МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования

«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»

ОТЧЕТ ПО ТЕОРИИ ГРАФОВ

Отчёт о практике

студента 3 курса 351 группы
направления 09.03.04 — Программная инженерия
факультета КНиИТ
Толстова Роберта Сергеевича

Саратов 2025

СОДЕРЖАНИЕ

1 Минимальные требования для класса Граф

1.1 Условие

Для решения всех задач курса необходимо создать класс (или иерархию классов - на усмотрение разработчика), содержащий:

1. Структуру для хранения списка смежности графа (не работать с графом через матрицы смежности, если в некоторых алгоритмах удобнее использовать список ребер - реализовать метод, создающий список рёбер на основе списка смежности);
2. Конструкторы (не менее 3-х):
   - конструктор по умолчанию, создающий пустой граф
   - конструктор, заполняющий данные графа из файла
   - конструктор-копию (аккуратно, не все сразу делают именно копию)
   - специфические конструкторы для удобства тестирования
3. Методы:
   - добавляющие вершину,
   - добавляющие ребро (дугу),
   - удаляющие вершину,
   - удаляющие ребро (дугу),
   - выводящие список смежности в файл (в том числе в пригодном для чтения конструктором формате).
   - Не выполняйте некорректные операции, сообщайте об ошибках.
4. Должны поддерживаться как ориентированные, так и неориентированные графы. Заранее предусмотрите возможность добавления меток иили весов для дуг. Поддержка мультиграфа не требуется.
5. Добавьте минималистичный консольный интерфейс пользователя (не смешивая его с реализацией!), позволяющий добавлять и удалять вершины и рёбра (дуги) и просматривать текущий список смежности графа.
6. Сгенерируйте не менее 4 входных файлов с разными типами графов (балансируйте на комбинации ориентированность-взвешенность) для тестирования класса в этом и последующих заданиях. Графы должны содержать не менее 7-10 вершин, в том числе петли и изолированные вершины.

   Замечание:

В зависимости от выбранного способа хранения графа могут появиться дополнительные трудности при удалении-добавлении, например, необходимость переименования вершин, если граф хранится списком (например, vector C++, List C). Этого можно избежать, если хранить в списке пару (имя вершины, список смежных вершин), или хранить в другой структуре (например, Dictionary C, map в C++, при этом список смежности вершины может также храниться в виде словаря с ключами - смежными вершинами и значениями - весами соответствующих ребер). Идеально, если в качестве вершины реализуется обобщенный тип (generic), но достаточно использовать строковый тип или свой класс.

1.2 Код

Описание графа многофайловое. Полный код проекта приложен в приложении.

Содержимое файла

1 graph/types.go

```
/*
 * This is a graph package, which contains graoh definition and basic
operations
 * on it. As you go through the file, you will see some comments, that
are
 * explaining this or that choice, etc.
 *
 * Author: github.com/tolstovrob
 */

package graph

type Option[T any] func(*T) // Type representing functional options
pattern

type TKey uint64   // Key type. Can be replaced with any UNIQUE type
type TWeight int64 // Weight type. Can be replaced with any COMPARABLE
type
```

Содержимое файла

1 graph/node.go

```go
/*
 * This is a graph package, which contains graoh definition and basic
operations
 * on it. As you go through the file, you will see some comments, that
are
 * explaining this or that choice, etc.
 *
 * Author: github.com/tolstovrob
 */

package graph

/*
 * Node struct represents graph node. It has a unique key and optional
label.
 * You can properly construct Node via this:
 *
 * node := MakeNode(1)
 *
 * or this:
 *
 * labeledNode := MakeNode(1, WithNodeLabel("Aboba"))
 */

type Node struct {
  Key   TKey   `json:"key"`
  Label string `json:"label"`
}

func MakeNode(key TKey, options ...Option[Node]) *Node {
  node := &Node{}
  node.Key = key
  for _, opt := range options {
    opt(node)
  }
  return node
}

func (node *Node) UpdateNode(options ...Option[Node]) {
  for _, opt := range options {
```

```
    opt(node)
  }
}

func WithNodeLabel(label string) Option[Node] {
  return func(node *Node) {
    node.Label = label
  }
}
```

Содержимое файла

1 graph/edge.go

```
/*
 * This is a graph package, which contains graoh definition and basic
operations
 * on it. As you go through the file, you will see some comments, that
are
 * explaining this or that choice, etc.
 *
 * Author: github.com/tolstovrob
 */

package graph

/*
 * Edge struct represents graph edge -- a connection between 2 nodes.
 * This edge implementation supposed to be direct. If you need to make
 * undirected edge, you should make 2 edges and work with this.
 *
 * Edge represents connection from Edge.Source to Edge.Destination,
optionally
 * labelled and weighted.
 *
 * I.e., there are 2 nodes given:
 *
 * src := MakeNode(1)
 * dst := MakeNode(2)
 *
 * You can properly construct this via this:
 *
```

```
 * edge := MakeEdge(1, src.Key, dst.Key)
 *
 * or with optional fields:
 *
 * fullyConstructedEdge := MakeEdge(1, src.Key, dst.Key,
WithEdgeLabel("Path"), WithEdgeWeight(69))
 */

type Edge struct {
  Key         TKey    `json:"key"`
  Source      TKey    `json:"source"`
  Destination TKey    `json:"destination"`
  Weight      TWeight `json:"weight"`
  Label       string  `json:"label"`
}

func MakeEdge(key, src, dst TKey, options ...Option[Edge]) *Edge {
  edge := &Edge{}
  edge.Key, edge.Source, edge.Destination = key, src, dst
  for _, opt := range options {
    opt(edge)
  }
  return edge
}

func (edge *Edge) UpdateEdge(options ...Option[Edge]) {
  for _, opt := range options {
    opt(edge)
  }
}

func WithEdgeWeight(weight TWeight) Option[Edge] {
  return func(edge *Edge) {
    edge.Weight = weight
  }
}

func WithEdgeLabel(label string) Option[Edge] {
  return func(edge *Edge) {
    edge.Label = label
```

```
  }
}
```

