EÖTVÖS LORÁND TUDOMÁNYEGYETEM
INFORMATIKAI KAR
ALGORITMUSOK ÉS ALKALMAZÁSAIK TANSZÉK

Heurisztikus algoritmusok az utazóügynök feladatra

Szakdolgozat

Témavezető:
Kovács Péter
Doktorandusz hallgató

Készítette:
Varga Gábor
Programtervező informatikus BSc
nappali tagozat

Budapest, 2010

Tartalomjegyzék

1. Bevezetés

1.1. Az utazóügynök probléma

Az utazóügynök feladat (Traveling Salesman problem, TSP) egy kombinatorikus optimalizálási probléma. Adott városok egy halmaza és páronként az egymástól való távolságuk. A feladat az, hogy meghatározzuk a legrövidebb túrát, amely minden várost pontosan egyszer érint.

Ezt a feladatot 1930-ban formalizálták először matematikai problémaként, és azóta is az egyik legismertebb és legintenzívebben kutatott optimalizálási probléma. Eredeti formájában is számos területen alkalmazzák, mint például a tervezés, logisztika és a mikrochipgyártás. Valamint kissé módosítva előfordul részproblémaként sok területen, például a DNS szekvencia meghatározásában. Ezeken a területeken a városok máshogy jelennek meg: vásárlókat, forrasztási pontokat vagy DNS töredékeket jelentenek. Az egyes területeken további megszorításokat is hozzávesznek, mint például a erőforrás- vagy időkorlátozás, és ezek még nehezebbé teszik a megoldás megtalálását.

Az utazóügynök probléma egy közismerten nehéz kombinatorikus optimalizálási feladat, gyakran használják különböző optimalizálási módszerek teljesítményméréséhez is. Eldöntési problémaként megfogalmazva az NP-teljes problémaosztályba tartozik, tehát nem ismert hatékony (polinomiális) megoldási módszere. Minden egzakt algoritmusra létezik egy olyan legrosszabb eset, amikor az algoritmus futási ideje exponenciálisan növekszik a városok számával [1]. Az egyszerűbb módszerek már 20-30 város esetén is használhatatlanná válnak, ezért sokféle heurisztikus módszert is kidolgoztak, amelyekkel bizonyos gyakorlati esetekben akár több tízezer város esetén is megoldható a probléma.

1.2. A dolgozat célkitűzése

Jelen dolgozat célkitűzése a klasszikusnak számító heurisztikus algoritmusok minél hatékonyabb implementálása a szimmetrikus és metrikus (lásd az 1.5. és 1.6. alfejezeteket) utazóügynök feladatra, valamint ezek összehasonlító elemzése. Az algoritmusokat a LEMON C++ hálózattervezési és optimalizálási programkönyvtár [5] felhasználásával valósítottuk meg, és a következő, 1.3-as kiadásának várhatóan részét képzik majd.

1.3. Mérföldkövek a TSP számolásában

1962-ben Held és Karp kifejlesztettek egy algoritmust, amely mai napig legkisebb futásidő garanciát adja a TSP megoldására. Ez a $O(n^2 2^n)$ korlát több mint 40 évig kiállta a próbát.

A jobb garanciát adó algoritmusok fejlődésének hiánya kiábrándító volt, de Edmonds azt tanácsolta a kutatóknak, hogy ne ragadjanak le a merev feltételeknél a gyakorlati al-

kalmazások fejlesztése során. A kutatói társadalom nem volt megelégedve az $O(n^2 2^n)$-es korláttal, Dantziggal kezdve a specifikus TSP esetekre fokuszáltak. Ennek a könnyen észrevehető jele az, hogy az évek során egyre nagyobb problémákat oldottak meg. A fontosabb mérföldköveket az 1. táblázatban láthatjuk a kutatást vezetők nevével és a problémák TSPLIB-ben használt nevével. Az elmúlt 50 év számos problémája megtalálható a TSPLIB könyvtárban [4]. A TSPLIB olyan gráfok gyűjteménye, amelyek tipikusan az utazóügynök problémánál vetődtek fel.

Év	Kutatók	Méret	TSPLIB
1954	G. Dantzig, R. Fulkerson, S. Johnson	49 város	dantzig42
1971	M. Held, R. M. Karp	57 város	-
1971	M. Held, R. M. Karp	64 város	véletlen pontok
1975	P. M. Camerini, L. Fratta, F. Maffioli	67 város	véletlen pontok
1975	P. Miliotis	80 város	véletlen pontok
1977	M. Grötschel	120 város	gr120
1980	H. Crowder, M. W. Padberg	318 város	lin318
1987	M. Padberg, G. Rinaldi	532 város	att532
1987	M. Grötschel, O. Holland	666 város	gr666
1987	M. Padberg, G. Rinaldi	1 002 város	pr1002
1987	M. Padberg, G. Rinaldi	2 392 város	pr2392

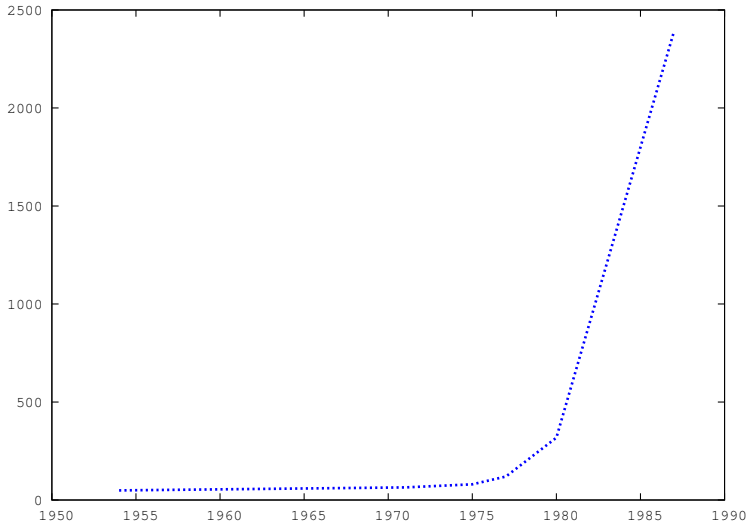
1. táblázat. Megoldott TSP feladatok mérete

Az eredeti 49 városból álló probléma USA 48 tagállamának fővárosait (Alaska és Hawaii csak 1959-ben csatlakoztak) és Washingtont jelentette. Dantzig, Fulkerson és Johnson egy 42 városból álló problémát oldottak meg Baltimore, Wilmington, Philadelphia, Newark, New York, Hartford és Providence városok eltávolításával. Végül úgy alakult, hogy a 42 városból álló probléma megoldása áthaladt az eltávolított 7 városon is, így a 49 városos problémának is megoldását adta.

Az 1. ábra a megoldott problémák méretét ábrázolja az idő függvényében. Látható, hogy a fejlődés rövid időn belül következett be.

A fejlődés egyik oka a növekvő számítási kapacitás. A hardware fejlődése egy ilyen bonyolultságú problémánál persze alapvetően másodrendű, de megadta a lehetőséget a kutatóknak, hogy új algoritmusokon kísérletezzenek. Ha a Held-Karp féle $O(n^2 2^n)$-es garanciát nézzük, akkor láthatjuk, hogy a teljes számítási kapacitást megduplázva pusztán annyit érünk el, hogy egyetlen várossal nagyobb problémát tudunk megoldani. Az viszont kétségtelen, hogy a gyors fejlődés a számítási kapacitás növekedése nélkül nem történhetett volna meg.

Amikor Reinelt 1990-ben először összeállította a TSPLIB-et, 24 olyan feladat került bele, amelyben a városok száma legalább ezer volt. Ez a szám 1995-ben 34-re emelkedett, amikor újabb kihívást jelentő problémákat adtak a gyűjteményhez. Kettőt ezen problémák közül (a pr1002-t és a pr2392-t) már előzőleg megoldotta Padberg és Rinaldi. Később a Concorde bevezetésével a maradék 32 problémát is megoldották.



1. ábra. Megoldott TSP feladatok mérete

A Concorde egy C nyelven írt, több mint 130 000 sorból álló program. A döntő modul a korai TSP kutatásokon alapul, legfőképp Dantzig, Fulkerson és Johnson munkája alapján. A Concorde-dal megoldott nagy méretű feladatok listáját a 2. táblázat tartalmazza. Mindegyik probléma a TSPLIB-ből származik egyetlen kivétellel: az sw24978, amely Svédország összes városát tartalmazza.

