**Общество с ограниченной ответственностью “Квантовые системы”**
**123112, г. Москва, Пресненская наб., д. 12, эт. 40, офис 12, info@qusolve.ru, +7 (964) 591 - 36 35**
**ОГРН 1157746910441, ИНН 7727269328**

# OptJet

## Руководство пользователя программного обеспечения

**СОДЕРЖАНИЕ**

# 1. Введение

ПО “OptJet” поставляется в виде пакета для языка программирования Python, которое позволяет описывать задачи в декларативном виде, далее применяя различные оптимизационные ядра для их решения. После получения решения результаты интерпретируются/преобразуются к форме значений исходных переменных, в которых была описана задача.

## 2. Требования к установке

К требованиям по установке и запуску пакета относятся:

- установленный на ПК Python, который доступен из командной строки;
- версия Python на ПК соответствует версии, которая указана в инструкции по установке пакета.

Пакет с решателем имеет также платную версию, при приобретении которой пользователю предоставляется лицензионный ключ.

Лицензионный ключ предоставляет пользователю возможность решать более крупные задачи, так как снимает ограничения на время решения и количество переменных в задаче.

## 3. Функциональные возможности пакета

ПО “**OptJet**” позволяет решать оптимизационные задачи в двух направлениях::

- Программирование в ограничениях, выраженных в виде комбинации формул логики высказываний и псевдобулевых неравенств.
- Линейное (в том числе целочисленное) и квадратичное программирование.

В зависимости от типа задачи посредством ПО “**OptJet**” описание преобразуется в представление, пригодное для использования соответствующего ядра решателя.

Поддерживаются задачи высокой размерности (десятки миллионов переменных).

## 4. Работа с пакетом

Любая задача оптимизации формулируется путем задания условий на некоторые сущности, которые описываются переменными. В зависимости от особенностей этих сущностей выбираются булевы, дискретные (целые, возможно из некоторого диапазона) и непрерывные переменные.

ПО “**OptJet**” позволяет решать оптимизационные задачи в двух направлениях (см. пункт 3). Причем это могут быть как небольшие задачи с несколькими переменными, так и задачи высокой размерности.

Для установки пакета необходимо в командной строке, либо в терминале среды разработки ввести команду

```
pip install optjet
```

или

```
python3 -m pip install -U optjet
```

### 4.1. Описание настроек ПО “OptJet”

Ниже в таблице 1 приводится список основных настроек, задаваемых в словаре, передаваемом в менеджер решателя при его инициализации (qparams в примерах выше).

*Таблица 1 - Описание настроек решателя.*

| Настройка решателя | Описание |
|---|---|
| input_cnf_file | Строка, входной CNF или WCNF файл, в случае если его генерация из модели не требуется. |
| log_file_enable | bool, писать ли строки логов в файл, по умолчанию включено. |
| log_console_enable | bool, писать ли строки логов в консоль, по умолчанию выключено. |
| log_file_folder | Строка, путь к папке с логами, по умолчанию пишет в папку "./logs" относительно текущей директории. |

| | |
|---|---|
| log_time | bool, показывать ли в начале строки лога время, по умолчанию включено. |
| log_console_output_level | Строка (warn/info/debug), уровень логирования для консоли (терминала). По умолчанию "info". |
| log_file_output_level | Строка (warn/info/debug), уровень логирования в файл. По умолчанию "info". |
| pkey | Строка, лицензионный ключ. По умолчанию пустая строка. |
| cdst.algorithm_enable | bool, включить ядро CDST (SAT), по умолчанию выключено. |
| cdst.witness_output | bool, вывести ответ решателя CDST в логах (сырой результат без интерпретации, занимает много места), по умолчанию выключено. |
| cdst.witness_in_file | bool, вывести ответ решателя CDST в отдельный файл, по умолчанию выключено. |
| cdst.search_time_limit | int, лимит времени на поиск решения алгоритма CDST в секундах, по умолчанию 300. |
| glcs.algorithm_enable | bool, включить ядро GLCS (SAT), по умолчанию выключено. |
| glcs.witness_output | bool, вывести ответ решателя CDST в логах (сырой результат без интерпретации, занимает много места), по умолчанию выключено. |

| | |
|---|---|
| glcs.witness_in_file | bool, вывести ответ решателя CDST в отдельный файл, по умолчанию выключено. |
| glcs.search_time_limit | int, лимит времени на поиск решения алгоритма CDST в секундах, по умолчанию 150. |
| msat.algorithm_enable | bool, включить ядро MaxSAT (wcnf), по умолчанию выключено. |
| msat.use_c_solver | bool, использовать ядро GLCS (false) или CDST (true) в качестве SAT-оракула, по умолчанию false. |
| msat.use_c_solver | bool, использовать ядро GLCS (false) или CDST (true) в качестве SAT-оракула, по умолчанию false. |
| msat.witness_output | bool, вывести ответ в логе, по умолчанию выключено. |
| msat.witness_in_file | bool, вывести ответ в отдельный файл, по умолчанию включено. |
| msat.witness_hard_in_file | bool, вывести ответ решателя после проверки задачи на совместность (найденное неоптимальное решение) в отдельный файл. |
| msat.global_time_limit | int, лимит времени на поиск решения в секундах (общее время поиска с учетом всех стадий), по умолчанию 250. |
| msat.iter_exp_time_factor | float, с точностью до десятых. Экспоненциальный коэффициент для прерывания по времени решения, если на |

