Дарвин vs Ламарк

Группа М06-006ск
Капицын В
Черенов А
Болтасов А
Уфимцев С

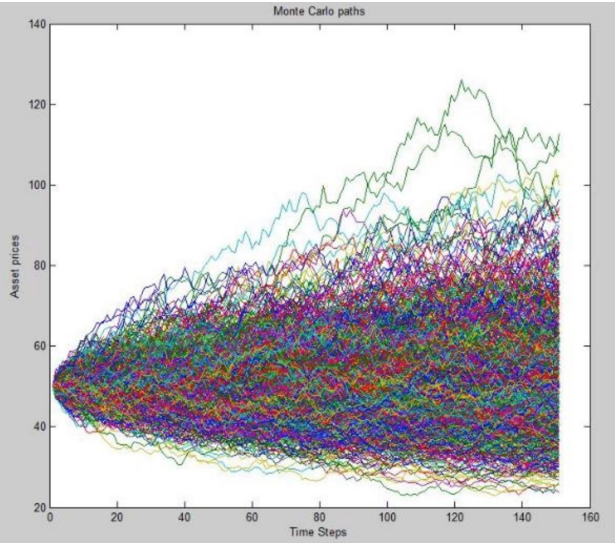
Содержание

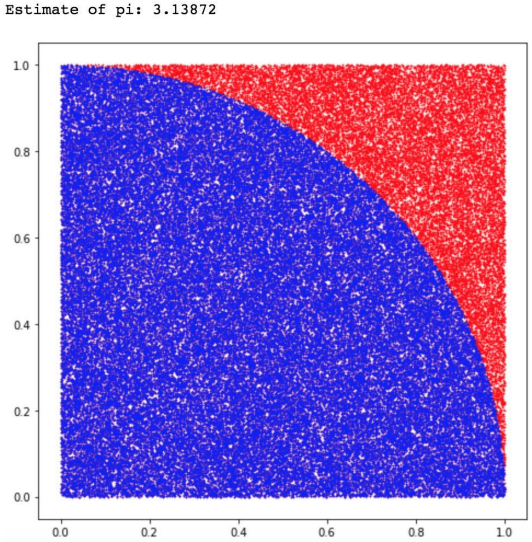
1. Про Марковские процессы и Монте-Карло
2. Сравнение на графе с 1024 узлами
    a. Дарвиновский алгоритм
    b. Муравьиный алгоритм
3. Сравнение в трёхмерном пространстве
    a. Дарвиновский алгоритм
    b. Муравьиный алгоритм

Про Марковские процессы и Монте-Карло

Ме́тоды Мо́нте-Ка́рло (ММК) — группа численных методов для изучения случайных процессов.

Суть метода заключается в следующем: процесс описывается математической моделью с использованием генератора случайных величин, модель многократно обсчитывается, на основе полученных данных вычисляются вероятностные характеристики рассматриваемого процесса.





Марковские и немарковские процессы

Ма́рковский проце́сс — случайный процесс, эволюция которого после любого заданного значения временно́го параметра ***t*** ***не зависит*** от эволюции, предшествовавшей ***t***, при условии, что значение процесса в этот момент фиксировано.

AR(1) процессы, Винеровский процесс и т.д.

Нема́рковский проце́сс — случайный процесс, эволюция которого после любого заданного значения времени ***t*** **зависит** от эволюции, предшествовавшей этому моменту времени.

Броуновское движение, цены акций и т.д.

Используемые инструменты:

NetworkX
Network Analysis in Python

Сравнение на графе с 1024 узлами

Алгоритм Дарвина на Графе

Считаем что точка A - это G[0], точка B - это G[nodes-1].

