Project 10: Graphs Part 2

Due: Thursday, April 21st @ 10:00 pm

This is not a team project, do not copy someone else's work.

This project is created by Brooke Osterkamp, Andrew Haas, Tanawan Premsri, Andrew McDonald

Assignment Overview

What is A* Search?

Instead of searching the "next closest" vertex as is done in Dijkstra's algorithm, A* Search (A-Star Search) picks the vertex which is "next closest to the goal" by weighting vertices more cleverly.

Recall that in Dijkstra's algorithm, vertices are stored in a priority queue with a priority key equal to the current shortest path to that vertex. If we denote the current shortest path to a vertex *V* by **g(v)**, then on each iteration of Dijkstra's algorithm, we search on the vertex with **min(g(v)).**

A* search takes the same approach to selecting the next vertex, but instead of setting the priority key of a vertex equal to **g(v)** and selecting **min(g(v))**, it uses the value **f(v)** and selects the vertex with **min(f(v))** where

f(v) = g(v) + h(v)

= current_shortest_path_to_v + estimated_distance_between_v_and_target

In English, Please...

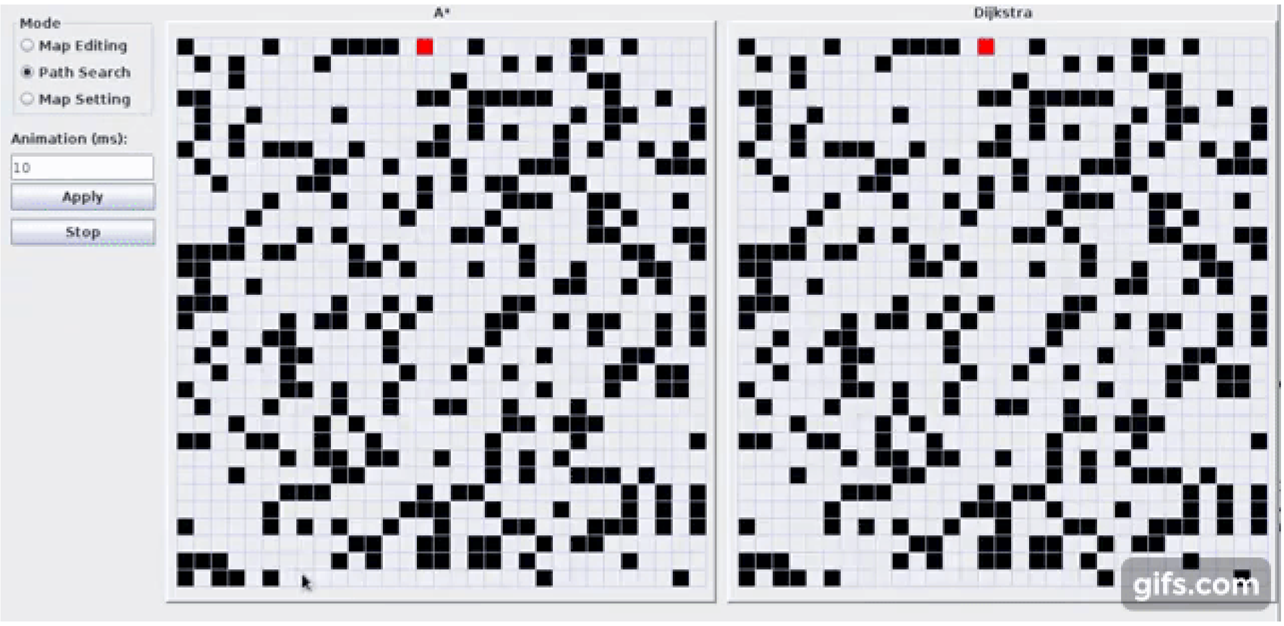
A* Search prioritizes vertices *V* to search based on the value **f(v)**, which is the sum of

- **g(v)**, or the current shortest known path to vertex *V*, and
- **h(v)**, which is the estimated (Euclidean or Taxicab) distance between the vertex *V* and the target vertex you're searching for

The result is that A* prioritizes vertices to search that (1) are *close to the origin along a known path* AND which (2) are *in the right direction towards the target.* Vertices with a small **g(v)** are *close to the origin along a known path* and vertices with a small **h(v)** are *in the right direction towards the target**,*** so we pick vertices with the smallest sum of these two values.

We strongly recommend you watch this video to build your intuition behind A* Search!

[A* is extremely versatile. Here we use Euclidean and Taxicab distances to prioritize certain directions of search, but note that any metric **h(v, target)** could be used should the need arise. See here for more information on situations where different metrics may be practical.] For additional information on Graphs, Shortest path and A* please go to D2L-Week-10-11



Assignment Notes

- A plotting function is provided to help you visualize the progression of various search algorithms
  - Be sure to read the specs explaining **plot()**
  - If you don't want to use it, just comment out the related import statements and **plot()** function
- Python allows representation of the value infinity using **float('inf')**
- No negative edge weights will ever be added to the graph
  - All edge weights are numeric values greater than or equal to zero
- Time complexities are specified in terms of *V* and *E*, where *V* represents the number of vertices in the graph and *E* represents the number of edges in a graph
  - Recall that E is bounded above by *V^2*; a graph has *E* = *V^2* edges if and only if every vertex is connected to every other vertex
- Recall that **list.insert(0, element)** and **list.pop(0)** are both *O(N)* calls on a Python list
  - Recall that python's 'lists' are not lists in the more common sense of the word: linked lists. They are dynamically managed tuples, stored in memory as contiguous arrays of pointers to elements elsewhere in memory. This allows indexing into a 'list' in constant time. The downside of this, however, is that adding to a python 'list' at a specific index, *i*, requires shifting the pointer to every element past *i* by one in the underlying array: a linear operation.
  - Be careful when implementing **dijkstra, astar,** and the Application Problem to ensure you do not break time complexity by popping or inserting from the front of a list when reconstructing a path
  - Instead of inserting into / popping from the front of the list, simply append to or pop from the end, then reverse the list *once* at the end
    - If you have N calls to **list.insert(0, element)**, that is *O(N^2)*
    - If you instead have N calls to **list.append(element)**, followed by a single call to **list.reverse()**, that is *O(N)*
    - Both methods will result in the same list being constructed, but the second is far more efficient

Assignment Specifications

class Vertex

Represents a vertex object, the building block of a graph.

