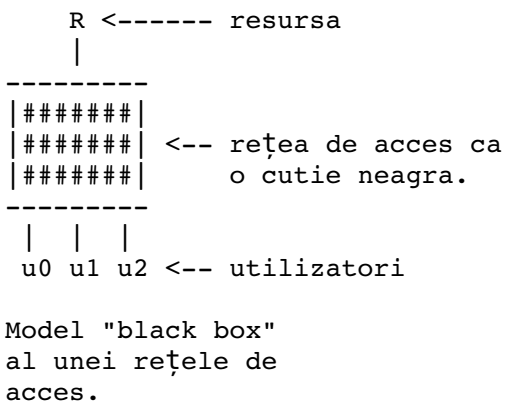
Algoritmi evolutivi aplicați în planificarea și proiectarea rețelelor de telecomunicații.

1.Introducere.

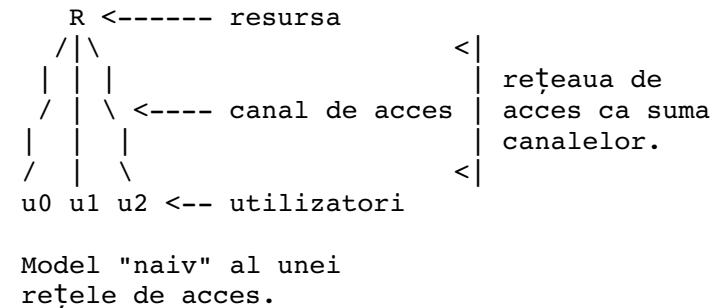
Această lucrare este un studiu al metodelor automate de construcție a rețelelor de acces optime din punct de vedere al costului, cu aplicabilitate în domeniul telecomunicațiilor. Realizarea de astfel de rețele automat prezintă interes pentru operatorii din domeniu, existând mari diferențe de preț intre o soluție funcționala, dar realizata de mana, și una realizata special de un calculator. Structural lucrarea are trei mari capitole: definirea problemei de rezolvat, descrierea unor metode pentru rezolvarea acesteia și descrierea implementării acestora într-un sistem real de optimizare. În final, sistemul este folosit pentru a găsii o rețea de acces pentru peste 1600 de utilizatori.

2.Problema.

Primul lucru pe care îl vom definii este chiar conceptul de "rețea de acces". Astfel, în termeni largi, o rețea de acces este un sistem ce permite unui număr de utilizatori sa comunice cu o resursa comuna. Ele sunt folosite pentru modelarea unei clase întregi de sisteme reale, ce include: rețele de electricitate, rețele de termoficare, sisteme de irigații, sisteme de canalizare și rețele de telecomunicații. Natura utilizatorilor, a resursei și a comunicației dintre aceștia este neimportanta momentan, astfel, rețeaua poate fi privita ca o cutie neagra (black box) ce permite fiecărui utilizator sa interacționeze cu resursa.



Fiindcă fiecare utilizator are un canal de comunicare separat cu resursa, putem reprezenta rețeaua de acces astfel:

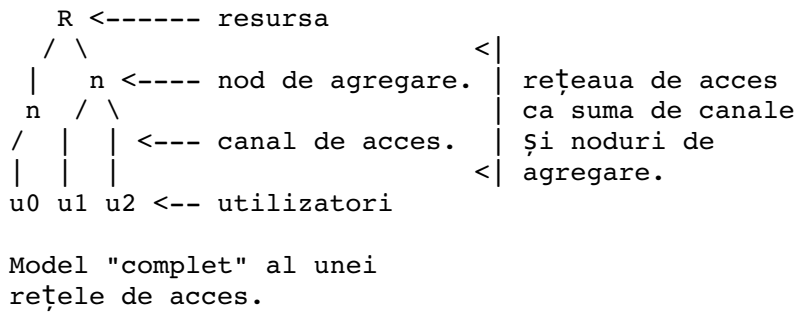


Putem spune ca un canal este o cale de comunicare dedicata intre un utilizator și resursa, iar o rețea de acces este multitudinea canalelor ce leagă toți utilizatorii de o resursa.

Având în vedere ca rețelele de acces sunt folosite pentru modelarea unor sisteme reale, trebuie ca

acestea sa tina cont de realitatea fizica. Acest lucru presupune introducerea unor limitări la care sistemele modelate sunt supuse. Doua dintre acestea sunt universale, iar introducerea lor în modelul rețelei de acces este de maxima importanta. Prima limitare este ca numărul de canale de acces legate direct la resursa este limitat, iar a doua este ca lungimea unui canal de comunicații este finita.

Folosindu-ne doar de conceptele definite pana acum putem modela un grup restrâns de sisteme reale. Pentru a putea modela rețele extinse, trebuie sa introducem conceptul de nod de agregare a comunicațiilor (nod). Nodul funcționează ca element ce unește mai multe canale de comunicație într-unul singur și segmentează canale prea lungi în sub-canale de lungimi acceptabile. Nodul este, deci, o entitate ce modelează mecanisme de depășire a limitărilor fizice mai sus enunțate.



Model "complet" al unei
rețele de acces.

Observam ca fiecare nod are un număr de canele downstream și un canal upstream (în figura, canalele downstream sunt canalele de sub nod, iar cel upstream, cel de deasupra lui). Funcția nodului este de a lua comunicările de pe toate canalele downstream și de a le transmite mai departe, către resursa, pe canalul upstream. Noduri pot fi introduse și pentru separarea unui canal prea lung în doua canale mai scurte, posibil sub limita de lungime a rețelei.

O consecința a acestui model este faptul ca nu mai exista o cale directa și dedicata intre un utilizator și resursa. În schimb, comunicarea utilizator-resursa trece printr-un număr de noduri și trebuie sa împartă un canal de acces cu alte comunicări. Spunem ca exista un canal virtual utilizator-resursa, vizibil doar acestuia dintâi. Funcția canalului de acces în noua taxonomie este de a fi o cale de transport pentru mai multe comunicații intre grupuri de utilizatori și resursa. Canalul de acces multiplexează comunicații virtuale pe un mediu fizic.

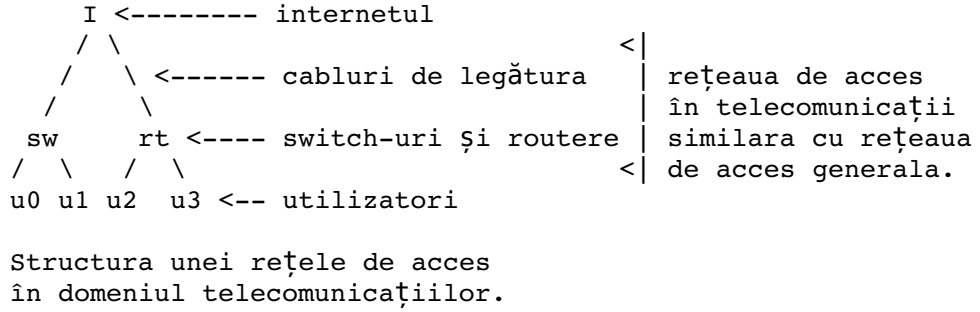
Așadar, o rețea de acces este un sistem ce permite unui număr de utilizatori sa comunice cu o resursa comuna. Rețeaua de acces este alcătuita din canale de acces și noduri de agregare. Rolul celor dintâi este de a transporta comunicări utilizator-resursa. Rolul celor din urma este, pe de-o parte, de a unii comunicări de pe canelele "downstream" și de a le transmite mai departe pe canalul "upstream", și, pe de alta parte, de a segmenta canale prea lungi în sub-canale mai mici. Rețeaua asigura un canal dedicat virtual intre fiecare utilizator și resursa. Din punct de vedere constructiv, o rețea de acces este o structura arborescenta, cu resursa ca nod rădăcina, utilizatorii ca nod frunza, și nodurile de agregare și canalele de acces pe post de noduri interne și arce intre acestea.

Ca și paranteza, când vorbim despre comunicare resursa-utilizator, sau comunicare de tip download, ne referim la faptul ca pe un anumit canal, printr-un anumit nod sau global în rețea exista o comunicare dinspre resursa spre utilizator, parcurgând rețeaua în jos de la rădăcina spre frunza țința. Același principiu se aplica și pentru comunicarea utilizator-resursa, numita și comunicare de tip upload, cu diferența ca parcurgerea rețelei se face de la frunza spre țința. În ambele cazuri prin comunicare se înțelege un schimb de materie sau un eveniment ce poate fi modelat ca atare. Exista o măsura a volumului de materie transmisa, un asa zis "bandwidth", măsurat în unități caracteristice

tipului de rețea studiat. Exista și o limita a volumului de materie ce poate circula la un moment dat printr-un canal sau printr-un nod, limita exprimata în unități de bandwidth. Prin agregarea comunicărilor de pe canalele downstream spre canalul upstream, volumul de materie transmis pe canalele upstream este egal cu suma volumului de materie de pe canalele downstream.

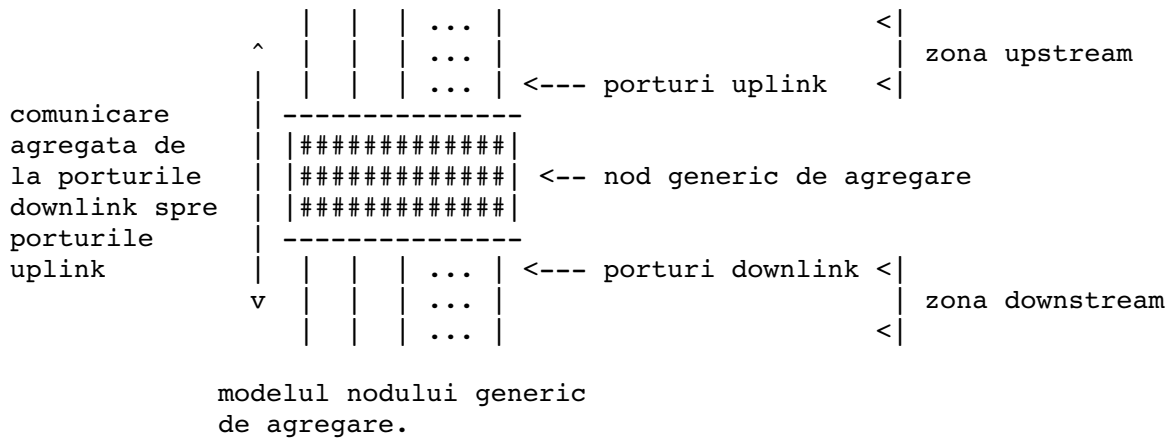
Revenind la exemplele de mai sus, pentru rețeaua de electricitate, resursa este energia electrica produsa într-o centrala, nodurile sunt diverse puncte de transformare, canalele sunt liniile de înalta tensiune ce leagă centrala de stațiile de transformare și stațiile de utilizatori, utilizatorii sunt consumatori domestici și industriali, iar comunicarea utilizator-resursa este transferul de energie electrica dinspre centrala înspre consumator, sub comanda celui din urma. Exprimarea ca materie a comunicării dintre uzina și consumator este, pe de-o parte, energia electrica, măsurata în KWh, și pe de alta parte, comenzi de cerere a unui volum de energie, exprimate ca cereri/secunda.

Interesante pentru noi, în schimb, sunt rețelele de acces din domeniul telecomunicațiilor, așa-numitele rețele de telecomunicații digitale. Exista o corespondență intre conceptele generale cu care am operat pana acum și conceptele specifice telecomunicațiilor. Resursa este un punct de acces spre o rețea mai mare, care de obicei este Internetul. Utilizatorii sunt entități care doresc acces la Internet. Depinzând de granularitatea modelarii, exemple de utilizatori sunt indivizi, companii sau alte rețele. Nodurile de agregare sunt echipamente de rețea precum switch-uri sau routere. Canelele de acces sunt diverse medii de transmisiune, dintre care enumeram: cabluri de cupru, cabluri de fibra optica, aer pentru transmisiuni wireless etc. În final, comunicarea resursa-utilizator este schimbul de date dintre Internet și client, sub controlul mixt al celor doi. Nu ne interesează în mod special cum funcționează rețeaua și ce forma iau transferurile de date, ci doar ca trăsăturile și limitările generale ale rețelelor de acces se aplica. Astfel, canalele de acces multiplexează comunicații de la multi utilizatori, iar fiecare utilizator, la un anumit nivel de abstracție, comunica independent cu resursa. De asemenea, switch-urile și routerele accepta un număr limitat de legături iar datorita atenuării semnalelor prin canalele de comunicare, lungimea acestora este limitata. În final, metodologia generala de a realiza rețele de acces în domeniul telecomunicațiilor presupune realizarea unei structuri arborescente, cu un nod rădăcina reprezentând conexiune la Internet, noduri de acces către utilizatori ca frunze, switch-uri și routere intermediare ca noduri și legăturile intre acestea ca arce ce definesc structura întregului copac.



Structura unei rețele de acces
în domeniul telecomunicațiilor.

În discuția ce urmează vom considera toate legăturile ca fiind cabluri de cupru. De asemenea, vom considera nodurile ca fiind echipamente de comutație generice. Nodurile sunt caracterizate de un număr de porturi ce accepta legături dinspre utilizatori, numite porturi de downlink, și un număr de porturi care se conectează spre rădăcina, numite porturi de uplink. Întreaga subrețea conectata la un nod se cheamă rețeaua downstream a nodului, iar rețeaua la care se conectează acesta se cheamă rețeaua upstream a nodului. Așadar, numărul de porturi downlink și numărul de porturi uplink sunt primele doua trăsături ce individualizează tipurile de noduri.



modelul nodului generic
de agregare.

Pornind de la o configurație de problema, ce consta într-o descriere a utilizatorilor ce trebuiesc conectați la resursa și o descriere a nodurilor ce pot fi folosite, ne punem doua probleme: construirea automata a unei rețele de acces și găsirea rețelei de acces optime.

Prima dintre probleme este destul de clara: trebuie sa găsim o metoda de a construii o rețea ce leagă resursa de toți utilizatorii folosind noduri alese dintr-un set de noduri posibile. Trebuie așadar sa construim un copac ce are ca rădăcina resursa, ca frunze utilizatorii și ca noduri intermediare, noduri aparținând de setul de noduri admise. Exista o distincție importanta intre rețele de acces valide și cele nevalide, iar aceasta este ca, în cazul rețelelor valide, toate nodurile sunt legate la resursa (copacul construit are toți utilizatorii ca noduri frunze). Procesul de construcție trebuie sa tina cont de criteriul de validitate și sa construiască doar rețele valide.

A doua problema introduce însa dificultatea definirii ideii de optim intre rețelele de acces. Folosind doar definițiile de pana acum, rețeaua de acces optima ar putea fi înțeleasa ca acea rețea ce folosește un număr de noduri cât mai mic. Aceasta modelare nu este însa suficienta, neputând reprezenta realități din procesul de realizare al unei rețele fizice, și aici ma refer, în special, la ideea ca, din punctul de vedere al constructorului, numărul de noduri este separat de costul propriu-zis al rețelei. Introducem așadar noțiunea de cost asociat fiecărui tip de nod din configurația de problema. Uzual, costul se reprezinta ca un număr, iar costuri mai mici corespund unor soluții mai bune. Introducem, de asemenea, și noțiunea de cost asociat unei rețele de acces, definita ca suma costurilor nodurilor folosite.

Avem așadar un criteriu de comparare a doua rețele de acces: consideram cea mai buna rețea dintr-un grup pe aceea cu costul cel mai mic. Putem definii, deci, problema găsirii rețelei de acces optim ca o problema de găsire a rețelei de acces de cost minim. De notat ca, momentan, nu introducem costurile legăturilor și alte costuri indirecte, gen costurile de întreținere a echipamentelor, în definiția problemei.

