УДК 004

Реализация алгоритма поиска в ширину на языке python

Радионов Сергей Владимирович
Приамурский государственный университет им. Шолом-Алейхема
Студент

Аннотация
Целью данного исследования является создание программы для нахождения кратчайшего пути в лабиринте. Поиск пути будет реализован на языке программирования python, используя алгоритм поиска в ширину (BFS). Результатом исследования является продукт, который находит кротчайший путь в заданном лабиринте.
Ключевые слова: python, графы, поиск в ширину, программа.

Python breadth-first algorithm implementation

Radionov Sergey Vladimirovich
Sholom-Aleichem Priamursky State University
Student

Annotation
The purpose of this study is to create a program for finding the shortest path in the maze. The path search will be implemented in the python programming language using the breadth-first search algorithm (BFS). The result of the study is a product that finds the shortest path in a given maze.
Keywords: python, graph, BFS, program.

Навигация является неотъемлемой частью жизни. Трудно представить поиск пути до нужной точки в большом городе без современного навигатора. Обусловлено это тем, что помнить всю карту города очень сложно. Человек, как правило, запоминает только места где он уже бывал, и возникают проблемы с местами, где человек еще не был. Для упрощения жизни существуют навигаторы. Одним из алгоритмов поиска пути по графу, в который преобразуется карта, является поиск в ширину (BFS).

В статье И.Кутепова рассматривается пример построения вычислителя на основе нечеткой логики и применение языка Python для написания интегрированной среды разработки [2]. И.Е.Бронштейн в своей статье описал вывод типов для программного кода на языке Python. Сначала производится обзор описанных в научной литературе алгоритмов вывода типов для языков с параметрическим полиморфизмом. Затем даётся описание нового алгоритма, являющегося модификацией одного из предыдущих: алгоритма декартова произведения. Показывается, как модуль вывода типов, использующий новый алгоритм, анализирует различные конструкции языка

Python. Представляются результаты работы над прототипом [3]. В работе В.В.Найденова протестирована производительность пары аналогичных приложений, реализующих CRUD логику с помощью прослойки ORM. Сравнивается SQLAlchemy – де-факто стандартный ORM для Python с динамическим ORM для C++ собственной разработки – YB.ORM. Сравнивается производительность при использовании CPython и PyPy. Проверяется влияние отключения логов на производительность [4]. В статье И.Е.Бронштейн рассматриваются виды дефектов, которые обычно встречаются в программном коде на языке Python. Показывается, что возможные дефекты для Python не похожи на те, что часто встречаются в коде на Си/Си++ и, следовательно, необходимо исследование дефектов в крупных проектах с открытым исходным кодом. Даётся классификация найденных дефектов на основе того, нужен ли для нахождения ошибки вывод типов. Показывается, что существует небольшая доля "простых" дефектов, но для обнаружения большинства дефектов вывод типов необходим. Рассматривается вопрос, какие конструкции языка Python должны поддерживаться при выводе типов для нахождения реальных дефектов [5]. В работе Д.А.Кузнецов рассматривается структура интерфейса программы «Фармацевтическая экономическая безопасность» для фармацевтических организаций. Описывается функциональное предназначение и возможности пунктов основного меню прикладной программы [6]. Ю.А.Котов, А.В.Шаповалов в своей статье рассмотрели интерфейс программной реализации экспертной системы для восстановления простой замены букв текста. Описаны базовые элементы интерфейса, включающие выбор функциональных методов и базовых операций (замена, сдвиг, перебор). Не менее значимы иностранные исследования в данной сфере [8-9].

Сначала необходимо создать лабиринт, будем считать, что символ ‘#’ будет означать стену а, ‘.’ свободную клетку. Напишем лабиринт в текстовом файле input.txt (Рис.1).

```
################
...............#
#.#.#.#.#.###.##
#...#.....#....#
#.#.###.#####.##
#...........#..#
#.#.#.#.########
#...#.#.#......#
#.###.#.#.#.#.##
#.#.....#.#.#..#
#.#.###.#.#.####
#...#.....#....#
#.#.#.#.#.#.#.##
#############.##
```

Рис. 1. Лабиринт

Сначала нужно подготовить для работы файлы input и output (Рис.2). Переменная st будет хранить символ, которым обозначается стена. Необходимо считать лабиринт и преобразовать его в граф. Пробегаем по всем точкам и для каждой точки указываем соседние, т.е. точки куда можно перейти их текущей (Рис.3). Таким образом получается именованный массив связей между клетками лабиринта, фармата «i-j = [i1-j1, i2-j2, …]».

```
fin = open('input.txt')
fout = open('output.txt', 'w')

st = '#'
```

