ОПТИМИЗАЦИЯ РАСЧЁТА БАЛАНСОВ ПРЕДПРИЯТИЙ ГОРНОДОБЫВАЮЩЕЙ ПРОМЫШЛЕННОСТИ

Миловидова Анна Александровна
Старший преподаватель кафедры САУ института САУ
Государственный университет "Дубна"
Порымов Иван Евгеньевич
бакалавр
Государственный университет "Дубна"

OPTIMIZATION OF BALANCE SHEET CALCULATIONS FOR MINING INDUSTRY

Milovidova Anna Aleksandrovna
Senior Lecturer
Dubna State University
Porymov Ivan
Bachelors degree
Dubna state university
DOI: 10.31618/nas.2413-5291.2020.1.57.256

АННОТАЦИЯ

Данная статья посвящена описанию способов сокращения времени для расчёта балансов ключевых документов по описанию деятельности предприятий горноперерабатывающей промышленности. Это позволит сэкономить временные и материальные ресурсы компании. Автором предлагается несколько способов ускорения расчёта, среди них: распрямление переносов продуктов, различные варианты сортировки продуктов, выделение приоритетных групп.

ABSTRACT

This article describes ways to reduce the time needed to calculate the balances of key documents describing the activities of mining companies. This will save the company's time and material resources. The author suggests several ways to speed up the calculation, among them: straightening product transfers, various options for sorting products, highlighting priority groups.

Ключевые слова: горнопереработывающая промышленность; оптимизация; программирование.

Keywords: mining industry; optimization; programming.

ВВЕДЕНИЕ

Основным документом, отражающим результаты деятельности по планированию работы предприятий горноперерабатывающей промышленности, является баланс. Его структура включает в себя названия используемой и производимой продукции, а также их массы в статьях прихода и расхода. Продукты состоят друг с другом в отношениях "зависимых" и "входных". Первый вид отношений подразумевает, что продукты должны пересчитаться после изменения массы данного, второй указывает на то, что расчёт данного продукта будет произведён на основе масс других. Продукты неразрывно связаны с технологическими переделами, где они были произведены или куда были отправлены на новый этап производства.

Все эти зависимости могут быть реализованы с помощью рекурсивного метода по расчёту масс продуктов и их компонентов, участвующих в составлении балансов. Однако, рекурсивные алгоритмы требуют большого количества ресурсов (особенно, если принять во внимание количество продуктов, исчисляемое десятками тысяч), что приводит к возрастанию длительности расчёта балансов. О способах сокращения этого времени на примере программной реализации на языке *C#* и пойдёт речь в данной статье.

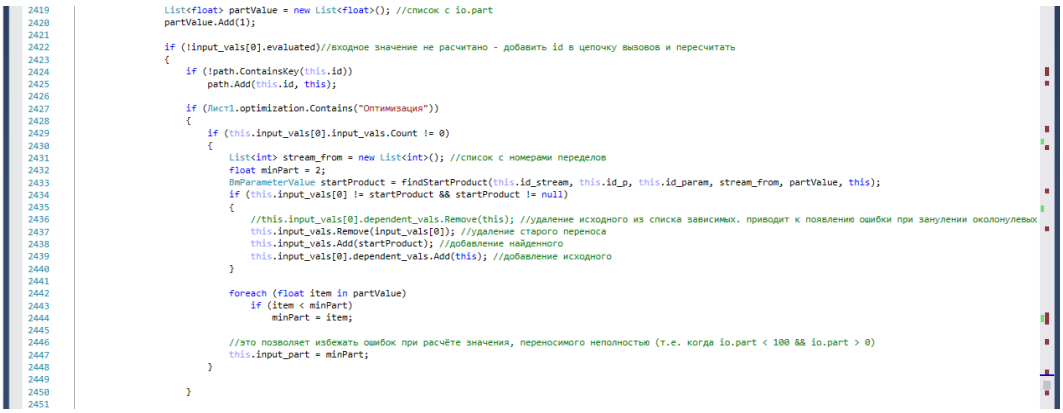
ОПТИМИЗАЦИЯ РАСЧЁТА БАЛАНСОВ

Ядром расчётного модуля является рекурсивный метод *Evaluate()*. Перед непосредственным запуском процедуры происходит сортировка продуктов в зависимости от количества зависимых значений (в исходной версии).

Прежде чем достичь пункта назначения часть продуктов проходит через несколько переделов, что естественно отражается на количестве пересчётов, а, следовательно, и на длительности расчёта. Поэтому была реализована процедура по «распрямлению» переделов – *findStartProduct*.

