Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего профессионального образования
**УРАЛЬСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ**
**имени первого Президента России Б. Н. Ельцина**

Институт математики и компьютерных наук
Кафедра алгебры и дискретной математики

# Разработка и исследование алгоритмов построения экстремальных цепей в вершинно-взвешенных орграфах Частный алгоритм

Допущен к защите

_____

«__»_____ 2014 г.

Квалификационная работа
на степень бакалавра наук
по направлению «Математика,
прикладная математика»
студентки группы МТ-401
Леоновой Светланы Игоревны

Научный руководитель
Вакула Игорь Александрович,
кандидат физико-математических
наук, заведующий сектором
адаптивных систем управления
ИММ УрО РАН

Екатеринбург
2014

блабла

# РЕФЕРАТ

Леонова С.И. РАЗРАБОТКА И ИССЛЕДОВАНИЕ АЛГОРИТМОВ ПОСТРОЕНИЯ ЭКСТРЕМАЛЬНЫХ ЦЕПЕЙ В ВЕРШИННО-ВЗЕШЕННЫХ ОРГРАФАХ. ЧАСТНЫЙ АЛГОРИТМ,

выпускная квалификационная работа на степень бакалавра наук : стр. 21, рис. 6, табл. 1, форм. 6, библ. 9 назв.

Ключевые слова: АЛГОРИТМЫ, ОРИЕНТИРОВАННЫЕ ГРАФЫ, ПРОСТЫЕ ЦЕПИ, ГОРЯЧАЯ ПРОКАТКА

Объект исследования - ориентированные графы специального вида.

Цель работы - построение простых цепей максимального веса в вершинно-взвешенных ориентированных графах специального вида.

В процессе работы исследуется структура графов, возникающих при моделировании задачи планирования графиков горячей прокатки. Разрабатываются алгоритмы поиска простых цепей максимального веса. Оценивается их сложность. Полученные результаты применяются для решения более общих задач в области планирования металлургического производства.

# Оглавление

**1 Введение** **4**

**2 Постановка задачи** **6**

**3 Метод решения** **8**

**4 Частный алгоритм поиска цепи максимального веса** **9**
4.1 Анализ структуры графа . . . . . 9
4.2 Неформальное описание алоритма . . . . . 10
4.3 Формальное описание алоритма . . . . . 13

**5 Программная реализация** **15**
5.1 Используемые структуры данных . . . . . 15
5.2 Методы . . . . . 17
5.3 Инструкция пользователю . . . . . 18
5.4 Результаты экспериментов . . . . . 19

**Заключение** **20**

**Список литературы** **21**

**Приложение** **22**

# Глава 1

# Введение

В настоящей работе рассматриваются ориентированные графы специального вида. Эти графы возникают при математической постановке задачи построения графиков прокатки для металлургических предприятий. Ресурсом для прокатки металла являются так называемые *партии* - металлические заготовки обладающие определеными геометрическими параметрами. *График прокатки* представляет собой упорядоченный набор партий, прокатываемых на стане горячей прокатки. Для получения качественного металла необходимо соблюдать ряд ограничений на порядок следования партий друг за другом. Таким образом, график прокатки должен удовлетворять всем технологическим ограничениям и при этом обладать "хорошим"сочетанием показателей по ряду критериев оценки. В настоящей работе в качестве критерия выбрана максимальная суммарная длина прокатанных рулонов.
В ряде работ (например, [2], [3]) приведены практические постановки задач совместного планирования непрерывной разливки стали и последующей горячей прокатки. Подробно обсуждаются различные варианты организации работы предприятия, когда разлитый металл полностью остывает, прежде, чем идет в нагревательные печи и на прокатку, либо остывает частично в виду наличия транспортного плеча, остывает мало и идет, как принято говорить на отечественных предприятиях, в прокатку горячим садом. Иллюстрируется эффект энергосбережния и прочее. Также в работах приводятся основные технологические ограничения, связанные с формированием графиков прокатки, которые используются в аналогичных условиях большинством предприятий. Приводятся некоторые математические модели, используемые при расчете графиков прокатки, для таких задач в иностранной литературе приняты названия "hot strip mill planning"и HRBPPs (hot rolling batch planning problems).
Приводимые в работах модели представляют собой аналоги и обобщения задач коммивояжера TSP (Travelling Salesman Problem) и планирования маршрутов транспорта VRT (vehicle routing problem, [7]). Существуют модели PCTSP (Prize Collecting TSP), предложенная Балашем ( [4], [5]) и PCVRP (Prize Collecting VRP, [6]), в которых для каждой пары партий прокатки задана стоимость перехода с одной на другую, то есть стоимость того, что в графике прокатки они непосредственно следуют друг за другом. Также задается награда (приз) за то, что партия входит в построенный график прокатки, и возможнен штраф за невхождение партии в график прокатки. Результатом является построение графиков с максимизацией получаемых призов, за вычетом штрафов за переходы с одной партии на другую.
Указанные задачи являются в общем случае труднорешаемыми.

