origoldbsc@gmail.com

תכנות מערכות 2

מטלה 2 – גרפים והעמסת אופרטורים

הגדרות

- גרף G הוא קבוצה סופית ולא ריקה של קודקודים.
- גרף ריק ($empty\ graph$) הוא גרף ללא צלעות, אך עם קודקודים.
- הצלע $u \rightsquigarrow v$ והצלע $v \rightsquigarrow u$ יצרו מעגל אם $w(uv) \neq w(vu)$. אם $w(uv) = w(vu)$ הצלע $u \leftrightsquigarrow v$ היא צלע לא מכוונת ולא תיצור מעגל בין u ל-v ולהיפך. בנוסף, מעגל יכול להיווצר בין קודקוד לעצמו אם הוא מכיל לולאה (כלומר, צלע מעצמו לעצמו).
- עבור משתנים ומתודות שאינם, לפי הגדרה, יכולים להחזיק מספרים שליליים נבחר הטיפוס $size\_t$.

$Graph$

עבור המחלקה לייצור גרף קיימים שני קבצים: $graph.cpp$ ו-$graph.hpp$.

לצורך כך נערך קובץ ה-$header$ בצורה הבאה:

```
namespace ariel {
class Graph
{
    private:
        vector<vector<int>> _adjacencyMatrix;
        size_t _numVertices;
        size_t _numEdges;
        bool _isDirected;

        bool checkDirected();
        size_t countEdges();
        bool isSubgraph(const Graph& other) const;
        bool Graph::compareMagnitude(const Graph& other) const

    public:
        Graph();
        void loadGraph(vector<vector<int>>& matrix);
        string printGraph() const;
        size_t getNumVertices() const;
        size_t getNumEdges() const;
        bool isGraphDirected() const;
        vector<vector<int>>& getAdjacencyMatrix();
        Graph& operator+();
        Graph operator+(const Graph& other) const;
        Graph& operator+=(int scalar);
        Graph& operator-();
        Graph operator-(const Graph& other) const;
        Graph& operator-=(int scalar);
        Graph operator*(const Graph& other) const;
```

```
        Graph& operator*=(int scalar);
        Graph& operator/=(int scalar);
        bool operator==(const Graph& other) const;
        bool operator!=(const Graph& other) const;
        bool operator<(const Graph& other) const;
        bool operator<=(const Graph& other) const;
        bool operator>(const Graph& other) const;
        bool operator>=(const Graph& other) const;
        Graph& operator++();
        Graph operator++(int);
        Graph& operator--();
        Graph operator--(int);
        friend ostream& operator<<(ostream& output, const Graph& graph);
    };
}
```

המחלקה כוללת את השדות **הפרטיים** הבאים:

שדה	הסבר
`vector<vector<int>> adjacencyMatrix`	מטריצת שכנויות המייצגת גרף
`size_t _numVertices`	משתנה המחזיק את מספר הקודקודים בגרף
`size_t _numEdges`	משתנה המחזיק את מספר הצלעות בגרף
`bool _isDirected`	דגל המסמל אם הגרף מכוון או לא

בנוסף, המחלקה גרף כוללת את המתודות **הפומביות** הבאות:

מתודה	הסבר
`Graph()`	בנאי ברירת מחדל. עם יצירת מופע חדש של גרף, מספר הקודקודים והצלעות מאותחלים להיות 0. בנוסף, הגרף מוגדר להיות גרף לא מכוון.
`void Graph::loadGraph(vector<vector<int>>& matrix)`	המתודה טוענת גרף ממטריצת שכנויות המתקבלת כקלט. אם המטריצה ריקה, מוחזרת הודעת שגיאה; וכן גם אם המטריצה אינה ריבועית. בנוסף, המתודה משתמשת בפונקציות עזר כדי לסמן אם הגרף מכוון או לא; וכן כדי לספור את מספר הקודקודים בהתאם לסוג הגרף (מכוון או לא).
	פונקציות עזר: - `checkDirected()` - `size_t countEdges()`
`void Graph::printGraph() const`	המתודה מדפיסה גרף בתצורה של מטריצת שכנויות.