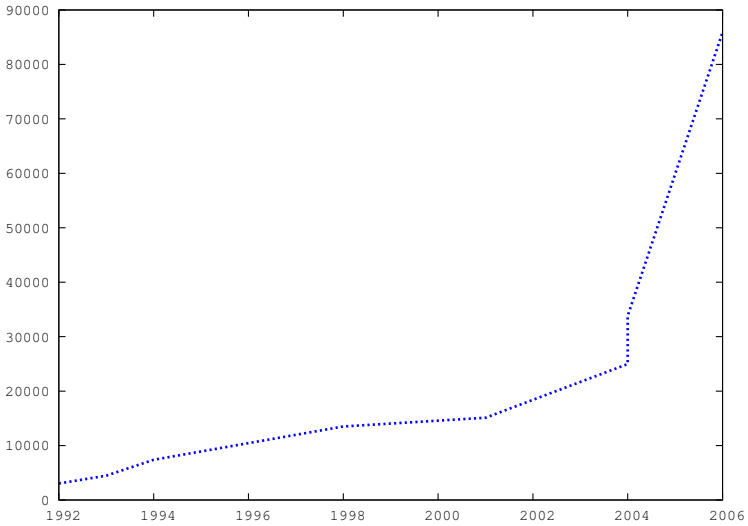
Év	Program	Méret	TSPLIB
1992	Concorde	3 038 város	pcb3038
1993	Concorde	4 461 város	gnl4461
1994	Concorde	7 397 város	pla7397
1998	Concorde	13 509 város	usa13509
2001	Concorde	15 112 város	dl15112
2004	Concorde	24 978 város	sw24978
2004	Concorde dominó-paritással	33 810 város	pla33810
2006	Concorde dominó-paritással	85 900 város	pla85900

2. táblázat. Concorde-dal megoldott feladatok

A 2. ábra mutatja a Concorde-dal megoldott feladatok gyorsan növekvő méretét. A legnagyobb, 85 900 várost tartalmazó probléma megoldásával a teljes TSPLIB gyűjtemény megoldásra került. Ez természetesen nem jelenti a TSP algoritmusok fejlődésének a végét. 2001-ben összeállítottak egy 1 904 711 várost tartalmazó feladatot a világ különböző pontjaiból. Ez a méret már messze túlhaladja a Concorde képességeit, de sok kutató újabb kihívást lát benne, és próbálnak minél jobb közelítéseket találni az optimumhoz [2].

1.4. Gráfelméleti modell

Az utazóügynök probléma modellezhető gráfként úgy, hogy a városok a gráf csúcspontjainak, az utak a gráf éleinek, az utak hossza pedig az élek költségeinek felelnek meg.



2. ábra. Concorde-dal megoldott feladatok mérete

A túra egy Hamilton körnek, az optimális túra pedig a legkisebb összköltségű Hamilton körnek felel meg.

Gyakran a gráf egy teljes gráf, azaz minden csúcspont összeköttetésben van a többivel. Ha az eredeti modellben két város nincs összekötve, akkor szokás gyengíteni a feltételeken annyiban, hogy egy csúcspontot többször is érinthet a túra, és a hiányzó él helyére felveszünk egy akkora költségű élet, mint a két pont közötti legrövidebb út költsége. A gyakorlatban ugyanis általában nem jelent gondot, hogy ha egy pontot többször bejárunk.

1.5. Szimmetrikus és Asszimetrikus TSP

A szimmetrikus TSP-ben a távolság két város között mindkét irányban azonos, azaz A városból B városba ugyan olyan hosszú az út, mint B-ből A-ba. Ilyenkor a feladatot irányítatlan gráffal modellezhetjük.

Az asszimetrikus TSP irányított gráffal modellezhető: itt két város között az oda-vissza út nem biztos, hogy azonos hosszúságú, sőt, nem is biztos, hogy létezik mindkettő. Ilyenre példa a forgalmi dugó vagy az egy irányú út esete.

A továbbiakban csak a szimmetrikus TSP feladattal foglalkozunk. Jelölje $G = (V, E)$ az irányítatlan gráfot, amelyen a problémát meg akarjuk oldani, $c : E \to \mathbb{R}$ pedig az éleken értelmezett költségfüggvényt.

1.6. Metrikus távolságok

A metrikus TSP-ben a városok közötti távolságok kielégítik a háromszög-egyenlőtlenséget. Előfordul még delta- vagy Δ-TSP néven is.

A háromszög-egyenlőtlenség azt mondja ki, hogy két pont között nincs rövidebb út a közvetlen útvonalnál, azaz i városból j városba vezető út hosszánál soha nem kisebb egy

k városon keresztülhaladó út hossza:

$$c(i,j) \leq c(i,k) + c(k,j)$$

Az ilyen távolságok egy metrikát definiálnak a csúcspontok halmazán. Ha a városok egy síkon helyezkednek el, akkor sok természetes távolságfüggvénnyel adódik metrikus TSP feladat:

- Az euklideszi TSP-ben a városok közötti távolságfüggvényt az euklideszi távolság adja.
- Az egyenes vonalú (rectilinear) TSP-ben két város közötti távolság a koordinátáik közötti különbségek összege. Ezt a metrikát Manhattan-távolságnak nevezik.
- A maximum metrikában a távolság két pont között a koordinátáik közötti különbségek maximuma.

Az utolsó két metrika előfordul a nyomtatott áramkörök lapjait kifúró gépeknél. A Manhattan-távolság esetén a gép egyszerre csak egy irányba tud mozogni, így először az x, majd az y tengelyen mozog el, a teljes mozgás hossza pedig ezek összege. Ha a gép egyszerre képes mindkét irányba fix sebességgel mozogni (átlósan is), akkor a teljes mozgás időtartama a maximum metrika szerint fog alakulni, azaz akkora lesz, mint amekkorát a nagyobb irányba meg kell tenni.

A metrikus tulajdonság jelentősen egyszerűsíti a problémát, de az így is NP-teljes marad.

1.7. Nem metrikus távolságok

Sok útvonal problémában előfordulnak nem metrikus távolságok. Például a közlekedés néhány formájánál, mint a repülés, előfordul, hogy a hosszabb útvonal ellenére is gyorsabb az utazás.

A definíció szerint a TSP nem engedi meg, hogy egy várost kétszer meglátogassunk, de sok gyakorlati alkalmazásban ez nem szükséges megszorítás. Az ilyen esetekben a szimmetrikus, nem metrikus feladatok átalakíthatóak metrikussá. Az eredeti gráfot egy teljes gráffá alakítjuk, ahol a $c(i,j)$ távolság az i és j város közötti legrövidebb út lesz.

1.8. Számítási kompexitás

A TSP probléma felírható mint élsúlyozott teljes gráfban való minimális összköltségű Hamilton-kör keresés. Feltéve, hogy P=NP nem teljesül, belátjuk, hogy a TSP probléma tetszőlegesen nagy k multiplikatív hibával sem közelíthető polinomiális időben.

Indirekt módon tegyük fel, hogy létezik olyan A_k polinom idejű algoritmus, amely bármely n pontú teljes gráfra és azon bármely c élköltségfüggvényre talál olyan Hamilton-kört, melynek összköltsége az optimum legfeljebb k-szorosa. Erre a feladatra visszavezetjük azt a problémát, hogy egy tetszőleges n pontú irányítatlan G gráfban létezik-e Hamilton-kör, ami egy közismert NP-teljes feladat.

Legyen a c költségfüggvény egy teljes gráfon definiálva úgy, hogy ha az eredeti G gráfban két pont között létezik él, akkor a teljes gráfban az él költsége egységnyi, egyébként pedig kn. Ha a G gráfban van Hamilton-kör, akkor a teljes gráfon az utazóügynök probléma optimuma n, ha nincs, akkor legalább $n - 1 + kn$, mivel legalább egy él nem létezett a G gráfban a Hamilton-körhöz. Az A_k közelítő algoritmus ezt az információt felhasználva el tudná dönteni polinom időben, hogy egy G gráfban létezik-e Hamilton-kör az alapján, hogy a teljes gráfon az utazóügynök problémára kapott közelítő megoldás költsége nagyobb-e mint kn [3].

2. Felhasználói dokumentáció

2.1. LEMON library

A LEMON egy nyílt forrású C++ template programkönyvtár, amely számos algoritmus és adatstruktúra hatékony implementációját biztosítja főként a gráfokon és hálózatokon értelmezett kombinatorikus optimalizálási feladatok megoldásához [5]. Az implementált TSP algoritmusok teljes mértékben kihasználják a LEMON adta lehetőségeket, sőt, alapvető cél a LEMON könyvtárral való kompatibilitás, az ott lefektetett konvenciók követése, hogy az elkészült implementációk bekerülhessenek a programcsomag következő verziójába.

Mivel a dolgozat célja nem a LEMON bemutatása, így továbbiakban a programkönyvtár alap szintű ismeretét feltételezzük. Ajánlatos a LEMON Tutorialt és a dokumentációt tanulmányozni [6, 7].

2.2. Tesztkörnyezet

Az algoritmusokat a TSPLIB két dimenziós euklideszi feladatain teszteljük. Ezzel lefedjük a teljes TSPLIB feladatok több mint felét, így viszonylag jó képet kapunk az algoritmusok hatékonyságáról.

2.3. Fordítás

Az algoritmusok forrásfájljainak lefordításához szükség van a LEMON könyvtárra. A LEMON forráskódja letölthető a weboldaláról [8], illetve az openSUSE rendszerhez a könyvtár előre lefordított változata letölthető az openSUSE Build Service tudományos respository-ból [9]. A fordításhoz a LEMON Tutorialban található útmutatásokat kell követni [6].

2.4. Általános interface

Mindegyik algoritmus nagyrészt hasonló public interface-szel rendelkezik, így először ezt a közös részt tárgyaljuk. Az egyes algoritmusoknak ettől való eltéréseit, illetve kiegészítéseit a részletes leírásukban említjük meg.

A CostMap a FullGraph-hoz rendelt EdgeMap rövid neve, ahogy a Cost pedig a CostMap::Value-val egyezik meg.