DO NOT MODIFY the following attributes/functions

- **Attributes**
  - **id:** A string used to uniquely identify a vertex
  - **adj:** A dictionary of type **{other_id : number}** which represents the connections of a vertex to other vertices; the existence of an entry with key **other_i****d** indicates connection from this vertex to the vertex with id **other_id** by an edge with weight **number**
    - Note that as of Python 3.7, insertion ordering in normal dictionaries is guaranteed and ensures traversals will select the next neighbor to visit deterministically
  - **visited:** A boolean flag used in search algorithms to indicate that the vertex has been visited
  - **x:** The x-position of a vertex (used in assignment) (defaults to zero)
  - **y:** The y-position of a vertex (used in assignment) (defaults to zero)
- **__init__(self, idx: str, x: float=0, y: float=0) -> None:**
  - Constructs a Vertex object
- **__eq__(self, other: Vertex) -> bool:**
  - Compares this vertex for equality with another vertex
- **__repr__(self) -> str:**
  - Represents the vertex as a string for debugging
- **__str__(self) -> str:**
  - Represents the vertex as a string for debugging
- **__hash__(self) -> int:**
  - Allows the vertex to be hashed into a set; used in unit tests

USE the following function however you'd like

- **deg(self) -> int:**
  - Returns the number of outgoing edges from this vertex; i.e., the outgoing degree of this vertex
  - *Time Complexity: O(1)*
  - *Space Complexity: O(1)*
- **get_outgoing_edges(self) -> Set[Tuple[str, float]]:**
  - Returns a **set** of tuples representing outgoing edges from this vertex
  - Edges are represented as tuples **(other_id, weight)** where
    - **other_id** is the unique string id of the destination vertex
    - **weight** is the weight of the edge connecting this vertex to the other vertex
  - Returns an empty set if this vertex has no outgoing edges
  - *Time Complexity: O(degV)*
  - *Space Complexity: O(degV)*
- **euclidean_distance(self, other: Vertex) -> float:**
  - Returns the euclidean distance [based on two-dimensional coordinates] between this vertex and vertex **other**
  - Used in AStar
  - *Time Complexity: O(1)*
  - *Space Complexity: O(1)*
- **taxicab_distance(self, other: Vertex) -> float:**
  - Returns the taxicab distance [based on two-dimensional coordinates] between this vertex and vertex **other**

- Used in AStar
- *Time Complexity: O(1)*
- *Space Complexity: O(1)*

class Graph

Represents a graph object

DO NOT MODIFY the following attributes/functions

- **Attributes**
  - **size:** The number of vertices in the graph
  - **vertices:** A dictionary of type **{id : Vertex}** storing the vertices of the graph, where **id** represents the unique string id of a **Vertex** object
    - Note that as of Python 3.7, insertion ordering in normal dictionaries is guaranteed and ensures **get_edges(self)** and **get_vertices(self)** will return deterministically ordered lists
  - **plot_show**: If true, calls to **plot()** display a rendering of the graph in matplotlib; if false, all calls to **plot()** are ignored (see **plot()** below)
  - **plot_delay**: Length of delay in **plot()** (see **plot()** below)
- **__init__(self, plt_show: bool=False) -> None:**
  - Constructs a Graph object
  - Sets **self.plot_show** to False by default
- **__eq__(self, other: Graph) -> bool:**
  - Compares this graph for equality with another graph
- **__repr__(self) -> str:**
  - Represents the graph as a string for debugging
- **__str__(self) -> str:**
  - Represents the graph as a string for debugging
- **add_to_graph(self, start_id: str, dest_id: str=None, weight: float=0) -> float:**
  - Adds a vertex / vertices / edge to the graph
    - Adds a vertex with id **start_id** to the graph if no such vertex exists
    - Adds a vertex with id **dest_id** to the graph if no such vertex exists and **dest_id** is not None
    - Adds an edge with weight **weight** if **dest_id** is not None
  - If a vertex with id **start_id** or **dest_id** already exists in the graph, this function will not overwrite that vertex with a new one
  - If an edge already exists from vertex with id **start_id** to vertex with id **dest_id**, this function will overwrite the weight of that edge
- **matrix2graph(self, matrix: Matrix) -> None:**
  - Constructs a graph from a given adjacency matrix representation

    - **matrix** is guaranteed to be a square 2D list (i.e. list of lists where # rows = # columns), of size **[V+1]** x **[V+1]**
        - **matrix[0][0]** is None
        - The first row and first column of **matrix** hold string ids of vertices to be added to the graph and are symmetric, i.e. **matrix[i][0] = matrix[0][i]** for i = 1, ..., n
        - **matrix[i][j]** is None if no edge exists from the vertex **matrix[i][0]** to the vertex **matrix[0][j]**
        - **matrix[i][j]** is a **number** if an edge exists from the vertex **matrix[i][0]** to the vertex **matrix[0][j]** with weight **number**

- **graph2matrix(self) -> None:**
    - Constructs and returns an adjacency matrix from a graph
    - The output matches the format of matrices described in **matrix2graph**
    - If the graph is empty, returns **None**

- **graph2csv(self, filepath: str) -> None:**
    - Encodes the graph (if non-empty) in a csv file at the given location

- **reset_vertices(self) -> None:**
    - Resets visited flags to False of all vertices within the graph
    - Used in unit tests to reset graphs between tests
    - Time Complexity: O(V)
    - Space Complexity: O(V)

- **get_vertex_by_id(self, v_id: str) -> Vertex:**
    - Returns the vertex object with id v_id if it exists in the graph
    - Returns None if no vertex with unique id v_id exists
    - Time Complexity: O(1)
    - Space Complexity: O(1)

