Министерство образования и науки Российской Федерации
Национальный исследовательский технологический университет
«МИСиС»
Институт информационных технологий и
автоматизированных систем управления
Кафедра Инженерной Кибернетики

Курсовая работа

по дисциплине «Технологии программирования»

на тему «Поиск оптимальных путей в неориентированном графе»

Выполнил:
студент БПМ-18-1
Кущ А. А.

Проверил:
доцент кафедры ИК, к.т.н.
Полевой Д. В.

2020

Оглавление

Введение

Цель работы

Поиск оптимальных непересекающихся путей в неориентированном графе и перемещение “муравьев” из начальной вершины в конечную.

Постановка задачи

Дан неориентированный граф, состоящий из конечного числа ребер и вершин. Нужно переместить некоторое число “муравьев” их некоторой “начальной” вершины в “конечную” за наименьшее число шагов, при этом нельзя допустить нахождение более одного “муравья” в промежуточной вершине.

Содержание

В данной курсовой работе будут кратко разобраны и реализованы следующие алгоритмы:

- поиск кратчайшего пути: алгоритм Беллмана-Форда и Дейкстры;
- поиск непересекающегося пути: алгоритм Суурбалле и Бхандари;
- перемещения муравьев;
- выбор оптимальных путей.

Теоретическая часть

Алгоритм поиска кратчайшего пути

Задача о кратчайшем пути является одной из классических задач теории графов. Широкое ее изучение началось во второй половине XX века, когда были предложены основные алгоритмы для оптимального решения задачи: алгоритм Дейкстры (1959) [1, 2, 4, 5], алгоритм Беллмана–Форда (1958) [1, 2, 4, 8], алгоритм поиска A* (1968), алгоритм Флойда–Уоршелла (1962), алгоритм Джонсона (1977), алгоритм Ли (1961), а так же несколько других решений для различных видов графов. Наиболее известные и эффективные из них, которые подходят для нахождения кратчайшего пути от одной из вершин графа до другой в текущей задаче – это алгоритм Дейкстры и алгоритм Белмана-Форда. Далее будут кратко разобраны оба алгоритма и выбран самый оптимальный.

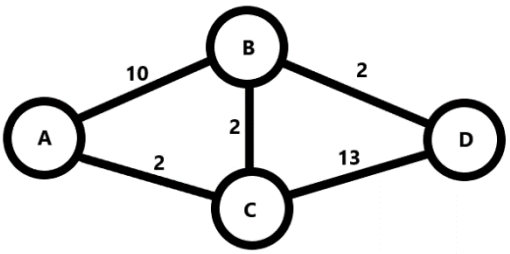
Алгоритм Беллмана — Форда

Данный алгоритм основан на том, что нужно пройти по всем ребрам графа V-1 раз (V – количество вершин), каждый раз пытаясь уменьшить значение кратчайшего пути до вершины. Алгоритм считается универсальным по той причине, что способен работать с ребрами, содержащими отрицательные веса. Стоит заметить, что у данного алгоритма есть одно важное ограничение: ему не поддаются графы с отрицательными циклами, т. е. те, проходя по которым и возвращаясь в исходную точку общий путь уменьшается.

Для большего понимания алгоритма разберем его на примере:

Начальная таблица весов:

A	B	C	D
0	∞	∞	∞



Шаг первый.

Пройдемся по всем вершинам в алфавитном порядке (синий фон обозначает текущую вершину).

№ итерации	A	B	C	D
	0	∞	∞	∞
1	0	10	∞	∞
2	0	10	2	∞
3	0	10	2	15

Шаг второй.

Переносим веса с последнего шага и снова запускаем алгоритм.

№ итерации	A	B	C	D
	0	10	2	15
1	0	4	2	15
2	0	4	2	15
3	0	4	2	6

Далее в таблице ничего меняться не будет на протяжении $\boldsymbol{V-3}$ раз (2 раза уже прошлись), т. к. самый короткий путь уже найден, и его вес составляет 14 единиц.
Сложность алгоритма составляет $\boldsymbol{O(VE)}$, где V – количество вершин, E – количество ребер.

А что, если останавливать алгоритм раньше в случаях неизменности весов за один шаг?

Модификация алгоритма Беллмана — Форда

Так зародилась идея модификации алгоритма. Можно значительно ускорить работу алгоритма, если завершать его выполнение при отсутствии изменении по окончании итерации.

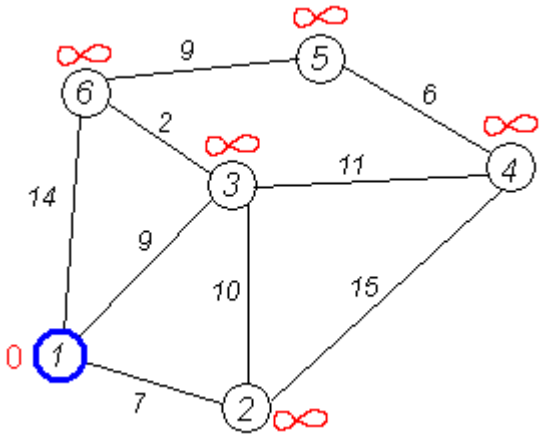
Помимо этого, была предложена идея, которая позволяет алгоритму обнаружить цикл отрицательного веса в графе. Для этого количество итераций в алгоритме не ограничивалось на количестве вершин в графе, уменьшенном на единицу, и проводилась еще одна дополнительная итерация. Соответственно, если на этой последней итерации веса определенных вершин обновились, то кратчайшего пути до них не существует из-за наличия отрицательного цикла на пути к ним.

Алгоритм Дейкстры

Данный алгоритм основан на фиксировании минимального пути до вершины.
Пошагово проходя по всем вершинам, запоминается минимальное значение расстояния до этой вершины. Чтобы достичь этого, изначальное расстояние начальной точки считается нулем, а все непосещенные вершины принимаются за бесконечности. Наиболее популярная реализация алгоритма использует список, в который помещаются соседние непосещенные вершины.

