**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №2**

**по дисциплине «Построение и Анализ Алгоритмов»**

Тема: Кратчайшие пути в графе: коммивояжёр

|  |  |  |
| --- | --- | --- |
| Студент гр. 3341 |  | Ягудин Д.Р. |
| Преподаватель |  | Фирсов М.А. |

Санкт-Петербург

2025

## Цель работы.

Целью данной работы является изучение задачи о комммивояжёре, различных методов её решения, а также написание программы, реализующей точный метод решения на основе метода ветвей и границ и приближенный способ.

## Задание.

Напишите программу, решающую задачу коммивояжера. Нужно найти кратчайший маршрут, который проходит через все заданные города ровно один раз и возвращается в исходный город. Не все города могут быть напрямую связаны друг с другом.

**Входные данные:**

* n - количество городов (5 ≤ n ≤ 15).
* Матрица расстояний между городами размером n×n , где graph[i][j] обозначает расстояние от города i до города j. Если graph[i][j]=0 (и i ≠ j ), это означает, что прямого пути между городами нет.

**Выходные данные:**

* Минимальная стоимость маршрута, проходящего через все города и возвращающегося в начальный город.
* Оптимальный путь в виде последовательности посещаемых городов, начинающегося и заканчивающегося в начальном городе.
* Если такого пути не существует, вывести "no path".

**Sample Input 1:**

5

0 1 13 23 7

12 0 15 18 28

21 29 0 33 28

23 19 34 0 38

5 40 7 39 0

**Sample Output 1:**

78

0 4 2 3 1 0

**Sample Input 2:**

3

0 1 0

1 0 1

0 1 0

**Sample Output 2:**

no path

## Индивидуализация.

8. Точный метод: динамическое программирование (не МВиГ), итеративная реализация.

Приближённый алгоритм: АЛШ-2.

Требование перед сдачей: прохождение кода в задании 3.1 на Stepik.

Замечание к варианту 8. АЛШ-2 начинать со стартовой вершины.

## Описание точного алгоритма.

**Идея алгоритма:**  
Используем динамическое программирование для поиска кратчайшего Гамильтонова цикла.

* dp[mask][i] — минимальная стоимость пути, проходящего через вершины маски mask и заканчивающегося в вершине i.
* parent[mask][i] — вершина, из которой пришли в i для маски mask (используется для восстановления пути).

**Параметры:**

* n — количество городов.
* graph — матрица смежности, где graph[i][j] — вес ребра между городами i и j.

**Инициализация:**

* size = 1 << n — количество возможных масок (2^n).
* dp[1][0] = 0 — база ДП: начинаем с города 0 (маска содержит только 0-й бит).

**Переходы между состояниями:**  
Для каждой маски mask и вершины i:

1. Если dp[mask][i] не вычислено (INF), переходим к следующей итерации.
2. Для всех вершин j, не входящих в маску mask:
   * Если существует ребро i → j (graph[i][j] > 0), вычисляем новую маску new\_mask = mask | (1 << j).
   * Обновляем dp[new\_mask][j] и parent[new\_mask][j], если найден более короткий путь.

**Восстановление пути:**

1. После заполнения таблицы dp проверяем возможность возврата в стартовый город (0) из всех вершин j.
2. Выбираем вершину best\_j, дающую минимальную стоимость полного цикла.
3. Восстанавливаем путь, двигаясь от best\_j к стартовой вершине с помощью массива parent.

**Сложность:**

* Время: O(2^n \* n^2) — три вложенных цикла (по маскам, вершинам i, вершинам j).
* Память: O(2^n \* n) — хранение массивов dp и parent.

**Описание приближенного алгоритма (метод ближайшего соседа + 2-opt)**

**Идея алгоритма:**

**Алгоритм ближайшего соседа:**

* + На каждом шаге выбирается город с минимальным расстоянием от текущего.
  + Построенный путь улучшается с помощью локальной оптимизации 2-opt.

**Шаги алгоритма:**

**Построение начального пути:**

* Стартуем из заданного города (start\_city).
* На каждом шаге:
  + Выбираем непосещенный город j с минимальным весом ребра current → j.
  + Добавляем j в путь.
* Проверяем возможность возврата в стартовый город.

**2. Оптимизация 2-opt:**

* Итеративно улучшаем путь, переставляя пары вершин:
  + Для всех пар (i, j) в пути проверяем, уменьшит ли стоимость перестановка подпути между i и j.
  + Если да, переворачиваем подпуть и обновляем стоимость.

