**ЧЕТИРИНАДЕСЕТА НАЦИОНАЛНА ОЛИМПИАДА**

**ПО ИНФОРМАЦИОННИ ТЕХНОЛОГИИ**

**1. ТЕМА: GRAPHical Learner**

**2. АВТОР:**

Косьо Николов, ЕГН 9803251047,

гр. Варна, кв. “Владислав Варненчик”, бл. 8, вх. 7, ет. 8, ап. 43,

тел: +359883700323, e-mail: [3c3code@gmail.com](mailto:3c3code@gmail.com), Математическа гимназия „Д-р Петър Берон“ - Варна, ученик от XI клас

**3. РЪКОВОДИТЕЛ:**

Юлия Илиева Димитрова, тел. +359887903407, [julidbg@gmail.com](mailto:jid@gmail.com), учител по

информатика и ИТ в Математическа гимназия „Д-р Петър Берон“ - Варна

**4. РЕЗЮМЕ:**

**4.1. Цели**

При решаване на задачи с графи се налага да се прави схема на графа. За да се тества резултата, даните от схемата трябва да се превърнат във формат, подходящ за компютъра. В някои случаи трябва да се прави и обратното - по дадени числови данни да се построи граф. При тези действия, особено, когато графът е по-голям, е възможно да се допуснат грешки, често незаслужено приписвани на тествания алгоритъм, което води до загуба на време в търсене на несъществуващи проблеми. Дори тези проблеми да бъдат разрешени, се налага проследяването на алгоритъм да се прави ръчно, което в повечето случаи означава отбелязване върху изображения – на хартия или на компютър. От програмиста се изисква да съпоставя числа на обекти от изображение, което отнема немалко време, особено когато „средата“ изисква значително време за триене. Също така, възможността за грешки отново е голяма. Всички тези фактори водят до редовно забавяне при работата с графи и усложняват учебния процес.

Приложението е създадено, с цел да разреши горепосочените проблеми и да улесни работата на ученици, студенти, преподаватели, програмисти, научни работници, любители и др. при решаване на теоретични и практически задачи, свързани с графи. То осигурява възможността за двете противоположни действия - създаване на граф с едновременното му визуално представяне и генериране на тестове за проверка на задания, използващи графи, както и обратното - по числови данни за графа, съхранени във файл, той да се визуализира, с възможност да се допълва схемата или да се коригира. Освен това, приложението притежава и функционалнстта на визуален дебъгер, чрез който може да се проследи алгоритъм. Последното неимоверно би помогнало много в класните стаи.

В практиката съществуват само решения на частични проблеми от целия набор функционалнисти, реализирани в този проект.

**4.2. Основни етапи в реализацията**

1. търсене на разнообразни източници по темата на проекта, с цел проверка за съществуващи решения и проучване на нуждата от разработката
2. избор и проектиране на софтуерната архитектура
3. избор на графична среда и библиотека
4. проектиране и разработка на потребителския интерфейс
5. създаване и имплементиране на алгоритъм за чертане на граф
6. реализиране на възможност за зареждане на данни от файл и съхраняване
7. добавяне на интерфейс за достъп до **модула за** алгоритми
8. добавяне на модул за алгоритми на C++
9. отстраняване на грешки и тестване за разрешаване на логически проблеми

**4.3. Ниво на сложност**

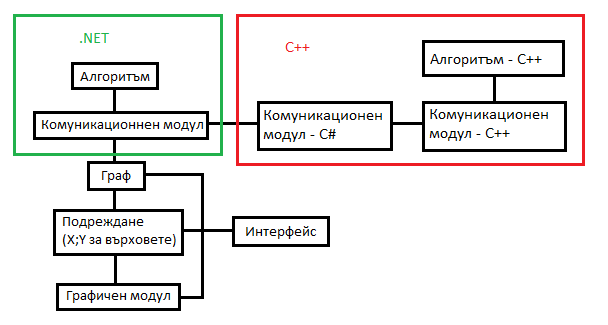
При реализацията трябваше да се решат няколко проблема:

* изграждане на софтуерна архитектура: след предварителен анализ бе съставена архитектура с висока модулярност, която позволява лесно надграждане
* избиране на подходящо вътрешно представяне на граф: използва се комбинирано представяне със списък на ребра и върхове. Към всеки връх и ребро могат да бъдат закачени свойства
* как да се подреди графа: трябваше да се избере подходящ алгоритъм и да се настрои. След проучване се избра алгоритъм, работещ със сили.
* какъв да е интерфейса: изборът беше между външен за основния прозорец интерфейс със Уиндоуски контроли и вграден интерфейс. Предопочетена бе втората опция и трябваше да бъдат създадени собствени интерфейсни елементи.
* как да се свърже приложението с алгоритми, написани на C++: използват се функции на Уиндоус за четене на памет и бе изграден протокол за комуникация.

