ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ – ПРОЦЕССОВ УПРАВЛЕНИЯ
НАПРАВЛЕНИЕ 02.03.02 «ФУНДАМЕНТАЛЬНАЯ ИНФОРМАТИКА И ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ»
ОБРАЗОВАТЕЛЬНАЯ ПРОГРАММА: «ПРОГРАММИРОВАНИЕ И ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ»

Киреев Сергей Андреевич

Выпускная квалификационная работа бакалавра

Теоретико-игровая модель передачи данных в беспроводных сетях с различной архитектурой

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

Рецензент:
ассистент кафедры вычислительных методов механики деформируемого тела
Шувалов Глеб Михайлович

Санкт-Петербург
2020

Оглавление

Введение

В данной работе рассматривается задача оптимизации передачи информации в самоорганизующихся сетях различных конфигураций с использованием теоретико-игрового подхода. Под самоорганизующимися подразумеваются сети, образуемые агентами (приёмо-передатчиками) без существующей заранее сетевой инфраструктуры (сотовые вышки, роутеры и т. п.). Часто такие сети находят применение в ситуациях, когда необходимо организовать связь между группами людей на местности, где сотовая связь недоступна. Например, при действиях спасательных групп [6,16] в горах, болотах и других труднопроходимых районах, одни группы используют переносное оборудование для общения с другими, причём, вследствие ограничений на дальность передачи радиосигнала, а также ландшафтных ограничений местности, при необходимости передать некоторое сообщение от одной группы до другой, информация зачастую идёт не напрямую, а через другие группы (узлы сети), которые выступают посредниками, транслируя сообщение по сети. В англоязычной терминологии такие сети называются Mobile Ad Hoc Network или Wireless Ad Hoc Network [10,15] и часто сокращаются до аббревиатур (MANET/WANET).

Основная особенность задачи состоит в том, что не все агенты сети принадлежат одному лицу. В сети присутствует несколько игроков, каждый из которых владеет некоторым набором агентов. Передавать друг другу информацию могут только агенты одного и того же игрока. В рассматриваемом варианте задачи агенты сети неподвижны. Оптимизация работы сети происходит за счёт изменения её структуры путём добавления в неё новых агентов – у каждого из игроков есть дрон с приёмо-передатчиком, который может быть помещён в некоторую позицию и включён в сеть, а кроме того, в отличие от неподвижных агентов, дрон может взаимодействовать с агентами любого игрока.

Таким образом, в постановке задачи вырисовывается теоретико-игровой подход: во-первых, есть множество игроков, владеющих дронами, во-вторых, у каждого из игроков есть стратегия – выбор конкретной позиции, куда поместить дрон, в-третьих, у каждого из игроков есть функция выигрыша, которая характеризует степень оптимизации сети каждого игрока (фактически, в данный момент не совсем корректно говорить, что функция выигрыша уже есть – она будет введена в процессе решения задачи, а здесь имеется ввиду, что для игроков возможно тем или иным образом определить меру улучшения их подсетей).

Здесь рассмотрен кооперативный подход к решению задачи, при котором игроки стремятся улучшить не каждый свою подсеть, а достичь максимального общего улучшения.

Постановка задачи

Основной целью ВКР являлось улучшение характеристик работы сети при помощи теоретико-игрового подхода к вопросу расстановки дополнительных узлов сети, которыми являются подвижные управляемые агенты.

В связи с поставленной целью можно сформулировать следующие основные задачи дипломной работы:

1) Формально описать сетевую структуру MANET/WANET при помощи теории графов;
2) Сформулировать кооперативную (однократную) игру между игроками, которым принадлежат различные агенты сети, намеревающиеся совместными усилиями улучшить характеристики общей сети;
3) Описать алгоритм поиска оптимальных стратегий в кооперативной игре;
4) Детализировать процедуру отбора оптимальных решений в случае их неединственности;

5) Реализовать предложенный алгоритм при помощи программных средств.

Обзор литературы

Варианты аналогичной некооперативной многошаговой игры с поиском равновесия по Нэшу рассмотрены в работах [7,12]. В [12] было выявлено, что существование такого равновесия не гарантировано и сформулированы некоторые условия, при которых оно существует. В [7] помимо прямоугольной используются также треугольная и шестиугольная решётки. Проводится тестирование с помощью симулятора сетей «Network Simulator 3» и выясняется, что предложенные методы действительно дают значительное улучшение производительности сети.

В [1] рассматривается кооперативный вариант, в котором максимизируется суммарный выигрыш игроков и, таким образом, вычисляется множество оптимальных решений задачи.

В работах [1,7] сеть рассматривается в виде графа, построенного на узлах, расположенных в вершинах решеток заранее определённой конфигурации (шестиугольных или прямоугольных), которые в дальнейшем будем называть неподвижными агентами. Рёбрами в графе соединяются вершины, которые соответствуют узлам сети, способным передавать информацию друг другу, т.е. тем, которые находятся в зоне доступа друг для друга. При решении вопроса об оптимизации передачи данных используются метрики графов, которые необходимо улучшить.

Различные коммуникационные сети, их моделирование с помощью графов, алгоритмы на них, а также сетевые метрики (в частности, меры центральности, используемые в данной работе), сетевые процессы и особенности генерации случайных сетей подробно описаны в книге [14].

Особенности применения теоретико-игрового подхода при моделировании сетевых коммуникационных структур и решении задач на них рассмотрены в книгах [3,10].

Глава 1. Математическая модель

Итак, сеть рассматривается в виде графа, состоящего из подграфов участвующих игроков. Задача состоит в поиске позиций для дронов каждого игрока, с целью максимизировать суммарный выигрыш игроков, где отдельный выигрыш каждого из игроков – это величина уменьшения диаметра его подграфа после «запуска» дрона. Каждая из найденных позиций и будет искомой стратегией соответствующего игрока. Задача решается в кооперативном варианте, поскольку оптимизируется именно суммарный выигрыш. Вследствие этого возникает множество оптимальных решений, дающих одинаковый максимальный суммарный выигрыш. Как правило, это множество содержит более одного решения, поэтому возникает потребность в выборе единственного решения из множества оптимальных – то есть, необходимо определить критерий отбора решения. В данной постановке в качестве критерия была использована мера центральности PageRank графа. Вычислив для каждого из наборов сумму мер PageRank вершин, входящих в него, можно найти набор с максимальной суммой – он и будет искомым решением задачи. Случай, когда суммы PageRank вершин для каких-либо двух и более наборов являются максимальными по всем наборам и при этом равны друг другу, на данный момент не встречался. Если такой случай возможен, то программа автоматически выберет первое полученное решение. Приведем формальное описание модели и методов, при помощи которой будет решаться поставленная задача об оптимизации сети.

1.1 Теоретико-игровой подход

Использование теоретико-игрового подхода к задаче предполагает введение обозначений для некоторых объектов.

Пусть $P = \{1, \dots, n\}$ – множество игроков. Каждый игрок $i \in P$ имеет непустое множество агентов $M_i \neq \emptyset$, участвующих в сети.

Поскольку решаемая задача имеет отношение к расположению некоторых объектов на местности, необходимо ввести координатное пространство, в котором будут располагаться агенты, а также функцию расстояния на этом пространстве.

Пусть X – пространство, ρ – функция расстояния:

$$\bar{X} \subset X \times X, \qquad \rho: \bar{X} \mapsto Y,$$

Y – множество значений функции расстояния. Как правило, $Y = \mathbb{R}$ и $\rho: \bar{X} \mapsto \mathbb{R}$.

В дальнейшем будем множество всех агентов сети обозначать M. $M = \bigcup_{i=1,\dots n} M_i$. Произвольных агентов сети будем обозначать a, $a \in M$. Агента игрока i будем обозначать a^i, $a_i \in M_i$.

Множество дронов игрока i будем обозначать $\overline{M_i}$, $\overline{M_i} \neq \emptyset$. В данной постановке у каждого игрока дрон имеется только один, поэтому все множества дронов – одноэлементные. Дрона игрока i будем обозначать за b_i, $b_i \in \overline{M_i}$.

Задача состоит в определении стратегии каждого из игроков. Стратегией S_i игрока i является позиция $\widehat{x_i}$, в которую будет поставлен его дрон.

Функции выигрыша игроков будут определены после введения определений, касающихся графов.

1.2 Описание сетевой структуры

Сеть в задаче рассматривается в виде графа.

Граф G – это математический объект, представляющий собой пару множеств: $G = (V, E)$, где V – произвольное множество, именуемое *множеством вершин* графа, E – множество упорядоченных (такой граф называется *ориентированным*) или неупорядоченных (такой граф называется *неориентированным*) пар (v_1, v_2), где $v_1, v_2 \in V$, именуемое *множеством дуг* графа (в случае ориентированного графа) или *множеством рёбер* графа (в случае неориентированного графа).

Данным определением графа можно пользоваться в том случае, когда дана конкретная сеть и дальнейших её изменений не планируется, что подходит при анализе особенностей структуры сети. Однако в поставленной задаче, помимо анализа структуры сети, будут происходить последовательные её преобразования – будут добавляться дроны, после чего будет проводиться оценка улучшения параметров сети, затем дроны удаляются и так несколько раз. Из-за необходимости динамического изменения сети стандартного определения графа оказывается недостаточно. Кроме того, необходимо учесть возможность добавления новых параметров в программу, которые могут повлиять на возможность образования связей между агентами (к примеру, допустим, что при переходе к реальным условиям оказалось, что устройства передачи информации, имеющиеся в распоряжении, имеют различную дальность действия – в таком случае расстояние, на котором агенты могут взаимодействовать, не фиксировано и внутри программы требуется использовать функцию, определяющую возможность взаимодействия).

Определим множество V вершин графа. V – множество пар вида (a, x), где $a \in M$ – это агент сети, $x \in X$ – координата, в которой он расположен. При этом a является произвольным математическим объектом и можно заметить, что координата x может быть рассмотрена как одно из его свойств. Тогда множество V будет состоять только из самих агентов, вместо пар агент-координата, однако здесь рассматривается более общий вариант, а случай включения координаты в свойства агента может быть сведён к нему.