**Условия завершения:**

* Если все города посещены и существует обратное ребро в стартовый город — путь найден.
* Если на любом этапе нет допустимых переходов — маршрут невозможен.

**Особенности реализации:**

* Фиксированный стартовый город (0 по умолчанию).
* Проверка на существование исходящих рёбер из стартового города.

**Сложность:**

* **Ближайший сосед:** O(n^2) — для каждого города перебираем всех соседей.
* **2-opt:** O(k \* n^2), где k — количество итераций улучшения.

## Описание функций и структур данных.

ДП (итеративная реализация):

• *class Solution* – класс, реализующий точное решение задачи коммивояжёра (TSP) методом динамического программирования с оптимизацией битовых масок. Содержит приватные поля: *\_dp* (таблица динамического программирования), *\_start* (стартовая вершина), *\_n* (количество вершин).

• метод *pair<int, vector<int>> TSP(vector<unordered\_map<int, int>>& g, int start=0)* – вычисляет минимальный гамильтонов цикл. Принимает: *g* (граф в виде списка смежности) и *start* (стартовая вершина, по умолчанию *0*). Возвращает пару: суммарный вес цикла (*-1* при отсутствии решения) и вектор вершин пути.

• метод *void fillDP(vector<unordered\_map<int, int>>& g)* – заполняет таблицу динамического программирования. Принимает граф *g*. Обрабатывает все возможные маски состояний, обновляя стоимости переходов между вершинами. Условия переходов: следующий узел не посещён или завершение цикла в стартовую вершину.

• метод *vector<int> restorePath(vector<unordered\_map<int, int>>& g)* – восстанавливает путь из таблицы *DP*. Принимает граф *g*. Возвращает вектор вершин пути, найденного обратным проходом от конечного состояния к начальному. Проверяет существование рёбер и соответствие стоимостям в таблице *DP*.

• метод *void printDP(int highlightMask = -1, int highlightNode = -1)* – выводит таблицу динамического программирования с форматированием. Принимает маску *highlightMask* и узел *highlightNode* для подсветки ячейки (значения по умолчанию: -1). Формат: колонки шириной 8 символов, *INT\_MAX* как *inf*, подсвеченные ячейки в [], заголовок содержит маски и узлы.

• метод *void printDPUpdateDebug(int mask, int new\_mask, int u, int v, int w, int old\_value)* – выводит информацию об обновлении таблицы *DP*. Принимает текущую маску *mask*, новую маску *new\_mask*, узлы *u* и *v*, вес ребра *w*, предыдущее значение *old\_value*. Формат: бинарные маски (16 бит), переход *u→v*, сравнение старых и новых значений.

• метод *void printDPResultDebug(int new\_mask, int v, bool is\_update, int new\_value)* – выводит результат обновления таблицы. Принимает новую маску *new\_mask*, узел *v*, флаг *is\_update* и значение *new\_value*. Формат: статус *ACCEPTED/REJECTED*, новое значение (*inf* при *INT\_MAX*), таблица с подсветкой при обновлении.

• метод *void printPathStep(int mask, int from, int to, int edge\_weight)* – выводит шаг восстановления пути. Принимает маску *mask*, узлы *from* и *to*, вес ребра *edge\_weight*. Формат: бинарные маски, переход *from* ← *to*, обновлённая маска.

• метод *void printFinalPath(const vector<int>& path, int total\_cost)* – выводит итоговый путь. Принимает вектор *path* и стоимость *total\_cost*. Формат: заголовок/футер (79 символов), путь в виде 0 → 1 → ... → 0, стоимость с выравниванием.

АЛШ-2:

• *class Solution* – класс, реализующий алгоритм АЛШ-2 для задачи коммивояжёра (TSP). Содержит методы и структуры для построения приближённого маршрута.

• *метод pair<int, vector<int>> TSP(vector<unordered\_map<int, int>>& g, int start=0)* – находит приближённое решение TSP. Принимает граф *g* (список смежности: вершина → {сосед: вес}) и стартовую вершину *start* (по умолчанию *0*). Возвращает пару: суммарный вес маршрута (*-1* при невозможности обхода) и вектор вершин пути.

• структура *Edge* – описывает ребро графа. Содержит поля: *from* (исходная вершина), *to* (конечная вершина), *w* (вес ребра).