**4.4. Логическо и функционално описание на решението**

Приложението се състои от шест модула:

* графичен модул - изобразява графичните обекти на интерфейса и на графа със съответните трансформации
* модул за подреждане на графа - реализира алгоритъма за автоматично естетическо подреждане на графа. Активира се по желание от потребителя, с цел по-прегледна визуализация на връзките в графа
* модул за управление на потребителския интерфейс
* модул за комуникация с алгоритъм на .NET
* модул за комункация с алгоритъм на C++: съдържа 2 части - една на C#, и една на C++
* модул за управление на работата с файлове



**4.5. Реализация**

Приложението е написано на езика за програмиране С#. За графичния модул е използвана библиотеката SFML.NET, която дава достъп от високо ниво до OpenGL. Използването на такава библиотека се налага, защото вградените графични библиотеки на Windows не са достатъчно бързи, за да се постигне желаната функционалност - получава се насичане на кадрите. Със SFML се получава плавна картина. Създадени са класове, представляващи ребра и върхове.

За подреждане на графа е използвана идеята на алгоритъма force-directed graph drawing, публикуван на сайта

<http://www.mathematica-journal.com/issue/v10i1/contents/graph_draw/graph_draw_3.html> със значителни авторови модификации. Идеята на този алгоритъм е, че на всеки връх се съпоставят сили на отблъскване, на върховетe, свързани с ребра - сили на привличане и на отблъскване. Тези сили в известна степен са аналогични на електрическите и еластичните(пружинни) сили от физиката. Енергията се дефинира като сбор на квадрата на силите, действащи върху върховете. На всяка стъпка алгоритъмът се опитва да намали общата енергия на системата, като движи върховете по посока на силата. Също така се променя коефициента, с който се прилагат силите, за да се получат по-бързи и точни резултати. Алгоритъмът за подреждане се изключва автоматично, когато подреди графа. Реализацията се намира в класа ForceSimulatorMkIIB. Създаден е интерфейс за алгоритъма(IForceSimulator) с цел лесна и бърза подмяна при нужда.

Интерфейсът на приложението е вграден в основната графична среда с цел програмата да има унитарен вид за потребителя. Създадена е йерархия от класове с цел ефективно визуализиране и обработка на събития от потребителя. Графичната среда се управлява от класа MainUI.

Работата с файлове е реализирана със стандарните .NET библиотеки.

Програмата съдържа вътрешна структура на граф, която позволява лесно приспособяване към различни условия. Върховете и ребрата(класове Vertex и Edge) наследяват класа PropertyHolder, който позволява към тях да се присвояват практически всякакви свойства. Свойство може да бъде например флаг, който да показва дали върха е обходен, цена на ребро, ниво на връх и т.н. Всяко свойство си има име, код(id) и стойност. Не е задължително стойността на свойството да е еднаква при всички елементи, които го притежават. За визуализацията е важно единствено типа да има смислен метод ToString. Има някои свойства със специални функции, които имат различен ефект – маркираност, цена на ребро, цвят.

За комуникация с алгоритъм, написан на .NET, е предоставен класа Connector. Той трябва да бъде наследен от клас в проекта на алгоритъма, и да се извикат и override-нат функции. Функциите, които могат направо да се използват, са:

* *SetVertices*: задава броя на върховете
* *AddEdge*: добавя ребро
* SetDirected: задава насоченост, стандартно – ненасочено(false)
* SetPollMode: указва дали програмата сама да търси промени в свойствата(активен режим), или тази отговорност да се прехвърли на потребителя(пасивен режим). Стандартно – активен(true)
* RegisterProperty: регистрира свойство. Свойствата могат да бъдат добавени към върхове и ребра. Могат да бъдат от всякакъв тип.
* SetVertexProperty: задава стойност на даденото свойство
* AddPropertyToVertices: добавя свойство към всички върхове с първоначална стойност
* SetEdgeProperty: аналогично на SetVertexProperty
* Pause: паузира алгоритъма - той може да бъде продължен от визуалната среда
* *SetupGui*: задължителна функция, която трябва да се извиква в началото на програмата
* *StartGui*: стартира визуалната среда

\*функциите в *Italic* са задължителни

Следните функции могат да се override-нат:

* GetVertexProperty
* GetEdgeProperty
* PreRun

Първите две са задължителни, когато са зададени свойства на ребрата или/и на върховете и програмата е в активен режим(SetPollMode(true)). Те се използват за извличане на информация от алгоритъма. Последната е по избор: тя се вика точно преди стартирането на визуалната среда. Connector създава нишка, в която се пуска MainUI.

За свързване на приложението с алгоритъм, написан на C++, са написани модули на C# и C++. За комуникацията се използват WinAPI функциите ReadProcessMemory и CreateRemoteThread. Следва описание на протокола за комуникация от страната на C++:

1. Програмата стартира приложението(C#) чрез CreateProcess. Локацията на приложението се взима от регистрите. На командния ред се предават 2 числа - id на процеса(взето с GetCurrentProcessId) и адреса на структура от типа StartUpData. Формата й е следния:

int threadId;

void\* pGraphData;

void\* pPropertyData;

void\* pVertexBuffer;

void\* pEdgeBuffer;

volatile bool\* pControlByte;

void\* fPtr;

Първият елемент е id на нишката на алгоритъма(взима се с GetCurrentThreadId). Той се използва от C# модула за отваряне на същата нишка. Следващите 4 елемента са указатели към различни буфери(описани по-надолу). pControlByte се използва за сигнализация кога алгоритъма е на пауза. Това се наложи, понеже алтернативните начини за проверка дали нишка е суспендирана не работят в друг процес, или са прекалено неудобни. Последният аргумент е пойнтър към функция, която приема C-string(такъв, който завършва с нула) и връша дължината му.

