**UML Diagrams:**

**struct idxNode**

**{**

**lNode \*list;**

**lNode head;**

**};**

**struct lNode**

**{**

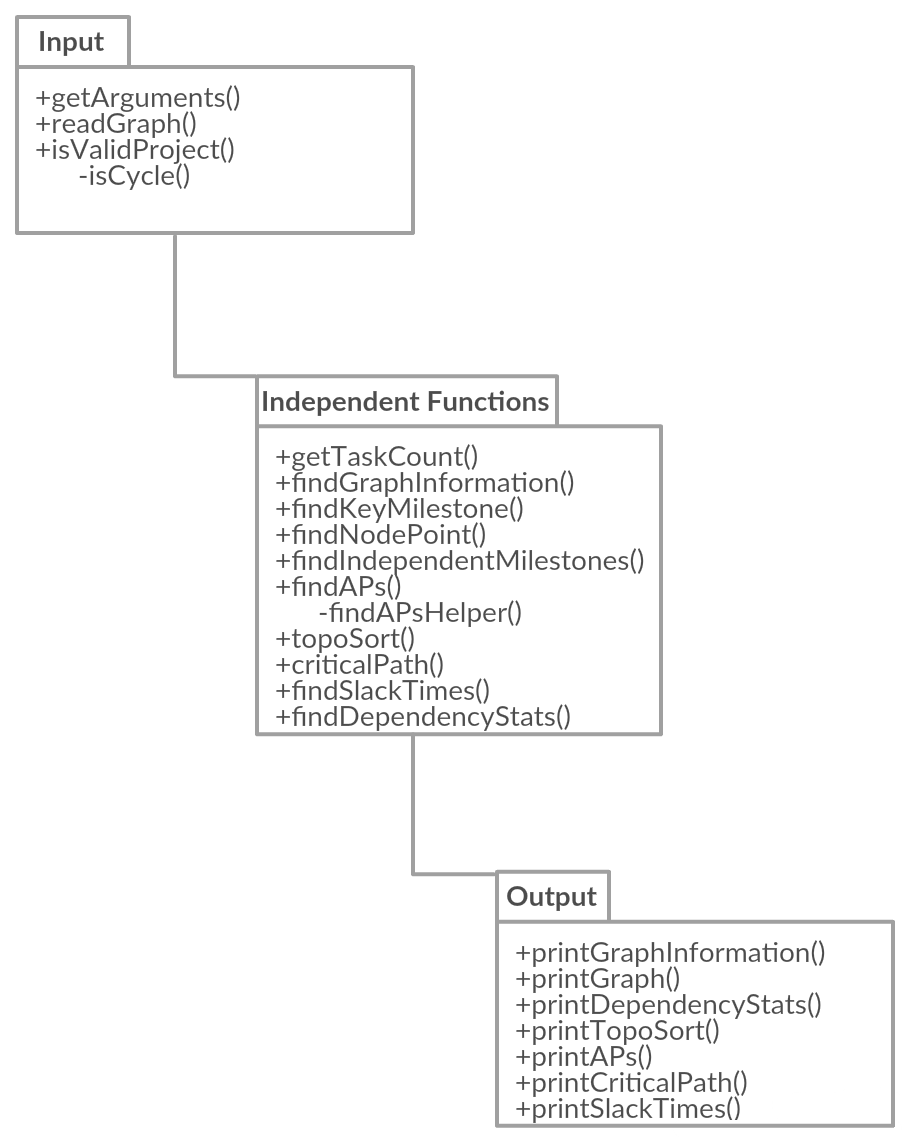
**int weight;**

**int vertex;**

**lNode \* next;**

**};**

|  |
| --- |
| **ganttUtils** |
| -adjList: \*idxNode   |  | | --- | | * *adjList is a variable that represents the adjacency list. It has a struct data type, that contains a list, and a head pointer. The head pointer points to the start of the list, and list is a linked list data type with the struct containing weight, vertex and next.* * *size should be equal to vertex count.* | |
| -visited: \*bool   |  | | --- | | * *visited[] is a variable that is used to keep track of visited elements in isValidProject(), it is also used findAPs(). (true if visited, false otherwise)* * *size should be equal to vertex count.* | |
| -aps: \*bool   |  | | --- | | * *aps[] is a variable used to store articulation points in findAPs() & findAPsHelper(). (true if index is articulation point, false otherwise)* * *size should be equal to vertex count.* | |
| -topoNodes: \*int   |  | | --- | | * *topoNodes[] is a variable that holds the topological sort of the graph.* * *size should be equal to vertex count.* | |
| -distances: \*int   |  | | --- | | * *distances[] is a variable that is used in criticalPath() to generate the distance of each neighbor connected to a vertex.* * *size should be equal to vertex count.* | |
| -crPath: \*int   |  | | --- | | * *crPath[] is a variable that is used in criticalPath() to store the critical path of the graph.* * *size should be equal to vertex count.* | |
| -slackTimes: \*int   |  | | --- | | * *slackTimes[] is a variable that is used in findSlackTimes() to store slackTime at each edge / vertex.* * *siuze should be equal to edge count at most.* | |
| -vCount: int   |  | | --- | | * *vCount is a variable that represents the vertex count of the graph.* | |
| -tCount: int   |  | | --- | | * *tCount is a variable that represents the task count of the graph.* | |
| -srcMile: int   |  | | --- | | * *srcMile is a variable that represents the source milestone of the graph.* | |
| -keyMile: int   |  | | --- | | * *keyMile is a variable that represents the key milestone of the graph (max in-degree).* | |
| -nodePoint: int   |  | | --- | | * *nodePoint is a variable that represents the node point of the graph (max out-degree).* | |
| -tmRatio: double   |  | | --- | | * *tmRatio is a variable that represents the tasks to milestone ratio of the graph.* * *task / milestone* | |
| -tDensity: double   |  | | --- | | * *tDensity is a variable that represents the task density of the graph.* * *(2\*tasks) / (milestones \* milestones - 1)* | |
| -title: string   |  | | --- | | * *title is a variable that represents the header / title of the graph.* | |
| -isCycle(int, int): bool   |  | | --- | | * *isCycle() is a function used in isValidProject() to determine if cycles exists in the graph.* | |
| -findAPsHelper(int, int, int, int): void   |  | | --- | | * *findAPsHelper() is a function used in findAPs() to determine articulation points for unvisited vertices in the graph.* | |
| +ganttUtils()   |  | | --- | | * *ganttUtils() constructor should initialize class variables to empty state.* | |