Учитывая специфичный способ описания ограничений предшествования партий в графике прокатки, принятый на отечественных предприятиях (например, ОАО "ММК"), ограничения удобно представлять в виде ориентированного графа. Предлагается удалить из целевой функции составляющую штрафов за переход, а сами

переходы отнести в состав ограничений. Таким образом, задача построения графиков прокатки в своих базовых ограничениях сводится к задаче поиска простой цепи достаточно большого веса в вершинно-взвешенном ориентированном графе. Смежность узлов в этом графе задается соотношением между ширинами и толщинами последовательно прокатываемых партий.
Основным результатом является то, что благодаря такому способу задания смежности удается построить алгоритм для нахождения простой цепи (графика прокатки) максимального веса, сложность которого ограничена кубом числа узлов (партий). Хотя в общем случае задача поиска простой цепи максимального веса труднорешаема.
Созданный алгоритм используется для оперативного планирования на станах прокатки в некоторых цехах ОАО "ММК".

Поясним причину выбора критерия оптимизации. Оценки графиков прокатки связаны с особенностями организации производства на конкретном предприятии. Вместе с тем общим является то, что график прокатки должен содержать достаточно таких партий, чтобы суммарный вес/длина партий были достаточно велики. Более того, наличие "большого"технологически допустимого графика прокатки позволяет на следующем этапе в автоматическом или ручном режиме его отредактировать, чтобы получить желаемый результат по всей совокупности практических требований. Также в качестве веса партии может выступать величина, характеризующая не только ее физические характеристики, но и предпочтительность по срокам, видам продукции и т.п. По указанным причинам для получения оптимизационной постановки задачи естественно потребовать максимальности суммарного веса партий, входящих в график прокатки.

# Глава 2

# Постановка задачи

Обозначим через $P$ множество партий $\{p_1, p_2, \ldots, p_k\}$, где $k \in \mathbb{N}$.
Пусть

- $w : P \to \mathbb{R}^+$ — ширина партии;
- $t : P \to \mathbb{R}^+$ — толщина партии;
- $l : P \to \mathbb{R}^+$ — длина партии; Для удобства в дальнейшем будем называть её весом, таким образом, $l$ - весовая функция.
- $r : \mathbb{R}^+ \to \mathbb{R}^+$ — монотонно неубывающая функция, определяющая максимальную величину допустимой разности значений толщины между двумя соседними партиями;
- $\mathbb{W} \in \mathbb{R}^+$ — величина, определяющая максимальную величину допустимой разности значений ширины между двумя соседними партиями.

Геометрические параметры партий являются основным источником накладываемых ограничений на порядок следования партий друг за другом в графике прокатки. Партия $q$ может непосредственно следовать за партией $p$ в графике прокатки в том и только в том случае, когда выполнены:

1. ограничение "для перехода по толщине":

$$|t_p - t_q| \leq \min\{r(t_p), r(t_q)\} \tag{2.1}$$

2. ограничение "для перехода по ширине":

$$0 \leq w_p - w_q \leq \mathbb{W} \tag{2.2}$$

Пусть $G$ ориентированный граф на множестве узлов $P$ с множеством дуг $E \subseteq P \times P$ таким, что $pq \in E$ в том и только в том случае, когда партии $p$ и $q$ различны, и $q$ может непосредственно следовать за $p$ в графике прокатки.

Множество всех допустимых графиков прокатки совпадает со множеством $\mathbb{B}(G)$ всех простых цепей в $G$.

Для простой цепи $z$ через $l(z)$ обозначим её вес:

$$l(z) \triangleq \sum_{p \in P(z)} l(p) \tag{2.3}$$

Сформулируем **задачу** оптимизации — найти в $\mathbb{B}(G)$ цепь максимального веса:

$$l(z) \to \max, \text{ где } z \in \mathbb{B}(G) \tag{2.4}$$

Пусть $p$ - произвольный узел в $G$. Обозначим через:

- $W = \{w_p | p \in P\}$ - всевозможные значения ширин узлов
- $P_w = \{p | w_p = w\}$, где $w \in W$, - множество узлов, имеющих данную ширину $w$.
- $G_w$ - граф, индуцированный множеством $P_w$.

Заметим, что любая наибольшая по весу простая цепь является максимальной по включению узлов. Поэтому для решения задачи 2.4 достаточно искать наибольшую по весу цепь среди максимальных по включению узлов цепей.

Таким образом, задача поиска наибольшей по весу цепи сводится к следующей задаче :

$$l(z) \to \max, \, z \in \mathbb{C}(G) \tag{2.5}$$

где $\mathbb{C}(G)$ - множество максимальных по включению узлов простых цепей в графе $G$.

В данной работе задача 2.5 решается для графа $G_w$.

Ниже будет приведен частный алгоритм решения задачи :

$$l(z) \to \max, \text{ где } z \in \mathbb{C}(G_w) \tag{2.6}$$

где $z$ - цепи с фиксированным началом $s$ и фиксированным концом $e$.

# Глава 3

# Метод решения

Для эффективного решения задачи предложен следующий подход :

1. Разрабатывается алгоритм, состоящий из двух частей - алгоритм, решающий задачу в подграфах $G_w$ и общий алгоритм, решающий задачу, в графе $G$. В настоящей работе рассматривается частный алгоритм для поиска максимальной простой цепи в графе $G_w$. Данный алгоритм используется в качестве подпрограммы для поиска максимальной простой цепи в графе $G$. Описание общего алгоритма приведено в работе [1].

