ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ «САНКТ-ПЕТЕРБУРГСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ – ПРОЦЕССОВ
УПРАВЛЕНИЯ

Лазарева Алина Вячеславовна

Выпускная квалификационная работа

Использование методов machine learning для улучшения характеристик самоорганизующихся сетей

Уровень образования: бакалавриат
Направление: 01.03.02 «Прикладная математика и информатика»
Основная образовательная программа «Прикладная математика, фундаментальная информатика и программирование»

Научный руководитель:
доктор физико-математических наук,
профессор кафедры теории игр
и статистических решений
Громова Екатерина Викторовна

Рецензент:
преподаватель University of Stirling, Phd
Кирпичникова Анна Сергеевна

Санкт-Петербург
2021

Оглавление

Введение 3

Постановка задачи 5

Обзор литературы 6

Глава 1. Математическая модель 7

1.1. Сетевая структура 7

1.2. Подвижный агент: дрон 8

1.3. Альтернативы и выигрыши 9

1.4. Меры центральности 10

Глава 2. Описание алгоритмов решения задачи 13

2.1. Алгоритм решения с использованием метрики betweenness centrality 13

2.2. «Жадный» алгоритм 14

Глава 3. Программная реализация 18

3.1. Алгоритм на основе метрики betweenness centrality. Описание структуры 18

3.2. Примеры решения задач 20

3.3. «Жадный» алгоритм 22

3.4. Примеры решения задачи 23

Глава 4. Сравнение полученных результатов 25

4.1. Сеть, состоящая из одного игрока 25

4.2. Сеть, состоящая из двух игроков 30

Глава 5. Моделирование в Network Simulator 3 32

5.1. Симулятор NS-3 32

5.2. Сравнение полученных результатов с помощью моделирования в NS-3 32

Выводы 37

Заключение 38

Список литературы 39

Приложение. Программный код 41

Введение

В данной работе рассматривается задача оптимизации передачи информации в самоорганизующихся сетях различной конфигурации [4] с точки зрения теории игр [3], а также с применением одного из методов машинного обучения. Самоорганизующиеся означают то, что сетевая инфраструктура изначально не существует, рассматриваемые сети формируются агентами с нуля без какой либо существующей сетевой инфраструктуры (вышки сотовой связи, маршрутизаторы и т. д.). Такие мобильные сети обычно называют Мобильными специальными сетями (MANET) [1] или Беспроводными специальными сетями (WANET) [2], [3].

Последнее время наблюдается повышенный интерес к изучение таких сетей. Одним из конкретных направлений исследований, которое оказалось ценным при решении различных задач, связанных с оптимизацией мобильных сетей, является применение методов теории игр. Мобильные ad-hoc сети используются в ситуациях, когда необходимо организовать связь между группами людей в районе, где сотовая связь недоступна. Применение этой технологии потенциально может спасти жизни. Например, во время поисково-спасательных работ связь между поисковыми группами является ключевым требованием. Когда происходят стихийные бедствия, нарушается коммуникационная инфраструктура, и чем быстрее устанавливается связь, тем быстрее координируются поисковые усилия.

Сетевая структура представляет собой граф, где вершины это агенты, а ребра — устойчивые связи между ними. Выигрыш каждого игрока зависит от диаметра подграфа, определенного на вершинах, принадлежащих игроку. Диаметром графа называется длина кратчайшего пути между двумя наиболее удаленными друг от друга вершинами. Диаметр графа используется для оценки максимального времени, требующегося для доставки пакета информации от одного агента к другому.

В рассматриваемом варианте задачи сетевые агенты неподвижны. Одним из эффективных способов уменьшения нагрузки на канал является использование беспилотных летательных аппаратов — дронов. Оптимизация работа сети осуществляется путем изменения ее структуры. У каждого

игрока есть дрон с приемопередатчиком, который может быть размещен в определенном положении и подключен к сети, и, кроме того, в отличие от стационарных агентов, дрон может взаимодействовать с агентами любого игрока. Следует подчеркнуть, что главной особенностью беспилотных летательных аппаратов, является их мобильность. В отличие от почти статичных агентов, беспилотник может быть размещен в любой позиции по желанию игроков.

В данной работе, как и в работах [5], [8], [16], задача сводится к поиску наилучшего местоположения для дронов. Под наилучшим местом понимается такое место, что добавление дрона в эту позицию приводит к увеличению производительности сети, а именно уменьшение диаметра подграфа игрока. Число доступных беспилотных летательных аппаратов фиксировано. Выигрыш каждого игрока зависит от диаметра подграфа, определенного на вершинах, принадлежащих игроку. Цель каждого игрока — уменьшение диаметра результирующего расширенного подграфа.

Для решения данной задачи рассматриваются два алгоритма. Первый основывается на кооперативном подходе, когда игроки стремятся достичь максимального общего улучшения. Затем применяется одна из мер центральности для отбора единственного решения среди множества оптимальных. А во втором алгоритме — «жадном» — игроки хотят улучшить каждый свою подсеть. Такой алгоритм относится к одному из методов машинного обучения.

Затем для сравнения полученных результатов было выполнено экспериментальное моделирование в среде NS-3. Моделирование сети было реализовано на основе алгоритмов организации сетей MANET посредством библиотек для моделирования NS-3 [13].

Некоторые результаты, полученные в ходе работы над выпускной работой, были опубликованы в [14], проиндексированы в системе Web of Science. Также работа [15] была принята к публикации.

Постановка задачи

Основной задачей в рассматриваемой работе является сравнение результатов работы двух алгоритмов поиска оптимальной позиции для дронов в сети с неподвижными агентами. Для этого нужно решить следующие задачи:

1. Описать сетевую структуру сети MANET.
2. Сформулировать игру между игроками, которым принадлежат агенты сети.
3. Описать алгоритм поиска множества оптимальных решений.
4. Описать и детализировать алгоритм отбора единственного решения среди оптимальных с помощью меры центральности.
5. Описать «жадный» алгоритм.
6. Реализовать программный код.
7. Сравнить результаты работы двух алгоритмов.
8. Провести моделирование сети с помощью симулятора Network Simulator 3.

Обзор литературы

В книге [1] описываются технологии самоорганизующихся сетей MANET. Рассмотрены принципы построения таких сетей и способы изменения их структуры для увеличения производительности. Сети MANET, применение теории игр для построения таких сетей и для решения задач рассмотрены в книге [2]. Аналогичная задача оптимизации передачи информации в самоорганизующихся сетях была рассмотрена в работе [3]. В работах [4], [5] задача состояла в поиске равновесия по Нэшу в многошаговой постановке игры. В [5] было выявлено, что существуют задачи определенного типа, в которых такое равновесие может не существовать, в связи с этим были сформулированы условия благожелательности игроков, при которых такое равновесие существует.

В [7] задача решается в кооперативном варианте, где максимизируется суммарный выигрыш игроков, а затем определяется множество оптимальных решений задачи. В работах [4], [7] сеть представлена с помощью графа, построенного в узлах целочисленной решетки заранее определенной конфигурации. Вершинам в графе соответствуют узлы сети, которые способны передавать информацию друг другу.

В работе [8] рассматривалась также кооперативная задача поиска оптимальных решений. Для отбора единственного решения среди множества оптимальных в работе была выбрана мера центральности PageRank. В данной работе в качестве критерия отбора используются метрика betweenness centrality, которая подробно описывается в [9]. В работе [10] рассматривается алгоритм для поиска betweenness centrality вершин различного типа графов, а также скорость его сходимости.

«Жадный» алгоритм, его принцип работы, особенности применения и его разновидности подробно рассмотрены в книгах [11] и [12]. Для сравнения полученных результатов работы рассмотренных алгоритмов было проведено моделирование работы сети в NS-3 симуляторе [13].

Глава 1. Математическая модель

1.1 Сетевая структура

Сетевая модель была предложена в работах [5], [6], [16]. Сеть задана в виде графа. Граф $R = (V, E)$ — совокупность двух множеств — множества самих объектов, называемого множеством вершин - V и множеством их парных связей - E, называемой множеством ребер. Элемент множества рёбер есть пара элементов множества вершин.

Пусть задано множество игроков $P = \{1, \ldots, n\}$. Каждый игрок $i \in P$ имеет непустое множество $M_i \neq \varnothing$ агентов, расположенных в подмножестве $X \subset \mathbb{N}^2$, т. е. в узлах целочисленной решетки. Дронами назовем дополнительных агентов, которые могут быть расположены в узлах сетки. Агенты разных игроков могут располагаться в одном узле решетки. Позиция каждого агента описывается положительными координатами в ДПСК. Будем говорить, что между агентами v_i^p и v_i^s $s \neq p$, игрока i установлена устойчивая связь $(v_i^p;\ v_i^s)$, если они находятся в соседних узлах целочисленной решетки.

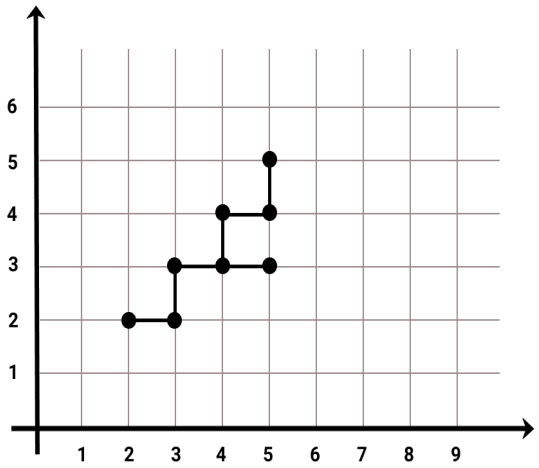


Рис. 1: Агенты и связи между ними

Таким образом, множество агентов $v_i^p \in M_i$, $i \in \{1, \ldots, N\}$ и связей $e = (v_i^p;\ v_i^s)$, $v_i^p \in M_i$, $v_i^s \in M_i$, $s \neq p$, $i \in \{1, \ldots, N\}$, определяет подграф $R_i = (\mathrm{V}_i, E_i) \in R = (\mathrm{V}, \mathrm{E})$, где V — непустое конечное множество элементов, называемых вершинами, E — конечное множество неупорядоченных

пар элементов из V, называемых ребрами.

Подграфом R_i графа R назовем граф, все вершины которого принадлежат V, а все ребра принадлежат E. Предполагается, что подграф R_i связный, что гарантирует существование пути между любыми двумя его вершинами. Взвешенный граф — граф, каждому ребру которого поставлено в соответствие некое значение (вес ребра). В противном случае граф называется невзвешанным.

Путь в подграфе R_i определим как последовательность $e = (e_1, \dots, e_k, \dots)$ ребер такую, что конец каждого предыдущего ребра является началом следующего. Длина пути графа $e = (e_1, \dots, e_k)$ — количество $d(e) = k$ ребер последовательности. Диаметром подграфа $D(R_i)$ называется число ребер в кратчайшем пути между двумя наиболее удаленными вершинами подграфа:

$$D(R_i) = \max_{(v_k, v_l) \in V_i \times V_i} d(v_k, v_l),$$

где $d(v_k, v_l)$ - путь между вершинами v_k и v_l. Так как каждый подграф R_i связный, то его диаметр может быть ограничен следующим образом:

$$2\lfloor \sqrt{M_i - 1} \rfloor \leq D(R_i) \leq M_i - 1.$$

1.2 Подвижный агент: дрон

В распоряжении у каждого игрока имеется набор подвижных агентов - дронов. Дроном $q_i \in N$ называется такой агент, положение которого может меняться в процессе игры, в то время как другие агенты имеют фиксированные позиции. Связь подвижного агента с другим агентом определяется аналогичным образом, как и связь между неподвижными агентами. Дрон может устанавливать связи с любым подвижным агентом q^i, $i \in N$, и с любыми неподвижными агентами v_i^p, $p \in M_i$, $i \in N$.

Подграф R_i^* будем называть расширенными подграфом, где множество вершин:

$$V(R_i^*) = M_i \cup \bigcup_{i=1}^{n} q_i, i \in 1, \dots n,$$

а множество ребер определяется связями на множестве $V(R_i^*)$. Заметим возможность установления связи между расширенными подграфами, которая зависит от расположения подвижных агентов.

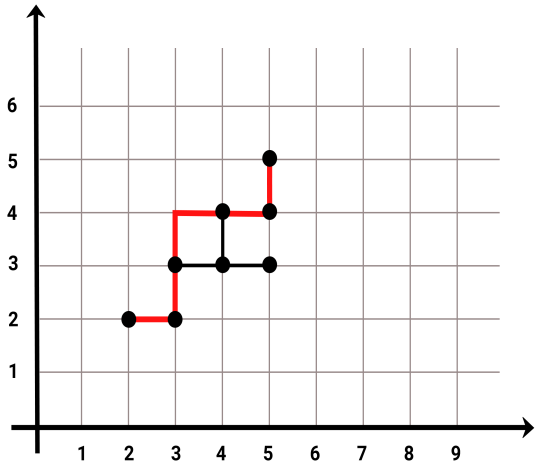


Рис. 2: Подвижный агент. Дрон

1.3 Альтернативы и выигрыши

Игроки ходят по очереди, каждый из игроков располагает дрона так, чтобы минимизировать диаметр на своем расширенном подграфе. На каждом шаге i игрок выбирает стратегию S_i. Стратегией S_i игрока i будем называть все те возможные положения на целочисленной решетке, в которые будет поставлен дрон, и которые дадут уменьшение диаметра расширенного подграфа игрока i по сравнению с текущим диаметром.

Функция выигрыша игрока i задается следующим образом:

$$H_i = d(R_i) - d(R_i^*)$$

Для некооперативной постановки задачи максимизируется функция выигрыша H_i каждого из игроков $i \in 1, \dots n$.

Для кооперативной постановки задачи будет сводиться к максимизации суммарного выигрыша, который будет состоять из суммы выигрышей

каждого из игроков:

$$H_i = \sum_{i=1}^{n} H_i$$

1.4 Меры центральности

После нахождения набора оптимальных в смысле наибольшего сокращения диаметра результирующего подграфа решений задачи возникает вопрос: как выбрать единственное решение. Чтобы ответить на него, необходимо, во-первых, обратить внимание на вклад установленных дронов в скорость передачи информации в сети в целом. Во-вторых, нужно выйти за рамки поставленной задачи и обратить внимание на другие аспекты функционирования системы. Одним из наиболее важных из этих аспектов является устойчивость сети к выходу из строя некоторых узлов. Например, если самый «крайний» узел, тот, который взаимодействует только с одним из узлов сети, терпит неудачу, тогда взаимодействие между другими узлами сети никаким образом нарушено не будет. Но если вышедший из строя узел окажется единственным, соединяющим две отдельные части сети, то структура разделится на две подсети, которые не смогут взаимодействовать друг с другом. Соответственно, дрон должен быть помещен в положение, которое в некоторой степени сохраняет общую структуру сети при выходе из строя центральных узлов. Обе эти задачи — улучшение работы сети в целом и повышение ее устойчивости к поломкам — могут быть решены одновременно с использованием мер централизации узлов сети.

Меры центральности узлов сети [9] позволяют определить величину «вклада», который каждый из узлов вносит в общую структуру. Чем больше это значение, тем важнее узел с точки зрения структуры сети. То есть, если узел потерпит неудачу (когда соответствующая вершина будет удалена из сетевого графа), взаимодействие оставшихся узлов будет значительно нарушено. Для оценки вклада соответствующего узла будут использованы различные меры центральности. Ниже рассмотрим несколько наиболее широко используемых мер центральности.

Самая простая из них — число соседей узла, называемая центрально-

стью по степени (Degree Centrality). Такая мера характеризует число связей данной вершины с другими вершинами сети и вычисляется по следующей формуле:

$$C_d(i) = \sum_{i \neq j} g_{i,j},$$

где $C_d(i)$ – степень центральности узла i, $g_{i,j}$ – связь между вершинами i и j. Вершина со степенью $(n-1)$ соединена со всеми остальными вершинами сети, а значит, такая вершина характеризуется максимально возможной степенью центральности узла.

Следующая мера, называемая центральностью близости (Closeness centralit), измеряет центральность узлов, определяя, насколько близок узел к другим. Если позиция центральна, то узел может быстро взаимодействовать с другими узлами. Таким образом, степень близости узла в сети вычисляется как обратное значение суммы его расстояний до всех других узлов:

$$C_c(i) = \frac{1}{\sum_{j=1}^{n} d_{i,j}},$$

где $C_c(i)$ – плотность центральности узла i, $d_{i,j}$ – расстояние между вершинами i и j.