`size_t Graph::getNumVertices() const`	המתודה מחזירה כמה קודקודים קיימים בגרף.
`size_t Graph::getNumEdges()const`	המתודה מחזירה כמה צלעות קיימות בגרף.
`bool Graph::isGraphDirected()const`	המתודה מחזירה אם גרף הוא מכוון או לא.
`vector<vector<int>>& Graph::getAdjacencyMatrix()`	המתודה מחזירה מצביע (reference) למטריצת השכנויות המייצגת כאמור גרף.
`Graph& Graph::operator+()`	**אופרטור אונרי +** : המתודה מחזירה את הגרף שעליו מופעל האופרטור ללא שינוי.
`Graph Graph::operator+(const Graph& other) const`	**אופרטור + בין שני גרפים (חיבור גרפים)** : תחילה המתודה בודקת אם הגרפים בעלי אותו סדר גודל, וששתיהן ריבועיות. אם הבדיקה נכשלת – נזרקת שגיאה. אחרת, המתודה עוברת כניסה-כניסה בשתי המטריצות המייצגות ומחברת את כניסות `other` ל-`this`. המתודה מחזירה את הגרף המתקבל.
`Graph& Graph::operator+=(int scalar)`	**אופרטור + סקלר (הוספת סקלר לגרף)** : המתודה מוסיפה סקלר, שניתן כפרמטר, לכל צלעות הגרף שאינן 0. משמע, המתודה לא מוסיפה/גורעת מגרף צלעות שאינן קיימות.
`Graph& Graph::operator-()`	**אופרטור אונרי -** : המתודה מחזירה את הגרף שעליו מופעל האופרטור כאשר כל צלעות הגרף מקבלות את המשקל ההופכי למשקלן הנוכחי.
`Graph Graph::operator-(const Graph& other) const`	**אופרטור - בין שני גרפים (חיסור גרפים)** : תחילה המתודה בודקת אם הגרפים בעלי אותו סדר גודל, וששתיהן ריבועיות. אם הבדיקה נכשלת – נזרקת שגיאה. אחרת, המתודה עוברת כניסה-כניסה בשתי המטריצות המייצגות ומחסרת את כניסות `other` מ-`this`. המתודה מחזירה את הגרף המתקבל.
`Graph& Graph::operator-=(int scalar)`	**אופרטור - סקלר (הפחתת סקלר מהגרף)** : המתודה מחסירה סקלר, שניתן כפרמטר, מכל צלעות הגרף שאינן 0. משמע, המתודה לא מוסיפה/גורעת מהגרף צלעות שאינן קיימות.
`Graph Graph::operator*(const Graph& other) const`	**אופרטור * בין שני גרפים (כפל גרפים)** : תחילה המתודה בודקת אם הגרפים בעלי אותו סדר גודל. אם הבדיקה נכשלת – נזרקת שגיאה. אחרת, המתודה מבצעת הכפלת מטריצות.

אופרטור * סקלר (הכפלת סקלר בגרף): המתודה מכפילה סקלר, שניתן כפרמטר, מכל צלעות הגרף.	`Graph&` `Graph::operator*=(int scalar)`
אופרטור / סקלר (חילוק סקלר מהגרף): המתודה מחלקת בסקלר, שניתן כפרמטר, את כל צלעות הגרף. ככל שהסקלר שהועבר כפרמטר הוא 0 המתודה זורקת שגיאה.	`Graph&` `Graph::operator/=(int scalar)`
אופרטור == (שוויון גרפים): גרפים G_1 ו-G_2 שווים אם הם מאותו סדר גודל ומכילים את אותן הצלעות <u>או</u> אם G_1 לא גדול מ-G_2 וגם G_2 לא גדול מ-G_1. לכן, המתודה פועלת באופן הבא: - אם מספר הקודקודים בשני הגרפים שונה, המתודה מחזירה שקר. - אחרת, המתודה בודקת כניסה-כניסה במטריצות המייצגות את הגרפים ובודקת אם משקלי הצלעות שקולים. אם משקלי הצלעות זהים, וגם שני הגרפים מאותו סדר גודל, המתודה מחזירה אמת. - אחרת, המתודה בודקת אם: `!(this > other)` **וגם** `!(this < other)`.	`bool` `Graph::operator==(const Graph& other) const`
אופרטור =! (אי-שוויון גרפים): גרפים מקיימים את היחס =! אם הם אינם שווים, ולכן המתודה מחזירה אמת או שקר בהתאם לביטוי הבא: `!(this == other*)`	`bool` `Graph::operator!=(const Graph& other) const`
אופרטור < (G_1 קטן ממש מ-G_2): גרף G_2 גדול ממש מגרף G_1 (שקול: G_1 קטן ממש מ-G_2) אם: 1. הגרף G_1 מוכל ממש בגרף G_2. 2. אם לא – גרף G_2 גדול מגרף G_1 אם מספר הצלעות ב-G_2 גדול ממספר הצלעות ב-G_1. 3. אם בכל זאת מספר הצלעות זהה, אז הגרף G_2 גדול מהגרף G_1 אם המטריצה המייצגת של G_2 בעלת סדר גודל גבוה יותר משל G_1. לכן, המתודה תחילה בודקת אם G_1 הוא תת-גרף ב-G_2 (או להיפך). אם לא – את מספר הצלעות; ואם לא – אז בודקת את סדר הגודל של שני הגרפים.	`bool` `Graph::operator<(const Graph& other) const`
פונקציית עזר: - `isSubgraph(other)` - `compareMagnitude (other)`	