2. Проводится анализ сложности алгоритма.

3. Приводятся результаты тестов и их анализ.

# Глава 4

# Частный алгоритм поиска цепи максимального веса

## 4.1 Анализ структуры графа

Для лучшего понимания структуры графа и удобства разработки алгоритма придуман удобный и естественный способ укладки графа на плоскости. Геометрические характеристики узлов выступают в роли координат. Для визуализации разработан плагин укладки для открытой программы Gephi.

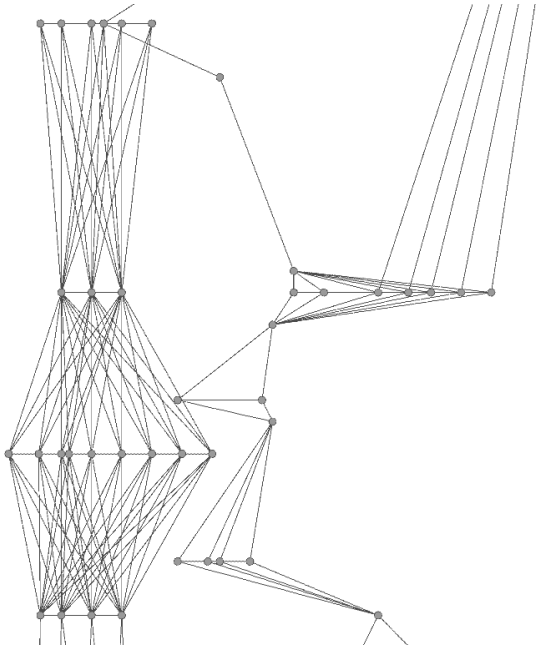

Рис. 4.1: Структура графа

На рис. (4.1) изображен фрагмент графа.

По оси абсцисс откладывается толщина узлов, по оси ординат откладывается ширина узлов, начало координат находится в левом нижнем углу. Видно, что граф складывается из подграфов $G_w$ таким образом, что дуги между подграфами направлены только вниз. Это обеспечивается условием предшествования партий по ширине. Внутри любого подграфа $G_w$ для каждой пары вершин $p, q \in P_w$ из существования дуги $pq$ следует существование обратной дуги $qp$. Эта симметрия обеспечивается модулем в условии предшествования по толщине. Также симметрия означает наличие циклов в подграфах $G_w$, что делает попытку решения задачи перебором крайне неэффективной. В данной работе речь пойдет о решении задачи в графах вида $G_w$.

## 4.2 Неформальное описание алоритма

Пусть заданы $w \in W$ и $s, e \in P_w$.

Алгоритм возвращает простую цепь $H = (q_1, q_2, \ldots, q_n)$, что $s = q_1 \to q_2 \to \ldots \to q_n = e$ -

- единственная максимальная по включению узлов $(s, e)$ - цепь для заданной ширины $w$, если $t_s \neq t_e$,
- максимальная по весу $(s, e)$ - цепь из двух максимальных по включению цепей для заданной ширины $w$, если $t_s = t_e$,
- сообщение, что простой $(s, e)$-цепи не существует.

Алгоритм заключается в последовательном добавлении узлов из множества $P_w$ в простую цепь $H$.

Рассмотрим случай, когда $t_s \neq t_e$. Пусть для определенности $t_s < t_e$. Множество $P_w \setminus \{s, e\}$ делится на три множества : $H^-, H^+, H^0$, где $H^-$ - узлы, имеющие толщину меньшую, чем у $s$. $H^+$ - узлы, имеющие толщину большую, чем у $e$. $H^0$ - все остальные узлы. Нетрудно видеть, что любой узел $p$ такой, что $t_s \leq t_p \leq t_e$, то есть узел из $H^0$, обязательно должен содержаться в максимальной по включению узлов $(s, e)$ - цепи. Поэтому изначально строится такая цепь : $H' = (s = p_1, p_2, \ldots, p_k = e)$, где $t_{p_1} \leq t_{p_2} \leq \ldots \leq t_{p_k}$. Если существует такой номер $i$, что не существует дуги $p_i p_{i+1}$, значит такую цепь $H'$ построить нельзя, следовательно простой $(s, e)$-цепи не существует.

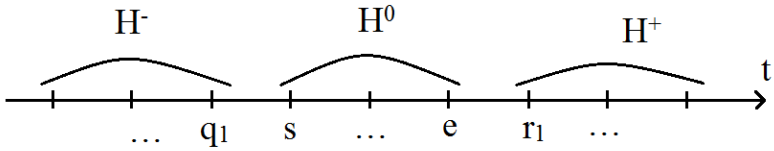


Рис. 4.2: Разбиение по толщине

Далее попытаемся добавить в $H'$ узлы из $H^-$. Выберем из этого множества узел $q_1$ с наибольшим значением толщины. Если существуют дуги $sq_1$ и $q_1 p_2$, значит этот узел можно добавить в $H'$.