| | |
|---|---|
| | следующую итерацию с учетом глобального лимита времени остается меньше, чем время текущей итерации, умноженное на экспоненту данного параметра, то следующая итерация не стартует и возвращается текущее решение. По умолчанию 4.5. |
| msat.prefatory_use | bool, использовать предварительные значения переменных (warm start), по умолчанию выключено. |
| msat.preprocess_minimize | bool, включить (использовать) минимизацию условий на оценку значения целевой функции при поиске решения, по умолчанию выключено. |
| msat.preprocess_use | bool, включить препроцессор, по умолчанию выключено. |
| msat.core_g_linear_strategy | int из диапазона (1,2), стратегия поиска решения (1=линейный поиск с уточнениями за счет оценки выполнимости, 2=только линейный поиск). |

### 4.2. Описание задачи и ее структура

ПО “**OptJet**” может использоваться для **любой оптимизационной задачи, удовлетворяющей п.3 документа.** Для того, чтобы описать как работать с ПО “**OptJet**” мы рассмотрим его работу на примерах.

Для ознакомления с работой ПО “**OptJet**” рассмотрим параллельно (работать нужно в разных файлах!) два примера - “Игра 15” и модельную задачу оптимальной загрузки вагонов.

“Игра 15” - обычная задача выстроить фишки по порядку на поле 4x4.

Загрузка вагонов - есть некоторый набор вагонов различной грузоподъёмности, есть правила укладки грузов в эти вагоны в виде возможных групп SKU, идущих в один вагон и есть сами SKU. Есть стоимость отправки каждого из возможных вагонов. Нужно достаточно эффективно использовать привлекаемые вагоны, уложить по возможности большее число SKU и при этом минимизировать цену.

На первой задаче будем демонстрировать применение методов булевой выполнимости (SAT и MaxSAT), на второй - решение задачи смешанного целочисленного линейного программирования (MILP).

Сначала нужно импортировать необходимые библиотеки:

```
import optjet.quant_engine as qe
import optjet.md_encoder as enc
#Дополнительно для задач LP и MILP
from optjet.md_encoder import linear_parser as lp
from optjet.md_encoder.linear_parser import *
```

Также дополнительно необходимо инициализировать настройки решателя.

Для задачи SAT (выполнимость):

```
qmng = qe.manager()
qparams = {
# "pkey": # ключ для полной версии
"cdst.algorithm_enable": "true", #используемый алгоритм
}
qmng.init(qparams)
```

Для задачи MaxSAT (булева оптимизация):

```
qparams = {
# "pkey": # ключ для полной версии
"log_console_output_level": "info",
"msat.preprocess_use": "false",
"glcs.search_time_limit": "500",
"msat.global_time_limit": "600",
```

```
"msat.core_g_linear_strategy": "2",
"msat.iter_exp_time_factor": "0",
"msat.witness_in_file": "false"
}
qmng = qe.manager()
qmng.init(qparams)
```

Смысл и возможные настройки для параметров конфигурации описываются в п. 4.12.

Для задачи (смешанного целочисленного) линейного программирования:

```
qmng = qe.manager()
qparams = {
#"pkey": ...
}
qmng.init(qparams)
```

Обычно задачи используют некоторые параметры, которые задают конкретный сценарий работы. Это простые переменные Python различных типов.

Для задачи "Игра 15":

```
cnt_m = 81 # сколько ходов максимально рассматриваем. Любая
решаемая задача # решается за такое количество шагов
SIDE = 4 # Длина стороны
cnt_p = SIDE * SIDE # номер фишки (пустая - HOLE)
cnt_r = SIDE + 2 # строка, на которой находится фишка p на
# ходу m
cnt_c = SIDE + 2 # столбец, на котором находится фишка p на
# ходу m
HOLE = cnt_p - 1 # номер пустой фишки
# начальные условия
table = [[14, 0, 2, 3],
[11, 5, 7, 6],
[8, 9, 10, 4],
[12, 13, HOLE, 1]]
```

Для удобства задания условий игровое поле дополнительно окружено пустыми ячейками.