Metódus	**Leírás**
template <CostMap> AlgoritmusNeve(const FullGraph&, const CostMap&)	Konstruktor, paraméterként megkapja a teljes gráfot és az azon értelmezett költségeket. Természetesen az AlgoritmusNeve helyett a megfelelő nevet kell behelyettesíteni.
Cost run()	Lefuttatja az algoritmust és visszaadja a túra költségét.
template <typename L> void tourNodes(L &container)	A paraméterül kapott containerbe másolja a túra pontjait. Minden olyan adatszerkezetet támogat, amelynek van olyan konstruktora, amely két iterátorból (kezdet és vég) dolgozik. Visszatérési értéke nincs. Előfeltétel: run() lefutása.
template <template <typename> class L> L<Node> tourNodes()	Template paraméterként egy containert vár, visszatérési értéke a megadott típusú containerbe másolt túra. Az adatszerkezetnek támogatnia kell a két iterátort (kezdet és vég) paraméterül kapó konstruktort. Előfeltétel: run() lefutása.
Path<FullGraph> tour()	LEMON Path adatszerkezetben adja vissza a túra éleit. Előfeltétel: run() lefutása.
Cost tourCost()	Visszaadja a túra összköltségét. A teljes kör költségét adja vissza. Előfeltétel: run() lefutása.

Példa a paraméteres tourNodes() használatára:

```
NearestNeighbor<CostMap> nn(graph, cost);
nn.run();
...
std::deque<FullGraph::Node> res;
nn.tourNodes(res);
for (int i=0; i<res.size(); ++i) {
  ...
}
```

Példa a második, visszatérési értékben eredményt visszaadó tourNodes() függvényre:

```
NearestNeighbor<CostMap> nn(graph, cost);
nn.run();
...
std::deque<FullGraph::Node> res = nn.tourNodes<std::deque>();
for (int i=0; i<res.size(); ++i) {
  ...
}
```

Példa a tour() függvény használatára:

```
NearestNeighbor<CostMap> nn(graph, cost);
nn.run();
...
Path<FullGraph::Node> res = nn.tour();
for (int i=0; i<res.length(); ++i) {
  ...
}
```

Példa a tourCost() függvény használatára:

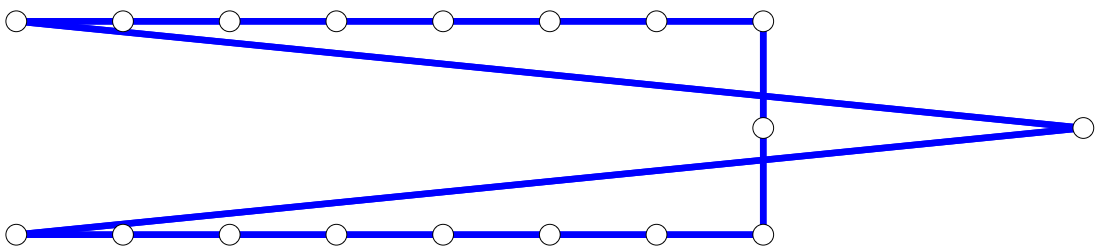
```
NearestNeighbor<CostMap> nn(graph, cost);
nn.run();
...
std::cout << nn.tourCost() << endl;
```

2.5. A legközelebbi szomszéd heurisztika

2.5.1. Az algoritmus

A legközelebbi szomszéd (Nearest Neighbor) heurisztika a legrövidebb élből indul ki. Minden lépésben a meglévő út valamelyik végpontjához a legközelebbi pontot veszi hozzá úgy, hogy kétszer ugyanazt a csúcsot nem érinti. Végül egy $n - 1$ hosszú út után kényszerűségből lépünk a kezdőpontba, és így egy túrát kapunk.

Az algoritmus egyszerűsége miatt rendkívül gyors, de mohó stratégiája miatt könnyen adhat rossz közelítést. Bár lokálisan mindig a legjobb lépéseket teszi meg, könnyen „kihagyhat” pontokat, amelyeket végül csak nagy költséggel tud bejárni, így sokszor igen kusza utakat kaphatunk. Könnyen adhatunk olyan példákat, amelyeken ez a módszer az optimálisnál jóval rosszabb túrát talál. A 3. ábra egy egyszerű gráfot mutat, amelyen a megtalált túra költsége az optimum közel kétszerese.



3. ábra. Példa a legközelebbi szomszéd heurisztika rossz közelítésére

2.5.2. Osztály leírása

A NearestNeighborTsp osztály a nearest_neighbor_tsp.h állományban van definiálva. Az általános publikus interface teljes mértékben része az osztálynak, kiegészítve a következő függvénnyel:

Metódus	**Leírás**
const std::deque<FullGraph::Node>& tourNodes()	A függvény az objektum belső adatszerkezetére ad konstans referenciát, így másolás nélkül használható. A deque tartalmazza a túra pontjait. Vigyázat: a referencia csak addig használható, amíg az objektum létezik! Előfeltétel: run() lefutása.

2.5.3. Példa

Az alábbi példa egy egyszerű 5 pontú teljes gráfon mutatja be az algoritmust. Érdemes megfigyelni, hogy nem a belső, deque adatszerkezetben kéri le a pontokat, hanem egy vektorként.

```
#include <iostream>
#include <vector>

#include "nearest_neighbor_tsp.h"
#include <lemon/full_graph.h>
#include <lemon/random.h>

using namespace std;
using namespace lemon;

int main() {
  FullGraph fullgraph; // teljes graf
  typedef FullGraph::EdgeMap<int> CostMap; // egesz sulyu elekkel

  CostMap cost(fullgraph);
  fullgraph.resize(5); // legyen 5 pont meretu

  rnd.seedFromTime();
  for (FullGraph::EdgeIt e(fullgraph); e!=INVALID; ++e)
    cost[e] = rnd[100]+1; // veletlenszeru elkoltsegek 1..100 kozott

  NearestNeighborTsp<CostMap> nn(fullgraph, cost);
  nn.run();

  cout << "tourCost: " << nn.tourCost() << endl;
```

```
  cout << "nodes: "; // kiiratjuk a pontokat
  vector<FullGraph::Node> v =
    nn.tourNodes<vector>(); // vektorkent kerjuk le
  for (unsigned int i=0; i<v.size(); ++i)
    cout << fullgraph.id(v[i]) << " ";

  cout << endl << "edges:" << endl; // majd az eleket
  Path<FullGraph> path = nn.tour();
  for (int i=0; i<path.length(); ++i)
    cout << fullgraph.id(fullgraph.source(path.nth(i))) << " -> "
         << fullgraph.id(fullgraph.target(path.nth(i))) << endl;

  return 0;
}
```

2.6. Mohó heurisztika

2.6.1. Az algoritmus

A Mohó (Greedy) heurisztika hasonló a minimális költségű feszítőfát megkereső Kruskal algoritmushoz.

Inicializálás Rendezzük sorba az éleket költségeik szerint, a kisebb költségű elemek kerüljenek előre.

Élválasztási lépés Vegyük az első még nem vizsgált élt a rendezett listából. Ha az élt hozzávéve az aktuális megoldáshalmazhoz nem alakul ki kör, és az él mindkét végpontja a megoldásban legfeljebb első fokú, akkor az élt hozzávesszük a megoldáshoz. A Kruskal-algoritmushoz képest a lényeges változtatás tehát abban áll, hogy nem engedünk meg kettőnél magasabb fokú pontokat, így $n - 1$ lépés után olyan feszítőfát kapunk, amely tulajdonképpen egy út. Ennek két végpontját összekötve megkapunk egy túrát.

Ez a módszer nagyon hasonlít az előző algoritmushoz. Szintén egy mohó stratégiát követ, de egy összefüggő út építése helyett egyszerre több, egymástól elválasztott utat építhet, amelyek aztán egy úttá kapcsolódnak össze, és legvégül ezt az utat zárjuk be egy túrává. Ez valamivel ügyesebb módszernek tűnik, mint a legközelebbi szomszéd heurisztika, de könnyen látható, hogy ugyanúgy adhat nagyon rossz közelítéseket. Például a 3. ábrán látható gráfra ugyanazt a megoldást adja, mint a legközelebbi szomszéd módszer.

2.6.2. Osztály leírása

A GreedyTsp osztály a greedy_tsp.h állományban van definiálva. Az általános publikus interface teljes mértékben része az osztálynak, kiegészítve a következő függvénnyel:

Metódus	Leírás
const std::vector<FullGraph::Node>& tourNodes()	A függvény az objektum belső adatszerkezetére ad konstans referenciát, így másolás nélkül használható. A vektor tartalmazza a túra pontjait. Vigyázat: a referencia csak addig használható, amíg az objektum létezik! Előfeltétel: run() lefutása.

2.6.3. Példa

Ez a példa az eredmény kiíratásánál GreedyTsp objektum belső adatszerkezetét közvetlenül használja, így nincs szükség felesleges másolásokra.

```
#include "greedy_tsp.h"
#include <lemon/full_graph.h>
#include <lemon/random.h>

#include <iostream>
#include <vector>

using namespace std;
using namespace lemon;

int main() {
  FullGraph fullgraph; // teljes graf
  typedef FullGraph::EdgeMap<int> CostMap; // egesz sulyu elekkel

  CostMap cost(fullgraph);
  fullgraph.resize(5); // legyen 5 pont meretu

  rnd.seedFromTime();
  for (FullGraph::EdgeIt e(fullgraph); e!=INVALID; ++e)
    cost[e] = rnd[100]+1; // veletlenszeru elkoltsegek 1..100 kozott

  GreedyTsp<CostMap> greedy(fullgraph, cost);
  greedy.run();

  cout << "tourCost: " << greedy.tourCost() << endl;

  cout << "nodes: "; // kiiratjuk a pontokat
```

```
  const vector<FullGraph::Node> &v =
    greedy.tourNodes(); // nincs masolas!
  for (unsigned int i=0; i<v.size(); ++i)
    cout << fullgraph.id(v[i]) << " ";

  cout << endl << "edges:" << endl; // majd az eleket
  Path<FullGraph> path = greedy.tour();
  for (int i=0; i<path.length(); ++i)
    cout << fullgraph.id(fullgraph.source(path.nth(i))) << " -> "
         << fullgraph.id(fullgraph.target(path.nth(i))) << endl;

  return 0;
}
```

2.7. Beszúró heurisztikák

2.7.1. Az algoritmus

Az alábbiakban nem egy, hanem négy hasonló algoritmust mutatunk be, amelyek egy beszúró (Insertion) módszeren alapulnak. Egy kisebb túrából indulnak ki, majd minden lépésben kiválasztanak egy következő pontot, és azt beszúrják a túrába arra a helyre, ahol a túra összköltsége a legkevesebbel nő. Vagyis megkeresik azt a két, az aktuális túrában egymás mellett lévő u, v pontot, amelyek esetén a beszúrandó w pontra a

$$c(uw) + c(vw) - c(uv)$$

érték minimális.

