Реализация общих шагов K-opt

В этом разделе описывается реализация общих ходов K-opt в LKH-2. Описание разделено на следующие четыре части:

1. Поиск последовательных ходов
2. Поиск непоследовательных ходов
3. Определение целесообразности хода
4. Выполнение возможного хода

Первые две части показывают, как можно систематически исследовать пространство поиска возможных ходов. Третья часть описывает, как можно решить, выполним ли данный ход, то есть приведет ли выполнение хода в текущем туре к туру. Наконец, показано, как это возможно эффективно выполняйте выполнимый ход.

Задействованные алгоритмы задаются с помощью языка программирования C. Код полностью соответствует фактической реализации в LKH-2.

5.1 Поиск последовательных ходов

Последовательный ход K-opt в туре T может быть задан последовательностью узлов $(t_1, t_2, \ldots, t_{2K-1}, t_{2K})$, где

- (t_{2i-1}, t_{2i}) принадлежит T $(1 \leq i \leq K)$ и
- (t_{2i}, t_{2i+1}) не принадлежит T $(1 \leq i \leq K$ и $t_{2K+1} = t_1)$

Требование о том, что (t_{2i}, t_{2i+1}) не принадлежит T, на самом деле, не является частью определения последовательного перемещения K-opt. Заметим, однако, что если какое-либо из этих ребер принадлежит T, то последовательный ход K-opt также является последовательным K'-opt ходом для некоторого K'< K. Таким образом, при поиске K-opt ходов это требование не исключает каких-либо ходов, которые должны быть найдены. Это требование упрощает кодирование, не нанося никакого вреда.

Таким образом, мы можем сгенерировать все возможные последовательные ходы K-opt, сгенерировав все t-последовательности длиной 2K, которые удовлетворяют двум требованиям. Генерация таких t-последовательностей может, например, выполняться итеративно в 2K вложенных циклах, где цикл на уровне i проходит через все возможные значения для t_i.

Предположим, что был выбран первый элемент, t_1, из t-последовательности. Затем может быть реализован итеративный алгоритм для генерации оставшихся элементов, как показано в псевдокоде ниже.

```
for (each t[2] in {PRED(t[1]), SUC(t[1])}) {
    G[0] = C(t[1], t[2]);
    for (each candidate edge (t[2], t[3])) {
        if (t[3] != PRED(t[2]) && t[3] != SUC(t[2]) &&
            (G[1] = G[0] - C(t[2], t[3]) > 0) {
            for (each t[4] in {PRED(t[3]), SUC(t[3])}) {
                G[2] = G[1] + C(t[3], t[4]);
                if (FeasibleKOptMove(2) &&
                    (Gain = G[2] - C(t[4], t[1])) > 0) {
                    MakeKOptMove(2);
                    return Gain;
                }
            }
            inner loops for choosing t[5], ..., t[2K]
        }
    }
}
```

Комментарии:

- t-последовательность хранится в массиве t узлов. Три крайних цикла выбирают элементы t[2], t[3] и t[4]. Операции PRED и SUCH возвращают для данного узла соответственно его предшественника и преемника в туре,
- Функция $C(t_a, t_b)$, где t_a и t_b - два узла, возвращает стоимость ребра (t_a, t_b).
- Массив G используется для хранения накопленного коэффициента усиления. Он используется для проверки выполнения критерия положительного выигрыша и для вычисления выигрыша от возможного хода.
- Функция $FeasibleKOptMove(k)$ определяет, представляет ли данная t-последовательность $(t_1, t_2, \ldots, t_{2K-1}, t_{2K})$ допустимый k-opt ход, где $2 \leq k \leq K$. Функция $MakeKOptMove(k)$ выполняет допустимое перемещение k-opt. Реализация этих функций описана в разделах 5.3 и 5.4.
- Обратите внимание, что, если во время поиска обнаруживается выгодный возможный ход, этот ход выполняется немедленно.

Внутренние циклы для определения $t_5, \ldots, t_{2K}$ могут быть реализованы аналогично приведенному выше коду. Однако у самого внутреннего цикла есть одна дополнительная задача, а именно зарегистрировать невыгодный K-opt

ход, который кажется наиболее многообещающим для продолжения цепочки K-opt ходов. Самый внутренний цикл может быть реализован следующим образом:

```
for (each t[2 * K] in {PRED(t[2 * K - 1], SUC[t[2 * K - 1]]}) {
    G[2 * K] = G[2 * K - 1] + C(t[2 * K - 1], t[2 * K]);
    if (FeasibleKOptMove(K)) {
        if ((Gain = G[2 * K] - C(t[2 * K], t[1])) > 0) {
            MakeKOptMove(K);
            return Gain;
        }
        if (G[2 * K] > BestG2K &&
            Excludable(t[2 * K - 1], t[2 * K])) {
            BestG2K = G[2 * K];
            for (i = 1; i <= 2 * K; i++)
                Best_t[i] = t[i];
        }
    }
}
```

Комментарии:

- Осуществимый ход K-opt, который максимизирует совокупный выигрыш, $G[2K]$, считается наиболее многообещающим ходом.
- Функция $Excludable$ используется для проверки того, было ли ранее добавлено последнее ребро, подлежащее удалению при K-opt перемещении, в текущую цепочку K-opt перемещений (критерий 4 в разделе 3).

Генерация 5-opt ходов в LKH-1 была реализована, как описано выше. Однако, если мы хотим генерировать ходы K-opt, где K может быть выбрано свободно, этот подход не подходит. В этом случае мы хотели бы использовать переменное количество вложенных циклов. Обычно это невозможно в императивных языках, таких как C, но хорошо известно, что это может быть смоделировано с помощью рекурсии. Рекурсивная реализация алгоритма приведена ниже.

```
GainType BestKOptMoveRec(int k, GainType G0) {
    GainType G1, G2, G3, Gain;
    Node *t1 = t[1], *t2 = t[2 * k - 2], *t3, *t4;
    int i;

    for (each candidate edge (t2, t3)) {
        if (t3 != PRED(t2) && t3 != SUC(t2) &&
            !Added(t2, t3, k - 2) &&
            (G1 = G0 - C(t2, t3)) > 0) {
            t[2 * k - 1] = t3;
            for (each t4 in {PRED(t3), SUC(t3)}) {
                if (!Deleted(t3, t4, k - 2)) {
                    t[2 * k] = t4;
                    G2 = G1 + C(t3, t4);
                    if (FeasibleKOptMove(k) &&
                        (Gain = G2 - C(t4, t1)) > 0) {
                        MakeKOptMove(k);
                        return Gain;
                    }
                    if (k < K &&
                        (Gain = BestKOptMoveRec(k + 1, G2)) > 0)
                        return Gain;
                    if (k == K && G2 > BestG2K &&
                        Excludable(t3, t4)) {
                        BestG2K = G2;
                        for (i = 1; i <= 2 * K; i++)
                            Best_t[i] = t[i];
                    }
                }
            }
        }
    }
}
```

Комментарий:

- Вспомогательные функции $Added$ и $Deleted$ используются для обеспечения того, чтобы ни одно ребро не добавлялось или не удалялось более одного раза в строящемся перемещении. Возможные реализации этих двух функций показаны ниже. Легко видеть, что временная сложность для каждой из этих функций равна $O(k)$.

```
int Added(Node *ta, Node *tb, int k) {
    int i = 2 * k;
    while ((i -= 2) > 0)
        if ((ta == t[i] && tb == t[i + 1]) ||
            (ta == t[i + 1] && tb == t[i]))
            return 1;
    return 0;
}
```

```
int Deleted(Node * ta, Node * tb, int k) {
    int i = 2 * k + 2;
    while ((i -= 2) > 0)
        if ((ta == t[i - 1] && tb == t[i]) ||
            (ta == t[i] && tb == t[i - 1]))
        return 1;
    return 0;
}
```

Учитывая два соседних узла в туре, t_1 и t_2, поиск перемещения K-opt инициируется вызовом функции драйвера $BestKOptMove$, показанной ниже.

```
Node *BestKOptMove(Node *t1, Node *t2,
                   GainType *G0, GainType *Gain) {
    t[1] = t1; t[2] = t2;
    BestG2K = MINUS_INFINITY;
    Best_t[2 * K] = NULL;
    *Gain = BestKOptMoveRec(2, *G0);
    if (*Gain <= 0 && Best_t[2 * K] != NULL) {
        for (i = 1; i <= 2 * K; i++)
            t[i] = Best_t[i];
        MakeKOptMove(K);
    }
    return Best_t[2 * K];
}
```

Эта реализация позволяет легко построить цепочку K-opt ходов:

```
GainType G0 = C(t1, t2), Gain;
do
    t2 = BestKOptMove(t1, t2, &G0, &Gain);
while (Gain <= 0 && t2 != NULL);
```

Цикл оставляется, как только цепочка представляет собой выгодный ход, или когда цепочку больше нельзя продлевать. Теоретически, в цикле будет до n итераций, где n - количество узлов. На практике, однако, количество итераций намного меньше (из-за критерия положительного усиления).

Временная сложность для каждой из итераций, то есть для каждого вызова $BestKOptMove$, может быть оценена исходя из временных сложностей для задействованных подопераций. В следующих разделах показано, что эти подоперации могут быть реализованы с учетом временных сложностей, приведенных в таблице 5.1.1.

Operation	*Complexity*
PRED	$O(1)$
SUC	$O(1)$
Excludable	$O(1)$
Added	$O(K)$
Deleted	$O(K)$
FeasibleKOptMove	$O(K\log K)$
MakeKOptMove	$O(\sqrt{N})$

Table 5.1.1. Complexities for operations involved in the search for sequential moves.

Пусть K обозначает максимальную степень узла в графе-кандидате. Тогда легко видеть, что наихудшая временная сложность для итерации равна $O(d^K KlogK + \sqrt{n})$. Если d и K малы по сравнению с n, что обычно имеет место, то наихудшая временная сложность равна $O(\sqrt{n})$.

Обратите внимание, однако, что количество $d^K KlogK$ растет экспоненциально с K, если $d \geq 2$, и что этот термин быстро становится доминирующим. Это подчеркивает важность выбора разреженного графа-кандидата, если требуются высокие значения K.

5.2 Поиск непоследовательных ходов

В оригинальной версии алгоритма Лина-Кернигана (LK) непоследовательные ходы используются только в одном особом случае, а именно, когда алгоритм больше не может найти никаких последовательных ходов, улучшающих тур. В этом случае он пытается улучшить тур с помощью непоследовательного 4-opt хода, так называемого двойного моста (см. рис. 2.4).

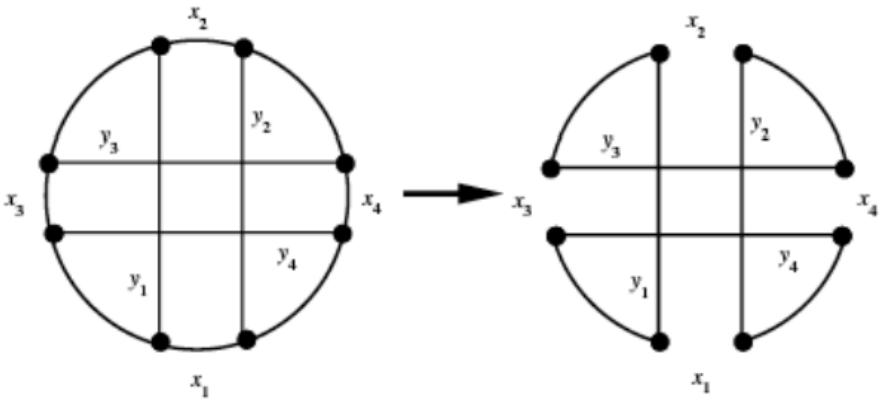

Figure 2.4. Non-sequential exchange (k = 4).

В LKH-1 этот вид постоптимизационных ходов расширен, чтобы включать в себя непоследовательные 5-opt ходы. Однако, в отличие от LK, поиск непоследовательных улучшений рассматривается не только как маневр после оптимизации. То есть, если найдено улучшение, предпринимаются дальнейшие попытки улучшить тур путем обычных последовательных, а также непоследовательных обменов.

LKH-2 развивает эту идею на шаг дальше. Теперь поиск непоследовательных ходов интегрирован с поиском последовательных ходов. Кроме того, можно искать непоследовательные ходы k-opt для любого значения $k \geq 4$.

Основная идея заключается в следующем. Если во время поиска последовательного хода будет найден невыполнимый ход, этот невыполнимый ход может быть использован в качестве отправной точки для построения возможного непоследовательного хода. Обратите внимание, что невыполнимый ход, если он будет выполнен в текущем туре, приведет к двум или более непересекающимся циклам. Следовательно, мы можем получить осуществимый ход, если эти циклы каким-то образом можно будет соединить вместе, чтобы сформировать один и только один цикл.

Решение этой проблемы с исправлением цикла простое. Учитывая набор непересекающихся циклов, мы можем исправить эти циклы одним или несколькими чередующимися циклами. Предположим, например, что выполнение невыполнимого хода k-opt, $k \geq 4$, привело бы к четырем непересекающимся циклам. Как показано на рисунке 5.2.1, четыре цикла могут быть преобразованы в тур с использованием одного чередующегося цикла, который представлен последовательностью узлов $(s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_1)$. Обратите внимание, что чередующийся цикл

поочередно удаляет ребро из одного из четырех циклов и добавляет ребро, которое соединяет два из четырех циклов.

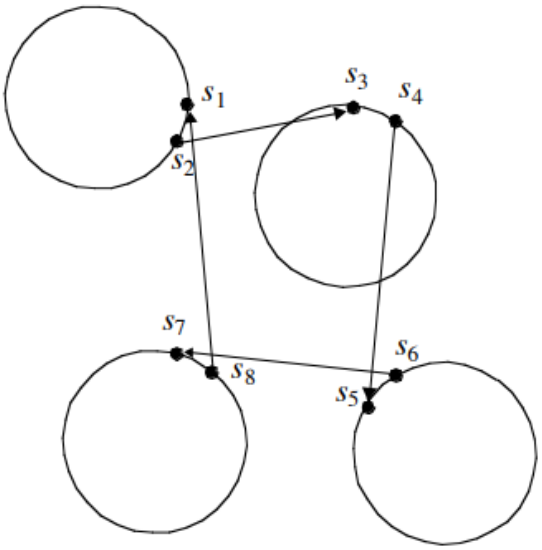

Figure 5.2.1. Four disjoint cycles patched by one alternating cycle.