אופרטור <= (G_1 קטן/שווה ל-G_2):
המתודה מחזירה אמת או שקר בהתאם לבדיקה, שמסתכמת על מתודות קיימות:
`this < other || this == other`
	`bool`
	`Graph::operator<=(const Graph& other)`
	`const`

אופרטור > (G_1 גדול ממש מ-G_2):
המתודה עושה שימוש במתודה עבור האופרטור <, ובהתאם ללוגיקה שם.
	`bool`
	`Graph::operator>(const Graph& other)`
	`const`

אופרטור >= (G_1 גדול/שווה ל-G_2):
המתודה מחזירה אמת או שקר בהתאם לבדיקה, שמסתכמת על מתודות קיימות:
`this > other || this == other`
	`bool`
	`Graph::operator>=(const Graph& other)`
	`const`

אופרטור ++ (הוספת 1 לצלעות גרף, prefix): המתודה מוסיפה 1 לצלעות גרף, ואינה מוסיפה זאת לצלעות שאינן קיימות.	`Graph&` `Graph::operator++()`
אופרטור ++ (הוספת 1 לצלעות גרף, postfix): המתודה מוסיפה 1 לצלעות גרף, ואינה מוסיפה זאת לצלעות שאינן קיימות.	`Graph` `Graph::operator++(int)`
אופרטור -- (החסרת 1 מצלעות גרף, prefix): המתודה מחסירה 1 מצלעות גרף, ואינה מפחיתה זאת מצלעות שאינן קיימות.	`Graph&` `Graph::operator--()`
אופרטור -- (החסרת 1 מצלעות גרף, postfix): המתודה מחסירה 1 מצלעות גרף, ואינה מפחיתה זאת מצלעות שאינן קיימות.	`Graph` `Graph::operator--(int)`

אופרטור פלט <<:
המתודה מדפיסה את הגרף תוך שימוש בפונקציה `printGraph()`.
	`ostream&`
	`operator<<(ostream& output, const`
	`Graph& graph)`

<u>הערה</u>: חלק מהמתודות הוגדרו כ-$const$, כהגדרה, מאחר שהן אינן אמורות לשנות את האובייקט גרף, בנוספת חלק מהפרטים של המתודות הוגדרו גם הם כ-$const$ כדי להגדירן באופן שהן לא ישנו את הערך המועבר.

בנוסף, המחלקה גרף כוללת את הפונקציות **הפרטיות** הבאות:

פונקציה	הסבר
`bool` `checkDirected()`	הפונקציה מחזירה אם גרף הוא מכוון או לא, על-ידי מעבר על כל הצלעות בגרף ובדיקה אם הערך במטריצת השכנויות $u \rightsquigarrow v$ שקול לערך $v \rightsquigarrow u$, או לאו.
`size_t` `countEdges()`	הפונקציה מחזירה את מספר הצלעות בגרף, תוך שימת דגש אם הגרף מכוון או לא.

`bool`
`Graph::isSubgraph(const Graph&`
`other) const`
	הפונקציה בודקת אם גרף הוא תת-גרף של גרף אחר, ומחזיקה אמת או שקר בהתאם לבדיקה. הפונקציה פועלת באופן הבא: תחילה הפונקציה בודקת את מספר

	הקודקודים כבדיקה ראשונית, שכן גרף "מוכל" לא יכול להכיל יותר קודקודים מגרף "מכיל". לאחר מכן, אם מספר הקודקודים אם לגרף הנוכחי ($this$) יש ערך שאינו אפס במיקום מסוים, ובמיקום המקביל במטריצה האחרת ($other$) נמצא 0, זה אומר שלגרף הנוכחי ישנה צלע שאינה קיימת בגרף השני. במקרה כזה, הפונקציה מחזירה מיד $false$ מכיוון שהגרף הנוכחי אינו יכול להיות תת-גרף של הגרף האחר. אם הפונקציה משלימה את הלולאות בלי למצוא שום צלע בגרף שאינה קיימת בגרף השני, זה אומר שכל הצלעות בגרף הנוכחי נמצאות גם בגרף השני. במקרה זה, הפונקציה מחזירה $true$.
`bool Graph::bool compareMagnitude(const Graph& other) const`	הפונקציה משווה את סדר הגודל של שתי המטריצות המייצגות את שני הגרפים ומחזירה אמת או שקר בהתאם לסיטואציה המתאימה.

