Решение задач линейного программирования с использованием GNU Octave, GLPK и Python

А.В. Пономарев

Оглавление

Версия 1.0 (22.03.2016)

Введение

Модели линейного программирования широко используются при рациональной организации производственных и транспортных систем. Более того, это один из тех немногих, в общем-то, классов моделей промышленных систем, допускающих нахождение оптимальных управляющих воздействий с помощью достаточно эффективных алгоритмов, позволяющих решать задачи большой размерности, встречающиеся на практике.

Данное пособие призвано познакомить студентов с основными классами современных инструментов, используемых для решения задач линейного программирования. Выбор именно этих

трех инструментов – GNU Octave, GLPK и Python – обусловлен тем, что они представляют совершенно разных сфер применения: универсальный математический пакет, специализированный пакет для решения линейных оптимизационных задач, и язык программирования общего назначения соответственно. Таким образом, данное пособие показывает, как именно решаются задачи линейного программирования в каждой из этих сфер, что позволит студентам осуществлять осознанный выбор инструмента в соответствии с особенностями решаемой задачи.

Помимо нахождения оптимального решения, в пособии рассматриваются и вопросы анализа чувствительности найденного решения к изменению параметров задачи, если это поддерживается соответствующими инструментами.

Предполагается, что читатели уже знакомы с основными положениями теории линейного программирования; наиболее важные теоретические понятия поясняются в пособии по мере необходимости, однако автор не ставил перед собой цели дать исчерпывающее введение в проблему.

Программные средства для решения задач линейного программирования

Можно выделить две категории средств, применяемых при решении задач линейного программирования: среды моделирования, предназначенные для записи моделей, и собственно решатели, реализующие тот или иной метод решения задачи (см. Рисунок 1). Физически решатель, как правило, представляет собой динамически подключаемую библиотеку; конкретная задача загружается в библиотеку либо посредством вызовов функций интерфейса прикладного программирования библиотеки (API), либо через файлы. Одним из широко распространенных файловых форматов для представления задач линейного программирования является MPS.

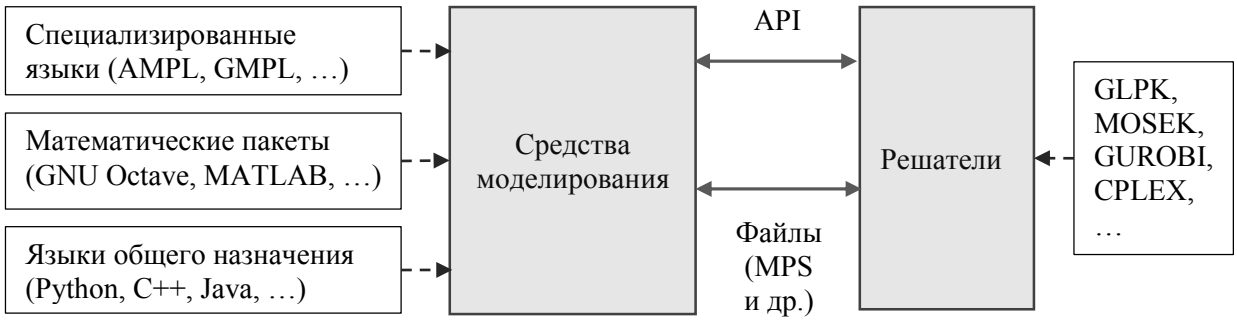


Рисунок 1 – Категории средств, применяемых при решении задач линейного программирования, и их взаимосвязь

На диаграмме «языки общего назначения» и «математические пакеты» показаны как подмножества «средств моделирования». Это следует понимать только в том смысле, что в качестве средств моделирования используются соответствующие языки и пакеты. При этом форма записи самой модели может в значительной степени варьироваться в соответствии с основными принципами построения высказываний соответствующего языка или пакета, используемого для моделирования. Так, при использовании для моделирования математических пакетов (типа GNU Octave) модель записывается как набор матриц и векторов (см. соответствующий раздел), при использовании языков общего назначения могут применяться как матрицы и вектора (поскольку такие конструкции допускаются большинством языков), так и специальные синтаксические конструкции (см. раздел по Python). Вполне предсказуемо, что в наибольшей степени для описания задач линейного программирования подходят специальные языки моделирования, сконструированные именно с этой целью. Примерами таких языков являются AMPL, GAMS, OPL (CPLEX), ZIMPL, GMPL и другие.

В пособии рассматривается язык моделирования GMPL (GNU Mathematical Programming Language). Выбор именно этого языка связан с двумя факторами:

- GMPL является подмножеством одного из наиболее популярных языков алгебраического моделирования – AMPL. Например, по данным сервера NEOS, предоставляющего возможность бесплатного доступа к широкому спектру решателей (в том числе, коммерческих), в 2014 году 83%

присланных на решение задач были записаны именно на языке AMPL; на втором месте по популярности оказался язык GAMS с 12% задач [1].

- GMPL поддерживается GLPK (GNU Linear Programming Kit), свободно распространяемым пакетом для решения задач линейного программирования.

Постановка задачи

В качестве сквозного примера, иллюстрирующего применение различных инструментов, рассмотрим достаточно типичную задачу линейного программирования производственного типа. Это даст возможность раскрыть экономический смысл понятий теневой и приведенной цены, используемых в анализе чувствительности.

Фабрика производит три вида продукции: П1, П2 и П3. Известна цена на продукцию для распространителей и приблизительный спрос на каждый из видов продукции в неделю (см. Таблицу 1). Процессы производства продукции разных видов имеют отличия. На фабрике есть три цеха: Ц1, Ц2 и Ц3. Для производства продукции П1 необходимы только технологические операции, производимые цехом Ц1, для П2 – Ц1 и Ц3, для производства П3 – необходима полная технологическая цепочка, включающая обработку во всех трех цехах. Причем, если в цехах Ц1 и Ц2 продукция разных видов обрабатывается одинаково, и известна общая производительность этих цехов в единицах обработанной продукции в неделю, то в цехе Ц3 предполагается ручная обработка, причем известны как временные затраты на обработку каждого из видов продукции, так и общий фонд рабочего времени сотрудников цеха в неделю (см. Таблицу 2). Из всех видов материалов, используемых при производстве продукции, ограниченным является только один, поставки его в неделю и потребности для каждого из видов продукции приведены в таблице 3.

Необходимо составить производственный план на неделю, максимизирующий выручку от реализации продукции.

Таблица 1 – Характеристики продукции

Вид продукции	Цена, руб.	Спрос, шт. в неделю
П1	1200	35
П2	2500	25
П3	1400	30

Таблица 2 – Производительность цехов

Ц1, шт. в неделю	Ц2, шт. в неделю	Ц3, часов (П2/П3/Общий фонд)
40	20	8/2/80

Таблица 3 – Материалы

Поставки в неделю, кг	Потребление на ед. продукта П1, кг	Потребление на ед. продукта П2, кг	Потребление на ед. продукта П3, кг
50	0,8	0,6	0,7

Формальная постановка

Пусть x_i – количество единиц продукции i-того вида, которое необходимо произвести за неделю ($i \in \{1, 2, 3\}$).

Тогда условие задачи можно формально записать следующим образом:

$$1200x_1 + 2500x_2 + 1400x_3 \to max$$

$$x_1 + x_2 + x_3 \leq 40 \quad (1)$$

$$x_3 \leq 20 \quad (2)$$

$$8x_2 + 2x_3 \leq 80 \quad (3)$$

$$x_1 \leq 35 \quad (4)$$

$$x_2 \leq 25 \quad (5)$$

$$x_3 \leq 30 \quad (6)$$

$$0{,}8x_1 + 0{,}6x_2 + 0{,}7x_3 \leq 50 \quad (7)$$

$$x_{1,2,3} \geq 0$$

Ограничения (1)-(3) диктуются производительностью цехов, ограничения (4)-(6) обусловлены спросом на продукцию, а (7) выражает ограничение на использование материала.

Табличная форма (которая, в частности, окажется полезной при решении задачи с помощью Octave) приведена в Таблице 4.

Таблица 4 – Табличная форма задачи линейного программирования

	x_1	x_2	x_3	Неравенство	Правая часть
c	1200	2500	1400	-	max
y1	1	1	1	<=	40
y2	0	0	1	<=	20
y3	0	8	2	<=	80
y4	1	0	0	<=	35
y5	0	1	0	<=	25
y6	0	0	1	<=	30
y7	0,8	0,6	0,7	<=	50

GNU Octave

GNU Octave – это свободная система для инженерных и математических вычислений и одноименный язык программирования. Одной из особенностей, способствовавшей распространению GNU Octave, является сходство используемого языка с языком популярной проприетарной системы аналогичного назначения MATLAB. Это сходство настолько велико, что многие скрипты могут без изменения запускаться как в Octave, так и в MATLAB.

В данном пособии затрагиваются только те возможности GNU Octave, которые связаны с решением задач линейного программирования, для более подробного знакомства с возможностями этой системы и языка рекомендуется обратиться к соответствующей литературе (например, [2, 3]).

Функция glpk

Для решения задач линейного программирования (в том числе, целочисленного и смешанного линейного программирования) в Octave служит функция `glpk`[1]. При вызове с тремя параметрами `glpk(c, A, b)` решает задачу

$$c^T x \to \min$$
$$Ax = b$$
$$x \geq 0$$

Передача функции дополнительных параметров позволяет решать задачи с нестрогими ограничениями, устанавливать тип искомого экстремума (минимум или максимум). Полный перечень параметров функции приведен ниже:

`c` – вектор-столбец коэффициентов целевой функции.

`A` – матрица с коэффициентами ограничений.

`b` – вектор-столбец со значениями свободных членов.

`lb` – вектор-столбец с нижними границами для каждой переменной (по умолчанию, 0).

`ub` – вектор-столбец с верхними границами для каждой переменной (если нет, то переменные считаются не ограниченными сверху).

`ctype` – массив символов с типами ограничений. Элементы массива интерпретируются следующим образом:

`F` – свободное ограничение (ограничение будет проигнорировано при решении задачи);

`U` – ограничение сверху (если этот символ стоит в позиции `i` параметра `ctype`, `A(i,:)` обозначает `i`-тую строку матрицы с коэффициентами ограничений `A`, а `b(i)` обозначает `i`-тый элемент вектора `b`, то для допустимого решения должно выполняться условие `A(i,:)*x <= b(i)`);

`S` – равенство (`A(i,:)*x = b(i)`);

`L` – ограничение снизу (`A(i,:)*x >= b(i)`);

`D` – двухстороннее ограничение: `A(i,:)*x >= -b(i)` и `A(i,:)*x <= b(i).`

[1] Название этой функции отражает тот факт, что в качестве решателя задач линейного программирования GNU Octave использует библиотеку GLPK.

`vartype` – строка с типами переменных:

`C` – непрерывная переменная;

`I` – дискретная переменная.

`sense` – если параметр sense принимает значение 1 (по умолчанию), то решается задача минимизации, а если -1, то задача максимизации.

param – позволяет в определенных рамках настроить то, как именно должна быть решена задача: какой использовать метод, применять ли масштабирование переменных, сколько итераций симплекс-метода проделать и т.п. Подробную информацию о возможных настройках можно получить, введя `help glpk` в командной строке Octave.

Функция возвращает:

`xopt` – значения переменных, соответствующие оптимальному значению целевой функции (в смысле, управляемом параметром sense).

`fopt` – оптимальное значение целевой функции.

`errnum` – код ошибки. Некоторые, наиболее часто встречающиеся коды ошибок:

0 – нет ошибки, найдено решение (дополнительные характеристики найденного решения содержатся в поле `extra.status`);

2 – матрица коэффициентов ограничений вырождена;

10 – (прямая) задача не имеет допустимых решений;

11 – двойственная задача не имеет допустимых решений (следовательно, в прямой задаче функция неограниченно возрастает или убывает, в зависимости от направления оптимизации);

15 – ни прямая, ни двойственная задачи не имеют допустимых решений.