Буферът с данни за графа(pGraphData) има следният формат: [бр. върхове - int] [бр. ребра - int] [насоченост - bool]. Следват двойки от числа(int), представляващи начална и крайна точка на ребро. Броят им съотвества на зададения вече брой ребра. Този буфер се попълва чрез функциите *SetVertices*, *SetEdges* и *AddEdge*, извиквани от потребителя - програмист.

Вторият буфер - pPropertyData, съдържащ данни за свойствата - съдържа 2 пойнтъра, които сочат към двете му части, наречени условно Primary(първичен) и Secondary(вторичен). Разделението е направено с цел опростена и улеснена имплементация.

Първичният буфер съдържа информация за код на свойство и първоначалната му стойност. Формата е [дължина - int] [брой - int] . След това следват структури, започващи с байт, означващ кода на свойството, и стойността му по подразбиране. Дължината на последното е променлива.

Вторичният буфер съдържа информация за типа и името на свойството. Форматът е [брой елементи - int], последван от толкова на брой [тип - байт] [име - char\*].

Попълването на буферите за свойствата е дело на класовете PropertyPrimaryBuffer и PropertySecondaryBuffer. Те от своя страна се контролират от PropertyManager - при извикването на неговите функции RegisterProperty в двата буфера се пише информация. Чрез това разделение се постигат кратки и ясни функции. RegisterProperty се вика от потребителския код. При извикването се указват типа на свойството, дали ще се отнася за връх или ребро, името и първоначалната стойност. Връща се код на свойството, който може да бъде използван по-късно със SetProperty. При регистрация PropertyManager записва кой код на кой тип съответсва.

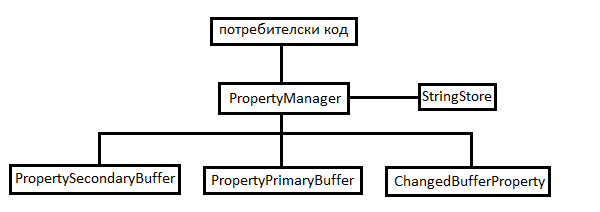
В момента на стартирането буферите за промени в ребра и върхове са празни(pVertexBuffer и pEdgeBuffer). pControlByte се попълва с адреса на контролния байт, използван от функцията Pause() - тя паузира алгоритъма. Пойнтърът към функция също се попълва при стартирането. Въпросната функция извиква в момента просто извиква strlen, но в бъдеще това може да бъде променено.

При създаването и обновяването на стойности се налага предаването на низове като информация. Очаква се, че повечето низове, подадени от потребителя, ще бъдат под формата на аргументи на функция, при което има опасност низа да бъде унищожен преди да бъде прочетен. За целта е създаден класа StringStore, който съхранява низове. За да се съхрани низ, се извиква Store - функцията приема char\* и връща char\*. Всеки низ се хешира и низ се добавя, само ако вече не съществува. StringStore се използва от PropertyManager.

2. След успешно стартиране на графичната среда, C++ програмата суспендира нишката си. Тя бива пускана след инициализирането на C# модула.

3. При изпълнението на алгоритъма може да се вика SetProperty на PropertyManager(с инстанция pm), за да се зададе стойност на свойство на даден елемент(връх или ребро). При извикването на тази функция PropertyManager проверява за коректност на типа и записва данни в буферите за промени във ребрата/върховете - те са от тип ChangedPropertyBuffer. Формата им е следния: [дължина - int], последвана от няколко [id на елемент - int] [код на свойство - char] [стойност - зависи]. Графичната среда може да прочете тази структура, защото вече знае дължината на стойностите за съответните кодове.

Диаграма на класовете в C++:

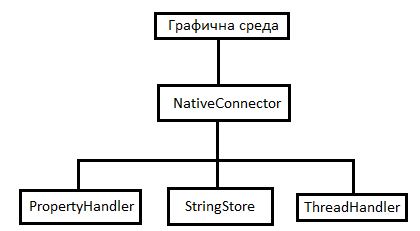


C# модула е поместен в проекта NativeAlgo. При пускането му се създава обект от тип NativeConnector, който наследява класа Connector. Той чете командната линия - уверява се, че са подадени аргументи, в противен случай изключва програмата. След това се четат данните за процеса с функцията *ReadProcess*. При успешно четене се викат функциите *SetupGui* и *StartGui* от Connector и графичната среда се стартира.