Az egyes algoritmusokat elsősorban az különbözteti meg, hogy milyen módon választják ki a következőnek beszúrandó csúcsot.

- **Legközelebbi pont beszúrása:** Valamelyik legrövidebb élből indulunk ki. Egy x pontnak egy meglévő $T \subseteq E$ túrától mért távolsága a túra pontjaitól vett távolságok minimuma, azaz:

  $$d(x) = \min_{u \in T} c(x, u)$$

  A legközelebbi pont beszúrása azt jelenti, hogy minden lépésben a túrához legközelebb lévő pontot szúrjuk be (a fenti általános szabály szerint, a legkisebb hossznövekedéssel). Tehát azt az $u \in V \setminus V[T]$ pontot választjuk, amelyre:

  $$d(u) = \min_{w \in V \setminus V[T]} d(w)$$

  Ahol $V[T]$ a T túrában szereplő élek végpontjainak halmazát jelöli.

- **Legtávolabbi pont beszúrása:** Valamelyik leghosszabb élből indulunk ki, és mindig a túrától legtávolabbi pontot szúrjuk be, azaz azt az $u \in V \setminus V[T]$ pontot, amelyre:

$$d(u) = \max_{w \in V \setminus V[T]} d(w)$$

- **Legolcsóbb beszúrás:** Valamelyik legrövidebb élből indulunk ki, és azt a pontot szúrjuk be a fenti általános szabály szerint, amelyre a túra hossznövekedése minimális.

- **Véletlen pont beszúrása:** Véletlenül választjuk ki a következő beszúrandó pontot.

2.7.2. Osztály leírása

Az InsertionTsp osztály az insertion_tsp.h állományban van definiálva. Az általános publikus interface-hez képest a következő változások vannak:

Metódus	**Leírás**
Cost run(InsertionMethod method = INSERT_FARTHEST)	A method paraméter a következő értékeket veheti fel: INSERT_NEAREST, INSERT_FARTHEST, INSERT_CHEAPEST, INSERT_RANDOM. Ezek rendre a megfelelő beszúró algoritmusokat jelentik. Az alapértelmezett a legtávolabbi pont beszúrása.
const std::vector<FullGraph::Node>& tourNodes()	A függvény az objektum belső adatszerkezetére ad konstans referenciát, így másolás nélkül használható. A vektor tartalmazza a túra pontjait. Vigyázat: a referencia csak addig használható, amíg az objektum létezik! Előfeltétel: run() lefutása.

2.7.3. Példa

Az alábbi példa az InsertionTsp alapvető használata mellett a tourNodes<T>() egy érdekesebb alkalmazási módját mutatja be, ahol saját containerbe rakjuk az adatokat.

```
#include "insertion_tsp.h"
#include <lemon/full_graph.h>
#include <lemon/random.h>

#include <iostream>
#include <list>
```

```
using namespace std;
using namespace lemon;

template <typename Type>
class Print {
  public:
    template <typename InputIterator>
    Print(InputIterator first, InputIterator last) {
      for (; first!=last; ++first)
        tour.push_back(*first);
    }
    void print(const FullGraph &g) {
      for (typename list<Type>::iterator i=tour.begin();
           i!=tour.end(); ++i) {
        cout << g.id(*i) << " ";
      }
      cout << endl;
    }
  private:
    list<Type> tour;
};

int main() {
  FullGraph fullgraph; // teljes graf
  typedef FullGraph::EdgeMap<int> CostMap; // egesz sulyu elekkel

  CostMap cost(fullgraph);
  fullgraph.resize(5); // legyen 5 pont meretu

  rnd.seedFromTime();
  for (FullGraph::EdgeIt e(fullgraph); e!=INVALID; ++e)
    cost[e] = rnd[100]+1; // veletlenszeru elkoltsegek 1..100 kozott

  InsertionTsp<CostMap> insert(fullgraph, cost);
  insert.run(InsertionTsp<CostMap>::INSERT_RANDOM);

  cout << "nodes: "; // kiiratjuk a pontokat
  Print<FullGraph::Node> p = insert.tourNodes<Print>();
  p.print(fullgraph);

  return 0;
}
```

2.8. 2-opt heurisztika

2.8.1. Az algoritmus

Definíció: Egy T túrát 2-optimálisnak nevezünk, ha nem javítható oly módon, hogy két élét elvágjuk, és a keletkező két utat máshogy kötjük össze. Tehát tetszőleges $uv, ab \in T$ élpárra teljesül, hogy $c(u,v) + c(a,b) \leq c(u,a) + c(b,v)$, ahol $T - \{uv, ab\} \cup \{ua, bv\}$ szintén túra.

Könnyen látható, hogy ha $T - \{uv, ab\} \cup \{ua, bv\}$ nem túrát, hanem két diszjunkt kört eredményez, akkor a $T - \{uv, ab\} \cup \{ub, av\}$ viszont túra.

A 2-opt algoritmus úgy működik, hogy megvizsgáljuk, hogy az adott T túrára teljesüle a 2-optimalitás feltétele. Ha nem, akkor keresünk egy olyan $uv, ab \in T$ élpárt, amelyeket lecserélve csökkenthető a túra költsége, és az így kapott túrával folytatjuk. $O(n^2)$ időben eldönthető, hogy egy túra 2-optimális-e, ám az algoritmus mégsem polinomiális idejű, nem feltétlenül áll le polinomiális mennyiségű javítás után.

A fentihez hasonló módon megfogalmazhatóak k-optimális túrák és k-opt algoritmusok is, de mivel k növekedtével nagyon gyorsan növekszik az ellenőrizendő túrák száma, így ezeket ritkán alkalmazzák.

2.8.2. Osztály leírása

Az Opt2Tsp osztály az opt2_tsp.h állományban van definiálva. Az általános publikus interface-hez képest a következő változások vannak:

Metódus	**Leírás**
Opt2Tsp(const FullGraph &gr, const CostMap &cost, const std::vector<Node> &path)	Egy második konstruktor az alapértelmezetten felül: a kezdőtúrát kapja paraméterként, így ha már van egy meglévő túránk, akkor így javíthatjuk azt. Előfeltétel: path egy tényleges túrát tartalmazzon. Hatékonysági okokból nincs ellenőrizve, hogy a beadott path input valódi túrát tartalmaz vagy sem.
const std::vector<FullGraph::Node>& tourNodes()	A függvény az objektum belső adatszerkezetére ad konstans referenciát, így másolás nélkül használható. A vektor tartalmazza a túra pontjait. Vigyázat: a referencia csak addig használható, amíg az objektum létezik! Előfeltétel: run() lefutása.

2.8.3. Példa

Az alábbi példakódban egy gráfra lefuttatjuk a beszúró heurisztikát, majd a kapott túrát próbáljuk tovább javítani a 2-opt algoritmussal. Ezután lefuttatjuk a 2-opt heurisztikát kezdőtúra megadása nélkül is.

```
#include "opt2_tsp.h"
#include "insertion_tsp.h"
#include <lemon/full_graph.h>
#include <lemon/random.h>

#include <iostream>
#include <list>

using namespace std;
using namespace lemon;

int main() {
  FullGraph fullgraph; // teljes graf
  typedef FullGraph::EdgeMap<int> CostMap; // egesz sulyu elekkel

  CostMap cost(fullgraph);
  fullgraph.resize(50); // legyen 50 pont meretu

  rnd.seedFromTime();
  for (FullGraph::EdgeIt e(fullgraph); e!=INVALID; ++e)
    cost[e] = rnd[100]+1; // veletlenszeru elkoltsegek 1..100 kozott

  // lefuttatjuk a legolcsobb beszuras heurisztikat
  InsertionTsp<CostMap> insert(fullgraph, cost);
  insert.run(InsertionTsp<CostMap>::INSERT_CHEAPEST);

  cout << "insert cost: " << insert.tourCost() << endl;

  // majd a kapott eredmenyt feljavitjuk, ha lehet
  Opt2Tsp<CostMap> opt2tsp(fullgraph, cost, insert.tourNodes());
  opt2tsp.run();

  cout << "opt2 improved insert cost: " << opt2tsp.tourCost() << endl;

  // megnezzuk az eredeti Opt2Tsp-t is
  Opt2Tsp<CostMap> opt2tsp2(fullgraph, cost);
  opt2tsp2.run();

  cout << "opt2 cost: " << opt2tsp2.tourCost() << endl;

  return 0;
```

```
}
```

2.9. Christofides heurisztikája

2.9.1. Az algoritmus

Christofides algoritmusa 1976-ból származik, és máig a legjobb elméleti korlátot adó algoritmus a szimmetrikus és metrikus TSP feladatra.