Стоит отметить, что мы только обозначили координатное пространство, а не определили его. В данной работе задача решается для сети на плоскости и координаты будут вводиться двухмерные. Но в дальнейшем, при совершенствовании модели до максимально приближенной к реальным условиям, а также непосредственно при работе в реальных условиях, будет необходимо оперировать трёхмерным пространством или пространством высших размерностей (в случае, например, когда на возможность связи будут

влиять факторы, неопределимые при помощи только координат, например, при работе групп по разные стороны от некоторого заграждения, не пропускающего сигнал). Поэтому в общем виде задачи предполагается, что агенты расположены в некотором неизвестном заранее пространстве.

Для определения возможности взаимодействия агентов в сети (наличия ребра между вершинами графа, соответствующего данной сети), введём функцию-инцедентор δ:

$$\delta(a_s, x_s, a_t, x_t) = \begin{cases} 1, & \text{если } a_s \text{ в координате } x_s \\ \text{может взаимодействовать с } a_t \text{ в координате } x_t & \\ 0, \text{иначе} & \end{cases}$$

где $x_s, x_t \in V$. Опять же, возможен вариант, когда координата – одно из свойств агента, тогда $\delta = \delta(a_s, a_t)$, но здесь рассмотрен более общий случай.

Теперь можно дать определение графа, моделирующего динамическую сеть:

$$G = (V, \delta, \rho)$$

Множество вершин графа, соответствующих добавляемым в сеть дронам обозначим за $\bar{V}$. Вершин, соответствующих дронам (дрону) игрока i – $\bar{V}_i$. Соответственно, $\bar{V} = \{(b, x) | b \in \bar{M}, x \in X\}$, $\bar{V}_i = \{(b_i, x) | b_i \in \overline{M_i}, x \in X\}$. Как только дроны помещены в сеть, исходный граф преобразуется в расширенный граф G^*:

$$G^* = (V^*, \delta, \rho), \quad V^* = V \cup \bar{V}$$

Заметим, что функция δ не меняется, так как она задаёт бинарное отношение не в виде множества пар, а в виде, собственно, некоторой функции.

Расширенным подграфом игрока i назовём граф:

$$G_i^* = (V_i^*, \delta, \rho), \quad V_i^* = V_i \cup \bar{V}$$

Важно отметить, что в расширенном подграфе конкретного игрока принимают участие все дроны всех игроков, поскольку дроны могут передавать информацию между любыми участниками сети, если эти участники могут взаимодействовать друг с другом.

В каждый отдельный момент, когда добавление/удаление агентов из сети не производится, её можно рассматривать как обычный граф $G = (V, E)$. Поскольку именно в такие моменты и производится оценка улучшения качества работы сети, дальнейшие определения, используемые для такой оценки, даны именно для обычных графов.

1.3 Основные определения

В данном разделе рассмотрены некоторые базовые понятия теории графов, используемые для оценки улучшения качества работы сети, а также приведены обоснования их выбора.

Рёбра (дуги) множества E будем обозначать e: $e = \{v_1, v_2\}$ ($e = (v_1, v_2)$), $v_1, v_2 \in V$

Инцидентными друг другу вершина v и ребро (дуга) e графа называются тогда, когда $e = \{v, v'\}$ ($e = (v', v)$ или $e = (v, v')$), где $v' \in V$.

Смежными называются два ребра (две дуги), инцидентные одной и той же вершине.

Взвешенный граф – это граф, каждому из ребёр (каждой из дуг) которого поставлено в соответствие некоторое число, называемое *весом* этого ребра. Если такого соответствия для графа нет, такой граф называется *невзвешенным*.

Маршрут в графе – конечная чередующаяся последовательность вершин и рёбер (дуг) $p = v_1, e_1, v_2, e_2, \dots, v_{n-1}, e_{n-1} v_n$, в которой $e_i = \{v_i, v_{i+1}\}$ ($e_i = (v_i, v_{i+1})$). *Длиной* маршрута в невзвешенном графе называется количество рёбер (дуг) в нём.

Расстоянием между парой вершин $v_1, v_2 \in V$ в невзвешенном графе называется длина кратчайшего из возможных маршрутов p, где $v_1, v_2 \in p$ (такое определение автоматически устанавливает вершины v_1 и v_2 как начальную и конечную вершины маршрута, а также исключает возможность наличия повторяющихся вершин и рёбер (дуг) в маршруте p, вследствие чего

не возникает необходимости вводить дополнительные определения перед определением расстояния).

Диаметром графа называется максимальное из возможных расстояний между какими-либо парами вершин в графе.

Диаметр исходного графа сети обозначим за d. Диаметр расширенного графа – за d^*.

Для численной оценки качества оптимизации сети в данном подходе используется именно изменение диаметра графа, так как для графа сети диаметр характеризует наихудшее время передачи информации между парами агентов. Так, функция выигрыша игрока i представляет собой численное значение уменьшения диаметра подграфа соответствующего игрока:

$$H_i = d_i - d_i^*$$

Поскольку рассматриваемый вариант игры носит кооперативный характер, было принято решение максимизировать суммарный выигрыш игроков. В данном случае функция суммарного выигрыша выглядит так:

$$H = \sum_{i=1,\dots n} H_i$$

1.4 Меры центральности и мера PageRank

После нахождения множества оптимальных решений задачи возникает вопрос – как выбрать единственное решение? Чтобы на него ответить, необходимо, во-первых, обратить внимание на вклад установленных дронов в скорость передачи информации в сети в целом. Во-вторых, постараться выйти за рамки поставленной задачи и обратить внимание на другие аспекты системы, с которой производится работа. Один из самых важных таких аспектов – это устойчивость работы сети к выходу некоторых узлов из строя. Например, если выйдет из строя самый «крайний» узел, взаимодействующий только с одним из узлов, то внутри оставшейся части сети взаимодействие никак нарушено не будет. Но если вышедший из строя узел окажется единственным соединяющим между двумя раздельными частями сети, то

структура распадётся надвое и две половины не смогут взаимодействовать друг с другом. Соответственно, следует поставить дрон в позицию, обеспечивающую, в некоторой степени, сохранение общей структуры сети при выходе центральных узлов из строя. Обе эти задачи – улучшение работы сети в целом и устойчивость к поломкам – можно решить одновременно, воспользовавшись мерами центральности [14] узлов сети.

Меры центральности узлов сети (вершин графа) позволяют определить значение «вклада», который каждый из узлов вносит в общую структуру. Под «вкладом» понимается численная характеристика узла, чем больше которая, тем важнее узел с точки зрения структуры сети – то есть при его выходе из строя (при удалении соответствующей ему вершины из графа сети), значительно нарушится взаимодействие оставшихся узлов.

В дальнейшем, поскольку описываемые модели непосредственно используют терминологию теории графов, помимо

Существует несколько подходов к расчёту мер центральности. Самый простой – подсчёт количества соседей узла – «центральность по степени»:

$$C_D(v) = \deg(v),$$

где $\deg(v)$ – количество соседей вершины $v \in V$.

В четырёх последующих иллюстрациях относительные значения мер центральности изображены при помощи раскраски узлов в разные цвета – узлы с бóльшим значением меры окрашены в более тёплые цвета (красный, бурый), с меньшим значением – в более холодные (голубой, тёмно-синий).

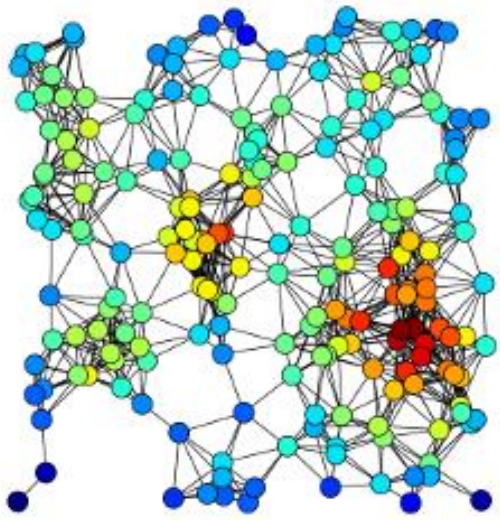

Рис. 1 – Иллюстрация меры центральности «центральность по степени»

Другая мера, называемая «центральность по близости», измеряет центральность узлов, определяя, насколько близко каждый из узлов расположен к другим. Для подсчёта степени близости некоторого узла в сети, необходимо суммировать его расстояния до каждого из остальных узлов и обратить полученное значение. Формула, таким образом, выглядит так [5,11]:

$$C_C(v) = \frac{1}{\sum_u dist(u, v)},$$

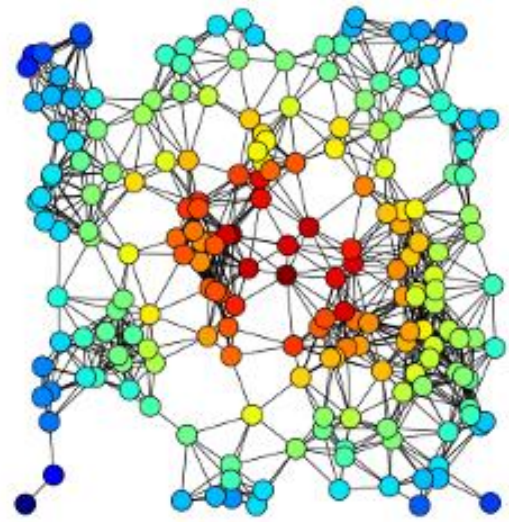

Рис. 2 – Иллюстрация меры центральности «центральность по близости»

Мера, называемая «центральность по посредничеству», более подходит под данную задачу с учётом исходной её постановки как задачи оптимизации передачи информации сети в целом, а также описанного выше требования сохранить структуру сети в случае выхода из строя «ключевых» узлов, через которые проходит наибольшее количество информации. При вычислении данной меры рассматриваются кратчайшие пути между всеми парами узлов в сети, учитывая при этом, через какие узлы эти пути проходят. Чем больше кратчайших путей проходит через некоторую вершину, тем больше её мера центральности с точки зрения передачи информации, так как алгоритмы маршрутизации вычисляют именно кратчайшие возможные пути в сети. Таким образом, формула вычисления данной меры выглядит следующим образом [17]:

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

где $\sigma_{st}(v)$ – количество кратчайших путей между узлами s и t, проходящими через узлы v, а σ_{st} – общее количество кратчайших путей между узлами s и t.

