{a, c, d\}, \{b, c, d\}|.$$

The family purchases a larger car that can carry four passengers:

$$\binom{4}{4} = 1 = |\{a, b, c, d\}|.$$

Finally, the family crashes the large car and is left with a bicycle. The bicycle has no capacity to carry passengers, and therefore

$$\binom{4}{0} = 1 = |\{\}| = |\emptyset|.$$

There is only one way to take an empty set from another set.

We will now construct a generalized function to count the number of subsets for any combination of r elements taken from a set of size n. Initially, consider the first element in the set. If we choose this element, then we still to select $r-1$ elements from the remaining $n-1$ elements. If we do not choose this element, then we still must choose r elements from the remaining $n-1$ elements. This gives us *Pascal's formula*, a *recursive* definition for counting combinations.

$$\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r-1}$$

We need at least one *base case* to prevent this function from entering an infinite loop. These identities should be intuitive from the earlier exercise, though the proof for the final case is left as an exercise to the reader.

$$\binom{n}{r} = \begin{cases} 1, & \text{if } n = r. \\ 1, & \text{if } r = 0. \\ n, & \text{if } r = 1. \\ 0, & \text{if } n < r. \end{cases}$$

Implemented in the R language (https://webr.r-wasm.org/latest/),

```
pascal <- function(n,r) {
  if (n < r) {
    return(0)
  } else if (n == r) {
    return(1)
  } else if (r == 0) {
    return(1)
  } else if (r == 1) {
    return(n)
  } else {
    return(pascal(n-1,r) + pascal(n-1,r-1))
  }
}
```

we can reproduce the results of our passengers example. The `sapply` function in R is comparable to the `map` operation (see section 2.4).

```
> sapply(0:4, function(r) pascal(4, r))
[1] 1 4 6 4 1
```

4.2 Permutations

An alternative definition for the combination formula requires *permutations* – ordered subsets. From set $S = \{a, b, c, d\}$ there are twelve two-element permutations, represented here as *tuples*: (a, b), (b, a), (a, c), (c, a), (a, d), (d, a), (b, c), (c, b), (b, d), (d, b), (c, d), and (d, c).

When counting the size of the permutation set of length r chosen from a set of size n, we begin wtih n possible elements for the first tuple element, then $n - 1$ possible elements for the second tuple element, and so on until all r tuple elements are filled.

$$nPr = n \times (n-1) \times (n-2) \times ... \times (n-r+1) = \frac{n!}{(n-r)!}$$

The *permutation formula* is usually defined using the *factorial* function, denoted by the "!" postfix operator.

$$n! = n \times (n-1) \times (n-2) \times ... \times 2 \times 1 = \prod_{i=1}^{n} i$$

$0! = 1$ by definition. The intuition for this is the bicycle: there was one way to choose an empty set from a set, and likewise there is one empty tuple of zero ordered elements taken from a set.

The number of $r = n$-length permutations of a set of size n is simply

$$nPn = \frac{n!}{(n-n)!} = \frac{n!}{0!} = \frac{n!}{1} = n!$$

Now we can define the combination formula in terms of the permutation formula. We count the number of permutations but de-duplicate this count, as combinations are unordered. The number of duplicated entries is $rPr = r!$.

$$nCr = \frac{nPr}{r!} = \frac{\frac{n!}{(n-r)!}}{r!} = \frac{n!}{r!\,(n-r)!}$$

4.3 The curse of combinatorics

The "curse of combinatorics" is simply that the number of possible combinations can become very large. Consider a bicycle factory that must manufacture a part in four materials (steel, aluminum, carbon fiber, and titanium), three sizes (small, medium, and large), five styles (road, mountain, touring, utility, and economy), and for five markets (North America, European Union, Latin America, East Asia, and Middle East) which each have different compliance requirements. There are $4 \times 3 \times 5 \times 5 = 300$ distinct variations of this part. Suppose a distributor wants to warehouse 50 of each part, but expects the factory to wait until the part is sold before receiving payment. Should the factory give the the distributor $300 \times 50 = 15\,000$ of this part?

Now suppose an investor wants a rigorous test of the bicycle factory's products. The investor demands that 30 copies of each part be tested in various ways. $300 \times 30 = 9000$ total parts being committed to this study might be unrealistic.

4.4 Sample spaces

Imagine one wanted to conduct a large study on exercise and health outcomes. Basic demographic variables include age, gender, location, height, weight, and race. Exercise variables in this study include weekly minutes performing cardiovascular training, minutes of resistance training, and minutes of flexibility training. Other exercise variables in this study include metrics of speed, endurance, strength, flexibility, blood pressure, resting heart rate, body composition, bone density, and sleep duration.

Suppose we discretize (see section 1.3) each continuous variable into discrete categories. For example, we might change the age variable from its numeric values to categories 1-10, 11-20, 21-30, and so on. We separate height into very short, short, average, tall, and very tall. We categorize minutes of weekly training into 0-20, 20-60, 60-120, and 120+. Some variables are divided into very low, low, medium, high, and very high. The process continues until all variables can be represented in discrete (sometimes ordered) categories instead of continuous numeric values.