• структура *DSU* – реализует систему непересекающихся множеств для построения МОД. Содержит массивы p (родители вершин) и r (ранги). Методы: *get(int x)* (возвращает корень множества для *x*), *join(int x, int y)* (объединяет множества *x* и *y*).

• метод *int next(vector<unordered\_map<int, int>>& g, int v)* – выбирает следующую вершину для добавления в путь. Принимает граф *g* и текущую вершину *v*. Возвращает индекс вершины с минимальной оценкой *(w + L)* или *-1*.

• метод *int spanningTreeW(vector<unordered\_map<int, int>>& g, int to)* – вычисляет вес МОД для оставшихся вершин. Принимает граф *g* и кандидата to (следующую вершину). Возвращает оценку стоимости завершения маршрута через вес МОД.

Разработанный программный код см. в приложении А.

Результаты тестирования см. в приложении Б.

## Выводы.

В результате выполнения данной лабораторной работы была изучена задача о коммивояжёре и различные методы её решения. На языке С++ написаны алгоритмы, реализующие точное решение методом динамического программирования (итеративная реализация) и приближенное решение методом АЛШ-2.

# Приложение А Исходный код программы

Название файла: *solution.cpp* (АЛШ-2)

#include <vector>

#include <unordered\_map>

#include <unordered\_set>

#include <climits>

#include <algorithm>

#include <numeric>

#define DEBUG 1

#if DEBUG == 1

#include <iostream>

#include <iomanip>

#include <sstream>

#endif

using namespace std;

/\*\*

\* Solution

\*

\* Этот класс предоставляет решение для TSP, используя жадный подход в сочетании с

\* оценкой минимального остовного дерева для локальных решений.

\*/

class Solution {

public:

/\*\*

\*

\* g Граф, представленный в виде списка смежности (узел -> {сосед: вес})

\* start Начальный индекс узла (по умолчанию = 0)

\* возвращается Пара, содержащая:

\* - Певое: общая стоимость тура (-1, если допустимого тура не существует)

\* - Второе: вектор, содержащий узлы пути в порядке

\*/

pair<int, vector<int>> TSP(vector<unordered\_map<int, int>>& g, int start=0) {

// Инициализация переменных

\_n = g.size();

\_vis.assign(\_n, false);

\_path.clear();

\_edges.clear();

#if DEBUG == 1

cout << "Initialization\n"

<< "Nodes: " << \_n << " Start: " << start << "\n"

<< "\n";

#endif

// Собираем все ребра для последующего их использования в построении МОД

for (int i = 0; i < \_n; ++i)

for (auto& [to, w] : g[i])

\_edges.push\_back({i, to, w});

// Сортируем веса ребер (в будущем пригодится для алгоритма Краскала)

sort(\_edges.begin(), \_edges.end(), [](Edge& a, Edge& b)

{ return a.w < b.w; }

);

// Строим путь с использованием эвристики

int v = start;

\_path.push\_back(start);

while(true) {

\_vis[v] = true; // Метим текущюю вершину как посещенную

int u = next(g, v); // Получаем следующюю лучшую ноду

if (u == -1) break; // Если вершин не осталось

\_path.push\_back(u);

v = u;

}

\_path.push\_back(start); // Завершаем путь в точке откуда пришли

// Проверяем все ли ноды вошли в путь

if (!all\_of(\_vis.begin(), \_vis.end(), [](bool el){ return el; }))

return {-1, {}};

// Считаем итоговую сумму по весам ребер

int sm = 0;

for (int i = 0; i < \_path.size()-1; ++i)

sm += g[\_path[i]][\_path[i+1]];

#if DEBUG == 1

cout << "\n" << "Path: ";

for(auto& node : \_path) cout << node << " ";

cout << "\n Total cost: " << sm << "\n\n";

#endif

return {sm, \_path};

}

private:

/\*\*

\* Стурктура Edge

\* Хранит данные ребер графа

\*/

struct Edge {

int from, to;

int w;

};

/\*\*

\* Система непересекающихся множеств (DSU)

\* Необходима для эффктивной проверки связности компонент в графе при поиске

\* минимального останового дерева по алгоритму Краскала

\*/

struct DSU {

DSU(int \_n)

: p(\_n)

, r(\_n, 0)

{ iota(p.begin(), p.end(), 0); } // Инициализируем каждую вершину как независимое множество

