

Poglavlje 5

Implementacija

Budući da je jedan od ciljeva ovog rada provjeriti dio prije opisanih operatora u stvarnosti, implementirano je nekoliko odgovarajućih struktura podataka i algoritama za rad sa strukturama. Strukture podataka će biti opisane tekstom koji prethodi kôdu, dok će funkcije biti opisane komentarima u samom kôdu.

Osnovni objekti u implementaciji su

1. objekt za primjerak problema usmjeravanja vozila (klasa PRV)
2. objekt za jedinke tj. kromosome genetskog algoritma (klasa Chromosome)
3. objekt za populaciju jedinki (klasa Population).

Za implementaciju je korišten programski jezik C++.

5.1 Problem usmjeravanja vozila

Primjeri problema usmjeravanja vozila se vrlo lako mogu naći na internetu, kao što je već navedeno u poglavlju o test primjerima problema (tzv. benchmark instances).

5.1.1 Struktura podataka

Budući da se primjeri problema usmjeravanja vozila najčešće nalaze u običnim tekstualnim datotekama (najčešće u ekstenzijom .vrp) s točno definiranim formatom, potrebno je podatke iz datoteka u određenom redoslijedu smjestiti u odgovarajuće varijable u radnu memoriju računala.

Potsjetimo, primjerak problema usmjeravanja vozila definiran je brojem vozila k , brojem klijenata n , kapacitetom vozila c , potražnjom klijenata d_i , $i = 1, 2, \dots, n$, te matricom cijena puta između klijenata i i j , $C = (c_{ij})$, $i, j = 1, 2, \dots, n$. Struktura podataka koja opisuje primjerak problema usmjeravanja vozila sadrži varijable za gore navedene podatke.

Budući da se u datotekama gotovo uvijek nalaze primjeri problema koji su po tipu Euklidski SCVPR (potsjetimo: klijenti su pridruženi točkama u ravnini), definirana je dodatna struktura podataka koja opisuje točke u dvodimenzionalnoj ravnini (klasa Point).

```
class Point {           // TOCKA U 2D RAVNINI
    public:
        float x;      // x koordinata klijenta
        float y;      // y koordinata klijenta
};
```

Također, u klasu PRV je ubačeno dodatno polje koje čuva položaje svih klijenata u primjerku problema usmjeravanja vozila.

```
class PRV {           //PROBLEM USMJERAVANJA VOZILA
    private:
    public:
        char filename[50];   // ime datoteke s problemom
        char name[50];       // ime problema
        int trucks;          // broj vozila (k)
        int dim;              // broj klijenata + skladiste (n+1)
        int cap;              // kapacitet vozila (c)
        Point* coord;         // položaji klijenata u 2D ravnini (x,y)
        float* demand;        // potražnja klijenata (d_i)
        float** distance;     // matrica cijena (C=c_ij)
        float maxx, maxy;     // najveća x i y iz coord
};
```

Važno je napomenuti da, budući da smo navikli označavati klijente prirodnim brojevima od 1 do n , u poljima coord, demand i distance imamo rezervirano jedno mjesto više nego je potrebno, tako da indeksiranje može ići od 1, kako smo navikli, a ne od 0 kao što je uobičajeno za C/C++. Osim toga, time je ostavljeno mjesto na poziciji 0 koje iskoristimo za skladište, što je također uobičajena oznaka (0). Ovakva situacija nam je vrlo korisna jer skladište ne mora biti u ravnini na lokaciji (0,0), pa ionako moramo negdje zapamtiti njegov položaj.

5.1.2 Osnovne funkcije

Za svaku klasu je važno da ima implementirane neke osnovne funkcije, a to su:

1. konstruktor
2. destruktor
3. copy-konstruktor
4. ostali konstruktori
5. operator pridruživanja (=)
6. operatori uspoređivanja (==, <, ≤, >, ≥, ≠).

U klasi PRV su gore navedene funkcije deklarirane ovako:

```
class PRV {           //PROBLEM USMJERAVANJA VOZILA
public:
    PRV() ;          // konstruktor
    PRV(PRV&) ;     // copy-konstruktor
    ~PRV() ;         // destruktor
```

* Konstruktor rezervira mjesto u memoriji za prihvatanje podataka koje ćemo kasnije učitavati.

```
PRV::PRV() {           // konstruktor
    strcpy(name, "");
    strcpy(filename, "");
    dim = 0;
    cap = 0;
    coord = NULL;
    demand = NULL;
    distance = NULL;
    maxx = 0.0;
    maxy = 0.0;
}
```

Destruktor briše primjerak problema i oslobađa zauzetu radnu memoriju računala.

```
PRV::~PRV() {
    if (coord!=NULL) {
        delete[] coord;
    }
    if (demand!=NULL) {
        delete[] demand;
    }
    if (distance!=NULL) {
        delete[] distance;
    }
    coord = NULL;
    demand = NULL;
    distance = NULL;
}
```

Budući da za testiranje koje želimo provesti nisu nužne ostale navedene funkcije, njihova implementacija je ostala prazna.

```
PRV::PRV(PRV& second) {           // copy-konstruktor
//TODO
}

const PRV& PRV::operator=(PRV&) {   // operator pridruzivanja
//TODO
}

bool PRV::operator==(PRV&) {       // operator usporedjivanja
//TODO
}

bool PRV::operator!=(PRV&) {       // operator usporedjivanja
//TODO
}
```

5.1.3 Ostale funkcije

Deklaracije svih ostalih funkcija dane su s

```
bool LoadDotVRP(char* file);
void PrintDotNB(char* );
void PrintDotNB();
bool CalculateDistances();
```

```

bool PrintDistances();
bool CheckDemands();
friend ostream& operator<<(ostream&, PRV&);

```

Budući da želimo učitavati podatke iz .vrp datoteke, implementirana je funkcija LoadDotVRP(). Osim nje, imamo funkciju koja provjerava da li su potrebe svih klijenata u skladu s kapacitetom vozila (CheckDemands()).

```

bool PRV::LoadDotVRP(char* file) {
    char buf[1024];
    char c;
    int br = 0;           // brojac linija u .vrp datoteci
    float trash;          // varijabla za nepotrebne podatke
    FILE* f;
    f = fopen(file, "r"); // otvori .vrp datoteku za citanje
    if (f == NULL) exit(0); // ako nisam uspio otvoriti datoteku
    strcpy(filename, file); // spremi ime datoteke
    while (!feof(f)) {    // dok nisam sve procitao, radi
        if (br==0)         // ucitaj ime problema
            fscanf(f, "%s %s %s", buf, buf, name);
        fscanf (f, "%c", &c); // citaj znak po znak
        if (c == '\n') br++; // kad naletis na "enter", (br + 1)
        if (br == 3) {      // u trecoj liniji pise broj klijenata
            fscanf(f, "%s %s %d", buf, buf, &dim); // ucitaj dim
            coord = new Point[dim + 1]; // rezerviraj memoriju
            demand = new float[dim + 1]; // za položaj,
            distance = new float*[dim + 1]; // d_i, c_ij
            for (int i = 0; i < dim + 1; i++)
                distance[i] = new float[dim + 1];
        }
        if (br==5) {        // u petoj liniji pise kapacitet vozila
            fscanf(f, "%s %s %d", buf, buf, &cap);
        }
        if (br==7) {        // od sedme linije nadalje, zapisane su
                           // koordinate položaja klijenata
            for (int i = 0; i < dim; i++) {
                fscanf(f, "%f %f %f", &trash, &coord[i].x,
                        &coord[i].y);
                br++;
            }
        }
        fflush(stdout);
    }
}

```

```

        fscanf(f, "%s", buf); // baci jednu liniju u smece
        for (int i = 0; i < dim; i++) {
            // u ostatku datoteke su zapisane potraznje klijenata
            fscanf(f, "%f %f", &trash, &demand[i]);
            br++;
        }
    }
    if (strcmp(name, "")!=0) {
        int i = strlen(name);
        while(name[i]!='k') i--;
        i++;
        int j = 0;
        while(i <= strlen(name)) buf[j++] = name[i++];
        trucks = atoi(buf);
    } // broj vozila se nalazi u imenu problema
    else
        trucks = 8;
    fclose(f); // zatvori datoteku
    return true;
}

bool PRV::CheckDemands() {
    for (int i = 0; i < dim; i++) { // skladiste je u vrhu 0
        // ako postoji klijent cija je potraznja veca od
        // kapaciteta vozila, prekini izvrsavanje programa
        if (demand[i] > cap) {
            cout << "Error: demand > capacity !\n"; cout.flush();
            exit(0);
            return false;
        }
    }
    return true;
}

```

Budući da su u .vrp datoteci zapisani samo položaji klijenata u 2D ravnini, potrebno je iz tih podataka izračunati međusobne udaljenosti za svaki par klijenata (matrica $C = (c_{ij})$, odnosno dvodimenzionalno polje $distance$). Taj posao obavlja funkcija `CalculateDistances()`.

```

bool PRV::CalculateDistances() {
    // za svaka dva klijenta i,j