Első lépésként keressünk a gráfban egy minimális költségű F feszítőfát, mondjuk a Kruskal algoritmus segítségével. Legyen W azon csúcsok halmaza, amelyek F-ben páratlan fokúak. Keressük meg a W által feszített súlyozott részgráf, $G[W]$ egy minimális költségű teljes párosítását. Jelöljük ezt M-mel.

Tekintsük az F és M élhalmazok által meghatározott gráfot, amely két példányban tartalmazza a mindkettőben szereplő éleket. Ebben a gráfban minden csúcs foka páros, így van benne Euler-séta. Keressünk egy Euler-sétát, amelyből alkalmas csúcsokat törölve egy túrát kapunk.

Tétel: (Christofides, 1976) Az algoritmus által a szimmetrikus és metrikus utazóügynök problémára adott megoldás hossza az optimális megoldás legfeljebb 1,5-szerese.

Bizonyítás: Jelöljünk T_{opt}-tal egy optimális túrát. Tudjuk, hogy $c(F) \leq c(T_{opt})$. Vegyük a $G[W]$ gráfnak azt a T_W túráját, amelyet a W pontjainak T_{opt}-beli sorrendje ad. A háromszög-egyenlőtlenség miatt $c(T_W) \leq c(T_{opt})$. Mivel T_W előáll mint W két teljes párosításának uniója, M definíciójának értelmében $2c(M) \leq c(T_W) \leq c(T_{opt})$, azaz $c(T \cup M) \leq \frac{3}{2}c(T_{opt})$. Megint a háromszög-egyenlőtlenség miatt a végső túra költsége is legfeljebb $c(T \cup M)$, így bebizonyítottuk a tételt.

2.9.2. Osztály leírása

Az ChristofidesTsp osztály a christofides_tsp.h állományban van definiálva. Az általános publikus interface-hez képest a következő változások vannak:

Metódus	**Leírás**
const std::vector<FullGraph::Node>& tourNodes()	A függvény az objektum belső adatszerkezetére ad konstans referenciát, így másolás nélkül használható. A vektor tartalmazza a túra pontjait .Vigyázat: a referencia csak addig használható, amíg az objektum létezik! Előfeltétel: run() lefutása.

2.9.3. Példa

Az alábbi példa a ChristofidesTsp algoritmus egy egyszerű használatát mutatja be.

```
#include "christofides_tsp.h"
#include <lemon/full_graph.h>
#include <lemon/random.h>

#include <iostream>
#include <list>

using namespace std;
using namespace lemon;

int main() {
  FullGraph fullgraph; // teljes graf
  typedef FullGraph::EdgeMap<int> CostMap; // egesz sulyu elekkel

  CostMap cost(fullgraph);
  fullgraph.resize(50); // legyen 50 pont meretu

  rnd.seedFromTime();
  for (FullGraph::EdgeIt e(fullgraph); e!=INVALID; ++e)
    cost[e] = rnd[100]+1; // veletlenszeru elkoltsegek 1..100 kozott

  ChristofidesTsp<CostMap> christofides(fullgrahp, cost);
  christofides.run();
  cout << "cost: " << christofides.tourCost() << endl;

  return 0;
}
```

3. Fejlesztői dokumentáció

3.1. Általános leírás

Mivel az algoritmusok mindegyikének van egy közös kódbázisa (főként lekérdező függvények), ezért először ezeket tárgyaljuk egyben. Az ezektől való implementációs eltéréseket minden esetben specifikusan külön jelezzük a különböző algoritmusok leírásánál.

3.1.1. Publikus típusdefiníciók

Az összes osztály egy CM template típusparamétert vár a specializáláshoz. Ez a CM a konstruktornak átadott FullGraph-hoz tartozó élsúlyozás típusa. A Mapokról bővebb leírás a LEMON dokumentációjában olvasható [10].

A könnyebb olvashatóság kedvéért a programokban két új típus nevet vezetünk be:

```
typedef CM CostMap;
typedef typename CM::Value Cost;
```

A CostMap magát az élköltségeket tároló map típusát jelenti, míg a Cost az élek költségének típusát. (Példa: CM = FullGraph::EdgeMap<int> esetén CostMap = FullGraph::EdgeMap<int>, Cost = int)

3.1.2. Adattagok

Három privát adattag része mindegyik algoritmusosztálynak: egy-egy konstans referencia a gráfhoz és a hozzá tartozó élköltségekhez, valamint egy Cost típusú változó a túra összköltségének nyilvántartásához.

```
private:
  const FullGraph &_gr;
  const CostMap &_cost;
  Cost _sum;
```

3.1.3. Default konstruktor

Minden algoritmus a konstruktorban egy konstans referenciát vár a teljes gráfra és a hozzá tartozó mapre. Az inicializáló listában beállítják a _gr és _cost referenciákat a paraméterül kapott értékekre.

Példa:

```
NearestNeighborTsp(const FullGraph &gr, const CostMap &cost) :
        _gr(gr), _cost(cost) {}
```

3.1.4. Helper névtér

Minden osztály rendelkezik egy névtérrel, amelyben különböző C++ containerek közötti átjárást biztosító függvények találhatóak. A legtöbb algoritmus std::vector-ból képes más típusokra konvertálni, így itt ezt mutatjuk be. Amennyiben valamelyik algoritmusnál változás van, azt ott kifejezetten jelezzük. Ezek a függvények azért nem az algoritmus osztályában vannak definiálva, mert beágyazott template típust nem lehet specializálni a teljes osztály specializálása nélkül, nekünk pedig szükségünk van a specializációra.

```
template <typename L>
L vectorConvert(const std::vector<FullGraph::Node> &_path) {
  return L(_path.begin(), _path.end());
}

template <>
std::vector<FullGraph::Node> vectorConvert(
        const std::vector<FullGraph::Node> &_path) {
  return _path;
}
```

A vectorConvert() egy olyan típust vár paraméterül, melynek van olyan konstruktora, amely két iterátort fogad: egy kezdetet és egy véget. Magának az L osztálynak nem kell feltétlenül tároló típusnak lennie, de rendelkeznie kell egy ilyen konstruktorral.

A template specializáció arra az esetre van megírva, ha a függvényt hívó fél vektort vár visszatérési értékül. Ilyenkor nem hozzuk létre újra a már meglévő vektort.

3.1.5. tourNodes() függvények

Háromféle tourNodes() függvény érhető el, mindegyiknek különböző a használata, de alapvetően ugyan azt csinálják: visszaadják a túra pontjait. Mindegyiknek előfeltétele a run() lefutása.

Lehetőség van az algoritmus belső adatszerkezetét használni, de a felület nem köti meg a felhasználója kezét azzal, hogy valamilyen speciális adatszerkezetre (például list-re vagy deque-re) kényszeríti az eredmény kiolvasásánál, hanem megadja a lehetőséget, hogy olyan adatszerkezetben kapja meg az adatot, amilyenben kívánja, amennyiben az megfelel a minimális elvárásoknak.

Első változat:

```
const std::vector<Node>& tourNodes() {
  return _path;
}
```

Ez a függvény template paraméter nélküli, mindig a belső adatszerkezetre ad konstans referenciát. Akkor érdemes használni, ha magát az adatszerkezetet csak olvasni kívánjuk,

módosítani nem. Ekkor másolás nélkül elérhető az eredmény. Szinte mindig std::vector a belső adatszerkezet, ahol nem, ott ez külön jelölésre kerül.

Futásidő: $O(1)$.

Második változat:

```
template <typename L>
void tourNodes(L &container) {
  container(alg_helper::vectorConvert<L>(_path));
}
```

Ez a függvény paraméterként várja a contanier osztályt, és ebbe fogja beleírni közvetlenül, ekkor csak a konvertálás miatt létezik átmeneti eltárolás, több másolás nem történik. Kényelmes, mivel nem kell specifikálni az osztálynak a típust, felismeri automatikusan. Feltétel a típusra: létezzen olyan konstruktora, amely egy kezdet és egy vég iterátort kap paraméterül.

Futásidő: $O(n)$, mivel le kell másolnunk a belső n elemű adatszerkezetet.

Harmadik változat:

```
template <template <typename> class L>
L<Node> tourNodes() {
  return alg_helper::vectorConvert<L<Node> >(_path);
}
```

Ennél a változatnál egy template paraméterben meg kell adni a container típusát, amelyben az eredményt várjuk. Feltétel a típusra: lennie kell olyan konstruktornak, amely egy kezdet és egy vég iterátort kap paraméterként, és L<Node> helyes legyen.

Futásidő: $O(n)$, mivel le kell másolnunk a belső n elemű adatszerkezetet.

3.1.6. tour() függvény

A tour() függvénnyel a túra éleit tudjuk lekérni LEMON Path<FullGraph> formában [11].

```
Path<FullGraph> tour() {
  Path<FullGraph> p;
  if (_path.size()<2)
    return p;

  for (unsigned int i=0; i<_path.size()-1; ++i) {
    p.addBack(_gr.arc(_path[i], _path[i+1]));
  }
  p.addBack(_gr.arc(_path.back(), _path.front()));
  return p;
}
```

A ciklus végigjárja a belső adatszerkezetet, amelyben Node-okat tárolunk, majd a jelenlegi és a következő Node által meghatározott élt hozzáadja a Path<FullGraph> típushoz. A függvény implementációja minden algoritmusnál ehhez hasonló, pusztán olyan különbségekkel, amelyek az eltérő reprezentációból adódnak.

Futásidő: $O(n)$, mivel le kell másolnunk a belső n elemű adatszerkezetet.

3.1.7. tourCost() függvény

Visszaadja a teljes túra költségét. Előfeltétel: run() lefutása.

```
Cost tourCost() {
  return _sum;
}
```

Futásidő: $O(1)$.

