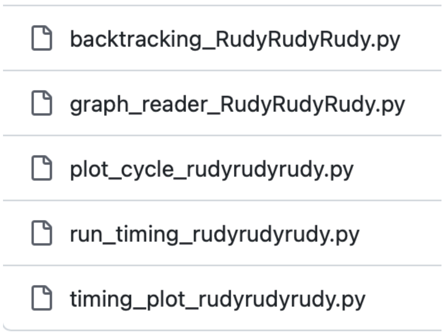
Project ReadMe

●	Team Name: RudyRudyRudy
●	Team members names and netid: Katherine Comito - kcomito Julia Lizak - jlizak
●	Overall project attempted, with sub-projects: Hamiltonian Paths/Cycles - The Traveling Salesman Problem Brute Force and Backtracking
●	Overall success of the project: **Backtracking**: The backtracking was fully successful. The program has a working depth-first search backtracking that finds the Hamiltonian cycles on weighted graphs. It is able to handle graphs with and without the cycles, reads the graph csv, and measures the runtime of each graph. **Brute Force**: Brute force is complete and successful. The program tries every possible ordering of vertices. It checks whether each ordering forms a Hamiltonian cycle or not. I was able to handle graphs with AND without cycles, read the csv test files, all while recording the timing results for each file. Also added some fun ways to visualize the graphs once they were all working correctly. Works cited in the readme file on github.
●	Approximate total time (in hours) to complete: **Backtracking**: 10 hours **Brute Force**: 9 hours
●	Link to GitHub repository: https://github.com/julializak/toc-project1-rudy.git
●	List of included files (if you have many files of a certain types, such as test files of different sizes, list just the folder): (Add more rows as necessary)

File/Folder Name: code	File Contents and Use: python files for all the processing of backtracking and brute force. Additional python files for plotting and visualizing the results

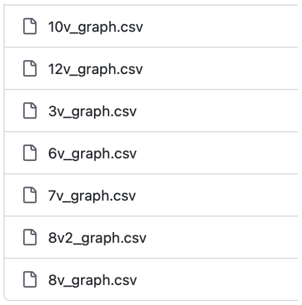
Code Files:



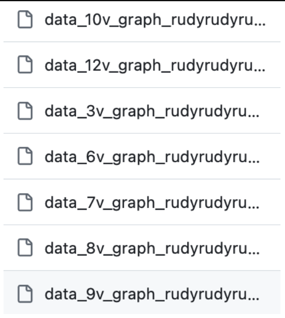
backtracking_RudyRudyRudy.py
graph_reader_RudyRudyRudy.py
plot_cycle_rudyrudyrudy.py
run_timing_rudyrudyrudy.py
timing_plot_rudyrudyrudy.py

File/Folder Name: data	File Contents and Use: All of the test files and different use cases for different parameters and sizes of the graphs

Test Files:



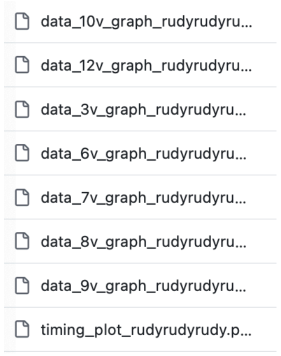
10v_graph.csv
12v_graph.csv
3v_graph.csv
6v_graph.csv
7v_graph.csv
8v2_graph.csv
8v_graph.csv



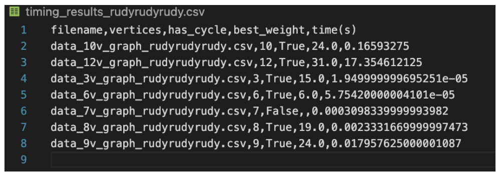
data_10v_graph_rudyrudyru...
data_12v_graph_rudyrudyru...
data_3v_graph_rudyrudyru...
data_6v_graph_rudyrudyru...
data_7v_graph_rudyrudyru...
data_8v_graph_rudyrudyru...
data_9v_graph_rudyrudyru...