Если входное значение переносимого продукта в свою очередь имеет входное значение, то будет вызван метод по определению первоначального передела (см. рис. 1).

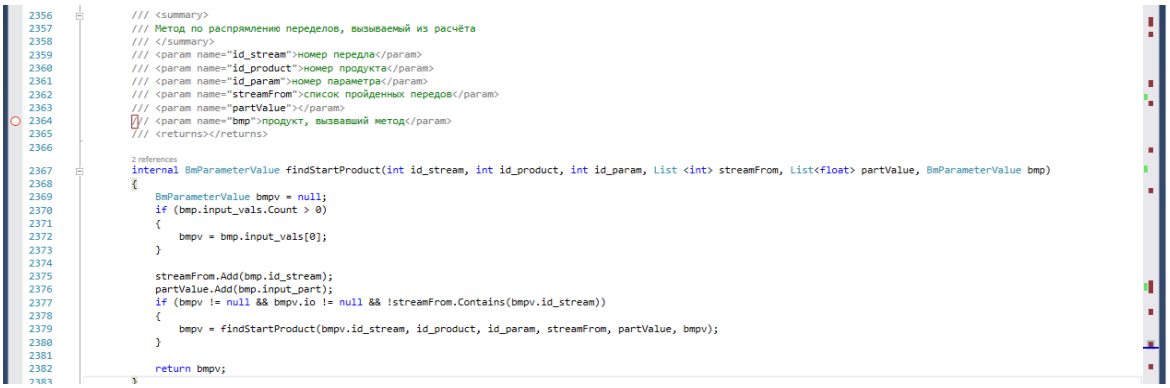
Если «родительский» продукт не равен найденному, то нужно изменить связи продуктов. Для сохранения точности расчёта, при перенесении продукта больше одного раза, удаляется ссылка на данный продукт у продукта с промежуточного передела и, при необходимости перезаписывается процент переноса (отдельные значения переходят не полностью).



```
List<float> partValue = new List<float>(); //список с io.part
partValue.Add(1);

if (!input_vals[0].evaluated)//входное значение не расчитано - добавить id в цепочку вызовов и пересчитать
{
    if (!path.ContainsKey(this.id))
        path.Add(this.id, this);

    if (Лист1.optimization.Contains("Оптимизация"))
    {
        if (this.input_vals[0].input_vals.Count != 0)
        {
            List<int> stream_from = new List<int>(); //список с номерами переделов
            float minPart = 2;
            BmParameterValue startProduct = findStartProduct(this.id_stream, this.id_p, this.id_param, stream_from, partValue, this);
            if (this.input_vals[0] != startProduct && startProduct != null)
            {
                //this.input_vals[0].dependent_vals.Remove(this); //удаление исходного из списка зависимых. приводит к появлению ошибки при занулении околонулевых
                this.input_vals.Remove(input_vals[0]); //удаление старого переноса
                this.input_vals.Add(startProduct); //добавление найденного
                this.input_vals[0].dependent_vals.Add(this); //добавление исходного
            }

            foreach (float item in partValue)
                if (item < minPart)
                    minPart = item;

            //это позволяет избежать ошибок при расчёте значения, переносимого неполностью (т.е. когда io.part < 100 && io.part > 0)
            this.input_part = minPart;
        }

    }
```

Рисунок 1. Вызов метода findStartProduct

Метод по определению исходного передела является рекурсивным, его код представлен на рисунке 2.



```
/// <summary>
/// Метод по распрямлению переделов, вызываемый из расчёта
/// </summary>
/// <param name="id_stream">номер передла</param>
/// <param name="id_product">номер продукта</param>
/// <param name="id_param">номер параметра</param>
/// <param name="streamFrom">список пройденных передов</param>
/// <param name="partValue"></param>
/// <param name="bmp">продукт, вызвавший метод</param>
/// <returns></returns>

2 references
internal BmParameterValue findStartProduct(int id_stream, int id_product, int id_param, List <int> streamFrom, List<float> partValue, BmParameterValue bmp)
{
    BmParameterValue bmpv = null;
    if (bmp.input_vals.Count > 0)
    {
        bmpv = bmp.input_vals[0];
    }

    streamFrom.Add(bmp.id_stream);
    partValue.Add(bmp.input_part);
    if (bmpv != null && bmpv.io != null && !streamFrom.Contains(bmpv.id_stream))
    {
        bmpv = findStartProduct(bmpv.id_stream, id_product, id_param, streamFrom, partValue, bmpv);
    }

    return bmpv;
}
```

