Государственное образовательное учреждение

высшего профессионального образования

«Нижегородский государственный университет

им. Н.И.Лобачевского»

Институт информационных технологий математики и механики

Кафедра информатики и автоматизации научных исследований

**Курсовая работа   
на тему:  
“Нахождение максимального пути в графе”**

**Выполнил:**

студент 3 курса гр.381507-2

Роман Даниил Валерьевич

Нижний Новгород

2018 г.

**СОДЕРЖАНИЕ**

Введение3

Цель4

Задачи4

Методы нахождения максимаьного пути в графе4

Генетический алгоритм5

Примеры входных данных5

Начальная популяция6

Мутация10

Селекция12

Результаты13

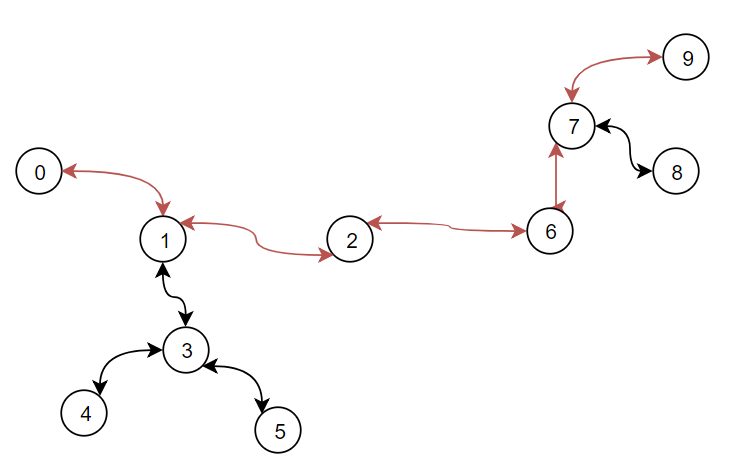
Заключение13

Используемая литература14

# **Введение**

# В теории графов нахождение максимального пути в графе означает нахождение простого пути максимальной длины.

Простой путь – это путь в графе, не имеющий повторяющихся вершин.



Длина пути это колличество ребер в простом пути в данном случаи длина равна 5(ребра составляющие простой путь в графе выделены красным цветом).

Впредь нахождение масксимального пути в графе будем называть проблемой максимального пути.

Проблема максимального пути это NP-трудная задача, что озачает что она не может быть разрешен за полиномиальное время.

Задача называется *NP*-*трудной*если каждая задача из *NP*полиномиально сводится к ней.

*NP-*труднаязадача имеет тот смысл, что эта задача не проще, чем «самая трудная в *NP*».

Однако в направленном ациклическом графе (DAG графе) можно найти решение за полиномиальное время.

Но мы будем рассматривать неориентированный ациклический граф. Для него нет алгоритма, который находит максимальный путь за полиномиальное время, поэтому будем находить приближенный максимальный путь.

# **Цель**

Найти максимальный путь в не ориентированным ациклическом графе.

# **Задачи**

* Изучить методы приближонного нахождения максимального пути в не ориентированном графе
* Реализовать метод нахождения максимального пути в графе

**Методы нахождения максимаьного пути в графе**

Основным способом нахождения максимального пути в графе является применение алгоритма Дейкстры. Алгоритм дейкстры находит минимальный путь в графе, но если заменить все веса в графе на отрицательные значения. Например, если ребро графа между вершинами A и B имеет вес 20, то мы заменяем его на -20 в том графе, на которым и будем производить поиск. Таким образом алгоритм Дейкстры даст нам минимальный путь, который на нашем изначальном графе будет являться максимальным.

Но у такого подхода есть существенный минус. Для большинства графов (назовем их G) такое преобразования создает в графе -G цикл отрицательной длинны.

Для нахождения приблизительного решения на графах очень часто приходит на ум алгоритмы на основе стохастики. Я выбрал для себя раздел эволюционных алгоритмов – генетические алгоритмы и решил с их помощью найти приблтженный максимальный путь в графе.

