Дискретные оптимизационные модели

Задача о рюкзаке. «Жадные» алгоритмы. Динамическое программирование. Оптимизация на графах.

Дискретные оптимизационные задачи

- Многие реальные задачи могут быть сформулированы в виде задач оптимизации.
- *Дискретная оптимизация* (или дискретное программирование) предполагает работу только с дискретными переменными. Дискретная оптимизация тесно связана с *комбинаторной оптимизацией*.
- Решение оптимизационных задач в общем случае весьма затратно с вычислительной точки зрения.
- Есть довольно много задач комбинаторной оптимизации, точное решение которых можно получить только полным перебором. Это может быть *очень* долго.
- Часто встречающимся в практике подходом является использование *«жадных» алгоритмов*, позволяющих найти хорошее приближенное решение оптимизационной задачи.

Задача о рюкзаке

- Рюкзак обладает ограниченной вместимостью / человек не может нести больше определенного веса.
- При этом хочется взять как можно больше вещей.
- Как выбрать, какие вещи положить в рюкзак, а какие оставить?
- Есть два варианта задачи о рюкзаке:
  - Рюкзак 0/1 (дискретная).
  - Непрерывная или дробная задача о рюкзаке.

VS

Приложения задачи о рюкзаке

- Задача о составлении плана питания (рациона).
- Размещение грузов на складе минимальной площади.
- Раскройка ткани – как из имеющегося куска получить максимальное число выкроек определенной формы.
- Расчет оптимальных капиталовложений.
- Автоматическое реферирование текста.
- Криптография (алгоритм ассиметричного шифрования).
- … и другие

Формализация 0/1 задачи о рюкзаке

- Каждый предмет представляется в виде пары чисел *<value, weight>*.
- Рюкзак может вместить такое количество вещей, что их суммарный вес не превышает *w*.
- Вектор *I* длины *n* представляет набор возможных предметов. Каждый элемент вектора – предмет.
- Вектор *V* длины *n* используется для того, чтобы обозначить, берется та или иная вещь или нет: если $V[i] = 1$, предмет $I[i]$ кладется в рюкзак; если $V[i] = 0$, то предмет $L[i]$ не кладется в рюкзак.
- Необходимо найти такой вектор *V*, что:

$$\sum_{i=0}^{n-1} V[i] * I[i].value \to \max,$$

при соблюдении ограничения:

$$\sum_{i=0}^{n-1} V[i] * I[i].weight \le w.$$

Алгоритм полного перебора (Brute Force)

- Поскольку множество предметов, которые можно взять, конечно, можно попробовать решить задачу *полным перебором*.
- Алгоритм в этом случае состоит в следующем:
  1. Пронумеровать все возможные сочетания предметов. Иными словами, построить все подмножества множества предметов. Это называется *булеан* (по-английски - *power set*).
  2. Исключить из получившегося набора сочетаний те сочетания, для которых суммарный вес превышает допустимый.
  3. Из оставшихся сочетаний выбрать то (или те), суммарная ценность которого максимальна.
- Сколько различных значений может быть у вектора V?
- Ровно столько, сколько различных двоичных чисел можно представить n битами: 2^n.
- Если предметов 100, то 1267650600228229401496703205376.

«Жадный» алгоритм

- *Принцип*: до тех пор, пока рюкзак не полон, класть «лучший» предмет в рюкзак.
- Но *что значит «лучший»?*
  - с наибольшей ценностью (value);
  - наименее тяжелый (weight);
  - с наилучшим отношением value / weight.
- Рассмотрим реализацию «жадного» алгоритма на примере.
- Предположим, мы хотим пообедать, но при этом не хотим, чтобы суммарная энергетическая ценность обеда превысила 800 ккал. Это *задача составления рациона питания*.
- Предположим, меню выглядит следующим образом:

Food	**wine**	**beer**	**pizza**	**burger**	**fries**	**coke**	**apple**	**donut**
Value	89	90	30	50	90	79	90	10
Calories	123	154	258	354	365	150	95	195

Класс Food: код

```
class Food(object):
      def __init__(self, n, v, w):
            self.name = n
            self.value = v
            self.calories = w
      def getValue(self):
            return self.value
      def getCost(self):
            return self.calories
      def density(self):
            return self.getValue()/self.getCost()
      def __str__(self):
            return self.name + ': <' + str(self.value)
                  + ', ' + str(self.calories) + ‘>’
def buildMenu(names, values, calories):
      menu = []
      for i in range(len(values)):
            menu.append(Food(names[i], values[i], calories[i]))
      return menu
```

Реализация «жадного» алгоритма: код

```
def greedy(items, maxCost, keyFunction):
      itemsCopy = sorted(items, key = keyFunction, reverse = True)
      result = []
      totalValue, totalCost = 0.0, 0.0
      for i in range(len(itemsCopy)):
            if (totalCost+itemsCopy[i].getCost()) <= maxCost:
                  result.append(itemsCopy[i])
                  totalCost += itemsCopy[i].getCost()
                  totalValue += itemsCopy[i].getValue()
      return (result, totalValue)

def testGreedy(items, constraint, keyFunction):
      taken, val = greedy(items, constraint, keyFunction)
      print('Total value of items taken =', val)
      for item in taken:
            print(' ', item)
```

Тестирование «жадного» алгоритма: код

```
def testGreedys(foods, maxUnits):
      print('Use greedy by value to allocate', maxUnits, 'calories’)
      testGreedy(foods, maxUnits, Food.getValue)
      print('\nUse greedy by cost to allocate', maxUnits, 'calories’)
      testGreedy(foods, maxUnits, lambda x: 1/Food.getCost(x))
      print('\nUse greedy by density to allocate', maxUnits, 'calories’)
      testGreedy(foods, maxUnits, Food.density)


names = ['wine', 'beer', 'pizza', 'burger', 'fries’, 'cola', 'apple’,
             'donut', 'cake']
values = [89,90,95,100,90,79,50,10]
calories = [123,154,258,354,365,150,95,195]
foods = buildMenu(names, values, calories)


testGreedys(foods, 750)
```

Какие можно сделать выводы?