Variable	Categories
Age	10
Gender	2
Location	5
Height	5
Weight	10
Cardio Minutes	4
Weights Minutes	4
Stretch Minutes	4
Speed	10

Variable	Categories
Strength	10
Endurance	10
Flexbility	10
Blood Pressure	5
Heart Rate	5
Composition	7
Bone Density	5
Sleep Duration	9

One might expect that, having discretized each variable, it would become easy to draw non-obvious conclusions from a reasonable sample size. Unfortunately, there are $10 \times 2 \times 5 \times 5 \times 10 \times 4 \times 4 \times 4 \times 4 = 320\,000$ possible combinations in the first eight variables alone. Is it unusual for a middle-aged, very tall, very heavy, zero-exercise male living in North America to have average fitness metrics with poor body composition? We would ideally like to sample many such persons, but even in a large study we likely would not have many individuals meeting exactly these characteristics.

Data mining is the search for non-obvious conclusions by analyzing data. Data mining efforts are especially characterized by the lack of *first principals*, meaning the researcher may not have any advance hypothesis about the relationships between variables.

Suppose our fitness research showed that heavy bodyweight predicts poor speed. This is quite obvious and likely not useful. Suppose our fitness research showed that minutes of stretching predicted not only flexibility but also strength and body composition. Such as result is less expected, and might (just as a hypothetical example) lead to a discovery that yoga develops muscle better than its reputation.

Data mining efforts in n-dimensional space are basically always complicated by this "curse of combinatorics." When we repeatedly multiply many variables together, we find that the space of possible combinations becomes so large that even very large samples cover only tiny portions. Our example health study has a total of $10 \times 2 \times 5 \times 5 \times 10 \times 4 \times 4 \times 4 \times 10 \times 10 \times 10 \times 10 \times 5 \times 5 \times 7 \times 5 \times 9 = 25\,200\,000\,000\,000$ possible states in its *sample space*.

4.5 Pareto fronts

4.6 Covariance

4.7 Discussion prompts

4.8 Practical exericses

1. Use nested `sapply` statements to improve `sapply(0:4, function(r) pascal(4, r))`. Iterate `pascal(n, r)` over $0 \leq n \leq 10$ and $0 \leq r \leq n$, generating the first 11 lines of Pascal's Triangle. Compare the result to `sapply(0:10, function(n) choose(n, 0:n))`. Why does the built-in `choose` function accept ranges (`0:n`) when our own `pascal` function does not?

Chapter 5

Graph Theory

5.1 Vertices, edges, and paths

A *graph* (G) is a collection of *vertices* (V; also known as *nodes* or *points*) and the *edges* (E; also known as *relations* or *lines*; see section 1.10) connecting them.

$$G = \{V, E\}$$

Edges are conventionally named for the vertices they connect. For example, let $V = \{u, v\}$ and $E = \{(u, v)\}$, then $G = \{V, E\}$ is a graph with two vertices, u and v, and one edge, (u, v).

Graphs can be used to model any form of *network* where the elements of sets bear relations. Graphs can be directed, where the source and destination vertices in an edge are significant ($(u, v) \neq (v, u)$), or undirected ($(u, v) = (v, u)$).

Figure 5.1: Graphs are conventionally visualized as circles (vertices) connected by lines (edges).

A *path* is a series of edges that *transitively* connect two vertices that are not directly connected. We say that $u \rightsquigarrow v$ (u leads to v) if the graph contains some path from u to v.

The edges of a graph may have a *distance function* (δ; also known as *weight* and *cost*), which relates each edge with some real number. Such graphs are *weighted graphs*. For example, suppose a high-speed railroad has train stations in Paris, Brussels, and the Hague. The distance from Paris to Brussels is about 350 km and the distance from Brussels to Hague is another 180 km. The total path length from Paris to Hague via Brussels is therefore 350 km + 180 km = 530 km.

$$\begin{aligned}\delta(\text{Paris}, \text{Hague}) &= \delta(\text{Paris}, \text{Brussels}) + \delta(\text{Brussels}, \text{Hague}) \\ &= 350\,\text{km} + 180\,\text{km} \\ &= 530\,\text{km}.\end{aligned}$$

Let us quickly reproduce this result using a *graph database* named Neo4j. Go to https://console.neo4j.org and click the "Clear DB" button. Enter the below *Cypher query* into the input field and press the triangle-shaped execute button.

```
CREATE
    (paris:CITY {name:"Paris"}),
    (brussels:CITY {name:"Brussels"}),
    (hague:CITY {name:"Hague"}),
    (paris)-[:ROUTE {dist:350}]->(brussels),
    (brussels)-[:ROUTE {dist:180}]->(hague);
```

This command created three vertices and two edges. Verify and visualize this graph with the below query. Neo4j's Cypher language uses `MATCH` as its selection (σ) operator.

```
MATCH (x:CITY)
RETURN x;
```

Finally, query the database for a path of any length (`*`) connecting Paris to Hague, returning the path (`p`) and its cumulative distance. Refer to section 2.4 for a description of the reduce operation.

```
MATCH (:CITY {name:'Paris'})-[p:ROUTE*]->(:CITY {name:'Hague'})
RETURN p, REDUCE(length=0, e IN p | length + e.dist) AS distance;
```

5.2 Special cases of graphs

A *directed acyclic graph* (DAG) is a special case of a directed graph. A *cycle* (also known as a *loop*) occurs in a directed graph when there is any path from some vertex to itself. For example, let G be a directed graph where

$$G = \{\{a, b, c, d\}, \{(a, b), (b, c), (c, d), (c, a)\}\}.$$

The edges (a, b), (b, c), and (c, a) form a cycle. Vertices a, b, and c together form a *connected component*. A directed graph with one or more connected

components is not a DAG, but the graph of components (where vertices of the component subgraph are merged into a "super vertex") is itself a DAG.