Для задачи погрузки в вагоны:

```
# Доступные грузоподъемности
M_w = [77.0, 75.0, 72.0, 71.9, 71.0, 70.6, 70.3, 70.0, 69.6,
69.5, 69.3, 69.0, 68.2, 68.0, 67.0, 66.0, 65.5, 65.0]
# Доступные количества SKU
N_t = [1, 1, 1, 1, 1, 1, 1]
# Доступное количество вагонов данной грузоподъемности
M_z = [0, 120, 0, 0, 0, 0, 0, 30, 0, 94, 0, 8, 0, 0, 0, 0, 0,
0]
# Стоимость использования одного вагона данной
# грузоподъемности
Price = [0, 23, 0, 0, 0, 0, 0, 21, 0, 21, 0, 21, 0, 0, 0, 0, 0,
0]
# Доступные варианты укладки SKU и масса варианта:[[номера SKU
# с нуля, соответствуют N_t], масса]
schemes = [[[2, 3], 49.0], [[2, 5], 52.95], [[3, 5], 53.01],
[[4, 6], 58.52]]
```

Далее создается объект, который будет собирать модель.

Для задачи выполнимости или булевой оптимизации:

```
model = enc.Model(qmng, "puzzle", False)
```

Для задачи линейного программирования:

```
model = lp.LinearModel(log_solver=True, presolve="on",
time_limit=Timeout)
```

Значение Timeout для примера можно задать равным 120 сек.

### 4.3. Задание переменных

Для задачи "Игра 15" введем переменные $s_m$, где m - номер шага (действия), начиная с нуля (как и все номера в Python) и флаги $x_{mprc}$, где m - номер шага, p - номер фишки, r, c - ряд и столбец игрового поля.

В коде это выглядит так:

```
s = model.get_variables("s", m=cnt_m) # s[m]=1, если на ходу m
# все фишки на своих местах
x = model.get_variables("x", m=cnt_m, p=cnt_p, r=cnt_r, c=cnt_c)
```

По каждому индексу переменные нумеруются с нуля до указанного значения минус единица (как range в Python).

Также декларируются индексы:

```
m, p, r, c = model.get_indexes(["m", "p", "r", "c"])
```

**Важно в коде не путать индексы (используемые как абстракции нумерации) и конкретные их значения в циклах и формулах.**

Для задачи погрузки в вагоны введем переменные $n_{iz}$ - число раз использования варианта укладки i в вагоне z и $\nu_t$ - количество использованных SKU типа (номера) t. В коде это будет:

```
T = len(N_t)
Z = len(M_z)
wagons_UB = int(max(M_z))
n = model.get_variables("n", lp.VariableType.INTEGER, lo=0,
hi=wagons_UB, i=N, z=Z)
nu = model.get_variables("nu", lp.VariableType.INTEGER, lo=0,
hi=max(N_t), i=T)
```

Заметим, что помимо индексов для переменных в этом случае также указываются допустимые диапазоны их изменения и типы.

Таким образом, для каждой задачи мы создали объект, в котором будет строиться модель, и задали основные сущности. Переходим к описанию ограничений.

### 4.4. Задание ограничений

Задание ограничений выполняется с помощью метода add_constraint объекта model, который несколько отличается для задач выполнимости и MILP.

Для задачи "Игра 15":

```
# Задачу всё же стоит решить хотя бы на последнем шаге
model.add_constraint(s[cnt_m - 1])
# Используются все фишки
model.add_constraint(Sum(x[m,   p,   r,   c],   p=range(cnt_p),
r=range(1, cnt_r - 1), \
c=range(1, cnt_c - 1)) == SIDE * SIDE, m=range(cnt_m))
# В суммах переменные (с коэффициентами) всегда слева, а все
# константные слагаемые справа от знака (не)равенства.
# Заметьте, что после условия равенства суммы константе есть
# еще один range по переменной m.
# Это универсальная квантификация, то есть условие
# автоматические повторяется для любого m из указанного
# диапазона. Только одна фишка в данной позиции на каждом шаге
model.add_constraint(Sum(x[m, p, r, c], p=range(cnt_p)) == 1,
m=range(cnt_m), r=range(1, cnt_r - 1), c=range(1, cnt_c - 1))

# Только одна фишка с данным p используется на поле
model.add_constraint(Sum(x[m, p, r, c], r=range(1, cnt_r - 1),
c=range(1, cnt_c - 1)) == 1, m=range(cnt_m), p=range(cnt_p))
# Пустая позиция перемещается, если не нашли решения
model.add_constraint(Implication(~s[m] & x[m, HOLE, r, c], ~x[m
+ 1, HOLE, r, c] & (x[m + 1, HOLE, r + 1, c] | x[m + 1, HOLE, r
- 1, c] | x[m + 1, HOLE, r, c + 1] | x[m + 1, HOLE, r, c - 1])),
m=range(cnt_m - 1), r=range(1, cnt_c - 1), c=range(1, cnt_c -
1))

# Начальные условия
for ri in range(1, cnt_r - 1):
     for ci in range(1, cnt_c - 1):
          model.add_constraint(x[0, table[ri - 1][ci - 1], ri,
          ci])
# Граничные условия
```

```
model.add_constraint(And(~x[m,  p,  r,  0],  m=range(cnt_m),
p=range(cnt_p), r=range(cnt_r)))

model.add_constraint(And(~x[m, p, r, SIDE + 1], m=range(cnt_m),

p=range(cnt_p), r=range(cnt_r)))
model.add_constraint(And(~x[m, p, 0, c], m=range(cnt_m),
p=range(cnt_p), c=range(cnt_c)))

model.add_constraint(And(~x[m, p, SIDE + 1, c], m=range(cnt_m),

p=range(cnt_p), c=range(cnt_c)))
# Решение найдено, если все фишки на своих местах
model.add_constraint(Equiv(s[m], AndIt([x[m, ci - 1 + SIDE * (ri
- 1), ri, ci]for ri in range(1, cnt_r - 1) for ci in range(1,
cnt_c - 1)])), m=range(cnt_m))

model.add_constraint(Implication(s[m], s[m + 1] &

And(Equiv(x[m + 1, p, r, c], x[m, p, r, c]), p=range(cnt_p),
r=range(cnt_r), c=range(cnt_c))),

m=range(cnt_m - 1))

# На шаге каждую фишку можно сдвинуть только на один шаг в
# сторону или не двигать
model.add_constraint(Implication(x[m, p, r, c], x[m + 1, p, r
- 1, c] & x[m + 1, HOLE, r, c] & x[m, HOLE, r - 1, c] | x[m +
1, p, r + 1, c] & x[m + 1, HOLE, r, c] & x[m, HOLE, r + 1, c]
| x[m + 1, p, r, c - 1] & x[m + 1, HOLE, r, c] & x[m, HOLE, r,
c - 1] | x[m + 1, p, r, c + 1] & x[m + 1, HOLE, r, c] & x[m,
HOLE, r, c + 1] | x[m + 1, p, r, c]), m=range(cnt_m - 1),
r=range(1, cnt_r - 1), c=range(1, cnt_c - 1), p=range(cnt_p -
1))
```