В ReadProcess последователността на действията е следната:

1. Отваря се handle към процеса на алгоритъма чрез OpenProcess. Използва се id-то от командната линия.
2. Чете се първоначалния буфер(StartUpData)
3. Отваря се handle към нишката на алгоритъма чрез OpenThread. Ползва се id от първоначалния буфер.
4. Създава се обект от типа ThreadHandler с новополучения handle. Този клас е отговорен за сигнализиране кога в алгоритъма е ударена Pause() и за продължаване на същата нишка. Тази функционалност е имплементирана съответно със събитието *ThreadSuspended* и функцията *Resume*. При създаването на класа се създава нова нишка в текущата програма, която периодично проверява за статуса на контролния байт в C++ модула. От WinAPI функциите се ползват *ResumeThread* и *ReadProcessMemory*
5. Четат се данните за графа(pGraphData) чрез *ReadGraphData*.
6. Създава се StringStore. Функциите му са аналогични на този от C++ - съхранява низове. Разликата е, че в случая се ползва речник указател - низ.
7. Четат се данните за свойствата. За целта се ползва PropertyHandler. Функциите му са донякъде сходни с тези на PropertyManager. В началото чете двата буфера и отваря 2 BinaryReader-а върху тях. Следва парарелно четене на буферите и регистриране на свойства. За последното в случая се ползват функциите *RegisterProperty* и *AddPropertyToVertices/Edges* на Connector. Създава се речник на кодовете(id) на свойствата, защото са възможни разминавания между C++ и C#.

След прочитането на първоначалните данни, NativeConnector има три задачи: да указва кога алгоритъма е на пауза, да чете променените свойства и да пуска алгоритъма отново по команда на визуалната среда. Първата и последната задача се изпълняват от ThreadHandler, а втората - от PropertyHandler, който чете буферите с промени(pVertexBuffer и pEdgeBuffer). Диаграма на класовете:



Разлики между модулите за комуникация с .NET и с C++: За разлика от C#, не се допускат промени в типа на свойството поради различията в езиците. Възможно е да се направи, но би се изисквало от потребителя да пише функции с указатели, което ще е сложно, а целта на проекта е да бъде лесен за използване. Друга разлика с .NET е, че няма възможност за активен режим на взимане на данни за свойства. В .NET това се осъществявя чрез функция, която се override-ва от потребителя, и се извиква от графичната среда. В C++ този подход не би бил удачен, защото би означавал извикване на CreateRemoteThread хиляди пъти в секунда. Други подходи биха изисквали много памет, затова в C++ присъства само пасивния режим, който не е сложен и е пестелив откъм памет.

Поддържането на приложението е улеснено от архитектурата - разделени са графичната среда, графовата логика, вход-изхода и комуникацията с алгоритъм.

**4.6. Описание на приложението**

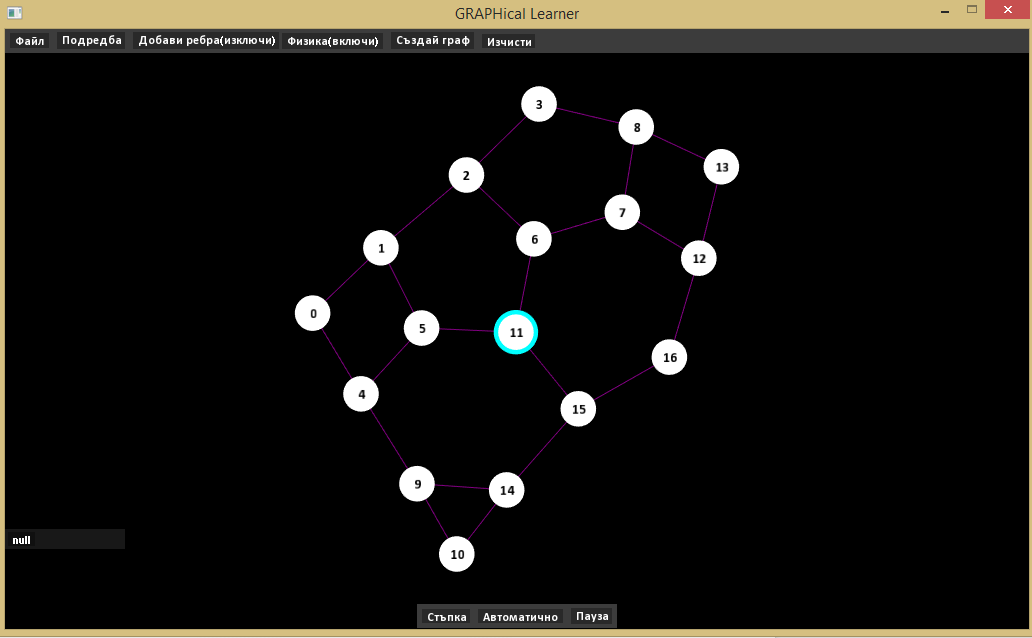
Приложението се инсталира лесно чрез Inno Setup.

Стартирането в режим редакция на граф става чрез GUI\_only.exe.

След стартиране, се зарежда заглавния екран, показан на фиг.1 Потребителят по лесен интуитивен начин може да създаде свой собствен граф или да зареди данни за граф от файл, с цел да го визуализира и при желание, да го допълни или коригира. Във всеки момент всичко може да се съхрани отново във файл.

При работа с приложението се използва единствено мишката. Възможности:

* маркиране на връх – чрез кликване върху връх с ляв бутон на мишката. В резутат, той се маркира със син контур (фиг. 1).