На рисунке 5.2.2 показано, как четыре непересекающихся цикла могут быть исправлены двумя чередующимися циклами: $(s_1, s_2, s_3, s_4, s_1)$ и $(t_1, t_2, t_3, t_4, t_5, t_6, t_1)$. Обратите внимание, что оба чередующихся цикла необходимы для того, чтобы совершить тур.

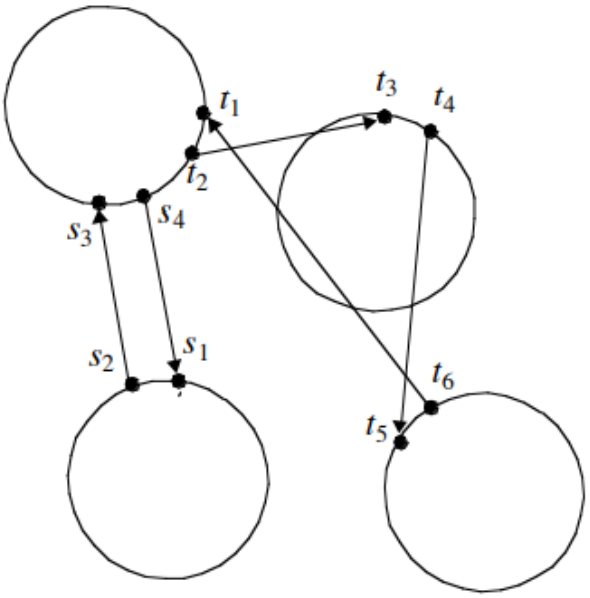

Figure 5.2.2. Four disjoint cycles patched by two alternating cycles.

Рисунок 5.2.3 иллюстрирует, что также возможно использовать три чередующихся цикла: $(s_1, s_2, s_3, s_4, s_1)$, $(t_1, t_2, t_3, t_4, t_1)$ и $(u_1, u_2, u_3, u_4, u_1)$. В общем, K циклов могут быть преобразованы в тур, используя до K-1 чередующихся циклов.

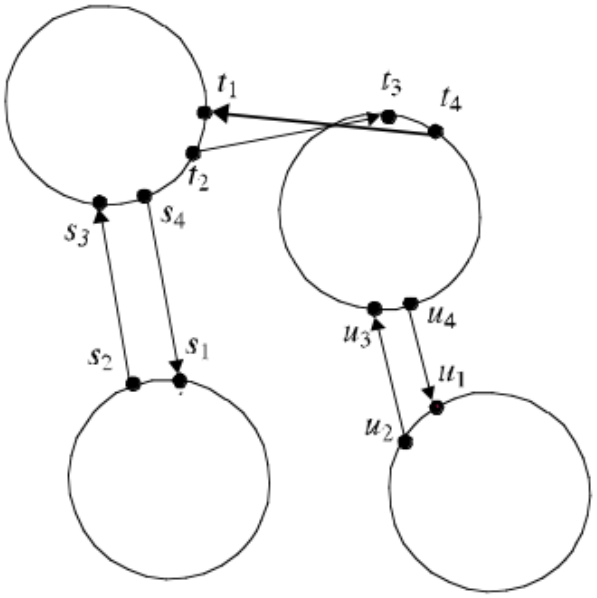

Figure 5.2.3. Four disjoint cycles patched by three alternating cycles.

С добавлением непоследовательных ходов количество различных типов ходов k-opt, которые алгоритм должен уметь обрабатывать, значительно увеличилось. Далее это утверждение дается как количественная оценка.

Пусть $MT(k)$ обозначает количество типов перемещений k-opt. Перемещение k-opt удаляет k ребер из тура и добавляет k ребер, которые повторно соединяют k сегментов тура в тур. Следовательно, перемещение k-opt может быть определено как циклическая перестановка k сегментов тура, где некоторые из сегментов могут быть инвертированы (поменяны местами). Пусть один из сегментов будет фиксированным. Тогда $MT(k)$ может быть вычислено как произведение количество инверсий k-1 сегментов и количество перестановок k-1 сегментов:

$$MT(k) = 2^{k-1}(k-1)!$$

Однако $MT(k)$ включает в себя количество ходов, которые повторно вставляют одно или несколько удаленных ребер. Поскольку такие ходы могут быть сгенерированы k'-opt ходами, где k' < k, нас больше интересует вычисление $PMT(k)$, число чистых k-opt ходов, то есть ходов, для которых множество удаленных ребер и множество добавленных ребер не пересекаются. Функе, Грюнерт и Ирних дают следующую формулу рекурсии для $PMT(k)$:

$$PMT(k) = MT(k) - \sum_{i=2}^{k-1} \binom{k}{i} PMT(i) - 1 \text{ для } k \geq 3$$

$$PMT(2) = 1$$

Явная формула для $PMT(k)$ может быть получена из формулы для ряда A061714 в Онлайн-энциклопедии целочисленных последовательностей [40]:

$$PMT(k) = \sum_{i=1}^{k-1} \sum_{j=0}^{i} (-1)^{k+j-1} \binom{i}{j} j! \, 2^j \text{ для } k \geq 2$$

Набор чистых ходов учитывает как последовательные, так и непоследовательные ходы. Чтобы проверить, насколько расширилось пространство поиска за счет включения непоследовательных ходов, мы вычислим $SPMT(k)$, количество последовательных, чистых k-opt ходов, и сравним это число с $PMT(k)$. Явная формула для $SPMT(k)$, показанная ниже, была получена Хэнлоном, Стэнли и Стембриджем [14].:

$$SPMT(k) = \frac{2^{3k-2} k! (k-1)!^2}{(2k)!} + \sum_{a=1}^{k-1} \sum_{b=1}^{min(a,k-a)} c_{a,b}(2) \left[\frac{2^{a-b-1} (2b)! (a-1)! (k-a-b-1)}{(2b-1) b!} \right]^2$$

где

$$c_{a,b}(2) = (-1)^k \frac{(-2)^{a-b+1} k (2a-2b+1)(a-1)!}{(k+a-b+1)(k+a-b)(k-a+b)(k-a+b-1)(k-a-b)! (2a-1)! (b-1)!}$$

В таблице 5.2.1 показаны $MT(k)$, $PMT(k)$, $SPMT(k)$ и соотношение $SPMT(k)/PMT(k)$ для выбранных значений k. Как видно, отношение $SPMT(k)/PMT(k)$ уменьшается по мере того, как k увеличивается. Для k ≥ 10 существует меньше типов последовательных ходов, чем типов непоследовательных ходов.

k	2	3	4	5	6	7	8	9	10	50	100
$MT(k)$	2	8	48	384	3840	46080	645120	1.0E7	1.9E8	3.4E77	5.9E185
$PMT(k)$	1	4	25	208	2121	25828	365457	5.9E6	1.1E8	2.1E77	3.6E185
$SPMT(k)$	1	4	20	148	1348	15104	198144	3.0E6	5.1E7	4.3E76	5.9E184
$\frac{SPMT(k)}{PMT(k)}$	1	1	0.80	0.71	0.63	0.58	0.54	0.51	0.48	0.21	0.17

Table 5.2.1. Growth of move types for k-opt moves.