0) Стартовая популция - число частиц в точке A. Задается, как гиперпараметр n_start. Частица имеет следюущую структуру:

> coord - координаты местонахожления (вершина графа)
> w - вес частицы (стартовое значение 1)
> r - расстояние (длина кратчайшего пути от coord до точки B)

1) Кажда частица совершает случайные блуждания по маршруту n_iterations эпох, пока не достигнет цели(точки B) или не "умрет":

> 1.1) Совершаем max_item_iterations случайных переходов частицей или пока частица не достигнем цели.
> 1.2) Если частица достигла цели, инкрементируем число достигших цели точек и "убиваем" частицу. Если частица не достигла цели, пересчитываем её вес.
> 1.3) Вес рассчитывается, как:
>
> $$w_i = w_{i-1} \times \left(\frac{e^{-r_i}}{r_i^2} \div \frac{e^{-r_{i-1}}}{r_{i-1}^2} \right)$$
>
> где i - текущая эпоха,
> r - расстояние от текущей точки до B
> e - константа ~2.72
> 1.4) Если новый w(вес частицы) < 1, с вероятностью 1-w "убиваем" частицу.
> 1.5) Если новый w(вес частицы) >= 2, расщепляем частицу на floor(w), с весами w/floor(w).

2) Завершаем алгоритм, если все выжившие частицы достигли цели или если все частицы погибли или пройден лимит эпох.
3) Выводим число пройденных эпох, число частиц достигших цели, время выполнения алгоритма.

Фрагменты кода

Инициализация алгоритма

```
colony = DarvinColony(g, 25, 20, 100, True)
colony.run()
```

Метод “run()” запускает алгоритм с заданными параметрами

```
def run(self):
  self.time_start = datetime.datetime.now()
  best_way = len(nx.dijkstra_path(self.graph, 0, len(g.nodes)-1)) # Кратчайший путь от A до B
  self.genQuantums(best_way)

  self.targetIterations = self.n_iterations
  for i in range(0, self.n_iterations):
    #print("---- Итерация #{} ----".format(i))
    self.gen_all_paths() #  Каждая частица совершает случайные блуждания (Шаги 1.1-1.2)
    self.recalcWeight() # Пересчет весов (Шаг 1.3)
    self.kill() # Убийство частиц с учетом весов (Шаг 1.4)
    self.reproduction() # Размножение чатиц с учетом весов (Шаг 1.5)
    if len(self.quantums) == 0:
      self.targetIterations = i
      break
  self.time_end = datetime.datetime.now()
```

Метод “recalcWeight()” - рассчитывает веса перед завершением эпохи

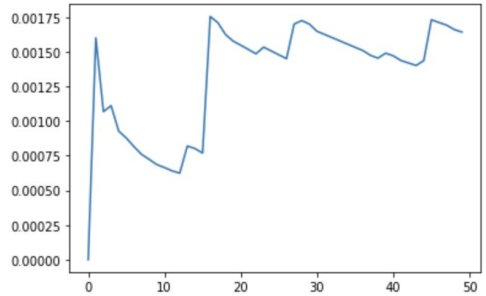
```
def recalcWeight(self):
  for i in range(0, len(self.quantums)):
    prev_w = self.quantums[i]['w']
    prev_r = self.quantums[i]['r']
    prev_f = (2.72**((-1)*prev_r)) / (prev_r * prev_r) # фитнес функция на предыдущем шаге

    self.quantums[i]['r'] = len(nx.dijkstra_path(self.graph, self.quantums[i]['coord'], len(g.nodes)-1)) #
    next_f = (2.72**((-1)*self.quantums[i]['r'])) / (self.quantums[i]['r'] * self.quantums[i]['r']) # новая
    self.quantums[i]['w'] = prev_w * (next_f/prev_f)
```

Испытания Дарвина на графе

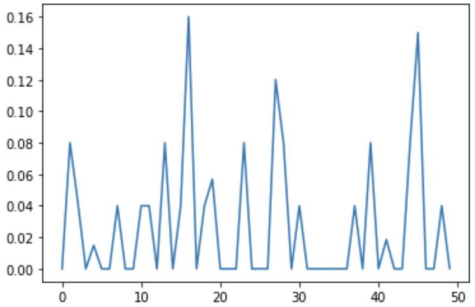
Гиперпараметры 50 запусков:

