Санкт-Петербургский государственный университет

Кафедра моделирования социально-экономических систем

Ринчинов Роман Цыдыпович

Многокритериальная задача оптимального размещения производства в сети

Магистерская программа:

Математическое и информационное обеспечение экономической деятельности

Научный руководитель:

кандидат физ.-мат. наук

Парфенов А. П.

Санкт-Петербург

2019

Содержание

Введение

Задача оптимального размещения производства в сети представляет практический интерес в экономике. При планировании развития производства часто возникает необходимость в решении задач оптимального размещения предприятий и складов. Во многих случаях такие задачи являются весьма сложными и требуют применения методов математического моделирования, разработки специальных алгоритмов и программного обеспечения. Исследованию таких задач посвящено множество работ [1]. В качестве критериев может служить множество различных параметров — это усложняет задачу, ведь не всегда понятно какие критерии более важны [2]. Для этой задачи в литературе рассмотрены многие подзадачи. Например:

— Многокритериальная задача размещения пунктов производства одного вида продукции [3];

— В своей публикации [4] я рассматривал алгоритм решения задачи размещения производства нескольких видов продукции;

— Задача о складировании [5];

В данной работе предлагается расширить задачу размещения производств добавлением новых объектов — пунктов хранения продукции(складов), а также применяя альтернативные подходы к принципам оптимальности. Складом будем называть место, где произведенная продукция может храниться, как в промежуточном пункте между производством и пунктами потребления товаров.

В работе рассмотрены различные методы решения многокритериальной задачи размещения, представляющие различные подходы к решению задачи(алгоритм, основанный на методе ветвей и границ, также эвристический алгоритм основанный на идеи генетической селекции множества парето оптимальных решений), выясняются их достоинства, недостатки и ограничения. В рамках данной работы выполнена программная реализация

этих методов на языках программирования C++ и PYTHON, позволяющая проверить эффективность и провести более детальное сравнение.

Обзор литературы

Исследованию таких задач размещения посвящено множество работ, например в статье [1] рассматриваются различные публикации по задачам размещениям. В качестве критериев для задачи рассматриваемой в этой работе может служить множество различных параметров — это усложняет задачу, ведь не всегда понятно какие критерии более важны [2].

Чаще всего рассматриваются задачи размещения пунктов производства одного вида продукции, а в качестве критериев оптимальности, которые следует минимизировать, берутся расстояния от пунктов производства до пунктов потребления. Так например Огрежак рассматривал [3] многокритериальную задачу, в которой необходимо разместить пункты производства одного вида продукции, а в качестве критериев он как раз рассматривает расстояния от пунктов производства до пунктов потребления. В качестве принципа оптимальности, согласующего эти критерии, ранее рассматривались оптимумы по Парето, сумма расстояний, максимум из расстояний. Оптимизация по принципу минимума суммы сводится к задаче о поиске медиан графа, а оптимизация по принципу минимума максимума — к задаче о поиске центров графа. Подобные задачи являются задачами целочисленного линейного программирования, для которых возможны различные алгоритмы точного и приближенного решения.

Я в статье [4] построил алгоритм поиска решения задачи размещения нескольких видов продукции для двух групп критериев. В своей бакалаврской выпускной работе я реализовал программный комплекс решающий задачу размещения нескольких видов продукции для двух видов критериев. А конкретнее — рассмотрены различные методы решения многокритериальной задачи размещения, представляющие различные подходы к задаче(алгоритм, основанный на методе ветвей и границ, также эвристический алгоритм находящий приближенные решения), выясняются их

достоинства, недостатки и ограничения. В рамках той работы выполнена программная реализация этих алгоритмов на языках программирования C++ и PYTHON, позволяющая проверить эффективность и провести детальное сравнение результатов работы алгоритмов.

1 Постановка задачи

Пусть имеется множество мест, в которых могут находится пункты производства, потребления и хранения. Эти места соединены между собой дорогами. Данную систему можно представить в виде ориентированного графа $G = (X,Y)$, X — множество вершин, Y — множество дуг между ними. Пронумеруем вершины $X = 1, \ldots, I$. Для удобства договоримся использовать символ i. Задана неотрицательная вещественная функция стоимости транспортировки одной единицы товара j из вершины i_1 в вершину i_2 обозначим ее $c_j(i_1,i_2)$. Будем рассматривать Q, последовательных промежутков времени. Пусть существует J различных товаров, которые расходуются в пунктах потребления и хранения в определенном количестве, т.е каждой вершине соответствует вектор $d_i^q = (d_{i1}^q, \ldots, d_{iN}^q)$, d_{ij}^q — потребление i-ым пунктом j-го товара в q-ый промежуток времени.

Для обеспечения всех городов товарами необходимо разместить $R = \sum_{j=1}^{J} r_j$ пунктов производства(r_j для продукции j) и p пунктов складирования. Будем считать, что в одном месте может находиться не более одного пункта производства. Также будем полагать, что пункт хранения(склад) может содержать различные виды и различный объем продукции. Будем полагать что для каждого места и каждого вида продукции задана стоимость хранения продукции. Также каждый тип заводов производит, в каждый из промежутков времени определенное количество товаров(p_j – количество товаров производимых заводом типа j в единицу времени).

Пусть $\hat{X}_j \subset X$ — подмножество вершин графа, в которых размещены пункты производства продукции j. Пусть $\overline{X} \subset X$ — подмножество вершин графа, в которых размещены пункты хранения.

Будем считать, в каждый интервал времени все, что произведено заводом отправляется в ближайший к нему склад. Пункт потребления получает необходимый товар с ближайшего склада.

Формулирование критериев

В качестве критериев оптимальности выбраны суммарная стоимость транспортировки товаров до пунктов потребления и стоимость хранения товаров. Стоит отметить, что в данной модели есть ограничения которые нужно учитывать во время поиска решений: в каждый момент времени на складе должно быть необходимое количество товаров для того чтоб все города получили их.

Суммарная стоимость транспортировки товаров

Пусть функция $\phi(i, Z)$ – возвращает ближайшую вершину из $Z \subset X$ к вершине i. В работе [4] были в критериев были взяты 1.1.

$$f_j(x) = \sum_{q=1}^{Q} \sum_{i=1}^{I} d_{ij}^q * \beta_j c_j(i, \phi(i,X_j)) \tag{1.1}$$

Перепишем критерии 1.1 для текущей задачи. Пусть $C_1(i,j)$ – стоимость транспортировки продукции j в пункт i с ближайшего склада, а $C_2(i,j)$ – стоимость транспортировки продукции с ближайшего завода производящего j к ближайшему складу от i.

$$C_1(i,j) = c_j(i, \phi(i, \overline{X}))$$

$$C_2(i,j) = c_j(\phi(i, \overline{X}), \phi(\phi(i, \overline{X}), X_j))$$

Тогда критерии 1.1 с учетом складирования примут вид

$$f_j = \sum_{q=1}^{Q} \sum_{i=1}^{I} d_{ij}^q * \beta_j [C_1(i,j) + C_2(i,j)] \tag{1.2}$$

– суммарная стоимость транспортировки товара j.

Ближайшие склады к пунктам потребления

Важным критерием для пунктов потребления является удаленность от складов, это позволяет быстрее и проще получать необходимые товары.

$$G_i = \hat{c}(i, \phi(i,\overline{X})) \tag{1.3}$$

– где длина кратчайшего пути i из в $\phi(i,\overline{X})$

Стоимость хранения товаров

С появлением складирования в модели, имеет смысл расширить пространство критериев, введем критерии связанные с хранением. Пусть

$$\eta_j(\delta) = \begin{cases} 1, & \text{if } \delta < \xi_j \\ 0, & \text{if } \delta >= \xi_j \end{cases}$$

– функция которая показывает испортится ли товар j за δ интервалов времени. Пусть $\Delta_{xj}^{\overline{q}}$ – разница между количеством поступившего и ушедшего товара j со склада x в промежуток времени q:

$$\Delta_{xj}^{\overline{q}} = \alpha_j * (p_j * |F_x| - \sum_{i \in G_x} d_{ij}^{\overline{q}}) \tag{1.4}$$

– где F_x заводы для которых склад x ближайший, G_x города для которых склад x ближайший, p_j – товара j которое производится на заводе за один интервал времени, α_j – стоимость хранения одной единицы товара j.