Содержимое файла

1 graph/graph.go

```
/*
 * This is a graph package, which contains graoh definition and basic
operations
 * on it. As you go through the file, you will see some comments, that
are
 * explaining this or that choice, etc.
 *
 * Author: github.com/tolstovrob
 */

package graph

import (
  "encoding/json"
  "fmt"
  "slices"
)

/*
 * Graph struct.
 *
 * First of all, we got TOptions struct, which represents all possible
graph
 * configuration. Now, it only has IsMulti and IsDirected for
multigraphs and
 * Directed graphs respectively, but it is easily scalable for other
options
 * if neccessary.
 *
 * Graph struct sa it is contains Nodes and Edges lists of Node and
Edge
 * pointers respectively, and Options configuration of TOptions.
 *
 * Graph represented via adjacency list of edges. But it also possible
to have
```

```
 * islands with no connections. You cannot find them in Edges, but in
the Nodes.
 *
 * You actually can use default constructor with this one. It will
build
 * non-multi undirected graph:
 *
 * gr := Graph{}
 *
 * But to make graph properly, use constructor with options:
 *
 * gr := MakeGraph(WithGraphMulti(true), WithGraphDirected(false))
 *
 * I.e., code above will create undirected multigraph.
 */

type TOptions struct {
  IsMulti    bool `json:"isMulti"`
  IsDirected bool `json:"IsDirected"`
}

type Graph struct {
  Nodes        map[TKey]*Node  `json:"nodes"`
  Edges        map[TKey]*Edge  `json:"edges"`
  AdjacencyMap map[TKey][]TKey `json:"adjacencyMap"`
  Options      TOptions        `json:"options"`
}

func MakeGraph(options ...Option[Graph]) *Graph {
  gr := &Graph{}
  gr.Nodes = make(map[TKey]*Node)
  gr.Edges = make(map[TKey]*Edge)
  gr.AdjacencyMap = make(map[TKey][]TKey)
  for _, opt := range options {
    opt(gr)
  }
  return gr
}

func (gr *Graph) Copy() *Graph {
```

```
  newGraph := MakeGraph(
    WithGraphDirected(gr.Options.IsDirected),
    WithGraphMulti(gr.Options.IsMulti),
  )

  for key, node := range gr.Nodes {
    newGraph.Nodes[key] = &Node{
      Key:   node.Key,
      Label: node.Label,
    }
  }

  for key, edge := range gr.Edges {
    newGraph.Edges[key] = &Edge{
      Key:         edge.Key,
      Source:      edge.Source,
      Destination: edge.Destination,
      Weight:      edge.Weight,
      Label:       edge.Label,
    }
  }

  newGraph.RebuildAdjacencyMap()
  return newGraph
}

func (gr *Graph) RebuildEdges() {
  newEdges := make(map[TKey]*Edge)
  edgeKeysUsed := make(map[TKey]bool)
  edgeKeyCounter := TKey(1)

  nextEdgeKey := func() TKey {
    for {
      if !edgeKeysUsed[edgeKeyCounter] {
        edgeKeysUsed[edgeKeyCounter] = true
        key := edgeKeyCounter
        edgeKeyCounter++
        return key
      }
      edgeKeyCounter++
```

```
    }
  }

  edgeID := func(src, dst TKey) string {
    if !gr.Options.IsDirected && src > dst {
      src, dst = dst, src
    }
    return fmt.Sprintf("%d-%d", src, dst)
  }

  seenEdges := make(map[string]bool)

  for _, edge := range gr.Edges {
    id := edgeID(edge.Source, edge.Destination)
    if !gr.Options.IsMulti {
      if seenEdges[id] {
        continue
      }
      seenEdges[id] = true
    }

    key := edge.Key
    if key == 0 || edgeKeysUsed[key] {
      key = nextEdgeKey()
    } else {
      edgeKeysUsed[key] = true
    }

    newEdge := &Edge{
      Key:         key,
      Source:      edge.Source,
      Destination: edge.Destination,
      Weight:      edge.Weight,
      Label:       edge.Label,
    }
    newEdges[key] = newEdge
  }

  gr.Edges = newEdges
}
```

```
func (gr *Graph) RebuildAdjacencyMap() {
  gr.AdjacencyMap = make(map[TKey][]TKey)
  for _, edge := range gr.Edges {
    gr.AdjacencyMap[edge.Source] = append(gr.AdjacencyMap[edge.Source],
edge.Destination)
    if !gr.Options.IsDirected {
      gr.AdjacencyMap[edge.Destination] =
append(gr.AdjacencyMap[edge.Destination], edge.Source)
    }
  }
}

/*
 * Later in the code, Graph.RebuildAdjacencyMap will be called many
times.
 * It could really affect performance on huge amount of edges, but
since
 * it is just academical example, we will pretend it never happens.
 *
 * Anyways, need to fix, so mark this part as WIP!
 */

func (gr *Graph) UpdateGraph(options ...Option[Graph]) {
  oldOptions := gr.Options

  for _, opt := range options {
    opt(gr)
  }

  if oldOptions != gr.Options {
    gr.RebuildEdges()
    gr.RebuildAdjacencyMap()
  }
}

func WithGraphNodes(nodes map[TKey]*Node) Option[Graph] {
  return func(gr *Graph) {
    gr.Nodes = nodes
  }
```

```
}

func WithGraphEdges(edges map[TKey]*Edge) Option[Graph] {
  return func(gr *Graph) {
    gr.Edges = edges
  }
}
func WithGraphAdjacencyMap(adj map[TKey][]TKey) Option[Graph] {
  return func(gr *Graph) {
    gr.AdjacencyMap = adj
  }
}

func WithGraphOptions(options TOptions) Option[Graph] {
  return func(gr *Graph) {
    gr.Options = options
  }
}

func WithGraphMulti(isMulti bool) Option[Graph] {
  return func(gr *Graph) {
    gr.Options.IsMulti = isMulti
  }
}

func WithGraphDirected(IsDirected bool) Option[Graph] {
  return func(gr *Graph) {
    gr.Options.IsDirected = IsDirected
  }
}

/*
 * Next coming finding, adding and removing handlers for nodes and
edges. I put
 * them apart main Graph struct because they contain both node and
edges and
 * graph. All of them will throw an error if the operation is not
allowed (I.e.
 * adding existing node or connecting nodes with more then one time in
multi).
```

```
 */

func (gr *Graph) GetNodeByKey(key TKey) (*Node, error) {
  if gr.Nodes == nil {
    return nil, ThrowNodesListIsNil()
  }

  if _, exists := gr.Nodes[key]; !exists {
    return nil, ThrowNodeWithKeyNotExists(key)
  }

  return gr.Nodes[key], nil
}

func (gr *Graph) AddNode(node *Node) error {
  if node, _ := gr.GetNodeByKey(node.Key); node != nil {
    return ThrowNodeWithKeyExists(node.Key)
  }

  gr.Nodes[node.Key] = node
  return nil
}

func (gr *Graph) RemoveNodeByKey(key TKey) error {
  if _, err := gr.GetNodeByKey(key); err != nil {
    return err
  }

  // Remove all edges connected to this node
  edgesToRemove := make([]TKey, 0)
  for edgeKey, edge := range gr.Edges {
    if edge.Source == key || edge.Destination == key {
      edgesToRemove = append(edgesToRemove, edgeKey)
    }
  }

  for _, edgeKey := range edgesToRemove {
    delete(gr.Edges, edgeKey)
  }
```

```
  // Remove the node
  delete(gr.Nodes, key)
  delete(gr.AdjacencyMap, key)

  // Remove references to this node from other nodes' adjacency lists
  for nodeKey, neighbors := range gr.AdjacencyMap {
    filteredNeighbors := make([]TKey, 0)
    for _, neighbor := range neighbors {
      if neighbor != key {
        filteredNeighbors = append(filteredNeighbors, neighbor)
      }
    }
    gr.AdjacencyMap[nodeKey] = filteredNeighbors
  }

  return nil
}

func (gr *Graph) GetEdgeByKey(key TKey) (*Edge, error) {
  if gr.Edges == nil {
    return nil, ThrowEdgesListIsNil()
  }

  if _, exists := gr.Edges[key]; !exists {
    return nil, ThrowEdgeWithKeyNotExists(key)
  }

  return gr.Edges[key], nil
}

func (gr *Graph) AddEdge(edge *Edge) error {
  if edge, _ := gr.GetEdgeByKey(edge.Key); edge != nil {
    return ThrowEdgeWithKeyExists(edge.Key)
  }

  if !gr.Options.IsMulti &&
    (slices.Contains(gr.AdjacencyMap[edge.Source], edge.Destination) ||
      !gr.Options.IsDirected &&
slices.Contains(gr.AdjacencyMap[edge.Destination], edge.Source)) {
    return ThrowSameEdgeNotAllowed(edge.Source, edge.Destination)
```

```
  }

  if src, _ := gr.GetNodeByKey(edge.Source); src == nil {
    return ThrowEdgeEndNotExists(edge.Key, edge.Source)
  }

  if dst, _ := gr.GetNodeByKey(edge.Destination); dst == nil {
    return ThrowEdgeEndNotExists(edge.Key, edge.Destination)
  }

  gr.Edges[edge.Key] = edge
  gr.RebuildAdjacencyMap()
  return nil
}

func (gr *Graph) RemoveEdgeByKey(key TKey) error {
  if edge, _ := gr.GetEdgeByKey(key); edge == nil {
    return ThrowEdgeWithKeyNotExists(key)
  }

  delete(gr.Edges, key)
  gr.RebuildAdjacencyMap()
  return nil
}

/*
 * File handling moved to CLI service -- here will be declared just
marshalling
 * and unmarshalling handlers
 */

func (gr *Graph) MarshalJSON() ([]byte, error) {
  type MarshalGraph Graph
  return json.Marshal(&struct {
    *MarshalGraph
  }{
    MarshalGraph: (*MarshalGraph)(gr),
  })
}
```

```go
func (gr *Graph) UnmarshalJSON(data []byte) error {
  type MarshalGraph Graph
  aux := &struct {
    *MarshalGraph
  }{
    MarshalGraph: (*MarshalGraph)(gr),
  }
  if err := json.Unmarshal(data, &aux); err != nil {
    return ThrowGraphUnmarshalError()
  }
  gr.RebuildAdjacencyMap()
  return nil
}

func (gr *Graph) ToJSON() (string, error) {
  b, err := json.Marshal(gr)
  if err != nil {
    return "", err
  }
  return string(b), nil
}

func (gr *Graph) FromJSON(jsonData string) error {
  return json.Unmarshal([]byte(jsonData), gr)
}

/*
 * Following methods are for checking some graph properties.
 * They are can be useful for some tasks
 */

func (gr *Graph) IsTree() bool {
  if len(gr.Nodes) == 0 {
    return true
  }

  // Check edge count: tree must have n-1 edges
  if len(gr.Edges) != len(gr.Nodes)-1 {
    return false
  }
```

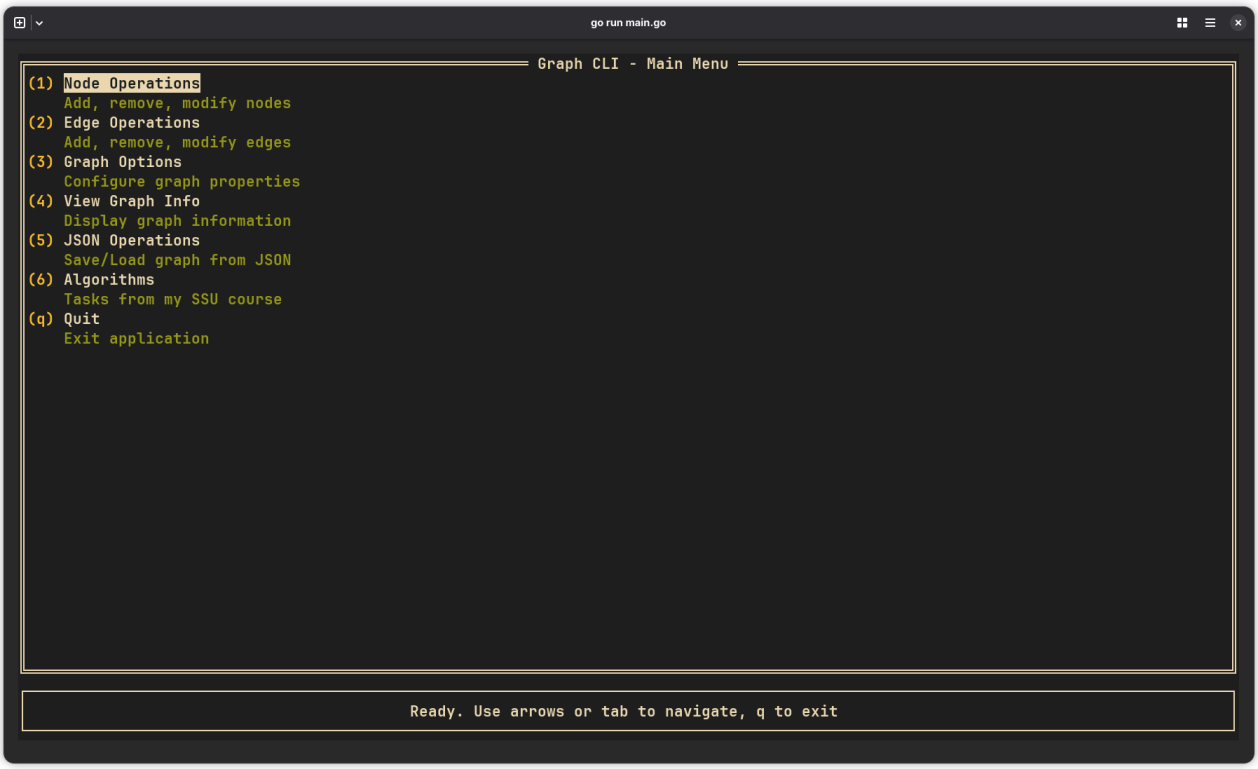
```
  // Check connectivity and acyclicity
  return gr.IsConnected() && !gr.HasCycle()
}

func (gr *Graph) IsConnected() bool {
  if len(gr.Nodes) == 0 {
    return true
  }

  visited := make(map[TKey]bool)

  // Start from first node
  var startKey TKey
  for key := range gr.Nodes {
    startKey = key
    break
  }

  // BFS traversal
  queue := []TKey{startKey}
  visited[startKey] = true

  for len(queue) > 0 {
    current := queue[0]
    queue = queue[1:]

    for _, neighbor := range gr.AdjacencyMap[current] {
      if !visited[neighbor] {
        visited[neighbor] = true
        queue = append(queue, neighbor)
      }
    }
  }

  return len(visited) == len(gr.Nodes)
}

func (gr *Graph) HasCycle() bool {
  if len(gr.Nodes) == 0 {
```

```
    return false
  }

  visited := make(map[TKey]bool)

  for node := range gr.Nodes {
    if !visited[node] {
      if gr.hasCycleDFS(node, 0, visited) {
        return true
      }
    }
  }

  return false
}

func (gr *Graph) hasCycleDFS(current, parent TKey, visited
map[TKey]bool) bool {
  visited[current] = true

  for _, neighbor := range gr.AdjacencyMap[current] {
    if !visited[neighbor] {
      if gr.hasCycleDFS(neighbor, current, visited) {
        return true
      }
    } else if neighbor != parent {
      return true
    }
  }

  return false
}

// GetConnectedComponents returns the number of connected components in
the graph
func (gr *Graph) GetConnectedComponents() int {
  if len(gr.Nodes) == 0 {
    return 0
  }
```

```
  visited := make(map[TKey]bool)
  componentCount := 0

  for node := range gr.Nodes {
    if !visited[node] {
      componentCount++
      gr.bfsComponent(node, visited)
    }
  }

  return componentCount
}

// bfsComponent performs BFS to mark all nodes in the same connected
component
func (gr *Graph) bfsComponent(start TKey, visited map[TKey]bool) {
  queue := []TKey{start}
  visited[start] = true

  for len(queue) > 0 {
    current := queue[0]
    queue = queue[1:]

    for _, neighbor := range gr.AdjacencyMap[current] {
      if !visited[neighbor] {
        visited[neighbor] = true
        queue = append(queue, neighbor)
      }
    }
  }
}

// GetComponentSizes returns the sizes of all connected components
func (gr *Graph) GetComponentSizes() []int {
  if len(gr.Nodes) == 0 {
    return []int{}
  }

  visited := make(map[TKey]bool)
  var sizes []int
```

```
  for node := range gr.Nodes {
    if !visited[node] {
      size := gr.bfsComponentWithSize(node, visited)
      sizes = append(sizes, size)
    }
  }

  return sizes
}

// bfsComponentWithSize performs BFS and returns the size of the
component
func (gr *Graph) bfsComponentWithSize(start TKey, visited
map[TKey]bool) int {
  queue := []TKey{start}
  visited[start] = true
  size := 1

  for len(queue) > 0 {
    current := queue[0]
    queue = queue[1:]

    for _, neighbor := range gr.AdjacencyMap[current] {
      if !visited[neighbor] {
        visited[neighbor] = true
        queue = append(queue, neighbor)
        size++
      }
    }
  }

  return size
}
```

