Graphs

Author	Uladzislau Volchyk
Created	@Dec 12, 2020 11:02 AM
Tags	Research Special Event

Лабораторная работа 1

Представление графов в памяти ЭВМ. (6 час.)

Разработать класс «Граф», для работы с графом, представленным

- матрицей смежности,
- матрицей инцидентности,
- списками смежности,
- списками дуг.

Класс «Граф» должен содержать следующие методы:

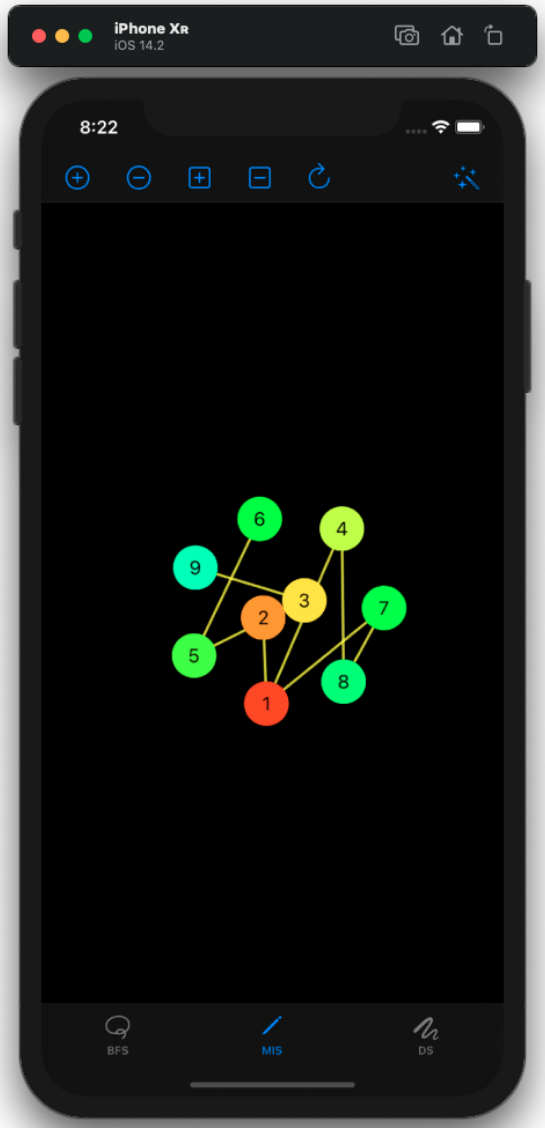
- конструктор/ деструктор;
- конструктор копирования;
- добавление/удаление вершины;
- добавление/удаление дуги;
- печать;

Код реализации:

```
import Foundation

public struct Graph: Codable {
  public struct Edge: Codable {
    public var nodes: Set<Int>
    public var weight: Float

    public init(nodes: Set<Int>, weight: Float) {
      self.nodes = nodes
      self.weight = weight
    }
  }

  public var nodes: Set<Int>
  public var edges: Set<Edge>

  public init<NodeSequence: Sequence, EdgeSequence: Sequence>(nodes: NodeSequence, edges: EdgeSequence) where NodeSequence.Elem
    self.nodes = Set(nodes)
    self.edges = Set(edges)
  }
```

```
}

extension Graph.Edge: Hashable {}

extension Graph: Hashable {}

extension Graph {
    func copy() -> Graph {
        return Graph(nodes: nodes, edges: edges)
    }

    mutating func addNode(_ node: Int) {
        nodes.insert(node)
    }

    mutating func removeNode(_ node: Int) {
        nodes.remove(node)
    }

    mutating func addEdge(_ edge: Edge) {
        edges.insert(edge)
    }

    mutating func removeEdge(_ edge: Edge) {
        edges.remove(edge)
    }

    func printAdjacencyMatrix() {
            var matrix = Array<Array<Int>>()
            nodes.sorted().forEach { (v1) in
                var row = Array<Int>()
                nodes.sorted().forEach { (v2) in
                    if (!edges.filter({ $0.nodes.sorted() == [v1,v2] }).isEmpty) {
                        row.append(1)
                    } else {
                        row.append(0)
                    }
                }
                matrix.append(row)
            }
            matrix.forEach({ print($0) })
        }

        func printIncidenceMatrix() {
            var matrix = Array<Array<Int>>()
            edges.sorted { (e1, e2) -> Bool in
                e1.nodes.sorted()[0] < e2.nodes.sorted()[1]
            }.forEach { (edge) in
                var row = Array<Int>()
                nodes.sorted().forEach { (vertex) in
                    if (edge.nodes.contains(vertex)) {
                        row.append(1)
                    } else {
                        row.append(0)
                    }
                }
                matrix.append(row)
            }
            matrix.forEach({ print($0) })
        }

        func printArcs() {
            edges.forEach { (edge) in
                let nodes = edge.nodes.sorted()
                print("\(nodes[0]) -> \(nodes[1])")
            }
        }

        func printAdjacencyLists() {
            var list = [Int:Array<Int>]()
            nodes.sorted().forEach { (vertex) in
                list[vertex] = edges
                    .filter{( $0.nodes.contains(vertex) )}
                    .map({ $0.nodes.subtracting([vertex]).first! })
            }
            list.forEach { (key, value) in
                print("\(key) -> \(value)")
            }
        }
}
```