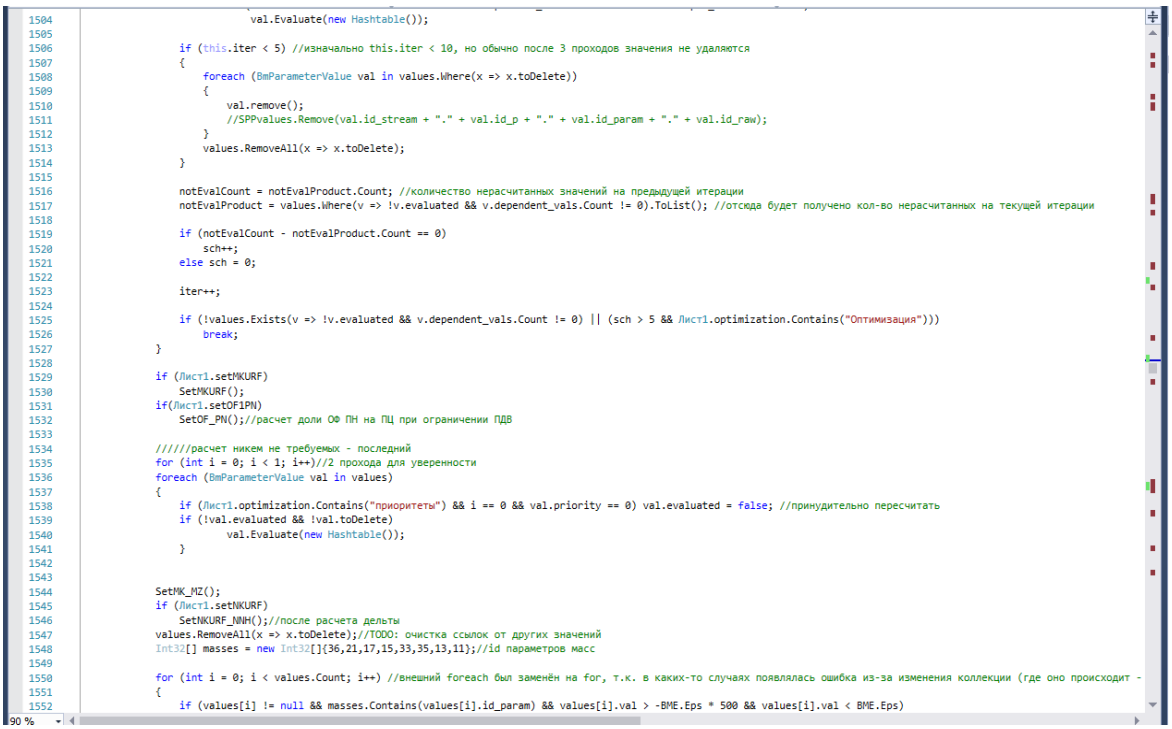
Рисунок 2. Код метода findStartProduct

Проведение тестирования продемонстрировало целесообразность использования данной процедуры.

Затем было решено пересмотреть сортировку продуктов, ранее производившуюся только по количеству зависимых значений, в неё было добавлено второе звено – сортировка получившейся последовательности продуктов по количеству входных продуктов. Таким образом, наиболее востребованные из них оказывались на поверхности списка. Это принесло свои плоды. Записи файлов логирования свидетельствовали о сокращении времени расчёта. Суммарно на балансах с 2021 по 2027 год было сокращено полминуты (31 секунда). Точность расчёта иногда терялась, но за допустимые пределы – 15 тонн – не выходила.

Следующий этап был посвящён анализу отношений между продуктами и назначению приоритетов. Первоначально все продукты получали одинаковый приоритет – 20, он сохранится для продуктов, не имеющих зависимых, часть из них, примерно 2%, считается рассчитанной на момент добавления в список.

В конечном итоге представляющие наибольшую важность продукты сходятся на один суммарный передел. Логично было предположить, что установка высокого приоритета (22) конечным продуктам и тем, что имеют длинную цепочку зависимостей (из двух и более звеньев), позволит запустить в первую очередь нужные цепочки пересчётов. Продукт, названный «Невязка», который рассчитывается как разница между массами прихода и расхода, используется только для проверок и в расчёте других продуктов не участвует. Он получил самый низкий приоритет – 0. С извлечениями и никем не требуемыми продуктами расхода схожая ситуация, их пересчёт, как зависимых, вызывался всякий раз после изменения массы главного по отношению к ним продукта, невзирая на их невостребованность. Значения продуктов данной категории рассчитываются в самом конце принудительно, то есть не в главном рекурсивном методе, а за его пределами. Для этого был организован цикл по пересчёту таких значений (рисунок 3).