- **get_all_vertices(self) -> Set[Vertex]:**
    - Returns a set of all Vertex objects held in the graph
    - Returns an empty set if no vertices are held in the graph
    - Time Complexity: O(V)
    - Space Complexity: O(V)

- **get_edge_by_ids(self, begin_id: str, end_id: str) -> Tuple[str, str, float]:**
    - Returns the edge connecting the vertex with id begin_id to the vertex with id end_id in a tuple of the form (begin_id, end_id, weight)
    - If the edge or either of the associated vertices does not exist in the graph, returns None
    - Time Complexity: O(1)
    - Space Complexity: O(1)

- **get_all_edges(self) -> Set[Tuple[str, str, float]]:**

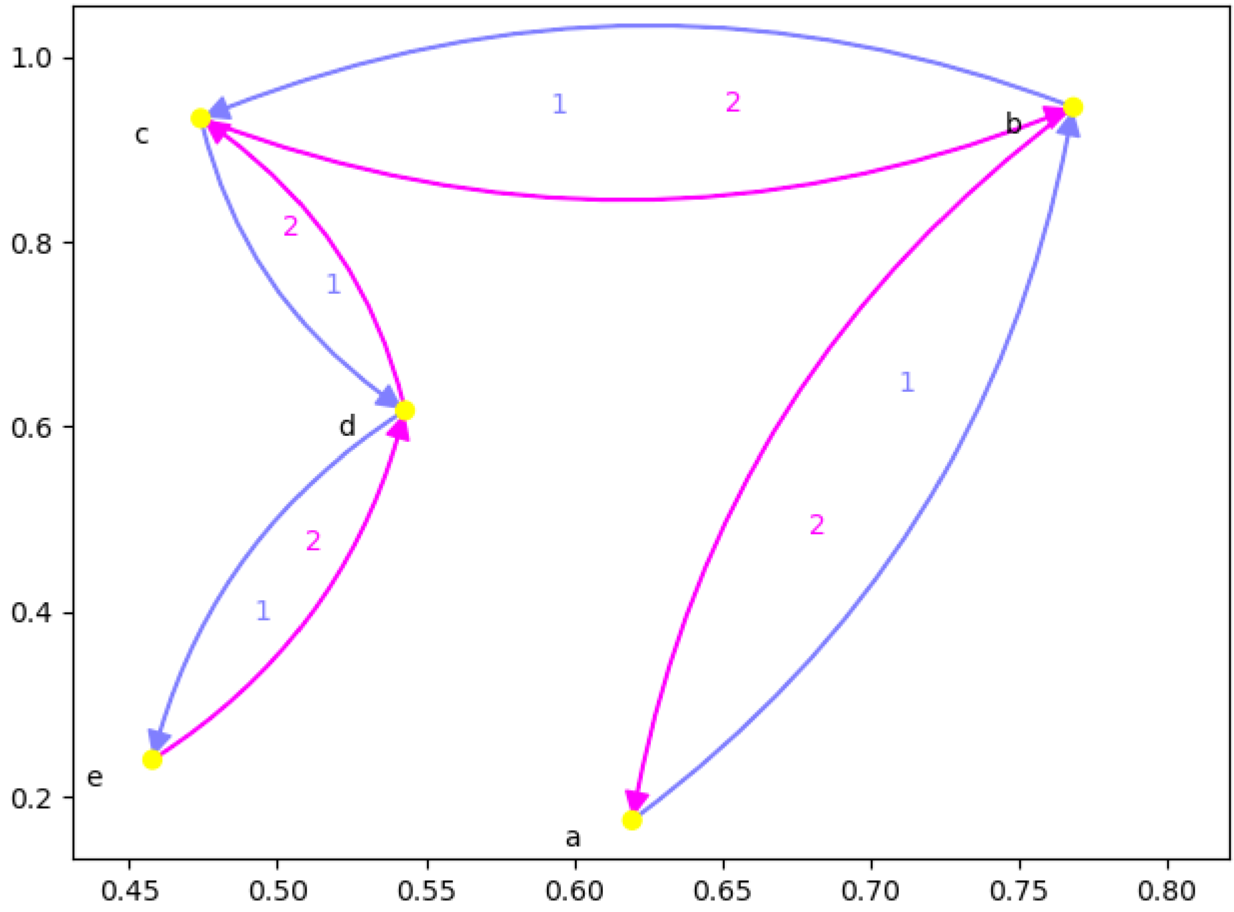
- Returns a set of tuples representing all edges within the graph
- Edges are represented as tuples (begin_id, end_id, weight) where
  - begin_id is the unique string id of the starting vertex
  - end_id is the unique string id of the starting vertex
  - weight is the weight of the edge connecitng the starting vertex to the destination vertex
- Returns an empty set if the graph is empty
- Time Complexity: O(V+E)
- Space Complexity: O(E)

- **build_path(self, back_edges: Dict[str, str], begin_id: str, end_id: str) -> Tuple[List[str], float]:**

  - Given a dictionary of back-edges (a mapping of vertex id to predecessor vertex id), reconstructs the path from start_id to end_id and computes the total distance
  - Returns tuple of the form ([path], distance) where [path] is a list of vertex id strings beginning with begin_id, terminating with end_id, and including the ids of all intermediate vertices connecting the two distance is the sum of the weights of the edges along the [path] traveled
  - Handle the cases where no path exists from vertex with id begin_id to vertex with end_id or if one of the vertices does not exist
  - Time Complexity: O(V)
  - Space Complexity: O(V)

