GCIS-124
Software Development & Problem Solving

8: Algorithms

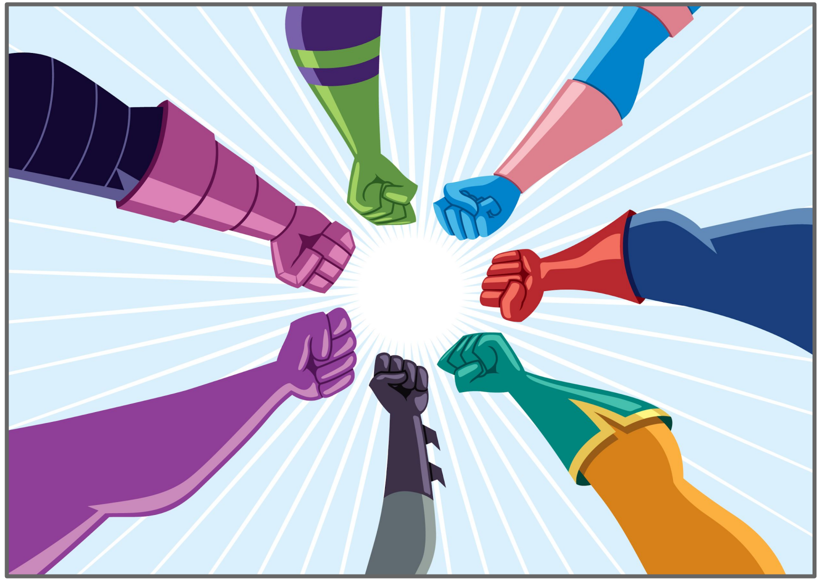
RIT | Golisano College of Computing and Information Sciences

SUN	MON (3/18)	TUE	WED (3/20)	THU	FRI (3/22)	SAT
	Unit 8: Algorithms					
	Unit 7 Mini-Practicum		Assignment 7.2 Due (start of class)		Assignment 8.1 Due (start of class)	
SUN	MON (3/25)	TUE	WED (3/27)	THU	FRI (3/29)	SAT
	Unit 9: Inner Classes, Anonymous Classes, & Lambdas				Project Kick-Off	
	Unit 8 Mini-Practicum Assignment 8.2 Due (start of class)				Project Team Formation Team Problem-Solving Assignment 9.1 Due (start of class)	

You Are Here

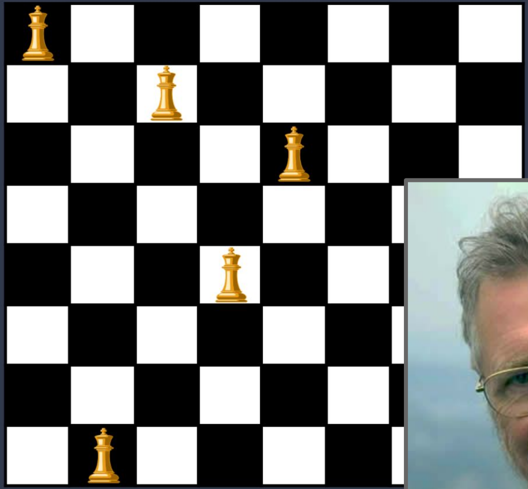
Next Week: Team Project

- Some software tasks are too large and complex for an individual software engineer to complete them alone in a reasonable amount of time.
- It is usually the case that a team of software engineers will work together on large scale projects.
- The team collaborate on all phases of the project.
  - Design
  - Implementation
  - Testing
- Working on a team also presents challenges that you will not generally face when working alone.
  - Including the dreaded ***merge conflict***.
- In support of this, you will be working with 2-3 other students on a team project beginning next week.



Next week we will begin the ***team project***. Your instructor will provide details about how your teams will be formed.

Overview of the Week



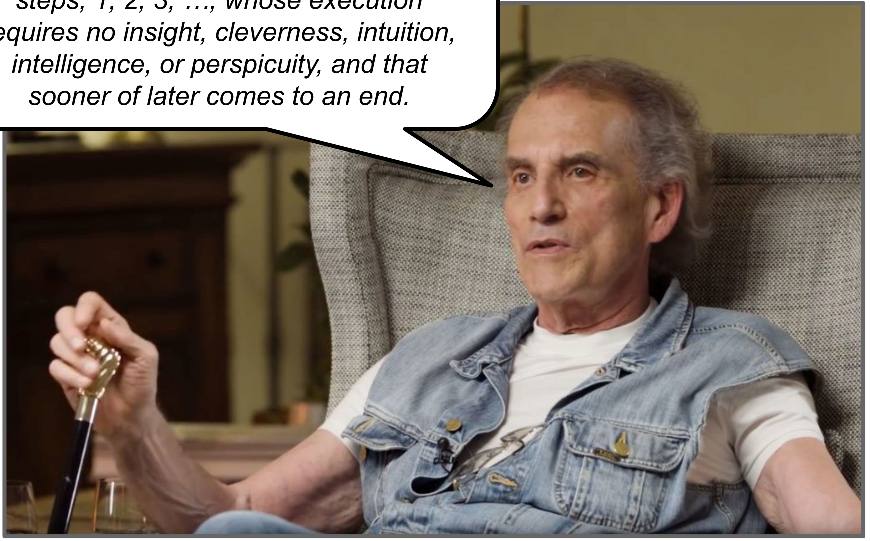
We will try to answer questions like *"Can I fit **8 queens** on an 8x8 chessboard?"* and *"What is the **lowest cost path** between two vertices in a weighted graph?"*

- We have spent the better part of two semesters implementing a variety of ***algorithms***. In this unit we will spend more time exploring exactly what an algorithm is.
  - How to apply an ***algorithmic approach*** to problem solving.
  - How to capture algorithms quickly using ***pseudocode***.
  - A few very ***well known algorithms*** in computer science.
- Specifically, we will explore:
  - Algorithms & Pseudocode
  - Algorithmic Problem Solving
  - Greedy Algorithms
    - Change Making
    - Nearest Neighbor
  - Dijkstra's Algorithm
  - Backtracking
- Today we will focus on an ***algorithmic approach*** problem solving, describing algorithms using ***pseudocode***, and ***greedy algorithms***.

Algorithms

- When programming in a high-level language like Python or Java, the programmer designs an ***algorithm***.
  - But what does this mean, exactly?
- An algorithm is like a ***recipe*** with instructions that tell a computer how to solve a problem.
  - First, break the problem into discrete ***steps***.
  - Implement each step as an ***instruction*** in a programming language.
  - ***Execute*** the program.
- We often begin by writing an algorithm in ***pseudocode***.
  - We use plain English (or something like it) to describe what it is that we'd like to do.
  - We then ***translate*** that pseudocode into the syntax of our language of choice.

An algorithm is a finite procedure, written in a fixed symbolic vocabulary, governed by precise instructions, moving in discrete steps, 1, 2, 3, …, whose execution requires no insight, cleverness, intuition, intelligence, or perspicuity, and that sooner of later comes to an end.



Believe it or not, this is *not* the creepy killer from Silence of the Lambs, but is actually David Berlinski, the person that wrote *The Advent of the Algorithm*.

An Algorithmic Approach to Problem Solving

- An ***algorithmic approach*** to problem solving begins by breaking the problem that we are trying to solve down into a ***series of discrete steps***.
  - If these steps are executed in the correct order, we will arrive at the solution to the problem at hand.
- Ultimately, we would like to capture these steps in a programming language like Python or Java.
- But before we start dealing with the rigid syntax of a real programming language, it is often useful to first try to express the algorithm that we wish to write in ***pseudocode***.
  - ***Pseudocode*** is a term for very high-level descriptions of what we would like the computer to do.
  - The description should be ***detailed***, but is not concerned with the syntax of any particular language.
  - Most pseudocode is very close to written or spoken language, and ***there are no rules*** of grammar or syntax.
- The purpose of using pseudocode to describe algorithms is to ***sketch out an algorithm quickly***, without spending time dealing with syntax errors or heavy weight tools like an IDE.

There are ***no rules*** to writing ***pseudocode***. The point is to avoid spending time on strict rules of syntax in order to express the algorithm as ***quickly as possible***.

Euclid's Method for finding the Greatest Common Divisor of two integers A and B:

```
1.  As long as A is not equal to B:
      a.  if A is greater than B:
            i.  Let A = A - B
      b.  otherwise
            i.  Let B = B - A
2.  Return A
```

Let's start by translating the Euclidean Algorithm for determining the GCD of two integers into Java code.

8.1 Euclidean Pseudocode

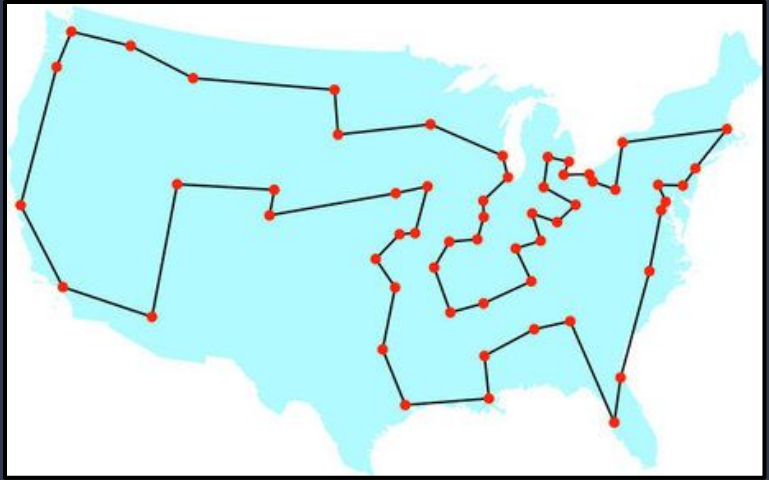
An algorithm is like a recipe that lists the ingredients needed and the steps required to solve a specific problem. We often sketch an algorithm out by first writing *pseudocode* that describes that we'd like the program to do. Next, the pseudocode needs to be translated into statements in a real programming language. Practice now by implementing Euclid's method for finding the greatest common divisor of two integers.

```
Euclid's Method for finding the Greatest Common
Divisor of two integers A and B:
  1.  As long as A is not equal to B:
        a.  if A is greater than B:
              i.  Let A = A - B
        b.  otherwise
              i.  Let B = B - A
  2.  Return A
```

- Create a new Java class named "`Euclid`" and implement the algorithm depicted to the left as a `static` method.
    - Begin by stubbing out the `gcd` method. It should declare parameters for A and B and return an integer.
    - Next, capture each of the steps as an ***inline comment*** in the method. You must ***commit*** this version of your code before moving any further!
    - ***Implement*** each of your comments as a Java statement.
- Define a `main` method with the appropriate signature and test your `gcd` method with several different combinations of A and B.

Greedy Algorithms

- A ***greedy algorithm*** is one that makes a decision based on what appears to be the optimal choice at the time.
  - "The optimal choice at the time" is another way of saying "the ***local optimum***" - what ***appears*** to be the best answer for that particular step in the problem given the information that we have.
- Greedy algorithms sometimes produce a result that is the ***global optimum***.
  - The best possible answer for the problem.
- But other times, greedy algorithms relatively ***efficiently*** produce a result that is ***reasonably good*** in a relatively short amount of time.
  - There are problems for which the optimal answer takes ***far too long*** to compute.
  - In these cases, a greedy algorithm may produce an answer that is "***close enough***" in a much shorter amount of time.
- Greedy algorithms can be a faster alternative to a more heavyweight algorithm ***if*** they produce a result that is good enough.



The ***traveling salesman problem*** is a classic example of an algorithm that ***cannot*** be solved optimally in a reasonable amount of time.

We will be looking at the traveling salesman problem presently, but first let's take a look at another classic example: ***making change***.

Making Change

Remember, a ***greedy algorithm*** makes the best possible choice at each step with whatever limited information is available.

The algorithm that chooses the ***largest*** denomination that is ***less than or equal to*** the amount due is an example of a greedy algorithm.

Does it also find the ***optimal*** answer?

- Consider a typical transaction at the grocery store.
  - A customer approaches a cashier with a cart full of goods.
  - The cashier scans the merchandise and presents the customer with the total price for all of the goods.
  - The customer hands over a payment in the form of an amount of cash greater than or equal to the total price.
  - The cashier gives any change due back to the customer.
- There are potentially many algorithms that the cashier might use to determine which coins and bills to use
  - Try every possible permutation of available coins and bills.
  - Choose the largest coin or bill that is less than or equal to the amount due, and subtract it from the total. Repeat.
  - Use the closest whole dollar amount.
  - Make change only in pennies.
  - etc.
- An ***optimal*** solution will meet a few simple goals:
  - Give the customer the ***correct change***.
  - Use the ***smallest number*** of bills and coins necessary.
  - Determine the correct answer ***quickly***.
- Which algorithm(s) if any meet these goals?

8.2 Making Change

Now it's your turn to practice writing pseudocode. Sketch out a greedy change-making algorithm using comments before you try to implement it in the next activity.

The ***greedy*** change making algorithm always picks the biggest denomination that will "fit" and subtracts it from the total.

- Take a few moments to examine the `Currency` `enum` that has been provided to you.
  - Your algorithm should return a `List` ***of*** `Currency` that sums to the correct amount of change.
  - For example, if `$2.26` was the change due back to the customer, your algorithm should return `[DOLLAR, DOLLAR, QUARTER, PENNY]`
- Create a new Java class named "`Greedy`" and define a method called "`makeChange`" that declares `double` parameters for `price` and `payment`. It should return a `List` ***of*** `Currency`.
  - The `price` represents the total cost of the items the customer wants to purchase, and the `payment` indicates how much money they are paying.
  - Using ***inline comments***, sketch out a ***greedy algorithm*** for computing the correct change to return to the customer in pseudocode.
  - When you have finished outlining your algorithm, ***commit*** your code.

8.3 Implementing Change

Once you are reasonably certain that you have sketched out a sufficiently detailed change-making algorithm, go ahead and try to implement it in Java. Translate each of your pseudocode comments into one or more Java statements. Does it work?

Thank you for shopping at Crossroads. Your total is $1,347.86.

- Open the `Greedy` class and navigate to the `makeChange` method.
  - Implement each of the steps in your algorithm using Java statements.
  - You should ultimately return a `List` of `Currency`.
- Define a `main` method with the appropriate signature.
  - ***Test*** the `makeChange` method with several combinations of prices and payments.
  - ***Print*** the results of each test to standard output.

The ***greedy*** change making algorithm always picks the biggest denomination that will "fit" and subtracts it from the total.

The Knapsack Problem

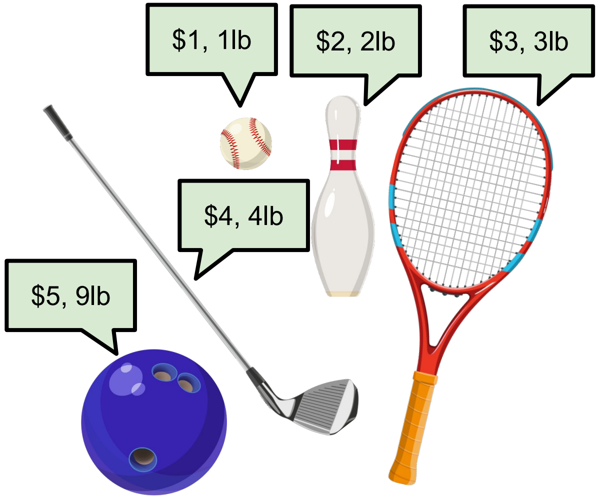
- The greedy change-making algorithm makes a ***locally optimal*** decision with each iteration:
  - "What is the largest bill or coin that is less than the current amount due?"
- Such decisions are ***quick and easy*** to make, but are based on ***limited information***.
  - No memory of what has come before.
  - No idea what is coming next.
- ***Sometimes*** making the locally optimal decision results in a solution that is also ***globally optimal***.
  - This is the case with the change-making algorithm, which quickly comes up with the best possible answer.
  - But this isn't always the case!
- Consider the ***Knapsack Problem***.
  - A ***knapsack*** has a ***maximum weight capacity*** that it can hold.
  - Each ***item*** has a ***weight*** and a ***value***.
  - What is the best way to ***maximize*** the ***value*** of the items in the knapsack without going over its ***weight limit***?

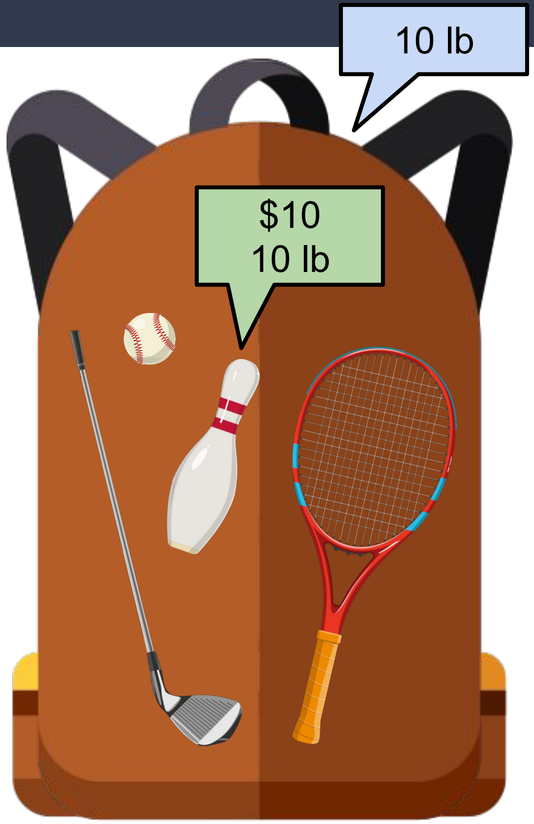
There are several possible greedy algorithms for choosing which items should be packed into the knapsack. Do any find an optimal solution?

The Illustrated Knapsack Problem

A ***knapsack*** has a ***maximum weight capacity*** that it can carry.

And each ***item*** has a ***value*** and a ***weight***.





The ***goal*** of the ***Knapsack Problem*** is to pack the combination of items that ***maximizes the total value*** without exceeding the capacity.

The ***optimal solution*** requires trying every possible combination of items to see which has the highest value; this is ***computationally expensive***.

A greedy algorithm ***quickly*** chooses the next item based on local optima like ***lowest weight*** or ***highest value***.

But will the greedy knapsack-packing algorithm ***guarantee*** the optimal result?

8.4 Knapsacks and Items

A few classes have been provided to you to help you implement a greedy knapsack-packing algorithm. Examine the classes and then think about your algorithm. What do you think will work best?

- Take a few minutes to examine the contents of the `knapsack` package that has been provided to you:
  - `Item` - a class that represents an item including its name, weight, and value. Items are naturally comparable by weight from lightest to heaviest.
  - `ValueComparator` - a comparator that arranges items from highest to lowest value. In the event that two items have the same value, they are compared by weight.
  - `Knapsack` - implements a knapsack. An item can be packed as long as its weight does not exceed the maximum capacity. The knapsack keeps track of items packed, current load, and total value.
  - `ItemSets` - provides helper methods to return pre-made sets of items that can be used for testing a knapsack-packing algorithm.
- Use this time to ask any questions that you have about the code.
- When you are finished, start thinking about the greedy algorithm that you want to write to pack a knapsack.
  - How will you choose the next item to pack?
  - Will you need to write your own `Comparator`?
  - How easy do you think it will be to foil your algorithm?

8.5 Packing the Knapsack

Using only pseudocode, sketch out your algorithm. Try to write one descriptive sentence or phrase for each statement you think that you will need to write.