Последняя мера, называемая степенью посредничества (Betweenness Centrality), основана на кратчайших путях. Для любой пары вершин в связном графе существует по меньшей мере один кратчайший путь между вершинами. В случае невзвешенного графа – минимальное число ребер, по которым проходит путь. Степень посредничества для каждой вершины равна числу этих кратчайших путей через вершину. Главную идею данного подхода можно сформулировать следующим образом: узел тем более централен, чем больше количество других узлов, между которыми он находится (чем больше маршрутов он контролирует). Формула для расчета этой меры выглядит следующим образом [10]:

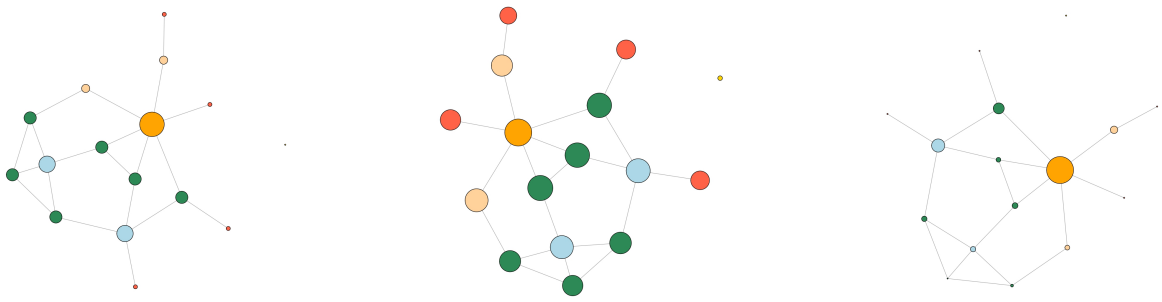
$$g(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}},$$

где σ_{st} – общее число кратчайших путей из узла st, $\sigma_{st}(v)$ равно числу этих

путей, проходящих через v.

В рассматриваемой задаче была выбрана именно эта метрика для отбора единственного решения среди оптимальных, поскольку она решает проблему оптимизации передачи информации в сети в целом, а также и описанное выше требование сохранения структуры сети в случае отказа ключевых узлов, которые обрабатывают наибольший объем информации. При расчете этой меры учитываются самые короткие пути между различными парами узлов в сети, при этом учитывается, через какие узлы проходят эти пути. Чем больше кратчайших путей проходит через определенный узел, тем больше его мера центральности с точки зрения передачи информации, поскольку алгоритмы маршрутизации точно вычисляют кратчайший возможный путь в сети.

Показаны (рис. 8) три различные меры центральности графа. Размеры вершин пропорциональны соответствующим мерам.



(a) (b) (c)

Рис. 3: Меры центральности. (a) Центральность по степени (Degree centrality). (b) Центральность по близости (Closeness centrality). (c) Степень посредничества (Betweenness centrality).

Глава 2. Описание алгоритмов решения задачи

2.1 Алгоритм решения с использованием метрики betweenness centrality

В рассматриваемом алгоритме у каждого из игроков в распоряжении находятся дроны, и задача решается в кооперативном варианте. Необходимо определить наборов стратегий игроков, которые дают наилучшее суммарное уменьшение диаметров подграфов игроков. Затем выбрать тот набор позиций дронов, в котором сумма мер betweenness centrality для этого набора позиций будет наибольшей. В случае, если сеть состоит из одного игрока, и в распоряжении у игрока имеется 1 дрон, множество позиций, которое будет оценено на способность уменьшить диаметр подграфов игроков, будет состоять из тех узлов целочисленной решётки, которые либо находятся на расстоянии 1 от вершин графа, либо, находясь внутри очерченного внутри графа прямоугольника, образуют связный подграф, хотя бы 2 вершины которого соседствуют с вершинами исходного графа [8].

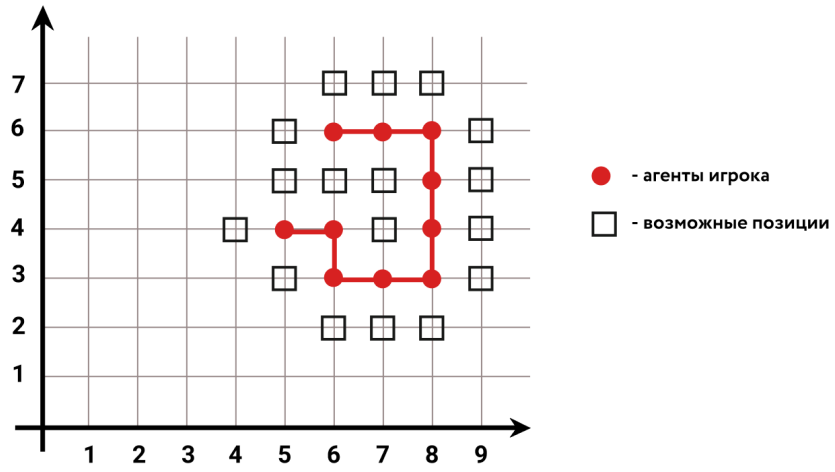


Рис. 4: Множество возможных позиций

Затем, необходимо отбросить из этого множества те наборы, которые точно не будут способствовать уменьшению диаметра графа, и выбрать наборы, дающие наибольшее улучшение.

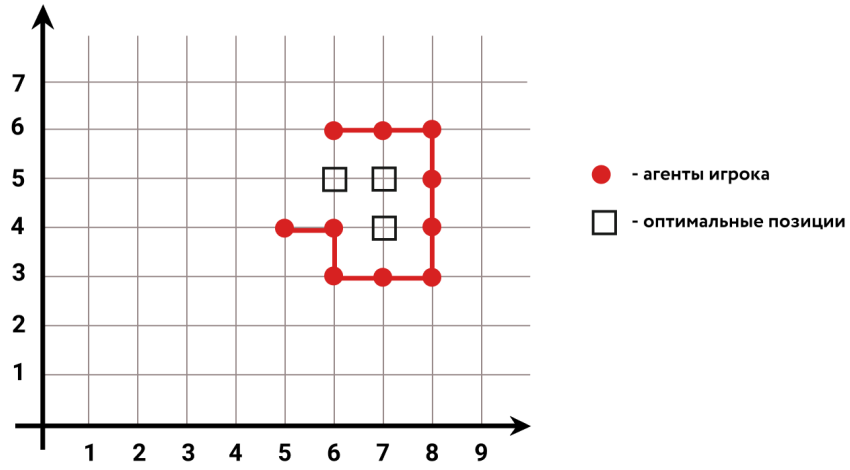


Рис. 5: Множество оптимальных позиций

А следующим шагом - отобрать единственное решение с помощью меры посредничества, т.е. выбрать из найденного множества оптимальных решений такое, что мера betweenness centrality позиций дрона игрока будет максимальной из всех возможных.

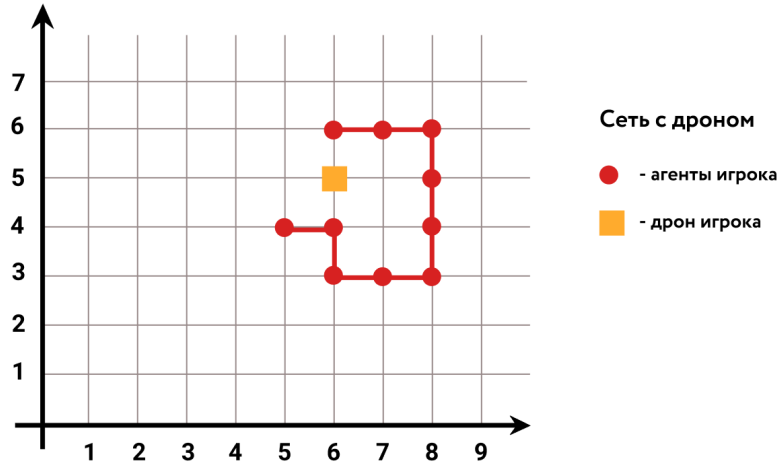


Рис. 6: Отобранное решение

2.2 «Жадный» алгоритм

«Жадный» алгоритм (greedy algorithm) — эвристический однопроходный итерационный алгоритм [11], [12]. Строит решение, добавляя на каждом шаге к текущему частичному решению новый элемент. Добавляемый элемент выбирается на основе локального оптимума («наилучший на

текущем шаге»). Т.е. производится локально оптимальный выбор в предположении о том, что он приведет к оптимальному решению глобальной задачи. Данный алгоритм основывается на методе машинного обучения, где стратегия поведения выбирается не случайно, а учитывается опыт предыдущего взаимодействия со средой.

В рассматриваемом алгоритме у каждого игрока в распоряжении находятся дроны, и каждый из игроков стремится улучшить свою подсеть с помощью подвижных агентов. В случае, когда сеть состоит из агентов одного игрока, у него в распоряжении имеется 2 дрона. В случае двух игроков у каждого из них имеется 1 дрон.

В данной задаче «жадность» заключается в выборе оптимального направления на каждом шаге. Для этого сначала определяются 2 самые удаленные точки, затем выбирается приоритетное направление движения от рассматриваемой точки к конечной.

Приоритетным направлением будем называть то, которое при проецировании на координатные оси, имеет наибольшую длину. В зависимости от знака проекции будет выбираться направление (вправо, влево, вверх, вниз).

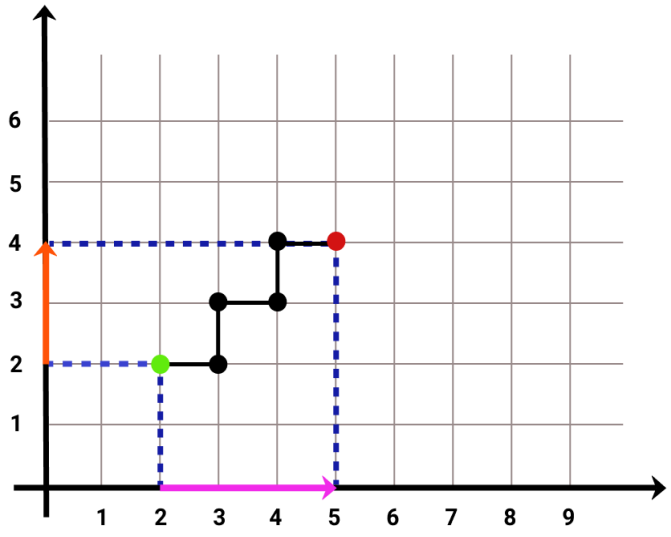


Рис. 7: Способ определения приоритетных направлений

«Жадный» алгоритм на каждой итерации, в зависимости от возможностей в рассматриваемой точке, делает оптимальный выбор в пользу од-

ной из следующих альтернатив: двигаться в самом приоритетном направлении, двигаться во втором по приоритетности направлении, добавить дрона в сеть, двигаться в третьем по приоритетности направлении. Т.е. если у рассматриваемого агента существует сосед, который располагается в приоритетном направлении, выбирается альтернатива движения в направлении к этому соседу. Если движение в самом приоритетном направлении невозможно, то проверяется условие существования соседа во втором по приоритетности направлении. Если такой сосед найден, то движение продолжается. Если же и в этом случае движение невозможно, то в сеть добавляется дрон. Далее он будет являться следующим рассматриваемым агентом.

Если же мы достигли такой точки, в которой переход к соседям невозможен, и количество оставшихся дронов равно нулю, то это означает, что необходимо проверить возможность движения в третьем по приоритетности направлении, если существует сосед у рассматриваемого агента в этом направлении, то переход осуществляется к нему. Если же такое движение невозможно, то следует вернуться к ключевой точке - такой точке, в которой было принято решение разместить последнего подвижного агента. В таком случае этот подвижный агент удаляется и выбирается другая альтернатива движения в зависимости от номера возврата к этой ключевой точке. Блок-схема работы алгоритма представлена на рис. (8)

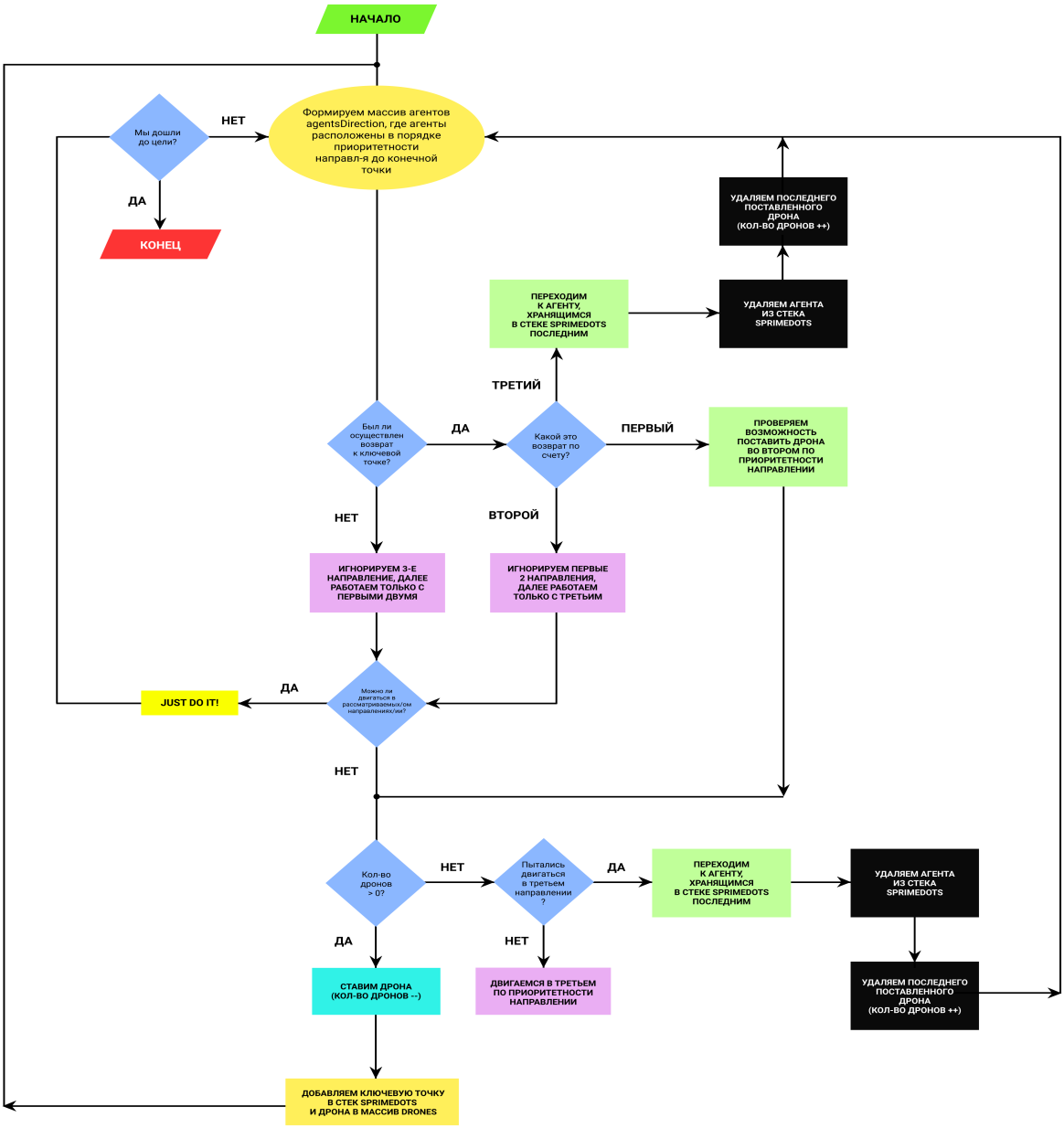


Рис. 8: Блок-схема реализации «жадного» алгоритма

Глава 3. Программная реализация

3.1 Алгоритм на основе метрики betweenness centrality. Описание структуры

За основу описания сетевой структуры была взять программа из работы [8], написанная на языке C++. Ключевую роль в формировании сетевой структуры играет класс Network с 6 шаблонными (template) параметрами, из которых 5 обязательных. Первый параметр – TPosition – определяет тип данных, которыми описываются координаты, в которых располагаются агенты сети. Этот параметр задаёт множество X. Второй параметр – TAgent – определяет тип данных, хранящий информацию об агентах сети – задаёт множество M. Третий параметр – DoNeighbor.

Для подсчета betweenness centrality вершин невзвешенного подграфа был реализован метод BetwCentral. Мера посредничества определяется с помощью алгоритма поиска по ширине [10], который сначала исследует соседа каждого узла, а затем соседей, в два этапа:

1. вычисляется длина и количество кратчайших путей между всеми парами вершин.

2. суммируются все парные зависимости.

Поиска по ширине для невзвешенных графов начинается с заданного источника $s \in V$ и на каждом шаге добавляет ближайшую вершину к набору уже обнаруженных вершин, чтобы найти кратчайший путь от источника ко всем другим вершинам.