A *tree* is another special case of an acyclic graph. Trees are often drawn in a vertical hierarchy where each *child node* has one *parent*, and the only parent node with no parent is called the *root node*. One's ancestoral family tree is an instance of a tree; without a time machine, it is impossible to one to form a loop with an ancestor.

5.3 Representation

Graphs are modeled using either an *adjacency list* or an *adjacency matrix*. An adjacency list might look like

$$\begin{aligned} \text{adj}\,(\text{Paris}) &= \{\text{Brussels}\} \\ \text{adj}\,(\text{Brussels}) &= \{\text{Paris}, \text{Hague}\} \\ \text{adj}\,(\text{Hague}) &= \{\text{Brussels}\}\,. \end{aligned}$$

A separate data structure would be necessary to represent edge weights. *Dictionary* types, such as `dict` in Python and `map` in Go, can be convenient implementations for both adjacency lists and edge weight functions.

An adjacency matrix represents edges with weights in a single data structure, such as

$$E = \begin{array}{r} \\ \text{Paris} \\ \text{Brussels} \\ \text{Hague} \end{array} \begin{array}{c} \begin{array}{ccc} \text{Paris} & \text{Brussels} & \text{Hague} \end{array} \\ \begin{bmatrix} 0 & 350 & \infty \\ 350 & 0 & 180 \\ \infty & 180 & 0 \end{bmatrix} \end{array}.$$

In this example, the distance of a vertex to itself is defined as zero,

$$\delta\,(u, u) = 0$$

and the distance between vertices is considered infinite if those vertices are not directly connected by an edge.

$$(u, v) \notin E \implies \delta\,(u, v) = \infty$$

The $\in$ operator and its negation, $\notin$, tests whether an object is an "element of" a set; $\in$ is read "in" and $\notin$ is read "not in." The symbol $\implies$ is for *conditional implication* and is read "implies." If the *statement* on the left of $\implies$ is true, then the statement on the right must also be true.

5.4 Search algorithms

5.4.1 Depth-first search

Imagine a video game where the player searches the kingdom for treasure. The player has no knowledge of where the treasure might be, so from a starting point they fully explore the forest, mountains, and sea. Both the starting point and the sea connect to opposite sides of the city. Upon entering the city from the sea-side, the player explores the city and discovers the treasure. This is an example of a *depth-first search* (DFS).

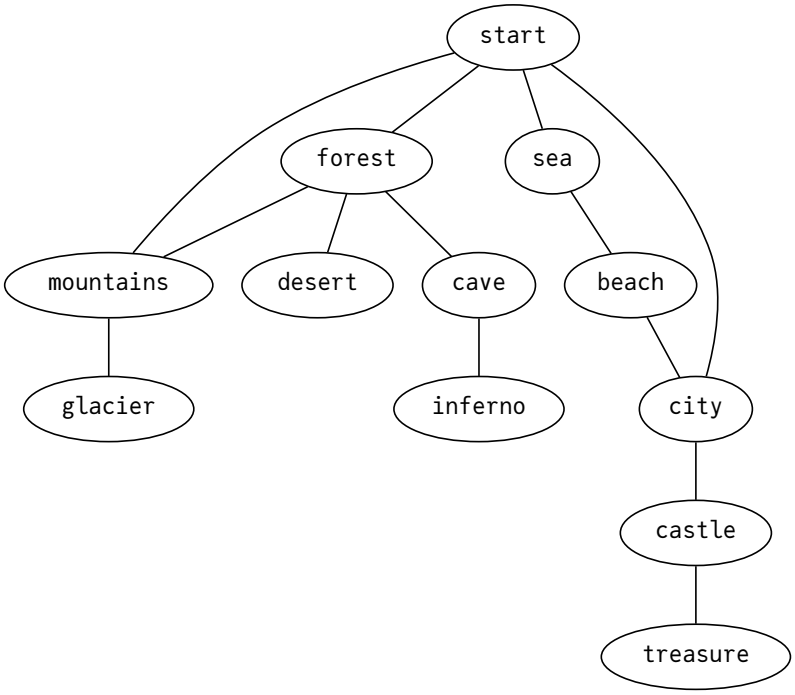

Figure 5.2: There are many paths from the starting point to the treasure in this kingdom.