Выпишем суммарную стоимость хранения товаров для каждого склада $x \in \overline{X}$ до момента q.

$$S_x = \sum_{j=1}^{J} \sum_{\overline{q}=1}^{Q} \Delta_{xj}^{\overline{q}} * \eta(q + 1 - \overline{q}) \tag{1.5}$$

Отметим, поскольку со складов не может убывать больше чем поступает должны выполняться следующие ограничения:

$$\sum_{\overline{q}=1}^{q} \Delta_{xj}^{\overline{q}} * \eta(q + 1 - \overline{q}) \geqslant 0 \quad q = 1, \ldots, Q; j = 1, \ldots, J \tag{1.6}$$

Рассмотрим многокритериальную задачу в которой нужно минимизировать издержки на транспортировку и хранение

$$\min_{\overline{X},X_j}\{\{f_j(X_j)\}_{j=1,\dots,J};\{S_x(\overline{X})\}_{x\in\overline{X}} : \overline{X},X_j \in W\} \tag{1.7}$$

Где W – множество допустимых размещений $\overline{X}; X_j \quad j = 1.\dots,J$

2 Описание методов решения поставленной задачи

Будем основываться на принципе оптимальности введенном Вильфредо Парето[1] В нашем случае поиск множества всех Парето-оптимальных решений затруднен, в силу большой мощности множества допустимых разрешений: не трудно заметить, что оно растет экспоненциально с ростом количества вершин в рассматриваемой модели или с ростом количества размещаемых производств и пунктов хранения. К примеру для 10 городов и 4 пунктов производства существует 210 различных размещений, для 20 и 8 — 125970 размещений, а для 50 и 12 — более 120 миллиардов возможных размещений. Данная задача является NP-полной, поэтому зачастую в прикладных задачах ограничиваются поиском одного решения которое является Парето-оптимальными, игнорируя тот факт, что вообще может быть множество Парето-оптимальных решений. Также существуют генетические алгоритмы – эвристические алгоритмы поиска, используемые для решения задач оптимизации и моделирования путём случайного подбора, комбинирования и вариации искомых параметров с использованием механизмов, аналогичных естественному отбору в природе. Генетические алгоритмы является разновидностью эволюционных вычислений, с помощью которых решаются оптимизационные задачи с использованием методов естественной эволюции, таких как наследование, мутации, отбор и кроссинговер[2]. Отличительной особенностью генетического алгоритма является акцент на использование оператора «скрещивания», который производит

[1]Вильфредо Парето — итальянский инженер, экономист и социолог. Разработал теории, названные впоследствии его именем: статистическое Парето-распределение и Парето-оптимум, широко используемые в экономической теории и иных научных дисциплинах.

[2]Кроссинговер — процесс обмена участками гомологичных хромосом во время конъюгации в профазе I мейоза.

операцию рекомбинации решений-кандидатов, роль которой аналогична роли скрещивания в живой природе.

Поиск частного Парето-оптимального решения

В этом пункте рассматривать взвешенную сумму 2.1 критериев с неотрицательными весами — минимум суммы будет является Парето-оптимальным решением [3, 7].

$$\min_{X_j}\{\sum_{j=1,\dots,J} f_j(X_j) + \sum_{x\in\overline{X}} S_x^Q(\overline{X}) : \overline{X}, X_j \in W\} \tag{2.1}$$

Эту задачу можно свести к задаче целочисленного линейного программирования, или же решить с помощью метода ветвей и границ.

Задача целочисленного линейного программирования(ЦЛП)

Пусть $[\xi_{i_1 i_2}]$ — матрица распределения, в которой

$$\begin{cases} 1, & \text{если город } i_1 \text{ прикреплен к складу } i_2 \\ 0, & \text{в противном случае.} \end{cases}$$

Далее примем $\xi_{ii} = 1$, в вершине i размещен склад, иначе $\xi_{ii} = 0$.

Пусть $[\Phi_{i_1 i_2}]$ — матрица распределения, в которой

$$\begin{cases} 1, & \text{если завод в вершине } i_1 \text{ поставляет к складу } i_2 \\ 0, & \text{в противном случае.} \end{cases}$$

Минимизировать функцию

$$z = \sum_{i_1=1}^{I}\sum_{i_2=1}^{I}\sum_{q=1}^{Q}\sum_{j=1}^{J}[\xi_{i_1 i_2} * c_j(i_1,i_2) * d^q_{i_1 j} * \beta_j$$

$$\Phi_{i_1 i_2} * c(i_1,i_2) * \sum_{\overline{i_2}=1}^{I} \xi_{i_1 \overline{i_2}} d^q_{\overline{i_2} j} * \beta_j] \qquad (2.2)$$

$$= \beta_j * c(i_1,i_2) * [\xi_{i_1 i_2} * d^q_{i_1 j} + \Phi_{i_1 i_2} * \sum_{\overline{i_2}=1}^{I} \xi_{i_1 \overline{i_2}} d^q_{\overline{i_2} j}]$$

при ограничениях

$$\begin{cases} \sum_{i_1=1}^{n} \xi_{i_1 i_2} = 1 \quad \text{для } i_2 = 1,\ldots,n, \\ \sum_{i=1}^{n} \xi_{ii} = p, \\ \sum_{i=1}^{n} \Phi_{ii} = R, \\ \xi_{i_1 i_2} \leqslant \xi_{i_1 i_1} \quad \text{для всех } i_1,i_2 = 1,\ldots,I, \\ \Phi_{i_1 i_2} \leqslant \Phi_{i_1 i_1} \quad \text{для всех } i_1,i_2 = 1,\ldots,I, \\ \xi_{i_1 i_2} \in \{0,1\}, \\ \Phi_{i_1 i_2} \in \{0,1\}, \end{cases} \qquad (2.3)$$

Таким образом, для исходной задачи получена задача, сформулированная в терминах целочисленного программирования. Задачу целочисленного линейного программирования можно решить методом ветвей и границ [8], в силу того, что задача является NP-трудной имеет место применения эвристических алгоритмов. Например, может быть использован поиск с запретами [9]

Метод ветвей и границ для поиска оптимального решения

Построим матрицу T, с элементами q_{kj}. Матрица T размера $I \times I$ описывает размещение складов. Ее строки соответствуют вершинам, в которых гипотетически могут склады, а l-ый столбец содержит вершины

графа t_{1l}, $t_{2l}, \ldots$, расположенные в неубывающем порядке по стоимости транспортировки всех необходимых товаров в вершину если бы в вершине l находился склад.

Замечания для перебора

а) Важно отметить, что на каждом шаге должны выполняться условия 1.6.

б) Если на некотором этапе проводимого поиска будет построено множество из p складов, то каждую оставшуюся не распределенную вершину можно прикрепить к ближайшему складу.

в) Для каждого построенного размещения нужно перебрать возможные размещения заводов

Вычисление нижней границы

Проведем нижнюю оценку для первой суммы и второй суммы в 2.1 по отдельности нижняя граница будет их суммой.

Вычислим нижнюю границу для слагаемых ответственных за стоимость хранения очевидно, что

$$S_x^q <= \sum_{j=1}^{J} \sum_{\overline{q}=1}^{q} [\alpha_j * (p_j * r_j - \sum_{i \in G_x} d_{ij}^{\overline{q}})] * \eta(q + 1 - \overline{q})$$

Теперь вычислим нижнюю границу для слагаемых относящихся к издержкам на транспортировку. Пусть на определенном шаге метода вычислены прикрепления для матрицы T. Пусть в матрице T выполненные прикрепления дали p' складов и i городов прикреплено к складам. Значит осталось прикрепить еще $h = p - p'$ складов. Пусть L — множество индексов еще нераспределенных вершин.

