ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ»
(Н И У « Б е л Г У »)

ИНСТИТУТ ИНЖЕНЕРНЫХ ТЕХНОЛОГИЙ И ЕСТЕСТВЕННЫХ НАУК

КАФЕДРА МАТЕМАТИЧЕСКОГО И ПРОГРАММНОГО ОБЕЧПЕЧЕНИЯ ИНФОРМАЦИОННЫХ СИСТЕМ

РАЗРАБОТКА МЕТОДА МАКСИМАЛЬНОГО СЕТЕВОГО ПОТОКА ПРИ ПЛАНИРОВАНИИ ДОБЫЧИ РУД В КАРЬЕРАХ НА ОСНОВЕ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

Выпускная квалификационная работа
обучающегося по направлению подготовки 02.04.01 Математика и компьютерные науки
очной формы обучения, группы 07001531
Яцкина Евгения Алексеевича

Научный руководитель
к.т.н., доцент
Васильев П.В.

Рецензент
к.т.н., доцент
Заливин А.Н.

БЕЛГОРОД 2017

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В настоящее время высокие информационные технологии играют важную роль не только в научной сфере, но и в повседневной жизни людей, так как они значительно упрощают и ускоряют любые процессы. Оптимизация алгоритмов является весьма актуальной задачей, поскольку позволяет выполнять расчеты быстрее и точнее, тем самым экономя рабочее время и ресурсы.

Целью научно-исследовательской работы является применение алгоритмов оптимизации границ карьеров по добыче руд для блочных моделей, а также наглядного представления (визуализации) данного процесса.

Основной задачей работы является реализация алгоритма максимального псевдопотока с использованием блочных моделей со структурой октодерева, что существенно сокращает время выполнения расчетов на сложных и сверхбольших блочных моделях. Разбиение исходной полигональной модели с помощью пирамидально-рекурсивного алгоритма, имитирует процесс сокращения крупности блоков и позволяет вычислять спектры распределения блоков каждого уровня по крупности и качеству. В данном исследовании:

- предлагается подход к моделированию карьеров по добыче руд на основе блочных моделей;

- производится приведение структуры блочной модели к структуре «ориентированный граф»;

- разрабатываются методы распараллеливания алгоритмов оптимизации, основанных на нахождении максимальных потоков в ориентированном графе блочной модели месторождения;

- создаётся программный продукт для тестирования имеющихся и оптимизированных методов распараллеливания;

Объект исследования — блочные модели карьеров по добыче руд в системах недропользования.

Предмет исследования — применение алгоритмов оптимизации границ карьеров, применяемых в программном обеспечении открытой разработки рудного сырья.

Данная выпускная квалификационная работа состоит из четырех глав.

В первой главе «Обзор и анализ предметной области» выполняется сбор необходимой информации и анализ современных методов оптимизации границ карьеров по добыче руд и их сопоставление по вычислительной сложности, особенности их применения и использования, возможность распараллеливания алгоритмов оптимизации. Также рассматриваются различные среды программирования и методы паралелльной разработки.

Во второй главе «Разработка теоретических основ оптимизации границ карьеров при помощи алгоритма псевдопотока» производится математическая постановка задачи и приводится описание алгоритма максимизации псевдопотока.

В третьей главе «Разработка программного обеспечения для решения задачи оптимизации границ карьеров» приводятся основные этапы реализации разработанного алгоритма.

Четвертая глава посвящена исследованию эффективности разработанных методов, в ней приводится описание тестовых наборов данных, использовавшихся в ходе тестирования.

Данная выпускная квалификационная работа выполнена на 84 страницах, содержит 16 рисунков, 1 приложение и 40 использованных литературных источников.

ГЛАВА 1. ОБЗОР И АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1. Задача поиска предельных границ рудных месторождений

Открытым способом разработки месторождения называется такой способ выполнения горных работ, при котором полезные ископаемые добывают непосредственно с поверхности земли. Горное предприятие, занятое открытой разработкой месторождения называется карьером.

Открытый способ имеет ряд преимуществ по сравнению с подземным:

- высокая безопасность и лучшая санитария,
- простая организация работ
- возможность селективной выемки,
- более полное извлечение полезных ископаемых,
- капитальные затраты и сроки строительства карьера меньше чем шахты.

С увеличением глубины карьера себестоимость добычи увеличивается. Предельной глубиной карьера считается такая глубина, при которой добыча открытым способом и подземным стоит одинаково. Все работы на карьере делятся на **вскрышные** и **добычные**.

Добычные работы начинаются отработкой одного или обоих бортов разрезной траншеи отдельными полосами – заходками. Вскрышные работы должны опережать добычные. Отработка верхних слоев опережает отработку нижележащих, поэтому боковая поверхность карьера имеет ступенчатую форму и называется бортом карьера. На каждый момент горные выработки карьера ограничены верхним и нижним контуром. Верхний контур ограничивает карьер на уровне поверхности земли, нижний контур на уровне

подошвы. Уступ состоит из верхней и нижней рабочей площадки, верхней и нижней бровки уступа, откоса уступа, угла откоса уступа.

При проектировании открытых карьеров по добычи полезных ископаемых рассматривается задача определения оптимального контура карьера. Решение данной задачи основано на учёте распределения добываемых полезных ископаемых и на основе допустимых, с точки зрения технологий, углов наклона контура карьера.

Процесс проектирования открытого карьера по добыче сырья состоит из нескольких шагов:

1. геологический анализ;
2. исследование механики горных пород;
3. моделирование залежей полезных ископаемых;
4. формирование дизайна окончательного карьера;
5. планирование большого радиуса добычи;
6. планирование небольшого радиуса добычи;
7. контроль качества добываемого сырья.

На основе полученных знаний формируется фиксированная 3D модель, состоящая из блоков фиксированных размеров. Блоки могут содержать в себе рудные тела, но также могут оказаться блоками с пустой породой. Блоки, содержащие в себе полезные породы имеют положительную ценность, тогда как блоки с пустой породой имеют отрицательную ценность. Сложность также заключается в том, что на каждом этапе, в процессе углубления, блоки перекрывают друг друга – это приводит к тому, что должны быть строго определены списки блоков, которые добываются первоначально, чтобы карьер приобрёл максимально допустимый контур своих границ и при этом не произошло обрушений его склонов.

Склоны в карьере формируются постепенно, по мере углубления, в ходе процесса добычи нового сырья. Для угла наклона склонов карьера существуют специальные ограничения, чтобы не произошёл обвал в результате смещения пород под своим весом.

Далее, появляется ещё один ряд задач, которые должны быть выполнены при разработке блочной 3D модели:

- инициализировать 3D модель в память ЭВМ;
- указать поверхностную топографию;
- указать геологическую информацию;
- для каждого блока, представленного в модели карьера, назначить его оценку, в зависимости от того, какие ископаемые он в себе содержит.

Благодаря компьютерному моделированию и вычислительным мощностям современных ЭВМ существенно сокращается время, необходимое для проведения расчётов и прогнозирования, а также сам процесс построения конечной модели контура открытого карьера.

Поскольку план предельно допустимого контура карьера является главной основой в планировании добычи полезных ископаемых, необходимо предельно точное его составление, поскольку он также влияет и на оценку времени переработки полученных пород и получения прибыли после реализации.

Существуют основные пункты, которые входят в проектирование плана предельного контура карьера:

1. оценка конкретного месторождения;
2. пространственное расположение будущего карьера;
3. возобновление ресурсов;
4. угол наклона границ карьера;
5. скорость добычи сырья;
6. последовательность добычи;
7. скорость добычи;
8. стоимость добычи сырья;
9. расходы на обработку и переработку сырья.

Эти параметры очень важны и тесно взаимосвязаны. При изменении одного из них будут изменяться и другие, оказывая влияние на затраты по добыче и реализации сырья.

Для выполнения оптимизационных расчетов строится дискретная блочная модель месторождения. Каждый блок модели имеет стоимостную оценку в зависимости от наличия или отсутствия в нем полезного минерала или металла. Исходными данными служат результаты опробования скважин и горных выработок. На их основе компьютерные методы построения полигональных, триангуляционных и интерполяционных моделей рудных тел позволяют получить детальные экономические блочные модели месторождений.

1.2. Модели карьера, задача определения границ

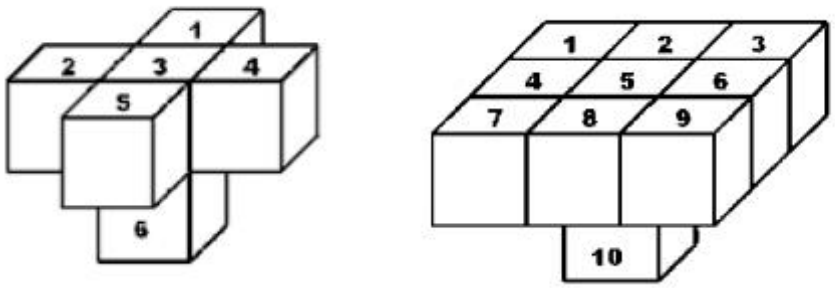
Задача определения границ *предельного карьера* (оболочки карьера на конец срока жизни горного предприятия) состоит в нахождении множества извлекаемых трехмерных блоков руды и породы с целью максимизации прибыли при наличии прецедентных ограничений, связанных с устойчивостью откосов бортов.

Геометрические ограничения на последовательность извлечения блоков (см. рис. 1.1) гарантируют, что откосы бортов карьера будут устойчивы, а горное оборудование будет иметь доступ к рабочим зонам. *Прецедентные ограничения* требуют выполнения следующего условия: при извлечении текущего блока на него непосредственно воздействуют вышележащие блоки, которые должны быть извлечены прежде, чем рассматриваемый блок. Прецедентная связь между блоками задается явно как связь транзитивного типа. Если для извлечения блока А необходимо извлечь блок В, а для извлечения блока В необходимо извлечь блок С, то для извлечения блока А также необходимо извлечь и блок С. Эта транзитивность отражается в исходных прецедентных ограничениях. Можно использовать свойство транзитивности для описания прецедентной связи как *непосредственной или прямой* - если на нее не влияет какая-либо иная пара

предшественников. Это позволяет моделировать прецедентные ограничения путем увеличения числа *прямых предшественников* в моделях.

При удалении 10 блоков угол наклона бортов для блоков лежит в пределах от 35 до 45 градусов, тогда как при удалении 6 вышележащих блоков углы крутизны склонов будут меняться в диапазоне от 45 до 55 градусов. Переходя от кубических блоков к блокам в виде параллелепипедов с различными размерами по осям *X*, *Y* и *Z* можно добиться изменения величин в необходимом диапазоне углов. Эти правила последовательности выемки блоков трактуются как некое приближение моделей стратегического планирования к реальным процессам добычи.

В этих задачах находится время, когда некоторый блок должен быть извлечен из карьера. Обычно целью при этом является максимизация чистого дисконтированного дохода (ЧДД) от добычи сырья. Ограничения включают порядок извлечения предшествующих блоков и верхние границы использования ресурсов производства.



а б

Рис.1.1. Схемы извлечения блоков, основанная на удалении

а) пяти блоков выше заданного 6-го блока

б) девяти блоков выше заданного 10-го блока

1.3. Условия оптимизации границ

Поиск оптимальных решений по извлечению запасов сырья из недр требует выполнения условий, обеспечивающих соблюдение ряда горно-геологических, геотехнических и производственных ограничений.

Используются следующие обозначения:

- *индексы и множества:*
  - t ϵ *T* - множество периодов времени *t* на уровне;
  - *b* ϵ *B* - множество блоков *b* ;
  - *b'* ϵ B_b - множество блоков *b',* предшествующих извлечению множества блоков *b* ;
  - *r* ϵ *R*- множество типов ресурсов r;
  - *d ϵ D* - множество пунктов назначения *d ;*
- *параметры:*
  - $P_b(P_{bt}, P_{bd}, P_{btd},)$ прибыль, получаемая от извлечения (и переработки) блока *b* (в период времени *t* при отправке по назначению *d*);
  - *a* - скорость дисконтирования, используемая в вычислении коэффициентов целевой функции (прибыли);
  - $q_{br}(q_{brd})$- количество ресурса *r,* используемого для извлечения и, при необходимости, переработки блока *b* (при отправке по назначению *d*) (в тоннах);
  - $\bar{R}_{rt}$ - максимально доступный операционный ресурс *r* в период времени *t* (в тоннах);
  - R_{rt} - минимально доступный операционный ресурс г в период времени / (в тоннах);
  - *A* - коэффициенты дополнительных общих ограничений;
  - $\underset{\circ}{a},\bar{a}$ - нижние и верхние границы, соответственно, для дополнительных общих ограничений (векторы с числом рядов равным этому числу в *A);*

- переменные:
  - x_b = 1, если блок b находится в проекте конечного карьера, а иначе о;
  - x_{bt}= 1, если блок b извлекается в период времени t, а иначе о;
  - y_{bdt} — количество блока b отправленное по назначению d в период времени t (%).

Приведем следующие математические формулировки трех главных задач оптимизации:

Задача нахождения предельных границ карьера, **(UPIT)**, или задача поиска замыкания с максимальным весом. Эта задача определяет наиболее прибыльную оболочку из блоков внутри рудного тела и, следовательно, в ней не учитываются факторы времени и какие-либо производственные ограничения. Набор ограничений состоит только из прецедентных отношений между блоками; соответствующая матрица коэффициентов, упорядоченных при обходе блоков против часовой стрелки, является полностью унимодулярной, представляющей эту задачу как задачу сетевого потока. По сути, используя ценность каждого блока без всяких ограничений на необходимые производственные ресурсы по выемке, решение этой задачи показывает немедленную прибыль от карьера и, соответственно, устанавливает то, какие блоки должны быть извлечены согласно прецедентным ограничениям для получения данной прибыли.

$$(\text{UPIT}) = \max \sum p_b x_{b,} \qquad (1.1)$$

При условии,

$$x_b \leq x_{b'}, b \in B,\ b' \in B + B_b \qquad (1.2)$$

$$x_b \in \{0,1\}\ b \in B$$

Цель состоит в том, чтобы максимизировать чистую прибыль от всех извлеченных блоков. Ограничения гарантируют, что каждый блок извлекается только в том случае, если добыты предшествующие ему блоки. Множество предшествующих блоков соответственно задают углы откосов проектируемого предельного карьера. Решение задачи (UPIT) определяет только проект карьера, то есть его границы. Решение этой задачи фактически может использоваться для исключения блоков из рассмотрения в дальнейших задачах.

Задача нахождения предельного карьера с ограничениями, (CPIT), обобщает вышеприведенную задачу нахождения предельных контуров карьера за счет ввода в набор данных фактора времени и связанных с ним ограничений. При этом для модели и её модификаций предполагается, что каждый добычной блок целиком отрабатывается за один период времени. В задаче (CPIT) рассматриваются не только прецедентные ограничения, но и ограничения на доступные операционные ресурсы в заданный период времени. На вход решения задачи (*CPIT)* поступают:

- доход по каждому блоку
- минимальная и максимальная допустимая величина операционного ресурса в заданный период времени
- множество предшественников для каждого блока

При указанных входных данных решение задачи (*CPIT)* предполагает составление плана графика добычи, обеспечивающего максимальную прибыль при наличии ограничений операционных ресурсов и прецедентных связей между блоками. В задаче (CPIT) не принимаются в расчет такие детали, как возможность промежуточного складирования.

$$(CPIT)$$

$$\max \sum_{b\in B}\sum_{t\in T} p_{bt}x_{bt} \quad (1.3)$$

$$\text{если } \sum_{\tau\le t} x_{b\tau} \le \sum_{\tau\le t} x_{b'\tau} \quad \forall b\in B, b'\in B_b, t\in T \quad (1.4)$$

$$\sum_{t\in T} x_{bt} \le 1 \quad \forall b\in B \quad (1.5)$$

$$\underline{R}_{rt} \le \sum_{b\in B} q_{br}x_{bt} \le \overline{R}_{rt} \quad \forall t\in T, r\in R \quad (1.6)$$

$$x_{bt}\in\{0,1\} \quad \forall b\in B, t\in T \quad (1.7)$$

В задаче (CPIT) ищется максимальный чистый доход от извлечения блоков за весь срок жизни горного предприятия. При этом p_{bt} вычисляется как

$$\frac{p_b}{(1+a)^t}$$

Условия (1.4) устанавливают предшественников. То есть, если блок b' является непосредственным предшественником блока b, то b' должен быть извлечен в тот же самый период времени, что и блок b или ранее. Условия (1.5) требуют, чтобы каждый блок добывался не более одного раза. Условия (1.6) гарантируют, чтобы ограничения значений минимального и максимального операционного ресурса соблюдались для каждого периода времени.