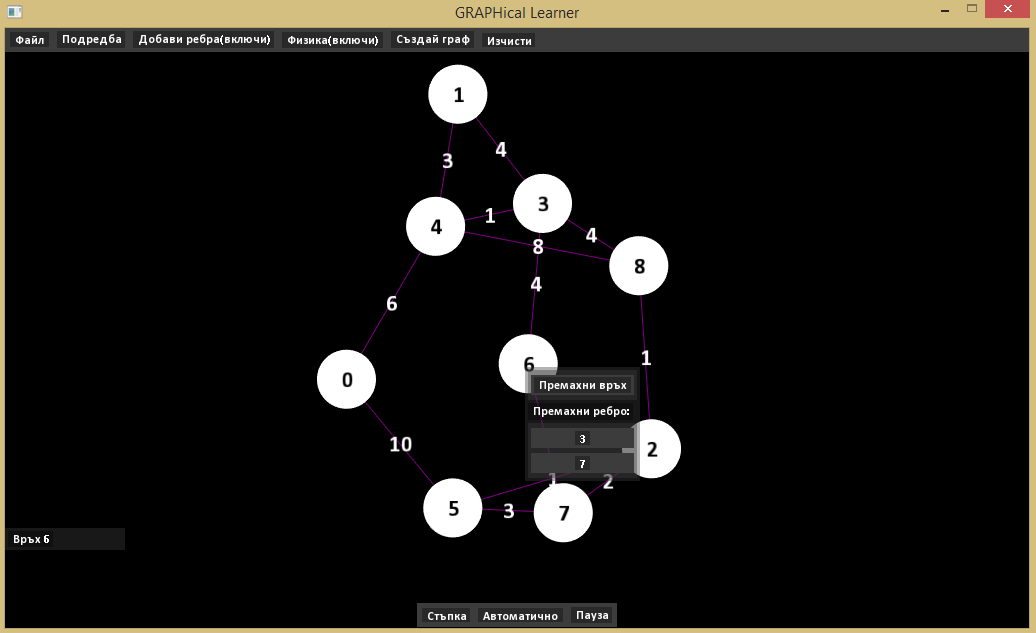


фиг 1. Марикиран връх

● добавяне, демаркиране или изтриване на връх - кликва се с десен бутон върху върха или на празно пространство от екрана и се избира опция от падащо меню.

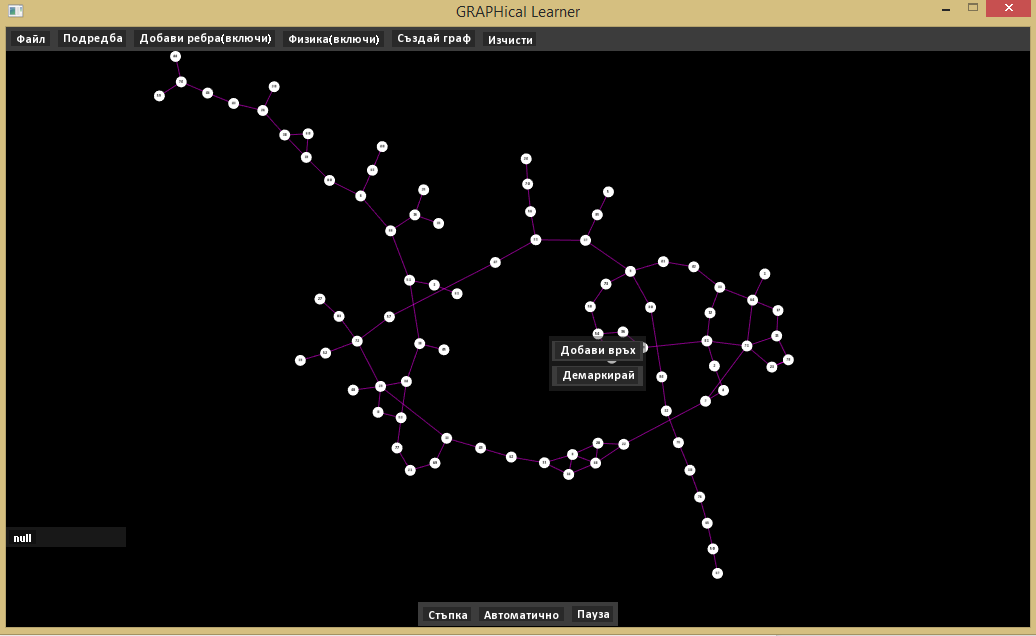
● добавяне на ребро между два върха - първо се кликва с ляв бутон върху единия връх, след това - върху другия, след което двата върха се свързват.

● премахване на ребро между два върха - кликва се с десен бутон върху единия връх или на празно пространство от екрана. В резултат се появява падащо меню за избор на операцията и списък на върховете, до които има ребро от текущия връх. Кликва се върху номера на реброто, което трябва да се изтрие (фиг. 2)