Ограничение эпох: 20
Ограничение шагов частицы за эпоху: 50
Число частиц на старте: 25

Дисперсия весов на 50 запусках



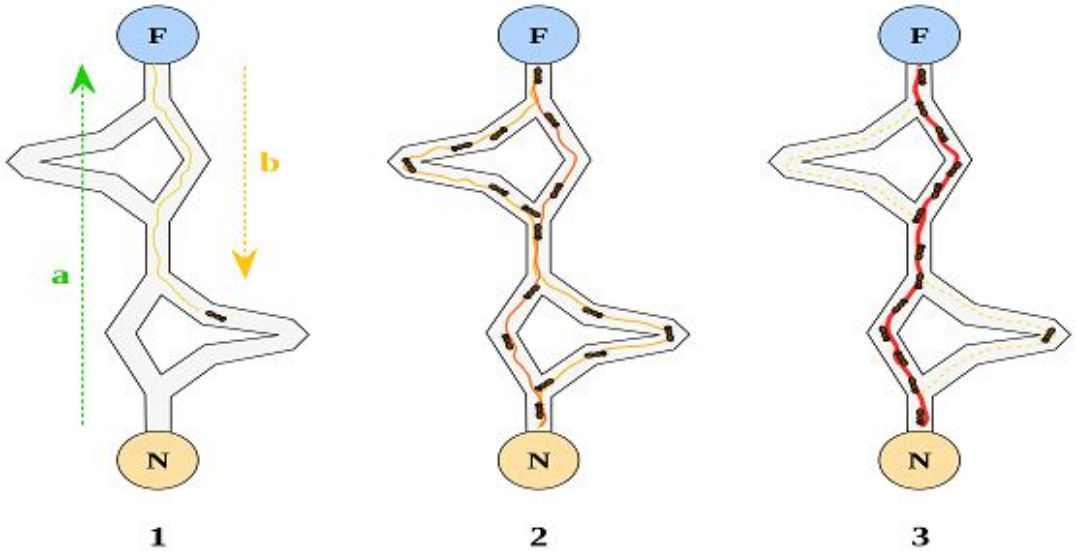
Вес на 50 запусках




```
Средн: 0.027253277558218617
Среднее время: 44.28231522
FOM: 17.041636349995144
```

$$FOM = \frac{1}{D * T}$$

Муравьиный алгоритм



Муравьиный алгоритм на графе

Алгоритм состоит из N итераций. Кол-во итераций задаётся в параметр n_iterations.

Считаем что точка A - это G[0], точка B - это G[nodes-1].

Есть K муравьёв. Кажда частица совершает случайные блуждания по маршруту n_iterations раз, то есть K раз происходит проход по графу для каждой итерации

Содержание каждой итерации:

1. Выпускается K муравьёв (задаётся параметром n_ants) в функции **gen_all_paths()**
2. Собираются путь каждого муравья (ф-я **gen_path()**) при том, что муравей выбирает, куда ему идти в функции **pick_move()**
    1. Если муравей достигает точки B, то его путь заканчивается
    2. Если муравей превышает число шагов, заданный параметром **max_moves**, то он "умирает" и его путь не защитывается
3. Для каждого собранного пути считается его дистанция (ф-я **gen_path_dist()**)
4. В зависимости от длинны дистанции выбираются **n_best** самых успешных дистанций. На них распыляется феромон (ф-я **spread_pheromone()**)
5. После этого происходит испарение феромона со всех рёбер графа, заданное параметром **decay**

$$Wnew = Wold * \frac{\frac{1}{Nways}}{f(i)}$$

Веса муравьёв обновляются при каждом переходе и считаются, как:

