Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет информационных технологий и управления

ОТЧЕТ

по лабораторной работе № 2

на тему: «Проектирование компонента решателя задач на основе многоагентных моделей решения задач»

Выполнил:

Студент группы 426401

Махмудов Асадулло Кобилжон угли

Проверил:

Минск 2025

**Выбор класса задач**

Для многоагентной модели решателя задач был выбран класс задач координации многопользовательской системы распределённой доставки (например, нескольких мобильных роботов‑курьеров, которые совместно доставляют грузы по разным точкам города).

Почему именно этот класс:

1. Распределённость среды. Каждый «агент‑курьер» действует в своей локальной области и имеет ограничённую информацию о положении других агентов и о загруженности дорог. Многоагентная архитектура позволяет описать каждого робота как самостоятельного интеллектуального агента, принимающего локальные решения.
2. Необходимость координации. Задача оптимизации маршрутов и распределения грузов между агентами требует обмена информацией (о свободных грузах, о времени прибытия к точке, о пробках). Модель взаимодействия через ACL‑сообщения, общую «доску объявлений» или систему договоров (market‑based) естественно реализуется в многоагентном подходе.
3. Параллельность и масштабируемость. По мере роста количества точек доставки и числа курьеров централизованный алгоритм становится узким местом. Многоагентная система легко масштабируется: каждый новый курьер (агент) добавляется к общей инфраструктуре, сохраняя локальную логику принятия решений и взаимодействия.
4. Широкая область применения. Подобные задачи встречаются в логистике (дроны‑доставщики, роботы‑курьеры), в интеллектуальных транспортных системах и «умных» складах. Поэтому опыт проектирования многоагентного решателя на примере доставки будет актуален и легко транслируется на другие предметные области.

Таким образом, выбрана задача совместной оптимизации маршрутов распределённой группы агентов‑курьеров, поскольку она демонстрирует ключевые преимущества МАС: децентрализованное принятие решений, координацию через обмен сообщениями и адаптивность в динамичной среде.

**Постановка задачи**

Система состоит из множества NN агентов‑курьеров A={a1,a2,…,aN}, каждый из которых перемещается по дорожной сети города и может доставлять грузы между точками. Набор задач (заказов) T={t1,t2,…,tM} задан заранее: у каждого заказа t\_j указана точка погрузки p\_j, точка разгрузки q\_j и требуемый промежуток времени [e\_j, l\_j], в который доставка должна быть совершена.

Каждый агент a\_i обладает своими текущими координатами x\_i и скоростью ν\_i. Известны оценки времени d(u,v) (или расстояния) между любыми двумя точками u,v на карте.

**Цель:** совместно (координированно) спланировать для каждого агента π\_i - последовательность заказов πi=(ti1,ti2,… ) и маршрутов между их точками погрузки/разгрузки, чтобы

1. все заказы T были выполнены (каждый заказ назначен и доставлен ровно одним агентом) при соблюдении таймаутов [e\_j, l\_j],
2. суммарное время выполнения (или общий пройденный путь) по всем агентам минимизировалось,
3. отсутствовали конфликты: два агента не должны одновременно занимать один и тот же участок дороги в противоположных направлениях (предотвращение коллизий), а объём груза у каждого агента не превышал его грузоподъёмности.

**Ограничения и допущения:**

* Среда динамична: заказы могут поступать во время работы, дорожная сеть может испытывать локальные пробки, поэтому принято периодически перестраивать план (реактивная корректировка).
* Каждый агент видит состояние своих ближайших соседей (локально ограниченная информация), но обмен данными возможен через сообщения (P2P или общая «доска объявлений»).
* Агент не знает заранее все будущие заказы, но может получить актуальный список доступных заказов в начале каждого планирования.

Таким образом, задача формулируется как комбинаторная оптимизация с распределённым принятием решений. Каждый агент должен выбрать, какие заказы из текущего пула взять на себя, и в каком порядке их выполнить, обменявшись с остальными агентами необходимой информацией для избежания конфликтов. При этом необходимо достигнуть глобальной сходимости к близкому к оптимальному распределению и маршрутам, минимизирующим общее время (или путь).

**Алгоритм решения**

Ниже приведён общий подход к распределённому планированию маршрутов и назначению заказов между агентами‑курьерами. Алгоритм сочетает в себе локальное планирование (для каждого агента) и механизм координации (обмен сообщениями при назначении заказов и предотвращении конфликтов).

**Инициализация**

1. Инициализация агентов

Каждый агент a\_i инициализируется со своим положением x\_i, скоростью v\_i и грузоподъёмностью C\_i.

Агент сохраняет «локальный чёрный список» занятых участков дорог (например, текущие брони дорожных отрезков на ближайшее время).