Данная мера предложена к использованию в данной задаче в [9]. На ближайших этапах развития программы она будет тестироваться и сравниваться с используемым сейчас критерием отбора.

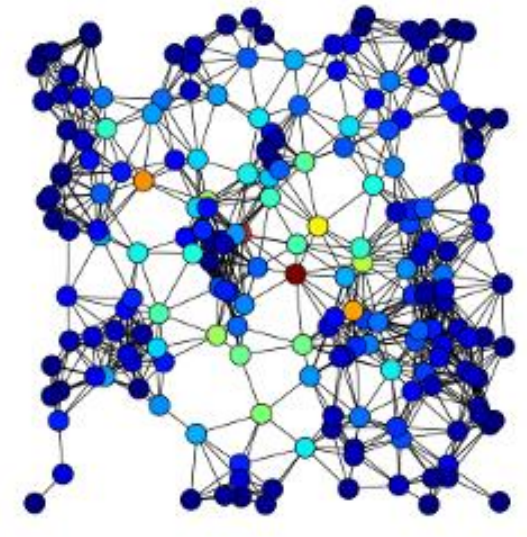

Рис. 3 – Иллюстрация меры центральности «центральность по посредничеству»

Матрицей смежности графа $G = (V, E)$ называется такая матрица A размера $|V| \times |V|$, в которой на пересечении строки i и столбца j стоит 1, если вершина с номером j является соседом вершины с номером i, и 0, если не является. Каждая вершина нумеруется уникальным номером от 1 до $|V|$.

Ещё один способ подсчитать центральность узлов в сети – воспользоваться мерой «центральности по собственному вектору». Подсчёт данной меры, соответственно названию, заключается в построении графа сети, нахождении его матрицы смежности A и вычислению собственного вектора v данной матрицы. Тогда i-е значение в собственном векторе будет являться мерой центральности для вершины i графа и, соответственно, мерой центральности узла, соответствующего i-й вершине графа. Формула для собственного v вектора матрицы A:

$$Av = \lambda v,$$

где λ – наибольшее собственное значение матрицы A [13].

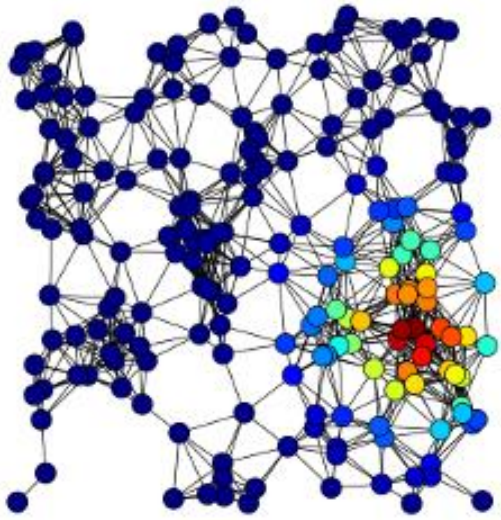

Рис. 4 – Иллюстрация меры центральности «центральность по собственному вектору»

Развитием меры центральности по собственному вектору стал метод ранжирования узлов сети (вершин графа), названный PageRank [8]. Данный метод был разработан в Стэнфордском университете Ларри Пэйджем и Сергеем Брином и в дальнейшем применялся для ранжирования результатов поисковых запросов в системе Google. Данная мера, а именно – её упрощённый вариант, была применена в данной работе для отбора единственного решения из множества оптимальных.

Данный метод работает с ориентированными графами сетей и интерпретирует дуги графа как придание конечной вершине дуги (соответствующему узлу сети) значимости, численное значение которой зависит, в свою очередь, от численного значения значимости вершины, являющейся началом дуги. Кроме того, когда вершина придаёт значимость нескольким вершинам сразу, она разделяет её поровну между всеми. Таким образом, получается взаимная зависимость (как и в случае с центральностью по собственному вектору). Эту зависимость можно описать следующим образом: пусть дан граф $G = (V, E)$, $B = \{b_{ij}\}$ – матрица, составленная по правилу:

$$b_{ij} = \begin{cases} \frac{1}{z_j}, \text{если } \exists(j,i) \in E, z_j - \text{количество соседей вершины } j \\ 0, \text{иначе,} \end{cases}$$

$A' = -I + B$. Тогда вектор $\mathbf{p}$ значений меры PageRank можно получить, решив системы линейных однородных уравнений: $A\mathbf{p} = 0$.

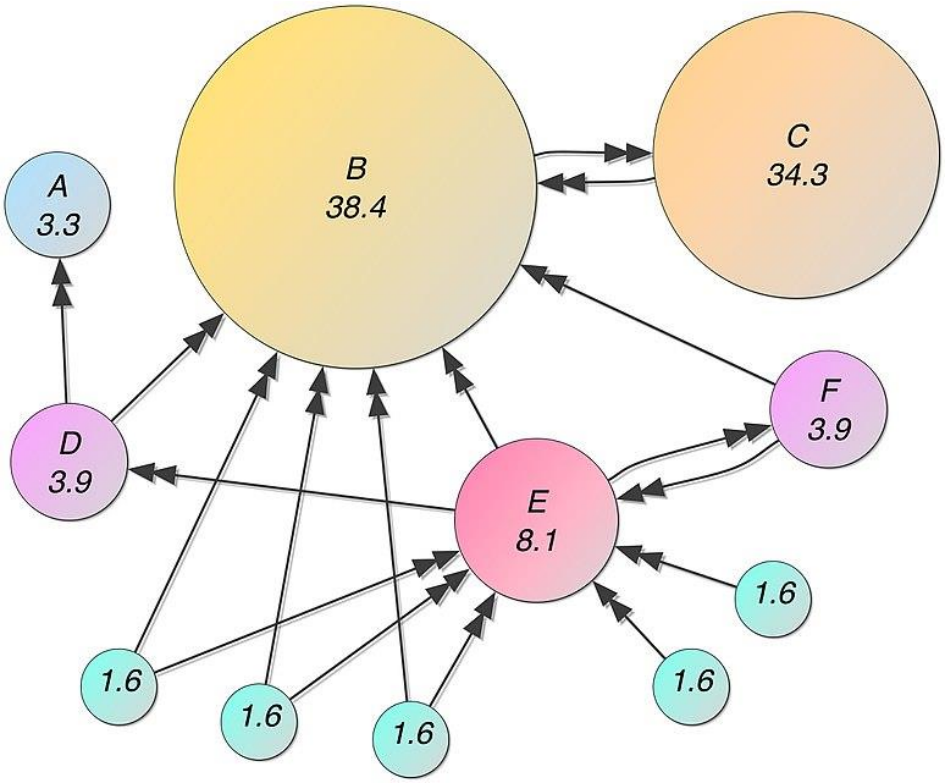

Рис. 5 – Иллюстрация меры центральности PageRank

Глава 2. Алгоритм решения задачи

2.1 Сеть на прямоугольной решётке

В пункте 1 главы 1 было упомянуто, что агенты сети располагаются в некотором координатном пространстве X. При переходе к реальным условиям, это пространство будет являться трёх- или более- мерным, однако на данном этапе моделирования сеть рассматривается на плоскости и, в целях дальнейшего упрощения, координаты, в которых могут располагаться агенты, ограничиваются целыми числами. В результате пространство X представляет собой квадратную решётку: $X = \mathbb{Z}^2$. Функция расстояния на нём – евклидово расстояние: $\rho(a, b) = \sqrt{(a.x - b.x)^2 + (a.y - b.y)^2}$

Устанавливать связи для взаимодействия могут только агенты сети, находящиеся в соседних узлах решётки, т.е. евклидово расстояние между которыми равно 1 (в данном случае, при проверке возможности взаимодействия, также оказывается корректным вычислять Манхэттенское расстояние вместо евклидового).

Здесь рассматривается случай с двумя игроками в сети. Обобщение на большее количество игроков для применённых методов выполняется тривиально.

2.2 Алгоритм

Итак, для решения задачи необходимо определить, какие из возможных наборов позиций дронов (наборов стратегий игроков) дают наилучшее суммарное уменьшение диаметров (максимальное значение функции суммарного выигрыша H) и после этого выбрать из них тот набор, в котором сумма мер PageRank позиций наибольшая. Однако сразу возникает вопрос – как выявить исходное множество возможных наборов позиций, которые будут затем оценены на способность уменьшить диаметры подграфов игроков? Понятно, что перебрать все возможные позиции на местности не получится, так как их бесконечно много.

Один из способов выбора – очертить вокруг имеющегося графа сети прямоугольник и последовательно перебирать пары узлов решётки в нём:

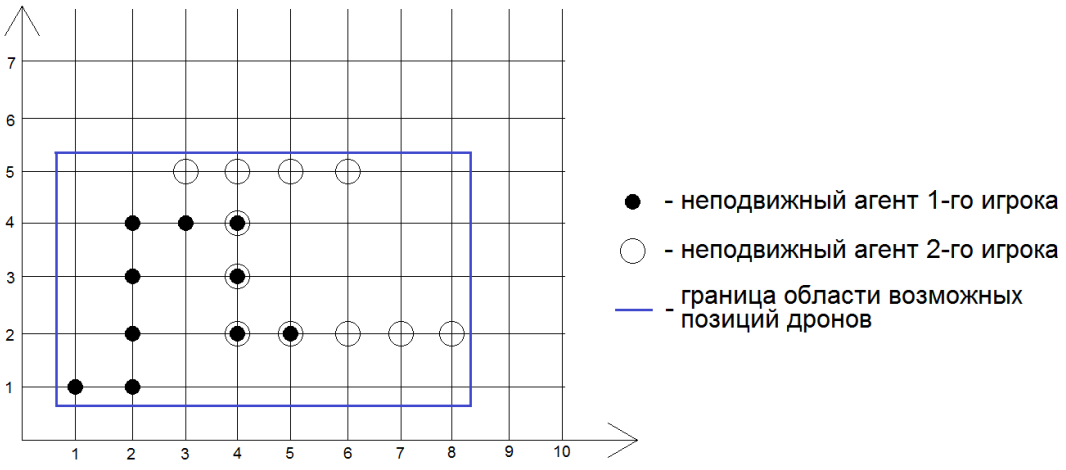

Рис. 6 – Ограничение графа прямоугольником

Действительно, очевидно, что при размещении дронов вне пределов такого прямоугольника, диаметр расширенного графа будет больше диаметра исходного графа, поэтому имеет смысл ставить дроны только внутрь. Но такой способ выбора исходного множества слишком грубый – зачастую при его использовании внутрь прямоугольника будет входить много пустых узлов решётки, стоящих далеко от графа, что негативно скажется на скорости работы алгоритма, так как сложность полного перебора n позиций в случае, когда в игре присутствуют всего 2 игрока – $O(n^2)$.