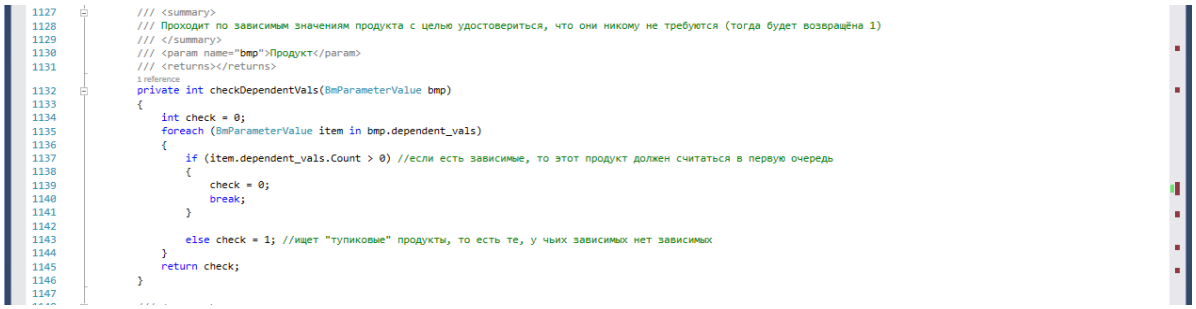


```
                        val.Evaluate(new Hashtable());

                if (this.iter < 5) //изначально this.iter < 10, но обычно после 3 проходов значения не удаляются
                {
                    foreach (BmParameterValue val in values.Where(x => x.toDelete))
                    {
                        val.remove();
                        //SPPvalues.Remove(val.id_stream + "." + val.id_p + "." + val.id_param + "." + val.id_raw);
                    }
                    values.RemoveAll(x => x.toDelete);
                }

                notEvalCount = notEvalProduct.Count; //количество нерасчитанных значений на предыдущей итерации
                notEvalProduct = values.Where(v => !v.evaluated && v.dependent_vals.Count != 0).ToList(); //отсюда будет получено кол-во нерасчитанных на текущей итерации

                if (notEvalCount - notEvalProduct.Count == 0)
                    sch++;
                else sch = 0;

                iter++;

                if (!values.Exists(v => !v.evaluated && v.dependent_vals.Count != 0) || (sch > 5 && Лист1.optimization.Contains("Оптимизация")))
                    break;
            }

            if (Лист1.setMKURF)
                SetMKURF();
            if(Лист1.setOF1PN)
                SetOF_PN();//расчет доли ОФ ПН на ПЦ при ограничении ПДВ

            //////расчет никем не требуемых - последний
            for (int i = 0; i < 1; i++)//2 прохода для уверенности
            foreach (BmParameterValue val in values)
            {
                if (Лист1.optimization.Contains("приоритеты") && i == 0 && val.priority == 0) val.evaluated = false; //принудительно пересчитать
                if (!val.evaluated && !val.toDelete)
                        val.Evaluate(new Hashtable());
                }


            SetMK_MZ();
            if (Лист1.setNKURF)
                SetNKURF_NNH();//после расчета дельты
            values.RemoveAll(x => x.toDelete);//TODO: очистка ссылок от других значений
            Int32[] masses = new Int32[]{36,21,17,15,33,35,13,11};//id параметров масс

            for (int i = 0; i < values.Count; i++) //внешний foreach был заменён на for, т.к. в каких-то случаях появлялась ошибка из-за изменения коллекции (где оно происходит -
            {
                if (values[i] != null && masses.Contains(values[i].id_param) && values[i].val > -BME.Eps * 500 && values[i].val < BME.Eps)
```

Рис. 3. Расчёт продуктов с самым низким приоритетом

Продукты, имеющие единственное звено в цепочке зависимых значений, получали приоритет, равный 5.