2. Загрузка пула заказов

В начале планирования (или при «перепланировке») все агенты получают (или считывают с общей «доски объявлений») актуальный список невыполненных заказов

T = {t\_j = (p\_j,q\_j, [e\_j,l\_j],w\_j) | j = 1..M},

где w\_j - объём груза, p\_j и q\_j - координаты погрузки/разгрузки, [e\_j,l\_j] - допустимый интервал сдачи.

3. Построение локального представления

Каждый агент формирует свою копию графа дорог (со взвешенными рёбрами по времени/расстоянию), а также оценивает «локальные» состояния (ближайшие пробки, занятые участки) в радиусе своей видимости.

**Распределённое назначение заказов (аукционный протокол)**

Для того чтобы минимизировать общее время/расстояние и распределить заказы между агентами, используется двухэтапный механизм:

1. Подача «заявок на заказы» (bid) - шаг аукциона

Каждый агент a\_i просчитывает для каждого свободного заказа t\_j (то есть ещё не назначенного) «локальную оценку» своей выгоды (затрат) при его выполнении. Оценка α\_i,j может включать:

1. Расстояние/время от текущей позиции x\_i до точки p\_j и далее до q\_j:

cost\_i,j = d(x\_i,p\_j) + d(p\_j,q\_j)

1. Учёт временных окон [e\_j,l\_j]: переходит ли агент успеет уложиться? Если прогнозируемое\_время > l\_j, то α\_i,j = + ∞ (не принимаем заявку).
2. Занятость грузового объёма: если w\_j > C\_i, агент не участвует в аукционе по этому заказу.

Далее агент формирует список {(t\_j,α\_i,j) и отправляет «заявку» (bid) на каждый подходящий заказ на общую «доску объявлений» или напрямую координирует с соседними агентами (Peer‑to‑Peer).

2. Выбор победителя аукциона для каждого заказа

Для каждого заказа t\_j агрегируется полученные от агентов заявки α\_i,j\_i=1..N. Побеждает тот агент a\_p, у которого α\_p,j = min\_i α\_i,j.

Обычно аукцион идёт раундами:

1. Каждый агент рассылает свои bids.

2. Когда поступают все bids (или истекает таймаут), каждый агент - либо «майстер» аукциона, либо договорённость о едином координаторе - называет победителя и публикует решение (назначение заказа t\_j → a\_p).

После каждого раунда назначенные заказы удаляются из пула свободных.

3. Обновление локальных планов

В конце раунда каждый агент получает список назначенных ему заказов π\_i ⊂ T.

Агент планирует последовательный маршрут: упорядочивает {t\_j ∈π\_i} в порядке возрастания e\_j (начало интервала) или на основе жадного минимума cost\_i,j.

Впоследствии, по мере приближения времени, агент может скорректировать порядок.

**Локальное планирование маршрута**

Для каждого агента a\_i после получения списка π\_i = (t\_{i\_1}, t\_{i\_2}, …) нужно проложить оптимальный маршрут, учитывая дорожную сеть и занятые участки. Последовательность действий:

1. Определение последовательности точек

Пусть текущая позиция агента - x\_i.

Агент формирует упорядоченный (временными рамками [e\_j,l\_j] или минимальным дополнительным расстоянием) список промежуточных точек:

[x\_i,p\_{i\_1},q\_{i\_1},p\_{i\_2},q\_{i\_2}, …].

Возможна жадная сортировка: выбираем следующий заказ ∈ π\_i с минимальным d(x\_i,p\_)+d(p\_,q\_), обновляем x\_i ← q\_, повторяем для оставшихся.

2. Поиск кратчайших путей между точками

Для каждого пары (u, v) в последовательности агент запускает локальный алгоритм поиска кратчайшего пути (например, Dijkstra или A) в своей локальной копии графа, при этом исключая участки, занятые сообщениями других агентов (обозначены как «забронированы» на ближайший интервал).

В результате получается последовательность P\_k, где P\_k - список узлов/рёбер от точки u до v.

Совокупный маршрут - конкатенация всех P\_k.

3. Составление расписания (тайминговая часть)

На основе скорости v\_i и весов рёбер (время прохождения) для каждого сегмента P\_k агент вычисляет прогноз времени прибытия в каждую точку (погрузки/разгрузки).

Если прогнозируемое время прибытия к точке q\_j превышает l\_j (позднее разрешённого интервала), агент отмечает конфликт и помещает соответствующий заказ t\_j в «перепланировку».

**Обнаружение конфликтов и динамическое перепланирование**

Поскольку среда динамична и у каждого агента информация может устаревать, необходимо периодически проверять план и корректировать его.

1. Динамические события

Новые заказы поступают в любой момент. При их появлении запускается новый раунд аукциона.

Изменение дорожной обстановки (локальные пробки, аварии) отмечается в локальном представлении графа. Агент, обнаруживший пробку, может поднять «сигнал» о занятости участков, и соседние агенты пересчитают маршруты.