Полный список операций и конструкторов для логических формул приводится в пункте 4.10. Отметим различие между замкнутым и итеративным вариантом для

формулы And (то есть конструкторами And и AndIt). Замкнутый вариант автоматически осуществляет итерацию по всем указанным индексам не используя внешних параметров. Итеративный вариант AndIt принимает список переменных, у которых часть значений индексов вычисляется и может быть связана с внешним лексическим окружением. Итеративные варианты SumIt также могут быть использованы для включения в формулу внешних коэффициентов, которые берутся из массива.

Возвращаясь к предупреждению выше: **в итеративных вариантах нельзя использовать в вычислениях имена индексов**, если они связаны с контекстом каким-либо образом кроме размножения формулы по range или суммирования по аналогичному итератору. Проще говоря, имена индексов используются только если они будут связываться в виде ind=range в соответствующей позиции конструктора.

Ограничения задачи погрузки вагонов описываются так:

```
### Вспомогательные функции:
def wag_is_ok(mass, z) -> bool: # Вагон пригоден для
# использования схемы
      return 70.0 <= mass <= M_w[z] if z < 3 else 50.0 <= mass
<= M_w[z]

def calc_SKUs_in_schemes(): # Матрица вхождений - сколько SKU
# данного типа находится в данной допустимой группе
      print("Calculating SKU in schemes")
      res = np.zeros((N, T), dtype=np.int32)
      for i, scheme in enumerate(schemes):
            skus = scheme[0]
            pairs = Counter(skus)
            for s, c in pairs.items():
                  res[i, s] = c
      return res

def get_allowed_placements(): # Какие комбинации в какие типы
# вагонов можно разместить
      res_ok = []
      res_not_ok = []
      for i, scheme in enumerate(schemes):
            for z, m in enumerate(M_w):
```

```
                        if wag_is_ok(scheme[1], z) and M_z[z] > 0:
                              res_ok.append((i, z))
                        else:
                              res_not_ok.append((i, z))

            return res_ok, res_not_ok

      def get_max_scheme_cnt(scheme): # Сколько раз максимально
      # можно использовать данную комбинацию
            cnt_sku = min([N_t[sku] for sku in scheme[0]])
            wags = []
            for z in range(Z):
                  if M_z[z] > 0 and wag_is_ok(scheme[1], z):
                        wags.append(z)
            cnt_wag = min([M_z[z] for z in wags])
            return min(cnt_sku, cnt_wag)

      ### Ограничения
      N = len(schemes)

      C = calc_slabs_in_schemes()
      ok, not_ok = get_allowed_placements()

      wagons_UB = int(max(M_z))
      not_placed_penalty = max(Price)
      timeout = 60

      print("Starting model build")
      model    =    lp.LinearModel(log_solver=True,    presolve=”on”,
time_limit=Timeout)
      # См. выше
      n = model.get_variables("n", lo=0, hi=wagons_UB, i=N, z=Z,
      variable_type=lp.VariableType.INTEGER)
      nu = model.get_variables("nu", lo=0, hi=max(N_t), i=T,
      variable_type=lp.VariableType.INTEGER)
      for i, z in not_ok:
            model.add_constraint(n[i, z], lo=0, hi=0) # или
```

```
        # model.add_constraint(n[i, z] == 0)
        # синтаксис с lo hi удобен, когда выражение
        # ограничивается с обеих сторон
for i, z in ok:
        max_cnt = get_max_scheme_cnt(schemes[i])
        model.add_constraint(n[i, z], hi=max_cnt)

for t in range(T):
        model.add_constraint(-1 * nu[t] + lp.SumIt([C[i, t] *
        n[i, z] for i, z in ok]), lo=0, hi=0)
        model.add_constraint(nu[t], hi=N_t[t])

for z in range(Z):
        model.add_constraint(lp.SumIt([n[i, z] for i in range(N)
        if (i, z) in ok]), hi=M_z[z])
```