Из таблицы также следует, что количество типов непоследовательных чистых ходов составляет 20% или более от всех типов чистых ходов для k ≥ 4. Поэтому очень важно, чтобы реализация генерации непоследовательных ходов была эффективной во время выполнения. В противном случае его практическая ценность будет ограничена.

Пусть задан набор циклов C, соответствующий некоторому невыполнимому последовательному движению. Затем LKH-2 ищет набор чередующихся циклов, AC, который при применении к C приводит к улучшенному туру. Набор AC строится элемент за элементом. Процесс поиска ограничен следующими правилами:

1. Никакие два чередующихся цикла не имеют общего ребра.

2. Все внешние ребра переменного цикла принадлежат пересечению текущего тура и C.

3. Все внутренние ребра чередующегося цикла должны соединять два цикла.

4. Начальный узел для чередующегося цикла должен принадлежат самому короткому циклу (циклу с наименьшим числом ребер).

5. Построение переменного цикла запускается только в том случае, если текущий коэффициент усиления положительный.

Правила 1-3 наглядно представлены на рисунках 5.2.1-3. Чередующийся цикл переходит от цикла к циклу, в конечном итоге соединяя последний посещенный узел с начальным узлом. Легко видеть, что чередующийся цикл с 2m ребрами ($m \geq 2$) уменьшает количество компонентов цикла на m - 1. Предположим, что неосуществимый ход K-opt приводит к $M \geq 2$ циклам. Затем эти циклы могут быть исправлены с использованием по меньшей мере одного и не более M - 1 чередующихся циклов. Если используется только один чередующийся цикл, он должен содержать ровно 2M ребер. Если M - 1 чередующихся циклов используется, каждый из них должен содержать ровно 4 ребра. Следовательно, сконструированный возможный ход является ходом k-opt, где $K + \frac{2M}{2} \leq k \leq K + \frac{4(M-1)}{2}$, то есть $K + M \leq k \leq K + 2M - 2$. Поскольку M может быть K, мы можем заключить, что Правила 1-3 допускают непоследовательные ходы k-opt, где $K + 2 \leq k \leq 3K - 2$. Например, если невыполнимый ход 5-opt приводит к 5 циклам, это может быть отправной точкой для поиска непоследовательного хода 7-13-opt.

Правило 4 сводит к минимуму количество возможных начальных узлов. Таким образом, алгоритм пытается минимизировать количество возможных чередующихся циклов, которые необходимо исследовать.

Правило 5 является аналогом критерия положительного выигрыша для последовательных ходов (см. Раздел 2). Во время построения хода ни один чередующийся цикл не будет закрыт, если совокупный выигрыш плюс стоимость близкого края не будут положительными.
Чтобы еще больше сократить поиск, используется следующее правило жадности:

6. Последние три ребра чередующегося цикла должны быть такими, которые внесут наибольший вклад в общий выигрыш. Другими словами, учитывая чередующийся цикл $(s_1, s_2, \ldots, s_{2m-2}, s_{2m-1}, s_{2m}, s_1)$, величина

$$- c(s_{2m-2}, s_{2m-1}) + c(s_{2m-1}, s_{2m}) - c(s_{2m}, s_1)$$

должно быть максимальным.

Кроме того, пользователь LKH-2 может ограничить поиск непоследовательных перемещений, указав верхний предел ($Patching\_C$) для количества циклов, которые могут быть исправлены, и верхний предел ($Patching\_A$) для количества чередующихся циклов, которые будут использоваться для исправления.

Далее я опишу, как циклическое исправление реализовано в LKH-2. Чтобы сделать описание более понятным, я сначала покажу реализацию алгоритма, который выполняет циклическое исправление с использованием только одного чередующегося цикла. Алгоритм реализован в виде функции $PatchCycles$, которая вызывает рекурсивную вспомогательную функцию $PatchCyclesRec$.

```
GainType PatchCycles(int k, GainType Gain) {
    Node *s1, *s2, *sStart, *sStop;
    GainType NewGain;
    int M, i;

    FindPermutation(k);
    M = Cycles(k);
    if (M == 1 || M > Patching_C)
        return 0;
    CurrentCycle = ShortestCycle(M, k);
    for (i = 0; i < k; i++) {
        if (cycle[p[2 * i]] == CurrentCycle) {
            sStart = t[p[2 * i]];
            sStop = t[p[2 * i + 1]];
            for (s1 = sStart; s1 != sStop; s1 = s2) {
                t[2 * k + 1] = s1;
                t[2 * k + 2] = s2 = SUC(s1);
                if ((NewGain =
                    PatchCyclesRec(k, 2, M,
                                   Gain + C(s1, s2))) > 0)
                    return NewGain;
            }
        }
    }
    return 0;
}
```

Комментарии:

- Функция вызывается из внутреннего цикла BestKOptMoveRec:

```
if (t4 != t1 && Patching_C >= 2 &&
    (Gain = G2 - C(t4, t1)) > 0 &&    // rule 5
    (Gain = PatchCycles(k, Gain)) > 0)
    return Gain;
```

- Параметр k указывает, что, учитывая невозможное перемещение k-opt,

представленное узлами в глобальном массиве t[1..2k], функция должна попытаться найти выгодное возможное перемещение путем циклического исправления. Параметр Gain определяет коэффициент усиления при невозможном перемещении входного сигнала.

- Нам нужно иметь возможность обходить те узлы, которые принадлежат наименьшему компоненту цикла (правило 4), и для данного узла быстро определять, к какому из текущих циклов он принадлежит (правило 3). Для этой цели мы сначала определяем перестановку p, соответствующую порядку, в котором узлы в t[1..2k] встречаются в туре по часовой стрелке, начиная с t[1]. Например, p = (1 2 4 3 9 10 7 8 5 6) для невыполнимого 5-opt перемещения, показанного на рисунке 5.2.4. Далее вызывается функция Cycles для определения количества циклов и для связи с каждым узлом в t[1..2k] номер цикла, к которому он принадлежит. Выполнение 5-opt на рисунке 5.2.4 ход производит два цикла, один из которых представлен последовательностью узлов $(t_1, t_{10}, t_7, t_6, t_1)$, а другой представлен последовательностью узлов $(t_2, t_4, t_5, t_8, t_9, t_3, t_2)$. Узлы первой последовательности помечены знаком 1, узлы второй последовательности - знаком 2.