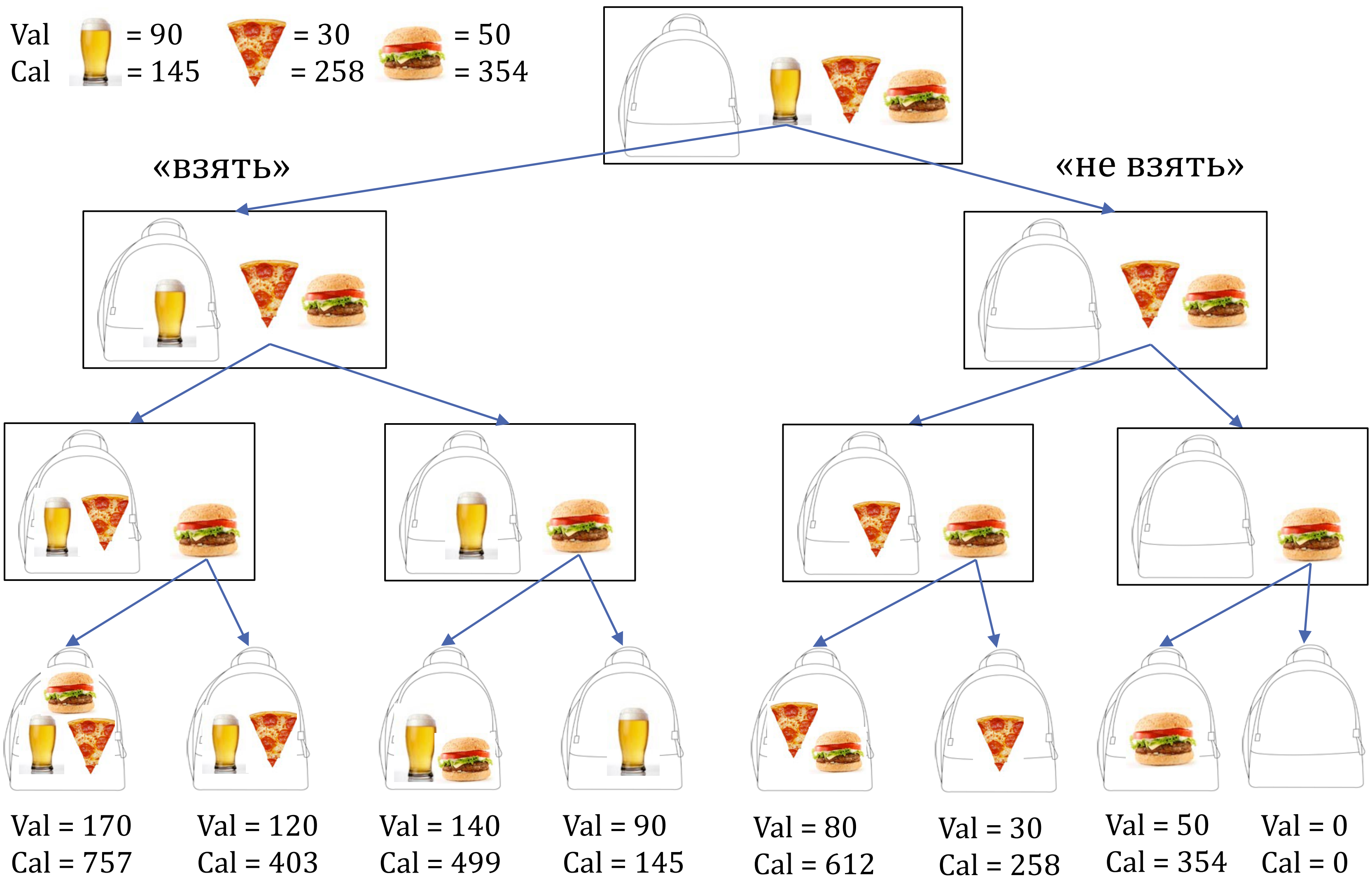
- Последовательность локально «оптимальных» решений не всегда приводит к глобальному оптимуму.
- Всегда ли `greedy by density` дает лучший результат?
- Нет. Попробуйте

  `testGreedys (foods, 1000)`.
- «Жадный» алгоритм легко реализуется.
- Он эффективен с вычислительной точки зрения.
- Однако, он не всегда находит оптимальное решение. Более того, совершенно неизвестно, насколько найденное приближенное решение близко к реальному оптимуму.

Дерево полного перебора

- Дерево перебора строится сверху-вниз. Вверху – корень.
- Первый предмет выбирается из оставшихся для рассмотрения:
  - Если для него еще имеется место в рюкзаке, строится ветвь, отражающая последствия выбора этого предмета. По умолчанию это левая ветвь.
  - Правая ветвь представляет собой рассмотрение последствий, если эту вещь не взять.
- Процесс продолжается рекурсивно до тех пор, пока не будут перебраны все варианты.
- В результате выбирается узел, в котором искомая функция максимальна при этом выполняются все ограничения.

Пример построения дерева перебора



Особенности алгоритма полного перебора

- Общее время вычисления зависит от количества узлов.
- Количество уровней дерева равно количеству предметов.
- Количество узлов на i-м уровне дерева равно 2^i.
- Если общее число предметов n, то

$$\sum_{i=0}^{n} 2^i \sim \mathcal{O}(2^{n+1}).$$

- Это значит, что *сложность алгоритма экспоненциальная*.
- Очевидная оптимизация алгоритма: не рассматривать части дерева, для которых заведомо не выполняются ограничения. Однако, *это не меняет сложности алгоритма*.
- Значит ли это, что полный перебор абсолютно бесполезен?
- Проверим… для этого вернемся к задаче о построении меню.

Алгоритм полного перебора: код

```
def maxVal(toConsider, avail):
      if toConsider == [] or avail == 0:
            result = (0, ())
      elif toConsider[0].getUnits() > avail:
            result = maxVal(toConsider[1:], avail)
      else:
            nextItem = toConsider[0]
            withVal, withToTake = maxVal(toConsider[1:],
                                        avail - nextItem.getUnits())
            withVal += nextItem.getValue()
            withoutVal, withoutToTake = maxVal(toConsider[1:], avail)
      if withVal > withoutVal:
            result = (withVal, withToTake + (nextItem,))
      else:
            result = (withoutVal, withoutToTake)
      return result
```

Что же получилось?

- Мы получили ответ, лучший, чем давал «жадный» алгоритм. *Это точный ответ*.
- Вычисления были проведены быстро.
- Однако, 2^8 это совсем не большое число.
- Посмотрим, что произойдет, если мы увеличим меню.