File/Folder Name: outputs (and plots)	File Contents and Use: Extra plot visuals for the different vertices graphs + an outputted CSV file for the graph values

Output Files:

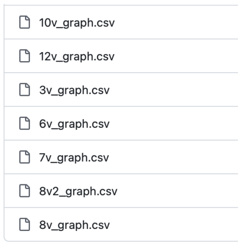


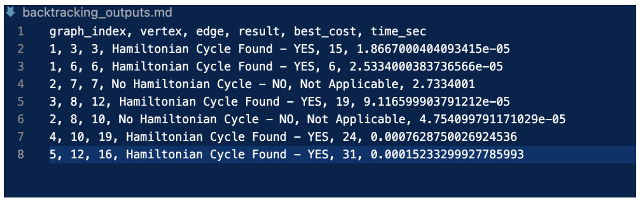
data_10v_graph_rudyrudyru...
data_12v_graph_rudyrudyru...
data_3v_graph_rudyrudyru...
data_6v_graph_rudyrudyru...
data_7v_graph_rudyrudyru...
data_8v_graph_rudyrudyru...
data_9v_graph_rudyrudyru...
timing_plot_rudyrudyrudy.p...



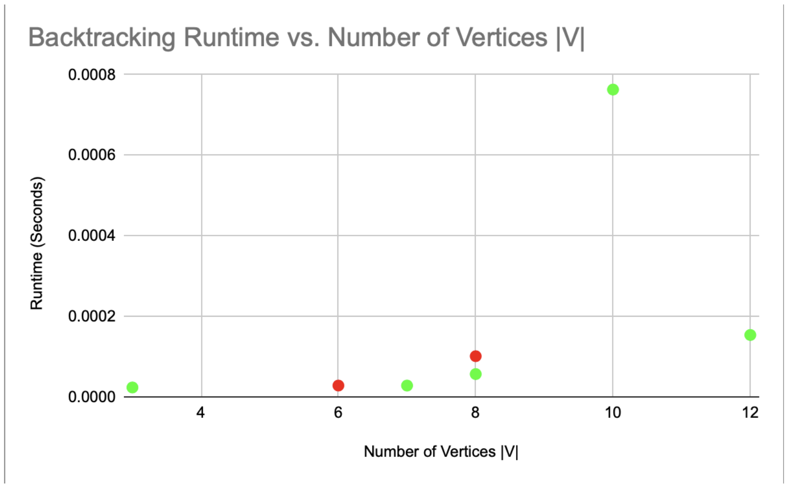
timing_results_rudyrudyrudy.csv

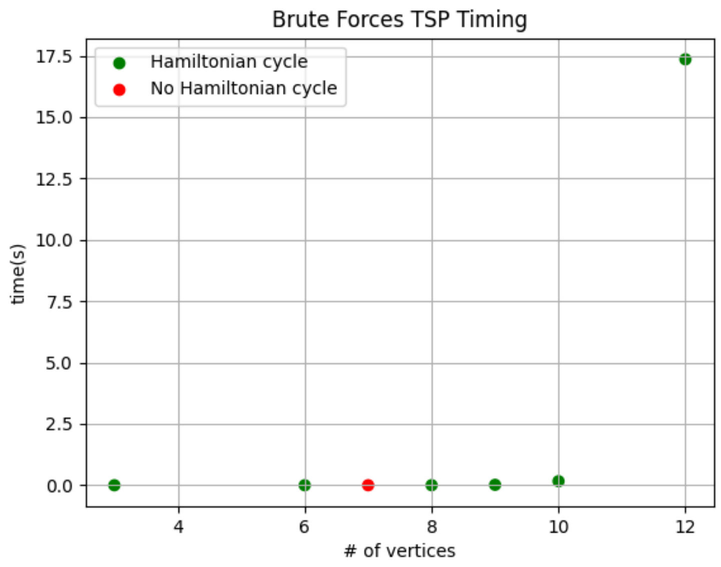
```
filename,vertices,has_cycle,best_weight,time(s)
data_10v_graph_rudyrudyrudy.csv,10,True,24.0,0.16593275
data_12v_graph_rudyrudyrudy.csv,12,True,31.0,17.354612125
data_3v_graph_rudyrudyrudy.csv,3,True,15.0,1.949999999695251e-05
data_6v_graph_rudyrudyrudy.csv,6,True,6.0,5.754200000004101e-05
data_7v_graph_rudyrudyrudy.csv,7,False,,0.0003098339999993982
data_8v_graph_rudyrudyrudy.csv,8,True,19.0,0.0023331669999997473
data_9v_graph_rudyrudyrudy.csv,9,True,24.0,0.017957625000001087
```


