Алгоритмы на графах

Общие обозначения

G - граф.
G.V - множество вершин графа (V).
G.E - множество ребер/дуг графа (E).
G.adj(u) - множесто смежных с вершиной u вершин (соединенных с u ребром/дугой).

Поиск в глубину

Из стартовой вершины просматриваются все смежные с ней вершины (соединенные ребром или дугой); если смежная вершина еще не посещена, процедура повторяется из нее рекурсивно; если все смежные вершины посещены или отсутствуют, выполняется “откат” в предыдущую вершину, или остановка процедуры, если предыдущая вершина отсутствует. Дерево посещения вершин - дерево поиска в глубину (при поиске в глубину на всем графе возможно получение нескольких деревьев).

```
DFS(G):
      for each u in G.V:
            u.visited = false
      for each u in G.V:
            if not u.visited:
                  DFS(G, u)
DFS(G, u):
      u.visited = true
      for each v in G.adj(u):
            if not v.visited:
                  DFS(G, v)
```

Временная сложность: O(V + E).

Поиск в ширину

Начиная из некоторой стартовой вершины, производится посещение всех смежных с ней вершин, затем посещение всех вершин, смежных с предыдущими вершинами и т.д. (т.е. последовательное посещение вершин на расстоянии 1-го перехода из стартовой вершины, 2-х переходов, 3-х переходов и т.д.). Процедура останавливается после посещения всех достижимых вершин. Дерево посещений вершин - дерево поиска в ширину.

```
BFS(G, s):
      for each u in G.V:
            u.visited = false
```

```
s.visited = true
Queue = {s}
while Queue not empty:
      u = Queue.pop()
      for each v in G.adj(u):
            if not v.visited:
                  v.visited = true
                  Queue.push(v)
```

Временная сложность: O(V + E).

Топологическая сортировка

Топологическая сортировка направленного ациклического графа (DAG - Directed Acyclic Graph) - расположение его вершин в линейном порядке, при котором все его дуги направлены в одну сторону, т.е.: если order: $V \to N$ - функция, задающая индекс вершины графа в отсортированном порядке, то $\forall$ (u,v) $\in$ E: order(u) < order(v).

Топологическая сортировка производится путем построения списка пройденых вершин при поиске в глубину на графе. По завершении процедуры порядок вершин в списке будет топологически отсортированным порядком вершин.

```
DFS(G):
      order = {}
      …
      return order

DFS(G, u):
      u.visited = true
      for each v in G.adj(u):
            if not v.visited:
                  DFS(G, v)
      order.push_front(u)
```

Временная сложность: O(V + E).

Сильно связные компоненты

Две вершины u и v ориентированного графа G являются сильно связными (strongly connected), если u достижима из v и v достижима из u (существует путь из одной вершины в другую и обратно). Сильно связная компонента (strongly connected component) ориентированного графа - максимальное подмножество вершин, все пары вершин в котором сильно связные: $C \subseteq V$, $\forall$ u,v $\in$ C, $u \nrightarrow v$, $v \nrightarrow u$.

Алгоритм Косарайю

Граф $G^T = (V, E^T)$, $E^T = \{(v, u) \mid (u, v) \in E\}$.

Поиск сильно связных компонент с помощью алгоритма Косарайю (Kosaraju):

1. Выполнить поиск в глубину на графе G для получения порядка вершин как при топологической сортировке
2. Выполнить поиск в глубину на графе G^T, обход вершин в основном цикле выполнять в порядке, полученном на шаге 1
3. Лес деревьев поиска в глубину, полученный на шаге 2 - сильно связные компоненты графа G

```
Kosaraju(G):
      order = DFS(G)
      SCCs = DFS(Transpose(G), order)
      return SCCs

DFS(G, order):
      trees = {}
      for each u in G.V:
            u.visited = false
      for each u in order:
            if not u.visited:
                  add tree from DFS(G, u) to trees
      return trees
```

Временная сложность: O(V + E).