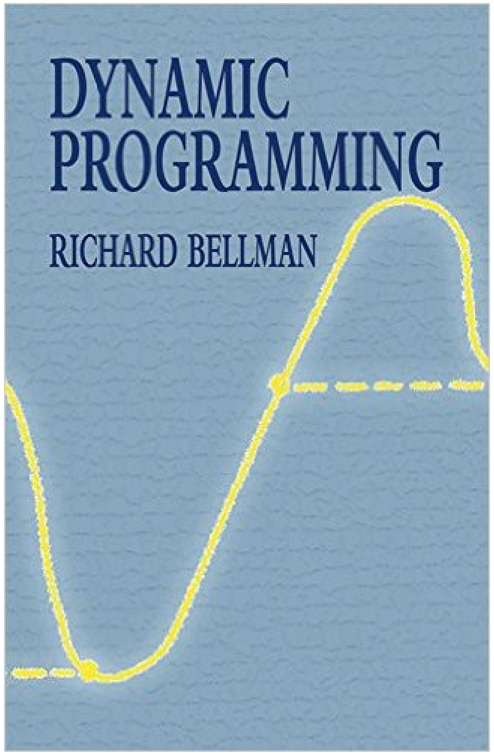
```
import random

def buildLargeMenu(numItems, maxVal, maxCost):
    items = []
    for i in range(numItems):
        items.append(Food(str(i), random.randint(1, maxVal),
                          random.randint(1, maxCost)))
    return items

for numItems in (5, 10, 15, 20, 25, 30, 35, 40, 45):
    items = buildLargeMenu(numItems, 90, 250)
    testMaxVal(items, 750, False)
```

Динамическое программирование

- Подход динамического программирования может помочь.
- Немного о происхождении названия: *“The 1950s were not good years for mathematical research…I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics... What title, what name, could I choose? ... It's impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.”*



Richard Bellman

Рекурсивное вычисление чисел Фибоначчи

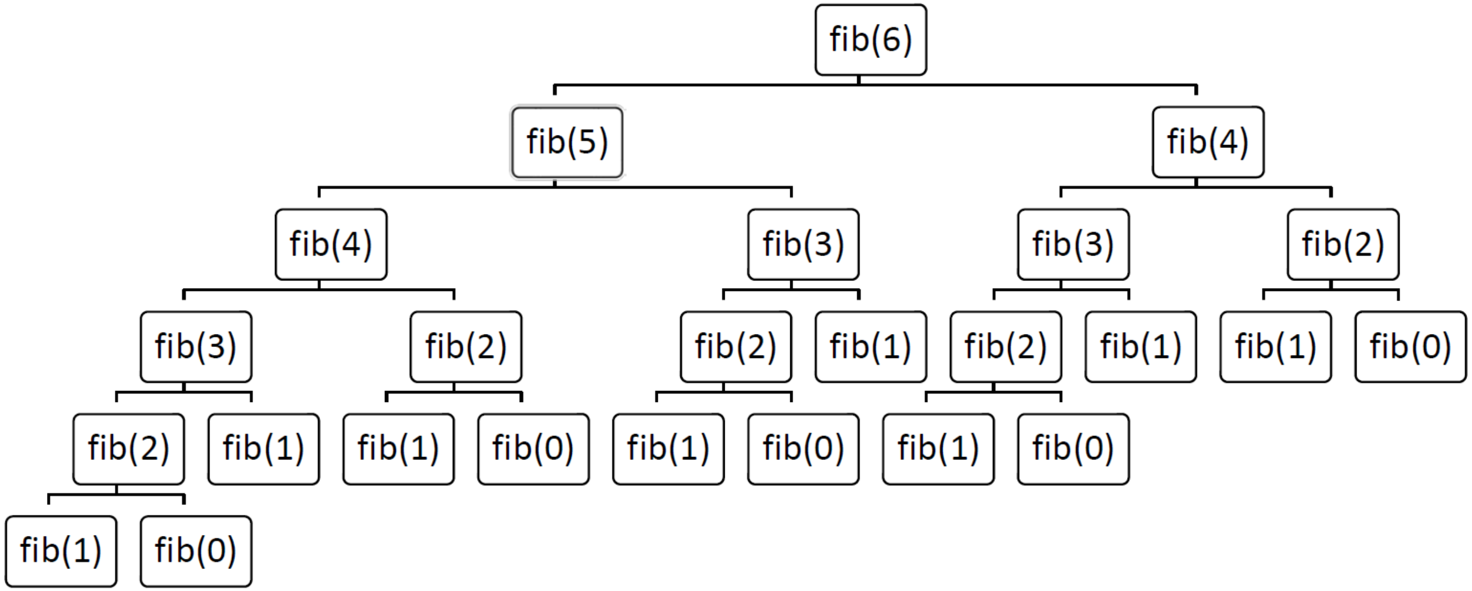
```
def fib(n):
        if n == 0 or n == 1:
                return 1
        else:
                return fib(n - 1) + fib(n - 2)


fib(120) = 8,670,007,398,507,948,658,051,921
```

Справка

$F_0 = 1, \; F_1 = 1,$

$F_n = F_{n-1} + F_{n-2}$



Повторяющиеся фрагменты

- *Принцип*: обмен «времени» на «пространство» (память).
- Необходимо создать таблицу, чтобы записывать уже сделанное:
  - Перед вычислением fib(x) проверить, не хранится ли уже значение fib(x) в таблице.
  - Если да, то найти его в таблице и использовать.
  - Если нет, вычислить его и записать в таблицу.
- Это называется *мемоизация*.

```
def fastFib(n, memo = {}):
      if n == 0 or n == 1:
            return 1
      try:
            return memo[n]
      except KeyError:
            result = fastFib(n-1, memo) + fastFib(n-2, memo)
            memo[n] = result
            return result
```

Когда динамическое программирование работает?

- *Оптимальная подструктура*: глобальный оптимум может найдем комбинацией решения подзадач меньшей размерности:

  ```
  For x > 1, fib(x) = fib(x -1) + fib(x –2)
  ```