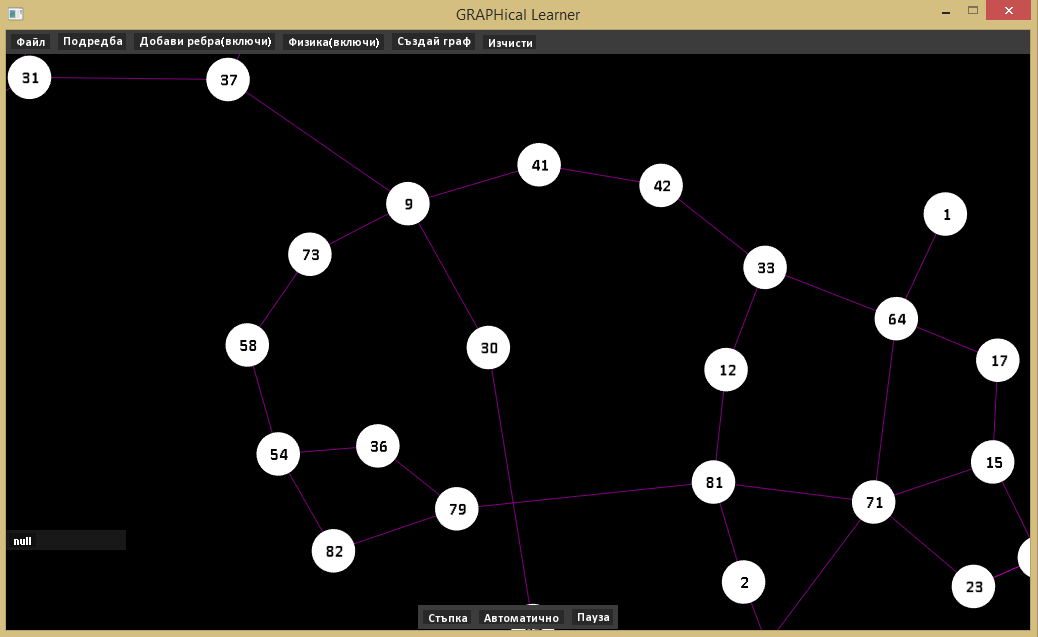
фиг. 2. Корекция на връх или ребро

● демаркиране на маркиран връх – кликва се с десен бутон върху празно пространство и се избира опцията от падащото меню(фиг. 3).



фиг. 3. Падащо меню

* промяна на гледната точка на граф(преместване на позицията на графа) - кликва се на празно място, задържа се левият бутон на мишката и се мести
* промяна на мащаб - чрез scroll-бутона на мишката(фиг. 4).



фиг. 4. Промяна на мащаб и позиция

* генериране на случайна схема на граф – опция “Разбъркай” от главното меню. Размества по случаен начин позициите на върховете на графа
* чертане на генериран по случаен принцип граф – чрез опцията “Създай граф” от главното меню.
* активиране и деактивиране на алгоритъма за подреждане – чрез бутон “Физика(включи/изключи)”. При включване, алгоритъмът подрежда графа, след което сам се изключва.
* преместването на върхове е изключително лесно – кликва се върху връх с ляв бутон на мишката, задържа се натиснат и се мести върха. При това двйствие, заедно с върха се преместват всички, свързани с него ребра, без да се променя схемата на свързване.
* граф може и да се зареди от файл(меню Файл), а впоследствие да се допълни, промени или да се подреди ръчно или чрез естетическия алгоритъм

За да се работи в режим проследяване на алгоритъм, написан на .NET, е необходимо потребителят програмист да добави като Reference GRAPHical\_Learner.dll. След това трябва да направи малки промени по кода си, за да може тестваният алгоритъм да се визуализира. Пример за това е даден в AlgoTest - обясненията по този пример следват.

Първото нещо, което програмистът трябва да направи, е да създаде клас, който наследява Connector от GRAPHical\_Learner, след което трябва да създаде инстанция от този клас. За целта може да се ползва класа с функция Main, както е демонстрирано в AlgoTest:

class Program : Connector

{

static void Main(string[] args)

{

new Program(); // необходима е инстанция на Connector

}

}

След това кода трябва да тръгне от конструктура на класа. Първата функция, която трябва да се извика, е *SetupGui*. Следва въвеждането на графа от клавиатурата. При него е нужно да се извикат две функции:

* *SetVertices*, за да се укаже броя на върхове на графа
* *AddEdge*, за да се добави ребро