// Сжатие пути для эффективного нахождения корня

int get(int x) {

return (p[x] == x) ? x : p[x] = get(p[x]);

}

// Объединение по рангу

void join(int x, int y) {

x = get(x); y = get(y);

if(r[x] > r[y]) swap(x, y);

p[x] = y;

if(r[x] == r[y]) ++r[y];

}

vector<int> p, r;

};

private:

int \_n; // Количество нод в графе

vector<Edge> \_edges; // Все ребра в графе

vector<bool> \_vis; // Вектор для просмотра какие вершины уже были посещены

vector<int> \_path; // Для запоминания пути

private:

/\*\*

\* Выбирает следующий узел для посещения, используя жадный подход с оценкой MST

\* g Список смежности графа

\* v Текущая нода

\* Возвращает самую лучшую ноду или -1 в противном случае

\*/

int next(vector<unordered\_map<int, int>>& g, int v) {

int nxt = -1;

int mn = INT\_MAX;

#if DEBUG == 1

cout << "\n" << "Current node: " << v << "\n";

#endif

// Оценка всех соседей текущего узла

for (auto& [to, w] : g[v]) {

if (\_vis[to]) continue; // Пропуск уже посещенных нод

#if DEBUG == 1

cout << "Considering node: " << to << " (w: " << w << ")\n";

#endif

// Расчет МОД для данной ноды

\_vis[to] = true; // временно заносим в путь

int L = spanningTreeW(g, to);

\_vis[to] = false; // выносим из пути

#if DEBUG == 1

cout << "MST: " << L << " Total: " << (w + L) << "\n";

#endif

// Выбрать соседа с минимальным значением (текущее ребро + оценка МОД)

if (w + L < mn) {

mn = w + L;

nxt = to;

}

}

#if DEBUG == 1

cout << "Selected: " << nxt << "\n";

#endif

return nxt;

}

/\*\*

\* Расчет МОД для оставшегося пути

\* g список смежности графа

\* to Потенциальный следующий узел

\* Возвращает вес МОД для оставшегося пути

\*/

int spanningTreeW(vector<unordered\_map<int, int>>& g, int to) {

int w = 0;

DSU dsu(\_n);

dsu.join(\_path.front(), to); // A - {B - ...} - C - to => A - to

#if DEBUG == 1

cout << "MST Construction \n" << "Current path: ";

for(auto& node : \_path) cout << node << " ";

cout << "\n| Trying node: " << to << "\n";

#endif

// Построение МОД алгоритмом Краскала

for (auto& e : \_edges) {

// если мы рассмотрим вершину из пути (e.from) и она не является концом -> недопустимое ребро

if (\_vis[e.from] && e.from != to) continue;

// если мы рассматриваем вершину из пути (e.to) и если это не начало -> недопустимое ребро

if (\_vis[e.to] && e.to != \_path.front()) continue;

// Добавляем ребро к МОД, если оно соединяет непересекающиеся компоненты

if (dsu.get(e.from) != dsu.get(e.to)) {

dsu.join(e.from, e.to);

w += g[e.from][e.to];

#if DEBUG == 1

cout << "+ Edge: " << e.from << "-" << e.to << " (w: " << e.w << ")\n";

#endif

}

}

#if DEBUG == 1

cout << "Total MST weight: " << w << "\n\n";

#endif

return w; // Возвращает итоговый вес МОД

}

};

int main() {

Solution s;

int n; cin >> n;

int start; cin >> start;

vector<unordered\_map<int, int>> g(n);

for (int i = 0; i < n; ++i) {

for (int j = 0; j < n; ++j) {

int g\_i\_j; cin >> g\_i\_j;

if (g\_i\_j != 0) g[i][j] = g\_i\_j;

}

}

auto [ans, path] = s.TSP(g, start);

if (ans == -1) {

cout << "no path\n";

return 0;

}

cout << ans << "\n";

for (auto& v : path) {cout << v << " ";}

cout << "\n";

}

Название файла: *solution.cpp* (ДП итеративное)

#include <vector>

#include <unordered\_map>

#include <climits>

#include <algorithm>

#define DEBUG 1

#if DEBUG == 1

#include <iostream>

#include <bitset>

#include <iomanip>

#include <sstream>

#endif

using namespace std;

class Solution {

public:

/\*\*

\* Находит минимальную стоимость гамильтонового цикла и путь

\* g список смежности графа

\* start Начальный индекс узла (по умолчанию: 0)

