**Санкт-Петербургский государственный УНИВЕРСИТЕТ**

**Факультет прикладной математики и процессов управления**

**отчет**

**по лабораторной работе №2**

**по дисциплине «Алгоритмы и структуры данных»**

**на тему «Алгоритм отжига»**

|  |  |  |
| --- | --- | --- |
| Студент гр. 22Б16 |  | Шувалов Ф.В. |
| Преподаватель |  | Дик А.Г. |

**Санкт-Петербург**

**2023 г.**

**Оглавление**

[1. Цель работы 3](#_Toc163604686)

[3. Введение 3](#_Toc163604687)

[4. Теоретическая часть 3](#_Toc163604688)

[5. Схема выполнения алгоритма 4](#_Toc163604689)

[7. Представление программы на языке программирования 6](#_Toc163604690)

[8. Раскрытие смысла работы алгоритма 6](#_Toc163604691)

[9. Контрольный пример 8](#_Toc163604692)

[10. Результаты тестирования 10](#_Toc163604693)

[11. Модификация 10](#_Toc163604694)

[12. Анализ результатов тестирования 10](#_Toc163604695)

[13. Вывод 11](#_Toc163604696)

[14. Литература 11](#_Toc163604697)

# **Цель работы**

Цель работы – исследование особенностей решения задачи о коммивояжере с помощью алгоритма отжига.

1. **Задачи**
2. Необходимо формализовать задачу о коммивояжере с помощью

алгоритма отжига.

1. Подготовить контрольный пример, используя взвешенный орграф
2. Найти кратчайший гамильтонов цикл.

# **Введение**

Название «Задача о коммивояжере» устойчиво закрепилось за одной из самых интересных, практически значимых и одновременно сложных задач теории графов. Задача, берущая свое начало из работ Гамильтона, состоит в определении кратчайшего гамильтонова цикла в графе. Ее решение связано с решением задачи о назначениях и с задачей об остове наименьшего веса.

Исследование алгоритма отжига для задачи нахождения гамильтонова цикла имеет важное значение, так как этот алгоритм представляет собой мощный метод оптимизации, способный находить приближенные решения сложных задач комбинаторной оптимизации. Понимание его работы, свойств и ограничений может быть полезным для разработки более эффективных методов решения задач, таких как маршрутизация в компьютерных сетях или оптимизация планирования маршрутов для транспортных средств, где требуется нахождение оптимальных или приближенных решений..

# **Теоретическая часть**

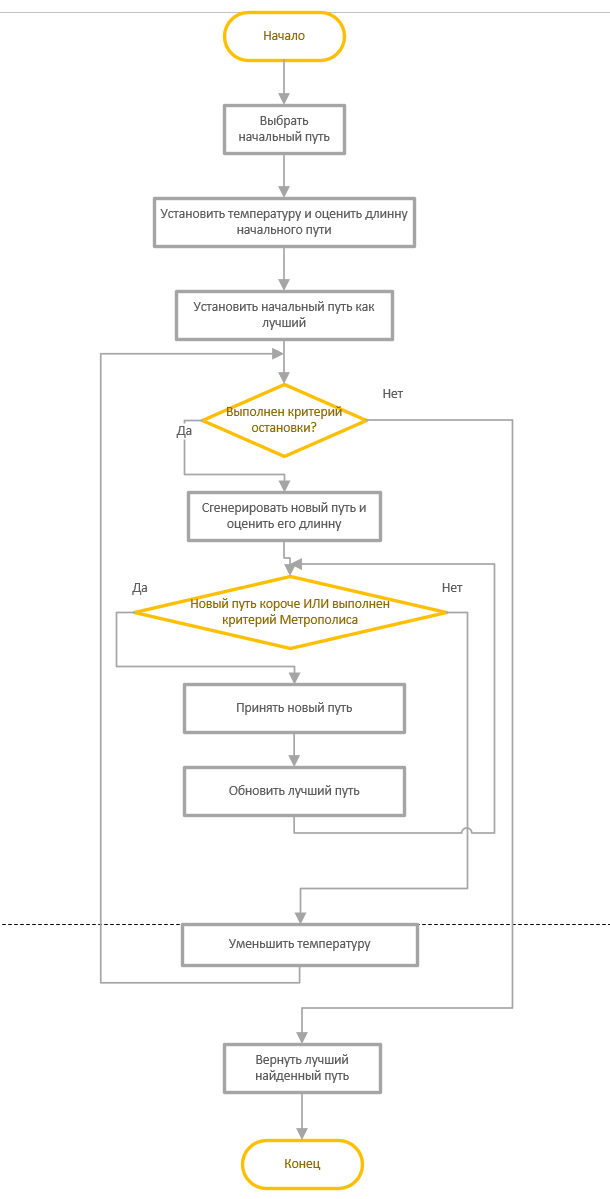
**Необходимые термины:**

Вот список терминов, связанных с алгоритмом отжига для решения задачи о нахождении гамильтонова цикла:

* 1. **Гамильтонов цикл**: Замкнутый путь в графе, который проходит через каждую вершину ровно один раз и возвращается в начальную вершину.
  2. **Алгоритм отжига (Simulated Annealing Algorithm)**: Метаэвристический алгоритм оптимизации, который моделирует процесс отжига металла и использует случайные изменения и вероятностные принятия решений для поиска приближенного оптимального решения задачи.
  3. **Граф**: Математическая структура, состоящая из вершин (узлов) и рёбер (связей), которые соединяют эти вершины.
  4. **Вершина графа**: Один из узлов или точек в графе.
  5. **Ребро графа**: Связь между двумя вершинами в графе.
  6. **Температура (Temperature):** Параметр алгоритма отжига, который управляет вероятностью принятия худшего решения для избежания застревания в локальном оптимуме.
  7. **Энергия (Energy):** Функция, которая оценивает качество текущего решения. В контексте алгоритма отжига, это может быть длина гамильтонова цикла или другая мера оптимальности решения.
  8. **Принятие решения (Decision Making):** Процесс выбора следующего состояния системы на основе текущего состояния, температуры и изменения энергии. В алгоритме отжига это происходит с использованием вероятностного критерия Метрополиса
  9. **Критерий Метрополиса (Metropolis Criterion):** правило, которое определяет вероятность принятия нового состояния системы на основе изменения энергии и текущей температуры. Согласно критерию Метрополиса, если изменение энергии отрицательное (то есть новое состояние лучше), то оно принимается всегда. В случае положительного изменения энергии (худшее состояние), оно принимается с вероятностью, которая уменьшается с увеличением температуры. Это позволяет алгоритму отжига временно принимать худшие решения, чтобы избежать застревания в локальном оптимуме и исследовать пространство решений более полно.

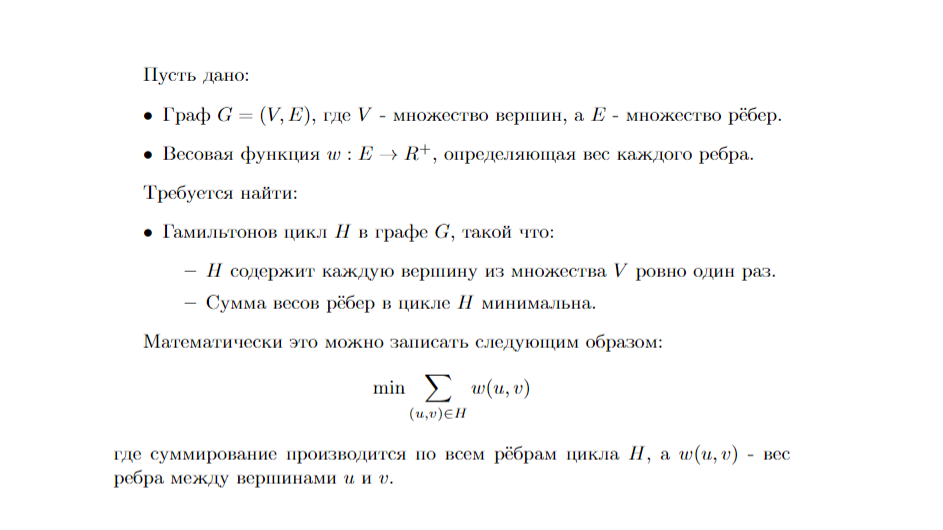
# **Схема выполнения алгоритма**

На рисунке 5.1 представлена блок-схема алгоритма.