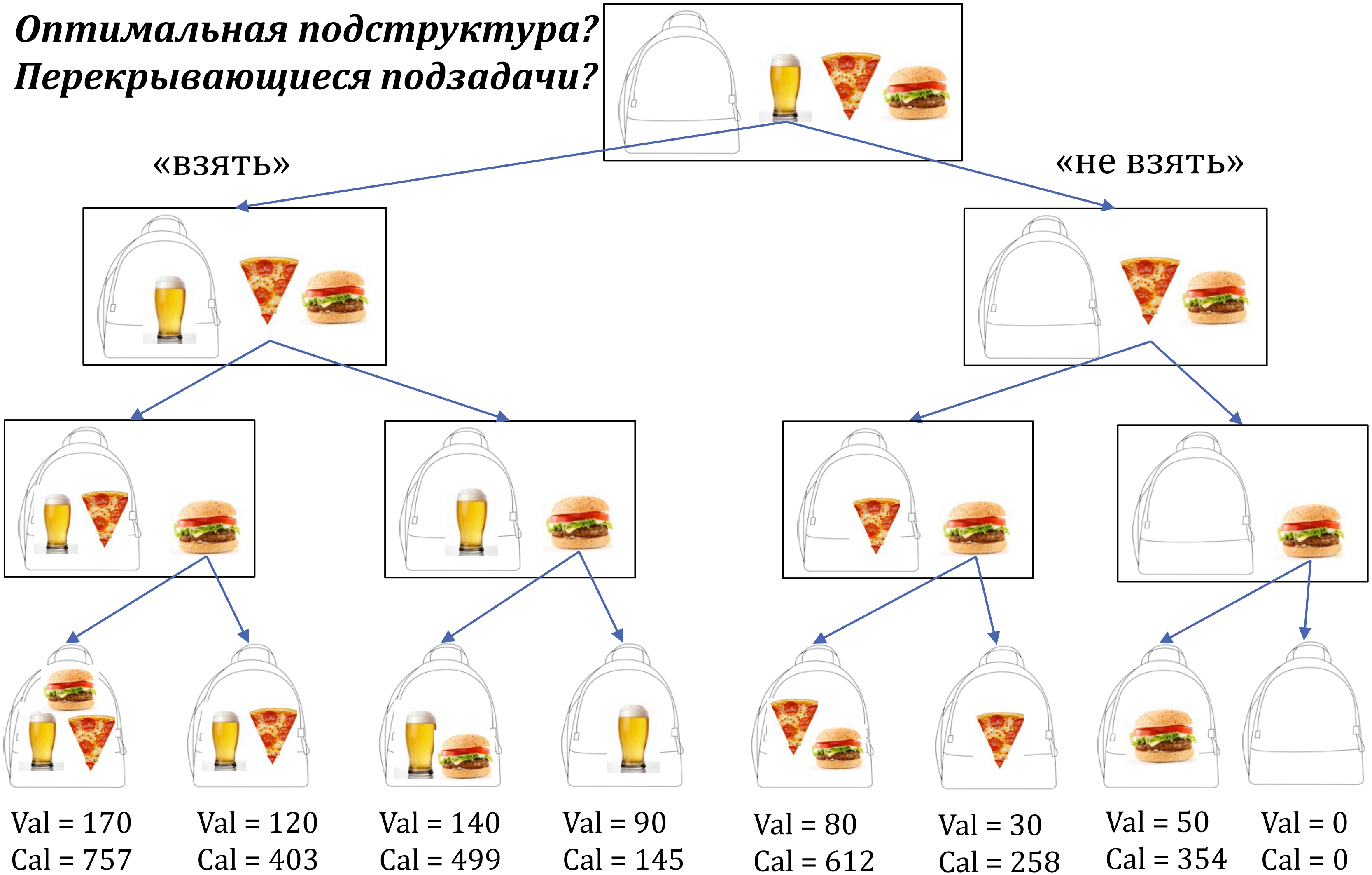
- *Перекрывающиеся подзадачи*: нахождение оптимального решения требует решения одной и той же задачи несколько раз:

  ```
  Compute fib(x) many times
  ```

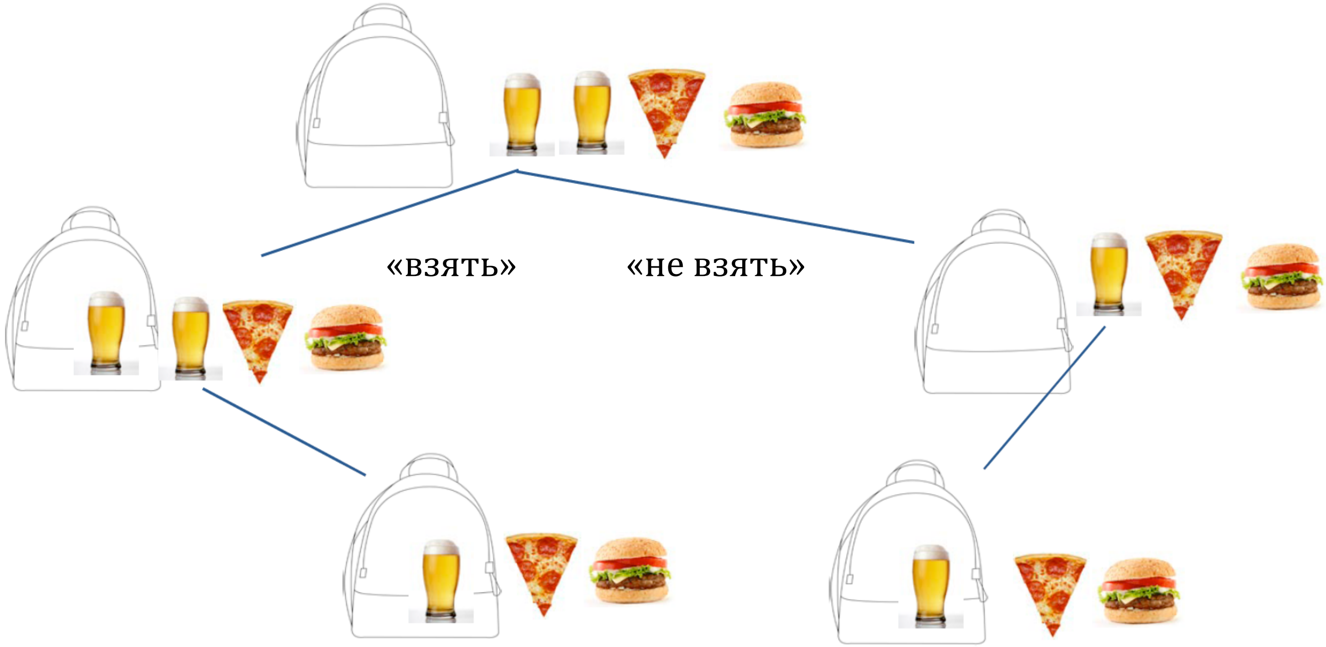
- Может ли этот подход быть применен к задаче о рюкзаке?

Вернемся к дереву перебора

Оптимальная подструктура?
Перекрывающиеся подзадачи?