Задача планирования продукции с прецедентными ограничениями (PCPSP), решается при выполнении следующих условий:

$$(PCPSP)$$

$$\max \sum_{b\in B}\sum_{d\in D}\sum_{t\in T} p_{bdt}y_{bdt} \quad (1.8)$$

$$\text{если } \sum_{\tau\le t} x_{b\tau} \le \sum_{\tau\le t} x_{b'\tau} \quad \forall b\in B, b'\in B_b, t\in T \quad (1.9)$$

$$x_{bt} = \sum_{d\in D} y_{bdt} \quad \forall b\in B, t\in T \quad (1.10)$$

$$\sum_{t\in T} x_{bt} \le 1 \quad \forall b\in B \quad (1.11)$$

$$\underline{R}_{rt} \le \sum_{b\in B}\sum_{d\in D} q_{bdr}y_{bdt} \le \overline{R}_{rt} \quad \forall t\in T, r\in R \quad (1.12)$$

$$\underline{a} \le Ay \le \overline{a} \quad (1.13)$$

$$y_{bdt}\in[0,1] \quad \forall b\in B, d\in D_b, t\in T \quad (1.14)$$

$$x_{bt}\in\{0,1\} \quad \forall b\in B, t\in T \quad (1.15)$$

Задача (PCPSP) максимизирует чистый доход от извлечения блоков за весь срок жизни рудника.

Ограничение (1.10) предписывает требование о непротиворечивости значений переменных для добычи и переработки. То есть, если блок не извлечен, то его материал не может быть отправлен ни по одному из пунктов назначения, а если извлечен, то весь его материал может быть отправлен куда угодно. Условие (1.11) вводит ограничение, что блок на любом горизонте (уровне) может быть извлечен только один раз. Ограничения (1.12) требуют, что кроме уже имеющихся больше никаких операционных ресурсов для целей добычи не используется. Условия (1.13) определяют общие дополнительные ограничения, подробное обсуждение которых приводится ниже. Заметьте, что поскольку x функционально зависит от y (см. условие (ю)), то предыдущая переменная не включается в это условие. Значение переменной, определяющей долю блока, направляемую по конкретному назначению в заданный период времени, должно лежать в диапазоне от 0 до единицы. Переменная, определяющая будет ли блок извлечен в конкретный период времени, является бинарной.

1.4. Методы оптимизации

Методы глобальной оптимизации

Оптимизация в широком смысле слова находит применение в науке, технике и в любой другой области человеческой деятельности.

Оптимизация – целенаправленная деятельность, заключающаяся в получении наилучших результатов при соответствующих условиях.

При постановке задачи оптимизации необходимы следующие критерии:

- Наличие объекта оптимизации и цели оптимизации. При этом формулировка каждой задачи оптимизации должна требовать

экстремального значения лишь одной величины, то есть одновременно в системе недолжно приписываться два и более критерия оптимизации, так как практически всегда экстремум одного критерия не соответствует экстремуму другого;

- Наличие ресурсов оптимизации, под которыми понимают возможность выбора значений некоторых параметров оптимизируемого объекта. Объект должен обладать определенными степенями свободы – управляющими воздействиями;
- Возможность количественной оценки оптимизируемой величины, поскольку только в этом случае можно сравнивать эффекты от выбора тех или иных управляющих воздействий;
- Учет ограничений [8].

Численные методы безусловной оптимизации

Задача безусловной оптимизации:

В каждом конкретном случае особо указывается (или это ясно из контекста), идет ли речь об отыскании локального или глобального минимума.

Значительное внимание уделено классическим методам минимизации — градиентному методу и методу Ньютона. Эти методы имеют важное значение в идейном отношении. Оба они явным образом основаны на идее замены минимизируемой функции в окрестности очередной точки x^k первыми членами ее разложения в ряд Тейлора. В градиентном методе берут линейную часть разложения, в методе Ньютона — квадратичную часть. Многие из этих методов оптимизации базируются на той же идее аппроксимации функций [11].

К численным методам безусловной оптимизации относятся:

- Градиентный метод;
- Метод Ньютона и его модификации;
- Методы сопряженных направлений;
- Эвристические методы нулевого порядка [11].

Численные методы условной оптимизации

Создание численных методов решения зада оптимизации с ограничениями является еще более трудной проблемой, чем построение методов безусловной оптимизации. Эффективные алгоритмы удается построить лишь для специальных классов условных задач, к которым в первую очередь следует отнести задачи линейного, квадратичного и выпуклого программирования.

К численным методам условной оптимизации относятся:

- Симплекс-метод задач линейного программирования;
- Метод проекции градиентов;
- Метод условного градиента;
- Конечный метод решения задач квадратичного программирования;
- Метод штрафных функций;
- Метод параметризации целевой функции;
- Метод линеаризации [11].

Методы дискретной оптимизации

Методы дискретной оптимизации решают задачи минимизации или максимизации функции f, определенной на множестве X с элементами $x = (x_1, \dots, x_n)$. При этом множество X дискретно, либо лишь некоторые из переменных $x_1, \dots, x_n$ пробегают дискретные множества, когда x пробегает X. Дискретными называют конечные множества и счетные множества без предельных точек. Часто X – это множество точек с целочисленными координатами, удовлетворяющими некоторым ограничениям.

К методам дискретной оптимизации относятся:

- Метод ветвей и границ;
- Метод динамического программирования [11].

1.5. Метод вариантов

В методе вариантов применяется традиционный подход к проектированию карьеров. Основные решения принимаются проектировщиком, а компьютер используется для вычислений площадей, объемов, содержания полезного компонента в руде и технико-экономических показателей.

При расчетах по блочной модели пространство месторождения разбивается на прямоугольные блоки и на основе данных геологоразведочных скважин каждому блоку присваивается кодовый признак породы и процентное содержание полезных компонентов. Базовый вариант границ карьера задается посредством назначения периметра дна, по которому автоматически проектируется верхний контур карьера в соответствии с допустимыми углами откосов рабочего и нерабочего бортов. Объемы пород и полезного ископаемого вычисляются в пределах каждого контура на всех горизонтах. Далее, путём расширения базового периметр, получают различные варианты приращения карьера. Для всех вариантов вычисляют объемы горной массы, коэффициенты вскрыши, затраты на разработку и т.д.

Критерием оценки варианта считается ожидаемая прибыль P . При этом из общего дохода от реализации продукции вычитают затраты на обогащение, переплавку, окомкование, транспортирование для сбыта. С учетом этого прибыль по вариантам рассчитывается по формуле:

$$P = \sum_{n=1}^{N} \frac{I_n - Q_n}{(1+p)^n} - C \rightarrow \max \qquad (1.16)$$

где I_n — ежегодный доход; Q_n — эксплуатационные затраты; p - доля прибыли; N — срок эксплуатации карьера; n — расчетный год; C — капитальные затраты. Детализация каждого из вариантов позволяет установить ожидаемое минимальное промышленного содержание металла в

руде, производственную мощность карьера и приступить к решению задачи календарного планирования добычи.

1.6. Плавающий конус

В данном алгоритме элементарная фигура формирования границ карьера — перевернутый усеченный конус, меньшее основание которого имеет размеры, соответствующие минимальной ширине дна карьера. Плоскость, образующая боковую поверхность конуса, наклонена к горизонтальной плоскости под углом, равным углу откоса конечного борта карьера. Этот метод имеет несколько вариантов, но его суть можно объяснить на основе использования двух типов блочных моделей с трехмерными массивами данных. Одна из моделей является моделью задачи, содержащая исходные значения по каждому блоку, вторая - моделью решения, первоначально имеющая нулевые значения.

Процедура поиска оптимальности границ карьера начинается с верхнего горизонта, содержащего полезное ископаемое.

В модели задачи отыскивается некоторый блок, который при его извлечении вместе со всеми блоками в конусе над ним обеспечивает прибыль. Когда такой блок найден, то он извлекается (добывается) - перемещается в модель решения, а нулевые значения помещаются в модель задачи. При построении конусов учитываются ограничения на допустимый наклон границ карьера.

При рассмотрении двумерного случая с углом откоса борта карьера в 45 градусов данное изображение показывает множество значений в начальный момент работы программы. Первый найденный конус с положительным значением закрашен желтым цветом. Заметим, что блок со значением +2 перекрыт сверху тремя блоками со значением -1. Желтые блоки теперь будут перемещены в модель решения.

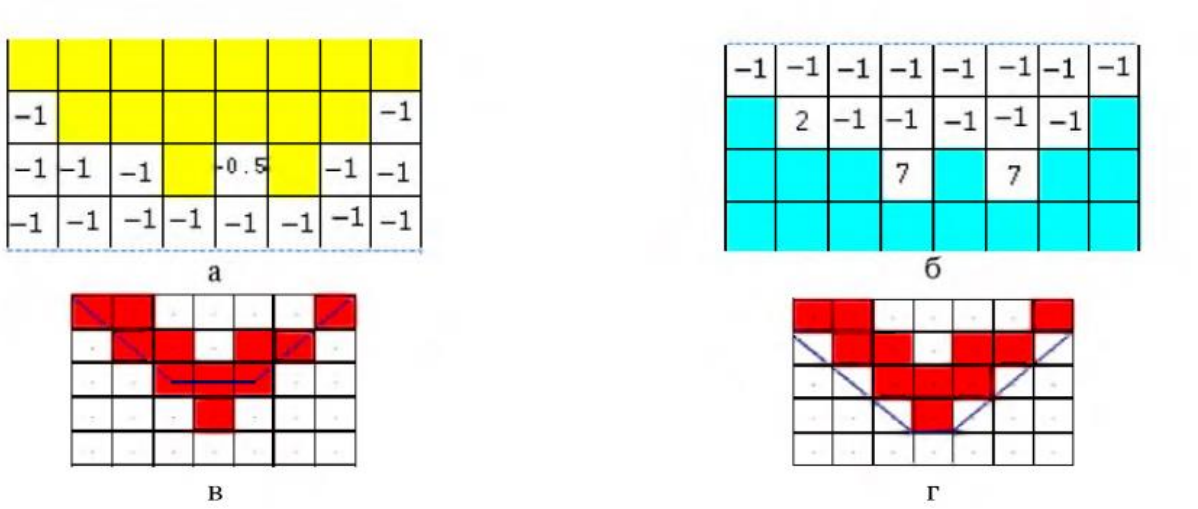


Рис.1.3. Варианты расположения конусов: а – модель задачи; б – модель решения; в - контур по центрам блоков; г - контур по границам блоков

-1	-1	-1	-1	-1	-1	-1	-1
-1	2	-1	-1	-1	-1	-1	-1
-1	-1	-1	7	-0.5	7	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1

Рис. 1.4. Конус с положительным значением

Поиск более выгодных конусов для модели решения может быть продолжен до тех пор, пока больше не останется вариантов. В ряде случаев этот процесс, к сожалению, будет перемещать в модель решения те блоки, которые не должны там находиться. Поэтому следующим этапом является поиск в файле модели решения таких блоков, которые будучи помещенными на вершину конуса, будут иметь отрицательное значение. Если такие конусы обнаруживаются, то они перемещаются обратно в блочную модель задачи и этим блокам приписываются нулевые значения.

0	-1	-1	-1	-1	-1	-1	-1
0	0	-1	-1	-1	-1	-1	0
0	0	0	7	-0.5	7	0	0
0	0	0	0	-1	0	0	0

Рис.1.5. Диаграмма блочной модели решения

Диаграмма показывает «модель решения» - это те блоки, которые, как было обнаружено на первом шаге, находятся в положительном конусе. В «модели решения» мы находим обратный конус с отрицательным значением и закрашиваем его синим цветом. Эти блоки необходимо вернуть обратно в исходную «модель задачи», присвоив нулевые значения.

Вновь возвращаясь к модели задачи мы обнаруживаем, что появился новый блок со значением 2, который теперь может быть извлечен с прибылью. После его удаления в модель решения в исходной модели задачи больше не будет каких-либо положительных конусов, а в модели решения нельзя будет найти каких-либо отрицательных конусов (обратных). В этом случае алгоритм находит наилучший из возможных контуров карьера.

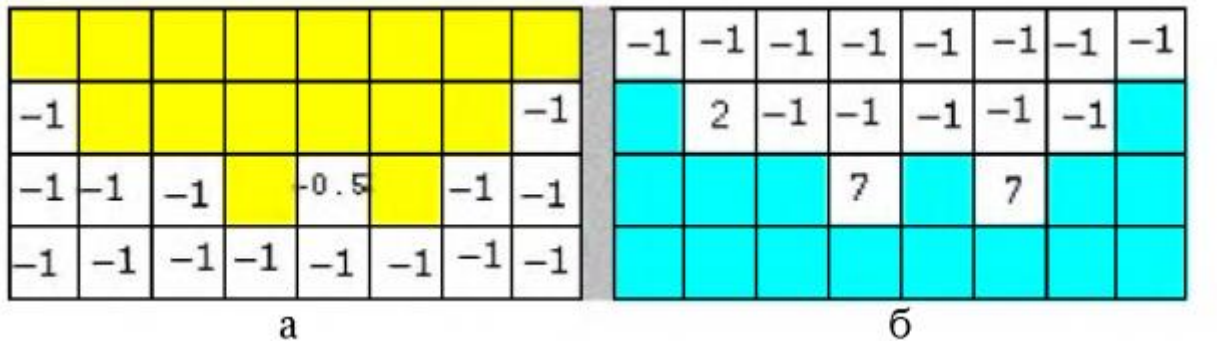


Рис. 1.6. Окончательный вид границ карьера: а)«модель задачи»; б)«модель решения»

Для большей эффективности алгоритма целесообразно использовать один массив значений стоимости блоков и создать средства переключения с

помощью флажка между моделями задачи и решения путём введения массива булевых значений.

По сути, существует не один, а множество алгоритмов «плавающего конуса». Они отличаются по способам поиска положительного конуса, по процедурам переключения и отыскания отрицательного конуса в модели решения. Методы плавающего конуса удобны для представления углов откоса бортов карьеров. Хотя первоначальный метод плавающего конуса не гарантирует нахождение наилучшего оптимального контура карьера, однако в последнее время появились варианты улучшающие его работ. Кроме того, в настоящее время он является единственным методом, для которого удалось использовать вместо регулярной блочной модели структуру представления данных в виде октодерева.