Algorithms

עבור המחלקה של האלגוריתמים קיימים שני קבצים: *Algorithms.cpp* ו-*Algorithms.hpp*.

המחלקה כוללת את המתודות **הפומביות** הבאות:

מתודה	הסבר
`bool isConnected(Graph& graph)`	המתודה בודקת אם גרף הוא קשיר או לא, בהנחה ששורש הגרף הוא הקודקוד 0. אם הגרף הוא בעל קודקוד אחד – מחזירה אמת מאחר שהגרף קשיר באופן טריוויאלי. אחרת, בודקת האם ניתן להגיע מקודקוד 0 לכל קודקוד אחר באמצעות האלגוריתם BFS. אם אחד הקודקודים לא "בוקר" מחזירה שקר; אחרת אמת.
	פונקציית עזר: - `bfs(graph, startVertex, visited's vector)`
`bool isStronglyConnected(Graph& graph)`	המתודה בודקת אם גרף הוא קשיר חזק או לא. אם הגרף הוא בעל קודקוד אחד – מחזירה אמת מאחר שהגרף קשיר חזק באופן טריוויאלי. אחרת, בודקת האם ניתן להגיע מכל קודקוד לכל קודקוד באמצעות האלגוריתם BFS. אם אחד מקודקודים הגרף באחת הריצות לא "בוקר" מחזירה שקר; אחרת אמת.
	פונקציית עזר: - `bfs(graph, startVertex, visited's vector)`
`string shortestPath(Graph& graph, size_t start, size_t end)`	המתודה בודקת מהו המסלול הקצר ביותר (אם קיים) בין קודקוד מקור לקודקוד יעד. אם קיים מסלול, המתודה מחזירה את המסלול כמחרוזת; אחרת מחזירה שלא קיים מסלול כזה. המתודה משתמשת באסטרטגיה הבאה כדי להחליט באיזה צורה לבצע את המשימה: אם הגרף אינו ממושקל, המתודה משתמשת באלגוריתם BFS; אם הגרף ממושקל אך עם משקלים חיוביים, המתודה משתמשת באלגוריתם של Dijkstra; ואם הגרף ממושקל אך עם צלעות בעלות משקל שלילי, המתודה משתמשת באלגוריתם של Bellman-Ford. המתודה גם בודקת תקינות קלט: (א) האם הקודקודים חוקיים (מאחר שנעשה שימוש בטיפוס *size_t* לא ניתן לקבל קודקוד מקור כשלילי); (ב) קודקוד המקור שונה מקודקוד היעד.
	פונקציית עזר: - `bfsShortestPath(graph, start, end)` - `bellmanFordShortestPath(graph, start, end)` - `dijkstraShortestPath(graph, start, end)` - `checkGraphType(graph)` - `extractSubgraph(graph, start, end, subgraph)`

`string isContainsCycle(Graph& graph)`	המתודה בודקת אם הגרף קיים מעגל בגרף. אם קיים מעגל, מחזירה אותו כמחרוזת; אחרת מחזירה "0". על מנת לבצע את המשימה המתודה משתמשת באלגוריתם DFS. המתודה עוברת על כל קודקודי הגרף ומנסה למצוא מעגל באמצעות איתור back-edge המהווה אינדיקציה למעגל.
	פונקציית עזר: - `dfs_cycle(graph, vertex, visited's vector, recStack's vector, parent's vector, isDirected)`
`string isBipartite(Graph& graph)`	המתודה בודקת אם הגרף הוא דו-צדדי. אם ניתן לחלק את קודקודי הגרף לשתי קבוצות, כך שלא תהיה צלע בין הצלעות בכל קבוצה, המתודה מחזירה את החלוקה; אחרת, מחזירה שהגרף אינו דו-צדדי. על מנת לבצע את המשימה, המתודה משתמשת בתצורה שונה של אלגוריתם DFS הכוללת צביעת קודקודים.
	פונקציית עזר: - `dfs_check(graph, currectVertex, colVec, color)` - `buildSet(set's vector)`
`string negativeCycle(Graph& graph)`	המתודה בודקת אם קיים מעגל **שלילי** בגרף. אם קיים מעגל שלילי, היא מחזירה אותו כמחרוזת; אחרת, מחזירה שאין מעגל שלילי בגרף. המתודה עושה שימוש בתצורה מסוימת של BF כאשר היא מנסה לבצע פעולת relax על צלעות הגרף לאיתור המעגל השלילי.
	פונקציית עזר: - `findNegativeCircle(graph)`