1.3 Пример интерфейса в консоли



Реализация TUI CLI находится в пакете cli.

2 Список смежности Ia

2.1 Условие

Вывести те вершины, полустепень захода которых меньше, чем у заданной вершины.

Для данной вершины орграфа вывести все «заходящие» соседние вершины.

2.2 Код

```
/*
 * This package contains algorithms and tasks for my SSU course
 */

package algo

import "github.com/tolstovrob/graph-go/graph"

/*
 * Task 1: Get all nodes, for which degree is greater then half-degree
of entry
 */

func InDegreeLessThan(gr *graph.Graph, targetKey graph.TKey)
[]graph.TKey {
  // Find all entries
  inDegree := make(map[graph.TKey]int)
  for _, edge := range gr.Edges {
    inDegree[edge.Destination]++
    if !gr.Options.IsDirected {
      inDegree[edge.Source]++
    }
  }
  targetInDegree := inDegree[targetKey]

  // Find all nodes satisfies task objective
  var result []graph.TKey
  for nodeKey := range gr.Nodes {
    if inDegree[nodeKey] < targetInDegree {
      result = append(result, nodeKey)
    }
```

```
  }
  return result
}

/*
 * Task 2: For directed graph node output all in-nodes
 */

func InNodesInDirected(gr *graph.Graph, targetKey graph.TKey)
([]graph.TKey, error) {
  // Check if graph is directed
  if !gr.Options.IsDirected {
    return nil, graph.ThrowGraphNotDirected()
  }

  // If graph directed, find all entries
  inNodes := []graph.TKey{}
  for _, edge := range gr.Edges {
    if edge.Destination == targetKey {
      inNodes = append(inNodes, edge.Source)
    }
  }
  return inNodes, nil
}
```

3 Список смежности Iб

3.1 Условие

Построить граф, полученный из исходного удалением висячих вершин.

3.2 Код

```
/*
 * This package contains algorithms and tasks for my SSU course
 */

package algo

import "github.com/tolstovrob/graph-go/graph"

/*
 * Task: Build graph obtained by removing pendant vertices from
original graph
 */

func RemovePendantVertices(gr *graph.Graph) (*graph.Graph, error) {
  if gr.Nodes == nil {
    return nil, graph.ThrowNodesListIsNil()
  }

  newGraph := gr.Copy()

  removed := true
  for removed {
    removed = false
    for key := range newGraph.Nodes {
      degree := len(newGraph.AdjacencyMap[key])
      if degree == 1 {
        newGraph.RemoveNodeByKey(key)
        removed = true
        break // Restart iteration since map changed
      }
    }
  }
```

```
    return newGraph, nil
}
```

4 Список смежности II

4.1 Условие

Выяснить, является ли граф связным.

4.2 Код

```
/*
 * This package contains algorithms and tasks for my SSU course
 */

package algo

import "github.com/tolstovrob/graph-go/graph"

/*
 * Task: Check if there exists a vertex that can be removed to make the
graph a tree
 */

func CanRemoveVertexToMakeTree(gr *graph.Graph) (bool, []graph.TKey,
error) {
  if gr.Nodes == nil {
    return false, nil, graph.ThrowNodesListIsNil()
  }

  var candidates []graph.TKey

  // For each vertex, check if removing it makes the graph a tree
  for key := range gr.Nodes {
    // Create a copy and remove the vertex
    tempGraph := gr.Copy()
    if err := tempGraph.RemoveNodeByKey(key); err != nil {
      continue
    }

    // Check if the resulting graph is a tree
    if tempGraph.IsTree() {
      candidates = append(candidates, key)
    }
  }
```

```
  return len(candidates) > 0, candidates, nil
}
```

Методы для проверки на связность описаны в структуре графа.

5 Список смежности II

5.1 Условие

Проверить, можно ли из графа удалить какую-либо вершину так, чтобы получилось дерево.

5.2 Код

```
/*
 * This package contains algorithms and tasks for my SSU course
 */

package algo

import "github.com/tolstovrob/graph-go/graph"

/*
 * Task: Count connected components and analyze their sizes
 */

type ComponentAnalysis struct {
  TotalComponents   int
  ComponentSizes    []int
  IsConnected       bool
  LargestComponent  int
  SmallestComponent int
}

func AnalyzeConnectedComponents(gr *graph.Graph) (*ComponentAnalysis,
error) {
  if gr.Nodes == nil {
    return nil, graph.ThrowNodesListIsNil()
  }

  totalComponents := gr.GetConnectedComponents()
  componentSizes := gr.GetComponentSizes()

  var largest, smallest int
  if len(componentSizes) > 0 {
    largest = componentSizes[0]
    smallest = componentSizes[0]
```

```
    for _, size := range componentSizes {
      if size > largest {
        largest = size
      }
      if size < smallest {
        smallest = size
      }
    }
  }

  return &ComponentAnalysis{
    TotalComponents:   totalComponents,
    ComponentSizes:    componentSizes,
    IsConnected:       totalComponents == 1,
    LargestComponent:  largest,
    SmallestComponent: smallest,
  }, nil
}
```

Методы для проверки на дерево описаны в структуре графа.

6 Список смежности III

6.1 Условие

Проверить, можно ли из графа удалить какую-либо вершину так, чтобы получилось дерево.

6.2 Код

```
/*
 * This package contains algorithms and tasks for my SSU course
 */

package algo

import "github.com/tolstovrob/graph-go/graph"

/*
 * Task: Count connected components and analyze their sizes
 */

type ComponentAnalysis struct {
  TotalComponents   int
  ComponentSizes    []int
  IsConnected       bool
  LargestComponent  int
  SmallestComponent int
}

func AnalyzeConnectedComponents(gr *graph.Graph) (*ComponentAnalysis,
error) {
  if gr.Nodes == nil {
    return nil, graph.ThrowNodesListIsNil()
  }

  totalComponents := gr.GetConnectedComponents()
  componentSizes := gr.GetComponentSizes()

  var largest, smallest int
  if len(componentSizes) > 0 {
    largest = componentSizes[0]
    smallest = componentSizes[0]
```

```
    for _, size := range componentSizes {
      if size > largest {
        largest = size
      }
      if size < smallest {
        smallest = size
      }
    }
  }

  return &ComponentAnalysis{
    TotalComponents:   totalComponents,
    ComponentSizes:    componentSizes,
    IsConnected:       totalComponents == 1,
    LargestComponent:  largest,
    SmallestComponent: smallest,
  }, nil
}
```

Методы для проверки на дерево описаны в структуре графа.