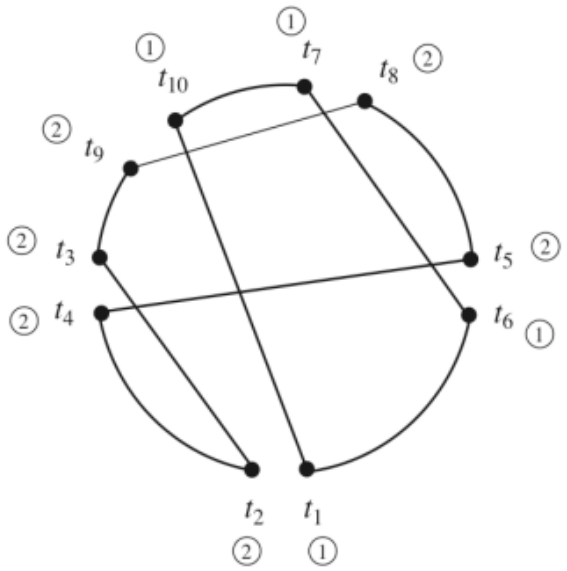

Figure 5.2.4. Non-feasible 5-opt move (2 cycles).

Затем вызывается функция $ShortestCycle$ (показанная ниже), чтобы найти кратчайший цикл. Функция возвращает номер цикла, который содержит наименьшее количество узлов.

```
int ShortestCycle(int M, int k) {
    int i, MinCycle, MinSize = INT_MAX;
    for (i = 1; i <= M; i++)
        size[i] = 0;
    p[0] = p[2 * k];
    for (i = 0; i < 2 * k; i += 2)
        size[cycle[p[i]]] +=
            SegmentSize(t[p[i]], t[p[i + 1]]);
    for (i = 1; i <= M; i++) {
        if (size[i] < MinSize) {
            MinSize = size[i];
            MinCycle = i;
        }
    }
    return MinCycle;
}
```

- Сегменты тура кратчайшего цикла проходят, используя тот факт, что t[p[2i]] и t[p[2i + 1]] являются двумя конечными точками для сегмента тура цикла, для $0 \leq i < k$ и p[0] = p[2k]. Частью какого цикла является сегмент тура, можно определить, просто извлекая номер цикла, связанный с одной из двух его конечных точек.
- Каждое ребро тура самого короткого цикла теперь может использоваться в качестве первого внешнего ребра (s1, s2) чередующегося цикла. Чередующийся цикл строится с помощью рекурсивной функции $PatchCycleRec$, показанной ниже.

```
GainType PatchCyclesRec(int k, int m, int M, GainType G0) {
    Node *s1, *s2, *s3, *s4;
    GainType G1, G2, G3, G4, Gain;
    int NewCycle, cycleSaved[1 + 2 * k], i;

    s1 = t[2 * k + 1];
    s2 = t[i = 2 * (k + m) - 2];
    incl[incl[i] = i + 1] = i;
    for (i = 1; i <= 2 * k; i++)
        cycleSaved[i] = cycle[i];
    for (each candidate edge (s2, s3)) {
        if (s2 != PRED(s2) && s3 != SUC(s2) &&
            (NewCycle = FindCycle(s3, k)) != CurrentCycle) {
            t[2 * (k + m) - 1] = s3;
            G1 = G0 - C(s2, s3);
            for (each s4 in {PRED(s3), SUC(s3)}) {
                if (!Deleted(s3, s4, k)) {
                    t[2 * (k + m)] = s4;
                    G2 = G1 + C(s3, s4);
                    if (M > 2) {
                        for (i = 1; i <= 2 * k; i++)
                            if (cycle[i] == NewCycle)
                                cycle[i] = CurrentCycle;
                        if ((Gain =
                                PatchCyclesRec(k, m + 1, M - 1,
                                               G2)) > 0)
                            return Gain;
                        for (i = 1; i <= 2 * k; i++)
                            cycle[i]= cycleSaved[i];
                    } else if (s4 != s1 &&
                               (Gain = G2 - C(s4, s1)) > 0) {
                        incl[incl[2 * k + 1] = 2 * (k + m)] =
                                                2 * k + 1;
                        MakeKOptMove(k + m);
                        return Gain;
                    }
                }
            }
        }
    }
    return 0;
}
```

Комментарии:

- Алгоритм довольно прост и очень напоминает функцию $BestKOptMoveRec$. Параметр k используется для указания того, что неосуществимый k-opt ход должен использоваться в качестве отправной точки для поиска выгодного возможного хода. Параметр m определяет текущее количество внешних ребер строящегося переменного цикла. Параметр M задает текущее количество компонентов цикла ($M \geq 2$). Последний параметр, G0, определяет накопленный выигрыш.
- Если найден выгодный выполнимый ход, то этот ход выполняется путем вызова функции $MakeKOptMove(k + m)$, прежде чем будет возвращен достигнутый выигрыш.
- Перемещение представлено узлами в глобальном массиве t, где первые

2k элементов являются узлами данного невыполнимого последовательного перемещения k-opt, а последующие элементы являются узлами чередующегося цикла. Для того, чтобы иметь возможность быстро определить, является ли ребро внутренним или внешним текущего перемещения, мы поддерживаем массив, $incl$, такой, что $incl[i] = j$ и $incl[j] = i$ истинно тогда и только тогда, когда ребро (t[i], t[j]) является внутренним ребром. Например, на рисунке 5.2.4 $incl = [10, 3, 2, 5, 4, 7, 6, 9, 8, 1]$. Это легко видеть, что нет причин сохранять аналогичную информацию о внешних ребрах, поскольку они всегда являются теми ребрами (t[i], t [i + 1]), для которых i нечетно.

- Пусть s2 - последний узел, добавленный в t-последовательность. При каждом рекурсивном вызове $PatchCyclesRec$ t-последовательность расширяется двумя узлами, s3 и s4, так что
  a. (s2, s3) является ребром-кандидатом,
  b. s3 принадлежит еще не посещенному компоненту цикла,
  c. s4 является соседом s3 в туре и
  d. ребро (s3, s4) ранее не удалялось.
- Перед рекурсивным вызовом функции все номера циклов этих узлов t[1..2k], которые принадлежат компоненту цикла s3, изменяются на номер текущего компонента цикла (который равен номеру компонента цикла s2).

Временная сложность для вызова $PatchCyclesRec$ может быть оценена исходя из временных сложностей для задействованных подопераций. Подоперации могут быть реализованы с учетом временных сложностей, приведенных в таблице 5.2.2. Пусть d обозначает максимальную степень узла в графе-кандидате. Тогда легко видеть, что наихудшая временная сложность для каждого вызова равна $O(nd^k KlogK + \sqrt{n})$. Коэффициент n обусловлен тем фактом, что самый короткий цикл в наихудшем случае содержит не более n/2 узлов. Коэффициент KlogK отражает, что перестановка p определяется путем сортировки узлов t-последовательности. Сравнения узлов выполняются с помощью операции $BETWEEN\ (a, b, c)$, который за постоянное время может определить, находится ли узел b между двумя другими узлами, a и c, в туре.

Operation	*Complexity*
PRED	$O(1)$
SUC	$O(1)$
FindPermutation	$O(K\log K)$
Cycles	$O(K)$
ShortestCycle	$O(K)$
FindCycle	$O(\log K)$
Deleted	$O(K)$
MakeKOptMove	$O(\sqrt{n})$

Table 5.2.2. Complexities for the sub-operations of PatchCyclesRec.

