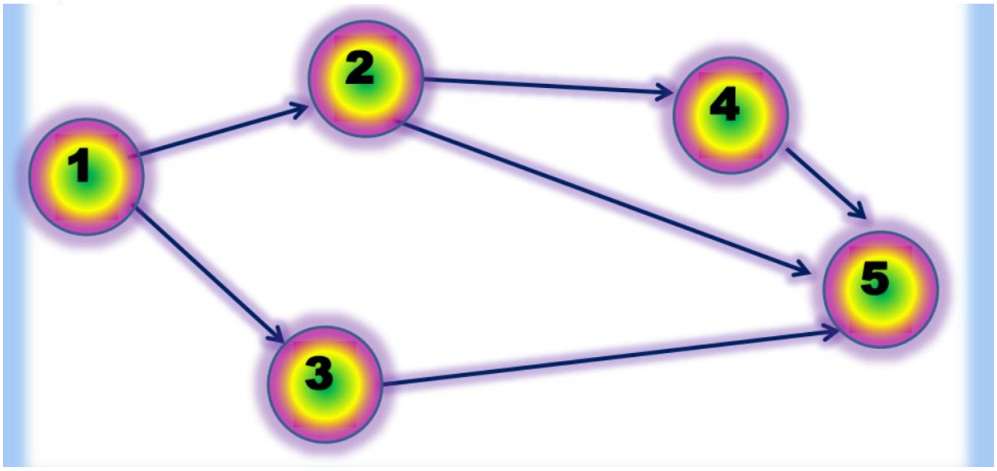
Задание 2

Поиск в ширину

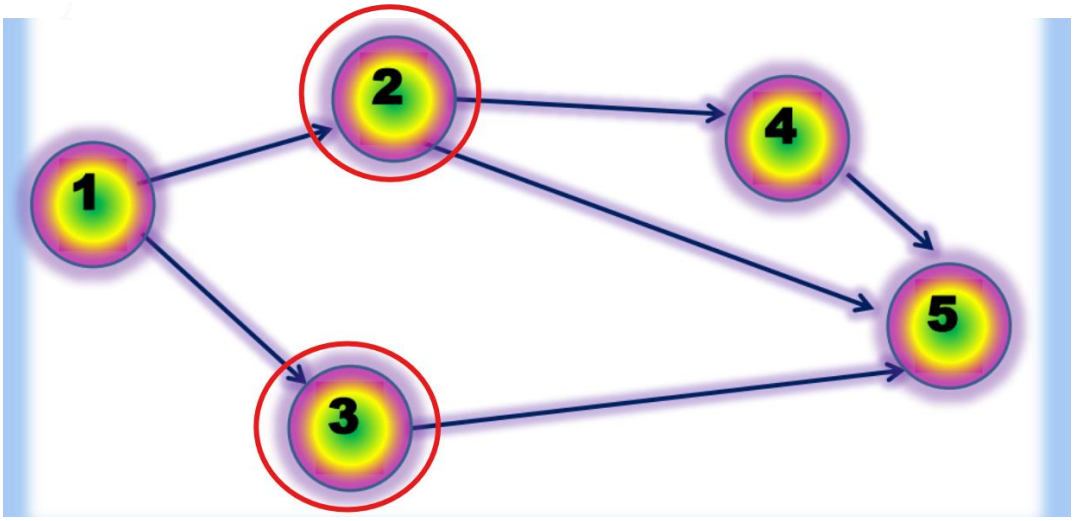
Ссылка на презентацию: https://prezi.com/view/FQLDZ4nPJ6uBDL48C0BT/

Поиск в ширину (англ. *breadth-first search*) — один из основных алгоритмов на графах, позволяющий находить все кратчайшие пути от заданной вершины и решать многие другие задачи.