USE the following function however you'd like

- **plot(self) -> None:**
  - Renders a visual representation of the graph using matplotlib and displays graphic in PyCharm
    - Follow this tutorial to install matplotlib and numpy if you do not have them, or follow the tooltip auto-suggested by PyCharm
  - Provided for use in debugging
  - If you call this in your searches and **self.**plot_show** is true, the search process will be animated in successive plot renderings (with time between frames controlled by **self.plot_delay**)
  - Not tested in any testcases
    - All testcase graphs are constructed with **self.plot_show** set to False
  - If vertices have (x,y) coordinates specified, they will be plotted at those locations
  - If vertices do not have (x,y) coordinates specified, they will be plotted at a random point on the unit circle
  - To install the necessary packages (matplotlib and numpy), follow the auto-suggestions provided by PyCharm
  - Vertices and edges are labeled; edges are color-coded by weight
    - If a bi-directional edge exists between vertices, two color-coded weights will be displayed



- **reset_vertices(self) -> None:**
  - Resets visited flags of all vertices within the graph
  - Used in unit tests to reset graph between searches
  - *Time Complexity: O(V)*
  - *Space Complexity: O(V)*
- **get_vertex(self, vertex_id: str) -> Vertex:**
  - Returns the Vertex object with id **vertex_id** if it exists in the graph
  - Returns None if no vertex with unique id **vertex_id** exists
  - *Time Complexity: O(1)*
  - *Space Complexity: O(1)*
- **get_vertices(self) -> Set[Vertex]:**
  - Returns a **set** of all Vertex objects held in the graph
  - Returns an empty set if no vertices are held in the graph
  - *Time Complexity: O(V)*
  - *Space Complexity: O(V)*
- **get_edge(self, start_id: str, dest_id: str) -> Tuple[str, str, float]:**
  - Returns the edge connecting the vertex with id **start_id** to the vertex with id **dest_id** in a tuple of the form **(start_id, dest_id, weight)**
  - If the edge or either of the associated vertices does not exist in the graph, returns **None**
  - *Time Complexity: O(1)*
  - *Space Complexity: O(1)*
- **get_edges(self) -> Set[Tuple[str, str, float]]:**
  - Returns a **set** of tuples representing all edges within the graph
  - Edges are represented as tuples **(start_id, other_id, weight)** where

    - **start_id** is the unique string id of the starting vertex
    - **other_id** is the unique string id of the destination vertex
    - **weight** is the weight of the edge connecting the starting vertex to the destination vertex
  - Returns an empty set if the graph is empty
  - *Time Complexity: O(V+E)*
  - *Space Complexity: O(E)*

IMPLEMENT the following functions

- **dijkstra(self, begin_id: str, end_id: str) -> Tuple[List[str], float]:**
  - Perform a dijkstra search beginning at vertex with id **begin_id** and terminating at vertex with id **end_id**
  - As you explore from each vertex, iterate over neighbors using **vertex.adj** (not vertex.get_edges()) to ensure neighbors are visited in proper order
  - Returns tuple of the form **([path], distance)** where
    - **[path]** is a list of vertex id strings beginning with **begin_id**, terminating with **end_id**, and including the ids of all intermediate vertices connecting the two
    - **distance** is the sum of the weights of the edges along the **[path]** traveled
  - If no path exists from vertex with id **begin_id** to vertex with **end_id** or if one of the vertices does not exist, returns tuple **([],0)**
  - Guaranteed that **begin_id != end_id** (since that would be a trivial path)
  - Because our adjacency maps use insertion ordering, neighbors will be visited in a deterministic order, and thus you do not need to worry about the order in which you visit neighbor vertices at the same depth
  - Use the given PriorityQueue class to simplify priority key updates in a search priority queue
  - *Time Complexity: O((E + V) log V)*
  - *Space Complexity: O(V)*
- **a_star(self, begin_id: str, end_id: str, metric: Callable[[Vertex, Vertex], float]) -> Tuple[List[str], float]**
  - Perform an A* search beginning at vertex with id **begin_id** and terminating at vertex with id **end_id**
  - As you explore from each vertex, iterate over neighbors using **vertex.adj** (not vertex.get_edges()) to ensure neighbors are visited in proper order
  - Use the **metric** parameter to estimate h(v), the remaining distance, at each vertex
    - This is a callable parameter and will always be **Vertex.euclidean_distance** or **Vertex.taxicab_distance**
  - Returns tuple of the form **([path], distance)** where
    - **[path]** is a list of vertex id strings beginning with **begin_id**, terminating with **end_id**, and including the ids of all intermediate vertices connecting the two
    - **distance** is the sum of the weights of the edges along the **[path]** traveled
  - If no path exists from vertex with id **begin_id** to vertex with **end_id** or if one of the vertices does not exist, returns tuple **([],0)**
  - Guaranteed that **begin_id != end_id** (since that would be a trivial path)
  - Recall that vertices are prioritized in increasing order of **f(v) = g(v) + h(v)**, where
    - **g(v)** is the shortest known path to this vertex
    - **h(v)** is the Euclidean distance from *V* to the target vertex
  - Use the given PriorityQueue class to simplify priority key updates in a search priority queue

  - **Implementations of this function which do not utilize the heuristic metric will not receive any credit, visible and hidden testcases included.**
    - **Do not simply implement Dijkstra's Algorithm**
  - *Time Complexity: O((E + V)log(V))*
  - *Space Complexity: O(V)*

To simplify the operation of updating a priority key in your search priority queue, use the following given class.

class PriorityQueue

DO NOT MODIFY the following attributes/functions

- **Attributes**
  - **data:** Underlying data list of priority queue
  - **locator:** Dictionary to locate vertices within the priority queue
  - **counter:** Used to break ties between vertices
- **__init__(self)**
  - Constructs an AStarPriorityQueue object
- **__repr__(self)**
  - Represents the priority queue as a string for debugging
- **__str__(self)**
  - Represents the priority queue as a string for debugging
- **empty(self)**
  - Returns boolean indicating whether priority queue is empty
- **push(self, priority, vertex)**
  - Push the **vertex** object onto the priority queue with a given **priority** key
  - This priority queue has been specially designed to hold Vertex objects as values ranked by priority keys; be sure you push Vertex objects and NOT vertex id strings onto the queue
- **pop(self)**
  - Visit, remove and return the Vertex object with the highest priority (i.e. lowest priority key) as a **(priority, vertex)** tuple
- **update(self, new_priority, vertex)**
  - Update the priority of the **vertex** object in the queue to have a **new_priority**

Application Problem

After finishing your school year for the summer, you decide to take a vacation to the lovely toll-ridden city of Chicago - modeled by a road network with **v vertices and e edges.** After cleaning out your dorm room, you find you have **m EZ-pass coupons** to spend that cover the cost of **half of any toll segment**, i.e., you have m EZ-pass coupons which effectively set the toll of a road segment to HALF.

