Design and Analysis of Algorithms COMP1002

S2 Assignment, 2018 – Documentation

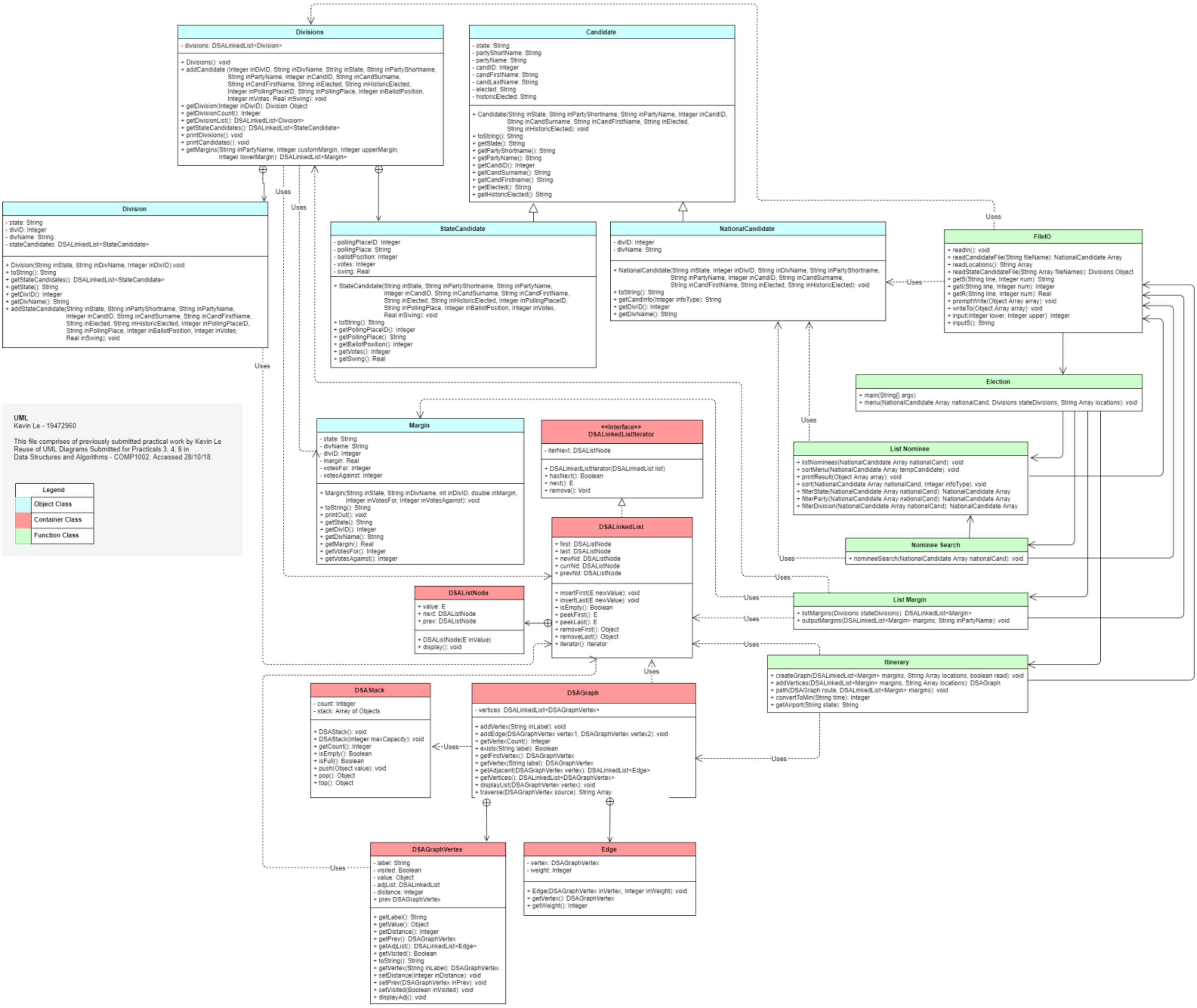
Kevin Le – 19472960 29/10/18

Contents:

1. **UML – Page 2**
2. **Justification of Design Choices (ADTs / Classes/ Major Decisions) – Page 3-8**
   - 2.1. Class Breakdown
     - 2.1.1. Classes
     - 2.1.2. Object Classes
     - 2.1.3. Container Classes
   - 2.2. List Nominees – Array, Insertion Sort
   - 2.3. Search Nominees – Array
   - 2.4. List Margins – Linked List
   - 2.5. Itinerary – Graph, DFS
3. **Class Overview – Page 8**
   - 3.1. Election
   - 3.2. FileIO
   - 3.3. ListNominee
   - 3.4. NomineeSearch
   - 3.5. ListMargin
   - 3.6. Itinerary
4. **Object Class Overview - Page 11**
   - 4.1. Candidate
   - 4.2. NationalCandidate
   - 4.3. StateCandidate
   - 4.4. Divisions
   - 4.5. Margin
5. **Container Classes Overview – Page 12**
   - 5.1. DSALinkedList
   - 5.2. DSAGraph
   - 5.3. DSAStack
6. **Static Method Testing – Page 13**
7. **References – Page 13**

Referenced code is identified with [x], matching the reference number in the References Chapter.