Был предложен более точный способ выбора исходного множества вершин, который, в случае небольших графов, будет иметь схожую, а иногда меньшую, по сравнению с методом очерчивания графа прямоугольником, производительность, но при работе с графами бóльших размеров, игнорирует значительную часть неподходящих позиций, которые попадают под рассмотрение в методе с прямоугольником. Данный способ оставляет для перебора только те узлы решётки, которые либо находятся непосредственно рядом с вершинами графа (на расстоянии 1 от них), либо, находясь внутри очерченного внутри графа прямоугольника, образуют связный подграф, хотя бы 2 вершины которого соседствуют с вершинами исходного графа (находятся

на расстоянии 1 от них). Второе условие проверяется уже непосредственно в ходе перебора для каждого конкретного набора вершин.

Теперь опишем весь алгоритм целиком. Первый этап: выбрать множество узлов решётки для перебора. Второй этап: перебрать выбранные на первом этапе узлы, отбрасывая в процессе перебора те наборы, которые точно не смогут способствовать уменьшению диаметра графа, и отобрать наборы, дающие наибольшее улучшение. Третий этап: для каждого из наборов подсчитать сумму мер PageRank узлов в нём. Четвёртый этап: выбрать набор с максимальной суммой, посчитанной на третьем шаге.

Для наглядности приведём блок-схему этого алгоритма:

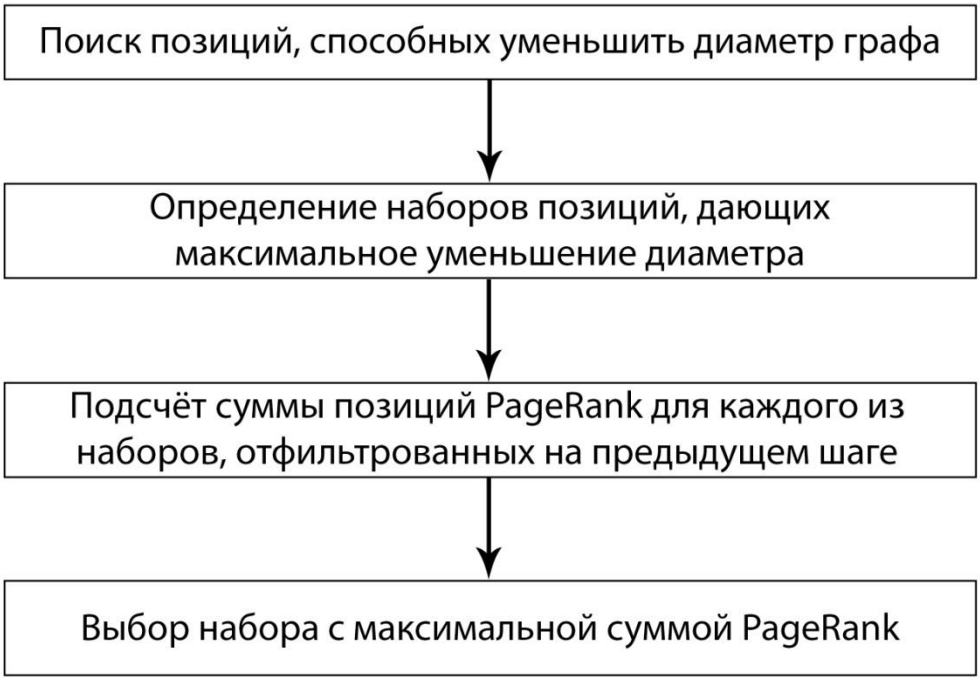

Рис. 7 – Блок-схема алгоритма решения задачи

Глава 3. Программная реализация решения задачи

3.1 Структура программы

Описанная здесь структура не является конечной для написанной программы, так как постоянно ведётся поиск способов её улучшения – как с точки зрения удобства поддержки программного кода, так и с точки зрения оптимизации времени работы программы и занимаемой ею памяти.

Итак, задача программы – реализовать описанный в предыдущем пункте алгоритм. Для разработки был выбран язык C++ по причине его универсальности и широкого круга разработчиков, владеющих им, благодаря чему дальнейшим развитием программы будет способен заняться практически любой программист, ознакомившись с уже имеющимся кодом.

Основной принцип, применённый при написании программы – удобство добавления новых параметров в сетевую структуру на дальнейших этапах, а также удобство добавления новых методов работы с данными, в связи с чем в описании структуры сети были в значительном объёме использованы обобщающие конструкции языка C++.

Программный код выложен в репозиторий на ресурсе GitHub: https://github.com/KKastaneda/NetworkOptimization

Сетевую структуру программы и работу с ней реализует 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. Фактически, этот параметр задаёт множество X. Второй параметр – `TAgent` – определяет тип данных, хранящий информацию об агентах сети – задаёт множество M. Третий параметр – `DoNeighbor`. На место этого параметра при создании экземпляра класса Network необходимо подставить класс или структуру, реализующую оператор `()` с двумя входными параметрами – два объекта

TAgent и возвращающий значение «истина», если агенты могут взаимодействовать друг с другом и значение «ложь», если не могут. Реализует функцию δ. Четвёртый параметр – Measure – задаёт тип возвращаемого значения функции расстояния и, таким образом, определяет множество Y. Пятый параметр – CalculateDistance. На место этого параметра при создании класса Network необходимо подставить класс или структуру, реализующую оператор () с двумя входными параметрами – два объекта TPosition и возвращающий значение расстояния между двумя координатами. Задаёт функцию ρ. Шестой параметр – Compare. На место этого параметра при создании класса Network необходимо подставить класс или структуру, реализующую оператор () с двумя входными параметрами – два объекта TPosition. Задаёт линейный порядок на множестве X, что требуется из-за выбранного способа хранения узлов сети. По умолчанию (если этот параметр пропущен), вместо него задаётся стандартный класс C++ std::less, что предполагает наличие линейного порядка на множестве X без явного определения функции сравнения координат.

Для хранения сети как графа используются списки смежности – то есть, при каждом узле хранится информация о его соседях. Чтобы реализовать списки смежности, в классе Network был создан внутренний класс NetworkAgentData, который хранит в себе указатель на объект класса TAgent, информацию о том, кто является соседом данного агента и для кого сам агент является соседом (список обратных соседей). Список обратных соседей нужен для возможности реализации ориентированного графа сети, чтобы при удалении узла из графа не было необходимости выполнять проход по всем узлам и проверять, если ли в их списке соседей удаляемый узел. Также реализован внутренний для класса Network класс NetworkAgentDataComparator, задающий линейный порядок на множестве классов NetworkAgentData, с целью возможности доступа к ним за логарифмическое время, что актуально при удалении узлов из сети.

Экземпляры класса NetworkAgentData хранятся в ассоциативном массиве по ключу TPosition: map<TPosition, set<NetworkAgentData*, NetworkAgentDataComparator>, Compare>. Ассоциативный контейнер set позволяет обеспечить быстрое добавление и удаление узлов из сети.

Исходно предполагалось, что класс Network будет выполнять значительную часть работы алгоритма, но после того, как было решено, что в нём будут решаться только задачи, связанные с графом сети, надобность в хранении узлов описанным выше образом пропала, так как класс не оперирует данными о положении агентов сети в пространстве (на множестве X). Однако, после рассмотрения способов оптимизации скорости работы программы, было выяснено, что, поскольку при добавлении агентов в сеть необходимо определять их соседей, можно передавать в Network координаты границы области, в которой соседи агента могут располагаться, а в методах класса будет реализован проход по узлам решётки внутри ограниченной области (такой способ будет эффективен при малых радиусах работы приёмо-передатчиков и большом количестве агентов). Тогда получается, что смысл хранить узлы описанным способом есть, и он был оставлен.

Реализован внутренний для класса Network класс Iterator, позволяющий упростить написание программного кода, где необходимо совершить проход по всем имеющимся узлам.

Методы класса, занимающие бóльшую часть программного кода, относятся к работе с сетевой структурой.

Вторая часть программы непосредственно реализует описанный алгоритм – создаёт экземпляр класса Network, добавляет в него агенты, определяет исходное множество узлов решётки для перебора, осуществляет перебор, вычисляя изменение диаметров подграфов игроков, пользуясь методом Network.GetDiameter(...), и определяет оптимальные наборы, после чего, пользуясь методом Network.GetPageRank(...), выбирает наилучший набор из найденных и выводит его на экран.

На следующих итерациях написания программы планируется во-первых: обобщить структуру хранения узлов сети с помощью таких механизмов языка C++, как виртуальные (`virtual`) методы (фактически, здесь имеется ввиду написание интерфейса) или уже упомянутые шаблоны (`template`). Это даст возможность быстро переключаться между способами хранения узлов сети, что положительно скажется на скорости обработки ситуаций, в которых граф удобно хранить не в виде списка смежности. Во-вторых, реорганизовать хранение узлов в сети таким образом, чтобы подграф каждого из игроков хранился отдельно. Поскольку в данной постановке неподвижные агенты игроков не могут взаимодействовать друг с другом, возможно разделение всех узлов на отдельные подграфы. Такой способ хранения повысит скорость добавления и удаления новых узлов в тех случаях, когда среди подграфов игроков не будет единственного, значительно превышающего размеры других, подграфа некоторого игрока.

3.2 Примеры решения задачи

Здесь приведены примеры задач, решённых при помощи написанной программы, с рисунками для каждого этапа решения.