**Генетичесие алгоритмы**

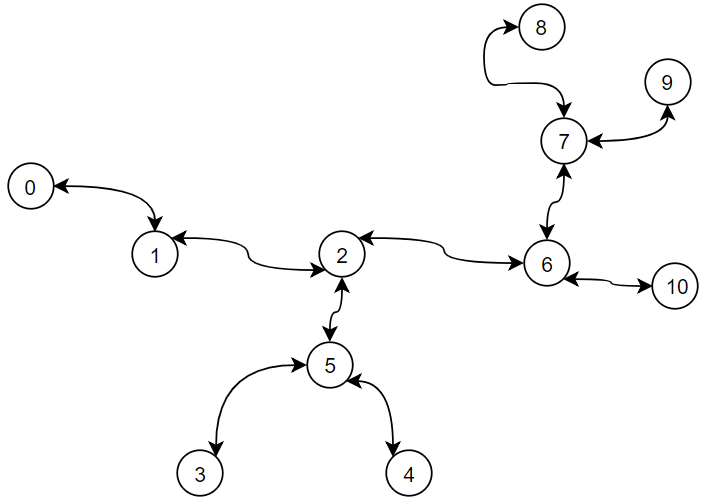
В компьютерных науках генетический алгоритм - это метаэвристический алгоритм который являедся подклассом более широкого класса эвалюционных алгоритмов.

Метаэвристика - это эвристика более высокого уровня, предназначенная для поиска, генерации или выбора эвристического алгоритма(частичного поиска), который может обеспечить достаточно хорошее решение проблемы оптимизации, особенно с неполной или несовершенной информацией или ограниченной вычислительной мощности, но она не гарантирует, что глобальное решение может быть найдено по некоторому классу проблем.

Класс генетических алгоритмов был создан под впечатлением от процессов естественного отбора. Генетические алгоритмы в основном используются для оптимизации и используют такие операторы отбора как мутация, кроссовер и селекция.

**Пример входных данных**

На данном этапе я буду рассматривать не взвешенный не ориентированный ациклический граф.



Граф я буду представлять в виде матрицы смежности.

Здесь “1” в ячейке 1, 0 означает связь, что на графе есть ребро между вершиной 1 и 0, а “0” в свою очередь означает обратное. Так как я рассматриваю не ориенированный граф, то матрица получается симетричной и при проходе по одному пути в лгоритме нужно не забыть удалить и путь в обратном направлении.

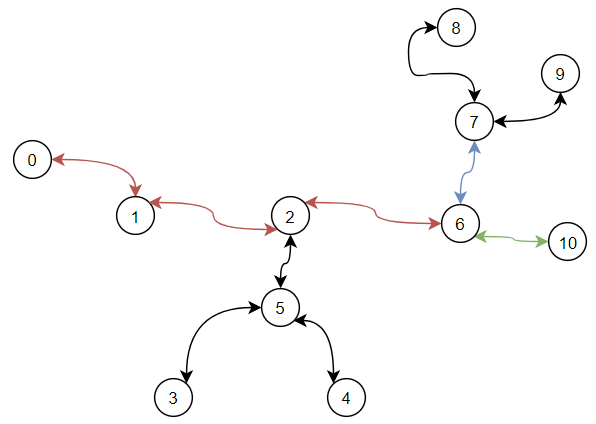
|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
| **0** | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **1** | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **2** | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| **3** | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| **4** | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| **5** | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| **6** | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| **7** | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| **8** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| **9** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| **10** | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

Для генетического алгоритма необходима начальная популяция. В нашем случаи это различные простые пути графа. Их можно составить рандомно, но это не дает не самый хороший результат. А для того, чтобы алгоритм сходился нам нужна хорошая начальная популяция из как можно более длинных простых путей.

Поэтому рандомно выбирается лишь первая точка от которой будет строиться путь, а каждая следующая выберается с определенной вероятностю.

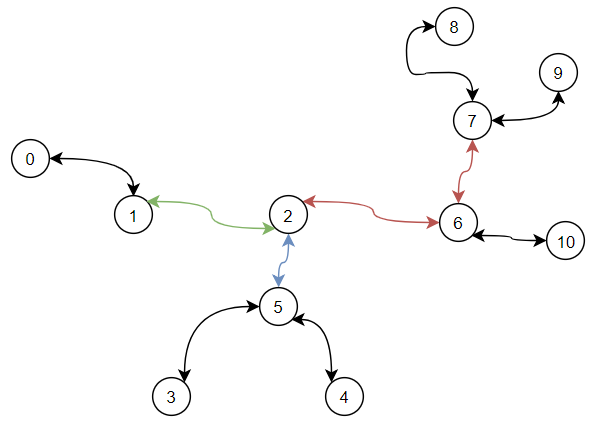
Вероятность перехода в следующую вершину из возможный определяется количеством вершин в которую можно перейти из этой новой вершины.