Зависимость вершины s от вершины v обозначим за величину:

$$\delta_s(v) = \sum_{t \in V} \frac{\sigma_{st}[v]}{\sigma_{st}}$$

Тогда мера посредничества будет определяться суммой зависимостей:

$$C_b(v) = \sum_{s \neq v} \delta_s(v)$$

Алгоритм 1 Подсчет betweenness centrality для вершины v

1: $C_b[v] \leftarrow 0$, $v \in V$;
2: **for** $s \in V$ **do**
3: $S \leftarrow$ empty stack; $P[w] \leftarrow$ empty list; $w \in V$;
4: $\sigma[t] \leftarrow 0$, $t \in V$, $\sigma[s] \leftarrow 1$;
5: $d[t] \leftarrow -1$, $t \in V$, $d[s] \leftarrow 0$;
6: $Q \leftarrow$ empty queue;
7: enqueue $s \rightarrow Q$;
8: **while** Q not empty **do**
9: dequeue $v \leftarrow Q$;
10: push $v \rightarrow S$;
11: **for** each neighbor w of v **do**
12: **if** $d[w] < 0$ **then**
13: enqueue $w \rightarrow Q$;
14: $d[w] \leftarrow d[v] + 1$;
15: **end if**
16: **if** $d[w] \leftarrow d[v] + 1$ **then**
17: $\sigma[w] \leftarrow \sigma[w] + \sigma[v]$;
18: append $v \rightarrow P[w]$;
19: **end if**
20: **end for**
21: **end while**
22: **end for**
23: $\delta[v] \leftarrow 0$, $v \in V$;
24: **while** S not empty **do**
25: pop $w \leftarrow S$;
26: **for** $v \in P[w]$ **do**
27: $\delta[v] \leftarrow \delta[v] + \frac{\sigma[v]}{\sigma[w]} \cdot (1 + \delta[w])$;
28: **end for**
29: **if** $w \neq s$ **then**
30: $C_b[w] \leftarrow C_b[w] + \delta[w]$;
31: **end if**
32: **end while**

Программный код выложен в репозиторий на ресурсе GitHub: https://github.com/alik220/Diplom$_L azal$.

3.2 Примеры решения задач

Рассмотрим примеры задач, решаемых при помощи написанной программы. Исходная сеть состоит из двух игроков. В распоряжении у каждого игрока находится 1 дрон. У первого из них (обозначенного синим цветом) диаметр - это расстояние от вершины (3; 2) до вершины (4; 6), он равен 7. У второго (обозначенного красным) диаметр – расстояние от вершины (5; 4) до вершины (6; 6) – равен 9.

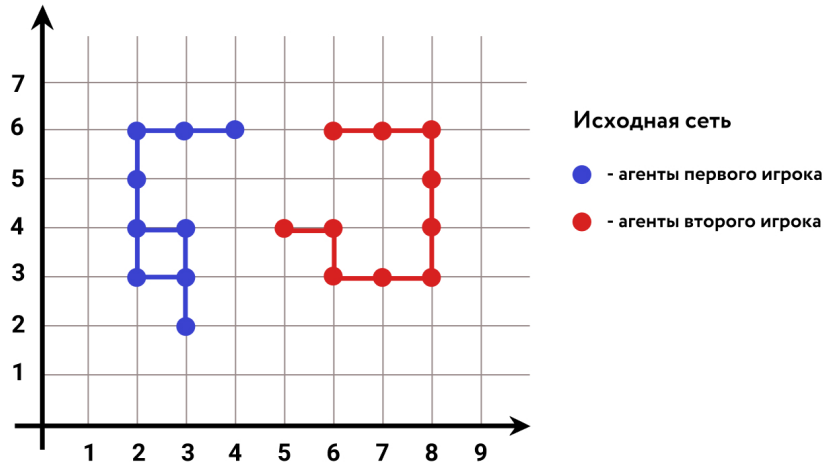


Рис. 9: Пример 1. Исходная сеть из двух игроков

Далее выбрано исходное множество узлов, которые можно использовать при переборе (обозначено черными квадратами). Т.к. в распоряжении у каждого игрока имеется 1 дрон, множество позиций, которое будет оценено на способность уменьшить диаметр подграфов игроков, будет состоять из тех узлов целочисленной решётки, которые либо находятся на расстоянии 1 от узлов графа, либо, находясь внутри очерченного внутри графа прямоугольника, образуют связный подграф, хотя бы 2 вершины которого соседствуют с вершинами исходного графа [8].

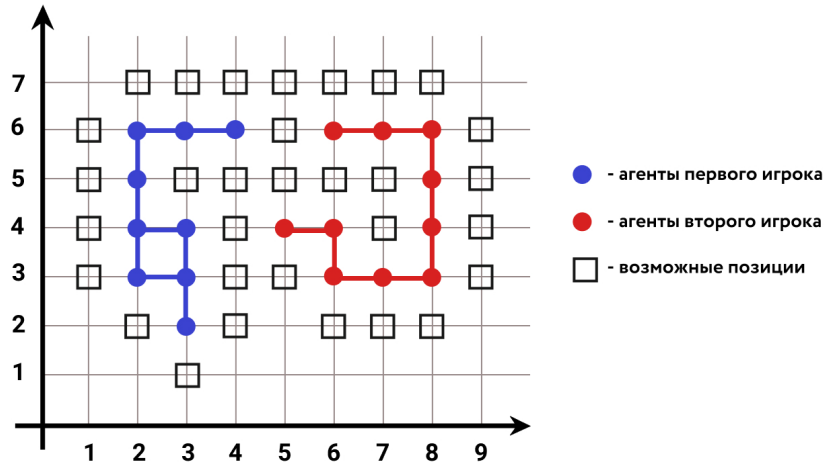


Рис. 10: Множество возможных позиций

Путем перебора были выявлены пары узлов решетки, дающих наибольшее значение суммарного выигрыша. На рис. 11 обозначены такие пары следующим образом: парой считается набор позиций, которые закрашены одинаковым цветом. Т.е. в данном примере получились следующие оптимальные пары, дающие наибольшее сокращение исходных подграфов игроков: (3; 5) и (6; 5), (6; 5) и (7; 5), (7; 4), (7; 4) и (6, 5).

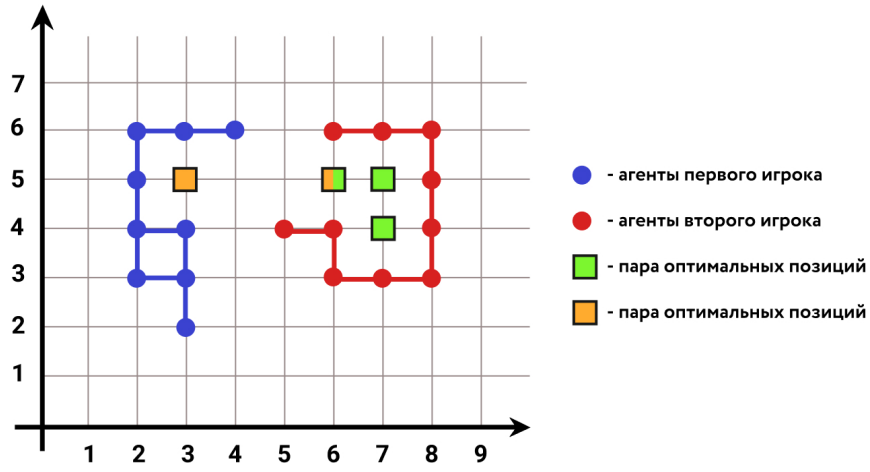


Рис. 11: Множество пар оптимальных позиций

После подсчета betweenness centrality для полученных на предыдущем этапе пар, была выбрана пара (3; 5), (6; 5), т.к. значение значение суммы betweenness centrality в этих вершинах дает наибольшие значение среди других оптимальных пар. Суммарный выигрыш составит $2 + 3 = 5$.

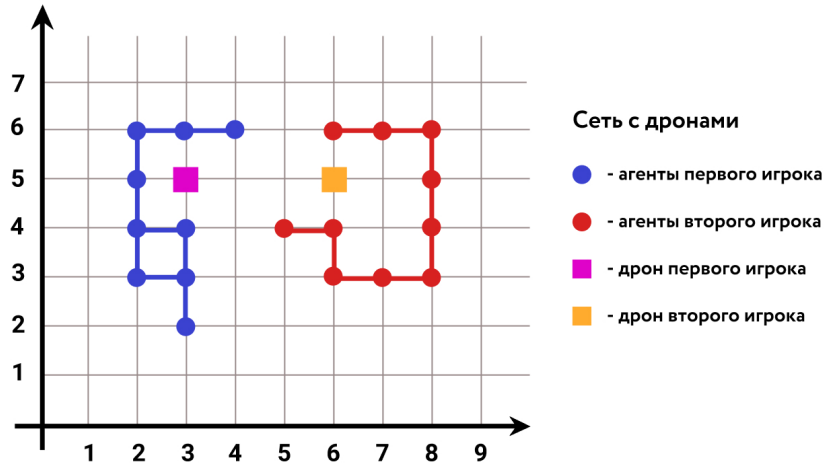


Рис. 12: Отобранное решение

3.3 «Жадный» алгоритм

Сетевую структуру программы и работу с ней реализует тот же класс Network. Используется те же структуры и функции. Помимо них добавляются новые, описанные ниже.

Метод CalcDiameterPoints возвращет две точки — самые удаленные точки в подграфе игрока, расстояние между которыми является диаметром подграфа. Метод FindDirectionNeighbors возвращает агентов в порядке приоритетности направления движения, т.е. первый агент в возвращаемом векторе будет являться самым желаемым для дальнейшего движения.

На вход алгоритму подается сеть, состоящая из агентов. Затем алгоритм вычисляет расстояние между двумя самыми удаленными точками. Количество дронов определяется переменной quantDrones. Движение от первой точки происходит в самом приоритетном направлении из трех возможных. Рассматривается только 3 направления, т.к. при движении в четвертом направлении возможно только отдаляться от цели. Если движение в приоритетном направлении невозможно, рассматривается второе по приоритетности. Если и такое движение невозможно, то в сеть добавляется дрон playerDrone в позицию, которая соответствует приоритетному направлению относительно рассматриваемого агента. Для того, чтобы вернуться к моменту добавления дрона - ключевой точке - запоминается в стеке currentPrimeDots текущий рассматриваемый агент - currentAgent, а

также previousAgent - предыдущий агент.

Количество возвратов к ключевой точке находится в переменной stepBack. Если осуществлен первый возврат, алгоритм проверяет возможность добавить дрона в сеть во втором по приоритетности направлении. Если это второй возврат, то аналогично поступаем с третьим по приоритетности направлении. Если же осуществлен третий возврат к рассматриваемой ключевой точке, то происходит удаление последнего добавленного в сеть дрона по причине его неэффективности. Движение продолжается с того агента, при рассмотрении которого было принято решение о добавлении последнего дрона.

3.4 Примеры решения задачи

Рассмотрим примеры задач, решаемых при помощи написанной программы для «жадного» алгоритма.

Исходная сеть состоит из одного игрока. Его диаметр - это расстояние от вершины (6; 5) до вершины (8; 2), он равен 9. В распоряжении у игрока находится 2 дрона.

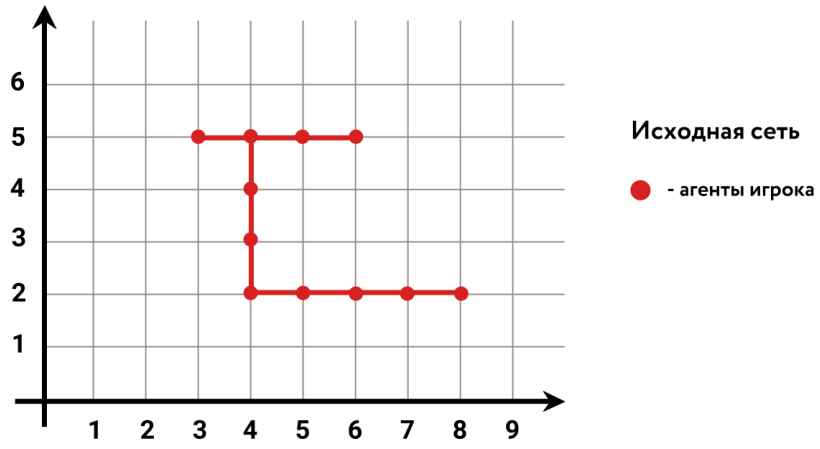


Рис. 13: Исходная сеть

Две самые удаленные точки - это (6; 5) и (8; 2). Если движение начинается из точки (6; 5), то приоритетным направлением будет являться движение вниз, т.к. до конечной точки 3 единицы вниз. Направление вправо будет вторым по приоритетности, т.к. расстояние между начальной и

конечной точками по оси Ox равно двум единицам. Третье по приоритетности, в таком случае, будет определено как движение влево.

В точке (6; 5) сначала проверяется возможность движения в самом приоритетном направлении - вниз. Т.к. такое движение невозможно, проверяется возможность добавления в сеть дрона в точку (6; 4).

Затем движение будет продолжаться в том же направлении, т.к. приоритетное направление не изменилось. Поэтому в точку (6; 3) добавится ещё один дрон, он и будет являться следующим рассматриваемым агентом, движение в выбранном направлении продолжиться.

Затем будет осуществлен переход в точку (6; 2), далее приоритетное направление измениться, поэтому конечная точка (9; 2) будет достигнута движением вправо по уже имеющимся агентам.

Таким образом, алгоритм добавит двух дронов в сеть в точки (6; 4) и (6; 3).

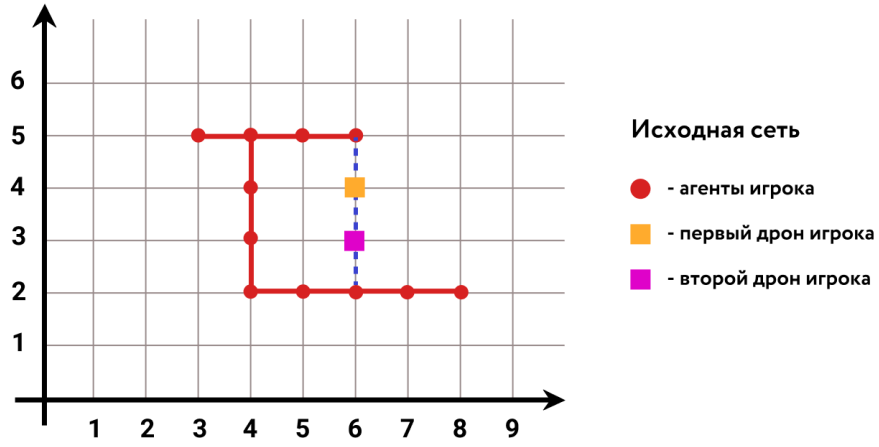


Рис. 14: Сеть с добавленными дронами

Глава 4. Сравнение полученных результатов

4.1 Сеть, состоящая из одного игрока

Рассмотрим на примере сети, состоящей из одного игрока, работу «жадного» алгоритма и алгоритма, который использует меру betweenness centrality для отбора единственного решения среди множества оптимальных.

В исходной постановке задачи у игрока имеется 2 дрона. Диаметр данного графа равен 17 — это расстояние между точками с координатами (1 ; 1) и (9 ; 2).

В алгоритме, который определяет единственное решение на основе метрики betweenness centrality, сначала ищется множество оптимальных пар позиций для размещения двух подвижных агентов. В рассматриваемом примере оптимальные пары получились следующие: (3; 2) и (5;4), (3; 2) и (3; 3), (3; 2) и (5; 3), (3; 2) и (7; 3), (3; 2) и (8; 3), (3; 3) и (5; 3), (3; 3) и (5; 4), (5; 4) и (5; 3), (5; 4) и (7; 3), (5; 4) и (8; 3). На рис. 15 обозначены некоторые из них.

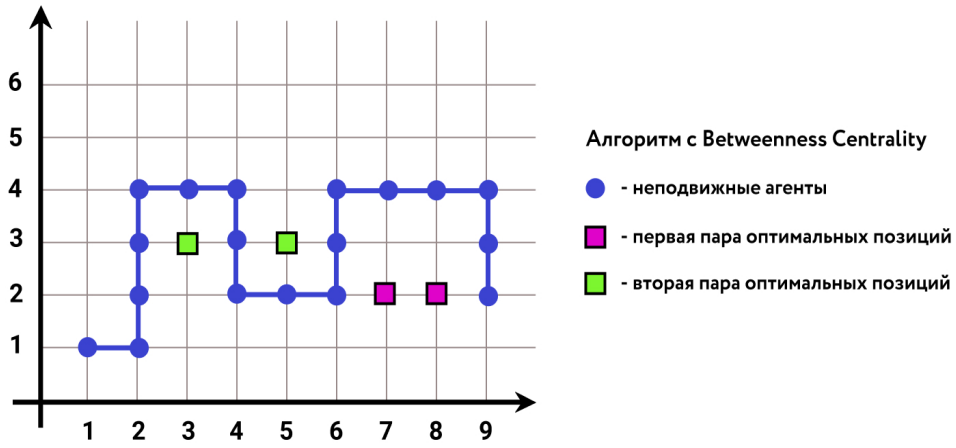


Рис. 15: Пары оптимальных позиций

В результате работы алгоритма дроны размещаются в точки (3; 3) и (5; 3), т.к значение суммы betweenness centrality в этих точках дает наибольшие значение среди всех возможных оптимальных пар.