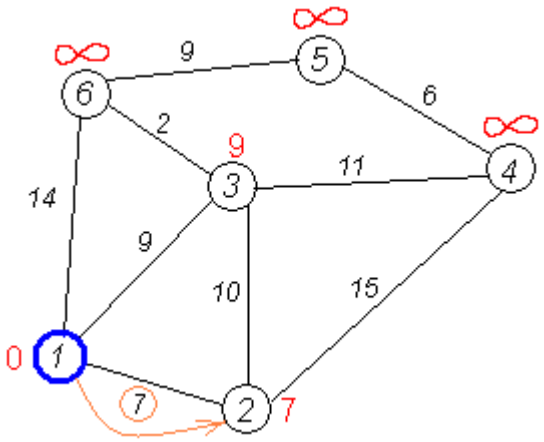
<u>Не применим к графу с отрицательными ребрами.</u>

Для большего понимания алгоритма разберем его на примере:



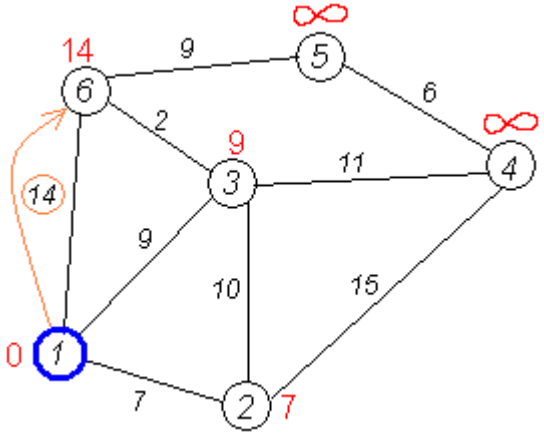
Первый шаг.

У вершины 1 минимальный вес. Начнем с неё. Ее соседи – 2, 3, 6.

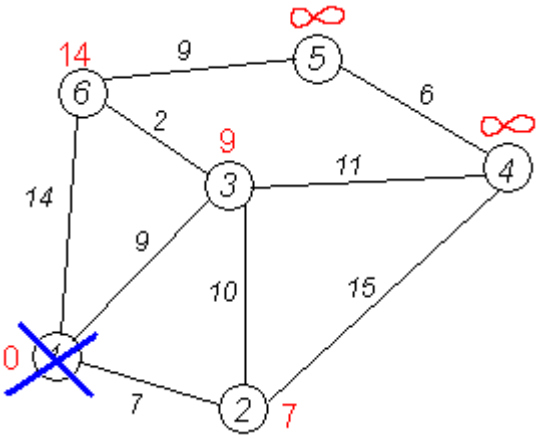


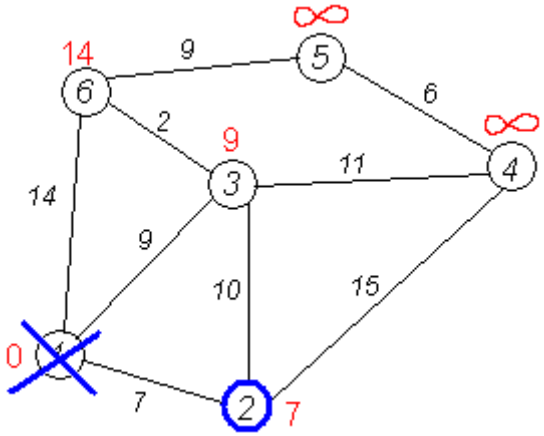
Первая по очереди вершина — 2, потому что вес ребра минимален, и она является соседом.
Длина пути через равна сумме веса в вершине 1 и веса ребра, следовательно, 0+ 7 = 7.
Это меньше текущего веса вершины 2, поэтому вес перезаписывается.

Аналогичную операцию проделываем с другими соседями вершины 1 — 3 и 6.



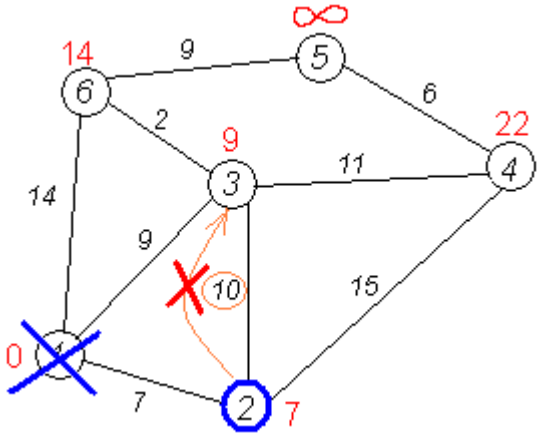
Все соседи вершины 1 проверены, исключаем их.
Вершину 1 мы больше не имеем права рассматривать и пересчитывать в ней веса.
Вычеркнем вершину из графа, ибо эта вершина посещена.



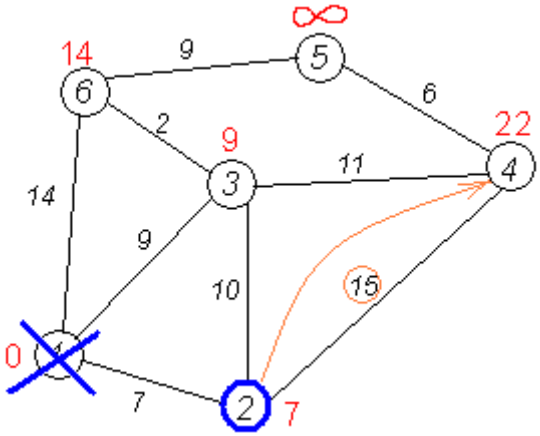


Второй шаг.