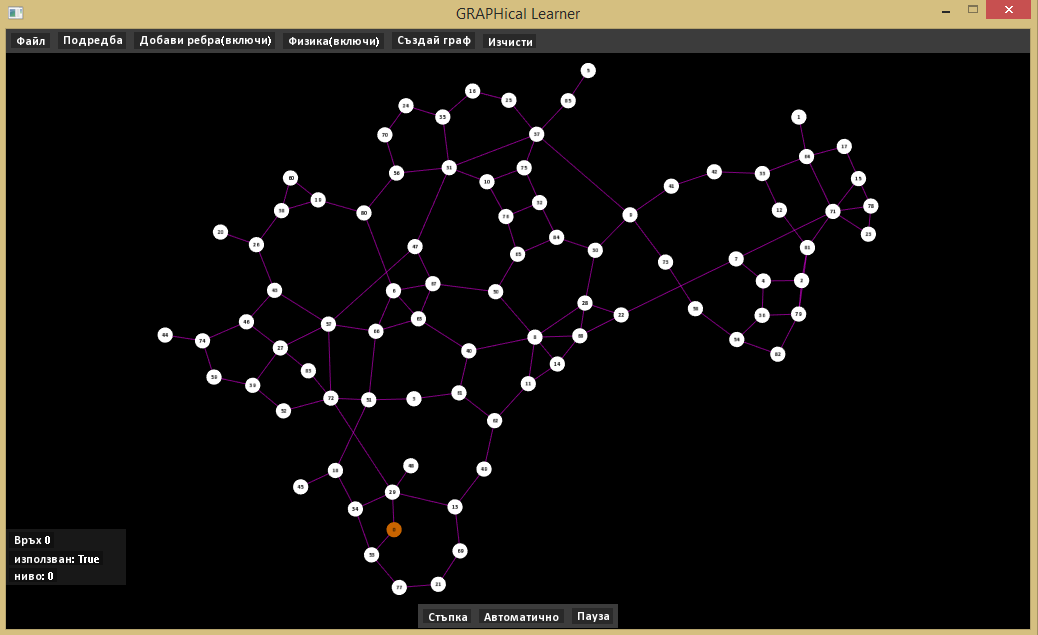
При нужда може да се извика и *SetDirected(bool)*, която указва дали графа е насочен. Следващата задължителна функция е *StartGui*. От своя страна тя извиква функцията *PreRun*, където могат да се направят приготовления, като например регистриране на свойства. Регистрацията се извършва чрез *RegisterProperty*. Функцията приема низ като аргумент и връща int - id(код) на свойството. Трябва да се отбележи, че едно свойство в случая може да приема стойности от различни типове. Това е възможно поради структурата на езика C# и .NET. Възможно е свойството да е обект от потребилски създаден клас. Силно се препоръчва в такъв случай да се override-не метода ToString на обекта, за да може да се покаже полезна информация. Не се препоръчва обаче смяна на типа на свойството за специалните свойства - обозначаващи маркираност(bool), цвят и т.н…. Също така, не е необходимо едно свойство да е общо за всички върхове или ребра. Всеки елемент има независим набор от свойства. С функцията *AddPropertyToVertices/Edges* може да се добави свойство на всички върхове/ребра със начална стойност.

Има два режима на работа на класа Connector, що се отнася до взимане на данни за свойствата: активен и пасивен. При активния начин, на всяка стъпка се извиква функцията *GetVertex/EdgeProperty*. За да работи, тя трабва да бъде override-ната. Аргументите й са id на връх/ребро и id на свойство. Очаква се функцията да връща съответната стойност. Този начин не изисква промени в кода на алгоритъма и е по-удобен, но за сметка на това е по-бавен, което не винаги е проблем.

При пасивният метод програмистът изрично трябва да извика *SetVertex/EdgeProperty*, за да се отрази промяната. Този метод е по-бърз, но изисква повече добавки в кода на алгоритъма, следователно е по-неудобен. Изборът на режим може да се осъществи чрез *SetPollMode(bool)*, като при true режима е активен. Това е и режима по подразбиране.

За да се укаже стъпка в алгоритъма се ползва функцията Pause.

При пуска на графичната среда графът автоматично се подрежда. Алгоритъмът в AlgoTest е стандартен BFS. Като свойства на върхове са зададени “използван” - дали върхът е обходен - и нивото му. Свойствата на върха се виждат в панела в лявата част на екрана:



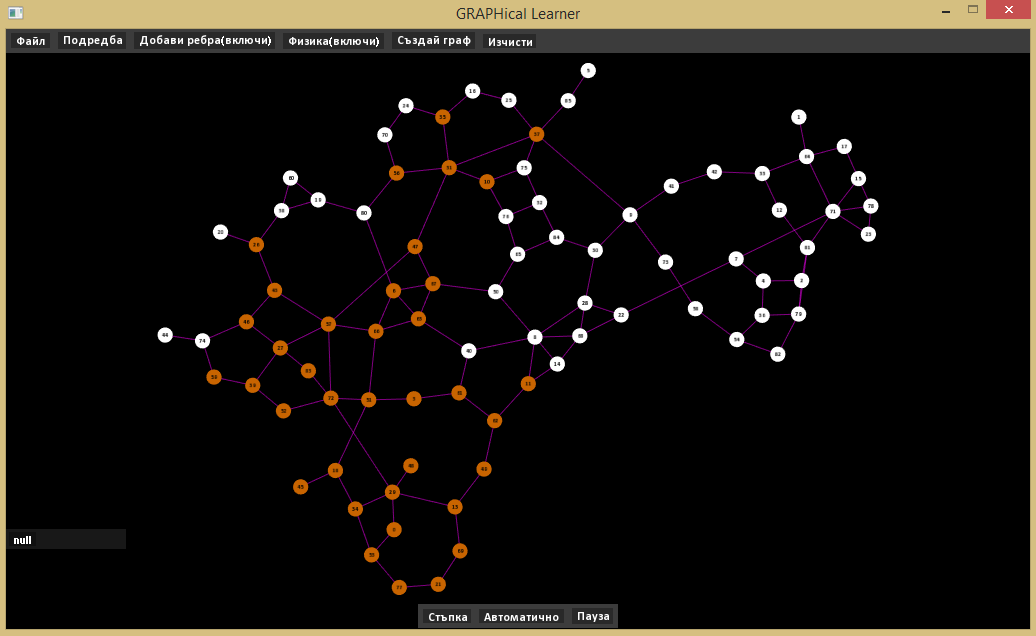
Максималният брой на свойствата е 20, което практически е предостатъчно. Те могат да бъдат абсолютно всичко - маркираност, ниво в BFS/DFS, цена на върха, цена на пътя от някой връх(например при Dijkstra или A\*), капацитет, поток, име и т.н.