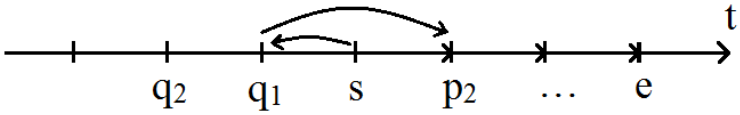


Рис. 4.3: Добавление узла из $H^-$

Затем в множестве $H^- \setminus \{q_1\}$ выбираем наибольший по толщине узел $q_2$. Если это возможно, добавляем его между узлами $s$ и $q_1$ или между $q_1$ и $p_2$, выбирая из них ту пару, чья разность по толщине наименьшая.

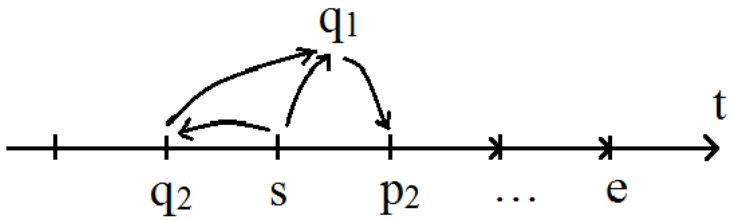


Рис. 4.4: Добавление следующего узла из $H^-$

Процесс завершится, если будут просмотрены все узлы из $H^-$ или в тот момент, когда на каком то шаге не удастся добавить узел в цепь. Это будет означать, что и все оставшиеся (меньшие по тощине) узлы так же не смогут быть добавлены.

После добавления узлов из $H^-$ получилась цепь $H''$. В нее попытаемся добавить узлы из $H^+$. Аналогично выбирается узел $r_1$ из $H^+$ с наименьшим значением толщины. Если существуют дуги $p_{k-1}r_1$ и $r_1e$ до узел $r_1$ добавляется в цепь и так далее пока не будут просмотрены все узлы из $H^+$ или пока в какой-то момент невозможно будет добавить узел из $H^+$ в текущую цепь.

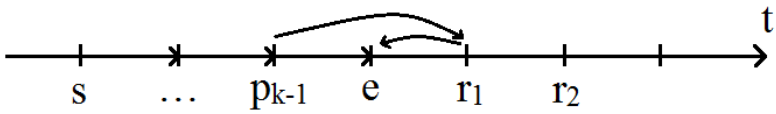


Рис. 4.5: Добавление узла из $H^+$

Пусть $t_s = t_e$. Если при этом множество $H^0$ не пусто, то данный случай аналогичен тому, что описан выше. Если $H^0$ пуст, то первыми элементами, между которыми будет вставляться наибольший по толщине узел из $H^-$ будут $s$ и $e$. Между ними же будет вставляться наименьший по толщине узел из $H^+$.

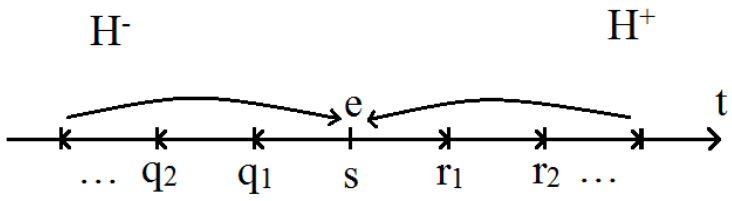


Рис. 4.6: Две максимальные по включению узлов цепи

Поэтому сначала добавляются узлы из $H^-$, затем из $H^+$, после чего получается первая максимальная по включению узлов цепь $H_1$. Затем к цепи $s \to e$ добавляются сначала узлы из $H^+$, затем из $H^-$, после чего получается вторая максимальная по включению узлов цепь $H_2$. И среди цепей $H_1$ и $H_2$ выбирается наибольшая по длине.

## 4.3 Формальное описание алоритма

Пусть $H = (p_1, p_2, \ldots, p_l)$ - произвольная простая цепь, где $l > 2$.

Пусть $i$ такой, что $p_i = \arg\max\limits_{p \in H} t_p$. Если $1 < i < l$, то *правой дугой* назовем дугу $p_i p_{i+1}$, если $t_{p_{i+1}} \geq t_{p_{i-1}}$ , иначе дугу $p_{i-1}p_i$. Если $i = 1$, то *правой дугой* будет дуга $p_i p_{i+1}$. Если $i = l$, то *правой дугой* будет дуга $p_{i-1}p_i$.

Аналогично *левой дугой* $H$ назовем дугу $p_i p_{i+1}$, если $t_{p_{i+1}} \leq t_{p_{i-1}}$, иначе дугу $p_{i-1}p_i$, где $p_i = \arg\min\limits_{p \in H} t_p$ и $1 < i < l$. Если $i = 1$, то *левой дугой* будет дуга $p_i p_{i+1}$. Если $i = l$, то *левой дугой* будет дуга $p_{i-1}p_i$.