1.7. Методы теории графов: алгоритм Лерча-Гроссмана

В практике проектирования карьеров алгоритм Лерча-Гроссмана, основанный на теории графов, получил наибольшее распространение. В алгоритме LG блочная модель месторождения представляется в виде ориентированного графа $G=(V,A)$. Каждому блоку трехмерной модели i соответствует узел графа с весом b_i, определяющим чистую экономическую ценность данного блока. Два узла i и j соединяются направленной дугой от i *к* j, если блок i не может быть извлечен раньше, чем блок j. Обычно блок j в таком случае располагается в верхнем слое, прямо над блоком i. Задача состоит в нахождение такого множества блоков, извлечение которых принесет максимальную прибыль. Это эквивалентно нахождения замкнутого набора узлов графа с максимальной суммой весов. Замкнутым считается множество узлов, которое для каждого узла содержит все его дочерние узлы.

Важной концепцией теории графов является *замыкание.* Для взвешенного ориентированного графа *замыкание* определяется как некое множество вершин C, таких, что если $u \in C$ и (u,v) есть дуга арки в графе, то v

є *C*. Вес замыкания равен сумме весов вершин в замыкании. Или, иначе, в контексте выполнения горных работ *замыкание* представляет собой некоторый подходящий контур карьера, суммарный вес которого равен ценности добытых блоков в замыкании. Таким образом, проблема определения контура карьера, удовлетворяющего требованиям безопасных углов наклона бортов и максимизирующего чистый доход, преобразуется в задачу оптимизации графа и определения замыкания максимального веса во взвешенном направленном графе.

По сути, алгоритм описывается следующими шагами:

Шаг 0. Первоначально дерево графа образуют ребра, соединяющие все блоки с корневым фиктивным блоком. Это, очевидно, есть нормализованный граф.

Шаг 1. Проверка - имеется ли направленное ребро (дуга) в графе откосов (α,β), такое, что α - является сильной вершиной, а β - нет? Если нет, то переходим к шагу 4 – конец.

Шаг 2. Пара вершин, найденная в шаге 1, добавляется к дереву графа. Немедленно граф дерева прекращает быть деревом, так как мы имеем структуру с фиктивным ребром на обоих концах. Для того чтобы исправить эту ситуацию удаляется это фиктивное ребро, поддерживающее (ранее) сильную ветвь. Шаг 2 достаточно легок в понимании, однако сложен для программирования. Хотя в оригинальной статье говорится о добавлении или удалении дуг из дерева, но на самом деле речь идёт о ребрах (деревья не направленны).

Шаг 3. Состоит в том, чтобы нормализовать граф циклически назад вплоть до шага 1. После того как граф нормализован до шага 1, нам остается пройти по вновь созданной ветви от листа назад до корневого блока. Внешняя часть этой ветви теперь будет иметь различный смысл для направления от корня и все вершины в ветви, очевидно, изменят поддерживаемый ими вес.

Шаг 4. Множество сильных вершин представляет собой оптимальный карьер. Множество сильных вершин является множеством, которое поддерживается сильными dummy вершинами. Завершение работы алгоритма.

1.8. Методы теории графов: алгоритм максимизации псевдопотока

В работе Хочбаум нормализованные деревья алгоритма LG были адаптированы к более общей модели сетевого потока на основе концепции псевдопотока, похожего на предпотоки (preflow). Сетевой псевдопоток удовлетворяет ограничениям пропускной способности, но в нём могут нарушаться условия баланса потока с созданием дефицита или избытка в узлах. Предпоток удовлетворяет ограничениям пропускной способности, но в нём может нарушаться баланс потока лишь путём создания избытка в узлах. Алгоритм псевдопотока решает задачу нахождения максимального потока в обобщенных сетях и работает с псевдопотоками вместо масс. Связь между алгоритмом псевдопотока и алгоритмом предпотока (переразметки, push-relabel) более очевидна, чем между алгоритмом LG и алгоритмом предпотока. Методы LG и псевдопотока имеют дело с множеством узлов (ветвей), способных аккумулировать либо избыток, либо дефицит. В подходе на основе псевдопотока, массы $M_{ге}$, поддерживаемые корневым узлом 7 * сильного дерева, трактуются как некий псевдопоток и выталкиваются к слабому корню r_w , а затем к фиктивному корневому узлу $x_г$, (одновременно являющемуся как источником, так и стоком). Алгоритм выталкивания с переразметкой работает с предпотоками (preflows). Алгоритм переразметки работает скорее с самими узлами, а не с множествами узлов, и избыток в некотором узле выдавливается в находящиеся ближе к стоку узлы в соответствие со значениями дистанционных меток, переразметка обновляет значения меток.

Алгоритм псевдопотока предоставляет несколько систематических путей обработки вершин типа слабый-над-сильным. Лучшими из этих методов являются варианты с нижней и верхней метками. Методы нижней и верхней разметки работают с концепцией дистанционной метки. Для некоторого узла дистанционная метка представляет собой неубывающую функцию и на этом уровне она является неубывающей в составе любого сгенерированного дерева. В статье доказывается, что дистанционная метка есть неубывающая функция на заданном уровне дерева и для слабого узла *v* она является некоторой нижней границей уровня *(v).* Функция дистанционной метки аналогична дистанционным меткам и меткам, используемым в методах сетевого потока.

Как отмечается в работе, Хочбаум развила алгоритм LG за счет включения в него концепции псевдопотока с формулировкой задачи сетевого потока. Такая формулировка дополняет базовый алгоритм LG структурированной стратегией определения последовательности обработки множества дуг.

Реализованный алгоритм систематически находит последовательность нормализованных вложенных остовных деревьев в виде расширенных замкнутых подграфов основного графа. Алгоритм объединяет на каждом этапе всё большее количество дуг из подграфа. Последовательность образующихся деревьев содержит подмножества (сильные ветви), чьи вершины формируют максимальное замыкание вложенного дерева и частичное замыкание подграфа и стремится в пределе к максимальному замыканию подграфа. Алгоритм может начинаться с любого нормализованного вложенного остовного дерева замкнутого расширенного подграфа G' исходного графа. В данной реализации создается фиктивный корневой узел x_0 и строится начальное связующее дерево, имеющее корень в фиктивном узле и ветви, отходящие к каждому узлу дерева в виде подграфа *G.* Таким образом, Tx_0 представляет собой нормализованное вложенное остовное дерево расширенного подграфа №. Предполагаться, что этот

подграф в целом должен быть графом без потери общности. Максимальное замыкание замкнутого подграфа содержится внутри максимального замыкания всего графа.

Алгоритм состоит из двух шагов, которые повторяются до тех пор, пока вершины сильных ветвей не сформируют максимальное замыкание взвешенного ориентированного графа G. Имеется шаг поглощения (merger) и шаг обрезки или нормализации (pruning). Таким образом, процесс начинается с некого супер оптимального множества (первоначально сильных ветвей Tx_0, образующих множество всех положительных вершин и которое может быть эффективно найдено, например, с помощью эвристического генетического алгоритма), которое не удовлетворяет ограничениям и преобразуется к подмножеству максимального замыкания, удовлетворяющего принятым ограничениям.

На каждом этапе имеются несколько переменных, связанных с каждой вершиной или ребром нормализованного дерева. Они представляют вес поддерева, исходящего из ребра или вершины, тип ребра (*р* или *пг*), связывающего вершину с её предком (первоначально с корнем x_0), и тип ребра или вершины (слабая или сильная). Ребро в нормализованном поддереве является сильным, если и только если оно является р-ребром и поддерживаемый этим ребром вес является положительным по определению и согласно “свойству 3”. Другие переменные являются счетчиками и housekeeping-переменными. Сама по себе древовидная структура является довольно сложной и представляет собой множество связанных списков, а структура данных спроектирована для оптимизации проходов по дереву и рекомбинирования. В нормализованном дереве все вершины у ветвей, связанных с корнем x_0, являются либо все сильными, либо все являются слабыми. В методах псевдопотока с нижней и верхней меткой для определения порядка обработки сильных деревьев применяются приоритетные очереди.

1.9. Среды разработки программного обеспечения

Разработка параллельных программ в настоящее время осуществляется с применением инструментальных и языковых средств, ориентированных на конкретные вычислительные архитектуры.

Intel Parallel Studio

Компания Intel считается одним из лидеров в индустрии параллельных вычислений. Intel расширяет собственную долголетнюю практику создания программных инструментов в сторону клиентских приложений для ПК. Практическая деятельность перенесения или «конверсии» технологий из области научных исследований и высокопроизводительных вычислений в масс-маркет постоянно приносила отличные результаты. В настоящее время начинается период, когда программистам клиентских Windows-приложений потребуются мощные и удобные инструменты с целью приспособления имеющихся или написания новейших программ, предельно использующих эффективность пользовательских систем на основе многоядерных процессоров. Данная компания решила назвать эту эко-систему мэйнстримом (mainstream), в которой используются требовательные к производительности приложения для десктопов и мобильных систем, написанные под Windows на языке C или C++. Очевидно, что зачастую графический интерфес приложений разработан с применением Java или .Net и managed-языков. Но компания надеется, что в большинстве ситуаций различные куски приложений, которые взыскательны к производительности, написаны непосредственно на C/C++, и в них немаловажно достичь задействования абсолютно всех потенциалов микропроцессора.

На сегодняшний день главным набором разработчика программного обеспечения считается Microsoft Visual Studio. Данная фирма дает отличное предложение для увеличения возможности MVS, чтобы упростить и улучшить цикл разработки масштабируемых параллельных программ для Windows. Ни для кого не секрет, что последующее увеличение

производительности приложений станет достигаться за счет превосходной распараллеленности и масштабируемости в связи с увеличением процессоров в системе. Совершенное программное обеспечение будет автоматически применять всю силу новейших процессоров, выпускаемых в ближайшее время, из-за вовлечения в работу большего числа ядер, количество которых на кристалле регулярно увеличивается с каждым поколением микроархитектуры чипов.

Intel® Parallel Studio – данный комплект из нескольких инструментов, который считается гармоничным продолжением либо расширением MVS и дает возможность за счет удобства применения, ясного интерфейса и уникальных технологий достигать превосходной производительность параллельных программ на многоядерных системах. Несмотря на то, что данный набор является плагином к VS, он целиком покрывает все без исключения этапы разработки программного обеспечения программистом, от создания скелета предстоящей параллельной программы вплоть до оптимизации релизной версии проекта. В состав данного набора включено 4 отдельных продукта. Они применяются в собственном сегменте цикла разработки, и любой может быть проинсталлирован и включен в VS как по-отдельности, так и абсолютно всем пакетом мгновенно.

В состав пакета входят:

- Intel® Parallel Advisor: поможет найти возможности распараллеливания кода с самого начала разработки приложения**
- Intel® Parallel Composer: предназначен для генерирования параллельного кода, т.е. создания программ с помощью компилятора и широкого набора библиотек для многопоточных алгоритмов
- Intel® Parallel Inspector: проверит ваше параллельное приложение на корректность и найдет ошибки работы с памятью
- Intel® Parallel Amplifier: обнаружит «узкие места» в программе, которые мешают масштабируемости и увеличению производительности на мультиядерных платформах [12].

Microsoft Visual Studio

Microsoft Visual Studio — линейка продуктов компании Microsoft, включающих интегрированную среду разработки программного обеспечения и ряд других инструментальных средств. Данные продукты дают возможность создавать как консольные приложения, так и приложения с графическим дизайном, в том числе с помощью технологии Windows Forms, а кроме того веб-сайты, интернет-приложения, интернет-службы как в родном, так и в контролируемом кодах для всех платформ, поддерживаемых Windows, Windows Mobile, Windows CE, .NET Framework, Xbox, Windows Phone .NET Compact Framework и Silverlight.

VS содержит в себе редактор исходного кода с поддержкой технологии IntelliSense и возможностью простейшего рефакторинга кода. Интегрированный отладчик способен работать как отладчик уровня исходного кода, так и как отладчик машинного уровня. Остальные встраиваемые инструменты включают в себя редактор форм для упрощения создания графического интерфейса приложения, веб-редактор, дизайнер классов и дизайнер схемы базы данных. VisualStudio дает возможность делать и подсоединять другие дополнения (плагины) с целью расширения функциональности почти на каждом уровне, в том числе добавление поддержки систем контроля версий исходного кода, добавление новейших комплектов инструментов (к примеру, с целью редактирования и визуального проектирования кода на предметно-ориентированных языках программирования либо инструментов для прочих аспектов процесса разработки программного обеспечения (например, клиент Team Explorer для работы с Team Foundation Server) [14].

Cuda Toolkit – это программный пакет, который содержит инструменты, библиотеки, заголовочные файлы для компиляции программ с помощью Microsoft Visual Studio.

GPU Computing SDK содержит в себе демонстрационные примеры, которые уже предварительно сконфигурированы для удобной работы в среде Microsoft Visual Studio.

Данные пакеты поставляются как для 32-х разрядных, так и для 64-х разрядных версий Windows.

Qt - кроссплатформенный инструментарий разработки ПО

Qt — кроссплатформенная библиотека классов C++. Со временем библиотека разрослась вширь и вглубь, и теперь, помимо разработки собственно графических интерфейсов, включает в себя сотни классов, охватывающих самые разные аспекты программирования — от разработки приложений баз данных до создания мультимедийных программ и от работы с динамически загружаемыми модулями до конечных автоматов и собственного интерпретатора языков сценариев. И все это работает практически одинаково на таких разных платформах как Microsoft Windows, Linux, Mac OS, многочисленные коммерческие версии UNIX, а также WinCE и Symbian. Библиотека Qt работает на 32- и 64-битных процессорах Intel, процессорах ARM старших моделей и некоторых других. Библиотеку весьма просто локализовать, то есть перевести её текстовые ресурсы на новый язык, что уже сделано (и постоянно делается) для большинства языков мира, так что после установки в русскоязычной системе многие компоненты Qt сразу же "заговорят" по-русски. Инструментарий, необходимый для локализации приложений, как и многие другие вспомогательные инструменты, входит в состав самой Qt.

Сравнительно недавно библиотека Qt перестала быть только библиотекой классов C++ и вспомогательных элементов и обзавелась собственной кроссплатформенной интегрированной средой разработки Qt Creator (что, однако, не мешает работать с библиотекой Qt в других популярных средах, таких как Microsoft Visual Studio и Eclipse).

Библиотека Qt используется для разработки приложений крупнейшими поставщиками программного продукта. Достаточно назвать такие компании

как Nokia (которая в данный момент и владеет библиотекой Qt), Oracle и Google. Кроме того, на основе Qt разрабатывается одна из самых популярных графических оболочек для Linux — KDE, и огромное количество открытых приложений для Linux и других UNIX-систем [2].

Embarcadero RAD Studio 10 (Parallel Programming Library)

Embarcadero RAD Studio представляет собой набор средств разработки приложений, который позволяет создавать приложения с графическим пользовательским интерфейсом для Windows, Mac OS X, .NET, PHP и веб-решений. В её состав входят:

- Embarcadero Delphi дает возможность создавать полнофункциональные приложения для Windows и Mac OS X.
- Embarcadero C++ Builder — это среда C++, которая полностью соответствует концепции быстрой разработки приложений, объединяет средства ANSI C++ и многофункциональную расширяемую инфраструктуру визуальных компонентов.
- Embarcadero Prism™ XE2 представляет собой кросс-платформенное решение для разработки и Delphi-образный язык программирования для быстрой разработки приложений .NET, Mono, ASP.NET и приложений создаваемых в рамках парадигмы Data Driven Design (ориентированных на работу с предопределенными наборами данных) для Windows, Linux и Mac OS X.
- Embarcadero RadPHP упрощает создание веб-приложений на PHP благодаря наличию визуальных средств проектирования интерфейсов, редактора, отладчика, средств подключения к базам данных и интегрированной библиотеки повторно используемых классов компонентов. Компоненты RadPHP XE2 позволяют делать веб-интерфейсы в стиле iOS и Android.
- ER/Studio 8.5 Developer Edition (в RAD Studio Architect — полная лицензия, в RAD Studio Professional и Enterprise — лицензия на ознакомительную версию). ER/Studio помогает проектировщикам баз данных

анализировать, документировать и повторно использовать данные и предоставляет средства обратного проектирования, анализа и оптимизации баз данных.