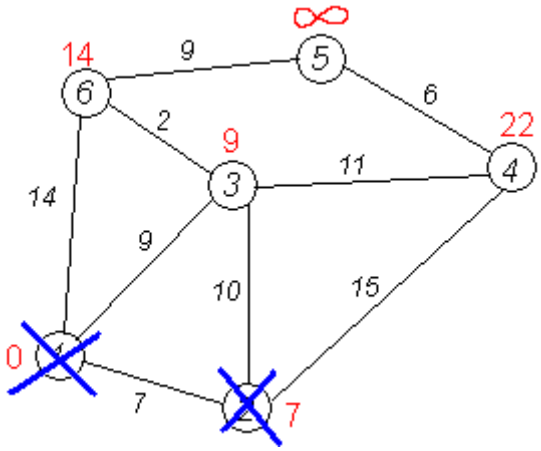
Снова находим «ближайшую» вершину из непосещенных.
Это вершина 2 с весом 7.
Снова пытаемся уменьшить вес соседа выбранной вершины, пытаясь пройти в них через вершину 2.
Соседями вершины 2 являются вершины 1, 3, 4.
Первый сосед вершины 2 — вершина 1. Но она уже была посещена, поэтому 1 вершину не рассматриваем.



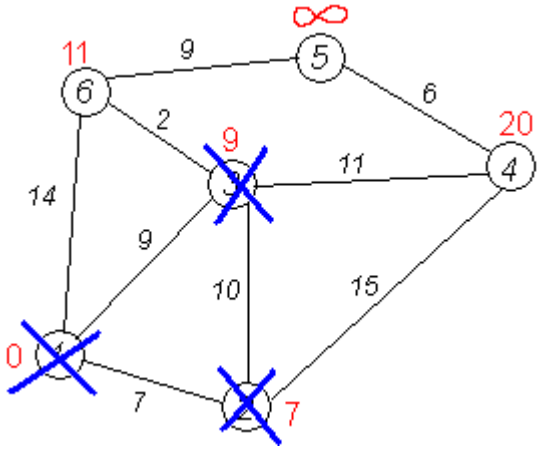
Следующий сосед — вершина 3, так как она имеет минимальный вес ребра после 1.
Если идти в неё через 2 вершину, длина пути будет равна 17. Но текущий вес 3 вершины равен 9, а 9 < 17, поэтому вес остается прежним. Идем далее.



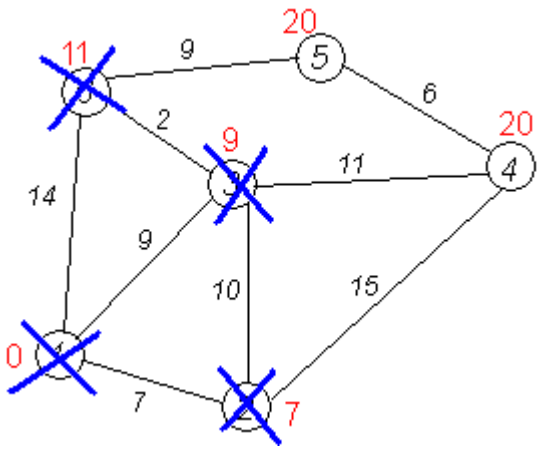
Ещё один сосед вершины 2 — это вершина 4.
Если идти в неё через 2, длина пути будет равна сумме веса в вершине 2 и веса ребра, соединяющего 2 и 4 вершины, то есть 7 + 15 = 22.
Поскольку 22 < ∞, устанавливаем вес вершины 4 равной 22 и идем далее.

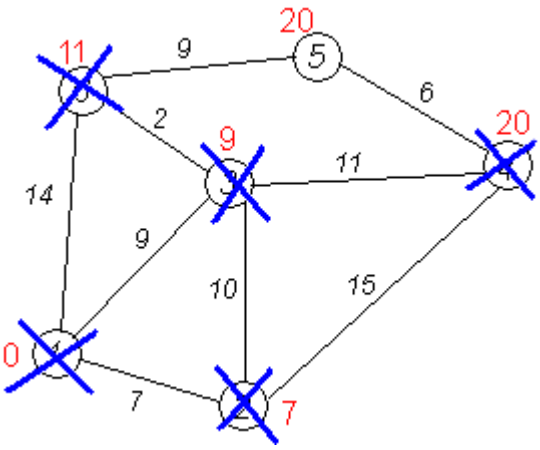


Все соседи вершины 2 просмотрены, замораживаем расстояние до неё и помечаем её как посещённую.



Третий шаг.
Повторяем шаг алгоритма, выбрав вершину 3. После её «обработки» получим такие результаты:





Как видим, кратчайший путь из вершины 1 в 5 прошёл через вершины 3, 6. Вес его составил 20 единиц. Если непосещённые вершины помечены бесконечностью, то это значит, до этих вершин нельзя добраться.
Сложность алгоритма составляет $\boldsymbol{O(V^2)}$, где V – количество вершин.
С использованием кучи Фибоначчи возможно улучшить время до $\boldsymbol{O(Vlog(V))}$.

Обоснование выбора

В качестве основного алгоритма был выбран алгоритм Беллмана-Форда в силу своей универсальности.

Кроме этого, были проведены собственные тесты алгоритмов на разных картах: 4 карты, по 5 тестов на каждой. Для сравнения был использован основной код, но упрощён: после нахождения кратчайшего пути выполнение программы прерывалось.

Результаты представлены в таблице ниже. Время замерялось в секундах. Все тесты проводились на одном оборудовании при одинаковой температуре.