**Частный алгоритм построения максимальной по весу простой цепи в графе $G_w$.**

1. Пусть

   $H_0 = (p_1, p_2, \ldots p_k)$, где $p_i \in P_w$ и $\forall i : t_s \leq t_{p_i} \leq t_{p_{i+1}} \leq t_t$.

   $H_0^+ = \{p | t_p > t_t\}$.

   $H_0^- = \{p | t_p < t_s\}$.

2. Проверить условие : $\forall i : p_i, p_{i+1} \in H_0 \Rightarrow p_i p_{i+1} \in E$.

   Если оно не выполнено, то нельзя построить путь максимальной длины.

3. Если $|H_0| > 2$, то

   (a) Пусть $i = 0$.
   Пока $H_i^+ \neq \emptyset$

   - $q_i = \arg\min\limits_{q \in H_i^+} t_q$, $p_j^i p_{j+1}^i$ - правая дуга $H_i = (p_1^i, p_2^i, \ldots, p_{n_i}^i)$
   - Если $p_j^i q_i \in E$ и $q_i p_{j+1}^i \in E$, то
     $H_{i+1}^+ = H_i^+ \setminus \{q_i\}$,
     $H_{i+1} = (p_1^i, p_2^i, \ldots, p_j^i, q_i, p_{j+1}^i, \ldots, p_{n_i}^i)$
     Иначе $H_{i+1}^+ = \emptyset$,
     $i = i + 1$;

   (b) Пусть $k = i, i = 0$.
   Пока $H_i^- \neq \emptyset$

   - $q_i = \arg\max\limits_{q \in H_i^-} t_q$, $p_j^i p_{j+1}^i$ - левая дуга $H_i = (p_1^i, p_2^i, \ldots, p_{n_i}^i)$
   - Если $p_j^i q_i \in E$ и $q_i p_{j+1}^i \in E$, то
     $H_{i+1}^- = H_i^- \setminus \{q_i\}$,
     $H_{k+i+1} = (p_1^{k+i}, p_2^{k+i}, \ldots, p_j^{k+i}, q_i, p_{j+1}^{k+i}, \ldots, p_{n_i}^{k+i})$,
     Иначе $H_{i+1}^- = \emptyset$,
     $i = i + 1$;

   (c) $H = H_{i+k}$ - максимальная по включению узлов Цепь.

4. Если $|H_0| = 2$, то имеет значение, какой из списков $H^+$ или $H^-$ первым добавлять в $H$. Поэтому сначала выполняются пункты 3a, 3b. Затем выполняются сначала 3b, а затем 3a. После чего получаются две максимальные по включению цепи. (Из них выбирается максимальная по длине).

5. В случае $s = t$ максимальная по длине простая цепь состоит из одного элемента $s$.

**Теорема (1).** *Частный алгоритм строит максимальную по включению цепь.*

*Доказательство.* 1. Пусть $H$ - цепь, построенная по алгоритму 1, а $H_m$ - максимальная по включению цепь. Предположим, $H$ - не максимальна, то есть существуют узлы из $H_m$, которые не попали в $H$.

2. Понятно, что если $\exists p \in H_m \setminus H$, то $t_p \notin [t_s, t_t]$, т. к. в противном случае $p$ обязательно попадет в $H$.

3. Пусть $p'p''$ - правая дуга цепи $H$. И пусть для определенности $t_{p''} \geq t_{p'}$. Тогда в силу построения $H$ и определения правой дуги справедливо равенство $t_{p'} = \max\limits_{p \in H \setminus \{p''\}} t_p$.

4. Рассмотрим множества $V^+ = \{p | p \in H_m \setminus H, t_p > t_t\}$, $V^- = \{p | p \in H_m \setminus H, t_p < t_s\}$. Пусть $V^+$ не пусто и $q = \arg\min\limits_{p \in V^+} t_p$. Покажем, что $q$ можно включить в $H$.

5. Заметим, что $H_m = V^- \cup H \cup V^+$. В силу того, что $H_m$ - цепь из $s$ в $t$, найдутся дуги из множества $V^- \cup H$ в $V^+$ и обратно. То есть найдутся узлы $v_1, v_2 \in V^- \cup H$ и $u_1, u_2 \in V^+$, что дуги $v_1u_1$ и $u_2v_2$ содержатся в $H_m$.

6. Из того, что $v_1u_1 \in H_m$ следует, что расстояние между $t_{v_1}$ и $t_{u_1}$ достаточно мало. Но расстояние между $t_{v_1}$ и $t_q$ еще меньше, поэтому дуга $v_1q \in E$. Аналогично, дуга $qv_2 \in E$.

7. В силу определения правой дуги расстояние до $t_q$ уменьшится, если заменить $v_1$ на $p'$, а $v_2$ на $p''$, то есть $p'q \in E$ и $qp'' \in E$. Тогда $q$ можно включить в $H$ по алгоритму.

8. Таким образом, любой элемент из $V^+$, так же как и любой элемент из $V^-$, можно включить в $H$.

9. Следовательно, $H$ - максимальная по включению узлов цепь. В силу пункта 4 алгоритма будет построена максимальная по длине цепь.