Un singur concept mai trebuie sa introducem în definirea problemei, anume acela de obligații fata de utilizatori. În sistemele reale, simpla conectare a utilizatorilor la resursa nu este suficienta. Astfel, fiecare utilizator are un număr de cerințe în legătura cu comunicarea sa. Cea mai întâlnita forma de cerința este aceea a asigurării unui volum al comunicațiilor la un anumit nivel, lucru exprimat prin faptul ca fiecare utilizator are garantata o lărgime de banda (bandwidth) anume, exprimata într-un multiplu de biți per secunda (bps,kbps,mbps,gbps). Rețeaua pe care o construim trebuie sa poată asigura utilizatorilor cel putin un nivel de trafic egal cu cel garantat. Cerințele la nivelul utilizatorului se

exprima ca doua numere: unul ce descrie bandwidth-ul pentru download (transmisia de date de la resursa către utilizator) și unul care descrie bandwidth-ul pentru upload (transmisia de date de la utilizator spre resursa). Porturile fiecărui nod sunt, la rândul lor, caracterizate de doua astfel de numere, ce descriu volumul de date în cele doua direcții maxim ce poate fi suportat. Pentru simplitate presupunem ca toate nodurile downlink și uplink sunt identice intre ele. Putem conecta un utilizator la un port al unui nod doar dacă volumul de trafic maxim pe un port de downlink al nodului este mai mare decât volumul de trafic necesar utilizatorului. Într-o rețea, fiecare nod agrega datele de la nodurile downstream și le transmite spre nodurile upstream. Astfel, volumul de trafic ce merge spre upstream este suma volumelor de trafic generat pe fiecare port downlink (nu volumul maxim, ci volumul actual). Fiecare port de uplink preia o parte egala din volumul total de date din downstream. Așadar, pentru conectarea unui port uplink al unui nod la un port downlink al unui alt nod, trebuie ca volumul de trafic pe portul uplink sa fie mai mic decât volumul maxim permis pe portul de downlink.

Avem, deci, încă o constrângere în realizarea de rețele, și încă un criteriu pentru a determina dacă o rețea de acces este valida sau nu. Astfel, rețele valide sunt doar acelea care conectează toți utilizatorii la resursa și care asigura necesarul de trafic pentru fiecare client (volumul de trafic pe fiecare port din fiecare nod este mai mic decât volumul maxim admis).

O observație importanta este ca prima problema, cea a generării automate de rețele de acces este o subprobleme a celei de-a doua. Discuția noastră în continuare se va concentra așadar pe aceasta.

În concluzie, problema pe care încercam sa o rezolvam este găsirea automata a unei rețele de telecomunicații de acces de cost minim, pornind de la o configurație de problema data, care sa permită accesul unui număr de utilizatori, în condiții stabilite, la o resursa comuna. Configurația de problema descrie un grup de utilizatori, cerințele acestora în materie de lărgime de banda, un grup de noduri folosibile și caracteristicile esențiale ale acestora: cost, număr și tipul de porturi downlink și număr și tipul de porturi uplink. Construirea unei rețele de acces presupune construirea unei structuri arborescente, având ca rădăcina resursa și ca frunze utilizatorii, cu noduri alese dintre cele din configurație și cu un număr de legături ce definesc canalele de comunicație intre utilizatori,resursa și noduri. Caracteristici importante ale rețelelor de acces sunt costul, calculat ca suma costurilor tuturor nodurilor folosite, și validitatea, determinata de facilitarea accesului tuturor utilizatorilor la resursa și asigurarea ratelor de transfer cerute de către fiecare utilizator. Găsirea rețelei de acces optime presupune găsirea acelei rețele de acces care este atât valida cât și de cost minim.

3.Modelare

Primul pas în modelarea matematica a problemei este reprezentarea configurației de problema ca o structura matematica. Considerând definiția configurației de problema ca o descriere a utilizatorilor ce trebuiesc acoperiți și a nodurilor ce pot fi folosite în construcția rețelei, putem folosii următorul model pentru aceasta:

```
CP : configurație de problema
CP = (CP:U,CP:N)
unde CP:U : descrierea utilizatorilor
     CP:U = { ui : i de la 1 la ||CP:U|| }
     CP:N : descrierea nodurilor
     CP:N = { ni : i de la 1 la ||CP:N|| }
```

Descrierea utilizatorilor și descrierea nodurilor sunt mulțimi de descriptori de utilizatori, respectiv noduri. Conform definiției informale a problemei, un utilizator este caracterizat de volumul de trafic ce ii trebuie asigurat, măsurat în unități de bandwidth separate pentru comunicarea download și upload.

```
ui = (ui:dl,ui:ul)
unde ui:dl : bandwidth-ul minim pentru download
     ui:ul : bandwidth-ul minim pentru upload

     ui:dl ∈ R+
     ui:ul ∈ R+
```

În același timp, un nod este caracterizat de: cost, numărul de porturi downlink, numărul de porturi uplink și volumul maxim de comunicații pe fiecare tip de port. Volumul maxim este dat în unități de bandwidth separate pentru comunicarea download și upload, similar cu definiția descriptorului de client.

```
ni = (ni:c,ni:nrd,ni:d:dl,ni:d:ul,ni:nru,ni:u:dl,ni:u:ul)
unde ni:c : costul nodului.
     ni:nrd : numărul de porturi downlink.
     ni:d:dl : bandwidth-ul maxim pentru download pe porturile de downlink.
     ni:d:ul : bandwidth-ul maxim pentru upload pe porturile de downlink.
     ni:nru : numărul de porturi de uplink
     ni:u:dl : bandwidth-ul maxim pentru download pe portul de uplink.
     ni:u:ul : bandwidth-ul maxim pentru upload pe porturile de uplink.

     ni:c ∈ R+
     ni:nrd ∈ N+
     ni:d:dl ∈ R+
     ni:d:ul ∈ R+
     ni:nru ∈ N+
     ni:u:dl ∈ R+
     ni:u:ul ∈ R+
```

Un exemplu de configurație cu 12 utilizatori, 6 tipuri de noduri folosibile, și unitatea de baza de măsura a lărgimii de banda este MB, este:

```
                          # ui:dl  ui:ul
    cp1 = (cp1:U = { u1  = (1.024, 0.128),
                     u2  = (0.512, 0.128),
                     u3  = (0.512, 0.064),
                     u4  = (1.024, 0.256),
                     u5  = (2.048, 1.024),
                     u6  = (0.512, 0.128),
                     u7  = (0.512, 0.256),
                     u8  = (1.024, 0.256),
                     u9  = (1.024, 0.128),
                     u10 = (0.512, 0.512),
                     u11 = (0.128, 0.064),
                     u12 = (0.256, 0.128) },
                           # ni:c   ni:nrd ni:d:dl ni:d:ul ni:nru ni:u:dl ni:u:ul
           cp1:N = { n1  = (100,   4,     10,     2,      1,     20,     2),
                     n2  = (40,    2,     10,     4,      1,     20,     2),
                     n3  = (150,   8,     10,     1,      2,     100,    10),
                     n4  = (180,   4,     12,     4,      1,     40,     10),
                     n5  = (250,   4,     20,     10,     1,     100,    30),
                     n6  = (500,   4,     40,     40,     1,     150,    150)})
```

Configurația de problema definește parametrii unei instante a problemei pe care încercam sa o rezolvam. Construcția unei rețele de acces va depinde astfel de configurația aleasa. Pentru reprezentarea acesteia vom folosii o structura liniara, în loc de structura intuitiva, arborescenta. Alegerea este motivata de practica, realizarea operațiilor necesare pentru implementările algoritmilor de optimizare făcându-se mult mai ușor.

Descrierea unei rețele de acces se face printr-o lista de indici ale nodurilor folosite.

```
RA(CP) : rețea de acces, dependenta de o configurație de problema anume
RA(CP) ∈ reuniune {1,2,..,||CP:N||}^i pentru i de la 1 la dimMax
unde dimMax : numărul maxim de noduri dintr-o rețea, calculat în Anexa 1
```

Exemple de astfel de rețele ar fi:

```
ra1(cp1) = (1,2,3,4)
ra2(cp1) = (6,4,4,1)
```

exemple de rețele de acces.

Fiecare element al unei descrieri de rețea de acces este înțeles ca indicele unui element din mulțimea descriptor de noduri din configurația de problema. Având doar lista de noduri folosite într-o rețea nu putem determina forma sa completa. Abordarea aleasa atribuie un înțeles și ordinii în care acestea apar, astfel ca, știind nodurile și ordinea acestora, și folosind o metoda de evaluare adecvata, putem construii forma arborescenta a rețelei.

Algoritmul de evaluare este următorul: consideram descrierea utilizatorilor și rețeaua de acces. Consideram ca descriptorii aranjați într-o anumita ordine constituie nivelul 0 al rețelei : nivelul nodurilor frunza. Primul nod din lista de noduri, are un număr de n:nrd porturi downlink și un număr de n:nru porturi uplink. Fiecare port downlink se leagă cu unul din primele n:nrd noduri downlink. Nodurile legate sunt retrase de pe nivelul 0, rămânând un număr mai mic de noduri de servit, iar pe nivelul 1 se introduc porturile uplink ale nodului curent. Procedeul continua pe același nivel, trecând prin nodurile rețelei de acces, pana când nu mai rămân utilizatori. Odată ce nivelul 0 a fost acoperit, întreg procesul

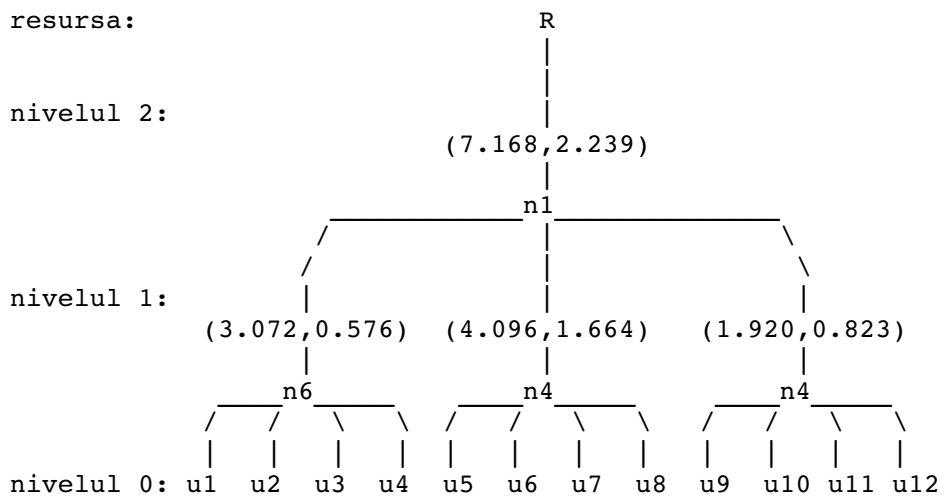
se repeta pentru nivelul 1, considerând porturile uplink introduse de nodurile de la nivelul 0 ca utilizatori pentru nivelul 1. Volumul de trafic ce trebuie garantat pentru utilizatorii de la nivelul 1 este dat de suma volumelor de trafic ai utilizatorilor de nivel 0 agregați de un nod, împărțit în mod egal la porturile uplink introduse de nod. Algoritmul se încheie când pe ultimul nivel exista un singur nod.

Nu orice lista de noduri din SP_RA(CP) pentru un CP, oarecare, este valida ca și rețea de acces. Trebuie ca toți utilizatorii sa fie conectați la resursa (lucru tradus prin faptul ca pe ultimul nivel în procesul de evaluare trebuie sa se afle un singur nod) și trebuie ca prin fiecare port sa treacă un volum de trafic mai mic decât limita portului respectiv.

Un exemplu de evaluare a unei rețele de acces:

```
ra2(cp1) = (6,4,4,1)

   inițial avem utilizatorii ce trebuiesc conectați la
   resursa așezați într-o ordine oarecare (în acest caz,
   crescător după indicele de utilizator)

resursa:                          R


nivelul 0: u1  u2  u3  u4  u5  u6  u7  u8  u9  u10 u11 u12

   primul nod este n6 care retrage 4 utilizatori de pe
   nivelul 0 și introduce un port de uplink/utilizator pe
   nivelul 1. Necesarul de trafic este acoperit. În paranteza
   deasupra lui n6 este traficul agregat al nodurilor u1,u2,u3
   și u4 ce trece prin portul uplink.

resursa:                          R


nivelul 1:         |
             (3.072,0.576)
                   |
                 ____n6_____
                /   /   \   \
                |   |   |   |
nivelul 0: u1  u2  u3  u4  u5  u6  u7  u8  u9  u10 u11 u12

   continuam pe același nivel cu nodurile n4 și n4 care
   acoperă utilizatorii u5,u6,u7 și u8 și, respectiv u9,u10
   u11 și u12.

resursa:                          R


nivelul 1:         |                |                |
             (3.072,0.576)  (4.096,1.664)   (1.920,0.823)
                   |                |                |
                 ____n6_____     ____n4_____     ____n4_____
                /   /   \   \   /   /   \   \   /   /   \   \
                |   |   |   |   |   |   |   |   |   |   |   |
nivelul 0: u1  u2  u3  u4  u5  u6  u7  u8  u9  u10 u11 u12
```

nivelul 0 este complet. Trecem la nivelul 1 și consideram
porturile de uplink ca și consumatori. Folosim ultimul
nod n1 pentru a completa rețeaua.



nivelul 2 este complet și el, și odată cu el întreaga rețea.
În paranteza deasupra nodului n1 este volumul de trafic
din întreaga rețea.

De observat ca ultimul nod de pe un nivel poate avea doar o parte din porturile de downlink ocupate. Depinzând de tipul de nod ar putea sa accepte legături de la porturile de uplink de pe același nivel. Ar rezulta un număr de noduri mai mic, dar rețeaua nu ar mai fi arborescenta, și, în general, ar fi mai greu de lucrat cu ea. Păstram aceasta mica ineficienta, iar, mai târziu, putem folosii porturile libere ca locuri de extins rețeaua, și putem optimiza rețeaua și în acest sens (obținerea unei rețele de cost minim dar maxim extensibila).

Pentru o anumita configurație de problema exista un spațiu al tuturor soluțiilor posibile. Dacă introducem constrângerea ca un nod sa nu aibă mai multe porturi de uplink decât porturi de downlink, atunci acest spațiu este finit, și mai mult, putem calcula numărul de elemente. Aceasta constrângere este plauzibila și vom opera în continuare ținând cont de ea. Mai multe despre detalii despre aceasta alegere se afla în Anexa 1. Numim spațiul tuturor rețelelor de acces construibile pentru o configurație de problema astfel:

```
SP_RA(CP) : spațiul rețelelor de acces construibile cu o configurație CP
```

Exista un spațiu al rețelelor de acces construibile cu o anumita configurație de problema care sunt și valide. Acesta este un sub-spatii al spațiului rețelelor de acces pentru aceeași configurație.

```
SP_RAV(CP) : spațiul rețelelor de acces valide construibile cu o configurație CP
SP_RAV(CP) ⊂ SP_RA(CP)
```

În Anexa 1 vom calcula numărul de elemente din acest spațiu și vom arata ca acesta depinde într-un mod exponențial de numărul de clienți și de numărul de noduri din configurație.

Definim în continuare funcția cost a unei rețele de acces, ca suma costurilor nodurilor ce o alcătuiesc. Funcția cost este parametrizata cu configurația de problema (așadar fiecare configurație de problema definește o funcție cost asociata):

```
ra_cost(CP) : SP_RA(CP) -> R
ra_cost(CP)((i0,i1,..,ik)) = n_i0 + n_i1 + … + n_ik
```

Problema noastră consta în găsirea unei rețele de acces de cost minim, lucru care se traduce prin minimizarea funcției ra_cost(CP).

```
min ra_cost(CP)
```

Acest lucru presupune găsirea unei rețele de acces din SP_RA(CP), RA* astfel încât:

```
ra_cost(CP)(RA*) < ra_cost(CP)(ra), oricare ra ∈ SP_RA(CP).
```

Avem așadar de-a face cu o problema de minimizare a unei funcții. Pentru tipul nostru spațiul nostru : discret și finit, aceasta problema este defapt o problema de căutare. Pentru găsirea de o soluție pentru problema de minimizare exista numeroase metode, asa zise metode de optimizare.

4.Metode de optimizare

Am spus așadar ca problema găsirii rețelei de acces optime este echivalenta problemei găsirii rețelei de cost minim din spațiul SP_RA(CP). Mai mult, aceasta rețea trebuie sa se găsească în SP_RAV(CP). Constrângerea din urma este cea care da dificultate problemei. Fără ea, rețeaua de cost minim se găsește ușor, fiind cea care conține un singur nod, anume acela de cost minim dintre nodurile folosibile. Evident, aceasta nu prezintă interes, reîndeplinind decât pentru rețelele foarte mici cerințele de trafic.