Your goal is to devise an algorithm to find a route from point **A** to point **B** on the toll roads of Chicago for the cheapest possible overall toll for your vacation.

- **tollways_algorithm(self, start_id: str, target_id: str, coupons: int) -> Tuple[float, float]**
  - Perform an toll algorithm beginning at vertex with id **start_id** and terminating at vertex with id **target_id**
  - Assume the distance between paths is the cost of going from one to another

- As you explore from each vertex, iterate over neighbors using **vertex.adj** (not vertex.get_edges()) to ensure neighbors are visited in proper order
- The **coupon** parameter is the maximum number of coupons
  - This parameter will always be an integer
- Returns tuple of the form **(cost, coupons)** where
  - **cost** is the cost of the best path beginning with **start_id**, terminating with **target_id**
  - When halfing coupon cost use integer division
  - float('inf') should be used for positive infinity
  - **coupons** is the sum of the number of coupons used along the best path traveled
- If no path exists from vertex with id **start_id** to vertex with **target_id** or if one of the vertices does not exist, returns tuple **(None,None)**
- Guaranteed that **start_id != target_id** (since that would be a trivial path)
- Use the given TollWayPriorityQueue class to simplify priority key updates in a search priority queue
- *Time Complexity: O((C*V+E)*log(V*C)), C is the number of coupons*
- *Space Complexity: O(V*C), C is the number of coupons*
- You can only use 1 coupon per edge
  - consider A -- 100 -- B -- 2 -- C example with 2 coupons, you can't apply both coupons to the 100 edge, you would have to apply 1 coupon to each, so the best you could do would be cost 51
- **HINTS**:
  - Use priority queue and dictionary similar to Dijkstra's but storing {cost, coupons used to reach the node, node} rather than just {distance, node}
  - When constructing the dictionary and priority queue think of a matrix V vertices rows by C columns starting at 0 coupons, similar to Cartesian Product

To simplify the operation of updating a priority key in your search priority queue, use the following given class.

class TollWayPriorityQueue:

DO NOT MODIFY the following attributes/functions

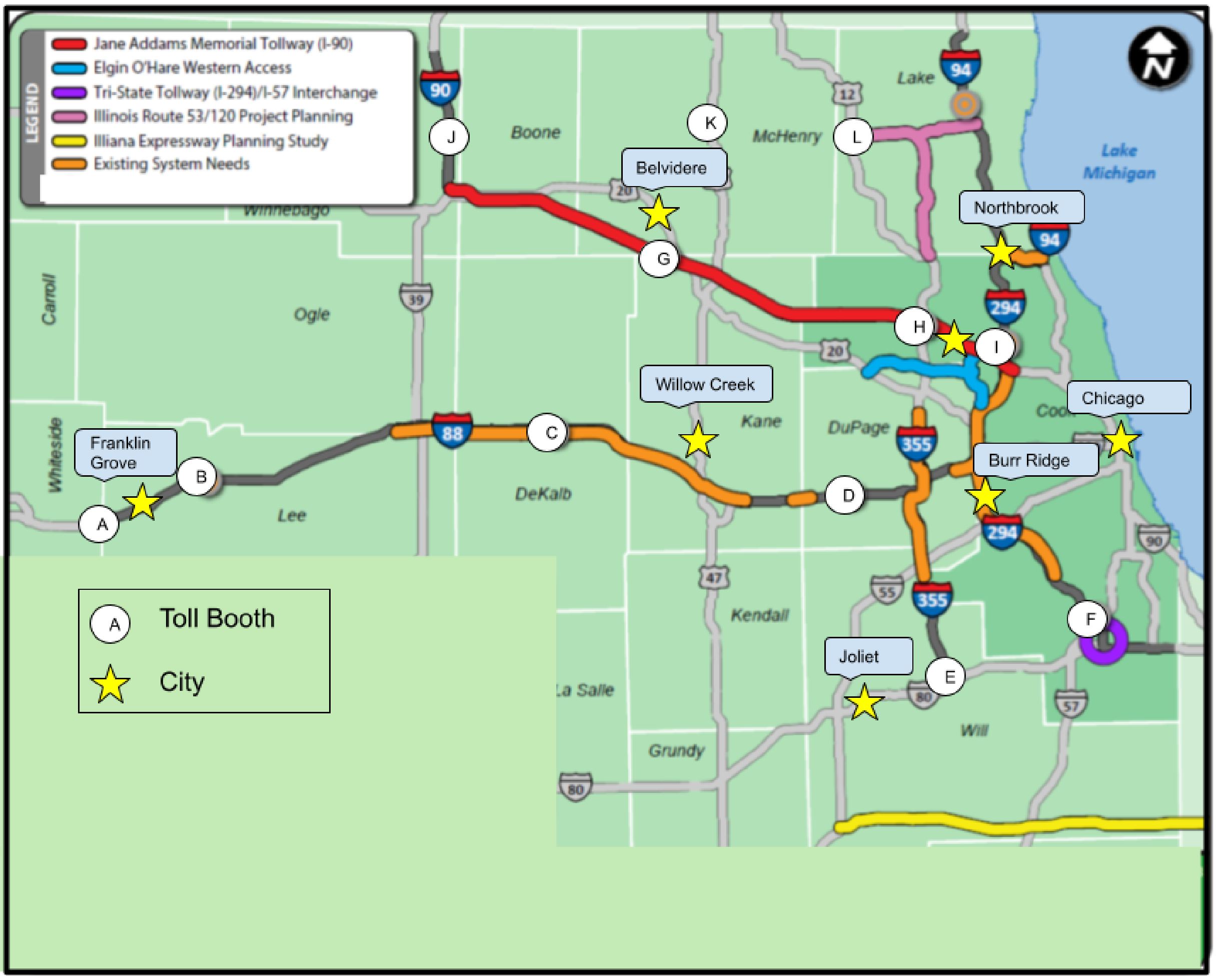
- **Attributes**
  - **data:** Underlying data list of priority queue
  - **locator:** Dictionary to locate vertices within the priority queue
  - **counter:** Used to break ties between vertices
- **__init__(self)**
  - Constructs an TollWayPriorityQueue object
- **__repr__(self)**
  - Represents the priority queue as a string for debugging
- **__str__(self)**
  - Represents the priority queue as a string for debugging
- **empty(self)**
  - Returns boolean indicating whether priority queue is empty
- **push(self, priority, vertex)**
  - Push the **vertex** object onto the priority queue with a given **priority** key