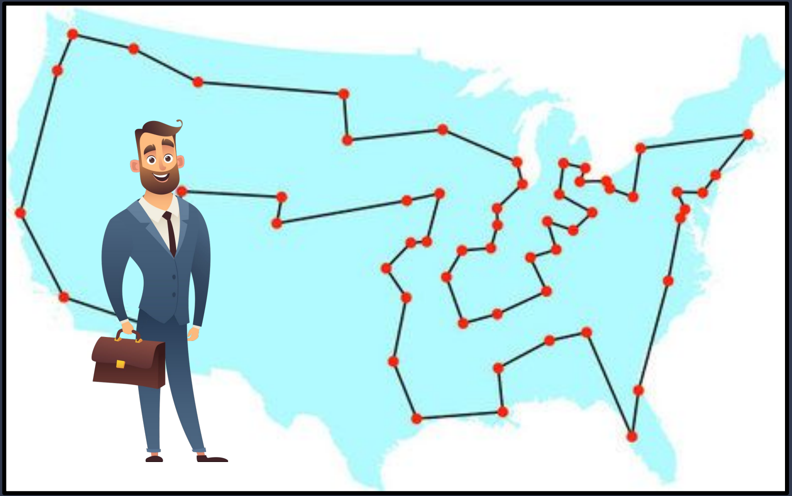
- Open the `Greedy` class and define a new method named "`packItems`" that declares parameters for a `Knapsack` and a `List` of `items`.
    - Using only ***inline comments***, sketch out your plan for a greedy algorithm to solve the Knapsack-Packing algorithm using ***pseudocode***.
    - What ***local optima*** will your algorithm consider as it chooses the next item to try to pack?
    - You should ***remove*** items from the list as they are packed (or discarded).
    - How will your algorithm know when to ***stop***?
- When you have finished your pseudocode algorithm ***commit*** your code to your repository.

8.6 Knacking the Packsack

Now it's time to put your algorithm to the test! Implement the greedy algorithm that you sketched out previously. How does it compare to the optimal solution for each item set?

- Open the `Greedy` class and implement your knapsack-packing algorithm in the `packItems` method.
  - Remember to ***remove*** items from the list as they are packed.
  - The `Knapsack` will not allow an `Item` to be packed if its weight exceeds the capacity.
  - ***Print*** the Knapsack once it is packed.
- Navigate to `main` and create a Knapsack with a capacity of 10 pounds. Test your knapsack-packing algorithm using the premade item sets created by the `ItemSets` class.
  - The Sports Set
  - The Electronics Set
  - The Toy Set
  - The Metal Set
- How does your algorithm compare to the optimal solution for each?

The Traveling Salesman Problem



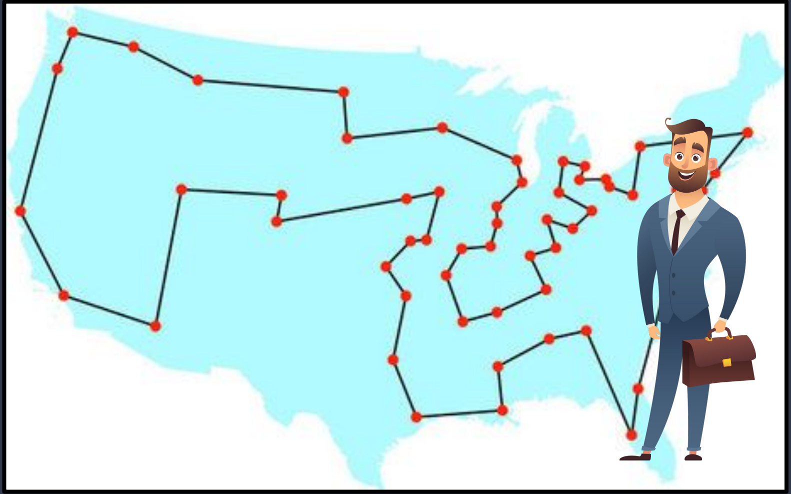
Before we can begin implementing an algorithm to find a solution to the ***Traveling Salesman Problem***, we will need to find a way to represent the 50 cities and the distance between them.

Would it be possible to use a ***graph***?

- A ***traveling salesman*** would like to travel to ***50 different cities*** in the contiguous United States.
- All 50 cities are ***connected to each other*** by the interstate highway system.
    - Meaning that the salesman can travel ***from*** any one city ***to*** any other.
- The salesman would like to ***plan a trip*** with a few goals in mind:
    - He would like to visit each of the 50 cities ***exactly once***.
    - In order to save on time, money, and gas, he would like to ***minimize*** the total distance traveled.
- The ***optimal*** solution to this problem would require mapping every possible route through all 50 cities.
    - There are ***50!*** possible routes!
- A ***greedy algorithm*** would attempt to plan a trip based on some local optima such as which cities are ***closest*** to each other or in the ***same general direction***.

- We could make a ***graph*** to represent the 50 cities.
    - Each city would be a ***vertex*** in the graph.
    - Each city would also be ***connected*** to the other 49 cities as neighbors.
- We could then run a BFS or DFS search to find a path through all 50 cities.
- The problem is that our graphs do not have any concept of ***cost*** or ***distance*** between two cities.
    - There is no way to tell the difference between a neighbor city that is 10 miles away, and one that is 3,000 miles away!
    - Meaning the BFS or DFS will be very unlikely to minimize the distance traveled.
- We will need a modified version of our graph data structure that assigns values to the edges between vertices.
    - We will refer to these values as ***weights***.
- Our ***weighted graph*** will also need modified versions of our search algorithms that take edge weights into consideration.

A Weighted Graph



In this problem, the number of vertices in the path ***does not change***: it will always be ***exactly 50***.

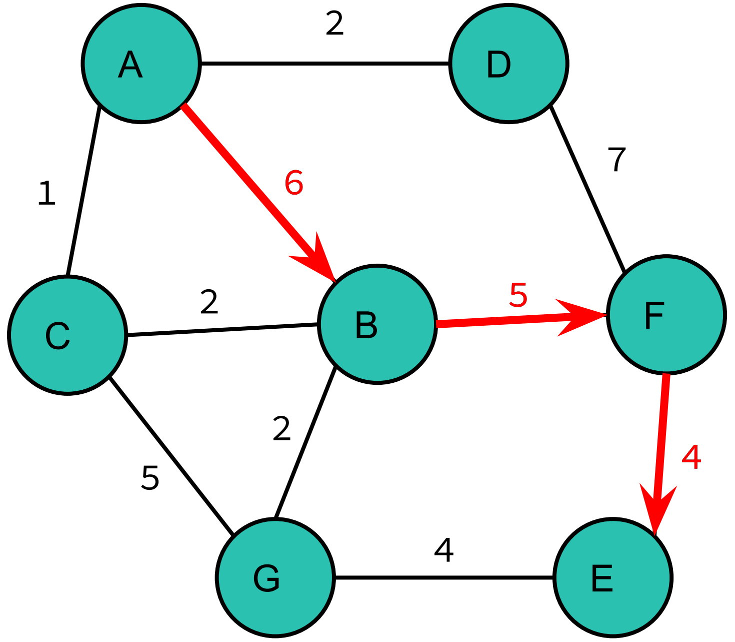
What ***will*** change (hopefully) are the specific edges along the path such that the ***total distance traveled*** is ***minimized***.

A Weighted Graph

The search algorithms that we have looked at so far do not differentiate between neighbors...

If ***BFS*** were looking for a path from ***A to E***, it would find one with the ***fewest edges***, for example...

But the ***total cost*** of this path is ***6 + 5 + 4 = 15***.



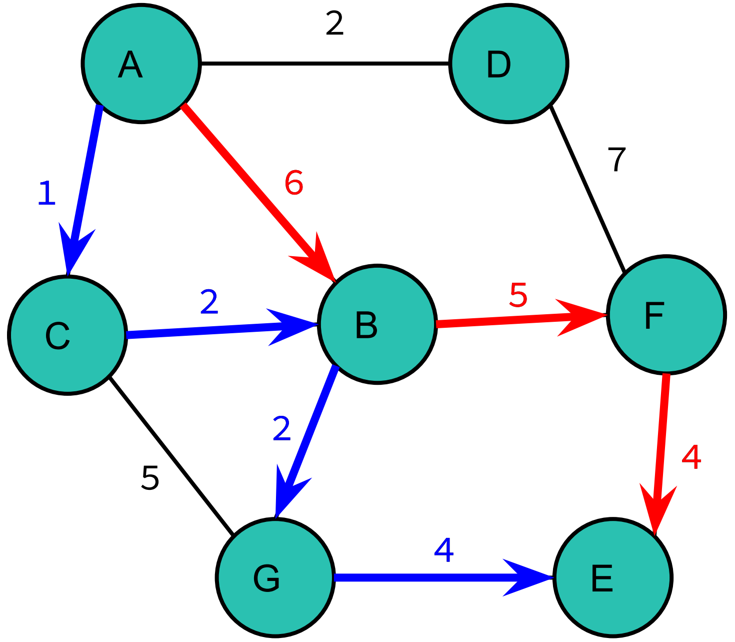
A Weighted Graph

But there is a ***lower cost*** path from ***A to E*** in this graph.

The Path ***A-C-B-G-E*** comprises ***four edges*** but has a total cost of only ***1 + 2 + 2 + 4 = 9***.

BFS would ***never*** find this path because it ***always*** finds paths with the fewest edges.

DFS ***might*** find it, but if it did, it would be a total coincidence. Unless...



8.7 A Weighted Graph

The Graph abstract data type that we have used previously keeps track of which vertices are connected as *neighbors*, but it doesn't assign a weight to those connections. We'll need a new Graph data structure that stores edges with weights.

<<Interface>>
WGraph<E>

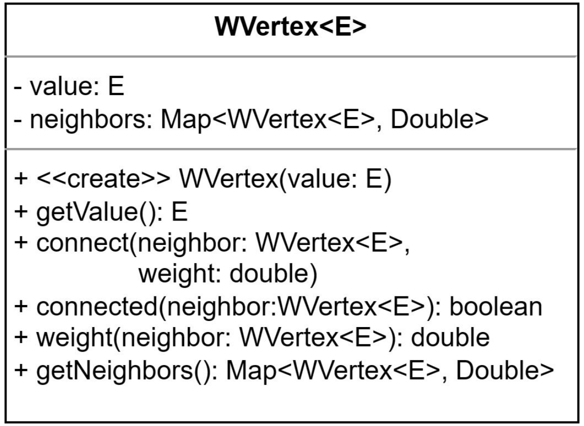
+ add(value: E)
+ contains(value: E): boolean
+ size(): int
+ connect(a: E, b: E, weight: double)
+ connected(a: E, b: E): boolean
+ weight(a: E, b: E): double

When two vertices are connected by an edge in a ***weighted graph***, a ***weight*** is associated with the edge.

- The `WGraph` interface has been provided to you in the `weighted` package.
    - Take a moment to review the code and the methods that it provides. Do you understand the purpose of each one?
    - If you have any questions, ask your instructor!

8.8 A Weighted Vertex

Before we can start implementing the *weighted adjacency graph*, we'll need a new vertex class that keeps track of the weight of the edges to its neighbors. This will be very similar to the `Vertex` class from the graphs unit, but it will use a `Map` to store neighbors and weights.

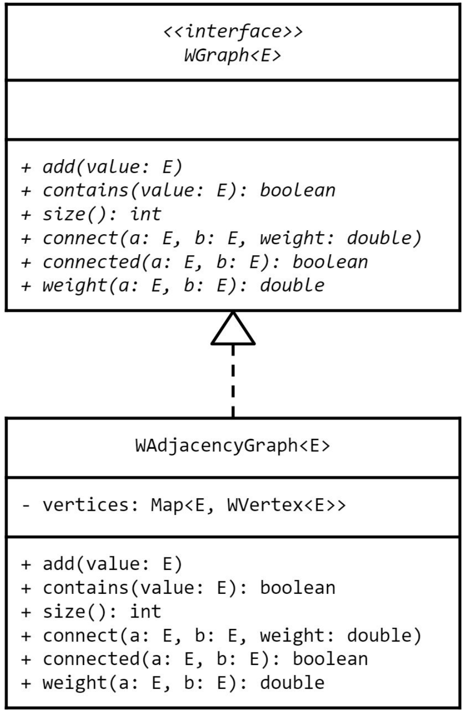


```
WVertex<E>
- value: E
- neighbors: Map<WVertex<E>, Double>
+ <<create>> WVertex(value: E)
+ getValue(): E
+ connect(neighbor: WVertex<E>,
              weight: double)
+ connected(neighbor:WVertex<E>): boolean
+ weight(neighbor: WVertex<E>): double
+ getNeighbors(): Map<WVertex<E>, Double>
```

- Create a new Java class named "`WVertex`" in the `weighted` package.
  - Use the UML to the left as a guide when implementing your `WVertex` class.
  - The ***constructor*** should create an empty map with ***neighbors*** as keys and ***weights*** (doubles) as values.
    - What kind of map should you use?
  - When ***connecting*** to a neighbor, put the neighbor and its weight in the map.
  - Implement a `toString()` that returns the ***vertex's*** `value` as a string.
- Use the provided `WVertexTest` to verify that the weighted vertex is working as expected.
  - ***Raise your hand*** if your tests are not passing.

8.9 A Weighted Adjacency Graph

Now it's finally time to work with a weighted graph! The implementation is very close to the `AdjacencyGraph`, so the implementation has been provided to you. Spend a few moments looking through the code with your instructor.

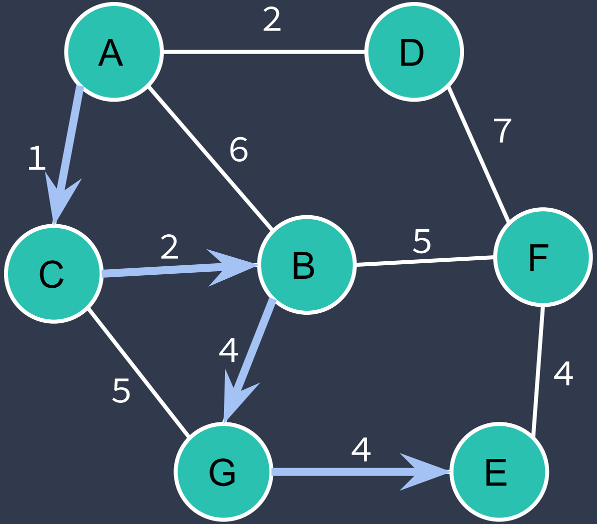


<<interface>>
WGraph<E>
+ add(value: E)
+ contains(value: E): boolean
+ size(): int
+ connect(a: E, b: E, weight: double)
+ connected(a: E, b: E): boolean
+ weight(a: E, b: E): double
WAdjacencyGraph<E>
- vertices: Map<E, WVertex<E>>
+ add(value: E)
+ contains(value: E): boolean
+ size(): int
+ connect(a: E, b: E, weight: double)
+ connected(a: E, b: E): boolean
+ weight(a: E, b: E): double

- Rename the "`WAdjacencyGraph.txt`" file that has been provided to you in the `weighted` package so that it has a `.java` extension.
  - Together with your instructor, examine the code and ask any questions that you have.
- Use the provided `WAdjacencyGraphTest` to verify that the graph working as intended with the new `WVertex` class.
  - ***Raise your hand*** if your tests are not passing!

Nearest Neighbor

The ***Nearest Neighbor*** algorithm is a slight modification of a ***depth-first search***; it always chooses the neighbor connected by the edge with the ***lowest weight***.

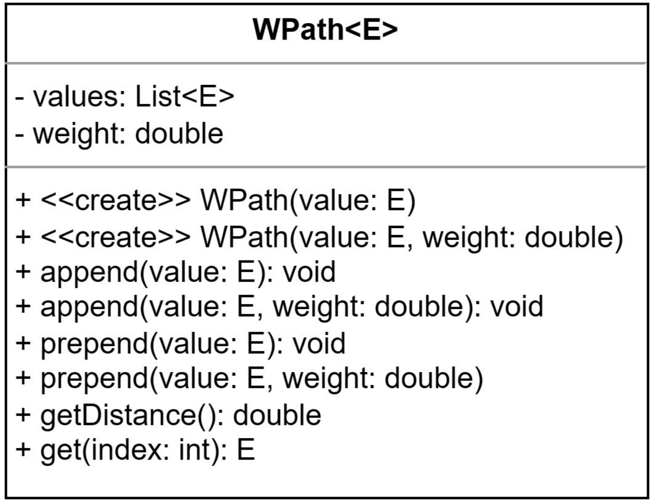


But will it always find the path with the lowest cost?

- As was previously mentioned, the optimal solution for the ***Traveling Salesman*** problem requires an exhaustive search of all possible paths through the graph.
  - This would take a while as there are are 50! possible paths.
- A ***greedy algorithm*** might not find the optimal path, but it should find one that is reasonably good in a much shorter amount of time.
- One such algorithm is the ***Nearest Neighbor*** algorithm.
  - Choose a starting point in the graph, e.g. the city closest to the salesman's current location.
  - Each time the salesman chooses a new city to travel to, he chooses the ***closest city that he has not yet visited***.
  - This continues until all 50 cities have been visited.
- Each time a new city is chosen, the algorithm heads ***deeper*** into the graph without looking at any other neighbors.
  - Does this sound familiar?

8.10 A Weighted Path

The *Nearest Neighbor* algorithm will not only need to return the *values* in a path, but the total distance (the sum of the weights of all of the edges on the path) as well. The `WPath` class has been provided to you to fulfill this function.



```
WPath<E>

- values: List<E>
- weight: double

+ <<create>> WPath(value: E)
+ <<create>> WPath(value: E, weight: double)
+ append(value: E): void
+ append(value: E, weight: double): void
+ prepend(value: E): void
+ prepend(value: E, weight: double)
+ getDistance(): double
+ get(index: int): E
```

- Open the provided `WPath` class in the `weighted` package.
  - ***Examine*** each of the methods to gain an understanding of what it does.
  - Notice that there are ***overloaded*** constructors that can be used to create a path with or without an initial weight.
  - Notice also that values can be ***added to either end*** of the path with or without a weight.
  - If you have any questions, ***ask***!

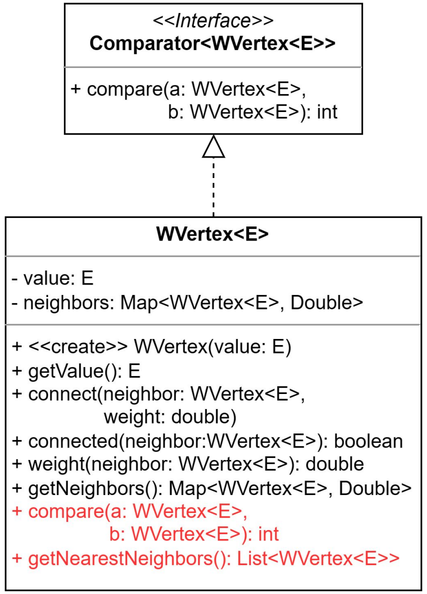
Sorting Neighbors
(Information Expert)

- In order to properly implement the ***Nearest Neighbor*** algorithm in a weighted graph, we will need to be able to iterate through each vertex's neighbors in order based on distance from shortest to longest.
- This means that we will need to implement a method that creates a ***sorted list of neighbors***.
    - Where should this method be implemented?
- ***Information Expert*** is an object-oriented design principle that is also sometimes called ***"behaviors follow state."***
    - This means that, if you need to add a new method (behavior) to existing code, you should add it to the class that already contains the necessary data (state).
- Which class in the system already contains all of the information necessary to sort a vertex's neighbors based on their distance from the vertex?

Information Expert is one of the ***GRASP*** design principles that were first cataloged by Craig Larman, one of the pioneers of the software engineering discipline.