| +~ganttUtils()   |  | | --- | | * *~ganttUtils() destructor should deallocate all dynamically allocated memory.* | |
| +getArguments(int, char\*[], string&, bool&): bool   |  | | --- | | * *getArguments() parses arguments. If no arguments, show usage of “./projectInfo -f <filename>. Ensure file can be opened and read. 1st argument is argument count, 2nd argument is argument vector, 3rd argument is filename, 4th argument is print flag. (true if all arguments are correct, false otherwise)* | |
| +readGraph(string): bool   |  | | --- | | * *readGraph() opens and reads file from passed argument, stores read data into class variables. (true if file can be opened and read, false otherwise)* | |
| +isValidProject(): bool   |  | | --- | | * *isValidProject() checks the graph data to ensure that it does not contain a path and is directed. (true if graph is directed and acyclic, false otherwise)* * *calls isCycle() to look for cycles in the graph.* | |
| +getTaskCount(): int   |  | | --- | | * *getTaskCount() returns the total amount of tasks in the graph.* | |
| +findGraphInformation(): void   |  | | --- | | * *findGraphInformation() calculates the tasks / milestones ratio, and task density ratio.* | |
| +findKeyMilestone(): void   |  | | --- | | * *findKeyMilestone() finds key milestone. (max in-degree)* | |
| +findNodePoint(): void   |  | | --- | | * *findNodePoint() finds node point. (max out-degree)* | |
| +findIndependentMilestones(): void   |  | | --- | | * *findIndependentMilestones() finds independent milestones. (out-degree of 0)* | |
| +findAPs(): void   |  | | --- | | * *findAPs() finds the articulation points of the graph. (if articulation point removed, graph disconnects)* * *calls findAPsHelper() to find articulation points for non-visited vertices.* | |
| +topoSort(): void   |  | | --- | | * *topoSort() does topological sorting with the graph data based on in-degrees.* | |
| +criticalPath(): void   |  | | --- | | * *criticalPath() finds the critical path of the graph to use as minimum starting time of activities.* | |
| +findSlackTimes(): void   |  | | --- | | * *findSlackTimes() finds slack time for each edge / vertex to determine possible delay time based on critical path.* | |
| +findDependencyStats(): void   |  | | --- | | * *findDependencyStats() finds values and vertices of the highest and lowest in-degree nodes excluding source node.* | |
| +printGraphInformation(): void   |  | | --- | | * *printGraphInformation() prints graph information with formatted output.* | |
| +printGraph(): void   |  | | --- | | * *printGraph() prints adjacency list with formatted output.* | |
| +printDependencyStats(): void   |  | | --- | | * *printDependencyStats() prints dependency statistics with formatted output.* | |
| +printTopoSort(): void   |  | | --- | | * *printTopoSort() prints topological sort with formatted output.* | |
| +printAPs(): void   |  | | --- | | * *printAPs() prints graph articulation points with formatted output.* | |
| +printCriticalPath(): void   |  | | --- | | * *printCriticalPath() prints critical path with formatted output.* | |
| +printSlackTimes(): void   |  | | --- | | * *printSlackTimes() prints slack times with formatted output.* | |