Thanks to
https://github.com/Akavall/AntColonyOptimization

Фрагменты кода

Инициализация алгоритма

```
ants = AntColony(G, n_ants = 35, n_best = 5, n_iterations = 20, decay = 0.95, max_moves = 100)
```

```
def run(self):
  shortest_path = None
  all_time_shortest_path = ("placeholder", np.inf)
  for i in range(self.n_iterations):
      self.weights = 0
      all_paths = self.gen_all_paths() #получаем все пути муравьёв за эту итерацию
      self.spread_pheronome(all_paths, self.n_best, shortest_path=shortest_path) #распыляем феромоны
      shortest_path = min(all_paths, key=lambda x: x[1]) #находим кратчайший путь из всех муравьиных

      #если нашли более короткий путь, чем был раньше, то заменяем
      if shortest_path[1] < all_time_shortest_path[1]:
          all_time_shortest_path = shortest_path
      self.pheromone = self.pheromone * self.decay #испаряем феромон
  self.min_path = shortest_path[1]
  print("shortest:", self.min_path, "weights:", self.weights)
  return all_time_shortest_path
```

Метод “run()” запускает алгоритм с заданными параметрами

Муравей выбирает путь и теряет вес

```
np.random.seed()
move = np_choice(a = neighbours, size = 1, p=norm_row)[0] # волшебство выбора
new_weight = old_weight / (len(neighbours) * norm_row[np.where(neighbours == move)])

return move, new_weight
```

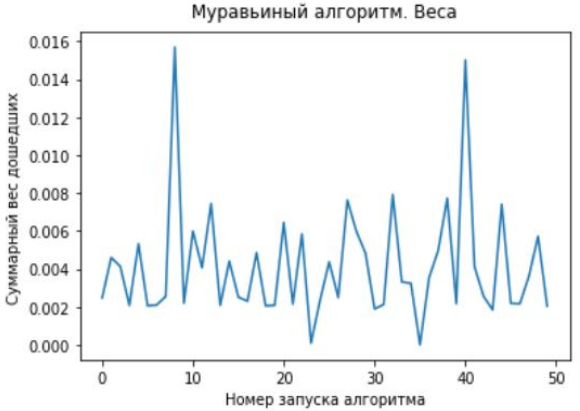
Испытание Ламарка на графе

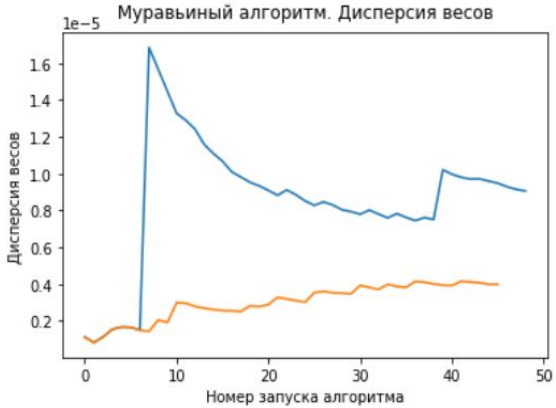
Гиперпараметры 50 запусков алгоритма:

Количество муравьёв: 35
Путей, по которым распыляют феромон: 25 лучших
Количество итераций(эпох): 20
Уровень испарения феромона: 0.95
Ограничение шагов муравья за итерацию: 500

```
Средний вес: 0.004135616631417681
Среднее время: 6.079779272079468
FOM1: 19289.651696559402
FOM2: 54429.643814767645
```

$$FOM = \frac{1}{D * T}$$





Выводы по сравнению на графе

Муравьиный алгоритм на несколько порядков увеличил FOM по сравнению с генетическим

С чем это связано:

- **отсутствие сложных операций**
  - в дарвиновском алгоритме - dijkstra для каждой частицы
  - в муравьином - распыление феромона на n_best путей (умножение) и умножение матрицы
- **ускорение работы алгоритма по мере обучения**
  - чем дальше муравьиный учится, тем короче блуждания и быстрее алгоритм
- **выбор на каждом шаге частицы**

Сравнение в трёхмерном пространстве

Краткое описание кода

def get_darwin_estimate(n_points, pogl_distance_min, pogl_distance_max):

Базовая функция, запускающая Дарвиновскую симуляцию с начальным количеством частиц *n_points* и лимитом расстояния между источником и поглотителем.