10v_graph.csv
12v_graph.csv
3v_graph.csv
6v_graph.csv
7v_graph.csv
8v2_graph.csv
8v_graph.csv


backtracking_outputs.md

```
graph_index, vertex, edge, result, best_cost, time_sec
1, 3, 3, Hamiltonian Cycle Found – YES, 15, 1.8667000404093415e-05
1, 6, 6, Hamiltonian Cycle Found – YES, 6, 2.5334000383736566e-05
2, 7, 7, No Hamiltonian Cycle – NO, Not Applicable, 2.7334001
3, 8, 12, Hamiltonian Cycle Found – YES, 19, 9.116599903791212e-05
2, 8, 10, No Hamiltonian Cycle – NO, Not Applicable, 4.754099791171029e-05
4, 10, 19, Hamiltonian Cycle Found – YES, 24, 0.0007628750026924536
5, 12, 16, Hamiltonian Cycle Found – YES, 31, 0.00015233299927785993
```





File/Folder Name: readme	File Contents and Use: Cited sources and project readme files

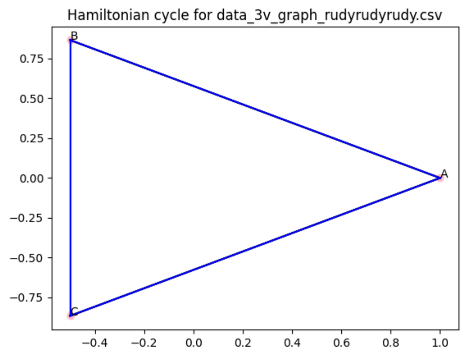
●

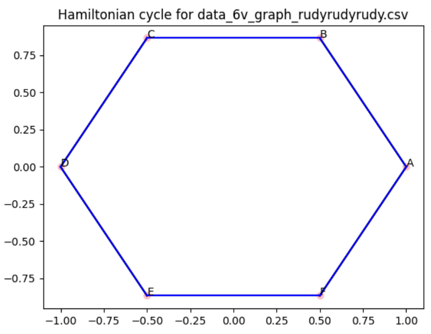
Programming languages used, and associated libraries:
Programming language: Python
Libraries: NetWorkx, time, csv, sys, itertools, math, time, matplotlib.pyploy

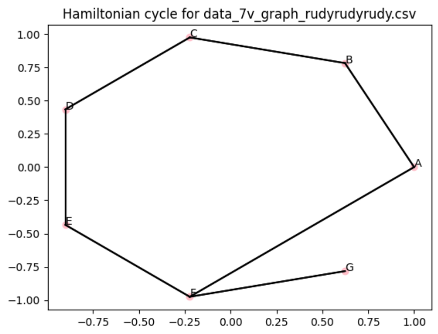
●	Key data structures (for each sub-project): Backtracking: Lists to store the current path during depth-first search, set used to keep track of all the visited vertices, and dictionaries to hold the best route and smallest cost of route. Brute Force - Lists: held each potential option for the cycles - Dictionaries: held the adjacent values for weighting and to help look up edges - Tuples: too from library to help with the permutations - Floats/ints: Helped for the weight values and counts
●	General operation of code (for each subproject): Backtracking: I used a depth-first search algorithm to find a Hamiltonian cycle. At the chosen start vertex the program extends the current path by trying all unvisited neighboring vertices. It would then track the total cost of the current partial path and remove a branch if the cost of a path exceeds the cheapest route found so far. After the path includes all the vertices to be listed, it checks if an edge returns to the start to complete the cycle. The best solution would be if the completed cycle had the lower cost. Finally, backtrack and remove the last vertex to continue exploring other possibilities. Traveling Salesman: The program reads the graphs from files, runs the solver, times the execution then prints the result. Brute Force - I read in the CSV file and put it into a vertex list and adjacency dictionary - Ran through to find all possible ways to make the cycle with the vertices - Through each of these, made a verifying function to check that the current pair did have an edge - This is also where the weight was calculated - Then kept track of the current best cycle option (accounting for the weight) - Returned the best cycle and weight, then calculated the time for how long each cycle took - Made an output CSV file for the weight, time, and exact cycle
●	What test cases you used/added, why you used them, what did they tell you about the correctness of your code? Backtracking: I began with a 3-vertex triangle to confirm the solver was functioning correctly. I then tested larger graphs, 6-vertex, 7-vertex (no cycle), an 8-vertex graph with no cycle, an 8-vertex graph with a cycle, a 10-vertex graph, and finally a 12-vertex graph. The smaller graphs confirmed