Go to https://go.dev/play/p/AuH2qOgSG-c to run the following DFS implementation, written in Go. This implementation uses a *recursive* definition of the DFS function (the DFS function invokes itself as it explores the graph). The function uses an external data structure (`quest`) to identify which vertices have already been discovered. Upon successful search, the function prints its position in the graph as the recursive calls "unwind."

```
package main

import "fmt"

var g = map[string][]string{
    "start":     []string{"forest", "mountains", "sea", "city"},
    "forest":    []string{"start", "mountains", "desert", "cave"},
    "mountains": []string{"start", "forest", "glacier"},
    "desert":    []string{"forest"},
    "cave":      []string{"forest", "inferno"},
    "inferno":   []string{"cave"},
    "glacier":   []string{"mountains"},
    "sea":       []string{"start", "beach"},
    "beach":     []string{"sea", "city"},
    "city":      []string{"beach", "start", "castle"},
    "castle":    []string{"city", "treasure"},
    "treasure":  []string{"castle"}}

var quest map[string]string = make(map[string]string)

func dfs(src, dst string) bool {
    quest[src] = "discovered"

    if src == dst {
        fmt.Printf("Discovered %s:", dst)
        return true
    }

    for _, neighbor := range g[src] {
        if quest[neighbor] != "discovered" {
            if dfs(neighbor, dst) == true {
                fmt.Printf(" %s", src)
                return true
            }
        }
    }

    return false
}

func main() {
    dfs("start", "treasure")
    fmt.Println()
}
```

The program should output `Discovered treasure: castle city beach sea start`.

DFS successfully discovers the treasure, but we have no guarantee that this algorithm will find the *optimal* (shortest) path.

5.4.2 Breadth-first search

Imagine the protagonist of our hypothetical adventure game was not a lone wanderer, but rather a field marshall commanding a large army. This army explores one region at a time, holding each area as adjacent units proceed into their respective area. The army incrementally expands the radius of the search *frontier* in a search technique called *breadth-first search* (BFS). Once one unit discovers the treasure, we are certain that no shorter path was possible thanks to an *invariant* in our search algorithm.

Maintaining an invariant is essential for *mathematical induction*, where we establish that some *predicate* P is true for the *base case* $P(0)$ and that $P(k)$ implies $P(k+1)$ and therefore $P(n)$ is true for all $n > 0$. The proof for the correctness of a BFS follows:

1. Along the frontier of radius $r = 0$, the BFS algorithm on graph G has not discovered a path from u to v. $\delta(u, v)$ is therefore at *closest* $r = 1$.
2. At $r = 1$, BFS has not found v and therefore $2 \leq \delta(u, v)$.
3. At $r = 2$, BFS has not found v and therefore $3 \leq \delta(u, v)$.
4. $\vdots$
5. At $r = k$, BFS has not found v and therefore $k + 1 \leq \delta(u, v)$.
6. $\vdots$
7. At $r = n$, BFS has located v and therefore $n = \delta(u, v)$. $\square$

Gp to https://go.dev/play/p/yMIcmcsK_V9 and run the following BFS implementation, written in Go. This implementation uses an *iterative* BFS function. The BFS function does not invoke itself. Instead, the procedure adds unexplored vertices to a queue and records the "parent" of each vertex. We "unwind" the resulting tree from child to parent nodes to construct the shortest path.

```
func bfs(src, dst string) map[string]string {
    parent := map[string]string{src: src}
    queue := []string{src}
    for len(queue) > 0 {
        position := queue[0]
        queue = queue[1:]
        if position == dst {
            break
        }
        for _, neighbor := range g[position] {
            if _, ok := parent[neighbor]; !ok {
                parent[neighbor] = position
                queue = append(queue, neighbor)
            }
        }
```

```
    }
    return parent
}

func main() {
    tree := bfs("start", "treasure")
    fmt.Printf("Discovered treasure:")
    position := "treasure"
    for position != tree[position] {
        position = tree[position]
        fmt.Printf(" %s", position)
    }
    fmt.Println()
}
```

This program should output `Discovered treasure: castle city start`. BFS finds the shortest path between two vertices by *hop count*, but it does not consider edge weights. In the following section, we will find that we can often explore graphs much faster by ordering our breadth-first traversal by cumulative path cost.