8.11 Sorting Neighbors

In order to implement *Nearest Neighbors*, we'll need to be able to create a sorted list of neighbors based on their distance from a vertex from shortest to farthest.



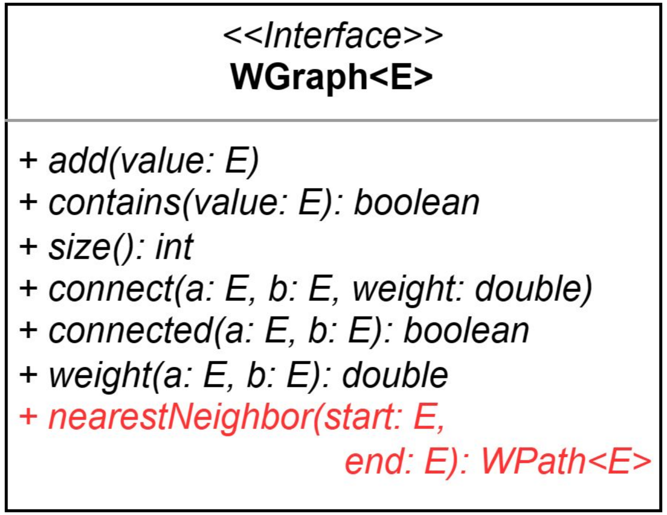
<<Interface>>
Comparator<WVertex<E>>
+ compare(a: WVertex<E>, b: WVertex<E>): int
WVertex<E>
- value: E
- neighbors: Map<WVertex<E>, Double>
+ <<create>> WVertex(value: E)
+ getValue(): E
+ connect(neighbor: WVertex<E>, weight: double)
+ connected(neighbor:WVertex<E>): boolean
+ weight(neighbor: WVertex<E>): double
+ getNeighbors(): Map<WVertex<E>, Double>
+ compare(a: WVertex<E>, b: WVertex<E>): int
+ getNearestNeighbors(): List<WVertex<E>>

The ***information expert*** principle states that we should add new methods to an existing class that has the needed data.

- The `WVertex` will need to compare its neighbors based on their distance (the weight of the edge). Open the class and update it so that it `implements Comparator`
  - The `compare` method should compare two neighbors, `a` and `b`, based on their distance from the vertex.
  - *Hint*: use the map of `vertices` to look up the weight of each vertex and then return `-1` if `a < b`, and `+1` otherwise.
- Next, define a new method named "`getNearestNeighbors`" that creates a list of neighbors and sorts it before returning it.
  - Use the `neighbors.keySet()` method to copy the neighbors into a list.
  - Use `Collections.sort(list, this)` to use the vertex as the comparator that helps sort its neighbors in the list.
- Add a new test method to the `WVertex` class that you can use to test the new method.
  - In the *setup* add at least *three* neighbors with different weights.

8.12 Update `WGraph`

We need to add a new method to the WGraph interface. As we have done in the past when adding new methods to an existing interface, we will provide a default implementation that throws an exception.

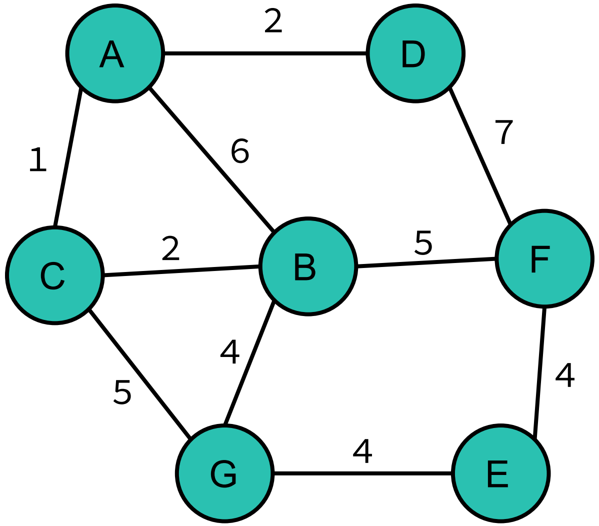


```
<<Interface>>
WGraph<E>
+ add(value: E)
+ contains(value: E): boolean
+ size(): int
+ connect(a: E, b: E, weight: double)
+ connected(a: E, b: E): boolean
+ weight(a: E, b: E): double
+ nearestNeighbor(start: E,
                  end: E): WPath<E>
```

- Open the `WGraph` interface and define a new, default method named "`nearestNeighbor`" that declares parameters for `start` and `end` values and returns a `WPath`.
    - The default implementation should throw an `UnsupportedOperationException`.

8.13 Nearest Neighbor, Part 1

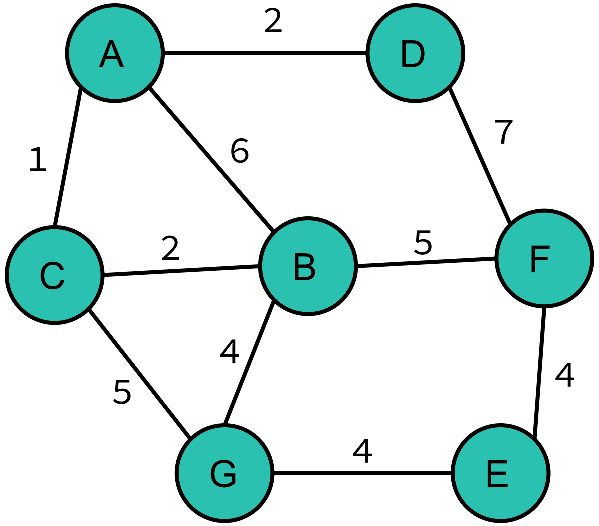
Nearest Neighbor is a greedy algorithm that attempts to quickly find a path through a graph that has a relatively low cost. It does this by choosing neighbors based on their distance from the current vertex. Begin Implementing it in your weighted graph now!



- The ***Nearest Neighbor*** algorithm is a modified DFS. Begin by ***opening*** the provided `AdjacencyGraph` class and ***copying*** the `visitDFPath` method.
- Open the `WAdjacencyGraph` class and paste the method at the bottom of the class.
  - Rename it to "`visitNearestNeighbor`".
  - Make sure to return a `WPath` instead of a `List`!
  - Modify the `for` loop so that it iterates over the ***nearest neighbors***.
  - If a non-`null` path is returned, `prepend` the neighbor and its weight onto the path!
- Make sure that there are no PROBLEMS in your code.

8.14 Nearest Neighbor, Part 2

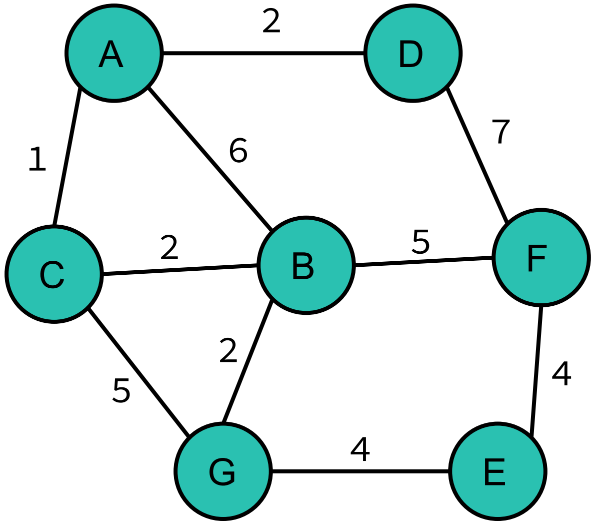
We're not done implementing *Nearest Neighbor* yet! Finish implementing and testing the method in your weighted graph now.



- ***Open*** the provided `AdjacencyGraph` class and ***copy*** the `dfPath` method.
- Open the `WAdjacencyGraph` class and paste the method at the bottom of the class.
    - Rename the method from "`dfPath`" to "`nearestNeighbor`". You will also need to update the code to use a `WPath` instead of a list.
- Rename the provided `GraphMaker` program and use it together with the provided `NearestNeighborTest` to test your algorithm.

Review: Graphs & Searches

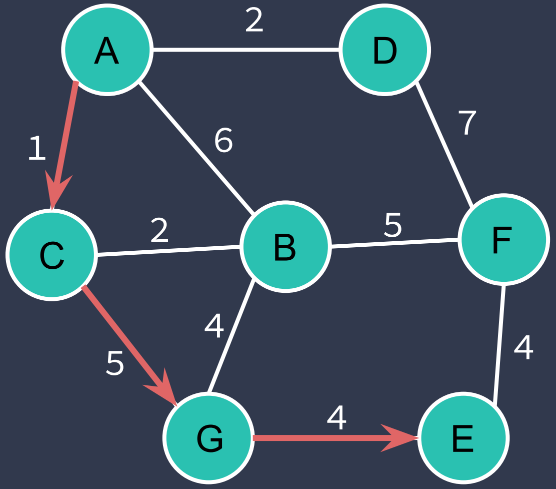
- A ***graph*** comprises ***vertices***, each of which contains a ***value*** of some kind.
- Two vertices can be connected by an ***edge***.
  - Edges may be ***directed*** or ***undirected***.
  - An edge may or may not have a ***weight*** that indicates the ***cost*** of traversing the edge.
  - Two vertices connected by an edge are said to be ***neighbors***.
- A ***path*** is the series of edges that connect two vertices together.
- A ***search*** is an algorithm that determines whether or not a path exists between two vertices.
- Some search algorithms consider the weight of the edge between vertices.
  - ***Breadth-First Search*** and ***Depth-First Search*** ignore the cost of edges.
  - ***Nearest Neighbor*** always chooses the ***unexplored*** edge with the ***lowest cost***.



Different searches will find different paths through a graph. Some look for ***fewer edges*** while others look for ***lowest cost***.

What does "*Shortest*" Mean?

- A ***breadth-first search*** will always find a path that has the fewest possible ***edges***.
  - There may be more than one path with the same "shortest" length.
  - The BFS will find one based on the order in which it visits the neighbors of each vertex.
- A ***depth-first search*** will only find a path with the fewest edges by coincidence.
  - Again, this will be determined by the order in which the search visits the neighbors of each vertex.
  - DFS is just as likely to find the path with the ***most*** edges.
- Neither algorithm is guaranteed (or even likely) to find the path with the lowest total ***cost***.
  - We also refer to total cost as ***total distance*** and the lowest cost as the ***shortest distance***.
  - If the shortest distance path has more edges, BFS will ***never*** find it.
  - DFS may find it, but only by coincidence.
- ***Nearest Neighbor*** is a greedy algorithm chooses the unexplored neighbor with the lowest cost, but this is ***not*** guaranteed to find the lowest cost path.

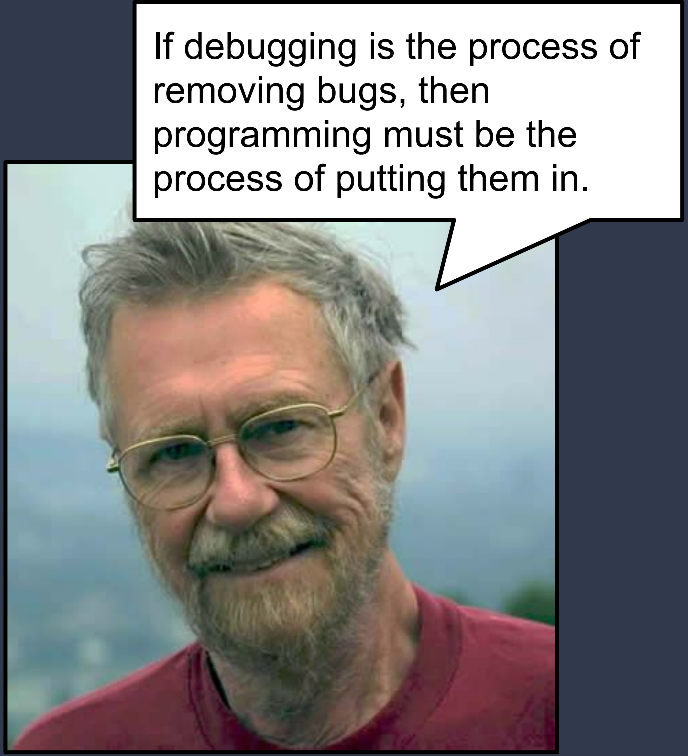


The definition of ***"shortest"*** depends on the problem that you are trying to solve. Is it the ***fewest hops*** or ***lowest cost***?

Today we will focus on finding the path with the ***lowest cost*** even if it means that it includes ***more edges***.

Dijkstra's Algorithm

- An ***optimal*** path ***P*** between two vertices ***S*** and ***E*** comprises edges that may be numbered from ***0 to N***.
- Let ***P'*** be the ***subpath*** of ***P*** that comprises the edges number ***0 to N-1*** that ends at some vertex ***E'***.
    - That is the subpath that contains all but the last edge of ***P*** ending at the last vertex before ***E***.
- If ***P*** is ***optimal***, then it must be true that ***P'*** is ***also optimal***.
    - That is to say that there ***cannot*** be some other path from ***S*** to ***E'*** that is shorter than ***P'***.
    - If there were such a shorter path, then ***P*** could not be optimal.
- It must also be true that the path ***P"*** that comprises the edges ***0 to N-2*** and ends at vertex ***E"*** is also optimal.
    - And so on.
- OK, so what?
- If we can methodically work through the graph and keep track of the shortest path from ***S*** to ***every other vertex***, we can find ***all*** of the shortest paths!


If debugging is the process of removing bugs, then programming must be the process of putting them in.

His last name is pronounced "Dike-stra."

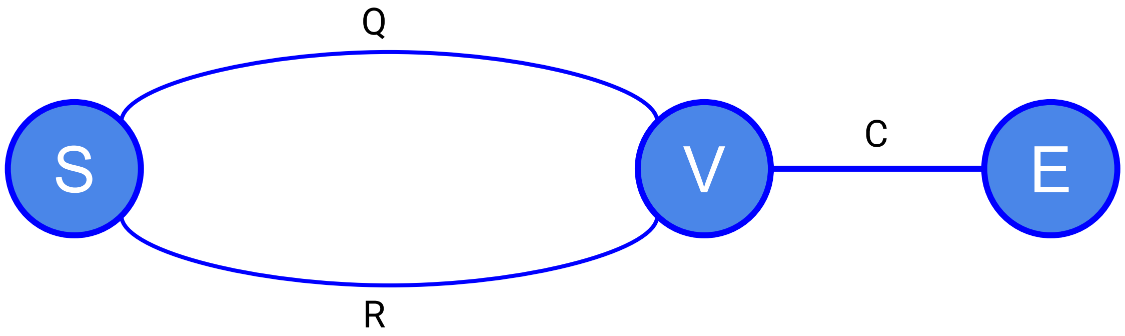
Dijkstra in a Nutshell

Assume that you are trying to find the ***optimal path*** from ***S*** to ***E*** in a weighted graph.

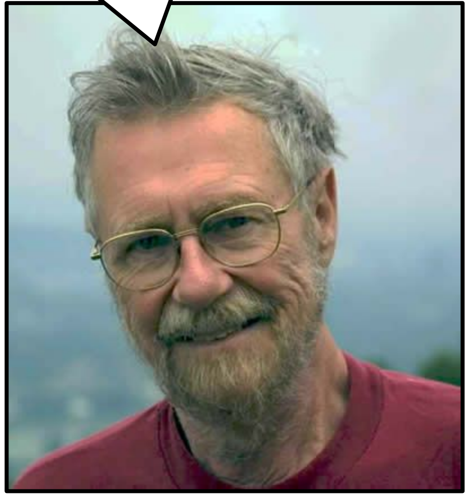
Now assume that there are ***exactly two*** possible paths from ***S*** to ***V*** the ***total distance*** of which are ***Q*** and ***R***.

The second to last vertex on that path is ***V*** and the ***distance*** between ***V*** and ***E*** is ***C***.

If ***Q < R***, then ***Q+C < R+C***. Therefore, the ***optimal path*** from ***S to E*** must include ***Q*** and ***not R***.



Therefore, we can find the ***optimal path*** by finding the ***optimal subpaths*** from ***S*** to every vertex that is along the path to ***E***.



An Illustrated Example

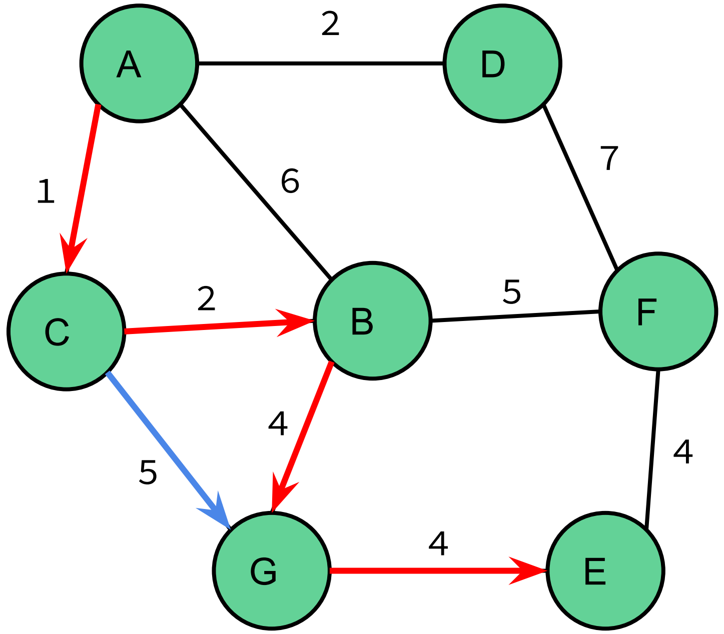
Let's represent a path from ***X*** to ***Z*** in the form X_dY_dZ where ***d*** is the distance between each pair of vertices.

When searching for a path from ***A*** to ***E***, ***Nearest Neighbor*** found the path $A_1C_2B_4G_4E$ with a total cost of ***11***.

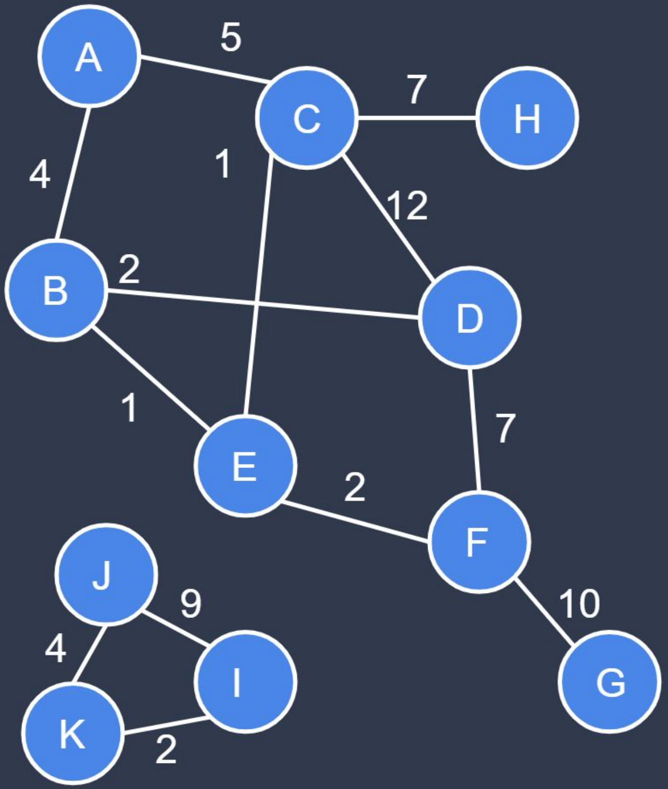
Dijkstra observed that if $A_1C_2B_4G_4E$ is optimal, then all of its ***subpaths*** must also be optimal. Which means that the subpath $A_1C_2B_4G$ must be the optimal path from ***A*** to ***G***. Is that true?