Пусть $m_{a_l l}$ и $m_{b_l l}$ — первые два неотмеченных элемента в столбце l матрицы T. Пусть число различных $m_{a_l l}$ для $l \in L$ равно h'. Наилучшим прикреплением вершины l будет $m_{a_l l}$. Если $h = h'$, то, прикрепив вершины

l к m_{a_l}, получим p складов, а значит, матрица T заполнена. Если же $h > h'$, для получения нижней границы нужно заменить $h - h'$ минимальных стоимостей на вторые по величине. Для поиска дополнительной стоимости нужно найти на $h - h'$ наименьших значений

$$\sum_{q=1}^{Q}\sum_{j=1}^{J} d_{lj}^{q} * \beta_j * [(c_j(l, m_{b_l l}) + c_j(m_{b_l l},\phi(m_{b_l l},X))) \\ -(c_j(l, m_{a_l l}) + c_j(m_{a_l l},\phi(m_{a_l l},X)))] \tag{2.4}$$

по всем вершинам из L. Нижняя граница будет получаться из суммирования уже выполненных распределений

$$\sum_{l\in L}\sum_{q=1}^{Q}\sum_{j=1}^{J} d_{lj}^{q} * \beta_j * (c_j(l, m_{b_l l}) + c_j(m_{b_l l},\phi(m_{b_l l},X)))$$

и $h - h'$ наименьших значений 2.5. Если же $h < h'$, то наилучшее пополнение частного решения даст меньше складов, чем p, видно $f_i(w)$ монотонно убывает с увеличением p, значит текущее частное решения наверняка не является частью оптимального [6].

Поиск оптимальных решений с помощью генетических алгоритмов

Метод ветвей и границ предлагает довольно наивный подход поиску оптимальных размещений — это объединение всех критериев в одну функцию приспособленности. Эта дает возможность найти частное Парето-оптимальное решение за приемлемое время. Для поиска множества оптимальных размещений предлагается использовать методы решения многокритериальных задач.

Алгоритм ветвей и границ пытается объединять критерии в отдельное значение приспособленности путем определения линейных соотношений между ними. Однако никак не использует понятия доминирования по Парето, чтобы более точно оценивать «хорошие» решения в многокрите-

риальном смысле. Для поиска решений многокритериальных задач с помощью генетических алгоритмов сущесвует несколько популярных алгоритмов: NSGA[1] [10] и SPEA[2] [11]. Они основываются на двух различных подходах к селекции потенциальных решений: в первом в качестве критериев для отбора лучшей популяции используются подход к выбору недоминируемых решений в популяции, во втором же используется Парето-сила решения(количество решений, которые доминируются по Парето данным решением).

Рассмотрим алгоритм основывающийся на предположении, что турнирная селекция, основанная на Парето-доминировании позволяет выбирать на каждой итерации все лучшее и лучшее множество допустимых размещений. Таким образом нам нужен оператор турнирной селекции.

Размещение A доминирует по Парето размещение B, если A не хуже B по всем критериям, а хотя бы по одному критерию лучше. В тех случаях когда оба размещения не доминируют по Парето друг друга, они являются одинаково хорошими, нужно оставить только одно размещение для оптимизации. Пусть размещение A многократно доминируемо другими размещениями в популяции, а – ни разу, тогда будет логичным оставить размещение A. хоть и B не доминирует по Парето A, но при этом в этой популяции является менее предпочтительнее чем B.

Будем пользоваться Рангом границы Парето. Напомним размещения находящиеся на границе имеют ранг 1. Если мы уберем эти размещения из популяции, а после посчитаем новую границу, то размещения то этой границе будут иметь ранг 2 и т.д.

[1]Dominated Sorting Genetic Algorithm

[2]Strength Pareto Evolutionary Algorithm

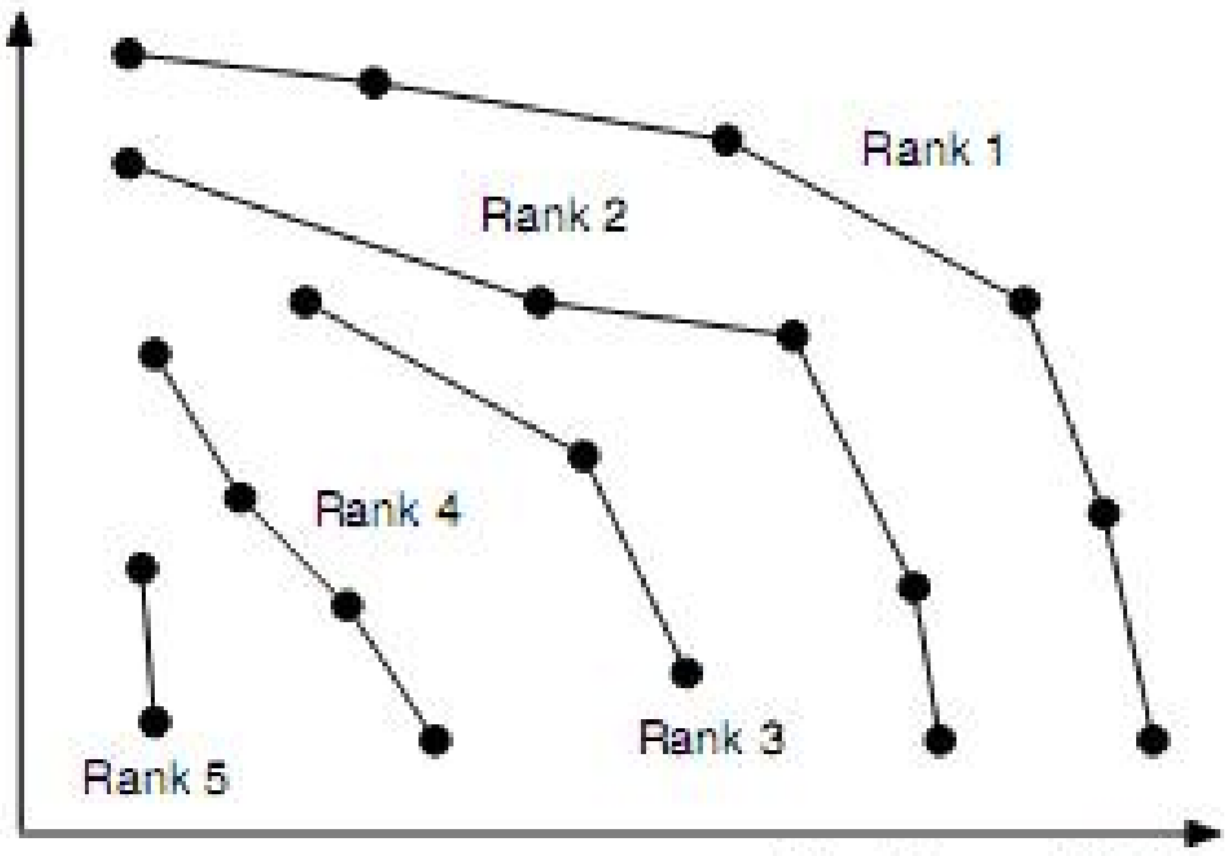

Рисунок 2.1 — Ранги

Рассмотрим, как вычисляется граница Парето. Суть заключена в том, что добавляем размещение к границе, если это размещение не доминируемо другими размещениями, уже принадлежащим на границе, и удаляем с границы тех размещений, которые доминируемыми этой новым размещением.

Вычислить ранги легко: находим первую границу, затем убираем размещения, снова находим границу и т.д. После того, как этой процедурой будут обработаны все размещения, можем использовать Ранг границы Парето для размещения в качестве ее приспособленности. Так как чем ниже ранг, тем лучше, его преобразование в приспособленность можно сделать следующим образом:

$$Fitness(i) = \frac{1}{1 + ParetoFronRank(i)} \tag{2.5}$$

Помимо прочего неплохо было бы, чтобы размещения были равномерно распределены вдоль границы. Для этого введем некоторую метрику для измерения расстояния между размещениями с одинаковым Рангом границы Парето, будем называть ее Разреженностью. Дадим определение разреженности для размещения: размещение находится в разреженной области, если ближайшее к ней размещения одного с ней ранга располагается как можно дальше.

