- To recompile and run the unit testing suite:

```
make clean && make test && ./test
```

For this assignment, you can run the same test suite that the autograder will use on Coursera. In addition, you can choose to run only the specific unit tests for a given exercise with a command-line argument. For example, to run only the Exercise 1 unit tests, you can do this:

```
./test [ex1]
```

The **`./main`** program doesn't support this command-line argument, but you can toggle various tests on or off in the source code itself. Look in the definition of the **`informalTests`** function in **`main.cpp`** for details.

### 2.4 Submitting Your Work

When you're ready to submit your work for final grading on Coursera, you'll need to bundle it into a zip file in a specific way. To do that, you can type this command from your main subdirectory, where the Makefile is located:

```
make zip
```

This will automatically package the files to be graded into a new zip file, **`graph_search_submission.zip`**, which will appear in the same directory. You can download the created zip file to your own computer from Cloud9 by right-clicking on the zip file in your Cloud9 file list and choosing "Download."

Once you've got the zip file on your local computer, you can log into Coursera and submit the zip file for this assignment. (If you need a reminder about how to use Cloud9 or how to submit work on Coursera, the first MOOC in this course sequence on Coursera has more detailed information about these topics in the lessons and homework assignments.)

## 3 Background Information

Here we discuss some of the topics, concepts, and syntax that the project touches upon. Much of this information is supplementary rather than essential, but please read to get the best understanding of what's going on in the project code.

### 3.1 Syntax for `std::unordered_set`

In this project, we use **`std::unordered_set`** in several places to represent collections of items having some shared property. A set contains value copies of objects, and it contains no duplicates; inserting the same item twice has no effect. In C++, the unordered set type uses hashing internally, so the key type must support equality comparison with the **`==`** operator as well as **`std::hash`** specialization. (We discussed that in the Week 1 project about **`std::unordered_map`**.) Unlike the unordered map type, the keys for an unordered set do not have associated values. We only care whether a given key has been inserted or not.

This is a convenient alternative to "labeling" vertices with some status, for example. If an algorithm tells you to label vertices as "visited", you could instead make a set for all the visited vertices, and insert copies of vertices to the visited set when you want to record that they have been visited.

For example, we could make a set to contain unique integers like this:

```
std::unordered_set<int> favoriteNumbers;
```

Then we can insert some items with **`insert()`**:

```
favoriteNumbers.insert(7);
favoriteNumbers.insert(42);
// Inserting again doesn't change anything:
favoriteNumbers.insert(42);
```

At this point, the set contains 7 and 42. (It only has a single copy of 42.) You can check the number of items with **`size()`** or **`empty()`**.

And then we can use **`count()`** to query if an item is in the set or not, and we can remove items from the set with **`erase()`**:

```
std::cout << favoriteNumbers.count(7) << std::endl; // output: 1
favoriteNumbers.erase(7);
std::cout << favoriteNumbers.count(7) << std::endl; // output: 0
```

### 3.2 Other C++ set implementations

It's worth mentioning that hashed containers that STL offers may not be optimal in some cases. Perhaps you wish to maintain ordered data or perhaps there isn't a fast enough hash function for your needs. There does also exist an ordered type **`std::set`** which is based on a binary tree implementation instead of hashing, and consequently uses the **`<`** and **`==`** operators to arrange items instead of **`std::hash`** and **`==`**. This would have different efficiency properties, but it may be appropriate when you need to maintain data in order.

It is also possible to use any of the other ordered STL containers such as **`std::vector`** as a general-purpose ordered set (as long as you maintain them in a sorted order). Then there are some utility functions in **`<algorithm>`** that can let you compute set union and intersection efficiently.[1] In your real-world projects, you would want to profile the performance of several options and see what works best.

For other "set" data structures like the disjoint-sets (or "union-find") structure or geometric partitioning structures, which have specific use cases, there are no STL classes but you can find robust open-source libraries for C++, or implement your own.

### 3.3 Initialization with type inference

As you examine the provided code, you may see objects being initialized in various ways. Remember that in C++11 and newer, there are compiler features that can infer (detect) the correct types automatically, which can help you write neater code. This was mentioned in the Unordered Map project, but now it's much more useful.