5.4.3 Dijkstra's algorithm

Dijkstra's algorithm uses a *priority queue* to visit nodes from shortest to longest path [6]. For this reason, Dijkstra's algorithm is also known as the *shortest-path first* (SPF). Like BFS, Dikjstra's algorithm is a *greedy algorithm* that discovered a globally optimal solution by repeatedly making locally optimal decisions. Let us turn to the Python language to demonstrate Dijkstra's algorithm on our same treasure-hunting graph, this time with edge weights. Run this program at https://www.python.org/shell/. (Note: the Python language is very "picky" about tabs. A plain text verion of this program is available at https://github.com/wjholden/Data-Literacy/blob/main/dijkstra.py.)

```
from heapq import *

g = dict(
    [
        ("start", ["forest", "mountains", "sea", "city"]),
        ("forest", ["start", "mountains", "desert", "cave"]),
        ("mountains", ["start", "forest", "glacier"]),
        ("desert", ["forest"]),
        ("cave", ["forest", "inferno"]),
        ("inferno", ["cave"]),
        ("glacier", ["mountains"]),
        ("sea", ["start", "beach"]),
        ("beach", ["sea", "city"]),
        ("city", ["beach", "start", "castle"]),
```

```
        ("castle", ["city", "treasure"]),
        ("treasure", ["castle"]),
    ]
)

w = dict(
    [
        (("start", "forest"), 70),
        (("start", "mountains"), 60),
        (("start", "sea"), 54),
        (("start", "city"), 81),
        (("forest", "start"), 42),
        (("forest", "mountains"), 51),
        (("forest", "desert"), 56),
        (("forest", "cave"), 63),
        (("mountains", "start"), 71),
        (("mountains", "forest"), 38),
        (("mountains", "glacier"), 72),
        (("desert", "forest"), 93),
        (("cave", "forest"), 19),
        (("cave", "inferno"), 17),
        (("inferno", "cave"), 71),
        (("glacier", "mountains"), 25),
        (("sea", "start"), 49),
        (("sea", "beach"), 88),
        (("beach", "sea"), 79),
        (("beach", "city"), 29),
        (("city", "beach"), 30),
        (("city", "start"), 33),
        (("city", "castle"), 36),
        (("castle", "city"), 39),
        (("castle", "treasure"), 76),
        (("treasure", "castle"), 76),
    ]
)


def dijkstra(src, dst):
    explored = set()
    distance = dict()
    distance[src] = 0
    queue = []
    heappush(queue, (0, src))
    while queue:
        _, current = heappop(queue)
        if current == dst:
```

```
            print("Path found": distance[dst])
            return distance[dst]
        if current in explored:
            continue
        explored.add(current)
        for neighbor in g[current]:
            if neighbor not in explored:
                d = distance[current] + w[(current, neighbor)]
                distance[neighbor] = d
                heappush(queue, (d, neighbor))
    print("No path found")

dijkstra("start", "treasure")
```

This program should output `Path found: 193`, revealing that the shortest path from `start` to `treasure` has a total path cost of 193.

Neo4j produces the same result. We reconstruct our graph in the Cypher language at https://console.neo4j.org:

```
CREATE
    (start:LOCATION {name:'start'}),
    (inferno:LOCATION {name:'inferno'}),
    (beach:LOCATION {name:'beach'}),
    (castle:LOCATION {name:'castle'}),
    (forest:LOCATION {name:'forest'}),
    (mountains:LOCATION {name:'mountains'}),
    (desert:LOCATION {name:'desert'}),
    (cave:LOCATION {name:'cave'}),
    (glacier:LOCATION {name:'glacier'}),
    (sea:LOCATION {name:'sea'}),
    (city:LOCATION {name:'city'}),
    (treasure:LOCATION {name:'treasure'}),
    (start)-[:CONN {distance:70}]->(forest),
    (start)-[:CONN {distance:60}]->(mountains),
    (start)-[:CONN {distance:54}]->(sea),
    (start)-[:CONN {distance:81}]->(city),
    (inferno)-[:CONN {distance:71}]->(cave),
    (beach)-[:CONN {distance:79}]->(sea),
    (beach)-[:CONN {distance:29}]->(city),
    (castle)-[:CONN {distance:39}]->(city),
    (castle)-[:CONN {distance:76}]->(treasure),
    (city)-[:CONN {distance:30}]->(beach),
    (city)-[:CONN {distance:33}]->(start),
    (city)-[:CONN {distance:36}]->(castle),
    (treasure)-[:CONN {distance:76}]->(castle),
    (forest)-[:CONN {distance:42}]->(start),
```

```
    (forest)-[:CONN {distance:51}]->(mountains),
    (forest)-[:CONN {distance:56}]->(desert),
    (forest)-[:CONN {distance:63}]->(cave),
    (mountains)-[:CONN {distance:71}]->(start),
    (mountains)-[:CONN {distance:38}]->(forest),
    (mountains)-[:CONN {distance:72}]->(glacier),
    (desert)-[:CONN {distance:93}]->(forest),
    (cave)-[:CONN {distance:19}]->(forest),
    (cave)-[:CONN {distance:17}]->(inferno),
    (glacier)-[:CONN {distance:25}]->(mountains),
    (sea)-[:CONN {distance:49}]->(start),
    (sea)-[:CONN {distance:88}]->(beach);
```

and then we can use the built-in `shortestPath` function to obtain the same result.

```
MATCH
    (src:LOCATION {name:'start'}),
    (dst:LOCATION {name:'treasure'}),
    path = shortestPath((src)-[:CONN*]->(dst))
RETURN
    path,
    REDUCE(d=0, e in relationships(path) | d + e.distance)
    AS distance;
```

Neo4j should also report a distance of 193.

As an exercise, change `->(dst)` to `-(dst)` and re-run the query with this change. The total distance is now 145. The difference in `()-[]->()` and `()-[]-()` is that one is a directed edge, the other is undirected. With `-()` instead of `->()`, Neo4j treats all edges in the graph as undirected. Neo4j allows *parallel edges* that connect the same two vertices.

5.4.4 Informed search with A*

DFS, BFS, and Dijkstra's algorithms are all *uninformed* search algorithms. The A* algorithm is an *informed* search algorithm: it uses some *heuristic* to explore its frontier ordered by minimum estimated distance to the destination [7].