*Рисунок 5.1 Блок-схема алгоритма*

1. **Формализация задачи**



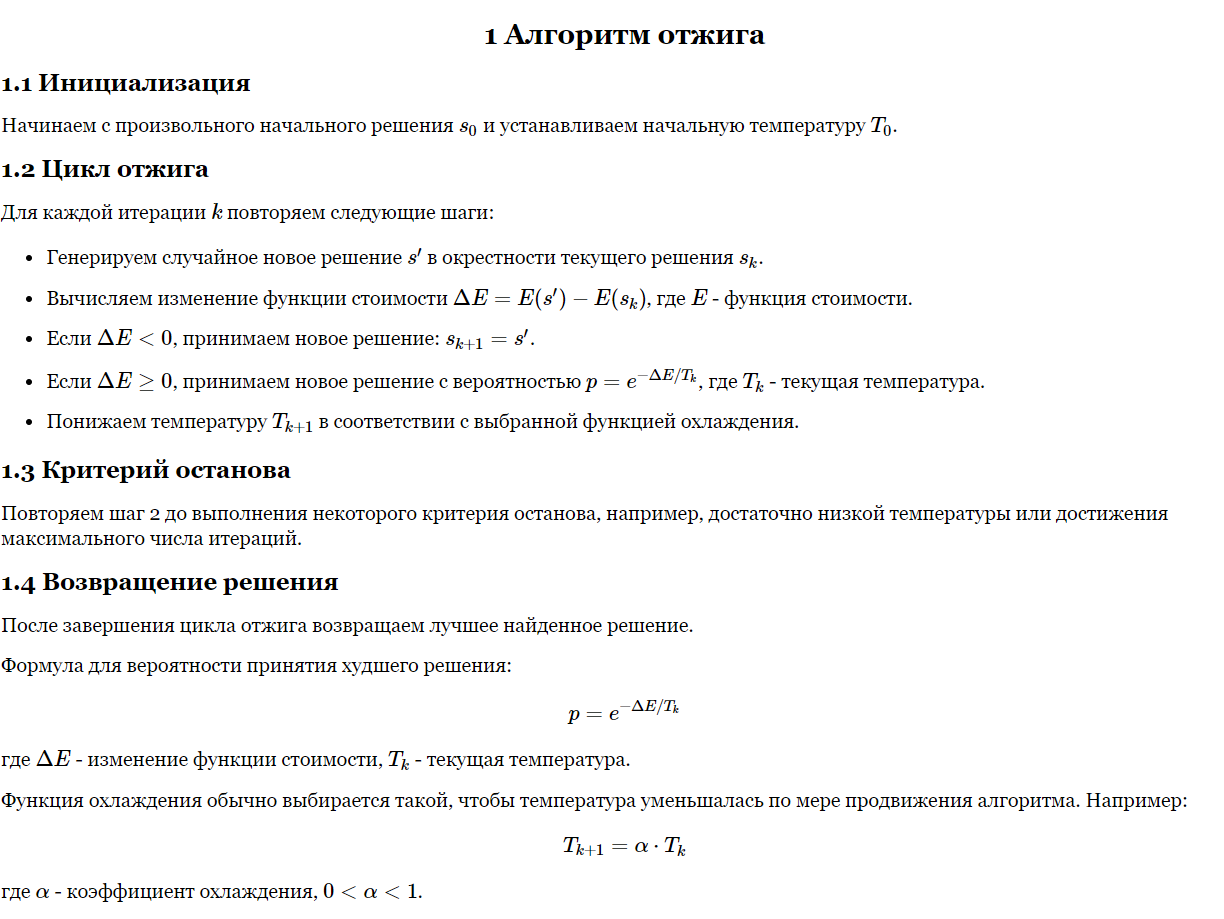
*Рисунок 6.1 Формализация задачи*

# **Представление программы на языке программирования**

Алгоритм имитации отжига реализован на выбранном языке программирования (Python3) с использованием объектно-ориентированного подхода. Программа состоит из двух основных классов: `SA` и `MSA`. Класс `SA` представляет реализацию алгоритма имитации отжига, а класс ` MSA ` представляет его модификацию.