```

```

// izracunaj euklidsku udaljenost (i,j)
for (int i = 0; i < dim; i++) {
    for (int j = 0; j < dim; j++) {
        if (i == j)
            distance[i][j] = 0.0;
        else
            distance[i][j] = sqrt (
                (coord[i].x-coord[j].x)*(coord[i].x-coord[j].x)
                +(coord[i].y-coord[j].y)*(coord[i].y-coord[j].y));
    }
}
return true;
}

```

Da bismo mogli grafički prikazati problem koji smo učitali, dana je funkcija `PrintDotNB()`, koja stvara tekstualnu datoteku s naredbom koju treba proslijediti programskom paketu "Mathematica". Nakon izvršavanja naredbe u paketu "Mathematica", dobijemo .eps datoteku sa slikom problema. Tako su stvorene i slike test problema A-n62-k8 i E-n101-k8.

```

void PRV::PrintDotNB() {
    char newname[50];
    strcpy(newname, name);
    strcat(newname, "-NB.txt");
    PrintDotNB(newname);
}

void PRV::PrintDotNB(char* outname) {
    int i;
    FILE* f;
    f = fopen(outname, "w");
    if (f == NULL) {
        cout << "Ne mogu otvoriti " << outname << "!\n";
        cout.flush();
        exit(0);
    }
    fprintf (f, "depot = Graphics[ListPlot[{{%.0f, %.0f}},",
    PlotStyle -> PointSize[0.015], AxesLabel ->
    TraditionalForm /@ {x, y}, TextStyle -> {FontFamily ->
    \"Times\", FontSize -> 14}, ImageSize -> 600,
    PlotLabel -> StyleForm[TraditionalForm[\"%s\"], FontFamily -> \"Times\",
    FontSlant -> \"Italic\",

```

```

FontSize -> 14]]];\n", coord[1].x, coord[1].y, name);

fprintf (f, "size = Graphics[ListPlot[{{%.0f, 0}, {0, %.0f}}, 
    PlotStyle -> PointSize[0.0], AxesLabel ->
    TraditionalForm /@ {x, y}, TextStyle -> {FontFamily ->
    \"Times\", FontSize -> 14}, ImageSize -> 600]];\n",
    maxx + 3, maxy + 3);

fprintf (f, "clients = Graphics[ListPlot[{{%.0f, %.0f}},
    coord[2].x, coord[2].y];
for (i = 3; i <= dim; i++)
    fprintf (f, ", {%.0f, %.0f}", coord[i].x, coord[i].y);
fprintf (f, "}, PlotStyle -> PointSize[0.008],
    AxesLabel -> TraditionalForm /@ {x, y},
    TextStyle -> {FontFamily -> \"Times\", FontSize -> 14},
    ImageSize -> 600]];\n");

fprintf (f, "pic = Show[depot, Graphics[{Text[\"1\",
    {%.0f, %.0f}, {0, -1.5},
    TextStyle -> {FontFamily -> \"Times\", FontSize -> 8},
    Background -> None}], coord[1].x, coord[1].y];
for (i = 2; i <= dim; i++)
    fprintf (f, ", Text[\"%d\", {%.0f, %.0f}, {0, -1.5},
    TextStyle -> {FontFamily -> \"Times\", FontSize -> 8},
    Background -> None]", i, coord[i].x, coord[i].y);
fprintf (f, "}], clients, size];\n");

fprintf (f, "Export[\"%s.eps\", pic, ImageSize -> 500];", name);

fclose(f);
}

```

Na samom kraju su nam ostale dvije funkcije koje samo ispisuju: funkcija `PrintDistances()` ispisuje matricu cijena $C = (c_{ij})$, a operator `<<` je redefiniran tako da "zna" ispisivati objekte VRP tipa.

```

bool PRV::PrintDistances() {
    for (int i = 0; i < dim; i++) {
        for (int j = 0; j < dim; j++) {
            cout << distance[i][j] << " ";
        }
    }
}

```

```

        cout << "\n";
    }
    cout.flush();
    return true;
}

ostream& operator<<(ostream& os, PRV& p) {
    os << "dim=" << p.dim << "\n";
    for (int i=1; i<p.dim + 1; i++) {
        os << i << ": (" << p.coord[i].x
            << ", " << p.coord[i].y
            << ")\\tdemand=" << p.demand[i] << "\n";
    }
    return os;
}

```

5.2 Jedinke evolucijskog algoritma

Kao i u ostalim klasama, jedinke su prikazane određenom strukturon podataka, na kojoj su definirane operacije potrebne za evoluciju.

5.2.1 Struktura podataka

U prethodnim poglavljima navedeno je nekoliko mogućnosti za prikaz jedinki evolucijskog algoritma (binarni prikaz, permutacija, vektor susjedstva, redoslijed, varijacija, itd.). U ovoj implementaciji evolucijskog algoritma izabran je najjednostavniji oblik: prikaz pomoću permutacije skupa klijenata.

Permutaciju $p = (p_1, p_2, \dots, p_n)$ duljine n prikazujemo poljem \mathbf{x} odgovarajuće duljine. Klasa također čuva duljinu polja \mathbf{x} (odnosno permutacije p) u varijabli `len`. Budući da se često mora računati vrijednost funkcije dobrote na permutaciji, što je prilično "skupa" operacija, u klasu je ubaćena varijabla `fit` koja čuva dobrotu jedinke. Sve funkcije funkcioniraju tako da, čim se promijeni jedinka, odmah se ažurira varjabla `fit` s novom dobrotom jedinke. Deklaracija klase `Permutation` izgleda ovako:

```

class Permutation { // JEDINKA EVOLUCIJSKOG ALGORITMA
private:
    int len;          // duljina kromosoma
    float fit;        // dobrota kromosoma
    int* x;           // polje gena
};

```

... osnov-
usmjeravanja vozila. Deklaracije su sljedeće:

```
class Permutation { // JEDINKA EVOLUCIJSKOG ALGORITMA
public:
    Permutation() ; // konstruktor
    Permutation(int, PRV&) ; // ostali konstruktori
    Permutation(Permutation&) ; // copy-konstruktor
    ~Permutation() ; // destruktor
    const Permutation& operator=(Permutation&) ;
        // operator pridruživanja
    bool operator==(Permutation&) ; // operator usporedjivanja
    bool operator!=(Permutation&) ; // operator usporedjivanja
    int& operator[](int); // operator koji vraca gen na poziciji i
};
```

Implementacije gore deklariranih funkcija su sljedeće:

```
Permutation::Permutation() { // konstruktor
    len = 0;
    x = NULL;
    fit = 0.0;
};
```

```
Permutation::Permutation(int n, PRV& pr) { // ostali konstruktori
    // stvara permutaciju duljine n i racuna dobrotu
    if (n < 1) exit(0);
    len = n;
    x = new int[len + 1];
    for (int i = 0; i < len + 1; i++)
        x[i] = i;
    setFitness(pr);
}
```

```
Permutation::Permutation(Permutation& pc) { // copy-konstruktor
    if ((pc.x!=NULL) && (pc.len>0)) { // ako imam sto kopirati
        len = pc.len; // iskopiraj duljinu
        x = new int[len + 1]; // rezerviraj dovoljno memorije
        for (int i = 0; i < len + 1; i++)
            x[i] = pc.x[i]; // iskopiraj gen po gen
```

```

        fit = pc.fit;           // iskopiraj dobrotu
    }
    else { // ako nemam sto kopirati
        x = NULL;
        len = 0;
        fit = 0.0;
    }
}

Permutation::~Permutation() { // destruktor
    if (x!=NULL) { // ako imam sto obisati
        delete[] x; // obrisi sve gene
    }
    x = NULL; // vazno
}

const Permutation& Permutation::operator=(Permutation& second) {
    // operator pridruzivanja
    if (len != second.len) {
        cout << "Greska: Permutation operator= !\n";
        cout.flush(); exit(0);
        // pokusavam kopirati permutacije razlicitih duljina
    }
    len = second.len; // iskopiraj duljinu
    fit = second.fit; // iskopiraj dobrotu
    for(int i = 1; i < len + 1; i++) {
        x[i] = (second.x)[i]; // iskopiraj gen po gen
    }
    return (*this);
}

```

Jednakost dviju permutacija p_1 i p_2 se definira tako da duljina (n odnosno len) od p_1 mora biti jednaka duljini od p_2 , te mora vrijediti

$$p_1(i) = p_2(i), \text{ za svaki } i = 1, 2, \dots, n.$$

```

bool Permutation::operator==(Permutation& second) {
    // operator usporedjivanja
    if (len != second.len)
        return false; // ako su duljine razlicite
    else
        for(int i=1; i<len + 1; i++)