□

**Теорема (2).** *Частный алгоритм имеет сложность $O(n \log n)$, где $n = |P_w|$.*

*Доказательство.* Список узлов в заданной ширине $w$ сортируется по толщине за $O(n \log n)$. После чего, в худшем случае, каждый узел из $P_w$ будет просмотрен один раз. Таким образом, сложность всего алгоритма остается $O(n \log n)$. □

# Глава 5

# Программная реализация

В работе использован язык программирования Java.

## 5.1 Используемые структуры данных

В программе были реализованы следующие структуры данных :

- Вершина графа, хранящая информацию о ширине, толщине и длине партии

  class Vertex

  {

  ...

  double width;

  double thickness;

  double length;

  ...

  }

- Критерий непосредственного следования партий друг за другом в графике прокатки

  class HotCrit

  {

  ...

  double w;

  double r(thickness);

  boolean widthCrit(from, to);

  boolean thicknessCrit(from, to);

  boolean test(from, to);

  ...

  }

  Величина w и функция r известны из ограничений. Методы widthCrit(from, to) и thicknessCrit(from, to) проверяют, удовлетворяют ли партии from и to ограничениям для переходов "по ширине"и "по толщине"соответственно. Метод test(from, to) проверяет, может ли партия to следовать за партией from в графике прокатки.

- Простая цепь, представленная набором узлов path, с началом в узле start, концом в узле end и длиной length

  class Path

  {

  ...

  Vertex start;

  Vertex end;

  ArrayList<Vertex> path;

  double length;

  ...

  }

## 5.2 Методы

На входе алгоритма заданы :

1. Vertex start - начало пути;
2. Vertex end - конец пути;
3. ArrayList<Vertex> vertex - список упорядоченных по возрастанию значений толщины узлов в заданной ширине;

- Метод divide формирует три списка :

  before - упорядоченный по убыванию значений толщины список узлов, меньших по толщине, чем start

  between - упорядоченный по возрастанию толщины список узлов, толщина которых не меньше, чем толщина start, и не больше, чем толщина end

  after - упорядоченный по возрастанию толщины список узлов, толщина которых больше, чем толщина end

- Метод addBefore добавляет в текущий путь узлы из списка before.
- Метод addAfter добавляет в текущий путь узлы из списка after.
- Метод layerPath принимает на вход start, end, vertex и критерий crit. Возвращает наибольшую по длине (start, end) - простую цепь.

  layerPath (vertex, start, end, crit)

  {

  ArrayList<Vertex> path = new ArrayList<Vertex>();

  ...

  divide(vertex, start, end, before, between, after);

  path.addAll(between);

  addBefore(before, path, crit);

  addAfter(after, path, crit);

  ...

  }

## 5.3 Инструкция пользователю

Входной файл представлен списком партий, имеющих одинаковую ширину, и двумя узлами - началом и концом пути. Партии задаются тремя действительными числами - шириной, толщиной и длиной (весом). Пример:

1. 1280 2,4 4,806 - начальный узел
2. 1280 2 6,283 - конечный узел

1. 1280 2,4 4,806
2. 1280 2 6,283
3. 1280 2,8 2,352
4. 1280 2 1
5. 1280 2 5,23
6. 1280 2 9,153
7. 1280 3 0,856
8. 1280 2,8 1,08

Выходной файл содержит длину построенной цепи, последовательность узлов, вошедших в данную цепь, и их количество. Пример:
30.76000000000001
(1, 8, 7, 3, 6, 5, 4, 2)
8

## 5.4 Результаты экспериментов

Данные для тестов берутся из корпоративной сети ОАО "ММК".

Критерии непосредственного следования партий друг за другом определяются следующим образом :

1. Величина $\mathbb{W}$ для "перехода по ширине"равна 250 мм;
2. Функция $r$ для "перехода по толщине"задается следующим образом :

$$r(t) = \begin{cases} 0.8, & \text{если } 1.29 \leq t \leq 2.0; \\ 1.5, & \text{если } 2 < t \leq 3.0; \\ 2, & \text{если } 3 < t \leq 16.0; \\ 4, & \text{если } 16 < t \leq 20.0. \end{cases} \tag{5.1}$$

Генерируются выборки порядка 200-300 партий. Из заданной выборки для каждого значения ширины генерируются списки узлов этой ширины. И для каждой пары узлов из списка партий в одной ширине строятся максимальные по весу простые цепи.

Ниже приведена таблица, в которой показано отношение количества узлов в максимальной цепи к общему количеству узлов в одной ширине. Нетрудно видеть, что обычно алгоритм генерирует цепи, содержащие больше 70 процентов от общего числа узлов. Данные, по которым составлена таблица, приведены в приложениях.

| Ширина | количество узлов | количество узлов в наибольшей по весу цепи | длина наибольшей цепи | суммарная длина всех узлов |
|---|---|---|---|---|
| 1829 | 13 | 11 | 20.096 | 22.89 |
| 1820 | 5 | 5 | 10.6 | 10.6 |
| 1800 | 6 | 6 | 5.8 | 5.8 |
| 1700 | 7 | 7 | 17.9 | 17.9 |
| 1550 | 5 | 3 | 7.2 | 7.7 |
| 1250 | 11 | 8 | 53.38 | 55.3 |
| 1115 | 8 | 8 | 51.783 | 51.783 |
| 1060 | 12 | 12 | 24.4 | 24.4 |
| 1055 | 8 | 8 | 41.5 | 41.5 |

Важным результатом является высокая скорость работы алоритма, так как он запускается многократно.