7 Список смежности III

7.1 Условие

Дан взвешенный неориентированный граф из N вершин и M ребер. Требуется найти в нем каркас минимального веса. Алгоритм Прима

7.2 Код

```
/*
 * This package contains algorithms and tasks for my SSU course
 */

package algo

import "github.com/tolstovrob/graph-go/graph"

/*
 * Task: Find Minimum Spanning Tree using Prim's algorithm
 *
 * Minimum Spanning Tree (MST) - a tree that connects all vertices with minimum total edge weight
 * Prim's Algorithm - greedy algorithm that grows the MST one vertex at a time
 */

// MSTResult represents the result of Minimum Spanning Tree calculation
type MSTResult struct {
  TotalWeight graph.TWeight // Total weight of all edges in MST
  Edges       []*graph.Edge // List of edges that form the MST
  IsPossible  bool          // Whether MST construction is possible (graph must be connected)
}

// FindMSTPrim finds Minimum Spanning Tree using Prim's algorithm
// Time Complexity: O(V^2) for this implementation, can be optimized to O(E log V) with priority queue
// Space Complexity: O(V + E)
func FindMSTPrim(gr *graph.Graph) (*MSTResult, error) {
  // Input validation: check if graph nodes exist
  if gr.Nodes == nil {
    return nil, graph.ThrowNodesListIsNil()
```

```
  }

  // Base case: empty graph is trivially a tree
  if len(gr.Nodes) == 0 {
    return &MSTResult{
      TotalWeight: 0,
      Edges:       []*graph.Edge{},
      IsPossible:  true,
    }, nil
  }

  // MST Requirement: graph must be connected
  // A disconnected graph cannot have a spanning tree that connects all
vertices
  if !gr.IsConnected() {
    return &MSTResult{
      TotalWeight: 0,
      Edges:       []*graph.Edge{},
      IsPossible:  false, // MST not possible for disconnected graphs
    }, nil
  }

  // Prim's algorithm works on undirected graphs
  // If graph is directed, create an undirected copy for MST
calculation
  if gr.Options.IsDirected {
    tempGraph := gr.Copy()
    tempGraph.UpdateGraph(graph.WithGraphDirected(false))
    return findMSTPrimInternal(tempGraph)
  }

  // For undirected graphs, proceed with normal MST calculation
  return findMSTPrimInternal(gr)
}

// findMSTPrimInternal implements the core Prim's algorithm logic
// Algorithm Strategy: Grow the MST by repeatedly adding the cheapest
edge
// that connects a vertex in MST to a vertex outside MST
func findMSTPrimInternal(gr *graph.Graph) (*MSTResult, error) {
```

```
  // Handle empty graph case
  if len(gr.Nodes) == 0 {
    return &MSTResult{
      TotalWeight: 0,
      Edges:       []*graph.Edge{},
      IsPossible:  true,
    }, nil
  }

  // Data Structures for Algorithm:
  // inMST: tracks which vertices are already included in the MST
  // mstEdges: stores the edges that form the MST
  inMST := make(map[graph.TKey]bool)
  var mstEdges []*graph.Edge

  // Step 1: Initialize with any vertex
  // Prim's algorithm can start from any vertex - choice doesn't affect
result
  var firstVertex graph.TKey
  for vertex := range gr.Nodes {
    firstVertex = vertex
    break // Pick the first available vertex
    // I do love working with queues in go
  }
  inMST[firstVertex] = true // Mark first vertex as included

  // Step 2: Repeat until all vertices are in MST
  // MST must contain exactly V-1 edges for V vertices
  for len(inMST) < len(gr.Nodes) {
    var bestEdge *graph.Edge             // The edge with minimum
weight
    bestWeight := graph.TWeight(1 << 30) // Initialize with "infinity"

    // Step 2.1: Find minimum weight edge connecting MST to non-MST
vertices
    // Strategy: Check all edges from vertices inside MST to their
neighbors outside MST
    for u := range inMST {
      // Explore all neighbors of vertex u (which is already in MST)
      for _, neighbor := range gr.AdjacencyMap[u] {
```

```
        // Only consider neighbors that are NOT yet in MST
        if !inMST[neighbor] {
          // Find the actual edge between u and neighbor
          edge := getEdgeBetweenReliable(gr, u, neighbor)

          // If edge exists and has lower weight than current best,
update best edge
          if edge != nil && edge.Weight < bestWeight {
            bestWeight = edge.Weight
            bestEdge = edge
          }
        }
      }
    }

    if bestEdge == nil {
      return &MSTResult{
        TotalWeight: 0,
        Edges:       []*graph.Edge{},
        IsPossible:  false,
      }, nil
    }

    mstEdges = append(mstEdges, bestEdge)

    if inMST[bestEdge.Source] {
      inMST[bestEdge.Destination] = true
    } else {
      inMST[bestEdge.Source] = true
    }

    // At this point, MST has grown by one vertex and one edge
    // The algorithm maintains the invariant that MST is always a tree
  }

  // Step 3: Calculate total weight of MST
  totalWeight := graph.TWeight(0)
  for _, edge := range mstEdges {
    totalWeight += edge.Weight
  }
```

```
  return &MSTResult{
    TotalWeight: totalWeight,
    Edges:       mstEdges,
    IsPossible:  true,
  }, nil
}

// getEdgeBetweenReliable finds an edge between two vertices in the
graph
// Handles both directed and undirected graphs correctly
func getEdgeBetweenReliable(gr *graph.Graph, u, v graph.TKey)
*graph.Edge {
  // First, search for edge u → v (forward direction)
  for _, edge := range gr.Edges {
    if edge.Source == u && edge.Destination == v {
      return edge
    }
  }

  // For undirected graphs, also search for edge v → u (reverse
direction)
  // In undirected graphs, edge A-B is the same as B-A
  if !gr.Options.IsDirected {
    for _, edge := range gr.Edges {
      if edge.Source == v && edge.Destination == u {
        return edge
      }
    }
  }

  // No edge found between u and v
  return nil
}
```

8 Список смежности IVa

8.1 Условие

Вывести длины кратчайших путей для всех пар вершин. Алгоритм Флойда.