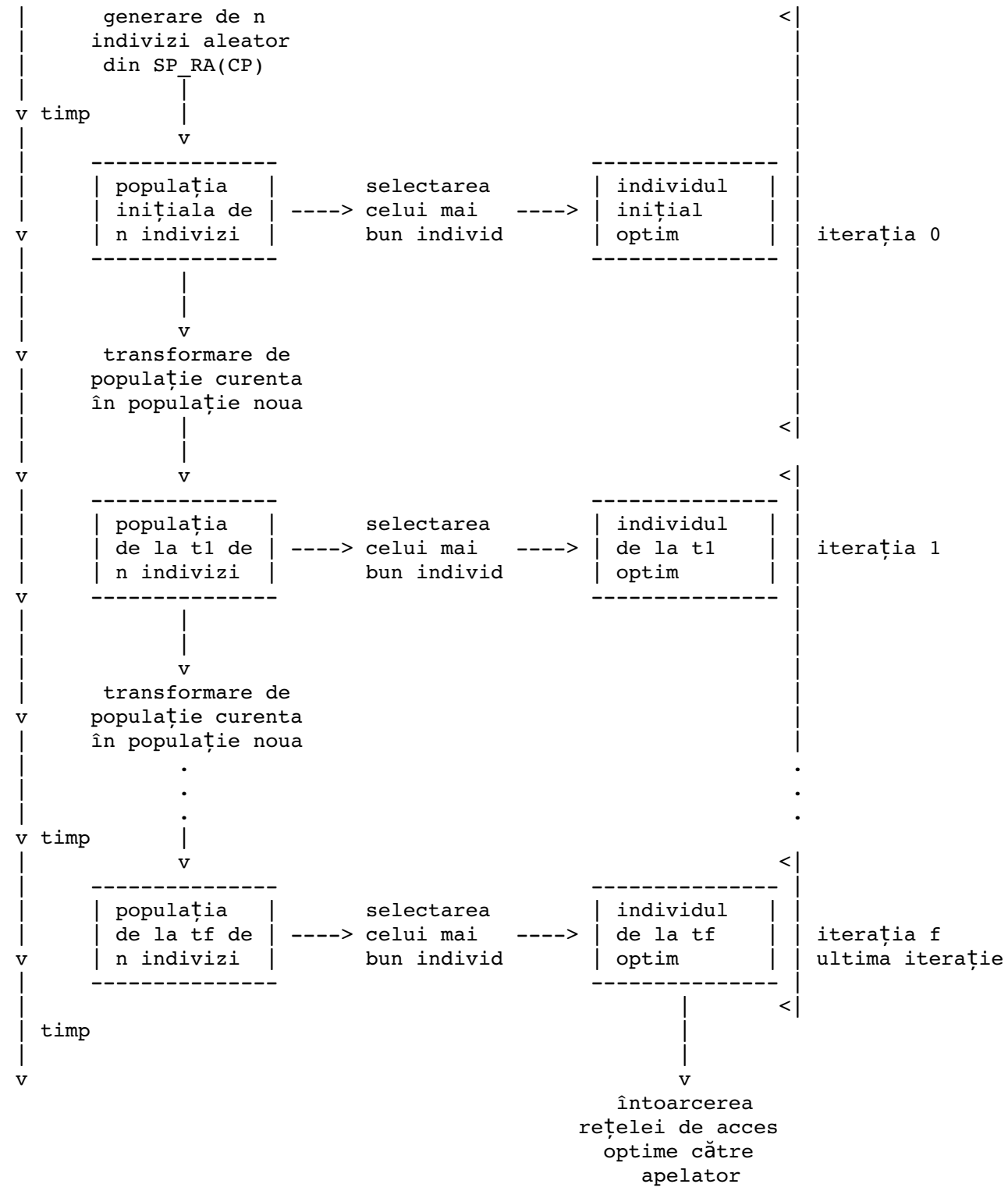
Singura metoda care produce garantat optimul este căutarea exhaustiva a lui SP_RA(CP) și retinerea membrului de cost minim întâlnit. Dimensiunea spațiului, conform Anexei 1, face aceasta metoda impracticabilă. Exista totuși o clasa de metode de găsire a minimului, numite tehnici de optimizare sau metode de optimizare, ce efectuează o căutare mai inteligenta decât cea bruta, găsind o soluție apropiata de optim. În aceasta idee, se fac un număr de presupuneri ce simplifica procesul de căutare. Dintre metodele de optimizare le vom studia pe cele stocastice, ce oferă cele mai bune rezultate pentru probleme cu spatii discrete și fără vreo structura anume.

Presupunerile făcute în procesul de căutare "inteligent" sunt doua la număr. Prima dintre ele spune ca rețele de acces asemănătoare au costuri asemănătoare. A doua spune ca prin efectuarea de mici modificări la o rețea de acces se produc modificări de cost mici. În general, spațiul de căutare trebuie sa aibă proprietatea ca nu conține zone cu schimbări bruște de cost, iar, în cel mai râu caz, aceasta proprietate trebuie sa fie valida local. Analizând SP_RA(CP) intuitiv, acesta poseda proprietatea, pe când SP_RAV(CP), nu. Așadar, în procesul de căutare, vom folosii diversele tehnici pentru explorarea spațiului SP_RA(CP) și vom "repara", separat, fiecare soluție produsa.

Evident, soluția obținuta prin aceste metode nu mai este garantat optimul, dar ele reprezinta un compromis intre eficienta de căutare și timpul de căutare.

Metodele de optimizare pe studiate fac parte din categoria de metode evolutive și se numesc: Evolution Strategies și Genetic Algorithm. În scopul dezvoltării sistemului de optimizare detaliat mai târziu, doua alte metode (neevolutive) au fost studiate și implementate: Random Search și Hill Climbing. Acestea sunt metode mai simple, și, deoarece introduc un număr de concepte importante celorlalte doua metode, le vom discuta mai întâi, lăsând discuția despre metodele evolutive pentru sfârșit.

În primul rând, exista un număr de concepte comune tuturor metodelor: individ, populație, iterație și individ optim. Un individ este o rețea oarecare de acces, considerata ca posibila soluție pentru problema de optimizare. O populație este o colecție de indivizi. Fiecare metoda întreține o populație curenta, ce constituie mare parte din starea algoritmului. Explorarea spațiului se face generând o populație noua pornind de la populația curenta prin aplicarea de diverși operatori membrilor ei. Populația noua înlocuiește populația veche la sfârșitul procesului de generare. Timpul de la înlocuirea populației curente pana la generarea uneia noi se numește iterație, iar algoritmii funcționează prin efectuarea acestui proces de generare-inlocuire de un număr de ori, numit număr de iterații. Separat fiecare metoda menține și un individ optim, care este rețeaua de acces de cost minim dintre toate rețelele din toate populațiile generate în procesul de optimizare. Menținerea separata a acestuia permite o explorare mai îndrăzneața a spațiului, fiind permise mișcări spre indivizi mai slabi, fără frica pierderii unui posibil optim. În final, la începutul procesului exista o populație inițiala, cu indivizi aleși aleator din spațiul de căutare.



evoluția în timp a procesului de optimizare văzut
prin prisma schimbărilor de stare a implementării
unei metode de optimizare.

Din acest moment prezentarea va fi mai degrabă algoritmica decât declarativa/matematica. Trecem de la descrierea obiectelor de studiu la descrierea acțiunilor asupra acestor obiecte. Vom face referire, de asemenea, și la sistemele reale care rezolva problema noastră, și care sunt implementări ale

diferitelor tehnici de optimizare. Vom vedea ca algoritmii definesc un număr de operații asupra indivizilor și populațiilor, dar pentru a avea imaginea completa asupra procesului de optimizare, trebuie sa facem referire și la detalii de implementare.

Din punct de vedere al notațiilor voi folosii o descriere în pseudo-cod a algoritmilor. Avem astfel flexibilitate, mai multi utilizatori sunt familiari cu "limbajul" folosit și putem introduce elemente de notație matematica ce nu sunt suportate, de obicei, în alte limbaje. Aceasta din urma trăsătura da și o oarecare continuitate notațiilor, deoarece structurile pe care operează algoritmii sunt aceleași descrise anterior.

Prezint aici câteva notații și funcții cu care vom opera, semantica acestora fiind în general ușor de înțeles pentru persoane familiare cu un limbaj de programare.

```
Obiecte cu care lucram:

  Numere
  Valori Logice
  Liste

  Exemple:
    10,20,3.17 # numere
    T,F        # valori logice True (T) și False (F)
    (1,2,3)    # lista de numere
    (1,(2,3))  # lista complexa de numere

Crearea unui nume și atribuirea unei valori acestuia:

  [nume] <- [valoare]

  Exemple:
    x <- 10
    pi <- 3.14
    t <- x + pi
    v <- (1,2,3,4,5)

Execuție condiționata:

  dacă [condiție]
  atunci
     [operații în caz de evaluare adevărata a condiției]
  altfel
     [operații în caz de evaluare falsa a condiției]

  Exemple:
    dacă x == 10
    atunci
      t <- 1
    altfel
      t <- 2
```

Execuție repetata:

```
  pentru [i] în [structura secvențiala]
    [operații asupra fiecărui element din structura secvențială]
    [i are rol de "obiectul curent" în procesare]

  cattimp [condiție]
    [operații]

  Exemple:
    s <- 0

    pentru i în (1,2,3,4):
      s <- s + i

    cattimp s > 0
      s <- s / 2 – 1
```

Funcții:

```
  [nume] ([parametru0],...,[parametruN]) =
    [operații asupra parametrilor]
    întoarce [rezultat]

  Exemple:

    patratGt10 (x) =
      y <- 0

      dacă x < 10
      atunci
        y <- x + 1
      altfel
        y <- x * x

      întoarce y
```

Funcții predefinite:

```
  adaugă [lista] [element]
  inserează [lista] [indice] [element]
  Șterge [lista] [indice]
  schimba [lista] [indice] [element]
  element [lista] [indice]
  primii [index] [lista]
  unește [lista1] [lista2]
  interval [lista] [start] [stop]

  random [start] [stop]

  Exemple:
    t <- (1,2,3)

    adaugă t 4 => (1,2,3,4)
    inserează t 2 7 => (1,2,7,3,4)
    Șterge t 4 => (1,2,7,4)
    schimba t 3 3 => (1,2,3,4)
    element t 2 => 2
```

```
primii 2 t => (1,2)
unește t (5,6) => (1,2,3,4,5,6)
interval t 2 4 => (2,3,4)

random 0 2 => 1.4
```

Starea curenta a unei implementări a unei metode oarecare de optimizare poate fi descrisa de următoarea structura matematica:

```
SO(CP,N,M) : starea curenta pentru o CP, N iterații și M indivizi în populație.
SO(CP,N,M) = (SO:it,SO:P,SO:B)
unde SO:it : iterația curenta.
     SO:P : populația curenta.
     SO:P = (SO:P:I0,SO:P:I1,...,SO:P:IM)
     SO:B : rețeaua de acces optima.

     SO:it ∈ {,2,..,N}
     SO:P ∈ SP_SAV(CP)^M
     SO:B ∈ SP_SAV(CP)
```

Ca exemplu de stare curenta, pentru configurația de problema folosita pana acum, avem:

```
so1(CP,10,2) = {so1:it = 3,
                so1:P = (so1:P:I0 = (1,2,3,4),
                so1:P:I1 = (6,4,4,1)),
                so1:B = (1,2,3,4)}
```

Asemănător descrierii unui individ/rețea de acces, exista și un spațiu al stărilor pentru o anumita configurație de problema, și, mai nou, pentru un anumit număr maxim de iterații și o anumita mărime a populației curente. Acest spațiu este:

```
SP_SO(CP,N,M)
```

Având în vedere ca fiecare componenta a unui element SO(CP,N,M) oarecare aparține de o mulțime finita, și SP_SO(CP,N,M) este un spațiu de dimensiuni finite.

Procesul de explorare a spațiului, din iterație în iterație, poate fi descris atunci de o funcție de forma:

```
explorare(CP,N,M) : funcția ce descrie procesul de explorare al unei metode
explorare(CP,N,M) : SP_SO(CP,N,M) -> SP_SO(CP,N,M)
```

Algoritmic aceasta funcție se exprima astfel, pentru cel mai general caz de metoda de optimizare:

```
explorare(CP,N,M) (stareCurenta) =
  stareNoua <- (0,(),())

  stareNoua:it <- stareCurenta:it + 1
  stareNoua:P <- evolutiePopulatie stareCurenta:P
  stareNoua:B <- individulDeCostMinim stareNoua:P stareCurenta:B

  întoarce stareNoua
```

Singura operație rămasa nedefinita este evolutiePopulatie. Pentru aceasta, avem:

```
evolutiePopulatie(CP,N,M) : funcția de transformare a unei populații într-alta.
evolutiePopulatie(CP,N,M) : SP_SAV(CP)^M -> SP_SAV(CP)^M
```

În imaginea cu evoluția în timp a procesului de optimizare, aceasta funcție corespunde blocului de "transformare de populație curenta în populație noua". Ce anume se întâmpla pentru a genera o populație noua dintr-o populație curenta depinde de tehnica, dar, în principiu, funcția va fi exprimata în termeni de operatori asupra indivizilor. Acești operatori, în realitate funcții definite pe SP_RA(CP), sunt în număr de patru, și anume: generarea unei de rețele aleatoare, modificarea unei rețele, combinarea a doua rețele pentru a forma o a treia și repararea unei rețele.

Generarea populației inițiale se face cu funcția generarePopulatie. Aceasta creează o populație formata din indivizi formați de către operatorul de generare de rețele aleator. Forma acestei funcții este:

```
generarePopulatie(CP,N,M) : SP_RAV(CP)
generarePopulatie(CP,N,M) =
   populație <- ()

   pentru i în (1,2,..M)
      adaugă populație (ra_reparare (ra_genereAleatoare))

   întoarce populație
```

Selectarea individului de cost minim se face cu funcția individulDeCostMimim ce are următoarea forma:

```
individulDeCostMinim(CP,N,M) : SP_RAV(CP)^M -> SP_RAV(CP) -> SP_RAV(CP)
invividulDeCostMinim(CP,N,M) (populatie,celMaiBunTrecut) =
  celMaiBun <- ()
  celMaiBunCost <- +inf

  pentru individ în populație
    dacă ra_cost(individ) < celMaiBunCost
    atunci
      celMaiBun <- individ
      celMaiBunCost <- ra_cost(individ)

  dacă celMaiBunTrecut != () si celMaiBunCost < ra_cost(celMaiBunTrecut)
  atunci
    întoarce celMaiBun
  altfel
    întoarce celMaiBunTrecut
```

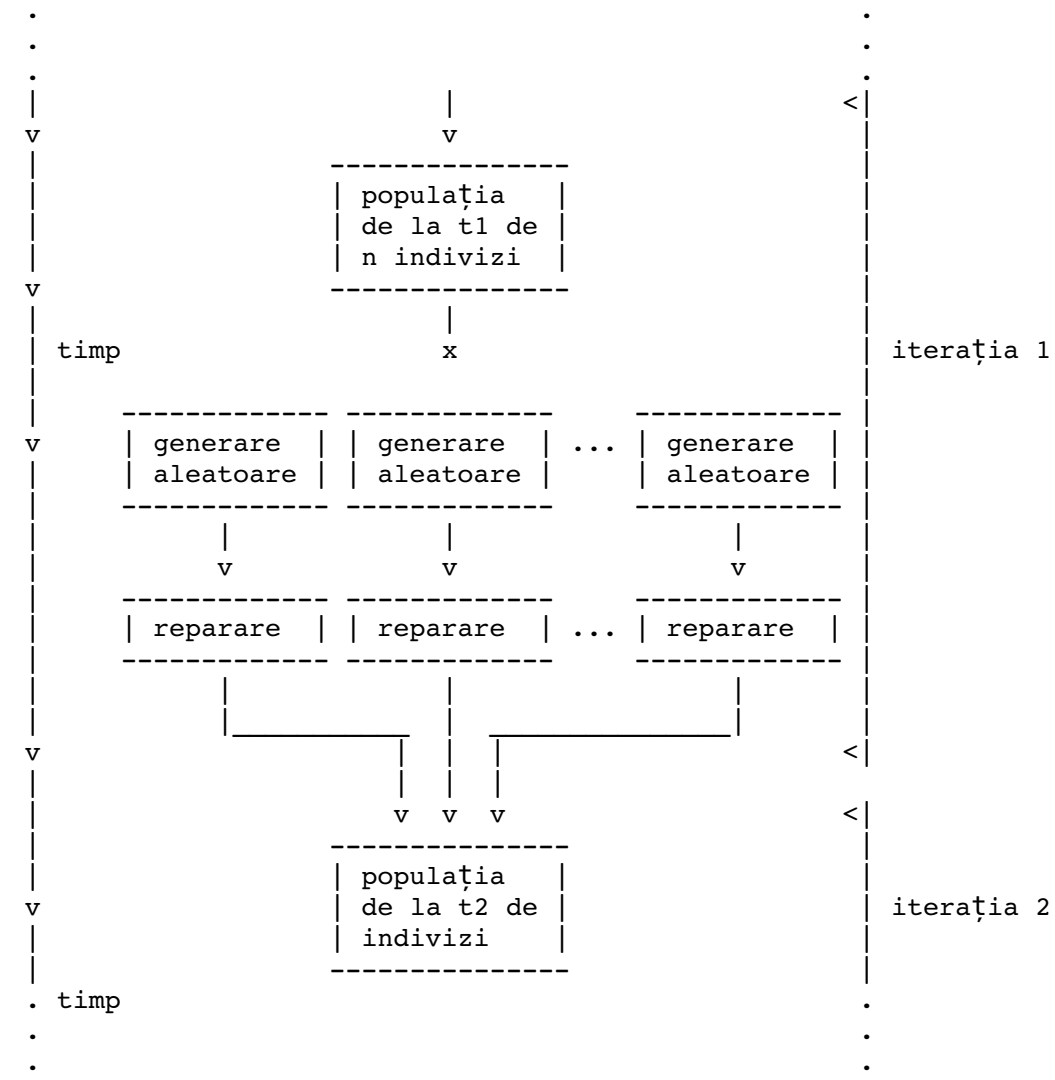
4.1.Random Search.