Алгоритм, описанный выше, допускает циклическое исправление только с помощью одного чередующегося цикла. Однако относительно просто расширить алгоритм таким образом, чтобы можно было использовать более одного чередующегося цикла. К функции $patchCycleRec$ необходимо добавить всего несколько строк кода.

Во-первых, следующий фрагмент кода вставляется сразу после рекурсивного вызова:

```
if (Patching_A >= 2 &&
    (Gain = G2 - C(s4, s1)) > BestCloseUpGain) {
    Best_s3 = s3;
    Best_s4 = s4;
    BestCloseUpGain = Gain;
}
```

Если пользователь решил, что для исправления могут использоваться два или более чередующихся цикла (Patching_A ≥ 2), этот код находит пару узлов (Best_s3, Best_s4), чтто максимизирует выигрыш от замыкания текущего переменного цикла (правило 6):

$$-c(s_2, s_3) + c(s_3, s_4) - c(s_4, s_1)$$

Текущий лучший коэффициент усиления крупным планом, $BestCloseUpGain$, инициализируется равным нулю (правило 5).

Во-вторых, последний оператор $PatchCyclesRec$ (оператор возврата) заменяется следующим кодом:

```
Gain = 0;
if (BestCloseUpGain > 0) {
    int OldCycle = CurrentCycle;
    t[2 * (k + m) - 1] = Best_s3;
    t[2 * (k + m)] = Best_s4;
    Patching_A--;
    Gain = PatchCycles(k + m, BestCloseUpGain);
    Patching_A++;
    for (i = 1; i <= 2 * k; i++)
        cycle[i]= cycleSaved[i];
    CurrentCycle = OldCycle;
}
return Gain;
```

Обратите внимание, что это вызывает рекурсивный вызов $PatchCycles$ (не $Patch\ Cycles\ Rec$).

Алгоритм, описанный в этом разделе, несколько упрощен по отношению к

алгоритму, реализованному в LKH-2. Например, когда возникают два цикла, LKH-2 попытается исправить их не только с помощью чередующегося цикла, состоящего из 4 ребер (2-opt ход), но также с помощью чередующегося цикла, состоящего из 6 ребер (3-opt ход).

5.3 Определение целесообразности перехода

Возьмем тур T и k-opt ход, как можно быстро определить, выполним ли ход, то есть будет ли результатом тур, если ход будет применен к T? Рассмотрим рисунок 5.3.1.

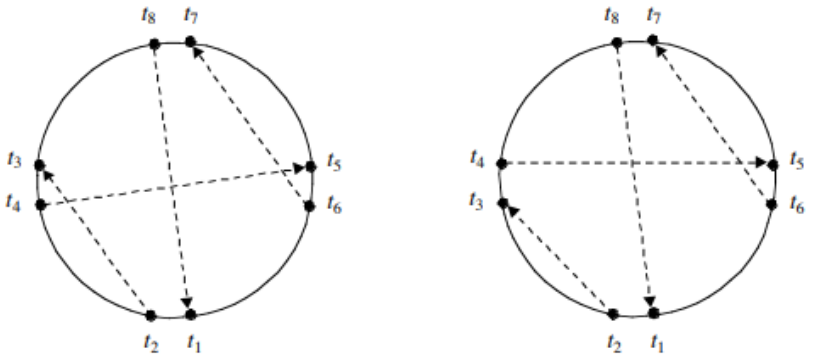

Figure 5.3.1. (a) Feasible 4-opt move. (b) Non-feasible 4-opt move.

На рисунке 5.3.1a показан возможный ход с 4 вариантами. Выполнение перемещения приведет ровно к одному циклу (туру), а именно $(t_2t_3 - t_8t_1 - t_6t_7 - t_5t_4 - t_2)$. С другой стороны, 4-opt ход на рисунке 5.3.1b неосуществим, поскольку результатом будут два цикла $t_2t_3 - t_2$ и $(t_4t_5 - t_7t_6 - t_1t_8 - t_4)$.

Решение о том, выполним ли k-opt ход, является частой проблемой в алгоритме. Каждый раз, когда обнаруживается выгодный ход, он проверяется на выполнимость. Невыигрышные ходы также проверяются, чтобы убедиться, что они могут войти в текущую цепочку последовательных ходов. Следовательно, очень важно, чтобы такие проверки проходили быстро.

Простой алгоритм проверки выполнимости состоит в том, чтобы начать с произвольного узла, а затем переходить от узла к узлу в графе, который возник бы, если бы перемещение было выполнено, пока снова не будет достигнут начальный узел. Перемещение осуществимо тогда и только тогда, когда количество посещенных узлов равно количеству узлов n в исходном туре. Однако сложность этого алгоритма составляет $O(n)$, что делает его непригодным для данной работы.

Можем ли мы построить более быстрый алгоритм? Да, потому что нам не нужно посещать каждый узел в цикле. Мы можем ограничиться только посещением t-узлов, которые представляют собой движение. Другими словами, мы можем переходить от t-узла к t-узлу. Перемещение осуществимо тогда и только тогда, когда все t-узлы перемещения посещаются таким образом. Например, если мы начнем с узла t_6 на рисунке 5.3.1a и перейдем от t-узла к t-узлу, следуя направлению стрелок, то все t-узлы будут посещены в следующей последовательности: $t_6, t_7, t_5, t_4, t_2, t_3, t_8, t_1$.

Легко перейти от одного t-узла, t_a, к другому, t_b, если ребро (t_a, t_b) является внутренним ребром. Нам нужно только поддерживать массив, $incl$, который представляет текущие внутренние ребра. Если (t_a, t_b) является преимуществом для k-opt хода $(1 \leq a, b \leq 2k)$, этот факт регистрируется в массиве путем установки incl[a] = b и incl[b] = a. Это означает, что каждый такой переход может быть выполнен за постоянное время (путем поиска по таблице).

С другой стороны, не очевидно, как мы можем пропустить те узлы, которые не являются t-узлами. Оказывается, небольшая предварительная обработка решает проблему. Если мы знаем циклический порядок, в котором t-узлы встречаются в исходном туре, то становится легко пропустить все узлы, которые лежат между двумя t-узлами в туре. Для данного t-узла нам просто нужно выбрать либо его предшественника, либо его преемника в это циклический порядок. Какой из двух случаев нам следует выбрать, можно определить за постоянное время. Таким образом, этот вид перехода может быть выполнен за постоянное время, если t-узлы были отсортированы. Теперь я покажу, что эта сортировка может быть выполнена за $O(klogk)$ времени.

Во-первых, мы понимаем, что нет необходимости сортировать все t-узлы. Мы можем ограничиться сортировкой половины узлов, а именно для каждого внешнего ребра первой конечной точки, встреченной при прохождении тура в заданном направлении. Если на рисунке 5.3.1 выбрано направление “по часовой стрелке”, мы можем ограничить сортировку четырьмя узлами t_1, t_4, t_5 и t_8. Если результат сортировки представлен перестановкой p_{half}, то p_{half} будет равен (1 4 8 5). Эта перестановка может быть легко расширена до полной перестановки, содержащей все индексы узлов, p = (1 2 4 3 8 7 5 6). Отсутствующие индексы узлов вставляются с использованием следующего правила: если значение $p_{half}[i]$ нечетное, то вставьте значение $p_{half}[i]$ + 1 после значения $p_{half}[i]$, в противном случае вставьте значение $p_{half}[i]$ - 1 после значения $p_{half}[i]$.