1. UML



Divisions
- divisions: DSALinkedList<Division>
+ Divisions(): void
+ addCandidate (Integer inDivID, String inDivName, String inState, String inPartyShortname, String inPartyName, Integer inCandID, String inCandSurname, String inCandFirstName, String inElected, String inHistoricElected, Integer inPollingPlaceID, String inPollingPlace, Integer inBallotPosition, Integer inVotes, Real inSwing): void
+ getDivision(Integer inDivID): Division Object
+ getDivisionCount(): Integer
+ getDivisionList(): DSALinkedList<Division>
+ getStateCandidates(): DSALinkedList<StateCandidate>
+ printDivisions(): void
+ printCandidates(): void
+ getMargins(String inPartyName, Integer customMargin, Integer upperMargin, Integer lowerMargin): DSALinkedList<Margin>
Candidate
- state: String
- partyShortName: String
- partyName: String
- candID: Integer
- candFirstName: String
- candLastName: String
- elected: String
- historicElected: String
+ Candidate(String inState, String inPartyShortname, String inPartyName, Integer inCandID, String inCandSurname, String inCandFirstName, String inElected, String inHistoricElected): void
+ toString(): String
+ getState(): String
+ getPartyShortname(): String
+ getPartyName(): String
+ getCandID(): Integer
+ getCandSurname(): String
+ getCandFirstname(): String
+ getElected(): String
+ getHistoricElected(): String
Uses
Uses
Uses
Division
- state: String
- divID: Integer
- divName: String
- stateCandidates: DSALinkedList<StateCandidate>
+ Division(String inState, String inDivName, Integer inDivID):void
+ toString(): String
+ getStateCandidates(): DSALinkedList<StateCandidate>
+ getState(): String
+ getDivID(): Integer
+ getDivName(): String
+ addStateCandidate(String inState, String inPartyShortname, String inPartyName, Integer inCandID, String inCandSurname, String inCandFirstName, String inElected, String inHistoricElected, Integer inPollingPlaceID, String inPollingPlace, Integer inBallotPosition, Integer inVotes, Real inSwing): void
Uses
StateCandidate
- pollingPlaceID: Integer
- pollingPlace: String
- ballotPosition: Integer
- votes: Integer
- swing: Real
+ StateCandidate(String inState, String inPartyShortname, String inPartyName, Integer inCandID, String inCandSurname, String inCandFirstName, String inElected, String inHistoricElected, Integer inPollingPlaceID, String inPollingPlace, Integer inBallotPosition, Integer inVotes, Real inSwing): void
+ toString(): String
+ getPollingPlaceID(): Integer
+ getPollingPlace(): String
+ getBallotPosition(): Integer
+ getVotes(): Integer
+ getSwing(): Real
NationalCandidate
- divID: Integer
- divName: String
+ NationalCandidate(String inState, Integer inDivID, String inDivName, String inPartyShortname, String inPartyName, Integer inCandID, String inCandSurname, String inCandFirstName, String inElected, String inHistoricElected): void
+ toString(): String
+ getCandInfo(Integer infoType): String
+ getDivID(): Integer
+ getDivName(): String
Uses
Uses
FileIO
+ readIn(): void
+ readCandidateFile(String fileName): NationalCandidate Array
+ readLocations(): String Array
+ readStateCandidateFile(String Array fileNames): Divisions Object
+ getS(String line, Integer num): String
+ getI(String line, Integer num): Integer
+ getR(String line, Integer num): Real
+ promptWrite(Object Array array): void
+ writeTo(Object Array array): void
+ input(Integer lower, Integer upper): Integer
+ inputS(): String
Election
+ main(String[] args)
+ menu(NationalCandidate Array nationalCand, Divisions stateDivisions, String Array locations): void
UML
Kevin Le - 19472960
This file comprises of previously submitted practical work by Kevin Le
Reuse of UML Diagrams Submitted for Practicals 3, 4, 6 in
Data Structures and Algorithms - COMP1002. Accessed 28/10/18.
Legend
🟦 Object Class
🟥 Container Class
🟩 Function Class
Margin
- state: String
- divName: String
- divID: Integer
- margin: Real
- votesFor: Integer