8.2 Код

```
/*
 * This package contains algorithms and tasks for my SSU course
 */

package algo

import (
  "fmt"
  "math"
  "sort"
  "strings"

  "github.com/tolstovrob/graph-go/graph"
)

/*
 * Task: Find eccentricity of vertices and graph radius using
Dijkstra's algorithm
 *
 * Eccentricity of a vertex = maximum shortest path distance from that
vertex to all other vertices
 * Graph Radius = minimum eccentricity among all vertices
 * Graph Diameter = maximum eccentricity among all vertices
 */

// EccentricityResult represents the result of eccentricity and radius
calculation
type EccentricityResult struct {
  Eccentricities     map[graph.TKey]int64 // Eccentricity value for
each vertex
  Radius             int64                // Graph radius (minimum
eccentricity)
  Diameter           int64                // Graph diameter (maximum
```

```
eccentricity)
  CenterVertices     []graph.TKey         // Vertices with eccentricity
= radius
  PeripheralVertices []graph.TKey         // Vertices with eccentricity
= diameter
  IsConnected        bool                 // Whether graph is connected
  Message            string               // Status message
}

// FindEccentricityAndRadius calculates eccentricity for all vertices
and graph radius
// Time Complexity: O(V * E log V) with Dijkstra for each vertex
func FindEccentricityAndRadius(gr *graph.Graph) (*EccentricityResult,
error) {
  if gr.Nodes == nil {
    return nil, graph.ThrowNodesListIsNil()
  }

  // Handle empty graph case
  if len(gr.Nodes) == 0 {
    return &EccentricityResult{
      Eccentricities:     make(map[graph.TKey]int64),
      Radius:             0,
      Diameter:           0,
      CenterVertices:     []graph.TKey{},
      PeripheralVertices: []graph.TKey{},
      IsConnected:        true,
      Message:            "Graph is empty",
    }, nil
  }

  // Step 1: Check for negative weights - Dijkstra cannot handle them
  for _, edge := range gr.Edges {
    if edge.Weight < 0 {
      return nil, fmt.Errorf("Dijkstra's algorithm cannot handle
negative weights. Edge %d has weight %d", edge.Key, edge.Weight)
    }
  }

  // For each vertex, find maximum distance to all other vertices
```

```
  eccentricities := make(map[graph.TKey]int64)
  for vertex := range gr.Nodes {
    // Run Dijkstra from current vertex to find shortest paths to all
others
    distances, err := dijkstra(gr, vertex)
    if err != nil {
      return nil, err
    }

    // Eccentricity = maximum distance to any reachable vertex
    eccentricity := int64(0)
    for _, dist := range distances {
      if dist > eccentricity && dist != math.MaxInt64 {
        eccentricity = dist
      }
    }

    // If any vertex is unreachable, graph is disconnected
    // In disconnected graphs, eccentricity is considered infinite
    for _, dist := range distances {
      if dist == math.MaxInt64 {
        eccentricity = math.MaxInt64
        break
      }
    }

    eccentricities[vertex] = eccentricity
  }

  // Step 3: Calculate radius and diameter
  radius := int64(math.MaxInt64) // Start with "infinity"
  diameter := int64(0)           // Start with 0
  var centerVertices, peripheralVertices []graph.TKey

  // Find minimum and maximum eccentricity values
  for _, ecc := range eccentricities {
    if ecc != math.MaxInt64 { // Only consider reachable vertices
      if ecc < radius {
        radius = ecc
      }
```

```
      if ecc > diameter {
        diameter = ecc
      }
    }
  }

  // Step 4: Find center and peripheral vertices
  for vertex, ecc := range eccentricities {
    if ecc == radius {
      centerVertices = append(centerVertices, vertex)
    }
    if ecc == diameter {
      peripheralVertices = append(peripheralVertices, vertex)
    }
  }

  // Step 5: Handle disconnected graphs
  if radius == math.MaxInt64 {
    radius = -1 // Special value for disconnected graphs
    diameter = -1
  }

  return &EccentricityResult{
    Eccentricities:     eccentricities,
    Radius:             radius,
    Diameter:           diameter,
    CenterVertices:     centerVertices,
    PeripheralVertices: peripheralVertices,
    IsConnected:        radius != -1,
    Message:            fmt.Sprintf("Found eccentricities for %d
vertices", len(gr.Nodes)),
  }, nil
}

// dijkstra implements Dijkstra's algorithm for single-source shortest
paths
// Returns distances from source vertex to all other vertices
func dijkstra(gr *graph.Graph, source graph.TKey)
(map[graph.TKey]int64, error) {
  distances := make(map[graph.TKey]int64)
```

```
visited := make(map[graph.TKey]bool)

// Initialize all distances to "infinity"
for vertex := range gr.Nodes {
  distances[vertex] = math.MaxInt64
}
distances[source] = 0 // Distance to self is 0

// Main Dijkstra loop - process all vertices
for len(visited) < len(gr.Nodes) {
  // Find unvisited vertex with minimum distance
  minVertex := graph.TKey(0)
  minDist := math.MaxInt64

  for vertex, dist := range distances {
    if !visited[vertex] && dist < int64(minDist) {
      minDist = int(dist)
      minVertex = vertex
    }
  }

  // If no more reachable vertices, stop
  if minDist == math.MaxInt64 {
    break
  }

  // Mark vertex as visited
  visited[minVertex] = true

  // Update distances to all neighbors
  for _, neighbor := range gr.AdjacencyMap[minVertex] {
    if !visited[neighbor] {
      edgeWeight := getEdgeWeight(gr, minVertex, neighbor)
      if edgeWeight == math.MaxInt64 {
        continue // No edge exists
      }

      // Relaxation step: update distance if shorter path found
      newDist := distances[minVertex] + edgeWeight
      if newDist < distances[neighbor] {
```

```
          distances[neighbor] = newDist
        }
      }
    }
  }

  return distances, nil
}

// getEdgeWeight finds the weight of an edge between two vertices
func getEdgeWeight(gr *graph.Graph, u, v graph.TKey) int64 {
  for _, edge := range gr.Edges {
    if (edge.Source == u && edge.Destination == v) ||
      (!gr.Options.IsDirected && edge.Source == v && edge.Destination
== u) {
      return int64(edge.Weight)
    }
  }
  return math.MaxInt64 // No edge found
}

// FormatEccentricityResult creates a formatted string representation
func (result *EccentricityResult) FormatEccentricityResult(gr
*graph.Graph) string {
  var sb strings.Builder

  sb.WriteString("ECCENTRICITY AND RADIUS ANALYSIS\n\n")
  sb.WriteString("Algorithm: Dijkstra's Algorithm\n")
  sb.WriteString(fmt.Sprintf("Total vertices: %d\n", len(gr.Nodes)))
  sb.WriteString(fmt.Sprintf("Graph connected: %v\n",
result.IsConnected))
  sb.WriteString(fmt.Sprintf("Radius: %s\n",
formatDistance(result.Radius)))
  sb.WriteString(fmt.Sprintf("Diameter: %s\n",
formatDistance(result.Diameter)))
  sb.WriteString(fmt.Sprintf("Center vertices: %d\n",
len(result.CenterVertices)))
  sb.WriteString(fmt.Sprintf("Peripheral vertices: %d\n\n",
len(result.PeripheralVertices)))
```

```
  sb.WriteString("ECCENTRICITIES BY VERTEX:\n")
  sb.WriteString(strings.Repeat("-", 50) + "\n")

  // Sort vertices for consistent output
  vertices := make([]graph.TKey, 0, len(result.Eccentricities))
  for v := range result.Eccentricities {
    vertices = append(vertices, v)
  }
  sort.Slice(vertices, func(i, j int) bool { return vertices[i] <
vertices[j] })

  // Display eccentricity for each vertex
  for _, vertex := range vertices {
    ecc := result.Eccentricities[vertex]
    node, _ := gr.GetNodeByKey(vertex)

    if node != nil && node.Label != "" {
      sb.WriteString(fmt.Sprintf("Vertex %d (%s): %s\n", vertex,
node.Label, formatDistance(ecc)))
    } else {
      sb.WriteString(fmt.Sprintf("Vertex %d: %s\n", vertex,
formatDistance(ecc)))
    }
  }

  // Display center vertices (vertices with minimum eccentricity)
  if len(result.CenterVertices) > 0 {
    sb.WriteString("\nCENTER VERTICES (eccentricity = radius):\n")
    for i, vertex := range result.CenterVertices {
      node, _ := gr.GetNodeByKey(vertex)
      if node != nil && node.Label != "" {
        sb.WriteString(fmt.Sprintf("%d. Vertex %d (%s)\n", i+1, vertex,
node.Label))
      } else {
        sb.WriteString(fmt.Sprintf("%d. Vertex %d\n", i+1, vertex))
      }
    }
  }

  // Display peripheral vertices (vertices with maximum eccentricity)
```

```
  if len(result.PeripheralVertices) > 0 && result.Diameter != -1 {
    sb.WriteString("\nPERIPHERAL VERTICES (eccentricity = diameter):
\n")
    for i, vertex := range result.PeripheralVertices {
      node, _ := gr.GetNodeByKey(vertex)