```

```

        if (x[i] != second.x[i]) return false;
        // ako postoji bilo koja pozicija s razlicitim genom
    return true;
}

bool Permutation::operator!=(Permutation& second) {
    // operator usporedjivanja; negacija od prethodnog op==
    return !(*this == second);
}

int& Permutation::operator[](int i) {
    // operator[] vraca gen sa pozicije i
    return x[i];
}

```

5.2.3 Ostale funkcije

Da bismo mogli s permutacijama obavljati križanje, mutaciju, lokalno pretraživanje, računati dobrotu, raspoređivati klijente po vozilima, ispisivati, itd. potrebne su nam i sljedeće funkcije:

```

class Permutation { // JEDINKA EVOLUCIJSKOG ALGORITMA
public:
    int Leng();
    float Cost(PRV&);
    float Fitness();
    bool Feasible(PRV&);
    bool Switch(Permutation&);
    int RouteStart(PRV&, int);
    int RouteEnd(PRV&, int);
    bool PrintRoutes(PRV&);
    void LoadFromFile(char*);
    void PrintDotNB(char*, PRV&);
    void PrintDotNB(PRV&);

    float TwoOptGlobal(PRV&);
    float TwoOptLocal(PRV&);
    float TwoOptLocal(PRV&, int, int);
    float RemoveAndReinsertLocalAll(PRV&);
    float RemoveAndReinsertLocalAll(PRV&, int, int);
    float RemoveAndReinsertGlobalAll(PRV& pr);
    Permutation OrderCrossover(Permutation&, PRV&);

```

```
    Permutation OrderCrossoverAll(Permutation&, PRV&);  
    Permutation OrderCrossover(Permutation&, PRV&, int, int);  
  
    void Random() ;  
    void Random(int) ;  
    inline void setFitness(PRV&) ;  
    inline void setFitness(float) ;  
    double getFitness() ;  
};
```

Dalje slijedi implementacija i objašnjenje svrhe gore navedenih funkcija.

```

int Permutation::Leng(void) {
    // jednostavna funkcija koji vraca duljinu jedinke
    return len;
}

float Permutation::Cost(PRV& pr) {
    // racuna cijenu tj. dobrotu jedinke
    Permutation ptmp = *this;
    float spaceleft = pr.cap;
    float cost = 0.0;
    int brK = 1;
    for (int i = 1; i < pr.dim; i++) {
        spaceleft = spaceleft - pr.demand[ x[i] ];
        if (spaceleft < 0) {
            i--;
            brK++;
            cost = cost + pr.distance[ptmp[i]][ptmp[0]];
            ptmp[i] = 0;
            spaceleft = pr.cap;
        }
        else
        {
            cost = cost + pr.distance[ptmp[i]][ptmp[i-1]];
        }
    }
    cost = cost + pr.distance[ptmp[0]][ptmp[i-1]];
    if (brK > pr.trucks) return 1E10; // nedopustiva permutacija
    return cost;
}

```

```

float Permutation::Fitness() {
    // vraca dobrotu jedinke
    return fit;
}

void Permutation::setFitness(PRV& pr) {
    // ponovno izracuna i postavlja dobrotu jedinke
    fit = Cost(pr);
}

void Permutation::setFitness(float f) {
    // postavlja dobrotu jedinke na vrijednost f
    fit = f;
}

bool Permutation::Feasible(PRV& pr) {
    // provjerava da li je rjesenje dopustivo
    int brK = 1;
    float spaceleft = pr.cap;
    for (int i = 2; i <= pr.dim; i++) {
        spaceleft = spaceleft - pr.demand[ x[i] ];
        if (spaceleft < 0) {
            i--;
            spaceleft = pr.cap;
            brK++;
        }
    }
    if (brK > pr.trucks) return false;
    return true;
}

bool Permutation::Switch(Permutation& with) {
    // zamijeni dvije jedinke (za lakse sortiranje
    float ftmp = fit;
    fit = with.fit;
    with.fit = ftmp;
    int itmp = len;
    len = with.len;
    with.len = itmp;
    int* ptmp = x;
    x = with.x;
}

```

```

        with.x = ptmp;
        return true;
    }

int Permutation::RouteStart(PRV& pr, int k) {
    // trazi poziciju pocetka rute za k-to vozilo u problemu pr
    if (k<1) {
        cout << "Greska: Nepostojece vozilo u VehicleNo!\n";
        exit(0);
    }
    if (k > pr.trucks) return (pr.dim + 1);
    int brK = 1;
    float spaceleft = pr.cap;
    int beg = 2;
    for (int i = 2; i <= pr.dim; i++) {
        spaceleft = spaceleft - pr.demand[ x[i] ];
        if (spaceleft < 0) {
            i--;
            spaceleft = pr.cap;
            brK++;
        }
        if (k == brK) break;
        beg = i + 1;
    }
    return beg;
}

int Permutation::RouteEnd(PRV& pr, int k) {
    // trazi poziciju kraja rute za k-to vozilo u problemu pr
    return (RouteStart(pr, k + 1) - 1 );
}

float Permutation::TwoOptGlobal(PRV& pr) {
    // 2-opt lokalno pretrazivanje cijele permutacije
    return TwoOptLocal(pr, 1, pr.dim);
}

float Permutation::TwoOptLocal(PRV& pr) {
    // "lokalno" 2-opt lokalno pretrazivanje:
    // pretrazujemo put jednog po jednog vozila
    float spaceleft = pr.cap;
}

```

```

float mincost = 1E10;
float cost = 1E10;
Permutation best = (*this);
int k = 2, l = 0;
spaceleft = pr.cap;
for (int i = k; i <= pr.dim; i++) {
    spaceleft = spaceleft - pr.demand[ x[i] ];
    if (spaceleft < 0) {
        i--;
        l = i;
        spaceleft = pr.cap;
        cost = TwoOptLocal(pr, k, l);
        if (cost < mincost)
        {
            mincost = cost;
            setFitness(cost);
        }
        k = i + 1;
    }
}
if(spaceleft >= pr.cap) {
    cost = TwoOptLocal(pr, k, pr.dim);
    if (cost < mincost)
    {
        mincost = cost;
        setFitness(cost);
    }
}
return cost;
}

float Permutation::TwoOptLocal(PRV& pr, int k, int l) {
    // obican 2-opt algoritam, samo djeluje od pozicija k do l
    if ( (k>pr.dim) || (l>pr.dim) || (k<1) || (l<1) )
    {
        cout << "Greska: Nedopustivi indeksi u TwoOptLocal!\n";
        exit(0);
    }
    float mincost = 1E10;
    float cost = 1E10;
    Permutation best = (*this);
}

```

```

Permutation p = (*this);
int xtmp;
for (int i = k; i < l - 1; i++)
{
    for (int j = k + 2; j < l + 1; j++)
    {
        p = (*this);
        for (int t = i + 1, int u = j; t <= u; t++, u--) {
            xtmp = p.x[t];
            p.x[t] = p.x[u];
            p.x[u] = xtmp;
        }
        cost = p.Cost(pr);
        if (cost < mincost)
        {
            mincost = cost;
            best = p;
        }
    } // for
} // for
(*this) = best;
setFitness(pr);
return cost;
}

bool Permutation::PrintRoutes(PRV& pr) {
    // ispisuje rjesenje tako da odvoji puteve vozila znakom ":".
    int brK = 1;
    float spaceleft = pr.cap;
    for (int i = 1; i < pr.dim; i++) {
        spaceleft = spaceleft - pr.demand[ x[i] ];
        if (spaceleft < 0) {
            i--;
            spaceleft = pr.cap;
            brK++;
            cout << " : ";
        }
        else
            cout << x[i] << ' ';
    }
    return true;
}

```

Najbolja rješenja, tj. permutacije, spremamo u datoteku. Funkcija `LoadFromFile` radi obrnut posao: iz datoteke učitava permutaciju.

```
void Permutation::LoadFromFile(char* inname) {
    int i, d;
    FILE* f;
    f = fopen(inname, "r");
    if (f == NULL) exit(0);
    x[0] = 0;
    x[1] = 1;
    for (i = 0; ( (i < len) || (!feof(f)) ); i++)
        fscanf (f, "%d", &x[i]);
    if (i < len) {
        cout << "Prekratka datoteka u Permutation::LoadFromFile!\n";
        cout.flush(); exit(0);
    }
    fclose(f);
}
```