- InterBase SMP 2009 Developer Edition предоставляет разработчикам кросс-платформенную базу данных для создания и тестирования приложений для встраиваемых приложений и приложений для малых и средних предприятий.

1.10. Методики разработки параллельных программ

OpenCL – открытый стандарт параллельного программирования для гетерогенных систем

OpenCL (от англ. Open Computing Language — открытый язык вычислений) — фреймворк для написания компьютерных программ, связанных с параллельными вычислениями на различных графических (англ. GPU) и центральных процессорах (англ. CPU), а также FPGA. Во фреймворк OpenCL входят язык программирования, который находится в стандарте C99, и интерфейс программирования приложений (англ. API). OpenCL обеспечивает параллелизм на уровне инструкций и на уровне данных и является реализацией техники GPGPU. OpenCL является полностью открытым стандартом, его применение не облагается лицензионными отчислениями.

OpenCL расширяет OpenGL и OpenAL. Они же, в свою очередь, считаются открытыми стандартами для трёхмерной компьютерной графики, а так же звука. OpenGL и OpenAL используют возможности GPU. OpenCL разрабатывается и поддерживается некоммерческим консорциумом Khronos Group. В консорциум входят такие компании, как Apple, AMD, Intel, Nvidia, ARM, Sun Microsystems, Sony Computer Entertainment и другие [15].

OpenCL предоставляет программисту низкоуровневый API, через который он взаимодействует с ресурсами устройства. OpenCL API может либо напрямую поддерживаться устройством, либо работать через промежуточный API (как в случае NVidia: OpenCL работает поверх CUDA Driver API, поддерживаемый устройствами), это зависит от конкретной реализации не описывается стандартом [20].

В OpenCL используется иерархия из четырех моделей, а именно:

- Модель платформы (Platform Model);
- Модель памяти (Memory Model);
- Модель исполнения (Execution Model);
- Программная модель (Programming Model).

CUDA – технология параллельного программирования для графических процессоров

CUDA – это архитектура параллельных вычислений от NVIDIA, позволяющая существенно увеличить вычислительную производительность благодаря использованию GPU (графических процессоров).

В настоящий период продажи CUDA процессоров достигли миллионов, а разработчики программного обеспечения, эксперты и ученые обширно применяют CUDA в разных сферах, в том числе обрабатывание видеоматериалов и изображений, вычислительную биологию и химию, прогнозирование динамики жидкостей, восстановление изображений, полученных посредством компьютерной томографии, микросейсмическое исследование, трассировку лучей и многое другое [17].

В отличии от предшествующих поколений GPU, в которых вычислительные ресурсы разделялись на вершинные и пиксельные шейдеры, в архитектуру CUDA интегрирован стандартизированный шейдерный конвейер, предоставляющий программе, выполняющей расчеты общего назначения, использовать любое арифметико-логическое устройство, в ходящее в микросхему. Так как NVIDIA полагала, что новейшее семейство графических процессоров станет применяться для вычислений общего

назначения, то арифметико-логическое устройства были сконструированы с учетом условий IEEE к арифметическим операциям над числами с плавающей точкой одинарной точности; помимо этого, был разработан комплект команд, направленный на вычисления общего назначения, а не только лишь на графику. В конечном итоге, исполняющим устройствам GPU был позволен случайный доступ к памяти с целью чтения и записи, а кроме того допуск к программно-управляемому кэшу, получившему название разделяемая память. Все без исключения средства были добавлены в архитектуру CUDA с целью создать GPU, который идеально справлялся бы с вычислениями общего назначения, а не только лишь с классическими вопросами компьютерной графики [10].

OpenACC

Стандарт OpenACC – это открытый стандарт для создания высокоуровневых гибридных (CPU + GPU) программ без явных инициализаций GPU и без явных передачи и синхронизации данных между CPU и GPU. На компиляторы взваливается главная деятельность, а именно распараллеливание программного обеспечения. Данный стандарт поддерживает разные ОС, CPU, GPU и компиляторы. Модель программирования OpenACC дает возможность программистам моментально и просто приступить к разработке параллельных программ на GPU. Стандарт OpenACC совместим с другими языками программирования на GPU и различными библиотеками: программы OpenACC могут взаимодействовать с CUDA C/Fortran- и GPU-библиотеками, например, cuFFT, cuBLAS, cuSPARSE, cuRAND...

Стандарт OpenACC содержит в себе комплект специальных директив и много runtime-функций. Директивы применяются с целью разметки параллельных областей. Данные области затем передаются автоматически компилятором в параллельный код для исполнения на GPU. Этот набор директив довольно гибкий, что дает возможность разработчику программного обеспечения осуществлять контроль распараллеливания на

GPU, добавляя при этом вспомогательные функции (clause) в имеющиеся в начальной программе директивы. Разработчик программного обеспечения с помощью таких директив способен специфицировать собственные данные – в каком месте они находятся, как выполняется передача данных между CPU и GPU, одновременно или нет, с какими исходными значениями инициализируются данные в памяти GPU.

При помощи директив разработчик программного обеспечения кроме того способен установить конкретное отображение определенных циклов в проекте на GPU и присоединить к программе разные функции оптимизации.

На официальном веб-сайте в 2011 г. была размещен полный стандарт OpenACC, а именно версии 1.0. Для разработчиков программного обеспечения, которые хотели применить данный стандарт в собственных научных проектах, было опубликовано краткое руководство пользователя, где коротко описан OpenACC и перечислены главные особенности OpenACC и сферы их использования. В данный момент существуют решения, поддерживающие OpenACC, от следующих ведущих компаний: Portland Group (PGI): Accelerator Compiler, CAPS Entreprise: HMPP Workbench, и CRAY: Compiling Environment (CRAY CE). 30-дневная лицензия на PGI Accelerator Compiler доступна всем зарегистрированным пользователям [18].

ГЛАВА 2. РАЗРАБОТКА ТЕОРЕТИЧЕСКИХ ОСНОВ ОПТИМИЗАЦИИ ГРАНИЦ КАРЬЕРОВ ПРИ ПОМОЩИ АЛГОРИТМА ПСЕВДОПОТОКА

2.1. Математическая постановка задачи.

Анализ современных методов оптимизации границ карьеров по добыче руд, выполненный в главе 1 и их сопоставление по вычислительной сложности показывает, что для поиска наиболее выгодных с экономической точки зрения оболочек карьеров наиболее эффективными являются алгоритмы теории графов, и, в частности, алгоритмы, основанные на нахождении максимальных потоков в ориентированном графе блочной модели месторождения. Ряд алгоритмов ориентированных графов и их обработка приведены в частности в [Axo, 2000]. Для регулярных блочных моделей карьеров общая схема может быть представлена в виде графа, показанного на рисунке 2.1.

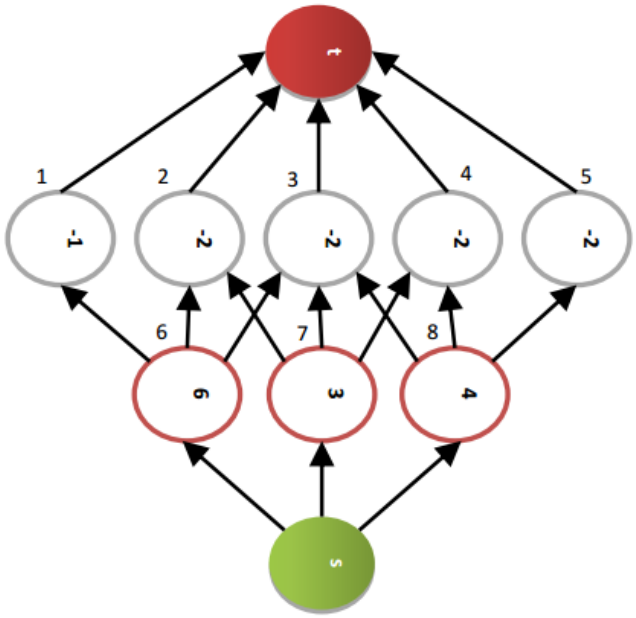


Рис. 2.1: Орграф сетевого потока от источника s к стоку t по регулярной решетке блоков

Схема структуры октодерева пространства месторождения, состоящего из однокомпонентной руды (черное) и пустой породы (белое), в виде октодерева представлена на рис. 2.2

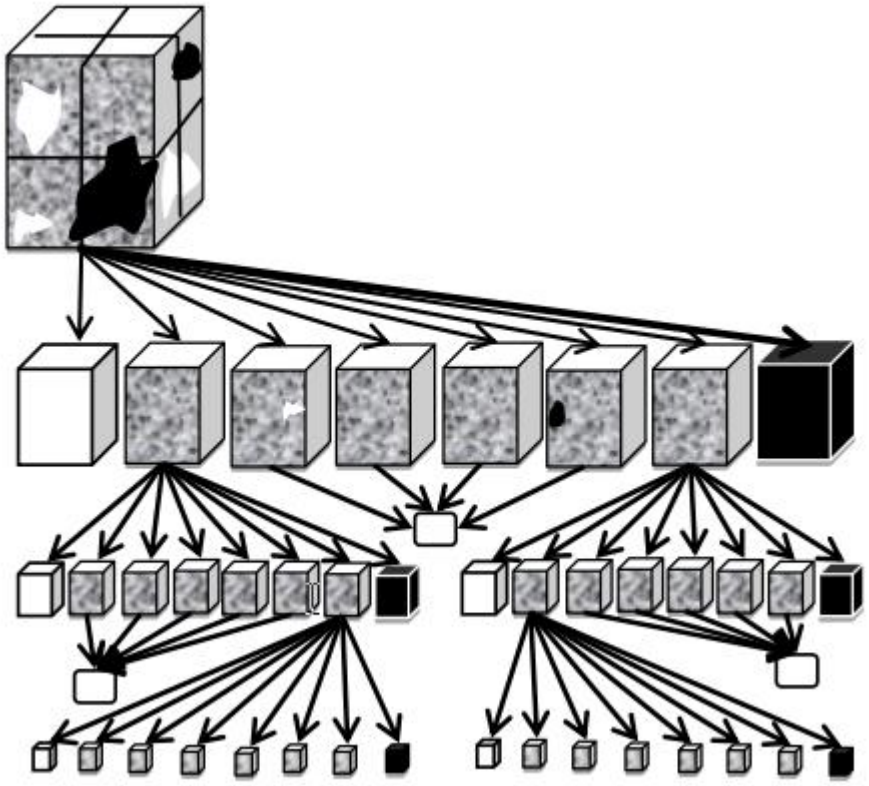


Рис 2.2: Структура октоэдрического дерева модели

Серые области данной модели на самом деле являются текстурой, состоящей из белых и черных тел (или «зерен»). Алгоритм формирования пирамидально рекурсивной структуры октодерева состоит в следующем.

Цвет каждого из *N* блоков может принимать значение 0 (w - "белое", порода), 1 (b -"черное", рудный минерал) и 2(g "серое", руда+порода). На нулевом уровне структуры исходный блок (суперблок) считается одним серым фрагментом со средним содержанием рудного компонента C*0*. На первом уровне объем суперблока нулевого уровня разбивается на *k* равных частей (для октодерева *k*=8) и мы получаем *k* фрагментов 1-го уровня

разбиения. На втором уровне число фрагментов равно *k2*. Каждому фрагменту сопоставляется среднее содержание рудного минерала, равное

$$C(i,j);\ i,j=0,k-1 \qquad (2.1)$$

Повторяя эту процедуру рекурсивно *t* раз мы получим фрагменты *t*-го разбиения, совпадающие с минимальной крупностью блока селективной добычи БСД со средним качеством $C^t(i,j);\ i,j=0,k^t-1$

Для всей структуры в целом формула имеет следующий вид:

$$C^t(i,j) = \frac{1}{k^2}\sum_{l=0}^{k-1}\sum_{n=0}^{k-1} C^t(ki+l, kj+n) \qquad (2.2)$$

Нетрудно показать, что качество каждого блока равно среднему качеству составляющих его подчиненных блоков (квантов) любого нижележащего уровня. В нашем случае качество блока есть доля полезной минеральной фазы в нем. Разбиение исходной полигональной модели (состоящей из полиэдров Вороного, оболочек рудных тел) с помощью пирамидально-рекурсивного алгоритма имитирует процесс сокращения крупности блоков (участков месторождения, кусков или частиц) и позволяет вычислить спектры распределения блоков каждого уровня по крупности по качеству.

Распределение средних содержаний блоков одного уровня разбиения представляет собой спектр раскрытия белых (породы) и черных (руды) блоков соответствующего класса крупности. Подобная усеченная структура представляется в виде несбалансированного октодерева. Все терминальные вершины этого дерева белые или черные, а нетерминальные - серые. С технологической точки зрения – построение оптимальной усеченной структуры октодерева эквивалентно тому, что при дроблении массива руды на каждой стадии фрагментируются лишь те куски (серые фрагменты) из данного класса крупности, которые состоят из двух или более фаз,

монофазные же фрагменты, (чистая пустая порода или чистый рудный минерал) далее не дробятся. Кроме того, это позволяет оценить число вершин различных "цветов" в октодереве и предсказать объемную долю руды, которую целесообразно измельчать дальше в каждом классе крупности для ОМГТ. Среднее число вершин различных типов определяется в зависимости от номера уровня *t* формулами:

$$w(t) = b(t) = k^t\left(k - \frac{1}{2}\right); g(t) = k^t; t = 1,2 \dots T \qquad (2.3)$$

где w(t),b(t),g(t) - соответственно число белых, черных и серых вершин на t – м уровне для простого сростка при ccp = 50% Число серых фрагментов пропорционально площадной доле сростков, появившихся в процессе сокращения крупности кусков руды после операции разделения качественного материала и пустой породы. До определенного уровня дробления t0 практически все вершины дерева остаются "серыми", а затем появляются полностью раскрытые блоки фаз.

При горнопромышленном освоении сложных месторождений твердых полезных ископаемых и решении задач управления запасами минерального сырья размеры каркасных и блочных геологических моделей могут достигать больших размеров. Создание совместных геолого-маркшейдерских моделей и систем управления качеством руд требует постоянного увеличения потоков обработки данных. В результате на время подсчета запасов и определение наилучших конфигураций выемки могут уходить многие часы работы быстродействующих компьютеров. В связи с этим становится актуальной проблема выбора структуры хранения компьютерной модели месторождения, а также – поиска наиболее эффективных методов оптимизации ведения горных работ.

Представление цифровой геологической модели в виде решетки блоков или массива прямоугольных ячеек позволяет использовать весь комплекс методов геостатистики и достаточно просто оценивать запасы.

Однако присутствие в массиве больших однородных областей, характеризуемых слабой анизотропией геопоказателей, никак не уменьшает объем моделей. Вместе с тем структура октодерева позволяет компактно хранить и с высокой скоростью обрабатывать большие массивы данных. Тем не менее, данная структура, также, как и регулярная матрица вокселей, не позволяет представить отдельно от вещественной составляющей поверхности пластов, линии складчатости и разломы.