It is not! There is a ***shorter*** path from ***A*** to ***G***: A_1C_5G with a total cost of ***6***.

It is impossible for $A_1C_2B_4G_4E$ to be optimal if it contains a subpath that is not optimal. In fact, the optimal path is $A_1C_5G_4E$ with a total cost of ***10***.



Search Fight!



- Consider searching for a path from ***A*** to ***G*** in the weighted graph to the left.
- The ***Breadth-First Search*** algorithm would visit each vertex in order based on ***distance from the start***.
  - All vertices that are ***one*** edge from start, then vertices that are ***two edges***, and so on.
  - In order to remain consistent, given the choice between two unexplored vertices, we choose the one that comes ***first alphabetically***.
  - The path the BFS would find is therefore ***$A_4B_2D_7F_{10}G$*** with a total cost of ***23***.
- ***Depth-First Search*** chooses ***one*** unexplored neighbor and heads ***deeper*** into the graph.
  - For consistency we also choose the neighbors in ***alphabetical order***.
  - The path that ***DFS*** would find is therefore ***$A_4B_2D_{12}C_1E_2F_{10}G$*** with a total cost of ***31***.
- ***Nearest Neighbor*** would find the path ***$A_4B_1E_1C_{12}D_7F_{10}G$*** with a total cost of ***35***!
- ***None*** of our algorithms find the optimal path, which is ***$A_4B_1E_2F_{10}G$*** with a total cost of ***17***.

A Path Tuple

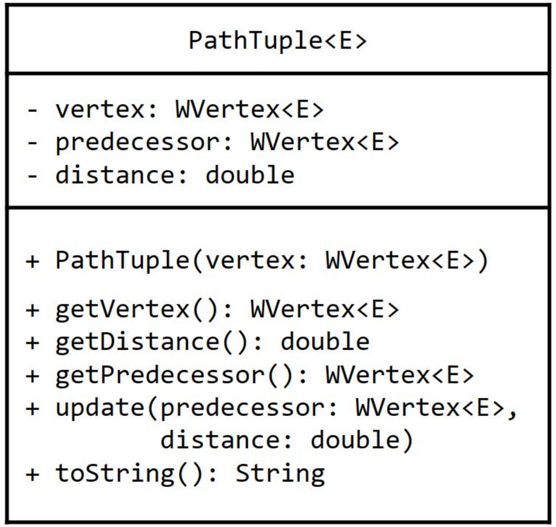
- When we implemented ***breadth-first search*** we used a map to keep track of each vertex ***V*** and its predecessor.
    - The predecessor was ***always*** the vertex through which we first discovered ***V*** as a neighbor.
    - This is one way the BFS insures that the paths that it finds will have the fewest edges.
- But BFS ignores the ***cost*** of the edges between neighbors.
    - We have already seen that it will not find the lowest cost path much of the time.
- Simply keeping track of the predecessor vertex will not be sufficient; we ***also*** need to know the ***total distance*** from the ***starting vertex*** through that predecessor.
- This means that we will need a new class to store both pieces of information: a ***path tuple***.
    - These tuples will be stored as the ***values*** in a ***map*** with each vertex as a ***key***.
    - The starting vertex will begin with a `null` predecessor and a ***distance*** of 0, e.g. `(null, 0)`
    - All other vertices will begin with a `null` predecessor and a ***distance*** of ***infinity***, e.g. `(null, ∞)`

Vertex	Path Tuple
A	(null, 0)
B	(null, ∞)
C	(null, ∞)
D	(null, ∞)
E	(null, ∞)
F	(null, ∞)
G	(null, ∞)

As the algorithm runs and each vertex is discovered (or *rediscovered*), the values in the path tuples will be updated to reflect the ***predecessor vertex*** and ***shortest known distance***.

8.15 The Path Tuple

In order to keep track of the shortest distance to each vertex, we will need a *path tuple* class. This class will keep track of a vertex, its predecessor, and the currently know shortest distance from the start vertex. We will update the predecessor and distance as we find new, shorter paths.

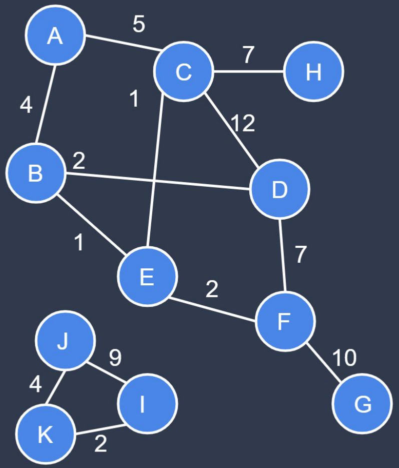


```
PathTuple<E>

- vertex: WVertex<E>
- predecessor: WVertex<E>
- distance: double

+ PathTuple(vertex: WVertex<E>)

+ getVertex(): WVertex<E>
+ getDistance(): double
+ getPredecessor(): WVertex<E>
+ update(predecessor: WVertex<E>,
         distance: double)
+ toString(): String
```

- Rename the "`PathTuple.txt`" file that has been provided to you in the `weighted` package so that it has a `.java` extension.
- A `PathTuple` has ***three*** essential pieces of information:
  - A `vertex`.
  - Its `predecessor`.
  - The ***currently known*** shortest `distance` from the starting vertex.
- Note that the ***constructor***:
  - Sets the `predecessor` to `null`.
  - Sets the `distance` to `Double.POSITIVE_INFINITY`.
- Note that the `update` method will ignore the update if the updated distance is not less than the current distance.
- ***Run*** the provided `PathTupleTest` to make sure that everything is working as expected.

- The search algorithms that we have implemented so far all have different ways of choosing which vertex is ***next***.
  - ***BFS*** chooses the next vertex from a ***queue***.
  - ***DFS*** chooses ***any*** unexplored neighbor of the ***current vertex***.
  - ***Nearest Neighbor*** chooses the neighbor with the ***shortest distance*** from the ***current vertex***.
- Dijkstra's Algorithm has its own way of choosing the next vertex: it will always choose the vertex that has the ***shortest distance*** from the ***starting vertex***.
  - Remember, each vertex has a ***path tuple*** that keeps track of its ***predecessor*** and the ***total distance*** from the ***starting vertex***.
  - We'll use this information to pick the next closest vertex each time.
- We will need a data structure to store the path tuples in such a way that we can find the next closest one.
  - This sounds like a ***priority queue*** of path tuples!

A Priority Queue



PRIORITY QUEUE

```
A:(null, 0)
K:(null, ∞)
J:(null, ∞)
D:(null, ∞)
F:(null, ∞)
H:(null, ∞)
G:(null, ∞)
E:(null, ∞)
B:(null, ∞)
C:(null, ∞)
I:(null, ∞)
```

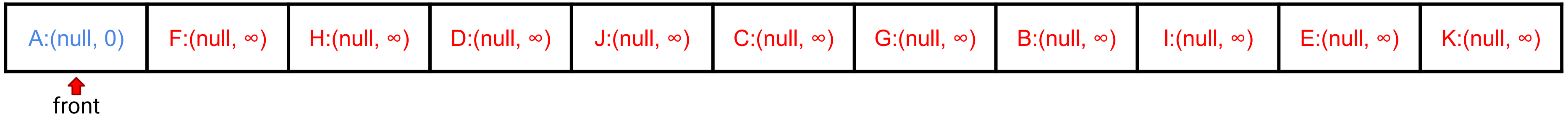
To begin with, the ***only vertex*** with a distance that is less than ***infinity*** is the ***starting vertex***, and so it will always be returned first.

Unfortunately...Java's Priority Queue Won't Work

Java's `PriorityQueue` internally uses an array-based ***heap*** to arrange the values in priority order.

As you know, a heap arranges values into priority order ***as they are added*** to the heap.

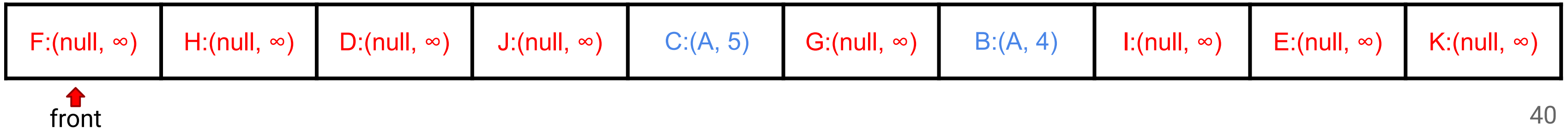
To start, all but one (the starting vertex) has a distance of ***infinity*** meaning that they have the ***same priority***.



As ***Dijkstra's Algorithm*** runs, the values of some of the tuples in the heap will be ***removed***...

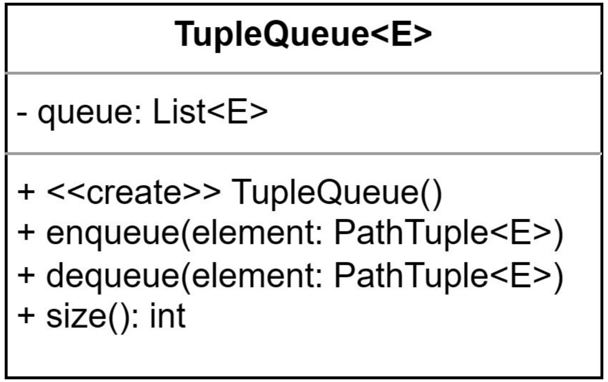
...and others will be ***updated*** with new values for the shortest known distance from the starting vertex.

As the tuples change, the heap ***will not*** automatically rearrange them into priority order. And so they will dequeue in the ***wrong order***.



8.16 The Tuple Queue

Unfortunately, Java's `PriorityQueue` will not rearrange values if/when they change. We'll need to use our own (and find tuples using a linear search). Examine the code now.

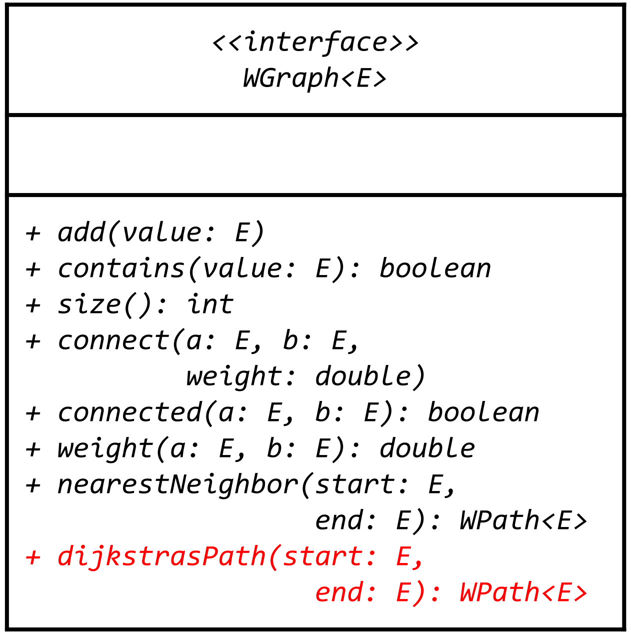


```
TupleQueue<E>
- queue: List<E>
+ <<create>> TupleQueue()
+ enqueue(element: PathTuple<E>)
+ dequeue(element: PathTuple<E>)
+ size(): int
```

- Open the "`TupleQueue`" class that has been provided to you in the `weighted` package.
  - `PathTuple` elements are stored in a linked list.
  - The `enqueue` method adds elements at the back.
  - The `dequeue` method performs a linear search for the `PathTuple` with the shortest distance from the start vertex.
- ***Test*** your code with the provided `TupleQueueTest` to make sure that everything is working as expected.
  - ***Raise your hand*** if your tests do not pass.

8.17 Step 0: Update `WGraph`

Once again, we're adding a new method to an existing interface. Add a default implementation of Dijkstra's Algorithm to the `WGraph` interface that throws an exception.



```
<<interface>>
WGraph<E>

+ add(value: E)
+ contains(value: E): boolean
+ size(): int
+ connect(a: E, b: E,
          weight: double)
+ connected(a: E, b: E): boolean
+ weight(a: E, b: E): double
+ nearestNeighbor(start: E,
                  end: E): WPath<E>
+ dijkstrasPath(start: E,
                end: E): WPath<E>
```

- Open the `WGraph` interface and define a default method named "`dijkstrasPath`" that declares parameters for `start` and `end` values and returns a `WPath`.
    - The default implementation should throw an ***unsupported operation exception***.
- Verify that your code ***compiles***.
    - The PROBLEMS tab in VSCode should be empty.

Step 1: Setup

The setup of Dijkstra's Algorithm requires two different data structures.

The first is a ***map*** to keep track of ***every*** vertex in the graph and its ***path tuple***. The vertices are the keys.

The path tuple for the ***start*** vertex has a `null` ***predecessor*** (because there isn't one) and a ***distance of 0*** (to itself).

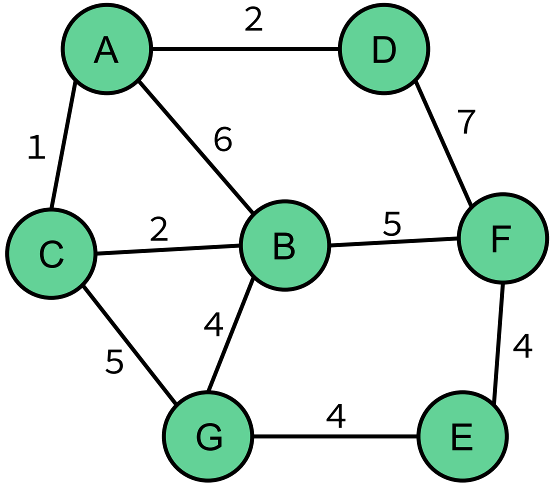
Every other vertex in the graph has a `null` ***predecessor*** and a ***distance of infinity*** (because we don't know yet).

All of the path tuples are also placed into a ***priority queue***.

Vertex	Path Tuple
A	(null, 0)
B	(null, ∞)
C	(null, ∞)
D	(null, ∞)
E	(null, ∞)
F	(null, ∞)
G	(null, ∞)

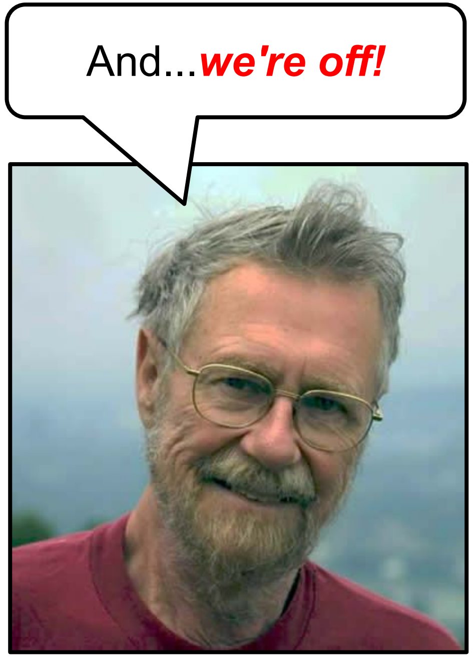
PRIORITY QUEUE

```
A:(null, 0)   C:(null, ∞)   B:(null, ∞)
D:(null, ∞)                 F:(null, ∞)
E:(null, ∞)   G:(null, ∞)
```



8.18 Step 1: Setup

Dijkstra's Shortest Path is probably the most complex algorithm that we will implement in this entire course. Let's take is slowly by implementing it in relatively small pieces, one at a time. As usual, we will begin be creating the necessary data structures.



- Open the `WAdjacencyGraph` class, override the `dijkstrasPath` method inherited from the `WGraph` interface, and implement ***step 1: setup***.
    - Use the `start` and `end` values to retrieve the corresponding vertices.
    - Create a `Map` to store `WVertex` keys and `PathTuple` values.
    - Create a `TupleQueue`.
    - Iterate over the vertices in the graph (***including the start vertex***):
        - ***Make*** a new `PathTuple` for the vertex.
        - ***Add both*** to the `Map`.
        - ***Add*** the tuple to the `TupleQueue`.
    - ***After*** the initial setup is complete, fetch the `PathTuple` for the start vertex from the map and update it to a ***distance of 0***.
    - For now, ***return*** `null`.
- Verify that your code ***compiles***.

Step 2: The Main Loop

Next, the main loop begins. The path tuple with the ***shortest distance*** from start is ***removed*** from the tuple queue.

*(this should be the **start vertex** first)*

The idea is that, if ***every other*** vertex still in the queue is ***farther away***, then we must already know the optimal path to ***this*** vertex.

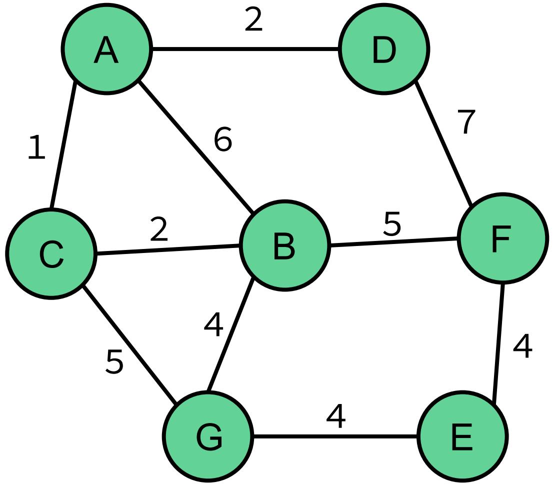
In other words, there can't be a ***shorter path*** through another vertex that is ***farther away***.

Let's call this vertex ***V***.

Vertex	Path Tuple
A	`(null, 0)`
B	`(null, ∞)`
C	`(null, ∞)`
D	`(null, ∞)`
E	`(null, ∞)`
F	`(null, ∞)`
G	`(null, ∞)`

PRIORITY QUEUE

```
C:(null, ∞)   B:(null, ∞)
D:(null, ∞)   F:(null, ∞)
E:(null, ∞)   G:(null, ∞)
```



`A:(null, 0)`

Step 2: The Main Loop

Next, we look at each of V's neighbors ***N*** and we compute the distance from the start to ***N*** through ***V***.

That distance is the ***sum*** of the distance in ***V***'s path tuple and the ***weight of the edge*** between ***V*** and ***N***.

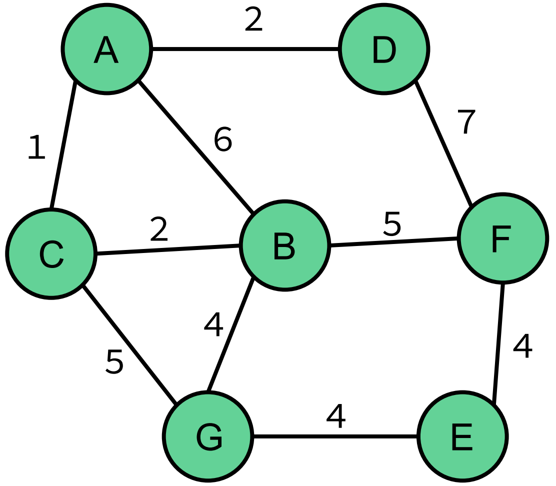
For example, the distance from the ***start*** to ***B*** through ***A*** is the ***distance*** in ***A***'s path tuple (***0***) plus the ***weight*** of the edge between ***A*** and ***B*** (***6***): ***0 + 6 = 6***

If this distance is ***shorter*** than the distance currently in ***N***'s path tuple, then the path tuple is updated with the new distance and ***V*** as the predecessor of ***N***.