- votesAgainst: Integer
+ Margin(String inState, String inDivName, int inDivID, double inMargin, Integer inVotesFor, Integer inVotesAgainst): void
+ toString(): String
+ printOut(): void
+ getState(): String
+ getDivID(): Integer
+ getDivName(): String
+ getMargin(): Real
+ getVotesFor(): Integer
+ getVotesAgainst(): Integer
<<Interface>>
DSALinkedListIterator
- iterNext: DSAListNode
+ DSALinkedListIterator(DSALinkedList list)
+ hasNext(): Boolean
+ next(): E
+ remove(): Void
Uses
List Nominee
+ listNominees(NationalCandidate Array nationalCand): void
+ sortMenu(NationalCandidate Array tempCandidate): void
+ printResult(Object Array array): void
+ sort(NationalCandidate Array nationalCand, Integer infoType): void
+ filterState(NationalCandidate Array nationalCand): NationalCandidate Array
+ filterParty(NationalCandidate Array nationalCand): NationalCandidate Array
+ filterDivision(NationalCandidate Array nationalCand): NationalCandidate Array
DSALinkedList
+ first: DSAListNode
+ last: DSAListNode
+ newNd: DSAListNode
+ currNd: DSAListNode
+ prevNd: DSAListNode
+ insertFirst(E newValue): void
+ insertLast(E newValue): void
+ isEmpty(): Boolean
+ peekFirst(): E
+ peekLast(): E
+ removeFirst(): Object
+ removeLast(): Object
+ iterator(): Iterator
Nominee Search
+ nomineeSearch(NationalCandidate Array nationalCand): void
Uses
Uses
DSAListNode
+ value: E
+ next: DSAListNode
+ prev: DSAListNode
+ DSAListNode(E inValue)
+ display(): void
Uses
Uses
List Margin
+ listMargins(Divisions stateDivisions): DSALinkedList<Margin>
+ outputMargins(DSALinkedList<Margin> margins, String inPartyName): void
Uses
Itinerary
+ createGraph(DSALinkedList<Margin> margins, String Array locations, boolean read): void
+ addVertices(DSALinkedList<Margin> margins, String Array locations): DSAGraph
+ path(DSAGraph route, DSALinkedList<Margin> margins): void
+ convertToMin(String time): Integer
+ getAirport(String state): String
Uses
DSAStack
- count: Integer
- stack: Array of Objects
+ DSAStack(): void
+ DSAStack(Integer maxCapacity): void
+ getCount(): Integer
+ isEmpty(): Boolean
+ isFull(): Boolean
+ push(Object value): void
+ pop(): Object
+ top(): Object
Uses
DSAGraph
- vertices: DSALinkedList<DSAGraphVertex>
+ addVertex(String inLabel): void
+ addEdge(DSAGraphVertex vertex1, DSAGraphVertex vertex2): void
+ getVertexCount(): Integer
+ exists(String label): Boolean
+ getFirstVertex(): DSAGraphVertex
+ getVertex(String label): DSAGraphVertex
+ getAdjacent(DSAGraphVertex vertex): DSALinkedList<Edge>
+ getVertices(): DSALinkedList<DSAGraphVertex>
+ displayList(DSAGraphVertex vertex): void
+ traverse(DSAGraphVertex source): String Array
Uses
DSAGraphVertex
- label: String
- visited: Boolean
- value: Object
- adjList: DSALinkedList
- distance: Integer
+ prev DSAGraphVertex
+ getLabel(): String
+ getValue(): Object
+ getDistance(): Integer
+ getPrev(): DSAGraphVertex
+ getAdjList(): DSALinkedList<Edge>
+ getVisited(): Boolean
+ toString(): String
+ getVertex(String inLabel): DSAGraphVertex
+ setDistance(Integer inDistance): void
+ setPrev(DSAGraphVertex inPrev): void
+ setVisited(Boolean inVisited): void
+ displayAdj(): void
Edge
- vertex: DSAGraphVertex
- weight: Integer
+ Edge(DSAGraphVertex inVertex, Integer inWeight): void
+ getVertex(): DSAGraphVertex
+ getWeight(): Integer

Higher resolution image located in main assignment directory UML.png/

UML made with Draw.IO

Justification of Design Choices

2.1 Class Breakdown

2.1.1 Classes

The List Nominee, Nominee Search, List Margin and Itinerary classes were created to separate each feature required into its own class to keep things organized and menu calls easy to understand.

The files were first read into the correct data structures by a call to readIn in FileIO, once the files were read in, the menu located in a class called Election was loaded. Which handled all the calling to the appropriate functions.