Prima metoda studiata este cea numita Random Search. Aceasta se folosește doar de operatorii de generare de rețele aleatoare și reparare de rețele. Ideea de baza este ca, la fiecare iterație, se generează o populație complet noua, selectata aleator din SP_RA(CP). Se face astfel o parcurgere nestructurata a spațiului, iar la sfârșit, cea mai buna dintre rețelele întâlnite este întoarsa ca rezultat. Evident algoritmul nu este eficient, el nefolosindu-se de presupunerile în băgătura cu forma spațiului de căutare.

Funcția de evoluție a populației se poate exprima atunci ca:

```
evolutiePopulatie(CP,N,M) (populație) =
    populație <- ()

    pentru i în (1,2,..,M):
       adaugă populație (ra_reparare (ra_genereAleatoare))

    întoarce populație
```

Procesul de evoluție poate fi vizualizat astfel:



Am stabilit ca folosim tehnicile de optimizare pentru explorarea spațiului SP_RA(CP), urmând ca soluțiile nevalide găsite sa fie constrânse la soluții valide din SP_RAV(CP). Operatorul de reparare realizează acest lucru. Mai mult, deoarece generarea de rețele de acces valide este un proces complicat, nu vom insista ca fiecare operator sa genereze o astfel de soluție, folosind în schimb operatorul de reparare pentru a aduce rețeaua generata la o forma valida.

Primul operator pe care îl vom descrie este cel de reparare. Acesta transforma o rețea de acces invalida într-una valida. Dacă rețeaua este deja valida, nu are loc nicio transformare.

```
  ra_reparare(CP) : operatorul de reparare de rețele de acces pentru o
configurație CP.
  ra_reparare(CP) : SP_RA(CP) -> SP_RAV(CP)
```

Cunoaștem ca o rețea de acces nu este valida dacă nu reușește sa conecteze toți utilizatorii la resursa sau dacă nu acoperă necesarul de trafic al acestora. Operatorul de reparare are rolul de a adaugă de ajunse noduri la retele pentru a conecta utilizatorii și de a schimba nodurile care nu fac fata la trafic. Pe de alta parte, nici rețelele cu prea multe noduri nu sunt de dorit, ele fiind oricum departe de optim. Așadar, operatorul de reparare trebuie sa se ocupe și de acestea.

Definirea în linii mari a acestuia este:

```
  ra_reparare(CP) (rețea) =
    pentru i în rețea
      dacă [n_i din CP:N nu satisface cerințele de trafic]
        i <- [nod suficient de puternic]

      [completează nivelele conform metodei de evaluare]

    dacă [pe ultimul nivel este un singur nod]
    atunci
      întoarce rețea
    altfel
      dacă [sunt mai multe noduri pe ultimul nivel]
      atunci
        cattimp [pe ultimul nivel nu este un singur nod]
          adauga rețea [nod suficient de puternic]
          [completează nivelele conform metodei de evaluare]
      altfel dacă [sunt prea multe noduri]
      atunci
        cattimp [pe penultimul nivel este un singur nod]
          [taie ultimul nod]
          [ajustează nivelele conform metodei de evaluare]

      întoarce rețea
```

Descrierea algoritmului este intenționat de nivel înalt, fără a intra în multe detalii, pentru a fi ușor de înțeles. Pentru o descriere în detaliu, se poate consulta anexa ce descrie pe larg sistemul de optimizare implementat.

Pentru următoarea rețea, ce asigura conectivitatea tuturor nodurilor dar nu îndeplinește cerințele de trafic, procesul de reparare este:

```
ra1 = (1,3,5)

operatorul de reparare evaluează descriptorul
de rețea ra1, încercând sa construiască forma
arborescenta.

resursa:                        R


nivelul 0: u1  u2  u3  u4  u5  u6  u7  u8  u9  u10 u11 u12
```

primul nod este n1 care retrage 4 utilizatori de pe nivelul 0 și introduce un port de uplink pe nivelul 1.

```
resursa:                        R


nivelul 1:         |
             (3.072,0.576)
                   |
                 ____n1_____
                /   /   \   \
                |   |   |   |
nivelul 0: u1  u2  u3  u4  u5  u6  u7  u8  u9  u10 u11 u12
```

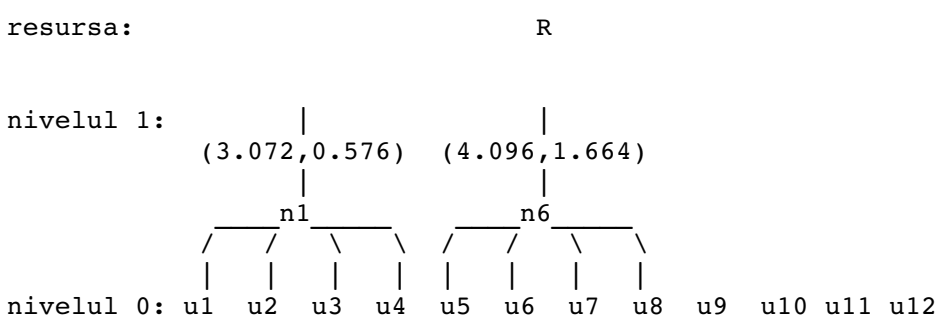
al doilea nod, n3, acoperă toți utilizatorii de pe nivelul 0 și introduce un singur port de uplink pe nivelul 1.

```
resursa:                        R


nivelul 1:         |                        |
             (3.072,0.576)            (6.016,2.496)
                   |                        |
                 ____n1_____     ____________n3_____________
                /   /   \   \   /   /   /   /   \   \   \   \
                |   |   |   |   |   |   |   |   |   |   |   |
nivelul 0: u1  u2  u3  u4  u5  u6  u7  u8  u9  u10 u11 u12
```

nodul de tipul n3 nu face fata cerințelor de trafic din downstream pe partea de upload. Utilizatorul u5 are nevoie de 1.024 Mb upload dar porturile de downlink ale lui n3 suporta doar 1 Mb upload. Așadar rețeaua este invalida. Trebuie schimbat nodul n3 cu un altul. Operatorul alege aleator nodul n6. Acesta înlocuiește pe n3 iar descriptorul rețelei devine:

```
ra1 = (1,6,5)
```

structura arborescenta se transforma în:

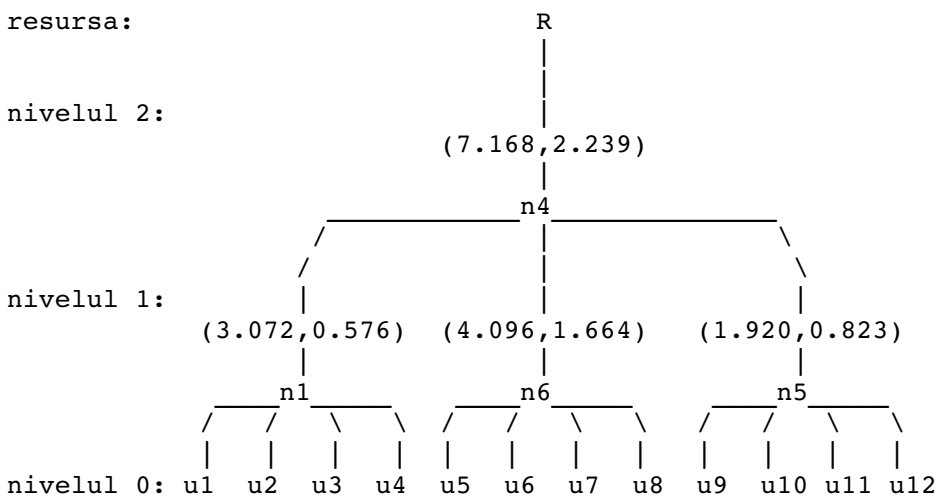


continuam procesul introducând și nodul n5, care conectează ultimii clienți de pe nivelul 0 și poate satisface toate cerințele de trafic. Deși inițial nodul n5 era pe nivelul 1, schimbarea lui n3 cu n6 îl aduce pe acesta pe nivelul 0.

pe moment am rămas fără noduri în ra1. Operatorul de reparare trebuie sa compenseze acest lucru prin introducerea de noi noduri, continuând procesul de evaluare, pana când rețeaua este completa. Se alege nodul n4 aleator ce completează rețeaua. Descriptorul de rețea devine:

ra1 = (1,6,5,4)

iar reprezentarea structurii arborescente este:



în final, operatorul întoarce rețeaua ra1 către apelatorul sau.

Munca grea a realizării unei retele valide se petrece în acest operator. Faptul ca el poate repara orice rețea produsa de alți operatori, ne da o mare flexibilitate în definirea acestora. Drept urmare, operatorul de generare de rețea aleatoare din spațiul SP_RAV(CP) nu face altceva decât sa întoarcă o rețea fără niciun nod, bazându-se pe operatorul de reparare sa construiască o rețea valida, pornind practic de la zero.

```
ra_generareAleatoare(CP) : SP_RA(CP)
ra_generareAleatoare(CP) = ()
```

De observat de asemenea ca operatorul de generare de rețele aleatoare este definit peste spațiul total al rețelelor de acces, nu doar peste cel al rețelelor valide. Toți ceilalți operatori vor fi, de asemenea, definiți pe acest spațiu, reflectând faptul ca nu produc garantat soluții valide, și ca soluțiile produse, trebuie reparate.

4.2.Hill Climbing.

A doua metoda de optimizare studiata poarta numele de Hill Climbing, și este prima care se folosește într-adevăr de constrângerile impuse și presupuse valide ale spațiului de căutare. Ideea de baza presupune trecerea de la populația curenta la populația iterației următoare, considerând fiecare individ separat. Astfel, pornind de la fiecare dintre indivizii din populația curenta, se generează cațe un grup de indivizi de explorare, diferind putin de individul sursa. Cel mai bun din fiecare grup este selectat ca înlocuitor al originalului în populația iterației viitoare. Putem vedea ca aceasta metoda se folosește într-adevăr de constrângerile impuse, presupunând ca, mergând din aproape în aproape și din mai bine în mai bine din punct de vedere al costului prin spațiul de explorare, se va ajunge la o soluție optima. Explorarea cu mai multi indivizi în paralel (ce nu face parte din algoritmul clasic) asigura o apropiere de optim mai buna.

Aceasta metoda introduce al treilea operator asupra unei retele, anume acela de mutație. Acest operator pornește de la un individ valid, și produce o mica schimbare în structura acestuia. Rețeaua de acces produsa nu este garantat valida, asa ca trebuie folosit și operatorul de reparare.

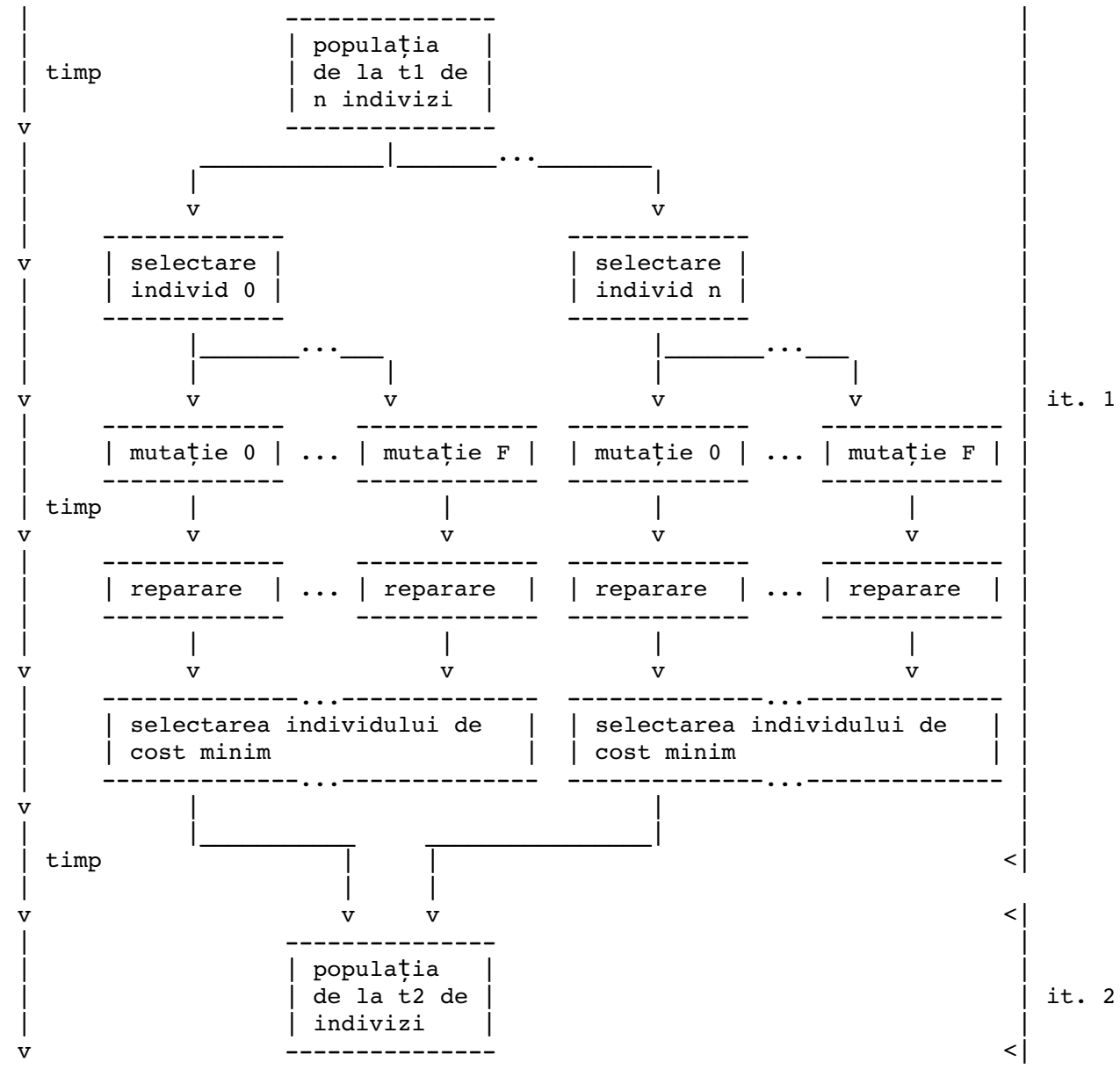
Metoda depinde și de un parametru în plus, numit factorul de explorare. Acesta determina câți indivizi sa conțină grupul de explorare din jurul unui membru al populației. Valori mari ale acestuia determina o deplasare rapida spre optimul local, de-a lungul evoluției metodei, la fiecare iterație fiind selectate cu o mai mare probabilitate soluții mai bune decât cea curenta. Aceasta explorare agresiva nu este întotdeauna de dorit, iar, pentru valori mici ale factorului de explorare se pot obține rezultate mai bune, depinzând de structura spațiului, deoarece deplasări spre zone de calitate mai slaba din punct de vedere al costului pot duce, ulterior, la atingerea, în final, a unor soluții mai bune. Acest factor este reprezentat matematic ca un număr întreg și pozitiv, strict mai mare ca zero.

```
F : factorul de explorare
F ∈ N+
```

Funcția de evoluție a populației se poate exprima astfel:

```
evolutiePopulatie(CP,N,M) (populație) =
   populatieNoua <- ()

   pentru i în (1,2,..,M)
      grupExplorare <- ()

      pentru j în (1,2,..,F)
         adauga grupExplorare (ra_reparare(ra_mutatie (element populație i)))

      adaugă populatieNoua (individulDeCostMinim grupExplorare ())

   întoarce populatieNoua
```

Procesul de evoluție poate fi vizualizat astfel:



Operatorul de mutație poate modifica o rețea de acces în trei feluri: prin schimbarea unui nod cu alt tip de nod, prim adăugarea unui nod sau prin scoaterea unui nod. Toate trei operațiile au riscul de a produce un rezultat invalid, asa ca este importanta folosirea operatorului de reparare după o mutație.