`extra` – структура данных, позволяющая получить дополнительную информацию о решении. Структура содержит следующие поля:

`lambda` – теневые цены (значения двойственных переменных).

`redcosts` – приведенные цены.

`time` – время (в секундах), затраченное на решение задачи.

`status` – статус решения задачи (содержащегося в `xopt` и `fopt`):

1 – решение не определено;

2 – допустимое решение;

3 – недопустимое решение;

4 – задача не имеет допустимых решений;

5 – оптимальное решение;

6 – задача не имеет конечного решения.

Рассмотрим несколько примеров применения этой функции.

1) Пусть необходимо решить следующую задачу линейного программирования:

$$2x_1 + x_2 \rightarrow max$$
$$x_1 \leq 1$$
$$x_2 \leq 1$$
$$x_{1,2} \geq 0$$

В задаче две переменных, следовательно, размерность вектора-столбца коэффициентов целевой функции 2×1. Этот вектор-столбец можно задать так:

```
c = [2; 1];
```

или так:

```
c = [2, 1]';
```

В соответствии с синтаксисом задания матриц Octave, точка с запятой (`;`) разделяет строки матрицы, а запятая (`,`) – значения в строке. Таким образом, в первом примере мы сформировали матрицу из двух строк, каждая из которых содержит по одному значению, а во втором мы сформировали матрицу строку и затем транспонировали ее (`'`).

Зададим матрицу коэффициентов переменных в ограничениях `A` и вектор-столбец правых частей `b`:

```
A = [1, 0;
     0, 1];
b = [1, 1]';
```

Поскольку по условиям задачи переменные должны быть неотрицательны и не ограничены сверху, то:

```
lb = [0, 0]';
ub = [];
```

Оба ограничения имеют вид (<=), следовательно, в строке ctype кодируются символами "U"; переменные непрерывные, следовательно, в строке vartype кодируются символами "C":

```
ctype = "UU";
vartype = "CC";
```

При вызове функции glpk() подставляем значения переменных, определенных выше, вместо соответствующих параметров. Кроме того, укажем, что необходимо решать задачу максимизации, передав -1 в качестве параметра sense:

```
[xopt, fopt, errnum, extra] = glpk(c, A, b, lb, ub, ctype, vartype, -1);
```

В результате выполнения функции:

```
xopt =

   1
   1

fopt =  3
errnum = 0
extra.status = 5
```

Значение errnum = 0 говорит о том, что было успешно найдено решение задачи оптимизации (это же подтверждается и значением extra.status = 5). В решении обе переменные принимают значение 1, давая значение целевой функции 3. Подробной интерпретации других значений возвращенной структуры extra посвящен подраздел Анализ чувствительности.

2) Модифицируем задачу, добавив еще одно ограничение:

$$2x_1 + x_2 \to max$$
$$x_1 \le 1$$
$$x_2 \le 1$$
$$x_1 + x_2 = 3$$
$$x_{1,2} \ge 0$$

Решение:

```
c = [2, 1]';
A = [1, 0;
     0, 1;
     1, 1];
b = [1, 1, 3]';
[xopt, fopt, errnum, extra] = glpk(c, A, b, [0, 0]', [], "UUS", "CC", -1);
```

Результат:

```
xopt =

   NA
   NA

fopt =  NA
errnum = 10
extra.status = 0
```

Код ошибки 10 говорит о том, что функции не удалось найти допустимых решений.

3) Модифицируем исходную задачу, изменив одно из ограничений:

$$2x_1 + x_2 \to max$$

$$-2x_1 + x_2 \leq 0.5$$
$$x_2 \leq 1$$
$$x_{1,2} \geq 0$$

Решение:

```
c = [2, 1]';
A = [-2, 1;
     0, 1];
b = [0.5, 1];
[xopt, fopt, errnum, extra] = glpk(c, A, b, [0, 0]', [], "UU", "CC", -1);
```

Результат:

```
errnum = 11
```

Данный код ошибки означает, что не удалось найти решение двойственной задачи. Дело в том, что решатель совершенно не обязательно действует в соответствии с алгоритмом прямого симплекс-метода. Используя связи между прямой и двойственной задачами линейного программирования, он может начать с решения двойственной, если по некоторым «соображениям» это покажется предпочтительнее. Видимо, в данном случае именно так и случилось. Отсутствие же решения двойственной задачи свидетельствует о неограниченности прямой.

Можно передать функции `glpk()` дополнительные настройки, слегка модифицировав обычный алгоритм решения задачи (дополнительный параметр `param`):

```
param.presol = 0
[xopt, fopt, errnum, extra] = glpk(c, A, b, [0, 0]', [], ...
                                   "UU", "CC", -1, param);
```

Результат:

```
xopt =

   0
   0
fopt = 0
errnum = 0
extra.status =  6
```

Результат, на первый взгляд, достаточно неожиданный. Код ошибки 0 говорит о том, что решение задачи завершилось успешно (то есть, при выбранном методе решения ошибок не возникло), однако `extra.status` указывает, что решение является неограниченным, следовательно, возвращенные `xopt` и `fopt` смысла не имеют. Данный пример призван продемонстрировать, что интерпретировать код ошибки и статус нужно совместно, принимая во внимание используемый метод решения и связь между прямой и двойственной задачами линейного программирования.

Пример решения задачи

Для решения с помощью GNU Octave задачи, приведенной в разделе Постановка задачи, необходимо подготовить набор матриц и векторов, которые должны быть переданы функции `glpk()` в качестве параметров. Все эти данные уже содержатся в табличной форме задаче (Таблица 4), необходимо добавить лишь специфические обозначения (например, кодирование ограничений <= с помощью символов **"U"** и пр.). Табличная форма задачи с применением обозначений Octave приведена в таблице 5. Изменения в таблице выделены полужирным шрифтом. Так, рядом с каждой переменной поставлен символ **"C"**, наличие которого в параметре `vartype` функции `glpk()` означает, что соответствующая переменная является непрерывной, рядом с каждым символом неравенства – **"U"**, потому что именно так кодируется ограничение «меньше, либо равно» в параметре `ctype` функции `glpk`, а рядом с символом максимизации – **"-1"**, потому что именно таким должен быть параметр `sense` для решения задачи максимизации.

Таблица 5 – Подготовка исходных данных для вызова `glpk`

	x_1 **[C]**	x_2 **[C]**	x_3 **[C]**	Неравенство	Правая часть
c	1200	2500	1400	-	max **[-1]**
y1	1	1	1	<= **[U]**	40
y2	0	0	1	<= **[U]**	20
y3	0	8	2	<= **[U]**	80
y4	1	0	0	<= **[U]**	35
y5	0	1	0	<= **[U]**	25
y6	0	0	1	<= **[U]**	30
y7	0,8	0,6	0,7	<= **[U]**	50

Полный код для подготовки параметров и решения задачи выглядит следующим образом:

```
c = [1200 2500 1400]';

A = [1   1   1;
     0   0   1;
     0   8   2;
     1   0   0;
     0   1   0;
     0   0   1;
     0.8 0.6 0.7];

b = [40 20 80 35 25 30 50]';

[x_max, z_max, error_code, extra] = glpk(c, A, b, zeros(3, 1), [], ...
                                         "UUUUUUU", "CCC", -1)
x_max =

   30
   10
    0

z_max =  61000
error_code = 0
extra.status = 5
```

Сочетание кода ошибки 0 и статуса 5 означает, что функции `glpk()` удалось найти значения переменных, при которых целевая функция принимает наибольшее значение. Решение задачи содержится в векторе `x_max`. В соответствии с введенными обозначениями, производственный план на неделю, максимизирующий выручку от реализации продукции, должен включать 30 единиц продукта П1 и 10 единиц продукта П2. Производить продукт в данных условиях П3 оказывается невыгодно. Наибольшее значение целевой функции, соответствующее наибольшей недельной выручке, содержится в переменной `z_max` и равно 61 тысяче.

Анализ чувствительности

Анализ чувствительности позволяет получить ответ на вопрос как изменится найденное оптимальное решение задачи ЛП, при *незначительном* изменении исходных данных. Как правило, интересует следующее:

1. Как изменится решение, если изменится количество ресурса (значения из вектора правых частей b_i)?

2. Как изменится решение задачи, если изменятся параметры целевой функции c_i?

Ответ на первый вопрос связан с понятием теневой цены ресурса (shadow price), а на второй – с понятием приведенной цены (нормированной стоимости, уменьшенной стоимости, reduced cost).

Теневые цены

Теневая цена λ_i ограничения y_i (с соответствующим значением вектора правых частей b_i) показывает, на сколько вырастет значение целевой функции в точке оптимума, если правая часть ограничения b_i будет увеличена на единицу (при условии, что не изменится состав базисных переменных).

Допустим, есть ограничение, заключающееся в 40 часовой рабочей неделе. Теневая цена этого ограничения в точке оптимума покажет, на сколько вырастет значение целевой функции, если ослабить ограничение на 1. Другими словами, сколько максимально можно доплачивать рабочим за дополнительный час, чтобы это было выгодно.

Сведения для анализа чувствительности содержатся в структуре `extra`, возвращаемой функцией `glpk()`. Получим значение этой структуры для рассматриваемого примера (значения входных матриц оставляем без изменения):

```
[x_max, z_max, ec, extra] = glpk(c, A, b, zeros(3, 1), [], ...
                                 "UUUUUUU", "CCC", -1)
extra =

  scalar structure containing the fields:

    lambda =

       1.2000e+03
       0.0000e+00
       1.6250e+02
       0.0000e+00
       0.0000e+00
       0.0000e+00
       0.0000e+00

    redcosts =

            0
            0
         -125

    time = 0
    status =  5
```

Теневые цены содержатся в поле `lambda` этой структуры (`extra.lambda`). Элемент вектора `extra.lambda` с индексом *i* соответствует теневой цене *i*-того ограничения (заданного *i*-той строкой матрицы `A` при вызове `glpk()`). В данном случае, мы видим, что у всех ограничений, кроме первого и третьего, теневые стоимости равны нулю, следовательно, ресурсы, соответствующие этим ограничениям, не являются дефицитными и увеличивать их количество смысла нет. Рассмотрим ограничения с ненулевой теневой ценой. Поскольку первое ограничение соответствует производительности первого цеха, то ненулевая теневая цена (1200 руб. – рублей, потому что именно в рублях измеряется наша целевая функция) для него означает, что при увеличении недельной производительности первого цеха (а именно для него составлено это ограничение) с 40 до 41 единицы продукции недельная выручка может увеличиться на 1200 руб. Это же число можно интерпретировать и иначе – если мы можем увеличить производительность цеха Ц1 на единицу, затратив меньше, чем 1200 руб., то это следует сделать, поскольку приведет к увеличению дохода. Ненулевая теневая цена третьего ограничения (162,50 руб.) означает, что при увеличении общего фонда рабочего времени сотрудников цеха Ц3 на 1 час (третье ограничение составлено для цеха Ц3 и его части измеряются в часах), выручка может вырасти на 162 руб. 50 коп. Опять же, альтернативно это означает, что если час рабочего времени сотрудников цеха Ц3 стоит меньше 162 руб. 50 коп., то можно доход можно увеличить.