Vertex	Path Tuple
A	`(null, 0)`
B	`(A, 6)`
C	`(A, 1)`
D	`(A, 2)`
E	`(null, ∞)`
F	`(null, ∞)`
G	`(null, ∞)`

PRIORITY QUEUE

```
C:(A, 1)    B:(A, 6)
D:(A, 2)
F:(null, ∞)
E:(null, ∞)    G:(null, ∞)
```



```
A:(null, 0)

A to B = 0 + 6 = 6
A to C = 0 + 1 = 1
A to D = 0 + 2 = 2
```

Step 2: The Main Loop

We continue to repeat this process, removing the ***V*** with the ***shortest distance*** from the tuple queue...

...computing the distance to each of its neighbors ***N*** through ***V***...

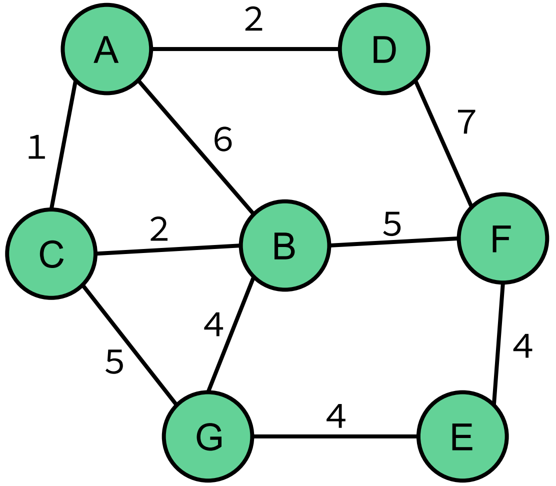
*(again, this is the distance in **V**'s path tuple plus the weight of the edge between **V** and **N**)*

If the computed distance is ***shorter*** than the distance already in ***N***'s path tuple, then the path tuple is updated with the new distance and ***V*** as the predecessor.

Vertex	Path Tuple
A	`(null, 0)`
B	`(C, 3)`
C	`(A, 1)`
D	`(A, 2)`
E	`(null, ∞)`
F	`(null, ∞)`
G	`(C, 6)`

PRIORITY QUEUE

```
B:(C, 3)
D:(A, 2)
F:(null, ∞)
E:(null, ∞)
G:(C, 6)
```



```
C:(A, 1)

C to A = 1 + 1 = 2
C to B = 1 + 2 = 3
C to G = 1 + 5 = 6
```

Step 2: The Main Loop

The loop continues as long as there are path tuples in the queue with a distance that is shorter than ***infinity***...

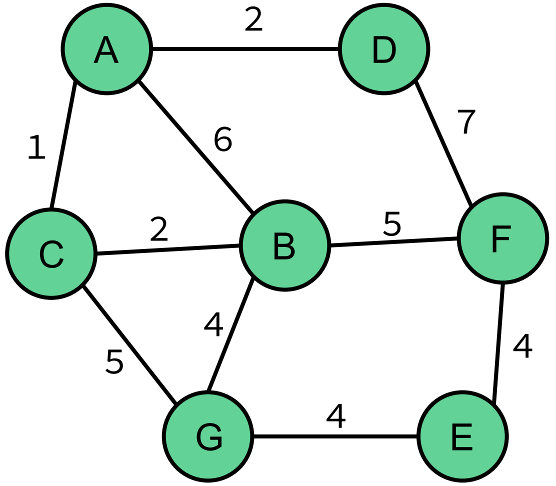
Vertex	Path Tuple
A	`(null, 0)`
B	`(C, 3)`
C	`(A, 1)`
D	`(A, 2)`
E	`(null, ∞)`
F	`(D, 9)`
G	`(C, 6)`

PRIORITY QUEUE

`B:(C, 3)`

`F:(D, 9)`

`E:(null, ∞)` `G:(C, 6)`



`D:(A, 2)`

```
D to A = 2 + 2 = 4
D to F = 2 + 7 = 9
```

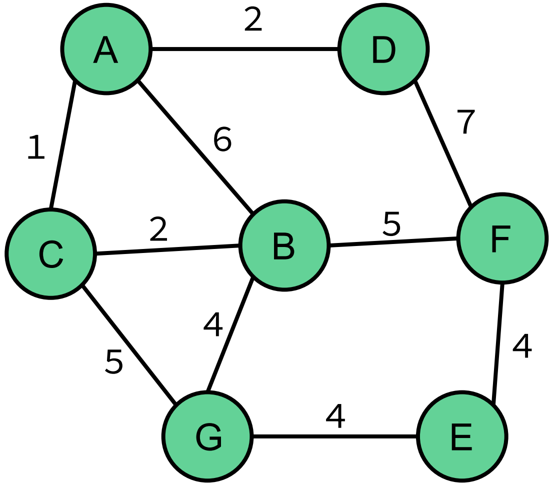
Step 2: The Main Loop

The loop continues as long as there are path tuples in the queue with a distance that is shorter than ***infinity***...

Vertex	Path Tuple
A	(null, 0)
B	(C, 3)
C	(A, 1)
D	(A, 2)
E	(null, ∞)
F	(B, 8)
G	(C, 6)

PRIORITY QUEUE

```
                              B:(C, 3)

                             F:(B, 8)
E:(null, ∞)             G:(C, 6)
```



```
B:(C, 3)

B to A = 3 + 6 = 9
B to C = 3 + 2 = 5
B to F = 3 + 5 = 8
B to G = 3 + 4 = 7
```

Step 2: The Main Loop

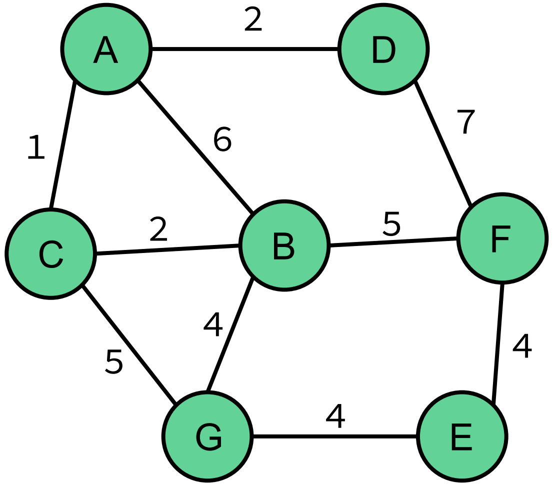
The loop continues as long as there are path tuples in the queue with a distance that is shorter than ***infinity***...

Vertex	Path Tuple
A	(null, 0)
B	(C, 3)
C	(A, 1)
D	(A, 2)
E	(G, 10)
F	(B, 8)
G	(C, 6)

PRIORITY QUEUE

F:(B, 8)

E:(G, 10)



G:(C, 6)

```
G to B = 6 + 4 = 10
G to C = 6 + 5 = 11
G to E = 6 + 4 = 10
```

Step 2: The Main Loop

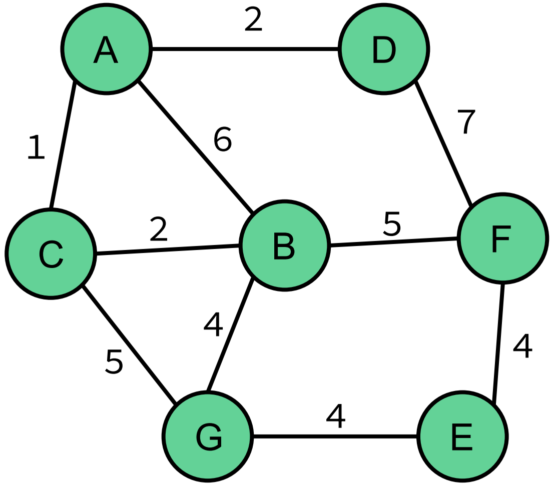
The loop continues as long as there are path tuples in the queue with a distance that is shorter than ***infinity***...

Sometimes, none of the neighbor vertices are updated, because the path through ***V*** is longer than the currently known ***shortest*** path.

Vertex	Path Tuple
A	(null, 0)
B	(C, 3)
C	(A, 1)
D	(A, 2)
E	(G, 10)
F	(B, 8)
G	(C, 6)

PRIORITY QUEUE

E:(G, 10)



F:(B, 8)

```
F to B = 8 + 5 = 13
F to D = 8 + 7 = 15
F to E = 8 + 4 = 12
```

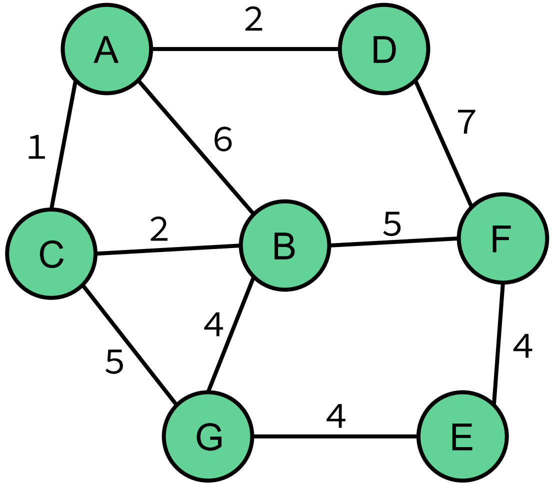
Step 2: The Main Loop

The loop continues as long as there are path tuples in the queue with a distance that is shorter than ***infinity***...

Once the ***tuple queue*** is empty, the main loop is complete. Unless...

Vertex	Path Tuple
A	`(null, 0)`
B	`(C, 3)`
C	`(A, 1)`
D	`(A, 2)`
E	`(G, 10)`
F	`(B, 8)`
G	`(C, 6)`

PRIORITY QUEUE



`E:(G, 10)`

```
E to F = 10 + 4 = 14
E to G = 10 + 4 = 14
```

Step 2: The Main Loop

Consider this hypothetical scenario where the tuple queue ***is not empty***, but the vertex that is ***closest*** to start has a distance of ***infinity***.

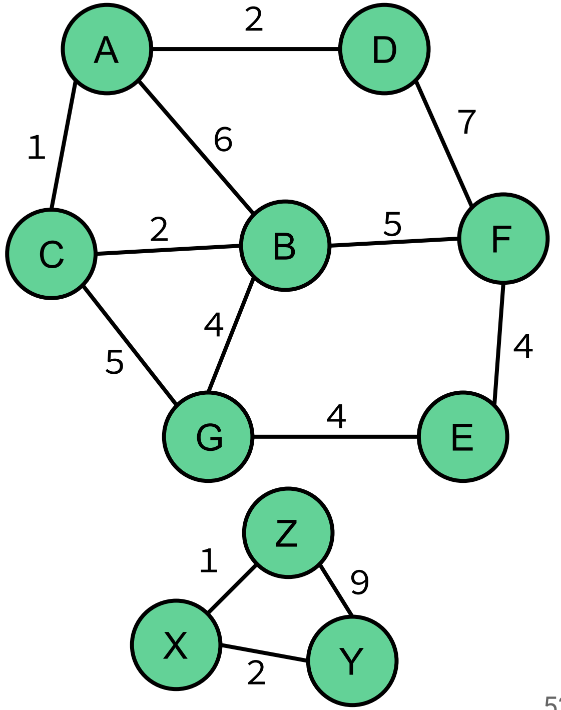
This means that any vertex still left in the tuple queue has never been updated.

This means that there is ***no path*** from start to any vertex left in the tuple queue. The main loop should ***stop***. even though the queue is not empty.

Vertex	Path Tuple
A	`(null, 0)`
B	`(C, 3)`
C	`(A, 1)`
	`...`
X	`(null, ∞)`
Y	`(null, ∞)`
Z	`(null, ∞)`

PRIORITY QUEUE

```
x:(null, ∞)    z:(null, ∞)
                    y:(null, ∞)
```



8.19 Step 2: The Main Loop

The goal of the main loop in Dijkstra's Shortest Path algorithm is to update the path tuple for *every* vertex that is reachable from the starting vertex. If a vertex is not reachable, its distance will remain set to infinity (and its predecessor will be null). Implement the main loop now.

- Open the `WAdjacencyGraph` class and navigate to the `dijkstrasPath` method, and implement the ***step 2: main loop***.
    - As long as the `TupleQueue` created in step 1 is not empty:
        - Dequeue the `PathTuple` with the next shortest distance from start.
        - If the distance in the tuple is ***infinity*** (`Double.POSITIVE_INFINITY`), stop.
        - Otherwise, get the vertex ***V*** from the tuple.
        - Get the ***neighbors*** map from ***V***.
        - For each neighbor ***N*** in ***neighbors***:
            - Get the ***distance*** from ***V*** to ***N*** from ***neighbors***.
            - Let ***DV*** be the *distance from start to* ***N*** through ***V***. ***Hint***: this is the distance in ***V***'s path tuple plus the ***distance*** from ***V*** to ***N***.
            - Use ***N*** to get its tuple from the ***map*** created in step 1.
            - ***Update*** the tuple with ***V*** and ***DV***. Remember, the update will be ignored if the new distance is not less than the current distance.

Step 3: Building the Path

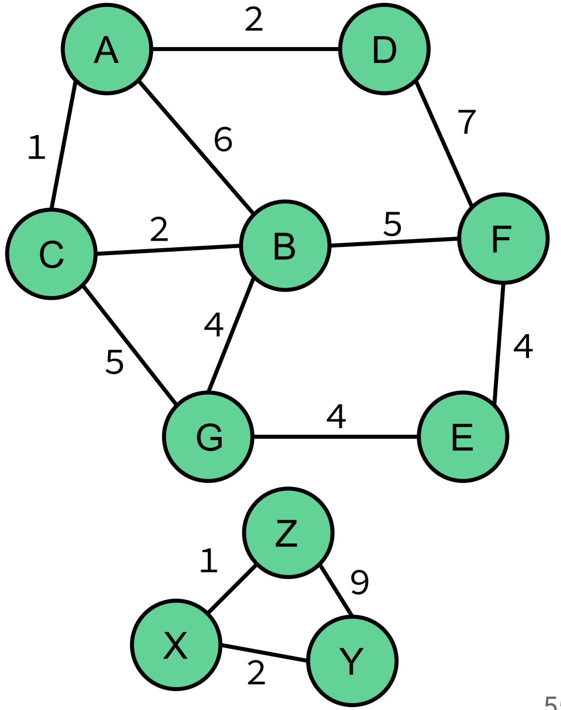
If there are any vertices left in the priority queue, you can ***forget them*** (they are unreachable from the start).

Vertex	Path Tuple
A	(null, 0)
B	(C, 3)
C	(A, 1)
D	(A, 2)
E	(G, 10)
F	(B, 8)
G	(C, 6)

PRIORITY QUEUE

x:(null, ∞) z:(null, ∞)

y:(null, ∞)



Step 3: Building the Path

Building a path from the start vertex to ***any reachable vertex*** works a lot like it does in BFS.

Start by using the ***end vertex*** (***E***) to retrieve its ***path tuple*** from the ***map***.

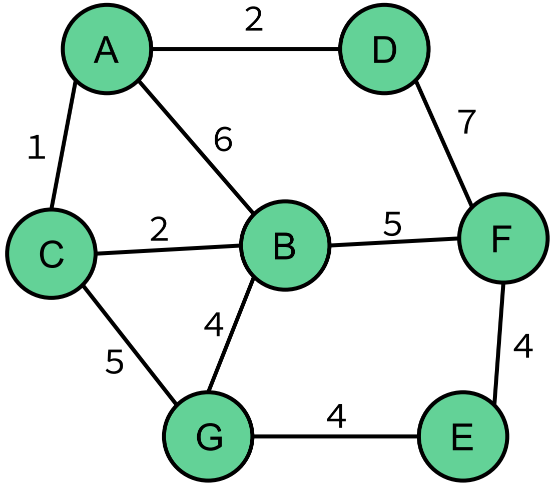
If the distance in the path tuple is ***infinity***, then ***there is no path*** from start. ***Return*** `null`!

Otherwise, ***create a path*** with the vertex's ***value*** and save the ***distance*** in a variable.

Vertex	Path Tuple
A	`(null, 0)`
B	`(C, 3)`
C	`(A, 1)`
D	`(A, 2)`
E	`(G, 10)`
F	`(B, 8)`
G	`(C, 6)`

			E

```
distance = 10
```



Step 3: Building the Path

Now, we will build the path ***backward***, just like the BF Path algorithm does.

Set the ***V*** to the end vertex's ***predecessor*** (get it from the tuple).

Loop as long as ***V is not*** `null`, add its ***value*** to the ***front*** of the path using the `distance` as its ***weight***, and then ***set*** `distance` ***to*** `0`*.

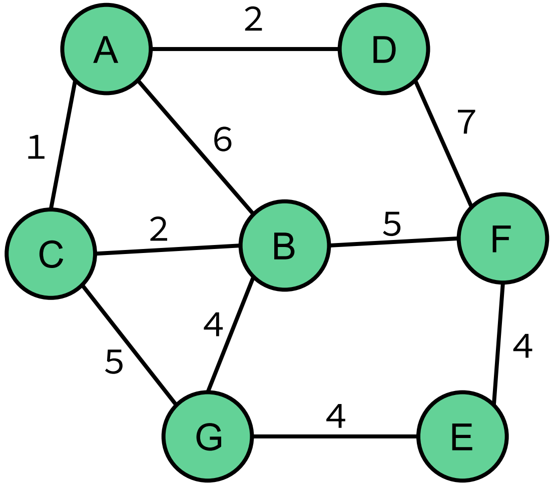
Then get ***V***'s tuple from the ***map*** and set ***V*** to ***its own predecessor***.

The loop should ***stop*** when you reach a `null` ***predecessor***; ***return*** the path.

Vertex	Path Tuple
A	`(null, 0)`
B	`(C, 3)`
C	`(A, 1)`
D	`(A, 2)`
E	`(G, 10)`
F	`(B, 8)`
G	`(C, 6)`

A	C	G	E

`distance = 0`



* Yes, this is a hack.

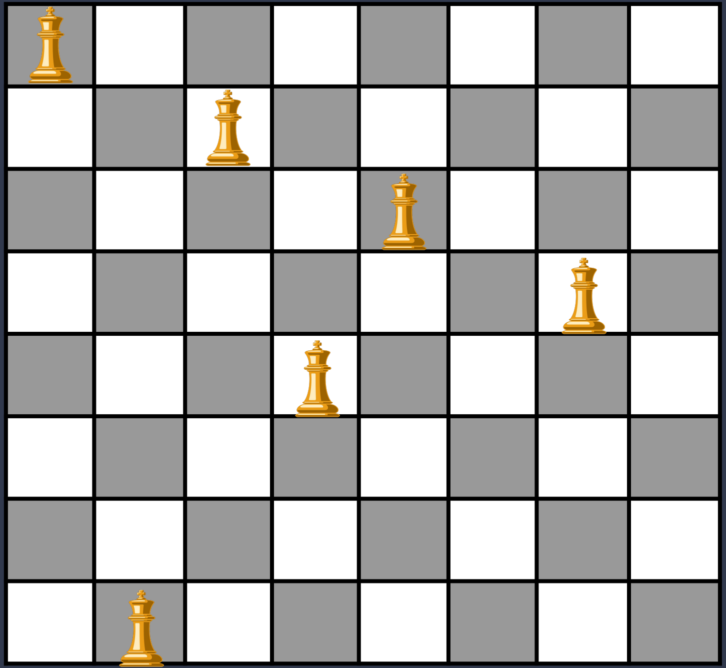
8.20 Step 3: Build the Path

The final step of Dijkstra's Shortest Path algorithm is to build the weighted path using the path tuples stored in the map. Let's do that now.