- **Election**
    - Purpose is to provide a main to start the program and the menu interface.
    - Created to provide a main call and a location for the menu
- **FileIO**
    - This class handled all the reading and storing of the data into the required data structures or ADTs and loaded the menu. Purpose is to create one centralized class that dealt with handling files and user input, including writing to a file which was called on by all 4 main function classes.
    - Contained functions which could be called on to input integers and strings, validating the input and returning the input.
    - Created to facilitate the FileIO features needed and keep it neatly in one class.
    - The generic user input functions could have been separated further into another class, but it was not worth creating a new class for 2 input functions.
- **List Nominee**
    - This class contained all the functions to form the features of filtering the national nominees, the state nominees and then sorting them. Displays the menu prompts to perform the operation required.
    - The purpose of this class is to form the List Nominee feature specified and provide reusable methods that can be called on to print, filter and/or sort an array of Objects.
- **Nominee Search**
    - Provided a user interface for the nominee search functionality
    - This class borrows calls a lot of functions from the List Nominee
    - Nominee search could have been combined with the List nominee class but separating each key functionality into separate classes made sense.
- **List Margin**
    - Provides the user interface for the listing of margins, its purpose is to call the getMargins function in the Divisions, returning the linked list of Marginal seats to be written and passed

  Itinerary
    - Purpose of this class is to load the marginal divisions, edges and required linking airports into the graph from the Linked List of Margin objects

2.1.2 Object Classes

These object classes were created for storing information on candidates in an easy to access and manipulate manner:

- **Divisions**
  - Purpose is to aggregate the divisions and store them. Stores a linked list of Division Obj.
  - Logically made sense to create this object. "Divisions contain candidates" and this is reflected in the class breakdown, as Divisions contains a linked list of Division objects which each contain another linked list of State Candidate objects.
  - Created to facilitate List Margins and Itinerary, which needed a method that separated candidates into divisions.
  - StateCandidate and Division were both made private classes inside since the only class using those objects was Divisions. So it was fine to hide it from other classes whilst allowing Divisions to access them.
- **Division - Private class inside Divisions**
  - Purpose is to hold information about the division and the candidates for that division
  - Created for the Divisions object linked list to store.
- **Candidate**
  - "A national candidate" is a Candidate, and "a state candidate" is a Candidate. Made sense to use inheritance and create a Candidate super class.
  - Purpose is to give each candidate their own objects allowing easy access and storing of the fields that contained the data.
  - Created to reduce the size of the National Candidate and State Candidate objects
- **National Candidate**
  - Inherits from Candidate class,
  - The purpose of this class is to add additional fields, divName and divID.
- **State Candidate - Private class inside Divisions**
  - Inherits from Candidate class
  - Purpose is to provide additional fields, pollingPlaceID, pollingPlace, ballotPosition, votes, and swing.
- **Margin**
  - Purpose is to store information required to create the vertices for the graphs
  - Created to make creation of vertices easier for Itinerary.

2.1.3 Container Classes

These classes store data, such as objects.

- **DSALinkedList** [1]
  - The Linked List was created to have a data structure that facilitates easy insertion and deletion of data (O(1) insertion). But especially for its dynamic sizing, for functions that would have to be storing large amounts of data such as the Division objects for List Margin, or functions that would be difficult determining the size of storage to allocate such as the Margin objects, a linked list was the best solution. Not needing to allocate size beforehand means there is less of a cost of having to count the number of items to allocate size for.
- **DSAGraph** [2]

    - A graph was the best solution to represent points and locations with distances in between, adding vertices is an easy task, and adding the edges in. DSAGraph is used by Itinerary to calculate the path to take to visit all vertices which are marginal seats.
- **DSAStack** [3]
    - Implemented due to Depth Firsth Search traversal method in DSAGraph requiring a stack for push pop operations with vertices.

—

2.2 List Nominees

- **Uses an array of National Candidate Objects**

The amount of data being stored in the array is relatively small (996 lines). The algorithm in FileIO for creating and adding the objects to the array performs 2 passes, first pass to count the number of lines and a second pass to initialize array and create the objects. An array was chosen as the data structure to hold the objects as the data would never be increasing in size, there was also no need for the insertion of values into the array at mid points or in front, so some of the disadvantages of arrays wouldn't be encountered. Access time using an array is also excellent as it is O(1) which is useful for filtering and sorting, and accessing is required for Listing Nominees and allows for easy implementation with the sort function. Filtering was easy as having the ability to create temporary arrays allowed separating the filter result from the actual main array. A viable alternative was the use of a Linked List but there was no need for the ability to dynamically grow and shrink. However, the arrays sometimes use less space than was allocated, especially with the temporary arrays, one of the disadvantages, forcing the need to have an if condition when iterating through, checking if its not null.