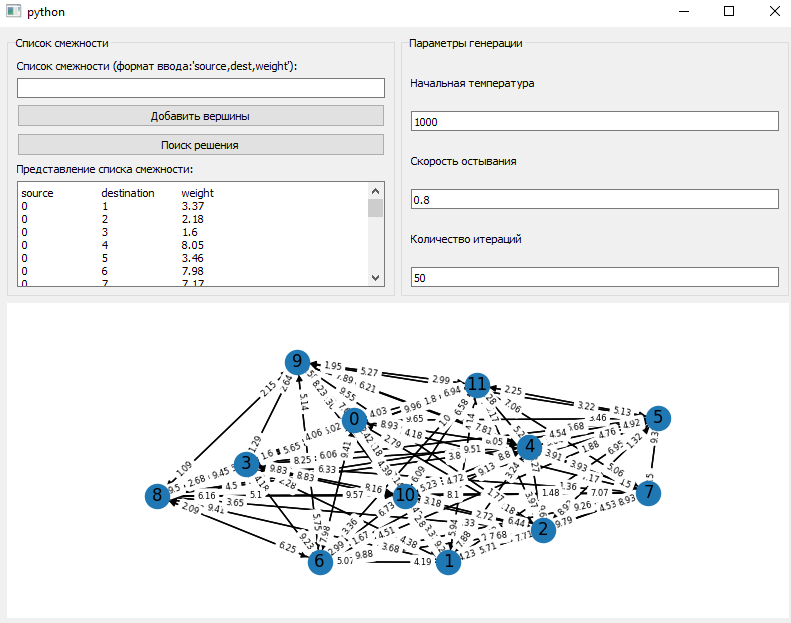
# **Раскрытие смысла работы алгоритма**

Алгоритм отжига является методом глобальной оптимизации, вдохновленным процессом отжига металлов. В этом алгоритме начальное решение выбирается случайным образом, а затем на каждом шаге предлагается новое решение, которое может быть принято или отклонено в зависимости от изменения функции стоимости и текущей "температуры". В начале процесса температура высока, что позволяет алгоритму принимать худшие решения, но по мере продвижения процесса температура постепенно уменьшается, что уменьшает вероятность принятия худших решений. Это позволяет избежать застревания в локальных оптимумах и исследовать большее пространство решений. Алгоритм отжига находит применение в широком спектре задач оптимизации, включая задачи коммивояжера, планирование производства, дизайн и другие. Более формальное описание алгоритма представлено на рисунке 8.1.



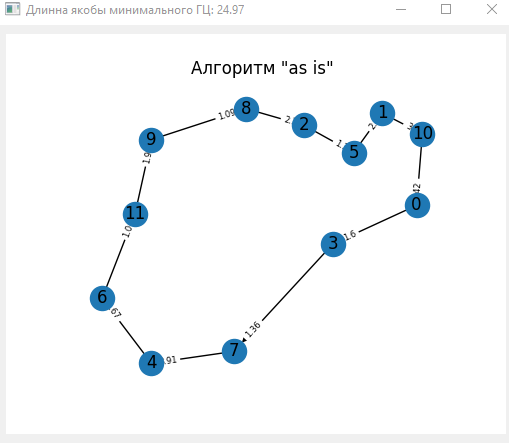
*Рисунок 8.1 Формальное описание алгоритма отжига*

# **Контрольный пример**



*Рисунок 10.1 Визуализация программы*

На рисунке 10.1 изображен пользовательский интерфейс программы. В левой части окна пользователь может вводить параметры, такие как: *граф, начальная температура, скорость остывания, количество итераций.*

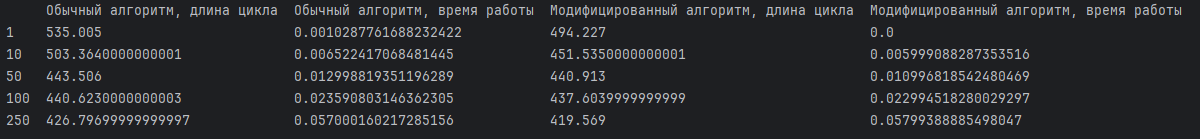
****

*Рисунок 10.2 результат работы алгоритма*

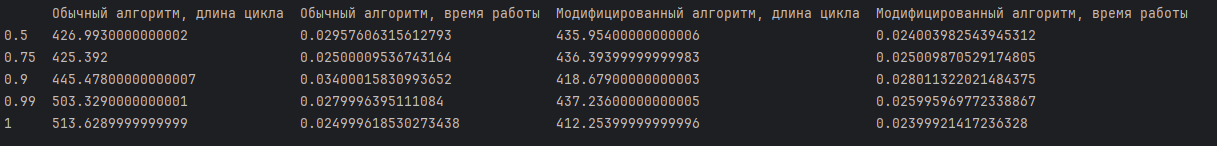
На рисунке 10.2 изображен результат работы алгоритма.