În linii mari, putem definii mutația astfel:

```
ra_mutatie(CP) (rețea) =
  operație <- random 0 1
  indice <- random 0 ||rețea||

  dacă operație < 0.2
  atunci
    # adauga un nod
    inserează rețea indice (random 0 (||CP:N||-1))
  altfel dacă operație < 0.4
  atunci
    # scoate un nod
    șterge rețea indice
  altfel
    # schimba un nod
    schimba rețea indice (random 0 (||CP:N||-1))

  întoarce rețea
```

De notat ca mutația este primul operator de natura genetica. El joaca un rol important în metodele Evolution Strategie și Genetic Algorithm, dar, întâmplator, modul de operație al metodei Hill Climbing se pretează și el la o descriere cu acest operator. Pentru a păstra descrierile mai concise, am adoptat astfel terminologie evolutiva pentru un algoritm care nu face parte din aceasta categorie.

4.3.Evolution Strategy

Aceasta a treia metoda de optimizare este prima din categoria celor evolutive, care au format direcția principala a studiului. Ideea de baza este următoarea: fiind dat un număr lambda=M de indivizi dintr-o populație la o anumita iterație, pentru a genera populația iterației următoare, triem populația la cei mai buni miu indivizi în jurul cărora efectuam mutații pentru a obține lambda/miu noi indivizi. Aceștia formează noua populație. Metoda este oarecum similara cu Hill Climbing (dacă aveam un factor de explorare F=1, deși, ca paranteza, și în acest caz putem genera în jurul unui individ câte F * lambda/miu indivizi, selectând din fiecare lambda/miu grup de F indivizi, pe cel mai bun pentru introducerea în populația viitoare), cu adăugarea ca se introduce un pas intermediar de selectare a unor indivizi din populația curenta. Procesul de selecție este general aplicabil mai multor metode, iar în acest caz poarta numele de selecție prin trunchiere : se selectează doar un număr de cele mai bune retele de acces din punct de vedere al costului.

Caracteristic metodelor evolutive este ca în procesul de generarea a unui individ nou, are influenta întreaga populație, nu doar individul părinte de baza. Acest lucru le diferențiază de metode de căutare precum Hill Climbing si Random Search sau metodele nestudiate Simulated Annealing, Quantum Annealing, Tabu Search etc. În cazul metodei Evolution Strategy, generarea unui nou individ este determinata de poporație prin folosirea procesului de selecție, în sensul ca, un număr de indivizi de calitate proasta nu vor mai ajunge candidați pentru mutație.

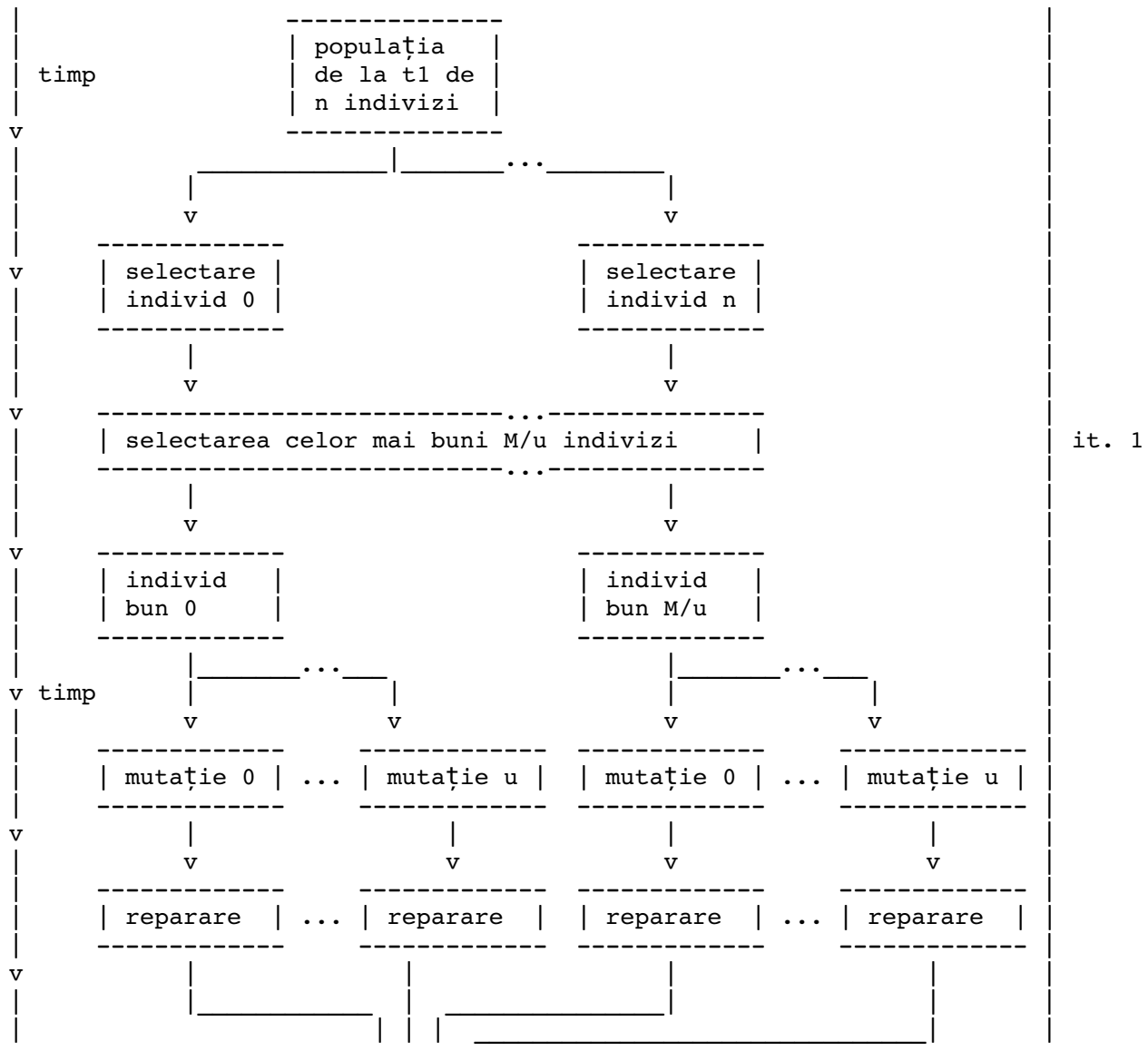
De notat ca exista multe variante de Evolution Strategy. Varianta prezentata este una dintre cele mai simple: ES (miu,lambda). Alte exemple sunt ES (miu + lambda), ES (miu/rho,lambda), ES (miu/rho +

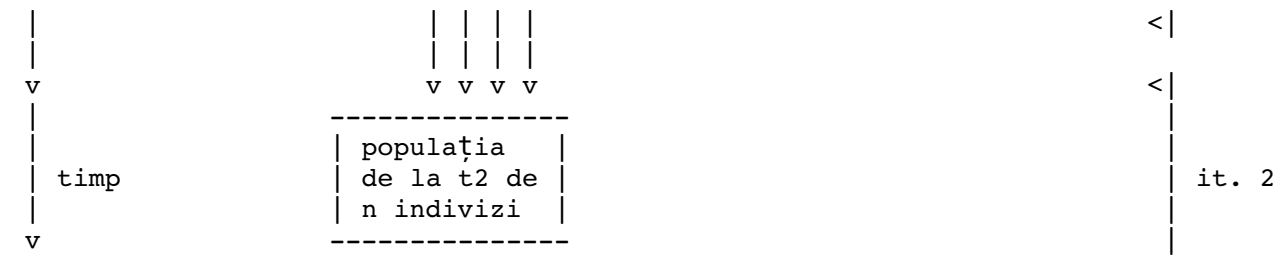
lambda), CMA-ES etc. Ca și operatori, nu folosește decât pe cei introduși pana acum: cel de generare aleatoarea a unui individ, cel de mutație și cel de reparare.

Funcția de evoluție a populației se poate exprima astfel:

```
u ∈ N+
M `mod` u == 0

evolutiePopulatie(CP,N,M) (populație) =
   populatieNoua <- ()
   ceiMaiBuni <- primii u [sortează_după_cost populație]

   pentru i în (1,2,..,M/miu)
      pentru j în (1,2,..,miu)
         adauga populatieNoua (ra_reparare (ra_mutatie ceiMaiBuni_i))

   întoarce rețea
```

Procesul de evoluție poate fi vizualizat astfel:





4.4.Genetic Algorithm.

Metoda Genetic Algorithm este cea mai sofisticata din punct de vedere al implementării, atât algoritmic, cât și la nivel de implementare : nu mai putin de opt versiuni ale metodei de baza sunt implementate. Metoda face parte, evident, din categoria celor evolutive, și este cea la care legătura cu sistemele biologice ce au fost sursa de inspirație este cea mai puternica.

Ideea de baza a metodei este următoarea: oprind de la populația iterației curente, selectam un număr de indivizi, egal cu cel al populației inițiale. Parcurgând în ordine populația, din fiecare pereche de doi indivizi se generează doi noi indivizi, printr-o asa zisa operație de combinare (exprimata prin operatorul de combinare). Aceștia doi sunt adaugați populației stării viitoare. Exista o mica șansa ca după ce procesul de combinare este terminat, un individ aleator sa sufere o mutație. Fiind data natura operatorului de combinare (încrucișarea a doi părinti, fără introducere de "informație" noua), rolul mutației este acela de a introduce noi trăsături ale indivizilor și de a forța explorarea unei zone mai mari a spațiului de căutare (în ideea ca făcând doar recombinări se poate explora acea parte a spațiului determinata de toate recombinările posibile intre părinti).

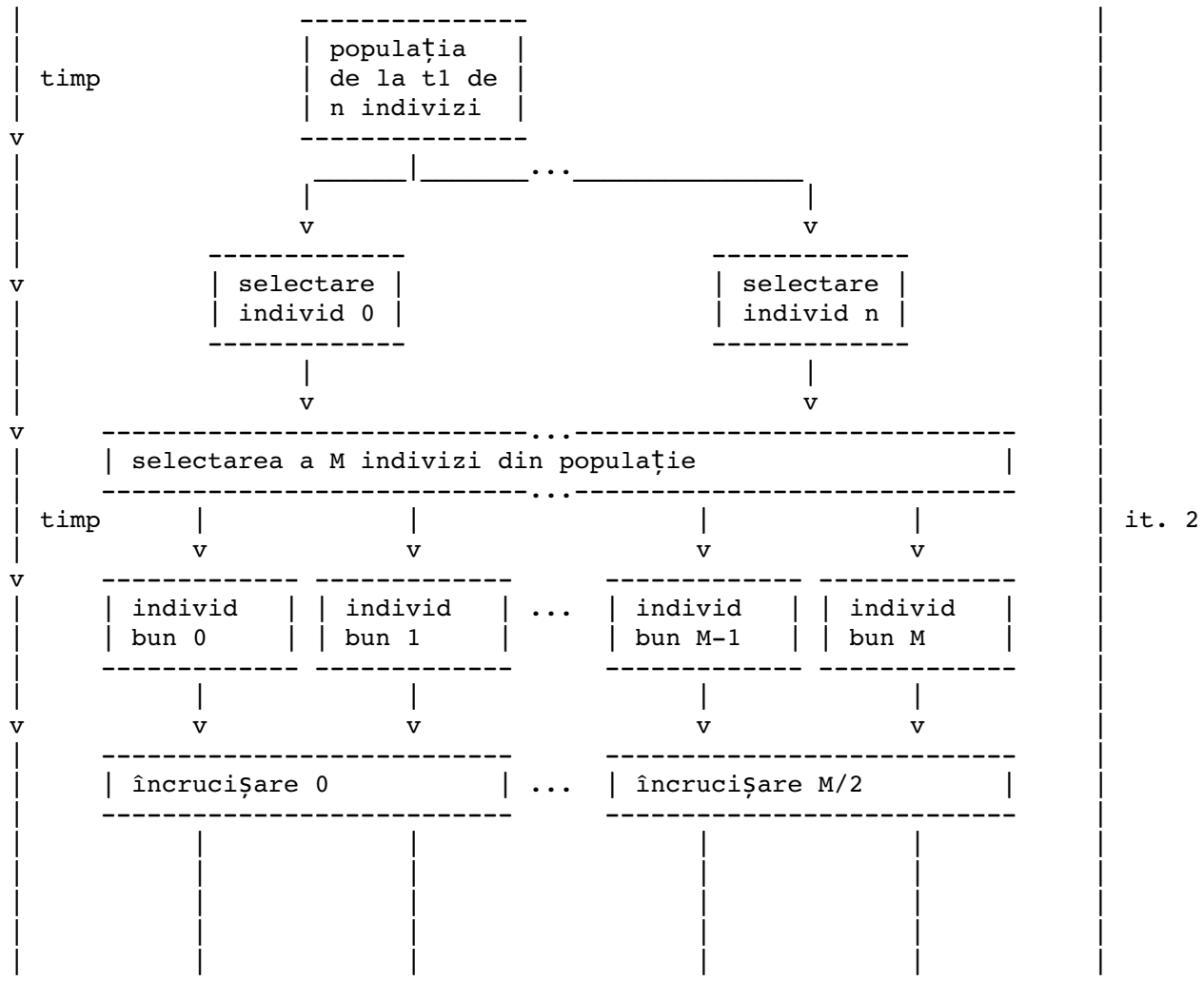
Exista o întreaga terminologie în teoria algoritmilor genetici ce include familiarele populație și individ, dar și gena, genotip, reprezentare genetica, fenotip, reprezentare fenotipica etc. În mod normal, trebuie un strat de adaptare a problemei la cerințele metodei. În cazul nostru, acest strat nu exista sau este foarte subțire, existând o apropiere foarte mare intre reprezentarea fenotipica (reprezentarea caracteristica problemei) și cea genetica (reprezentarea caracteristica Genetic Algorithm). În speța, atât problema cât și metoda vad un individ ca o lista de numere. Un număr (identificator al unui nod) este o "gena" pentru în individ și este unitatea atomica de lucru în procesele de combinare și mutație. Funcția de evaluarea a unui individ este, de asemenea, în ambele cazuri, funcția cost, cu însemnătatea ca un cost mai mic este asociat unui individ mai bun.

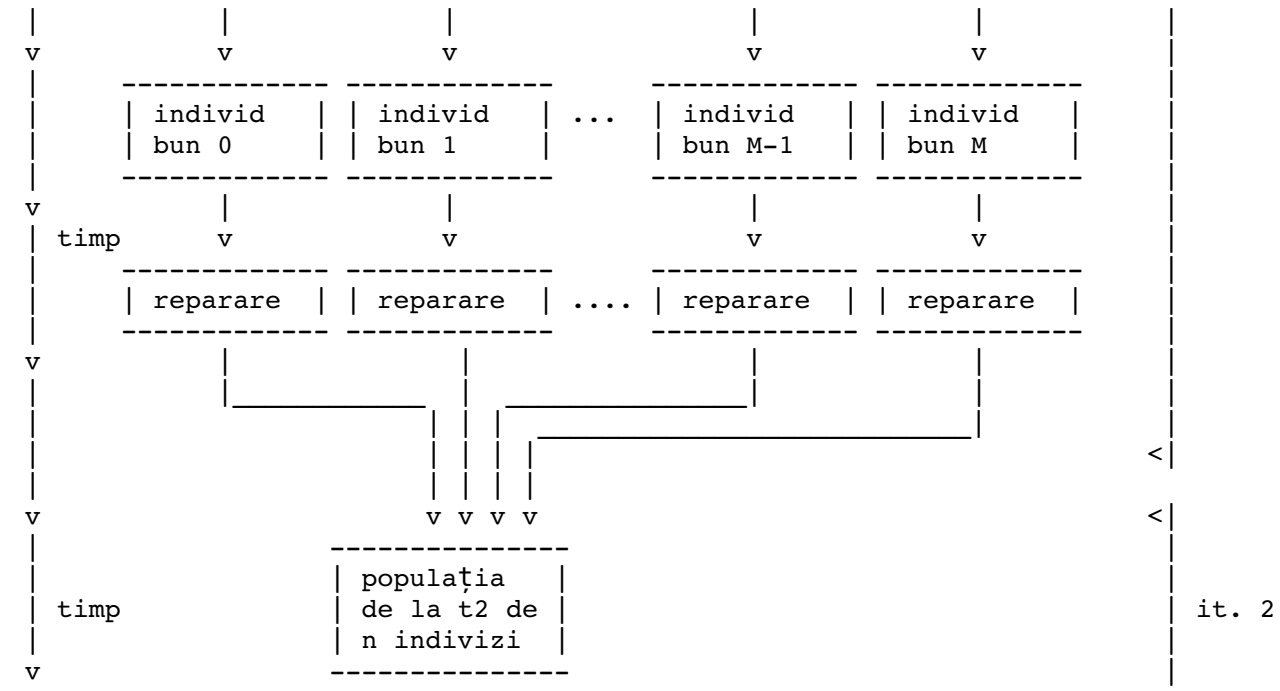
Ca și la Evolution Strategy, exista numeroase variante de Genetic Algorithm. Pe lângă variații în metoda de selectare și încrucișare a indivizilor "părinti", exista variante ce folosesc mai mult de doi părinti, variante care păstrează un număr de soluții foarte bune în orice populație (o practica numita în general elitism), variante care efectuează un mic Hill Climb pe fiecare soluție rezultata pentru a obține minimele locale din apropierea fiecărui individ etc.