Выглядит это следующим образом:



Красным выделен тот путь, который мы уже совершили и сейчас мы находимся в вершине под номером 6. У нас есть два варианта долнейшего развития событий: вершина 10 или вершина вершина 7. Из 10 вершины мы не сможем дальше пройти никуда кроме как вернуться, а возвращаться мы не можем, поэтому это плохое решение для дальнейшего пути. А вот из вершины 7 мы сможем попасть аж в две другие вершины. Таким образом в даннном случаи получается так, что вершину 7 мы выберем с вероятностью 1.

Рассмотрим еще один случай:



В данном случаи мы находимся в вершине 2 и мы выберем вершину 5 для дальнейшего следования с вероятностью 0.66, а вершину 1 с вероятностью 0.33

В коде это выглядит следующим образом:

Метод «getNextVertex» возвращает нам вершину которую мы прибавим к нашему пути. Ему на вход мы подаем список всех возможных вершин ,в которые мы можем перейти т.е. «nextVertices». В нашем примере это получается 5 и 1. И список количества дальнейших путей из каждой веришны списка «nextVertices» в нашем случаи это 2 для вершины 5 и 1 для вершины 1. Получаем их методом «getAgrees». Далее мы строим массив вероятностей следующим образом. Мы добавляем туда вершину столько раз, какой число ей соответсвует в массиве «agrees». Т.е. в массиве «probability» число 5, которое обозначает 5ую вершину будет содержаться 2 раза, так как из 5 вершины мы можем попасть в 2 другие, а число 1 в массиве будет содержаться лишь один раз. И наконец мы рандомно выбираем число из этого массива и добавляем полученную вершину в наш путь. Использование функции рандома с равномерным распределением гарантирует то, что неважно на каком месте в массиве расположены вершины в начале или в конце, вероятность их выбора зависит лишь от их колличества в массиве, а не от расположения.

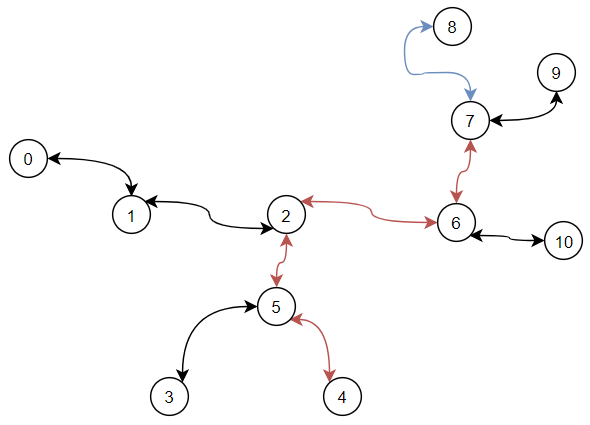
1. **public** **static** Integer getNextVertex(List<Integer> nextVertices, List<Integer> agrees) {
2. ArrayList<Integer> probability = **new** ArrayList<Integer>();
3. **for** (**int** j = 0; j < nextVertices.size(); j++) {
4. **for** (**int** k = 0; k < agrees.get(j); k++) {
5. probability.add(nextVertices.get(j));
6. }
7. }
8. Random random = **new** Random();
9. **int** i = random.nextInt(probability.size());
10. **return** probability.get(i);
11. }
13. **public** **static** List<Integer> getAgrees(List<List<Integer>> a,
14. List<Integer> nextVertices) {
15. List<Integer> agrees = **new** ArrayList<Integer>();
16. **for** (Integer nextVertex : nextVertices) {
17. **int** sum = 0;
18. **for** (**int** j = 0; j < a.size(); j++) {
19. sum += a.get(nextVertex).get(j);
20. }
21. agrees.add(sum);
22. }
23. **return** agrees;
24. }

Когда больше нельзя добавить вершину в наш путь мы не останавливаемся, а смотрим на первую вершину, от которой начали строить наш путь. И если из нее мы можем двигаться в другое направление, т.е. из первой вершины количество дальнейших вариантов пути больше единицы, то мы строим еще один путь. И сравниваем их.