Algorithm		test_0	test_1	test_2	test_3
Bellman-Ford	1	0,36690	275,33910	0,24090	38,87630
	2	0,37360	292,84790	0,24180	42,26852
	3	0,36800	NULL	0,02304	NULL
	4	0,36570	NULL	0,23170	NULL
	5	0,36790	NULL	0,23440	NULL
	res	**0,36842**	**284,09350**	**0,19437**	**40,57241**
Bellman-Ford (modified)	1	0,00790	0,04320	0,00096	0,01690
	2	0,00780	0,03760	0,00094	0,01700
	3	0,00800	0,03560	0,00091	0,01700
	4	0,00790	0,03530	0,00095	0,01710
	5	0,00790	0,03750	0,00094	0,01730
	res	**0,00790**	**0,03784**	**0,00094**	**0,01706**
Dijkstra	1	0,00100	0,28000	0,00093	0,07180
	2	0,00094	0,28490	0,00092	0,07170
	3	0,00100	0,28380	0,00093	0,07260
	4	0,00098	0,29410	0,00092	0,07120
	5	0,00093	0,28390	0,00096	0,07490
	res	**0,00097**	**0,28534**	**0,00093**	**0,07244**
edges		3545	39825	2045	19973
rooms		1500	25000	1000	10000

Проанализируем таблицу:
Алгоритм Беллмана-Форда работает медленно, что полностью исключает целесообразность его использования.

Модифицированный алгоритм Беллмана-Форда, включающий в себя раннюю остановку в случае неизменности весов на протяжении цикла, по скорости в большинстве случаев сопоставим с алгоритмом Дейкстры. Модифицированный Беллман-Форд проявил себя лучше в задачах с большим количеством вершин, а Дейкстры – в задачах с малым количеством вершин.

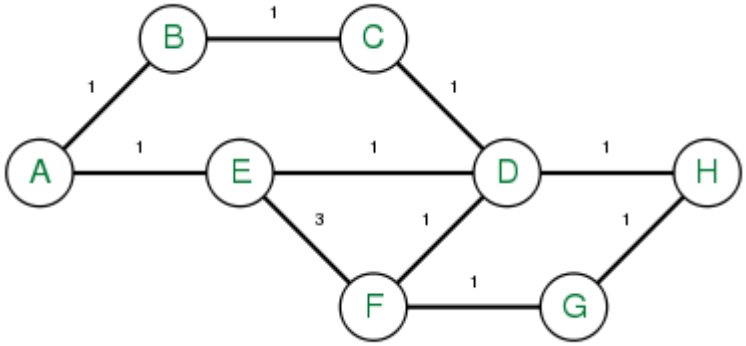
Алгоритм поиска непересекающегося пути

Алгоритм поиска непересекающегося пути подразумевает в себе использование одного из вышеупомянутых алгоритмов поиска кратчайшего пути. Проще говоря, сначала находится самый короткий путь, после чего он “изолируется” с помощью алгоритма Суурбалле [3, 7] или Бхандари [7]. Действия повторяются до тех пор, пока не закончатся варианты новых путей.

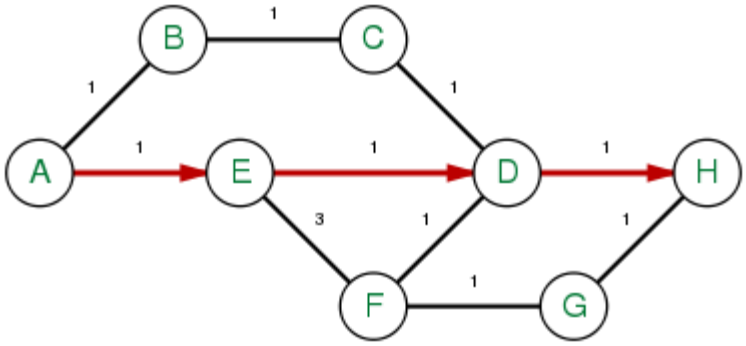
Алгоритм Суурбалле

Суурбалле придумал свой алгоритм и опубликовал его в 1974 году. Он находит k (2 или более) реберно-непересекающихся путей в графе. Стоимость всех путей минимальна. Данный алгоритм более требователен, чем алгоритм Бхандари. Он не позволяет двум путям использовать один и тот же узел при пересечении. За счет этого алгоритм работает всегда безошибочно, но более ресурсозатратно.

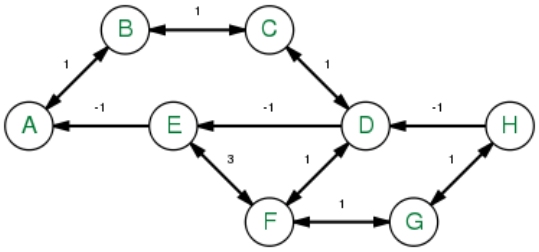
Изучим на примере работу алгоритма:

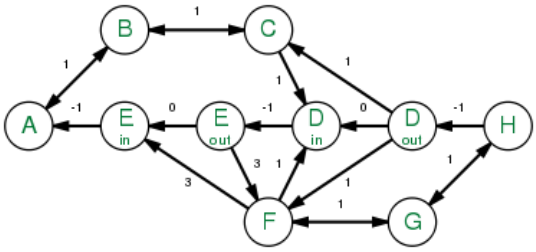


Шаг первый.
Найдем самый короткий путь.



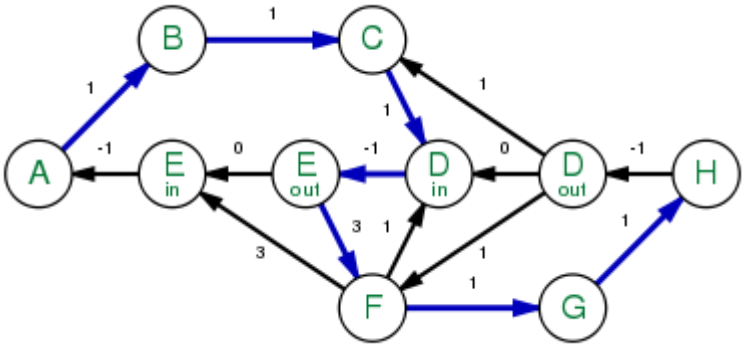
Шаг второй.
Делаем граф ориентированным, разворачивая путь в обратную сторону и меняя веса на отрицательные. Кроме этого, дублируем промежуточные вершины и соединяем их ориентированными ребрами с весом 0. Таким образом, мы установим единственный вариант следования алгоритма при последующих проходах. Важно: начальную и конечную вершину не трогаем.