Our **`IntPair`** type, defined in **`IntPair2.h`**, is based on **`std::pair<int,int>`**. The lines below all construct the same **`IntPair`**. The last one uses an "initializer list" in braces , which can be used wherever the type can be understood correctly by the compiler. (This is not the same thing as the "member initializer" lists that can follow a colon **`:`** in a class constructor.)

[1] https://en.cppreference.com/w/cpp/algorithm/set_union

```
IntPair point1 = IntPair(1,2);
IntPair point2 = std::make_pair(1,2);
auto point3 = IntPair(1,2);
IntPair point4 = {1,2};
```

However, if you're not explicit enough, the compilation may fail. The compiler needs to be able to tell if you mean **`IntPair`**, **`std::vector`**, or something else, based on the structure of the variable and how you use it. The following may or may not work, depending on the context of your code:

```
auto point5 = {1,2}; // Probably too vague
```

### 3.4 Implementing adjacency lists

For this project, we'll use the "adjacency lists" model for a graph data structure. We can rely on STL containers to get a somewhat simpler implementation that the one that is explained thoroughly in the lectures. That is, instead of maintaining a linked list of pointers for each vertex, and a separate, shared lookup table of explicit edge records, we will use this scheme:

- Each vertex object has a unique key. We create a map where each vertex has a separate entry based on its own key.
- Each vertex should be mapped to a set of other keys, one key for each adjacent vertex (connected by an edge, implicitly).

We can do this because we guarantee that there are no duplicate vertices, so keys are unique, and because the keys themselves are small, it's efficient enough to store value copies of them instead of pointers to a shared storage structure elsewhere in memory.

Our class **`GridGraph`** uses this internal representation. However, we'll see in the last exercise that in some cases, an even more implicit representation can be achieved, when we do not need to maintain records of all existing vertices and their adjacencies, but rather can calculate that information on demand.

### 3.5 The `GridGraph` class

Our **`GridGraph`** class is defined in **`GridGraph.h`** and **`GridGraph.cpp`**. A **`GridGraph`** is meant to represent undirected graphs on a 2D grid, where vertices are "points" with integer coordinates, each point represented by an **`IntPair`**.

**`IntPair`** is defined in **`IntPair2.h`**, where it is implemented as a **`std::pair<int, int>`**. (We add a few things like **`std::hash`** support.) The first item in the pair is the row index, and the second item is the column index. We want our GridGraph class to only allow for edges between grid points that are one unit apart, so we'll only have horizontal or vertical edges that are exactly one unit in length. For example, we would allow an edge between (0,0) and (1,0), but not between (0,0) and (1,1). We'll also allow for "isolated points," that is, individual points that have no incident edges.

Note that you can display a plot of a **`GridGraph`** in the terminal. We overload the **`<<`** operator so that you can output a **`GridGraph`** directly to **`std::cout`**, and this delegates to the **`plot`** member function by default. If the verbose diagrams in the terminal cause problems with screen readers or for any other reason, you can prevent the plotting output by changing options listed in **`informalTests`** in **`main.cpp`**.

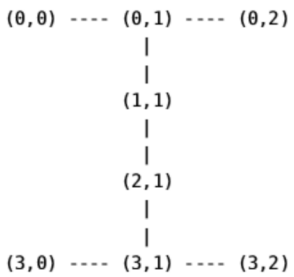


Fig. 2: Example terminal output of an "I"-shaped GridGraph plot.

In the **GridGraph** class namespace, we define a type for the "adjacency set" belonging to a single vertex:

```
using NeighborSet = std::unordered_set<IntPair>;
```

Therefore this type could be referred to globally as **GridGraph::NeighborSet**. Each **GridGraph** graph stores all of its internal data in its member variable **adjacencyMap**, which is declared like this:

```
std::unordered_map<IntPair, GridGraph::NeighborSet> adjacencyMap;
```

That is, the **adjacencyMap** associates one **IntPair**, the key point, with a set of several other **IntPair**, which are all points that are adjacent to the key point, connected to it implicitly by edges. (We say two points are adjacent only if they are connected by an edge. In Fig. 3, although points $(7,2)$ and $(7,3)$ are one space apart, they are not connected by an edge, and therefore not "adjacent.")