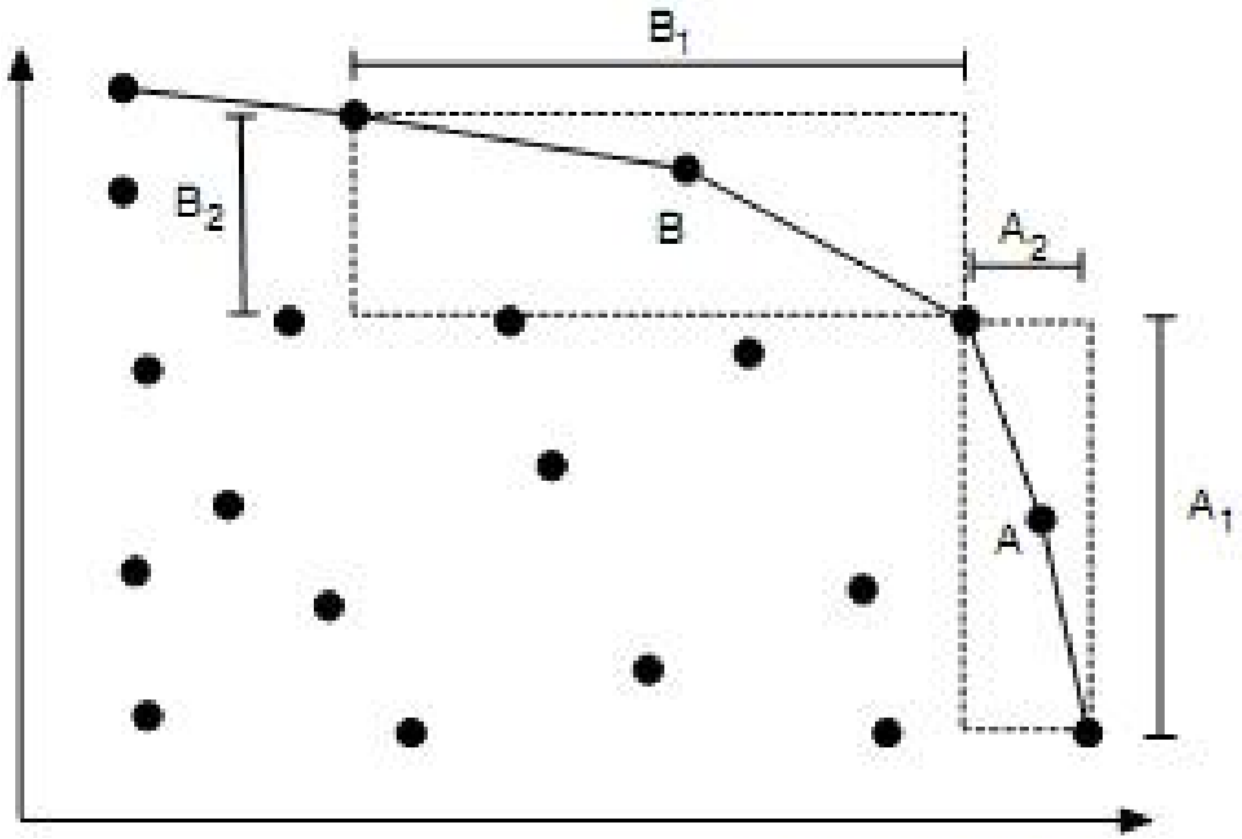

Рисунок 2.2 — Разреженность

Будем вычислять разреженность с использованием Манхеттеновского расстояния [?] по всем критериям для правого и левого соседа каждого размещения для рассматриваемого критерия. Размещения на концах Ранга границы Парето считаются находящимися в бесконечно разреженной области. Будем использовать разреженность, чтобы дополнительно отсекать решения.

Окончательно определим турнирную селекцию таким образом, чтобы размещения прежде всего отбирались по Рангу границы Парето, а неоднозначные ситуации будем разрешать с использованием разреженности. Идея состоит в том, чтобы выбирать размещения, которые не только ближе остальных к настоящей границе Парето, но и равномерно распределены по ней.

Будем запоминать все хорошие размещения найденные в процессе поиска. Общая идея заключается в использовании архива для лучший найденных размещений. Произведя новую популяцию, устраивается отбор, чтобы решить, какие размещения останутся в популяции.

Численный пример

В рамках магистерской диссертации была осуществлена программная реализация описанных выше алгоритмов решающих поставленную задачу. Были реализованы методы поиска поиска решений как с помощью точных алгоритмов: методом ветвей и границ, так и с помощью эвристических алгоритмов.

Для реализации были использованы языки программирования C++, PYTHON. Как язык программирования высокого уровня, C++ поддерживает необходимый уровень абстракции для удобной платформонезависимой разработки алгоритмов рассматриваемых методов, при этом предоставляя большое количество стандартных структур данных, дополнительных подключаемых библиотек алгебры матриц и линейного программирования, а также обеспечивая высокое время исполнения скомпилированной программы. PYTHON язык с множеством библиотек, а также удобный для вспомогательных вычислений и постройки иллюстраций.

2.1 Программная реализация

Этот параграф посвящен описанию структуры программного кода, реализованных алгоритмов. С частичным листингом кода можно ознакомиться в Приложении. Были реализованы алгоритмы описанные в предыдущей главе, проведен сравнительный анализ скорости работы, качества полученных результатов. В основе реализации лежит класс `OptimalAlloc`, предназначенный для создания графа, описывающего систему городов и дорог, а также методов поиска оптимального размещения пунктов производств. Непосредственно метод ветвей и границ реализован public-методом `solveByBranchAndBoundMethod()`, при этом метод задействует ряд вспомогательных private-методов. Для хранения промежуточных структур(матриц, списков, множеств) используется стандартная библиотека шаб-

лонов в языке программирования C++(Standard Template Library, STL) — это удобные шаблоны, которые поддерживают динамическое выделение памяти, что позволяет более просто и эффективно реализовать данный алгоритм.

Метод `solveByBranchAndBoundMethod()` реализует алгоритм, приведенный в параграфе 2. Для отыскания матрицы расстояний между произвольными двумя вершинами реализован алгоритм Флойда в качестве private-метода `floyd()`. Также были реализованы private-методы `makeQ()` и `AllocQ()` — создание матриц T их перебор. Для поиска нижней границы был реализован еще один private-метод `countLB()`.

Для эвристического алгоритма был использован язык программирования PYTHON, также использован класс `LocationProblem`, предназначенный все также для создания графа, описывающего систему городов и дорог. Реализованы функции подсчета критериев, а также реализована функция выполняющая генетический алгоритм с селекцией основанной на недоминируемой сортировке с использованием лучших результатов.

2.2 Анализ эффективности работы алгоритмов

Благодаря программной реализации мы можем провести более детальный анализ методов оптимизации, применив их к конкретным задачам. Для этого сгенерируем наборы задач размещения производств в сети. В качестве параметров будут выступать такие величины как количество городов P, количество видов продукции N, количество пунктов хранения, а также количество пунктов производства M.

В нашей задаче трудно оценить качество полученных решений для задач с большой размерностью. Алгоритм работающий по утилитарному принципу оптимальности работает за приемлемое время для графов с размерностью 15-25 вершин при нескольких различных товарах.

Для того чтоб оценить точность эвристического алгоритма были рассмотрены задачи с небольшим количеством параметров, 15 вершин, от 2-3 различных вида товаров и 2-3 пункта хранения. При таких параметрах множество возможных размещений производств и пунктов хранения достаточно мало чтоб алгоритм основанный на методе ветвей и границ работал за приемлемое время.

Была произведена генерация 40 различных связных графов. Каждому графу соответствовали параметры описывающие систему: стоимости хранения, транспортировки и количество потребления товаров. На всех графах были запущены три алгоритма: полный перебор, метод основанный на обобщении критериев и эвристический алгоритм с размером начальной популяции в 30 размещений. По завершении 400 итераций эвристического алгоритма из популяции выбиралось множество недоминируемых размещений.

В качестве метрики оценки качества генетического алгоритма были использованы следующие метрики:

— R_1 — среднее среди кратчайших расстояний от всех элементов недоминируемых размещений, полученных эвристическим алгоритмом, до множества Парето-оптимальных размещений полученных полным перебором.

— R_2 — среднее расстояние от размещения полученного с помощью утилитарного принципа(метода ветвей и границ) до всех недоминируемых решений генетичекого алгоритма.