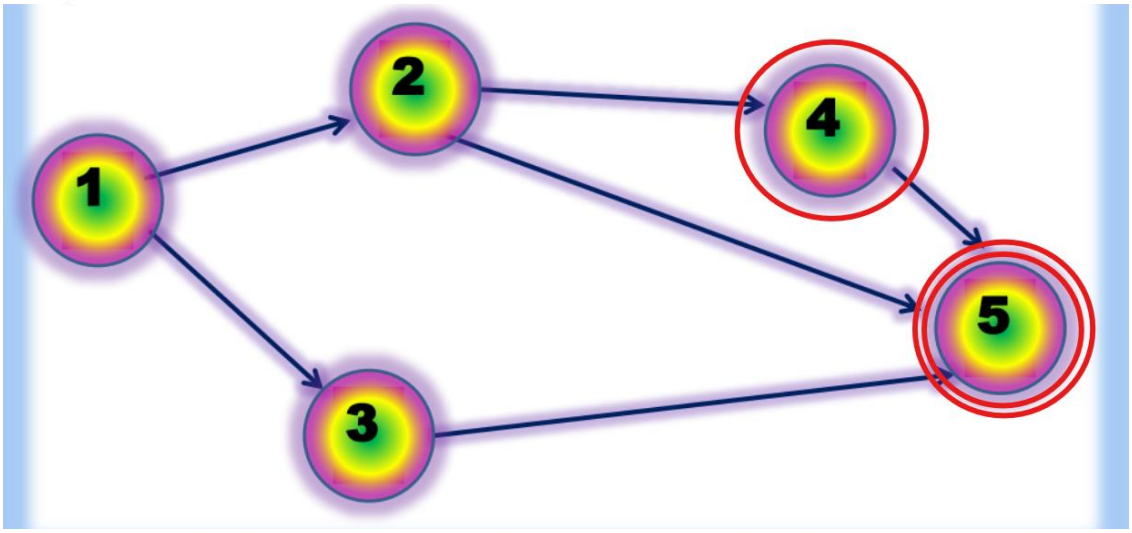
Предположим, вы находитесь в точке 1 и хотите добраться в точку 5. Возможные варианты:



Можно ли добраться до точки за один шаг? На следующем рисунке выделены все места, в которые можно добраться за один шаг.



Точка 5 на схеме не выделена, до неё нельзя добраться за один шаг. А можно ли добраться за два шага?



На этот раз точка 5 выделена. Следовательно, чтобы добраться из точки 1 к точке 5 по этому маршруту, необходимо сделать два шага.

Существуют и другие маршруты, но они требуют больше шагов (три). Алгоритм определил, что самый короткий путь состоит из двух шагов. Задача такого рода называется задачей поиска кратчайшего пути. Часто нужно находить кратчайшие маршруты, например, до университета или для достижения победы в шахматах за минимальное количество ходов. Метод, используемый для решения задачи поиска кратчайшего пути, называется поиском в ширину.

Чтобы определить кратчайший путь от точки 1 до точки 5, нужно было выполнить два действия:

1. Смоделировать задачу в виде графа.

2. Применить метод поиска в ширину для её решения.

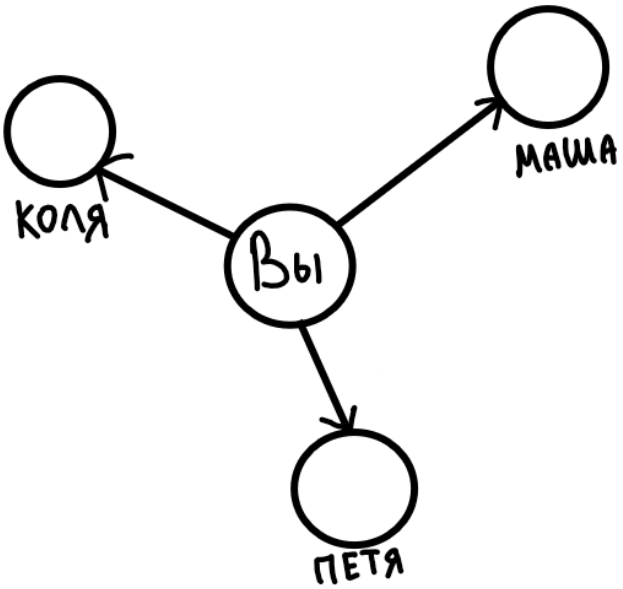
Поиск в ширину также относится к алгоритмам поиска, но работает с графами. Он позволяет ответить на два типа вопросов:

• тип 1: существует ли путь от узла A к узлу B?

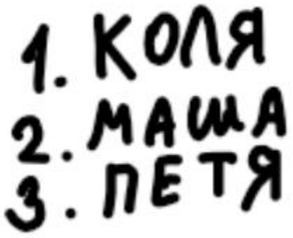
• тип 2: каков кратчайший путь от узла A к узлу B?

Мы уже рассматривали пример поиска в ширину, когда определяли кратчайший путь от 1 до 5, что было вопросом типа 2. Теперь рассмотрим работу алгоритма подробнее с вопросом типа 1: существует ли путь?