Пример 1:

Исходный граф:

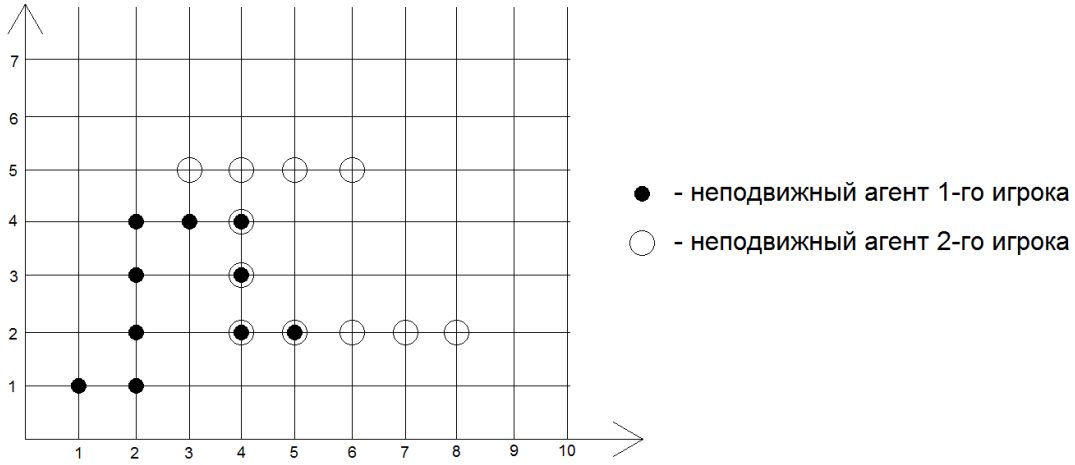

Рис. 8 – Пример 1 – Граф сети

Здесь изображены 2 подграфа игроков. У одного из них (обозначенного чёрными кругами) диаметр равен 9 (расстояние от вершины (1;1) до вершины (5;2)), у другого (обозначенного окружностями) – также 9 (расстояние от вершины (6;5) до вершины (8;2)).

Далее выбрано исходное множество узлов, которые можно использовать при переборе.

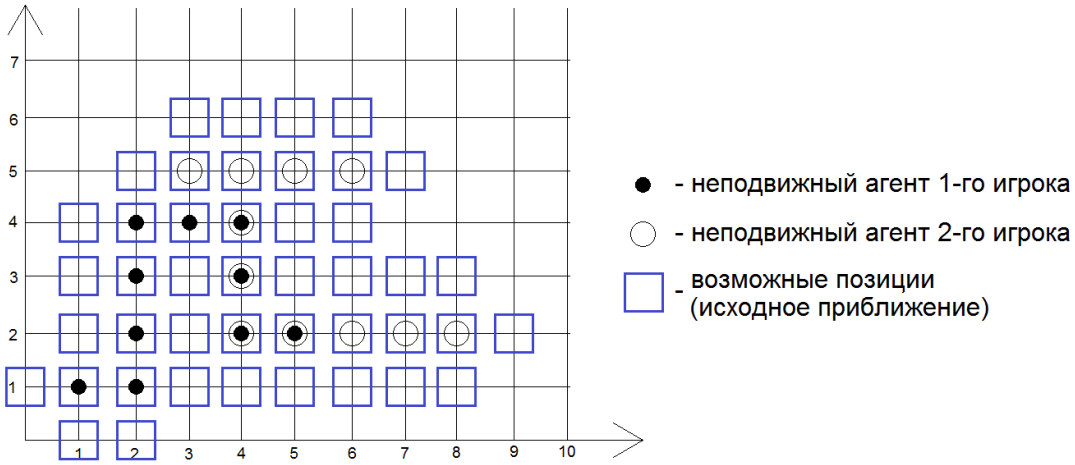

Рис. 9 – Пример 1 – Исходное множество

Путём перебора были выявлены пары узлов решётки, дающих наибольшее значение суммарного выигрыша. На данной иллюстрации пары изображены следующим образом: образующие одну пару узлы отмечены одним цветом, а предназначенные для разных игроков узлы имеют разную форму. Для этих наборов пар полученное улучшение диаметра подграфа 1-го игрока составляет 3 единицы, 2-го игрока – 0.

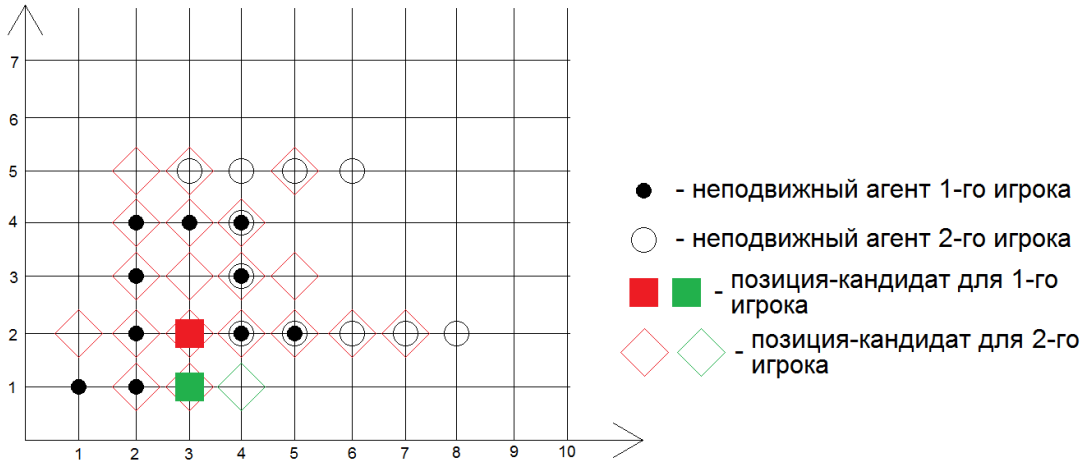

Рис. 10 – Пример 1 – Множество оптимальных наборов

После подсчёта PageRank для полученных на предыдущем этапе пар, была выбрана пара $(3; 2), (4; 2)$. Суммарный выигрыш составил $3 + 0 = 3$.

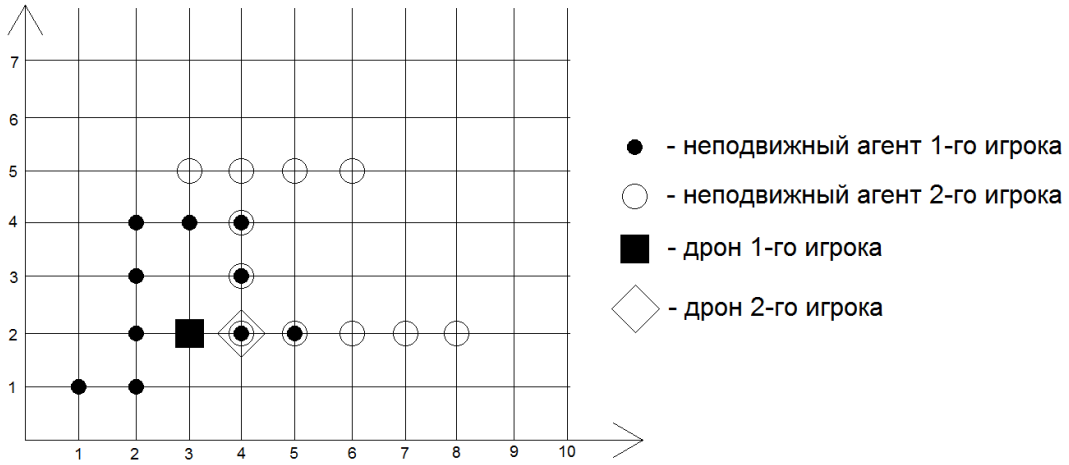

Рис. 11 – Пример 1 – Решение задачи

Пример 2:

Исходный граф:

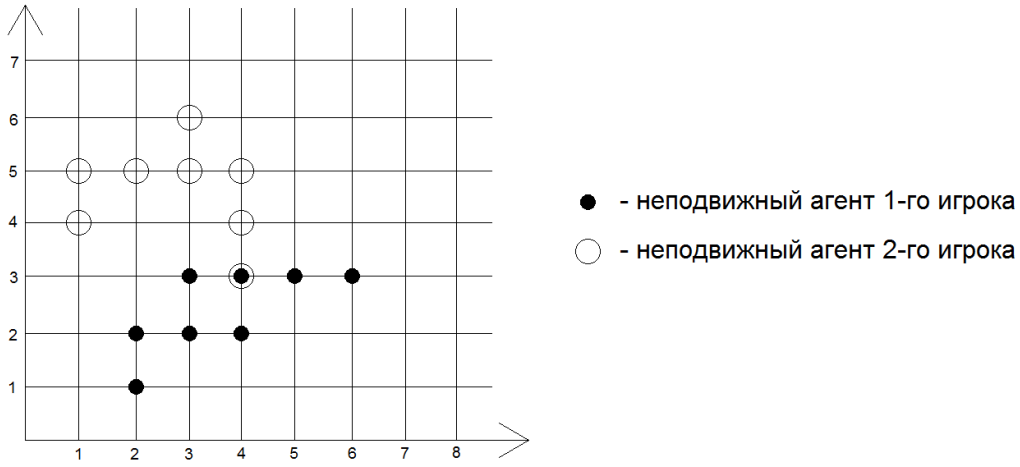

Рис. 12 – Пример 2 – Граф сети

В данном примере диаметр обоих подграфов игроков равен 6. Для подграфа первого игрока – это расстояние от вершины (2;1) до вершины (6;3), для подграфа второго игрока – от вершины (4;3) до вершины (1;4).