Важно заметить, что при задании линейных или псевдобулевых ограничений переменные всегда слева, а постоянные слагаемые - справа от знака неравенства.

### 4.5. Задание целевой функции

Возможны несколько сценариев. При решении задач выполнимости целевая функция не задаётся и обычно используется SAT-решатель.

Для “Игра 15” при использовании MaxSAT ядра отдельные "мягкие" дизъюнкты задаются по отдельности с весами:

```
for mm in range(cnt_m):
model.add_soft_constraint(s[mm]) # Чем больше s[m] выполнено,
# тем раньше решение найдено
```

Для задачи о вагонах в случае MILP:

```
model.add_target(-0.5 * lp.SumIt([nu[t] for t in range(T)]) +
lp.SumIt([Price[z] * n[i,z] for i, z in ok]), minimization=True)
```

### 4.6. Построение модели

Для задачи MILP модель собирается по мере интерпретации ограничений и целевой функции автоматически.

Для задач выполнимости требуется финальная сборка низкоуровневого представления модели.

```
model.build_cnf() # для задачи SAT
model.build_wcnf() # для задачи MaxSAT
```

### 4.7. Запуск и отладка

Запуск осуществляется по-разному для различных ситуаций.

Для задачи MILP:

```
model.solve()
```

В случае решения задачи булевой выполнимости или максимизации выбираются подходящие ядра:

```
#SAT или MaxSAT
rc = qmng.run()
# где "rc" - это число, которое является результатом решения
# 30 = optimum found, оптимум найден
# 10 = sat, решение найдено, но оно не оптимальное, если
# прервано таймерами.
```

### 4.8. Интерпретация результатов

Для решения переменные преобразуются во внутреннее представление соответствующего решателя, которое обычно является низкоуровневым и неудобным для непосредственного использования человеком. Для удобства реализован функционал обратного преобразования (интерпретации результатов), который приводит ответ к виду numpy массива в размерности исходных переменных.

Для задачи MILP:

```
n_res = model.get_result("n") # Используем имена данные в
# аргументах get_variables при создании переменных
nu_res = model.get_result("nu")
for i in range(N):
      for z in range(Z):
            val = n_res[i][z]
            if n_res[i][z] > 0:
                  print(f'Scheme {i} used {val} times in wagons
                  of type {z}')
```

Для “Игра 15” и в случае выполнимости, и в случае оптимизации:

```
wi = qmng.get_witness_interpreter()
res_x = wi.get_var_values("x")
res_s = wi.get_var_values("s")
s_res = None
if res_s is not None:
      s_res = res_s.get_ndarray()
      print("res_s : " + res_s.to_string())
      print("s_res:\n" + str(s_res))
else:
      # Решение не найдено
```

### 4.9. Оптимизация сборки модели

Если генерация происходит долго, то имеет смысл пересмотреть форму ограничений, уменьшить количество переменных и писать более общие формулы в случае (Max)SAT, чтобы использовать неитеративные версии конструкторов (And вместо AndIt и т.п.). Также имеет смысл повысить подробность журналирования и проанализировать тайминги и сообщения в логе.

### 4.10. Решение проблем

Стандартным способом отладки является метод дихотомии - ограничивать создание модели рядом условий и следить за поведением (скоростью генерации,

наличием решения) деля пополам подмножество ограничений, сходясь к проблемной точке.

Для задач (Max)SAT нужно при инициализации менеджера выключить опцию очистки накопленных дизъюнктов:

```
qparams = {
"msat.algorithm_enable": "true",
"cnf_storage_clean_clauses": "false",
}
qmng.init(qparams)
```

Затем можно вызывать метод build_cnf после любого add_constraint и запускать SAT-оракулов для оценки поведения, выделяя проблемные блоки ограничений.

```
# Выше закодирована часть модели
model.build_cnf()
status = qmng.run()
```

Код возврата (status в примере выше) принимает значения:

- 0 - статус не определён (не найдено целостное состояние решения задачи, например, не успела пройти инициализация внутри);
- 10 - SAT (задача разрешима, либо есть промежуточное решение для задачи MaxSAT);
- 20 - UnSAT (задача неразрешима);
- 30 - Найден оптимум задачи MaxSAT;
- 101 - Ошибка;
- 102 - Таймаут.

Для задач MILP - решать модель без части ограничений, сходясь к проблемным.