Пусть перемещение представлено узлами t[1..2k]. Затем функция $FindPermutation$, показанная ниже, способна найти перестановку p[1..2k], которая соответствует порядку их посещения, когда тур проходит в правильном направлении. Кроме того, функция определяет q[1..2k] как обратную перестановку к p, то есть перестановку, для которой q[p[i]] = i для $1 \leq i \leq 2k$.

```
void FindPermutation(int k) {
    int i,j;
    for (i = j = 1; j <= k; i += 2, j++)
        p[j] = (SUC(t[i]) == t[i + 1]) ? i : i + 1;
    qsort(p + 2, k - 1, sizeof(int), Compare);
    for (j = 2 * k; j >= 2; j -= 2) {
        p[j - 1] = i = p[j / 2];
        p[j] = i & 1 ? i + 1 : i - 1;
    }
    for (i = 1; i <= 2 * k; i++)
        q[p[i]] = i;
}
```

Комментарии:

- Сначала k индексов, подлежащих сортировке, помещаются в p. Это занимает O(k) времени. Далее выполняется сортировка. Первый элемент p, p[1] является фиксированным и не принимает участия в сортировке. Таким образом, сортируются только элементы k-1. Сортировка выполняется библиотечной функцией сортировки C, qsort, с использованием функции сравнения, показанной ниже:

```
int Compare(const void *pa, const void *pb) {
    return BETWEEN(t[p[1]], t[*(int *) pa], t[*(int *) pb])
           ? -1 : 1;
}
```

Операция $BETWEEN(a, b, c)$ определяет в постоянное время, находится ли узел b между узлом a и узлом c на маршруте в $SUC - direction$. Поскольку количество сравнений, выполняемых qsort, в среднем равно $O(klogk)$, и каждое сравнение занимает постоянное время, процесс сортировки в среднем занимает $O(klogk)$ времени.

- После сортировки строится полная перестановка p и ее обратная функция q. Это можно сделать за $O(k)$ времени.
- Из приведенного выше анализа мы видим, что среднее время для вызова $FindPermutation$ с аргументом k равно $O(klogk)$.

Определив p и q, мы можем быстро определить, выполнимо ли перемещение k-opt, представленное содержимым массивов t и incl. Функция, показанная ниже, показывает, как этого можно достичь.

```
int FeasibleKOptMove(int k) {
    int Count = 1, i = 2 * k;
    FindPermutation(k);
    while ((i = q[incl[p[i]]] ^ 1) != 0)
        Count++;
    return (Count == k);
}
```

Комментарии:

- На каждой итерации цикла while посещаются два конечных узла внутреннего ребра, а именно t[p[i]] и t[incl[p[i]]]. Обратная перестановка, q, позволяет пропустить возможные t-узлы между t [incl[p[i]]] и следующим t-узлом в цикле. Если позиция incl[p[i]] в p четная, то на следующей итерации i должна быть равна этой позиции плюс один. В противном случае оно должно быть равно этой позиции минус единица.
- Начальное значение для i равно 2k. Цикл завершается, когда i становится равным нулю, что происходит при посещении узла t[p[1]] (начиная с 1 ^ 1 = 0, где ^ - исключающий оператор OR). Цикл всегда завершается, поскольку оба t[p[2k]] и t[p[1]] принадлежат циклу, который проходит цикл, и ни один узел не посещается более одного раза.
- Легко видеть, что цикл выполняется за O(k) времени. Поскольку сортировка, выполняемая вызовом $FindPermutation$, в среднем занимает $O(klogk)$ времени, мы можем заключить, что средняя временная сложность для функции $FeasibleKOptMove$ составляет $O(klogk)$. Обычно k очень мало по сравнению с общим числом узлов n. Таким образом, мы получили алгоритм, который эффективен на практике.
- Чтобы увидеть конкретный пример того, как работает алгоритм, рассмотрим 4-opt ход на рисунке 5.3.1a. Перед циклом while верно следующее:

p = (1 2 4 3 8 7 5 6)
q = (1 2 4 3 7 8 6 5)
incl = (8 3 2 5 4 7 6 1)

Управляющей переменной i присваиваются значения 8, 7, 2, 5, 0 в таком порядке, и количество переменных увеличивается в 3 раза. Поскольку количество становится равным 4, возможен ход с 4 вариантами.

Для перемещения с 4-opt на рисунке 5.3.b цикл while запускается с

p = (1 2 3 4 8 7 5 6)
q = (1 2 3 4 7 8 6 5)
incl = (8 3 2 5 4 7 6 1)

Управляющей переменной i присваиваются значения 8, 7, 5, 0 в таком порядке. Поскольку количество здесь становится 3 (что не равно 4), ход с 4 вариантами невозможен.

5.4 Выполнение возможного хода

Чтобы упростить выполнение выполнимых ходов k-opt, можно использовать следующий факт: любой ход k-opt $(k \geq 2)$ эквивалентен конечной последовательности ходов 2-opt [9, 28]. В случае 5-opt ходов можно показать, что любой 5-opt ход эквивалентен последовательности не более чем из пяти 2-opt ходов. Любой 3-opt ход, а также любой 4-opt ход эквивалентны последовательности не более чем из трех 2-выбор ходов. В общем, любой возможный ход k-opt может быть выполнен не более чем k ходами 2-opt. Доказательство см. в [30].

Пусть $FLIP(a,\ b,\ c,\ d)$ обозначает операцию замены двух ребер (a, b) и (c, d) тура двумя ребрами (b, c) и (d, a). Затем 4-opt ход, изображенный на рисунке 5.4.1, может быть выполнен с помощью следующей последовательности $FLIP\ -$ операций:

$$FLIP(t_2, t_1, t_8, t_7)$$
$$FLIP(t_4, t_3, t_2, t_7)$$
$$FLIP(t_7, t_4, t_5, t_6)$$

Выполнение flip проиллюстрировано на рисунке 5.4.2.

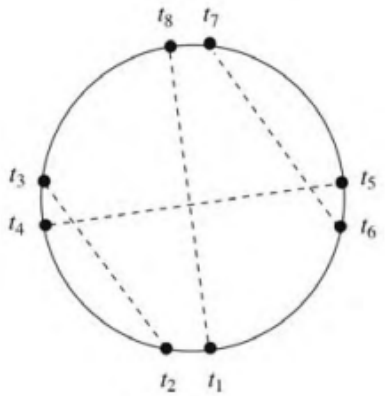

Figure 5.4.1. Feasible 4-opt move.