Обновление статуса соседних агентов (они сами объявляют, что заняли те или иные дорожные сегменты на будущий интервал); позволяет избежать столкновений.

2. Триггеры перепланировки

Если в ходе следования по маршруту агент видит, что прогнозируемое время приезда к запланированной точке погрузки e\_j превышено или близко к l\_j, он «откладывает» текущий заказ, пересылая запрос на «перевыставление» (re‑bid) другим агентам.

При поступлении новых заявок каждый агент проверяет, не появился ли заказ, который он может взять выгоднее (меньшие затраты). В этом случае он «сдаёт» менее приоритетный заказ (если успевает его завершить) и участвует в новом аукционе.

3. Перепланировка локального маршрута

Агент составляет новый список π\_i (с учётом сданных и добавленных заказов).

При каждом поиске маршрута заново исключаются уже «забронированные» (имеющие приоритет) участки у соседей.

4. Избежание коллизий

Каждый агент, прежде чем «бронировать» отрезок дороги (ребро графа) на интервал t\_start, t\_end, посылает своим ближайшим соседям сообщение «Request\_Reserve(edge, t\_start, t\_end)».

Все соседи, у которых в этом же интервале стоит бронь на тот же «edge» в противоположном направлении, отвечают «Reject». Агент ищет альтернативный путь. Если никто не занят, соседи отвечают «Accept», и агент «резервирует» этот отрезок.

**Итоговый порядок действий агентов**

1. Получение начальных заказов - формируем общий пул T.

2. Распределённый аукцион - назначаем заказы между агентами.

3. Локальное планирование - каждый агент строит маршрут по своим заказам, бронируя участки дорог.

4. Исполнение маршрута - агент движется по текущему маршруту, следуя своей скорости v\_i; при приближении к каждому «node» выполняет нужные действия (погрузка/разгрузка).

5. Мониторинг и перепланировка - при изменении дорожных условий, появлении новых заказов или угрозе нарушения временных окон агент запускает новый аукционный раунд (либо обновляет текущий список локально).

6. Завершение задачи - когда все заказы T распределены и выполнены, все агенты переходят в ждущий режим до поступления новых заказов.

**Особенности и преимущества алгоритма**

1. Децентрализация. Нет единого «мастера», все агенты равноправно участвуют в аукционах и локальном планировании.
2. Гибкость. При каждом изменении (новые заказы, пробки) запускается локальная перепланировка, не требующая полной переработки всего расписания.
3. Устойчивость к отказам. Если один агент выходит из строя, его незавершённые заказы попадают обратно в пул и переаукционируются между оставшимися.
4. Параллельность. Каждый агент умеет одновременно строить свой маршрут и бронировать отрезки дорог, не дожидаясь «сигнала» от централизованной системы.

**Вывод:**

Предложенный многоагентный алгоритм сочетает в себе периодические аукционные раунды для назначения заказов и непрерывное локальное планирование маршрутов с бронированием участков дорог. Это обеспечивает сбалансированное заполнение курьерского пула, позволяет минимизировать общее время/путь и гарантирует отсутствие столкновений при работе нескольких агентов в одной дорожной сети.

**Решение**

**Структура проекта**

multiagent-solver/

├── requirements.txt

├── app/

│ └── main.py

└── src/

├── config.py

├── graph.py

├── environment.py

├── auction.py

├── agent.py

└── simulation.py

requirements.txt - список Python‑библиотек (если понадобятся сторонние пакеты).

app/main.py - точка входа (запускает симуляцию).

src/config.py - глобальные параметры (число агентов, тайминги, настройки аукциона).

src/graph.py - класс Graph и алгоритм Дейкстры (из лабораторной 1, чуть адаптирован).

src/environment.py - описание дорожной сети, динамика (например, блокировки дорог).

src/auction.py - логика аукциона: сбор заявок (bids) и назначение заказов.

src/agent.py - класс Agent: хранит состояние агента, методы bid/plan/execute.

src/simulation.py - главный цикл симуляции: время, передача сообщений, триггеры перепланировки.

**Запуск**

1. Установите Python 3.x (если ещё не установлен).