В случае проблем и невозможности установить их причину рекомендуется обратиться к поставщику.

### 4.11. Синтаксис, конструкции ограничений и целевой функции

#### Объявление модели

При формализации и передачи задачи решателю необходимо создать модель, в которую будут добавляться различные типы ограничений.

Конструктор класса модели Model для булевых задач имеет следующие параметры:

- manager — используемый менеджер выполнения;
- model_name — имя модели;
- Флаг алгоритма генерации (стабильный - False, быстрый - True).

Конструктор класса MILP модели LinearModel имеет следующие параметры:

- log_solver — флаг (True/False) включения логирования работы солвера в консоль. По умолчанию отключено.
- timeout — таймаут в секундах, заданных как вещественное положительное число.
- presolve — включает или выключает пресолвер, значения “on” или “off”. По умолчанию включено.

**Задание ограничений**

- add_constraint — добавить в модель “жесткое” ограничение. Все ограничения данного типа необходимо создать до определения мягких ограничений.
- add_soft_constraint — добавить в модель “мягкое” ограничение. Используется только для MaxSAT.

**Задание целевой функции**

Целевая функция задается методом линейной модели add_target.

**Создание и использование переменных и индексов**

Переменная (фактически - группа, многомерная матрица) определяется именем, измерениями и количеством элементов в них, т.е. индексами и диапазонами их изменения. Диапазоны являются отрезками натурального ряда, начинающимися с 0. Для булевых задач переменная - многомерный массив двоичных флагов {0, 1}.

Для MILP задач указывается диапазон изменения переменных (именованные параметры lo и hi), а также тип переменной

(variable_type=lp.VariableType.BINARY|INTEGER|REAL).

Индекс определяется своим именем. Приведем примеры различных способов создания индексов:

```
# Создание одиночного односимвольного индекса
i = model.get_indexes("i")
# Создание одиночного индекса с произвольной длиной имени
```

```
idx = model.get_indexes(["idx"])
# Создание группы односимвольных индексов
i, j, k = model.get_indexes("ijk")
# Создание группы односимвольных индексов с произвольной
# длиной имени
idx, j = model.get_indexes(["idx", "j"])
```

**Запись ограничений**

Для булевых задач SAT и MaxSAT парсер поддерживает смешанные булевы выражения, состоящие из булевых переменных и конструкций из них:

- И
- ИЛИ
- ИСКЛЮЧАЮЩЕЕ ИЛИ
- ОТРИЦАНИЕ
- ЭКВИВАЛЕНТНОСТЬ
- ИМПЛИКАЦИЯ
- УСЛОВНЫЙ ОПЕРАТОР

и псевдобулевых операций:

- СЛОЖЕНИЕ
- ВЫЧИТАНИЕ
- УМНОЖЕНИЕ НА КОНСТАНТУ

Приведем синтаксис для каждой их них

- first_expression + \ - second_expression - бинарная операция сложения или вычитания для псевдобулева выражения (линейная форма с булевыми переменными, целыми коэффициентами и правой частью), expression - допустимая комбинация переменных.
- const * expression - умножение на константу для псевдобулева выражения. expression - допустимая комбинация переменных, const - целая константа, которая может быть целым числом или выражением вида: pow_expression(base, exp), где base - целое положительное число, exp - доступный индекс или целое число, что необходимо для задания натуральных чисел в виде

$$\sum_l 2^l x_l$$

- Sum(expression, indexes) - суммирование по индексу для псевдобулева выражения. expression - допустимая комбинация переменных, indexes - набор индексов, по которым проводится суммирование (каждый задается как итерируемая последовательность в языке Python).
- SumIt(expressions) - суммирование последовательности псевдобулевых выражений. expressions - итерируемая последовательность допустимых комбинаций переменных (обычно - список).
- expression <= \ == \ >= const - сравнение для псевдобулева выражения. expression - допустимая комбинация переменных, const - константа (целое число). Сравнение обязательно должно быть использовано при применении псевдобулевых выражений (любое выражение, содержащее вышеприведенные операции).
- first_expression & \ | \ ^ second_expression - бинарная операция (И \ ИЛИ \ ИСКЛЮЧАЮЩЕЕ ИЛИ) для булевого выражения. expression - допустимая комбинация переменных.
- expression - унарная операция, отрицание для булевого выражения.
- And(expression, indexes) - логическое умножение по индексу для булевого выражения. expression - допустимая комбинация переменных, indexes - набор индексов, по которым проводится умножение (каждый задается как итерируемая последовательность в языке Python).
- AndIt(expressions) - логическое умножение последовательности булевых выражений. expressions - итерируемая последовательность допустимых комбинаций переменных.
- Or(expression, indexes) - логическое сложение по индексу для булевого выражения. expression - допустимая комбинация переменных, indexes - набор индексов, по которым проводится сложение (каждый задается как итерируемая последовательность в языке Python).
- OrIt(expressions) - логическое сложение последовательности булевых выражений. expressions - итерируемая последовательность допустимых комбинаций переменных.
- Xor(expression, indexes) - исключающее ИЛИ по индексу для булевого выражения. expression - допустимая комбинация переменных, indexes - набор индексов, по которым проводится операция (каждый задается как итерируемая

последовательность в языке Python). Задает выражение по форме схожее с полиномом Жегалкина, в котором вместо конъюнкций произвольное булево выражение.