Structure of returned dictionary - total:

Start_time - time of function launch: datetime object

Finish_time - time of function end (excluding plotting of paths): datetime object

pogl - coordinates of Absorber: [x, y, z]

paths - array with paths: *[[[x00, x01, ...], [y00, y01, ...], [z00, z01, ...]], [[x10, x11, ...], [y10, y11, ...], [...]], ...]*

number of int type - object of type: *[[weight0, weight1, ...], [time0, time1, ...]]* of points, which achieved Absorber in *number* steps

Краткое описание кода

def get_lamark_estimate_v2(darwin_results):

Функция для запуска симуляции алгоритма Ламарка. На вход принимается словарь, полученный из симуляции алгоритма Дарвина.

Structure of returned dictionary - total:

Start_time - time of function launch: datetime object

Finish_time - time of function end (excluding plotting of paths): datetime object

number of int type - object of type: *[[weight0, weight1, ...]*, *[time0, time1, ...]]* of points, which achieved Absorber in *number* steps

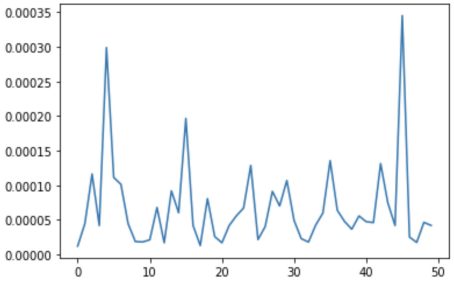
Испытания Дарвина в Трехмерном пространстве

Гиперпарамерты: 50 запусков

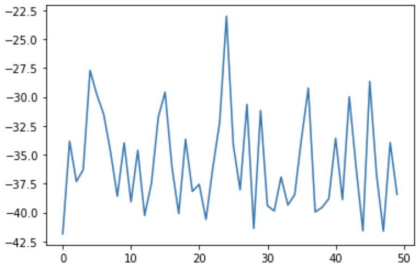
Мин. расстояние от частицы до поглотителя: 5
Макс, расстояние от частицы до поглотителя : 6
Число частиц на старте: 100

Средний вес: `1.7796216782810938e-05`
Среднее время: `12.560156`
FOM: `69.05340676664457`

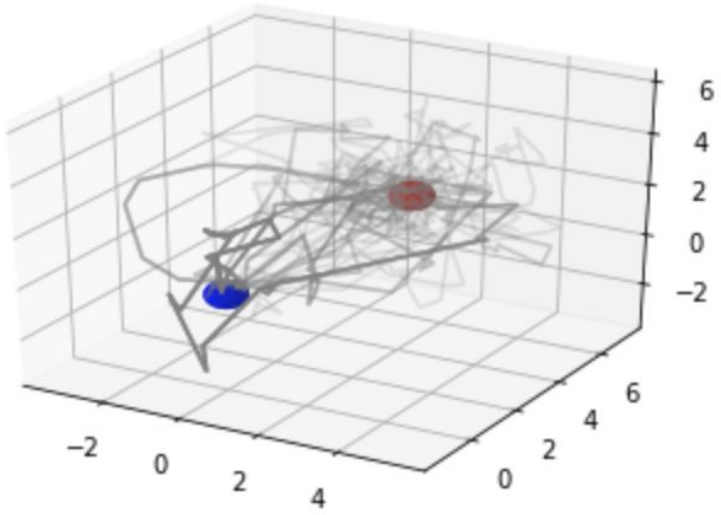
Веса



Дисперсия весов



Запуск Дарвина в 3д пространстве в разбивке по длиннам путей



Исходный код

https://github.com/VKapicyn/m06-006sk-course-work-var2

Наша команда

Сережа Уфимцев

Владислав Капицын

Александр Болтасов

Алексей Черенов

Благодарим за внимание!