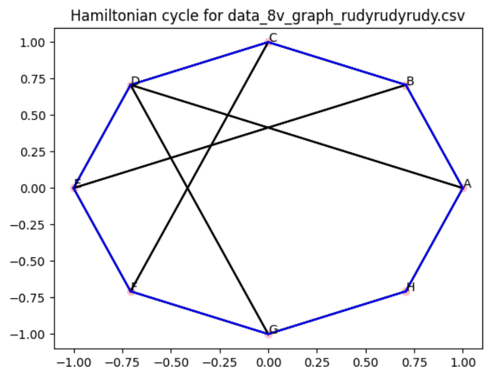
	that the algorithm produced the correct cycle or correctly identified when no cycle existed. The larger graphs demonstrated how runtime increases with the number of vertices, which aligns with the exponential worst-case complexity of backtracking. Brute Force: Pretty much the same test files for each. Had a 3 vertex triangle, 6 vertex, 7 vertex (but with no cycle), 8, 9, 10, and 12 vertex graphs. The larger graphs were used to show how it is an exponential growth. This variety of graphs showed that the brute force works for both types of graphs and the run time increases pretty quickly.
●	How did you manage the code development? Backtracking: I implemented easier functions to tackle first, like edge_weight and edge_exists. I struggled with recursion so I wanted to have enough time to work on it so I did that next. Next I added in small text graphs to verify minimum correctness before moving on. Then added the full graph file for the project. I then looked at google images to find Hamiltonian graphs to add into as test cases. Finally adding the timing measurement and printing outputs. Brute Force: - Worked slowly and tested almost line by line or function by function - First just made sure i could read the graphs and any csv files I was importing - Then started implementing the way to cycle through all of the different options for a cycle/path - Implemented a verifier to triple check that each pair of vertices did in fact have an edge between them - Once getting good results and I could verify with my own knowledge and reading the outputs, I decided to try implementing actual visuals to display - I just created separate python files to try not to mess with the main file. This is where I did the timing, plotting, and creating the visuals.
13	Detailed discussion of results: Backtracking: The runtime increased as the number of vertices increased. After running all graph files, I created an Excel scatter plot of runtime vs. |v|. Green markers were used for graphs where a Hamiltonian cycle was found, and red markers were used when no cycle existed. The resulting plot showed a noticeable upward trend as |v| increased, showing the exponential growth characteristic of backtracking approaches to NP-hard problems like the Hamiltonian cycle problem. This confirmed that the solver behaves as expected. Brute Force: - For the smaller size graphs (like 3-6) the code ran pretty fast