Следует иметь в виду, что описанные оценки верны только для случая сохранения структуры решения. Если структура изменится, то есть, активизируются новые ограничения, то реальный прирост дохода от ослабления заданного ограничения может оказаться меньше. Попробуем решить задачу заново с измененными ограничениями. Начнем с первого:

```
db = [1 0 0 0 0 0 0]';  % приращение к вектору правых частей
[x_max, z_max] = glpk(c, A, b + db, zeros(3, 1), [], "UUUUUUU", "CCC", -1);
x_max =
```

```
   10
    0

z_max = 62200
```

Действительно, значение выручки увеличилось на 1200 руб., как и «предсказывалось» теневой ценой первого ограничения. Попробуем перебирать различные приращения, чтобы найти то, которое приведет к изменению структуры решения (определяемому косвенно как приращение выручки менее, чем на теневую цену, то есть, менее, чем на 1200 руб.).

```
db = [1 0 0 0 0 0 0]';
prev_z = z_max;
a = 1;
while (1)
  [x_max, z_max, ec, extra] = glpk(c, A, b + a*db, zeros(3, 1), [], ...
                                   "UUUUUUU", "CCC", -1);
  if ec != 0 || extra.status != 5
    printf("glpk(): can't find solution.\n");
    break;
  endif
  printf("Increment %d: z_max = %f delta = %f\n", a, z_max, z_max - prev_z);
  if abs(z_max - prev_z - 1200) > 1e-6
    printf("Basis changed at increment %d\n", a);
    break;
  endif
  prev_z = z_max;
  a = a + 1;
endwhile
```

Результат выполнения скрипта:

```
Increment 1: z_max = 62200.000000 delta = 1200.000000
Increment 2: z_max = 63400.000000 delta = 1200.000000
Increment 3: z_max = 64600.000000 delta = 1200.000000
Increment 4: z_max = 65800.000000 delta = 1200.000000
Increment 5: z_max = 67000.000000 delta = 1200.000000
Increment 6: z_max = 68033.333333 delta = 1033.333333
Basis changed at increment 6
```

Как видно из вывода, сгенерированного скриптом, без изменения состава базисных переменных данное ограничение может быть ослаблено на 5 единиц, что даст, в общей сложности увеличение выручки на 6 тыс. руб. Видно, также, что при дальнейшем увеличении выручка так же продолжает расти (просто теневые цены, вычисленные при исходных ограничениях, оказываются уже неактуальными). В принципе, подобный перебор можно продолжать и дальше, вплоть до момента, когда ослабление ограничения вообще перестанет влиять на значение целевой функции. В каком-то смысле, такой перебор является упрощенной версией анализа чувствительности, методом «грубой силы».

Исследование теневой цены третьего ограничения оставляю читателю в качестве упражнения.

Приведенные цены

Приведенная цена (*reduced cost*), в отличие от теневой цены, ассоциируется уже не с ограничениями, а с переменными.

Приведенной ценой u_i небазисной переменной x_i является величина, на которую уменьшится значение целевой функции в точке оптимума, если x_i будет увеличено с 0 до 1 (войдет в базис), при условии, что это изменение мало.

Комментарии:

1) Для базисных переменных приведенная цена всегда 0.

2) Приведенную цену также можно назвать ценой возможности (opportunity cost). Предположим, нас вынудили произвести единицу x_i, продукта, который мы не собирались производить. Эта вынужденная возможность будет нам чего-то стоить, поскольку связана с

добавлением нового ограничения. Насколько именно снизится значение ЦФ – и есть приведенная цена.

3) Приведенная цена u_i – это величина, на которую должен измениться коэффициент c_i перед x_i, чтобы x_i стал ненулевым в точке оптимума. Предположим, мы производим товары x_1, …, x_n, которые приносят нам доход c_1, …, c_n соответственно. Мы связаны определенными ограничениями, которые здесь не играют роли. Сформировав и решив задачу ЛП, чтобы оптимизировать прибыль, получаем оптимальный план производства: x_1*, …, x_n*, соответствующий прибыли z*. Предположим, в этом плане x_2* = 0. Очевидно, прибыль от производства товаров этого типа недостаточно велика, чтобы нам было выгодно производить их (а не какие-то другие товары). Тогда мы можем спросить себя: а какой должна быть прибыль от x_2, чтобы нам все-таки было выгодно их производить, хотя бы в небольших количествах? Ответ $c_2 - u_2$ (или $c_2 + u_2$, в зависимости от того, как интерпретируется знак). Это означает, что прибыль должна увеличиться по крайней мере на размер reduced cost, чтобы стало выгодно производить этот продукт.

Значение приведенной цены находится в поле `redcosts` структуры `extra`, возвращаемой `glpk`:

```
[x_max, z_max, s, extra] = glpk(c, A, b, zeros(3, 1), [], "UUUUUUU", "CCC", -1);
extra.redcosts

ans =

     0
     0
  -125
```

Видим, что для единственной небазисной переменной, соответствующей продукту П3, приведенная цена равна 125 руб. (для базисных она всегда равна нулю). То есть, если мы включим в план производства одну штуку продукта П3, то выручка должна уменьшиться на 125 руб. И наоборот, если мы увеличим цену на продукт П3 хотя бы на 125 руб., то производить этот продукт станет выгодно. Для этого добавим нижнюю границу для переменной x_3 (параметр `lb` при вызове `glpk`), потребовав, чтобы ее значение было не меньше 1:

```
[x_max, z_max, error_code] = glpk(c, A, b, [0 0 1]', [], "UUUUUUU", "CCC", -1)

x_max =

   29.2500
    9.7500
    1.0000

z_max =  60875
error_code =  0
```

Действительно, выручка уменьшилась на 125 руб. Теперь попробуем увеличить цену на 125 руб., чтобы проверить, действительно ли при этих условиях станет выгодно производить продукт П3:

```
[x_max, z_max, error_code, extra] = glpk(c + [0 0 125]', A, b, zeros(3,1), ...
                                         [], "UUUUUUU", "CCC", -1)
x_max =

   30
   10
    0

z_max =  61000
error_code =  0
extra =

  scalar structure containing the fields:
```

```
    lambda =

      1.2000e+003
      0.0000e+000
      1.6250e+002
      0.0000e+000
      0.0000e+000
      0.0000e+000
      0.0000e+000

    redcosts =

       0
       0
       0

    time = 0
    status = 5
```

Сам план не изменился, но обратите внимание, что приведенные цены теперь для всех продуктов равны нулю. Попробуем еще немного увеличить цену на третий продукт (на 126 руб.):

```
[x_max, z_max, error_code] = glpk(c + [0 0 126]', A, b, zeros(3,1), [], "UUUUUUU",
"CCC", -1)
x_max =

   15
    5
   20

z_max =  61020
error_code =  0
```

В результате видим серьезное изменение плана производства (но незначительное приращение целевой функции) – теперь продукту П3 отдается предпочтение.

GLPK

GLPK (GNU Linear Programming Kit) представляет собой свободно распространяемый пакет для решения задач линейного программирования (в том числе, целочисленного и смешанного линейного программирования). Основой этого пакета является библиотека, в которой реализованы алгоритмы решения задач линейного программирования (прямой и двойственный симплекс-метод, метод внутренней точки) и метод ветвей и границ для решения задач целочисленного линейного программирования. Помимо библиотеки, пакет включает несколько «оберток», позволяющих использовать эту библиотеку в программах, написанных на разных языках программирования (C++, Java, Python), а также программу, позволяющую решать задачи, используя интерфейс командной строки.

GLPK поддерживает язык математического моделирования GMPL (GNU Mathematical Programming Language), также известный как GNU MathProg, являющийся подмножеством популярного языка AMPL (для знакомства с прообразом можно порекомендовать, например, [4]).

Сам по себе пакет GLPK не предоставляет средств визуального моделирования задач и отображения результатов, однако, как это часто бывает с консольными утилитами GNU, существуют программы сторонних разработчиков, предоставляющие графический интерфейс пользователя и формирующие обращения к консольной программе на основе настроек, указанных пользователем графического интерфейса. Примером такой программы для GLPK является GUSEK[2].

Элементы GMPL

Модель задачи линейного программирования записывается на языке GMPL с использованием следующих видов объектов:

[2] https://sourceforge.net/projects/gusek/

- множества;
- параметры;
- переменные;
- ограничения;
- цели.

Пользователь определяет эти объекты с помощью языковых конструкций. Каждый объект снабжается уникальным именем, используемым для ссылок на этот объект при построении модели. Объекты могут представлять собой многомерные массивы, адресуемые с помощью индексных множеств. Для обращения к одному элементу такого объекта используется нотация `object[index]`, где `object` – это имя объекта, а `index` – это перечисление значений индексных множеств. Например, если `demand` является двумерным параметром, определенным на множестве $I \times J$, то конкретный элемент параметра `demand` будет адресоваться как `demand[i, j]`, где `i` $\in I$ и `j` $\in J$.

В ряде случаев оказывается удобным описывать модель задачи и фактические данные раздельно. Это позволяет использовать одну и ту же модель для решения различных экземпляров задач одного класса. С этой целью исходный файл языка GMPL состоит из двух секций: секции описания модели и секции данных. Секция описания модели содержит описание всех объектов, а секция данных содержит, как правило, значения параметров для конкретного экземпляра задачи. Более того, это разделение можно усилить, поместив секции описания модели и секцию данных в различные файлы, чтобы изменение данных вообще не приводило к необходимости изменять файл, содержащий описание модели.

Описание модели

Предложения

Предложения являются основными элементами описания модели. Все предложения GMPL можно разделить на две категории: декларативные и функциональные. Декларативные предложения (`set`, `param`, `var`, `s.t.`, `goal`) используются для определения объектов модели и свойств этих объектов. Функциональные предложения (`solve`, `check`, `display`, `printf`, `loop`, `table`) предназначены выполнения действий.

Каждое предложение должно заканчиваться точкой с запятой (‘`;`’).

Определение множества (`set`)

```
set name alias domain, attrib, …, attrib;
```

`name` – символическое имя определяемого множества;
`alias` – необязательная строка, задающая псевдоним множества;
`domain` –выражение, задающее пространство индексов объекта;
`attrib, …, attrib` – необязательные атрибуты множества:

`dimen` *`n`* – задает размерность элементов множества;

`within` *`expression`* – указывает множество, элементами которого должны являться элементы определяемого множества (надмножество);

`:=` *`expression`* – определяет состав множества;

`default` *`expression`* – определяет состав множества в том случае, если его состав не был определен в секции описания данных.

Предложение `set` определяет множество. Если не задано пространство индексов, то `set` определяет простое множество, в противном случае- массив множеств по количеству элементов пространства индексов.

Примеры:

```
set nodes;
```

Это предложение определяет множество вершин, не указывая, какие элементы этому множеству принадлежат. Подобная запись имеет смысл, если предполагается, что элементы множества `nodes` будут заданы в секции данных.

```
set arcs within nodes cross nodes;
```

Данное предложение определяет множество дуг, указывая, с помощью атрибута `within`, что это множество должно быть подмножеством декартова произведения множества nodes на само себя.

Значения элементов множества также должны быть заданы в секции данных, а атрибут `within` будет использован для контроля правильности заполнения этого множества.

```
set DaysOfWeek := {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
```

В этом предложении определяется множество дней недели и сразу задаются элементы множества.

Определение параметра (`param`)

```
param name alias domain, attrib, …, attrib;
```

`name` – символическое имя определяемого параметра;
`alias` – необязательная строка, задающая псевдоним параметра;
`domain` – выражение, задающее пространство индексов параметра;
`attrib, …, attrib` – необязательные атрибуты параметра:

`integer, binary, symbolic` – указывает, что параметр является целочисленным, двоичным или символическим соответственно, если этот атрибут не указан, то параметр может принимать любые числовые значения;

`relation expression` – (где *`relation`* может быть одним из `<`, `<=`, `=`, `==`, `>=`, `>`, `<>`, `!=`) задает условие, которому должны удовлетворять значения параметра;

`in` *`expression`* – указывает, что значения параметра должны принадлежать множеству (задаваемому указанным выражением *`expression`*);

`:=` *`expression`* – определяет значение параметра;

`default expression` – определяет значение параметра в том случае, если его значение не было определено в секции описания данных.

Предложение `param` определяет параметр задачи. Если не задано пространство индексов, то `param` определяет один параметр (скаляр), в противном случае – массив параметров по количеству элементов пространства индексов. Наиболее естественной областью применения параметров определение числовых значений, указанных в тексте задачи (стоимости перевозок, потребности, пропускные способности) – определение параметров для этих значений позволяет записать обобщенную модель и многократно использовать ее для различных исходных данных.