2. Создайте виртуальное окружение и активируйте его:

python -m venv venv

source venv/bin/activate # Linux/MacOS

venv\Scripts\activate # Windows

3. (Если потребуется) установите зависимости:

pip install -r requirements.txt

4. Перейдите в папку проекта и запустите:

python -m app.main

**Код файла config.py**

В этом файле задаются все глобальные параметры симуляции: число агентов, количество генерируемых заказов, длительность симуляции, параметры аукциона, диапазоны координат, грузоподъёмности и скорости агентов, а также фиксируется фиксированный seed для детерминированной генерации случайных величин.

import random

NUM\_AGENTS = 5

INITIAL\_ORDERS = 10

SIMULATION\_END\_TIME = 300

AUCTION\_INTERVAL = 20

BID\_TIMEOUT = 2

ACCEPT\_TIMEOUT = 1

MAX\_PICKUP\_COORD = 10

MAX\_DELIVERY\_COORD = 10

MAX\_TIME\_WINDOW = 100

MIN\_TIME\_WINDOW = 10

MIN\_CAPACITY = 5

MAX\_CAPACITY = 15

MIN\_SPEED = 1.0

MAX\_SPEED = 2.0

RANDOM\_SEED = 42

random.seed(RANDOM\_SEED)

**Код файла graph.py**

Реализован класс Graph, который хранит взвешенный неориентированный граф (списки смежности) и позволяет находить кратчайшие пути с помощью алгоритма Дейкстры. Доступны методы для добавления рёбер и восстановления самого маршрута.

import heapq

class Graph:

def \_\_init\_\_(self):

self.adj = {}

def add\_edge(self, u, v, w):

if u not in self.adj:

self.adj[u] = []

if v not in self.adj:

self.adj[v] = []

self.adj[u].append((v, w))

self.adj[v].append((u, w))

def neighbors(self, u):

return self.adj.get(u, [])

def vertices(self):

return list(self.adj.keys())

def dijkstra(self, start):

INF = float('inf')

dist = {v: INF for v in self.adj}

prev = {v: None for v in self.adj}

dist[start] = 0

heap = [(0, start)]

while heap:

d\_u, u = heapq.heappop(heap)

if d\_u > dist[u]:

continue

for v, w in self.neighbors(u):

alt = d\_u + w

if alt < dist[v]:

dist[v] = alt

prev[v] = u

heapq.heappush(heap, (alt, v))

return dist, prev

def shortest\_path(self, start, target):

dist, prev = self.dijkstra(start)

if dist.get(target, float('inf')) == float('inf'):

return None, float('inf')

path = []

cur = target

while cur is not None:

path.append(cur)

cur = prev[cur]

path.reverse()

return path, dist[target]

**Код файла** **environment.py**

Здесь создаётся квадратная сетка (величина по X и Y определяется в config.py) и хранится словарь координат и обратный словарь. Поддерживается динамическая блокировка рёбер (список blocked\_edges), а также поиск кратчайшего пути с учётом текущих блокировок.

import random

from typing import List, Tuple

from src.graph import Graph

from src.config import MAX\_PICKUP\_COORD, MAX\_DELIVERY\_COORD

class Environment:

def \_\_init\_\_(self):

self.graph = Graph()

self.coordinates = []

self.\_build\_grid\_graph()

self.blocked\_edges: List[Tuple[Tuple[int,int], float, float]] = []

def \_build\_grid\_graph(self):

size\_x = MAX\_PICKUP\_COORD + 1

size\_y = MAX\_DELIVERY\_COORD + 1

self.coord\_map = {}

self.reverse\_map = {}

vertex\_id = 0

for x in range(size\_x):

for y in range(size\_y):

self.coord\_map[vertex\_id] = (x, y)

self.reverse\_map[(x, y)] = vertex\_id

self.coordinates.append((x, y))

vertex\_id += 1

for vid, (x, y) in self.coord\_map.items():

if x + 1 < size\_x:

u = vid

v = self.reverse\_map[(x + 1, y)]

self.graph.add\_edge(u, v, 1)

if y + 1 < size\_y:

u = vid

v = self.reverse\_map[(x, y + 1)]

self.graph.add\_edge(u, v, 1)

def random\_block\_edge(self, current\_time: float, duration: float = 10.0):

all\_edges = []

for u in self.graph.adj:

for v, \_ in self.graph.adj[u]:

if u < v:

all\_edges.append((u, v))

if not all\_edges:

return

edge = random.choice(all\_edges)

self.blocked\_edges.append((edge, current\_time, current\_time + duration))

def is\_edge\_blocked(self, u: int, v: int, time: float) -> bool:

e = (u, v) if u < v else (v, u)

for (edge\_uv, t\_start, t\_end) in self.blocked\_edges:

if edge\_uv == e and t\_start <= time <= t\_end:

return True

return False

def get\_neighbors(self, u: int, time: float) -> List[Tuple[int, float]]:

nbrs = []

for v, w in self.graph.neighbors(u):

if not self.is\_edge\_blocked(u, v, time):

nbrs.append((v, w))

return nbrs

def shortest\_path(self, start: int, target: int, time: float):