Минимальное остовное дерево

Остовное дерево (spanning tree) неориентированного взвешенного графа G - связный ациклический подграф (дерево), содержащий в себе (связывающий) все вершины G. Минимальное остовное дерево (minimum spanning tree) - остовное дерево с минимальным весом, т.е суммой весов всех входящих в него ребер. Для поиска минимального остовного дерева используются алгоритмы Крускала и Прима.

Алгоритм Крускала

На первом шаге для каждой вершины графа создается отдельное множество из одной вершины. Далее все ребра графа обходятся в порядке неубывания весов. Если вершины ребра принадлежат разным множествам (добавление ребра не создаст цикла), ребро добавляется в остовное дерево, а множества объединяются. Работа алгоритма завершается после просмотра всех ребер, при этом полученное остовное дерево будет минимальным.

```
Kruskal(G, weight):
      MST = {}
      for each u in G.V:
            MakeSet(u)
      for each (u,v) in SortedEdges(G, weight):
            if Set(u) != Set(v):
                  MST.add((u, v))
                  UniteSets(u, v)
      return MST

MakeSet(u):
      make set from one vertex u

Set(u):
      return set to which vertex u belongs

UniteSets(u, v):
      unite sets of vertices u and v into one set

SortedEdges(G, w):
      return edges G.E sorted by weight w in ascending order
```

Временная сложность: O(E*log(V)).

Алгоритм Прима

Построение остовного дерева начинается с некоторой стартовой вершины. На каждом шаге выбирается ребро с минимальным весом, соединяющее вершины, уже находящиеся и еще не находящиеся в дереве, вершина в выбранном ребре добавляется в дерево. Для выбора ребра с минимальным весом используется очередь с приоритетами с атрибутом вершины minWeight в качестве ключа; атрибут вершины minWeight равен весу минимального ребра от нее до вершины в дереве. Работа алгоритма завершается после помещения всех вершин в дерево, полученное остовное дерево будет минимальным; ребра минимального остовного дерева восстанавливаются с помощью аттрибута вершины parent, содержащего смежную в ребре вершину (родителя в дереве).

```
Prim(G, weight, s):
      for each u in G.V:
            u.minWeight = Infinity
            u.parent = None
      s.priority = 0
      Queue = G.V
      while Queue not empty:
            u = Queue.popMin()
```

```
            for each v in G.adj(u):
                  if v in Queue and weight(u, v) < v.minWeight:
                        v.minWeight = weight(u, v)
                        v.parent = u
                        update Queue with new priority of v
      return BuildMST(G, s)

BuildMST(G, s):
      MST = {}
      for each u in G.V - {s}:
            MST.add((u, u.parent))
      return MST
```

Временная сложность: O(E*log(V)).

Кратчайшие пути из одной вершины

Кратчайший путь между двумя вершинами в ориентированном взвешенном графе - последовательность переходов по дугам между вершинами, имеющая минимальную сумму весов дуг (минимальный вес пути).

Алгоритм Дейкстры

Алгоритм находит кратчайшие пути из одной стартовой вершины во все остальные. Предполагается, что веса всех дуг неотрицательны: $\forall(u, v) \in E$: weight(u, v) ≥ 0. В алгоритме строится множество вершин S - вершин, для которых уже вычислены окончательные веса кратчайших к ним путей. На каждом шаге алгоритма выбирается вершина с минимальной оценкой веса кратчайшего пути (атрибут minPathWeight) и добавляется в множество S; для выбора вершины используется очередь с приоритетами с атрибутом minPathWeight в качестве ключа. Для смежных с ней вершин оценка пересчитывается. Работа алгоритма завершается после добавления в S всех вершин графа. Кратчайший путь до каждой вершины можно восстановить с помощью атрибута вершины parent.

```
Dijkstra(G, weight, s):
      Init(G, s)
      S = {}
      Queue = G.V
      while Queue not empty:
            u = Queue.popMin()
            S.add(u)
            for each v in G.adj(u):
                  Relax(u, v, weight)
                  update Queue with new priority of v if necessary
```

```
Init(G, s):
      for each u in G.V:
            u.minPathWeight = Infinity
            u.parent = None
      s.minPathWeight = 0

Relax(u, v, weight):
      if v.minPathWeight > u.minPathWeight + weight(u, v):
            v.minPathWeight = u.minPathWeight + weight(u, v)
            v.parent = u

BuildPath(u):
      current = u
      path = {}
      while current != None:
            path.add((current, current.parent))
            current = current.parent
      return path
```

Временная сложность: $O(E*\log(V))$ или $O(V^2)$, в зависимости от реализации очереди с приоритетами.