\* возвращает - Пара, содержащая общую стоимость и вектор пути. Возвращает {-1, {}}, если цикла не существует

\*/

pair<int, vector<int>> TSP(vector<unordered\_map<int, int>>& g, int start=0) {

\_n = g.size();

\_start = start;

\_dp.assign(1 << \_n, vector<int> (\_n, INT\_MAX));

\_dp[0][\_start] = 0;

// Основной алгоритм дп

fillDP(g);

// Пути не существует

if (\_dp[(1 << \_n) - 1][\_start] == INT\_MAX) return {-1, {}};

return {\_dp[(1 << \_n) - 1][\_start], restorePath(g)};

}

private:

vector<vector<int>> \_dp;

int \_start;

int \_n;

/\*\*

\* Заполнение таблицы DP

\* g Список смежности графа

\*/

void fillDP(vector<unordered\_map<int, int>>& g) {

// Иткрируемся по всем маскам

for (int mask=0; mask<(1 << \_n); ++mask) {

// Проходим по всем нодам

for (int u=0; u<\_n; ++u) {

if (\_dp[mask][u] == INT\_MAX) continue;

// Обновляем маску с текущей нодой

int new\_mask = mask | (1 << u);

// Исслкдуем все ребра

for (auto& e : g[u]) {

int v = e.first; int w = e.second;

// Разрешаем переход, если:

// - Следующий узел не посещен

// ИЛИ

// - Завершение цикла обратно в начало

if ((new\_mask & (1 << v)) == 0 || v == \_start && new\_mask == ((1 << \_n) - 1)) {

#if DEBUG == 1

bool is\_update = (\_dp[new\_mask][v] > \_dp[mask][u] + w);

printDPUpdateDebug(mask, new\_mask, u, v, w, \_dp[new\_mask][v]);

#endif

\_dp[new\_mask][v] = min(\_dp[new\_mask][v], \_dp[mask][u] + w);

#if DEBUG == 1

printDPResultDebug(new\_mask, v, is\_update, \_dp[new\_mask][v]);

#endif

}

}

}

}

}

vector<int> restorePath(vector<unordered\_map<int, int>>& g) {

vector<int> path = {\_start};

int i = \_start;

int mask = (1 << \_n) - 1;

// Возврат из конечного состояния

while (mask != 0) {

// Находим предшественника

for (int j=0; j<\_n; ++j) {

if (g[j][i] != 0 && (mask & (1 << j)) && (\_dp[mask - (1 << j)][j] + g[j][i] == \_dp[mask][i])) {

#if DEBUG == 1

printPathStep(mask, i, j, g[j][i]);

#endif

path.push\_back(j);

i = j;

mask -= (1 << j);

continue;

}

}

}

// реверс для корректного ответа

reverse(path.begin(), path.end());

#if DEBUG == 1

printFinalPath(path, \_dp[(1 << \_n) - 1][\_start]);

#endif

return path;

}

#if DEBUG == 1

private:

/\*\*

\* Выводит отладочную информацию об обновлении таблицы DP

\* mask Текущая маска

\* new\_mask Новая маска после обновления

\* u Текущий узел

\* v Соседний узел

\* w Вес ребра

\* old\_value Предыдущее значение в DP

\*/

void printDPUpdateDebug(int mask, int new\_mask, int u, int v, int w, int old\_value) {

cout << "\n---------------------------------------Iteration------------------\n"

<< "Mask: " << setw(4) << mask

<< " (" << bitset<16>(mask) << ")"

<< " -> New: " << setw(4) << new\_mask

<< " (" << bitset<16>(new\_mask) << ")\n"

<< "Current node: " << u

<< " -> Neighbor: " << v

<< " (w: " << setw(3) << w << ")\n"

<< "\n"

<< "Current cost: "

<< ((\_dp[mask][u] == INT\_MAX) ? "inf" : to\_string(\_dp[mask][u]))

<< " + " << w << " = " << (\_dp[mask][u] + w) << "\n"

<< "Previous best: "

<< ((old\_value == INT\_MAX) ? "inf" : to\_string(old\_value)) << "\n"

<< "------------------------------------------------------------------\n";