  - This priority queue has been specially designed to hold Vertex objects as values ranked by priority keys; be sure you push Vertex objects and NOT vertex id strings onto the queue
- **pop(self)**
  - Visit, remove and return the Vertex object with the highest priority (i.e. lowest priority key) as a **(priority, vertex)** tuple
- **update(self, new_priority: float, new_coupon: int, vertex: Vertex)**
  - Update the priority of the **vertex** object in the queue to have a **new_priority**

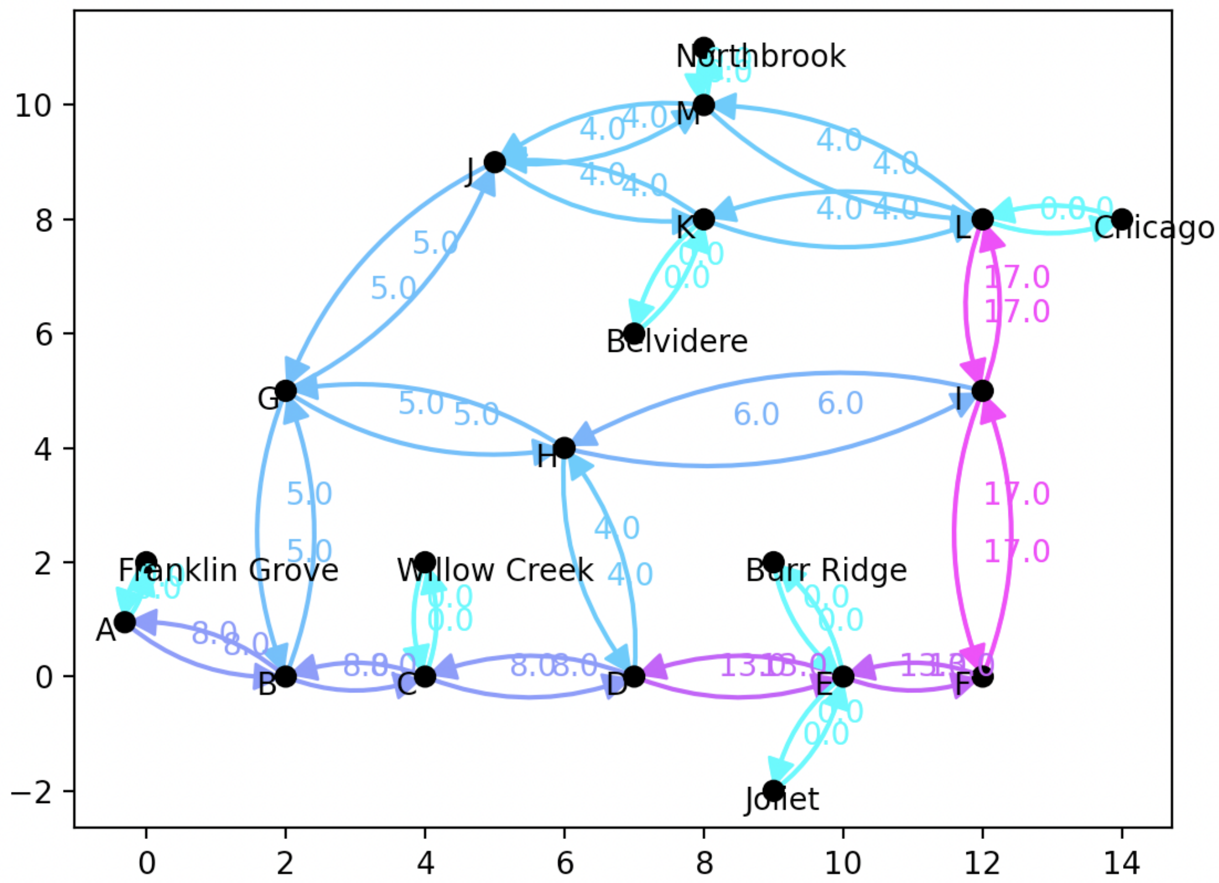
Examples

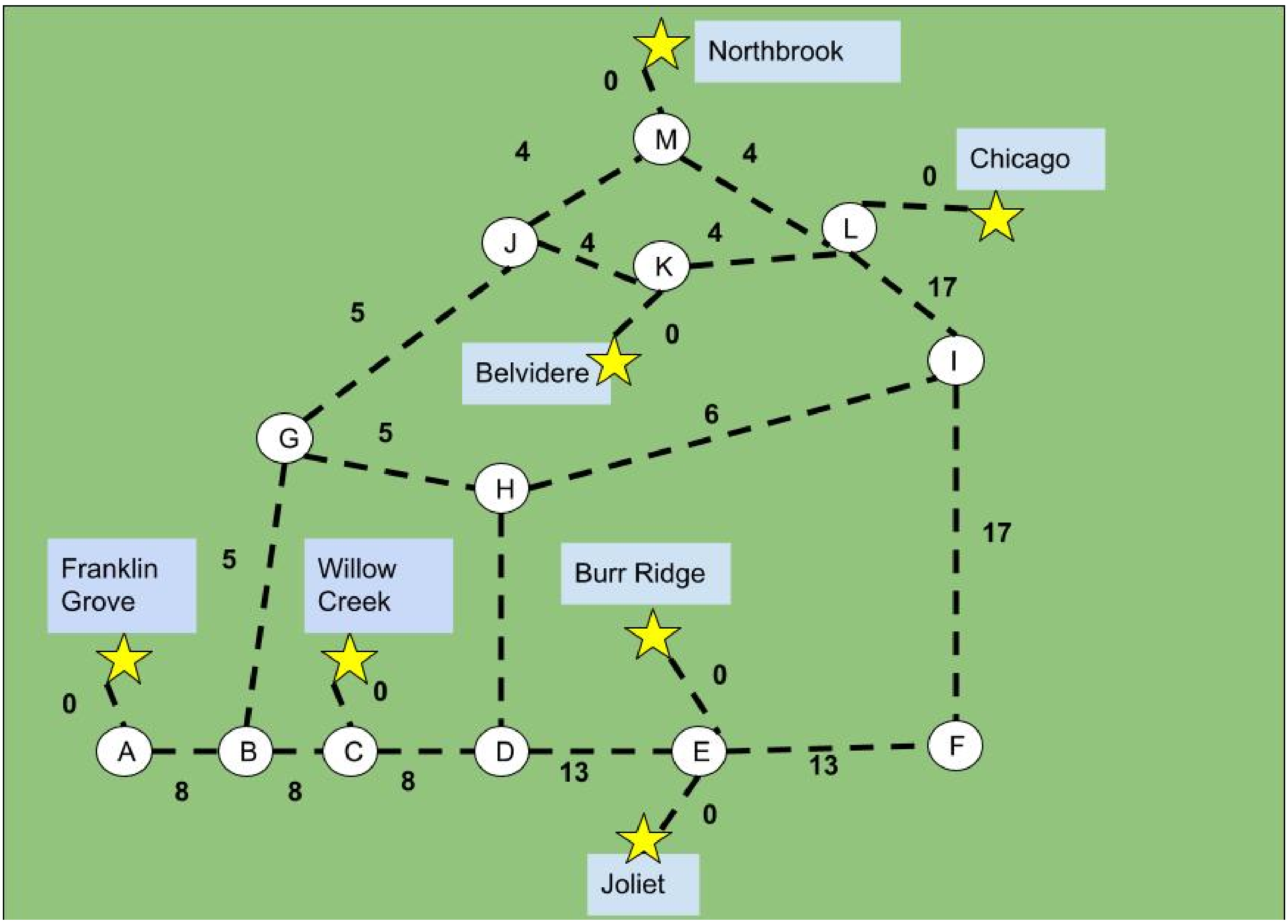
Use the map below of the Chicago Tollways,

Letters A-M represent toll booths, and the stars are cities you are visiting.



and below is a modified and simplified version of the map represented as a graph.





Note: maps/graphs may not be accurate to actual tollway maps.

Examples for a_star

- **a_star('Franklin Grove', 'Northbrook')** would return
  - (['Franklin', 'A', 'B', 'G', 'J', 'M', 'Northbrook'], 22)
  - This finds the same optimal path as Dijkstra
- **a_star('Franklin Grove', 'Joliet')** would return
  - (['Franklin Grove', 'A', 'B', 'G', 'H', 'D', 'E', 'Joliet'], 35)
  - This finds the optimal path (note that BFS would find a sub-optimal path of path length 37)
- **a_star('Northbrook', 'Belvidere')** would return
  - (['Northbrook', 'M', 'J', 'K', 'Belvidere'], 8)
  - Although two equally-optimal paths exist, A* chooses the one that's closer to the straight line connecting the Cafeteria to the Med Bay

Examples for tollways_algorthim

- For Test 3: **tollways_algorithm('A', 'C', 1)** would return