— W — среднее расстояние между всеми Парето-оптимальными решениями.

Оказалось, что отношение этих двух расстояний в среднем

$$D_1 = R_1/W \ = 2.1$$

$$D_2 = R_2/W \ = 3.4$$

. Также были выбраны множества случайные недоминируемые размещения. Были посчитаны те же метрики будто это есть решения задачи(обозначим их соответственно $R3$,$R4$,D_3,D_4), получилось:

$$D_3 = R_3/W \ = 6.1$$

$$D_4 = R_4/W \ = 7.5$$

Среднем получается неплохой результат приближенного решения. Образно можно сказать, что решения полученные генетическим алгоритмом в среднем удалены от множества Парето-оптимальных решений в примерно два раза дальше, чем среднее расстояние между всеми Парето-оптимальными решениями, и примерно в три раза ближе, чем если выбирать размещения случайным образом.

Результаты вычислений для примера

Пусть $P = 25$, $N = 3$, $G(X,Y)$ — граф, моделирующий систему городов, изображенную на рисунке 2.1. Пусть есть два вида продукции A и B каждый из которых должен производиться на двух и трех заводах соответственно и 4 пункта хранения.

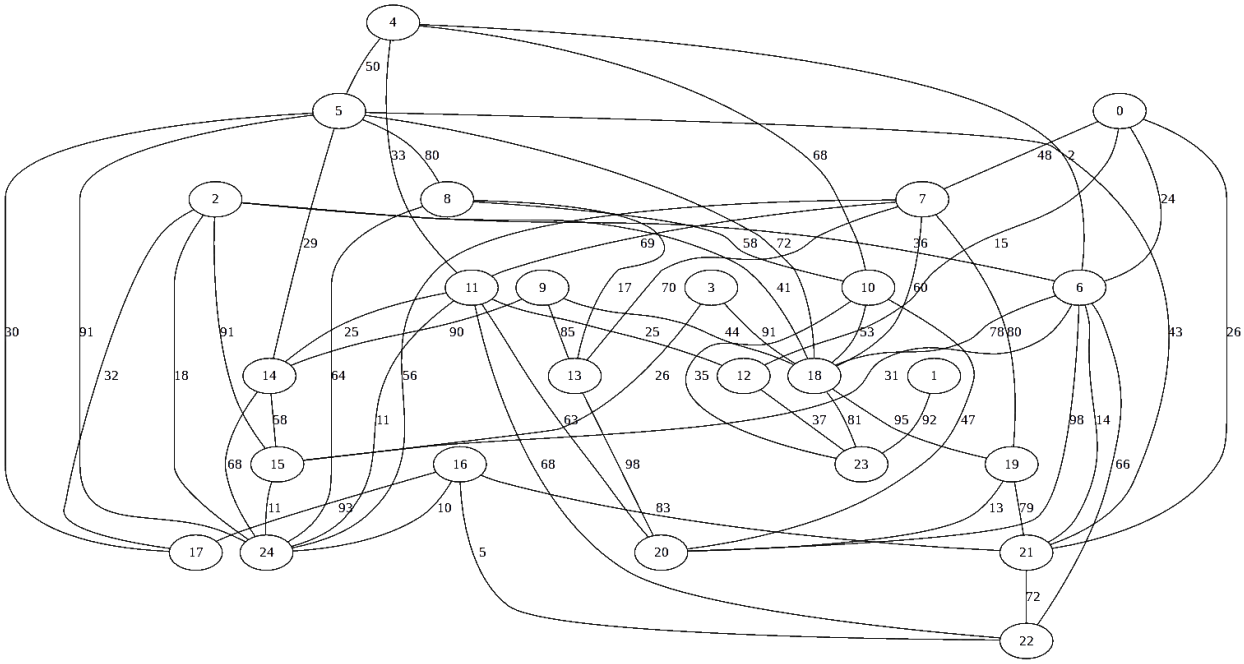

Рисунок 2.1 — $G(X,Y)$

На рисунках 2.2 и 2.3 изображены — размещения, полученные соответственно алгоритмом, основанным на методе ветвей и границ, а также же одно из размещений полученных алгоритмом(самое ближайшее к размещению полученному алгоритмом ветвей и границ).