3.1.8. run() függvény

Az algoritmusok implementációja itt jelenik meg, ezt a részt külön kifejtjük algoritmusonként.

3.2. Tesztelési terv

Az algoritmus lefutása után három tesztelésen esik át.

Túra teszt Az algoritmus által pontok formájában visszaadott túra ellenőrzésre kerül. Csak akkor teljesíti ezt a tesztet, ha minden pont szerepel benne, és egyik pont sem szerepel kétszer.

Költség teszt 1 Az algoritmus által élek formájában visszaadott túra összköltsége ellenőrzésre kerül. Ha a tourCost() által visszaadott érték nem felel meg a visszaadott túra tényleges költségének, akkor az algoritmus nem teljesíti a tesztet.

Költség teszt 2 Az algoritmus által pontok formájában visszaadott túra összköltsége ellenőrzésre kerül. Ha a tourCost() által visszaadott érték nem felel meg a visszaadott túra tényleges költségének, akkor az algoritmus nem teljesíti a tesztet.

Mindegyik algoritmus teljesítette ezeket a teszteket az összes input esetén.

3.3. A legközelebbi szomszéd heurisztika

3.3.1. Tárolt adatok

Az algoritmus teljes futása során ismerni kell a felépített utat és annak két végpontját. Az általunk megvalósított verzióban az út mindkét végére szúrhatunk be új pontot, és mindig oda szúrjuk be, ahol kisebb a túra összköltségének növekedése. Ennél fogva szükség van a túra két végpontjának tárolására és karbantartására.

Nyilván kell még tartani a már meglátogatott pontokat, hogy ne járjunk be kétszer egy pontot.

3.3.2. Megvalósítás

A belső adatszerkezet, amelyben a túrát nyilvántartjuk egy std::deque. Ez azért célszerű, mert $O(1)$ időben mindkét végébe tudunk új elemet beszúrni, nekünk pedig pontosan erre lesz szükségünk.

Két Node-dal tartjuk nyilván a meglévő túránk két végpontját, valamint két Edge-et használunk a végpontokból kimenő legrövidebb él eltárolására. Egy FullGraph::NodeMap <bool>-t használunk a már elért csúcsok nyilvántartásához, egy Node elértté nyilvánítása $O(1)$ időt igényel.

Inicializálás: felvesszük a legkisebb költségű élt, a végpontokat az él két végpontjára, a legkisebb kimenő éleket INVALID-ra, azaz nem ismertre állítjuk. A legkisebb költségű él megkeresésére a LEMON mapMin() függvényét használjuk.

Minden lépés kezdetén, ha nem ismerjük valamelyik végpontból a legrövidebb kiinduló élt, akkor először ezt keressük meg. Ezután a rövidebb élhez tartozó pontot beszúrjuk, majd felvesszük a már bejárt pontok közé és ismeretlenné tesszük az ehhez a végponthoz tartozó legrövidebb élt. Ha a beszúrt pont a másik végponthoz tartozó legrövidebb él végpontja is egyben, akkor a másik végponthoz tartozó legrövidebb élt is ismeretlenné tesszük. n lépés után a túra elkészül, az algoritmus végén kiszámoljuk a túra összköltségét.

Mivel minden beszúrási lépés $O(n)$ időben megvalósítható, így a teljes algoritmus lépésszáma $O(n^2)$, vagyis a teljes gráf éleinek számában lineáris.

3.3.3. Általános leírástól való eltérések

A Helper névtér neve: nn_helper.

Mivel az algoritmus belső adatszerkezete std::deque, így az általános leírásban tárgyalt std::vector-os konvertáló függvények ebben az esetben std::deque alapokon szerepelnek.

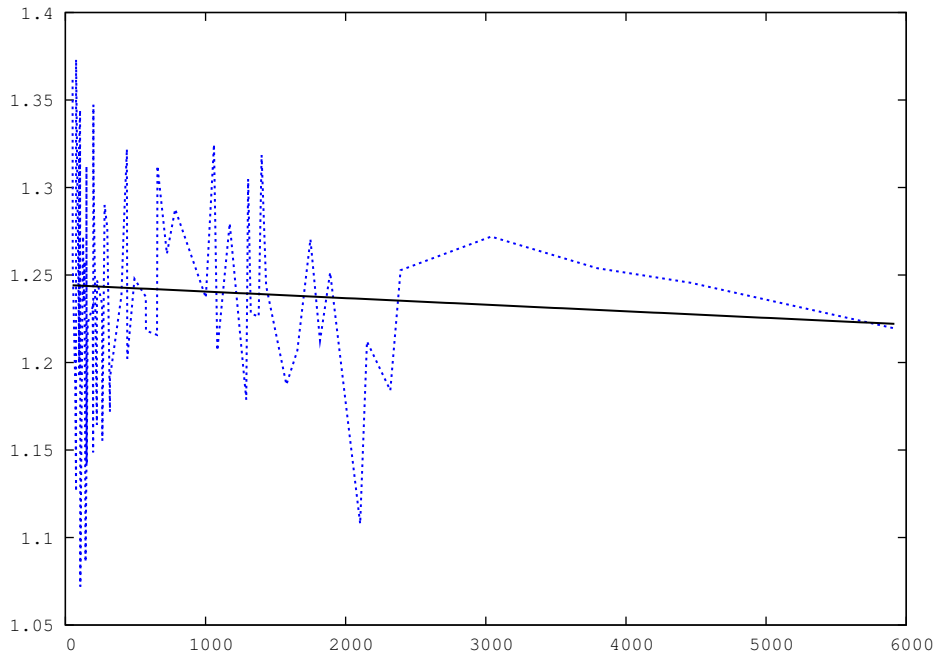
```
namespace nn_helper {
  template <typename L>
  L dequeConvert(const std::deque<FullGraph::Node> &_path) {
```

```
    return L(_path.begin(), _path.end());
  }

  template <>
  std::deque<FullGraph::Node> dequeConvert(
          const std::deque<FullGraph::Node> &_path) {
    return _path;
  }
}
```

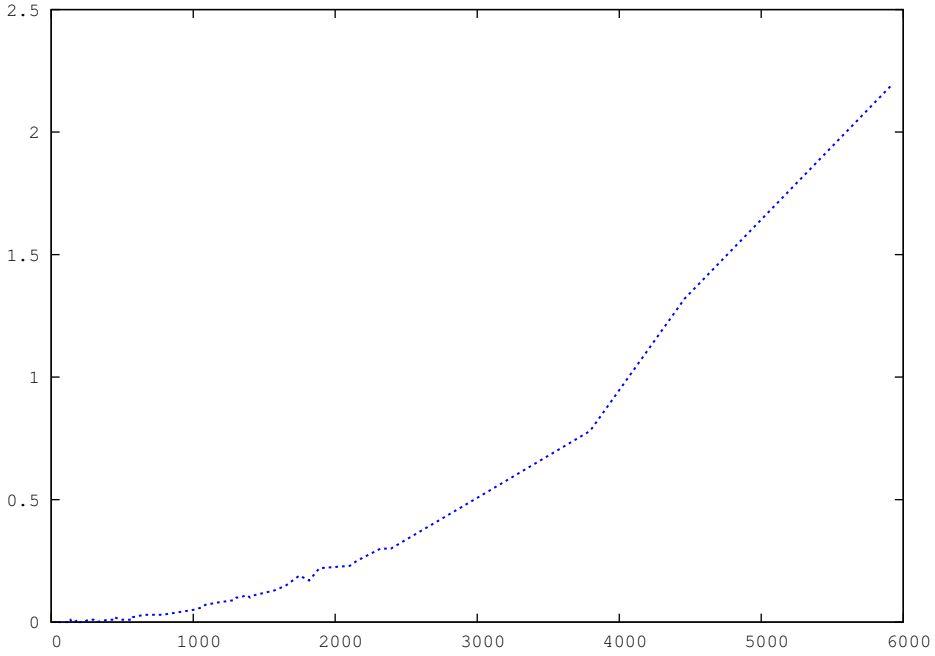
3.3.4. Hatékonyságelemzés

A megoldott TSPLIB problémákon elért eredményeket a 4. ábra mutatja méret szerint növekvő sorba rendezve. Az y tengely az algoritmus által adott eredmény és az optimális megoldás hányadosát jelenti. A képen megjelenítjük az adathalmazra legjobban illeszkedő lineáris függvényt is, amely közel vízszintes, így azt mutatja, hogy az optimális megoldástól való távolság közel független a feladat méretétől. Az egyenes lefelé tartása valószínűleg pusztán annak tudható be, hogy a TSPLIB feladatokból jóval több kisméretű probléma lett megoldva, mint nagy.

A futási időket az 5. ábra mutatja. Még közel 6000 város esetén sem érte el a 3 másodpercet. A 3. táblázat tartalmazza a minimális, maximális és átlagos eredményeket.



4. ábra. Legközelebbi szomszéd heurisztika: TSPLIB eredmények



5. ábra. Legközelebbi szomszéd heurisztika: TSPLIB futásidők

	Min	**Max**	**Átlag**
Futásidő (másodperc)	0.000000	2.190000	0.109722
Összköltség/optimum	1.072004	1.373164	1.241208

3. táblázat. A legközelebbi szomszéd heurisztika eredményei

3.4. Mohó heurisztika

3.4.1. Tárolt adatok

Egyrészt szükségünk van az élek rendezett halmazára. Másrészt el kell tárolnunk a már megtalált utakat, és nyilván kell tartanunk, hogy ezek milyen csúcsokat tartalmaznak, hogy egy új él beszúrásánál ellenőrizni lehessen, hogy kört alkot majd vagy sem. Ezen kívül a pontok fokát is nyilván kell tartanunk.