За проследяване на алгоритъма се ползва контролния панел, намиращ се в долната част на екрана. Бутонът “Стъпка” изпълнява алгоритъма до следващото извикване на Pause. В случая на AlgoTest - “Стъпка” маркира 1 връх като използван. При “Автоматично” стъпките следват една след друга със скорост, лесна за проследяване. “Пауза” спира автоматичното проиграване.

За да се проследява алгоритъм на C++, стъпките са следните:

1. Включване на хедъра GraphicalLearnerNative.h
2. Извикване на *Initialise()* в началото на *main()*. Ако резултата е false, значи има грешка и не бива да се продължава
3. Въвеждане на графа от клавиатурата. Тук е необходимо да се извикат функциите *SetVertices(int)* и *SetEdges(int)*, за да се зададат съответно броя върхове и ребра. При въвеждане на ребро е необходимо извикване на *AddEdge(int, int)*, за да се добави реброто и към визуалния граф. Насочеността се задава чрез *SetDirected(bool)*.
4. Задаване на свойствата, които ще бъдат използвани. Задаването на свойство става чрез функцията *pm.RegisterProperty(char, char\*, T)*. **Първият** аргумент представлява типа на свойството. В повечето случаи е достатъчно да се укаже дали свойството се отнася за връх(VERTEX\_PROPERTY) или за ребро(EDGE\_PROPERTY). Може и изрично да се укаже типа на свойството: bool(TYPE\_BOOL), int(TYPE\_INT), long(TYPE\_LONG), float(TYPE\_FLOAT), double(TYPE\_DOUBLE), цвят(TYPE\_RGB24), низ(TYPE\_STRING), но това не е задължително - в зависимост от типа на последния аргумент автоматично се добавя типа. Същестувават 2 специални типа(и двата bool) - TYPE\_MARKED и TYPE\_VISIBLE - съответно показват дали елемента е маркиран(маркираните елементи си сменят цвета) или видим(ако е невидим, елемента ще спре да бъде показван). Пример - при много алгоритми върховете биват обходени и необходени. За това е удачно да се ползва TYPE\_MARKED. <Да напиша за свойство цена на ребро, когато го напиша, и за цвят> Пример за комбинация: VERTEX\_PROPERTY | TYPE\_MARKED. **Вторият** аргумент е името на свойството като C string - приема се директно да се напише низа във извикването на функцията. **Третият** аргумент е първоначалната стойност на свойството. Функцията връща char, в който се пази id на създаденото свойство. Пример за ползване: *char idUsed = pm.RegisterProperty(VERTEX\_PROPERTY | TYPE\_MARKED, "used", false)*. Това създава свойство използваност за върхове.
5. Извикване на *StartGui()*. Това ще стартира визуалната среда
6. Следва изпълнението на алгоритъма. Има 2 налични функции за комуникация с визуалната среда: *pm.SetProperty(int, char, T)* и *Pause()*. Първата приема id на елемент(връх или ребро), id на свойство(получено от *RegisterProperty*) и стойност. Типа на стойността трябва да съвпада с този при регистрирането. Програмата автоматично засича дали елемнтът е връх или ребро в зависимост от id на свойството. Примерно използване: *pm.SetProperty(cand, idUsed, true)*. Това указва, че върхът *cand* е използван - резултата във визуалната среда е, че се оцветява. Функцията *Pause()* се използва за паузиране на алгоритъма. Когато това се случи, натрупаните промени се пращат към визуалната среда. Тя продължава изпълнението на алгоритъма.
7. <Да напиша за изключването>

Пример за свързване на C++ е даден във NativeGraph.cpp.



Приложението не изисква специална поддръжка.

**4.7. Заключение** */какъв е основният резултат, дали има приложения до момента, какви възможности съществуват за развитие и усъвършенстване/*

В резултат е създаден продукт с лесен за научаване потребителски и програмен интерфейс, който вече намира приложение. Тестван е в школата по програмиране при Математическа гимназия - гр. Варна.

Използването на приложението не само улеснява потребителите и спестява времеемките процедури по съставяне на граф по дадени числови данни и създаване на тестове по схемата му, но и предотвратява риска от допускане на грешки при ръчно извличане на информация за съставяне на тестове.

Спестява се и време при тестване на алгоритъм - чрез този проект стъпките на алгоритмите за графи са ясно видими. Това значително улеснява тестването, а също така и обясняването на алгоритми. Проследяването на алгоритъм е налично за два езика – C# и C++. Вторият се изучава в училище и е най-масовият език за състезателно програмиране, което значително увеличава достъпността на проекта. Графите са в учебното съдържание по информатика, което означава, че проектът може да бъде директно интегриран в класните стаи.

Перспективата е проектът да се разшири с добавяне на допълнителни функции за дебъгване и връзка с други езици за програмиране, а също така и за допълнително улесняване на потребителите чрез промени в графичния и програмния интерфейс.