וכן את הפונקציות **הפרטיות** הבאות:

פונקציה	הסבר
`void bfs(Graph& graph, size_t startVertex, vector<bool>& visited, vector<size_t>& parent, size_t end, bool reverse = false)`	פונקציית עזר זו מבצעת תהליך של BFS על קודקוד בגרף על מנת לבדוק אם ניתן להגיע ממנו לכל קודקודי הגרף. התוצאות נשמרות בווקטור המועבר לפונקציה כפרמטר בשם visited. <u>הערה</u>: הפונקציה "הורחבה" כך שתקבל גם וקטור עבור לשמירת הורי הקודקודים וקודקוד יעד, וכן ביצוע של BFS בכיוון ההפוך על מנת שתהיה שמישה גם עבור מציאת המסלול הקצר ביותר בין שתי נקודות.
	מתודות קשורות: - `isConnected(graph)` - `isStronglyConnected(graph)` - `shortestPath(graph, start, end)`

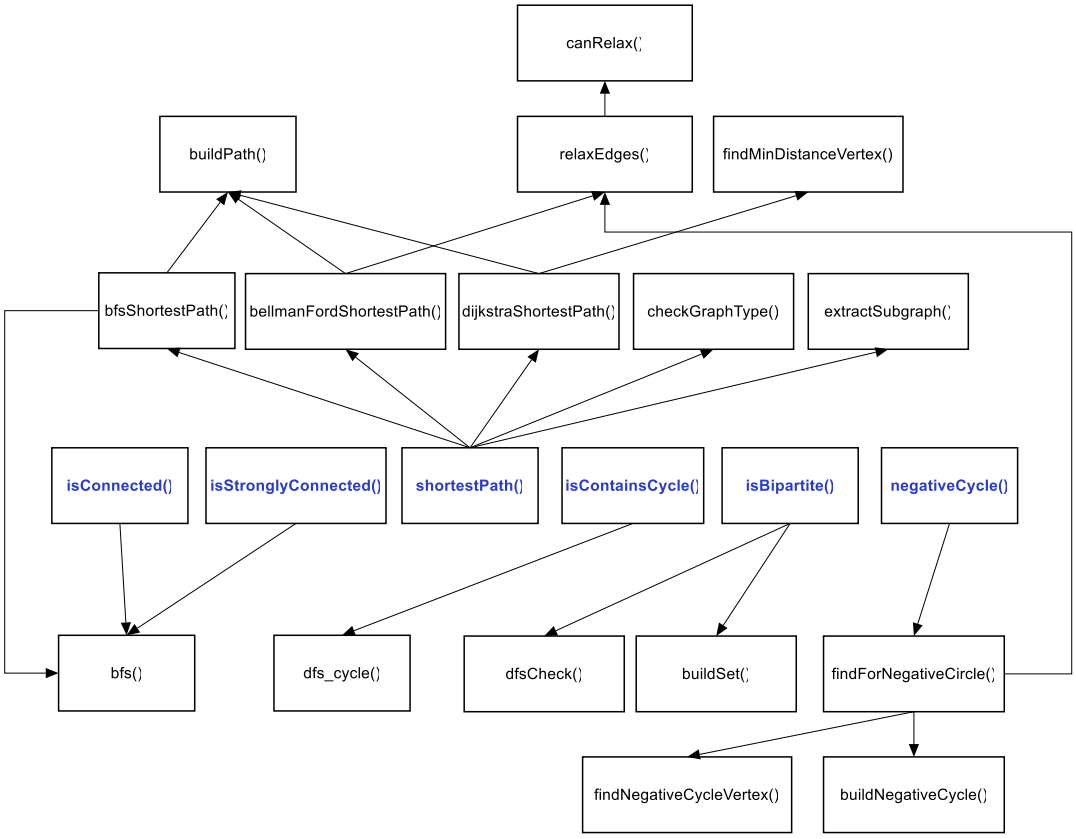
`void extractSubgraph(Graph& graph, size_t start, size_t end, Graph& subgraph)`	פונקציית עזר זו מקבלת גרף ומחלצת את תת-הגרף שמכיל את כל הקודקודים והצלעות שעשויים להכיל מסלול בין קודקוד ההתחלה לקודקוד הסיום במסלול. את התוצאה היא שומרת לתוך הכתובת של `subgraph` המועבר כפרמטר לפונקציה. הפונקציה עובדת בצורה הבאה: היא מריצה BFS מקודקוד ההתחלה כדי לאתר את כל הקודקודים שניתנים להגעה ממנו; לאחר-מכן היא מריצה BFS פעם נוספת הפעם מקודקוד הסיום, על הגרף ההפוך. לאחר מכן, הפונקציה לוקחת את החיתוך בין שתי התוצאות. באופן זה הפונקציה גם מייעלת את בדיקת המסלול הקצר ביותר; וכן מטפלת במקרה מיוחד עבור BF כאשר קיים מסלול קצר ביותר בין שני קודקודים, אך גם מסלול שלילי (במקום אחר בגרף שאינו קשר למסלול בין שתי הקודקודים הנתונים).