Выигрыш в таком случае будет являться разностью изначального

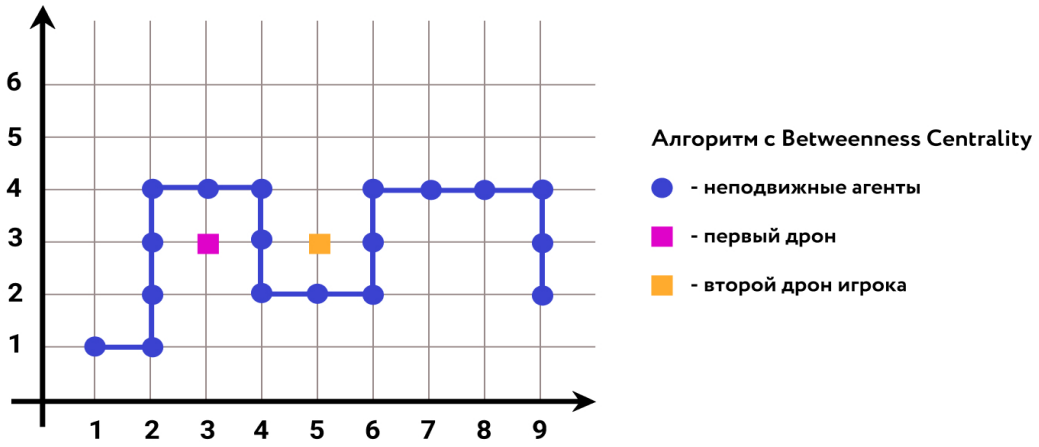


Рис. 16: Результат работы алгоритма на основе метрики betweenness centrality

диаметра и нового улучшенного диаметра, полученного с помощью дронов. Таким образом, выигрыш это разность 17 и 13, итого 4.

В случае «жадного» алгоритма движение начинается из точки с (1; 1) в точку (9; 2). Приоритетное направление будет определено как наибольшая проекция на координатные оси между начальной и конечной точкой. В данном случае, эта проекция будет равна 7, а направление движения будет определено вправо.

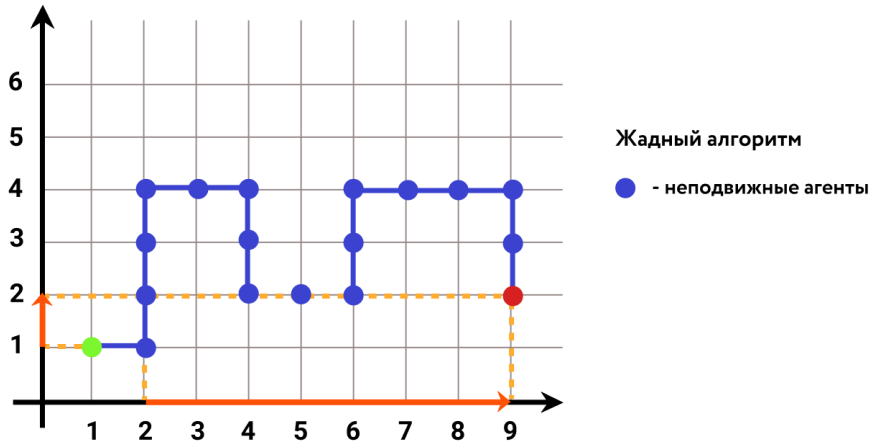


Рис. 17: Определение приоритетных направлений

Второе по приоритетности будет найдено как проекция на вторую ось, эта проекция отрицательна, а значит движение будет определено как движение вверх. До тех пор, пока существуют соседи у рассматриваемого

агента, движение в приоритетном и во втором по приоритетности направлениях будет продолжаться. В данном примере из точки (1; 1) возможно перейти в приоритетном направлении в точку (2; 1).

Далее будет рассмотрено второе по приоритетности направление относительно точки (2; 1). Оно будет по-прежнему вверх. Соответственно следующим рассматриваемым агентом будет точка (2; 2). В данной точке остается только самое приоритетное направление — вправо, т.к. движение во всех оставшихся направлениях будет только отдалять от конечной точки. Поэтому рассматривается возможность разместить дрона в точку (3; 2). Движение в приоритетном направлении можно продолжать, поэтому алгоритм дойдет до точки (6; 2). Затем будет проверяться возможность добавить дрона в точку (7; 2) — в приоритетном направлении. Количество дронов на данном этапе равно единице. Дрон будет размещен в точку (7; 2), но продолжать движение в приоритетном направлении из этой точки будет невозможно.

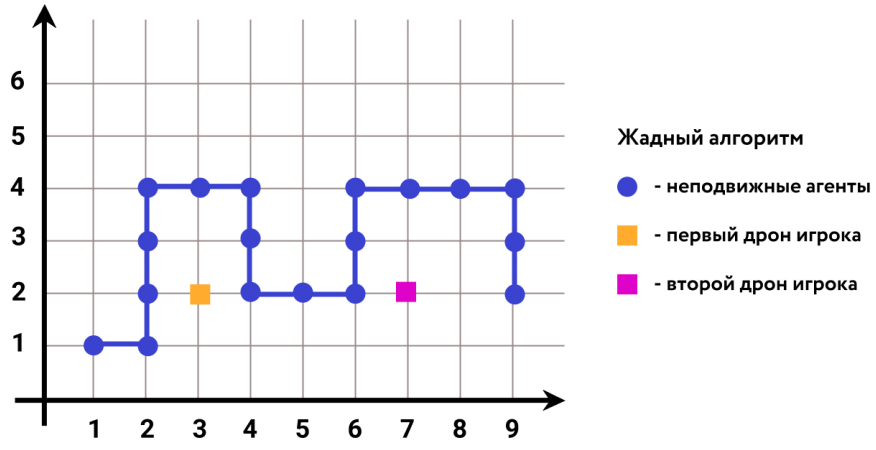


Рис. 18: Размещенные дроны в случае «жадного» алгоритма

Поэтому будет осуществлен возврат к последнему агенту, в котором было принято решение разместить дрона, т.е. в точку (6; 2), а дрон из сети будет удален. Количество дронов будет равно единице.

Затем движение будет продолжаться в менее приоритетном направлении, где существуют соседи у рассматриваемого агента. В данном случае выбрано будет направление вверх, и следующей рассматриваемой точкой

будет агент в узле (6; 3). Затем вновь будет проверяться возможность добавить дрона в приоритетном направлении — в точку (7; 3). Такой дрон будет иметь соседа в третьем по приоритетности направлении - вверх. Поэтому следующей точкой будет (7; 4). Затем движение вниз и вправо будут равнозначны по приоритетности. Т.к. рассматриваемый агент имеет соседа в точке (8; 4), движение продолжиться вправо, а затем вниз, вплоть до конечной точки. Выигрыш в данном случае составит 17 - 13 = 4.

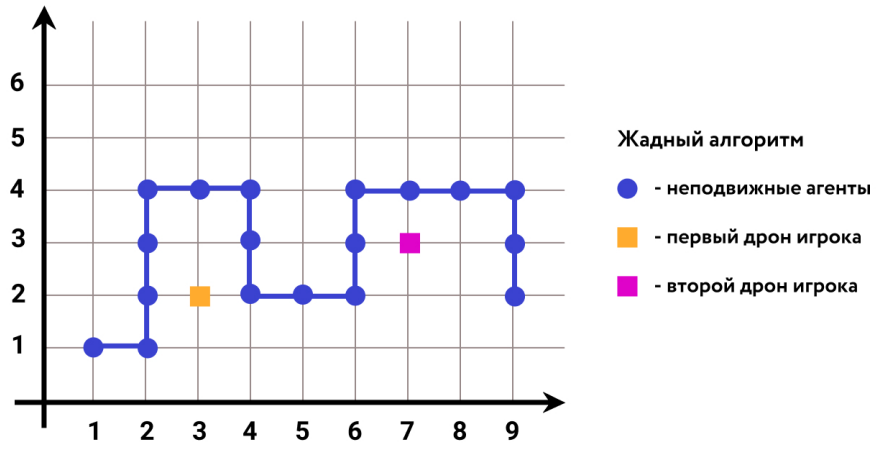


Рис. 19: Результат работы «жадного» алгоритма

Таким образом, не смотря на то, что позиции дронов были определены в различные точки, выигрыш в обоих алгоритмах получился одинаковым.

Рассмотрим следующий пример. Аналогично, сеть состоит из одного игрока, в распоряжении которого имеется 2 дрона. Диаметр исходного подграфа равен 25. Алгоритм, основанный на метрике betweenness centrality, определит сначала множество пар оптимальных решений. Затем будет выбрана та пара, где сумма мер betweenness centrality максимальна. Дроны будут помещены в точки с координатами (4; 7) и (5; 7). Диаметр подграфа сократится с 25 до 14, поэтому выигрыш составит 11.

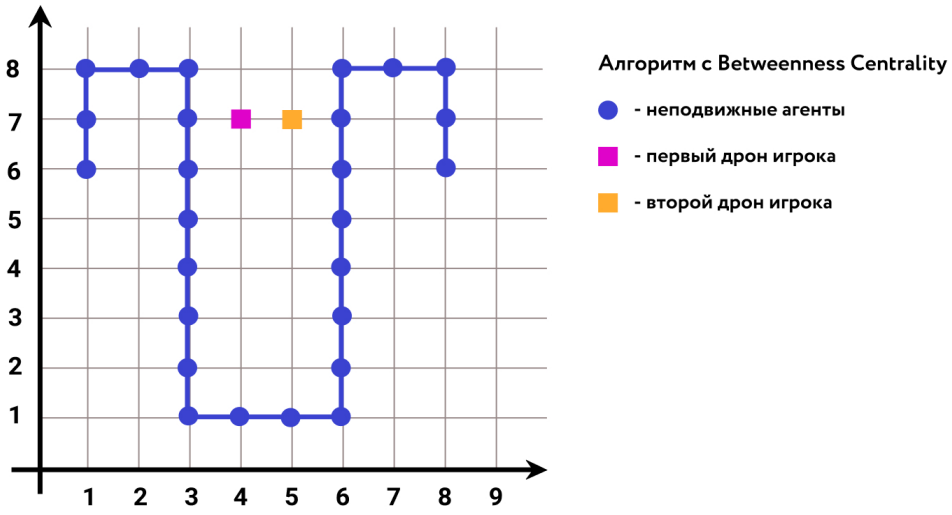


Рис. 20: Результат работы алгоритма на основе метрики betweenness centrality

В «жадном» алгоритме приоритетное направление от точки (1; 6) до точки (8; 6) будет определено как движение вправо, поэтому в первую очередь дрон разместится в точку (2; 6). А второй дрон будет размещен в точку с координатами (4; 2). Выигрыш в таком случае составит 5 (диаметр графа сократится с 25 до 20).

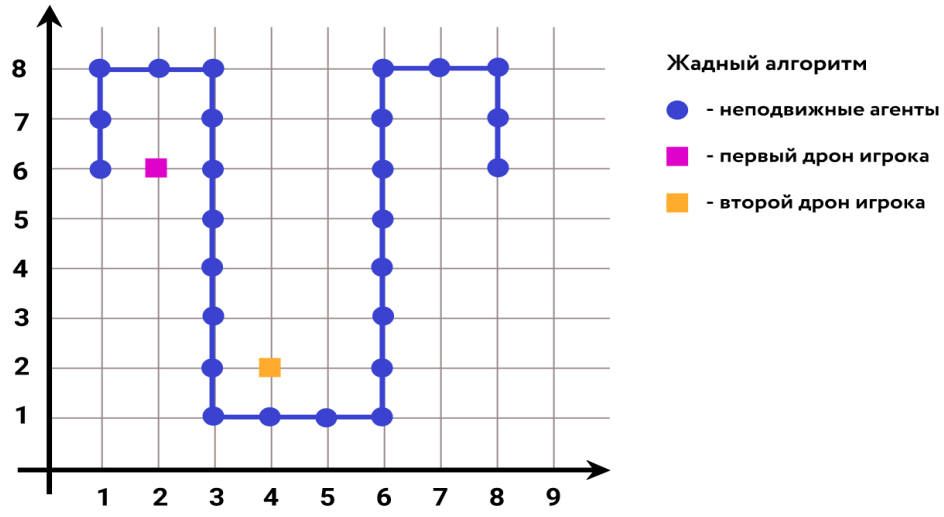


Рис. 21: Результат работы «жадного» алгоритма

4.2 Сеть, состоящая из двух игроков

Рассмотрим пример сети, состоящей из двух игроков (рис. 22), и продемонстрируем результат работы двух алгоритмов. В распоряжении у каждого игрока находится 1 дрон.

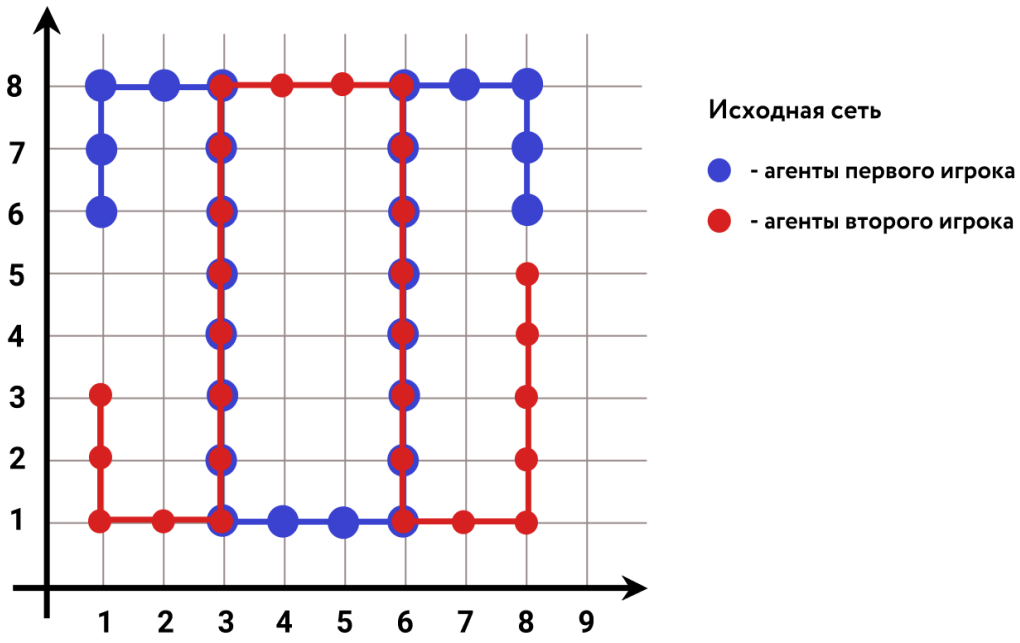


Рис. 22: Исходная сеть, состоящая из двух игроков

Диаметр исходного подграфа первого и второго игроков равен 25. Алгоритм, основанный на метрике betweenness centrality, определит сначала множество пар оптимальных решений. Затем будет выбрана та пара, где сумма мер betweenness centrality максимальна. Для первого игрока дроны будут помещены в точку с координатами (4; 2), а для второго игрока в точку (5; 2) (рис. 23). Суммарный выигрыш игроков составит 11.

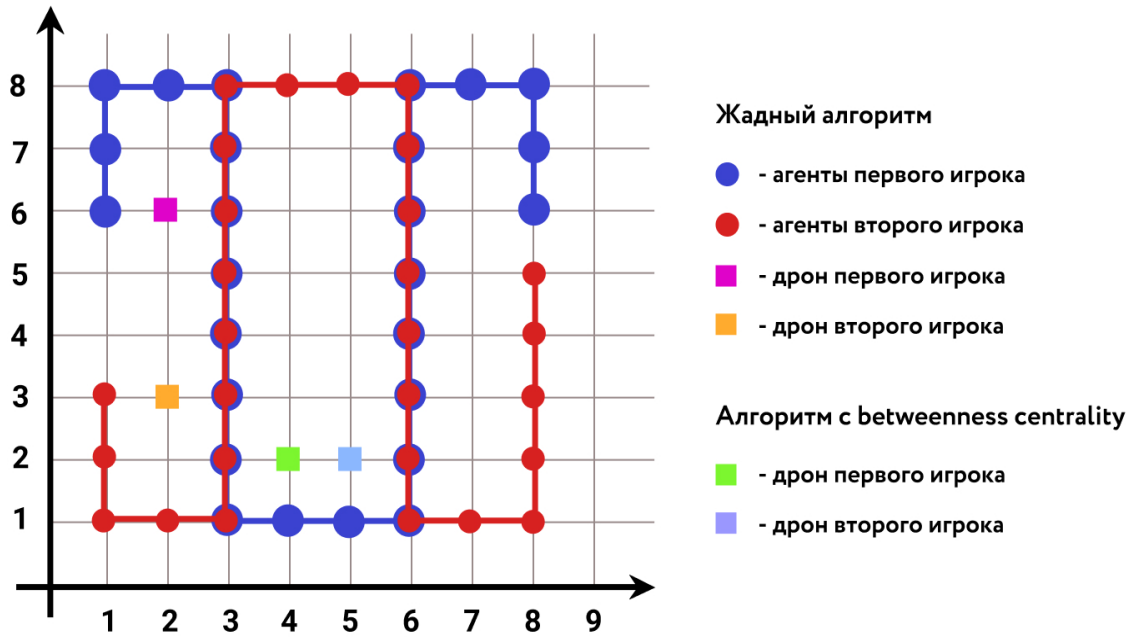


Рис. 23: Результат работы двух алгоритмов для сети, состоящей из двух игроков

Если применить «Жадный» алгоритм для сети такой же конфигурации, то найденные позиции для дронов будут точки с координатами (2; 6) и (2; 3) для первого и второго игроков соответственно (рис. 23). Суммарный выигрыш составит 4.