- XorIt(expressions) - исключающее ИЛИ для последовательности булевых выражений. expressions - итерируемая последовательность допустимых комбинаций переменных.
- Equiv(expressions) - эквивалентность для булевого выражения. expressions - допустимые комбинации переменных.
- Implication(if_expression, then_expression) - импликация, if_expression - достаточное условие для выполнения then_expression, then_expression - необходимое условие для выполнения if_expression.
- IfThenElse(if_expression, then_expression, else_expression) - условный оператор, if_expression - проверяемое допустимое выражение, then_expression - допустимое выражение, которое необходимо удовлетворить, если проверяемое выражение истинно, else_expression - допустимое выражение, которое необходимо удовлетворить, если проверяемое выражение ложно.
- Zero - константа, соответствующая булевому False, в псевдобулевых выражениях необходимо использовать 0.
- One - константа, соответствующая булевому True, в псевдобулевых выражениях необходимо использовать 1.

Для MILP задач поддерживаются только линейные формы в ограничениях и конструкторы Sum и SumIt. Важно отметить, что условие на целочисленность коэффициентов и констант в правой части при этом снимается.

**Добавление ограничений в модель**

Добавление ограничений в модель производится в фиксированном порядке. Сначала добавляются все “жесткие” ограничения. Затем, если требуется, добавляются одно целевое ограничение, либо одно или несколько “мягких” ограничений (для задач MaxSAT).

Для псевдобулевых ограничений всегда в левой части выражения находится переменная часть линейной формы, а в правой - константа.

Для задач MILP возможно как задание условий на сравнение через >=, <=, ==, так и парное задание ограничений через именованые параметры lo, hi (нижняя и верхняя граница).

Для булевых задач у каждого из методов для добавления ограничений имеются следующие общие аргументы add_{soft}_constraint(constraint, foralls, write_type):

- constraint — добавляемое в модель выражение, описывающее ограничение.
- foralls — набор диапазонов для индексов, применяемых в операции «для любых». Задается как набор выражений вида <имя индекса>=<итерируемая последовательность>.

Все параметры, кроме constraint, являются необязательными. Использование foralls предпочтительно по производительности, но в случае выборочной работы с индексами допускается использование внешнего цикла со связыванием значений в конкретных позициях.

**Важно не использовать имена переменных-индексов в циклах и вычислениях при использовании модели вне описания ограничений constraint и foralls. Это может приводить к сложноуловимым ошибкам.**

Рассмотрим “жесткие” и “мягкие” ограничения:

1) Жесткие ограничения задаются при помощи метода add_constraint. В качестве constraint необходимо подавать смешанное булево выражение. Вызов данного метода после вызова add_target или add_soft_constraint приведет к ошибке.
2) Мягкие ограничения задаются при помощи метода add_soft_constraint. В качестве constraint необходимо подавать литерал (переменную или ее отрицание) в качестве выражения. Опционально задаются вес для данного выражения (по умолчанию равный 1) и направление работы - параметр minimization. В случае значения True для минимизации выражение обращается (автоматически навешивается отрицание) и фактически решается задача на минимум.

Приведем пример добавления целевого и мягких ограничений в модель:

```
# Добавление “мягких” ограничений
for ii in range(I):
      for ss in range(S):
            model.add_soft_constraint(y[ii,ss], cost[ii],
            minimization=True)
```

**Формирование модели**

Для булевых моделей формирование проводится двумя методами:

- build_cnf — создает модель в формате CNF. Для корректной работы требуется наличие хотя бы одного “жесткого” ограничения и отсутствие “мягких” ограничений.
- build_wcnf — создает файл в формате WCNF. Для корректной работы требуется наличие хотя бы одного “жесткого” ограничения, наличие хотя бы одного “мягкого” ограничения и отсутствие целевого ограничения.

Для задач MILP в силу более компактного представления модель хранится в памяти и строится моментально.

### 4.12. Алгоритм работы парсера

Создание КНФ (конъюнктивной нормальной формы, CNF) из заданных ограничений состоит из трех этапов: компиляция шаблонных выражений, вычисление необходимых смещений для переменных, формирование модели.