1. Результаты тестирования

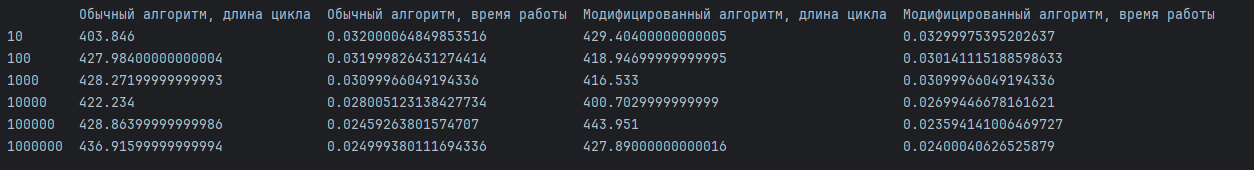
На рисунках 10.1-10.5 приведены результаты тестирования алгоритма.



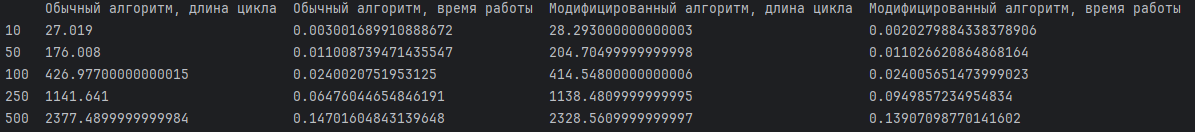
*Рисунок 10.1 Результаты тестирования (количество итераций как переменная)*



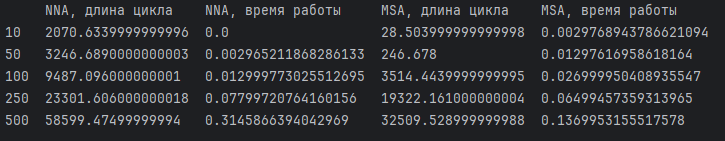
*Рисунок 10.2 Результаты тестирования (скорость охлаждения как переменная)*

**

*Рисунок 10.3 Результаты тестирования (начальная температура как переменная)*

**

*Рисунок 10.4 Результаты тестирования (количество вершин в графе как переменная)*

**

*Рисунок 10.4 Результаты тестирования (количество вершин в графе как переменная), сравнительная таблица*

1. Модификация

В данной лабораторной работе была реализована также модификация алгоритма отжига, включающая в себя два ключевых аспекта:

1. Жадная эвристика для начального решения (greedy\_initial\_solution): В этой модификации используется жадная эвристика для выбора начального пути. Этот метод помогает быстро получить исходное решение, которое может быть использовано в качестве отправной точки для алгоритма отжига.
2. Адаптивное управление температурой (adaptive\_cooling): Вместо использования фиксированного коэффициента охлаждения температуры, алгоритм изменяет температуру на каждой итерации согласно адаптивной схеме. Это позволяет более эффективно исследовать пространство состояний в различных фазах выполнения алгоритма отжига.

Вместе эти два аспекта обеспечивают более эффективный поиск гамильтонова цикла в графе. Жадная эвристика помогает ускорить начальный этап поиска, а адаптивное управление температурой помогает более эффективно настраивать процесс отжига, что может привести к нахождению более качественных решений.

1. Анализ результатов тестирования

Оптимальные параметры для алгоритма, полученные по результатам тестирования отображены на рисунке 12.1.



*Рисунок 12.1 Оптимальные параметры для алгоритма отжига*

Анализ результатов тестирования выявил следующие ключевые особенности алгоритма SA:

1. Незначительное ухудшение скорости сходимости при увеличении размера (количества ребер и вершин) графа; для полного графа зависимость времени работы от числа вершин приблизительно O(n).
2. Увеличение количества итераций не оказывает значительного эффекта на улучшение скорости сходимости.
3. Алгоритм SA значительно превосходит по точности алгоритм NNA
4. Алгоритм SA превосходит алгоритм NNA по скорости работы в полном графе при увеличении количества вершин.