- Open the `WAdjacencyGraph` class and navigate to the `diskstrasPath` method, and implement the ***step 3: build the path***.
  - Use the ***end vertex*** to retrieve its `PathTuple` from the ***map*** created in step 1 and save the `distance` in a temporary variable.
  - If the `distance` in the tuple is ***infinity*** (`Double.POSITIVE_INFINITY`), there is no path. ***Return*** `null`.
  - Otherwise, ***create a new*** `WPath` with the end vertex's ***value*** and ***distance***. Remember that this is the *total distance from start*, so there is no need to add any additional distance later.
  - Let ***V*** be the end vertex's ***predecessor*** (get it from the tuple).
  - As long as ***V is not*** `null`:
    - Add ***V***'s value to the ***front*** of the path.
    - Get ***V***'s tuple from the map.
    - Set ***V*** to ***its own predecessor***.
  - When ***V*** is `null`, you have reached the start vertex. ***Return*** the path.
- ***Test*** your implementation using the provided `DijkstrasPathTest`.

The 8-Queens Problem

- In chess, the ***queen*** may move an unlimited number of squares in any direction: ***up***, ***down***, or ***diagonally***.
- Imagine that you are given a standard 8x8 chessboard with 64 squares, and 8 queen pieces and you are challenged to ***place all 8 queens*** on the board in a ***configuration*** such that no two queens can attack each other.
  - This means that no two queens can be in the same ***row***, ***column***, or ***diagonal*** on the board.
  - Is this even possible?
  - How would you go about trying to find if a solution to the problem exists?
- You might start by placing the first queen on any square on the board, e.g. the top left square.
  - For each subsequent queen, you would choose a square that cannot be attacked by any previously placed queen.
- What would you do if you'd placed fewer than 8 queens on the board, but none of the remaining squares is safe?
  - You will have reached a ***dead end***.
  - You would need to ***backtrack*** by picking up the last queen you played and trying it in a different square.
  - You may need to ***backtrack*** further and pick up two or more queens.



Today we will implement an algorithm that keeps trying, and backtracking, until it finds a solution (or tries every possible combination).

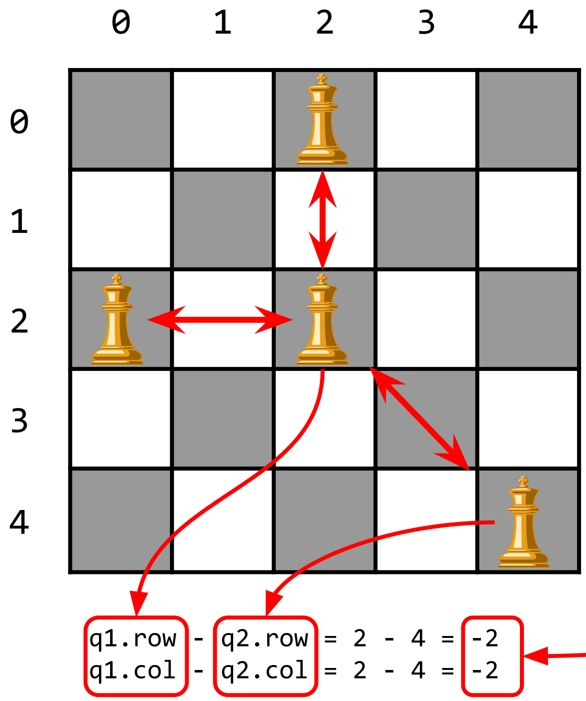
Queen Fight!

An essential part of solving the ***N-Queens*** problem is being able to quickly determine whether or not two queens that have been placed on the board can ***attack each other***.

This is true if two queens are in the same ***row***...

...or the same ***column***...

...or on the same ***diagonal***.



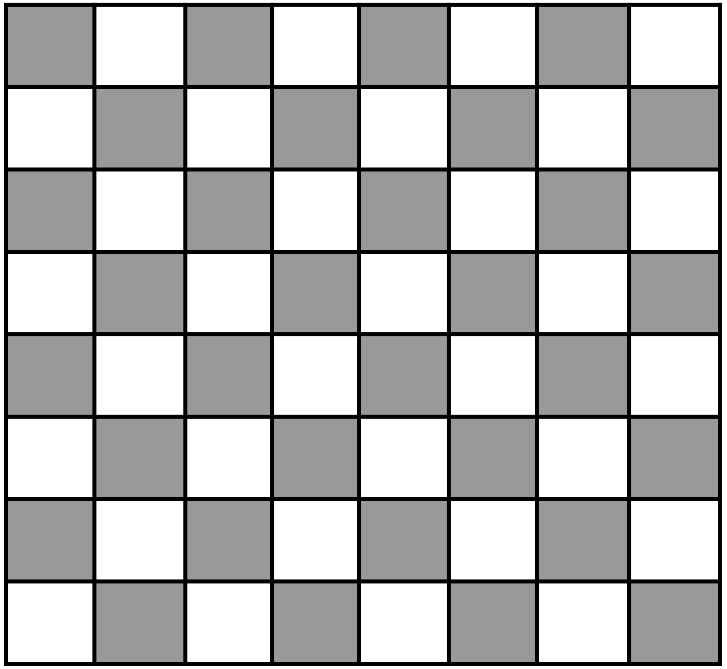
```
q1.row - q2.row = 2 - 4 = -2
q1.col - q2.col = 2 - 4 = -2
```

Assuming that we know the ***row*** and ***column*** of each queen on the board, comparing those is easy.

Determining whether or not two queens are on the same diagonal is a ***little*** more challenging...

First, calculate the difference between their ***rows*** and ***columns***. If the ***absolute values*** are the same, they are in the same diagonal.

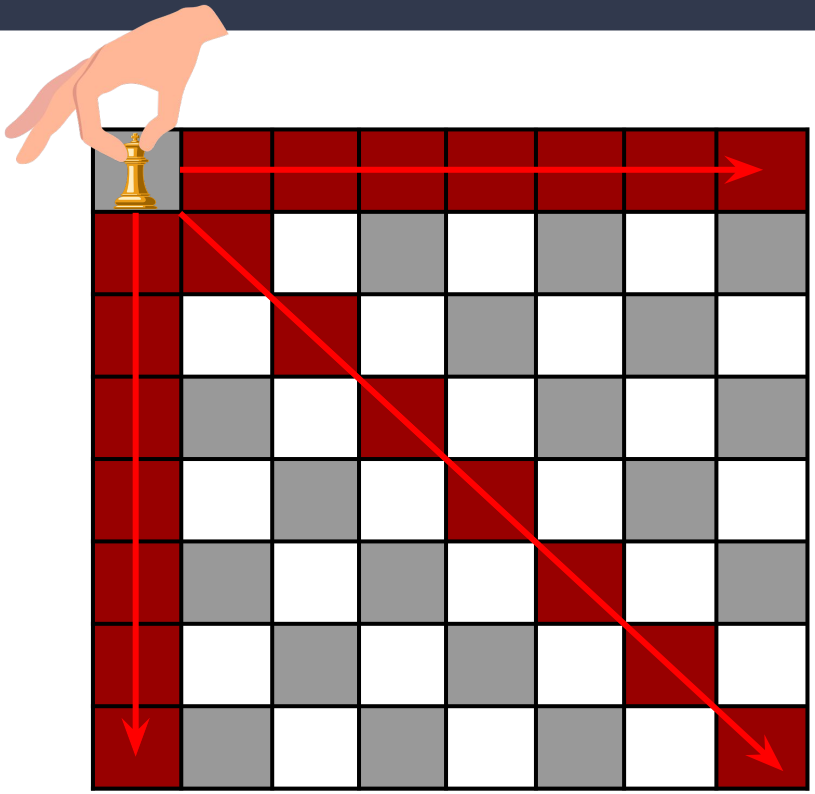
An Illustrated Example



The goal of the Eight Queens Problem is to place eight queens on a chessboard in such a way that no two queens can attack each other.

Q: If given a chessboard and 8 queens, how might you go about finding a solution to this problem?

An Illustrated Example



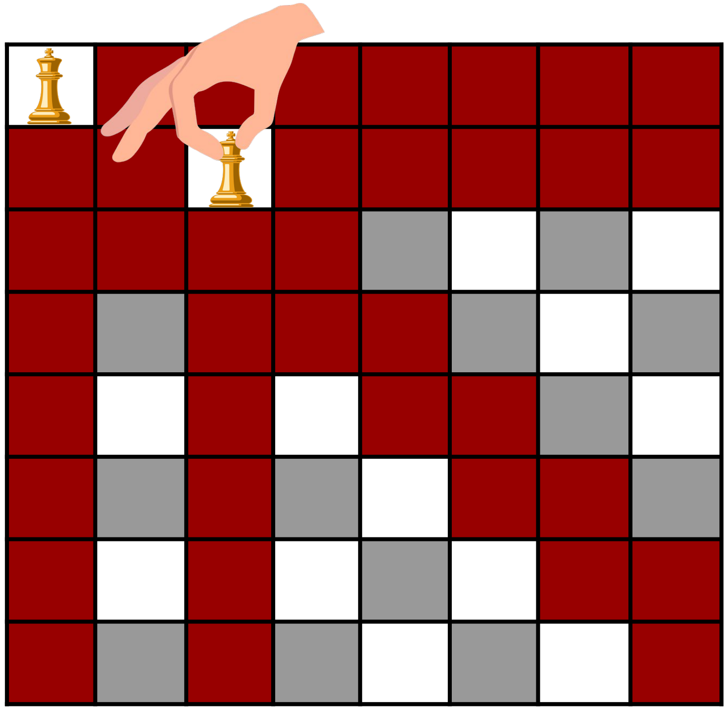
You would begin by placing a queen somewhere on the board.

Assuming that you wanted to take a methodical approach, you might choose to start in the top left corner.

Any square that the queen can attack is eliminated from consideration for the placement of a future queen.

This includes any squares in the same row, column, or on the same diagonal. We'll mark ***invalid*** squares in ***red***.

An Illustrated Example

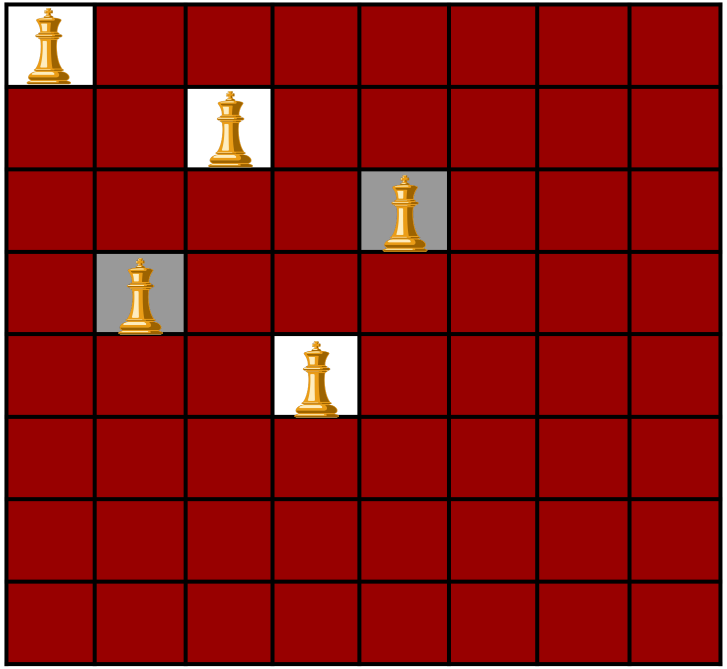


Continuing with your methodical approach, you might choose to place the next queen in the first open square in the next row.

This would eliminate any additional squares that *this* queen can attack.

You could repeat this process until you have either placed 8 queens on the board or...

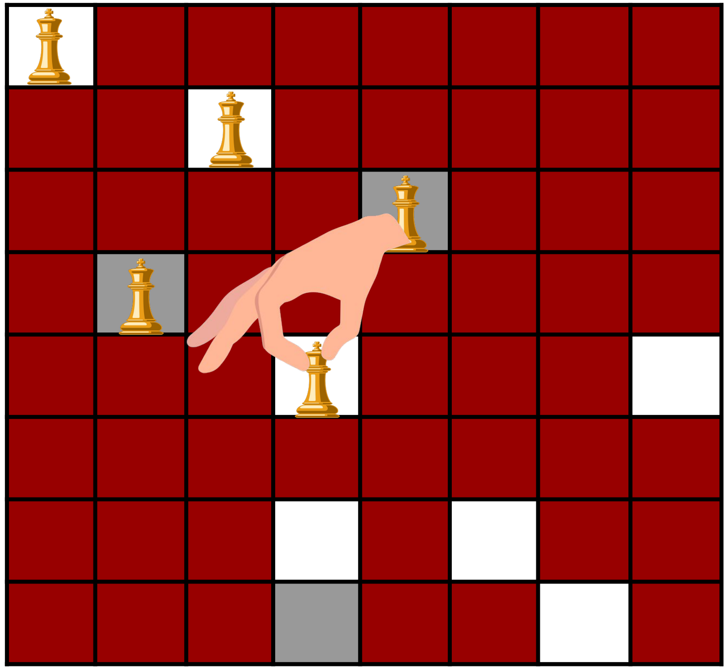
An Illustrated Example



...you run out of valid squares on which to place a new queen.

At this point you have two choices. You can give up and go do something else, or...

An Illustrated Example

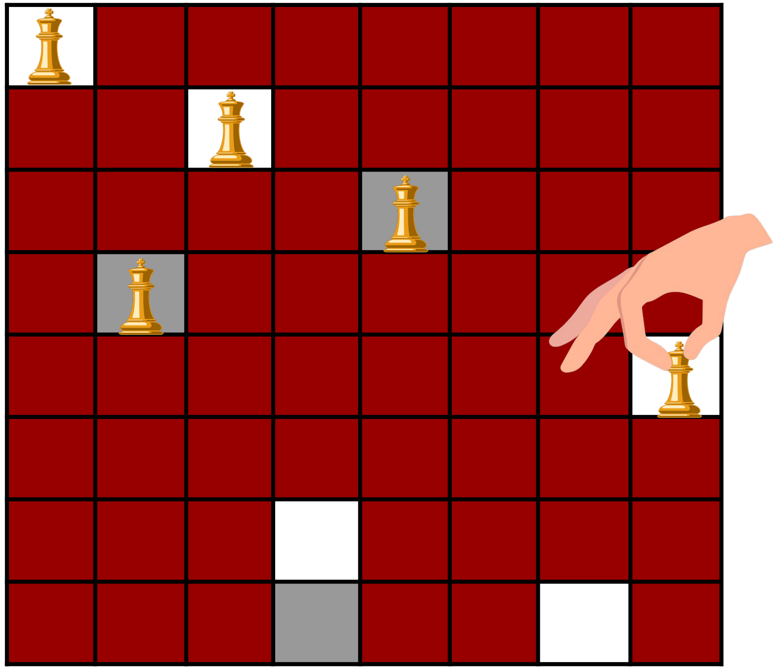


...you run out of valid squares on which to place a new queen.

At this point you have two choices. You can give up and go do something else, or...

...you can ***backtrack*** and pick up the last queen that you placed on the board...

An Illustrated Example



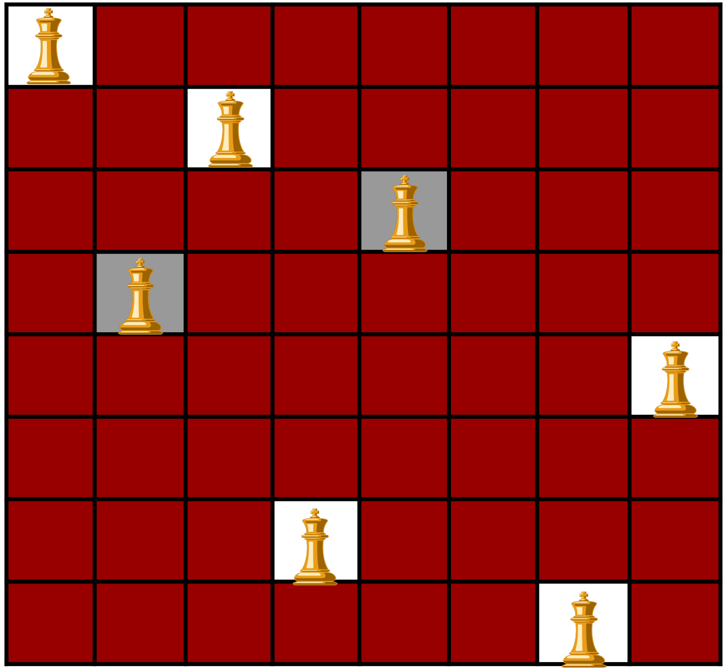
...you run out of valid squares on which to place a new queen.

At this point you have two choices. You can give up and go do something else, or...

...you can ***backtrack*** and pick up the last queen that you placed on the board...

...and then try playing it on a different square instead, which will hopefully leave you with a few more options.

An Illustrated Example

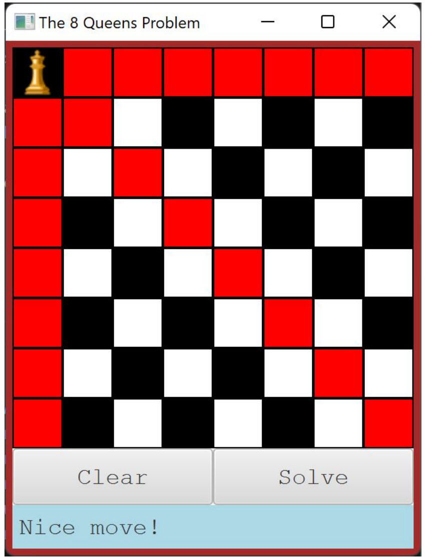


Eventually, no matter where you try to play the last queen, you may have no choice but to ***backtrack farther*** and pick up the last several queens.

You may even have to ***backtrack*** all the way to the ***very first queen*** that you placed on the board and try ***it*** in a different spot!

8.21 The N Queens Problem

Before designing a backtracking algorithm to solve the N-Queens problem, you should try it yourself! Your repository includes a JavaFX GUI that you can use to try to solve the N-Queens problem. Run the application now and see if you can solve the default configuration (8 queens).

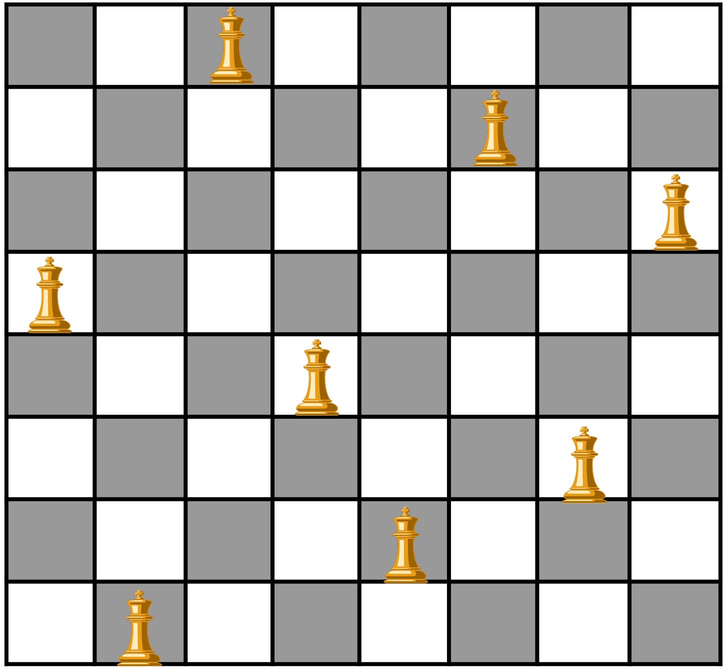


```
Nice move!
```

Note that the "`Solve`" button is not yet implemented!