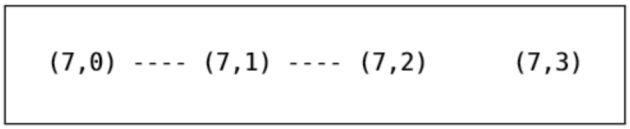


Fig. 3: Some connected points and an isolated point.
$(7,0)$ is connected to $(7,1)$, and $(7,1)$ is connected to $(7,2)$.
$(7,3)$ is an isolated point with no edge connections.

Because **GridGraph** represents *undirected* graphs, if any point X includes a point Y as an adjacency, then Y must also include X as an adjacency. Consider the points shown in Fig. 3. If we look up the point $(7,1)$ in the **adjacencyMap**, we should get a **std::unordered_set** containing the two adjacent points $(7,0)$ and $(7,2)$. If we look up $(7,0)$, we should get a set containing only $(7,1)$. If we look up $(7,3)$, we should get an empty set.

The fact that there is a key in **adjacencyMap** for a given point is enough to declare that the point exists in the graph. Therefore, if we want to insert an isolated point with no edge connections, we would insert the key for that point with an empty value, that is, an empty set of neighbors. If you recall how the unordered map type works, just by referring to a non-existent key with **[]**, we cause it to be created with a default value. This is actually all that **GridGraph::insertPoint** does to insert an isolated point, as you can see in the class definition. **GridGraph::insertEdge** will also initialize the endpoints of the given edge if necessary, so points do not need to be insert individually before inserting an edge.

Let's avoid a point of confusion. Although our graph data structure doesn't store edges explicitly, sometimes we may want to refer to edges separately as objects. For example, we may want to enumerate

all the unique edges in a graph and list them separately. The function **`GridGraph::printDetails`** does this. Then it's good to remember that even as a single **`GridGraph`** point is a pair of two integers, an edge could be represented as a pair of two points. For these reasons, in **`IntPair2.h`** we've also defined **`IntPairPair`** as a pair of **`IntPair`**.

There are some comments in **`IntPair2.h`** warning about unique edge representations. For example, for an edge between $(1, 7)$ and $(1, 8)$, the explicit edge representations $((1, 7), (1, 8))$ and $((1, 8), (1, 7))$ must mean the same edge, but our **`IntPairPair`** type won't automatically consider these to be equal, nor hash them the same way. You need to take care about that if you are trying to enumerate edges explicitly. (It would be possible to make a class type similar to **`IntPairPair`** that ensures that undirected edges are correctly compared automatically, but that's unnecessary to complete this assignment.)

### 3.6 Graph search algorithms overview

In this project you'll be dealing with breadth-first search, but there are some other commonly discussed algorithms and they all have various strengths. We mention them here just for your reference. Note that these can all be used for both directed and undirected graphs, with some modifications.

- **breadth-first search (BFS):** This can be used to find the shortest paths from one vertex to other vertices in the graph, as long as all of the edges have the same length (or "weight," or "cost," which are basically synonyms). Since BFS doesn't consider different edge lengths, it finds the shortest path only in terms of the number of steps.

  This works because as the algorithm expands the frontier of its search, it always visits all of the vertices that are reachable by a length-1 path before it visits any of the vertices that are reachable by a length-2 path (but not a length-1 path), etc. If you are searching for a single goal vertex, then as soon as BFS discovers it, you know you've found a path of shortest length to that vertex. (There may be more than one shortest path between two points, if multiple paths have the same length, and that length is the shortest. In that case, we're just finding *one* of these shortest paths.)

- **Dijkstra's algorithm:** This one is also discussed in this course. It finds the shortest paths from one vertex to all others, where the edge weights can all be different, as long as they have non-negative values.

- **depth-first search (DFS):** This has various creative uses, but note that it won't necessarily find shortest paths. An "iterative deepening" variation on this algorithm can be used for that purpose efficiently, though. That has the benefit of using less memory than BFS.