Метод, который возвращает нам список возможных вершин для дальнейшего построения пути называется «getNextVertices», на вход подается матрица смежности графа и текущая вершина для которой нам интересно узнать следующие возможные вершины.

1. **public** **static** List<Integer> getNextVertices(List<List<Integer>> a, **int** currentVertex) {
2. List<Integer> array = a.get(currentVertex);
3. List<Integer> nextPath = **new** ArrayList<Integer>();
4. **for** (**int** j = 0; j < a.size(); j++) {
5. **if** (array.get(j) == 1) {
6. nextPath.add(j);
7. }
8. }
9. **return** nextPath;
10. }

Например, если мы начинаем строить путь из вершины 7, то можем получить два пути:

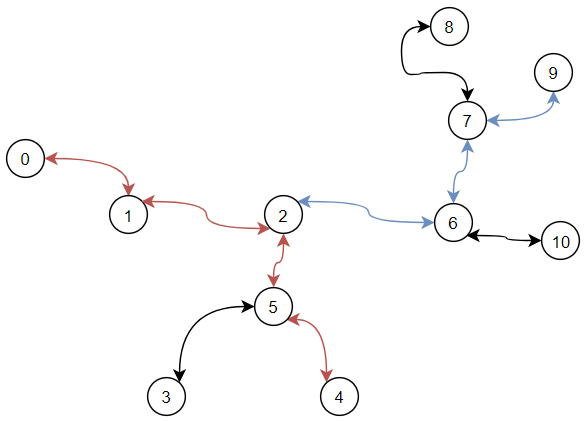


Они выделены красным и синем. Мы выбираем наиболее длинный из них и заносим его в начальную популяцию.

В моем варианте генетического алгоритма для нахождения максимального пути в графе я не буду использовать кроссовер. Для создания новых потомков я буду оперировать лишь оператором мутации.

Для этого я прохожусь по всем путям в популяции. В каждом пути я генерирую значение возмущения. Это велечина, которая определяет вершину в текущем родительском пути от которой мы будем строить нового потомка.

Например:



У нас есть путь 4->5->2->1->0. Мы получили значение возмущения равное 2 это велечина соответсвует вершине под номером 2. От нее мы строим новый путь по тому же принципу, что и в начальной популяции. И в результате новый потомок имеет путь 4->5->2->6->7->9. Потом мы образуем еще одного потомка, но в этот раз мы идем не от начала пути родителя, а от его конца, т.е. просто делаем реверс предка. Таким образом на одного предка у нас получается по два потомка такую мытацию мы называем «успешной». Не на каждом пути из популяции возможно произвести мутацию например значение возмущения может ссылаться на точку, которая имеет меньше 3 вариантов следующих вершин, тогда мы не генерируем значение возмущения снова, а переходим к следующему пути в популяции и такая мутация не считается успешной. Количество успешных мутаций контролируется и мы подаем это значение на вход алгоритму.

В коде это выглядит следующим образом:

1. **public** **static** List<List<Integer>> applyMutation(List<List<Integer>> a,
2. List<List<Integer>> parents, **int** mutationCount) **throws** FileNotFoundException {
3. List<List<Integer>> children = **new** ArrayList<>();
4. List<Integer> tmpParent, child1,child2;
5. **for**(List<Integer> parent: parents){
6. tmpParent = parent;
7. child1 = getChildren(a,parent);
8. Collections.reverse(tmpParent);
9. child2 = getChildren(a, tmpParent);
10. **if**(child1 == **null** || child2 == **null**) {**continue**;}
11. children.add(child1);
12. children.add(child2);
13. mutationCount--;
14. **if**(mutationCount<=0){**break**;}
15. }
16. **return** children;
17. }
19. **private** **static** List<Integer> getChildren(
20. List<List<Integer>> a, List<Integer> parent) **throws** FileNotFoundException {
21. List<List<Integer>> tmpMatrixA = **new** ArrayList<>(a);
22. List<Integer> child = **new** ArrayList<Integer>();
24. **int** pressureValue = **new** Random().nextInt(Math.round(parent.size()/2));
25. **for**(**int** i=0;i<=pressureValue;i++){
26. child.add(parent.get(i));
27. }
28. **if**(pressureValue>0){
29. **for** (**int** i=1;i<=pressureValue+1;i++){
30. MatrixTool.deletePath(tmpMatrixA,parent.get(i-1),parent.get(i));
31. }
32. }
33. **int** currentVertex = parent.get(pressureValue);
34. List<Integer> nextVertices = MatrixTool.getNextVertices(tmpMatrixA,currentVertex);
35. **if**(nextVertices.size()<3 || pressureValue==0){**return** **null**;}
36. **int** preventVertex = parent.get(pressureValue-1);
37. nextVertices.remove(parent.get(pressureValue+1));
38. nextVertices.remove(parent.get(pressureValue-1));
39. List<Integer> availableVertices = nextVertices;
40. **int** nextVertex;
41. **while** (**true**) {
42. **if** (availableVertices.isEmpty()) {
43. **break**;
44. }
45. List<Integer> agrees = MatrixTool.getAgrees(tmpMatrixA, availableVertices);
46. nextVertex = availableVertices.size() > 1 ?
47. MatrixTool.getNextVertex(availableVertices, agrees) : availableVertices.get(0);
48. child.add(nextVertex);
49. currentVertex = nextVertex;
50. MatrixTool.deletePath(tmpMatrixA, preventVertex, currentVertex);
51. preventVertex = nextVertex;
52. availableVertices = MatrixTool.getNextVertices(tmpMatrixA, nextVertex);
53. }
54. **return** child;
55. }