Funcția de evoluție a populației se poate exprim astfel:

```
evolutiePopulatie(CP,N,M) (populație) =
   populatieNoua <- ()
   populatieParinti <- ga_selectie populație

   pentru i în (1,3,..,M)
        copil1 <- ga_incruciseaza populatieParinti_i populatieParinti_i+1
        copil2 <- ga_incruciseaza populatieParinti_i+1 populatieParinti
        adaugă populatieNoua (ra_reparare copil1)
        adaugă populatieNoua (ra_reparare copil2)

   sansaMutatie <- random 0 1

   dacă sansaMutatie < 0.1
   atunci
      idIndividMutat <- random 0 (||populatieNoua||-1)
      individMutat <- ra_mutatie (element populatieNoua idIndividMutat)
      schimba populatieNoua individMutata (ra_reparare individMutat)

   întoarce populatieNoua
```

Procesul de evoluție poate fi vizualizat astfel:





Putem observa ca procesul de evoluție depinde de doi operatori, de selecție și, respectiv, de încrucișare. Aceștia sunt ortogonali, așadar vom avea un număr mare de variante de Genetic Algorithm, depinzând de perechea de operator de selecție și de încrucișare folosita.

4.4.1.Selectia.

Operatorul de selecție generează o populație de "părinti", pornind de la populația curenta. În general, rolul selecției este de a da șansa de a se reproduce numai unor indivizi considerați "buni". Calitatea nu este data însa numai de cost, astfel, depinzând de metoda, pot fi selectați indivizi depărtați de optim, dar care aduc alte avantaje, precum diversitatea setului de gene.

Am studiat doua metode de selecție: Selecția Aleatoare (Random Selection) și Selecția prin Turnament (Tournament Selection). Prima dintre acestea formează setul de părinti selectând aleator M indivizi din populația curenta.

```
ga_selectie(CP,N,M) : SP_RAV(CP)^M -> SP_RAV(CP)^M
ga_selectie(CP,N,M) (populație) =
   părinti <- ()

   pentru i în (1,2,..,M)
      adaugă părinti (element populație (random 0 (||populație||-1)))

   întoarce părinti
```

Observam ca operatorul de selecție produce indivizi valizi, el nemodificant structura rețelelor de acces, ci doar rearanjând sau eliminând indivizi din populație.

A doua metoda, Torunament Selection, formează M grupuri de indivizi aleși aleator din populație, și păstrează în setul de părinti pe cel mai bun individ din fiecare grup (câștigătorul turnamentului). Mărimea grupului de turnament T este o caracteristica a metodei. Exemple de valori sunt 2,3,5,7,10 etc.

```
T : mărimea turnamentului
T ∈ N+

ga_selectie(CP,N,M) : SP_RAV(CP)^M -> SP_RAV(CP)^M
ga_selectie(CP,N,M) (populație) =
   părinti <- ()

   pentru i în (1,2,..,M)
      grupTurnament <- ()

      pentru j în (i,2,..T)
         adaugă grupTurnament (element populație (random 0 (||populație||-1)))

      adaugă părinti (individulDeCostMinim grupTurnament ())

   întoarce părinti
```

Aceasta metoda nu explorează asa de puternic spațiul de căutare, dar poate profita de cai spre minime locale atunci când le detectează. Mărimea turnamentului influențează raportul căutare/exploatare, iar pe măsura ce T tinde la M, selecția devine din ce în ce mai elitista, rămânând pentru T=M un grup de părinti consistând doar din cel mai bun individ.

4.4.2.Incrucisarea.

Procesul de încrucișare produce un individ nou pornind de la doi indivizi existenți. Modul în care se produce acest lucru este controlat de "masca de încrucișare". Știim ca modul de reprezentare atât pentru problema cât și genetic este o lista de numere. Fiecare element din lista de numere este o "gena". Încrucișarea nu reprezinta decât formarea unui nou material genetic prin selectarea de gene din fiecare reprezentare genetica în parte. Masca de încrucișare controlează procesul de producție a unui individ, specificând, pentru fiecare gena în parte părintele de la care provine.

Definirea operatorului de încrucișare este:

```
ga_incrucisare(CP,N,M) : SP_RAV(CP) -> SP_RAV(CP) -> SP_RA(CP)
ga_incrucisare(CP,N,M) (părinte1) (părinte2) =
   individNou <- ()
   masca <- ga_genereazaMasca (min ||părinte1|| ||părinte2||)

   pentru i în (1,2,..,(min ||părinte1|| ||părinte2||))
      dacă masca_i == 0
      atunci
         adaugă individNou <- element părinte1 i
      altfel
         adaugă indvidNou <- element părinte2 i

   dacă ||părinte1|| == min ||părinte1|| ||părinte2||
   atunci
      unește individNou (interval părinte2 ||părinte1|| ||părinte2||)
   altfel
      unește individNou (interval părinte1 ||părinte2|| ||părinte1||)

   întoarce individNou
```

Algoritmul copiază gene din părinti în individul nou sub controlul măștii. Cum exista șansa ca doi indivizi sa aibă lungimi ale listei de noduri diferite, consideram ca din nodul mai lung se ia și porțiunea de "coada" ce nu are echivalent în nodul mai scurt.

Și aici ca și la selecție nu se introduc gene noi, doar se rearanjează sau se elimina. Așadar, folosirea unui pas separat de mutație este justificata. Pe de alta parte, individul produs nu mai este valid. Prin schimbarea de noduri se pot produce indivizi ce nu conectează toți utilizatorii sau nu acoperă necesarul de trafic, conform procesului de evaluare.

Drept exemplu, considerând doi indivizi, ra1 și ra2, și o masca m, procesul de încrucișare produce următorul individ:

```
ra1 = (1,6,5,4)
ra2 = (2,2,2,3,5)

m = (0,1,1,0)

nou = ()

   inițial individul nou este gol. la primul
   pas al încrucișării se selectează prima
   gena din ra1 (conform valorii din masca : 0)

nou = (1)

   la al doilea și al treilea pas se selectează
   genele din al doilea părinte.

nou = (1,2,2)

   ultima gena determinata de masca provine de la
   părintele ra1.

nou = (1,2,2,4)

   în final se adauga "coada" lui ra2.

nou = (1,2,2,4,5)
```

După cum spuneam, exista mai multe metode de încrucișare, dar acestea diferă doar prin modul de generare al măștii, adică prin implementarea funcției ga_genereazaMasca. Am ales patru variante de implementat: One Point CrossOver, Two Point CrossOver, Single Point CrossOver si Uniform CrossOver.

One Point CrossOver partiționează masca într-o zona inițiala umpluta cu zero și restul măștii umpluta cu unu. Astfel, prima parte din genom provine de la primul părinte iar a doua de la al doilea. Alegerea punctului de partiționare se face aleator. Rezultatele obținute suferă din cauza ca intre "genele" reprezentării genetice exista o legătura. Ruperea și refacerea acestor legături afectează calitatea indivizilor.

Exemplu de masca generata de aceasta metoda este:

```
m = (0,0,0,1,1,1,1)
```

Two Point CrossOver partiționează masca în trei zone. Prima și ultima sunt umplute cu zero, iar cea din mijloc este umpluta cu unu. Alegerea punctelor de partiționare se face de asemenea aleator.

Exemplu de masca generata de aceasta metoda este:

```
m = (0,0,1,1,1,0,0)
```

Single Point CrosOver determina generarea unui individ nou, identic cu primul dintre părinti cu excepția ca o anumita gena, aleasa aleator, este obținuta de la celalalt părinte. Este o metoda oarecum similara cu mutația ce schimba un nod cu alt nod.

Exemplu de masca generata de aceasta metoda este:

```
m = (0,0,0,1,0,0,0)
```

Uniform CrossOver generează o masca aleator. Aceasta produce cele mai bune rezultate (lucru oarecum neintuitiv: strica foarte mult din structurile bune găsite, dar forțează o explorare puternica a spațiului).

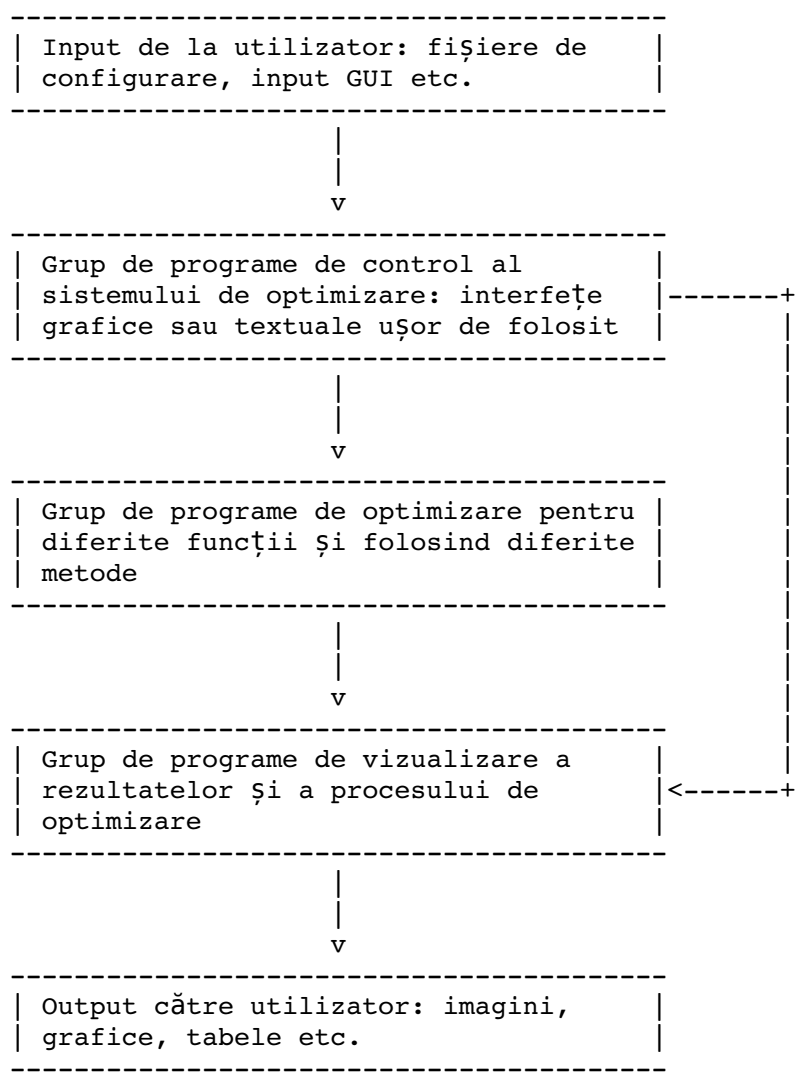
```
m = (0,1,1,0,1,0,0)
```

În total, avem doua metode de selecție și patru metode de încrucișare. Cum am spus ca acești operatori sunt ortogonali, putem combina fiecare operator de selecție cu fiecare operator de încrucișare, obținând în final 8 variante de Genetic Algorithm.

5.Sistemul de Optimizare.

Pana în acest moment am definit problema pe care încercam s-o rezolvam și un număr de metode pentru rezolvarea ei. În aceasta secțiune voi prezenta implementarea acestor metode ca un sistem software. Algoritmii descriși pana acum își găsesc implementarea, într-o forma mai mult sau mai putin asemănătoare cu cea prezentata, în acest sistem.

Sistemul de optimizare se prezintă ca un număr de programe ce implementează metodele de optimizare propriu-zise și un număr de programe ajutătoare, cu rol de control, de vizualizare a proceselor de optimizare sau de ușurare a interacțiunii. Programele de optimizare sunt implementate în C și compilate pentru sistemul de operare Linux rulând pe un PC obișnuit: o configurație destul de comuna. Momentan, software auxiliar exista doar pentru vizualizarea rezultatelor unei sesiuni de optimizare, și este scris în Tcl.

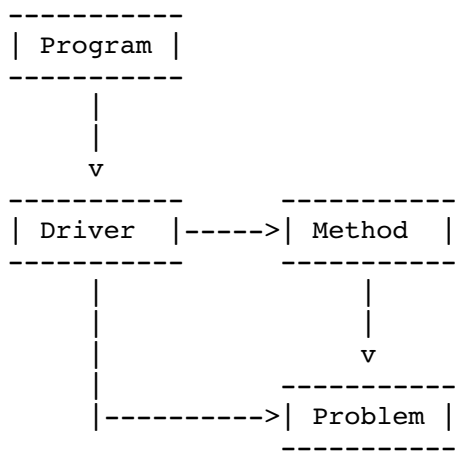


Vizualizare a diverselor componente ale sistemului de optimizare.

Partea cea mai importanta este suita de programe de optimizare. Exista o multitudine de variante de programe, fiecare specializat pe o anumita metoda de optimizare și o anumita problema. Sistemul în sine își propune sa fie mai general decât ce ar fi fost necesar pentru rezolvarea stricta a problemei. Astfel, exista trei tipuri de probleme rezolvate (optimizare de funcții reale 1 dimensionale, optimizare de funcții reale 2 dimensionale și optimizarea de retele de acces) și patru metode de optimizare, fiecare cu, posibil, mai multe variante . Este foarte ușoara extinderea atât din punct de vedere al problemelor cât și a metodelor, iar un număr de alegeri la nivelul implementării în C permit extinderea într-o anumita direcție independent de cealaltă. Astfel, definirea unei noi probleme nu presupune vreo schimbare în implementarea metodelor de optimizare, toate fiind direct folosibile. De asemenea, o noua metoda poate fi definita și folosita direct pe fiecare problema existenta.

Din punct de vedere structural fiecare program este alcătuit din trei mari module: modulul de comanda (Driver), modulul de optimizare (Method) și modulul de definire a problemei (Problem). Acesta din urma specifica reprezentarea unui individ și a unei configurații de problema. De asemenea, implementează un număr de operații asupra acestora precum generarea unei individ aleatoare, mutația sau încrucișarea. Modulul de optimizare definește reprezentarea stării interne a algoritmului de optimizare și operațiile specifice precum generarea unei populații inițiale, evoluția sau găsirea celui mai bun individ. Acest modul depinde de modulul de definire a problemei. Ultimul modul definește felul în care algoritmul de optimizare se executa pe sistemul gazda : dacă folosește mai multe fire paralele de execuție pe același calculator, dacă distribuie munca spre mai multe alte calculatoare într-un cluster, dacă procesele de optimizare executate în paralel suporta vreo forma de comunicare de indivizi intre ele (faptul ca pot comunica soluții din când în când aduce un avantaj în plus metodelor ce folosesc mai multe procese de execuție în paralel (pe lângă creșterea de viteza și explorarea unui spațiu mai mare) anume îmbunatățirea soluției găsite prin introducerea periodica de indivizi dintr-o cu totul alta populație). Acest modul se folosește de modulul de optimizare și modulul de definire a problemei.

În continuare este o reprezentare a structurii programului. Modulul Program conține partea organizatorica a definirii reale a unui program ce tine de operațiile specifice limbajului de programare folosit (C) și a sistemului ținta (Linux,PC x86).



Reprezentare idealizata a structurii unui program de optimizare.