Глава 5. Моделирование в Network Simulator 3

5.1 Симулятор NS-3

В рассмотренной задаче средой для симуляции был выбран симулятор сетей Network Simulator 3 (NS-3) [13]. Это симулятор дискретных событий, каждое событие в котором связано со временем выполнения. Моделирование в NS-3 выполняется в ограниченном дискретном промежутке времени. Симулятор разработан для того, чтобы обеспечить открытую расширяемую платформу для моделирования сетей, исследований сети и образования. NS-3 предоставляет модели того, как реализованы и работают сети пакетной передачи данных и позволяет провести имитацию экспериментов. Причинами использовать NS-3 могут быть ситуации, когда требуется провести исследование, которое трудно или невозможно выполнять на реальной системе. Симулятор дает возможность изучать поведение системы в строго контролируемой воспроизводимой среде. В связи с этим уменьшается трудоемкость процесса, обеспечивается достаточная скорость и эффективность при моделировании.

Такие программные утилиты как NetAnim и NS-3 PyVis позволяют визуализировать моделируемую сеть. С помощью такого функционала можно проверить корректность построенной сети, проанализировать состояние узлов, передаваемые и получаемые пакеты. В ходе работы программы генерируется файл формата flowmon, который использует синтаксис xml и содержит информацию о всех протекающих во время моделирования потоках. Данные, которые содержатся в файле используются для анализа сети. Отсюда можно зафиксировать, когда именно начался поток, в какое время закончился, сколько было передано пакетов, каких размеров и сколько времени заняла передача данных.

5.2 Сравнение полученных результатов с помощью моделирования в NS-3

В примере, который был рассмотрен ранее (рис. 24), где позиции для размещения дронов у двух алгоритмов были определены в различные узлы

решетки, выигрыш был одинаков.

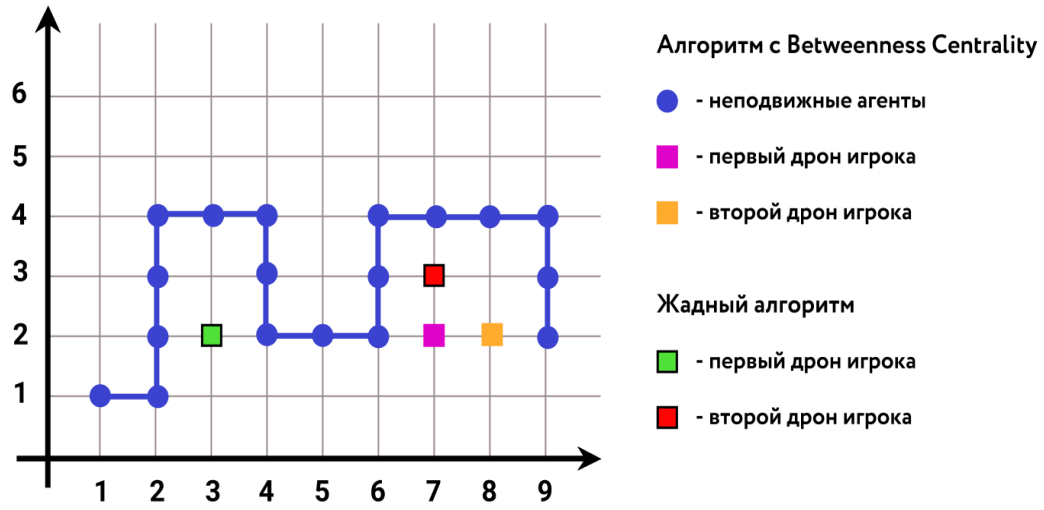


Рис. 24: Результат двух алгоритмов. Пример 1

При моделировании получились следующие результаты:

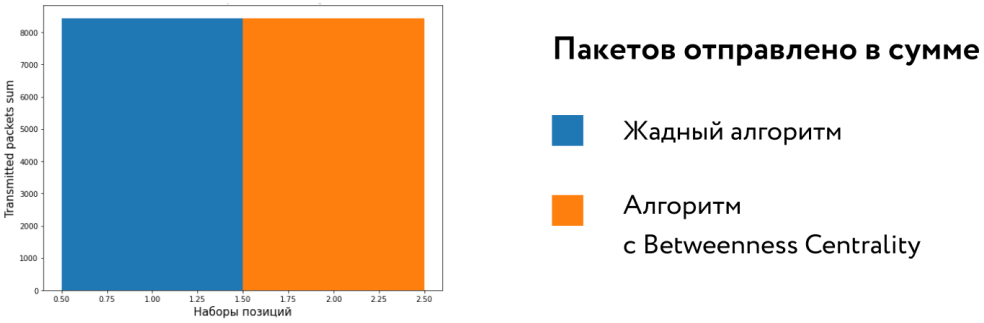


Рис. 25: Отправленные пакеты. Пример 1

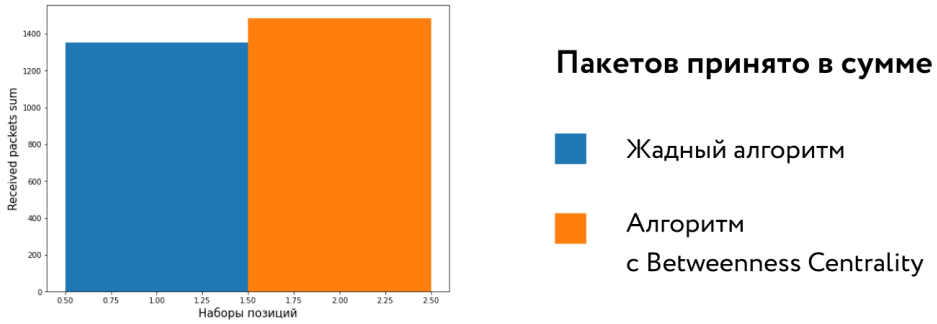


Рис. 26: Принятые пакеты. Пример 1

Ребра графа сети имеют одинаковую пропускную способность. Данные передавались от агентов каждого игрока ко всем агентам рассматриваемого игрока. Отправлено было в сумме одинаковое количество пакетов. Нужно отметить, что одним из важных показателей улучшения сети является число принятых и потерянных пакетов. Из (рис. 26) видно, что при симуляции сети с использованием «жадного» алгоритма было принято на 10% пакетов меньше, чем при симуляции сети с помощью алгоритма на основе betweenness centrality.

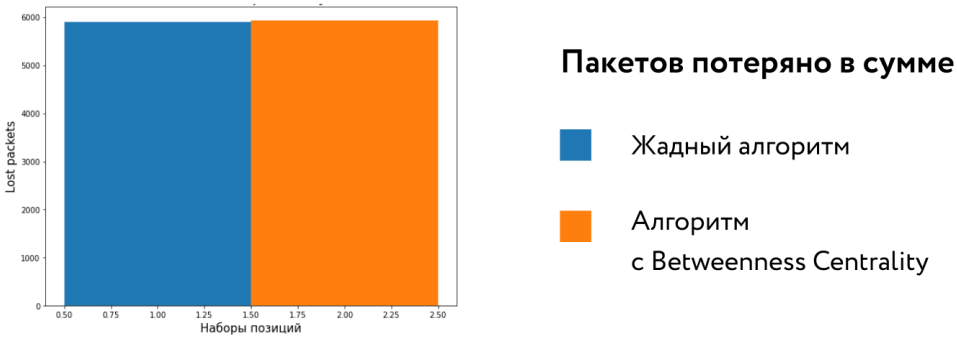


Рис. 27: Потерянные пакеты. Пример 1

Объем потерянных пакетов (рис. 27) в «жадном» алгоритме незначительно меньше, чем в алгоритме на основе меры посредничества. А задержки (рис. 28) алгоритм с betweenness centrality показал на 2, 8 нс больше, чем в случае с «жадным» алгоритмом. По полученным результатам можно сделать вывод о том, что алгоритм с betweenness centrality несмотря на

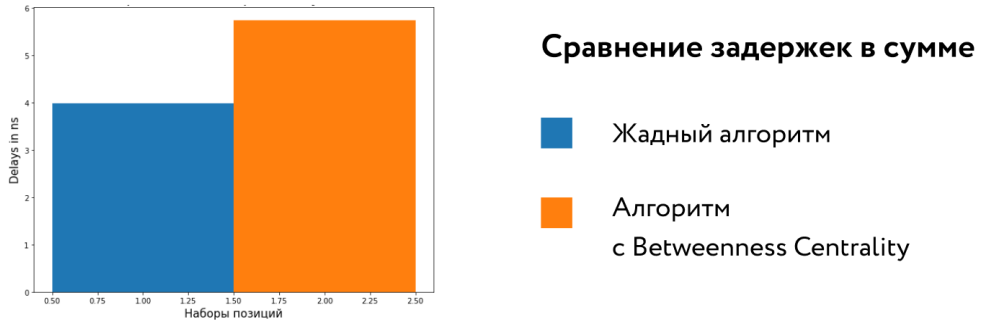


Рис. 28: Задержки пакетов. Пример 1

небольшое преимущество в количестве принятых пакетов, показал более высокие задержки пакетов в сумме по сравнению с «жадным» алгоритмом.

Рассмотрим следующий пример, где выигрыш игроков оказался различным (рис. 29).

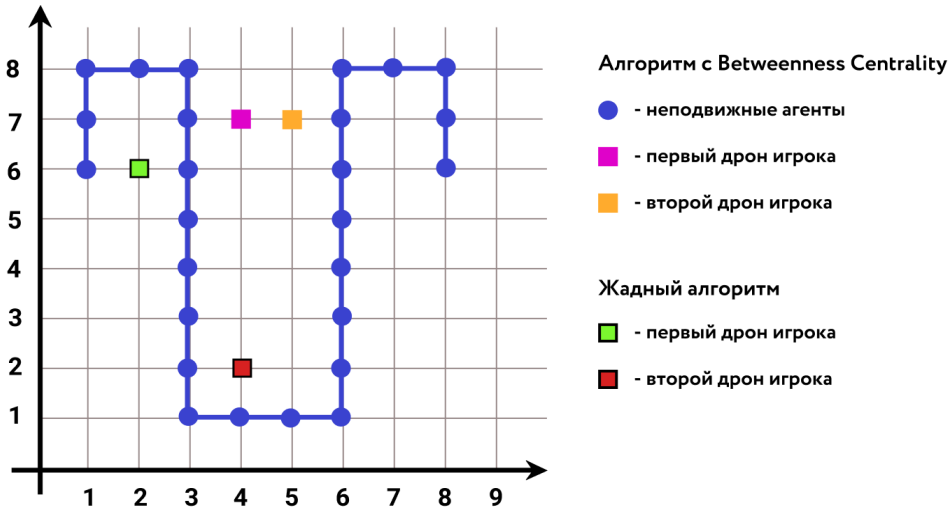


Рис. 29: Результат двух алгоритмов. Пример 2

В данном примере наибольший интерес представляет сравнение принятых и потерянных пакетов при передачи по сети, а также задержки, которые получились в сумме.

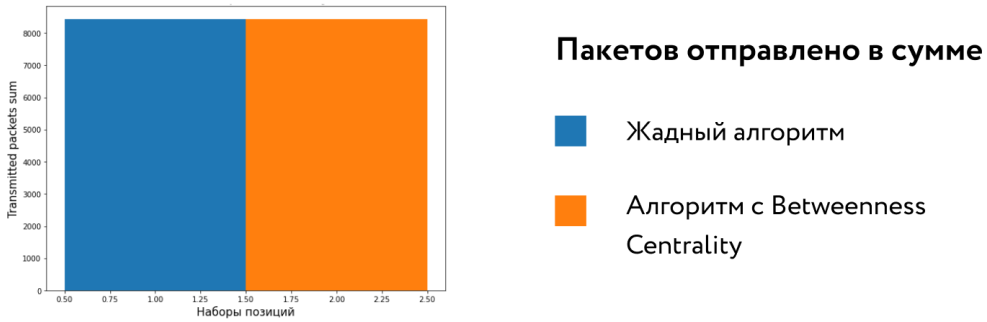


Рис. 30: Отправленные пакеты. Пример 2

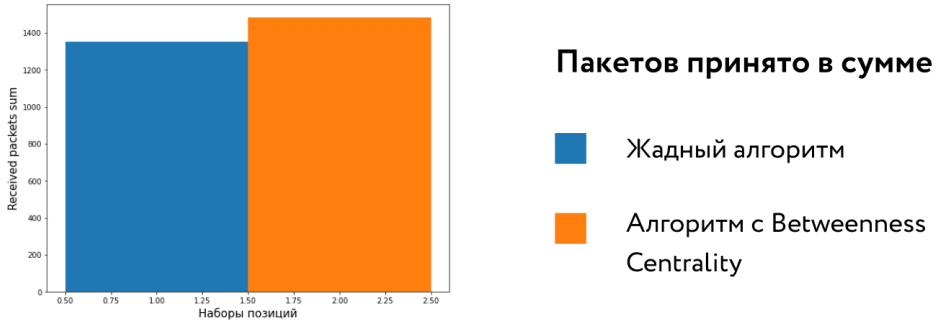


Рис. 31: Принятые пакеты. Пример 2

Отправлено было в сумме одинаковое количество пакетов (рис. 30). При симуляции сети с использованием «жадного» алгоритма принятых пакетов получилось также на 10% меньше числа принятых пакетов при симуляции сети с помощью алгоритма на основе betweenness centrality (рис. 31). А объем потерянных пакетов в данном примере оказалось значительно больше в результате работы «жадного» алгоритма, чем в алгоритме с мерой посредничества. «Жадный» алгоритм показал значительную потерю пакетов в сумме (более, чем в 2 раза по сравнению с другим алгоритмом (рис. 32)). А задержки (рис. 33) алгоритм с betweenness centrality показал на 2, 6 нс больше, чем в случае с «жадным» алгоритмом.

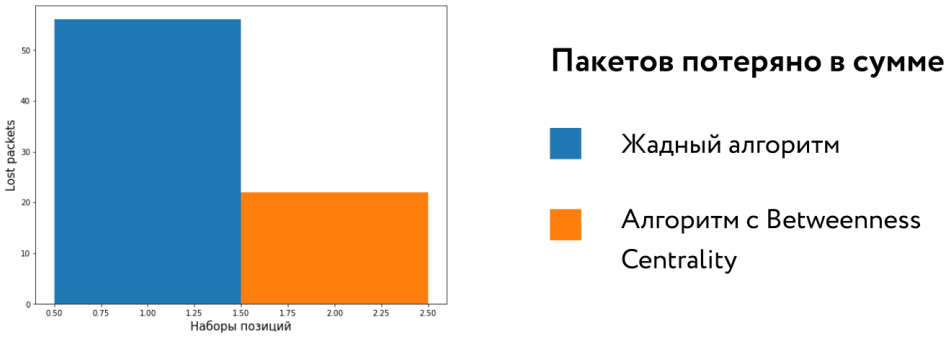


Рис. 32: Потерянные пакеты. Пример 2

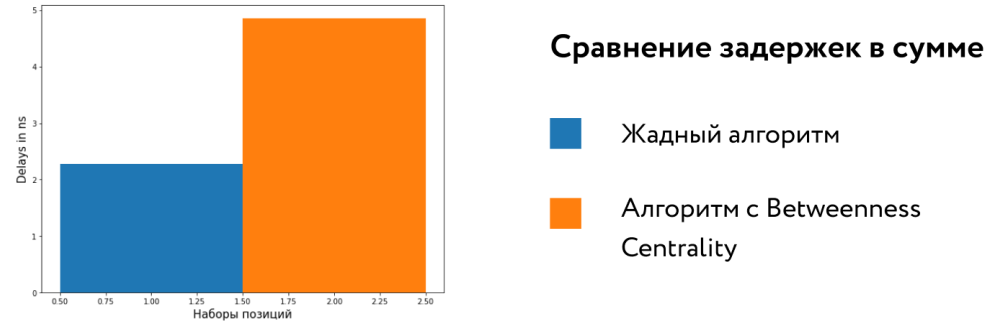


Рис. 33: Задержки пакетов. Пример 2

Выводы

Таким образом, по полученным результатам можно сделать вывод о том, что качество работы построенной сети, где использовалось оптимальное решение, найденное на основе алгоритма с мерой посредничества, выше качества работы сети, где оптимальное решение найдено с помощью «жадного» алгоритма.