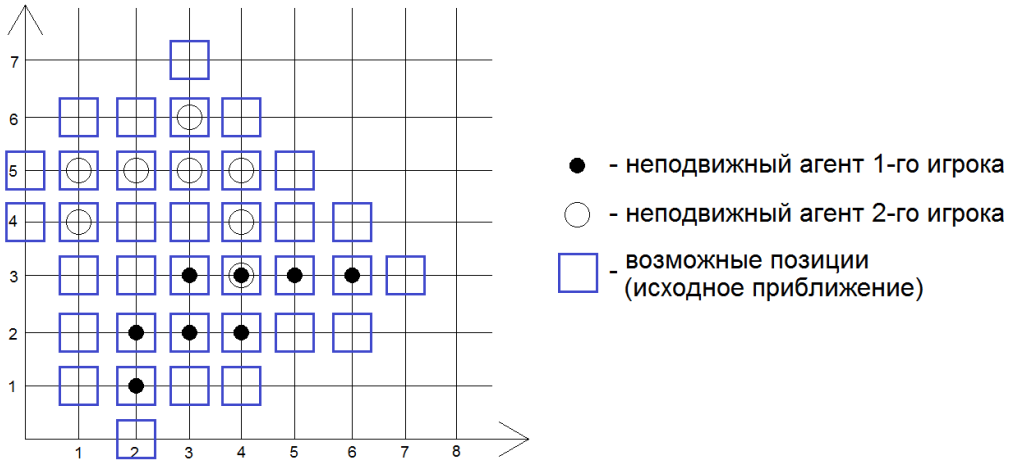

Рис. 13 – Пример 2 – Исходное множество

Исходное множество узлов для перебора было построено. Теперь из него необходимо оставить только оптимальные наборы пар узлов.

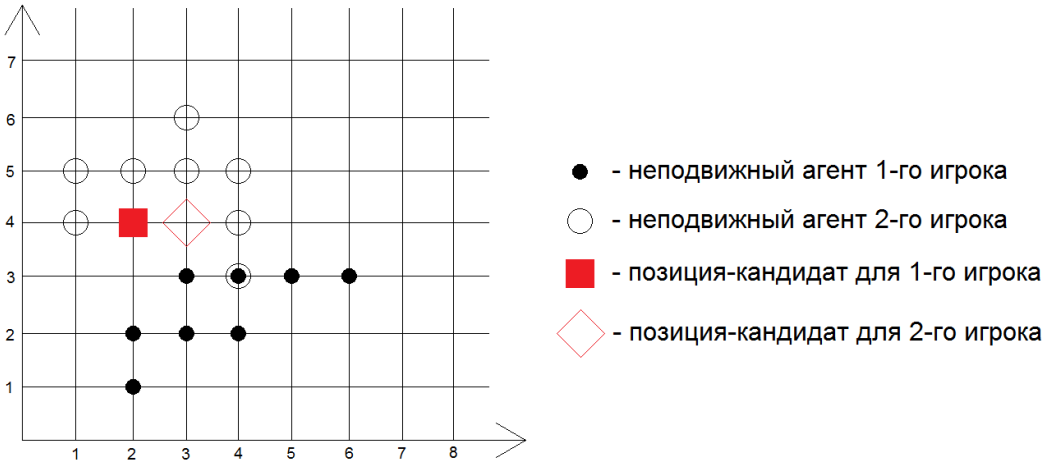

Рис. 14 – Пример 2 – Множество оптимальных наборов

В данном примере, в отличие от предыдущего, множество оптимальных наборов состоит только из одной пары – (2;4), (3;4). Этот набор уменьшает диаметр подграфа второго игрока на 1 – теперь диаметром является расстояние между узлами (4;3) и (1;5).

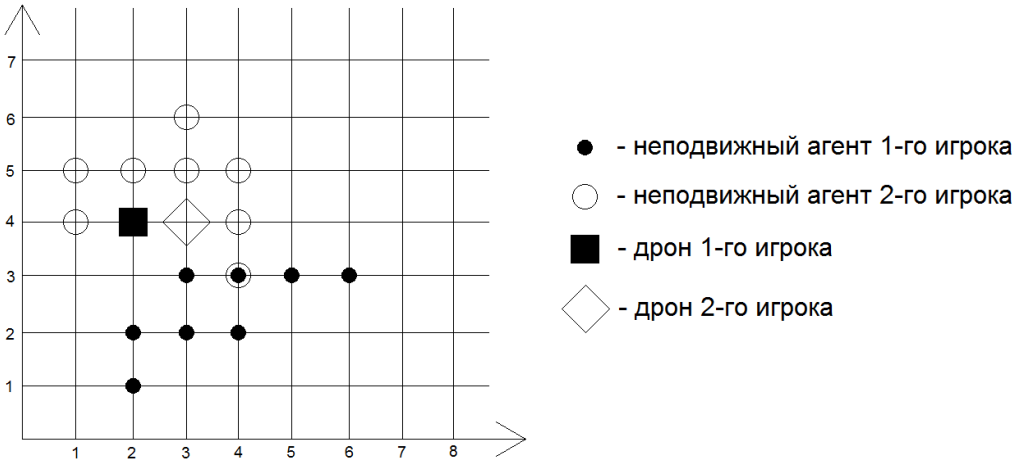

Рис. 15 – Пример 2 – Решение задачи

Единственный набор, полученный на предыдущем шаге, очевидно, и будет решением задачи.

Заключение

В данной работе сформулирована задача оптимизации передачи информации в самоорганизующихся сетях с использованием графов в качестве модели сети, а также рассмотрен теоретико-игровой подход к постановке задачи. За численную меру оптимизации выбрано уменьшение диаметров подграфов каждого из игроков. В качестве критерия отбора решения из множества оптимальных выбрана мера PageRank.

Написана и протестирована программа, реализующая алгоритм поиска решения задачи. Программа позволяет быстро добавлять новые параметры в задачу для исследования их влияния на результат.

Некоторые результаты, полученные в ходе работы над ВКР, также опубликованы в [2].

В дальнейшем планируется протестировать меру центральности по посредничеству как критерия отбора решения из множества оптимальных, усовершенствовать программу, сделав её ещё более гибкой, и улучшить ее время работы.

Кроме того, может быть изучена коалиционная структура игры на сети с дальнейшим распределением суммарного максимального выигрыша между игроками при помощи построения так называемой характеристической функции. В качестве принципа оптимальности в кооперативной игре может быть выбран вектор Шепли.

Отметим, что анализ работы программы для конкретных конфигураций сетей при помощи симулятора Network Simulator -3 проведен в ВКР Голубевой Марии «О совместном использовании узлов децентрализованной беспроводной самоорганизующейся сети», в которой показано существенное улучшение характеристик сети при добавлении дополнительных узлов согласно алгоритму, представленному к защите в настоящей ВКР.

Список литературы

1. Воронцов А. О поиске местоположения дронов для оптимизации работы сети типа MANET // Процессы управления и устойчивость. 2019. Т. 6. Вып. 1. С. 404–408.
2. Киреев С. А. «Оптимизация передачи информации в самоорганизующихся сетях» // Процессы управления и устойчивость. 2020
3. Мазалов В.В., Чиркова Ю.В. Сетевые игры. 1 изд. Лань, 2018.
4. Тимонин Н. О. «Одна динамическая игра управления мобильными агентами в сети» https://nauchkor.ru/uploads/documents/587d36465f1be77c40d58b3f.pdf
5. Alex Bavelas Communication Patterns in Task-Oriented Groups // The Journal of the Acoustical Society of America. 1950. Vol. 22. P. 725–730.
6. Anjum S. S., Noor R. M., Anisi M. H. Survey on MANET Based Communication Scenarios for Search and Rescue Operations // 2015 5th International Conference on IT Convergence and Security (ICITCS). IEEE, 2015. P. 1–5
7. *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.
8. 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
9. Ekaterina 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

10. Game Theory in Wireless and Communication Networks. Theory, Models, and Applications. / Han Z., Niyato D., Saad D., Basar T., Hjorungnes A., New York: Cambridge University Press, 2012.
11. Gert Sabidussi The centrality index of a graph // Psychometrika. 1966. Vol. 31. P. 581–603.
12. 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.
13. M. E. J. Newman The mathematics of networks. 2006
14. Mark Newman. Networks: An Introduction. 1st Edition. Oxford University Press, 2010.
15. Novikov D. A. Games and networks // Automation and Remote Control. 2014. Vol. 75. №6 P. 1145–1154.
16. O. Erim and C. Wright, "Optimized mobility models for disaster recovery using UAVs," 2017 IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC), Montreal, QC, 2017, P. 1–5. doi: 10.1109/PIMRC.2017.8292716
17. Ulrik Brandes A faster algorithm for betweenness centrality // Journal of Mathematical Sociology. 2001. Vol. 25. P. 163–177.