3.4.2. Megvalósítás

Az éleket egy std::vector-ban helyezzük el, majd az std::sort() algoritmusa segítségével rendezzük őket növekvően. Az utak eltárolására egy $2n$ hosszú vektort használunk. Minden i. ponthoz a vektor $2i$ és $2i + 1$-edik eleme tartozik. A gráf minden pontjához eltároljuk, hogy milyen két másik ponthoz kapcsolódik. Alapértelmezettként minden elem -1, azaz egyik pont sem kapcsolódik sehova.

A körök ellenőrzéséhez egy hatékony speciális adatszerkezetet használunk, amely kifejezetten diszjunkt halmazok nyilvántartására és uniójuk számolására lett kidolgozva. Egy ilyen implementáció UnionFind néven a LEMON könyvtár részét képzi. A UnionFind képes elemeket halmazokba szervezni, azokat egyesíteni, illetve megmondani, hogy két elem ugyanabban a halmazban van-e vagy sem.

A gráfpontok fokát egy FullGraph::NodeMap<int>-tel tároljuk el, kezdetben minden pont foka nulla.

Minden lépésben megnézzük az éppen soron következő (növekvő sorrend szerint) él végpontjait. Ha mindkét végpont foka kisebb vagy egyenlő mint egy, és még külön halmazban vannak, akkor a két pont halmazát egyesítjük, a pontok fokát megnöveljük eggyel, majd eltároljuk a létrejött kapcsolatokat a $2n$ hosszú vektorban. Mivel csak olyan pontokhoz húzhatunk be élt, amelyeknek foka legfeljebb egy, ezért a $2n$ hosszú vektorban mindkét csúcsnál lesz olyan hely, ahova beírható az új él.

Az algoritmus addig fut, amíg $n - 1$ élt be nem húzott. Az utolsó él pedig az első és az utolsó pontot összekötő él lesz.

3.4.3. Általános leírástól való eltérések

A Helper névtér neve: greedy_tsp_helper.

Az általános résznél tárgyalt konvertáló függvények itt is része a névtérnek kiegészítve az alábbi osztállyal:

```
template <typename CostMap>
class KeyComp {
  typedef typename CostMap::Key Key;
  const CostMap &cost;
```

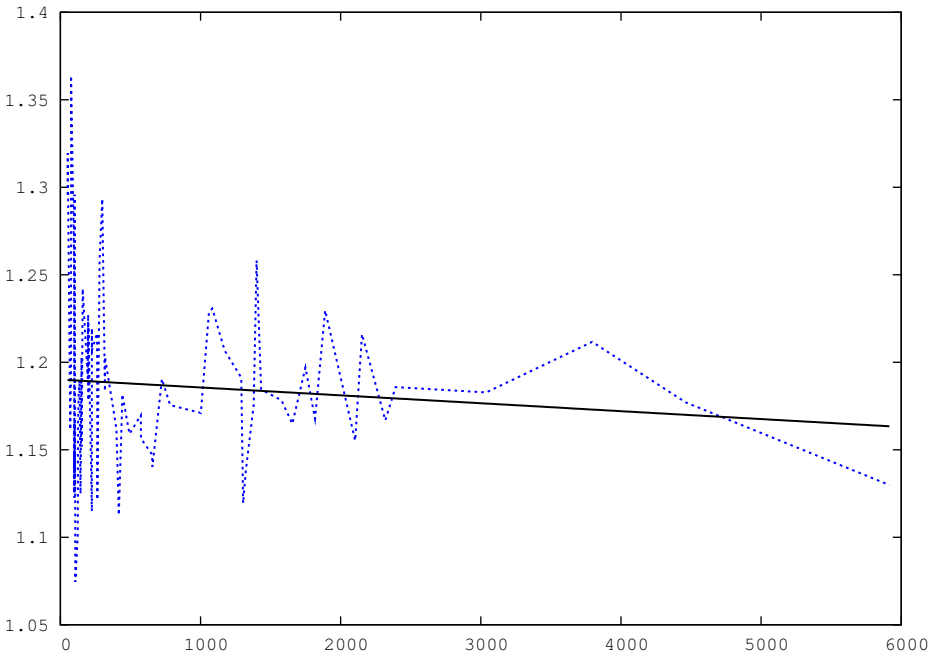
```
  public:
    KeyComp(const CostMap &_cost) : cost(_cost) {}

    bool operator() (const Key &a, const Key &b) const {
      return cost[a] < cost[b];
    }
};
```

Ez egy komparátor osztály, amelyet az std::sort() algoritmus használ élek összehasonlítására.

3.4.4. Hatékonyságelemzés

A megoldott TSPLIB problémákon elért eredményeket az előző algoritmushoz hasonlóan a 6. ábra mutatja. Itt is megjelenítjük az adathalmazra legjobban illeszkedő lineáris függvényt, amely itt is közel vízszintes.

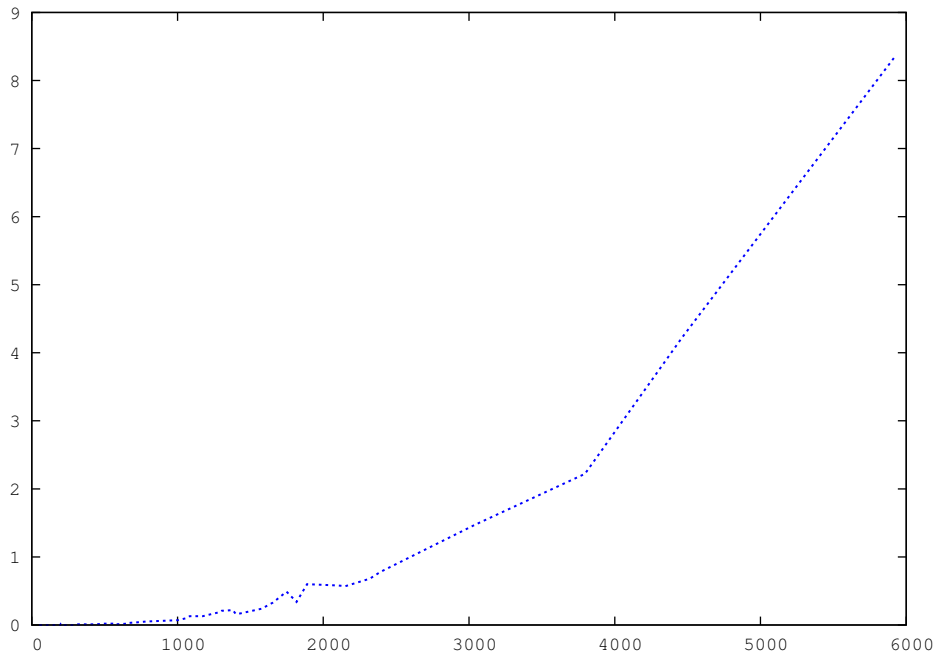
A futási időket a 7. ábra mutatja. Ez az algoritmus 6000 város esetén 8 másodpercig számolt. A 4. táblázat tartalmazza a minimális, maximális és átlagos eredményeket.



6. ábra. A mohó heurisztika: TSPLIB eredmények

	Min	**Max**	**Átlag**
Futásidő (másodperc)	0.000000	8.330000	0.315416
Összköltség/optimum	1.074577	1.363696	1.186397

4. táblázat. A mohó heurisztika eredményei



7. ábra. A mohó heurisztika: TSPLIB futásidők

3.5. Beszúró heurisztikák

3.5.1. Tárolt adatok

Nyilván kell tartanunk, hogy mely csúcsokat érintettük már, és képesnek kell lennünk beszúrni a kiválasztott pontot a már meglévő túránkba.

3.5.2. Megvalósítás

Mivel négy alapvetően hasonló algoritmusról van szó, ezért egy általános implementációt készítettünk, amelyben a megfelelő részek lecserélhetőek.

A run() függvény a következőképpen néz ki:

```
Cost run(InsertionMethod method = INSERT_FARTHEST) {
  switch (method) {
    case INSERT_NEAREST:
      start<InitTour<true>, NearestSelection, DefaultInsert>();
      break;
    case INSERT_FARTHEST:
      start<InitTour<false>, FarthestSelection, DefaultInsert>();
      break;
    case INSERT_CHEAPEST:
      start<InitTour<true>, CheapestSelection, CheapestInsert>();
      break;
    case INSERT_RANDOM:
      start<InitTour<true>, RandomSelection, DefaultInsert>();
      break;
```

```
    }
    return sum;
  }
```

Ahogy láthatjuk a start() template függvényt a megfelelő beszúró algoritmushoz illeszkedve példányosítjuk. Vegyük sorra a megfelelő osztályok működését.

InitTour Az InitTour osztály init() függvénye állítja be a kezdőtúrát. Egy bool template argumentumot vár a specializáláshoz, ez dönti el, hogy a legkisebb vagy a legnagyobb él legyen a kezdőtúra. Visszaadja a kezdőtúrát tartalmazó vektort, beállítja a csúcsok használatát nyilvántartó vektorokat.

NearestSelection A select() függvénye kiválasztja a meglévő túrához legközelebb lévő pontot és törli a még nem használt pontok közül. Visszaadja a kiválasztott Node-ot.

FarthestSelection A select() függvénye kiválasztja a meglévő túrától legtávolabb lévő pontot és törli a még nem használt pontok közül. Visszaadja a kiválasztott Node-ot.