- **Uses Insertion Sort** [4]

This sort was chosen despite being an O(n^2) sort due it still being relatively faster than other O(n^2) algorithms, it was also chosen as its both an in-place *and* stable sort with better efficiency than bubble/ selection sort. Insertion sort uses much less temporary storage and guarantees that the result would keep its ordering. [7]

Its implementation is simple and less complex than other stable or in-place sorts such as alternatives, MergeSort (complex, requires more temporary storage) and QuickSort (Can still have O(n^2) time complexity and has a risk of stack overflow due to recursion, especially with large data). [7]

- **Functions: listNominees, sortMenu, sort, filterState, filterParty, and filterDivision**

This layout was chosen in the order the tasks would be performed. It was separated into the filtering stage, sort stage and print/write stage. listNominees provided a menu to select the filter operation, which then called the appropriate filter. Then sortMenu was called, prompting for the sort to perform. When entering Party names for filtering, it only accepts the abbreviated party name form, as the CSV file data appeared to have some inconsistencies with full party naming (e.g. different naming), so it was assumed that the abbreviated form would be the best and most consistent. Entering the abbreviated format is also assumed easier for the user, and less likely to make spelling mistakes for a short 3 letter word.

The filter methods were split as that allowed other classes to make use of them. Such as the filter State and party options in NomineeSearch.

These methods could have been further separated into more classes, since both List Nominees and Nominee Search need the same methods, it was more elegant to have the functions in one of the 2 classes instead of the 2 classes calling another one.

2.3 Nominee Search

- **Uses an array of National Candidate Objects**

Nominee Search used the same array as ListNominees for the same reasons. This allowed for the reuse/ compatibility with the filter methods in the ListNominee class. Simple operations on a relatively small data set, with no resizing of data needed, and just filtering through the array twice, one for a search, one for the filter meant arrays were well suited for a small feature.

- **Contained in Two methods, nomineeSearch and promptFilter**

When first called, the search method is run, and once the search results have been loaded, the function promptFilter is called. The class is separated into functions representing the order of operations it would run.

—

2.4 List Margin

- **Divisions Object with private inner classes Division and State Candidate**

List Margin required a more efficient data structure to store all the division information, a special Divisions object is made, with a Division private inner class and a StateCandidate private inner class. The divisions object has a method that aggregates the divisions as candidates are added and has the appropriate getters and setters. The Division is then stored in a linked list of Divisions. Each division object contains a linked list of candidates, which logically makes sense as each division has candidates. This is done so multiple divisions are aggregated into one division which allows for easy calculations to perform at division levels and less work needed for Itinerary.

- **Divisions Object uses Linked List to store aggregated divisions**

The Divisions object is declared in the function readStateCandidateFile in the FileIO class. FileIO extracts the necessary data then passing it to addCandidate. Aggregating 8 csv files (representing the candidates per state and territory) which contained >70,000 lines in total meant a Linked List was suited. If an array was used counting through to get number of lines for allocating the array would be expensive. So a linked list was optimal, as its dynamically allocated (no need to determine size) and an insertion complexity on the end of O(1). However despite the low insert complexity, aggregating the actual candidates into divisions was an $O(N^2)$ task, having to check the linked list of Division objects every time to determine if the division already exists before adding it, for each row in the CSVs. This perhaps could have been countered with Hash Tables/ Hash Maps but the large files would still have to be read in.

- **Functions: listMargins, outputMargins**

The ListMargin class has functions listMargins and outputMargins. These functions provide menus to prompt for which party, custom margins for the margin search, and to write the results, the actual calculations to run were calling the margin calculation functions located in the Divisions class. Functions located in the order they would be run.

- **Linked List of Margin Objects**

The margin calculation was performed in the Divisions object class by getMargin which iterates through the linked list of Divisions with a for each loop. For each division, the candidates of that division are iterated through, calculating the number of votes for the selected party and the rest against. Then the margin is calculated. If a custom margin was specified those values are used, else the default is used, and if a division for that party is within the party's bounds, a Margin object is created and added to a new Margins linked list. Using a linked list since it has dynamic sizing with no knowledge beforehand of the amount of the margins for the specified party and low insertion complexity on the end of O(1). The linked list of Margin objects is returned for use by the Itinerary class to create routes. This structure to calculate the margins was chosen to allow the ListMargin class to just be used as an interface for the getMargins function in the Divisions. getMargins was created in the Divisions object so it easily has access to all the candidate fields and division fields, without the high coupling calls and passing in the objects to and from a method, if it were to be in ListMargin for example.