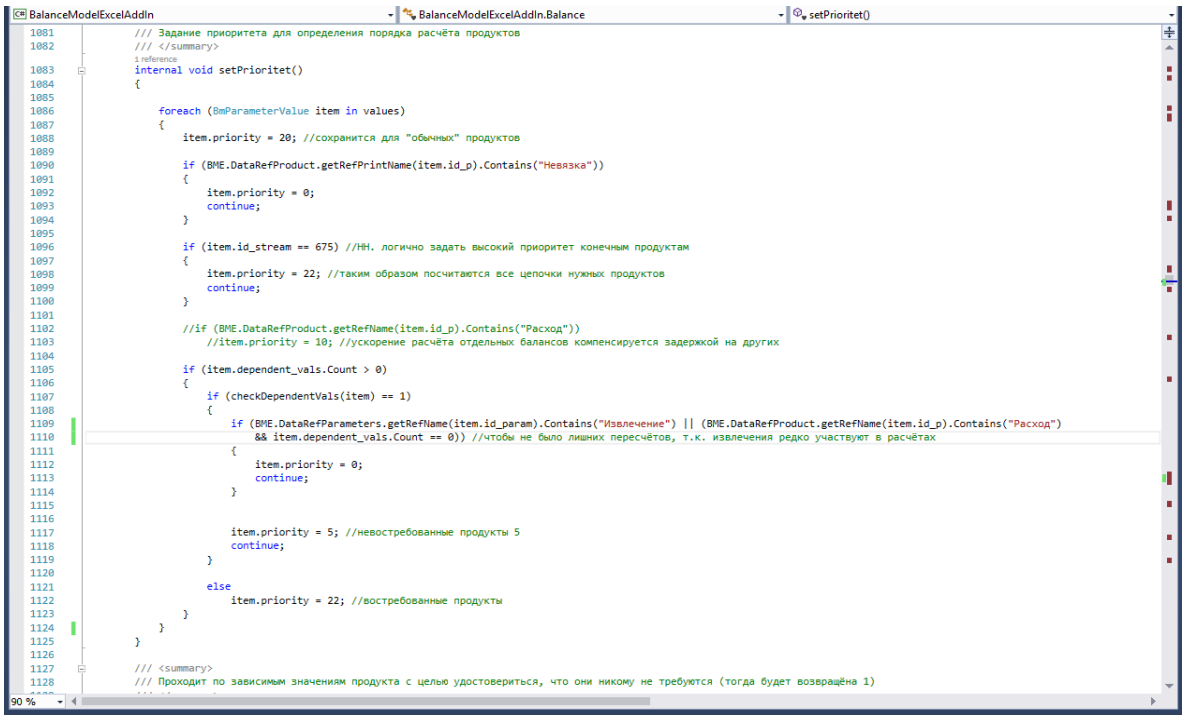
Определение количества звеньев было возложено на процедуру *checkDependentVal*, код которой представлен на рисунке 4. Она вызывается от каждого продукта, хранящегося в списке *values* и если количество зависимых значений у зависимых значений данного продукта больше одного, то процедура вернёт 0, в ином случае 1.



```
        /// <summary>
        /// Проходит по зависимым значениям продукта с целью удостовериться, что они никому не требуются (тогда будет возвращёна 1)
        /// </summary>
        /// <param name="bmp">Продукт</param>
        /// <returns></returns>
        1 reference
        private int checkDependentVals(BmParameterValue bmp)
        {
            int check = 0;
            foreach (BmParameterValue item in bmp.dependent_vals)
            {
                if (item.dependent_vals.Count > 0) //если есть зависимые, то этот продукт должен считаться в первую очередь
                {
                    check = 0;
                    break;
                }

                else check = 1; //ищет "тупиковые" продукты, то есть те, у чьих зависимых нет зависимых
            }
            return check;
        }
```