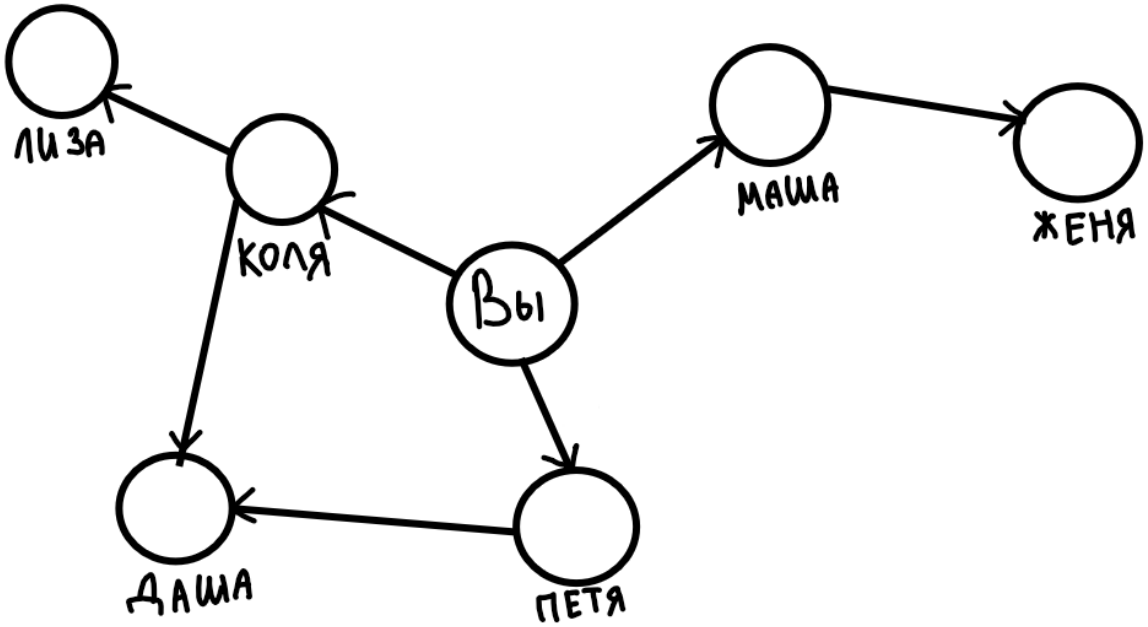
Представьте, что вы хотите снять квартиру и ищете арендодателя. Попробуйте найти его в списке своих друзей в соц. сетях.



Построим список своих друзей, потом проверим каждого человека в списке



Проверяем Колю. Если он сдает квартиру - завершаем программу, если нет - проверяем следующего человека в списке. Маша сдает квартиру? Нет, проверяем Петю. Если и Петя не сдает квартиру, значит среди наших друзей нет арендодателей. В таком случае поиск продолжается среди друзей ваших друзей.



Каждый раз, когда вы проверяете кого-то из списка, вы добавляете в конец списка всех его друзей.

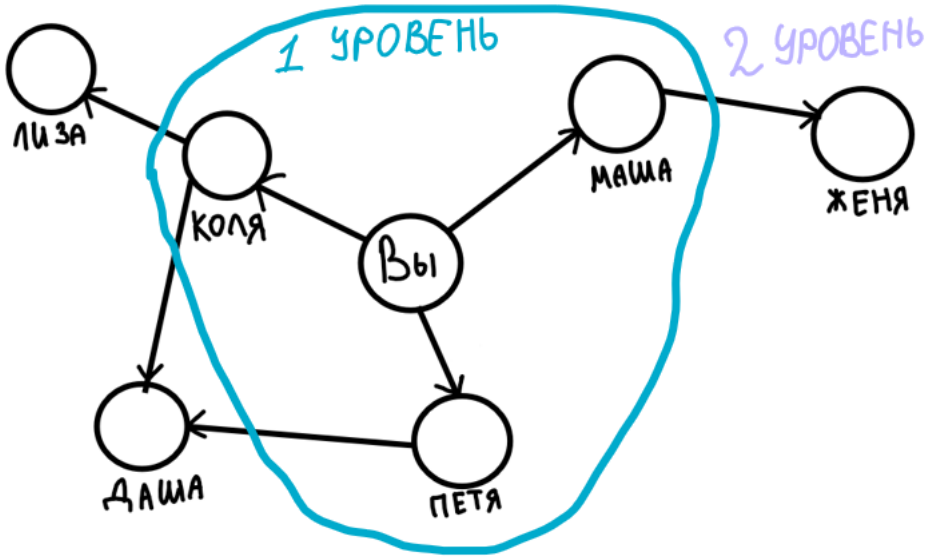
В таком случае поиск ведется не только среди друзей, но и среди друзей друзей тоже. Нам надо найти в сети хотя бы одного арендодателя. Это означает, что со временем вы проверите всех друзей, а потом их друзей и т.д. С этим алгоритмом поиск рано или поздно пройдет по всей сети, пока вы все-таки не наткнетесь на нужного вам человека. Такой алгоритм и называется поиском в ширину.