Примеры:

```
param N;
```

Определение числового параметра `N`, который может принимать любые значения. Значение параметра не определено; предполагается, что оно будет задано в секции данных.

```
param N integer;
```

Определение параметра `N`, значениями которого могут быть только целые числа.

```
param price{e in arcs};
```

При определении параметра `price` используется пространство индексов (задаваемое множеством `arcs`), следовательно, `price` задает массив параметров – по одному числовому значению для каждого элемента множества `arcs`.

```
param demand{i in 1..N};
```

Еще один пример параметра с пространством индексов. В данном случае, в роли этого пространства выступает множество целых чисел от `1` до `N`. Обращение к отдельным значениям параметра возможно с помощью явного указания индекса. Например, `demand[3]`.

Определение переменной (`var`)

```
var name alias domain, attrib, …, attrib;
```

`name` – символическое имя определяемой переменной;
`alias` – необязательная строка, задающая псевдоним переменной;
`domain` –выражение, задающее пространство индексов переменной;
`attrib, …, attrib` – необязательные атрибуты переменной:

`integer, binary` – указывает, что переменная является целочисленной или бинарной соответственно, если этот атрибут не указан, то переменная может принимать любые числовые значения;

`relation expression` – (где отношение (*`relation`*) может быть одним из `<=`, `=`, `>=`) задает ограничение на множество значений, которые может принимать переменная, если атрибут не указан, то ограничений на диапазон значений, принимаемых переменной, не накладывается (кроме бинарных переменных);

Предложение `var` определяет переменную задачи или целый массив переменных (в случае, если задано пространство индексов). Значения переменные получают в ходе решения задачи оптимизации.

Примеры:

```
var x >= 0;
```

Определение неотрицательной переменной. Типичная ситуация для задач линейного программирования.

```
var flow{a in arcs} >= 0;
```

При определении переменной `flow` используется пространство индексов, следовательно, это предложение определяет целый массив переменных – по одной для каждого элемента множества `arcs`, причем каждая из этих переменных должна быть неотрицательной.

```
var take{o in objects} binary;
```

Такое определение может быть использовано, например, в задаче о рюкзаке – оно задает массив бинарных переменных по одной для каждого элемента множества `objects` (предметов).

Определение ограничения (`subject to`)

```
subject to name alias domain : expression, = expression ;
subject to name alias domain : expression, <= expression ;
subject to name alias domain : expression, >= expression ;
subject to name alias domain : expression, <= expression, <= expression;
subject to name alias domain : expression, >= expression, >= expression;
```

`name` – символическое имя определяемого ограничения;

`alias` – необязательная строка, задающая псевдоним ограничения;

`domain` –выражение, задающее пространство индексов ограничения;

`expression` – линейное выражение, используемое для вычисления компонента ограничения.

Предложение `subject to` определяет ограничение или целую группу ограничений (в случае, если задано пространство индексов). Ключевое слово `subject to` может быть записано сокращенно как `subj to`, `s.t.` или даже пропущено.

Примеры:

```
s.t. r: x[1] + x[2] <= 17;
```

Данное ограничение требует, чтобы сумма двух переменных не превосходила 17. Из самой записи не следует, что `x` – это переменные, однако это предполагается. Следует также заметить, что подобное ограничение является образцом плохого стиля, поскольку константа 17 присутствует непосредственно в определении модели. Правильней было бы определить соответствующий параметр и использовать его в этом ограничении.

```
subject to weight: sum{o in objects} w[o] * take[o] <= W_max;
```

Ограничение, которое могло бы быть использовано в задаче об упаковке рюкзака. При такой интерпретации `weight` – это имя ограничения, выражение `sum` используется для получения суммарного веса набора объектов, для которых переменная `take` принимает значение 1, `W_max` – параметр, задающий вместительность рюкзака, а `w` – параметр-массив (индексируемый элементами множества `objects`), в котором содержатся веса объектов.

```
subject to ration{p in products}: sum{c in crops} x[c] * r[p, c] <= R[p];
```

Это предложение задает сразу семейство однотипных ограничений – по одному для каждого элемента множества `products`.

Определение целевой функции (maximize/minimize)

```
maximize name alias : expression ;
minimize name alias : expression ;
```

name – символическое имя целевой функции;

alias – необязательная строка, задающая псевдоним целевой функции;

expression – линейное выражение, используемое для вычисления значения целевой функции.

Предложение `minimize/maximize` определяет целевую функцию. При добавлении в модель нескольких целевых функций, использоваться в процессе решения задачи будет только первая из них.

Примеры:

```
minimize total_cost: sum{w in warehouses, c in consumers} c[w, c] * x[w, c] ;
```

Пример целевой функции, которая могла бы быть использована в транспортной задаче, если множество пунктов хранения обозначено `warehouses`, множество потребителей – `consumers`, цена перевозки единицы груза обозначена параметром `c`, а объем перевозок – переменной `x`.

```
maximize total_cost: sum{o in objects} price[o] * take[o] ;
```

Пример целевой функции, которая могла бы быть использована в задаче о рюкзаке.

Решение задачи (solve)

Для решения задачи предусмотрено необязательное предложение `solve`, которое может быть использовано в файле модели не более одного раза. Если оно отсутствует, то предполагается, что оно находится в конце секции описания модели.

В момент обработки предложения `solve` происходит решение задачи оптимизации, поэтому никакое из декларативных предложений не может находиться после `solve`. Зато после обработки предложения `solve` переменные получают значения, соответствующие найденному решению, и к ним можно применять функциональные предложения.

Вывод найденного решения

Для вывода в GMPL предусмотрено несколько функциональных предложений. Самое простое из них – `display`. По аналогии с многими другими предложениями, для `display` может быть задано пространство индексов, а выводимые значения разделяются запятой:

```
display : ‘x= ‘, x, ‘y = ‘, y ;
display{o in objects} : o, take[o] ;
```

С помощью `display` можно выводить не только значения выражений, но и объекты модели (множества, ограничения, целевую функцию), однако вывод всегда производится в стандартный поток (как правило, консоль) и возможности форматирования существенно ограничены.

Несколько большую гибкость предоставляет предложение `printf`, которое позволяет осуществлять форматированный вывод, используя синтаксис описания формата одноименной функции языка C. Кроме того, `printf` позволяет направлять вывод в указанный файл (как с перезаписью (“>”), так и с добавлением в конец файла (“>>”)):

```
printf : “Hello, world!\n" ;
printf{i in 1..4} : “x[%d] = %.3f\n”, i, x[i] > “result.txt” ;
```

Кроме того, в GMPL предусмотрен и высокоуровневый интерфейс для ввода-вывода табличных данных с помощью предложения table:

```
table name alias IN driver arg … arg: set <- [fld, …, fld] , par ~ fld, …, par ~ fld
;
table name alias domain OUT driver arg … arg : expr ~ fld, … , expr ~ fld ;
```

name –имя таблицы;

alias – необязательная строка, задающая псевдоним таблицы;

domain –выражение, задающее пространство индексов таблицы;

IN – означает, что данные должны быть прочитаны из таблицы;

OUT – означает, что данные должны быть записаны в таблицу;

driver - идентификатор драйвера, предоставляющего доступ к таблице (поддерживаются файлы формата CSV, базы данных ODBC, xBASE, MySQL);

arg – необязательные параметры, которые должны быть переданы драйверу (например, имя файла для CSV);

set – имя контрольного множества;

fld – имя поля;

par – имя параметра модели (пространство индексов этого параметра должно содержать контрольное множество);

expr – числовое или символьное выражение.

Примеры:

```
table data IN “CSV” “data.csv”: S <- [FROM, TO], d~DISTANCE, c~COST;
```

Параметры модели d и c будут заполнены из CSV-файла data.csv.

```
table res{(f, t) in arcs} OUT “CSV” “flow.csv” : f~FROM, t~TO, flow[f, t]~FLOW;
```

Поток, содержащийся в переменной flow, будет сохранен в файл flow.csv.

Описание данных

Описание данных расположено либо в отдельной секции файла модели, начинающейся предложением data, либо в отдельном файле (этот файл не обязательно начинать с предложения data). Это описание должно задавать элементы всех множеств, определенных в секции описания модели, и значения всех параметров. В методическом пособии рассматриваются только наиболее простые и распространенные синтаксические формы описания данных, более подробную информацию можно получить в руководстве по GMPL.

Задание элементов множества

```
set name, record, …, record ;
set name [symbol, …, symbol] , record, …, record ;
```

name – имя множества;

symbol, …, *symbol* – индексы, задающие конкретный элемент, если у множества соответствующего имени name, определено пространство индексов;

record, …, *record* – записи с данными:

:= - необязательная запись, которая может использоваться для повышения читаемости кода;

simple_data – данные в простой форме;

: *matrix_data* – данные в матричной форме.

Примеры:

```
set nodes := LED VKO KGD TJM ;
```

В этом примере элементы множества nodes задаются в простой форме. Элементами этого множества являются символьные константы (в данном случае, являющиеся международными кодами аэропортов).

```
set arcs := (LED, VKO) (VKO, LED) (VKO, KGD) (VKO, TJM) ;
```

В подразделе пособия, посвященном определению множеств, множество arcs было определено как подмножество декартова произведения множества nodes самого на себя. Таким образом, элементами этого множества должны являться пары, что и имеет место в задании arcs выше.

```
set arcs : LED VKO KGD TJM :=
        LED -   +   -   -
        VKO +   -   +   +
        KGD -   -   -   -
        TJM -   -   -   - ;
```

А это пример задания элементов множества пар в матричной форме. Содержимое множества arcs будет таким же, как в предыдущем примере.

Задание значений параметров

```
param name, record, …, record ;
param name default value, record, …, record ;
```

name – имя параметра;

value – необязательное значение по умолчанию для параметра;

record, …, record – записи с данными:

`:=` - необязательная запись, которая может использоваться для повышения читаемости кода;

plain-data – значения параметра в простой форме;

`:` *tabular-data* – значения параметра в матричной форме.

Примеры:

```
param N := 5 ;
param demand := 1 8 2 14 3 3 4 9 5 11 ;
```

Параметр `N` был определен как скалярный параметр, задание значения для него имеет очевидный синтаксис. С параметром `demand` – немного сложнее; он был определен как параметр-массив, поскольку для него было задано индексное пространство `1..N`. Поскольку `N` присвоено значение 5, то `demand` должен состоять из пяти значений, которые и перечисляются с указанием соответствующих индексов. То есть, например, `demand[1]` присвоено значение 8, а `demand[5]` – значение 11.

```
param r :  1   2    3 :=
     1   0.5 0.3 0.12
     2   0.2 0.1 0.22 ;
```

Это пример использования записи *tabular-data* для задания параметра с двумерными индексами. При таком задании `r[2,1]` будет присвоено значение 0.2.

```
param trans_cost :=
   [*,*,wool]:    1   2    3 :=
             1   0.3 0.2 0.11
             2   0.1 0.1 0.32
   [*,*,steel]:   1   2    3 :=
             1   0.7 0.4 0.45
             2   0.2 0.2 0.64 ;
```

Такая конструкция потребуется, если необходимо задать значения для параметра, пространство индексов которого образовано кортежами из трех элементов. Например, стоимость перевозки единицы товара зависит не только от того, откуда и куда товар нужно перевезти, но и от самого товара. В данном примере, товаров рассматривается два: шерсть (wool) и сталь (steel). Конструкция `[*,*,wool]` называется спецификация среза и говорит о том, что следующий за ней блок данных следует воспринимать как задание двумерного параметра (по количеству символов “*”) описывающего стоимости перевозки шерсти (wool). То есть, задание значений многомерного параметра происходит по срезам, размерность каждого из которых не превышает 2.