Рисунок 4. Код метода checkDependentVals

Назначение приоритетов происходило в процедуре *setPriotet*. Её код на рисунке 5.




```
1081        /// Задание приоритета для определения порядка расчёта продуктов
1082        /// </summary>
            1 reference
1083        internal void setPrioritet()
1084        {
1085
1086            foreach (BmParameterValue item in values)
1087            {
1088                item.priority = 20; //сохранится для "обычных" продуктов
1089
1090                if (BME.DataRefProduct.getRefPrintName(item.id_p).Contains("Невязка"))
1091                {
1092                    item.priority = 0;
1093                    continue;
1094                }
1095
1096                if (item.id_stream == 675) //НН. логично задать высокий приоритет конечным продуктам
1097                {
1098                    item.priority = 22; //таким образом посчитаются все цепочки нужных продуктов
1099                    continue;
1100                }
1101
1102                //if (BME.DataRefProduct.getRefName(item.id_p).Contains("Расход"))
1103                    //item.priority = 10; //ускорение расчёта отдельных балансов компенсируется задержкой на других
1104
1105                if (item.dependent_vals.Count > 0)
1106                {
1107                    if (checkDependentVals(item) == 1)
1108                    {
1109                        if (BME.DataRefParameters.getRefName(item.id_param).Contains("Извлечение") || (BME.DataRefProduct.getRefName(item.id_p).Contains("Расход")
1110                            && item.dependent_vals.Count == 0)) //чтобы не было лишних пересчётов, т.к. извлечения редко участвуют в расчётах
1111                        {
1112                            item.priority = 0;
1113                            continue;
1114                        }
1115
1116
1117                        item.priority = 5; //невостребованные продукты 5
1118                        continue;
1119                    }
1120
1121                    else
1122                        item.priority = 22; //востребованные продукты
1123                }
1124            }
1125        }
1126
1127        /// <summary>
1128        /// Проходит по зависимым значениям продукта с целью удостовериться, что они никому не требуются (тогда будет возвращёна 1)
```

Рисунок 5. Код метода setPriotet

Затем были совершены реформы в других частях кода. Сортировка обогатилась третьим звеном – идущие по убыванию приоритеты продуктов.

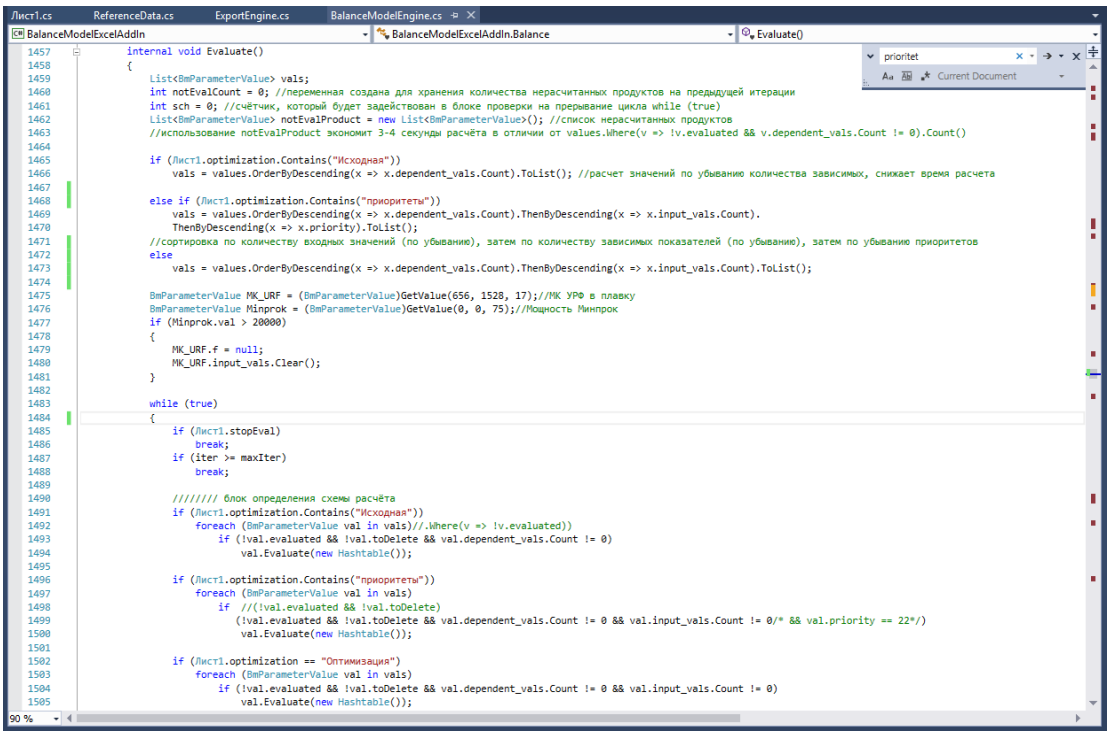
Расширение вариантов расчёта спровоцировало новое изменений интерфейсной части, в книгу балансовой модели потребовалось добавить *comboBox* со следующими элементами: исходная версия, оптимизация, оптимизация + приоритеты. Эти варианты различаются способом сортировки продуктов и условиями прерывания расчёта, распрямление переделов распространяется только на два последних.