- You will find the `queens.view.NQueensGUI` application in your repository.
  - It is configured to launch with an 8x8 board by default.
  - Click to place a queen on any square. Click again to remove the queen.
  - Invalid squares are marked in ***red*** - you won't be able to place a queen there.
- You may change the size of the board by closing the application, and launching with a ***command line argument***.
  - Press the up-arrow in the terminal to recall the last command.
  - Type the size of the board after the command, e.g. `4`, then press enter to run it.

An Illustrated Example



There ***is*** a valid solution for the 8-Queens Problem.

There are actually ***lots*** of solutions.

Today we are going to design an algorithm capable of solving the ***N-Queens*** problem: placing ***N*** queens on an ***N*N*** chessboard.

*(we should be able to find solutions for any **N>3**)*

8.22 It's Good to be the Queen

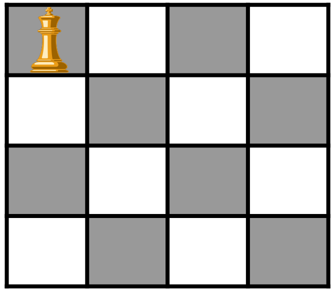
Not surprisingly, the core of the N-Queens problem is the `Queen` class. Take a few moments to examine the code and ask any questions that you may have about the class.

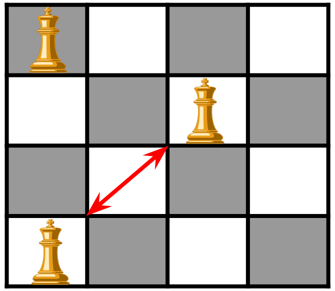
- Open the "`queens.model.Queen`" class and examine the code inside.
  - You will see that a `Queen` is essentially a row, column location on the board and a few utility functions.
  - You should also note that instances of the class can be used in hashing data structures.
- You will need to have a good understanding of the `Queen` class in order to use it to implement your backtracking algorithm. If you have any questions, now is the time to ask!

Backtracking Vocabulary

A ***configuration*** is an attempted solution at any stage.

In the ***N-Queens*** problem this would be a chessboard with ***0 or more*** queens placed on it.



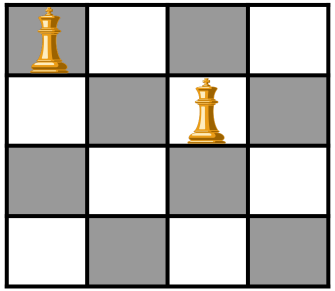


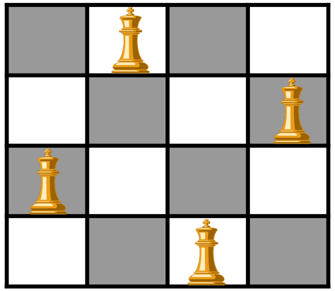
A configuration is ***invalid*** if it breaks the rules of the problem being solved.

In the ***N-Queens*** problem this would mean two queens can ***attack each other***.

A ***successor*** configuration is one where one of the ***possible next choices*** has been made.

In the ***N-Queens*** problem, this would mean that ***an additional queen*** has been placed in an unoccupied square.





A configuration is the ***goal*** if it is a valid solution to the problem.

In the ***N-Queens*** problem this means ***N*** queens have been placed and ***no two*** can attack each other.

The Configuration Interface

- We will be using a ***backtracking algorithm*** to attempt to find a solution to the ***N-Queens*** problem on chessboards of various sizes.
- The first step in utilizing this algorithm is to create an implementation of the `Configuration` ***interface***.
  - Remember that a ***configuration*** represents an attempted solution to a problem that includes the results of making ***zero or more*** choices.
- The `Configuration` interface defines ***three*** required methods, the implementations for which will vary based on the problem being solved.
  - `isValid()` returns `true` if the configuration is valid. Which is to say it does not break the rules of the problem.
  - `isGoal()` returns `true` if the configuration is a ***valid solution*** to the problem.
  - `getSuccessors()` returns a ***collection*** of ***successors***, each of which represents one of the possible next choices.
- The class that implements `Configuration` must include the ***state*** needed to implement each of the required methods.

At first glance, the `Configuration` interface is deceptively simple. It only contains three methods, two of which return a `boolean` value.

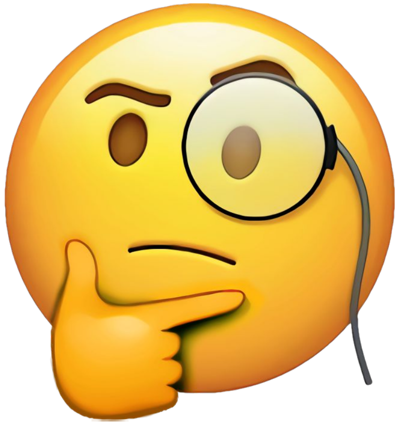
```java
public interface Configuration<C extends Configuration<C>> {
   public boolean isValid();

   public boolean isGoal();

   public Collection<C> getSuccessors();
}
```

But, depending on the nature of the problem being solved, any one of them may be very challenging to implement.

The first step in implementing a `Configuration` that can be used to attempt to solve a problem using backtracking is to answer a few questions...

8.23 A Few Questions

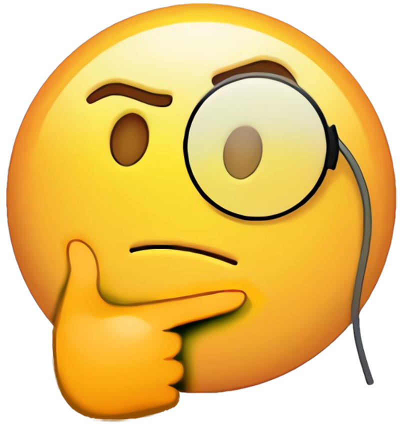
Think about how you would approach implementing a `Configuration` for solving the N-Queens problem using backtracking. Take a few moments to consider each of the following questions. Answer them in a file named "`queens.txt`".



How will you represent the ***state*** of a configuration? What fields and constructors will be needed?	How will you determine if a configuration is ***valid*** or not?
How will you determine whether or not a configuration is the ***goal***?	What do the ***successors*** of each configuration represent? How will you create them?

8.23 A Few Questions

Think about how you would approach implementing a `Configuration` for solving the N-Queens problem using backtracking. Take a few moments to consider each of the following questions. Answer them in a file named "`queens.txt`".



How will you represent the ***state*** of a configuration? What fields and constructors will be needed? The queens that have been placed The number of queens placed? The last move made? Do we need a chessboard?!	How will you determine if a configuration is ***valid*** or not? Does the canAttack method return true for any pair of queens?
How will you determine whether or not a configuration is the ***goal***? It is valid N queens have been placed on the board	What do the ***successors*** of each configuration represent? How will you create them? The set of next possible moves Create one for every empty square on the board?

A Closer Look at Implementing `Configuration`

```
public class TicTacToe
                 implements Configuration< TicTacToe > {

    @Override
    public Collection<TicTacToe> getSuccessors() {
        List<TicTacToe> successors = new ArrayList<>();

        return successors;
    }

    @Override
    public boolean isValid() {
        return true;
    }

    @Override
    public boolean isGoal() {
        return false;
    }
}
```

A Closer Look at Implementing `Configuration`

As is common when implementing an interface, the methods are initially stubbed out.

`getSuccessors()` should return a list of "next steps" towards possible solutions. Initially, the list will be empty.

`isGoal()` should return `true` if the configuration is a valid solution to the problem. Initially it will return `false`.

```
public class TicTacToe
                implements Configuration< TicTacToe > {

    @Override
    public Collection<TicTacToe> getSuccessors() {
        List<TicTacToe> successors = new ArrayList<>();

        return successors;
    }

    @Override
    public boolean isValid() {
        return true;
    }

    @Override
    public boolean isGoal() {
        return false;
    }
}
```

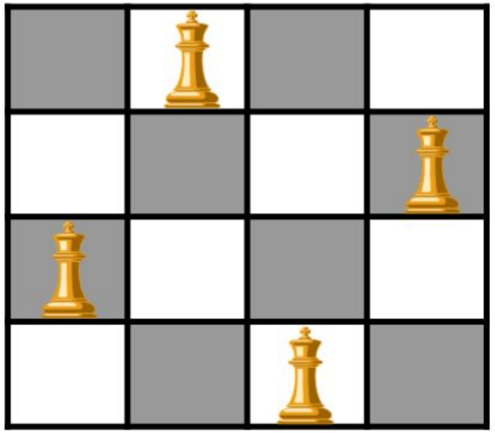
`Configuration` is ***generic*** and the implementing class specifies *its own type*.

This is similar to implementing `Comparable`.

`isValid()` should return `false` if the configuration breaks the rules of the problem. Initially it will return `true`.

8.24 Implementing a Configuration

Implementing a solution to the N-Queens problem will require us to write an implementation of the `Configuration` interface that can be used with the `Backtracker`. Let's begin by creating a class and stubbing out the required functions.



For any given value of ***N>3*** there are multiple possible solutions.

- Create a new Java class named "`NQueens`" in the `queens.model` package.
    - ***Implement*** the `Configuration` interface. You will find it in the `backtracker` package. Remember that it is ***generic***. Your implementation should specify *its own type* in place of the type parameter, similar to implementing `Comparable`.
    - ***Stub*** out the required methods.
    - Add any ***fields*** that you think that you will be needing.
    - Add an initializing ***constructor***.
    - Add a `toString()` that prints the configuration to standard output.
        - You don't need to do anything fancy.
- Define a `main` method with the appropriate signature.
    - ***Create*** an instance of your class representing an empty 4x4 chessboard.
    - ***Print*** the configuration to standard output.

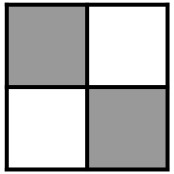
What's a Successor, Anyway?

Each configuration represents an attempt at a solution to the problem after which 0 or more choices have been made.

Let's take a look at an extremely simple version of the *N-Queens* problem: *"Can 2 queens be played on a 2x2 chessboard?"*

It's immediately obvious to humans that there is no solution to this problem, but a computer will need to use trial and error to figure that out.

The ***initial configuration*** represents 2x2 chessboard on which ***no queens*** have yet been played.



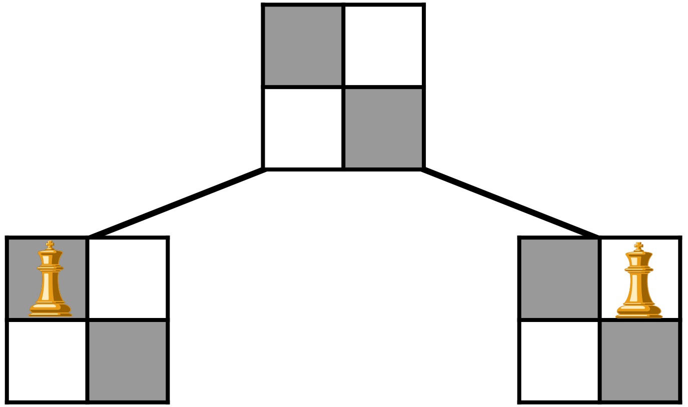
What's a Successor, Anyway?

A ***successor*** is a ***new*** configuration that includes all of the choices made by its parent plus ***one*** additional choice.

We ***could*** say that there are ***four*** such potential choices that can be made on an empty chessboard.

But for now, let's focus on choices in just the ***top row*** on the board. On such a small board, there are only ***two*** such choices.

Each of these choices is a ***successor*** of the original configuration with ***one*** additional choice made.



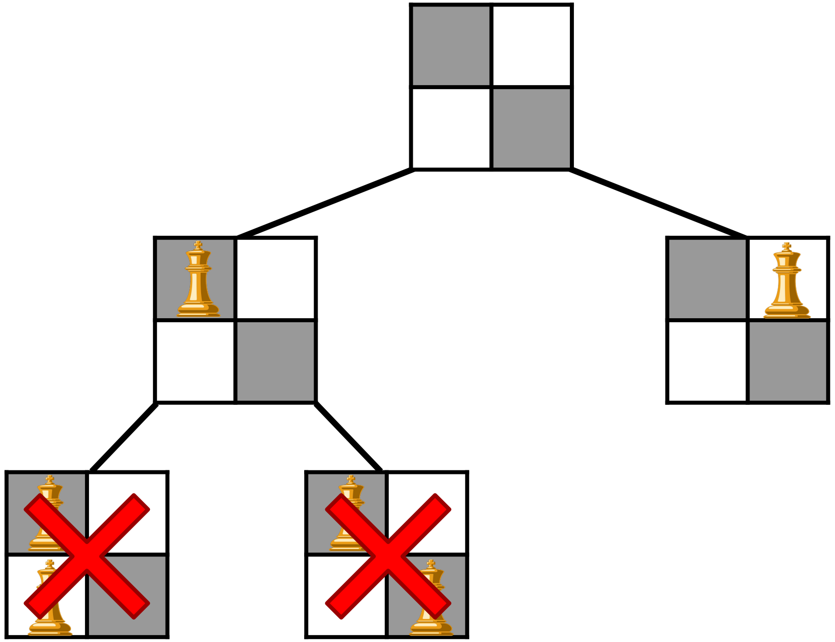
What's a Successor, Anyway?

The ***goal*** of this problem is to see if there is a way to play ***two*** queens on the ***2x2*** board.

Neither successor is the goal, but neither is ***invalid***, and so the ***backtracking algorithm*** will choose ***one*** and ask it for ***its successors***.

We will avoid wasting time by trying to play another queen in the top row. Instead, we will create a ***successor*** for each of the possible moves in the ***bottom row***.

Both of these are ***invalid*** configurations with illegal moves. These are ***dead ends***.



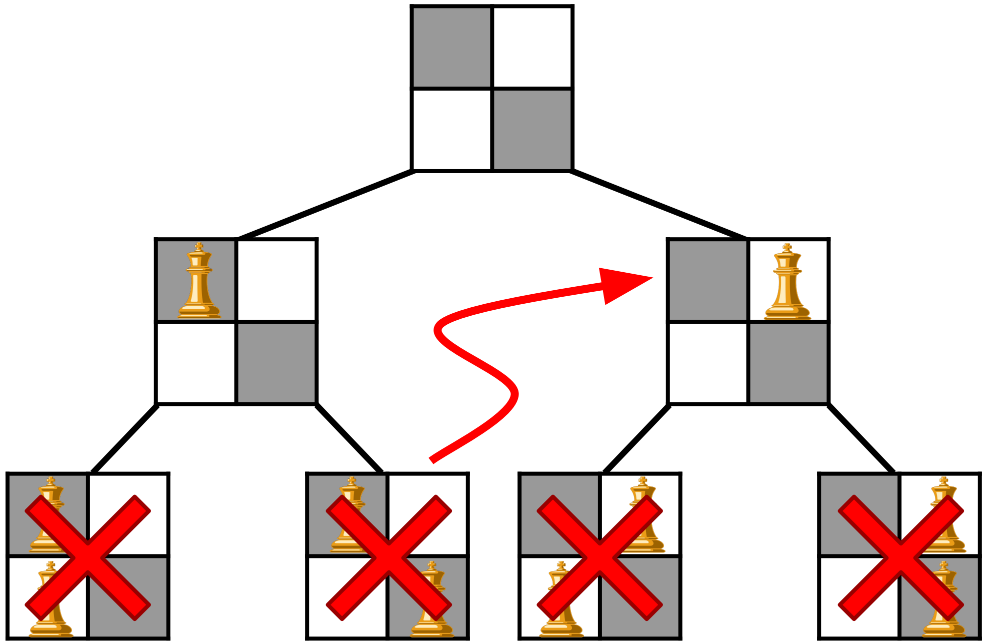
What's a Successor, Anyway?

At this point, the algorithm will ***backtrack*** to a previous, valid configuration and use ***it*** to create another collection of successors.

Once again, each successor contains the same state as its parent configuration plus ***one*** additional move in the second row.

Also once again, both configurations are ***invalid***.

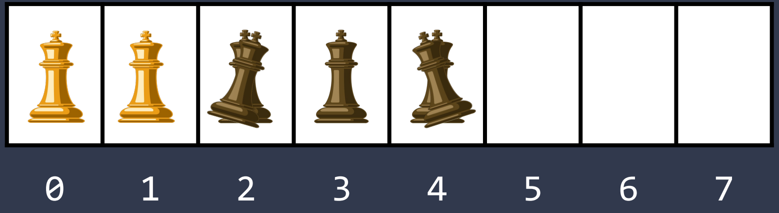
At this point the algorithm has exhausted all possible configurations without finding a ***goal***. There is no solution to this problem.



- Each configuration needs to store the queens that have already been placed on the board in some kind of ***data structure***.
- As you should know by now, all data structures (including ***arrays***) are ***reference types***.
  - If a function declares a parameter of an array type, a ***reference*** to the array is passed into the function.
  - Any changes made inside the function ***persist***!
  - This includes ***constructors!***
- This means that if a configuration simply passes its array of queens into each of its successors through the constructor, they will all share ***the same array***.
  - Any new moves made will overwrite each other, or add additional (potentially invalid!) queens to the board.
- In order to avoid this problem, a ***deep copy*** of the data structure must be made.
  - This includes making a new array of the required size and copying all of the elements into the new array before giving the ***copy*** to a successor.
- The same will need to be done with ***any mutable reference type*** that is used by a configuration.

Deep Copies

Data structures like arrays are ***reference types***. When used as an argument to a function, a ***reference*** to the real object in memory is passed in.



This means that any changes made to the reference will be made to the ***single***, ***shared copy***.

In the N-Queens problem, this means that successor configurations will add queens to the ***same*** array that is used by the parent and even ***overwrite*** each other's queens!!

A ***deep copy*** of the array must be made for each successor to prevent this from happening.

The most ***common error*** that students make when implementing a backtracking configuration is ***forgetting to make deep copies***!

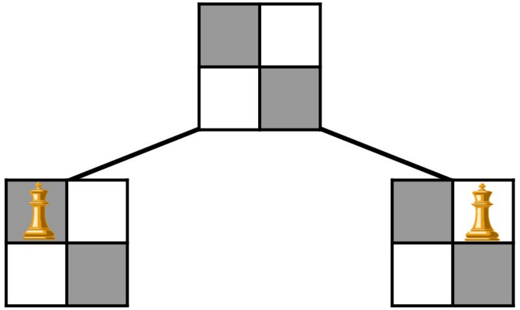
If your configuration makes changes to some mutable value, e.g. adding queens to an array, it ***must*** have its own copy!

Otherwise, two different configurations will ***change each other's data***!

Lesson 1
MAKE DEEP COPIES OF:
– Arrays
– Data Structures
– Mutable classes
– Everything!

8.25 Lines of Succession

A successor to an `NQueens` configuration includes one of the next possible moves that can be made on the chessboard. For now, we will just focus on playing one queen in each of the available squares in the next row.



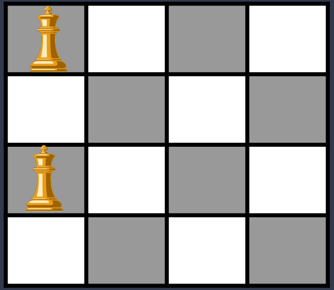
There should be one possible move for each successor in the next row.

If the board is empty, the "next" row is the first row.

- Open the `NQueens` class and implement the `getSuccessors()` method.
    - ***Don't waste time*** trying to place queens in the ***same row*** as the last queen that was played.
        - What if no queen has been played yet?
    - Generate a successor for each possible move in the ***next*** row.
        - This means that you should make ***N*** successors.
    - ***Hint***: Use the `Arrays.copyOf()` method to make a ***deep copy*** of your array of queens with room enough for ***one more queen***.
    - Be sure to make sure that you are not already in the last row!
- ***Test*** your `getSuccessors()` function from `main`.
    - There should be ***four*** successors on a 4x4 board, all in the top row. Print each of the successors.