	מתודות קשורות: - `shortestPath(graph, start, end)`
`string bfsShortestPath(Graph& graph, size_t start, size_t end)`	פונקציית עזר זו מקבלת קודקוד מקור וקודקוד יעד ובודקת עבור גרף לא ממושקל אם קיים מסלול בין הקודקודים. אם קיים מסלול, הפונקציה מחזירה לפונקציית המקור את המסלול כמחרוזת; אחרת, מחזירה שאין מסלול בין קודקוד המקור לקודקוד היעד. ביצוע התהליך ממומש באמצעות תור, שבאמצעותו בכל פעם בודקים את השכנים של הקודקוד שנמצא בראש התור, ומעדכנים בין היתר את האב על מנת לשחזר את המסלול הקצר (ככל שיימצא). שחזור המסלול מבוצע באמצעות פונקציית עזר.
	מתודות קשורות: - `shortestPath(graph, start, end)` **פונקציות עזר**: - `bfs(graph, startVertex, visited's vector, parent's vector, end)` - `buildPath(start, end, parent's vector)`
`string bellmanFordShortestPath(Graph& graph, size_t start, size_t end)`	פונקציית עזר זו מקבלת קודקוד מקור וקודקוד יעד ובודקת עבור גרף ממושקל עם צלעות שליליות אם קיים מסלול בין הקודקודים. אם קיים מסלול, הפונקציה מחזירה לפונקציית המקור את המסלול כמחרוזת; אחרת, מחזירה שאין מסלול בין קודקוד המקור לקודקוד היעד. ביצוע התהליך מבוצע באמצעות ביצוע פעולת relax על קודקודי הגרף $|V| - 1$ פעמים, באמצעות פונקציית עזר המיועדת לכך. לאחר מכן, מבוצעת פעולה נוספת שכזו, המבוצעת גם היא באמצעות פונקציית עזר. ככל שלא נמצא מעגל שלילי, אזי נכנסת לפעולה פונקציית עזר נוספת שמחזירה את המעגל כמחרוזת.