—

2.5 Itinerary

- **Graph Container Class** [2]

The Itinerary class is implemented with a Graph container class, with a Node and Edge private inner class. Creating the graph requires a Margin object Linked List and a Locations String array containing all the CSV location data for airports and electorates. An extra edge class was required since the Edge must hold the weight of the link (in this case the distance). A graph logically makes sense to represent locations with distances, so it is the best option for this component, especially when it comes to traversal as there are well made algorithms to traverse the graphs in the shortest path.

This class also justifies the use of the Margins linked list, it ensures that no unnecessary vertices are added and only the marginal seats are added, isolating the class from the extremely large Divisions linked list. The margins linked list is useful as the Margin object is constructed in the Divisions class to contain all the information required for the creation of the vertices in the graph. The difficult operation of aggregating the divisions has been completed by List Margins in the Divisions class, so adding all the nodes is an O(1) operation, since we aren't processing any data and since we aren't removing nodes or edges, we don't encounter the O(N) operations of a graph. Edges however take more work as we must iterate through the locations string array to add the matching links, adding edges is at least an O(N) operation.

- **Location data stored all in one array**

The location data for both airports and electorate locations was stored in one array. This allowed for smaller function calls and easier passing of data to the functions requiring the location information. An array was used instead of a linked list to store location info as it is possible to iterate through the array from the middle/ end, where the Airport data was located (e.g. iteration can start at 2000). So instead of using 2 linked lists or having to traverse through one large Linked List with the airport data at the end (therefore makes access O(N)). A simple array sufficed.

- **Depth First Search (Previously Djikstras)** [5][6]

To traverse the graph, I had to revert to the Depth First Search algorithm. Which is a significantly less short method of traversal through the graph, forfeiting the option of having the shortest path. It doesn't consider edge weights making it an extremely bad algorithm. Unfortunately, it still seemed to have trouble traversing the graph still. The complexity of DFS is O(Vertices + Edges).

There was an attempt to implement Dijkstra's algorithm as it's a much more optimal algorithm vs Depth First search as it is better at finding shorter paths and doesn't just plainly traverse the graph, but implementation failed due to some vertices never being traversed, it was likely too complex for me, or simply didn't suit how I set up my graph. The attempt can be found in the Testing folder. Testing failed to uncover the problem as it successfully traversed edges with small test cases. Plausible that the edges are disconnected, or the algorithm has a major logic error. Else it would be so much better than just a simple DFS traverse. Djikstras complexity is O(V^2) but yields a much more useable result.

- **Functions: createGraph, addVertices, path, convertToMin, getAirport.**

An assumption is made for the function getAirport, as there was a large increase in complexity having to read through the locations array to retrieve the airport name from the files base on state, so it was assumed that state airports would not be changed as often, and the airport names were hardcoded per state in a simple return function that returned the airport based on the state.

convertToMin was used to convert the airport travel times to the same time unit as the Electorate times, an assumption was made the times in the airport csv would follow an exact format, HH:MM:SS

—

3. Class Overview

Extra function information – Self-explanatory classes excluded

3.1 Election

3.1.1 menu

After files have been read in, menu receives data structures holding the information required for each method. (List Nominees, Nominee Search, List by Margin, Itinerary). Prompts user for desired action calling appropriate class, passing in the required data.

3.2 FileIO

3.2.1 readIn

This function calls the functions that read in the files for the data structure and sets the file names to variables. Using a string array for the State Candidate files to allow the method readStateCandidateFIle to loop through per file. The array of National Candidate objects, Divisions object and String array of locations are all read in and sent to the menu. Assumes files would be contained in same directory and filenames not changed.

3.2.2 readCandidateFile

Reads through the file twice, first time it counts the number of lines in the file, the second time it initializes the National Candidate object array, which has the objects created and added to it. Once finished, the array full of candidate objects is returned.

3.2.3 readLocations

Reads the Airport and Electorate Location files first, to get the array size to initialize, and then reads both files again this time loading the lines into a String array. This function combines both the Airport and Electorate locations into one array for easy passing into other functions.

3.2.4 readStateCandidateFile

Creates a new Divisions object, and uses the array of location names to read each State/ Territory file into the one object, using the addCandidate method.

3.2.5 getS

A string split method that is called, passing in a String line and number index on what to split by. Returns the string at that index using split. Uses a special Regex to force ignore of commas in quotes.

3.2.6 getI

A string split method that is called, passing in a String line and number index on what to split by. Returns the integer at that index using split. Uses a special Regex to force ignore of commas in quotes.

3.2.7 getR

A string split method that is called, passing in a String line and number index on what to split by. Returns the real value at that index using split. Uses a special Regex to force ignore of commas in quotes.