There are also some other commonly-discussed algorithms that we haven't talked about:

- **A*:** Pronounced "A-star," this algorithm is somewhat similar to Dijkstra's, but it uses an estimation function (called a heuristic) to guess which nodes to try exploring next, with the intention of finding a single goal node and stopping. The heuristic must be carefully designed to ensure correctness. Traditionally, this was the algorithm used for computing shortest paths in video games and so on, but many newer variations on the algorithm exist.

- **Bellman-Ford:** This algorithm is similarly in principle to BFS, but it performs iterations that re-check edges multiple times to see if shorter path information can be updated. Because of this, it is somewhat slow, but it's also compatible with negative edge lengths—unless there is a negative cycle somewhere in the graph. In that case, there may exist no such thing as a shortest path, but this algorithm can at least be used to detect that.

### 3.7 Graph modeling

Many applications of graph theory and graph algorithms deal with things that are not obviously graphs shaped like road maps. Sometimes you'll have to think creatively to realize what the "vertices" and "edges" are for a given application, and what the weights of the edges should be.

One typical application is where a graph represents a sequence of choices or events, where each vertex in the graph represents *state*, that is, the current conditions at any given moment. Then the graph edges would represent possible transitions from one state to another, and the edge weights would signify the relative cost of choosing one option instead of another.

In the simplest case where edges all have the same weight, we can use BFS to compute the shortest path from some initial state to some desired goal state. Many single-player puzzles with no hidden or random game elements, such as a Rubik's cube or sliding tile puzzle, theoretically can be solved this way. In the last exercise we'll look at a simple example.

However, for many problems, even a small puzzle, the number of potential states (and thus graph vertices and edges) is incredibly large, and so an exhaustive graph search can take a very long time and consume lots of memory. That's why in practice, an algorithm like A* (or something more advanced) can perform much better, assuming thee is some way to estimate a cost for a given choice. In our implementation of BFS, we'll at least discuss a few minor improvements that can reduce the memory and preprocessing requirements.

## 4 Exercises

Please look at the starter code in **`GraphSearchExercises.cpp`**. You'll need to write all of your graded code in this file, because the autograder will ignore changes you make to any others.

### 4.1 Exercise 1: Adjacency list utilities

In this exercise, you'll finish implementing the **`GridGraph`** member functions **`countEdges`** and **`removePoint`**. This should help you understand the Week 3 materials and become more comfortable with **`GridGraph`** for the next exercise. To code these, it's okay to use any of the **`GridGraph`** class member functions, or anything else useful from the provided code.

- **`countEdges`**: This function should return the number of unique edges in the GridGraph. Remember that **`GridGraph`** doesn't explicitly store edge objects, but instead it stores point adjacency information. Since the graph edges are undirected, for any two vertices A and B, the edges $(A,B)$ and $(B,A)$ are the same edge. This means if we add up the sizes of the **`GridGraph::NeighborSet`** sets mapped in **`adjacencyMap`**, we'll be double-counting the number of edges. We can still use that to get the answer, or instead, the implementation of **`GridGraph::printDetails`** shows another method, by constructing a set of the unique edges only.

- **`removePoint`**: This function takes a point (an **`IntPair`**) as input, and it should remove all references to that point from the data structure. That means it should remove the input point's key from the adjacency map, but *first* it should remove the edges connected to this point by removing any references to the input point from among the adjacency sets of the other points that were originally adjacent to the input point. You shouldn't change any other data in the graph, such as adding anything new.

### 4.2 Exercise 2: `graphBFS`

You need to finish implementing the **`graphBFS`** function that performs a breadth-first search within a **`GridGraph`**. Note that the provided code is almost complete, and you only need to implement the few parts marked with "TODO." The provided code is heavily documented with comments, and there are some hints about what exactly you need to do.

Here are some significant details about this implementation of BFS; in particular, some aspects that are different from how the algorithm is listed in pseudocode in the lectures:

- This implementation of BFS is geared toward finding the shortest path from a single start point to a single goal point. So, it only explores within a single connected component of the graph (if there are several), and it may report that the goal is unreachable. As soon as the goal point is found, our function stops searching, unlike the pseudocode given in lecture that describes exploring and labeling every node and every edge in every connected component.