Заключение

В работе была решена задача оптимизации передачи информации в сетях MANET с использованием теоретико-игрового подхода и одного из методов машинного обучения. Параметром оптимизации работы сети был выбран диаметр подграфа каждого из игроков, а сеть была представлена в виде графа. Были рассмотрены и проанализированы два алгоритма. Первый алгоритм основан на поиске и отборе единственного решения среди множества оптимальных с помощью метрики betweenness centrality. Второй - «жадный» алгоритм - основан на принятии локально оптимальных решений на каждом этапе, допуская, что конечное решение также окажется оптимальным.

Была написана и протестирована программа, реализующая алгоритм поиска решения задачи. Программа позволяет быстро изменять количество игроков, количество подвижных агентов, а также добавлять новые параметры для того, что определить их влияние на результат. Для сравнения полученных результатов работы двух алгоритмов было проведено моделирование сети с помощью симулятора Network Simulator-3.

В дальнейшем планируется рассмотреть задачу, где учитывается betweenness centrality вершин расширенного подграфа, т.е. тех узлов, где расположены неподвижные агенты. В зависимости от величины «вклада» каждого узла в сеть принимать решение о постановке дрона. Также планируется протестировать другие меры центральности как критерия отбора решения из множества оптимальных, усовершенствовать «жадный» алгоритм, улучшив качество его работы. Также планируется работа над программной реализацией алгоритма: оптимизация времени работы.

Список литературы

[1] Yu F. R. Cognitive Radio Mobile Ad Hoc Networks. Springer, 2011 507 c.

[2] Han Z., Niyato D., Saad D., Basar T., Hjorungnes A. Game Theory in Wireless and Communication Networks. Theory, Models and Applications. New York: Cambridge University Press, 2012. 554 p.

[3] Novikov D. A. Games and networks // Automation and Remote Control. 2014. Vol. 75. No 6. P. 1145–1154.

[4] Blakeway S., Gromov D. V., Gromova E. V., Kirpichnikova A. S., Plekhanova T. M. Increasing the performance of a Mobile Ad-hoc Network using a game-theoretic approach to drone positioning // Вестник Санкт-Петербургского университета. Прикладная математика. Информатика. Процессы управления. 2019 Т. 15 Вып. 1 С. 22–38.

[5] Gromova, E., Gromov, D., Timonin, N., Kirpichnikova, A., Blakeway, S. (2016, June). A dynamic game of Mobile Agent placement in a MANET. In 2016 International Conference on Systems Informatics, Modelling and Simulation (SIMS) (pp. 153-158). IEEE.

[6] E. Gromova, A. Vorontsov, S. Blakeway, A. Kirpichnikova. Pareto-optimal Solutions in a Game of Mobile Agents Placements in a MANET. Lancaster Game Theory conference, November, 2019, DOI:10.13140/RG.2.2.17885.64485

[7] Воронцов А. О поиске местоположения дронов для оптимизации работы сети типа MANET. // Процессы управления и устойчивость. 2019. Т. 6. Вып. 1. С. 404–408.

[8] Киреев С. А. Оптимизация передачи информации в самоорганизующихся сетях // Процессы управления и устойчивость. 2020. Т. 7. № 1. С. 381-386.

[9] Newman M. Networks: An Introduction. 1st Edition. Oxford University Press, 2010. 772 p.

[10] Ulrik Brandes. A Faster Algorithm for Betweenness Centrality. // Journal of Mathematical Sociology. 2001 Vol. 25 P. 163–177.

[11] Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн, С.. Алгоритмы: построение и анализ. // 2-е издание, С. 443-481.

[12] В.Е. Алексеев, В.А. Таланов. Графы и алгоритмы. Структуры данных. Модели вычислений. // Москва, Интернет-университет информационных технологий, 2006.

[13] NS-3 Documentation [Электронный ресурс]: `URL:https://www.nsnam.org/doxygen/index.html` (дата обращения: 21.03.21).

[14] Gromova E., Kireev S., Lazareva A., Kirpichnikova A., Gromov D. MANET Performance Optimization Using Network-Based Criteria and Unmanned Aerial Vehicles. // Journal of Sensor and Actuator Networks.

[15] Лазарева А., Бусел В. Использование различных мер центральности в задаче оптимизации передачи информации в самоорганизующихся сетях // Процессы управления и устойчивость. 2021.

[16] Тимонин Н. О. Одна динамическая игра управления мобильными агентами в сети. // https://nauchkor.ru/uploads/documents/587d36465f1be77c40d58b3f.pdf

[17] Alex Bavelas. Communication Patterns in Task-Oriented Groups. // The Journal of the Acoustical Society of America. 1950. Vol. 22. P. 725–730.

[18] Brin S., Page L. The anatomy of a large-scale hypertextual Web search engine. // Computer Networks and ISDN Systems. 1998. Vol. 30. P. 107–117

[19] Plekhanova T., Gromova E., Gromov D., Kirpichnikova A., Blakeway S. The strategic placement of mobile agents on a hexagonal graph using game theory // Proc. of the IEEE conference ICAT 2017. doi:10.1109/ICAT.2017.8171635.

[20] Мазалов В. В., Чиркова Ю. В. Сетевые игры. 1-е изд. СПб.: Лань, 2018. 320 с.