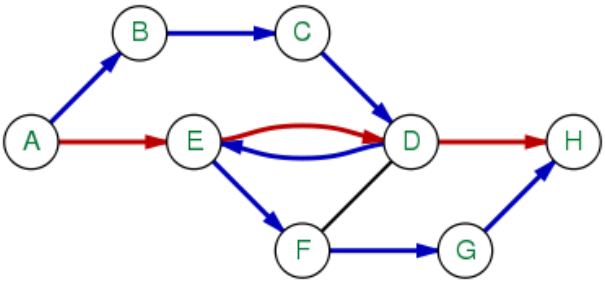
Шаг третий.

Запускаем алгоритм поиска ближайшего пути до тех пор, пока не найдутся все пути.



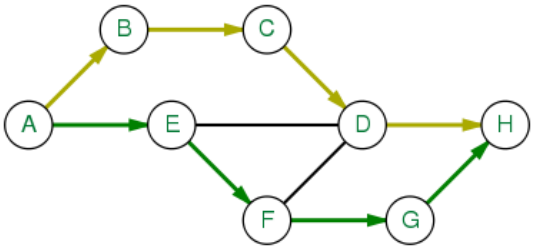
Шаг четвертый.

Изобразим все пути на графе.



Шаг пятый.

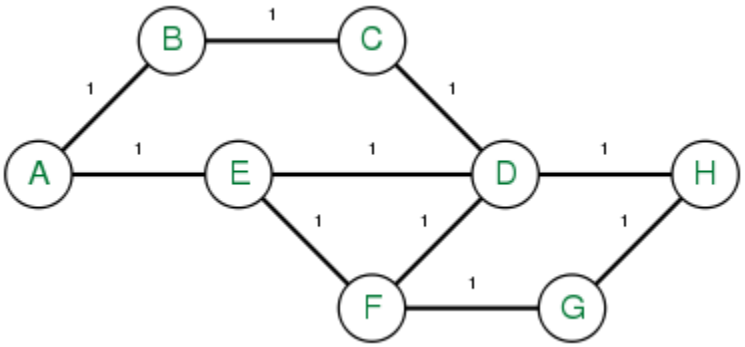
Удалим все дублирующиеся ребра и получим конечный результат.



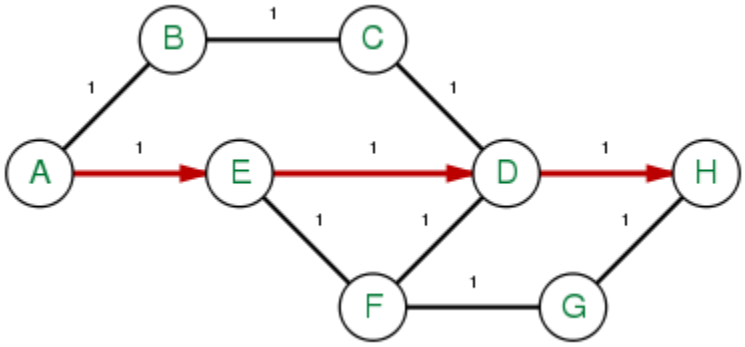
Алгоритм Бхандари

Рамеш Бхандари опубликовал свой алгоритм в книге “Survivable Networks: Algorithms for Diverse Routing” (1999 г.) Он основан на оригинальном алгоритме Суурбалле, но прост в реализации, занимает меньше памяти и работает быстрее. Он также находит k (2 или более) реберно-непересекающихся путей в графе. Стоимость всех путей минимальна.

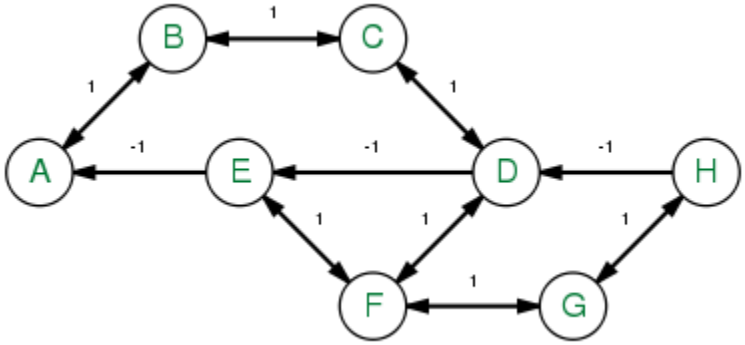
Рассмотрим на примере работу алгоритма:



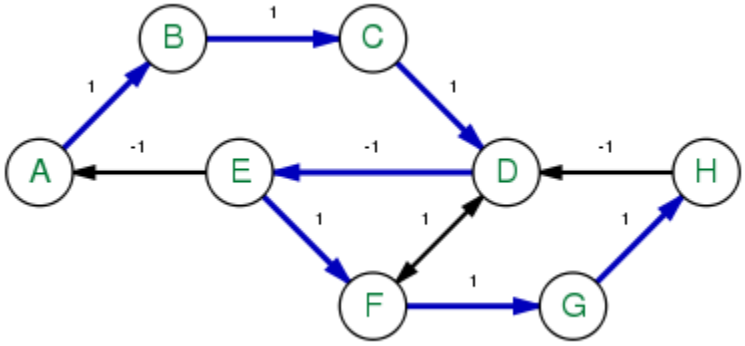
Шаг первый.
Найдем самый короткий путь.



Шаг второй.
Делаем граф ориентированным, разворачивая путь в обратную сторону и меняя веса на отрицательные. Если некоторые ребра уже были ориентированными, то просто выбрасываем их.