3.2.8 promptWrite

Menu to prompt the user if they would like to write the printed data to a file (Stored in an Object array). Calls writeTo if requested.

3.2.9 writeTo

Writes the specified object array to a file. Can write any object array with a toString method, generic and is used by all four menu features to write to a file.

3.2.10 input

Simple integer input method for input validation. Called with a minimum value that can be entered and a maximum value that can be entered, returning the input if valid, prompting for input if not. Called on by all menu functions, allows reuse of code instead of multiple duplicate lines.

3.2.11 inputS

String input validation, returns string on valid string input.

3.3 List Nominee

3.3.1 listNominees

Menu to perform filtering calls appropriate filter methods then sends to sort() to prompt user for sort to perform on filtered result.

3.3.2 sortMenu

Prompts user for sort type. then performs the specified sort by calling sort function, passing in a number parameter for the sort function to use to call National Candidate, which passes the appropriate data depending on number. (Surname/State/Party/Division).

3.3.3 sort [4]

Insertion sort method. Takes a variable infotype based on the sort option. Used to call getCandInfo in the National Candidate object, which returns the variable to sort ‘on’ based on the number. 1 = Surname, 2 = State, 3 = Party, 4 = Division.

3.3.4 filterState

Iterates through array checking for matches with specified state name. If found, it’s printed and stored into a temporary array for writing (if the user requests so)

3.3.5 filterParty

Iterates through array checking for matches with specified party name. If found, it’s printed and stored into a temporary array for writing (if the user requests so)

3.3.6 filterDivision

Iterates through array checking for matches with specified division name. If found, it’s printed and stored into a temporary array for writing (if the user requests so)

—

3.4 Nominee Search

3.4.1 Nominee Search

Search for nominee based on last name, creates 2 temporary arrays. 1 to store search result and 1 to store the filtered result to write to a file. Iterates through National Candidate object array finding if the last name matches with the search term using startsWith. Search term converted to upper case also to match CSV file. Then filtering function is called performing selected filter/s on search result if candidates have been found, if no candidates found, outputs not found message and returned to menu.

3.5 List Margin

3.5.1 listMargins

Menu options for the getMargins function in Divisions object. Manages input for party name and custom margins if specified. Calls the getMargins returning the Margin linked list for passing to getMargins.

3.5.2 outputMargins

Print out the marginal seats if there are any, if no seats, print out the party name that couldn’t be found. Else prompt the user if they want to write the result. The margin linked list is then written to a string array to allow compatibility with the writing method in FileIO

3.6 Itinerary

3.6.1 createGraph [2]

Checks if the graph has been created yet, if not then it displays error and returns to menu. Else it calls the add vertices function passing in the margins and locations. Returning the graph.

3.6.2 addVertices [2]

Iterates through Margins linked list adding all margins in the Margin linked list as vertices in the graph. If a new state is also found, the airports for that state and arrival state are added.

3.6.3 path [5]

Calls on traversal method in DSAGraph to create a route to take to visit the vertices. Traversal method returns string array with weights and visited vertices, for loop iterates through array extracting certain indexes of the array to print out the from, to and time.

3.6.4 convertToMin

Converts a string in the format HH:MM:SS to seconds.

3.6.5 getAirport

Returns the airport based for the state name that has been passed in. Airports hardcoded (Assumption)

—

Object Class Overview

4.1 Candidate (Super Class)

Contains all required constructors, accessors and mutators to form a Candidate Object. See comments.

4.2 National Candidate

4.2.1 getCandInfo

Returns the required information based on the sort menu option specified in ListNominee. For implementation with insertion sort based on users option on what to sort by. Call this method with appropriate param to get desired return (only returns string). Used by sort to reduce method calls needed.

4.3 State Candidate – Private Class in Divisions

Contains all required constructors, accessors and mutators. See comments.

4.4 Divisions

Constructor creates a new linked list of Division Objects

Division objects contain another linked list of candidates

4.4.1 addCandidate

Checks if the candidate being added already has a division or not, if the candidates division already exists in the divisions linked list, than they’re added to the Division candidates linked list. If the candidates division does not exist yet, then a new division is created, the candidate is added to the new division and the new division is added to the linked list.

4.4.2 getDivision

Finds the matching division based on division ID search, iterates through linked list of division objects.

4.4.3 getDivisionCount

Counts the number of divisions in the linked list of Divisions

4.4.4 printCandidates

Prints the candidates in each division for each division in the divisions linked list.

4.4.5 getMargins

For each Division in the "divisions" linked list, iterate through the Division objects candidates linked list summing up the votes for the parties matching the search term. If the search term doesnt match, count the votes against. Calculate margin for that division, saving to Margins linked list if it is considered significant. Once entire divisions linked list is searched through. Return Margin linked list.