	מתודות קשורות: - shortestPath(graph, start, end) **פונקציות עזר**: - relaxEdges(g, distance's vector, parent) - buildPath(start, end, parent's vector)
String dijkstraShortestPath(Graph& graph, size_t start, size_t end)	פונקציית עזר זו מקבלת קודקוד מקור וקודקוד יעד ובודקת עבור גרף ממושקל עם צלעות אי-שליליות אם קיים מסלול בין הקודקודים. אם קיים מסלול, הפונקציה מחזירה לפונקציית המקור את המסלול כמחרוזת; אחרת, מחזירה שאין מסלול בין קודקוד המקור לקודקוד היעד. ביצוע בתהליך מבוצע באופן הבא: נעבור כל קודקודי הגרף; נמצא בכל פעם את הקודקוד עם המרחק הקטן ביותר, באמצעות פונקציית עזר, ונצבע פעולת relax על הקודקודים שטרם "בוקרו". נחזור על התהליך האמור עבור כל הקודקודים. אם בסוף התהליך, קודקוד היעד הוא במרחק של INT_MAX מקודקוד המקור, הרי שאין מסלול. אחרת, נשתמש בפונקציית עזר לשחזור המסלול הקצר ביותר באמצעות שמירת ההורה של כל קודקוד שנעשתה בתהליך הריצה.
	מתודות קשורות: - shortestPath(graph, start, end) **פונקציות עזר**: - findMinDistanceVertex(distance's vector, visited's vector) - buildPath(start, end, parent's vector)
pair<bool, bool> checkGraphType(Graph& graph)	פונקציית עזר זו בודקת אם הגרף ממושקל, וכן אם קיימת צלע קטנה מ-0. אם הגרף ממושקל, מחזירה אמת; אחרת, שקר. התהליך הוא בדיקת כל צלעות הגרף כך שהן לא 0 (משמע, לא קיימת צלע) וגם לא 1 (משמע, הצלע חסרת משקל). בנוסף, אם נמצאה צלע שלילי, מחזירה אמת; אחרת, שקר. הפונקציה מחזירה את התוצאה כזוג של משתנים בוליאניים.
	מתודות קשורות: - shortestPath(graph, start, end)
string dfs_cycle(Graph& graph, size_t vertex, vector<bool>& visited, vector<bool>& recStack, vector<size_t>& parent, bool isDirected)	פונקציית עזר זו מבצעת ריצה של DFS באופן רקורסיבי על מנת לאתר מעגל בגרף, באמצעות ניסיון לאתר back-edge (כאשר מדובר בגרף לא מכוון). ככל שנמצאת צלע שכזו, הפונקציה בונה את המעגל תוך התחקות אחרת הורי-הקודקודים; אחרת מחזירה ערך ריק. אם הגרף הוא מכוון, אזי הפונקציה מחפש מעגל פשוט.
	מתודות קשורות: - isContainsCycle(Graph& graph)

`bool dfsCheck(Graph& graph, size_t currectVertex, vector<int>& colorVec, int color)`	פונקציית עזר זו מקבלת בין היתר גרף, צביעה של קודקוד התחלה וקודקוד התחלה, ובודקת אם הגרף הוא 2-צביע. זאת, באמצעות תהליך של מעבר על כל קודקודי הגרף וצביעתם לסירוגין. אם נוצרת סתירה בצביעה – משמע שני קודקודים סמוכים נצבעו באותו צבע – הפונקציה מחזירה שקר; אחרת, אמת.
	מתודות קשורות: - `isBipartite(graph)`
`string buildSet(vector<size_t>& set)`	פונקציית עזר זו מקבלת קבוצה של קודקודים, ומחזירה אותם כמחרוזת.
	מתודות קשורות: - `isBipartite(graph)`
`String findNegativeCircle(Graph& graph)`	פונקציית עזר זו מנסה לאתר מעגל בגרף באמצעות אלגוריתם Bellman-Ford (עם שינוי מסוים, שכן האלגוריתם הקלאסי מאתר מסלול בין קודקוד מקור לקודקוד יעד). בפונקציה מבצעים פעולת relax על כל הקודקודים, ואם הפעולה האחרונה מביאה ליצירת מעגל, אזי הפונקציה מתחקה אחר הורי הקודקודים כדי לשחזר אותו. זהו גם הערך המוחזר אם קיים; אחרת, הפונקציה מחזירה שאין מעגל שלילי.
	מתודות קשורות: - `negativeCycle(graph)` **פונקציות קשורות**: - `relaxEdges(g, distance's vector, parent's vector` - `findNegativeCycleVertex(graph, distance's vector, parent's vector)` - `buildNegativeCycle(graph, vertex, parent's vector)`
`String buildPath(size_t start, size_t end, vector<size_t>& parent)`	פונקציית עזר זו מקבל קודקוד מקור ויעד, וכן וקטור של הורי הקודקודים ומשחזרת את המסלול מקודקוד המקור לקודקוד היעד. זו גם הערך המוחזר.
	פונקציות קשורות: - `bfsShortestPath(graph, start, end)` - `bellmanFordShortestPath(g, start, end)` - `dijkstraShortestPath(graph, start, end)`
`bool relaxEdges(Graph& graph, vector<int>& distance, vector<size_t>& parent)`	פונקציית עזר זו מבצעת פעולת relax על קודקודי גרף. בעבור פעולה זו היא משתמשת בפונקציית עזר.
	פונקציות קשורות: - `canRelax(graph, vertex_u, vertex_v, weight, distance's vector, parent's vector)`