Приложение. Программный код

Листинг 1: Network.cpp

```
#include "Header.h"
#include <map>
#include <set>
#include <string>
#include <iostream>
#include <vector>
#include <climits>

using namespace std;

class Pos
{
public:
 int i;
 int j;
 Pos() : i(0), j(0) {}
 Pos(int iInitial, int jInitial) : i(iInitial), j(jInitial) { }
 Pos& operator=(Pos rhs)
 {
  i = rhs.i;
  j = rhs.j;
  return *this;
 }
 Pos(const Pos& another) : i(another.i), j(another.j) {}
 Pos Left() { return Pos(this->i - 1, this->j); }
 Pos Right() { return Pos(this->i + 1, this->j); }
 Pos Up() { return Pos(this->i, this->j - 1); }
 Pos Down() { return Pos(this->i, this->j + 1); }
 bool operator<(const Pos another) const
 {
  return(this->i < another.i)  ((this->i == another.i) && (this->j <
      another.j));
 }
};

class PosComp
```

```
{
public:
 bool operator()(const Pos& lhs, const Pos& rhs) const
 {
  return (lhs.i < rhs.i)  ((lhs.i == rhs.i) && (lhs.j < rhs.j));
 }
};

class Agent
{
public:
 Pos position;
 int playerID;
 ~Agent() {}

 Agent() : position(), playerID(-1) {}
 Agent(int iInitial, int jInitial, int playerID) : position(iInitial,
    jInitial), playerID(playerID) {}

 Agent(Pos positionInitial, int playerID) : position(positionInitial),
    playerID(playerID) {}

 Agent(int playerID) : playerID(playerID) {}

 Agent(Agent& another) : position(another.position),
    playerID(another.playerID) {}
};

class DoNeighbor
{
public:
 bool operator()(const Agent& lhs, const Agent& rhs)
 {
  int idist = abs(lhs.position.i - rhs.position.i);
  int jdist = abs(lhs.position.j - rhs.position.j);

  bool distanceIsRight = ((idist + jdist) <= 1) && ((idist + jdist) > 0);
  bool eitherIsADrone = ((lhs.playerID  rhs.playerID) & 0x10) != 0;
  bool samePlayer = (lhs.playerID == rhs.playerID);
```

```
  return distanceIsRight;
 }
};

class CalculateDistance
{
public:
 int operator()(const Agent& lhs, const Agent& rhs)
 {
  int idist = abs(lhs.position.i - rhs.position.i);
  int jdist = abs(lhs.position.j - rhs.position.j);
  return (idist + jdist);
 }
};

template<class _T>
class IBase
{
public:
 virtual ~IBase() {};
 virtual _T GetData() = 0;
 virtual bool Comp(IBase& another) = 0;
};

template<class _T>
class Inherited : public IBase<_T>
{
public:
 ~Inherited() {}

 _T GetData()
 {
  return 4;
 }

 bool Comp(IBase<_T>& another)
 {
  _T data = this->GetData();
```

```
  _T anotherdata = another.GetData();
  return data == anotherdata;
 }
};


int GreedyAl(Network<Pos, Agent, DoNeighbor, int, CalculateDistance,
    PosComp>& network, int quantDrones, int pID)
{
 map< int, Agent*> indicesToAgentsMapping;
 map< Agent*, int> agentsToIndicesMapping;
 pair<int, int> diameterPoints;
 int maxPoints[3];
 Agent* currentAgent;
 Agent* previousAgent;
 vector<Agent*> agentDirections;
 bool flag_move = true;
 int x_delta, y_delta;
 string direction[3];
 map<Agent*, int> thirdDirection;
 pair <Agent*, string> dir;
 map<Agent*, int> stepBack;
 Agent* endDot;
 stack <Agent*> drones;
 stack < pair <Agent*, Agent*> > sPrimeDots;
 pair <Agent*, Agent*> currentPrimeDots;
 Agent* playerDrone = new Agent(0x11);
 Pos DronePos;
 previousAgent = NULL;
 dir.first = NULL;
 dir.second = "";
 agentDirections.resize(3);
 auto filter1 = [](Agent x) {
  if (x.playerID == 0x01)
   return true;
  else
   return false;
 };
 auto filter2 = [](Agent x) {
```

```
 if (x.playerID == 0x02)
  return true;
 else
  return false;
};
if (pID == 1)
{
 diameterPoints = network.CulcDiameterPoints(filter1, (INT_MAX - 1) / 2);
 indicesToAgentsMapping = network.CulcAgentsMap(filter1);
}
if (pID == 2)
{
 diameterPoints = network.CulcDiameterPoints(filter2, (INT_MAX - 1) / 2);
    //                                (2     - first   second)
 indicesToAgentsMapping = network.CulcAgentsMap(filter2);
}
for (int i = 0; i < indicesToAgentsMapping.size(); i++)
{
 agentsToIndicesMapping[indicesToAgentsMapping[i]] = i;
}
currentAgent = indicesToAgentsMapping[diameterPoints.first];
endDot = indicesToAgentsMapping[diameterPoints.second];

stepBack[currentAgent] = 0;
thirdDirection[currentAgent] = 0;

while (currentAgent != endDot)
{
 for (int i = 0; i < 3; i++)
 {
  agentDirections[i] = NULL;
 }
 flag_move = false;
 x_delta = (currentAgent->position.i) - (endDot->position.i);
 y_delta = (currentAgent->position.j) - (endDot->position.j);

 if ((abs(x_delta) > abs(y_delta)) && x_delta > 0 && y_delta >= 0)
 {
  direction[0] = "l";
```

```
 direction[1] = "d";
 direction[2] = "u";
}
else if ((abs(x_delta) > abs(y_delta)) && x_delta > 0 && y_delta < 0)
{
 direction[0] = "l";
 direction[1] = "u";
 direction[2] = "d";
}
else if ((abs(x_delta) > abs(y_delta)) && x_delta < 0 && y_delta >= 0)
{
 direction[0] = "r";
 direction[1] = "d";
 direction[2] = "u";
}
else if ((abs(x_delta) > abs(y_delta)) && x_delta < 0 && y_delta < 0)
{
 direction[0] = "r";
 direction[1] = "u";
 direction[2] = "d";
}
if ((abs(x_delta) < abs(y_delta)) && x_delta >= 0 && y_delta > 0)
{
 direction[0] = "d";
 direction[1] = "l";
 direction[2] = "r";
}
else if ((abs(x_delta) < abs(y_delta)) && x_delta >= 0 && y_delta < 0)
{
 direction[0] = "u";
 direction[1] = "l";
 direction[2] = "r";
}
else if ((abs(x_delta) < abs(y_delta)) && x_delta < 0 && y_delta > 0)
{
 direction[0] = "d";
 direction[1] = "r";
 direction[2] = "l";
}
```

```
else if ((abs(x_delta) < abs(y_delta)) && x_delta < 0 && y_delta < 0)
{
 direction[0] = "u";
 direction[1] = "r";
 direction[2] = "l";
}
else if (abs(x_delta) == abs(y_delta))
{
 if (dir.first != currentAgent)
 {
  dir.second = "";
  for (int i = 0; i < 3; i++)
  {
   dir.second = dir.second + direction[i];
  }
  dir.first = currentAgent;
 }
 else
 {
  for (int i = 0; i < 3; i++)
  {
   direction[i] = dir.second[i];
  }
 }
}

if (pID == 1)
{
 agentDirections = network.FindDirectionNeighbors(filter1,
    agentsToIndicesMapping[currentAgent], direction);
}

if (pID == 2)
{
 agentDirections = network.FindDirectionNeighbors(filter2,
    agentsToIndicesMapping[currentAgent], direction);
}

if (stepBack[currentAgent] == 0)
```

```
{
 for (int i = 0; i < 2; i++)
 {
  if (agentDirections[i] != NULL && agentsToIndicesMapping[previousAgent]
     != agentsToIndicesMapping[agentDirections[i]])
  {
   previousAgent = currentAgent;
   currentAgent = agentDirections[i];
   flag_move = true;
   break;
  }
 }
}

else if (stepBack[currentAgent] == 1)
{
 if (((agentsToIndicesMapping[previousAgent] !=
    agentsToIndicesMapping[agentDirections[1]] && agentDirections[1] !=
    NULL)  (agentDirections[1] == NULL)) && quantDrones > 0)
 {
  if (direction[1] == "u")
  {
   DronePos.i = currentAgent->position.i;
   DronePos.j = currentAgent->position.j + 1;
  }
  else if (direction[1] == "d")
  {
   DronePos.i = currentAgent->position.i;
   DronePos.j = currentAgent->position.j - 1;
  }
  else if (direction[1] == "l")
  {
   DronePos.i = currentAgent->position.i - 1;
   DronePos.j = currentAgent->position.j;
  }
  else if (direction[1] == "r")
  {
   DronePos.i = currentAgent->position.i + 1;
   DronePos.j = currentAgent->position.j;
```

```
  }
  playerDrone = new Agent(0x11);
  playerDrone->position = DronePos;
  network.Add(DronePos, playerDrone);
  currentPrimeDots.first = currentAgent;
  currentPrimeDots.second = previousAgent;
  sPrimeDots.push(currentPrimeDots);
  drones.push(playerDrone);
  quantDrones--;

  if (pID == 1)
  {
   indicesToAgentsMapping = network.CulcAgentsMap(filter1);
  }
  if (pID == 2)
  {
   indicesToAgentsMapping = network.CulcAgentsMap(filter2);
  }
  for (int i = 0; i < indicesToAgentsMapping.size(); i++)
  {     agentsToIndicesMapping[indicesToAgentsMapping[i]] = i;

  }
  previousAgent = currentAgent;
  currentAgent = playerDrone;
  playerDrone = NULL;
  flag_move = true;
 }
 else
 {
  stepBack[currentAgent]++;
  flag_move = true;
 }
}
else if (stepBack[currentAgent] == 2)
{
 if (agentDirections[2] != NULL)
 {
  previousAgent = currentAgent;
  currentAgent = agentDirections[2];
```

```
  flag_move = true;
 }
 else
 {
  previousAgent = sPrimeDots.top().second;
  currentAgent = sPrimeDots.top().first;
  sPrimeDots.pop();
  network.Remove(drones.top()->position, drones.top());
  drones.pop();
  quantDrones++;
  stepBack[currentAgent]++;
  flag_move = true;
 }
}
else if (stepBack[currentAgent] == 3)
{
 currentAgent = sPrimeDots.top().first;
 sPrimeDots.pop();
 network.Remove(drones.top()->position, drones.top());
 drones.pop();
 quantDrones++;
 stepBack[currentAgent]++;
 flag_move = true; //
}
 if (!flag_move)
 {
  if (quantDrones > 0 && agentDirections[0] == NULL)
  {
   if (direction[0] == "u")
   {
    DronePos.i = currentAgent->position.i;
    DronePos.j = currentAgent->position.j + 1;
   }
   else if (direction[0] == "d")
   {
    DronePos.i = currentAgent->position.i;
    DronePos.j = currentAgent->position.j - 1;
   }
   else if (direction[0] == "l")
```

```
 {
  DronePos.i = currentAgent->position.i - 1;
  DronePos.j = currentAgent->position.j;
 }
 else if (direction[0] == "r")
 {
  DronePos.i = currentAgent->position.i + 1;
  DronePos.j = currentAgent->position.j;
 }
 playerDrone = new Agent(0x11);
 playerDrone->position = DronePos;
 network.Add(DronePos, playerDrone);
 drones.push(playerDrone);
 currentPrimeDots.first = currentAgent;
 currentPrimeDots.second = previousAgent;
 sPrimeDots.push(currentPrimeDots); //

 quantDrones--;
 if (pID == 1)
 {
  indicesToAgentsMapping = network.CulcAgentsMap(filter1);
 }
 if (pID == 2)
 {
  indicesToAgentsMapping = network.CulcAgentsMap(filter2);
 }
 for (int i = 0; i < indicesToAgentsMapping.size(); i++)
 {
  agentsToIndicesMapping[indicesToAgentsMapping[i]] = i;

 }
 previousAgent = currentAgent;
 currentAgent = playerDrone;
 playerDrone = NULL;
}
else if (agentDirections[2] != NULL)
{
 previousAgent = currentAgent;
 currentAgent = agentDirections[2];
```

```
     flag_move = true;
    }
    else if (agentDirections[2] == NULL)
    {
     previousAgent = sPrimeDots.top().second;
     currentAgent = sPrimeDots.top().first;
     sPrimeDots.pop();
     quantDrones++;
     stepBack[currentAgent] ++;
     network.Remove(drones.top()->position, drones.top());
     drones.pop();
     if (pID == 1)
     {
      indicesToAgentsMapping = network.CulcAgentsMap(filter1);
     }
     if (pID == 2)
     {
      indicesToAgentsMapping = network.CulcAgentsMap(filter2);
     }
     for (int i = 0; i < indicesToAgentsMapping.size(); i++)
     {
           agentsToIndicesMapping[indicesToAgentsMapping[i]] = i;
     }
    }
   }
 }
 return quantDrones;
}

int main()
{
 Network<Pos, Agent, DoNeighbor, int, CalculateDistance, PosComp> network;
 const int player1AgentsNumber = 9;
 const int player2AgentsNumber = 10;
 unsigned int quantDrones = 2;
 unsigned int quantPlayers = 2;
 int remainder;
 double currentBetwCentr = 0.0;
 vector <unsigned int> quantDronesForAnotherPlayer;
```

```
 quantDronesForAnotherPlayer.resize(quantPlayers);
 remainder = quantDrones % quantPlayers;

 for (int i = 0; i < quantPlayers; i++)
 {
  quantDronesForAnotherPlayer[i] = quantDrones / quantPlayers;
  if (quantDrones % quantPlayers != 0)
  {
   if (remainder != 0)
   {
    quantDronesForAnotherPlayer[i] += 1;
    remainder--;
   }
  }
 }
 int player1Pos[player1AgentsNumber][2] = {
 {1,6},{1,7},{1,8},
 {2,8},{3,8},{3,7},
 {3,6},{3,5},{3,4},
 {3,3},{3,2},{3,1},
 {4,1},{5,1},{6,1},
 {6,2},{6,3},{6,4},
 {6,5},{6,6},{6,7},
 {6,8},{7,8},{8,8},
 {8,7},{8,6}
};
 int player2Pos[player2AgentsNumber][2] = {
 {1,3},{1,2},{1,1},
 {2,1},{3,1},{3,2},
 {3,6},{3,5},{3,4},
 {3,3},{3,7},{3,8},
 {4,8},{5,8},{6,2},
 {6,3},{6,4},{6,5},
 {6,6},{6,7},{6,1},
 {6,8},{7,1},{8,1},
 {8,2},{8,3}

};
 Agent** player1Agents = new Agent * [player1AgentsNumber];
```

```
Agent** player2Agents = new Agent * [player2AgentsNumber];

for (int i = 0; i < player1AgentsNumber; i++)
{
 player1Agents[i] = new Agent(player1Pos[i][0], player1Pos[i][1], 0x01);
 network.Add(player1Agents[i]->position, player1Agents[i]);
}
for (int i = 0; i < player2AgentsNumber; i++)
{
 player2Agents[i] = new Agent(player2Pos[i][0], player2Pos[i][1], 0x02);
 network.Add(player2Agents[i]->position, player2Agents[i]);
}

// greedy
if (quantPlayers != 1)
{
 for (int i = 0; i < quantPlayers; i++)
 {
  if (i < quantPlayers - 1)
  {
   quantDronesForAnotherPlayer[i + 1] += GreedyAl(network,
       quantDronesForAnotherPlayer[i], i + 1);
  }
  else
   quantDronesForAnotherPlayer[i] = GreedyAl(network,
       quantDronesForAnotherPlayer[i], i + 1);
 }
}
else
{
 quantDronesForAnotherPlayer[0] = GreedyAl(network, quantDrones, 1);
}

int player1Diameter = network.GetDiameter(
 [](Agent x) {return (x.playerID == 0x01); },
 [](Agent x) {return (x.playerID == 0x01); },
 (INT_MAX - 1) / 2
);
int player2Diameter = network.GetDiameter(
```

```
 [](Agent x) {return (x.playerID == 0x02); },
 [](Agent x) {return (x.playerID == 0x02); },
 (INT_MAX - 1) / 2
);

set<Pos, PosComp> existingPositions;
set<Pos, PosComp> candidatesUniquifier;

for (int i = 0; i < player1AgentsNumber; i++)
{
 candidatesUniquifier.insert(Pos(player1Pos[i][0] - 1, player1Pos[i][1]));
 candidatesUniquifier.insert(Pos(player1Pos[i][0] + 1, player1Pos[i][1]));
 candidatesUniquifier.insert(Pos(player1Pos[i][0], player1Pos[i][1] - 1));
 candidatesUniquifier.insert(Pos(player1Pos[i][0], player1Pos[i][1] + 1));

 existingPositions.insert(Pos(player1Pos[i][0], player1Pos[i][1]));
}
for (int i = 0; i < player2AgentsNumber; i++)
{
 candidatesUniquifier.insert(Pos(player2Pos[i][0] - 1, player2Pos[i][1]));
 candidatesUniquifier.insert(Pos(player2Pos[i][0] + 1, player2Pos[i][1]));
 candidatesUniquifier.insert(Pos(player2Pos[i][0], player2Pos[i][1] - 1));
 candidatesUniquifier.insert(Pos(player2Pos[i][0], player2Pos[i][1] + 1));

 existingPositions.insert(Pos(player2Pos[i][0], player2Pos[i][1]));
}

vector<Pos> candidates;

for (set<Pos, PosComp>::iterator iter = candidatesUniquifier.begin(); iter
    != candidatesUniquifier.end(); iter++)
{
 candidates.push_back(*iter);
}
CalculateDistance distanceCalculator;
Agent* player1Drone = new Agent(0x11);
Agent* player2Drone = new Agent(0x11);

map<pair<int, int>, set<pair<Pos, Pos>>> paretoOptimal;
```

```
for (int i = 0; i < candidates.size(); i++)
{
 for (int j = i; j < candidates.size(); j++)
 {
  int idist = abs(candidates[i].i - candidates[j].i);
  int jdist = abs(candidates[i].j - candidates[j].j);
  int player1NeighborsNumber =
   (existingPositions.find(candidates[i].Left()) !=
      existingPositions.end()) +
   (existingPositions.find(candidates[i].Right()) !=
      existingPositions.end()) +
   (existingPositions.find(candidates[i].Up()) != existingPositions.end())
      +
   (existingPositions.find(candidates[i].Down()) !=
      existingPositions.end());
  int player2NeighborsNumber =
   (existingPositions.find(candidates[j].Left()) !=
      existingPositions.end()) +
   (existingPositions.find(candidates[j].Right()) !=
      existingPositions.end()) +
   (existingPositions.find(candidates[j].Up()) != existingPositions.end())
      +
   (existingPositions.find(candidates[j].Down()) !=
      existingPositions.end());
  bool connected = (idist + jdist) == 1;
  bool haveNeighbors = ((player1NeighborsNumber > 0) &&
     (player2NeighborsNumber > 0));
  bool haveMoreThanOneNeighbor = ((player1NeighborsNumber > 1) &&
     (player2NeighborsNumber > 1));
  if ((haveNeighbors && connected)  haveMoreThanOneNeighbor)
  {
   player1Drone->position = candidates[i];
   player2Drone->position = candidates[j];
   network.Add(candidates[i], player1Drone);
   network.Add(candidates[j], player2Drone);
   int player1Gain =
    player1Diameter - network.GetDiameter(
     [](Agent x) {return ((x.playerID == 0x01)  ((x.playerID & 0x10) !=
```

```
    0)); },
  [](Agent x) {return (x.playerID == 0x01); },
  (INT_MAX - 1) / 2
 );
int player2Gain =
 player2Diameter - network.GetDiameter(
  [](Agent x) {return ((x.playerID == 0x02)  ((x.playerID & 0x10) !=
    0)); },
  [](Agent x) {return (x.playerID == 0x02); },
  (INT_MAX - 1) / 2
 );
pair<int, int> newSolution = make_pair(player1Gain, player2Gain);
bool goesInside = true;
for (
 map<pair<int, int>, set<pair<Pos, Pos>>>::iterator iter =
    paretoOptimal.begin();
 iter != paretoOptimal.end();
 )
{
 int checkAgainstPlayer1Gain = iter->first.first;
 int checkAgainstPlayer2Gain = iter->first.second;

 if (player1Gain == checkAgainstPlayer1Gain && player2Gain ==
    checkAgainstPlayer2Gain)
 {
  iter++;
  continue;
 }
 if (player1Gain <= checkAgainstPlayer1Gain && player2Gain <=
    checkAgainstPlayer2Gain)
 {
  goesInside = false;
  break;
 }
 if (!(player1Gain >= checkAgainstPlayer1Gain && player2Gain >=
    checkAgainstPlayer2Gain))
 {
  iter++;
 }
```

```
    else
    {
     iter = paretoOptimal.erase(iter);
    }
   }
   if (goesInside)
   {
    paretoOptimal[newSolution].insert(make_pair(candidates[i],
        candidates[j]));
   }
   network.Remove(candidates[i], player1Drone);
   network.Remove(candidates[j], player2Drone);
  }
 }
}

double bestBetwCentr = 0.0;
pair<Pos, Pos> solution2;
vector<pair<Agent*, double>> betwCentr;

auto filter1 = [](Agent x) {
 if (x.playerID == 0x01)
  return true;
 else
  return false;
} ;
auto filter2 = [](Agent x) {
 if (x.playerID == 0x02)
  return true;
 else
  return false;
};

for (map<pair<int, int>, set<pair<Pos, Pos>>>::iterator iter =
    paretoOptimal.begin(); iter != paretoOptimal.end(); iter++)
{
 for (set<pair<Pos, Pos>>::iterator setiter = iter->second.begin();
     setiter != iter->second.end(); setiter++)
 {
```

```
Pos p1pos = setiter->first;
Pos p2pos = setiter->second;

player1Drone->position = p1pos;
player2Drone->position = p2pos;
network.Add(p1pos, player1Drone);
network.Add(p2pos, player2Drone);
currentBetwCentr = 0.0;

if (quantPlayers == 1)
{
 betwCentr = network.BetwCentral(filter1);

 for (int i = 0; i < betwCentr.size(); i++)
 {
  if (betwCentr[i].first == player1Drone  betwCentr[i].first ==
     player2Drone)
  {
   currentBetwCentr += betwCentr[i].second;
  }
 }
}
else
{
 betwCentr = network.BetwCentral(filter1);

 for (int i = 0; i < betwCentr.size(); i++)
 {
  if (betwCentr[i].first == player1Drone  betwCentr[i].first ==
     player2Drone)
  {
   currentBetwCentr += betwCentr[i].second;
  }
 }
 betwCentr = network.BetwCentral(filter2);
 for (int i = 0; i < betwCentr.size(); i++)
 {
  if (betwCentr[i].first == player1Drone  betwCentr[i].first ==
     player2Drone)
```

```
    {
     currentBetwCentr += betwCentr[i].second;
    }
   }
  }

  if (currentBetwCentr > bestBetwCentr)
  {
   bestBetwCentr = currentBetwCentr;
   solution2 = make_pair(p1pos, p2pos);
  }
  network.Remove(p1pos, player1Drone);
  network.Remove(p2pos, player2Drone);
 }
}

delete player1Drone;
delete player2Drone;

for (int i = 0; i < player1AgentsNumber; i++)
{
 network.Remove(player1Agents[i]->position, player1Agents[i]);
 delete player1Agents[i];
}

for (int i = 0; i < player2AgentsNumber; i++)
{
 network.Remove(player2Agents[i]->position, player2Agents[i]);
 delete player2Agents[i];
}

delete[] player1Agents;
delete[] player2Agents;

cout << "(" << solution2.first.i << ", " << solution2.first.j << ") (" <<
 solution2.second.i << ", " << solution2.second.j << ")" << "; "<< "Best
    betweenes centrality: " << bestBetwCentr << endl;

fgetc(stdin);
```

```
 return 0;
}
```