Slično kao i kod klase PRV, implementirana je funkcija koja stvara tekstualnu datoteku s naredbom koja nakon izvršavanja u paketu "Mathematica", daje .eps datoteku s grafičkim prikazom permutacije. Tako su dobijene sve slike rješenja u ovom radu.

```
, void Permutation::PrintDotNB(PRV& pr) {
    char newname[50];
    strcpy(newname, pr.name);
    strcat(newname, "-RJ.txt");
    PrintDotNB(newname, pr);
}

void Permutation::PrintDotNB(char* outname, PRV& pr) {
    int i, d;
    FILE* f;
    f = fopen(outname, "w");
    if (f == NULL) {
        cout << "Ne mogu otvoriti " << outname << "!\n";
        cout.flush(); exit(0);
    }
    setFitness(pr);

    fprintf (f, "depot = Graphics[ListPlot[{%.0f, %.0f}],
```

```

PlotStyle -> PointSize[0.015], AxesLabel ->
TraditionalForm /@ {x, y},
TextStyle -> {FontFamily -> "Times\", FontSize -> 14},
ImageSize -> 600, PlotLabel -> StyleForm[TraditionalForm
["%s (cijena: %.0f)"], FontFamily -> "Times\",
FontSlant -> "Italic\", FontSize -> 14]]];\n",
pr.coord[1].x, pr.coord[1].y, pr.name, fit);

fprintf (f, "size = Graphics[ListPlot[{{%.0f, 0}, {0, %.0f}},",
PlotStyle -> PointSize[0.0], AxesLabel ->
TraditionalForm /@ {x, y},
TextStyle -> {FontFamily -> "Times\", FontSize -> 14},
ImageSize -> 600]]; \n", pr.maxx + 3, pr.maxy + 3);

fprintf (f, "clients = Graphics[ListPlot[{{%.0f, %.0f}",
pr.coord[x[2]].x, pr.coord[x[2]].y);
for (i = 3; i <= len; i++)
fprintf (f, ", {%.0f, %.0f}", pr.coord[x[i]].x,
pr.coord[x[i]].y);
fprintf (f, "}, PlotStyle -> PointSize[0.008],
AxesLabel -> TraditionalForm /@ {x, y},
TextStyle -> {FontFamily -> "Times\", FontSize -> 14},
ImageSize -> 600]]; \n");

fprintf (f, "paths = Graphics[ListPlot[{{%.0f, %.0f}",
pr.coord[x[1]].x, pr.coord[x[1]].y);
int brK = 1;
float spaceleft = pr.cap;
for (int i = 2; i <= pr.dim; i++) {
spaceleft = spaceleft - pr.demand[ x[i] ];
if (spaceleft < 0) {
i--;
spaceleft = pr.cap;
brK++;
fprintf (f, ", {%.0f, %.0f}",
pr.coord[x[1]].x, pr.coord[x[1]].y);
}
else
fprintf (f, ", {%.0f, %.0f}",
pr.coord[x[i]].x, pr.coord[x[i]].y);
}
}

```

```

fprintf (f, ", {%.0f, %.0f}", pr.coord[x[1]].x, pr.coord[x[1]].y);
fprintf (f, "}, PlotStyle ->PointSize[0.008],
AxesLabel -> TraditionalForm /@ {x, y},
TextStyle -> {FontFamily -> \"Times\", FontSize -> 14},
ImageSize -> 600, PlotJoined->True]];\n");

fprintf (f, "pic = Show[depot, Graphics[{Text[\"1\",
{%.0f, %.0f}, {0, -1.5},
TextStyle -> {FontFamily -> \"Times\", FontSize -> 8},
Background -> None}], pr.coord[1].x, pr.coord[1].y];
for (i = 2; i <= len; i++)
fprintf (f, "Text[\"%d\", {%.0f, %.0f}, {0, -1.5},
TextStyle -> {FontFamily -> \"Times\", FontSize -> 8},
Background -> None]", i, pr.coord[i].x, pr.coord[i].y);
fprintf (f, "}], clients, size, paths];\n");

fprintf (f, "Export[\"%s-RJ.eps\", pic, ImageSize -> 500];",
pr.name);

fclose(f);
}

float Permutation::RemoveAndReinsertGlobalAll(PRV& pr) {
// mutacija ubacivanjem koja pretrazuje cijelu okolinu
return RemoveAndReinsertLocalAll(pr, 2, pr.dim);
}

float Permutation::RemoveAndReinsertLocalAll(PRV& pr,
int beg, int end) {
// mutacija ubacivanjem koja pretrazuje cijelu okolinu
// unutar puta jednog vozila
float mincost = 1E10;
float cost = 0.0;
int begl;
int endl;
if ( (beg > end) || (beg > pr.dim) || (end > pr.dim) ||
(beg < 1) || (end < 1) )
{
cout << "Greska: Nedopustivi indeksi u RARLocAll!\n";
cout.flush(); exit(0);
}
}

```

```

if (end - beg < 1) {
    setFitness(pr);
    return fit;
}
Permutation p = (*this);
Permutation best = (*this);
for (int k = beg; k <= end ; k++) {
    for (int l = beg; l <= end ; l++) {
        p = (*this);

        if (k < l) {
            begl = k;
            endl = l;
            int tmp = p.x[k];
            for (int i = k; i < l; i++) {
                p.x[i] = p.x[i + 1];
            }
            p.x[l] = tmp;
        }
        else if (k > l) {
            begl = l;
            endl = k;
            int tmp = p.x[k];
            for (int i = k; i > l; i--) {
                p.x[i] = p.x[i - 1];
            }
            p.x[l] = tmp;
        }
        cost = p.Cost(pr);
        if (cost < mincost) {
            mincost = cost;
            best = p;
            best.setFitness(cost);
        }
    }
    (*this) = best;
    return mincost;
}

```

```

float Permutation::RemoveAndReinsertLocalAll(PRV& pr) {
    // mutacija ubacivanjem koja pretrazuje cijelu okolinu
    // unutar puta svakog vozila
    float mincost = 1E10;
    float cost = 0.0;
    Permutation best = (*this);

    for (int v = 1; v <= pr.trucks; v++) {
        // int k = rand() % (pr.trucks) + 1;
        int beg = RouteStart(pr, v);
        int end = RouteEnd(pr, v);

        cost = RemoveAndReinsertLocalAll(pr, beg, end);

        if (cost < mincost) {
            mincost = cost;
            best = (*this);
        }
    }
    (*this) = best;
    setFitness(pr);
    return mincost;
}

Permutation Permutation::OrderCrossover(Permutation& mom, PRV& pr)
{
    // OX krizanje
    Permutation& dad = (*this);
    Permutation child = (*this);
    int k = 0, l = 0;
    k = rand() % len + 1;
    do
        l = rand() % len + 1;
    while (k == l);
    if (k > l) {
        int tmp = k;
        k = l;
        l = tmp;
    }

    int j = l + 1;
}

```

```

if (j > child.len) j = 2;
int n = j;
int no = child.len - (l - k + 2);
int i = 1;
while (i <= no) {
    bool ok = true;
    for (int m = k; m <= l; m++) {
        if (mom[j] == child[m]) {
            ok = false;
            break;
        }
    }
    if (ok == true) {
        i++;
        child[n] = mom[j];
        n++;
        if (n > child.len) n = 2;
    }
    j++;
    if (j > child.len) j = 2;
}
child.setFitness(pr);
return child;
}

Permutation Permutation::OrderCrossoverAll(
    Permutation& mom, PRV& pr) {
// radi sva moguca krizanja mame i tate
float mincost = 1E10;
float cost = 0.0;
Permutation& dad = (*this);
Permutation child = (*this);
Permutation best = child;

for (int k = 2; k < pr.dim; k++)
    for (int l = k + 1; l <= pr.dim; l++) {
        child = dad.OrderCrossover(mom, pr, k, l);
        cost = child.Cost(pr);
        if (cost < mincost) {
            mincost = cost;
            best = child;
        }
    }
}

```

```

        best.setFitness(cost);
    }
}
return best;
}

Permutation Permutation::OrderCrossover(
    Permutation& mom, PRV& pr, int k, int l) {
// krizanje s tockama prekida k i l
Permutation& dad = (*this);
Permutation child = (*this);
int j = l;
if (j > child.len) j = 2;
int n = j;
int no = child.len - (l - k + 2);
int i = 1;
while ( i <= no) {
    bool ok = true;
    for (int m = k; m <= l; m++) {
        if ( mom[j] == child[m] ) {
            ok = false;
            break;
        }
    }
    if (ok == true) {
        i++;
        child[n] = mom[j];
        n++;
        if (n > child.len) n = 2;
    }
    j++;
    if (j > child.len) j = 2;
}
child.setFitness(pr);
return child;
}

void Permutation::Random(void) {
// stvara slucajnu permutaciju mijesanjem postojece
if ((x!=NULL)&&(len>0)) {
    int k = 0, l = 0;
}
}

```

```

int temp;
for (int i = 0; i < (len + 1) / 2; i++) {
    k = rand() % len + 1;
    l = rand() % len + 1;
    temp = x[k];
    x[k] = x[l];
    x[l] = temp;
}
}

void Permutation::Random(int n) {
// stvara slucajnu permutaciju duljine n mjesanjem postojece
if (n < 1) exit(0);
len = n;
x = new int[len + 1];
for (int i = 0; i < len + 1; i++)
    x[i] = i;

if ((x!=NULL)&&(len>0)) {
    int k = 0, l = 0;
    int temp;
    for (int i = 0; i < (len + 1) / 2; i++) {
        k = rand() % len + 1;
        l = rand() % len + 1;
        temp = x[k];
        x[k] = x[l];
        x[l] = temp;
    }
}
}

ostream& operator<<(ostream& os, Permutation& p) {
// ispisivanje permutacije
for (int i=1; i<p.len + 1; i++) {
    os << p.x[i] << " ";
}
return os;
}