Другое меню

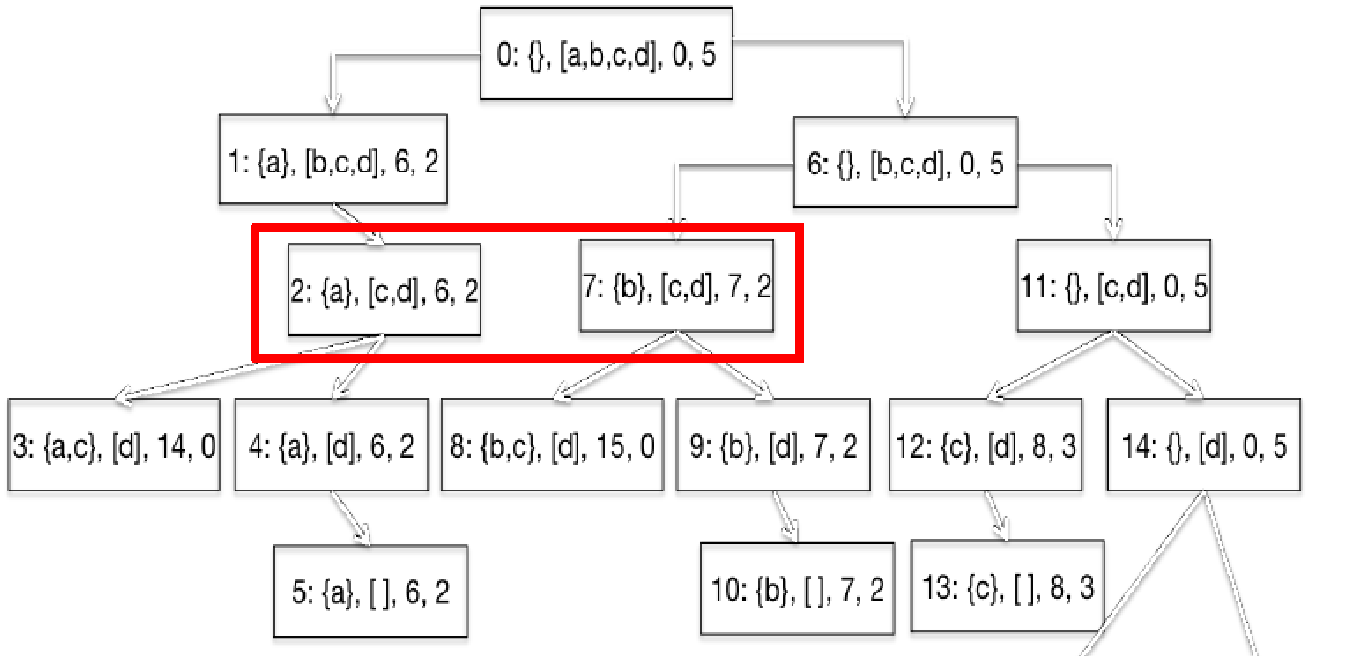


Какая задача решается в каждом узле?

- Дан оставшийся вес, нужно максимизировать ценность, выбирая из оставшихся предметов.
- Набор выбранных до этого предметов и их ценность не имеет значения.

Пример дерева решений

- Each node = <taken, left, value, remaining calories>



Item	Value	Calories
a	6	3
b	7	3
c	8	2
d	9	5

Модификация maxVal с учетом мемоизации

- Прибавим memo в качестве третьего аргумента:
  - `def fastMaxVal(toConsider, avail, memo = {}):`
- Ключ memo – tuple:

  `(len(toConsider), avail)`
  - toConsider - предметы, которые остались
  - avail - доступный вес;
  - предметы, которые нужно рассмотреть, представлены длиной вектора len(toConsider)
- Первое действие, осуществляемой в теле функции, проверка того, что оптимальный выбор предметов, отвечающий оставшемуся весу, уже записан в memo.
- Последнее действие – обновление memo.

Производительность

len(items)	2**len(items)	Number of calls
2	4	7
4	16	25
8	256	427
16	65,536	5,191
32	4,294,967,296	22,701
64	18,446,744,073,709,551,616	42,569
128	Big	83,319
256	Really Big	176,614
512	Ridiculously big	351,230
1024	Absurdly big	703,802

Как это может быть?

- Сложность задачи экспоненциальная.
- Удалось ли нам обойти законы мироздания?
- Динамическое программирование это чудо?

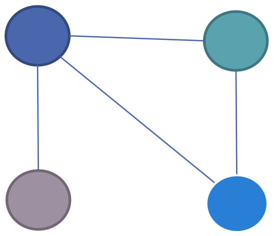
- Нет, но вычислительная сложность – тонкая вещь.
- Количество запусков fastMaxVal зависит от количества разных пар значений <toConsider, avail>:
  - Число возможных значений toConsider ограничено len(items).
  - Возможные значения avail сложнее определить, но они ограничены числом различных сумм весов.

Оптимизационные задачи на графах

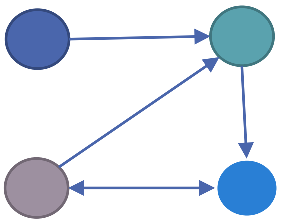
- Что такое граф?
- *Набор вершин*, с которыми могут ассоциироваться некоторые свойства.
- *Набор ребер*, каждое из которых содержит две вершины.

Графы могут быть:

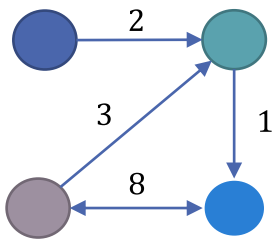
- *Неориентированные* (graph)
- *Ориентированные* (digraph от directed graph): у ребра есть направление, т. е. начало (вершина) и окончание (друга вершина).
- *Взвешенные* – каждому ребру соответствует некоторое число (вес).
- *Невзвешенные*