Приложение. Программный код

Файл «Network.hpp»

```
#pragma once

#ifndef MANET_PROJECT_NETWORK_HEADER_DEFINED
#define MANET_PROJECT_NETWORK_HEADER_DEFINED

#include <map>
#include <set>
#include <functional>
#include <queue>

using namespace std;

/*
	We imply that anything can be a position
	for an agent of a network and anything
	can be an agent. So both of them are templates -
	TPosition and TAgents respectively.

	Compare must define a total order on a set
	of all possible TPositions.

	DoNeighbor is supposed to be a class or a struct
	that overrides an invocation operator - that is "()".
	The invokation operator must take 2 TAgents and return
	bool - true if these agents can communicate in
	some way, and false otherwise.

	Measure represents a type of data used to calculate
	distances on the network. And CalculateDistance must
	be a struct or a class that overrides an invocation
	operator so that it takes 2 TAgents and returns a
	value of type Measure, indicating distance between
	the TAgents.
*/
template <
	class TPosition,
	class TAgent,
	typename DoNeighbor,
	class Measure,
	typename CalculateDistance,
	typename Compare = less< TPosition >
>
class Network
{
private:
	int agentsCount = 0;

	/*
	 * This subclass is used to store network
	 * agents information.
	 */
	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)
		{
			/*
				This is an absolutely awesome way to
				sort these classes inside STL containers,
				since we may have two agents with absolutely
				identical properties and location (position),
				but we may never have two identical agent
				pointers direct to different agents
			*/
			return ((int)lhs->agent) < ((int)rhs->agent);
		}
	};

	/*
	 * From the definition you can figure out that
	 * networkAgents maps positions (coordinates)
	 * to the network agents that are located in
	 * those positions. Note, that a single position
	 * may contain as many agents as needed.
	 *
	 * Compare is a functor-class that defines a
	 * linear order over a set of all possible positions.
	 *
	 * It's not std::set because
	 */
	map<TPosition, set<NetworkAgentData*, NetworkAgentDataComparator>, Compare>
networkAgents;

	/*
	 * class Iterator - simplifies iterating through networkAgents
	 *
	 * If you'd like to iterate through the whole set of existing
	 * agents, which is stored in a map of position - set_of_agents
```

```
         * pairs, you would need a nested loop where the outer loop
         * iterator would be a map<...>::iterator and the inner loop
         * iterator would be a set<...>::iterator. This class flattens
         * those nested loops, making the iteration process possible
         * with a single loop. The usage is very similar to stl iterators -
         * you initialize a beginning iterator with a BeginIterator() method,
         * then you advance it one step forward with a pre-increment
         * or post-increment operator and then to figure out when to stop
         * you compare the current iterator against the EndIterator().
         * Accessing the element to which the iterator points is done
         * by using either dereferencing operator: * or ->
         *
         * Defines:
         *            operator== and operator!= - check if iterators
         *                  are equal or if they are not equal respectively.
         *
         *            operator++() and operator++(int) - pre-increment
         *                  and post-increment respectively. Move iterator
         *                  one step forward, returning either an old or
         *                  an updated value, correspondingly to the type
         *                  of increment used.
         *
         *            operator*()
         */
        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(function<bool(TAgent)> filter)
		{
			for (
				Iterator agentsIterator = BeginIterator();
				agentsIterator != EndIterator();
				agentsIterator++
				)
			{
				NetworkAgentData* candidate =
	const_cast<NetworkAgentData*>(*agentsIterator);
				if (filter(*(candidate->agent)))
				{
					return candidate;
				}
			}

			return NULL;
		}

	#pragma region Functions for matrix operations

		/*
		 * The functions below are written to simplify
		 * operations with matrices. SwapColumns is much
		 * slower then SwapRows due to the how the matrix
		 * are stored (rowswisely).
		 *
		 * TODO:
		 * Addition and multiplication operations are
		 * not parallelled, though they should be.
		 */

		bool IsZero(double value)
		{
			return( abs(value) < 1e-7 );
		}

		bool NonZero(double value)
		{
			return( !IsZero(value) );
		}
```

```
	void SwapRows(vector<vector<double>> & matrix, int row1, int row2)
	{
		swap(matrix[row1], matrix[row2]);
	}

	void SwapColumns(vector<vector<double>> & matrix, int col1, int col2)
	{
		int fin = martix.size();

		for (int i = 0; i < fin; i++)
		{
			swap(matrix[i][col1], matrix[i][col2]);
		}
	}

	void AddRow(vector<vector<double>> & matrix, int srcrow, int dstrow, double
multiplier)
	{
		for (int i = 0; i < matrix[0].size(); i++)
		{
			matrix[dstrow][i] += matrix[srcrow][i] * multiplier;
		}
	}

	void AddColumn(vector<vector<double>> & matrix, int srccol, int dstcol, double
multiplier)
	{
		for (int i = 0; i < matrix.size(); i++)
		{
			matrix[i][dstcol] += matrix[i][srccol] * multiplier;
		}
	}

	void MulRow(vector<vector<double>> & matrix, int row, double multiplier)
	{
		for (int i = 0; i < matrix[0].size(); i++)
		{
			matrix[row][i] *= multiplier;
		}
	}

	void MulColumn(vector<vector<double>> & matrix, int col, double multiplier)
	{
		for (int i = 0; i < matrix.size(); i++)
		{
			matrix[i][col] *= multiplier;
		}
	}

#pragma endregion

	/*
	 * bool SolveSLE(vector<vector<double>> matrix, vector<double> rhs) -
	 *     solves a system of linear equations.
	 *
	 *     Returns bool - true if the system if consistent, false otherwise
	 *
	 *     vector<vector<double>> matrix - the matrix of the system
	 *
	 *     vector<double> rhs - the right hand side column, if it's
	 *         a zero vector, the system is homogeneous
	 *
	 *     vector<double> solution - stores the solution of the SLE
	 *
```

```
         *     The 'solution' values are unknown if the function
         *     returns 'false'. If the system has an infinite amount
         *     of solutions, the free variables will be set to 1.0
         *     and the dependent ones will be calculated correspondingly
         */
        bool SolveSLE(vector<vector<double>> & matrix, vector<double> & rhs,
vector<double> & solution)
        {
                int equations_number = matrix.size();
                int vars_number = matrix[0].size();

                int current_row = 0;

                /*
                 * the loop below performs a downward
                 * walk of the gaussian elimination
                 */
                // 'i' iterates through each variable
                for (int i = 0; i < vars_number; i++)
                {
                        if (current_row == equations_number)
                        {
                                break;
                        }

                        /*
                         * first, we need to find a row, such that the
                         * i-th value in it (the coefficient of the current
                         * variable) is a non-zero value, we call this
                         * a nonzero_row
                         */
                        int nonzero_row = current_row;
                        while (IsZero(matrix[nonzero_row][i]))
                        {
                                nonzero_row++;
                                // if there's no nonzero row, move to the next variable
                                if (nonzero_row == equations_number)
                                {
                                        break; // check out the 'if' block right below this
while loop
                                }
                        }

                        if (nonzero_row == equations_number)
                        {
                                continue; // jumps to the next iteration with i = i + 1
                        }

                        // replace the current row with the first found nonzero_row
                        SwapRows(matrix, current_row, nonzero_row);
                        swap(rhs[current_row], rhs[nonzero_row]);

                        /*
                         * 'j' iterates through the rows below the current, nullifying
                         * corresponding values in the matrix
                         */
                        for (int j = current_row + 1; j < equations_number; j++)
                        {
                                /*
                                 * if the corresponding value is already nullified,
                                 * move to the next row
                                 */
                                if (IsZero(matrix[j][i]))
                                {
```

```
                                continue;
                        }

                        // the number we multiply the current_row before subtracting
it from the row j
                        double coefficient = matrix[j][i] / matrix[current_row][i];

                        for (int k = i; k < vars_number; k++)
                        {
                                matrix[j][k] -= matrix[current_row][k] * coefficient;
                        }

                        rhs[j] -= rhs[current_row] * coefficient;
                }

                current_row++;
        }

        if (current_row < equations_number)
        {
                /*
                 * in this case we need to check if
                 * any of the right-hand-side values
                 * below the current_row (including it)
                 * are non-zero - if this is true, then
                 * the SLE is inconsistent
                 */
                for (int i = current_row; i < equations_number; i++)
                {
                        if (NonZero(rhs[i]))
                        {
                                return false;
                        }
                }
        }

        // the only non-zero rows by now are 0 through current_now-1

        /*
         * this int below holds
         * vars_number-the number of calculated variables by the current moment
         * made for convenience, so that we don't need to iterate
         * until vars_number-the_other_possible_value (check out the inner loops)
         */
        int calculated_offset = vars_number;

        /*
         * this loop down below performs an upward walk
         * of the gaussian elimination
         * it handles both cases - when there's a unique
         * solution and when there is an infinite amount
         * of solutions - as mentioned in the description
         * of the function, in this last case, it fills
         * free variables with 1.0 values
         */

        current_row--;

        for (; current_row >= 0; current_row--)
        {
                /*
                 * the leftmost nonzero value in any row
                 * for this moment can only be the diagonal
                 * value, which in-row index corresponds
```

```
                        * to the row number
                        */
                       int leftmost_nonzero = current_row;
                       while (IsZero(matrix[current_row][leftmost_nonzero]))
                       {
                              leftmost_nonzero++;
                       }

                       double lhs_calculated_part = 0.0;

                       // in this loop we fill up free variables with 1.0 values
                       for (int j = leftmost_nonzero + 1; j < calculated_offset; j++)
                       {
                              solution[j] = 1.0;
                              lhs_calculated_part += matrix[current_row][j];
                       }

                       for (int j = calculated_offset; j < vars_number; j++)
                       {
                              lhs_calculated_part += matrix[current_row][j] * solution[j];
                       }

                       solution[leftmost_nonzero] = (rhs[current_row] - lhs_calculated_part)
/ matrix[current_row][leftmost_nonzero];

                       /* okay, we've figured out the value for
                        * the current dependent variable
                        * (one for each row of the matrix)
                        * now we need to clean up the rows above
                        * the current one so that none of them
                        * contain this dependent variable as
                        * a summand (essentialy, this means that)
                        * we set the values in the corresponding
                        * column to 0, except the current row
                        */

                       for (int j = current_row - 1; j >= 0; j--)
                       {
                              double multiplier = matrix[j][leftmost_nonzero] /
matrix[current_row][leftmost_nonzero];
                              for (int k = leftmost_nonzero; k < vars_number; k++)
                              {
                                     matrix[j][k] -= matrix[current_row][k] * multiplier;
                              }
                       }

                       calculated_offset = leftmost_nonzero;
                }

                /*
                 * this happens if we end up with a matrix
                 * with zero columns on the left by now
                 * (this is only possible if we were given
                 * them filled with zeros initially)
                 */
                for (int i = 0; i < calculated_offset; i++)
                {
                       solution[i] = 1.0;
                }

                return true;
        }

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 we do not have any agents at this position
	if (networkAgents.count(position) == 0)
	{
		return;
	}

	NetworkAgentData dummyToFind(agent);

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

	// if there's no such agent
	if (agentData == networkAgents[position].end())
	{
		return;
	}

	NetworkAgentData* foundAgent = (*agentData);

	for (
		set<NetworkAgentData*>::iterator it = foundAgent->neighbors.begin();
		it != foundAgent->neighbors.end();
		it++
		)
	{
		(*it)->reverseNeighbors.erase(foundAgent);
```

```
		}

		for (
			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);
		}
	}

	/*
	 * Calculates the diameter of the subgraph constructed of
	 * the vertices that satisfy the filter.
	 *
	 * When calling this function you need to be sure that
	 * the subgraph is connected, otherwise you're gonna
	 * get a diameter of any of its connectivity components.
	 */
	Measure GetDiameter(function<bool(TAgent)> filter, function<bool(TAgent)>
findFirstFilter, 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 (
                                set<NetworkAgentData*>::iterator neighbors = current-
>neighbors.begin();
                                neighbors != current->neighbors.end();
                                neighbors++
                                )
                        {
                                NetworkAgentData* currentNeighbor = *neighbors;

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

                                // this condition is true if we've never met the agent before
                                if (agentsToIndicesMapping.find(currentNeighbor) ==
agentsToIndicesMapping.end())
                                {
                                        agentsToIndicesMapping[currentNeighbor] = subgraphsize;
                                        dp[subgraphsize][subgraphsize] = 0;
                                        subgraphsize++;
                                        bfsQueue.push(currentNeighbor);
                                }
                                // now we're sure that currentNeighbor has an index mapped to
it

                                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;
        }

        void GenerateAdjacencyMatrix(function<bool(TAgent)> filter,
vector<vector<double>>& buffer, vector<TAgent*>& indexedAgents)
        {
                buffer.resize(agentsCount);
```

```
		if (agentsCount == 0)
		{
			return;
		}

		for (int i = 0; i < agentsCount; i++)
		{
			buffer[i].resize(agentsCount, 0.0);
		}

		NetworkAgentData* root = FindFirst(filter);

		int subgraphsize = 0;
		map<NetworkAgentData*, int> agentsToIndicesMapping;

		queue<NetworkAgentData*> bfsQueue;

		agentsToIndicesMapping[root] = subgraphsize;
		subgraphsize++;
		bfsQueue.push(root);

		indexedAgents.push_back(root->agent);

		while (!bfsQueue.empty())
		{
			NetworkAgentData* current = bfsQueue.front();
			bfsQueue.pop();

			int currentFilteredNeighbors = 0;

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

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

				currentFilteredNeighbors++;

				// this condition is true if we've never met the agent before
				if (agentsToIndicesMapping.find(currentNeighbor) ==
agentsToIndicesMapping.end())
				{
					agentsToIndicesMapping[currentNeighbor] = subgraphsize;
					subgraphsize++;
					bfsQueue.push(currentNeighbor);

					indexedAgents.push_back(currentNeighbor->agent);
				}
				// now we're sure that currentNeighbor has an index mapped to
it

				// since these guys are neighbors, we set the value to 1.0

	buffer[agentsToIndicesMapping[currentNeighbor]][agentsToIndicesMapping[current]] =
1.0;
			}
```

```
		}

		for (int i = 0; i < buffer.size(); i++)
		{
			buffer[i].resize(subgraphsize);
		}

		buffer.resize(subgraphsize);
	}

	vector<pair<TAgent*, double>> GetPageRank(function<bool(TAgent)> filter,
function<bool(TAgent)> findFirstFilter)
	{
		vector<pair<TAgent*, double>> result;

		vector<vector<double>> ranks;

		ranks.resize(agentsCount);

		for (int i = 0; i < agentsCount; i++)
		{
			ranks[i].resize(agentsCount, 0.0);
		}

		NetworkAgentData* root = FindFirst(findFirstFilter);

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

		int subgraphsize = 0;
		map<NetworkAgentData*, int> agentsToIndicesMapping;

		queue<NetworkAgentData*> bfsQueue;

		agentsToIndicesMapping[root] = subgraphsize;
		subgraphsize++;
		bfsQueue.push(root);

		vector<int> filteredNeighbors;
		result.push_back(make_pair(root->agent, 0.0));

		while (!bfsQueue.empty())
		{
			NetworkAgentData* current = bfsQueue.front();
			bfsQueue.pop();

			int currentFilteredNeighbors = 0;

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

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