# **Вывод**

При выполнении данной работы были получены все необходимые навыки и знания по реализации алгоритма отжига. В ходе реализации задачи была создана программа для визуализации его работы.

# **Литература**

* + 1. Статья, описывающая принципы работы SA [Электронный ресурс] - URL:  [https://algorithmica.org/ru/annealing](https://habr.com/ru/articles/105639/)(дата обращения: 03.03.2024)
    2. Документация библиотеки PyQT [Электронный ресурс] - URL: <https://doc.qt.io/qtforpython-6/> (дата обращения: 25.09.2023)

1. **Листинг**

import networkx as nx  
import random  
import math  
  
  
class SA:  
 def \_\_init\_\_(self, graph, initial\_temperature=1000, cooling\_rate=0.95):  
 self.graph = graph  
 self.current\_path = list(self.graph.nodes())  
 self.best\_path = list(self.graph.nodes())  
 self.best\_path\_length = self.calculate\_path\_length(self.best\_path)  
 self.temperature = initial\_temperature  
 self.cooling\_rate = cooling\_rate  
  
 def run(self, iterations: int) -> list:  
 for \_ in range(iterations):  
 self.iteration()  
 return self.best\_path  
  
 def iteration(self):  
 # Перемешиваем текущий путь  
 random.shuffle(self.current\_path)  
 # Оцениваем длину текущего пути  
 current\_path\_length = self.calculate\_path\_length(self.current\_path)  
 # Оцениваем длину лучшего пути  
 best\_path\_length = self.calculate\_path\_length(self.best\_path)  
  
 # Если текущий путь лучше или имеет вероятность принятия по критерию Метрополиса  
 if current\_path\_length < best\_path\_length or random.random() < math.exp(  
 -(current\_path\_length - best\_path\_length) / self.temperature):  
 self.best\_path = self.current\_path[:]  
 self.best\_path\_length = current\_path\_length  
  
 # Уменьшаем температуру  
 self.temperature \*= self.cooling\_rate  
  
 def calculate\_path\_length(self, path):  
 length = 0  
 for i in range(len(path) - 1):  
 length += self.graph[path[i]][path[i + 1]]['weight']  
 return length  
  
 def to\_DiGraph(self):  
 return self.convert\_path\_to\_graph(self.best\_path)  
  
 def convert\_path\_to\_graph(self, path):  
 path\_graph = nx.DiGraph()  
 if len(path) == 0:  
 return path\_graph  
 if path[0] != path[-1]:  
 path.append(path[0])  
 for i in range(len(path) - 1):  
 source = path[i]  
 target = path[i + 1]  
 weight = self.graph[source][target]['weight'] # Получаем вес ребра между текущей и следующей вершинами  
 path\_graph.add\_edge(source, target, weight=weight)  
 # Добавляем ребро между последней и первой вершинами, чтобы закрыть цикл  
 return path\_graph  
  
  
class MSA(SA):  
 def \_\_init\_\_(self, graph, initial\_temperature=1000, cooling\_rate=0.95):  
 super().\_\_init\_\_(graph, initial\_temperature, cooling\_rate)  
  
 def run(self, iterations: int) -> list:  
 for i in range(iterations):  
 self.temperature = self.adaptive\_cooling(i)  
 self.iteration()  
 return self.best\_path  
  
 def iteration(self):  
 random.shuffle(self.current\_path)  
 current\_path\_length = self.calculate\_path\_length(self.current\_path)  
 best\_path\_length = self.calculate\_path\_length(self.best\_path)  
  
 if current\_path\_length < best\_path\_length or random.random() < math.exp(  
 -(current\_path\_length - best\_path\_length) / self.temperature):  
 self.best\_path = self.current\_path[:]  
 self.best\_path\_length = current\_path\_length  
  
 self.temperature \*= self.cooling\_rate  
  
 def greedy\_initial\_solution(self):  
 current\_node = random.choice(list(self.graph.nodes()))  
 current\_path = [current\_node]  
 remaining\_nodes = set(self.graph.nodes()) - {current\_node}  
 while remaining\_nodes:  
 next\_node = min(remaining\_nodes, key=lambda x: self.graph[current\_node][x]['weight'])  
 current\_path.append(next\_node)  
 remaining\_nodes.remove(next\_node)  
 current\_node = next\_node  
 return current\_path  
  
 def adaptive\_cooling(self, iteration):  
 return self.temperature / (1 + iteration)