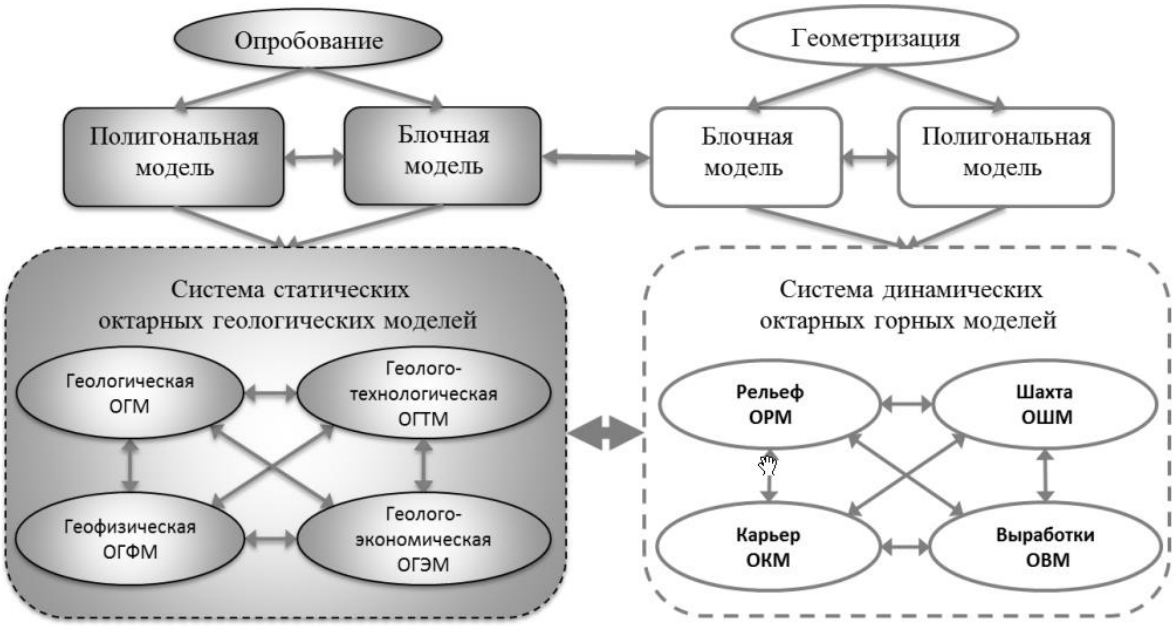


Рис. 2.3: Система блочных моделей со структурой октодерева для решения задач оптимизации в недропользовании

2.2. Модификация алгоритма псевдопотока для структуры данных в виде октодерева

Как отмечается в работе [Muir, 2008] нормализованные деревья алгоритма LG были адаптированы к более общей модели сетевого потока на основе концепции псевдопотока, по аналогии с предпотоками (preflow). Сетевой псевдопоток удовлетворяет ограничениям пропускной способности, но в нем могут нарушаться условия баланса потока с созданием дефицита или избытка в узлах. Предпоток удовлетворяет ограничениям пропускной

способности, но в нем может нарушаться баланс потока лишь путем создания избытка в узлах. Алгоритм псевдопотока решает задачу нахождения максимального потока в обобщенных сетях и работает с псевдопотоками вместо масс.

Связь между алгоритмом псевдопотока и алгоритмом предпотока (переразметки, push-relabel) более очевидна, чем между алгоритмом LG и алгоритмом предпотока. Методы LG и псевдопотока имеют дело с множеством узлов (ветвей), способных аккумулировать либо избыток, либо дефицит. В подходе на основе псевдопотока, массы M_r, поддерживаемые корневым узлом r_0 сильного дерева, трактуются как некий псевдопоток и выталкиваются в слабый корень r_s , а затем к фиктивному корневому узлу x_0 (одновременно являющемуся как источником, так и стоком). Алгоритм выталкивания с переразметкой работает с предпотоками (preflows). Алгоритм переразметки работает скорее с самими узлами, а не с множествами узлов, и избыток в некотором узле выдавливается в находящиеся ближе к стоку узлы в соответствие со значениями дистанционных меток, переразметка обновляет значения меток.

Алгоритм псевдопотока предоставляет несколько способов обработки вершин типа «слабый над-сильным». Лучшими из этих методов являются варианты с нижней и верхней метками [Hochbaum, 2008]. Для некоторого узла дистанционная метка представляет собой неубывающую функцию и на этом уровне она является неубывающей в составе любого сгенерированного дерева. В статье [Hochbaum, 2008] доказывается, что дистанционная метка есть неубывающая функция на заданном уровне дерева и для слабого узла она является некоторой нижней границей уровня (v). Функция дистанционной метки аналогична дистанционным меткам, введенным Голдбергом и меткам, используемым в методах сетевого потока, таких как метод переразметки [Ахо, 2000]. Так, метод Голдберга-Тарьяна дает оценку сложности для максимального потока. В методе же псевдопотока при первоначальной нормализации вложенного остовного дерева Tv, всем

сильным узлам блоков присваивается метка 2, а всем слабым узлам присваивается метка 1. Для эффективного управления сильными ветвями создается и поддерживается приоритетная очередь с индексом. Счетчик отслеживает количество сильных корневых узлов и индексированный список, указывающий на первый сильный корневой узел, для которых вводятся метки (первоначально все имеют индекс 0).

Модифицированный алгоритм псевдопотока на структуре октодерева выполняется в следующей последовательности:

Шаг 1. Определяется тип данных структуры блочной модели.

Шаг 2. Первоначально все положительные узлы считаются сильными и являются, соответственно, корнями своих ветвей. На этой стадии определяется индекс блока, его принадлежность к уровню и ветви дерева.

Шаг 3. Если после размер блока таков, что угол наклона борта превышает допустимый, то целый текущий блок разбивается на следующую восьмерку блоков меньшего размер, пока угол наклона борта не будет лежать в допустимых пределах.

Шаг 4. Выполняется расстановка меток. Всем таким узлам присваивается метка 2 и они помещаются в очередь. Порядок произвольный, поскольку все сильные узлы имеют пометку 2, хотя фактический порядок определяет последовательность обработки ветви. Указатель на первый узел с меткой 2 заносится в индексированный список.

Шаг 5. При выборе для обработки следующего сильного дерева, верх очереди (при упорядочивании либо сверху, либо снизу) выбирается и удаляется из очереди.

Шаг 6. После того, как процессы слияния и нормализации генерируют новую сильную ветвь, то она вставляется в очередь в верхнюю позицию для этой метки.

ГЛАВА 3. РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ДЛЯ РЕШЕНИЯ ЗАДАЧИ ОПТИМИЗАЦИИ ГРАНИЦ КАРЬЕРОВ

3.1. Разработка интерфейсов ввода-вывода

Для хранения данных о блочной модели месторождения используется текстовый формат определенной структуры. Блочные модели *MINELIB* описываются несколькими файлами. Нас интересуют, в частности, файлы *blocks, upit* и *prec* – они потребуются, чтобы выстроить ориентированный граф на основе блочной модели.

Файл *blocks* содержит основную информацию о модели карьера. Он требуется, чтобы в дальнейшем имелась возможность выполнить визуализацию модели и проведенных вычислений. Каждая строка файла соотносится с одним конкретным блоком модели. Все линии имеют одинаковое число столбцов. Столбцы организованы следующим образом:

<int id><int x><int y><int z><str$_1$>…<str$_k$>

- *int id* – уникальный id конкретного блока модели;
- *x, y, z* – координаты x,y,z конкретного блока, где *z=0* – самый нижний слой блочной модели. Ось *y* направлена по направлению взгляда наблюдателя, а ось *x* – влево от наблюдателя.
- *str1…str$_k$* – необязательные дополнительные пользовательские поля, которые могут содержать информацию о тоннаже, ценности или каких-либо иных параметрах блока. Такая гибкость используется для того, чтобы избежать каких-либо ограничений, связанных с одной конкретной блочной моделью.

```
0 0 0 0 61200.012 0 0 -4
1 0 0 1 61200.012 0 0 -4
2 0 0 2 61200.012 0 0 -4
3 0 0 3 61200.012 0 0 -4
4 0 0 4 61200.012 0 0 -4
5 0 0 5 61200.012 0 0 -4
6 0 0 6 61200.012 0 0 -4
7 0 0 7 61200.012 0 0 -4
8 0 0 8 61200.012 0 0 -4
9 0 0 9 61200.012 0 0 -4
10 0 0 10 61200.012 0 0 -4
11 0 0 11 61200.012 0 0 -4
12 0 0 12 61200.012 0 0 -4
13 0 0 13 52503.559 0 0 -4
14 0 1 0 61200.012 0 0 -4
15 0 1 1 61200.012 0 0 -4
```

Рис 3.1: Пример файла *blocks*

Файл **prec** (англ. *precedence* - приоритет) содержит информацию об отношениях между блоками в модели. Этот файл нужен, чтобы определить связи между узлами ориентированного графа. Каждая строка файла содержит информацию для одного блока. Приоритетные отношения блоков модели описываются следующим образом:

<*int b*> <*int n*> <*int* p_1>…<*int* p_n>

- переменная b хранит уникальный идентификатор блока (аналогично *int id* в файле **blocks**);
- переменная *n* хранит количество односторонних связей блока *b* (в рамках логики блочной модели – информацию о том, какое количество блоков нужно «вынуть», перед тем как вынуть блок *b*);
- переменные $p_1…p_n$ хранят уникальные идентификаторы блоков, связанных с блоком *b*.

Как правило, мы принимаем что $p_1…p_n$ – прямые предшественники блока *b*, однако это не обязательное требование. Также мы предполагаем, что два блока не могут иметь одинаковый идентификатор. Если же конкретный блок *b* не имеет блоков-предшественников, то значение *n* для него будет равняться нулю, и значений $p_1…p_n$ для него нет.

```
0 6 1 891 15 1815 3630 2746
1 6 2 892 16 1816 3631 2747
2 6 3 893 17 1817 3632 2748
3 6 4 894 18 1818 3633 2749
4 6 5 895 19 1819 3634 2750
5 6 6 896 20 1820 3635 2751
6 6 7 897 21 1821 3636 2752
7 6 8 898 22 1822 3637 2753
8 6 9 899 23 1823 3638 2754
9 6 10 900 24 1824 3639 2755
10 4 11 901 25 1825
11 4 12 902 26 1826
12 3 13 903 27
13 0
```

Рис 3.2: Пример файла *prec*

Файл *UPIT* (англ. Ultimate Pit – предельный карьер) содержит информацию о прибыли, которую возможно получить из блоков карьера. Всего в нём два столбца: уникальный идентификатор блока и прибыль, которую возможно получить из этого блока. Данный файл требуется, чтобы иметь возможность задать не только дуги между узлами выстраиваемого орграфа, но и задать для этих дуг емкость (capacity).

```
NAME: marvin
TYPE: UPIT
NBLOCKS: 53271
OBJECTIVE_FUNCTION:
0 -55080.0108
1 -55080.0108
2 -55080.0108
3 -55080.0108
4 -55080.0108
5 -55080.0108
6 -55080.0108
7 -55080.0108
8 -55080.0108
9 -55080.0108
10 -55080.0108
11 -55080.0108
12 -55080.0108
13 -47253.2031
```

Рис 3.3: Пример файла UPIT

Эти три файла библиотеки *MINELIB* позволяют нам выстроить для конкретной модели ориентированный граф, определить дуги этого графа и их емкость. Получив эти сведения, мы можем создать файл максимального

формата DIMACS, который будет содержать всю информацию о графе, а следовательно – и о рассчитываемой модели.

Структура должна быть следующей:

- исследуемая сеть – это ориентированный граф с количеством узлов *n* и количеством дуг *m;*
- узлы идентифицируются целыми числами (*1…n*);
- на графе не должно быть параллельных дуг;
- на графе не должно быть петель;
- емкости дуг – 32-битные целые числа;
- источник и сток графа разделены;
- сток может быть недоступен из источника.

Файл формата DIMACS для максимального потока состоит из строк с комментариями, строк с описанием проблемы и строк с описанием дуг и узлов.

Строки комментариев предоставляют информацию непосредственно для людей, в них можно помещать какие-либо сообщения или определения – эти строки будут игнорироваться программой. Строка с комментарием начинается с символа *c*. Пример: *c This is an example of a comment line*.

Строка с описанием проблемы может быть всего одна в одном файле. В ней содержится информация о количестве узлов и количестве дуг в ориентированном графе. Строка начинается с символа *p*. Пример: *p max 6 8* (граф, в котором 6 узлов и 8 дуг).

Строки с описанием узлов указываются строго после строки с описанием кол-ва узлов и дуг. Эти строки указывают, какой из узлов является источником, а какой – стоком. Начинается строка с символа *n*. Пример такой строки: *n 1 s* (узел 1 является источником) *n 2 t* (узел 2 является стоком).

Строки с описанием дуг указываются после строк описания узлов. Они указывают, от какого блока к какому следует дуга, а также – емкость этой дуги. Строка начинается с символа *a*, после нее указывается три числа –

идентификатор узла-источника, идентификатор узла-приемника и емкость самой дуги. Пример такой строки: *a 1 2 5* (дуга с емкостью 5 связывает узел 1 т узел 2).

```
c This is a simple example file to demonstrate the DIMACS
c input file format for maximum flow problems. The solution
c vector is [5,10,5,0,5,5,10,5] with cost at 15.
c Problem line (nodes, links)
p max 6 8
c source
n 1 s
c sink
n 6 t
c Arc descriptor lines (from, to, capacity)
a 1 2 5
a 1 3 15
a 2 4 5
a 2 5 5
a 3 4 5
a 3 5 5
a 4 6 15
a 5 6 5
c
c End of file
```

Рис 3.4: Пример файла DIMACS для поиска максимального потока в сети.

На рисунке 3.4 изображен dimacs-файл описывающий небольшой ориентированный граф который состоит из 6 узлов, узел 1 является источником, узел 6 является стоком.

3.2 Реализация алгоритма псевдопотока

Для того чтобы иметь возможность рассчитать предельный карьер в блочной модели методом, использующим максимального псевдопотока, следует конвертировать блочную модель в файл DIMACS, который описывает орграф. Для этого потребуется извлечь информацию из файлов **prec** и **upit** – для определения дуг между блоками и для определения емкости этих дуг соответственно.

В дальнейшем программа использует для вычислений модифицированную версию реализации алгоритма псевдопотока, предоставленную Dorit Hochbaum (см. Приложение).

Циклы for, используемые в функции простой инициализации (simpleInitialization) и функции слияния (merge) в коде программы были заменены на обращение к class function библиотеки PPL для повышения быстродействия программы. Кроме того, для исключения вероятных конфликтов в исполнении многопоточных вычислений, стандартное приращение i++ было заменено командой *TInterlocked->Increment*.

Как было уже сказано выше, при использовании стандартного алгоритма *pseudoflow* разработанного Дорит Хочбаум, на каждом этапе выполнения имеются несколько переменных, связанных с каждой вершиной или ребром нормализованного дерева. После стандартной инициализации, когда все ребра, примыкающие к источнику, насыщаются – запускается серия итераций из следующих шагов:

1. Каждая из задействованных активных вершин, обрабатывается параллельно. Для каждой из вершин арки проверяются последовательно. Возможные выталкивания (pushes) масс M_r, поддерживаемых корневым узлом r_0 сильного дерева в слабый корень r_s, но изменения применяются только к копии старых избыточных значений (окончательные значения будут скопированы на шаге 4);

2. Новые временные метки вычисляются параллельно для вершин, которые были обработаны в ходе шага 1, но еще являются активными;

3. Новые метки применяются при повторной итерации ко множеству активных вершин параллельно, устанавливая значения, вычисленные в ходе шага 2;

4. Изменения избыточных значений, полученные на шаге первом, применяются к новому полученному набору активных вершин параллельно.

Данные шаги повторяются, пока не останется активных вершин с меткой, меньшей чем N. Алгоритм является детерминированным в том

смысле, что он требует такого же объема работы независимо от количества потоков, что является явным преимуществом по сравнению с другими параллельными подходами, которые демонстрируют значительное увеличение работы при добавлении большего количества потоков.