Пример решения задачи

Запишем модель задачи из раздела Постановка задачи на языке GMPL. Начнем с определения множеств:

```
set P := 1..3;
set W := 1..3;
set W_Mass within W := {1, 2};
set W_Manual within W := W diff W_Mass;
```

Множество `P` соответствует видам производимой продукции. Поскольку в задаче не уточняются названия продукции, в качестве элементов множества `P` используются просто порядковые номера. Множество `W` соответствует цехам. Поскольку выделяется два вида цехов,

обладающих несколько разной логикой формирования ограничений, это отражено и в наборе используемых множеств – выделены подмножества `W_Mass` и `W_Manual` цехов с массовым и ручным производством соответственно, причем множество цехов с ручным производством определено как дополнение множества цехов с массовым производством до множества всех цехов.

По условию задачи известно, что в производстве каждого из видов продукции участвуют не все цеха. Смоделировать это условие можно по-разному. Введем, например, отношение «вид продукции-цех», указывающее, что для производства заданного вида продукции необходимы операции, выполняемые в определенном цехе. Отношение задается как множество пар, являющееся подмножеством декартова произведения исходных множеств:

```
set Ops within P cross W := {(1, 1), (2, 1), (2, 3), (3, 1), (3, 2), (3, 3)};
```

Для удобства задания ограничений определим массив множеств, каждый элемент которого соответствует цеху, а значениями являются те виды продукции, в производстве которых цех принимает участие. Важно, что этот массив является производным и должен быть определен через `Ops`:

```
set W_P{w in W} := setof {(i, w) in Ops} i;
```

Параметрами задачи являются цена каждого вида продукции, спрос на продукцию, потребление материалов, а также характеристики производительности цехов. Введем эти параметры в модель:

```
param price{p in P};      # Цена по видам продукции, руб.
param demand{p in P};     # Спрос в неделю по видам продукции, шт.
param cost{p in P};       # Расход материала на производство продукции, кг
param prod{w in W_Mass};  # Производительность цехов, шт. в неделю
param prod_m_cost{w in W_Manual, p in P};  # Затраты времени на единицу
                                           # продукции, часы
param prod_m{w in W_Manual};  # Фонд рабочего времени, часы
param materials;              # Поставки материалов в неделю, кг
```

Переменными задачи являются количество продукции каждого из видов, которое необходимо произвести за неделю:

```
var x{p in P} >= 0;
```

Составим ограничения. Для цехов, в которых предполагается машинная обработка, известно максимальное количество изделий, которые могут пройти машинную обработку в неделю:

```
s.t. mass{w in W_Mass}: sum{p in W_P[w]} x[p] <= prod[w];
```

Для цеха с ручной обработкой должно выполняться требование по рабочему времени:

```
s.t. manual{w in W_Manual}: sum{p in W_P[w]} x[p] * prod_m_cost[w, p] <= prod_m[w];
```

Производство каждого из видов продукции не должно превосходить спрос:

```
s.t. dem{p in P}: x[p] <= demand[p];
```

Затраты материала на производство не должны превышать поставок:

```
s.t. mat: sum{p in P} x[p] <= materials;
```

Необходимо найти план производства, максимизирующий выручку:

```
maximize profit : sum{p in P} price[p] * x[p];
```

Создадим файл `example.mod` и добавим в него следующий текст:

```
set P := 1..3;
set W := 1..3;
set Ops dimen 2, within P cross W := {(1, 1),
                                      (2, 1), (2, 3),
```

```
                                        (3, 1), (3, 2), (3, 3)};
set W_Mass within W := {1, 2};
set W_Manual within W := W diff W_Mass;

set W_P{w in W} := setof {(i, w) in Ops} i;

param price{p in P};      # Цена по видам продукции, руб.
param demand{p in P};     # Спрос в неделю по видам продукции, шт.
param cost{p in P};       # Расход материала на производство продукции, кг
param prod{w in W_Mass};  # Производительность цехов, шт. в неделю
param prod_m_cost{w in W_Manual, p in P};  # Затраты времени на единицу
                                           # продукции, часы
param prod_m{w in W_Manual};  # Фонд рабочего времени, часы
param materials;              # Поставки материалов в неделю, кг

var x{p in P} >= 0;

s.t. mass{w in W_Mass}:
     sum{p in W_P[w]} x[p] <= prod[w];
s.t. manual{w in W_Manual}:
     sum{p in W_P[w]} x[p] * prod_m_cost[w, p] <= prod_m[w];
s.t. dem{p in P}: x[p] <= demand[p];
s.t. mat: sum{p in P} x[p] <= materials;

maximize profit : sum{p in P} price[p] * x[p];

solve;

end;
```

Далее необходимо задать значения параметров, указанные в условии задачи. Для этого создадим файл `example.dat` и добавим в него следующий текст:

```
data;
# Цена по видам продукции, руб.
param price := 1 1200 2 2500 3 1400;
# Спрос в неделю по видам продукции, шт.
param demand := 1 35 2 25 3 30;
# Расход материала на производство продукции, кг
param cost := 1 0.8 2 0.6 3 0.7;
# Производительность цехов, шт. в неделю
param prod := 1 40 2 20;
# Затраты времени на единицу продукции, часы
param prod_m_cost default 0 := [3, 2] 8 [3, 3] 2;
# Фонд рабочего времени, часы
param prod_m := 3 80;
# Поставки материалов в неделю, кг
param materials := 50;
end;
```

Для решения задачи следует:

а) перейти в директорию, где находится `glpsol.exe` – исполняемый файл GLPK или убедиться в том, что эта директория находится в списке, задаваемом переменной окружения `PATH`;

б) набрать в консоли (при условии, что `example.mod` и `example.dat` находятся в текущей директории):

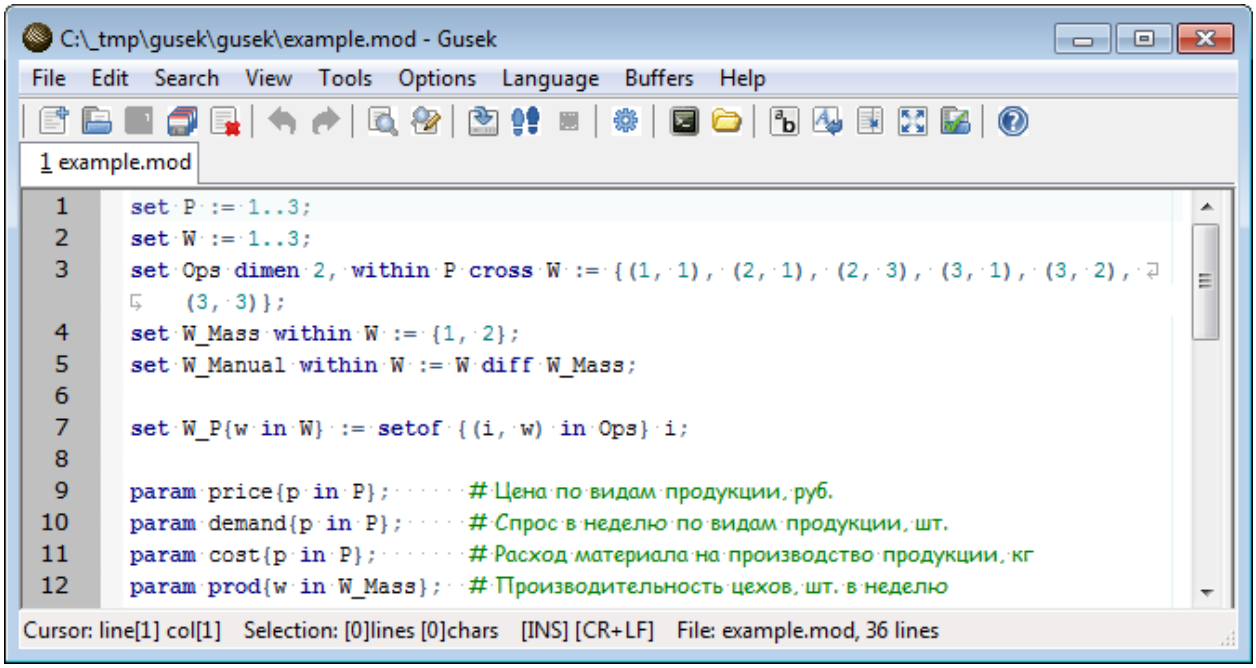
```
> glpsol –m example.mod –d example.dat –o example.sol
```

С помощью параметров `–m` и `–d` указывается размещение файлов модели и данных соответственно. Параметр `–o` служит для задания имени файла, в которой должны быть помещены сведения о найденном решении. Если параметр `–o` не указан, то файл с решением создан не будет и

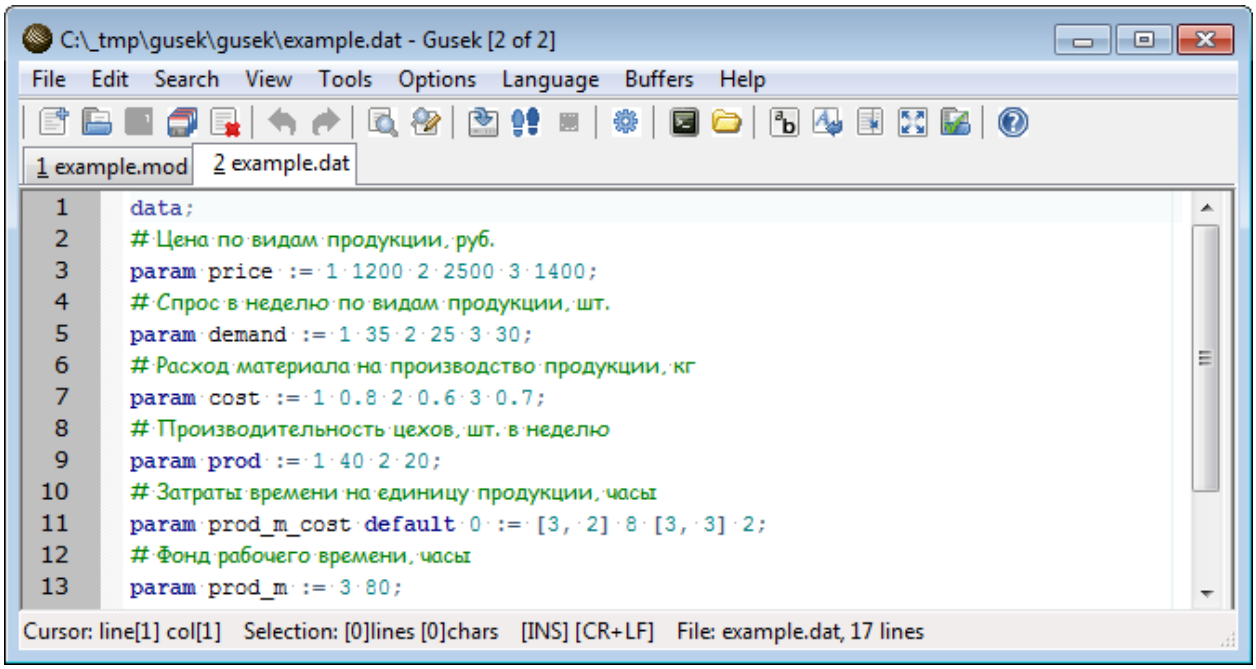
единственным способом выяснить оптимальные значения переменных будет использование в файле модели предложений `display`, `printf` и `table`.

Для автоматизации процесса формирования командной строки можно воспользоваться свободно распространяемой оболочкой GUSEK. Последовательность действий следующая:

а) Открыть файл модели (`File | Open…`).



б) Открыть файл данных (`File | Open…`).



в) В меню `Tools` отметить пункт «`Use External .dat`». В противном случае GUSEK попробует запустить `glpsol`, передав ему только файл модели, что неизбежно закончится ошибкой, если этот файл не содержит определения всех параметров. Если же значения параметров заданы в файле модели (.mod), то отметку напротив этого пункта необходимо снять.

г) Убедиться, что в меню `Tools` отмечен пункт «`Generate Output File on Go`». В противном случае GUSEK не добавит параметр `-o` при вызове `glpsol` и сведения о решении получить будет невозможно.