Результат:



Вид исходного графа

```
[0, 1, 0, 1, 0, 0, 1, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Матрица смежности

```
[1, 0, 0, 0, 0, 0, 1, 0, 0]
[1, 1, 0, 0, 0, 0, 0, 0, 0]
[0, 1, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 0, 1, 0, 0, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0, 1]
[1, 0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 1, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 1, 1, 0]
[0, 0, 0, 1, 0, 0, 0, 1, 0]
```

Матрица инцидентности

```
5 -> 6
3 -> 5
1 -> 2
4 -> 8
1 -> 7
7 -> 8
2 -> 3
3 -> 9
1 -> 4
```

Списки дуг

```
2 -> [1, 3]
4 -> [8, 1]
9 -> [3]
5 -> [3, 6]
7 -> [8, 1]
8 -> [4, 7]
3 -> [5, 2, 9]
1 -> [4, 2, 7]
6 -> [5]
```

Списки смежности

Лабораторная работа 2

Обходы графов в глубину и в ширину (4 час.)

Расширить класс «Граф» (Л.Р. 1) методом, реализующим следующий алгоритм:

2. Алгоритм обхода графа в ширину с использованием внутренней очереди.

Алгоритм обхода графа в ширину с использованием внутренней очереди

Объекты:

- Ориентированный граф G и некоторая вершина q, не принадлежащая графу.
- Вершины графа могут находиться в одном из двух состояний: “ непомечена”, “ помечена”.
- Вершине q и каждой вершине графа G приписан атрибут СЛЕД, значением которого может быть имя любой вершины графа.

Операция:

- ПРОЙТИ_ВЕРШИНУ(q) переводит вершину q из состояния “непомечена” в состояние “помечена”.
- НЕПОМЕЧЕНА(q) - выдает значение истина, если q находится в состоянии “непомечена”.

Дано:

- Каждая вершина графа G находится в состоянии “ непомечена”, а значением q.СЛЕД является имя некоторой вершины p0 графа G.

Требуется:

- Перевести в состояние “ помечена” все вершины графа G, достижимые из p0, выполняя над ними операцию ПРОЙТИ_ВЕРШИНУ в порядке первого попадания в них при обходе графа G в ширину, начиная с p0.

Алгоритм: ОБХОД_В_ШИРИНУ1

p=q.СЛЕД;

ПРОЙТИ_ВЕРШИНУ(p);

пока p<>q ***цикл***

q=q.СЛЕД

для всех r из множества смежных вершин с q ***цикл***

если НЕПОМЕЧЕНА (r)

то ПРОЙТИ_ВЕРШИНУ(r); p.СЛЕД=r; p=r;

все

все

p.СЛЕД=ничто

все

Пояснение:

- В процедуре с помощью вершины q и атрибута СЛЕД реализуется S в режиме очереди: если p=q, то очередь S пуста; при p<>q очередь состоит из элементов q.СЛЕД, (q.СЛЕД).СЛЕД,..., p0. Оператор (q=q.СЛЕД) исключает элемент из очереди, а оператор p.СЛЕД=r; p=r; добавляет новый элемент в очередь.

Код реализации алгоритма:

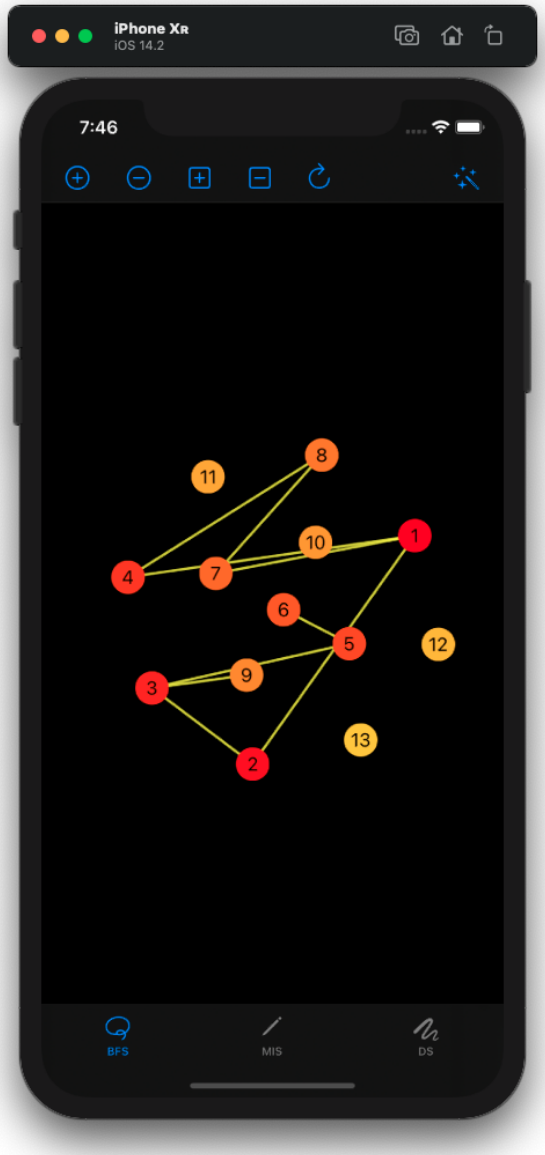
```
struct BFS {
    var markCompletion: ((Int) -> ())?