```
                        currentFilteredNeighbors++;

                        // this condition is true if we've never met the agent before
                        if (agentsToIndicesMapping.find(currentNeighbor) ==
agentsToIndicesMapping.end())
                        {
                                agentsToIndicesMapping[currentNeighbor] = subgraphsize;
                                subgraphsize++;
                                bfsQueue.push(currentNeighbor);
                                result.push_back(make_pair(currentNeighbor->agent,
0.0));
                        }
                        // now we're sure that currentNeighbor has an index mapped to
it

                        // since these guys are neighbors, we set the value to 1.0

        ranks[agentsToIndicesMapping[currentNeighbor]][agentsToIndicesMapping[current]] =
1.0;
                }

                filteredNeighbors.push_back(currentFilteredNeighbors);
        }

        if (subgraphsize == 1)
        {
                result.push_back(make_pair(root->agent, 0.0));
                return result;
        }

        for (int i = 0; i < subgraphsize; i++)
        {
                ranks[i][i] = -1.0;

                for (int j = 0; j < subgraphsize; j++)
                {
                        if (i != j)
                        {
                                ranks[j][i] /= ((double)filteredNeighbors[i]);
                        }
                }
        }

        result.resize(subgraphsize);
        for (
                std::map<NetworkAgentData*, int>::iterator it =
agentsToIndicesMapping.begin();
                it != agentsToIndicesMapping.end();
                it++
                )
        {
                result[it->second].first = it->first->agent;
        }

        vector<double> rightHandSide;
        rightHandSide.resize(subgraphsize, 0.0);

        vector<double> sleSolution;
        sleSolution.resize(subgraphsize);

        // now we have a matrix built

        if (!SolveSLE(ranks, rightHandSide, sleSolution))
        {
```

```
                        throw new exception("The SLE for calculating the PageRank is
invalid");
                }

                for (int i = 0; i < subgraphsize; i++)
                {
                        result[i].second = sleSolution[i];
                }

                return result;
        }

        Network() {}
};

#endif /* MANET_PROJECT_NETWORK_HEADER_DEFINED */
```

Файл «Entry.cpp»

```
#include "Network.hpp"

#include <map>
#include <set>
#include <string>
#include <iostream>
#include <vector>
#include <fstream>

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)
	{
		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 && (eitherIsADrone || samePlayer));
		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);
	}
};

int main()
{
	Network<Pos, Agent, DoNeighbor, int, CalculateDistance, PosComp> network;

	const int player1AgentsNumber = 8;
	const int player2AgentsNumber = 8;
```

```
int player1Pos[player1AgentsNumber][2] = {
	{2,1},{2,2},{3,2},
	{3,3},{4,2},{4,3},
	{5,3},{6,3}
};

int player2Pos[player2AgentsNumber][2] = {
	{1,4},{1,5},{2,5},
	{3,5},{3,6},{4,5},
	{4,4},{4,3}
};

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]);
}

int player1Diameter = network.GetDiameter(
	[](Agent x) {return (x.playerID == 0x01);},
	[](Agent x) {return (x.playerID == 0x01);},
	LONG_MAX / 2
);

int player2Diameter = network.GetDiameter(
	[](Agent x) {return (x.playerID == 0x02);},
	[](Agent x) {return (x.playerID == 0x02);},
	LONG_MAX / 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(0x12);

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

	for (int i = 0; i < candidates.size(); i++)
	{
		for (int j = i; j < candidates.size(); j++)
		{
			/*
				checking if the pair of positions
				can possibly provide some improvement
			*/

			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);},
```

```
                                        LONG_MAX / 2
                                );

                        int player2Gain =
                                player2Diameter - network.GetDiameter(
                                        [](Agent x) {return ((x.playerID == 0x02) ||
((x.playerID & 0x10) != 0));},
                                        [](Agent x) {return (x.playerID == 0x02);},
                                        LONG_MAX / 2
                                );

                        /*
                                At this point we have the drones set
                                and diameter calculated so we can perform
                                any computations needed to figure out if
                                this set of positions is valuable or not
                        */

                        // finding Pareto optimal solutions
                        int newResult = player1Gain + player2Gain;

                        if (paretoOptimal.size() != 0)
                        {
                                if (paretoOptimal.begin()->first <= newResult)
                                {
                                        if (paretoOptimal.begin()->first < newResult)
                                        {
                                                paretoOptimal.clear();
                                        }


        paretoOptimal[newResult].insert(make_pair(player1Drone->position, player2Drone-
>position));
                                }
                        }
                        else
                        {
                                paretoOptimal[newResult].insert(make_pair(player1Drone-
>position, player2Drone->position));
                        }

                        network.Remove(candidates[i], player1Drone);
                        network.Remove(candidates[j], player2Drone);
                }
        }
    }

    double bestPageRank = 0.0;
    pair<Pos, Pos> solution;

    for (map<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);
```

```
                        vector<pair<Agent*, double>> pageRank = network.GetPageRank(
                                [](Agent) {return true;},
                                [](Agent) {return true;}
                        );

                        double currentPageRank = 0.0;

                        for (int i = 0; i < pageRank.size(); i++)
                        {
                                if (pageRank[i].first == player1Drone || pageRank[i].first ==
player2Drone)
                                {
                                        currentPageRank += pageRank[i].second;
                                }
                        }

                        if (currentPageRank > bestPageRank)
                        {
                                bestPageRank = currentPageRank;
                                solution = make_pair(p1pos, p2pos);
                        }

                        network.Remove(p1pos, player1Drone);
                        network.Remove(p2pos, player2Drone);
                }
        }

        player1Drone->position = solution.first;
        player2Drone->position = solution.second;

        network.Add(solution.first, player1Drone);
        network.Add(solution.second, player2Drone);

        ofstream outfile("output.txt");

        vector<vector<double>> adjMatrix;
        vector<Agent*> indexedAgents;

        const int numberOfPlayers = 2;

        outfile << numberOfPlayers << endl;

        for (int playerNumber = 1; playerNumber <= numberOfPlayers; playerNumber++)
        {
                network.GenerateAdjacencyMatrix(
                        [playerNumber](Agent x) {return ((x.playerID == playerNumber) ||
((x.playerID & 0x10) != 0));},
                        adjMatrix,
                        indexedAgents
                );

                outfile << indexedAgents.size() << endl;

                for (int i = 0; i < indexedAgents.size(); i++)
                {
                        outfile << indexedAgents[i]->position.i << " " << indexedAgents[i]-
>position.j << " " << indexedAgents[i]->playerID << endl;
                }

                for (int i = 0; i < adjMatrix.size(); i++)
                {
                        for (int j = 0; j < adjMatrix.size(); j++)
                        {
                                outfile << adjMatrix[i][j] << " ";
```

```
                }

                outfile << endl;
            }

            indexedAgents.clear();

            for (int i = 0; i < adjMatrix.size(); i++)
            {
                adjMatrix[i].clear();
            }
            adjMatrix.clear();
        }

        outfile.close();

        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 << "(" << solution.first.i << ", " << solution.first.j << ") (" <<
            solution.second.i << ", " << solution.second.j << ")" << endl;

        cout << "Press Enter to quit...";

        fgetc( stdin );

        return 0;
}
```