Următoarele module au fost realizate:

```
  * Problem:
    * RealFunction1D: descrie funcții reale 1 dimensionale.
    * RealFunction2D: descrie funcții reale 2 dimensionale.
    * AccessNetwork: descrie retele de acces.
  * Method:
    * RandomSearch
    * HillClimbing
    * EvolutionStrategy
    * GeneticAlgorithm:
       * Selecție:
          * RandomSelection
          * TournamentSelection
       * Încrucișare:
          * OnePointCrossOver
          * TwoPointCrossOver
          * SinglePointCrossOver
          * UniformCrossOver
  * Driver
    * SingleThreadedDriver: cea mai simpla metoda de control - un singur proces de
optimizare cu execuție secvențială.
```

Deoarece tipurile de module sunt ortogonale, putem combina orice modul Driver cu orice module Method și orice modul Problem pentru a realiza 33 de programe separate de optimizare. Acest număr poate părea ridicat, iar pe măsura ce metode, probleme și driveri sunt adaugați acesta va creste, sporind temerile deja prezente. Aceasta structura pentru sistem și pentru programele de optimizare a fost aleasa intenționat, însa. Astfel, în loc de un program monolitic, folosim o sumedenie de programe simple. Per total, rezulta un sistem mai putin complex: fiecare program fie știe sa optimizeze o problema într-un mod specific (driver,metoda,problema), fie știe sa prelucreze datele de ieșire într-un anume mod (vizualizarea evoluției unui algoritm genetic pe o funcție reala 2 dimensionala de exemplu), fie știe sa lege într-un mod anume celelalte programe sau sa ofere o interfața de un anumit tip pentru acestea. Mai mult, modul de comunicare inter-program este foarte simplu (în general citirea datelor de pe canalul standard de intrare (stdin) și tipărirea lor pe canalul standard de ieșire (stdout)). Pentru reprezentarea datelor pentru comunicare/stocare se folosește serializare pe baza de text și descrieri în limbaje regulate pentru interiorul sistemului și fișiere de configurație flexibile, imagini (formate prin excelenta binare) și fișiere text complexe pentru exteriorul sistemului (comunicarea cu utilizatorul). Acestea sunt principii generale de buna practica în structurarea de sisteme software mari. Unul dintre principalele avantaje, pe lângă complexitatea mai scăzuta, este ca putem folosii tehnici de implementare adecvate problemei. Pentru programele de optimizare acest lucru se traduce prin folosirea de limbaje de nivel jos (C) cu control al execuției și memoriei strict, și testarea exhaustiva de stări de eroare la nivelul codului, împreuna cu alte teste de validitate (rulare prin programul valgrind de verificare a corectitudinii acceselor la memorie de exemplu). Pentru programele auxiliare se folosesc în schimb limbaje de nivel înalt și nu se pune accentul pe performanta sau corectitudine în execuție asa mare.

5.1.Modul de folosire.

Pentru a folosii un program de optimizare, trebuie sa îl invocam dintr-o line de comanda (când folosim direct ca utilizator) sau cu ajutorul unui alt program (când folosim prin intermediul unui program auxiliar). Indiferent de caz, toate programele de optimizare se vor aștepta sa poată citi o configurație de pe canalul standard de intrare (stdin) și vor tipării rezultatul pe canalul standard de ieșire (stdout).

Sa luam ca exemplu programul de optimizare a rețelei de acces, ce folosește metoda Evolution Strategy și driverul Single Threaded. Numele executabilului programului este: "opt-st-es-an". Convenția în general este ca numele programelor de optimizare este compus din trei elemente variabile și prefixul "opt", separate de "-". Astfel, după "opt", urmează un identificator pentru driverul ales ("st"), un identificator pentru metoda aleasa ("es") și un identificator pentru problema ("an").

Lista identificatorilor este următoarea:

```
* Problem:
  * RealFunction1D: rf1d
  * RealFunction2D: rf2d
  * AccessNetwork: an
* Method:
  * RandomSearch: rs
  * HillClimbing: hc
  * EvolutionStrategy: es
  * GeneticAlgorithm:
    * RandomSelection si OnePointCrossOver: ga-rs-opc
    * RandomSelection si TwoPointCrossOver: ga-rs-tpc
    * RandomSelection si SinglePointCrossOver: ga-rs-spc
    * RandomSelection și UniformCrossOVer: ga-rs-uc
    * TournamentSelection și OnePointCrossOver: ga-tos-opc
    * TournamentSelection și TwoPointCrossOver: ga-tos-tpc
    * TournamentSelection și SinglePointCrossOver: ga-tos-spc
    * TournamentSelection și UniformCrossOVer: ga-tos-uc
```

Revenind la exemplu, pentru cazul interacțiunii directe (folosind un shell de exemplu), programul este invocat ca oricare alt program/comanda :

```
./opt-st-es-an
```

Odată pornit acesta va aștepta descrierea unei configurații de la utilizator. Deoarece majoritatea configurațiilor sunt complexe, acestea sunt salvate în fișiere separate, iar conținutul lor este redirectat spre canalul de intrare al programului la momentul invocației. Astfel, de obicei, programul se se pornește astfel:

```
./opt-st-es-an < fișier_configurație
```

sau

```
cat fișier_configurație | ./opt-st-es-an
```

Pornit astfel, programul trece direct la execuția procesului de optimizare și, după un timp, rezultatul optimizării este afișat. Mai mult, din când în când sunt afișate și stări intermediare ale procesului. Deoarece ieșirea poate fi destul de mare, de obicei canalul de ieșire este redirectat spre un fișier.

```
./opt-st-es-an < fișier_configurație > fișier_ieșire
```

sau

```
cat fișier_configurație | ./opt-st-es-an > fișier_ieșire
```

Folosirea doar a canalelor stdin si stdout pentru comunicare cu lumea exterioara programului se

dovedește astfel avantajoasa. Utilizatorul are control flexibil asupra modului de comunicare: poate tasta direct configurația sau o poate salva într-un fișier si introduce de câte ori are nevoie, poate urmării direct procesul de optimizare sau îl poate salva pentru vizualizare ulterioara etc. De asemenea, dacă executabilul este invocat de către un alt program, acesta poate comunica ușor cu acesta, majoritatea metodelor de execuție a unui program extern din limbajele de nivel înalt dând acces ușor la canalele stdin si stdout (astfel configurația poate fi generata pe loc, si nu salvata în vreun fișier temporar etc.), nefiind, de asemenea, nevoie de mecanisme mai sofisticate de comunicare inter-proces (IPC).

Fișierul de configurare este special gândit pentru a fi ușor de parsat. Astfel, limbajul de descriere este unul regulat, ce este parsat doar cu ajutorul funcțiilor standard *scanf din C. Structural, fișierul de configurare este împărțit în 3 parți: configurarea pentru driver, configurarea pentru metoda și configurarea pentru problema. Fiecare dintre acestea este specific modulului folosit, așadar un fișier de configurare pentru un anume program de optimizare nu poate fi folosit pentru altul. Pentru exemplul nostru, fișierul de configurație poate avea următoarea forma:

```
SingleThreadedParams:          <|
  IterationsNr: 100              | porțiunea de configurare a
  StatePrintInterval: 50       <| modulului de control
EvolutionStrategyParams:       <|
  Miu: 2                         | porțiunea de configurare a
  Lambda: 4                    <| modulului de optimizare
AccessNetworkParams:           <|
  Name: Test_Problem             | porțiunea de configurare a
  NodesCnt: 6                    | modului de descriere a
  Nodes:                         | problemei
    Node:                        |
      Name: SW_01                |
      Cost: 100.0                |
      DownPortsNr: 4             |
      DownPort: 10 2             |
      UpPortsNr: 1               |
      UpPort: 20 2               |
    Node:                        |
      Name: SW_02                |
      Cost: 40                   |
      DownPortsNr: 2             |
      DownPort: 10 4             |
      UpPortsNr: 1               |
      UpPort: 20 2               |
    Node:                        |
      Name: SW_03                |
      Cost: 150                  |
      DownPortsNr: 8             |
      DownPort: 10 1             |
      UpPortsNr: 2               |
      UpPort: 100 10             |
    Node:                        |
      Name: SW_04                |
      Cost: 180                  |
      DownPortsNr: 4             |
      DownPort: 12 4             |
      UpPortsNr: 1               |
      UpPort: 40 10              |
    Node:                        |
```

```
    Name: SW_05                 |
    Cost: 250                   |
    DownPortsNr: 4              |
    DownPort: 20 10             |
    UpPortsNr: 1                |
    UpPort: 100 30              |
  Node:                         |
    Name: SW_06                 |
    Cost: 500                   |
    DownPortsNr: 4              |
    DownPort: 40 40             |
    UpPortsNr: 1                |
    UpPort: 150 150             |
UsersCnt: 12                    |
Users:                          |
  User:                         |
    Name: User_01               |
    Speed: 1.024 0.128          |
  User:                         |
    Name: User_02               |
    Speed: 0.512 0.128          |
  User:                         |
    Name: User_03               |
    Speed: 0.512 0.064          |
  User:                         |
    Name: User_04               |
    Speed: 1.024 0.256          |
  User:                         |
    Name: User_05               |
    Speed: 2.048 1.024          |
  User:                         |
    Name: User_06               |
    Speed: 0.512 0.128          |
  User:                         |
    Name: User_07               |
    Speed: 0.512 0.256          |
  User:                         |
    Name: User_08               |
    Speed: 1.024 0.256          |
  User:                         |
    Name: User_09               |
    Speed: 1.024 0.128          |
  User:                         |
    Name: User_10               |
    Speed: 0.512 0.512          |
  User:                         |
    Name: User_11               |
    Speed: 0.128 0.064          |
  User:                         |
    Name: User_12               |
    Speed: 0.256 0.128         <|
```

Structura descrierilor pare ierarhica, dar, datorita faptului ca adâncimea copacului construit este limitata, iar ordinea și conținutul fiecărui element structurat este bine stabilit, limbajul este unul regulat, și, deci, simplu de parsat, iar, ca exemplu, metoda de parsare din fiecare program de optimizare se limitează la a folosii funcții din familia de funcții standard din C, scanf (deci, nici măcar pe expresii regulate). De asemenea, este de observat ca în descrierea problemei, numărul de tipuri de noduri și

utilizatori este specificat dinainte. Acest lucru simplifica din nou procesul de parsare a configurației.

Prin variarea parametrilor, adăugarea de noduri sau utilizatori noi etc. se pot obține diverse instante ale problemei noastre și diverse moduri de a o rezolva.

În continuare este prezentat rezultatul execuției programului de mai sus. Am omis o retipărire a configurației ce prezenta modul în care programul a înțeles fișierul de configurare (se adăugau un număr mic de informații suplimentare, dar, în principiu, rolul retipăririi era ca validare pentru utilizator ca fișierul de configurație a fost înțeles cum trebuie). De asemenea, rezultatele pot diferi de la execuție la execuție, algoritmii conținând o componenta aleatoare pana la urma.

```
SingleThreadedState:                  <|
  Iteration: 0                          | starea inițiala a programului
  HillClimbingState:                    | de optimizare (suma stărilor
    Iteration: 0                        | modulelor Driver,Method si
    ProblemStatesCnt: 4                 | Problem)
    ProblemStates:                      |
      AccessNetworkState:               |
        NodeIdsUsedCnt: 5               |
        NodeIdsCnt: 12                  |
        NodesIds: 4 1 3 1 3             |
        Cost: 690.000                   |
      AccessNetworkState:               |
        NodeIdsUsedCnt: 5               |
        NodeIdsCnt: 12                  |
        NodesIds: 0 1 0 3 3             |
        Cost: 600.000                   |
      AccessNetworkState:               |
        NodeIdsUsedCnt: 4               |
        NodeIdsCnt: 12                  |
        NodesIds: 3 3 2 4               |
        Cost: 760.000                   |
      AccessNetworkState:               |
        NodeIdsUsedCnt: 7               |
        NodeIdsCnt: 12                  |
        NodesIds: 4 1 0 0 1 1 5         |
        Cost: 1070.000                  |
  AccessNetworkState:                 <<|
    NodeIdsUsedCnt: 5                   |
    NodeIdsCnt: 12                      | cel mai bun individ întâlnit.
    NodesIds: 0 1 0 3 3                 |
    Cost: 600.000                     <|
SingleThreadedState:                  <|
  Iteration: 49                         | starea programului de
  HillClimbingState:                    | optimizare la a 50-ea iterație.
    Iteration: 49                       |
    ProblemStatesCnt: 4                 |
    ProblemStates:                      |
      AccessNetworkState:               |
        NodeIdsUsedCnt: 5               |
        NodeIdsCnt: 12                  |
        NodesIds: 3 1 4 3 5             |
        Cost: 1150.000                  |
      AccessNetworkState:               |
        NodeIdsUsedCnt: 4               |
        NodeIdsCnt: 12                  |
```

```
        NodesIds: 4 1 2 3                |
        Cost: 620.000                    |
      AccessNetworkState:                |
        NodeIdsUsedCnt: 4                |
        NodeIdsCnt: 12                   |
        NodesIds: 3 3 2 5                |
        Cost: 1010.000                   |
      AccessNetworkState:                |
        NodeIdsUsedCnt: 4                |
        NodeIdsCnt: 12                   |
        NodesIds: 3 3 4 4                |
        Cost: 860.000                    |
  AccessNetworkState:                  <<|
    NodeIdsUsedCnt: 5                    |
    NodeIdsCnt: 12                       | cel mai bun individ întâlnit.
    NodesIds: 1 0 0 1 3                  |
    Cost: 460.000                     <  |
SingleThreadedState:                  <  |
  Iteration: 99                          |
  HillClimbingState:                     | starea finala a programului de
    Iteration: 99                        | optimizare.
    ProblemStatesCnt: 4                  |
    ProblemStates:                       |
      AccessNetworkState:                |
        NodeIdsUsedCnt: 4                |
        NodeIdsCnt: 12                   |
        NodesIds: 3 3 4 3                |
        Cost: 790.000                    |
      AccessNetworkState:                |
        NodeIdsUsedCnt: 4                |
        NodeIdsCnt: 12                   |
        NodesIds: 4 3 3 4                |
        Cost: 860.000                    |
      AccessNetworkState:                |
        NodeIdsUsedCnt: 4                |
        NodeIdsCnt: 12                   |
        NodesIds: 4 3 4 4                |
        Cost: 930.000                    |
      AccessNetworkState:                |
        NodeIdsUsedCnt: 5                |
        NodeIdsCnt: 12                   |
        NodesIds: 4 1 0 3 4              |
        Cost: 820.000                    |
  AccessNetworkState:                  <<|
    NodeIdsUsedCnt: 5                    |
    NodeIdsCnt: 12                       | cel mai bun individ întâlnit
    NodesIds: 1 0 0 1 3                  | și rezultatul optimizării.
    Cost: 460.000                     <  |
```

6.Experimente.

În aceasta secțiune voi prezenta rezultatele testării implementării pe doua instante de problema. Prima dintre acestea este exemplul de mici dimensiuni cu care am lucrat pana acum. A doua, însa, este o problema de dimensiuni mari, apropiate de ce s-ar putea întâlnii în medii de producție.