Также было установлено, что выход из метода *Evaluate* после завершения расчётов всех продуктов почти никогда не срабатывал, обычно прерывание происходило из-за превышения количества итераций, поэтому была создано условие, которое завершало расчёт в случае отсутствия изменений в количестве рассчитанных продуктов на протяжении пяти итераций подряд (см. рис. 6).

```
1518
1519            notEvalCount = countNotEvalProduct;
1520            countNotEvalProduct = values.Where(v => !v.evaluated && v.dependent_vals.Count != 0).Count();  //отсюда будет получено кол-во нерасчитанных на текущей итера
1521
1522            if (notEvalCount == countNotEvalProduct)
1523                sch++;
1524            else sch = 0;
1525
1526            iter++;
1527
1528            if (!values.Exists(v => !v.evaluated && v.dependent_vals.Count != 0) || (sch > 5 && Лист1.optimization.Contains("Оптимизация")))
1529                break;
```

Рисунок 6. Дополнительное условие прерывания расчёта

После этого первая часть метода расчёта приобрела вид, закреплённый на рисунке 7.



```
internal void Evaluate()
{
    List<BmParameterValue> vals;
    int notEvalCount = 0; //переменная создана для хранения количества нерасчитанных продуктов на предыдущей итерации
    int sch = 0; //счётчик, который будет задействован в блоке проверки на прерывание цикла while (true)
    List<BmParameterValue> notEvalProduct = new List<BmParameterValue>(); //список нерасчитанных продуктов
    //использование notEvalProduct экономит 3-4 секунды расчёта в отличии от values.Where(v => !v.evaluated && v.dependent_vals.Count != 0).Count()

    if (Лист1.optimization.Contains("Исходная"))
        vals = values.OrderByDescending(x => x.dependent_vals.Count).ToList(); //расчет значений по убыванию количества зависимых, снижает время расчета

    else if (Лист1.optimization.Contains("приоритеты"))
        vals = values.OrderByDescending(x => x.dependent_vals.Count).ThenByDescending(x => x.input_vals.Count).
        ThenByDescending(x => x.priority).ToList();
    //сортировка по количеству входных значений (по убыванию), затем по количеству зависимых показателей (по убыванию), затем по убыванию приоритетов
    else
        vals = values.OrderByDescending(x => x.dependent_vals.Count).ThenByDescending(x => x.input_vals.Count).ToList();

    BmParameterValue MK_URF = (BmParameterValue)GetValue(656, 1528, 17);//МК УРФ в плавку
    BmParameterValue Minprok = (BmParameterValue)GetValue(0, 0, 75);//Мощность Минпрок
    if (Minprok.val > 20000)
    {
        MK_URF.f = null;
        MK_URF.input_vals.Clear();
    }

    while (true)
    {
        if (Лист1.stopEval)
            break;
        if (iter >= maxIter)
            break;

        //////// блок определения схемы расчёта
        if (Лист1.optimization.Contains("Исходная"))
            foreach (BmParameterValue val in vals)//.Where(v => !v.evaluated))
                if (!val.evaluated && !val.toDelete && val.dependent_vals.Count != 0)
                    val.Evaluate(new Hashtable());

        if (Лист1.optimization.Contains("приоритеты"))
            foreach (BmParameterValue val in vals)
                if  //(!val.evaluated && !val.toDelete)
                   (!val.evaluated && !val.toDelete && val.dependent_vals.Count != 0 && val.input_vals.Count != 0/* && val.priority == 22*/)
                    val.Evaluate(new Hashtable());

        if (Лист1.optimization == "Оптимизация")
            foreach (BmParameterValue val in vals)
                if (!val.evaluated && !val.toDelete && val.dependent_vals.Count != 0 && val.input_vals.Count != 0)
                    val.Evaluate(new Hashtable());
```

Рисунок 7. Считывание способа расчёта

Как было сказано ранее, важность имеет не только ускорение расчёта, но и сохранение точности. Предыдущие версии способов выделения групп продуктов с помощью приоритетов не подходили именно из-за проблем с точностью, описанная здесь полностью удовлетворяет заявленным требованиям.

Файлы логирования свидетельствуют о суммарном ускорении расчёта балансов 2021-2027 годов по сравнению с исходной версией на 51-53 секунды, по сравнению с предыдущим вариантом оптимизации на 21-22 секунды. Длительность расчётов балансов разных лет в каждой из описанных версий расчёта представлены на рисунке 8.