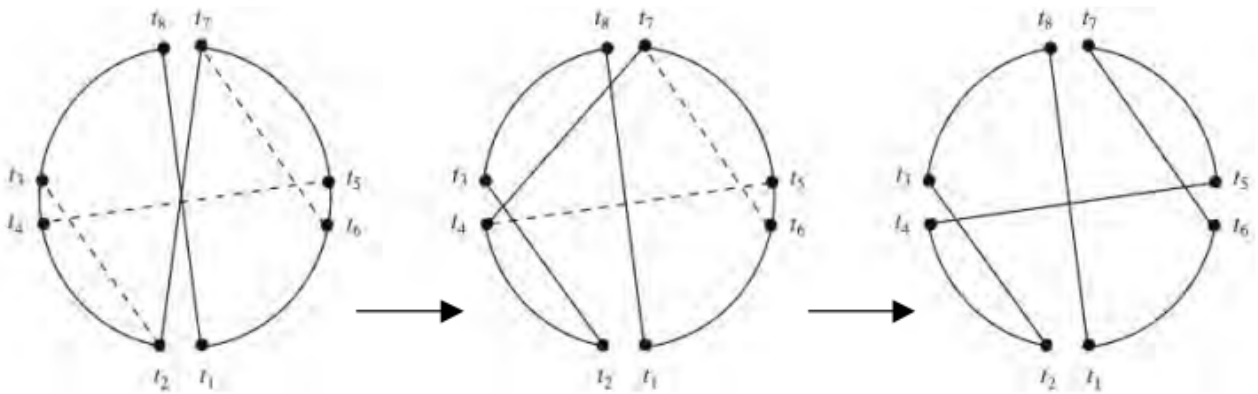

Figure 5.4.2. Execution of the 4-opt move by means of 3 flips.

4-opt ход на рисунке 5.4.1 может быть выполнен многими другими последовательностями переворотов, например:

$$FLIP(t_2, t_1, t_6, t_5)$$
$$FLIP(t_5, t_1, t_8, t_7)$$
$$FLIP(t_3, t_4, t_5, t_7)$$
$$FLIP(t_2, t_6, t_7, t_3)$$

Однако эта последовательность содержит на одну операцию FLIP больше, чем предыдущая последовательность. Следовательно, первый из этих двух является предпочтительным.

Центральный вопрос теперь заключается в том, как для любого возможного хода k-opt найти последовательность переворотов, соответствующую ходу. Кроме того, мы хотим, чтобы последовательность была как можно короче.

В дальнейшем я покажу, что ответ на этот вопрос можно найти с помощью преобразование задачи в эквивалентную задачу, имеющую известное решение. Рассмотрим рисунок 5.4.3, на котором показан итоговый тур после применения 4-опционного хода. Обратите внимание, что любой ход с 4 опциями может привести к отмене до 4 сегментов тура. На рисунке каждый из этих сегментов помечен целым числом, числовое значение которого соответствует порядку, в котором сегмент встречается в результирующем туре. Знак целого числа указывает, имеет ли сегмент в результирующем туре ту же (+) или противоположную ориентацию (-) как в оригинальном туре. Начиная с узла t2, сегменты в новом туре выполняются в порядке с 1 по 4. Отрицательный знак, связанный с сегментами 2 и 4, указывает на то, что они были перевернуты относительно их направления в исходном туре (по часовой стрелке).

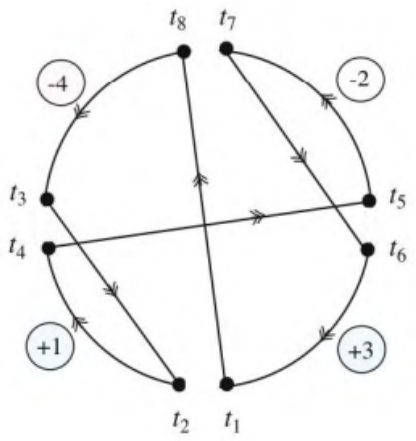

Figure 5.4.3. Segments labeled with orientation and rank.

Если мы запишем метки сегментов в том порядке, в котором сегменты встречаются в исходном туре, мы получим следующую последовательность, так называемую подписанную перестановку.

(+1 -4 -2 +3)

Мы хотим, чтобы эта последовательность была преобразована в последовательность (перестановка идентификаторов):

(+1 +2 +3 +4)

Теперь обратите внимание, что операция FLIP соответствует изменению как порядка, так и знаков элементов сегмента знаковой перестановки. На рисунке 5.4.3 показано выполнение последовательности переворачивания

$$FLIP(t_2, t_1, t_8, t_7)$$

$$FLIP(t_4, t_3, t_2, t_7)$$

$$FLIP(t_7, t_4, t_5, t_6)$$

соответствует следующей последовательности разворотов со знаком:

(+1 -4 -2 +3)

(+1 -4 -3 +2)

(+1 -2 +3 +4)

(+1 +2 +3 +4)

Перевернутые сегменты подчеркнуты.

Предположим теперь, что, учитывая знаковую перестановку {1, 2, ..., 2k}, мы можем определить кратчайшую возможную последовательность разворотов со знаком, которая преобразует перестановку в тождественную перестановку(+1, +2, ..., + 2k). Затем, учитывая возможный ход k-opt, мы также сможем найти кратчайшую возможную последовательность операций FLIP, которые могут быть использованы для выполнения хода.

Однако эта проблема, называемая сортировкой знаковых перестановок по обратным, является хорошо изученной проблемой в вычислительной молекулярной биологии. Проблема возникает, например, когда кто-то хочет определить генетическое расстояние между двумя видами, то есть минимальное количество мутаций, необходимых для преобразования генома одного вида в геном другого. Наиболее важными мутациями являются те, которые перестраивают геномы путем переворота, и поскольку порядок генов в геноме может быть описан перестановкой, проблема состоит в том, чтобы найти наименьшее число переворотов, которые преобразуют одну перестановку в другую.

Проблему можно более формально определить следующим образом. Пусть $\pi = (\pi_1, \ldots, \pi_n)$ быть перестановка {1, ..., n}. Разворот $\rho(i, j)$ - это инверсия сегмента $(\pi_i, \ldots, \pi_j)$ из π, то есть он преобразует перестановку $(\pi_1 \ldots \pi_i \ldots \pi_j \ldots \pi_n)$

в $(\pi_1 \dots \pi_j \dots \pi_i \dots \pi_n)$. Проблема сортировки по разворотам (SBR) - это проблема нахождения кратчайшей возможной последовательности разворотов ($\rho_1, \dots, \rho_{d(n)}$) такой, что $\pi\rho_1 \dots \rho_{d(n)} = (1\ 2 \dots\ n - 1\ n)$, где d(n) называется расстоянием разворота для π.

Специальная версия задачи определена для подписанных перестановок. Знаковая перестановка $\sigma = (\sigma_1, \dots, \sigma_m)$ получается из обычной перестановки $\pi = (\pi_1 \dots \pi_m)$ путем замены каждого из его элементов π_i либо на $+\pi_i$, либо на $-\pi_i$. Изменение $\rho(i,j)$ знаковой перестановки σ изменяет как порядок, так и знаки элементов $(\sigma_i \dots \sigma_j)$. Проблема сортировки знаковых перестановок по разворотам (SSBR) - это проблема нахождения кратчайшей возможной последовательности разворотов $(\rho_1 \dots \rho_{d(n)})$ такой, что $\sigma\rho_1 \dots \rho_{d(n)} = (+1\ +2 \dots\ +(m-1)\ m)$.