}

/\*\*

\* Выводит отладочную информацию о результате обновления DP

\* new\_mask Обновленная маска

\* v Узел, который был обновлен

\* is\_update Было ли принято обновление

\* new\_value Новое значение в таблице DP

\*/

void printDPResultDebug(int new\_mask, int v, bool is\_update, int new\_value) {

cout << "Result: " << (is\_update ? "ACCEPTED" : "REJECTED")

<< " -> New value: "

<< ((new\_value == INT\_MAX) ? "inf" : to\_string(new\_value))

<< endl;

}

/\*\*

\* Выводит отладочную информацию о шаге реконструкции пути

\* mask Текущая маска

\* from Текущий узел

\* to Следующий узел в пути

\* edge\_weight Вес соединительного ребра

\*/

void printPathStep(int mask, int from, int to, int edge\_weight) {

cout << "---------------------------------- Path Reconstruction -------------------------\n"

<< "Current mask: " << setw(10) << mask << " (" << bitset<16>(mask) << ")\n"

<< "Moving: " << setw(3) << from << " ← " << setw(3) << to << " (w: " << setw(3) << edge\_weight << ")\n"

<< "New mask: " << setw(10) << (mask - (1 << to)) << " (" << bitset<16>(mask - (1 << to)) << ")\n"

<< "-----------------------------------------------------------------------------------\n";

}

/\*\*

\* Выводит окончательный реконструированный путь

\* path Вектор реконструированного пути

\* total\_cost Общая стоимость пути

\*/

void printFinalPath(const vector<int>& path, int total\_cost) {

cout << "Final Path " << "\n"

<< " Total cost: " << setw(58) << total\_cost << "\n"

<< " Path: ";

for (size\_t i = 0; i < path.size(); ++i) {

if (i != 0) cout << " -> ";

cout << path[i];

}

}

#endif

};

int main() {

Solution s;

int n;

cin >> n;

vector<unordered\_map<int, int>> g(n);

for (int i = 0; i < n; ++i) {

for (int j = 0; j < n; ++j) {

int g\_i\_j; cin >> g\_i\_j;

if (g\_i\_j != 0) g[i][j] = g\_i\_j;

}

}

auto [ans, path] = s.TSP(g);

if (ans == -1) {

cout << "no path\n";

return 0;

}

cout << ans << "\n";

for (auto& v : path) {cout << v << " ";}

cout << "\n";

}

# Приложение Б

Результаты тестирования точного метода дп (итеративный) приведены в табл.1.

Таблица 1 – Тестирование точного алгоритма

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
| 1. | 5  0 1 13 23 7  12 0 15 18 28  21 29 0 33 28  23 19 34 0 38  5 40 7 39 0 | 78  0 4 2 3 1 0 | Верно |
| 2. | 3  0 1 0  1 0 1  0 1 0 | no path | Верно |
| 3. | 3  0 27 7  15 0 28  25 21 0 | 43  0 2 1 0 | Верно |
| 4. | 4  0 25 23 11  19 0 37 18  20 31 0 24  15 39 2 0 | 63  0 3 2 1 0 | Верно |

Результаты тестирования приближенного алгоритма АЛШ-2 приведены в табл.2.

Таблица 2 – Тестирование приближенного алгоритма

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
| 1. | 9  3  0 78 62 27 77 15 64 80 95  78 0 65 39 48 68 58 85 88  62 65 0 28 33 43 53 9 98  27 39 28 0 22 6 25 8 46  77 48 33 22 0 82 46 30 80  15 68 43 6 82 0 40 42 58  64 58 53 25 46 40 0 68 60  80 85 9 8 30 42 68 0 14  95 88 98 46 80 58 60 14 0 | 306 | +7.00% |
| 2. | 3  1  0 48 14  48 0 58  14 58 0 | 120 | +0.00% |
| 3. | 7  5  0 0 100 96 85 68 26  0 0 0 0 0 0 0  100 0 0 50 83 30 66  96 0 50 0 96 9 47  85 0 83 96 0 32 90  68 0 30 9 32 0 7  26 0 66 47 90 7 0 | -1 | +0.00% |
| 4. | 9  4  0 57 56 50 3 99 66 77 6  57 0 22 39 78 74 32 84 81  56 22 0 39 52 49 24 23 23  50 39 39 0 66 43 28 93 20  3 78 52 66 0 98 85 44 3  99 74 49 43 98 0 76 2 96  66 32 24 28 85 76 0 5 95  77 84 23 93 44 2 5 0 53  6 81 23 20 3 96 95 53 0 | 189 | +9.00% |