Pruning

On larger boards it is not only ***possible***, but ***very likely*** that your algorithm will create an ***invalid*** successor long before making every possible move.



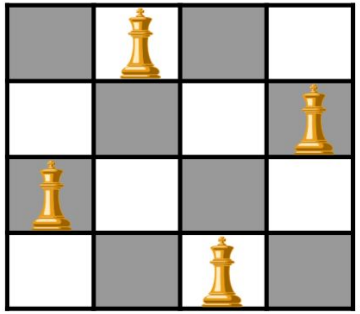
It is ***impossible*** for a valid solution to be reached after a single bad placement of a queen. And so there is no sense in wasting time continuing to create successors after this point.

The backtracking algorithm that we will be using will automatically ***prune*** the search tree whenever it finds an ***invalid configuration***.

- The 2-Queens problem is a fairly trivial instance of N-Queens that very quickly runs out of moves.
  - It just so happens that the ***last moves*** made always result in invalid configurations.
- Consider a larger chessboard where there are ***many more*** possible choices that can be made at each iteration.
  - Your configuration may create an ***invalid successor*** long before the number of possible choices is exhausted.
  - Is there any way for an ***invalid configuration*** to lead to a ***valid solution***?
- Depending on the number of possible choices that have to be tested, a backtracking algorithm can be ***extremely computationally intensive***.
  - You should try to ***optimize*** in any way you can to reduce the amount of time that it takes to find a solution.
- One optimization technique is called ***pruning***.
  - If a configuration is ***invalid***, no additional successor configurations are tried from that point.
  - This branch of the search tree is ***abandoned***.
- Pruning is built into the backtracker.
  - If the `isValid()` method returns `false`, configuration and all of its potential successors are ***pruned***.

8.26 Validating a Configuration

An N-Queens configuration is valid if no two queens have been played that can attack each other. Assuming that all previous configurations have been valid, which queen(s) do we need to look at closely? Which can we ignore?



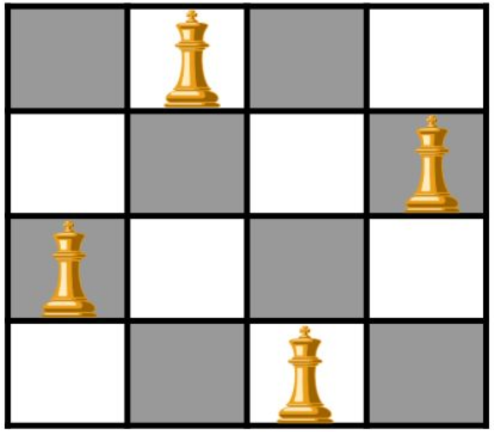
The `isValid()` method is ***only*** concerned with whether or not any two queens can attack each other.

But does it have to check them ***all***?

- Open the `NQueens` class and implement the `isValid()` method.
  - You should ***return*** `true` if none of the queens played (so far) can attack each other.
  - Do you need to compare ***every*** pair of queens?
- Test your method by calling it from `main`.
  - It should return `true` for an empty board. This is fine for now.

8.27 Gooooooooooal!

An N-Queens configuration is the goal if it is valid (no two queens can attack each other) and all N-Queens have been placed on the chessboard. Implement this functionality in your N-Queens configuration now.



For any board larger than 3x3 there are multiple possible solutions. The one that the backtracker finds depends on the order in which moves are made.

- Open the `NQueens` class and implement the `isGoal()` method.
  - If the configuration is ***valid***, and ***N queens*** have been placed on the chessboard, return `true`.
  - Otherwise, return `false`.
- ***Test*** your `isGoal()` function from `main`.
  - It should return `false`. This is good enough for now.

The Backtracking Algorithm

- Now that we have an implementation of the `Configuration` interface, it can be executed using a simple, generic ***backtracking algorithm*** to solve the problem.
- Call the ***solve*** method with a ***configuration C***:
  - If ***C*** is the ***goal***, then return ***C***. Problem solved!
  - Otherwise, for each of ***C***'s successors ***S***:
    - If ***S*** is ***valid***, recursively call ***solve*** with ***S***
    - If a ***non-*`null`** ***solution*** is returned, problem solved! return it!
  - If none of ***C***'s successors returns a non-null solution, then there is no solution from ***C***. Return `null`.
- The ***backtracking*** occurs whenever one of ***C***'s successors fails to return a solution.
  - The algorithm ***backtracks*** to ***C*** and then moves on to its next successor.
- An implementation of this algorithm has been provided to you in the `Backtracker` class.
  - You can create a Backtracker with or without ***debugging enabled***.
  - The `solve()` method will work with ***any*** class that implements `Configuration`.

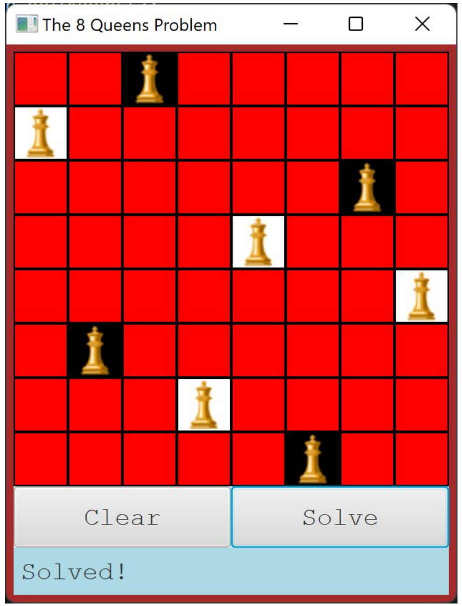
```
public C solve(C config) {
    if(config.isGoal()) {
        return config;
    } else {
        for(C child :
                        config.getSuccessors()) {
            if(child.isValid()) {
                C sol = solve(child);
                if(sol != null) {
                    return sol;
                }
            }
        }
    }
    return null;
}
```

Note that the `Backtracker` is also generic and declares a type parameter ***C***. This must be replaced with a real class that implements `Configuration`.

The `Configuration` returned may be `null` if no solution is found.

8.28 Plug in the Backtracker

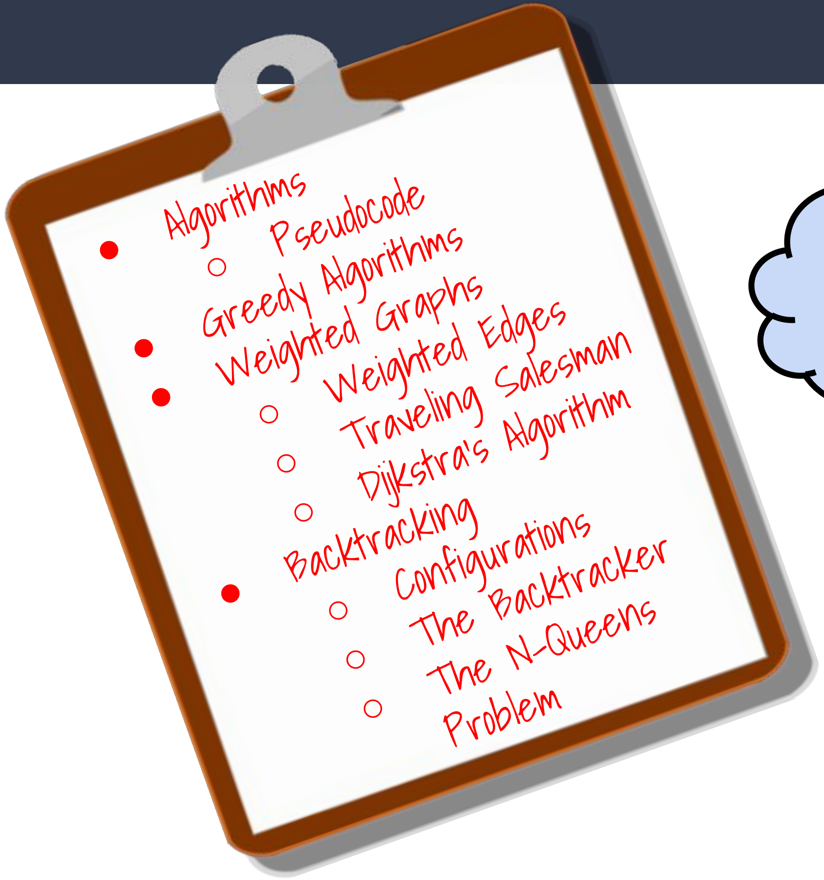
The "solve" method in the NQueens GUI has been commented out. If your backtracking configuration is ready, you can uncomment the code and try it out!



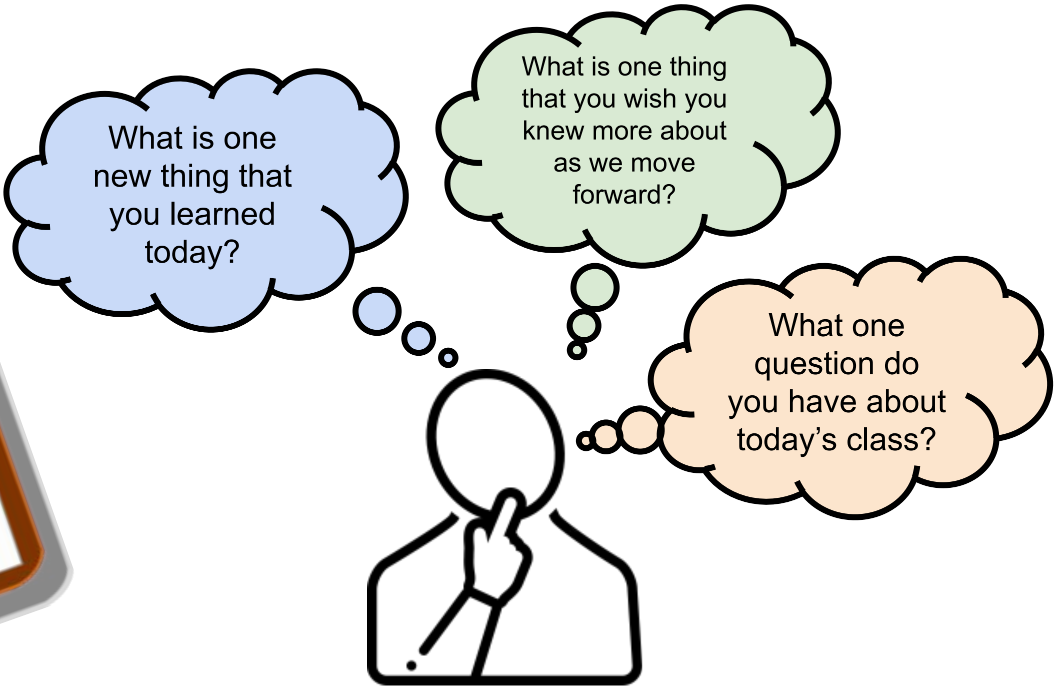
```
Solved!
```

- Open the `NQueens` class and implement a `getQueens()` method that returns the array of queens.
- Open the `NQueensGUI` class and navigate to the `solve` method at the top of the class.
    - ***Uncomment*** the code there.
    - You will need to import the classes required to use the backtracker and your configuration.
    - Take a few moments to examine the code, and ask any questions that you may have!
- Run the application and try it out!
    - Press solve right away.
    - Try placing a few queens first. You will need to place them top-to-bottom starting with the first row.
    - Can you find a set of moves that is unsolvable?
    - Try running with different board sizes. How long does it take to solve for 16 queens? 20?

Summary & Reflection



- Algorithms
  - Pseudocode
- Greedy Algorithms
- Weighted Graphs
  - Weighted Edges
  - Traveling Salesman
  - Dijkstra's Algorithm
- Backtracking
  - Configurations
  - The Backtracker
  - The N-Queens Problem



What is one new thing that you learned today?

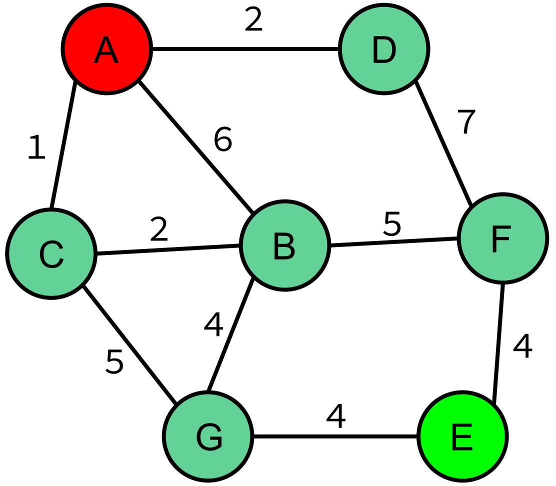
What is one thing that you wish you knew more about as we move forward?

What one question do you have about today's class?

Please answer the questions above in your notes for today.

Dijkstra’s Shortest Path Diagram

Iteration	Finalized Vertex	A	B	C	D	E	F	G
0	▨	0,null	∞,null	∞,null	∞,null	∞,null	∞,null	∞,null
1								
2								
3								
4								
5								
6								
7								

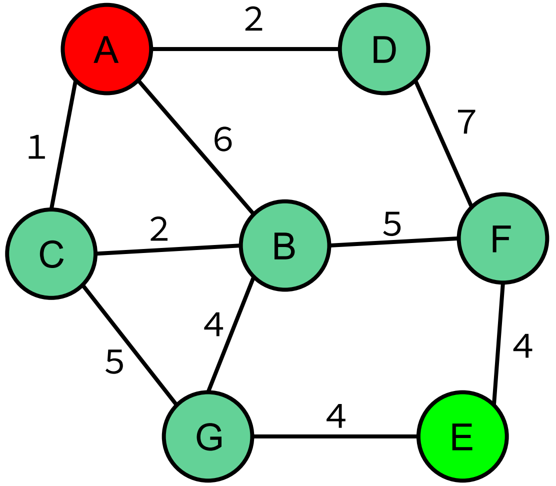


Path A -> E

Distance = ???

Dijkstra’s Shortest Path Diagram

Iteration	Finalized Vertex	A	B	C	D	E	F	G
0		0,null	∞,null	∞,null	∞,null	∞,null	∞,null	∞,null
1	A	X	6,A	1,A	2,A			
2	C		3,C	X				6,C
3	D				X		9,D	
4	B		X				8,B	
5	G					10,G		X
6	F						X	
7	E					X		



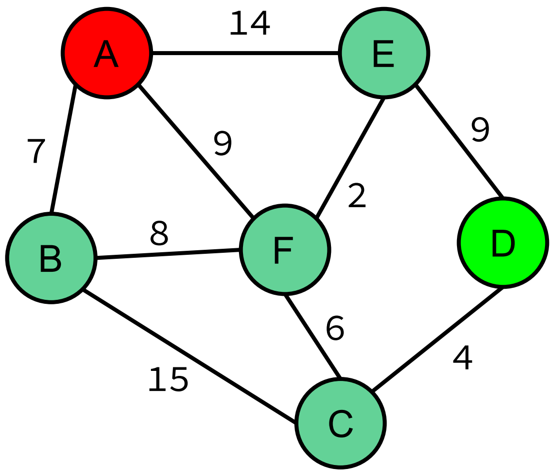
Path A -> E

A	C	G	E

Distance = 10

Dijkstra’s Shortest Path Diagram

Iteration	Finalized Vertex	A	B	C	D	E	F
0	▨	0/null	∞,null	∞,null	∞,null	∞,null	∞,null
1							
2							
3							
4							
5							
6							
7							

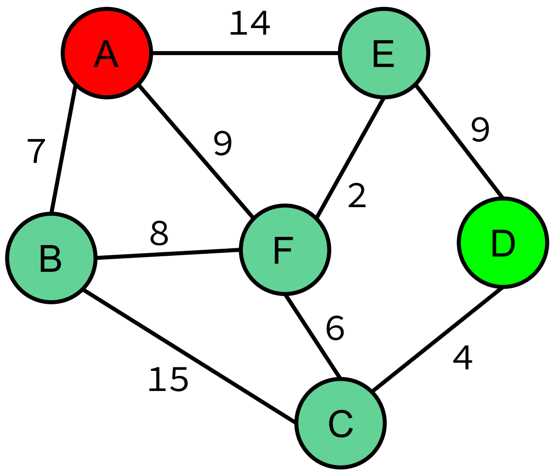


Path A -> D

Distance = ???

Dijkstra’s Shortest Path Diagram

Iteration	Finalized Vertex	A	B	C	D	E	F
0		0/null	∞,null	∞,null	∞,null	∞,null	∞,null
1	A	X	7,A			14,A	9,A
2	B		X	22,B			
3	F			15,F		11,F	X
4	E				20,E	X	
5	C			X	19,C		
6	D				X		
7							



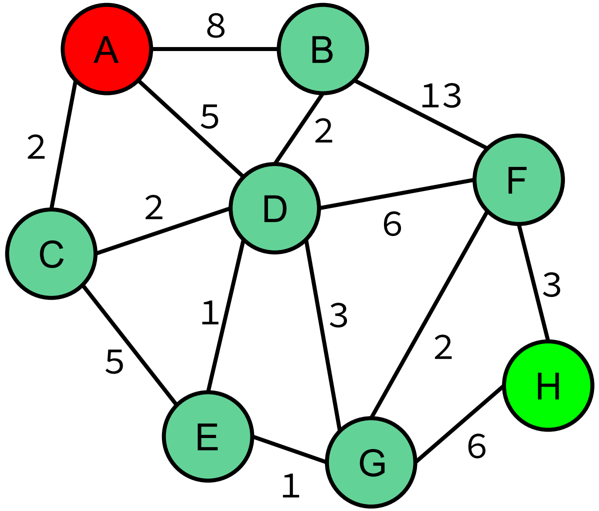
Path A -> D

A	F	C	D

Distance = 19

Dijkstra’s Shortest Path Diagram

Iteration	Finalized Vertex	A	B	C	D	E	F	G	H
0	▨	0,null	∞,null	∞,null	∞,null	∞,null	∞,null	∞,null	∞,null
1									
2									
3									
4									
5									
6									
7									
8									

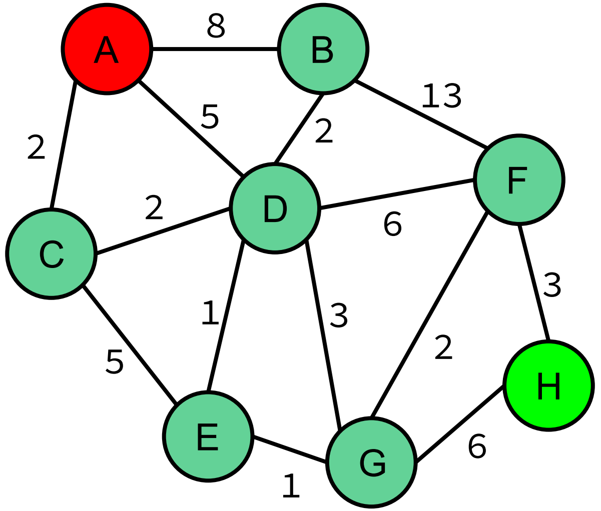


Path A -> H

Distance = ???

Dijkstra's Shortest Path Diagram

Iteration	Finalized Vertex	A	B	C	D	E	F	G	H
0		0,null	∞,null	∞,null	∞,null	∞,null	∞,null	∞,null	∞,null
1	A	X	8,A	2,A	5,A				
2	C			X	4,C	7,C			
3	D		6,D		X	5,D	10,D	7,D	
4	E					X		6,E	
5	B		X						
6	G						8,G	X	12,G
7	F						X		11,F
8	H								X



Path A -> H

A	C	D	E	G	F	H

Distance = 11