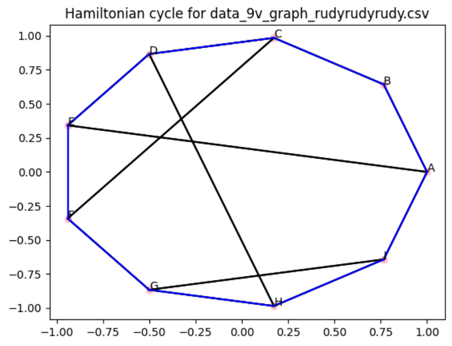
	(~0.0000195), but as we got up to about 8-12 vertices, it definitely started slowing down. - The 12 vertex graph took ~17 seconds, which really does show that exponential growth - For an example of no hamiltonian cycle, we used the 7 vertex graph - All the cycles for the complete and solvable graphs were also able to be outputted to a visual so we can see the actual path/cycle
14	How team was organized: Katherine: In charge of the backtracking code, test graph files for backtracking, and collecting timing data for each test graph file Julia: In charge of the brute force algorithm, test files, the timing script, produces a “result” timing csv file and plot, as well as creating some visuals for all the graph cycles. Helped put together the readme file.
15	What might you do differently if you did the project again? Katherine: I would try to explore other algorithms that could be useful instead of using depth-first search. I wanted to stick with algorithms I was familiar with so I only looked at doing depth-first, breadth-first, or Dykstra's for potential options to solve the problem. Julia: I would definitely try more strategies to cut down on the timing. Obviously if I had more time, I could try to write up and learn a few more algorithms and compare the results, but I tried to just stick with what I was comfortable with. Also, I would maybe try to look at more libraries within Python. I am sure there are a few out there that would definitely help out with efficiency.
16	Any additional material: Graph Visual Tool - Separate plotting script - Draws out each test graph with the vertices on an approximate circle, and also the cycle lines (if it exists) - Outputted this into a PNG that automatically saves into the local folder I was connected to through VSCode Automated the process of reading the input graph files - Looked up how sort of use regex/search for specific pathnames in python - Came across the glob library (pattern matching) - I think this would be helpful for future projects because it definitely speeds up the process of having to constantly upload/input the file names when testing (I am sure all the other coders in the class think this is light work, but I thought it was cool)

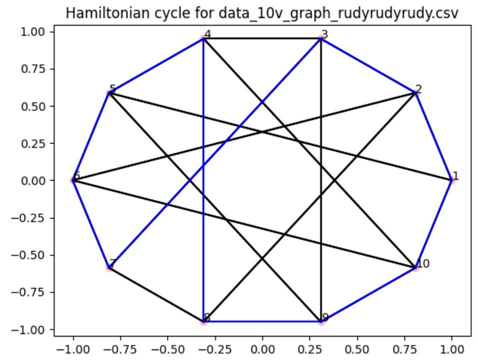

Hamiltonian cycle for data_3v_graph_rudyrudyrudy.csv
B
A
C

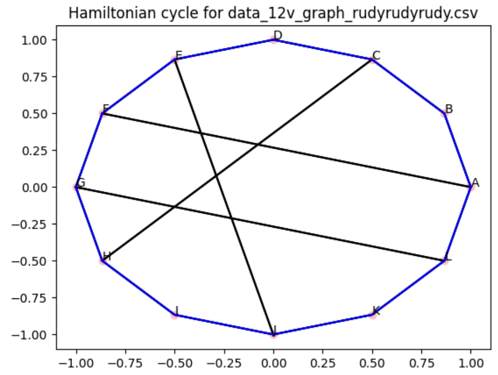

Hamiltonian cycle for data_6v_graph_rudyrudyrudy.csv
C
B
D
A
E
F


Hamiltonian cycle for data_7v_graph_rudyrudyrudy.csv
C
B
D
A
E
G
F


Hamiltonian cycle for data_8v_graph_rudyrudyrudy.csv
C
D
B
E
A
F
H
G


Hamiltonian cycle for data_9v_graph_rudyrudyrudy.csv
D
C
B
E
A
F
I
G
H


Hamiltonian cycle for data_10v_graph_rudyrudyrudy.csv
4
3
5
2
6
1
7
10
8
9


Hamiltonian cycle for data_12v_graph_rudyrudyrudy.csv
E
D
C
F
B
G
A
H
L
I
K
J