A* can solve hard problems that are not immediately recognizable as searches. The canonical example is the 8-piece puzzle problem. An 8-piece puzzle arranges 8 movable square tiles into a 3×3 grid. One position is empty. The challenge is to slide the pieces until all pieces are in order.

$$\begin{bmatrix} 5 & 4 & 1 \\ & 2 & 8 \\ 3 & 6 & 7 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & \end{bmatrix}$$

The choice of heuristic function influences the path A* chooses as it searches

for a solution. One heuristic function for the 8-piece puzzle problem returns the count of mismatched pieces. If two pieces are out of place then the heuristic distance is 2, if three pieces are out of place then the heuristic distance is 3, and so on. An alternative, more sophsticated heuristic function, uses the *Manhattan distance* (also known as *Taxicab distance*) of each puzzle piece to its destination. Manhattan distance is the sum of unsigned differences of each dimension between two coordinates in n-dimensional space. The intuition is that a taxicab cannot fly in a straight line, but rather has to corner the rectangular blocks of Manhattan.

$$d_T(p, q) = \sum_{i=i}^{n} |p_i - q_i|$$

For example, if $p = (x_1, y_1)$ and $q = (x_2, y_2)$, then $d_T(p, q) = |x_1 - x_2| + |y_1 - y_2|$. A heuristic function for an A* solution to the 8-piece puzzle problem uses Manhattan distance to quantify the closeness of the current game state to the desired solution. This approach works because progress *towards* the solution ultimately results in a global solution. The design of the heuristic function is key to the informed search technique.

Some problems contain *local extrema* (local but not global minimal or maximal values) that might stop the search at a suboptimal solution. If the problem has an extremely large space (there are too many candidate solutions to search exhaustively), then it may be acceptable to accept a "good-enough" local best candidate solution as an approximation for the global optimum. The *local search* technique uses an *ensemble* of *search agents* which independently search *neighborhoods* of the problem space. A local search could be built upon A* searches from different origins. Ideally, the A*-based local search should explore the problem with reasonable depth, mobility, and coverage [8].

Interestingly, the 8-piece and comparable 15-piece puzzle problems are not guaranteed to be solvable. Exactly half of all possible $9! = 362\,880$ and $15! = 1\,307\,674\,368\,000$ permutations are reachable from any random 8- and 15-piece puzzle game state [9]. n-piece puzzles ($n \geq 3$) contain two partitions and are therefore classified as *bipartite* graphs. The puzzle is unsolvable if started in the partition that does not contain the solution. We must understand that search algorithms might not be able to solve a problem if the graph of the problem space contains a partition.

5.4.5 A* and the Stable Marriage Problem

We will demonstrate the A* informed search algorithm on the *Stable Marriage Problem* [10]. The Stable Marriage Problem seeks to pair the members of two equal-sized sets to one another based upon their mutual preferences. This problem and its solution are applied to many practical situations, including the Army Talent Alignment Process (ATAP); see

https://www.youtube.com/watch?v=9mEBe7fzrmI for an official and detailed explanation. Explained with marriages, the problem has all of the men rank all of the women from most to least preferred. Correspondingly, the women also rank all of the men from most to least preferred.

$$M = \begin{array}{r} \\ \text{Man 1} \\ \text{Man 2} \\ \text{Man 3} \end{array} \begin{array}{c} \begin{array}{ccc} \text{Woman 1} & \text{Woman 2} & \text{Woman 3} \end{array} \\ \begin{bmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \\ 2 & 3 & 1 \end{bmatrix} \end{array}$$

$$W = \begin{array}{r} \\ \text{Woman 1} \\ \text{Woman 2} \\ \text{Woman 3} \end{array} \begin{array}{c} \begin{array}{ccc} \text{Man 1} & \text{Man 2} & \text{Man 3} \end{array} \\ \begin{bmatrix} 3 & 1 & 2 \\ 2 & 3 & 1 \\ 1 & 2 & 3 \end{bmatrix} \end{array}$$

Each man is paired to one woman. The set of matching is considered "stable" when there is no "*rogue couple*": where a man x who prefers some other woman y to his current wife *and* where woman y prefers man x to her current husband. (The solution does not need to give everyone their first preference. It is acceptable for some person to prefer someone other than their current spouse; instability occurs only when that person requites.) In the above example, the pairing $[1, 2, 3]$ (man 1 paired to woman 1, man 2 with woman 2, 3 with 3) is stable. The pairing $[3, 2, 1]$ (man 1 paired to woman 3, man 2 with woman 2, 3 with 1) is not stable because man 1 perfers woman 2 to his current spouse (woman 3) and woman 2 prefers man 1 to her current spouse (man 2).

The Stable Marriage Problem is known to be solvable with a simple and predictable algorithm by Gale and Shapley [10]. First, each man proposes to his most-preferred woman. Women who receive multiple proposals maintain only their most-preferred proposal and reject the others. Second, each rejected man proposes to his next-preferred woman; women receiving new proposals continue to maintain only the one most-preferred. The process continues until no man is rejected in a round. Finally, the women accept their most-preferred proposal and the algorithm terminates.

A *reduction* is a method of solving one problem by restating it in terms of another. Reductions can be a powerful but difficult technique for solving hard problems. By reducing the stable marriage problem to a graph search problem, we can solve the problem with an A* algorithm. Our A* solution will be slower and less efficient than the Gale-Shapley algorithm, but it will demonstrate a method for solving difficult problems using general and reusable techniques. The 8-piece puzzle problem could likely also be solved with a very fast and direct algorithm, but it may be very difficult for us to discover this solution. Likewise, there may be an optimal algorithm to solve a Rubik's cube, but given an A* solver and a fast computer we may be able to find an economically-acceptable solution.