Теперь нам нужно привести нашу разросшуюся популяцию к исходному количеству особей. Для этого мы используем селекцию. Я использую элитарную стратегию т.е. задается процент лучших особей, который точно перейдет в следующее поколение.

Остальная часть популяции отбирается с помощью колеса рулетки (или фитнес-пропорцилнального выбора). Это один из возможных вариантов селекции. Для каждой особи определяется вероятность ее выбора следующим образом:

 где  это фитнес функции или ее приспособнотсь, в нашем случаи это длина пути.

В коде это выглядит следующим образом:

1. **private** **static** List<List<Integer>> rouletteSelect
2. (List<List<Integer>> currentPopulation, **int** count) {
4. List<List<Integer>> selectedPart = **new** ArrayList<>();
5. **for** (**int** i = 0; i < count; i++) {
6. **double** weight\_sum = 0;
7. **for** (List<Integer> path : currentPopulation) {
8. weight\_sum += path.size();
9. }
10. **double** value = **new** Random().nextDouble() \* weight\_sum;
11. **int** removeIndex = -1;
12. **for** (List<Integer> path : currentPopulation) {
14. removeIndex++;
15. value -= path.size();
16. **if** (value < 0) {
17. selectedPart.add(path);
18. currentPopulation.remove(removeIndex);
19. **break**;
20. }
21. }
22. **return** selectedPart;
23. }

Таким образом я реализовал алгоритм поиска максимального пути в графе при помощи генетического алгоритма. Рассмотрим результаты некоторых запусков:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Best path** | **Result path** | **Time(ms)** | **Population Size** | **Elite %** | **Mutation count** | **Generation count** |
| 5 | 5 | 869 | 80 | 25 | 20 | 40 |
| 5 | 5 | 611 | 40 | 25 | 5 | 20 |
| 5 | 5 | 500 | 20 | 25 | 5 | 10 |
| 13 | 11 | 1569 | 80 | 25 | 20 | 40 |
| 13 | 12 | 1567 | 80 | 25 | 20 | 40 |
| 13 | 12 | 1629 | 100 | 25 | 40 | 80 |
| 13 | 13 | 1473 | 80 | 25 | 40 | 150 |

**Заключение:**

На не больших графах алгоритм отрабатывает быстро и находит самый длинный путь почти при любых входных параметрых. На более существенных графах алгоритм находит близкое значение к самому длинному пути в графе, но редко получает самый длинный путь. Я выяснил, что размер популяции и колличество успешных мутаций слабее влияют на итоговый результат нежели количество популяций или же это можно назвать колличеством итераций алгоритма, что вполне логично. Чем большее количество раз мы применили нашу генетическую схему тем более вероятно получить наилучшее решение. Алгоритм не паказал все на что он способен и я знаю, какие моменты еще нужно доработать и улучшить чем и займусь в дальнейшем.

# **Используемая литература.**

1. A Study of Genetic Algorithms for Approximating the Longest Path in Generic Graphs (David Portugal)
2. Approximating Longest Path(Björklund, Andreas)
3. Algorithms(Robert Sedgewick and Kevin Wayne)