Результат работы алгоритма записывается в текстовый файл для последующей визуализации. Таким образом, список «помеченных» узлов может быть использован в системе Geoblock для определения вынимаемых блоков при построении модели предельного карьера.

ГЛАВА 4. ТЕСТИРОВАНИЕ ПРОГРАММЫ ДЛЯ РЕШЕНИЯ ЗАДАЧИ ОПТИМИЗАЦИИ ГРАНИЦ КАРЬЕРОВ

4.1. Проведение тестирования работы программы

Для проведения тестирования разработанного параллельного алгоритма максимального псевдопотока использовались блочные модели, представленные в библиотеке Minelib (marvin, kd, newman). Также, помимо блочных моделей, в качестве тестовых файлов использовались уже конвертированные в DIMACS DTF-графы (Decision Tree Field), которые, хоть и не являются графами, основанными на блочных моделях, но позволяют оценить быстродействие разработанного алгоритма ввиду схожести решаемых задач.

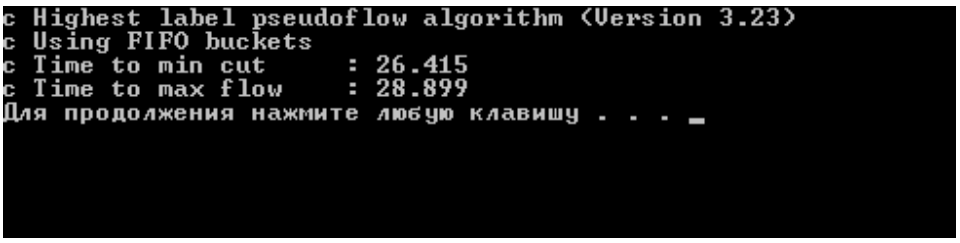


Рис 4.1: Результат работы последовательного алгоритма для модели с кол-вом блоков 994840

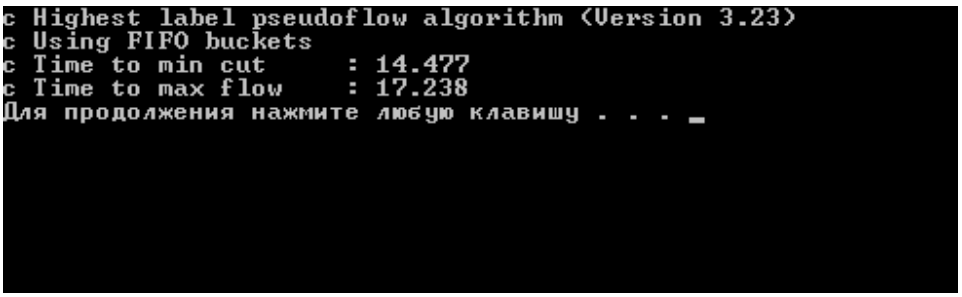


Рис 4.2: Результат работы параллельного алгоритма для модели с кол-вом блоков 994840

На данных рисунках можно увидеть, что для одной и той же модели время вычисления отличается для последовательной и параллельной реализаций. Что характерно, в иных ситуациях распараллеливание алгоритма, если оно применено корректно, дает значительно больший прирост производительности. В нашей же ситуации, небольшой прирост производительности обусловлен особенностями алгоритма, поскольку некоторые циклы его работы не представляется возможным распараллелить выбранными средствами.

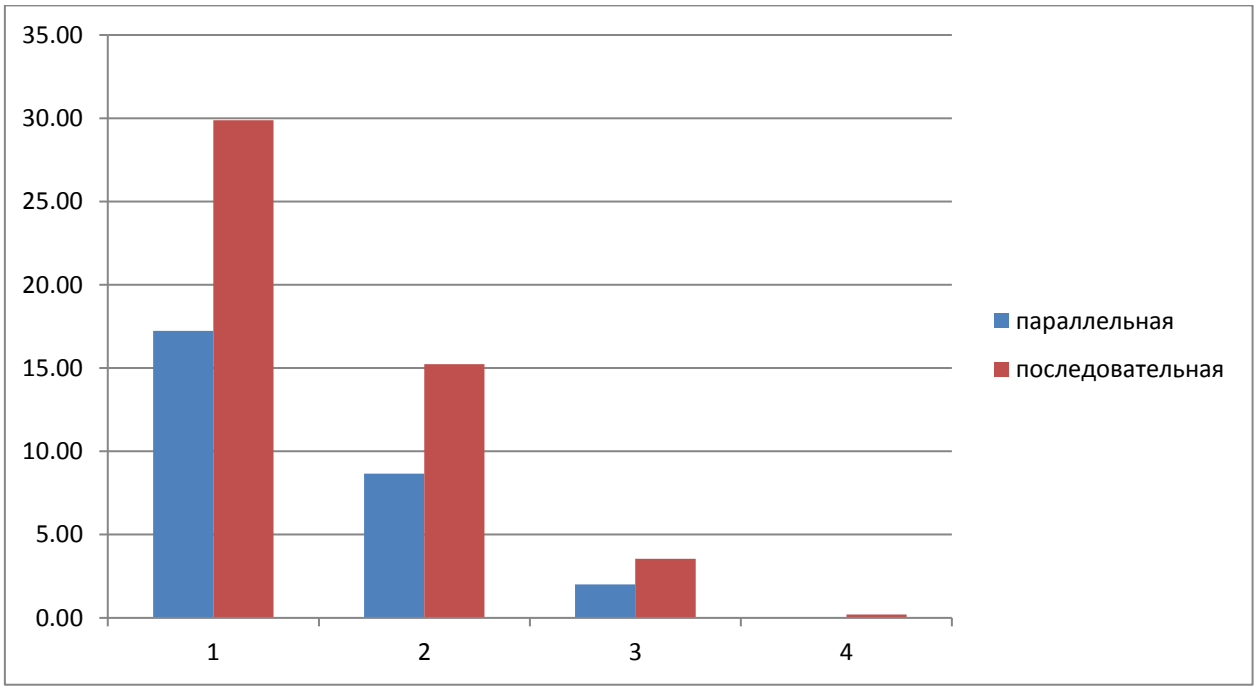


Рис.4.3. Гистограмма для наглядного сравнения времени, затраченного на выполнение последовательного и параллельного алгоритмов

Тем не менее, как видно из гистограммы на рисунке 4.3, такая разница во времени исполнения версий алгоритма вполне ощутима. Параллельная реализация, будучи почти в два раза более быстрой в вычислениях, может позволить практически мгновенно вычислять небольшие блочные модели и значительно ускорить вычисление гигаблочных и тераблочных моделей.

ЗАКЛЮЧЕНИЕ

При выполнении научно-исследовательской работы были выполнены следующие задачи:

1) было сформулировано определение открытого способа разработки месторождения;

2) описан процесс проектирования открытого карьера по добыче сырья;

3) рассмотрена задача определения границ предельного карьера;

4) исследованы различные алгоритмы оптимизации границ открытого карьера;

5) выбранный алгоритм максимального псевдопотока был модифицирован для использования с блочными моделями;

6) было разработано и реализовано ПО, вычисляющее контур предельного карьера блочной модели на основе выбранного метода.

7) Произведена оценка эффективности разработанной системы.

Алгоритм нахождения максимального псевдопотока найден наиболее быстрым из представленных, вследствие чего поставленные задачи по проекту заключаются в том, чтобы найти способ каким-либо образом уменьшить время выполнения данного алгоритма для крупных моделей. В качестве основного способа ускорения используется применение параллельного программирования (библиотек PPL в составе среды разработки Embarcadero RAD Studio) и доработка имеющихся алгоритмов с применением данных средств.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Lerchs H., Grossman I.F. Optimum design of open pit mines/H. Lerchs // Canadian Mining and Metallurgical Bulletin. 1965. - Vol.58. - № 633. - P. 47 – 56

2. Бобровский А.Н. Qt4.7+. Практическое программирование на C++. – СПб.: БХВ-Петербург, 2012. – 496 с.

3. Гергель В.П. Современные языки и технологии параллельного прогаммирования. - Издательство Московского Университета, 2012. -408 с.

4. Гладков Л.А., Курейчик В.В., Курейчик В.М. – Генетические алгоритмы– 2-е изд., исправл. и доп. – М.: ФИЗМАТЛИТ, 2010. – 310 с.

5. Капутин Ю.Е. Информационные технологии планирования горных работ - СПб., 2004.-334 с.

6. Малышкин В.Э., Корнеев В.Д. Параллельное программирование мультикомпьютеров. Изд-во НГТУ, 2011.

7. Немнюгин С.А., Стесик О.Л. Параллельное программирование для многопроцессорных вычислительных систем, БХВ-Петербург, 2002, 400 стр.

8. Рейзлин В.И. Численные методы оптимизации: учебное пособие – Томск: Изд-во Томского политехнического университета, 2011. – 105 с.

9. Рутковская Д., Пилиньский М., Рутковский Л. – Нейронные сети, генетические алгоритмы и нечеткие системы, 452, Горячая Линия – Телеком, 2007.

10. Сандерс Дж., Кэндрот Э. Технология CUDA в примерах: введение в програмирование графических процессоров: Перевод с английского Сликина А.А., научный редактор Боресков А.В. – М.: ДМК Пресс, 2011. -232 с.

11. Сухарев А.Г., Тимохов А.В., Федоров В.В. Курс методов оптимизации: Учебное пособие. – 2-е издание. – М.:ФИЗМАЛИТ, 2005. – 368 с.

12. Intel® Parallel Studio – инструмент для создания параллельных приложений / [Электронный ресурс] / Режим доступа: https://software.intel.com/ru-ru/articles/intel-parallel-studio-parallelism-toolkit

13. Steps3D – Графика, ООП / [Электронный ресурс] / Режим доступа: http://steps3d.narod.ru/tutorials/cuda-tutorial.html

14. Википедия — свободная энциклопедия / [Электронный ресурс] / Режим доступа: https://ru.wikipedia.org/wiki/Microsoft_Visual_Studio

15. Википедия — свободная энциклопедия/ [Электронный ресурс] / Режим доступа: https://ru.wikipedia.org/wiki/OpenCL

16. Национальный открытый университет «Интуит» / [Электронный ресурс] / Режим доступа: http://www.intuit.ru/studies/courses/3735/977/lecture/14689?page=5

17. Параллельные вычисления CUDA / [Электронный ресурс] / Режим доступа: http://www.nvidia.ru/object/cuda-parallel-computing-ru.html

18. Сибирский суперкомпьютерный центр СО РАН / [Электронный ресурс] / Режим доступа: http://www2.sscc.ru/SORAN-INTEL/paper/2012/Open%20ACC.PDF

19. Факультет информационных технологий и прикладной математики / [Электронный ресурс] / Режим доступа: http://knit.bsu.edu.ru/knit/resources/docs.php?ID=000

20. Хабрахабр – интересные публикации / [Электронный ресурс] / Режим доступа: http://habrahabr.ru/post/72650/

21. Minelib 2011: A library of open pit production scheduling problems. Espinoza D, Goycoolea M, Moreno E, Newman A. 206(l), 2013, Ann. Oper. Res., pp. 93-114.

22. Ху Т. Целочисленное программирование и потоки в сетях. Москва : Мир, 1974

23. The Pseudoflow Algorithm: A New Algorithm for the Maximum-Flow Problem. Hochbamn D.S. 56, 2008, Oparations Research, Vol. 4, pp. 992-1009.

24. Астафьев Ю.П., Зеленский А.С., Горлов Н.И. и др. Компьютеры и системы управления в горном деле за рубежом. М.: Недра, 1989. стр. 264.

25. Система многоуровневых октарных моделей горно-геологических объектов. Васильев П.В. 2012 г., Материалы XII Международной конференции Информатика. (XII International Conference "Informatics", 20-22 February 2012) 20-22 February 2012) в 2т, стр. 28-32

26. Петров Д.В. Применение методов глобальной оптимизации для поиска предельных границ рудных месторождений. Научные ведомости БелГУ. Серия Экономика. Информатика, 2015. №7 (204) Выпуск 34/1

27. Михелев В.М., Васильев П.В., Петров Д.В. «Суперкомпьютеры, как средства моделирования граничных контуров карьеров рудных месторождений», Вопросы радиоэлектроники. Серия "Электронная вычислительная техника (ЭВТ)" Выпуск 1, Москва 2013., с. 5-10

28. Muir D.C.W. 2008. Pseudoflow, New Life for Lerchs-Grossmann Pit Optimisation. Spectrum Series, Orebody Modelling and Strategic Mine Planning, 14: 97-104.

29. Ахо А.И., Хопкрофт Д., Ульман Д.Д. 2003. Структуры данных и алгоритмы.: Перв. с англ., М., Издательский дом "Вильямс", 384 (Alfred V. Aho, John E. Hopkroft, Jeffrey D. Ullman 2000. Data Structures and Algorithms, Addison-Wesley publishing company, London: 386)

30. Черепанов, Е.В., Макаров, И.В. Фисенко, А.И. Оценка экономической эффективности вовлечения в разработку площадей с прогнозными ресурсами категории Р1, с применением горно-геологических информационных систем [Электронный ресурс] / Сибирский федеральный университет. – Режим доступа: *conf.sfu-kras.ru/sites/mn2010/pdf/4/80.pdf.*

31. Валуев, А.М. О моделях и методах оптимизации в задачах проектирования разработки месторождений открытым способом

[Электронный ресурс] / Горный информационно-аналитический бюллетень. – Режим доступа: http://www.giab-online.ru/files/Data/2015/02/29_197-206_Valuev.pdf.

32. Элкингтон, Т., Дурхэм, Р. Объединение задач определения размера приконтурных блоков и оптимизации производственной мощности карьера [Текст] / Т. Элкингтон, Р. Дурхэм // Физико-технические проблемы разработки полезных ископаемых. – 2011. – № 2. – С. 41-56.

33. Solving ultimate pit limit problem through graph closure (L-G Algorithm) and the fundamental tree algorithm [Электронный ресурс] / SpotiDoc. – Режим доступа: http://www.spotidoc.com/doc/725786

34. Ческидов, В.И., Саканцев, Г.Г., Саканцев, М.Г. Комплексное обоснование границ карьеров и способов вскрытия глубоких горизонтов при разработке крутопадающих пластообразных залежей [Текст] / В.И. Ческидов, Г.Г. Саканцев, М.Г. Саканцев // Известия высших учебных заведений. Горный журнал. – 2015. – № 2. – С. 17-23.

35. Шариф, Д.А. Регулярная проверка конечного контура рабочей зоны карьера на основе алгоритма скользящего конуса [Текст] / Шариф Д.А. // Горный информационно-аналитический бюллетень (научно-технический журнал). – 2007. – № 6. – С. 352-356.

36. Петров Д.В., Михелев В.М. Высокопроизводительные алгоритмы решения задачи поиска предельных границ открытых карьеров [Текст] / Д.В. Петров, В.М. Михелев // В сборнике: Актуальные проблемы вычислительной и прикладной математики труды Международной конференции, посвященной 90-летию со дня рождения академика Г. И. Марчука. – 2015. С. 580-584.

37. Петров Д.В., Михелев В.М. Моделирование карьеров рудных месторождений на высокопроизводительных гибридных вычислительных системах [Текст] / Д.В. Петров, В.М. Михелев // В сборнике: Параллельные вычислительные технологии (ПАВТ'2014) Труды международной научной конференции. Ответственные за выпуск: Л.Б. Соколинский, К.С. Пан. – 2014. С. 299-302.