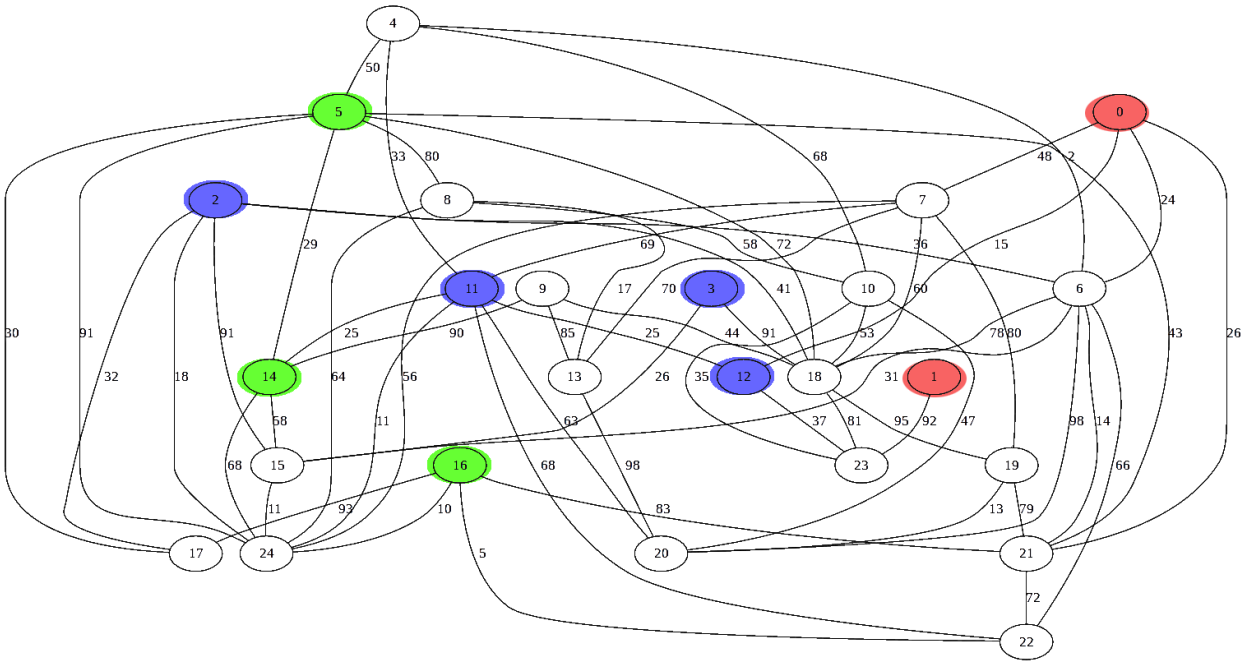

Рисунок 2.2

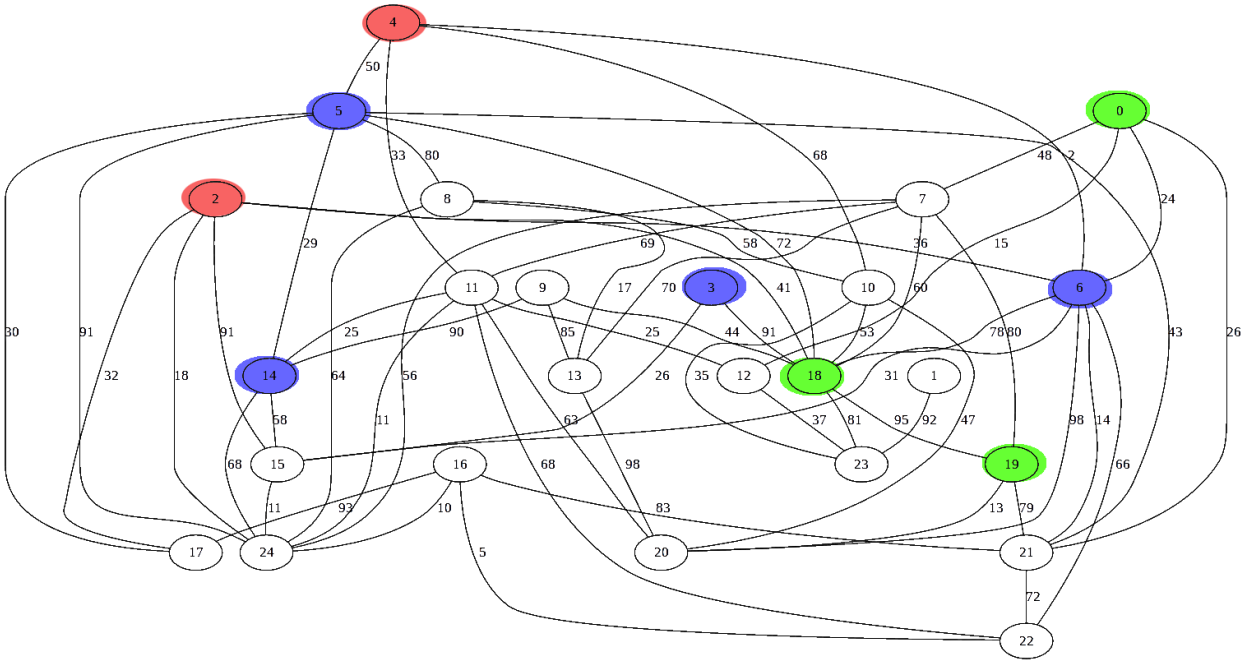

Рисунок 2.3

Оценка зависимости времени исполнения реализованных алгоритмов от параметров.

Для тестов программы было написано скриптов на языке Python3: программа для генерации векторов потребления продукции городами, а также взвешенных связных графов с заданным количеством вершин.

На рисунке 2.4 изображена диаграмма. На данной диаграмме по вертикальной оси — время исполнения в секундах, по горизонтальной — количество вершин в графе, ломаные — запуск для разного количества типов производимой продукции. На рисунке 2.4 показаны результаты запуска алгоритма поиска оптимального размещения, в качестве принципа берётся утилитарный принцип оптимальности. Алгоритм основан на методе ветвей и границ.

Данный алгоритм имеет высокую скорость исполнения для количества производимых различных товаров $N = \{1,3\}$, при количестве вершин в графе $P < 65$, однако, при $N = 5$ удалось вычислить только для $P = 55$

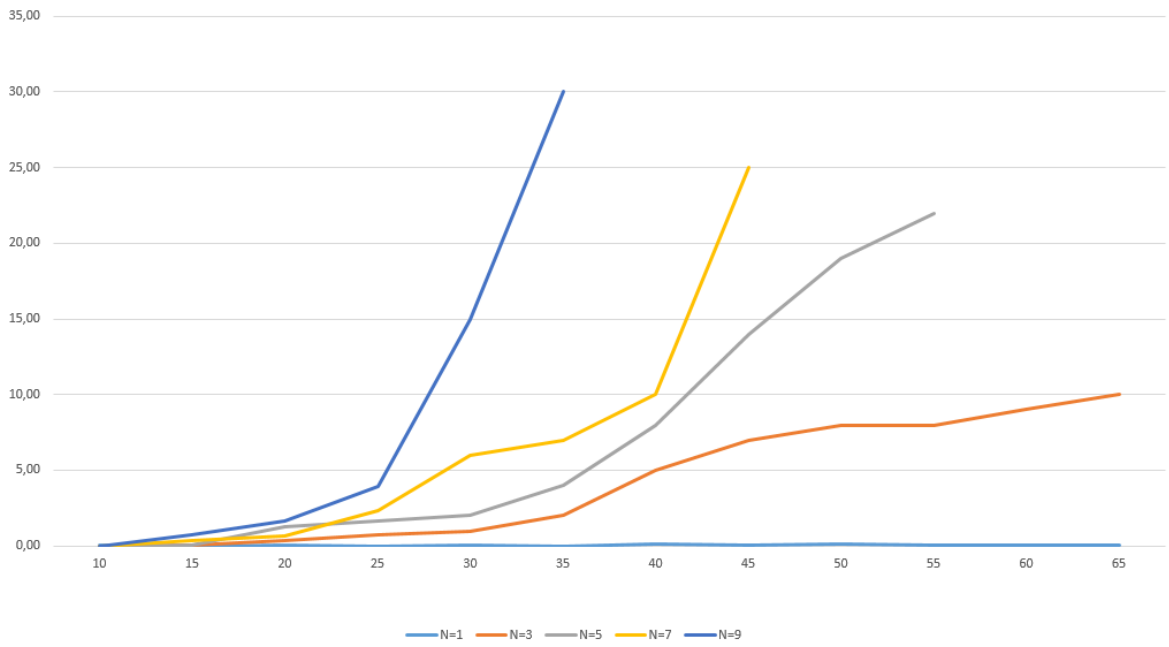

Рисунок 2.4

вершин, при $N = 7$ — $P = 45$ вершин, а при $N = 9$ — только $P = 35$ вершин.

Итак алгоритм ветвей и границ работает достаточно быстро для $P \leqslant 25$ вершин в графе. Далее время алгоритма довольно быстро возрастает. Анализ скорости работы генетического алгоритма не производился, так как он довольно быстро работает для данных размерностей графа.

Заключение

В работе рассмотрена задача многокритериальной оптимизации размещения пунктов производства и складирования. Данная задача рассматривалась в литературе [1], так например в [3] автор рассматривает задачу размещения производств одного вида продукции в сети. Применяя простейшие принципы оптимальности: эгалитарный и утилитарный, автор сводит задачи размещения к задачам о поиске медиан и центров в графе. В той работе, обобщается данная задача: необходимо разместить пункты производства, производящих несколько видов продукции, а также пункты складирования. Были выбраны методы для решения. Первый метод основан на объединении критериев в один и использовании метода ветвей и границ для поиска наименьшего значения полученного обобщенного критерия. Второй базируется на подходе к решению многокритериальных задач с помощью эвристических алгоритмов. Также разработана программная реализация алгоритмов, решающих задачу.

Среднем для генетического алгоритма получается неплохой результат приближенного решения. Можно сказать, что решения полученные генетическим алгоритмом в среднем удалены от множества Парето-оптимальных решений в примерно два раза дальше, чем среднее расстояние между всеми Парето-оптимальными решениями, и примерно в три раза ближе, чем если выбирать размещения случайным образом.

Исследование методов не только показало их применимость к решению задачи, но также и наглядно продемонстрировало различия подходов, определяющих эти методы. Подходы к поиску оптимального размещения разнообразны, и даже в рамках различных методов оптимальности методы решения могут быть различны. Поэтому исследование задачи оптимального размещения может быть продолжено. Можно комбинировать методы

использовать приближенные решения полученные на алгоритмом ветвей и границ(прервав его исполнение через некоторое время).

Список использованных источников

1. Farahani R. Z, SteadieSeifib M., Asgari N. Multiple criteria facility location problems: A survey // Applied Mathematical Modelling, Vol. 34, Iss. 7. 2010. P. 1689–1709.

2. *В.Д. Ногин* Принятие решений в многокритериальной среде. Москва. 2005 год.

3. Ogryczak W. Location Problems from the Multiple Criteria Perspective: Effcient Solutions. // Archives of Control Sciences, 7 (XLIII). 1998. P. 161–180.

4. Ринчинов Р.Ц., Парфенов А.П. Многокритериальная задача оптимального размещения производства в сети // ЖУРНАЛ: ПРОЦЕССЫ УПРАВЛЕНИЯ И УСТОЙЧИВОСТЬ; Издательство: Смирнов Николай Васильевич; ISSN: 2313-7304

5. *Б.А.Аникина* Логистика: Учебник М.: ИНФРА-М, 2008. — 368 с. — (Высшее образование)

6. *Н. Кристофидес* Теория графов. Алгоритмический подход. Москва. 1978 год.

7. Ногин В. Д Линейная свертка критериев в многокритериальной оптимизации; журнал: искусственный интеллект и принятие решений; Издательство: Федеральный исследовательский центр "Информатика и управление"Российской академии наук (Москва); ISSN: 2071-8594

8. *А. Схрейвер* Теория линейного и целочисленного программирования. Москва. 1991 год.