Неориентированный невзвешенный граф



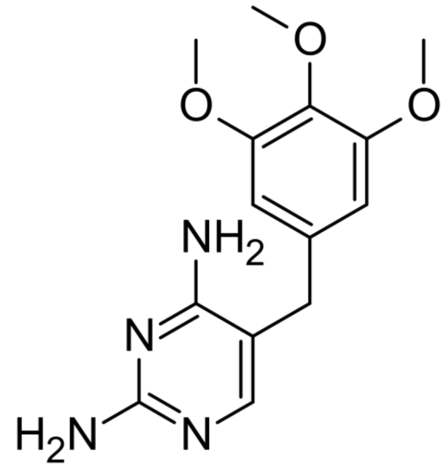
Ориентированный невзвешенный граф

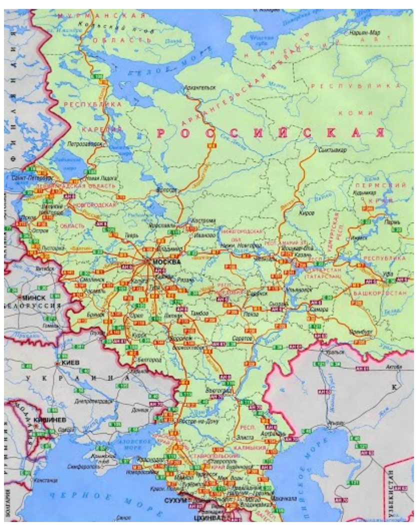


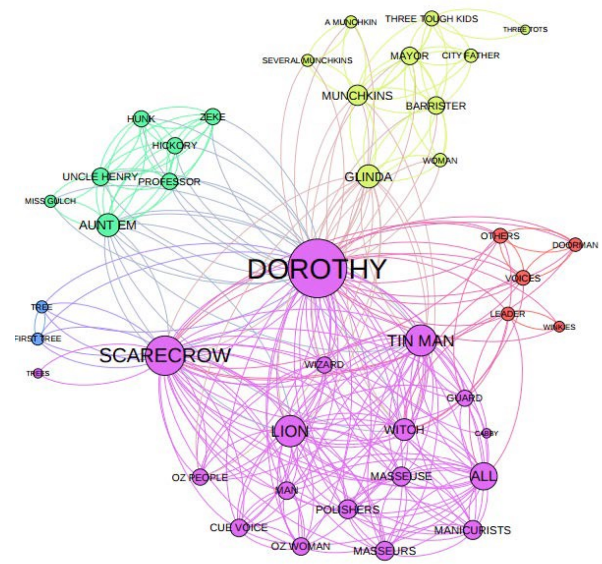
Ориентированный взвешенный граф

Примеры графов

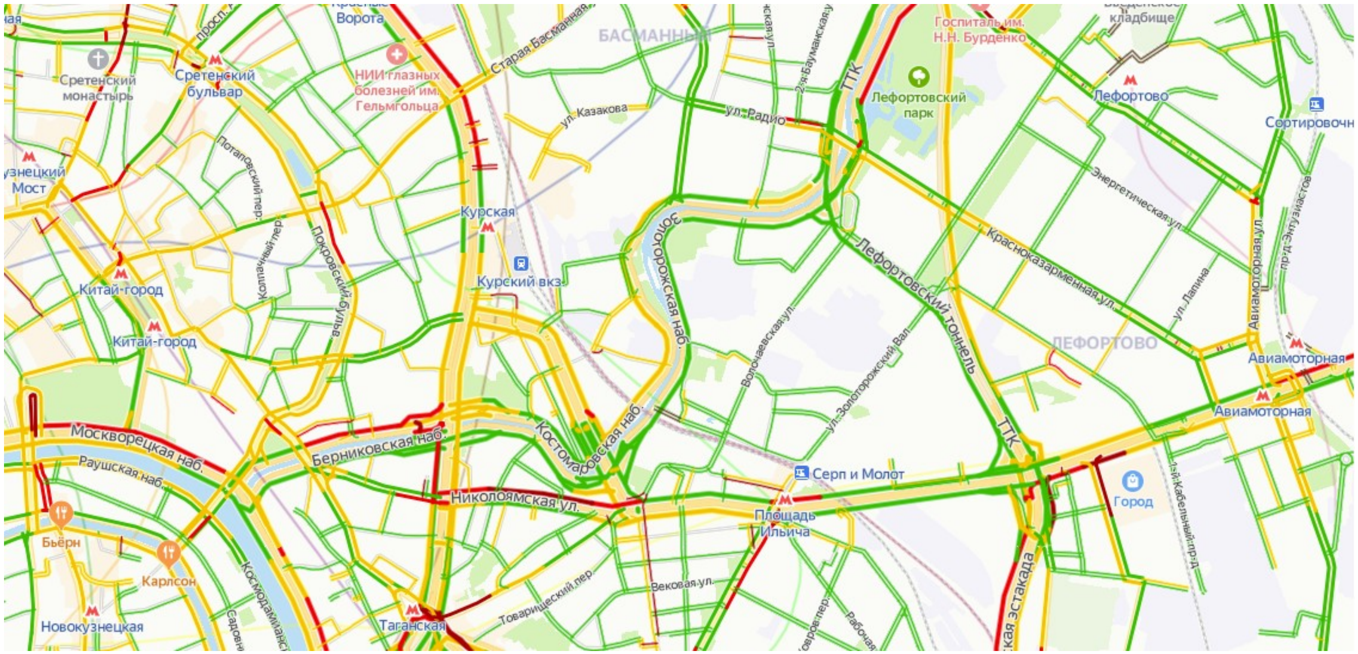
- Сеть дороги, соединяющих разные города.
- Связь атомов в молекуле (химическая структура).
- Генеалогическое дерево.
- Социальная сеть.
- Дерево – важный частный случай графа. Это ориентированный граф, в котором каждая пара вершин соединена ребром.







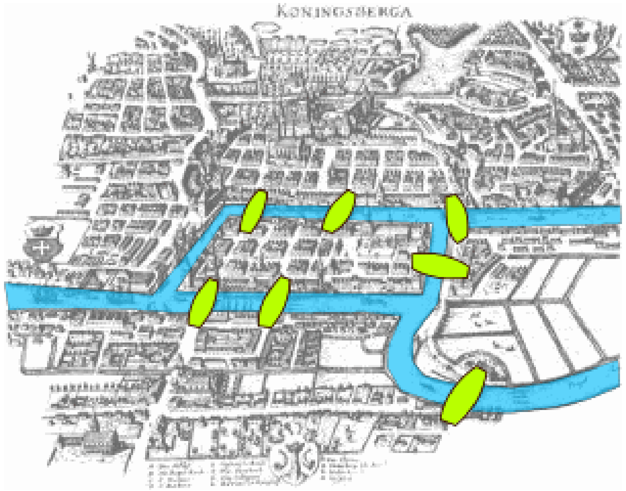
Теория графов для построения кратчайшего маршрута

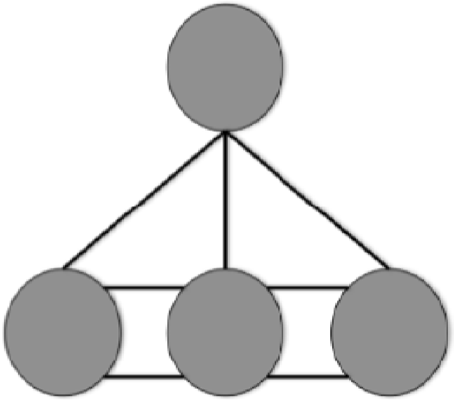


- Вершины: точки, где дороги пересекаются или заканчиваются.
- Ребра: соединения между точками.
- Каждое ребро имеет вес, характеризующий время проезда между двумя точками.

Мосты Кёнигсберга

- Задача о семи мостах Кёнигсберга (Л. Эйлер, 1735 г.).
- Возможно ли построить маршрут так, чтобы перейти через каждый мост ровно один раз?
- Каждый остров – вершина (их 4).
- Каждый мост – неориентированное ребро.
- Модель абстрагируется от несущественных деталей (размер островов, длины мостов).