  - (3,1) aka (cost, coupon)
  - We only can spend one coupon. It is applied on the path A->C which costs 6. The coupon halfs the cost, 6/2 = 3.
- For Test 3: **tollways_algorithm('A', 'D', 1)** would return
  - (5,1)
  - The path A->B costs 2 then B->D cost 6. We can only use one coupon so applied to the higher cost toll. Therefore 2 (A->B)+3 (B->D with coupon) = 5.
  - The following is the steps of the data structure updating through Test 3

#1 INITIAL	0 coupons	1 coupons
A	inf	inf
B	inf	inf
C	inf	inf
D	inf	inf
E	inf	inf

We create this matrix format in a dictionary dp['A']= [inf,inf,inf] the dictionary key represents the row and the columns index represent the number of coupons.
We create a priority queue that holds this information as well by pushing in the format [cost, coupons, node] e.g. [inf, 0, A].... [inf,1,E]

#2 START	0 coupons	1 coupons
A	**0**	**0**
B	inf	inf
C	inf	inf
D	inf	inf
E	inf	inf

Initialize the starting vertex in PQ and dictionary to start at 0
E.g. For start at A… dp['A'] = [0]*(coupons+1) = [0, 0]

#3 A TO B	0 coupons	1 coupons
A	0	0
B	**2**	**1**
C	inf	inf
D	inf	inf
E	inf	inf

Perform a dijkstra's search and update PQ & dictionary along the way

#4 A TO C	0 coupons	1 coupons
A	0	0
B	2	1
C	**6**	**3**
D	inf	inf
E	inf	inf

LOOK AT NEIGHBORS AHEAD

Next step involves If the number of coupons is within the limit of coupons AND the **neighbor** vertex cost indexed at coupon+1 is greater than the cost of the path to vertex_to_neighbor_path//2 + vertex_path_cost update neighbor vertex cost in the dictionary and update the PQ.