д) Запустить решение: `Tools | Go`.

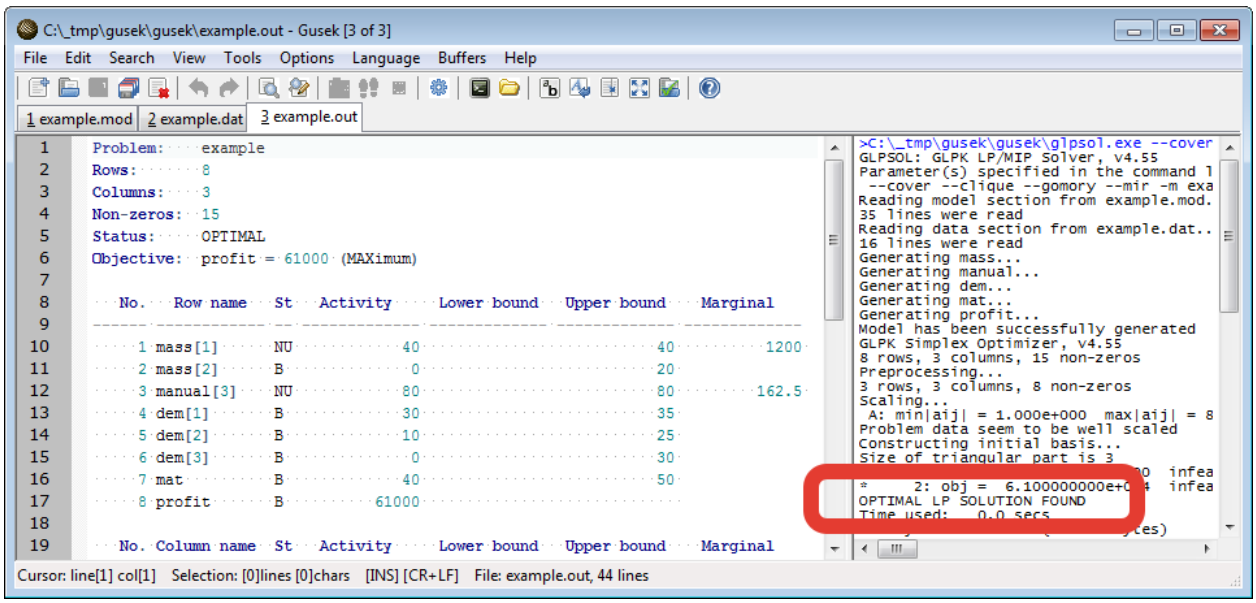


Рисунок 2 – Результаты решения задачи в GUSEK

В результате в правой части будет отображено содержимое стандартного потока вывода, полученного от `glpsol`, в том числе, статус решения (он обведен на рисунке 2). В рассматриваемом примере задача имеет решение, о чем свидетельствует строка “OPTIMAL LP SOLUTION FOUND”. Если файл модели содержит предложения `display` или `printf` (без перенаправления потока вывода), то результат обработки этих предложений также будет в правой части после сведений об использованном времени и памяти.

В левой части будет автоматически открыт еще один файл, содержащий найденное решение в текстовой форме.

В файле с решением можно выделить три основных блока: 1) общая информация о решении, 2) информация о значениях переменных и ограничений в оптимальном решении, включая теневые и приведенные цены, 3) условия оптимальности Каруша-Куна-Таккера.

В первую очередь, нас интересует второй блок, рассмотрим его более подробно. Этот блок, в свою очередь включает две таблицы: в первой содержится информация о состоянии ограничений, во второй – о состоянии переменных. Перечень столбцов для ограничений и переменных приблизительно одинаков, назначение их следующее:

- `No.` – порядковый номер ограничения (переменной);
- `Row name` – название ограничения, заданное в файле модели;
- `Column name` – название переменной, заданное в файле модели;
- `St` – статус ограничения или переменной:
  - `B` – неактивное ограничение (в первой таблице) / базисная переменная (во второй таблице);
  - `NL` – неравенство с активным ограничением снизу (>=) / небазисная переменная, значение которой находится на нижней границе;
  - `NU` – неравенство с активным ограничением сверху (<=) / небазисная переменная, значение которой находится на верхней границе;
  - `NS` – активное ограничение равенства / фиксированная небазисная переменная;
  - `NF` – активное свободное ограничение / свободная небазисная переменная.
- `Activity` – значение левой части ограничения или значение переменной;
- `Lower bound` и `Upper bound` – нижняя и верхние границы для ограничений и переменных;
- `Marginal` – теневая цена (shadow price) для ограничений и приведенная цена (reduced cost) для переменных.

Содержание таблиц, помещенных в результате решения в файл `example.out`, следующее:

```
   No.   Row name   St   Activity     Lower bound   Upper bound    Marginal
------ ------------ -- ------------- ------------- ------------- -------------
```

```
     1 mass[1]      NU             40                            40           1200
     2 mass[2]      B               0                            20
     3 manual[3]    NU             80                            80          162.5
     4 dem[1]       B              30                            35
     5 dem[2]       B              10                            25
     6 dem[3]       B               0                            30
     7 mat          B              40                            50
     8 profit       B           61000

   No. Column name  St   Activity     Lower bound   Upper bound    Marginal
------ ------------ -- ------------ ------------- ------------- -------------
     1 x[1]         B              30             0
     2 x[2]         B              10             0
     3 x[3]         NL              0             0                        -125
```

Таким образом, оптимальное значение целевой функции (строка `profit` – по имени целевой функции, введенному в файле модели) 61000 достигается при производстве 30 штук продукции первого вида (значение из столбца `Activity` напротив переменной, соответствующей плану выпуска первой продукции – `x[1]`) и 10 штук продукции второго вида (аналогично, для переменной `x[2]`).

Переменная `x[3]` помечена статусом `NL` (небазисная переменная, значение которой находится на нижней границе); действительно, в решении эта переменная принимает значение 0 (столбец `Activity`), а это является нижней границей области допустимых значений неотрицательной переменной. Поскольку данная переменная является небазисной, она имеет ненулевую приведенную цену -125 (столбец `Marginal`). То есть, увеличение значения переменной `x[3]` на единицу повлечет за собой снижение значения целевой функции на 125 единиц.

Судя по значениям столбца `St` первой таблицы, из всех ограничений задачи активными являются два: `mass[1]` (ограничение производительности первого цеха) и `manual[3]` (ограничение производительности третьего цеха). Статус `NU` (неравенство с активным ограничением сверху) прекрасно согласуется с тем, что значение левой части ограничения (столбец `Activity`) для данных ограничений совпадает с верхней границей (столбец `Upper bound`). Столбец `Marginal` содержит теневые цены для этих ограничений (1200 и 162,5 соответственно).

Анализ чувствительности

Несмотря на то, что основные показатели, используемые при анализе чувствительности – теневую и приведенные цены – можно извлечь из таблицы результатов, в GLPK предусмотрен специальный вид отчета по анализу чувствительности, содержащий расширенную информацию. В частности, в этом отчет можно использовать для определения интервалов осуществимости, то есть, границ, в которых могут изменяться правые части ограничений и коэффициенты целевой функции без изменения состава базисных переменных.

Для создания такого отчета при вызове `glpsol` следует указать параметр `--ranges`, задав имя файла, куда отчет должен быть помещен. Например, при выполнении следующей команды отчет по анализу чувствительности будет помещен в файл `example_sens.sol`:

```
> glpsol –m example.mod –d example.dat --ranges example_sens.sol –o example.sol
```

При использовании интерфейса GUSEK следует отметить в меню пункт `Tools | Generate LP Sensitivity Analysis`.

Пример отчета приведен на Рисунке 3.

```
GLPK 4.55 - SENSITIVITY ANALYSIS REPORT                                         Page   1

Problem:    meth
Objective:  profit = 61000 (MAXimum)

   No. Row name     St      Activity         Slack   Lower bound       Activity      Obj coef  Obj value at Limiting
                                          Marginal   Upper bound          range         range   break point variable
------ ------------ -- ------------- ------------- -------------  ------------- ------------- ------------- ------------
     1 mass[1]      NU      40.00000             .          -Inf       10.00000   -1200.00000   25000.00000 x[1]
                                       1200.00000      40.00000       45.00000          +Inf   67000.00000 dem[1]

     2 mass[2]      BS             .      20.00000          -Inf              .          -Inf   61000.00000
                                              .      20.00000       30.00000     125.00000   61000.00000 x[3]
```

```
     3 manual[3]    NU      80.00000         .              -Inf       40.00000    -162.50000   54500.00000 dem[1]
                                        162.50000       80.00000      200.00000          +Inf   80500.00000 dem[2]

     4 dem[1]       BS      30.00000       5.00000          -Inf       15.00000    -166.66667   56000.00000 x[3]
                                             .           35.00000       40.00000    1300.00000  100000.00000 manual[3]

     5 dem[2]       BS      10.00000      15.00000          -Inf        5.00000    -500.00000   56000.00000 x[3]
                                             .           25.00000       10.00000          +Inf          +Inf

     6 dem[3]       BS         .          30.00000          -Inf          .               -Inf   61000.00000
                                             .           30.00000       20.00000     125.00000   61000.00000 x[3]

     7 mat          BS      40.00000      10.00000          -Inf       10.00000   -1200.00000   13000.00000 mass[1]
                                             .           50.00000       40.00000          +Inf          +Inf

     8 profit       BS   61000.00000  -61000.00000          -Inf    25000.00000      -1.00000         .      mass[1]
                                             .              +Inf    61000.00000          +Inf          +Inf

GLPK 4.55 - SENSITIVITY ANALYSIS REPORT                                                                 Page   2

Problem:    meth
Objective:  profit = 61000 (MAXimum)

   No. Column name  St      Activity      Obj coef   Lower bound       Activity      Obj coef  Obj value at Limiting
                             Marginal   Upper bound          range         range   break point variable
------ ------------ -- ------------- ------------- -------------  ------------- ------------- ------------ ------------
     1 x[1]         BS      30.00000    1200.00000         .             15.00000    1033.33333   56000.00000 x[3]
                                             .              +Inf       35.00000    2500.00000  100000.00000 manual[3]

     2 x[2]         BS      10.00000    2500.00000         .              5.00000    2000.00000   56000.00000 x[3]
                                             .              +Inf       10.00000          +Inf          +Inf

     3 x[3]         NL         .        1400.00000         .             -6.66667          -Inf   61833.33333 dem[1]
                                        -125.00000          +Inf       20.00000    1525.00000   58500.00000 mass[2]

End of report
```

Рисунок 3 – Пример отчета по анализу чувствительности

Анализ чувствительности активных границ

Такого рода анализ проводится только для активных ограничений. Для каждого такого ограничения в ходе исследования активная граница (правая часть ограничения) изменяется как в сторону уменьшения, так и в сторону увеличения. Каждому ограничению в отчете соответствует две строки: первая характеризует результат, полученный при уменьшении активной границы, вторая – при увеличении. Изменение активной границы ведет к изменению значений переменных и продолжается до тех пор, пока какая-либо из базисных переменных не достигает границы области допустимых значений. Соответствующее значение активной границы называется в GLPK break point и по сути является границей интервала осуществимости решения по отношению к исследуемому ограничению.

В отчете по анализу чувствительности диапазон исследуемых значений границы приводится в столбце `Activity range`. Соответствующие значения целевой функции указаны в столбце `Obj value at break point`, а символические имена ограничивающих базисных переменных – в столбце `Limiting variable`. Если активную границу можно увеличивать или уменьшать бесконечно, то в столбце `Activity range` будет указано `+Inf` или `-Inf` соответственно, а столбец `Limiting variable` будет пуст.