Кратчайшие пути между всеми парами вершин

Можно применить алгоритм Дейкстры из каждой вершины или специальный алгоритм Флойда-Уоршелла.

Алгоритм Флойда-Уоршелла

В алгоритме используется матрица весов W: W_{ij} = weight(i, j), $(i,j) \in E$; 0, $i = j$; ∞, $i \neq j$, $(i,j) \notin E$. Алгоритм вычисляет матрицу весов путей между всеми парами вершин. Для нахождения самих путей требуется построение матрицы предшествования P: $P_{ij}^{(0)} = i$, $(i,j) \in E$; $P_{ij}^{(k)} = P_{kj}^{(k-1)}$, $D_{ij}^{(k-1)} > D_{ik}^{(k-1)} + D_{kj}^{(k-1)}$.

```
FloydWarshall(W):
      n = num of rows in W
      D(0) = W
      for k = 1..n:
            D(k) = make new n*n matrix
            for i = 1..n:
                  for j = 1..n:
                        Dij(k) = min(Dij(k-1), Dik(k-1) + Dkj(k-1))
      return D(n)
```

Временная сложность: $O(V^3)$.

Домашнее задание

Требуется реализовать указанный в задании алгоритм, провести тестирование алгоритма на указанном графе/графах. Замерить время работы алгоритма. Сравнить время работы с любой библиотечной реализацией (например, The Boost Graph Library для C++).

Для своей и библиотечной реализации также найти:

- поиск сильно связных компонент: число компонент, минимальный, максимальный и средний размер компонент
- поиск минимального остовного дерева: вес дерева
- поиск кратчайшего пути: минимальные, максимальные и средние вес пути и количество дуг в пути

В отчет включить:

- постановку задачи
- описание использованных алгоритмов и структур данных
- результаты тестов (таблицы, графики)
- выводы
- текст программы включать в отчет не обязательно, но требуется показать работающую программу на занятии

Подключение библиотек в C/C++

Header-only библиотеки (состоят только из заголовочных .h/.hpp файлов):

1. Распаковать/установить заголовочные файлы библиотеки на компьютер.
2. Если заголовочные файлы установлены в нестандартную директорию, добавить путь к ним в опции компилятора: Additional Include Directories в Visual Studio или -I<dir path> (Linux).
3. Добавить требуемые #include для использования библиотеки, собрать проект.

Компилируемые библиотеки:

1. Распаковать/установить файлы библиотеки на компьютер.
2. Скомпилировать библиотеку, если требуется, для получения библиотечных .lib/.dll (Windows) или .a/.so(Linux) файлов.
3. Указать путь к заголовочным файлам (как и с header-only библиотеками).
4. Если библиотечные файлы находятся в нестандартной директории, указать путь к ним в опциях линковщика: Additional Library Directories в Visual Studio или -L<dir path> (Linux).
5. Указать сами библиотеки в опциях линковщика: Additional Dependencies в Visual Studio или -l<library name> (Linux; префикс lib и расширение опускаются).
6. Добавить требуемые #include, собрать проект.