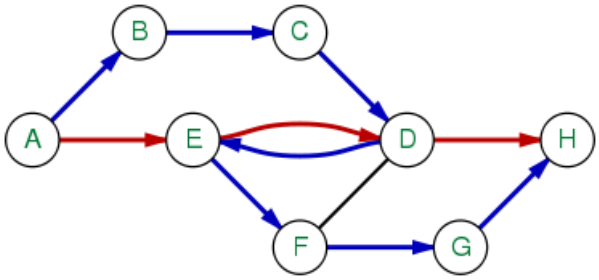


Шаг третий.
Проделываем шаг 1 и 2 (пока необходимо).



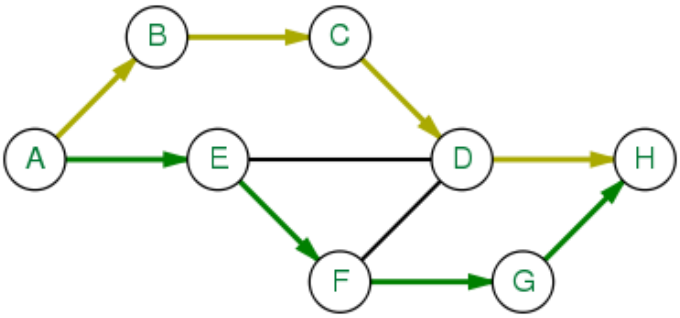
Когда больше нет возможности найти путь, переходим к следующему шагу.

Шаг четвертый.
Изобразим все пути на графе.



Шаг пятый.

Удалим все дублирующиеся ребра и получим конечный результат.



Стоит отметить, что у данного алгоритма есть один очень серьезный минус: иногда он работает некорректно с весами в случае пересекающихся путей, как было рассмотрено выше.

Обоснование выбора

Сравнивая два алгоритма, очевидное преимущество одерживает идея Суурбалле в силу своей универсальности.

Оптимальный выбор путей и “путешествие” муравьев

Особенность отправки муравьев на каждом шаге

Теперь мы знаем, как получить кратчайшие непересекающиеся пути. Но этого недостаточно. Если отправлять муравьев по всем подряд путям, то могут возникнуть серьезные проблемы. Для этого нам нужно проверять каждый путь перед отправкой в него нового муравья, а затем смотреть, не будет ли этот путь ухудшать наш алгоритм, добавляя лишние шаги. Данное опасение справедливо в том случае, если в графе нашлось более 1 непересекающегося пути различной длины.

Алгоритм “Правильной отправки муравьев”:

Суть алгоритма заключается в проходе по всем путям графа и проверки выполнения условия:

$$\sum_{i=1,i\neq m}^{m-1} (L_m - L_i) \leq N, m \in M,$$

где N – количество муравьев, M – все непересекающиеся пути, L – длина пути, m – текущий рассматриваемый путь.

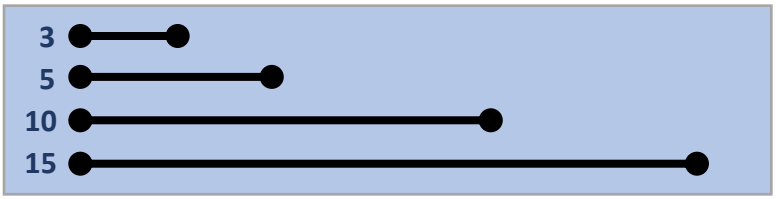
Если условие выполнилось успешно, переходим к следующему пути и вычитаем одного муравья:

$$N = N - 1$$

Для большей ясности рассмотрим пример.

Допустим, есть 4 пути разной длины, начинающихся из общей стартовой вершины и приходящих в общую конечную вершину. Задача: переместить $N = 14$ муравьев за меньшее число шагов.

Первым делом, отсортируем все пути в порядке возрастания длины для удобства и возможности раннего прерывания шага в случае первого невыполнения условия.

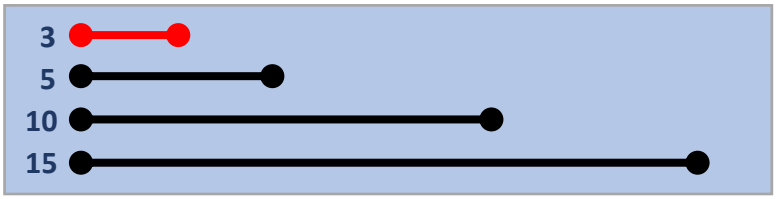


Шаг первый.

Выбираем первый путь. Вычтем из рассматриваемого пути длину предыдущих путей и сложим:

$$0 \leq N = 14$$

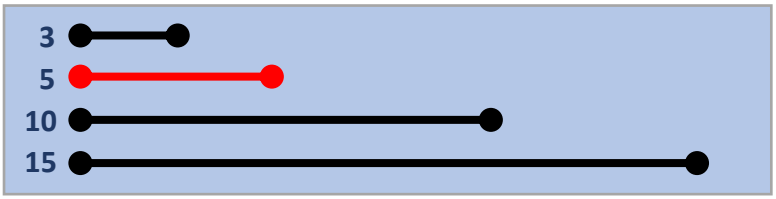
Условие выполняется, отправляем на этот путь нового муравья. Теперь муравьев $N - 1 = 13$.