Рис.2. Подготовка файлов

```
l = list(map(list, fin.read().split('\n')))

graph = {}
for i, line in enumerate(l):
    for j, val in enumerate(line):
        if val==st:
            continue
        key = '{}-{}'.format(i, j)
        graph[key] = {'value': 1}
        edges = []
        if i>0 and l[i-1][j]!=st:
            edges.append('{}-{}'.format(i-1, j))
        if j>0 and l[i][j-1]!=st:
            edges.append('{}-{}'.format(i, j-1))
        if i<len(l)-1 and l[i+1][j]!=st:
            edges.append('{}-{}'.format(i+1, j))
        if j<len(l[-1])-1 and l[i][j+1]!=st:
            edges.append('{}-{}'.format(i, j+1))

        graph[key]['relations'] = set(edges)
```

Рис.3. Создание графа

Для обхода графа будем использовать алгоритм поиска в ширину, т.е. BFS. Под обходом понимается последовательное посещение (обработка) вершин графа в определённом порядке. Одним из двух часто использующихся способов обхода является обход в ширину, или BFS (англ. breadth-first search, поиск в ширину). Его иногда также называют волновым, по аналогии с распространяющейся волной.

Суть BFS достаточно проста. Обход начинается с посещения определённой вершины (для обхода всего графа часто выбирается произвольная вершина). Затем алгоритм посещает соседей этой вершины. За ними - соседей соседей, и так далее.

Теперь самая важная часть – функция нахождения кратчайшего пути (Рис.5). Создаем список с отправной точкой, откуда будет начинаться обход. Эта отправная точка состоит из 3ех значений, текущая координата, путь до этой координаты и пройденное расстояние. В таком же формате будут добавляться и другие возможные пути. Пока список не будет пустым будет выполняться цикл, в котором, если соседняя от текущей координаты еще не содержится в текущем пути, в список добавляется новый путь, концом которого является эта соседняя точка. Когда все соседние точки перебраны, старый путь удаляется из списка. Если координата точки соответствует конечной цели, функция возвращает данные этого пути, а именно: расстояние и путь от начала до конца.

```
def dfs_paths(graph, start, goal):
    stack = [(start, [start], graph[start]['value'])]
    while stack:
        vertex, path, distance = stack.pop(0)
        for next in graph[vertex]['relations'] - set(path):
            if next == goal:
                path = path + [next]
                distance+=graph[next]['value']
                return {'dist': distance, 'path': path}
            else:
                stack.append((next, path + [next], distance+graph[next]['value']))
```

Рис.5. Функция поиска пути

Осталось записать результат в файл output (Рис.6). Для этого вызовем функцию поиска пути, в параметры которой передадим граф, начальную точку и конечную цель. Запишем в файл расстояние кротчайшего пути. Также в массиве символов, который является лабиринтов перепишем все клетки состоящие в кротчайшем пути на символ ‘o’. Запишем этот массив в файл.

```
min_path = dfs_paths(graph, '1-0', '13-13')
print(min_path['dist'], file=fout)
import copy
l2 = copy.deepcopy(l)
for vertex in min_path['path']:
    i,j = map(int, vertex.split('-'))
    l2[i][j]='o'
for line in l2:
    print(''.join(line), file=fout)
```

Рис.6. Запись результата поиска

Открыв файл output можно увидеть результат выполнения поиска пути в ширину (Рис.7). В первой строке написана длина пути, а далее с помощью

символов нарисован лабиринт и кротчайший путь от начальной до конечной точки.

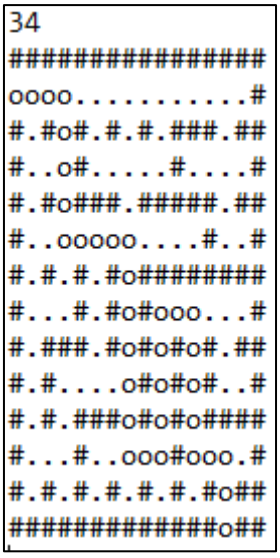

Рис.7. Содержание output.txt

Таким образом, была разработана программа поиска пути в лабиринте с помощью алгоритма поиска пути в ширину.

Библиографический список

1. Лутц М. Изучаем python. М., 2009.
2. Кутепов И. Применение языка python при проектировании нечеткого контроллера // Компоненты и технологии. 2013. № 8 (145). С. 148-154.
3. Бронштейн И.Е. Вывод типов для языка python // Труды Института системного программирования РАН. 2013. Т. 24. С. 161-190.
4. Найденов В.В. Тестирование производительности orm в языках python и c++ // RSDN Magazine. 2014. № 1. С. 05-08.
5. Бронштейн И.Е. Исследование дефектов в коде программ на языке python // Программирование. 2013. Т. 39. № 6. С. 25-32.
6. Кузнецов Д.А. Интерфейс программы "фармацевтическая экономическая безопасность" // Российский медико-биологический вестник им. академика И.П. Павлова. 2009. № 2. С. 162-165.
7. Котов Ю.А., Шаповалов А.В. Интерфейс программы восстановления простой замены букв текста // Современные тенденции развития науки и технологий. 2016. № 4-4. С. 57-59.
8. Smith A. W. et al. Application program interface that enables communication for a network software platform : пат. 7117504 США. – 2006.
9. Parikh V., Moore R., Cheng H. Application program interface for a graphics system : пат. 6456290 США. – 2002.