      if node != nil && node.Label != "" {
        sb.WriteString(fmt.Sprintf("%d. Vertex %d (%s)\n", i+1, vertex,
node.Label))
      } else {
        sb.WriteString(fmt.Sprintf("%d. Vertex %d\n", i+1, vertex))
      }
    }
  }

  return sb.String()
}

// formatDistance converts numerical distance to readable string
func formatDistance(d int64) string {
  if d == math.MaxInt64 {
    return "inf (unreachable)"
  }
  if d == -1 {
    return "inf (graph disconnected)"
  }
  return fmt.Sprintf("%d", d)
}
```

9 Список смежности IVb

9.1 Условие

Эксцентриситет вершины — максимальное расстояние из всех минимальных расстояний от других вершин до данной вершины. Найти радиус графа — минимальный из эксцентриситетов его вершин. Алгоритм Дейкстры.

9.2 Код

```
/*
 * This package contains algorithms and tasks for my SSU course
 */

package algo

import (
  "fmt"
  "sort"
  "strings"

  "github.com/tolstovrob/graph-go/graph"
)

/*
 * Task: Find shortest paths between all pairs of vertices using Floyd-Warshall algorithm
 *
 * Floyd-Warshall Algorithm - dynamic programming algorithm that finds shortest paths
 * between all pairs of vertices in a weighted graph
 * Can handle negative weights but not negative cycles
 */

// AllPairsShortestPath represents the result of Floyd-Warshall algorithm
type AllPairsShortestPath struct {
  Distances map[graph.TKey]map[graph.TKey]int64      // Shortest distance between every pair
  Next      map[graph.TKey]map[graph.TKey]graph.TKey // Next vertex in shortest path
  IsValid   bool                                     // Whether result
```

```
is valid (no negative cycles)
  Message   string                                  // Status message
about computation
}

// FindAllPairsShortestPath finds shortest paths between all vertex
pairs using Floyd-Warshall
// Time Complexity: O(V^3) where V is number of vertices
// Can handle: directed/undirected graphs, negative weights (but not
negative cycles)
func FindAllPairsShortestPath(gr *graph.Graph) (*AllPairsShortestPath,
error) {
  if gr.Nodes == nil {
    return nil, graph.ThrowNodesListIsNil()
  }

  // Handle empty graph case
  if len(gr.Nodes) == 0 {
    return &AllPairsShortestPath{
      Distances: make(map[graph.TKey]map[graph.TKey]int64),
      Next:      make(map[graph.TKey]map[graph.TKey]graph.TKey),
      IsValid:   true,
      Message:   "Graph is empty",
    }, nil
  }

  return findFloydWarshall(gr)
}

// findFloydWarshall implements the core Floyd-Warshall algorithm
// Algorithm Strategy: Dynamic Programming - gradually improve shortest
path estimates
// by considering each vertex as an intermediate point
func findFloydWarshall(gr *graph.Graph) (*AllPairsShortestPath, error)
{
  // Step 1: Get sorted node keys for consistent processing
  // This ensures we always process vertices in the same order
  keys := getSortedKeys(gr.Nodes)

  // Step 2: Initialize distance and next matrices
```

```
// dist[i][j] = shortest distance from i to j
// next[i][j] = next vertex after i in shortest path to j
dist := make(map[graph.TKey]map[graph.TKey]int64)
next := make(map[graph.TKey]map[graph.TKey]graph.TKey)

infinity := int64(1 << 30)

// Step 3: Initialize matrices with base cases
for _, i := range keys {
  dist[i] = make(map[graph.TKey]int64)
  next[i] = make(map[graph.TKey]graph.TKey)

  for _, j := range keys {
    if i == j {
      dist[i][j] = 0
    } else {
      dist[i][j] = infinity
    }
    next[i][j] = 0
  }
}

// Step 4: Initialize with direct edges
// Set distances for edges that exist directly in the graph
for _, edge := range gr.Edges {
  weight := int64(edge.Weight)

  // Set direct edge distance if it's better than current value
  if weight < dist[edge.Source][edge.Destination] {
    dist[edge.Source][edge.Destination] = weight
    next[edge.Source][edge.Destination] = edge.Destination
  }

  // For undirected graphs, set reverse edge as well
  if !gr.Options.IsDirected {
    if weight < dist[edge.Destination][edge.Source] {
      dist[edge.Destination][edge.Source] = weight
      next[edge.Destination][edge.Source] = edge.Source
    }
  }
```

```
  }

  // Step 5: Floyd-Warshall Algorithm
  // Consider each vertex as an intermediate point
  for _, k := range keys {
    for _, i := range keys { // Source vertex
      if dist[i][k] == infinity {
        continue
      }

      for _, j := range keys { // Destination vertex
        if dist[k][j] == infinity {
          continue
        }

        if dist[i][k]+dist[k][j] < dist[i][j] {
          dist[i][j] = dist[i][k] + dist[k][j]
          next[i][j] = next[i][k] // Path goes through k next
        }
      }
    }
  }

  // Step 6: Check for negative weight cycles
  // Negative cycle exists if any dist[i][i] < 0 (distance to self
becomes negative)
  for _, k := range keys {
    if dist[k][k] < 0 {
      return &AllPairsShortestPath{
        IsValid: false,
        Message: "Graph contains negative weight cycles",
      }, nil
    }
  }

  // Step 7: Return successful result
  return &AllPairsShortestPath{
    Distances: dist,
    Next:      next,
    IsValid:   true,
```

```
    Message:   fmt.Sprintf("Computed shortest paths for %d vertices",
len(keys)),
  }, nil
}

// GetPath reconstructs the shortest path from start to end using the
next matrix
// Returns the sequence of vertices in the shortest path
func (apsp *AllPairsShortestPath) GetPath(start, end graph.TKey)
[]graph.TKey {
  // Check if no path exists
  if apsp.Next[start][end] == 0 {
    return nil
  }

  // Reconstruct path by following next pointers
  path := []graph.TKey{start}
  current := start

  for current != end {
    current = apsp.Next[current][end]
    path = append(path, current)
  }

  return path
}

// FormatDistanceMatrix creates a formatted string representation of
the distance matrix
// Useful for displaying results in CLI
func (apsp *AllPairsShortestPath) FormatDistanceMatrix(gr *graph.Graph)
string {
  if !apsp.IsValid {
    return apsp.Message
  }

  var sb strings.Builder
  keys := getSortedKeys(gr.Nodes)

  // Header information
```

```
  sb.WriteString("SHORTEST PATH DISTANCES BETWEEN ALL PAIRS OF
VERTICES\n\n")
  sb.WriteString("Algorithm: Floyd-Warshall\n")
  sb.WriteString(fmt.Sprintf("Total vertices: %d\n", len(keys)))
  sb.WriteString(fmt.Sprintf("Total edges: %d\n", len(gr.Edges)))
  sb.WriteString(fmt.Sprintf("Directed: %v\n\n",
gr.Options.IsDirected))

  // Table header row
  sb.WriteString(fmt.Sprintf("%-8s", "From\\To"))
  for _, j := range keys {
    node, _ := gr.GetNodeByKey(j)
    if node != nil && node.Label != "" {
      sb.WriteString(fmt.Sprintf("%-12s", fmt.Sprintf("%d(%s)", j,
node.Label)))
    } else {
      sb.WriteString(fmt.Sprintf("%-12d", j))
    }
  }
  sb.WriteString("\n")

  // Table separator
  sb.WriteString(strings.Repeat("-", 8+len(keys)*12) + "\n")

  // Distance matrix rows
  infinity := int64(1 << 30)
  for _, i := range keys {
    node, _ := gr.GetNodeByKey(i)
    if node != nil && node.Label != "" {
      sb.WriteString(fmt.Sprintf("%-8s", fmt.Sprintf("%d(%s)", i,
node.Label)))
    } else {
      sb.WriteString(fmt.Sprintf("%-8d", i))
    }

    // Distance values for this row
    for _, j := range keys {
      dist := apsp.Distances[i][j]
      if dist == infinity {
        sb.WriteString(fmt.Sprintf("%-12s", "inf"))
```

```
      } else if i == j {
        sb.WriteString(fmt.Sprintf("%-12s", "0"))
      } else {
        sb.WriteString(fmt.Sprintf("%-12d", dist))
      }
    }
    sb.WriteString("\n")
  }

  // Connectivity analysis
  sb.WriteString("\nCONNECTIVITY:\n")
  sb.WriteString(strings.Repeat("-", 50) + "\n")

  reachablePairs := 0
  totalPairs := len(keys) * (len(keys) - 1)

  for _, i := range keys {
    for _, j := range keys {
      if i != j && apsp.Distances[i][j] < infinity {
        reachablePairs++
      }
    }
  }

  sb.WriteString(fmt.Sprintf("Reachable vertex pairs: %d/%d (%.1f%
%)\n",
    reachablePairs, totalPairs, float64(reachablePairs)/
float64(totalPairs)*100))

  return sb.String()
}

// getSortedKeys returns sorted node keys for consistent processing
// Important for Floyd-Warshall to maintain consistent vertex ordering
func getSortedKeys(nodes map[graph.TKey]*graph.Node) []graph.TKey {
  keys := make([]graph.TKey, 0, len(nodes))
  for key := range nodes {
    keys = append(keys, key)
  }
  sort.Slice(keys, func(i, j int) bool { return keys[i] < keys[j] })
```

```
  return keys
}
```