INF = float('inf')

dist = {v: INF for v in self.graph.adj}

prev = {v: None for v in self.graph.adj}

dist[start] = 0

import heapq

heap = [(0, start)]

while heap:

d\_u, u = heapq.heappop(heap)

if d\_u > dist[u]:

continue

if u == target:

break

for v, w in self.get\_neighbors(u, time):

alt = d\_u + w

if alt < dist[v]:

dist[v] = alt

prev[v] = u

heapq.heappush(heap, (alt, v))

if dist[target] == INF:

return None, INF

path = []

cur = target

while cur is not None:

path.append(cur)

cur = prev[cur]

path.reverse()

return path, dist[target]

**Код файла** **auction.py**

Модуль, реализующий простой распределённый аукцион. Хранит заявки (bids) и по окончании таймаута выбирает для каждого заказа (task\_id) агента с минимальной ставкой. Результаты доступны через словарь assignment.

from threading import Event

from collections import defaultdict

import time

class Auction:

def \_\_init\_\_(self):

self.bids = defaultdict(list)

self.assignment = {}

self.round\_complete = Event()

def submit\_bid(self, agent\_id: int, task\_id: int, bid\_value: float):

bt = time.time()

self.bids[task\_id].append((agent\_id, bid\_value, bt))

def run\_auction\_round(self, task\_ids: list, timeout: float):

time.sleep(timeout)

for t in task\_ids:

bids\_for\_t = self.bids.get(t, [])

if not bids\_for\_t:

continue

winner = min(bids\_for\_t, key=lambda x: x[1])

self.assignment[t] = winner[0]

self.round\_complete.set()

def get\_assignment(self, task\_id: int):

return self.assignment.get(task\_id, None)

def reset\_round(self):

self.bids.clear()

self.assignment.clear()

if self.round\_complete.is\_set():

self.round\_complete.clear()

**Код файла** **agent.py**

Класс Agent, наследующийся от threading.Thread. Каждый агент хранит своё текущее положение, скорость, грузоподъёмность, список назначенных задач, локальный план и текущее состояние маршрута. Внутри run() происходят циклы аукционных раундов, построения маршрутов, исполнения и мониторинга дедлайнов. Когда агент доезжает до точки доставки, он помечает задачу как выполненную.

import threading

import time

from random import uniform, choice

from src.config import (MIN\_SPEED, MAX\_SPEED, MIN\_CAPACITY, MAX\_CAPACITY,

AUCTION\_INTERVAL, BID\_TIMEOUT, ACCEPT\_TIMEOUT)

from src.environment import Environment

class Agent(threading.Thread):

def \_\_init\_\_(self, agent\_id: int, env: Environment, auction, all\_tasks: dict, sim\_start\_time: float):

super().\_\_init\_\_()

self.agent\_id = agent\_id

self.env = env

self.auction = auction

self.all\_tasks = all\_tasks

self.sim\_start\_time = sim\_start\_time

self.position = self.random\_start\_position()

self.speed = uniform(MIN\_SPEED, MAX\_SPEED)

self.capacity = int(uniform(MIN\_CAPACITY, MAX\_CAPACITY))

self.current\_load = 0

self.assigned\_tasks = []

self.plan = []

self.current\_route = []

self.route\_index = 0

self.completed\_tasks\_count = 0

self.locked\_edges = {}

self.stop\_event = threading.Event()

def random\_start\_position(self):

verts = list(self.env.graph.vertices())

return choice(verts) if verts else 0

def run(self):

next\_auction\_time = time.time()

while not self.stop\_event.is\_set():

now\_real = time.time()

now\_sim = now\_real - self.sim\_start\_time

if now\_real >= next\_auction\_time:

self.execute\_auction()

next\_auction\_time = now\_real + AUCTION\_INTERVAL

if self.plan:

self.follow\_route(now\_real)

self.monitor\_plan(now\_real)

time.sleep(0.1)

def execute\_auction(self):

free\_tasks = [t\_id for t\_id, info in self.all\_tasks.items()

if info.get('assigned\_to') is None]

for t\_id in free\_tasks:

cost = self.estimate\_cost(t\_id)

if cost < float('inf'):

self.auction.submit\_bid(self.agent\_id, t\_id, cost)

time.sleep(BID\_TIMEOUT)

for t\_id in free\_tasks:

winner = self.auction.get\_assignment(t\_id)

if winner == self.agent\_id:

self.all\_tasks[t\_id]['assigned\_to'] = self.agent\_id

self.assigned\_tasks.append(t\_id)

time.sleep(ACCEPT\_TIMEOUT)

self.plan\_tasks()

def estimate\_cost(self, task\_id: int) -> float:

task = self.all\_tasks[task\_id]

pickup = task['pickup\_node']

delivery = task['delivery\_node']

e\_j = task['e']

l\_j = task['l']

w\_j = task['w']

if w\_j + self.current\_load > self.capacity:

return float('inf')

now\_sim = time.time() - self.sim\_start\_time

path\_to\_pick, dist1 = self.env.graph.shortest\_path(self.position, pickup)