Классы «вершина» (Node) и «ребро» (Edge): код

```
class Node(object):
        def __init__(self, name):
                """Assumes name is a string"""
                self.name = name
        def getName(self):
                return self.name
        def __str__(self):
                return self.name

class Edge(object):
        def __init__(self, src, dest):
                """Assumes src and dest are nodes"""
                self.src = src
                self.dest = dest
        def getSource(self):
                return self.src
        def getDestination(self):
                return self.dest
        def __str__(self):
                return self.src.getName() + '->’+ self.dest.getName()
```

Представление ориентированных графов

- Матрица принадлежности:
  - Строки: вершины начала ребра (source);
  - Столбцы: вершина конца ребра (destination);
  - Cell[s, d] = 1, если существует ребро, соединяющее s и d.
    В противном случае Cell[s, d] = 0.
- Вектор принадлежности (list):
  - Каждой вершине сопоставляется список вершин назначения.

```
class Digraph(object):
     """edges is a dict mapping each node to a list of
     its children"""
     def __init__(self):
          self.edges = {}

     def addNode(self, node):
          if node in self.edges:
               raise ValueError('Duplicate node’)
          else:
               self.edges[node] = []
```

```
def addEdge(self, edge):
        src = edge.getSource()
        dest = edge.getDestination()
        if not (src in self.edges and dest in self.edges):
                raise ValueError('Node not in graph’)
        self.edges[src].append(dest)

def childrenOf(self, node):
        return self.edges[node]
def hasNode(self, node):
        return node in self.edges
def getNode(self, name):
        for n in self.edges:
                if n.getName() == name:
                        return n
        raise NameError(name)
def __str__(self):
        result = ‘’
        for src in self.edges:
                for dest in self.edges[src]:
                        result = result + src.getName() + '->’\
                                        + dest.getName() + '\n’
                return result[:-1] #omit final newline
```

Неориентированный граф: класс Graph

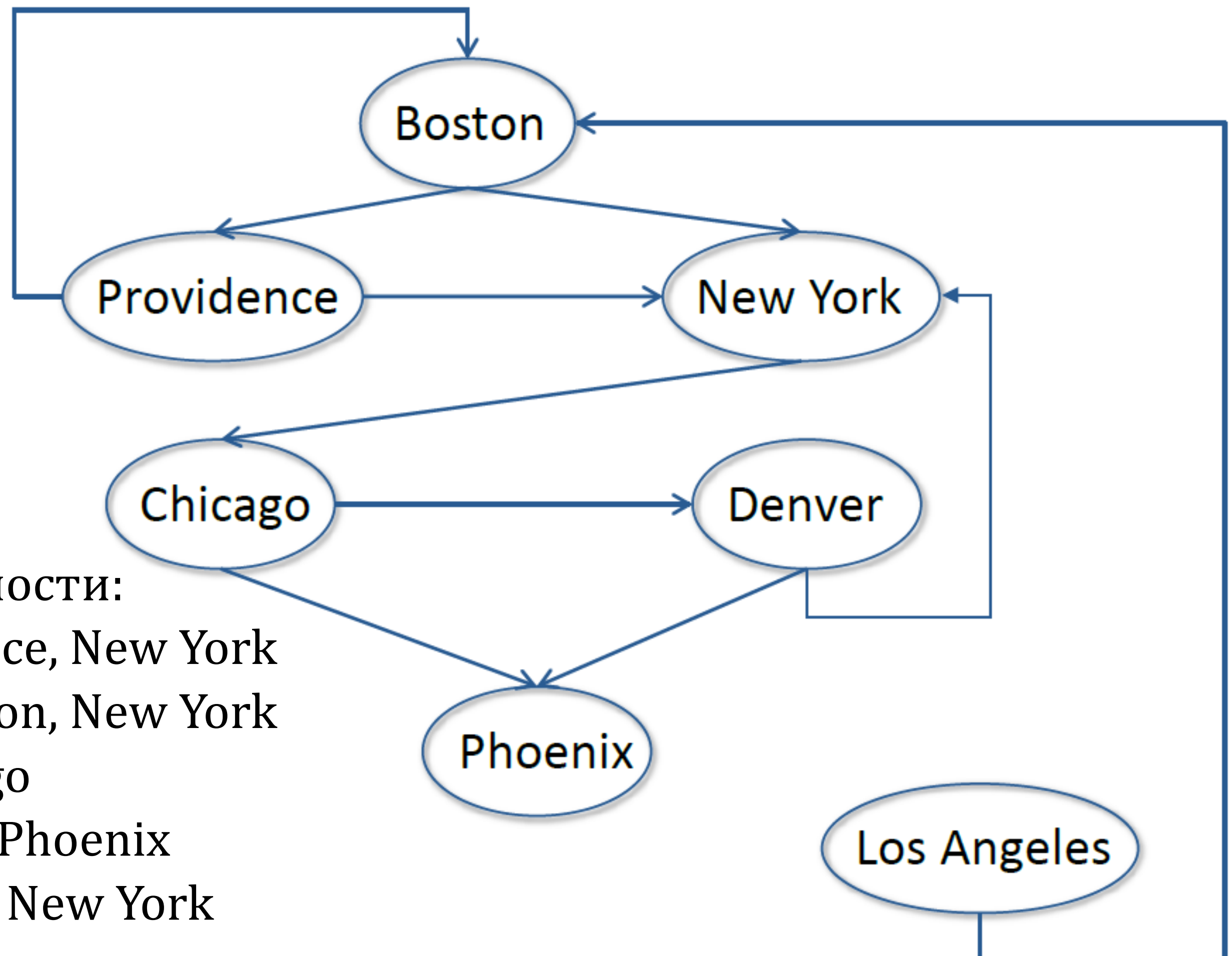
```
class Graph(Digraph):
    def addEdge(self, edge):
        Digraph.addEdge(self, edge)
        rev = Edge(edge.getDestination(), edge.getSource())
        Digraph.addEdge(self, rev)
```

- В этом представлении неориентированный граф – подкласс ориентированного.

Классические оптимизационные задачи на графах:

- Найти кратчайший путь, соединяющий вершины n_1 и n_2, такой, что: начало первого ребра в n_1, конец последнего ребра в n_2. Для ребер e_1 и e_2 в последовательности, если e_2 следует за e_1, то начало e_2 есть конец e_1.
- Найти кратчайший взвешенный пусть: минимизировать сумму весов ребер.

Пример



Лист принадлежности:

- Boston: Providence, New York
- Providence: Boston, New York
- New York: Chicago
- Chicago: Denver, Phoenix
- Denver: Phoenix, New York
- Los Angeles: Boston

Построение графа: код

```
def buildCityGraph():
        g = Digraph()
        for name in ('Boston', 'Providence', 'New York', 'Chicago’,
                     'Denver', 'Phoenix', 'Los Angeles'): #Create 7 nodes
                g.addNode(Node(name))
        g.addEdge(Edge(g.getNode('Boston'), g.getNode('Providence')))
        g.addEdge(Edge(g.getNode('Boston'), g.getNode('New York')))
        g.addEdge(Edge(g.getNode('Providence'), g.getNode('Boston')))
        g.addEdge(Edge(g.getNode('Providence'), g.getNode('New York')))
        g.addEdge(Edge(g.getNode('New York'), g.getNode('Chicago')))
        g.addEdge(Edge(g.getNode('Chicago'), g.getNode('Denver')))
        g.addEdge(Edge(g.getNode('Denver'), g.getNode('Phoenix')))
        g.addEdge(Edge(g.getNode('Denver'), g.getNode(’New York')))
        g.addEdge(Edge(g.getNode('Chicago'), g.getNode('Phoenix')))
        g.addEdge(Edge(g.getNode('Los Angeles'), g.getNode('Boston')))
```