10 Список смежности IVc

10.1 Условие

Вывести все отрицательные циклы. Алгоритм Беллмана-Форда.

10.2 Код

```
/*
 * This package contains algorithms and tasks for my SSU course
 */

package algo

import (
  "fmt"
  "sort"
  "strings"

  "github.com/tolstovrob/graph-go/graph"
)

/*
 * Task: Find all negative cycles using Bellman-Ford algorithm
 *
 * Negative Cycle: A cycle in a graph where the sum of edge weights is
negative
 * Bellman-Ford Algorithm: Single-source shortest path algorithm that
can detect negative cycles
 * Time Complexity: O(V * E) for each source vertex, O(V^2 * E) total
 */

// NegativeCycle represents a single negative cycle found in the graph
type NegativeCycle struct {
  Vertices    []graph.TKey  `json:"vertices"`     // Ordered list of
vertices in the cycle
  Edges       []graph.TKey  `json:"edges"`        // Ordered list of
edges in the cycle
  TotalWeight graph.TWeight `json:"total_weight"` // Sum of all edge
weights in the cycle
}
```

```go
// NegativeCyclesResult contains the complete result of negative cycle
detection
type NegativeCyclesResult struct {
  Cycles            []NegativeCycle `json:"cycles"`              //
List of all unique negative cycles found
  HasNegativeCycles bool            `json:"has_negative_cycles"` //
Whether any negative cycles exist
  TotalCycles       int             `json:"total_cycles"`        //
Count of unique negative cycles
  Message           string          `json:"message"`             //
Status message describing the result
}

// FindNegativeCycles finds all negative cycles in the graph using
Bellman-Ford algorithm
// This is the main entry point for negative cycle detection
func FindNegativeCycles(gr *graph.Graph) (*NegativeCyclesResult, error)
{
  if gr.Nodes == nil {
    return nil, graph.ThrowNodesListIsNil()
  }

  // Handle empty graph case - no cycles possible
  if len(gr.Nodes) == 0 {
    return &NegativeCyclesResult{
      Cycles:            []NegativeCycle{},
      HasNegativeCycles: false,
      TotalCycles:       0,
      Message:           "Graph is empty",
    }, nil
  }

  // Bellman-Ford requires directed graphs for negative cycle detection
  if !gr.Options.IsDirected {
    return &NegativeCyclesResult{
      Cycles:            []NegativeCycle{},
      HasNegativeCycles: false,
      TotalCycles:       0,
      Message:           "Bellman-Ford algorithm for negative cycles
requires directed graph",
```

```
    }, nil
  }

  allCycles := findAllNegativeCycles(gr)

  return &NegativeCyclesResult{
    Cycles:            allCycles,
    HasNegativeCycles: len(allCycles) > 0,
    TotalCycles:       len(allCycles),
    Message:           fmt.Sprintf("Found %d negative cycle(s)",
len(allCycles)),
  }, nil
}

// findAllNegativeCycles executes Bellman-Ford from each vertex to find
all negative cycles
// This is the core algorithm implementation
func findAllNegativeCycles(gr *graph.Graph) []NegativeCycle {
  keys := getSortedNodeKeys(gr.Nodes)    // Get sorted vertices for
consistent processing
  allCycles := []NegativeCycle{}         // Store all found cycles
  visitedCycles := make(map[string]bool) // Track seen cycles to avoid
duplicates

  // Try each vertex as a potential starting point for cycle detection
  for _, start := range keys {
    dist := make(map[graph.TKey]graph.TWeight)  // Shortest distance
estimates
    prev := make(map[graph.TKey]graph.TKey)     // Predecessor vertices
for path reconstruction
    edgePrev := make(map[graph.TKey]graph.TKey) // Predecessor edges
for cycle tracing

    // Initialize with large values representing infinity
    infinity := graph.TWeight(1 << 30)
    for _, key := range keys {
      dist[key] = infinity
    }
    dist[start] = 0 // Distance to start vertex is 0
```

```
    // Relaxation phase: |V| - 1 iterations of edge relaxation
    for i := 0; i < len(keys)-1; i++ {
      changed := false // Track if any distances were updated
      for _, edge := range gr.Edges {
        u, v, w := edge.Source, edge.Destination, edge.Weight
        // If we found a shorter path through u to v, update
        if dist[u] != infinity && dist[u]+w < dist[v] {
          dist[v] = dist[u] + w
          prev[v] = u
          edgePrev[v] = edge.Key
          changed = true
        }
      }
      // Early termination if no improvements in this iteration
      if !changed {
        break
      }
    }

    // Negative cycle detection phase: check if we can still relax
edges
    for _, edge := range gr.Edges {
      u, v, w := edge.Source, edge.Destination, edge.Weight
      // If we can still improve after |V|-1 iterations, negative cycle
exists
      if dist[u] != infinity && dist[u]+w < dist[v] {
        // Trace and reconstruct the actual cycle
        cycle := traceCycle(gr, u, v, prev, edgePrev)
        if cycle != nil {
          // Normalize cycle representation and check for duplicates
          normalized := normalizeCycle(*cycle)
          cycleKey := generateCycleKey(normalized)
          if !visitedCycles[cycleKey] && normalized.TotalWeight < 0 {
            visitedCycles[cycleKey] = true
            allCycles = append(allCycles, normalized)
          }
        }
      }
    }
  }
```

```
  return allCycles
}

// traceCycle traces back from a negatively-weighted edge to find the
actual cycle
// Uses Floyd's cycle-finding algorithm (tortoise and hare)
func traceCycle(gr *graph.Graph, u, v graph.TKey, prev, edgePrev
map[graph.TKey]graph.TKey) *NegativeCycle {
  // Use two pointers to detect cycle: slow moves 1 step, fast moves 2
steps
  slow, fast := v, v

  // Find meeting point inside the cycle
  for i := 0; i < len(gr.Nodes); i++ {
    if prev[fast] == 0 || prev[prev[fast]] == 0 {
      return nil // Invalid pointers, no cycle found
    }
    slow = prev[slow]
    fast = prev[prev[fast]]
    if slow == fast {
      break // Cycle detected
    }
  }

  if slow != fast {
    return nil // No cycle found
  }

  // Find the start node of the cycle
  cycleStart := findCycleStart(slow, prev)
  if cycleStart == 0 {
    return nil
  }

  // Reconstruct the complete cycle
  return reconstructCycle(gr, cycleStart, prev, edgePrev)
}

// findCycleStart finds the starting node of a cycle using cycle length
```

```
detection
func findCycleStart(meetingPoint graph.TKey, prev
map[graph.TKey]graph.TKey) graph.TKey {
  // Determine cycle length by traversing until we return to meeting
point
  cycleLength := 0
  current := meetingPoint
  for {
    current = prev[current]
    cycleLength++
    if current == meetingPoint {
      break
    }
  }

  // Use two pointers to find cycle start: one starts cycleLength steps
ahead
  ptr1 := meetingPoint
  for i := 0; i < cycleLength; i++ {
    ptr1 = prev[ptr1]
  }

  // Move both pointers until they meet at cycle start
  ptr2 := meetingPoint
  for ptr1 != ptr2 {
    ptr1 = prev[ptr1]
    ptr2 = prev[ptr2]
  }

  return ptr1
}

// reconstructCycle builds the complete cycle from predecessor
information
func reconstructCycle(gr *graph.Graph, start graph.TKey, prev, edgePrev
map[graph.TKey]graph.TKey) *NegativeCycle {
  cycleVertices := []graph.TKey{start}
  cycleEdges := []graph.TKey{}
  totalWeight := graph.TWeight(0)
  visited := make(map[graph.TKey]bool)
```

```
visited[start] = true

// Reconstruct vertices in the cycle by following predecessor links
current := prev[start]
for current != start {
  if current == 0 || visited[current] {
    return nil // Invalid cycle
  }
  visited[current] = true
  cycleVertices = append([]graph.TKey{current}, cycleVertices...)
  current = prev[current]

  // Safety check to prevent infinite loops
  if len(cycleVertices) > len(gr.Nodes) {
    return nil
  }
}

// Reconstruct edges and calculate total weight
for i := 0; i < len(cycleVertices); i++ {
  from := cycleVertices[i]
  to := cycleVertices[(i+1)%len(cycleVertices)]

  edge := findEdgeBetween(gr, from, to)
  if edge == nil {
    return nil
  }

  cycleEdges = append(cycleEdges, edge.Key)
  totalWeight += edge.Weight
}

if len(cycleVertices) < 2 {
  return nil
}

return &NegativeCycle{
  Vertices:    cycleVertices,
  Edges:       cycleEdges,
  TotalWeight: totalWeight,
```

```
  }
}

// normalizeCycle rotates the cycle to start with the smallest vertex
for consistent comparison
func normalizeCycle(cycle NegativeCycle) NegativeCycle {
  if len(cycle.Vertices) == 0 {
    return cycle
  }

  // Find the vertex with smallest key value
  minIndex := 0
  minVertex := cycle.Vertices[0]
  for i, vertex := range cycle.Vertices {
    if vertex < minVertex {
      minVertex = vertex
      minIndex = i
    }
  }

  // Rotate cycle to start with smallest vertex
  normalizedVertices := make([]graph.TKey, len(cycle.Vertices))
  copy(normalizedVertices, cycle.Vertices[minIndex:])
  copy(normalizedVertices[len(cycle.Vertices)-minIndex:],
cycle.Vertices[:minIndex])

  return NegativeCycle{
    Vertices:    normalizedVertices,
    Edges:       cycle.Edges, // Edge order doesn't affect cycle
identity
    TotalWeight: cycle.TotalWeight,
  }
}

// generateCycleKey creates a unique string identifier for a cycle
func generateCycleKey(cycle NegativeCycle) string {
  vertices := make([]string, len(cycle.Vertices))
  for i, v := range cycle.Vertices {
    vertices[i] = fmt.Sprintf("%d", v)
  }
```

```
  return strings.Join(vertices, "-")
}

// Helper functions for graph operations

// getSortedNodeKeys returns vertices sorted by key for consistent
processing
func getSortedNodeKeys(nodes map[graph.TKey]*graph.Node) []graph.TKey {
  keys := make([]graph.TKey, 0, len(nodes))
  for key := range nodes {
    keys = append(keys, key)
  }
  sort.Slice(keys, func(i, j int) bool { return keys[i] < keys[j] })
  return keys
}

// findEdgeBetween finds the edge between two vertices in the graph
func findEdgeBetween(gr *graph.Graph, from, to graph.TKey) *graph.Edge
{
  for _, edge := range gr.Edges {
    if edge.Source == from && edge.Destination == to {
      return edge
    }
  }
  return nil
}

// FormatNegativeCyclesResult creates a human-readable formatted output
func (result *NegativeCyclesResult) FormatNegativeCyclesResult(gr
*graph.Graph) string {
  var sb strings.Builder

  sb.WriteString("NEGATIVE CYCLES ANALYSIS\n\n")
  sb.WriteString("Algorithm: Bellman-Ford\n")
  sb.WriteString(fmt.Sprintf("Total vertices: %d\n", len(gr.Nodes)))
  sb.WriteString(fmt.Sprintf("Total edges: %d\n", len(gr.Edges)))
  sb.WriteString(fmt.Sprintf("Graph directed: %v\n",
gr.Options.IsDirected))
  sb.WriteString(fmt.Sprintf("Found negative cycles: %v\n",
result.HasNegativeCycles))
```

```
  sb.WriteString(fmt.Sprintf("Total unique cycles: %d\n\n",
result.TotalCycles))

  if !result.HasNegativeCycles {
    sb.WriteString("No negative cycles found in the graph.\n")
    if !gr.Options.IsDirected {
      sb.WriteString("Note: Bellman-Ford for negative cycles requires
directed graphs.\n")
    }
    return sb.String()
  }

  // Display each found cycle with detailed information
  for i, cycle := range result.Cycles {
    sb.WriteString(fmt.Sprintf("NEGATIVE CYCLE %d:\n", i+1))
    sb.WriteString(strings.Repeat("─", 40) + "\n")
    sb.WriteString(fmt.Sprintf("Total weight: %d\n",
cycle.TotalWeight))
    sb.WriteString(fmt.Sprintf("Length: %d vertices, %d edges\n\n",
len(cycle.Vertices), len(cycle.Edges)))

    sb.WriteString("CYCLE PATH:\n")
    // Display each edge in the cycle
    for j := 0; j < len(cycle.Vertices); j++ {
      current := cycle.Vertices[j]
      next := cycle.Vertices[(j+1)%len(cycle.Vertices)]

      currentNode, _ := gr.GetNodeByKey(current)
      nextNode, _ := gr.GetNodeByKey(next)

      var currentLabel, nextLabel string
      if currentNode != nil && currentNode.Label != "" {
        currentLabel = fmt.Sprintf(" (%s)", currentNode.Label)
      }
      if nextNode != nil && nextNode.Label != "" {
        nextLabel = fmt.Sprintf(" (%s)", nextNode.Label)
      }

      edge := findEdgeBetween(gr, current, next)
      var edgeLabel string
```

```
      if edge != nil && edge.Label != "" {
        edgeLabel = edge.Label
      }

      sb.WriteString(fmt.Sprintf("  %d%s → %d%s", current,
currentLabel, next, nextLabel))
      sb.WriteString(fmt.Sprintf(" [Weight: %d", edge.Weight))
      if edgeLabel != "" {
        sb.WriteString(fmt.Sprintf(", Edge: %s", edgeLabel))
      }
      sb.WriteString("]\n")
    }

    sb.WriteString(fmt.Sprintf("\nCycle completes with total weight:
%d\n", cycle.TotalWeight))

    if i < len(result.Cycles)-1 {
      sb.WriteString("\n" + strings.Repeat("=", 50) + "\n\n")
    }
  }

  return sb.String()
}
```