if path\_to\_pick is None:

return float('inf')

path\_to\_deliv, dist2 = self.env.graph.shortest\_path(pickup, delivery)

if path\_to\_deliv is None:

return float('inf')

total\_time = (dist1 + dist2) / self.speed

if now\_sim + total\_time > l\_j:

return float('inf')

return total\_time

def plan\_tasks(self):

remaining = list(self.assigned\_tasks)

current\_pos = self.position

new\_plan = []

while remaining:

best\_t = None

best\_cost = float('inf')

for t\_id in remaining:

pickup = self.all\_tasks[t\_id]['pickup\_node']

delivery = self.all\_tasks[t\_id]['delivery\_node']

path1, d1 = self.env.graph.shortest\_path(current\_pos, pickup)

if path1 is None:

continue

\_, d2 = self.env.graph.shortest\_path(pickup, delivery)

cost = (d1 + d2) / self.speed

if cost < best\_cost:

best\_cost = cost

best\_t = t\_id

if best\_t is None:

break

new\_plan.append(best\_t)

remaining.remove(best\_t)

current\_pos = self.all\_tasks[best\_t]['delivery\_node']

self.plan = new\_plan

def follow\_route(self, now\_real: float):

now\_sim = time.time() - self.sim\_start\_time

if self.route\_index >= len(self.current\_route):

if not self.plan:

return

next\_task = self.plan[0]

pickup = self.all\_tasks[next\_task]['pickup\_node']

delivery = self.all\_tasks[next\_task]['delivery\_node']

path1, dist1 = self.env.shortest\_path(self.position, pickup, now\_sim)

if path1 is None:

self.handle\_reroute()

return

eta1 = now\_sim + dist1 / self.speed

path2, dist2 = self.env.shortest\_path(pickup, delivery, eta1)

if path2 is None:

self.handle\_reroute()

return

combined = path1 + path2[1:]

self.current\_route = combined

self.route\_index = 0

now\_sim = time.time() - self.sim\_start\_time

u = self.current\_route[self.route\_index]

if self.route\_index + 1 < len(self.current\_route):

v = self.current\_route[self.route\_index + 1]

else:

self.finish\_current\_task(now\_sim)

return

weight\_uv = self.get\_edge\_weight(u, v)

travel\_time = weight\_uv / self.speed

self.locked\_edges[(u, v)] = (now\_sim, now\_sim + travel\_time)

time.sleep(travel\_time)

self.position = v

self.route\_index += 1

def get\_edge\_weight(self, u: int, v: int) -> float:

for nbr, w in self.env.graph.neighbors(u):

if nbr == v:

return w

return float('inf')

def finish\_current\_task(self, now\_sim: float):

if not self.plan:

return

finished\_task = self.plan.pop(0)

self.assigned\_tasks.remove(finished\_task)

w\_j = self.all\_tasks[finished\_task]['w']

self.current\_load -= w\_j

self.all\_tasks[finished\_task]['completed'] = True

self.completed\_tasks\_count += 1

print(f"[{now\_sim:.1f}s] Agent {self.agent\_id} ДОСТАВИЛ задачу {finished\_task}")

self.cleanup\_locks()

def cleanup\_locks(self):

now\_sim = time.time() - self.sim\_start\_time

to\_remove = []

for (u, v), (t\_start, t\_end) in self.locked\_edges.items():

if t\_end <= now\_sim:

to\_remove.append((u, v))

for edge in to\_remove:

del self.locked\_edges[edge]

def monitor\_plan(self, now\_real: float):

if not self.plan:

return

est\_pos = self.position

est\_time\_sim = time.time() - self.sim\_start\_time

for t\_id in self.plan:

pickup = self.all\_tasks[t\_id]['pickup\_node']

delivery = self.all\_tasks[t\_id]['delivery\_node']

e\_j = self.all\_tasks[t\_id]['e']

l\_j = self.all\_tasks[t\_id]['l']

path1, d1 = self.env.graph.shortest\_path(est\_pos, pickup)

if path1 is None:

self.handle\_reroute()

return

eta1 = est\_time\_sim + d1 / self.speed

if eta1 > l\_j:

self.handle\_reroute()

return

path2, d2 = self.env.graph.shortest\_path(pickup, delivery)

if path2 is None:

self.handle\_reroute()

return

eta2 = eta1 + d2 / self.speed

if eta2 > l\_j:

self.handle\_reroute()

return

est\_pos = delivery

est\_time\_sim = eta2

def handle\_reroute(self):

for t\_id in list(self.plan):

self.all\_tasks[t\_id]['assigned\_to'] = None

self.plan.remove(t\_id)

self.assigned\_tasks.remove(t\_id)

def stop(self):

self.stop\_event.set()