Листинг 2: Header.h

```
#pragma once

#ifndef MANET_PROJECT_NETWORK_HEADER_DEFINED
#define MANET_PROJECT_NETWORK_HEADER_DEFINED

#include <map>
#include <set>
#include <functional>
#include <queue>
#include <stack>
#include <string>
#include <iostream>
#include <vector>
#include <climits>

using namespace std;

template <
 class TPosition,
 class TAgent,
 typename DoNeighbor,
 class Measure,
 typename CalculateDistance,
 typename Compare = less< TPosition >
>

class Network
{
private:
 int agentsCount = 0;

 class NetworkAgentData
 {
 public:
```

```
 TAgent* agent;
 set<NetworkAgentData*> neighbors;
 set<NetworkAgentData*> reverseNeighbors;
 NetworkAgentData()
 {
  agent = NULL;
  neighbors = set<NetworkAgentData*>();
  reverseNeighbors = set<NetworkAgentData*>();
 }
 NetworkAgentData(TAgent* agentPtr) : agent(agentPtr)
 {
  neighbors = set<NetworkAgentData*>();
  reverseNeighbors = set<NetworkAgentData*>();
 }
 NetworkAgentData(NetworkAgentData& original)
 {
  this->agent = original.agent;
  this->neighbors = original.neighbors;
  this->reverseNeighbors = original.reverseNeighbors;
 }
 ~NetworkAgentData()
 {
  neighbors.clear();
  reverseNeighbors.clear();
  agent = NULL;
 }
};
class NetworkAgentDataComparator
{
public:
 bool operator()(NetworkAgentData* lhs, NetworkAgentData* rhs) const
 {
  return ((int)lhs->agent) < ((int)rhs->agent);
 }
};
map<TPosition, set<NetworkAgentData*, NetworkAgentDataComparator>,
   Compare> networkAgents;
class Iterator
{
```

```
public:
 typename map<TPosition, set<NetworkAgentData*,
    NetworkAgentDataComparator>, Compare>::iterator mapIterator;
 typename set<NetworkAgentData*, NetworkAgentDataComparator>::iterator
    setIterator;
 typename map<TPosition, set<NetworkAgentData*,
    NetworkAgentDataComparator>, Compare>::iterator mapEnd;
 typename set<NetworkAgentData*, NetworkAgentDataComparator>::iterator
    currentSetEnd;
 bool operator==(const Iterator& rhs)
 {
  return((this->mapIterator == rhs.mapIterator) \
  && ((this->mapIterator == this->mapEnd)  (this->setIterator ==
     rhs.setIterator)));
 }
 bool operator!=(const Iterator& rhs)
 {
  return(!((*this) == rhs));
 }

 Iterator& operator++()
 {
  if (mapIterator == mapEnd)
  {
   throw exception("Cannot␣move␣the␣iterator␣further");
  }
  setIterator++;
  if (setIterator == currentSetEnd)
  {
   mapIterator++;

   if (mapIterator != mapEnd)
   {
    setIterator = mapIterator->second.begin();
    currentSetEnd = mapIterator->second.end();
   }
  }
  return (*this);
 }
```

```
Iterator operator++(int)
{
 Iterator newIter(*this);
 operator++();
 return newIter;
}

NetworkAgentData* const& operator*()
{
 if (setIterator == currentSetEnd)
 {
  return NULL;
 }

 return (*setIterator);
}

NetworkAgentData** operator->()
{
 if (setIterator == currentSetEnd)
 {
  return NULL;
 }
 return (*setIterator);
}
Iterator() {}
Iterator(const Iterator& src)
{
 mapIterator = src.mapIterator;
 setIterator = src.setIterator;

 currentSetEnd = src.currentSetEnd;
 mapEnd = src.mapEnd;
}

Iterator& operator=(const Iterator& src)
{
 if (this != &src)
 {
```

```
   this->mapIterator = src.mapIterator;
   this->setIterator = src.setIterator;

   this->currentSetEnd = src.currentSetEnd;
   this->mapEnd = src.mapEnd;
  }
  return (*this);
 }
};

Iterator BeginIterator()
{
 Iterator begin;

 begin.mapIterator = networkAgents.begin();
 begin.mapEnd = networkAgents.end();

 if (begin.mapIterator != begin.mapEnd)
 {
  begin.setIterator = begin.mapIterator->second.begin();
  begin.currentSetEnd = begin.mapIterator->second.end();
 }
 return begin;
}

Iterator EndIterator()
{
 Iterator end;
 end.mapIterator = networkAgents.end();
 end.mapEnd = networkAgents.end();
 return end;
}

NetworkAgentData* FindFirst(bool(*filter)(TAgent))
{
 for (
  Iterator agentsIterator = BeginIterator();
  agentsIterator != EndIterator();
  agentsIterator++
```

```
   )
  {
   NetworkAgentData* candidate =
       const_cast<NetworkAgentData*>(*agentsIterator);
   if (filter(*(candidate->agent)))
   {
    return candidate;
   }
  }
  return NULL;
 }

public:
 void Add(TPosition position, TAgent* agent)
 {
  NetworkAgentData* newAgent = new NetworkAgentData(agent);

  for (
   Iterator agentsIterator = BeginIterator();
   agentsIterator != EndIterator();
   agentsIterator++
   )
  {
   NetworkAgentData* currentAgent = (*agentsIterator);

   if (currentAgent == newAgent)
   {
    continue;
   }

   DoNeighbor neighborChecker;

   if (neighborChecker(*(newAgent->agent), *(currentAgent->agent)))
   {
    newAgent->neighbors.insert(currentAgent);
    currentAgent->reverseNeighbors.insert(newAgent);
   }

   if (neighborChecker(*(currentAgent->agent), *(newAgent->agent)))
```

```
  {
   currentAgent->neighbors.insert(newAgent);
   newAgent->reverseNeighbors.insert(currentAgent);
  }
 }
 networkAgents[position].insert(newAgent);
 agentsCount++;
}

void Remove(TPosition position, TAgent* agent)
{
 if (networkAgents.count(position) == 0)
 {
  return;
 }

 NetworkAgentData dummyToFind(agent);
 typename set<NetworkAgentData*, NetworkAgentDataComparator>::iterator
    agentData;
 agentData = networkAgents[position].find(&dummyToFind);

 if (agentData == networkAgents[position].end())
 {
  return;
 }
 NetworkAgentData* foundAgent = (*agentData);
 for (
  typename set<NetworkAgentData*>::iterator it =
     foundAgent->neighbors.begin();
  it != foundAgent->neighbors.end();
  it++
  )
 {
  (*it)->reverseNeighbors.erase(foundAgent);
 }

 for (
  typename set<NetworkAgentData*>::iterator it =
     foundAgent->reverseNeighbors.begin();
```

```
 it != foundAgent->reverseNeighbors.end();
 it++
 )
{
 (*it)->neighbors.erase(foundAgent);
}

networkAgents[position].erase(foundAgent);
delete foundAgent;
agentsCount--;

if (networkAgents[position].size() == 0)
{
 networkAgents.erase(position);
}
}

Measure GetDiameter(bool(*filter)(TAgent), bool(*findFirstFilter)(TAgent),
   Measure maxval)
{
Measure diameter = 0;
vector<vector<int>> dp;
dp.resize(agentsCount);
for (int i = 0; i < agentsCount; i++)
{
 dp[i].resize(agentsCount, maxval);
}

NetworkAgentData* root = FindFirst(findFirstFilter);
if (root == NULL)
{
 return ((Measure)0);
}

int subgraphsize = 0;
map<NetworkAgentData*, int> agentsToIndicesMapping;
queue<NetworkAgentData*> bfsQueue;
agentsToIndicesMapping[root] = subgraphsize;
dp[subgraphsize][subgraphsize] = 0;
```

```
subgraphsize++;
bfsQueue.push(root);

while (!bfsQueue.empty())
{
 NetworkAgentData* current = bfsQueue.front();
 bfsQueue.pop();
 for (
  typename set<NetworkAgentData*>::iterator neighbors =
     current->neighbors.begin();
  neighbors != current->neighbors.end();
  neighbors++
  )
 {
  NetworkAgentData* currentNeighbor = *neighbors;

  if (!filter(*(currentNeighbor->agent)))
  {
   continue;
  }

  if (agentsToIndicesMapping.find(currentNeighbor) ==
     agentsToIndicesMapping.end())
  {
   agentsToIndicesMapping[currentNeighbor] = subgraphsize;
   dp[subgraphsize][subgraphsize] = 0;
   subgraphsize++;
   bfsQueue.push(currentNeighbor);
  }
  CalculateDistance distanceCalculator;
  dp[agentsToIndicesMapping[current]] \
     [agentsToIndicesMapping[currentNeighbor]] =
     distanceCalculator(*(current->agent), *(currentNeighbor->agent));
 }
}

for (int currentAgent = 0; currentAgent < subgraphsize; currentAgent++)
{
 for (int i = 0; i < subgraphsize; i++)
```

```
  {
   for (int j = 0; j < subgraphsize; j++)
   {
    dp[i][j] = min(dp[i][j], dp[i][currentAgent] + dp[currentAgent][j]);
   }
  }
 }
 for (int i = 0; i < subgraphsize; i++)
 {
  for (int j = 0; j < subgraphsize; j++)
  {
   diameter = max(diameter, dp[i][j]);
  }
 }
 return diameter;
}

vector<pair<TAgent*, double>> BetwCentral(bool(*filter)(TAgent))
{
 vector<pair<TAgent*, double>> result;
 NetworkAgentData* root = FindFirst(filter);
 int numbVrtx = 0;
 map< NetworkAgentData*, int> agentsToIndicesMapping;
 map< int, NetworkAgentData*> indicesToAgentsMapping;
 queue<NetworkAgentData*> Q;
 stack <NetworkAgentData*> S;
 vector <double> CB;
 vector <long> d;
 vector <long> sigma;
 vector <double> delta;
 vector< vector <unsigned long> > Pred;
 NetworkAgentData* u;
 NetworkAgentData* currentNeighbor;
 vector<int> filteredNeighbors;
 agentsToIndicesMapping[root] = numbVrtx;
 indicesToAgentsMapping[numbVrtx] = root;
 result.push_back(make_pair(root->agent, 0.0));

 if (root == NULL)
```

```
{
 throw "No␣agents␣correspond␣to␣the␣filter";
}
Q.push(root);

while (!Q.empty())
{
 u = Q.front();
 for (
  typename set<NetworkAgentData*>::iterator neighbors =
     u->neighbors.begin();
  neighbors != u->neighbors.end();
  neighbors++
  )
 {
  currentNeighbor = *neighbors;

  if (currentNeighbor->agent->playerID != root->agent->playerID &&
     currentNeighbor->agent->playerID != 17)
  {
   continue;
  }
  if (agentsToIndicesMapping.find(currentNeighbor) ==
     agentsToIndicesMapping.end())
  {
   numbVrtx++;
   agentsToIndicesMapping[currentNeighbor] = numbVrtx;
   indicesToAgentsMapping[numbVrtx] = currentNeighbor;
      result.push_back(make_pair(currentNeighbor->agent, 0.0));
   Q.push(currentNeighbor);
  }
 }
}

CB.assign(agentsToIndicesMapping.size(), 0);
queue<int> Q2;
for (int i = 0; i < agentsToIndicesMapping.size(); i++)
{
 d.assign(agentsToIndicesMapping.size(), ULONG_MAX);
```

```
Pred.assign(agentsToIndicesMapping.size(), vector <unsigned long>(0, 0));
sigma.assign(agentsToIndicesMapping.size(), 0);
delta.assign(agentsToIndicesMapping.size(), 0);
root = indicesToAgentsMapping[i];
d[i] = 0;
sigma[i] = 1;
Q2.push(i);

while (!Q2.empty())
{
 u = indicesToAgentsMapping[Q2.front()];
 Q2.pop();
 S.push(u);

 for (
  typename set<NetworkAgentData*>::iterator neighbors =
     u->neighbors.begin();
  neighbors != u->neighbors.end();
  neighbors++
  )
 {
  currentNeighbor = *neighbors;

  if (currentNeighbor->agent->playerID != root->agent->playerID &&
     currentNeighbor->agent->playerID != 17)
  {
   continue;
  }
  if (d[agentsToIndicesMapping[currentNeighbor]] == ULONG_MAX)
  {
   d[agentsToIndicesMapping[currentNeighbor]] =
      d[agentsToIndicesMapping[u]] + 1;
   Q2.push(agentsToIndicesMapping[currentNeighbor]);
  }
  if (d[agentsToIndicesMapping[currentNeighbor]] ==
     d[agentsToIndicesMapping[u]] + 1)
  {       sigma[agentsToIndicesMapping[currentNeighbor]] +=
     sigma[agentsToIndicesMapping[u]];
   Pred[agentsToIndicesMapping[currentNeighbor]].
```

```
        push_back(agentsToIndicesMapping[u]);
    }
   }
  }
  while (!S.empty())
  {
   u = S.top();
   S.pop();
   for (int j = 0; j < Pred[agentsToIndicesMapping[u]].size(); j++)
   {
    delta[Pred[agentsToIndicesMapping[u]][j]] +=
        ((double)sigma[Pred[agentsToIndicesMapping[u]][j]] /
        sigma[agentsToIndicesMapping[u]]) * (1 +
        delta[agentsToIndicesMapping[u]]);
   }
   if (agentsToIndicesMapping[u] != i)
    CB[agentsToIndicesMapping[u]] += delta[agentsToIndicesMapping[u]];
  }
  Pred.clear();
  d.clear();
  sigma.clear();
  delta.clear();
 }
 for (int i = 0; i < CB.size(); i++)
 {
  result[i].second = CB[i];
 }
 return result;
}

map<int, TAgent*> CulcAgentsMap(bool(*filter)(TAgent))
{
 map<int, TAgent*> result;
 NetworkAgentData* u;
 map< NetworkAgentData*, int> agentsToIndicesMapping;
 map< int, NetworkAgentData*> indicesToAgentsMapping;
 NetworkAgentData* root = FindFirst(filter);
 int numbVrtx = 0;
 queue<NetworkAgentData*> Q;
```

```
NetworkAgentData* currentNeighbor;
agentsToIndicesMapping[root] = numbVrtx;
indicesToAgentsMapping[numbVrtx] = root;

if (root == NULL)
{
 throw "No␣agents␣correspond␣to␣the␣filter";
}
Q.push(root);

while (!Q.empty())
{
 u = Q.front();
 Q.pop();

 for (
  typename set<NetworkAgentData*>::iterator neighbors =
     u->neighbors.begin();
  neighbors != u->neighbors.end();
  neighbors++
  )
 {
  currentNeighbor = *neighbors;
  if (currentNeighbor->agent->playerID != root->agent->playerID &&
     currentNeighbor->agent->playerID != 17)
  {
   continue;
  }

  if (agentsToIndicesMapping.find(currentNeighbor) ==
     agentsToIndicesMapping.end())
  {
   numbVrtx++;
   agentsToIndicesMapping[currentNeighbor] = numbVrtx;
   indicesToAgentsMapping[numbVrtx] = currentNeighbor;
   Q.push(currentNeighbor);
  }
```

```
 }
}
for (int i = 0; i < agentsToIndicesMapping.size(); i++)
{
 result[i] = indicesToAgentsMapping[i]->agent;
}

return result;
}

pair<int, int> CulcDiameterPoints(bool(*filter)(TAgent), Measure maxval)
{
pair<int, int> result;
NetworkAgentData* u;
map< NetworkAgentData*, int> agentsToIndicesMapping;
map< int, NetworkAgentData*> indicesToAgentsMapping;
NetworkAgentData* root = FindFirst(filter);
int numbVrtx = 0;
queue<NetworkAgentData*> Q;
vector<vector<int>> dp;
NetworkAgentData* currentNeighbor;
int max[3] = { 0, 0, 0 };
agentsToIndicesMapping[root] = numbVrtx;
indicesToAgentsMapping[numbVrtx] = root;
dp.resize(agentsCount);
for (int i = 0; i < agentsCount; i++)
{
 dp[i].resize(agentsCount, maxval);
}
agentsToIndicesMapping[root] = numbVrtx;
indicesToAgentsMapping[numbVrtx] = root;

if (root == NULL)
{
 throw "No agents correspond to the filter";
}
Q.push(root);

while (!Q.empty())
```

```
{
 u = Q.front(); //1                                    = u
 Q.pop();

 for (
  typename set<NetworkAgentData*>::iterator neighbors =
     u->neighbors.begin();
  neighbors != u->neighbors.end();
  neighbors++
  )
 {
  currentNeighbor = *neighbors;

  if (currentNeighbor->agent->playerID != root->agent->playerID &&
     currentNeighbor->agent->playerID != 17)
  {
   continue;
  }
  if (agentsToIndicesMapping.find(currentNeighbor) ==
     agentsToIndicesMapping.end())
  {
   numbVrtx++;
   agentsToIndicesMapping[currentNeighbor] = numbVrtx;
   indicesToAgentsMapping[numbVrtx] = currentNeighbor;
   Q.push(currentNeighbor);
  }

  CalculateDistance distanceCalculator;

  dp[agentsToIndicesMapping[u]][agentsToIndicesMapping[currentNeighbor]] =
   distanceCalculator(*(u->agent), *(currentNeighbor->agent));
 }
}

for (int current = 0; current < agentsToIndicesMapping.size(); current++)
{
 for (int i = 0; i < agentsToIndicesMapping.size(); i++)
 {
  for (int j = 0; j < agentsToIndicesMapping.size(); j++)
```

```
   {
    dp[i][j] = min(dp[i][j], dp[i][current] + dp[current][j]);
    if (dp[i][j] > max[0] && dp[i][j] < maxval)
    {
     max[0] = dp[i][j];
     max[1] = i;
     max[2] = j;
    }
   }
  }
 }
 result.first = max[1];
 result.second = max[2];
 return result;
}

vector<TAgent*> FindDirectionNeighbors(bool(*filter)(TAgent), int agentNum
    , string *directionPointer)
{
 vector<TAgent*> result;
 NetworkAgentData* u;
 map< NetworkAgentData*, int> agentsToIndicesMapping;
 map< int, NetworkAgentData*> indicesToAgentsMapping;
 NetworkAgentData* root = FindFirst(filter);
 int numbVrtx = 0;
 queue<NetworkAgentData*> Q;
 int g = 0;
 string dir;
 string direction[3];
 NetworkAgentData* currentAgent;
 NetworkAgentData* currentNeighbor;
 result.resize(3);

 for (int i = 0; i < 3; i++)
 {
  result[i] = NULL;
 }
 for (int i = 0; i < 3; i++)
 {
```

```
 direction[i] = *directionPointer;
 directionPointer++;
}
agentsToIndicesMapping[root] = numbVrtx;
indicesToAgentsMapping[numbVrtx] = root;
if (root == NULL)
{
 throw "No␣agents␣correspond␣to␣the␣filter";
}
Q.push(root);
while (!Q.empty())
{
 u = Q.front();
 Q.pop();
 for (
  typename set<NetworkAgentData*>::iterator neighbors =
     u->neighbors.begin();
  neighbors != u->neighbors.end();
  neighbors++
  )
 {
  currentNeighbor = *neighbors;

  if (currentNeighbor->agent->playerID != root->agent->playerID &&
     currentNeighbor->agent->playerID != 17)
  {
   continue;
  }

  if (agentsToIndicesMapping.find(currentNeighbor) ==
     agentsToIndicesMapping.end())
  {
   numbVrtx++;
   agentsToIndicesMapping[currentNeighbor] = numbVrtx;
   indicesToAgentsMapping[numbVrtx] = currentNeighbor;
   Q.push(currentNeighbor);
  }
 }
}
```

```
currentAgent = indicesToAgentsMapping[agentNum];
for (
 typename set<NetworkAgentData*>::iterator neighbors =
    currentAgent->neighbors.begin();
 neighbors != currentAgent->neighbors.end();
 neighbors++
 )
{
  currentNeighbor = *neighbors;
  g++;
  if (currentNeighbor->agent->playerID != root->agent->playerID &&
     currentNeighbor->agent->playerID != 0x17)
  {
   continue;
  }
  if (currentAgent->agent->position.i ==
     currentNeighbor->agent->position.i)
  {
   if (currentAgent->agent->position.j >
      currentNeighbor->agent->position.j)
    dir = "d";
   else
    dir = "u";
  }

  else
  {
   if (currentAgent->agent->position.i >
      currentNeighbor->agent->position.i)
    dir = "l";
   else
    dir = "r";
  }

  for (int i = 0; i < 3; i++)
  {
   if (direction[i] == dir)
    result[i] = currentNeighbor->agent;
  }
```

```
  }
  return result;
 }
 Network() {}
};

#endif
```