Например, ограничение `mass[1]` является активным в верхней границе (статус `NU`), по условию задачи равной 40 (`Upper bound`). Соответственно, в ходе анализа чувствительности эта граница изменялась в сторону уменьшения до тех пор, пока при значении 10 переменная `x[1]` не приняла граничного значения (стала равной нулю), и в сторону увеличения до тех пор, пока при значении 45 не активизировалось ограничение `dem[1]` (спрос на продукцию первого вида). Соответствующие значения целевой функции – 25000 и 67000.

Анализ чувствительности коэффициентов целевой функции при небазисных переменных

Этот вид анализа является достаточно простым, поскольку изменение коэффициента целевой функции при небазисной переменной ведет к эквивалентному изменению ее приведенной цены.

Для каждой небазисной переменной коэффициент при этой переменной в целевой функции изменяется как в сторону уменьшения, так и в сторону увеличения. Каждой переменной в отчете соответствует две строки: первая описывает результат, полученный при уменьшении

коэффициента, вторая – при увеличении. Изменение коэффициента ведет к изменению приведенной цены анализируемой переменной и влияет на приведенные цены других переменных. Базисное решение остается допустимым и оптимальным до тех пор, пока приведенная стоимость сохраняет свой знак.

В отчете по анализу чувствительности диапазон значений коэффициента выводится в столбце `Obj coef range`. Если коэффициент можно увеличивать или уменьшать бесконечно без смены базиса, то в столбце `Obj coef range` будет указано `+Inf` или `-Inf` соответственно.

Например, переменная `x[3]` – небазисная переменная, принимающая в решении наименьшее допустимое значение (статус `NL`). Коэффициент при этой переменной в исходной задаче равен 1400, а приведенная стоимость равна -125 (столбец `Obj coef/Marginal`). При уменьшении коэффициента при `x[3]` потенциальный вклад `x[3]` в целевую функцию становится еще меньше; что отражается и в значении `-Inf` в столбце `Obj coef range`. То есть, можно сколько угодно уменьшать данный коэффициент без изменения базиса. Возвращаясь к содержательной интерпретации задачи – уменьшение цены на третий вид продукции не приведет к пересмотру производственного плана, в частности, к включению этого вида продукции в план. При увеличении же коэффициента до 1525 состав базисных переменных изменяется и разница между исходным значением коэффициента и пороговым равна 125, то есть, объяснимо совпадает с приведенной ценой переменной `x[3]`.

Анализ чувствительности коэффициентов целевой функции при базисных переменных

Для каждой базисной переменной коэффициент при этой переменной в целевой функции изменяется как в сторону уменьшения, так и в сторону увеличения. Каждой переменной в отчете соответствует две строки: первая описывает результат, полученный при уменьшении коэффициента, вторая – при увеличении. Изменение коэффициента ведет к изменению приведенной цены небазисных переменных. При определенном изменении какая-то из цен становится равной нулю, а это означает, что даже малейшее дальнейшее изменение приведет к включению в базис соответствующей переменной.

В отчете по анализу чувствительности диапазон значений коэффициента выводится в столбце `Obj coef range`. Соответствующие значения целевой функции приведены в столбце `Obj value at break point`, символические имена ограничивающих небазисных переменных – в столбце `Limiting variable`, а значения исследуемой переменной, которые она принимает на концах диапазона – в столбце `Activity range`. Если коэффициент можно увеличивать или уменьшать бесконечно без смены базиса, то в столбце `Obj coef range` будет указано `+Inf` или `-Inf` соответственно.

Например, переменная `x[1]` является базисной и принимает в решении значение 30 (столбец `Activity`), а коэффициент в целевой функции при этой переменной равен 1200 (столбец `Obj coef/Marginal`). Анализ показал, что переменная остается базисной при коэффициенте в целевой функции не ниже 1033.33333, при дальнейшем снижении коэффициента `x[1]` выйдет из базиса и ее место займет `x[3]`. Значение переменной `x[1]` при граничном значении коэффициента будет равно 15 (столбец `Activity range`), а значение целевой функции – 56000 (столбец `Obj value at break point`). Увеличивать коэффициент при `x[1]` без смены базиса можно вплоть до значения 2500. Значения переменных при этом будут изменяться, значение целевой функции возрастет до 100000, но состав ненулевых переменных и активных ограничений будет неизменен. Дальнейшее повышение коэффициента ограничено `manual[3]`. Обратим внимание на то, что ограничение `manual[3]` в решении является активным. Это ограничение является ограничением нестрогого неравенства, а значит, при переходе к канонической форме в него должна быть добавлена дополнительная переменная, значение которой равно разнице между левой и правой частями исходного ограничения. При активном ограничении левая и правая части равны, а значит, дополнительная переменная не содержится в базисе. По всей видимости, при повышении коэффициента целевой функции при `x[1]` выше 2500 дополнительная переменная входит в базис, то есть, левая его часть соответствующего ограничения становится строго меньше правой. С точки зрения интерпретации, это означает, что при ценах на продукцию первого вида выше 2500 план производства изменяется таким образом (видимо, в пользу продукции первого вида), что третий цех становится недогруженным.

Python

Scipy

Scipy – это популярная библиотека для языка программирования Python, предназначенная для выполнения научных и инженерных расчётов. Библиотека включает компоненты для решения дифференциальных уравнений, интегрирования, обработки изображений, визуализации, оптимизации и многих других задач. В частности, для решения задач линейного программирования Scipy предлагает функцию `linprog` (входящую в компонент `optimize`):

```
scipy.optimize.linprog(c, A_ub=None, b_ub=None, A_eq=None, b_eq=None,
                       bounds=None, method='simplex', callback=None,
                       options={'disp': False,
                                'bland': False,
                                'tol': 1e-12,
                                'maxiter': 1000})
```

Функция решает следующую задачу оптимизации:

$$c^T x \to min$$
$$A_{ub} x \leq b_{ub}$$
$$A_{eq} x = b_{eq}$$
$$x \geq 0$$

То есть, она поддерживает только два вида ограничений; впрочем, не поддерживаемые явно ограничения «≥» легко преобразуются в «≤», так что сложностей это не добавляет.

В соответствии с синтаксисом языка Python, те параметры, в определении которых есть оператор присваивания (например, `=None`), имеют значение по умолчанию, которое будет использовано в том случае, если при вызове функции данный параметр не будет задан. Таким образом, функцию `linprog` можно вызвать с единственным параметром `c`.

В качестве представления для векторов и матриц, передаваемых функции `linprog`, можно использовать как стандартные списки Python, так и многомерные массивы `ndarray`, оптимизированные для вычислений, определяемые библиотекой `NumPy`.

Описание параметров:

`c` – коэффициенты целевой функции;

`A_ub` – двумерный массив `ndarray` (или список списков) с коэффициентами ограничений – верхних границ («≤»);

`b_ub` – правая часть ограничений – верхних границ;

`A_eq` – двумерный массив `ndarray` (или список списков) с коэффициентами ограничений равенства;

`b_eq` – правая часть ограничений равенства;

`bounds` – ограничения на значения переменных. Этот параметр может принимать одну из трех форм:

`None` – все переменные неотрицательны;

`(lb, ub)` – если параметр представляет собой один кортеж, то все переменные должны принимать значения в диапазоне [*lb*; *ub*] включительно;

`[(lb_0, ub_0), (lb_1, ub_1), …]` – если параметр представляет собой список, то в этом списке должно быть столько элементов, сколько переменных, и *i*-тый элемент списка задает диапазон значений *i*-той переменной;

`callback` – функция обратного вызова, которая должна иметь сигнатуру `callback(xk, **kwargs)`, где `xk` – это вектор с текущим решением, а `kwargs` – ассоциативный массив, содержащий следующие ключи:

`tableau` – текущая симплекс-таблица;

`nit` – номер текущей итерации;

`pivot` – разрешающий элемент, который будет использован для следующей итерации;

`phase` – фаза алгоритма (функция реализует двухфазный симплекс-метод);

`bv` – структурированный массив, содержащий строковое представление каждой переменной и ее текущего значения;

Возвращаемым значением является объект `scipy.optimize.OptimizeResult`, содержащий следующие поля:

x – массив значений переменных, доставляющих максимум целевой функции;

slack – значения дополнительных переменных. Каждая переменная соответствует ограничению-неравенству. Нулевое значение переменной означает, что соответствующее ограничение активно.

success – True, если функции удалось найти оптимальное решение;

status – статус решения:

0 – поиск оптимального решения завершился успешно;

1 – достигнут лимит на число итераций;

2 – задача не имеет решений;

3 – целевая функция не ограничена.

nit – количество произведенных итераций;

message – строка с описанием статуса.

Дополнительные опции, задаваемые через параметр options:

maxiter – лимит на количество итераций;

disp – если True, то выводить статус в стандартный поток вывода;

tol – порог, используемый при сравнении с нулем;

bland – если True, то использовать алгоритм Блэнда для выбора разрешающего элемента, позволяющее избежать возможного зацикливания симплекс-метода. В противном случае, выбирается элемент, который быстрее всего приведет к нахождению решения, однако, в редких случаях не исключается возможность зацикливания.

Пример:

В качестве примера решим с помощью linprog задачу, описанную в разделе Постановка задачи.

В данной задаче значение целевой функции необходимо максимизировать, а linprog решает задачу минимизации, следовательно, коэффициенты целевой функции нужно задать с обратным знаком. Все ограничения этой задачи являются ограничениями сверху, которые напрямую поддерживаются функцией, а значит, никаких преобразований производить не нужно. Ограничений типа равенства нет, значит, соответствующие параметры можно не задавать, поскольку для них и так есть значения по умолчанию:

```
from scipy.optimize import linprog

c = [-1200, -2500, -1400]

A_ub = [[1,   1,   1],
        [0,   0,   1],
        [0,   8,   2],
        [1,   0,   0],
        [0,   1,   0],
        [0,   0,   1],
        [0.8, 0.6, 0.7]]

b_ub = [40, 20, 80, 35, 25, 30, 50]

print(linprog(c, A_ub, b_ub))
```

Исполнение скрипта приведет к выводу функцией print текстового представления всех полей объекта scipy.optimize.OptimizeResult, возвращенного linprog:

```
success: True
     nit: 2
       x: array([ 30.,  10.,   0.])
  status: 0
     fun: -61000.0
 message: 'Optimization terminated successfully.'
   slack: array([  0.,  20.,   0.,   5.,  15.,  30.,  20.])
```

То есть, поиск оптимального решения завершился успехом (success, message) и потребовал всего двух итераций (nit). Оптимальные значения переменных – 30, 10 и 0 (x); значение целевой функции при этом -61000 (fun). Поскольку мы при формировании параметров для linprog

использовали коэффициенты, взятые с обратным знаком, то значение целевой функции оригинальной задачи (максимизации) 61000. Активными ограничениями являются первое и третье, потому что для них дополнительные переменные (`slack`) принимают нулевое значение.

Для того, чтобы проследить за последовательностью получения решения, можно определить функцию и передать ее в качестве параметра `callback`:

```
def print_iteration(x, **kwargs):
    print(kwargs)

linprog(c, A_ub, b_ub, callback=print_iteration)
```

CVXOPT

Библиотека CVXOPT[3] предназначена для решения более широкого класса задач – задач выпуклого программирования. Соответственно, в ней используются более универсальные алгоритмы, за счет чего время решения аналогичных задач может оказаться несколько больше, чем у специализированных библиотек, реализующих разновидности симплекс-метода.

Функция `solvers.lp`

Для различных классов задач выпуклого программирования в CVXOPT предназначены различные функции. Так, для решения задач линейного программирования предназначена функция `lp`, определенная в модуле `solvers`, являющаяся «оберткой» для другой функции `conelp` (решение задач конического программирования методом внутренней точки). Однако `lp` позволяет использовать и сторонние решатели, в том числе, уже знакомый нам GLPK.