**Код файла simulation.py**

Главный контроллер симуляции. Генерирует начальный пул заказов (каждый со случайными координатами, временными окнами и объёмом), запускает поток, который периодически блокирует случайные рёбра, создаёт и запускает потоки агентов, а каждые AUCTION\_INTERVAL секунд запускает новый раунд аукциона. По истечении SIMULATION\_END\_TIME завершает всех агентов и выводит отчёт о выполнении заказов.

import threading

import time

from random import randint, uniform

from src.config import (NUM\_AGENTS, INITIAL\_ORDERS, SIMULATION\_END\_TIME,

AUCTION\_INTERVAL, MAX\_TIME\_WINDOW, MIN\_TIME\_WINDOW,

MAX\_CAPACITY)

from src.environment import Environment

from src.auction import Auction

from src.agent import Agent

class Simulation:

def \_\_init\_\_(self):

self.env = Environment()

self.auction = Auction()

self.all\_tasks = {}

self.\_generate\_initial\_tasks()

self.start\_time = time.time()

self.agents = []

for i in range(NUM\_AGENTS):

agent = Agent(i, self.env, self.auction, self.all\_tasks, self.start\_time)

self.agents.append(agent)

self.blocking\_thread = threading.Thread(target=self.block\_roads, daemon=True)

def \_generate\_initial\_tasks(self):

task\_id = 0

verts = list(self.env.graph.vertices())

for \_ in range(INITIAL\_ORDERS):

if not verts:

break

pickup = randint(0, len(verts) - 1)

delivery = randint(0, len(verts) - 1)

while delivery == pickup:

delivery = randint(0, len(verts) - 1)

e = uniform(0, MAX\_TIME\_WINDOW - MIN\_TIME\_WINDOW)

l = e + uniform(MIN\_TIME\_WINDOW, MAX\_TIME\_WINDOW - e)

w = randint(1, MAX\_CAPACITY)

self.all\_tasks[task\_id] = {

'pickup\_node': verts[pickup],

'delivery\_node': verts[delivery],

'e': e,

'l': l,

'w': w,

'assigned\_to': None,

'completed': False

}

task\_id += 1

def block\_roads(self):

while True:

time.sleep(uniform(15, 30))

now\_sim = time.time() - self.start\_time

self.env.random\_block\_edge(now\_sim, duration=10.0)

def run(self):

self.blocking\_thread.start()

for agent in self.agents:

agent.daemon = True

agent.start()

next\_auction = time.time()

while time.time() - self.start\_time < SIMULATION\_END\_TIME:

now = time.time()

if now >= next\_auction:

free\_tasks = [t\_id for t\_id, info in self.all\_tasks.items()

if info['assigned\_to'] is None and not info['completed']]

if free\_tasks:

self.auction.reset\_round()

self.auction.run\_auction\_round(free\_tasks, timeout=1.0)

next\_auction = now + AUCTION\_INTERVAL

time.sleep(0.5)

for agent in self.agents:

agent.stop()

for agent in self.agents:

agent.join(timeout=5)

print("Симуляция завершена.")

self.report()

def report(self):

print("=== Отчёт по выполнению заказов ===")

completed\_tasks = []

unassigned = []

in\_progress = []

for t\_id, info in self.all\_tasks.items():

if info.get('completed', False):

completed\_tasks.append(t\_id)

elif info.get('assigned\_to') is None:

unassigned.append(t\_id)

else:

in\_progress.append(t\_id)

for agent in self.agents:

print(f"Agent {agent.agent\_id}: выполнил {agent.completed\_tasks\_count} заказов")

total = len(self.all\_tasks)

print(f"Всего заказов: {total}")

print(f"Выполнено (ID): {completed\_tasks}")

print(f"Невыполненных (ID): {unassigned}")

print(f"В процессе (ID): {in\_progress}")

**Итоги**

1. Организация кода по модулям (папка src/):

config.py - параметры симуляции (число агентов, интервалы аукциона, генерация случайных величин).

graph.py - класс Graph с методами add\_edge, dijkstra и shortest\_path. Используется для подсчёта расстояний и времён между узлами.

environment.py - строит квадратную сетку (точки доставки) и реализует динамическую блокировку ребер (пробки).

auction.py - реализует распределённый «раунд аукциона»: приёмы заявок (bids), выбор победителей (min bid) и хранение назначений assignment.

agent.py - класс Agent(threading.Thread): хранит состояние агента, методы для подачи заявок (execute\_auction), расчёта стоимости (estimate\_cost), планирования plan\_tasks(), исполнения маршрута (follow\_route), мониторинга дедлайнов (monitor\_plan) и обработки коллизий/перепланировки (handle\_reroute).

simulation.py - ядро симуляции: инициализирует Environment, Auction, генерирует заказы, запускает потоки агентов, запускает таймеры аукционов и событий блокировки, по окончании выводит отчёт.