****

**Big-Oh for each function:**

* getArguments(): O(1)
* readGraph(): O(|E|)
* isValidProject(): O(|V|)
* getTaskCount(): O(1)
* findGraphInformation(): O(1)
* findKeyMileStone(): O(|V| + |E|)
* findNodePoint(): O(|V| + |E|)
* findIndependentMilestones(): O(|V| + |E|)
* findAPs(): O(|V| + |E|)
* topoSort(): O(|V| + |E|)
* criticalPath(): O(|V| + |E|)
* findSlackTimes(): O(|V|)
* findDependencyStats(): O(1)
* printGraphInformation(): O(1)
* printGraph(): O(|V| + |E|)
* printDependencyStats(): O(1)
* printTopoSort(): O(|V|)
* printAPs(): O(|V|)
* printCriticalPath(): O(|V|)
* printSlackTimes(): O(|V|)

**Detailed description of data structure for graph (node type):**

* I plan to use an adjacency list for this assignment, I have two structs.

**struct idxNode**

**{**

**lNode \*list;**

**lNode head;**

**};**

**struct lNode**

**{**

**int weight;**

**int vertex;**

**lNode \* next;**

**};**

* Macro struct being idxNode, which contains a linked list, and a head that points to the start of the linked list. Micro struct being lNode, which is the linked list node structure, which contains a weight variable, vertex variable and next variable.
* idxNode represents the “array” part of the adjacency matrix, representing an index via base vertex, and that index containing a linked list that consists of the linked vertices. lNode represents the individual nodes of the linked list, which contains the vertex and the weight.
* For example, if v1 would be index of 1, connected to v2 would mean that lNode’s head → vertex = 2, etc.

**Summary of other data structures proposed with explanation:**

* Another data structure proposed to this program was adjacency matrix, but since we have a ridiculously large amount of vertices in bigger test files, using O(|V|2) would take up a tremendous amount of memory, which is why adjacency lists were done instead. In addition, the graphs data is also very sparse, making it more efficient to use adjacency lists’ O(|V| + |E|).

Total word count: 1,036 words