פונקציית עזר זו בודקת מספר תנאים כדי לקבוע אם ניתן לבצע פעולת relax על צלע, ביניהן, אם משקלה לא 0, וגם אם המרחק שלה לא INT_MAX, וגם המרחק של u בתוספת הצלע uv קטן מהמרחק ישירות ל-v. כמו כן, נעשית בדיקה אם מדובר בגרף מכוון, ועבור גרף לא מכוון, אם ההורה של u אינו v כדי למנוע בדיקות כפולות.	`bool canRelax(Graph& graph, size_t vertex_u, size_t vertex_v, int weight, vector<int>& distance, vector<size_t>& parent)`
פונקציות קשורות: `canRelax(graph, vertex_u, vertex_v, weight, distance's vector, parent's vector)`	
פונקציית עזר זו מאתרת קודקוד שנמצא על מעגל שלילי. הפונקציה עוברת על כל הצלעות לאחר שכבר בוצע להם פעולת relax. אם ניתן לבצע פעולה נוספת, הרי שהקודקוד שממנו יצאה הצלע הוא חלק ממעגל שלילי. קודקוד זה יוחזר.	`size_t findNegativeCycleVertex(Graph& graph, vector<int>& distance, vector<size_t>& parent)`
`canRelax(graph, vertex_u, vertex_v, weight, distance's vector, parent's vector)`	
פונקציית עזר זו בונה מעגל שלילי שנמצא בגרף, ובונה ייצוג מחרוזת של המעגל בסדר הפוך מזה שנשמר. אם אכן נמצא מעגל שלילי, הפונקציה מחזירה מחרוזת של המעגל השלילי.	`String buildNegativeCycle(Graph& graph, vector<int>& distance, vector<size_t>& parent)`

להלן תרשים המתאר את היחס בין המתודות והפונקציות השונות במחלקת האלגוריתמים:



$Makefile$

לצורך הרצת התוכנית והבדיקות נערך קובץ $makefile$ כדלקמן:

```
# General macros
CXX = clang++
CXXFLAGS = -std=c++11 -Werror -Wsign-conversion
VALGRIND_FLAGS = -v --leak-check=full --show-leak-kinds=all --error-
exitcode=99

# Macros for source files and headers files
SOURCES = Graph.cpp Algorithms.cpp
HEADERS = Graph.hpp Algorithms.hpp
DEMO_SRC = Demo.cpp
TEST_SRC = Test.cpp
TEST_COUNTER_SRC = TestCounter.cpp

# Macros for object and headers files
OBJECTS = Graph.o Algorithms.o

# Main target: Build and run the demo
run: demo
    ./demo

# Build the demo exe file from object files
demo: Demo.o $(OBJECTS)
    $(CXX) $(CXXFLAGS) Demo.o $(OBJECTS) -o demo

# Build the test exe file that includes the tests
test: TestCounter.o Test.o $(OBJECTS)
    $(CXX) $(CXXFLAGS) TestCounter.o Test.o $(OBJECTS) -o test

# Run clang-tidy
tidy:
    clang-tidy $(SOURCES) -checks=bugprone-*,clang-analyzer-
*,cppcoreguidelines-*,performance-*,portability-*,readability-*,-
cppcoreguidelines-pro-bounds-pointer-arithmetic,-cppcoreguidelines-owning-
memory --warnings-as-errors=-* --

# Run valgrind
valgrind: demo test
    valgrind --tool=memcheck $(VALGRIND_FLAGS) ./demo 2>&1 | { egrep "lost|
at " || true; }
    valgrind --tool=memcheck $(VALGRIND_FLAGS) ./test 2>&1 | { egrep "lost|
at " || true; }

# Rule to compile Graph object file
Graph.o: Graph.cpp Graph.hpp
    $(CXX) $(CXXFLAGS) -c Graph.cpp -o Graph.o
```

```makefile
# Rule to compile Algorithms object file
Algorithms.o: Algorithms.cpp Algorithms.hpp
	$(CXX) $(CXXFLAGS) -c Algorithms.cpp -o Algorithms.o

# Rule to compile Demo object file
Demo.o: Demo.cpp $(HEADERS)
	$(CXX) $(CXXFLAGS) -c Demo.cpp -o Demo.o

# Rule to compile Test object file
Test.o: Test.cpp $(HEADERS)
	$(CXX) $(CXXFLAGS) -c Test.cpp -o Test.o

# Rule to compile TestCounter object file
TestCounter.o: TestCounter.cpp $(HEADERS)
	$(CXX) $(CXXFLAGS) -c TestCounter.cpp -o TestCounter.o

# Clean up command to remove all compiled files
clean:
	rm -f *.o demo test
```

$Testing$

הבדיקות בוצעו עבור המחלקות $Graph$, תוך בדיקת תקינות המחלקה $Algorithms$.

הבדיקות המלאות מצורפות לקובץ $Test.cpp$.