Введем несколько используемых далее терминов:

- Целевые или бизнес-переменные — набор многомерных переменных, созданный пользователем с помощью метода get_variables. Целевой переменной мы будет называться не весь массив, а конкретный экземпляр: x – массив целевых переменных, x[0, 3, 7] – целевая переменная.
- Дополнительные переменные — переменные, возникшие в ходе компиляции шаблонного выражения. Их появление обусловлено применением компактных по размеру результатов методов трансляции псевдобулевых выражений в КНФ, а также использованием преобразования Цейтина.
- Шаблонная переменная — целевая или дополнительная переменная, полученная в результате компиляции выражения для ограничения и преобразования его в КНФ. У целевой переменной в этом случае часть измерений может быть определена как индекс, который используется при подставлении выражений типа «для любых из». Такие переменные шаблонные в том смысле, что в зависимости от содержимого итераторов (range, foralls в конструкторах формул) они определяют не одну конкретную переменную, а массив.

КНФ представляется в формате DIMACS со сквозной нумерацией всех входящих переменных, начиная с единицы.

Далее будет описан алгоритм для компиляции одного из ограничений различных типов. Для всех ограничений задачи он применяется аналогично.

### Компиляция шаблонного выражения

Для ограничения на вход подается некоторое выражение, у которого часть значений индексов переменных может быть явно задана, а остальные относятся к выражению «для любого из» (перечисления foralls, range в формуле).

Компиляция выражения происходит в два этапа: сначала строится КНФ для шаблонного выражения (не связанные внешними привязками переменные рассматриваются как абстрактные позиции), а затем эта КНФ размножается по диапазонам не связанных внешне переменных с соответствующим тиражированием вспомогательных переменных и определением конкретных номеров для бизнес-переменных.

Первым этапом для каждой шаблонной переменной создаются вспомогательные переменные, возникающие при применении преобразований Цейтина и трансляции псевдобулевых ограничений в КНФ соответственно. Затем компиляция выражения производится в следующем порядке:

1) Собираются и компилируются псевдобулевы выражения. На выходе для них получаются КНФ.
2) Используя результаты шага 1 собирается общее выражение в булевой логике.
3) К полученному выражению применяется преобразование Цейтина, в результате которого получается шаблонная КНФ.

### Постобработка шаблонного выражения

После компиляции ограничения в полученном после преобразования Цейтина выражении существуют два вида шаблонных переменных: связанные с целевыми переменными и дополнительные.

Заметим, что при подстановке в шаблон индексов из foralls шаблонная целевая переменная обращается в существующую целевую переменную, для которой явно известен номер в глобальной системе переменных, а дополнительная шаблонная переменная становится новой дополнительной переменной для каждого конкретного набора индексов. То есть производится поочередная подстановка всех индексов foralls в шаблон с замещением шаблонных дополнительных переменных на конкретные дополнительные и фиксированием всех индексов у целевых переменных. При этом лексическая область видимости и привязки не нарушается.

Следовательно, для того, чтобы пронумеровать все переменные в глобальной системе можно использовать следующую раскладку: первыми будут идти целевые

переменные, затем дополнительные переменные для первого выражения, после для второго и так далее.

Внутри для каждого выражения дополнительные переменные делятся на pb и bx переменные (псевдобулевы и от преобразования Цейтина соответственно), которые укладываются в порядке pb и bx для первого набора foralls, pb и bx для второго и т.д.

Будем полагать, что переменные pb и bx пронумерованы подряд, начиная с нуля и их количество соответственно count_pb и count_bx. Тогда глобальный индекс для pb с номером local_pb_index:

```
global_pb_index = shift_target_variables +
shift_previous_expressions + cur_number_foralls * (count_pb +
count_bx) + local_pb_index
```

глобальный индекс для bx с номером local_bx_index:

```
global_bx_index = shift_target_variables + shift_previous_expressions + cur_number_foralls
* (count_pb + count_bx) + count_pb + local_bx_index
```

Вычисление count_pb и count_bx производится после компиляции шаблона и равно количеству соответственно переменных pb и bx в шаблоне.

## Термины, сокращения и определения

Термины, сокращения и определения, используемые в настоящем документе, приведены в таблице 1.

*Таблица 1 - Термины, сокращения и определения.*

| **Термины/Сокращения** | **Определения/Пояснения** |
|---|---|
| SAT | Задача выполнимости булевых (пропозициональных) формул (SAT) — это задача нахождения по булевой формуле такой подстановки значений переменным, что при применении данной подстановки формула обращается в тождественную истину. Если такая подстановка существует, то формула называется выполнимой; если же нет, то функция, задаваемая данной формулой, является тождественной ложью, и формула является невыполнимой. |
| MaxSAT | Задача максимальной выполнимости (MaxSAT) является оптимизационной версией задачи булевой выполнимости (SAT) и заключается в выполнении наибольшего количества дизъюнктов в булевой формуле. |
| MILP | Смешанное целочисленное линейное программирование (MILP - Mixed-Integer Linear Programming) - это задача оптимизации, которая включает линейную целевую функцию и линейные ограничения с целочисленными и непрерывными переменными решения. |
| Линейное программирование | Набор математических и вычислительных инструментов, позволяющих найти конкретное решение системы, которое соответствует максимуму или минимуму какой-либо другой линейной функции. |
| Квадратичное программирование | Задача оптимизации квадратичной функции нескольких переменных при линейных ограничениях на эти переменные. |