	Версия		
Год	Исходная	Старая	Приоритеты
2021	45	29	28
2022	47	34	33
2023	48	47	45
2024	35	46	39
2025	17	17	17
2026	9	15	9
2027	51	33	28
	252	221	199

Рисунок 8. Длительности расчётов балансов для конфигурации 1

Далее были проведены расчёты на другой конфигурацияи (см. рис. 9).

Год	Исходная	Старая	Приоритеты
2021	44	37	29
2022	44	41	31
2023	42	39	36
2024	27	15	28
2025	17	17	17
2026	8	16	9
2027	47	36	48
	229	201	198

Рисунок 9. Длительности расчётов балансов для конфигурации 2

ЗАКЛЮЧЕНИЕ

Данная работа демонстрирует зарекомендовавшие себя способы оптимизации расчёта балансов предприятий горноперерабатывающей промышленности. Среди них распрямление переносов продуктов, изменение способов сортировки, а также создание групп продуктов в зависимости от их приоритета.

СПИСОК ЛИТЕРАТУРЫ

1. Рабочий проект Балансовая модель: Техническое задание на балансовую модель по цветным, драгоценным металлам и сере ЗФ ОАО «ГМК «Норильский никель» 2010 г.

2. Кайтмазов Н. Г. Производство металлов за Полярным кругом. Технологическое пособие — Норильск, 2006.

3. Металлоторг – режим доступа к изд.: http://www.metaltrade.ru/abc/b/balans.htm

УДК 658.512

УПОРЯДОЧЕНИЯ РАБОТ НА ТЕХНОЛОГИЧЕСКОЙ ЛИНИИ ПРИ НАЛИЧИИ ПЕРЕНАЛАДОК ОБОРУДОВАНИЯ

Сошников А.В.
Санкт-Петербургский государственный университет промышленных технологий и дизайна

RATIONAL ORDERING OF WORK ON THE PRODUCTION LINE IN THE PRESENCE OF EQUIPMENT CHANGEOVERS

Soshnikov A.V.
St. Petersburg state University industrial technology and design
DOI: 10.31618/nas.2413-5291.2020.1.57.260

АННОТАЦИЯ

Рассмотрена задача выбора рациональной очередности выполнения работ в технологических комплексах с последовательной структурой, широко распространенных в различных отраслях промышленности. Постановка задачи отличается учетом переналадок машин, входящих в линию, при смене выполняемых работ. Предложена многокритериальная модель задачи. Предложен подход и конкретные методы поиска рациональных вариантов упорядочения работ при запуске их на линию. Подход базируется на использовании принципов систематической эвристики, позволяющих генерировать ограниченное и управляемое с учетом производственных условий количество вариантов, среди которых, как предполагается, имеются и рациональные (практически приемлемые) варианты.

ABSTARCT

The method of rational ordering of work on the production line in the presence of equipment changeovers The problem of choosing a rational sequence of work in technological complexes with a consistent structure, widespread in various industries, is considered. The problem statement differs taking into account changeovers of the cars entering the line, at change of the performed works. A multicriteria model of the problem is proposed. The approach and specific methods of search of rational variants of ordering of works at their start on the line are offered. The approach is based on the use of the principles of systematic heuristics, which allow to generate a limited and manageable number of options, taking into account the production conditions, among which there are supposed to be rational (practically acceptable) options.

Ключевые слова: технологические линии, переналадки машин, очередность выполнения работ, многокритериальная модель, эвристические методы, Парето-оптимальные решения

Keywords: process lines, machine changeovers, sequence of work, multi-criteria model, heuristic methods, Pareto-optimal solutions.

Задача поиска оптимального или субоптимального порядка выполнения заданного множества работ на технологической линии при одинаковых маршрутах их движения подробно изучена в теории расписаний, и для нее имеются различные алгоритмы, в основном построенные на эвристических правилах назначения приоритетов [2],[3].

Критерием качества расписания часто выступает общее время выполнения работ. В традиционной постановке не учитываются возможные переналадки машин при смене выполняемых на них работ. Среди производственных структур часто встречаются комплексы машин, формально не связанных в единую линию, работающих автономно, но по отношению к объекту обработки представляющие цепочки операций, выполняемых последовательно. Переналадки отдельных машин при смене работ достаточно часто имеют место на практике. В этом случае в такой технологической системе правомерно ставить задачу выбора очередности выполнения работ, при которой приближаются к своим наилучшим или нормативным значениям такие показатели как общее время выполнения работ, затраты на переналадки как отдельных