9. *F. Glover.* Tabu search-Part II // ORSA Journal on computing. — 1989. — Т. 1, вып. 3. — С. 4–32. — DOI:10.1287/ijoc.2.1.4.

10. *K. Deb ; A. Pratap ; S. Agarwal ; T. Meyarivan* A fast and elitist multiobjective genetic algorithm: NSGA-II; IEEE Transactions on Evolutionary Computation (Volume: 6 , Issue: 2 , Apr 2002)

11. *Eckart Zitzler , Marco Laumanns , Lothar Thiele* SPEA2: Improving the Strength Pareto Evolutionary Algorithm (2001);

12. *Revelle C. S.,Swain R. W.* Central facilites location, Geographical Analysis, 2, стр.30 1970 год

13. Brian Wesley Baugh Generate a randomly connected graph with N nodes and E edges.: [Электронный ресурс]: Github. URL: https://gist.github.com/bwbaugh/4602818 (дата обращения: 9.05.2017).

Программный код реализации алгоритма ветвей и границ

```
    class OptimalAlloc{
public:
  OptimalAlloc(const char* file);
  OptimalAlloc(const char* graph,const char* prod);

  vector<set <int> > BranchAndBound();
  vector<set <int> > Center();

private:
  vector<vector<float> > X;
  vector<int> M;
  int P;
  vector<vector<int> > t;
  vector<vector<int> > c;
  void print(vector<vector<int> > vec);
  void print(list<list<int> > l);
  vector<vector<int> >  read(const char *filename, int
    *N, vector<vector<float> > *X,vector<int> *M);
  vector<vector<int> >  floyd(int N, vector<vector<int>
     > vec);
  list<list<int> >  makeQ(int N, vector<vector<int> >
    vec);
  float count_ng(list<list<int> > Q,int p,set<int>A,
    vector<float> x,vector<vector<int> > c);
  set<int> pereborQ(list<list<int> > Q,int p, vector<
    float> x,vector<vector<int> > c, set<int>A_old);
```

```
    set <int>  centerAlg(int p,set<int> A_old,list<float>
        L);
    void read(const char *prod, vector<vector<float> > *X
       ,vector<int> *M);
    vector<vector<int> >  read(const char *graph,int *N);
};


list<list<int> >  OptimalAlloc::makeQ(int N, vector<
   vector<int> > vec){
  list<list<int> > result;

  for ( int i=0;i<N;i++ ) {
    list<int> l;

    l.push_back(i);

    for ( int j=0;j<N;j++){
      if (i!=j){
        bool flag = true;
        for (list<int>::iterator it = l.begin(); it != l.
           end(); it++){
          if (vec[*it][i]>=vec[j][i]){
            flag=false;
            l.insert(it,j);
            break;
          }
        }
```

```
            if (flag)
              l.push_back(j);
          }

      }
      l.push_front(1);
      result.push_back(l);
    }
    return result;
}


float OptimalAlloc::count_ng(list<list<int> > Q,int p,set
    <int>A, vector<float> x,vector<vector<int> > c){
    float result=0;

    int h=0; // количество не распределенных вершин
    list<vector<int> > m;
    int co=0;
    for (list<list<int> >::iterator lis = Q.begin(); lis !=
        Q.end(); lis++){

      if (*((*lis).begin())!=0){
        vector<int> v;
        v.push_back(co);
        for (list<int>::iterator it = (*lis).begin();it !=
            (*lis).end(); it++){
          if ( it == (*lis).begin())
```

```
            it++;
          if (A.find(*it)==A.end()){
            v.push_back(*it);
            if (v.size()==3)
              break;
          }
        }
        h++;
        m.push_back(v);
      }
      else{
        for (list<int>::iterator it = (*lis).begin(); it !=
            (*lis).end(); it++){
          if ( it == (*lis).begin())
            it++;
          if (*it >99) {
            result += x[co]*c[co][*it-100];
            break;
          }
        }
      }
      co++;

  }
```

```
if (p-A.size()<=h){//прикрепляем к самым верхним элемен
    там не прикрепленных столбцов
   for (list<vector<int> >::iterator it = m.begin(); it
      != m.end(); it++)
      result += x[(*it)[0]]*c[(*it)[0]][(*it)[1]];

   if (p-A.size()<h){//найти h-p+A.size() наименьших раз
      ностей 6.20
      list<float> mmm;
      for (list<vector<int> >::iterator it = m.begin();
         it != m.end(); it++)
        mmm.push_back(x[(*it)[0]]*(c[(*it)[0]][(*it)[2]]-
           c[(*it)[0]][(*it)[1]]));
     mmm.sort();
     mmm.reverse();
     co=0;
     for (list<float>::iterator it = mmm.begin();it !=
        mmm.end() and co!=h-p+A.size();it++){
        co++;
        result+=*it;
     }

   }
}
else
   result=1E101;

return result;
```

```
}

set<int> OptimalAlloc::pereborQ(list<list<int> > Q,int p,
    vector<float> x,vector<vector<int> > c, set<int>A_old
  ){//перебирает все вершины которые можно добавить
  list<list<int> >::iterator list_;
  list<int>::iterator it_;
  float ng_=1e100;
  int p_=0;
  set<int>A;
  while(p_<p){
    bool flag = false;

      for (list<list<int> >::iterator lis = Q.begin();
        lis != Q.end(); lis++){

      if (*((*lis).begin())!=0)

        for (list<int>::iterator it = (*lis).begin(); it
          != (*lis).end(); it++){
          if ( it == (*lis).begin())
            it++;
          if (A.find(*it)==A.end() and A_old.find(*it)==
            A_old.end() ){
            A.insert(*it);
            *it=*it+100;
            *((*lis).begin())=0;
```

```
                float ng=count_ng(Q,p,A,x,c);
                *it=*it-100;
                *((*lis).begin())=1;
                A.erase(*it);
                if (ng<ng_){
                  flag=true;
                  ng_=ng;
                  list_=lis;
                  it_=it;
                }
              }

            }
          if (flag){
            *((*list_).begin())=0;
            *it_=*it_+100;
            p_++;
            A.insert(*it_-100);
            flag= false;
          }
        }

    }
    return A;

}
```

```
vector<set <int> > OptimalAlloc::BrachAndBound(){
  vector<set <int> > result;
  list<list<int> > Q= makeQ(P,c);
  set<int> A_old;
  for (int i=0; i<X.size() ; i++){
    set< int> A_ = pereborQ(Q,M[i],X[i],c,A_old);
    A_old.insert(A_.begin(),A_.end());
    result.push_back(A_);
  }
  return result;
}

class Covering
  {
  public:
    std::vector<bitset <100> > bitset_;

    list<pair<bitset<100>, set<int> > > list_;
    int p;
    Covering(std::vector<vector<int> > c,int P,float l){
      p=P;
      for (int i=0;i<c.size();i++){
        pair<bitset<100>, set<int> > temp;
        for (int j=0;j<c.size();j++){
          if (c[i][j] <=l){
            temp.first[j]=1;
          }
          if (j!=i)
            temp.second.insert(j);
```

```
        }

        list_.push_back(temp);
        bitset_.push_back(temp.first);
        }


}
set <int> find_covering(int step,set<int> A_old){
  set <int> result;
  list<pair<bitset<100>, set<int>>> temp;
  for ( pair<bitset<100>  , set<int>>  l : list_){
    for (int k : l.second){

      if (A_old.find(k)==A_old.end()){
        if (p==1 and bitset_[k].count()==bitset_.size
          ()){
          set<int> t;
          t.insert(k);
          return t;
        }
        bitset<100> t = l.first|bitset_[k];
        if (t.count()==bitset_.size() and p!=1){
          return makeset(l.second,k,bitset_.size());
        }
        pair<bitset<100>, set<int>> r(l);
        r.second.erase(k);
        r.first=t;
```

```
            temp.push_back(r);
          }
        }
      }

      if (step+1 == p or p==1)
        return result;
      else{
        list_ = temp;
        return find_covering(step+1,A_old);
      }
  }
private:
  set <int> makeset(set<int> t, int p,int P){
    set <int> result;
    for (int i=0;i<P;i++)
      if (t.find(i)==t.end())
        result.insert(i);
    result.insert(p);
    return result;


  }

};

vector<set <int> >  OptimalAlloc::Center(){
  vector<set <int> > result;
```

```
  set<int> A_old;
  list<float> L;
  for (int i=0; i<M.size() ; i++){
    for (int j=0; j<M.size() ; j++){
      if (c[i][j]!=0)
        L.push_back(c[i][j]);
  }
  L.sort();
  }
  for (int i=0; i<M.size() ; i++){
    set< int> A_ = centerAlg(M[i],A_old,L);
    A_old.insert(A_.begin(),A_.end());
    result.push_back(A_);
  }
  return result;
}


set <int> OptimalAlloc::centerAlg(int p,set<int> A_old,
   list<float> L){
  set <int> result;
  list<float> ::iterator t =L.begin();
  do{
    if (t==L.end()){
      result.clear();
      break;
    }
```

```
        Covering cov(c,p,*t);
        result = cov.find_covering(1,A_old);
        t++;

    }while(result.size()<p);
    return result;
}
```