Our informed search algorithm is implemented in Julia. This is comparable

our implementation of Dijkstra's algorithm (see section 5.4.3), but instead of searching for a named destination this function instead uses its heuristic function. If the heuristic function returns the value zero, then the search is considered successful and the program terminates. This A* program also prints its search in the DOT graph description language, which can be rendered as a graphic using the GraphViz program.

```
using DataStructures

function informed_search(source, edges::Function, heuristic::Function)
    println("digraph {")

    pq = PriorityQueue()
    visited = Set()
    enqueue!(pq, source=>heuristic(source))

    while !isempty(pq)
        u = dequeue!(pq)
        push!(visited, u)
        println("\"$(u)\" [color=\"blue\"];")

        if heuristic(u) == 0
            println("}")
            return
        end

        for v ∈ edges(u)
            if v ∉ visited && !haskey(pq, v)
                enqueue!(pq, v=>heuristic(v))
                println("\"$(u)\" -> \"$(v)\";")
            end
        end
    end

    error("Failed to find a solution.")
end
```

The heuristic function for the stable marriage function seeks to quantify and differentiate instability by returning the sum of the squared distance (see section 3.1) of a rogue couple's candidate and current preferences.

```
function stability(men::Matrix, women::Matrix, matching)
    n = length(matching)
    wife = matching
    husband = Dict(values(matching) .=> keys(matching))
    metric = 0
```

```
    for man ∈ 1:n
        for woman ∈ 1:n
            # Candidate and current preferences for the man
            x1, x2 = men[man, woman], men[man, wife[man]]
            # Candidate and current preferences for the woman
            y1, y2 = women[woman, man], women[woman, husband[woman]]
            # The matching is unstable if, and only if, both the man
            # and the woman prefer each other to their current matches.
            if x1 < x2 && y1 < y2
                metric += (x1 - x2)^2
                metric += (y1 - y2)^2
            end
        end
    end

    return metric
end
```

The size of our problem comes from the number of possible matchings. If there are n men and n women, then the first man can be matched to n women, the second man can be matched to $n-1$ women, and so on until the last man can only be matched to the only remaining woman. Thus,

$$\text{Problem size} = n \cdot (n-1) \cdot (n-2) \cdot ... \cdot 3 \cdot 2 \cdot 1 = n!$$

A problem of factorial size is a large combinatorial problem. We will not attempt to visit all possible nodes in the problem space. Instead, our edge function will generate three permutations of the current position by switching two matchings.

```
using StatsBase

function e(u)
    v = Set()
    for _ ∈ 1:3
        x = copy(u)
        # Sample, without replacement, two random elements to swap.
        y = sample(collect(eachindex(u)), 2, replace=false)
        x[y[1]], x[y[2]] = x[y[2]], x[y[1]]
        push!(v, x)
    end
    return v
end
```

Our ranking matrices for men, M, and women, W are taken from example 2 from Gale and Shapley [10].

$$M = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 4 & 3 & 2 \\ 2 & 1 & 3 & 4 \\ 4 & 2 & 3 & 1 \end{bmatrix}$$

$$W = \begin{bmatrix} 3 & 4 & 2 & 1 \\ 3 & 1 & 4 & 2 \\ 2 & 3 & 4 & 1 \\ 3 & 2 & 1 & 4 \end{bmatrix}$$

The Julia language has a compact notation to create matrix literals.

```
M = [1 2 3 4; 1 4 3 2; 2 1 3 4; 4 2 3 1]
W = [3 4 2 1; 3 1 4 2; 2 3 4 1; 3 2 1 4]
```

We construct a *closure* to encapsulate M and W with our stability metric function into the heuristic function.

```
h(u) = stability(M, W, u)
```

This syntax declares a unary function `h` that will enable our A* to navigate matrices M and W without direct knowledge of either.

Finally, we invoke the A* informed search algorithm to navigate the Stable Marriage Problem as a graph, starting from pairings $[4, 3, 2, 1]$ (man 1 matched to woman 4, man 2 to woman 3, 3 to 2, and 4 to 1).

```
julia> informed_search([4,3,2,1], e, h)
digraph {
"[4, 3, 2, 1]" [color="blue"];
"[4, 3, 2, 1]" -> "[4, 2, 3, 1]";
"[4, 3, 2, 1]" -> "[4, 1, 2, 3]";
"[4, 3, 2, 1]" -> "[4, 3, 1, 2]";
"[4, 3, 1, 2]" [color="blue"];
"[4, 3, 1, 2]" -> "[4, 2, 1, 3]";
"[4, 3, 1, 2]" -> "[4, 1, 3, 2]";
"[4, 2, 1, 3]" [color="blue"];
"[4, 2, 1, 3]" -> "[2, 4, 1, 3]";
"[4, 2, 1, 3]" -> "[1, 2, 4, 3]";
"[2, 4, 1, 3]" [color="blue"];
"[2, 4, 1, 3]" -> "[2, 1, 4, 3]";
"[2, 4, 1, 3]" -> "[3, 4, 1, 2]";
"[3, 4, 1, 2]" [color="blue"];
}
```

(Note: this *stochastic* algorithm uses randomness in the `sample` operation. Outputs are not determininstic. In rare cases, this procedure may not discover

the one and only solution, $[3, 4, 1, 2]$. See https://github.com/wjholden/Data-Literacy/blob/main/StableMarriageSearch.jl for an expanded version of this program which uses a seeded random number generator for reproducibility.)