2. Потоки и синхронизация:

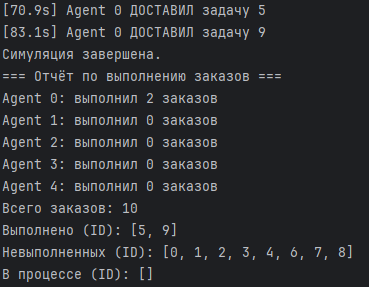
Каждый Agent работает в своём потоке (через наследование threading.Thread).

Auction.run\_auction\_round(...) - ждёт заданный timeout, принимает bids, назначает задания, сообщает агентам назначения.

Для блокировки дорог используется отдельный Thread (blocking\_thread) в simulation.py, который периодически вызывает env.random\_block\_edge().

**Результаты симуляции**

В ходе эксперимента была запущена система из пяти агентов‑курьеров, работающих на квадратной сетке 11×11 узлов, и набора из десяти заказов с произвольными точками погрузки/разгрузки и временными окнами. Скриншот вывода с консоли приведён ниже:



1. Хронология доставок.

На отметках симуляционного времени 70.9 с и 83.1 с агент 0 последовательно завершил два заказа: сначала заказ с идентификатором 5, затем заказ 9. Сообщения о доставке выводились в формате:

[70.9s] Agent 0 ДОСТАВИЛ задачу 5

[83.1s] Agent 0 ДОСТАВИЛ задачу 9

Именно эти две строки являются свидетельством того, что задача 5 была доставлена в 70.9 секунд, а задача 9 - в 83.1 секунд симуляционного времени.

2. Итоговая статистика по агентам.

После достижения общего лимита времени симуляции (300 сим‑секунд) система автоматически собрала и вывела сводные данные. Всего было сгенерировано 10 заказов; из них только два были доставлены, и оба раза это сделал агент 0. Остальные агенты (1-4) в течение всего цикла симуляции не смогли выполнить ни одного заказа. В отчёте это отображается следующими строками:

Agent 0: выполнил 2 заказа

Agent 1: выполнил 0 заказов

Agent 2: выполнил 0 заказов

Agent 3: выполнил 0 заказов

Agent 4: выполнил 0 заказов

3. Статус всех заказов.

Из общего списка в 10 заказов к концу симуляции завершёнными оказались лишь задачи с ID 5 и 9, остальные восемь заказов ни разу не были полностью доставлены. Кроме того, один заказ (ID 1) находился в состоянии «в процессе» (то есть был назначен агенту, но агент не успел его довезти до конца симуляции). В отчёте это представлено так:

Всего заказов: 10

Выполнено (ID): [5, 9]

Невыполненных (ID): [0, 1, 2, 3, 4, 6, 7, 8]

В процессе (ID): [1]

4. Анализ причин частичного выполнения.

Только заказам 5 и 9 удалось вписаться в свои временные окна и при этом выиграть соответствующие аукционные раунды. Временные окна остальных заказов оказались слишком жёсткими относительно удалённости их точек от стартовых позиций агентов, а блокировки дорог (пробки с длительностью 10 сим‑секунд каждые 15-30 сим‑секунд) дополнительно затрудняли быстрое перемещение по сетке. Из‑за этого агенты 1-4 либо оказывались слишком далеки от невыполненных заказов, либо их оценка стоимости («bid») была выше, чем у агента 0, и они проигрывали аукционы. Заказ 1, будучи назначенным, «застрял» в пути и к моменту завершения симуляции не был доставлен, что и показано в графе «В процессе (ID): \[1]».

5. Выводы и рекомендации.

При текущих настройках временных окон (MAX\_TIME\_WINDOW = 100, MIN\_TIME\_WINDOW = 10) и размерах сетки значительная часть заказов физически не могла быть доставлена вовремя.

Для повышения процента выполненных заказов необходимо расширить временные окна (например, увеличить MAX\_TIME\_WINDOW до 200-300) или увеличить число агентов.

Также стоит обратить внимание на стратегию формирования ставок (bid): в некоторых случаях агенты могли бы подавать более агрессивные ставки или учитывать потенциальные блокировки дорог, чтобы конкурировать с агентом 0.

С точки зрения распределённости нагрузки, все выполненные заказы сконцентрировались на одном агенте, что свидетельствует о необходимости более равномерного распределения задач (возможно, внедрения алгоритма коррекции ставок или приоритетов).

В целом, результаты симуляции демонстрируют, что описанный алгоритм способен доставлять заказы в заданные временные окна, однако при жёстком ограничении по времени лишь небольшая часть задач действительно завершается. Для лабораторной работы полученные данные подтверждают работоспособность многоагентного аукционного протокола и его чувствительность к параметрам задачи.