4.5 Margin

Contains all required constructors, accessors and mutators to form a Margin object. See comments.

—

5. Container Classes Overview
Most documentation is contained in class comments – Self-Explanatory methods excluded

5.1 DSALinkedList [1]

Contains all required constructors, accessors and mutators for the linked list container class. See comments.
Contains a DSALinkedList node private class for DSALinkedList to construct when adding node.
Contains an Iterator private class to be implemented by the Linked List

5.2 DSAGraph [2]

Contains a DSAGraphVertex private inner class and an Edge private inner class

5.2.1 addVertex
Creates a new Vertex object using the Label passed in and stores an object alongside in the vertex.

5.2.2 addEdge
Creates a `link` by adding the connections to the vertex adjacency lists along with the weight. See addEdge method in DSAGraphVertex class.

5.2.3 getVertexCount
Counts the number of vertices contained in the vertices linked list.

5.2.4 exists
Checks if a vertex label specified exists in the graph. Iterates through vertices linked list, if there is a match with the label, then the vertex exists. Else false.

5.2.5 getVertex
Gets the vertex matching the label. Find if a vertex has the same label as the label to search by, if true, vertex is found, return that vertex.

5.2.6 traverse
Algorithm for Depth First search graph traversal, utilises a stack. Adds vertices onto a stack until all have been visited and stack is empty. Once stack is empty all the vertices have been visited. Used by Itinerary.

- 5.2.7 DSAGraphVertex (Private Class in DSAGraph) [2]

Required in the graph as graph requires a vertex (node) object to represent the points.

> **5.2.7.1 getVertex**
> Gets the vertex matching the label. Find if a vertex has the same label as the label to search by, if true, vertex is found, return that vertex.

5.2.7.2 displayAdj

Displays the Linked List of edges for the current vertex, in this case the Edge objects contained in the adjList

5.2.7.3 checkExists

Checks if a DSAGraphVertex object exists in the current vertex adjacency list

5.7.2.4 addEdge

Checks if the vertex connection already exists in the adjacency list, if not then the vertex is added as new edge/link.

- 5.2.8 Edge (Private Class in DSAGraph)

Private class representing the edge, required as Edge information must also hold the weight of the edge and not just the label connecting to.

5.3 DSAStack [3]

Contains all required constructors, accessors and mutators for the stack container class. See comments.

—

7. Static Method Testing

Functionality testing located in /Testing/ModuleTests.pdf

—

6. References

[1] DSALinkedList.java

Re-Use of Previously Submitted Practical 4 - Linked Lists, in unit: *Data Structures and Algorithms - COMP1002.* Implementation of Linked List and Iterator based on Pseudocode from Lecture 4 - Linked Lists, Iterators and Generics. Curtin University - Department of Computing. Accessed 9/10/2018.

[2] DSAGraph.java

Re-Use of Previously Submitted Practical 6 - Graphs, in unit: *Data Structures and Algorithms* - COMP1002. Implementation of Graph based on Pseudocode from Lecture 6 - Graphs. Curtin University - Department of Computing. Accessed 9/10/2018.

[3] DSAStack.java

Re-Use of Previously Submitted Practical 3 - Stacks, Queues and Recursion, in unit: *Data Structures and Algorithms* - COMP1002. Implementation of Stack based on Pseudocode from Lecture 3 - Stacks, Queues and Recursion. Curtin University - Department of Computing. Accessed 18/10/2018.

[4] ListNominee.java – sort()

Insertion sort translated from Practical 2 of *Data Structures and Algorithms* COMP1002 - Insertion Sort. Implementation of Insertion sort based on Pseudocode from Lecture 2 - Sorting. Curtin University - Department of Computing. Accessed 26/9/2018.

Parts adapted from *java2s* Sorting Objects using insertion sort : Sort « Collections « Java Tutorial http://www.java2s.com/Tutorial/Java/0140__Collections/SortingObjectsusinginsertionsort.htm

[5] DSAGraph.java – traverse()

Obtained from *GeeksforGeeks* at "Iterative Depth First Traversal of Graph". Retrieved from: https://www.geeksforgeeks.org/iterative-depth-first-traversal/ Accessed 26/10/2018

[6] DjikstrasAlgorithm.java

Obtained from *Wikipedia* at "Dijkstra's algorithm". Retrieved from: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#Pseudocode Accessed 20/10/2018.

[7] Sorting

Justification of Insertion sort based on data from Lecture 2 - Sorting. Curtin University - Department of Computing. Accessed 29/10/2018.