# Заключение

Автоматическое планирование металлургического производства является сложной задачей, которая до сих пор не реализована полностью ни на одном предприятии в мире. Сопутствующие ей математические постановки зачастую являются труднорешаемыми. За счет исследования технологических ограничений на конкретном предприятии удалось сформулировать математическую задачу планирования графика прокатки. В настоящей работе разработан полиномиальный алгоритм решения сотпутствующей подзадачи, существенно использующийся в общем алгоритме. Удалось разработать полиномиальный алгоритм решения общей задачи. Результат уже частично внедрён и продолжается внедрятся на ОАО "ММК". В реальной жизни существуют более общие и сложные постановки задачи планирования графиков прокатки, решение которых основывается на описанных в настоящей работе алгоритмах.

# Литература


[1] Березин А. А. Разработка и исследование алгоритмов построения экстремальных цепей в вершинно-взвешенных орграфах. Общий алгоритм.

[2] Lixin Tang, Jiyin Liu, Aiying Rong, Zihou Yang. A review of planning and scheduling systems and methods for integrated steel production. European Journal of Operational Research. Volume 133, Issue 1, 16 August 2001, PP. 1–20.

[3] P. Cowling, W. Rezig. Integration of continuous caster and hot strip mill planning for steel production. Journal of Scheduling, Volume 3, Issue 4, July/August 2000, PP. 185–208.

[4] E. Balas, The prize collecting travelling salesman problem, Networks 19 (1989) 621-636.

[5] E. Balas and H. M. Clarence. Combinatorial optimization in steel rolling. Workshop on Combinatorial Optimization in Science and Technology, April, 1991.

[6] Shixin Liu. Model and Algorithm for Hot Rolling Batch Planning in Steel Plants. International Journal of Information and Management Sciences. 21 (2010), PP. 247-263.

[7] G. B. Dantzig and J. H. Ramser. Management Science, Vol. 6, No. 1 (Oct., 1959), pp. 80-91.

[8] Andreas Stenger, Michael Schneider, Dominik Goeke. The prize-collecting vehicle routing problem with single and multiple depots and non-linear cost. EURO Journal on Transportation and Logistics, May 2013, Volume 2, Issue 1-2, pp 57-87.

[9] K. Ioannidou, S. D. Nikolopoulos. The Longest Path Problem is Polynomial on Cocomparability Graphs. Algorithmica. January 2013, Volume 65, Issue 1, pp 177-205.

# Приложение

## Данные

Ниже приведена выборка, на основе который были проведены тесты.

1829 9,27 2,179
1829 9,27 0,622
1829 4,5 0,321
1829 4,5 1,603
1829 6,1 1,419
1829 4,55 0,634
1829 4,45 4,862
1829 6,1 0,709
1829 4,5 3,847
1829 6,1 4,02
1829 6,1 0,236
1829 6,1 2,128
1829 4,55 0,317
1820 6 0,483
1820 6 2,416
1820 5 2,61
1820 4 2,175
1820 6 2,9
1800 8 0,531
1800 6 0,708
1800 5 0,85
1800 4 1,063
1800 8 1,88
1800 9,99 0,851
1700 5 0,587
1700 5 1,469
1700 2,1 4,215
1700 5 0,472
1700 2,1 7,377
1700 2,1 3,162
1700 3,3 0,716
1550 3,9 0,718
1550 12 0,318
1550 10 0,198
1550 3,5 3,635
1550 3,5 2,909
1250 5 0,894
1250 2 2,528
1250 2 1,609
1250 2,8 12,211
1250 2,8 12,211

1250 2,8 12,211
1250 2,8 10,333
1250 2 1,267
1250 2 1,018
1250 5 0,447
1250 7 0,58
1115 3,8 3,842
1115 3,35 11,767
1115 3,35 8,281
1115 3,35 1,743
1115 3,35 8,281
1115 3,35 10,024
1115 3,35 0,872
1115 3,35 6,973
1060 4,5 2,431
1060 4,5 0,617
1060 3,9 3
1060 3,9 3,687
1060 3,5 0,794
1060 3,5 0,794
1060 4,2 0,618
1060 5,8 0,524
1060 5,8 2,098
1060 4,8 7,922
1060 4,2 0,372
1060 4,2 1,601
1055 3,9 0,741
1055 3,9 7,409
1055 2 14,661
1055 2 6,527
1055 3,3 2,537
1055 2,8 1,449
1055 2 6,527
1055 2,5 1,623

# Исходный код

## Примеры структур данных

Реализация класса Vertex

```
public class Vertex implements Comparable<Object> {

    double width;
    double thickness;
    int id;
    double length;

    public Vertex(double width, double thickness, double length, int i) {
        this.width = width;
        this.thickness = thickness;
        this.length = length;
        this.id = i;
    }
```