Интерфейс функции следующий:

```
solvers.lp(c, G, h, [, A, b [, solver [, primalstart [, dualstart] ] ] ])
```

Функция решает прямую и двойственную задачи линейного программирования:

$$\begin{gathered} c^T x \to min \\ Gx + s = h \\ Ax = b \\ s \geq 0 \end{gathered} \qquad \begin{gathered} -h^T z - b^T y \to max \\ G^T z + A^T y + c = 0 \\ z \geq 0 \end{gathered}$$

Параметры функции `lp`, взятые в квадратные скобки, в соответствии с широко распространенной нотацией, означает, что соответствующие параметры не являются обязательными. То есть, в простейшем случае функцию можно вызвать всего с тремя параметрами: `c`, `G`, и `h`.

Параметр `solver` позволяет задать вид решателя: если параметр не задан или имеет значение `None`, то используется функция `conelp`. Значением параметра также может быть строка `'glpk'` или `'mosek'` (второй вариант работает только в том случае, если установлен коммерческий пакет оптимизации MOSEK). При использовании внешнего решателя начальные значения переменных задавать нельзя; даже если соответствующие параметры указаны при вызове `lp`, они игнорируются.

Все параметры-матрицы задаются в виде экземпляров класса `matrix`, определенного в CVXOPT. Особенностью этого класса является представление матрицы по столбцам, то есть объект этого класса, созданный следующим кодом:

```
matrix([[1., 3., 5.], [2., 4., 6.]])
```

соответствует матрице

$$\begin{pmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \\ 5.0 & 6.0 \end{pmatrix}.$$

Другой, не вполне очевидной при первом знакомстве с библиотекой, особенностью использования функции `lp` является необходимость явно задавать все матрицы как матрицы вещественных чисел. Простейшим способом добиться этого является использование десятичной точки во всех числовых литералах, участвующих в задании матрицы. Другим способом является

[3] Для установки под Windows проще всего использовать неофициальный бинарный пакет: http://www.lfd.uci.edu/~gohlke/pythonlibs/#cvxopt

явное указание символа типа (‘d’ для вещественных матриц) в параметрах конструктора экземпляра матрицы. Так, матрица выше могла быть создана и следующим кодом:

```
matrix([[1, 3, 5], [2, 4, 6]], tc='d')
```

Или даже таким (здесь используется тот факт, что при наличии явно заданного параметра size (размер матрицы), само содержимое может быть простым одномерным списком, который будет преобразован в матрицу указанного размера по колонкам):

```
matrix([1, 3, 5, 2, 4, 6], (3,2), 'd')
```

Из приведенных формальных постановок задач, решаемых функцией, видно, что вводимые дополнительные переменные s всегда неотрицательны, следовательно, матрица `G` должна заполняться коэффициентами переменных в ограничениях сверху (“≤”). При наличии ограничений снизу (“≥”), коэффициенты и правую часть для них нужно брать с обратным знаком, изменив тем самым тип ограничения.

Функция `lp` не предполагает автоматически неотрицательности переменных, так что для ее обеспечения необходимо вводить явные ограничения.

Возвращаемым значением функции является ассоциативный массив, наиболее важными ключами которого являются следующие:

`status` - статус решения. Строка ‘optimal’ означает, что было найдено оптимальное решение;

`x` – значения переменных, при которых целевая функция достигает минимального значения;

`primal objective` – значение целевой функции;

`s` – значения дополнительных переменных прямой задачи. Могут быть использованы, например, для анализа того, какие ограничения являются активными;

`z` – значения переменных двойственной задачи, соответствующих ограничениям-неравенствам исходной задачи. Для ограничений неотрицательности переменных – это приведенные цены соответствующих переменных, а для всех остальных ограничений – теневые цены ресурсов;

`x` – значения переменных двойственной задачи, соответствующих ограничениям-равенствам исходной.

Пример:

В качестве примера решим с помощью `cvxopt.solvers.lp` задачу, описанную в разделе Постановка задачи.

В данной задаче значение целевой функции необходимо максимизировать, а `lp` решает задачу минимизации, следовательно, коэффициенты целевой функции нужно задать с обратным знаком. Все ограничения этой задачи являются ограничениями сверху, которые напрямую поддерживаются функцией, а значит, никаких преобразований производить не нужно. Ограничений типа равенства нет, значит, соответствующие параметры (`A` и `b`) можно не задавать, поскольку для них и так есть значения по умолчанию. Однако для обеспечения неотрицательности переменных необходимо ввести еще по одному ограничению для каждой из переменных; причем, поскольку эти ограничения снизу (≥ 0), то их необходимо преобразовать в ограничения сверху, взяв коэффициенты с обратным знаком. Получается следующий скрипт:

```
import numpy  # По некоторым причинам для используемой сборки CVXOPT под Windows
              # это оказывается важно
from cvxopt import matrix, solvers

c = matrix([-1200., -2500., -1400.])

G = matrix([[1, 0, 0, 1, 0, 0, 0.8, -1,  0,  0],
            [1, 0, 8, 0, 1, 0, 0.6,  0, -1,  0],
            [1, 1, 2, 0, 0, 1, 0.7,  0,  0, -1]], tc='d')
                                # ^^^^^^^^^^
                                #  -x_i <= 0
                                #
h = matrix([40, 20, 80, 35, 25, 30, 50, 0, 0, 0], tc='d')

solution = solvers.lp(c, G, h)
```

```
print('Status: ', solution['status'])
print('x = \n', solution['x'])    # Значения переменных
print('z = \n', solution['z'])    # Значения двойственных переменных
                                  # (Теневые цены и приведенные цены)
```

В результате выполнения этого скрипта в стандартном потоке вывода (обычно, консоль) сначала будет распечатан краткий отчет о ходе решения задачи, а потом появится результат работы функций `print`, выводящих компоненты найденного решения:

```
Status:  optimal
x =
 [ 3.00e+01]
[ 1.00e+01]
[ 9.82e-08]

z =
 [ 1.20e+03]
[ 5.74e-07]
[ 1.63e+02]
[ 6.09e-07]
[ 7.65e-07]
[ 4.84e-07]
[ 5.95e-07]
[ 3.96e-07]
[ 1.09e-06]
[ 1.25e+02]
```

Видно, что полученное решение, в целом, совпадает с решениями этой задачи, полученными другими средствами, за исключением несколько меньшей точности, что особенно заметно на переменных, равных нулю в оптимальном решении – здесь такие переменные имеют небольшие по модулю, но не нулевые значения (около 10^{-9}-10^{-6}). Привлекает также внимание теневая цена третьего ограничения (1.63e+02 = 163), которая при других способах решения равнялась 162.5. Однако это лишь следствие округления – вывод третьего элемента вектора `z` дает результат 162.5000008500826. Меньшая точность является результатом того, что CVXOPT решает более общую задачу конического программирования, используя для этого более общий алгоритм. Если же при вызове функции `lp` явно указать, что должен использоваться внешний решатель GLPK

```
solution = solvers.lp(c, G, h, solver='glpk')
```

то результат в точности совпадет с полученными ранее.

Python как среда моделирования

Помимо низкоуровневых функций для решения разных видов задач выпуклого программирования, библиотека CVXOPT предоставляет набор инструментов, позволяющих использовать Python в качестве среды моделирования задач выпуклого программирования, то есть для описания таких задач в форме, удобной понимания и обработки человеком. В данном пособии приводятся лишь базовые сведения о таком использовании CVXOPT; заинтересованному читателю рекомендуется обратиться к соответствующему разделу Руководства пользователя этой библиотеки [5].

По аналогии с тем, как это делается в GMPL – другом языке моделирования, описанном в данном пособии – для задания модели задачи необходимо определить переменные, ограничения и целевую функцию. Следует заметить, что уровень абстракции инструментов моделирования CVXOPT несколько ниже, чем в GMPL – в частности, здесь невозможно задавать множества и параметры. С другой стороны, в CVXOPT есть возможность задавать элементы модели в матричном виде, что может оказаться весьма удобно.

Переменная задачи определяется с помощью класса `cvxopt.modeling.variable`. Конструктор класса содержит два необязательных параметра – размерность и имя переменной:

```
x = variable()         # скалярная переменная
y = variable(5)        # вектор из пяти переменных
```

За счет того, что арифметические и логические операции для класса `variable` являются перегруженными, ограничения могут определяться как логические выражения с одним из операторов **“<=”**, **“==”**, **“>=”**, операндами которого являются линейные выражения относительно объектов `variable`. В том числе, могут использоваться и векторные операции:

```
cnstr1 = (x >= y[0] + y[1])
cnstr2 = (y <= 5)
```

Так, ограничение `cnstr2` означает, что каждый из пяти компонентов вектора `y` не должен быть больше 5.

Функция `cvxopt.modeling.op` по переданным в качестве параметров целевой функции и набору ограничений создает объект модели:

```
problem = op(5*x, [cnstr1, cnstr2])
```

Нахождение минимального значения целевой функции при указанных ограничениях производится методом `solve()` объекта модели. По умолчанию используется тот же алгоритм, что и в `cvxopt.solvers.lp()`, однако, как и в упомянутой функции, можно явно указать, что решение должно производиться одним из поддерживаемых CVXOPT внешних решателей – GLPK или MOSEK:

```
problem.solve()
```

После вызова метода `solve()`, статус решения можно получить через поле `status` объекта модели, а в случае успешного нахождения оптимального решения, объекты-переменные приобретают соответствующие значения. Получить же информацию о значении двойственных переменных при таком способе решения оказывается невозможно.

Пример:

В качестве примера решим с помощью `cvxopt.modeling` задачу, описанную в разделе Постановка задачи.

```
import numpy  # По некоторым причинам для используемой сборки CVXOPT под Windows
              # это оказывается важно
from cvxopt.modeling import variable, op

x = variable(3, 'x')

mass1 = (x[0] + x[1] + x[2] <= 40)
mass2 = (x[2] <= 20)
manual3 = (8*x[1] + 2*x[2] <= 80)
demand1 = (x[0] <= 35)
demand2 = (x[1] <= 25)
demand3 = (x[2] <= 30)
material = (0.8*x[0] + 0.6*x[1] + 0.7*x[2] <= 50)
x_non_negative = (x >= 0)

problem = op(-1200*x[0] -2500*x[1] - 1400*x[2],
             [mass1,
              mass2,
              manual3,
              demand1,
              demand2,
              demand3,
              material,
              x_non_negative])
problem.solve(solver=’glpk’)

problem.status
print(problem.objective.value())
print(x.value)
```

Заключение

В пособии рассмотрены основные виды инструментов, используемых для решения задач линейного программирования. Для каждого из инструментов показано, как подготовить задачу к решению, как интерпретировать результат решения, и как провести анализ чувствительности решения к изменению исходных данных задачи, если инструмент поддерживает такую возможность.

Одним из критериев отбора описываемых инструментов была возможность свободного доступа к ним. Вместе с тем, в категории специализированных инструментов для решения задач линейного и выпуклого программирования существует ряд мощных коммерческих продуктов – IBM ILOG CPLEX, Gurobi Optimizer, MOSEK и другие. В качестве следующего этапа заинтересованному читателю рекомендуется знакомство с одним из таких продуктов. Изучение материала настоящего пособия закладывает хорошую базу для этого, поскольку языки математического моделирования, поддерживаемые перечисленными продуктами, отчасти схожи с рассмотренным здесь языком GMPL.

Литература

1. Сервер численной оптимизации NEOS: статистика использования. URL: http://www.neos-server.org/neos/report.html (время доступа 16.11.2015)
2. Е.Р. Алексеев, О.В. Чеснокова Введение в Octave для инженеров и математиков, М.: ALT Linux, 2012. – 368 с.
3. J. W. Eaton, GNU Octave Manual. URL: https://www.gnu.org/software/octave/doc/interpreter/ (время доступа 20.02.2016)
4. Fourer, Robert; Gay, David M.; Kernighan, Brian W. (2002). AMPL: a Modelling Language for Mathematical Programming (2 ed.). Duxbury. Web: http://ampl.com/resources/the-ampl-book/chapter-downloads/ (время доступа 24.02.2016)
5. CVXOPT Modeling. URL: http://cvxopt.org/userguide/modeling.html