```

5.3 Populacija jedinki

5.3.1 Struktura podataka

Svaki evolucijski algoritam barata s populacijom jedinki. Populacija je predstavljena poljem jedinki `chroms`. Duljina polja odnosno veličina populacije je `size`, a `chrLen` je duljina, odnosno broj gena, svih jedinki u populaciji. Jedinke su u populaciji uvijek sortirane tako da je najbolja jedinka na vrhu (pozicija 0). Ugrađen je elitizam (najbolja jedinka ne može odumrijeti) i zaštita od duplikata (u populaciju ne možemo ubaciti jedinku koja se već nalazi u populaciji). Deklaracija klase `Population` izgleda ovako:

```
class Population {           // POPULACIJA
    private:
        int size;           // velicina populacije
        Permutation* chroms; // populacija
        int chrLen;         // duljina kromosoma u populaciji
};
```

5.3.2 Osnovne funkcije

Kao i u prethodna dva slučaja (klase `PRV` i `Chromosome`), imamo sljedeće deklaracije osnovnih funkcija.

```
class Population {           // POPULACIJA
public:
    Population() ;          // konstruktor
    Population(int, int, PRV&) ; // ostali konstruktori
    Population(Permutation&) ; // ostali konstruktori
    Population(Population&) ; // copy-konstruktor
    ~Population() ;          // destruktur

    const Population& operator=(Population&) ;
        // operator pridruživanja
    bool operator==(Population&) ; // operator usporedjivanja
    bool operator!=(Population&) ; // operator usporedjivanja
    Permutation& operator[](int); // vraca i-tu jedinku
}
```

```

Population::Population() { // konstruktor
    size = 0;
    chroms = NULL;
    chrLen = 0;
}

Population::Population(int s, int c, PRV& pr) {
    // stvara slucajnu populaciju velicine s,
    // s jedinkama duljine c i postavlja dobrotu
    size = s;
    chrLen = c;
    chroms = new Permutation[size];
    for(int i=0; i<size; i++) {
        chroms[i].Random(chrLen);
        chroms[i].setFitness(pr);
    }
    SortPop(); // sortira populaciju
}

Population::Population(Permutation& c) {
    // stvara populaciju koja sadrzi 1 jedinku
    if (c.Leng()==0) {
        chroms = NULL;
        size = 0;
        chrLen = 0;
    }
    else {
        size = 1;
        chrLen = c.Leng();
        chroms = new Permutation[size];
        chroms[0] = c;
    }
}

Population::Population(Population& src) {
    // copy-konstruktor
    size = src.size;    // iskopiraj broj jedinki
    chrLen = src.chrLen; // iskopiraj duljinu jedinki
    chroms = new Permutation[size]; // rezerviraj memoriju
    for(int i=0; i<size; i++) {
}

```

```

chroms[i] = src.chroms[i];
// iskopiraj populaciju jedinku po jedinku
}
}

Population::~Population() {
    // obrije sve jedinke
    if (chroms!=NULL) delete[] chroms;
    chroms = NULL;
}

const Population& Population::operator=(Population& second) {
    // operator pridruzivanja
    if (chroms != NULL) delete[size] chroms;
    size = second.size;
    chrLen = second.chrLen;
    chroms = new Permutation[size];
    for(int i = 0; i < size; i++) {
        chroms[i].Random(chrLen);
        chroms[i] = (second.chroms)[i];
    }
    return (*this);
}

```

• Dvije populacije su po definiciji jednake ako imaju jednak broj jedinki, te ako se svaka jedinka jedne populacije nalazi negdje u drugoj populaciji.

```

bool Population::operator==(Population& second) {
    if (chrLen != second.chrLen) // || (size != second.size)
        return false;
    else {
        for(int i=0; i<size; i++)
            if (!have(second.chroms[i])) return false;
        for(i=0; i<size; i++)
            if (!second.have(chroms[i])) return false;
    } //else
    return true;
}

bool Population::operator!=(Population& second) {
    return (!( (*this) == second ));
}

```

```
Permutation& Population::operator[](int i) {
    // vraca jedinku s pozicije i
    return chroms[i];
}
```

5.3.3 Ostale funkcije

Populaciju treba znati ispisati, provjeriti da li je određena jedinka možda već u populaciji, obavljati selekciju, dohvatiti i -tu jedinku u populaciji, usporediti da li su dvije populacije jednake, treba moći sortirati populaciju po dobroti jedinki, generirati slučajnu populaciju, itd. Zato su deklarirane sljedeće funkcije

```
void Population::SortPop() {
    // sortiranje populacije upotrebom ranije opisanog switch-a
    if ((size !=0) && (chroms!=NULL)) {
        for (int i=0; i<size; i++)
            for (int j=i+1; j<size; j++)
                if (chroms[j].Fitness() < chroms[i].Fitness()) {
                    chroms[j].Switch(chroms[i]);
                }
    }
}

int Population::popSize() {
    // vraca broj jedinki u populaciji
    return size;
}

Permutation& Population::selectBest(float p) {
    // turnirska selekcija najbolje od p% jedinki
    if ((chroms!=NULL)&&(size>0)) {
        int pos = rand() % size;
        //      (int)( ((float)size-1) * rand() ) / RAND_MAX );
        if ((p>=1)|| (p<=0)) {
            cout << "Greska: Neispravan p u selectBest!\n";
            cout.flush();
            exit(0);
        }
        else {
            int n = (int) ((float)size*p);
            if ((n>=size)|| (n<=0)) {
```

POGLAVLJE 5. IMPLEMENTACIJA

```

cout << "Greska: Neispravan n u selectBest!\n";
cout.flush();
exit(0);
}
else {
    for (int i=1; i<n; i++) {
        int temp = rand() % size;
        if (temp<pos) pos=temp;
    } //for
    return chroms[pos];
} //else
} //else
}
else {
    cout << "Greska: Neispravna populacija u selectBest!\n";
    cout.flush();
    exit(0);
}
return chroms[0];//NULL;
}

Permutation& Population::selectWorst(float p) {
    // turnirska selekcija najgore od p% jedinki
    if ((chroms!=NULL)&&(size>0)) {
        int pos = rand() % size;
        if (pos==0) pos++; //ne zelim unistiti najbolju jedinku
        if ((p>=1)|| (p<=0)) {
            cout << "Greska: Neispravan p u selectWorst!\n";
            cout.flush();
            exit(0);
        }
    else {
        int n = (int) ((float)size*p);
        if ((n>=size)|| (n<=0)) {
            cout << "Greska: Neispravan n u selectBest!\n";
            cout.flush();
            exit(0);
        }
    else {
        for (int i=1; i<n; i++) {
            int temp = rand() % size;

```

```

        if (temp>pos) pos=temp;
    } //for
//    if (pos==0) pos++;
//ne zelim unistiti najbolju jedinku
    return chroms[pos];
} //else
} //else
}
else {
    cout << "Greska: Neispravna populacija u selectWorst!\n";
    cout.flush();
    exit(0);
}
return chroms[0];//NULL;
}

bool Population::have(Permutation& c) {
    //provjerava da li populacija sadrzi kromosom
    if (c.Leng() != chrLen) return false;
    else for(int i=0; i<size; i++)
        if (c == chroms[i]) return true;
    return false;
}
;
ostream& operator<<(ostream& os, Population& p) {
    // ispisivanje populacije
    cout << "Populacija:\n";
    for (int i=0; i<p.popSize()-1; i++)
        cout << p[i] << "\t" << p[i].Fitness() << ",\n";
    cout << p[i] << "\t" << p[p.popSize()-1].Fitness() << " !\n\n";
    return os;
}

```

5.4 Glavni program

Glavni program koji koristi sve prethodno opisane klase izgleda ovako:

```
#include "stdlib.h"
#include "time.h"
#include "Permutation.h"
#include "Population.h"
```

```

#include "PRV.h"

int main(int argc, char* argv[]) {
    cout << "Starting...\n";
    srand(time(0));

    PRV x;
    // x.LoadDotVRP("E-n101-k8.vrp");
    x.LoadDotVRP("A-n62-k8.vrp");
    x.CalculateDistances();
    x.CheckDemands();

    Permutation p(x.dim, x);
    Permutation best(x.dim, x);
    Permutation dad(x.dim, x);
    Permutation mom(x.dim, x);
    Population pop(30, x.dim, x);
    cout << pop;

    best = pop[0];
    float mincost = best.Fitness();
    float cost     = 1E10;
    for (int g = 0; true ; g++) {
        cout.flush();
        dad   = pop.selectBest(0.7);
        mom   = pop.selectBest(0.7);

        // for (int k = 2; k < x.dim; k++)
        // for (int l = k + 1; l <= x.dim; l++) {
        //     p = dad.OrderCrossover(mom, x, k, l);
        for (int m = 0; m < 20; m++) {
            p = dad.OrderCrossover(mom, x);

            for (int l = 0; l < 20; l++) {
                float rarLocA = p.RemoveAndReinsertLocalAll(x);
                if (rarLocA < mincost) {
                    mincost = rarLocA;
                    cout << " " << g << ":" l=" << l << ":" mincost = "
                        << mincost << ". (RARlocAll) \n"; cout.flush();
                }
            }
            p.PrintRoutes(x);
            best = p;
        }
    }
}