Графы для тестов

Требуемые типы графов:

- поиск сильно связных компонент: ориентированный граф
- поиск минимального остовного дерева: неориентированный взвешенный граф
- поиск кратчайшего пути: ориентированный взвешенный граф

Варианты графов:

1. https://snap.stanford.edu/data/soc-Epinions1.html (ориентированный)
2. https://snap.stanford.edu/data/soc-Slashdot0902.html (ориентированный)
3. https://snap.stanford.edu/data/email-EuAll.html (ориентированный)
4. https://snap.stanford.edu/data/email-Enron.html (неориентированный)
5. https://snap.stanford.edu/data/web-Stanford.html (ориентированный)
6. https://snap.stanford.edu/data/amazon0302.html (ориентированный)
7. https://snap.stanford.edu/data/com-DBLP.html (неориентированный)
8. https://snap.stanford.edu/data/cit-HepPh.html (ориентированный)

Если требуется взвешенный граф, сгенерировать веса для ребер/дуг случайным образом. Для превращения ориентированного графа в неориентированный заменить дуги на ребра (учесть симметрию ребер). Если граф несвязный и требуется связный граф (поиск минимального остовного дерева), сделать граф связным путем добавления дополнительных ребер.

Варианты заданий

Номер варианта для выполнения = свой номер в списке группы % количество вариантов

1. Поиск сильно связных компонент, граф 1.
2. Поиск минимального остовного дерева (алгоритм Крускала), граф 3.
3. Поиск минимального остовного дерева (алгоритм Прима), граф 7.
4. Поиск кратчайших путей между всеми парами вершин (алгоритм Дейкстры), граф 2.
5. Поиск кратчайших путей между всеми парами вершин (алгоритм Флойда-Уоршелла), граф 8.
6. Поиск сильно связных компонент, граф 2.
7. Поиск минимального остовного дерева (алгоритм Крускала), граф 7.
8. Поиск минимального остовного дерева (алгоритм Прима), граф 4.
9. Поиск кратчайших путей между всеми парами вершин (алгоритм Дейкстры), граф 6.
10. Поиск кратчайших путей между всеми парами вершин (алгоритм Флойда-Уоршелла), граф 1.
11. Поиск сильно связных компонент, граф 5.
12. Поиск минимального остовного дерева (алгоритм Крускала), граф 8.

13. Поиск минимального остовного дерева (алгоритм Прима), граф 2.
14. Поиск кратчайших путей между всеми парами вершин (алгоритм Дейкстры), граф 3.
15. Поиск кратчайших путей между всеми парами вершин (алгоритм Флойда-Уоршелла), граф 6.
16. Поиск сильно связных компонент, граф 8.
17. Поиск минимального остовного дерева (алгоритм Крускала), граф 2.
18. Поиск минимального остовного дерева (алгоритм Прима), граф 5.
19. Поиск кратчайших путей между всеми парами вершин (алгоритм Дейкстры), граф 1.
20. Поиск кратчайших путей между всеми парами вершин (алгоритм Флойда-Уоршелла), граф 3.
21. Поиск сильно связных компонент, граф 3.
22. Поиск минимального остовного дерева (алгоритм Крускала), граф 1.
23. Поиск минимального остовного дерева (алгоритм Прима), граф 8.
24. Поиск кратчайших путей между всеми парами вершин (алгоритм Дейкстры), граф 4.
25. Поиск кратчайших путей между всеми парами вершин (алгоритм Флойда-Уоршелла), граф 2.

Контрольные вопросы

1. Виды графов. Представление графов. Основные операции с графами.
2. Поиск в глубину и в ширину на графе.
3. Топологическая сортировка.
4. Сильно связные компоненты. Алгоритм Косарайю.
5. Жадный алгоритм. Определение и примеры.
6. Минимальное остовное дерево. Алгоритм Крускала.
7. Минимальное остовное дерево. Алгоритм Прима.
8. Поиск кратчайшего пути в графе. Алгоритм Дейкстры.
9. Поиск кратчайшего пути между всеми парами вершин. Алгоритм Флойда-Уоршалла.

Литература

1. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: Построение и анализ, 3-е издание:
    - Глава 16, Жадные алгоритмы
    - Часть 6, Алгоритмы для работы с графами, главы 22-25
2. Ахо А., Хопкрофт Д., Ульман Д. Структуры данных и алгоритмы:
    - Глава 6, Ориентированные графы
    - Глава 7, Неориентированные графы
3. Седжвик Р. Фундаментальные алгоритмы на C++:

- Часть 5, Алгоритмы на графах, главы 17-21