We can input this `digraph` data into https://dreampuf.github.io/GraphvizOnline/ to visualize the search tree, as shown in figure 5.3.

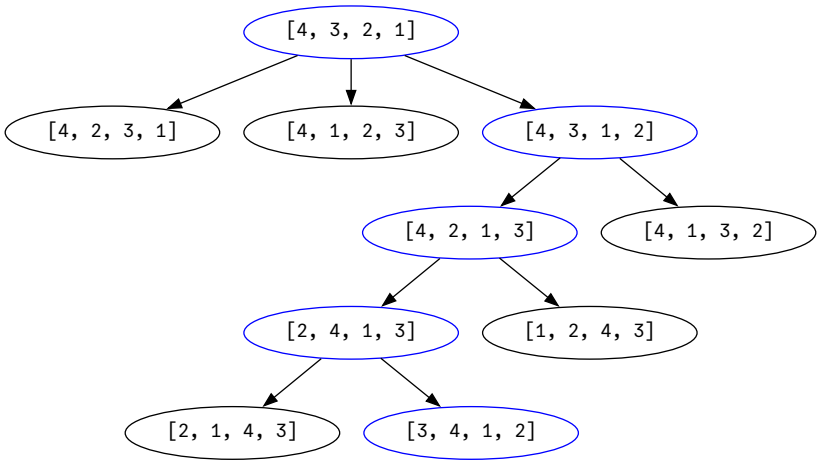

Figure 5.3: A search tree of the Stable Marriage Problem, reduced to an informed search that is solvable with A*.

Again, we applied informed search to the Stable Marriage Problem as an exercise in artificial intelligence methods. Though slow, an informed search can navigate difficult problems with very little information: a simple heuristic function tells A* whether it has gotten closer or farther from the solution. This technique can be useful for solving challenging problems where an optimal solution is not known. Moreover, we can also apply informed search to *intractable* problems where computational complexity forces us to accept approximate solutions as a compromise.

5.5 Discussion prompts

1. A graph can be represented with an adjacency list or a matrix. What are the advantages and disadvantages of each approach?
2. What algorithm can be used to solve the "seven ways to Kevin Bacon" problem?
3. Is a Gantt chart a graph? How can one find the critical path of a project

if represented as a graph?

5.6 Practical exercises

1. Convert currency exchange rates from multiplication to addition using a logarithm, then prove that infinite arbitration is impossible given a set of exchange rates and Bellman-Ford implementation.
2. Define a topological sorting and relate it to a workplace problem.
3. Define the Traveling Salesman Problem (TSP) and explain the computational difficulty of this problem.
4. Determine the minimum paving needed to fully connect a tent complex using a list of coordinates and a Prim or Kruskal implementation.
5. Simulate an infection model in a dense social graph where edge weights represent probability of infection.

References

[1] S. S. Stevens, "On the theory of scales of measurement," *Science*, vol. 103, no. 2684, pp. 677–680, 1946, doi: 10.1126/science.103.2684.677

[2] E. F. Codd, "A relational model of data for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970, doi: 10.1145/362384.362685

[3] A. M. Legendre, *Nouvelles méthodes pour la détermination des orbites des comètes*. Paris, France: Firmin Didot, 1805, pp. 72–75.

[4] A. M. Legendre, "On least squares," in *A source book in mathematics*, New York, NY, USA: McGraw-Hill, 1929, pp. 576–579.

[5] K. PEARSON, "'DAS FEHLERGESETZ UND SEINE VERALLGEMEINER-UNGEN DURCH FECHNER UND PEARSON*.' A REJOINDER," *Biometrika*, vol. 4, no. 1–2, pp. 169–212, Jun. 1905, doi: 10.1093/biomet/4.1-2.169. Available: https://doi.org/10.1093/biomet/4.1-2.169

[6] E. W. Dijkstra, "A note on two problems in connexion with graphs," in *Edsger Wybe Dijkstra: His Life,Work, and Legacy*, 1st ed.New York, NY, USA: Association for Computing Machinery, 2022, pp. 287–290. doi: 10.1145/3544585.3544600

[7] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968, doi: 10.1109/TSSC.1968.300136

[8] D. Schuurmans and F. Southey, "Local search characteristics of incomplete SAT procedures," *Artificial Intelligence*, vol. 132, no. 2, pp. 121–150, 2001, doi: 10.1016/S0004-3702(01)00151-5

[9] Wm. W. Johnson and W. E. Story, "Notes on the "15" puzzle," *American Journal of Mathematics*, vol. 2, no. 4, pp. 397–404, 1879, doi: 10.2307/2369492. Available: http://www.jstor.org/stable/2369492

[10] D. Gale and L. S. Shapley, "College admissions and the stability of marriage," *The American Mathematical Monthly*, vol. 69, no. 1, pp. 9–15, 1962, doi: 10.1080/00029890.1962.11989827