```

```

        cout << '\n'; cout.flush();
    }

float rarGloA = p.RemoveAndReinsertGlobalAll(x);
if (rarGloA < mincost) {
    mincost = rarGloA;
    cout << " " << g << ": l=" << l << ": mincost = "
        << mincost << ". (RARgloAll) \n"; cout.flush();
    p.PrintRoutes(x);
    best = p;
    cout << '\n'; cout.flush();
}

float twoOptLocCost = p.TwoOptLocal(x);
if (twoOptLocCost < mincost) {
    mincost = twoOptLocCost;
    cout << " " << g << ": l=" << l << ": mincost = "
        << mincost << ". (20ptLoc) \n"; cout.flush();
    p.PrintRoutes(x);
    best = p;
    cout << '\n'; cout.flush();
}

float twoOptGloCost = p.TwoOptGlobal(x);
if (twoOptGloCost < mincost) {
    mincost = twoOptGloCost;
    cout << " " << g << ": l=" << l << ": mincost = "
        << mincost << ". (20ptGlo) \n"; cout.flush();
    p.PrintRoutes(x);
    best = p;
    cout << '\n'; cout.flush();
}
}

if (!pop.have(best)) {
    (pop.selectWorst(0.15)) = best;
    pop.SortPop();
}
}

```

```

cout << "End!\n"; cout.flush();
return 0;
}

```

5.5 Rezultati

Implementacije opisana u prethodnom poglavlju je ustvari jedan vrlo, vrlo jednostavan oblik evolucijskog algoritma koji pruža još jako puno mogućnosti za poboljšanje. Cilj je bio samo da se u praksi provjeri dio teoretski proučenih i opisanih operatora križanja, mutacije i lokalnog pretraživanja. Ipak, napravljena je i jedna malena modifikacija do sada korištenih tehnika. Naime, svi evolucijski algoritmi se gotovo slijepo drže algoritma "selektiraj, križaj, mutiraj", određen broj generacija. Malena modifikacija koja je napravljena u gornjoj implementaciji pojavila se ustvari sasvim slučajno, a pokazala se prilično važnom i bitnom za brzinu i kvalitetu rješenja evolucijskog algoritma (barem na testiranim primjerima). Radi se o tome da se nije išlo strogo po principu "jedno križanje, pa jedna mutacija, pa lokalno pretraživanje, pa ispočetka", nego se nakon jednog križanja, različitim operatorima mutacije pretražilo malu okolinu dobijenog djeteta. Zatim se nisu birali novi roditelji, nego se određen broj puta pokušalo dobiti drugačije dijete od istih roditelja, pa ga se opet raznim mutacijama poboljšavalo. Naravno, nakon mutacije je opet slijedilo 2-opt lokalno pretraživanje.

Zanimljivo je da se ovakvom načinom mutiranja prilično rijetko događalo da 2-opt algoritam uspije poboljšati trenutno rješenje. Također, od tri implementirana operatora mutacije, neki su se pokazali znatno boljim od drugih. Kao daleko najbolja pokazala se mutacija ubacivanjem (RAR), i to ne lokalna, nego obična globalna verzija.

Također je primjećeno da brzina konvergencije tj. brzina nalaženja kvalitetnih rješenja jako ovisi o raznim parametrima evolucijskog algoritma. Takvi parametri su sljedeći:

1. broj jedinki u populaciji
2. postotak jedinki od kojih tražimo najbolju - parametar selekcije za križanje
3. postotak jedinki od kojih tražimo najgoru - parametar selekcije za odumiranje
4. broj uzastopnih mutiranja iste jedinke.

Pokazalo se da preko 100 jedinki samo nepotrebno usporava algoritam, a ne donosi gotovo nikakvu korist u kvaliteti najbolje jedinke. Također, manje od 10 jedinki donosi premalu genetsku različitost populacije (eng. diversity) [21], pa je vrlo vjerojatno brzo zaglavljivanje u lokalnom optimumu. Najbolje su se pokazale populacije s 30 do 50 jedinki.

Postotak jedinki od kojih tražimo najbolju je također jako bitan. Pokazuje se da, ako je jako visok (preko 0.8), uzrokuje vrlo brzu konvergenciju algoritma k prilično kvalitetnom rješenju, ali se zato uskoro "zaglavimo", jer visok postotak prisiljava stalno na izbor najkvalitetnijih jedinki, pa se nakon što pretražimo prostor oko njih, počnemo vrtiti u krug. S druge strane, ako je postotak jedinki od kojih tražimo najbolju jako nizak (ispod 0.2), pokazuje se da je konvergencija algoritma vrlo spora, jer dobar genetski materijal kvalitetnih jedinki ne može doći do izražaja. Najboljim postotkom se pokazao 0.7.

Slična, samo obrnuta, je situacija s postotkom jedinki od kojih tražimo najgoru. Ako je postotak previšok, uništavamo uvijek samo loše jedinke, zajedno s dijelovima dobrog genetskog materijala koje one nose, a time si povećavamo mogućnost zaglavljivanja u lokalnom optimumu. S druge strane, ako je postotak preveliki, možemo često uništiti previše dobrih jedinki, koje bi ipak trebale imati prioritet. Kao dobar izbor pokazala se vrijednost 0.15.

Što se tiče malene modifikacije standardnog načina - broja uzastopnih mutacija iste jedinke, pokazuje se da je najbolja metoda da jedna mutacija pretraži cijelu jednu malenu okolinu rješenja, zatim isto s ostalim vrstama mutacija. Također se pokazuje da je dobro da se cijeli postupak još ponovi do desetak puta prije idućeg križanja.

Zanimljivo je da, ukoliko bilo koji operator pustimo da djeluje samostalno (OX križanje, mutacija ubacivanjem, mutacija inverzijom, 2-opt lokalno pretraživanje) dobijemo prilično slabe rezultate, neznatno bolje od slučajnog pretraživanja. Pogotovo se kao loša metoda pri samostalnom djelovanju pokazalo OX križanje. Nasuprot tome, čim bilo koje dvije metode, ili više njih, pustimo da djeluju zajedno, kvaliteta rješenja strahovito poraste.

5.5.1 Rezultati za A-n62-k8 i E-n101-k8

Implementacija je kompajlirana i pokrenuta na dva benchmark problema `A-n62-k8.vrp` i `E-n101-k8.vrp`, koji se mogu naći na

[http://neo.lcc.uma.es/radi-aeb/WebVRP/
index.html?algorithms/TabuSearch.html](http://neo.lcc.uma.es/radi-aeb/WebVRP/index.html?algorithms/TabuSearch.html)

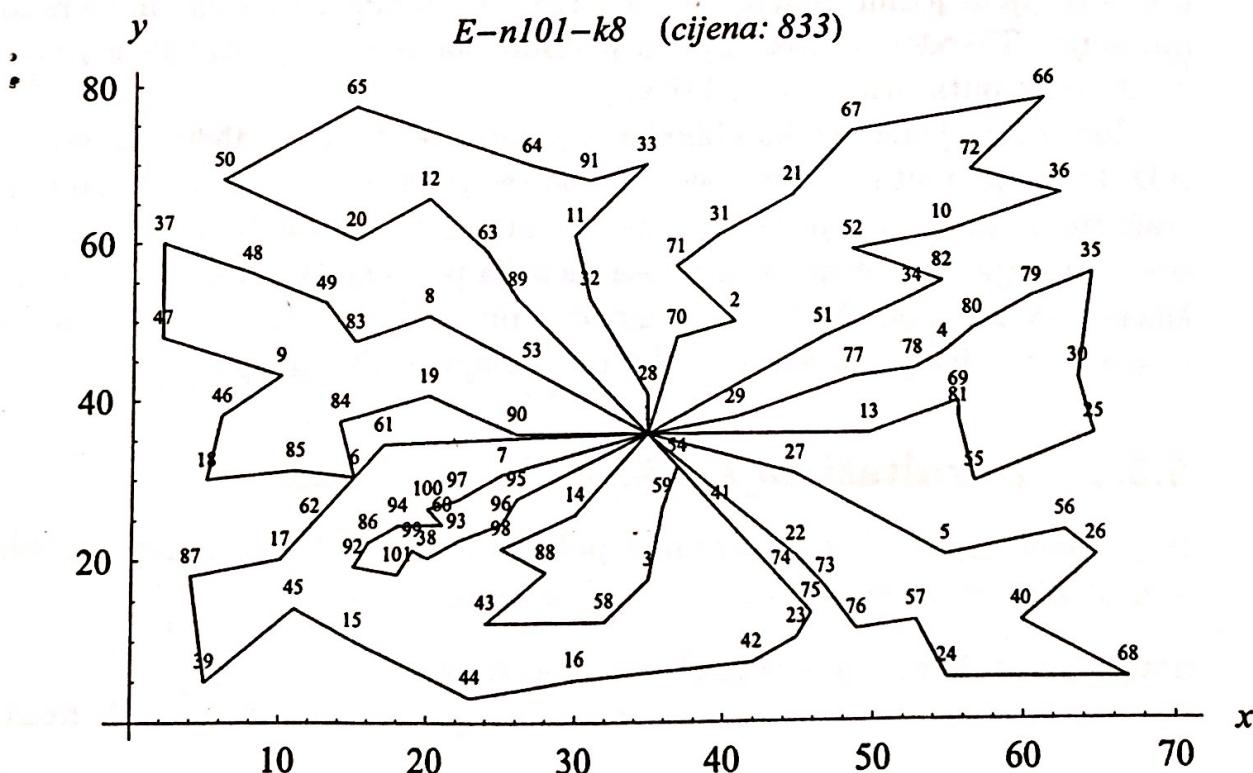
`E-n101-k8` je također Euklidski CVRP iz "Christophides and Eilon" biblioteke problema s 8 vozila i 101 klijentom. Kapacitet vozila je 200 jedinica