- This implementation uses sets of points to implicitly "label" points with some status such as "visited," instead of assigning a label property to a point itself. This lets us associate more than one kind of status with any given point at the same time, by putting the point in several sets. It's convenient to do this with STL containers like **`std::unordered_set`**, since this way, we don't have to add some kind of additional status member variables to our own classes like **`IntPair`** just to record meta-data about the search process. It also means we don't have to initialize a status for every vertex from the beginning. In some graph search problems, such as for solving a puzzle with graph modeling, the number of vertices is extremely large, so we'd rather not have to initialize them all, or even loop over them all just once!

- We use a **`std::unordered_map`**, **`pred`**, to record the predecessor vertex of any newly-discovered vertex during the search. (You may also see this concept referred to as "parent" or "previous" instead of "predecessor.") This implicitly records what the "discovery" edges are, as described in lecture. We can use that to rebuild the path from goal to start (in reverse) when we're done searching. Then it's trivial to arrange the path in the reverse order and return the path from start to goal.

  The terms "visit," "discover," and "explore" are used somewhat interchangeably when discussing graph search. To restate what we're doing here, a node X becomes the predecessor of a node Y if we are currently viewing the adjacencies of node X when we visit Y for the first time, adding Y to the exploration queue. It's as though X has discovered Y, so call this a "discovery edge," and then we know that a shortest path from the start to Y uses this edge.

- We do not assign status directly to edges, and we aren't explicitly storing the edges in a separate structure, only implicitly in the **`adjacencyMap`** that the **`GridGraph`** class has. However, the vertex predecessor information we record during the search is basically revealing the "discovery" edges.

- We use yet another map, **`dist`**, to record information about the shortest-path distance from the start to any given node that has been discovered so far. This is similar to something that Dijkstra's algorithm does, although with BFS, when the edge lengths are all equal, we know that as soon as we discover a node we have found the shortest path to it. We can still use this to detect if we've taken more steps than expected and quit, since in some cases, we know a definite upper bound on how many steps should be required before we give up.

  (For example, you could set the maximum allowed distance to be the same as the number of edges you counted in the graph structure. When there are no negative edge weights, the path can never

be shorter by repeating an edge, and so in the worst case, the shortest path would use every graph edge exactly once. But you may be able to calculate an even more helpful upper bound for some types of problems.)

- Redundantly, we have also created a "dequeued" set of vertices that we use to help you check for mistakes that could cause an infinite loop. This isn't normally part of the BFS algorithm itself. In a way, this is mirroring what the "visited" set is already supposed to accomplish.

Some of these features are mainly intended to help make our implementation of BFS more efficient and less prone to infinite looping mistakes the programmer might make, which could be especially helpful for the next exercise where the graph is much larger. In practice, you would probably want to use a more advanced graph search algorithm for a heavy graph search operation with a single start and single goal. (For example, you might use an iterative version of the A* algorithm mentioned earlier in this document. Some other courses at the University of Illinois discuss these algorithms.) However, the simple principles behind how BFS works also inform many of those other algorithms.

Please test your work with both the **`./main`** and **`./test`** programs as described earlier in this document. When you run **`./main`**, you can see a graphical printout of your shortest path results if you have enabled the Exercise 2 test option listed in **`informalTests`** in **`main.cpp`**.

### 4.3 Exercise 3: `puzzleBFS`

This time, we will use BFS to solve a graph modeling problem. This is where we model a realistic problem in terms of an imaginary graph, and then we can use graph search concepts to solve the modeled problem. This is mostly a conceptual exercise; if you've done the previous exercise, you'll realize there is very little work to do for this one.