Выбираем второй путь. Вычтем из рассматриваемого пути длину предыдущих путей и сложим:

$$(5-3)=2\leq N=13$$

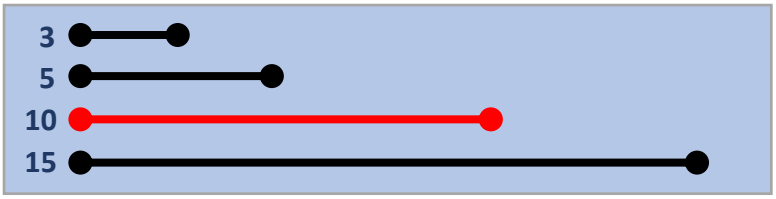
Условие выполняется, отправляем на этот путь нового муравья. Теперь муравьев $N-1=12$



Выбираем третий путь. Вычтем из рассматриваемого пути длину предыдущих путей и сложим:

$$(10-5)+(10-3)=12\leq N=12$$

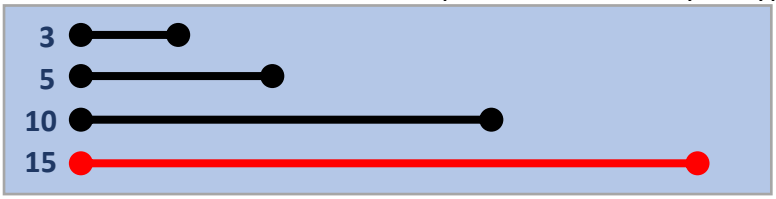
Условие выполняется, отправляем на этот путь нового муравья. Теперь муравьев $N-1=11$



Выбираем третий путь. Вычтем из рассматриваемого пути длину предыдущих путей и сложим:

$$(15-10)+(15-5)+(15-3)=27>N=11$$

Условие не выполняется, не отправляем на этот путь муравья. Муравьев $N=11$



Первый шаг завершен.

Шаг второй.

Повторяем вышеперечисленные действия до тех пор, пока не закончатся муравьи.

Последующие шаги…

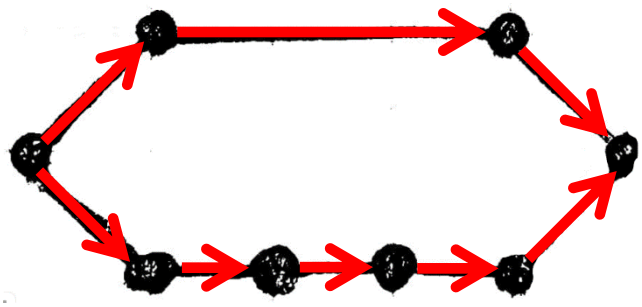
После того, как закончатся муравьи в стартовой вершине, останется только выполнить шаги для перемещения оставшихся муравьев в пути до конечной вершины.

Оптимальный выбор путей

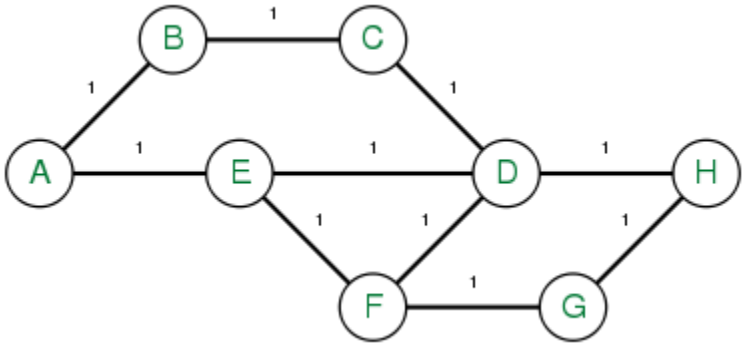
Немаловажную роль в скорости и корректности работы программы играет правильное использование непересекающихся путей.

В тривиальном случае, когда во время работы алгоритма поиска непересекающегося пути при наложении решений не возникает дублирующихся ребер, проблем также не возникает.

Сбоку представлен пример такого графа.



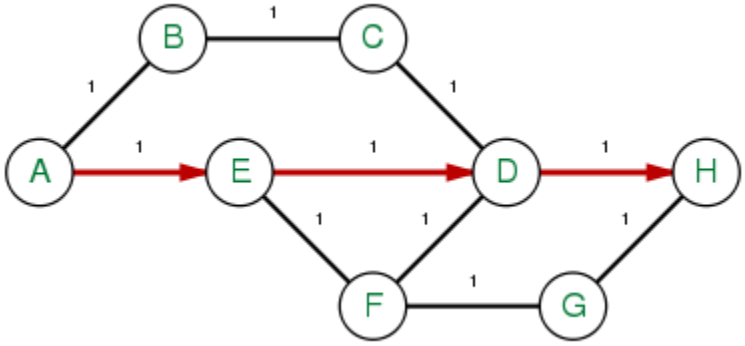
В ином случае все интереснее и сложнее. Снова рассмотрим на примере для наглядности уже знакомый граф:



Первый путь, который найдет алгоритм, будет самым коротким:

$$A - E - D - H \ (4 \text{ вершины})$$

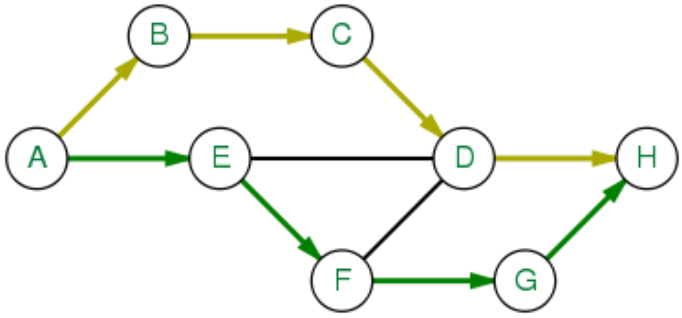
Выделим его на графе:



Если запустить алгоритм дальше, то как было рассмотрено в предыдущих параграфах, в итоге получится два пути, каждый из которых будет длиннее первого пути:

$$\left.\begin{matrix} A - B - C - D - H \\ A - E - D - G - H \end{matrix}\right\} \ (5 \text{ вершин})$$

Изобразим их на графе:



Возникает закономерный вопрос: всегда ли большее количество более длинных путей лучше меньшего количества путей, но более коротких?

Чтобы получить ответ на данный вопрос, нужно применить алгоритм “Правильной отправки муравьев”, рассмотренный выше. С помощью него можно посчитать, при каком наборе путей получается наименьшее число шагов.