11 Список смежности V

11.1 Условие

Максимальный поток. Алгоритм Эдмонса Карпа

11.2 Код

```
/*
 * This package contains algorithms and tasks for my SSU course
 */

package algo

import (
  "fmt"
  "sort"
  "strings"

  "github.com/tolstovrob/graph-go/graph"
)

/*
 * Task: Find maximum flow using Edmonds-Karp algorithm (BFS-based
Ford-Fulkerson)
 */

// FlowEdge represents an edge with flow information
type FlowEdge struct {
  Source      graph.TKey    `json:"source"`
  Destination graph.TKey    `json:"destination"`
  Capacity    graph.TWeight `json:"capacity"`
  Flow        graph.TWeight `json:"flow"`
}

// MaxFlowResult contains the result of maximum flow calculation
type MaxFlowResult struct {
  MaxFlowValue graph.TWeight `json:"max_flow_value"`
  Source       graph.TKey    `json:"source"`
  Sink         graph.TKey    `json:"sink"`
  FlowEdges    []FlowEdge    `json:"flow_edges"`
  MinCut       []graph.TKey  `json:"min_cut"`
```

```
  Message      string        `json:"message"`
}

// FindMaxFlow finds maximum flow from source to sink using Edmonds-
Karp algorithm
func FindMaxFlow(gr *graph.Graph, source, sink graph.TKey)
(*MaxFlowResult, error) {
  if gr.Nodes == nil {
    return nil, graph.ThrowNodesListIsNil()
  }

  // Validate source and sink nodes
  if _, err := gr.GetNodeByKey(source); err != nil {
    return nil, fmt.Errorf("source node %d does not exist", source)
  }

  if _, err := gr.GetNodeByKey(sink); err != nil {
    return nil, fmt.Errorf("sink node %d does not exist", sink)
  }

  if source == sink {
    return nil, fmt.Errorf("source and sink cannot be the same node")
  }

  // Create residual graph and initialize flow
  residualGraph := createResidualGraph(gr)
  flowMap := initializeFlowMap(gr)

  maxFlow := graph.TWeight(0)

  // Edmonds-Karp algorithm: repeatedly find augmenting paths using BFS
  for {
    // Find augmenting path in residual graph
    path, parent := findAugmentingPath(residualGraph, source, sink)
    if path == nil {
      break // No more augmenting paths
    }

    // Find bottleneck capacity in the path
    pathFlow := findBottleneckCapacity(residualGraph, path, parent,
```

```
sink)

    // Update residual capacities and flow along the path
    updateResidualGraph(residualGraph, flowMap, path, parent, pathFlow,
sink, gr)

    maxFlow += pathFlow
  }

  // Build result
  flowEdges := buildFlowEdges(gr, flowMap)
  minCut := findMinCut(residualGraph, source)

  return &MaxFlowResult{
    MaxFlowValue: maxFlow,
    Source:       source,
    Sink:         sink,
    FlowEdges:    flowEdges,
    MinCut:       minCut,
    Message:      fmt.Sprintf("Maximum flow from %d to %d is %d",
source, sink, maxFlow),
  }, nil
}

// createResidualGraph creates the residual graph from original graph
func createResidualGraph(gr *graph.Graph)
map[graph.TKey]map[graph.TKey]graph.TWeight {
  residual := make(map[graph.TKey]map[graph.TKey]graph.TWeight)

  // Initialize residual capacities
  for u := range gr.Nodes {
    residual[u] = make(map[graph.TKey]graph.TWeight)
    for v := range gr.Nodes {
      residual[u][v] = 0
    }
  }

  // Set initial capacities from original edges
  for _, edge := range gr.Edges {
    capacity := edge.Weight
```

```
    if capacity <= 0 {
      capacity = 1 // Default capacity for zero/negative weights
    }
    residual[edge.Source][edge.Destination] = capacity
  }

  return residual
}

// initializeFlowMap creates initial flow map with zero flow
func initializeFlowMap(gr *graph.Graph)
map[graph.TKey]map[graph.TKey]graph.TWeight {
  flowMap := make(map[graph.TKey]map[graph.TKey]graph.TWeight)
  for u := range gr.Nodes {
    flowMap[u] = make(map[graph.TKey]graph.TWeight)
    for v := range gr.Nodes {
      flowMap[u][v] = 0
    }
  }
  return flowMap
}

// findAugmentingPath finds a path from source to sink using BFS
func findAugmentingPath(residualGraph
map[graph.TKey]map[graph.TKey]graph.TWeight, source, sink graph.TKey)
([]graph.TKey, map[graph.TKey]graph.TKey) {
  visited := make(map[graph.TKey]bool)
  parent := make(map[graph.TKey]graph.TKey)
  queue := []graph.TKey{source}
  visited[source] = true

  for len(queue) > 0 {
    u := queue[0]
    queue = queue[1:]

    // Check all neighbors with positive residual capacity
    for v, capacity := range residualGraph[u] {
      if !visited[v] && capacity > 0 {
        parent[v] = u
        visited[v] = true
```

```
            queue = append(queue, v)

            // If we reached sink, reconstruct path
            if v == sink {
              return reconstructPath(parent, source, sink), parent
            }
          }
        }
      }

      return nil, parent
}

// reconstructPath builds the path from parent pointers
func reconstructPath(parent map[graph.TKey]graph.TKey, source, sink
graph.TKey) []graph.TKey {
      path := []graph.TKey{}
      current := sink

      for current != source {
        path = append([]graph.TKey{current}, path...)
        current = parent[current]
      }
      path = append([]graph.TKey{source}, path...)

      return path
}

// findBottleneckCapacity finds the minimum residual capacity along the
path
func findBottleneckCapacity(residualGraph
map[graph.TKey]map[graph.TKey]graph.TWeight, path []graph.TKey, parent
map[graph.TKey]graph.TKey, sink graph.TKey) graph.TWeight {
      bottleneck := graph.TWeight(1 << 30) // Large number
      v := sink

      for v != path[0] { // While not at source
        u := parent[v]
        if residualGraph[u][v] < bottleneck {
          bottleneck = residualGraph[u][v]
```

```
    }
    v = u
  }

  return bottleneck
}

// updateResidualGraph updates residual capacities and flow after
augmenting path
func updateResidualGraph(residualGraph
map[graph.TKey]map[graph.TKey]graph.TWeight, flowMap
map[graph.TKey]map[graph.TKey]graph.TWeight, path []graph.TKey, parent
map[graph.TKey]graph.TKey, pathFlow graph.TWeight, sink graph.TKey, gr
*graph.Graph) {
  v := sink

  for v != path[0] { // While not at source
    u := parent[v]

    // Update residual capacities
    residualGraph[u][v] -= pathFlow
    residualGraph[v][u] += pathFlow

    // Update flow
    if hasOriginalEdge(gr, u, v) {
      flowMap[u][v] += pathFlow
    } else {
      // Backward edge - subtract flow
      flowMap[v][u] -= pathFlow
    }

    v = u
  }
}

// hasOriginalEdge checks if an edge exists in the original graph
func hasOriginalEdge(gr *graph.Graph, u, v graph.TKey) bool {
  for _, edge := range gr.Edges {
    if edge.Source == u && edge.Destination == v {
      return true
```

```
    }
  }
  return false
}

// buildFlowEdges creates the list of flow edges from flow map
func buildFlowEdges(gr *graph.Graph, flowMap
map[graph.TKey]map[graph.TKey]graph.TWeight) []FlowEdge {
  flowEdges := []FlowEdge{}

  for u := range flowMap {
    for v := range flowMap[u] {
      flow := flowMap[u][v]
      if flow > 0 {
        // Find original capacity
        capacity := graph.TWeight(1)
        for _, edge := range gr.Edges {
          if edge.Source == u && edge.Destination == v {
            if edge.Weight > 0 {
              capacity = edge.Weight
            }
            break
          }
        }

        flowEdges = append(flowEdges, FlowEdge{
          Source:      u,
          Destination: v,
          Capacity:    capacity,
          Flow:        flow,
        })
      }
    }
  }

  // Sort for consistent output
  sort.Slice(flowEdges, func(i, j int) bool {
    if flowEdges[i].Source == flowEdges[j].Source {
      return flowEdges[i].Destination < flowEdges[j].Destination
    }
```

```
    return flowEdges[i].Source < flowEdges[j].Source
  })

  return flowEdges
}

// findMinCut finds the minimum cut (reachable nodes from source in
residual graph)
func findMinCut(residualGraph
map[graph.TKey]map[graph.TKey]graph.TWeight, source graph.TKey)
[]graph.TKey {
  visited := make(map[graph.TKey]bool)
  queue := []graph.TKey{source}
  visited[source] = true

  for len(queue) > 0 {
    u := queue[0]
    queue = queue[1:]

    for v, capacity := range residualGraph[u] {
      if !visited[v] && capacity > 0 {
        visited[v] = true
        queue = append(queue, v)
      }
    }
  }

  // Convert visited map to sorted slice
  minCut := []graph.TKey{}
  for node := range visited {
    minCut = append(minCut, node)
  }
  sort.Slice(minCut, func(i, j int) bool { return minCut[i] <
minCut[j] })

  return minCut
}

// FormatMaxFlowResult creates a formatted string representation
func (result *MaxFlowResult) FormatMaxFlowResult(gr *graph.Graph)
```

```
string {
  var sb strings.Builder

  sb.WriteString("MAXIMUM FLOW ANALYSIS\n\n")
  sb.WriteString("Algorithm: Edmonds-Karp (BFS-based Ford-
Fulkerson)\n")
  sb.WriteString(fmt.Sprintf("Source: %d", result.Source))
  if node, _ := gr.GetNodeByKey(result.Source); node != nil &&
node.Label != "" {
    sb.WriteString(fmt.Sprintf(" (%s)", node.Label))
  }
  sb.WriteString(fmt.Sprintf("\nSink: %d", result.Sink))
  if node, _ := gr.GetNodeByKey(result.Sink); node != nil &&
node.Label != "" {
    sb.WriteString(fmt.Sprintf(" (%s)", node.Label))
  }
  sb.WriteString(fmt.Sprintf("\nMaximum Flow Value: %d\n\n",
result.MaxFlowValue))

  sb.WriteString("FLOW DISTRIBUTION:\n")
  sb.WriteString(strings.Repeat("-", 60) + "\n")
  sb.WriteString(fmt.Sprintf("%-8s %-8s %-12s %-12s %-12s\n", "From",
"To", "Capacity", "Flow", "Utilization"))
  sb.WriteString(fmt.Sprintf("%-8s %-8s %-12s %-12s %-12s\n", "----",
"--", "--------", "----", "----------"))

  totalCapacity := graph.TWeight(0)
  totalFlow := graph.TWeight(0)

  for _, edge := range result.FlowEdges {
    fromLabel := fmt.Sprintf("%d", edge.Source)
    if node, _ := gr.GetNodeByKey(edge.Source); node != nil &&
node.Label != "" {
      fromLabel = fmt.Sprintf("%d(%s)", edge.Source, node.Label)
    }

    toLabel := fmt.Sprintf("%d", edge.Destination)
    if node, _ := gr.GetNodeByKey(edge.Destination); node != nil &&
node.Label != "" {
      toLabel = fmt.Sprintf("%d(%s)", edge.Destination, node.Label)
```

```
    }

    utilization := "0%"
    if edge.Capacity > 0 {
      utilization = fmt.Sprintf("%.1f%%", float64(edge.Flow)*100/
float64(edge.Capacity))
    }

    sb.WriteString(fmt.Sprintf("%-8s %-8s %-12d %-12d %-12s\n",
      fromLabel, toLabel, edge.Capacity, edge.Flow, utilization))

    totalCapacity += edge.Capacity
    totalFlow += edge.Flow
  }

  sb.WriteString("\nSUMMARY:\n")
  sb.WriteString(strings.Repeat("-", 40) + "\n")
  sb.WriteString(fmt.Sprintf("Total capacity: %d\n", totalCapacity))
  sb.WriteString(fmt.Sprintf("Total flow: %d\n", totalFlow))
  if totalCapacity > 0 {
    sb.WriteString(fmt.Sprintf("Flow efficiency: %.1f%%\n",
float64(totalFlow)*100/float64(totalCapacity)))
  }

  sb.WriteString(fmt.Sprintf("\nMINIMUM CUT (%d nodes):\n",
len(result.MinCut)))
  for i, node := range result.MinCut {
    nodeLabel := fmt.Sprintf("%d", node)
    if nodeObj, _ := gr.GetNodeByKey(node); nodeObj != nil &&
nodeObj.Label != "" {
      nodeLabel = fmt.Sprintf("%d (%s)", node, nodeObj.Label)
    }
    sb.WriteString(fmt.Sprintf("%d. %s\n", i+1, nodeLabel))
  }

  return sb.String()
}
```