38. Васильев П.В., Михелев В.М., Петров Д.В. Параллельные алгоритмы оптимизации границ карьеров по методу псевдопотока на модели данных со структурой октодерева [Текст] / П.В. Васильев, В.М. Михелев, Д.В. Петров // Научные ведомости Белгородского государственного университета. Серия: Экономика. Информатика. – 2016. том. 38. – № 9 (230). С. 123-128.

39. Петров, Д.В., Дроник, В.И., Михелев, В.М. Реализация алгоритма Лерча-Гроссмана для поиска предельных границ карьеров рудных месторождений [Текст] / Д.В. Петров, В.И. Дроник, В.М. Михелев // Информатика: проблемы, методология, технологиим сборник материалов XVII международной научно-методической конференции: в 5 т. Секция 6. – 2017. – С. 68-72.

40. Петров Д.В., Михелев В.М. Решение задачи оптимизации блочных моделей при проектировании открытых горных работ с использованием гибридных вычислительных систем [Текст] / Д.В. Петров, В.М. Михелев // Научные ведомости Белгородского государственного университета. Серия: Экономика. Информатика. том. 35. – № 13-1 (210). – 2015. – С. 93-98.

ПРИЛОЖЕНИЕ

```
#pragma hdrstop
#pragma argsused
#ifdef _WIN32
#include <tchar.h>
#else
  typedef char _TCHAR;
  #define _tmain main
#endif
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
        #define PROGRESS
        #define STATS
typedef long long int uint;
typedef long int lint;
typedef long long int llint;
typedef unsigned long long int ullint;
double
timer (void)
{
        clock_t cl = clock( );
        if ( cl != (clock_t)-1 )
                return (double)cl / (double)CLOCKS_PER_SEC;
}
struct node;
typedef struct arc
{
        struct node *from;
        struct node *to;
        uint flow;
        uint capacity;
        uint direction;
} Arc;

typedef struct node
{
        uint visited;
```

```
        uint numAdjacent;
        uint number;
        uint label;
        int excess;
        struct node *parent;
        struct node *childList;
        struct node *nextScan;
        uint numOutOfTree;
        Arc **outOfTree;
        uint nextArc;
        Arc *arcToParent;
        struct node *next;
} Node;
typedef struct root
{
        Node *start;
        Node *end;
} Root;

//---------------  Global variables ------------------
static uint numNodes = 0;
static uint numArcs = 0;
static uint source = 0;
static uint sink = 0;

#ifdef LOWEST_LABEL
static uint lowestStrongLabel = 1;
#else
static uint highestStrongLabel = 1;
#endif
static Node *adjacencyList = NULL;
static Root *strongRoots = NULL;
static uint *labelCount = NULL;
static Arc *arcList = NULL;
//-----------------------------------------------------
#ifdef STATS
static ullint numPushes = 0;
static uint numMergers = 0;
static uint numRelabels = 0;
static uint numGaps = 0;
static ullint numArcScans = 0;
#endif
```

```
#ifdef DISPLAY_CUT
static void
displayCut (const uint gap)
{
        uint i;
        printf("c\nc Nodes in source set of min s-t cut:\n");
        for (i=0; i<numNodes; ++i)
        {
                if (adjacencyList[i].label >= gap)
                {
                        printf("n %d\n", adjacencyList[i].number);
                }
        }
}
#endif
#ifdef DISPLAY_FLOW
static void
displayFlow (void)
{
        uint i;
        printf("c\nc Flow values on each arc:\n");
        for (i=0; i<numArcs; ++i)
        {
                printf("a %d %d %d\n", arcList[i].from->number, arcList[i].to->number, arcList[i].flow);
        }
}
#endif
static void
initializeNode (Node *nd, const uint n)
{
        nd->label = 0;
        nd->excess = 0;
        nd->parent = NULL;
        nd->childList = NULL;
        nd->nextScan = NULL;
        nd->nextArc = 0;
        nd->numOutOfTree = 0;
        nd->arcToParent = NULL;
        nd->next = NULL;
        nd->visited = 0;
        nd->numAdjacent = 0;
        nd->number = n;
```

```
        nd->outOfTree = NULL;
}
static void
initializeRoot (Root *rt)
{
        rt->start = NULL;
        rt->end = NULL;
}
static void
freeRoot (Root *rt)
{
        rt->start = NULL;
        rt->end = NULL;
}

#ifndef LOWEST_LABEL
static void
liftAll (Node *rootNode)
{
        Node *temp, *current=rootNode;
        current->nextScan = current->childList;
        -- labelCount[current->label];
        current->label = numNodes;
        for ( ; (current); current = current->parent)
        {
                while (current->nextScan)
                {
                        temp = current->nextScan;
                        current->nextScan = current->nextScan->next;
                        current = temp;
                        current->nextScan = current->childList;

                        -- labelCount[current->label];
                        current->label = numNodes;
                }
        }
}
#endif
#ifdef FIFO_BUCKET
static void
addToStrongBucket (Node *newRoot, Root *rootBucket)
{
```

```
		if (rootBucket->start)
		{
			rootBucket->end->next = newRoot;
			rootBucket->end = newRoot;
			newRoot->next = NULL;
		}
		else
		{
			rootBucket->start = newRoot;
			rootBucket->end = newRoot;
			newRoot->next = NULL;
		}
	}
	#else
	static void
	addToStrongBucket (Node *newRoot, Root *rootBucket)
	{
		newRoot->next = rootBucket->start;
		rootBucket->start = newRoot;
	}
	#endif
	static void
	createOutOfTree (Node *nd)
	{
		if (nd->numAdjacent)
		{
			if ((nd->outOfTree = (Arc **) malloc (nd->numAdjacent * sizeof (Arc *))) == NULL)
			{
				printf ("%s Line %d: Out of memory\n", __FILE__, __LINE__);
				exit (1);
			}
		}
	}

	static void
	initializeArc (Arc *ac)
	{
		ac->from = NULL;
		ac->to = NULL;
		ac->capacity = 0;
		ac->flow = 0;
		ac->direction = 1;
```

```
}
static void
addOutOfTreeNode (Node *n, Arc *out)
{
        n->outOfTree[n->numOutOfTree] = out;
        ++ n->numOutOfTree;
}
static void
readDimacsFileCreateList (char *fullpath)
{
        FILE *file;
        char *fname = fullpath;
        uint capacity;
        uint lineLength=1024, i,  numLines = 0, from, to, first=0, last=0;
        char *line, *word, ch, ch1;
        file = fopen(fname,"rt");
        if ((line = (char *) malloc ((lineLength+1) * sizeof (char))) == NULL)
        {
                printf ("%s, %d: Could not allocate memory.\n", __FILE__, __LINE__);
                exit (1);
        }
        if ((word = (char *) malloc ((lineLength+1) * sizeof (char))) == NULL)
        {
                printf ("%s, %d: Could not allocate memory.\n", __FILE__, __LINE__);
                exit (1);
        }
        while (fgets (line, lineLength, file))
        {
                ++ numLines;
                switch (*line)
                {
                case 'p':
                        sscanf (line, "%c %s %d %d", &ch, word, &numNodes, &numArcs);

                        if ((adjacencyList = (Node *) malloc (numNodes * sizeof (Node))) == NULL)
                        {
                        printf ("%s, %d: Could not allocate memory.\n", __FILE__, __LINE__);
                                exit (1);
                        }
                        if ((strongRoots = (Root *) malloc (numNodes * sizeof (Root))) == NULL)
                        {
                        printf ("%s, %d: Could not allocate memory.\n", __FILE__, __LINE__);
```

```
                exit (1);
        }
        if ((labelCount = (uint *) malloc (numNodes * sizeof (uint))) == NULL)
        {
        printf ("%s, %d: Could not allocate memory.\n", __FILE__, __LINE__);
                exit (1);
        }
        if ((arcList = (Arc *) malloc (numArcs * sizeof (Arc))) == NULL)
        {
        printf ("%s, %d: Could not allocate memory.\n", __FILE__, __LINE__);
                exit (1);
        }
        for (i=0; i<numNodes; ++i)
        {
                initializeRoot (&strongRoots[i]);
                initializeNode (&adjacencyList[i], (i+1));
                labelCount[i] = 0;
        }

        for (i=0; i<numArcs; ++i)
        {
                initializeArc (&arcList[i]);
        }

        first = 0;
        last = numArcs-1;

        break;

case 'a':
        sscanf (line, "%c %d %d %d", &ch, &from, &to, &capacity);
        if ((from+to) % 2)
        {
                arcList[first].from = &adjacencyList[from-1];
                arcList[first].to = &adjacencyList[to-1];
                arcList[first].capacity = capacity;
                ++ first;
        }
        else
        {
                arcList[last].from = &adjacencyList[from-1];
                arcList[last].to = &adjacencyList[to-1];
```

```
                        arcList[last].capacity = capacity;
                        -- last;
                }
                ++ adjacencyList[from-1].numAdjacent;
                ++ adjacencyList[to-1].numAdjacent;
                break;
        case 'n':
                sscanf (line, "%c %d %c", &ch, &i, &ch1);
                if (ch1 == 's')
                {
                        source = i;
                }
                else if (ch1 == 't')
                {
                        sink = i;
                }
                else
                {
                        printf ("Unrecognized character %c on line %d\n", ch1, numLines);
                        exit (1);
                }
                break;
        }
}
for (i=0; i<numNodes; ++i)
{
        createOutOfTree (&adjacencyList[i]);
}

for (i=0; i<numArcs; i++)
{
        to = arcList[i].to->number;
        from = arcList[i].from->number;
        capacity = arcList[i].capacity;
        if (!((source == to) || (sink == from) || (from == to)))
        {
                if ((source == from) && (to == sink))
                {
                        arcList[i].flow = capacity;
                }
                else if (from == source)
                {
```

```
                                        addOutOfTreeNode (&adjacencyList[from-1], &arcList[i]);
                                }
                                else if (to == sink)
                                {
                                        addOutOfTreeNode (&adjacencyList[to-1], &arcList[i]);
                                }
                                else
                                {
                                        addOutOfTreeNode (&adjacencyList[from-1], &arcList[i]);
                                }
                        }
                }
                free (line);
                line = NULL;
                free (word);
                word = NULL;
        }

        static void
        simpleInitialization (void)
        {
                uint i, size;
                Arc *tempArc;
                size = adjacencyList[source-1].numOutOfTree;
                for (i=0; i<size; ++i)
                {
                        tempArc = adjacencyList[source-1].outOfTree[i];
                        tempArc->flow = tempArc->capacity;
                        tempArc->to->excess += tempArc->capacity;
                }
                size = adjacencyList[sink-1].numOutOfTree;
                for (i=0; i<size; ++i)
                {
                        tempArc = adjacencyList[sink-1].outOfTree[i];
                        tempArc->flow = tempArc->capacity;
                        tempArc->from->excess -= tempArc->capacity;
                }
                adjacencyList[source-1].excess = 0;
                adjacencyList[sink-1].excess = 0;
                for (i=0; i<numNodes; ++i)
                {
                        if (adjacencyList[i].excess > 0)
```

```
                {
                    adjacencyList[i].label = 1;
                        ++ labelCount[1];

                        addToStrongBucket (&adjacencyList[i], &strongRoots[1]);
                }
        }
        adjacencyList[source-1].label = numNodes;
        adjacencyList[sink-1].label = 0;
        labelCount[0] = (numNodes - 2) - labelCount[1];
}
static inline int
addRelationship (Node *newParent, Node *child)
{
        child->parent = newParent;
        child->next = newParent->childList;
        newParent->childList = child;

        return 0;
}
static inline void
breakRelationship (Node *oldParent, Node *child)
{
        Node *current;
        child->parent = NULL;

        if (oldParent->childList == child)
        {
                oldParent->childList = child->next;
                child->next = NULL;
                return;
        }
        for (current = oldParent->childList; (current->next != child); current = current->next);
        current->next = child->next;
        child->next = NULL;
}
static void
merge (Node *parent, Node *child, Arc *newArc)
{
        Arc *oldArc;
        Node *current = child, *oldParent, *newParent = parent;
#ifdef STATS
```

```
            ++ numMergers;
#endif
            while (current->parent)
            {
                        oldArc = current->arcToParent;
                        current->arcToParent = newArc;
                        oldParent = current->parent;
                        breakRelationship (oldParent, current);
                        addRelationship (newParent, current);
                        newParent = current;
                        current = oldParent;
                        newArc = oldArc;
                        newArc->direction = 1 - newArc->direction;
            }
            current->arcToParent = newArc;
            addRelationship (newParent, current);
}
static inline void
pushUpward (Arc *currentArc, Node *child, Node *parent, const uint resCap)
{
#ifdef STATS
            ++ numPushes;
#endif
            if (resCap >= child->excess)
            {
                        parent->excess += child->excess;
                        currentArc->flow += child->excess;
                        child->excess = 0;
                        return;
            }
            currentArc->direction = 0;
            parent->excess += resCap;
            child->excess -= resCap;
            currentArc->flow = currentArc->capacity;
            parent->outOfTree[parent->numOutOfTree] = currentArc;
            ++ parent->numOutOfTree;
            breakRelationship (parent, child);
#ifdef LOWEST_LABEL
            lowestStrongLabel = child->label;
#endif

            addToStrongBucket (child, &strongRoots[child->label]);
```

```
}
static inline void
pushDownward (Arc *currentArc, Node *child, Node *parent, uint flow)
{
#ifdef STATS
        ++ numPushes;
#endif

        if (flow >= child->excess)
        {
                parent->excess += child->excess;
                currentArc->flow -= child->excess;
                child->excess = 0;
                return;
        }
        currentArc->direction = 1;
        child->excess -= flow;
        parent->excess += flow;
        currentArc->flow = 0;
        parent->outOfTree[parent->numOutOfTree] = currentArc;
        ++ parent->numOutOfTree;
        breakRelationship (parent, child);
#ifdef LOWEST_LABEL
        lowestStrongLabel = child->label;
#endif

        addToStrongBucket (child, &strongRoots[child->label]);
}
static void
pushExcess (Node *strongRoot)
{
        Node *current, *parent;
        Arc *arcToParent;
        int prevEx=1;
        for (current = strongRoot; (current->excess && current->parent); current = parent)
        {
                parent = current->parent;
                prevEx = parent->excess;
                arcToParent = current->arcToParent;
                if (arcToParent->direction)
                {
                        pushUpward (arcToParent, current, parent, (arcToParent->capacity -
```

```
arcToParent->flow));
                        }
                        else
                        {
                                pushDownward (arcToParent, current, parent, arcToParent->flow);
                        }
                }
                if ((current->excess > 0) && (prevEx <= 0))
                {
        #ifdef LOWEST_LABEL
                        lowestStrongLabel = current->label;
        #endif
                                addToStrongBucket (current, &strongRoots[current->label]);
                }
        }
        static Arc *
        findWeakNode (Node *strongNode, Node **weakNode)
        {
                uint i, size;
                Arc *out;
                size = strongNode->numOutOfTree;
                for (i=strongNode->nextArc; i<size; ++i)
                {

        #ifdef STATS
                        ++ numArcScans;
        #endif
        #ifdef LOWEST_LABEL
                        if (strongNode->outOfTree[i]->to->label == (lowestStrongLabel-1))
        #else
                        if (strongNode->outOfTree[i]->to->label == (highestStrongLabel-1))
        #endif
                        {
                                strongNode->nextArc = i;
                                out = strongNode->outOfTree[i];
                                (*weakNode) = out->to;