Ca și testare, rulam cele 11 variante de programe de optimizare pentru aceasta problema. Fiecare va porni folosind aceeași "sămânța" pentru generatorul de numere aleatoare și fiecare ca avea mărime a populației egala. Aceste constrângeri asigura ca populația inițiala este identica pentru toate implementările și ca metodele vor fi egale din punct de vedere al posibilităților de lucru cu populația. Pentru găsirea timpului mediu de execuție, fiecare program este rulat de 50 de ori. Folosirea aceleiași semințe se asigura ca toate cele 50 de rulări ale unei probleme vor avea același rezultat. Exista un singur driver implementat momentan, anume SingleThreadedDriver. Același set de parametrii sunt folosiți pentru acesta pentru toate metodele de rezolvare.

```
Parametrii SingleThreadedDriver:

  Număr de Iterații: 100
  Interval Printare a Stării: 50 iterații
```

Metoda	Parametrii Metoda	Nume Program	Optim Găsit	Timp
Random Search	Numar de Stari: 10	opt-st-rs-an	n1 n2 n1 n2 n4 / 460	00:00
Hill Climbing	Factor de Explorare: 4 Numar de Stari: 10	opt-st-hc-an	n2 n1 n1 n2 n4 / 460	00:00
Evolution Strategy	Miu: 5 Lambda: 10	opt-st-es-an	n2 n2 n1 n1 n4 / 460	00:00
GeneticAlgorithm RandomSelection OnePointCrossOver	Sansa Mutatie: 15 Numar de Stari: 10	opt-st-ga-rs-opc-an	n1 n2 n4 n1 n4 / 540	00:00
GeneticAlgorithm RandomSelection TwoPointCrossOver	Sansa Mutație: 15 Număr de Stări: 10	opt-st-ga-rs-tpc-an	n1 n2 n2 n1 n3 / 460	00:00
GeneticAlgorithm RandomSelection SinglePointCrossOver	Șansa Mutație: 15 Număr de Stări: 10	opt-st-ga-rs-spc-an	n1 n2 n1 n2 n4 / 460	00:00
GeneticAlgorith RandomSelection UniformCrossOver	Șansa Mutație: 15 Număr de Stări: 10	opt-st-ga-rs-upc-an	n1 n2 n1 n2 n4 / 460	00:00
GeneticAlgorithm TournamentSelection OnePointCrossOver	Șansa Mutație: 15 Număr de Stări: 10 Mărime Turnament: 4	opt-st-ga-tos-opc-an	n1 n2 n4 n1 n4 / 540	00:00
GeneticAlgorithm TournamentSelection TwoPointCrossOver	Șansa Mutație: 15 Număr de Stări: 10 Mărime Turnament: 4	opt-st-ga-tos-tpc-an	n1 n4 n1 n4 / 560	00:00
GeneticAlgorithm TournamentSelection SinglePointCrossOver	Șansa Mutație: 15 Număr de Stări: 10 Mărime Turnament: 4	opt-st-ga-tos-spc-an	n1 n2 n3 n4 / 470	00:00
GeneticAlgorith TournamentSelection UniformCrossOver	Șansa Mutație: 15 Număr de Stări: 10 Mărime Turnament: 4	opt-st-ga-tos-upc-an	n1 n4 n2 n2 n4 / 540	00:00

În final, pentru o problema cu 1600 de utilizatori și folosind echipamente similare cu cele de la problema anterioara (doar niște noduri mai puternice au fost adăugate) s-au obținut următoarele rezultate:

```
  Număr de Iterații: 100
  Interval Printare a Stării: 50 iterații
```

Metoda	Parametrii Metoda	Nume Program	Optim Găsit	Timp
Random Search	Număr de Stări: 2000	opt-st-rs-an	269800	04:08
Hill Climbing	Factor de Explorare: 5 Număr de Stări: 2000	opt-st-hc-an	251490	04:59
Evolution Strategy	Miu: 400 Lambda: 2000	opt-st-es-an	222340	01:27
GeneticAlgorithm RandomSelection OnePointCrossOver	Șansa Mutație: 15 Număr de Stări: 2000	opt-st-ga-rs-opc-an	280640	01:15
GeneticAlgorithm RandomSelection TwoPointCrossOver	Șansa Mutație: 15 Număr de Stări: 2000	opt-st-ga-rs-tpc-an	280640	01:31
GeneticAlgorithm RandomSelection SinglePointCrossOver	Șansa Mutație: 15 Număr de Stări: 2000	opt-st-ga-rs-spc-an	280640	01:29
GeneticAlgorith RandomSelection UniformCrossOver	Șansa Mutație: 15 Număr de Stări: 2000	opt-st-ga-rs-uc-an	280640	01:20
GeneticAlgorithm TournamentSelection OnePointCrossOver	Șansa Mutație: 15 Număr de Stări: 2000 Mărime Turnament: 5	opt-st-ga-tos-opc-an	166500	01:19
GeneticAlgorithm TournamentSelection TwoPointCrossOver	Șansa Mutație: 15 Număr de Stări: 2000 Mărime Turnament: 5	opt-st-ga-tos-tpc-an	155250	01:24
GeneticAlgorithm TournamentSelection SinglePointCrossOver	Șansa Mutație: 15 Număr de Stări: 2000 Mărime Turnament: 5	opt-st-ga-tos-spc-an	264670	01:19
GeneticAlgorith TournamentSelection UniformCrossOver	Șansa Mutație: 15 Număr de Stări: 2000 Mărime Turnament: 5	opt-st-ga-tos-uc-an	76160	01:34

Individul optim a fost gasit de metoda Genetic Algorithm folosind Tournament Selection si Uniform CrossOver. Acesta era un rezultat asteptat. Celelalte metode au fost in general de doua pana la patru ori mai slabe in privinta individului gasit.

Forma individului optim este urmatoare:

```
  n1 n1 n3 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n4 n1 n1 n1 n1 n1 n1 n1 n1
n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1
n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1
n1 n1 n2 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1
n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1
n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1
n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1
n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1
n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1
n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1
```

```
n1 n1 n2 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1
n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1
n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1
n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1 n1
n1 n1 n1 n1 n1 n1 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4
n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 4 n4 n4 n4 n4 n4 n4 n4
n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4
n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n4 n5 n5 n5 n5 n5 n5 n5
n5 n5 n5 n5 n5 n5 n5 n5 n5 n5 n5 n5 n5 n5 n5 n5 n5 n7 n7 n7 n7 n7 n7 n8 n8 n8 n8
n8 n8
```

Nodurile n7 si n8 sunt specifice instantei mari de problema si sunt descrise de:

```
Node:
  Name: SW_07
  Cost: 1200
  DownPortsNr: 4
  DownPort: 100 100
  UpPortsNr: 1
  UpPort: 1000 1000
Node:
  Name: SW_08
  Cost: 1200
  DownPortsNr: 2
  DownPort: 1000 1000
  UpPortsNr: 1
  UpPort: 1000 1000
```

7.Concluzii.

În aceasta lucrare am descris o problema de optimizare matematica și un sistem pentru rezolvarea acesteia. Firește, sunt încă multe îmbunatățiri de adus sistemului, iar în acest capitol voi vorbii despre unele dintre acestea.

Prima adăugare, și discutabil, cea mai importanta, este extinderea definiției problemei. Momentan, nu se iau în considerare tipurile sau costurile conexiunilor ce leagă nodurile, utilizatorii și resursa. Aceste lucruri afectează procesul de construire fizica a rețelei, iar informații privind delimitarea cea mai buna a zonelor cu acoperire de fibra optica sau cu acoperire de cablu de cupru pot ajuta. Mai mult, selectarea tipurilor de legături poate suferi de constrângeri în plus. De exemplu: conexiuni intre anumite noduri se pot realiza doar prin fibra deoarece nodurile poseda doar astfel de conexiuni sau, ca regula generala, legăturile folosind fibra optica trebuie sa fie mai apropiate de rădăcina, cât timp cele ce folosesc cablu de cupru trebuie sa fie mai aproape de utilizatori. Introducerea conceptelor de cost al conexiunii impune introducerea și a noțiunii de poziție: pentru a putea calcula costul unei legături (pentru care avem doar metrica de "cost per metru") trebuie sa știm distanta intre doua entități conectate. Acest lucru presupune extinderea definiției utilizatorului și a individului în sensul includerii de informații de poziție. Alegerea poziției, ca și alegerea nodurilor și a tipurilor de legături, trebuie sa fie făcuta astfel încât costul final al rețelei sa fie cât mai mic, deci și acest parametru trebuie "optimizat". Din fericire, utilizatorii au poziții cunoscute, iar nodurile pot fi plasate doar în poziții fixe numite "cabinete", astfel, spațiul de căutare rămâne de o dimensiune cu care putem lucra.

O alta adăugare este optimizarea pornind de la o rețea deja existenta. Aceasta problema modelează situația reala în care trebuie sa adaugăm niște noduri la o rețea deja existenta (și asupra căreia nu putem opera) pentru acoperirea de noi utilizatori. Aceasta problema prezintă dificultăți deoarece introducerea de noi noduri trebuie făcuta considerând limitările reprezentării lineare a unei retele.

Pornind de jos în sus, pe partea de implementare, prima extindere se referă la adăugarea de metode noi de optimizare. Candidați imediați sunt variante ale metodelor deja studiate și tehnici precum Simulated Annealing sau Tabu Search. De asemenea, un driver ce profita de capabilitățile de multiprocesoare ale PC-urilor recente (sisteme multicore / multiprocesor) pare în categoria îmbunatățirilor simplu de introdus. Modele mai complexe de control precum Island Models sau paralelism la nivel de metoda sau problema pot urma după implementarea acesteia. În final, pentru rezolvarea de instante foarte mari de problema, se pot realiza sisteme ce se executa pe clustere de calculatoare. Pe partea de control și vizualizare se poate lucra mult, iar ca proiect imediat, am realizarea unei interfețe web pentru sistem, ce permite mai multor utilizatori sa optimizeze diverse instante pe un sistem dedicat (server singur sau cluster). De asemenea, sisteme de vizualizare și comanda pot fi realizate fără a afecta dezvoltarea programelor de optimizat, fiind candidați buni pentru includere.

În final, sistemul produce rezultate promițătoare pe problemele de test, reușind sa găsească soluții bune (vizibil diferite de cele generate doar cu Random Search) în intervale de timp acceptabile (mai putin de 5 minute, ținând cont ca testele au fost făcute pe un sistem vechi (Pentium 4) și fără optimizări serioase (nici măcar din partea compilatorului)). Este desigur mult loc de îmbunatățiri, dar o baza solida, atât conceptual cât și din punct de vedere al implementării a fost realizata.

Anexa 1 : Dimensiunea spațiului de căutare.

În aceasta anexa prezentam un calcul al dimensiunii spațiului de căutare SP_RA(CP). Dimensiunea acestuia depinde de numărul de noduri folosibile (||CP:N||) și numărul de utilizatori (||CP:U||). Cunoscând ca o rețea de acces este reprezentata ca o lista de numere, și ca aceasta poate varia de la 1 la o mărime maxima, putem exprima spațiului astfel:

```
dimMax : mărimea maxima a unei soluții
dimMax ∈ N+

SP_RA(CP) = reuniune {1,2,...,||CP:N||}^i pentru i de la 1 la dimMax
```

Putem exprima mărimea spațiului astfel:

```
||SP_RA(CP)|| = suma ||{1,2,...,||CP:N||}^i|| pentru i de la 1 la dimMax
```

Cunoaștem ca pentru o mulțime cu x elemente, produsul cartezian de puterea i are x^i elemente. Putem rescrie expresia de mai sus astfel:

```
||SP_RA(CP)|| = suma ||CP:N||^i pentru i de la 1 la dimMax
```

Ce rămâne de demonstrat acum este ca dimMax este finit si ca poate fi exprimat în funcție de parametrii ||CP:N|| si ||CP:U|| ai configurației de problema. Considerând ca fiecare nod folosibil are strict mai puține porturi uplink decât downlink, si ca în procesul de construcție prezentat vrem sa folosim cât mai multe din porturile unui nod, atunci este intuitiv ca, pornind de la rădăcina si construind spre frunze, se va obține un copac finit. Cu alte cuvinte, la fiecare nivel al rețelei, pornind de la resursa spre utilizatori, frontul de porturi downlink creste. Invers, pornind de la utilizatori spre resursa, frontul de porturi uplink creste. Prin front înțelegem totalitatea porturilor de pe un nivel, fie ele uplink sau downlink. Evident, dacă un nod are mai multe noduri uplink decât downlink si este folosit de multe ori într-un nivel, frontul nu mai creste spre utilizatori si rețeaua poate fi construita indefinit. Același lucru se întâmpla si dacă nu impunem ca nodurile să-și folosească la maxim porturile.

Pentru calculul dimensiunii maxime a reprezentării unei retele trebuie sa găsim numărul maxim de noduri dintr-o rețea. Ca exemplu, presupunem ca avem în configurația problemei un singur nod definit, cu 4 porturi downlink și unul uplink. Avem de asemenea 64 utilizatori de acoperit pentru care ignoram, momentan, cerințele de trafic.

Primul nivel al rețelei are 64 de utilizatori de deservit și folosește 16 noduri pentru aceasta, introducând în nivelul următor 16 porturi. Al doilea nivel folosește doar 4 noduri pentru deservirea celor 16 noduri, iar ultimul nivel folosește un singur nod. În total avem 16 + 4 + 1 = 21 de noduri.

```
64 <----------- utilizatori.

64/4 = 16 <---- primul nivel din rețea.

16/4 = 4 <----- al doilea nivel din rețea.

4/4 = 1 <------ ultimul nivel din rețea.
```

În general, pentru ||CP:U|| utilizatori și un nod cu D porturi de downlink și 1 port de uplink, putem exprima numărul de noduri ce compun rețeaua astfel:

$$nrNod = \frac{|CP:U|}{D} + \frac{|CP:U|}{D^2} + |CP:U|/D^3 + ... + |CP:U|/D^t$$

Fiecare nivel scade numărul de noduri de D ori, pana când se ajunge la un singur nod pe ultimul nivel. Numărul t este indicele ultimului nivel și trebuie calculat astfel încât:

```
||CP:U||/D^t = 1 => <- pe ultimul nivel trebuie sa fie un singur nod.

||CP:U|| = D^t =>

D^t = ||CP:U|| =>

t = logD (||CP:U||)
```

Putem rescrie nrNod, ținând seama ca este o suma de termeni ai unei serii geometrice:

```
nrNod = ||CP:U||*(1/D + (1/D)^2 + ... + (1/D)^t)
      = ||CP:U||*(1 - (1/D)^(t+1)) / (1 - (1/D))
      = D*||CP:U||*(1 - (1/D)^(t+1)) / (D - 1)
      = D*||CP:U||*(1 - (1/D)^(logD(||CP:U||)+1)) / (D - 1)
```

Deoarece nrNod este pana la urma un întreg, valoarea finala este obținuta prin aproximare în sus:

```
nrNod = ceil D*||CP:U||*(1 - (1/D)^(logD(||CP:U||)+1)) / (D - 1)
```

Dacă nodul folosit introduce U porturi uplink, trebuie sa revizuim formulele de calcul. La fiecare nivel numărul de noduri scade de D ori dar numărul de porturi introduse scade doar de D/U ori (este multiplicat cu U/D).

```
nrNod = ||CP:U||/D + ||CP:U||*U/D^2 + ... + ||CP:U||*U^(t-1)/D^t
```

Observam ca numărul de noduri de pe primul nivel este ||CP:U||/D. Se introduc U*(||CP:U||/D) porturi uplink pe următorul nivel, iar numărul de noduri de pe acesta este de D ori mai mic decât numărul de porturi de pe primul nivel : U*(||CP:U||/D)/D.

În continuare, efectuam aceleași calcule ca mai înainte.

```
||CP:U||*U^(t-1)/D^t = 1 =>
||CP:U||/U = (D/T)^t =>
(D/T)^t = ||CP:U||/U =>
t = log(D/T) (||CP:U||/U)

nrNod = ||CP:U||*(1/D + U/D^2 + U^2/D^3 + ... + U^(t-1)/D^t)
      = ||CP:U||/U*(U/D + (U/D)^2 + ... (U/D)^t)
      = (||CP:U||/U)*(1 - (U/D)^(t+1))/(1 - U/D)
      = (D/U) * ||CP:U|| * (1 - (U/D)^(log(D/T)(||CP:U||/U) + 1)) / (D - U)
```

În final:

```
   nrNod = ceil (D/U) * ||CP:U|| * (1 - (U/D)^(log(D/T)(||CP:U||/U) + 1)) / (D -
U)
```

Desigur, aceste formule sunt valabile pentru cazul în care avem un singur nod în configurația de problema. Cazul interesant este când avem mai multi. În aceasta situație, rețeaua de dimensiune maxima se selectează alegând pe D ca numărul de porturi downlink minim dintre toate nodurile definite (acoperind cazul unei retele construite doar din acel nod care își extinde fronturile de porturi intre nivele spre utilizatori greu), și alegând pe U ca numărul de porturi uplink maxim dintre toate nodurile definite (acoperind cazul unei retele construite doar din acel nod care își extinde fronturile de porturi intre nivele spre resursa greu).

Așadar numărul de noduri maxim într-o rețea este:

```
   dimMax = ceil (D/U) * ||CP:U|| * (1 - (U/D)^(log(D/T)(||CP:U||/U) + 1)) / (D -
U)
     unde D : numărul minim de porturi downlink dintre toate nodurile definite.
        U : numărul maxim de porturi uplink dintre toate nodurile definite.
```

Cunoscând acest număr, dependent după cum spuneam de configurația de problema, putem spune ca într-adevăr SP_RA(CP) este finit, iar dimensiunea sa este calculabila conform expresiei:

```
  ||SP_RA(CP)|| = suma ||CP:N||^i pentru i de la 1 la dimMax
```

Dimensiunea spațiului SP_RA(CP) este evident dependenta într-un mod exponențial de configurația de problema. Acest fapt validează presupunerile anterioare și decizia de a folosii metode de optimizare.