#5 PATH to D(based on PQ ordering)	0 coupons	1 coupons
A	0	0
B	2	1
C	6	3
D	**inf**	**7**
E	inf	inf

#6 PATH TO D	0 coupons	1 coupons
A	0	0
B	2	1
C	6	3
D	**8**	**UPDATE VALUE! 5 < 7 SO 5**
E	inf	**9**

#7 PATH TO E	0 coupons	1 coupons
A	0	0
B	2	1
C	6	3
D	8	5
E	**10**	**6**

#8 PATH TO E	0 coupons	1 coupons
A	0	0
B	2	1
C	6	3
D	8	5
E	**9**	6

tollway(A,B, 1): look at the minimum cost for row B, use your dictionary.
The minimum cost is $1, and is at column 1 which means one coupon is used.
tollway(A,C,1): look at the minimum cost for row C
The minimum cost is $3, and is at column 1 which means one coupon is used.
tollway(A,D,1): look at the minimum cost for row D
The minimum cost is $4, and is at column 1 which means one coupon is used.
tollway(A,E,1): look at the minimum cost for row E
The minimum cost is $6, and is at column 1 which means one coupon

- For Test 4: **tollways_algorithm('A', 'C', 2)** would return
    - We have a direct path from A->C that costs 4. Only need one coupon.
- For Test 4: **tollways_algorithm('A', 'E', 2)** would return
    - (3,2) aka (cost, coupon)
    - We only can spend two coupons. The path is from A->B (2) and B->E (5). 1(A->B) + (5(B->E)//2) (integer division - no remainder) = 1+2 = 3
- For Test 7: **tollways_algorithm('Franklin Grove', 'Chicago', 5)** City by Coupon Number Data Table for debugging

	0	1	2	3	4	5
M	22	18	15	12	10	10
Burr Ridge	35	28	24	21	18	16
Chicago	26	22	19	16	14	12
C	16	12	8	8	8	8
E	35	28	24	21	18	16
B	8	4	4	8	8	12
I	24	20	17	14	11	11
H	18	14	11	8	8	12
Willow Creek	16	12	8	8	8	8
L	26	22	19	16	14	12
Franklin Grove	0	0	0	0	0	0
J	18	14	11	8	8	12
G	13	9	6	6	10	10
Northbrook	22	18	15	12	10	10
Belvidere	22	18	15	12	10	10
F	41	32	28	25	22	19
A	0	0	8	8	12	12
K	22	18	15	12	10	10
D	22	18	15	12	10	10
Joliet	35	28	24	21	18	16

Writing Application Problem

- **Complete the questions below - copy and paste them into a word doc - and include this pdf document as you are submitting your Project10 on Mimir.**
- Use project10 comparison.py file to answer questions
- Matplotlib installation resources: how-to-install-matplotlib-in-python and install python packages on PyCharm
- **Writing Questions**

- Question 1 (2pts): Run the first test (test1) from comparison.py file and include the figure produced here.
- Question 2 (2pts): The time complexity of both breadth-first and depth-first graph searches is O(|V | + |E|), where V is the set of vertices of the graph, and E is the set of edges. In the figure produced above, the "connectivity factor" of a graph is the probability that given any two vertices, say v1 and v2, there exists an edge from v1 to v2. Note that in a fully-connected graph, where every vertex is connected to every other vertex, we have E=V^2. Why in the figure above does BFS/DFS performance appear linear for small connectivity factors, but not so far larger factors?
- Question 3 (2pts): Run the second test (test2) and include the figure produced here.
- Question 4 (2pts): Notice that AStar search notably outperforms Dijkstras algorithm on the graphs provided. In one or two sentences, informally explain why. What does AStar take advantage of?
- Question 5 (2pts): Suppose you were to call the AStar search as follows: What issues(s) may this cause?

```
graph.a_star(vertex_1, vertex_2, metric = lambda v1, v2 :
random.rand())
```

Submission

Deliverables:

Be sure to upload the following deliverables in a .zip folder to Mimir by 10:00 pm EST, on Thursday, April 21st.

Your .zip folder can contain other files (for example, description.md and tests.py), but must include (at least) the following:

```
|- Project10.zip
    |— Project10/
        |— feedback.xml     (for project feedback, make sure to fill this out)
        |— __init__.py      (for proper testcase loading)
        |— solution.py      (contains your solution source code)
        |— project10.pdf    (contains your written solution)
```

Grading

The following 100-point rubric will be used to determine your grade on Project 10

- Policies
    - **Making all of these policies bold or italic would get too visually fatiguing but read them all because they're important!**

    - Using a different algorithm than the one specified for some function will result in the loss of all automated and manual points for that function.
    - You will not receive any points on this project if you use Python's built-in sorting functions or sorting functions imported from any library.
    - You will not receive any points on the project if you use any list-reversing function such as reversed, list.reverse, or a homemade alternative to these. You must sort the lists in ascending or descending order directly.

- Tests (70)

    - Test feedback.xml:__/3
    - Test coding standard:__/3
    - **dijkstra:__**/14
        - **dijkstra_basic:__**/7
        - **dijkstra_comprehensive:__**/7
    - **a_star**:__/20
        - **a_star_basic:__**/10
        - **a_star_comprehensive:__**/10
    - **tollways_algorithm**:__/20
        - **application_basic:__**/10
        - **application_comprehensive:__**/10

- Manual (30)

    - Time and space complexity points are **all-or-nothing** for each function. If you fail to meet time or space complexity in a given function, you do not receive manual points for that function. (30)
    - Manual points for each function require passing all tests for that function, except for the comprehensive tests.
    - Use of non-local in any function will result with loss of manual points for that function.
    - Time & Space Complexity (30)
        - M1 - **dijkstra** :__/10
            - *Time Complexity: O((E + V)log(V))*
            - *Space Complexity: O(V)*
        - M2 - **a_star**: __/10
            - *Time Complexity: O((E + V)log(V))*
            - *Space Complexity: O(V)*
        - M3 - **tollways_algorithm**:__/10
            - *Time Complexity: O((C*V+E)*log(V*C)), C is the number of coupons*
            - *Space Complexity: O(V*C), C is the number of coupons*
        - M4 - **WRITING ASSIGNMENT**:__/10

Congratulations on Finishing your Last Project!

- [What time is it?!]