Практическая часть

Классы, структуры и функции

Интерфейсы проекта

Класс ***Room*** – хранит вершины, их параметры и связи.
Класс ***Neigh*** – хранит параметры связи.
Класс ***Solutions*** – хранит временные и лучшие пути.
Класс ***FileReader*** – хранит считанную информацию из файла.
Класс ***Lemin*** – наследует класс Solutions, хранит информацию о комнатах, муравьях и связях.

```
class Room;
class Neigh;
class Solutions;
class Lemin;
class FileReader;
```

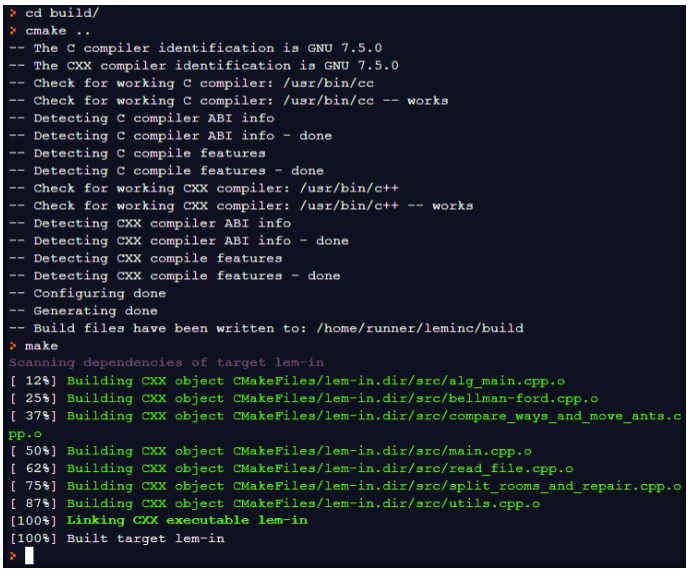
Инструкция по сборке

Исходный код

https://github.com/artemk1337/lem-in_cpp

Компиляция проекта

Компиляция проекта осуществляется с помощью утилиты cmake.



Создание карты (графа)

Карта должна соответствовать определенному формату:

- → Отсутствие пустых строк;
- → Ненулевое количество муравьев;
- → Комментарий начинается с “#”;
- → Идентификация стартовой и конечной вершины начинается с помощью ##start и ##end соответственно;
- → При указании комнаты (вершины) обязательно нужны координаты; в дальнейшем можно визуализировать или применить на реальных задачах;
- → Строгий порядок ввода данных: количество муравьев, все комнаты (вершины), все ребра;
- → Вершины не могут иметь одинаковое название, а ребра не должны повторяться;
  *напомним, что ребра не ориентированы, поэтому порядок соединения вершин не важен, пр.: A-B эквивалентно B-A.

Примеры валидных карт:

```
# Это комментарий
# Ниже задается количество муравьев
100
# Ниже инициализируются вершины с координатами
##start
0 0 0
1 0 0
2 0 0
3 0 0
4 0 0
##end
5 0 0
# Ниже инициализируются ребра
0-1
1-2
2-3
3-4
4-5
```

```
10
##start
start 0 0
1 0 0
b 0 0
c 0 0
4 0 0
##end
finish 0 0
start-1
1-a
a-b
b-c
c-finish
a-finish
c-start
```

Запуск программы

1) Компилируем проект с помощью утилиты cmake.
2) Создаем карту на подобии примера выше или берем готовую. Генератор карт присутствует в проекте, но работает только на Mac OS.
3) Запуск осуществляется через терминал в следующем виде:

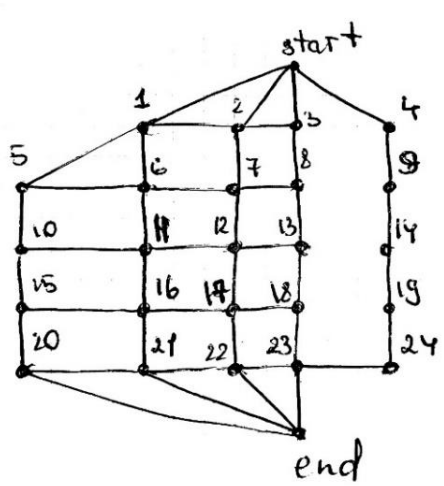
>> ./lem-in <название карты>

Формат вывода программы (разделитель – пустая строка):

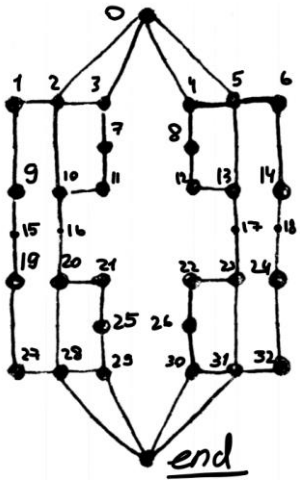
1) Данные из файла
2) Найденные пути
3) Перемещение муравьев: L<номер муравья>-<название комнаты> … |step <номер шага>

В папках *“maps_***”* находятся карты для тестирования.
Карты, с которыми возникло больше всего трудностей во время реализации проекта.

Карта ***gemerald4***:



Карта ***super_hard***:



Список литературы

1. Перепелица В. А., Тебуева Ф. Б. «Дискретная оптимизация и моделирование в условиях неопределенности данных» - 3.2. - “Алгоритмы нахождения кратчайшего пути (цепи) между вершинами графа”.
2. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein «Introduction To Algorithms» - 24.1 - стр. 589
3. Suurballe J. W. «Disjoint paths in a network» - T.4. - стр. 125-145
4. https://ru.wikipedia.org/wiki/Задача_о_кратчайшем_пути
5. https://ru.wikipedia.org/wiki/Алгоритм_Дейкстры
6. https://habr.com/ru/company/otus/blog/484382/
7. http://www.macfreek.nl/memory/Disjoint_Path_Finding
8. https://ru.wikipedia.org/wiki/Алгоритм_Беллмана_—_Форда