Алгоритмы

- Алгоритм 1: «Поиск в глубину» (depth-first search, DFS)
- Алгоритм 2: «Поиск в ширину» (breadth-first Search, BFS)
- Второй алгоритм похож на обход дерева полного перебора. Основное отличие в том, что у графа возможны петли (циклы), поэтому нужно отслеживать посещенные узлы.

```
def DFS(graph, start, end, path, shortest):
      path = path + [start]
      if start == end:
            return path
      for node in graph.childrenOf(start):
            if node not in path: #avoid cycles
                  if shortest == None or len(path) < len(shortest):
                        newPath = DFS(graph, node, end, path,
                                      shortest, toPrint)
                        if newPath != None:
                              shortest = newPath
      return shortest
```

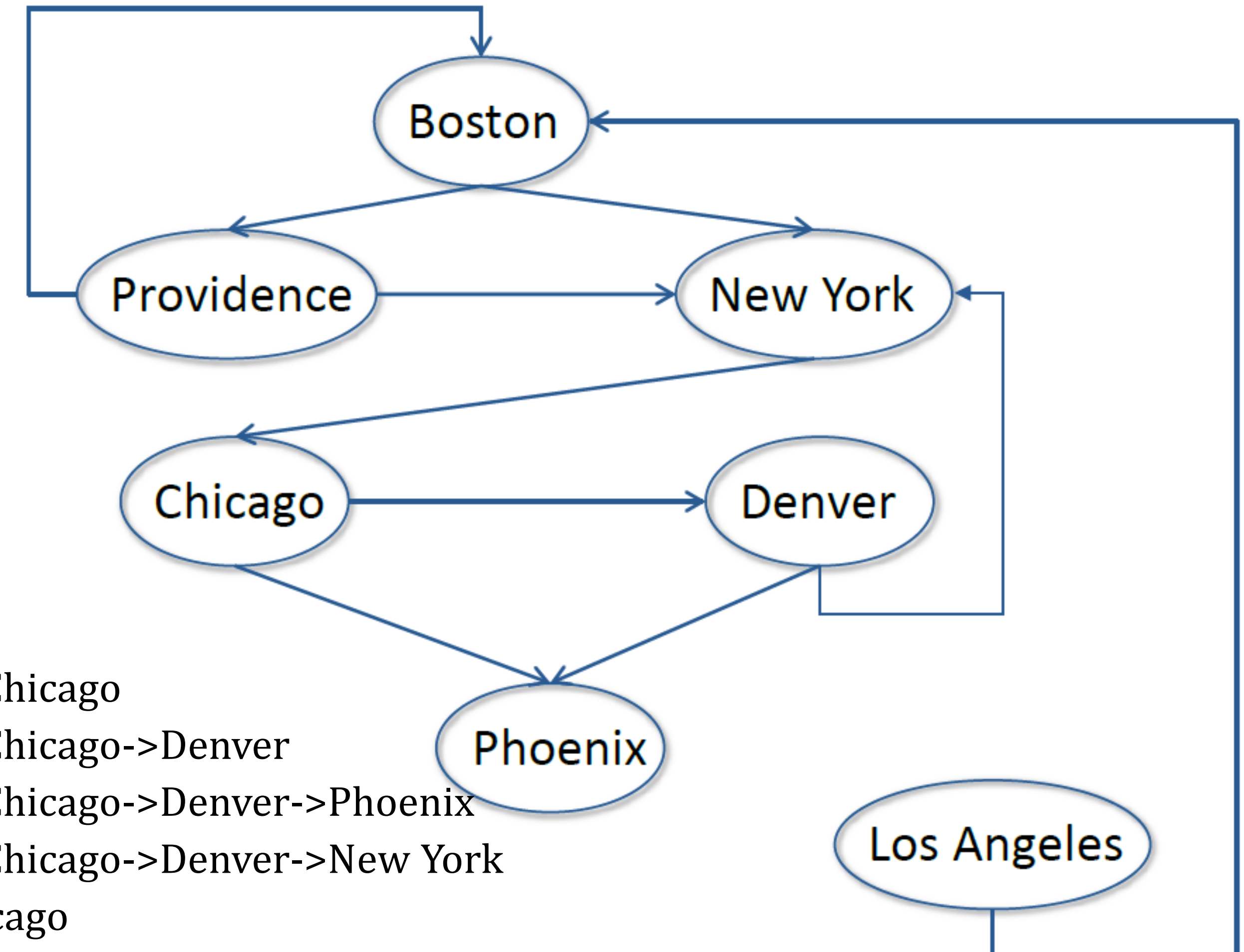
Depth First Search (продолжение)

```
def shortestPath(graph, start, end):
        return DFS(graph, start, end, [], None, toPrint)

def testSP(source, destination):
        g = buildGraph()
        sp = shortestPath(g, g.getNode(source), g.getNode(destination)
        if sp != None:
                print('Shortest path from', source, 'to’, destination, 'is’,
                        printPath(sp))
        else:
                print('There is no path from', source, 'to', destination)

testSP('Boston', ’Chicago')
```

DFS, результат (Chicago to Boston)



- Current DFS path: Chicago
- Current DFS path: Chicago->Denver
- Current DFS path: Chicago->Denver->Phoenix
- Current DFS path: Chicago->Denver->New York
- Already visited Chicago
- There is no path from Chicago to Boston

Breadth-first Search (BFS)

```
def BFS(graph, start, end, toPrint = False):
        initPath = [start]
        pathQueue = [initPath]
        if toPrint:
                print('Current BFS path:', printPath(pathQueue))
        while len(pathQueue) != 0:
                #Get and remove oldest element in pathQueue
                tmpPath = pathQueue.pop(0)
                print('Current BFS path:', printPath(tmpPath))
                lastNode = tmpPath[-1]
                if lastNode == end:
                        return tmpPath
                for nextNode in graph.childrenOf(lastNode):
                        if nextNode not in tmpPath:
                                newPath = tmpPath + [nextNode]
                                pathQueue.append(newPath)
        return None
```

Некоторые заключительные замечания

- Задача о нахождении минимального взвешенного пути на графе:
  - Необходимо минимизировать сумму весов ребер, а не количество узлов
  - «Поиск в глубину» (DFS) может быть легко модифицирован под эту задачу.
  - «Поиск в ширину» (BFS) нет, потому что кратчайший взвешенный путь может иметь больше, чем минимальное число переходов (узлов).
- Графы – это полезно
  - Лучший способ создать модель связи между объектами
  - Многие важные практические задачи могут быть сформулированы в терминах задач оптимизации на графах, для которых решение уже известно.
- Поиск в ширину и поиск в глубину – широко распространенные алгоритмы обхода графов, и могут быть использованы для решения широкого круга задач.