Реализация класса Crit

```
public class Crit {

    public double w;
    public double r(double t) {
        if ((t >= 1.29) && (t <= 2))
            return 0.8;
        else if ((t > 2) && (t <= 3))
            return 1.5;
        else if ((t > 3) && (t <= 16))
            return 2;
        else if ((t > 16) && (t <= 20))
            return 4;
        return -1;
    }

    public boolean widthCrit(Vertex from, Vertex to) {
        if ((from.width - to.width <= w) && (from.width - to.width >= 0))
            return true;
        return false;
    }

    public boolean thicknessCrit(Vertex from, Vertex to) {
        if (Math.abs(to.thickness - from.thickness) <= Math.min(r(to.thickness), r(from.thickness)))
            return true;
        return false;
    }

    public boolean test(Vertex from, Vertex to) {
        if ((from.id != to.id) && (widthCrit(from, to)) && (thicknessCrit(from, to)))
            return true;
        else
            return false;
    }
```

Реализация класса Path

```
public class Path {

    Vertex start;
    Vertex end;
    ArrayList<Vertex> path;
    double length;

    public Path(Vertex start, Vertex end, ArrayList<Vertex> path, double length) {
        this.start = start;
        this.end = end;
        this.path = path;
        this.length = length;
    }
```

## Примеры методов

Реализация методов addBefore, addAfter и divide

```
public static void addBefore(ArrayList<Vertex> before, ArrayList<Vertex> path, Crit crit) {
    int pos = 0;
    for (Vertex v : before) {
        if (crit.thicknessCrit(path.get(pos), v)    && crit.thicknessCrit(v, path.get(pos + 1))) {
            path.add(pos + 1, v);
            if (v.thickness - path.get(pos).thickness > v.thickness - path.get(pos + 2).thickness)
                pos++;
        } else
            break;
    }
}
```

```
public static void addAfter(ArrayList<Vertex> after, ArrayList<Vertex> path, Crit crit) {
    int pos = path.size() - 1;
    for (Vertex v : after) {
        if (crit.thicknessCrit(path.get(pos), v)    && crit.thicknessCrit(v, path.get(pos - 1))) {
            path.add(pos, v);
            if (v.thickness - path.get(pos - 1).thickness > v.thickness - path.get(pos + 1).thickness)
                pos++;
        } else
            break;
    }
}
```

```
// before упорядочен по убыванию, a between и after по возрастанию
public static void divide(ArrayList<Vertex> vertex, Vertex start, Vertex end, ArrayList<Vertex> before,
                          ArrayList<Vertex> between, ArrayList<Vertex> after) {
    for (int i = 0; i < vertex.size(); i++) {
        if ((vertex.get(i).id != start.id) && (vertex.get(i).id != end.id)) {
            if (vertex.get(i).thickness < start.thickness)
                before.add(0, vertex.get(i));
            else if (vertex.get(i).thickness > end.thickness)
                after.add(vertex.get(i));
            else
                between.add(vertex.get(i));
        }
    }
}
```

Реализация частного алгоритма

```
// принимает предварительно отсортированный список вершин
public static Path sortedLayerPath(ArrayList<Vertex> vertex, Vertex start, Vertex end, Crit crit) {
    double length = 0;
    ArrayList<Vertex> path = new ArrayList<Vertex>();
    if (start.id == end.id) {
        length = start.length;
        path.add(start);
    } else {
        // start меньше end по толщине
        boolean reverse = false;
        if (start.thickness > end.thickness) {
            Vertex v = start;
            start = end;
            end = v;
            reverse = true;
        }
        ArrayList<Vertex> before = new ArrayList<Vertex>();
        ArrayList<Vertex> between = new ArrayList<Vertex>();
        ArrayList<Vertex> after = new ArrayList<Vertex>();
        divide(vertex, start, end, before, between, after);
        path.addAll(between);
        path.add(0, start);
        path.add(end);
        // если нет пути от start до end
        for (int i = 0; i < path.size() - 1; i++) {
            if (!crit.thicknessCrit(path.get(i), path.get(i + 1)))
                path.clear();
        }



            // если есть путь от start до end
            if (!path.isEmpty()) {
                if (path.size() == 2) {
                    // в одном порядке
                    addBefore(before, path, crit);
                    addAfter(after, path, crit);
                    for (Vertex v : path)
                        length += v.length;
                    // и другом порядке
                    ArrayList<Vertex> path1 = new ArrayList<Vertex>();
                    path1.addAll(path);
                    addAfter(after, path1, crit);
                    addBefore(before, path1, crit);
                    double length1 = 0;
                    for (Vertex v : path)
                        length1 += v.length;
                    // выбираем лучший из них
                    if (length1 >= length) {
                        path = path1;
                        length = length1;
                    }


                ,
            } else {
                // если в path больше двух вершин, то нет разницы, в каком порядке
                addBefore(before, path, crit);
                addAfter(after, path, crit);
                for (Vertex v : path)
                    length += v.length;
            }
            if (reverse == true) {
                Collections.reverse(path);
                Vertex v = start;
                start = end;
                end = v;
            }
        }
    }
    return new Path(start, end, path, length);
}
```