Напомню два вопроса, на которые алгоритм поиска в ширину может дать ответ:

• Вопрос 1: существует ли путь от узла A к узлу B? (Есть ли арендодатель в вашей сети?)

• Вопрос 2: каков кратчайший путь от узла A к узлу B? (Кто из арендодателей ближе всего к вам?)

Вы уже знаете, как решить первый вопрос; теперь попробуем ответить на второй. Сможете ли вы найти ближайшего арендодателя? Предположим, что ваши друзья — это связи первого уровня, а друзья друзей — второго уровня.



Связи первого уровня имеют приоритет над связями второго уровня, а связи второго уровня — над третьими и так далее. Это означает, что искать среди контактов второго уровня следует только после того, как вы убедитесь, что среди связей первого уровня нет подходящих кандидатов. Таким образом, алгоритм поиска в ширину начинает с начальной точки и сначала проверяет связи первого уровня, добавляя их в список поиска раньше связей второго уровня.

Мы двигаемся вниз по списку, проверяя каждого человека, и связи первого уровня будут проверены до связей второго уровня, поэтому мы найдем ближайшего человека, подходящего нам.

Для того, чтобы добавлять связи второго уровня после связей первого уровня существует специальная структура данных, которая называется очередь.

Очереди

Очередь — это структура данных, которая работает по принципу FIFO (первым пришёл — первым вышел). Представьте, что вы стоите в очереди за мороженым. Тот, кто пришёл первым, получает мороженое первым, а новички становятся в конец очереди.

Основные операции очереди:

1. enqueue (добавление): Это как когда вы приходите в очередь и встаете в конец. Например, если вы пришли после своих друзей, вы встанете за ними.

2. dequeue (удаление): Это когда кто-то из очереди получает мороженое и уходит. Например, ваш друг, который пришёл раньше вас, берёт мороженое и уходит, освобождая место.

Пример:

Предположим, у нас есть очередь людей, ожидающих на автобусе:

- Сначала в очереди стоят: Алиса, Боб, Карина.

- Если приходит Даша, она встает в конец очереди (enqueue): Алиса, Боб, Карина, Даша.

- Когда автобус приезжает, сначала уходит Алиса (dequeue), затем Боб, потом Карина, и наконец Даша.

Таким образом, очередь позволяет организовать порядок, в котором элементы обрабатываются, что делает её полезной в разных задачах, например, в очередях на печать документов или в системах обработки данных.

Реализация графа

Сначала нужно создать граф в программе. Граф включает несколько узлов, каждый из которых соединён с соседними. Выразить эти отношения можно с помощью хэш-таблиц.

Хэш-таблицы это структура данных. Они хранят данные в виде пар "ключ-значение" и используют хэш-функцию для определения места хранения значения по ключу.

Пример:

Предположим, у нас есть список с именами и номерами телефонов:

- "Аня" — 123456

- "Вася" — 654321

1. Ключ и значение: Ключ — это имя, значение — номер телефона.

2. Хэш-функция: Когда мы добавляем "Аня", хэш-функция может преобразовать "Аня" в индекс, например, 2. Мы сохраним номер 123456 по этому индексу.

3. Поиск: Чтобы найти номер Ани, мы снова применяем хэш-функцию к "Аня", получаем индекс 2 и быстро находим 123456.

Таким образом, хэш-таблицы позволяют быстро добавлять и находить данные по ключам.

Таким образом мы можем показать связи вида: Вы -> (Коля, Маша, Петя)

А вот как это записывается на Python:

```
graph = {}

graph["you"] = ["Kolya", "Masha", "Petya"]
```

Обратите внимание: элемент «вы» (you) отображается на массив. Следовательно, результатом выражения graph["you"] является *массив* всех ваших соседей.

Для представления графа на Python ничего больше не потребуется.