Генетический алгоритм

```
import networkx as nx
import numpy as np
from itertools import combinations
from networkx import drawing, readwrite
import matplotlib.pyplot as plt


class LocationProblem(Problem):
    def __init__(self, G, J, R, Q, p):
        super(LocationProblem, self).__init__(1 + J, J +
            R)
        self.G, self.J, self.R, self.Q, self.p = G, J, R,
            Q, p
        self.distances = list(nx.
            all_pairs_dijkstra_path_length(G))
        self.types[:] = [Subset(G.nodes, p), *(Subset(G.
            nodes, 3) for _ in range(self.J))]
        self.I = G.number_of_nodes()
        self.D = np.ones((self.I, self.J, self.Q), dtype=
            np.int) * 2
        self.L = np.sum(self.D, axis=2)
        self.B = np.ones((self.J,), dtype=np.int)
        self.ap = np.ones(self.J)

    def get_all_solutions(self):
        """Returns the non-dominated solutions."""
        archive = Archive()
```

```
        i = 0
        for s in self._get([tuple(combinations(G.nodes,
            self.p)),
                            *(tuple(combinations(G.nodes,
                                3)) for _ in range(self.J
                               ))]):
            i+=1
            if i % 10000 == 0:
                print("total steps: {} len: {}".format(i,
                    len(archive._contents)))
            solution = Solution(self)
            solution.variables = s
            self.evaluate(solution)
            archive.add(solution)
        return archive._contents

    def _get(self, sol):
        if len(sol) > 1:
            for n in self._get(sol[1:]):
                for s in sol[0]:
                    yield (s, *n)
        else:
            for s in sol[0]:
                yield s,

    def evaluate(self, solution):
        solution.objectives[:] = [*self.obj1(solution), *
            self.obj2(solution)]
```

```
        # print(solution.objectives)

    def c(self, i, j):
        return self.distances[i][1][j]

    def fi(self, i, subset):
        d = {_: self.c(i, _) for _ in subset}
        x = min(d, key=d.get)
        return x

    def _c1(self, i, X_hat):
        return self.c(i, self.fi(i, X_hat))

    def _c2(self, i, X_hat, X_j):
        return self.c(self.fi(i, X_hat),
                      self.fi(self.fi(i, X_hat), X_j))

    def _C(self, X_hat, X):
        """X_hat — sklads; X dict with subsets of
           manafacturing"""
        C1, C2 = np.zeros(self.I), np.zeros((self.I, self
           .J))
        for i in G.nodes:
            C1[i] = self._c1(i, X_hat)
            for j in range(self.J):
                C2[i][j] = self._c2(i, X_hat, X[j])
        return C1, C2
```

```
def _obj1(self, C1, C2, j):
    return np.dot(self.L[:, j], (C1 + C2[:, j]))

def obj1(self, solution):
    X_hat = solution.variables[0]
    X = solution.variables[1:]
    C1, C2 = self._C(X_hat, X)
    return [self._obj1(C1, C2, j) for j in range(self
        .J)]

def _G(self, solution):
    X_hat = solution.variables[0]
    G = np.zeros((self.I, self.I, self.J, self.Q))  #
        j, q
    for i in range(self.I):
        G[i][self.fi(i, X_hat)] = self.D[i, :, :]
    return G

def _F(self, solution):
    X_hat = solution.variables[0]
    X = solution.variables[1:]
    F = np.zeros((self.J, self.I, self.I))  #
    for j in range(self.J):
        for i in X[j]:
            F[j][self.fi(i, X_hat)][i] = 1
    return F

def obj2(self, solution):
```

```
        F = self._F(solution)
        G = self._G(solution)
        X_hat = solution.variables[0]
        F_ = F.sum(axis=(2))
        res = list()
        for x in X_hat:
            a = self.Q * np.dot(self.ap, F_[:, x])
            b = np.sum(G[:, x, :, :])
            res.append(a + b)
        return res


def draw(g, pos, title, *nodes_groups):
    print({k: k for k in g.nodes()})
    nx.draw_networkx_labels(g, {k: (v[0]+10, v[1]+10) for
        k,v in pos.items()}, {k: k for k in g.nodes()})
    if not nodes_groups:
        nodes_groups = g.nodes(),
    colors = ['red', 'blue', 'green', 'yellow', 'pink', '
      brown', 'black', 'orange', 'gray']
    _nodes = list()
    for index, nodes in enumerate(nodes_groups):
        nx.draw_networkx_nodes(g, pos,
                               nodelist=nodes,
                               node_color=colors[index],
                               node_size=100,
                               alpha=0.8)
        _nodes.extend(nodes)
```

```
        nx.draw_networkx_nodes(g, pos,
                               nodelist=[n for n in g.nodes()
                                   if n not in _nodes],
                               node_color='gray',
                               node_size=100,
                               alpha=0.8)

        nx.draw_networkx_edges(g, pos,
                               edgelist=g.edges(),
                               width=1, alpha=0.5, edge_color
                                  ='black')

        plt.title(title)
        plt.axis('off')
        plt.show()

DATA_PATH = "calc/data/{}/{}"


if __name__ == '__main__':
    import argparse
    import pickle
    parser = argparse.ArgumentParser()
    parser.add_argument('--type')
    parser.add_argument('-R', type=int)
    parser.add_argument('-J', type=int)
    parser.add_argument('-Q', type=int)
```

```
parser.add_argument('-p', type=int)
parser.add_argument('--Gn', type=int)
parser.add_argument('--Gp', default=0.5, type=float)
parser.add_argument('--Gseed', default=124, type=int)
parser.add_argument('--out')
parser.add_argument('--inp')
args = parser.parse_args()
if args.type == "generate-graph":
    if None in (args.Gn, args.Gp, args.Gseed):
        raise ValueError("need args")
    with open(DATA_PATH.format("graphs","Gn{}.Gp{}.
      Gseed.{}.pickle".format(args.Gn, args.Gp, args
      .Gseed)), 'wb') as f:
        G = nx.erdos_renyi_graph(args.Gn, args.Gp,
          seed=args.Gseed, directed=False)
        pos = nx.drawing.nx_agraph.graphviz_layout(G)
        pickle.dump([G, pos], f)
elif args.type.startswith("algorithm"):
    if None in (args.J, args.R, args.Q, args.p, args.
      inp):
        raise ValueError("need args")
    with open(DATA_PATH.format("graphs", args.inp), "
      rb") as f:
        G, pos = pickle.load(f)
        problem = LocationProblem(G, args.J, args.R,
          args.Q, args.p)

        if args.type == "algorithmSPEA2":
```

```
        algorithm = SPEA2(problem)
        algorithm.run(1)

    elif args.type == "algorithmNSGAII":
        algorithm = NSGAII(problem, 10000)
        algorithm.run(800)

    elif args.type == "algorithmCalcFull":
        algorithm = problem.get_all_solutions()
    else:
        raise ValueError()
    import os
    os.makedirs(os.path.dirname(DATA_PATH.format(
        args.type, args.inp)), exist_ok=True)
    with open(DATA_PATH.format(args.type, args.
        inp), 'wb') as f:
        pickle.dump(algorithm, f)
```