The "8 puzzle" is a sliding tile puzzle played in a $3 \times 3$ square grid with eight tiles, labeled 1 through 8, where the ninth space on the grid is blank. Whenever a tile is adjacent to the blank space, we can slide it into the blank space, and this counts as one puzzle move.

```
 -------
| 1 2 3 |
| 4 5 6 |
| 7 8   |
 -------
```

Fig. 4: Example terminal output of a **`PuzzleState`** object in the solved configuration, $[1,2,3,4,5,6,7,8,9]$

When the puzzle is solved, it looks like Fig. 4 above. Reading the numbers row by row, using 9 for the blank space, we can write the state compactly as an array of numbers. So this state is: $[1,2,3,4,5,6,7,8,9]$. We have a class **`PuzzleState`**, defined in **`PuzzleState.h`**, that implements such a puzzle state as a fixed-length array, **`std::array<int, 9>`**. You can think of this like a **`std::vector`** that always has nine items.

When the puzzle begins in a scrambled state, we want to slide the tiles until the grid looks like the solution state again. We can use graph search to find the solution steps. At first, one might think that we would make edges between grid squares or something, but that's not the right approach. Instead, the current state of the entire puzzle grid is a single vertex in our imaginary graph model. This state vertex is connected by implicit edges to other state vertices that can be reached by performing a single puzzle move. For example, beginning from the puzzle state shown in Fig. 5, there are two adjacent states, which are the result states of the two potential moves.


Fig. 5: A **PuzzleState** in the configuration $[9,1,3,4,2,5,7,8,6]$

First, we could slide the 1 to the left. (This is the same idea as swapping the blank space to the right. Our **PuzzleState** member functions refer to the moves as if the blank space itself is moving through the grid.) Then this is the neighboring state we would reach (Fig. 6):


Fig. 6: A **PuzzleState** in the configuration $[1,9,3,4,2,5,7,8,6]$

Or, instead, we could slide the 4 up. (That's the same as swapping the blank space down.) Then we'd reach this state (Fig. 7):


Fig. 7: A **PuzzleState** in the configuration $[4,1,3,9,2,5,7,8,6]$

There are helper functions in the **PuzzleState** class that can query what these neighboring states are for any given state. Since we calculate these adjacent states implicitly, we don't need a separate graph data structure with a map of adjacencies. We also don't need to keep a list of all the possible vertices that exist, although we can "label" vertices as we go by inserting them into sets. It's good that we don't need to label them all from the start, because the number of potential puzzle state configurations is huge!

If we ignore the sliding rules and just rearrange tiles however we want, we could potentially make 362,880 different permutations, but not all of those arrangements are even reachable from the solved state by valid puzzle moves. So simply listing all the vertices in our graph model would be very inefficient, but even if we wanted to make a reduced list of valid puzzle states, that would require figuring out which of all the vertices are reachable, too. Therefore we won't make a list of all the vertices either way. We'll just rely on the start state and goal states we know about, and we'll follow the implicit adjacency information based on the rules of the puzzle.

(There is actually a way to figure out if a given configuration is solvable based on the number of ordering flips.[2] But apart from that, if you recall the "landmark path" problem from the Week 4 lectures, we could initiate BFS from the solved state with no particular goal in mind, and exhaustively explore *every* reachable puzzle configuration. In the process of doing that, we'd find the shortest path solution for every valid puzzle—but this would take a lot of time and memory.)

[2] http://mathworld.wolfram.com/15Puzzle.html

Each puzzle move slides a single tile by a single space, and since we want to find the solution that takes the shortest sequence of moves, we can naively suppose that all potential moves are equally costly, so we can think of all the implicit edges in our graph model having the same weight. Then we can implement **`puzzleBFS`** to perform BFS for the 8 puzzle *very* similarly to **`graphBFS`**.

We can also use the **`dist`** record to see if we've taken too many steps and should give up. In the case of the 8 puzzle, there is a known maximum number of steps that it should take to solve any given puzzle (reportedly 31 steps,[3] but we've hard-coded 35 as the limit). Therefore if we take more steps than that, we know the puzzle can't be solved or we have some bug, and we can immediately quit.

There are other kinds of problems you can model as graphs where such upper bound information is not known. However, the concept of the "best-yet-found" distance record is still useful in many cases, and other graph search algorithms use this to detect errors or optimize where to search next. If we could think of a way to correctly assign a score to potential moves, it might be possible to use one of the other graph search algorithms with much greater efficiency. But, we'll set that aside for now.



[3] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.40.9889