robe. Dobrota dosad navedenog najboljeg rješenja je 817 jedinica. Najbolje rješenje koje je dala u prethodnom poglavlju navedena implementacija je 833 jedinice, što je za 2% lošije od dosad najboljeg poznatog rješenja. Samo rješenje se grafički može predviđati koristeći E-n101-k8.vrp datoteku iz dorađenja ovog rada i prikazano je na slici 5.1. Raspored klijenata po vozilima izgleda ovako:

Vozilo 1 :	89 63 12 20 50 65 64 91 33 11 32 28
Vozilo 2 :	22 73 76 57 24 68 40 26 56 5 27
Vozilo 3 :	29 77 78 4 80 79 35 30 25 55 81 69 13
Vozilo 4 :	95 96 93 38 99 101 92 86 94 60 100 97 7
Vozilo 5 :	61 62 17 87 39 45 15 44 16 42 23 75 74 41
Vozilo 6 :	51 34 82 52 10 36 72 66 67 21 31 71 2 70
Vozilo 7 :	53 8 83 49 48 37 47 9 46 18 85 6 84 19 90
Vozilo 8 :	14 98 88 43 58 3 59 54

Važno je napomenuti da je ovdje skladište označeno s 1, tako da indeksi klijenata idu od 2 do 101, umjesto od 1 do 100. Tako su naime podaci složeni u E-n101-k8.vrp datoteci.

Slika 5.1:



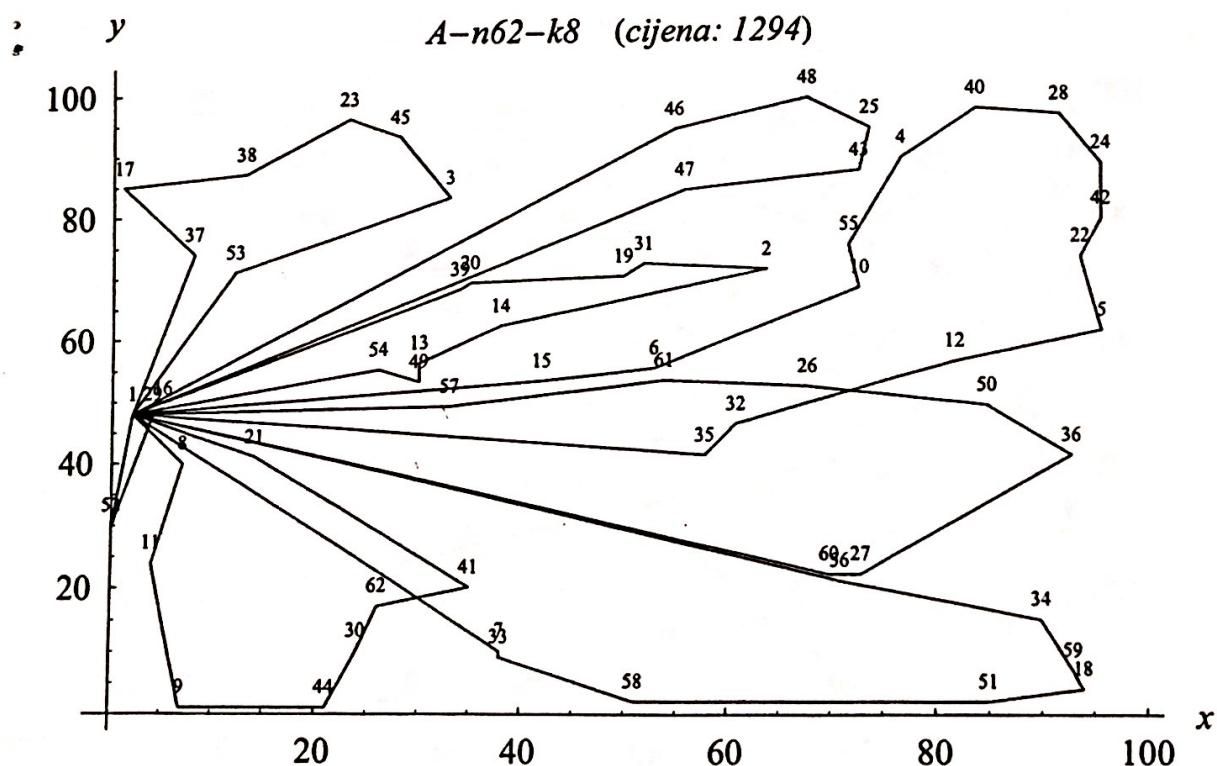
5.5. REZULTATI

A-n62-k8 je Euklidski CVRP iz "Augerat" biblioteke problema s 8 vozila i 62 klijenta. Kapacitet vozila je 100 jedinica robe. Dobrota optimalnog rješenja je 1288 jedinica. Najbolje rješenje koje je dala u prethodnom početku navedena implementacija je 1294 jedinice, što je samo 0,5% lošije od optimuma. Samo rješenje se, kao i u prethodnom slučaju, grafički može predviđati koristeći A-n62-k8.vrp datoteku iz dodatka ovog rada i prikazano je na slici 5.2. Raspored klijenata po vozilima izgleda ovako:

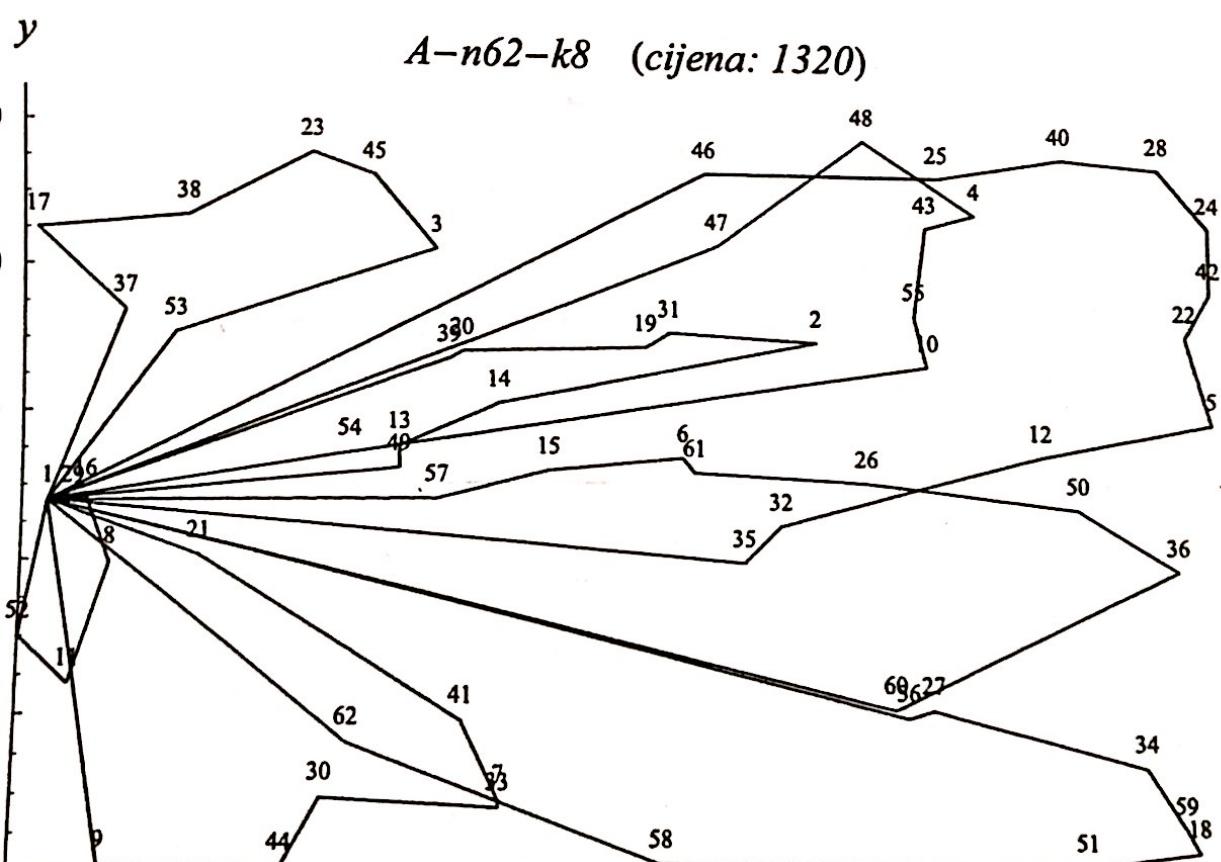
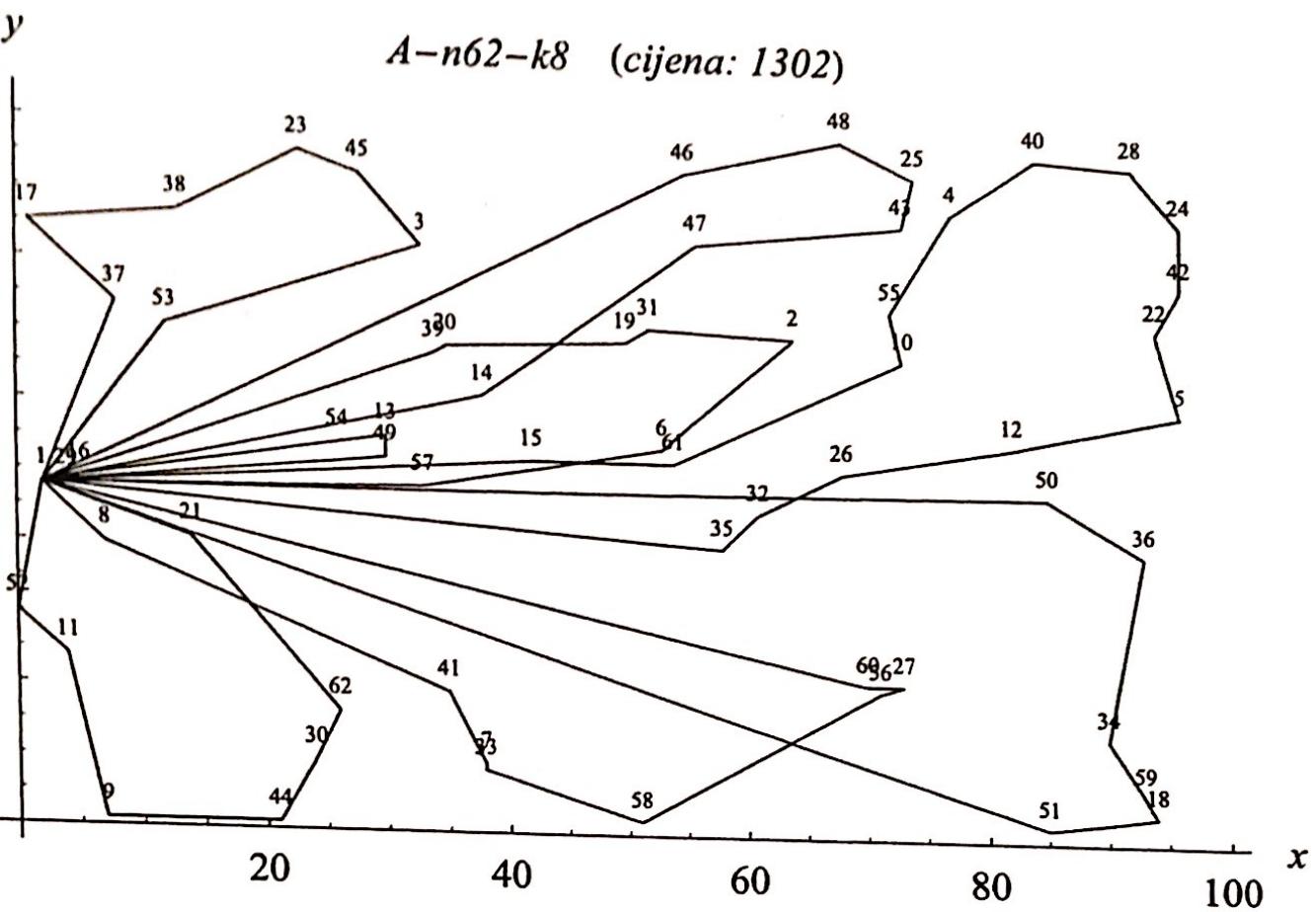
Vozilo 1 :	35 32 12 5 22 42 24 28 40 4 55 10 6 15
Vozilo 2 :	37 17 38 23 45 3 53
Vozilo 3 :	46 48 25 43 47
Vozilo 4 :	39 20 19 31 2 14 13 49 54
Vozilo 5 :	57 61 26 50 36 27 60
Vozilo 6 :	8 11 9 44 30 62 41 21
Vozilo 7 :	56 34 59 18 51 58 33 7
Vozilo 8 :	52 29 16

Ovdje je također važno napomenuti da je skladište opet označeno s 1, tako da indeksi klijenata idu od 2 do 62.

Slika 5.2:

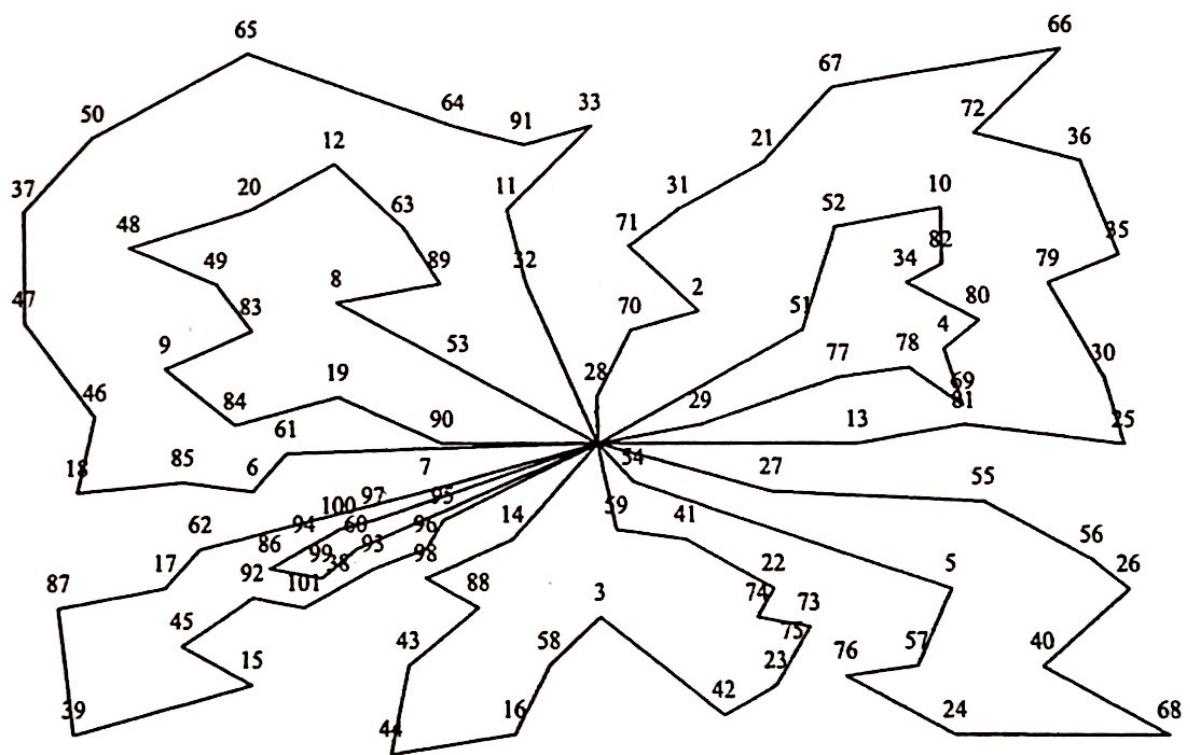


Uzima ponovljim pokretanjem programa iz prethodnog poglavlja.



10 20 30 40 50 60 70

E-n101-k8 (cijena: 842)



Zanimljivo je da su, i za jedan i za drugi test primjer, sva tri nezavisno dobijena rješenja prilično slična, odakle bi se dalo zaključiti da algoritam doista ide prema nekakvom globalnom optimumu. Također, ukoliko grafički pogledamo rješenja, možemo utvrditi da ona doista imaju smisla. Naime, povezani su oni klijenti koje bi i mi sami, raspoređujući klijente "na prste", najvjerojatnije povezali.