    private var graph: Graph
    private var nodes: [Int:Bool]

    var pq = Queue<Int>()

    init(_ graph: Graph) {
        self.graph = graph
        nodes = graph.nodes
            .reduce([Int:Bool](), { (dict, node) -> [Int:Bool] in
                var dict = dict
                dict[node] = false
                return dict
            })
    }

    mutating func settle(_ node: Int) {
        markCompletion?(node)
        nodes[node] = true
    }

    func isSettled(_ node: Int) -> Bool { nodes[node] ?? false }

    mutating func bfs(start: Int, goal: Int, completion: (Bool) -> ()) {
        pq.enqueue(start)
        settle(start)
        while(!pq.isEmpty) {
            if let q = pq.dequeue() {
                graph.edges
                    .filter { edge -> Bool in edge.nodes.sorted()[0] == q }
                    .map { edge -> Int in edge.nodes.sorted()[1] }
```

```
                    .forEach { vertex in
                        if (vertex == goal) {
                            settle(goal)
                            completion(true)
                            return
                        }
                        if (!isSettled(vertex)) {
                            pq.enqueue(vertex)
                            settle(vertex)
                        }
                    }

            }
        }
        completion(false)
    }
}
```

Примечание:

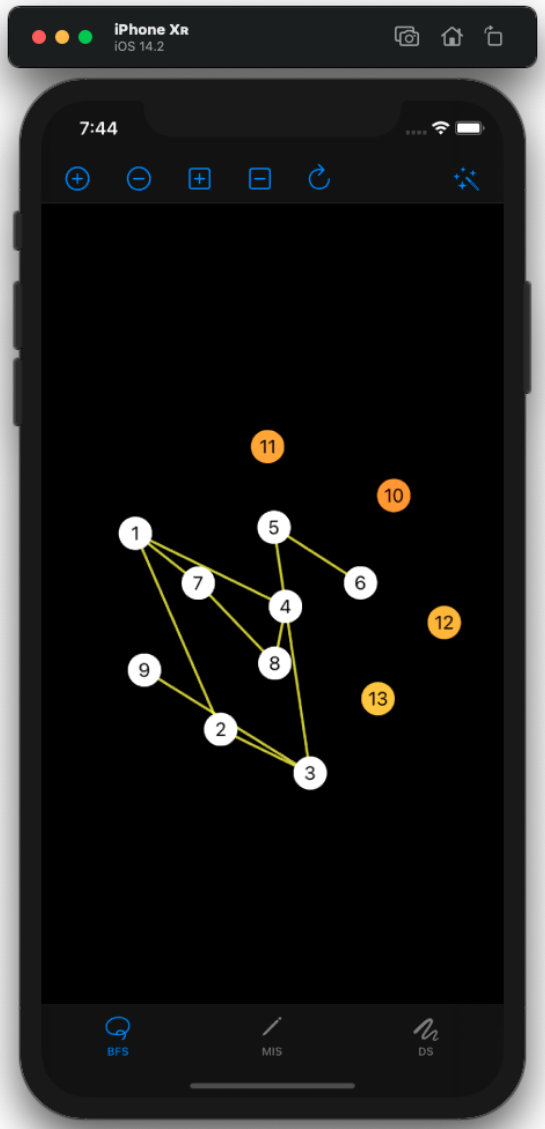
- Направления дуг соответствуют направлениям от меньших вершин к большим

Результат:



Вид графа до поиска в ширину

Задание вершин (1 - начало, 10 - конец)



Результат поиска в ширину

Лабораторная работа 3

Основные задачи теории графов, алгоритмы их решения и области их приложения (7 час.)

Расширить класс «Граф» (лабораторная работа 1,2) методом, решающим следующую задачу теории графов:

2. Независимые и доминирующие множества.

Алгоритм нахождения максимального независимого множества:

1. Initialize I to an empty set.
2. While V is not empty:
   - Choose a node v∈V;
   - Add v to the set I;
   - Remove from V the node v and all its neighbours.

3. Return I.

Реализация алгоритма:

```
func mis(_ completion: (Set<Int>) -> ()) {
        var i = Set<Int>()
        var unsettled = Set(graph.nodes)
        while (!unsettled.isEmpty) {
            let vertex = unsettled.first!
            i.insert(vertex)
            unsettled.subtract([vertex])
            unsettled = unsettled.filter({ (candidate) -> Bool in
                graph.edges
                    .filter({ $0.nodes.intersection([candidate, vertex]).count == 2 })
                    .isEmpty
            })
        }
        completion(i)
    }
```

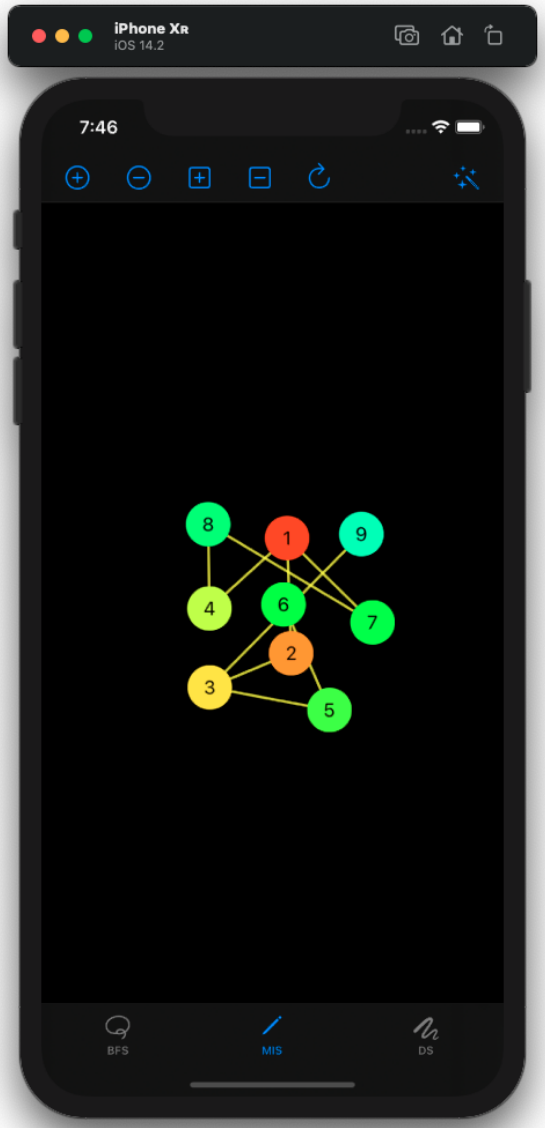
Алгоритм нахождения минимального доминирующего множества:

1. First we have to initialize a set ‘S’ as empty
2. Take any edge ‘e’ of the graph connecting the vertices (say A and B)
3. Add one vertex between A and B (let say A) to our set S
4. Delete all the edges in the graph connected to A
5. Go back to step 2 and repeat, if some edge is still left in the graph
6. The final set S is a Dominant Set of the graph

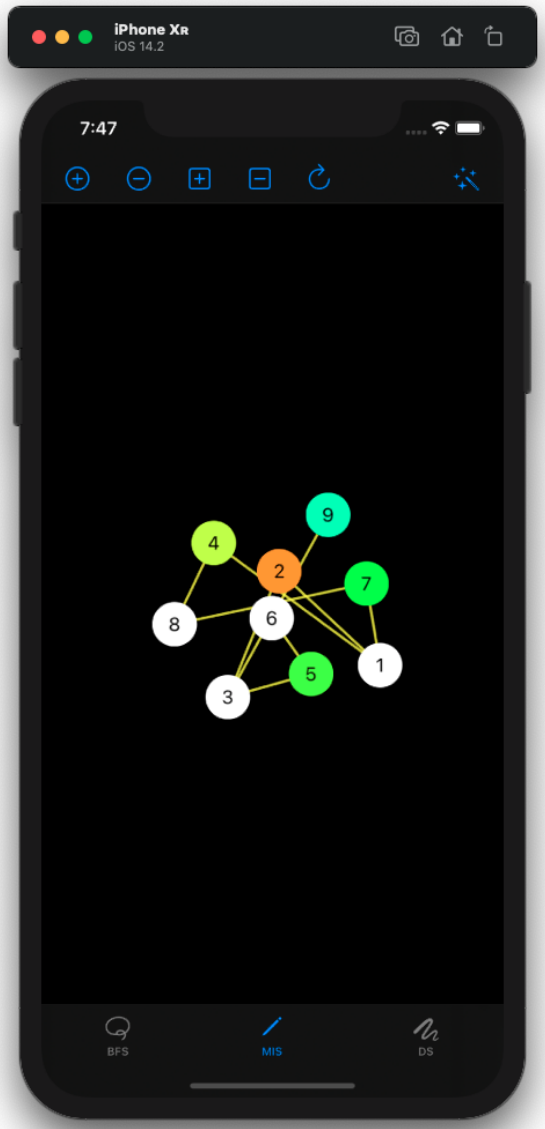
Реализация алгоритма:

```
func ds(_ completion: (Set<Int>) -> ()) {
        var s = Set<Int>()
        var edgesCopy = Set(graph.edges)
        while (!edgesCopy.isEmpty) {
            let nodes = edgesCopy.first!.nodes.sorted()
            s.insert(nodes[0])
            edgesCopy = edgesCopy.filter({ (edge) -> Bool in
                !edge.nodes.contains(nodes[1])
            })
        }
        completion(s)
    }
```

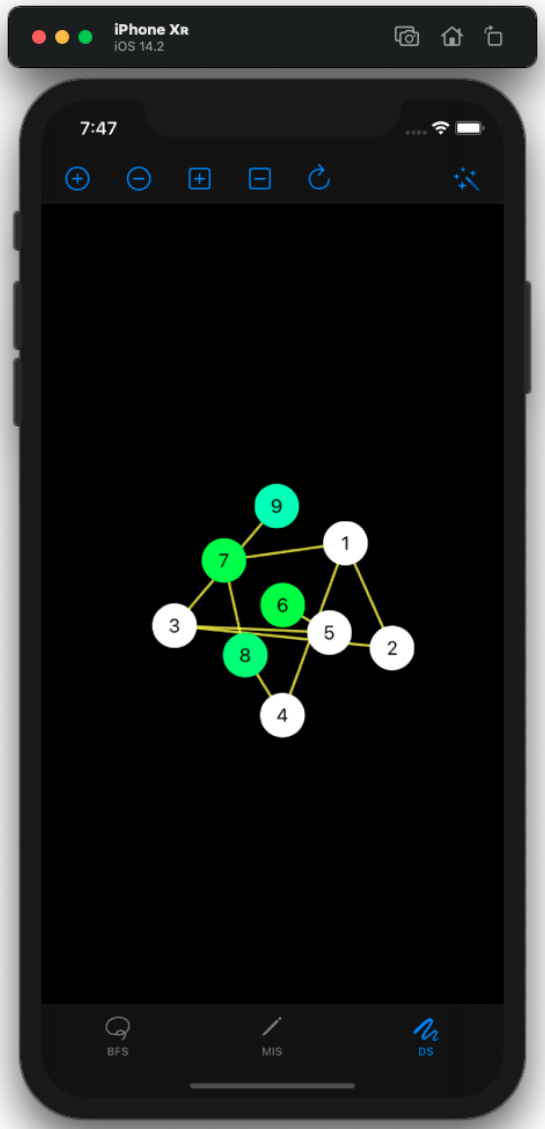
Результат:



Исходный вид графа



Результат поиска максимального независимого подграфа (белые вершины)



Результат поиска минимального доминирующего подграфа (белые вершины)

trotnic - Overview

Dismiss Sign up for your own profile on GitHub, the best place to host code, manage projects, and build software alongside 50 million developers. Sign up I'm an iOS developer. Inspired by people, who change the world with their ideas, i try everyday to learn something new.

https://github.com/trotnic