                                -- strongNode->numOutOfTree;
                                strongNode->outOfTree[i]        =       strongNode->outOfTree[strongNode-
>numOutOfTree];
                                return (out);
                        }
        #ifdef LOWEST_LABEL
```

```
                        else if (strongNode->outOfTree[i]->from->label == (lowestStrongLabel-1))
        #else
                        else if (strongNode->outOfTree[i]->from->label == (highestStrongLabel-1))
        #endif
                        {
                                strongNode->nextArc = i;
                                out = strongNode->outOfTree[i];
                                (*weakNode) = out->from;
                                -- strongNode->numOutOfTree;
                                strongNode->outOfTree[i]        =       strongNode->outOfTree[strongNode-
>numOutOfTree];
                                return (out);
                        }
                }
                strongNode->nextArc = strongNode->numOutOfTree;
                return NULL;
        }
        static void
        checkChildren (Node *curNode)
        {
                for ( ; (curNode->nextScan); curNode->nextScan = curNode->nextScan->next)
                {
                        if (curNode->nextScan->label == curNode->label)
                        {
                                return;
                        }
                }
                -- labelCount[curNode->label];
                ++      curNode->label;
                ++ labelCount[curNode->label];
        #ifdef STATS
                ++ numRelabels;
        #endif
                curNode->nextArc = 0;
        }
        static void
        processRoot (Node *strongRoot)
        {
                Node *temp, *strongNode = strongRoot, *weakNode;
                Arc *out;
                strongRoot->nextScan = strongRoot->childList;
                if ((out = findWeakNode (strongRoot, &weakNode)))
```

```
        {
                merge (weakNode, strongNode, out);
                pushExcess (strongRoot);
                return;
        }
        checkChildren (strongRoot);
        while (strongNode)
        {
                while (strongNode->nextScan)
                {
                        temp = strongNode->nextScan;
                        strongNode->nextScan = strongNode->nextScan->next;
                        strongNode = temp;
                        strongNode->nextScan = strongNode->childList;

                        if ((out = findWeakNode (strongNode, &weakNode)))
                        {
                                merge (weakNode, strongNode, out);
                                pushExcess (strongRoot);
                                return;
                        }

                        checkChildren (strongNode);
                }

                if ((strongNode = strongNode->parent))
                {
                        checkChildren (strongNode);
                }
        }
        addToStrongBucket (strongRoot, &strongRoots[strongRoot->label]);
#ifndef LOWEST_LABEL
        ++ highestStrongLabel;
#endif
}
#ifdef LOWEST_LABEL
static Node *
getLowestStrongRoot (void)
{
        uint i;
        Node *strongRoot;
        if (lowestStrongLabel == 0)
```

```
        {
                while (strongRoots[0].start)
                {
                        strongRoot = strongRoots[0].start;
                        strongRoots[0].start = strongRoot->next;
                        strongRoot->next = NULL;
                        strongRoot->label = 1;

#ifdef STATS
                        ++ numRelabels;
#endif
                        -- labelCount[0];
                        ++ labelCount[1];
                        addToStrongBucket (strongRoot, &strongRoots[strongRoot->label]);
                }
                lowestStrongLabel = 1;
        }

        for (i=lowestStrongLabel; i<numNodes; ++i)
        {
                if (strongRoots[i].start)
                {
                        lowestStrongLabel = i;

                        if (labelCount[i-1] == 0)
                        {
#ifdef STATS
                                ++ numGaps;
#endif
                                return NULL;
                        }

                        strongRoot = strongRoots[i].start;
                        strongRoots[i].start = strongRoot->next;
                        strongRoot->next = NULL;
                        return strongRoot;
                }
        }
        lowestStrongLabel = numNodes;
        return NULL;
}
#else
```

```
static Node *
getHighestStrongRoot (void)
{
	uint i;
	Node *strongRoot;
	for (i=highestStrongLabel; i>0; --i)
	{
		if (strongRoots[i].start)
		{
			highestStrongLabel = i;
			if (labelCount[i-1])
			{
				strongRoot = strongRoots[i].start;
				strongRoots[i].start = strongRoot->next;
				strongRoot->next = NULL;
				return strongRoot;
			}

			while (strongRoots[i].start)
			{
#ifdef STATS
				++ numGaps;
#endif
				strongRoot = strongRoots[i].start;
				strongRoots[i].start = strongRoot->next;
				liftAll (strongRoot);
			}
		}
	}
	if (!strongRoots[0].start)
	{
		return NULL;
	}
	while (strongRoots[0].start)
	{
		strongRoot = strongRoots[0].start;
		strongRoots[0].start = strongRoot->next;
		strongRoot->label = 1;
		-- labelCount[0];
		++ labelCount[1];

#ifdef STATS
```

```
                ++ numRelabels;
#endif

                addToStrongBucket (strongRoot, &strongRoots[strongRoot->label]);
        }
        highestStrongLabel = 1;
        strongRoot = strongRoots[1].start;
        strongRoots[1].start = strongRoot->next;
        strongRoot->next = NULL;
        return strongRoot;
}
#endif
static void
pseudoflowPhase1 (void)
{
        Node *strongRoot;

#ifdef LOWEST_LABEL
        while ((strongRoot = getLowestStrongRoot ()))
#else
        while ((strongRoot = getHighestStrongRoot ()))
#endif
        {
                processRoot (strongRoot);
        }
}
static void
checkOptimality (const uint gap)
{
        uint i, check = 1;
        ullint mincut = 0;
        llint *excess = NULL;
        excess = (llint *) malloc (numNodes * sizeof (llint));
        if (!excess)
        {
                printf ("%s Line %d: Out of memory\n", __FILE__, __LINE__);
                exit (1);
        }
        for (i=0; i<numNodes; ++i)
        {
                excess[i] = 0;
        }
```

```
for (i=0; i<numArcs; ++i)
{
        if ((arcList[i].from->label >= gap) && (arcList[i].to->label < gap))
        {
                mincut += arcList[i].capacity;
        }

        if ((arcList[i].flow > arcList[i].capacity) || (arcList[i].flow < 0))
        {
                check = 0;
        printf("c Capacity constraint violated on arc (%d, %d). Flow = %d, capacity = %d\n",
                        arcList[i].from->number,
                        arcList[i].to->number,
                        arcList[i].flow,
                        arcList[i].capacity);
        }
        excess[arcList[i].from->number - 1] -= arcList[i].flow;
        excess[arcList[i].to->number - 1] += arcList[i].flow;
}
for (i=0; i<numNodes; i++)
{
        if ((i != (source-1)) && (i != (sink-1)))
        {
                if (excess[i])
                {
                        check = 0;
                printf ("c Flow balance constraint violated in node %d. Excess = %lld\n",
                                i+1,
                                excess[i]);
                }
        }
}
if (check)
{
        printf ("c\nc Solution checks as feasible.\n");
}

check = 1;

if (excess[sink-1] != mincut)
{
        check = 0;
```

```
                printf("c Flow is not optimal - max flow does not equal min cut!\nc\n");
        }
        if (check)
        {
                printf ("c\nc Solution checks as optimal.\nc \n");
                printf ("s Max Flow          : %lld\n", mincut);
        }

        free (excess);
        excess = NULL;
}


static void
quickSort (Arc **arr, const uint first, const uint last)
{
        uint i, j, left=first, right=last, x1, x2, x3, mid, pivot, pivotval;
        Arc *swap;

        if ((right-left) <= 5)
        {// Bubble sort if 5 elements or less
                for (i=right; (i>left); --i)
                {
                        swap = NULL;
                        for (j=left; j<i; ++j)
                        {
                                if (arr[j]->flow < arr[j+1]->flow)
                                {
                                        swap = arr[j];
                                        arr[j] = arr[j+1];
                                        arr[j+1] = swap;
                                }
                        }

                        if (!swap)
                        {
                                return;
                        }
                }
                return;
        }
        mid = (first+last)/2;
```

```
x1 = arr[first]->flow;
x2 = arr[mid]->flow;
x3 = arr[last]->flow;
pivot = mid;
if (x1 <= x2)
{
	if (x2 > x3)
	{
		pivot = left;
		if (x1 <= x3)
		{
			pivot = right;
		}
	}
}
else
{
	if (x2 <= x3)
	{
		pivot = right;
		if (x1 <= x3)
		{
			pivot = left;
		}
	}
}

pivotval = arr[pivot]->flow;
swap = arr[first];
arr[first] = arr[pivot];
arr[pivot] = swap;
left = (first+1);
while (left < right)
{
	if (arr[left]->flow < pivotval)
	{
		swap = arr[left];
		arr[left] = arr[right];
		arr[right] = swap;
		-- right;
	}
	else
```

```
                {
                        ++ left;
                }
        }

        swap = arr[first];
        arr[first] = arr[left];
        arr[left] = swap;
        if (first < (left-1))
        {
                quickSort (arr, first, (left-1));
        }
        if ((left+1) < last)
        {
                quickSort (arr, (left+1), last);
        }
}
static void
sort (Node * current)
{
        if (current->numOutOfTree > 1)
        {
                quickSort (current->outOfTree, 0, (current->numOutOfTree-1));
        }
}
static void
minisort (Node *current)
{
        Arc *temp = current->outOfTree[current->nextArc];
        uint i, size = current->numOutOfTree, tempflow = temp->flow;
        for(i=current->nextArc+1; ((i<size) && (tempflow < current->outOfTree[i]->flow)); ++i)
        {
                current->outOfTree[i-1] = current->outOfTree[i];
        }
        current->outOfTree[i-1] = temp;
}

static void
decompose (Node *excessNode, const uint source, uint *iteration)
{
        Node *current = excessNode;
        Arc *tempArc;
```

```
uint bottleneck = excessNode->excess;
for ( ;(current->number != source) && (current->visited < (*iteration));
                        current = tempArc->from)
{
        current->visited = (*iteration);
        tempArc = current->outOfTree[current->nextArc];
        if (tempArc->flow < bottleneck)
        {
                bottleneck = tempArc->flow;
        }
}
if (current->number == source)
{
        excessNode->excess -= bottleneck;
        current = excessNode;
        while (current->number != source)
        {
                tempArc = current->outOfTree[current->nextArc];
                tempArc->flow -= bottleneck;
                if (tempArc->flow)
                {
                        minisort(current);
                }
                else
                {
                        ++ current->nextArc;
                }
                current = tempArc->from;
        }
        return;
}
++ (*iteration);
bottleneck = current->outOfTree[current->nextArc]->flow;
while (current->visited < (*iteration))
{
        current->visited = (*iteration);
        tempArc = current->outOfTree[current->nextArc];
        if (tempArc->flow < bottleneck)
        {
                bottleneck = tempArc->flow;
        }
        current = tempArc->from;
```

```
		}
		++ (*iteration);
		while (current->visited < (*iteration))
		{
			current->visited = (*iteration);
			tempArc = current->outOfTree[current->nextArc];
			tempArc->flow -= bottleneck;

			if (tempArc->flow)
			{
				minisort(current);
				current = tempArc->from;
			}
			else
			{
				++ current->nextArc;
				current = tempArc->from;
			}
		}
	}
	static void
	recoverFlow (const uint gap)
	{
		uint i, j, iteration = 1;
		Arc *tempArc;
		Node *tempNode;
		for (i=0; i<adjacencyList[sink-1].numOutOfTree; ++i)
		{
			tempArc = adjacencyList[sink-1].outOfTree[i];
			if (tempArc->from->excess < 0)
			{
				if ((tempArc->from->excess + (int) tempArc->flow)  < 0)
				{
					tempArc->from->excess += (int) tempArc->flow;
					tempArc->flow = 0;
				}
				else
				{
			tempArc->flow = (uint) (tempArc->from->excess + (int) tempArc->flow);
					tempArc->from->excess = 0;
				}
			}
```

```
}
for (i=0; i<adjacencyList[source-1].numOutOfTree; ++i)
{
        tempArc = adjacencyList[source-1].outOfTree[i];
        addOutOfTreeNode (tempArc->to, tempArc);
}

adjacencyList[source-1].excess = 0;
adjacencyList[sink-1].excess = 0;
for (i=0; i<numNodes; ++i)
{
        tempNode = &adjacencyList[i];

        if ((i == (source-1)) || (i == (sink-1)))
        {
                continue;
        }
        if (tempNode->label >= gap)
        {
                tempNode->nextArc = 0;
                if ((tempNode->parent) && (tempNode->arcToParent->flow))
                {
                addOutOfTreeNode (tempNode->arcToParent->to, tempNode->arcToParent);
                }

                for (j=0; j<tempNode->numOutOfTree; ++j)
                {
                        if (!tempNode->outOfTree[j]->flow)
                        {
                                -- tempNode->numOutOfTree;
        tempNode->outOfTree[j] = tempNode->outOfTree[tempNode->numOutOfTree];
                                -- j;
                        }
                }

                sort(tempNode);
        }
}
for (i=0; i<numNodes; ++i)
{
        tempNode = &adjacencyList[i];
        while (tempNode->excess > 0)
```

```
                {
                        ++ iteration;
                        decompose(tempNode, source, &iteration);
                }
        }
}
static void
freeMemory (void)
{
        uint i;
        for (i=0; i<numNodes; ++i)
        {
                freeRoot (&strongRoots[i]);
        }
        free (strongRoots);
        for (i=0; i<numNodes; ++i)
        {
                if (adjacencyList[i].outOfTree)
                {
                        free (adjacencyList[i].outOfTree);
                }
        }
        free (adjacencyList);
        free (labelCount);
        free (arcList);
}
int
main(int argc, char ** argv)
{
        double readStart, readEnd, initStart, initEnd, solveStart, solveEnd, flowStart, flowEnd;
        uint gap;
#ifdef LOWEST_LABEL
        printf ("c Lowest label pseudoflow algorithm (Version 3.23)\n");
#else
        printf ("c Highest label pseudoflow algorithm (Version 3.23)\n");
#endif
#ifdef FIFO_BUCKET
        printf ("c Using FIFO buckets\n");
#endif
#ifdef LIFO_BUCKET
        printf ("c Using LIFO buckets\n");
#endif
```

```
        readStart = timer ();
        readDimacsFileCreateList ("D://dim.txt");
        readEnd=timer ();
#ifdef PROGRESS
        printf ("c Finished reading file.\n"); fflush (stdout);
#endif
        initStart = readEnd;
        simpleInitialization ();
        initEnd=timer ();
#ifdef PROGRESS
        printf ("c Finished initialization.\n"); fflush (stdout);
#endif
        solveStart=initEnd;
        pseudoflowPhase1 ();
        solveEnd=timer ();
#ifdef PROGRESS
        printf ("c Finished phase 1.\n"); fflush (stdout);
#endif
#ifdef LOWEST_LABEL
        gap = lowestStrongLabel;
#else
        gap = numNodes;
#endif
        flowStart = solveEnd;
        recoverFlow(gap);
        flowEnd=timer ();
        printf ("c Number of nodes     : %d\n", numNodes);
        printf ("c Number of arcs      : %d\n", numArcs);
        printf ("c Time to read        : %.3f\n", (readEnd-readStart));
        printf ("c Time to initialize  : %.3f\n", (initEnd-initStart));
        printf ("c Time to min cut     : %.3f\n", (solveEnd-initStart));
        printf ("c Time to max flow    : %.3f\n", (flowEnd-initStart));
#ifdef STATS
        printf ("c Number of arc scans : %lld\n", numArcScans);
        printf ("c Number of mergers   : %d\n", numMergers);
        printf ("c Number of pushes    : %lld\n", numPushes);
        printf ("c Number of relabels  : %d\n", numRelabels);
        printf ("c Number of gaps      : %d\n", numGaps);
#endif
        checkOptimality (gap);
#ifdef DISPLAY_CUT
```

```
                displayCut (gap);
        #endif
        #ifdef DISPLAY_FLOW
                displayFlow ();
        #endif
                freeMemory ();
                system("pause");
                return 0;
}
```