Как будет выглядеть конечный граф со всеми связями? Вот так:

```
graph = {}

graph["you"] = ["Kolya", "Masha", "Petya"]
```

```
graph["Kolya"] = ["Liza", "Dasha"]
graph["Masha"] = ["Zhenya"]
graph["Petya"] = ["Dasha"]
graph["Liza"] = []
graph["Dasha"] = []
graph["Zhenya"] = []
```

Реализация алгоритма

1. Создать очередь с именами проверяемых людей
2. Извлечь из очереди одного человека
3. Проверить является ли он арендодателем

4.1 Если ДА, то Завершить

4.2 Если НЕТ, то Добавить всех соседей этого человека в очередь

5. Зациклить, вернувшись к пункту 2

6. Если очередь пуста, среди ваших друзей никто не сдает квартиры

Теперь рассмотрим написание кода. В Python для создания двусторонней очереди (дека) используется функция deque:

```
from collections import deque
search_queue = deque()   #Создание новой очереди
search_queue += graph["you"]   #Все соседи добавляются в очередь поиска
```

Разберем код далее:

```
while search_queue:   #Пока очередь не пуста…
   person = search_queue.popleft()  #из очереди извлекается первый человек
    if person_is_lessor(person): #Проверяем, сдает ли человек квартиры
      print person + " is a lessor!"  #Да, это арендодатель
      return True
    else:
```

search_queue += graph[person]   #Нет, не является. Все друзья этого человека добавляются в очередь поиска

return False   #Если выполнение дошло до этой строки, значит, в очереди нет арендодателя

И последнее: нужно определить функцию person_is_lessor, которая сообщает сдает ли человек квартиру. Для примера будем считать, что арендодатель тот, чье имя заканчивается на g. Тогда получим такую функцию:

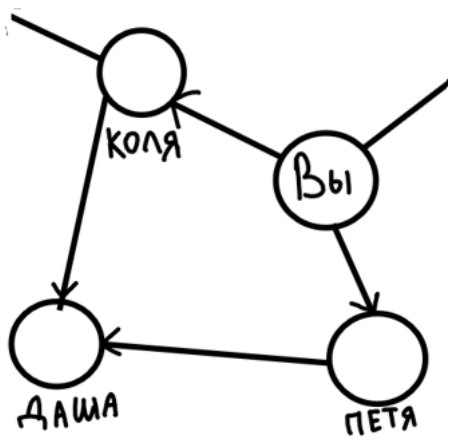
```
def person_is_lessor(name):
    return name[-1] == 'g'
```

Алгоритм будет функционировать до тех пор, пока:

- не обнаружит арендодателя

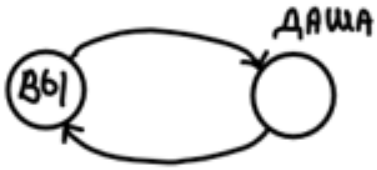
- очередь не станет пустой (если среди друзей арендодателя нет)

У Коли и Пети есть общий знакомый — Даша. Таким образом, она будет добавлена в очередь дважды: один раз через Колю и один раз через Петю. Это приведет к тому, что Даша окажется в очереди поиска дважды.



Однако проверять, сдает ли Даша квартиру, нужно лишь один раз. Если вы проверите её дважды, это займет ненужное время, и даже может создать бесконечный цикл.

Представим граф, состоящий из двух узлов: вас и Даши.



В начале очередь поиска содержит всех ваших соседей.

Сначала вы проверяете Дашу. Она не сдаёт квартиру, и все её соседи — вы — добавляются в очередь поиска.

Затем вы проверяете себя. Вы также не арендодатель, и все ваши соседи — Даша — снова добавляются в очередь.

Таким образом, возникает бесконечный цикл, так как очередь будет постоянно переключаться между вами и Дашей.

Во избежание таких ситуаций заведем список с уже проверенными людьми.

Вот так будет выглядеть финальный код, со всеми нюансами:

```
def search(name):
    search_queue = deque()
    search_queue += graph[name]
    searched = []    #Этот массив используется для отслеживания уже проверенных людей
     while search_queue:
        person = search_queue.popleft()
         if not person in searched:    #Человек проверяется только в том случае, если он не проверялся ранее
            if person_is_lessor(person):
               print person + " is a lessor!"
               return True
            else:
               search_queue += graph[person]
               searched.append(person)       # Человек помечается как уже проверенный
```

```
    return False
search("you")
```

Список литературы:

1. https://ru.algorithmica.org/cs/shortest-paths/bfs/
2. https://habr.com/ru/companies/yandex_praktikum/articles/705178/
3. https://evileg.com/ru/post/512/
4. https://intuit.ru/studies/courses/101/101/lecture/2951
5. https://ru.wikipedia.org/wiki/Очередь_(программирование)
6. https://habr.com/ru/articles/422259/
7. https://ru.wikipedia.org/wiki/Хеш-таблица
8. https://ru.hexlet.io/courses/python-dicts/lessons/hash-table/theory_unit
9. https://habr.com/ru/companies/ruvds/articles/705368/
10. https://external.software/archives/12528