CheapestSelection A select() függvénye kiválasztja a meglévő túrához a legolcsóbban beszúrható pontot és be is szúrja azt a túra megfelelő helyére. Ez azért más mint a többi függvény, mert ekkor már a kiválasztás idejében ismert az, hogy hova kell beszúrni, így helyben meg lehet tenni. Visszatérési értéke a túra összköltségének változása.

RandomSelection A select() függvénye kiválaszt egy véletlenszerű pontot és törli a még nem használt pontok közül. Visszaadja a kiválasztott Node-ot.

DefaultInsert Az insert(Node n) függvénye beszúrja a meglévő túrában arra a helyre, ahol a legkisebb összköltség növekedést éri el, majd ennek megfelelően karbantartja az összköltséget tároló változót.

CheapestInsert Az insert(Cost diff) függvénye hozzáadja a változást az összköltséget karbantartó változóhoz.

A start függvény a következőképpen néz ki:

```
template <typename InitFunctor, typename SelectionFunctor,
          typename InsertFunctor>
void start() {
  InitFunctor init(_gr, _cost, nodesPath, notused, sum);
  SelectionFunctor selectNode(_gr, _cost, nodesPath, notused);
  InsertFunctor insertNode(_gr, _cost, nodesPath, sum);
```

```
    nodesPath = init.init();

    for (int i=0; i<_gr.nodeNum()-2; ++i) {
      insertNode.insert(selectNode.select());
    }

    sum = _cost[ _gr.edge(nodesPath.front(), nodesPath.back()) ];
    for (unsigned int i=0; i<nodesPath.size()-1; ++i)
      sum += _cost[ _gr.edge(nodesPath[i], nodesPath[i+1]) ];
  }
```

Azért _gr.nodeNum()-2 beszúrás van, mert 2 csúcsot már elhelyezett a grában az init függvény. A függvény végén azért kerül újra összeszámolásra az összköltség, mert a beszúró algoritmusok sok összeadást és kivontást végeznek, amelyek nem egészértékű élköltségek esetén numerikus hibával terheltek lehetnek.

3.5.3. Általános leírástól való eltérések

A segítő névtér osztály neve: insertion_tsp_helper.

Alapvető eltérés az általános szerkezettől, hogy a run(InsertionMethod) függvény egy paramétert vár a következőkből:

```
enum InsertionMethod {
  INSERT_NEAREST,
  INSERT_FARTHEST,
  INSERT_CHEAPEST,
  INSERT_RANDOM
};
```

Ezzel lehet kiválasztani a beszúró módszert.

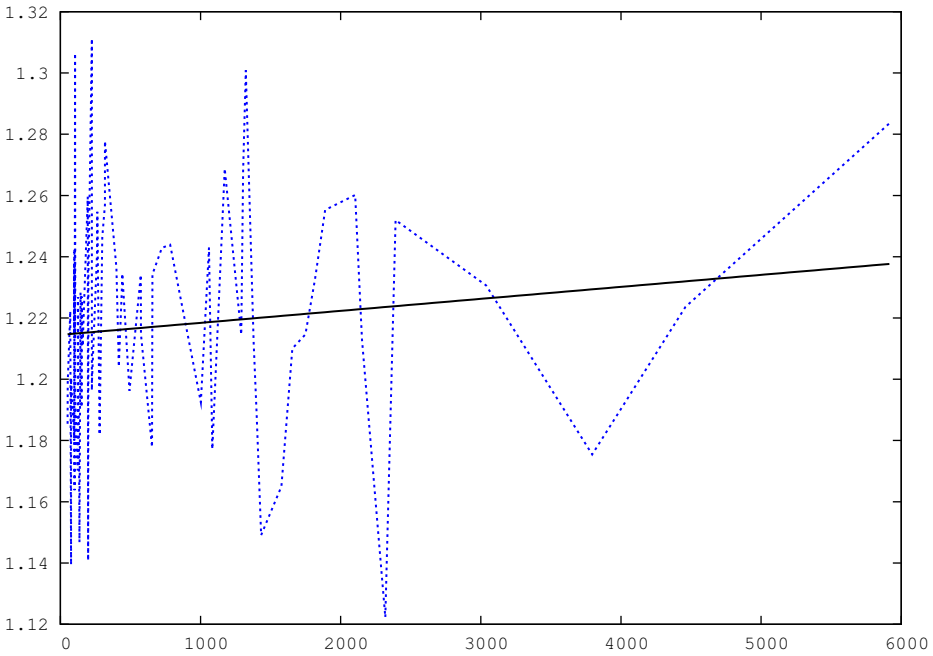
3.5.4. Hatékonyságelemzés

Legközelebbi beszúrás A TSPLIB eredményeket az előzőekhez hasonló módon a 8. ábra, a futási időket pedig a 9. ábra mutatja. Ez az algoritmus 6000 város esetén 12 percig számolt. Az 5. táblázat tartalmazza a minimális, maximális és átlagos eredményeket.

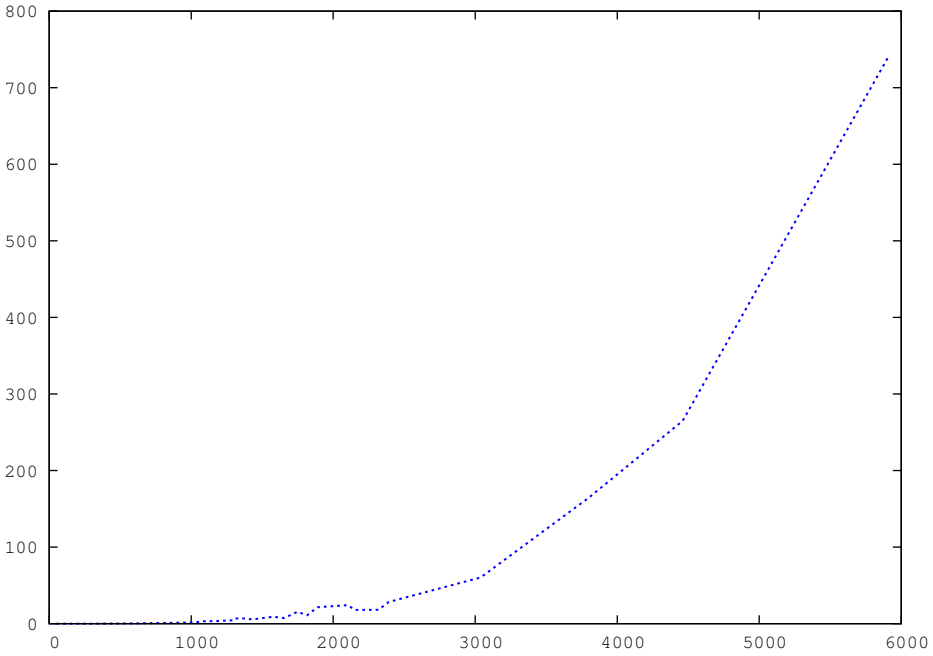
	Min	**Max**	**Átlag**
Futásidő (másodperc)	0.000000	742.270000	19.932777
Összköltség/optimum	1.122281	1.311221	1.217736

5. táblázat. A legközelebbi beszúrás heurisztika eredményei

Legtávolabbi beszúrás A TSPLIB eredményeket az előzőekhez hasonló módon a 10. ábra, a futási időket pedig a 11. ábra mutatja. Ez az algoritmus 6000 város esetén 12 percig számolt. A 6. táblázat tartalmazza a minimális, maximális és átlagos eredményeket.



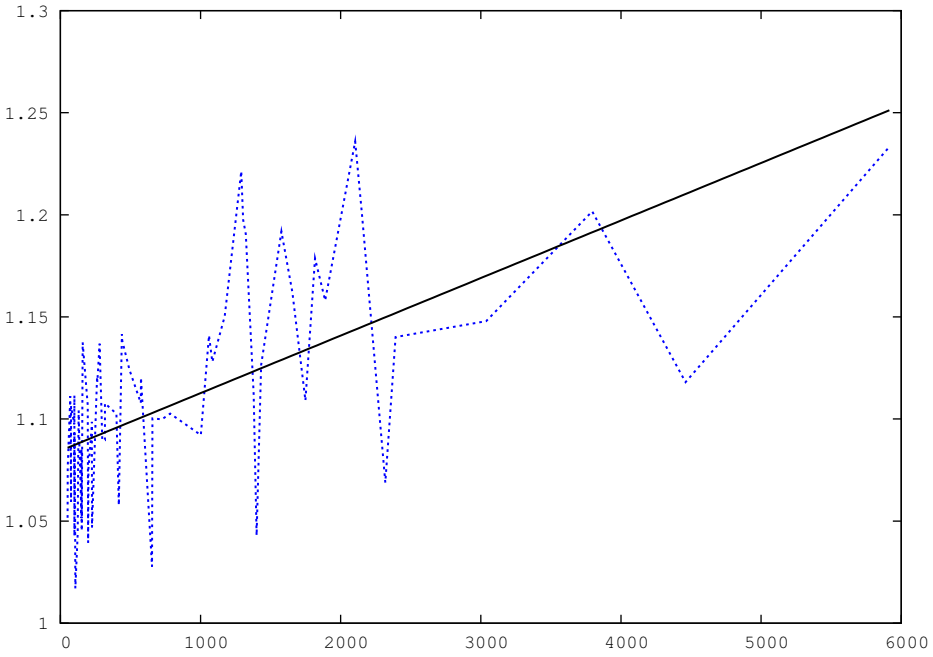
8. ábra. A legközelebbi beszúrás heurisztika: TSPLIB eredmények



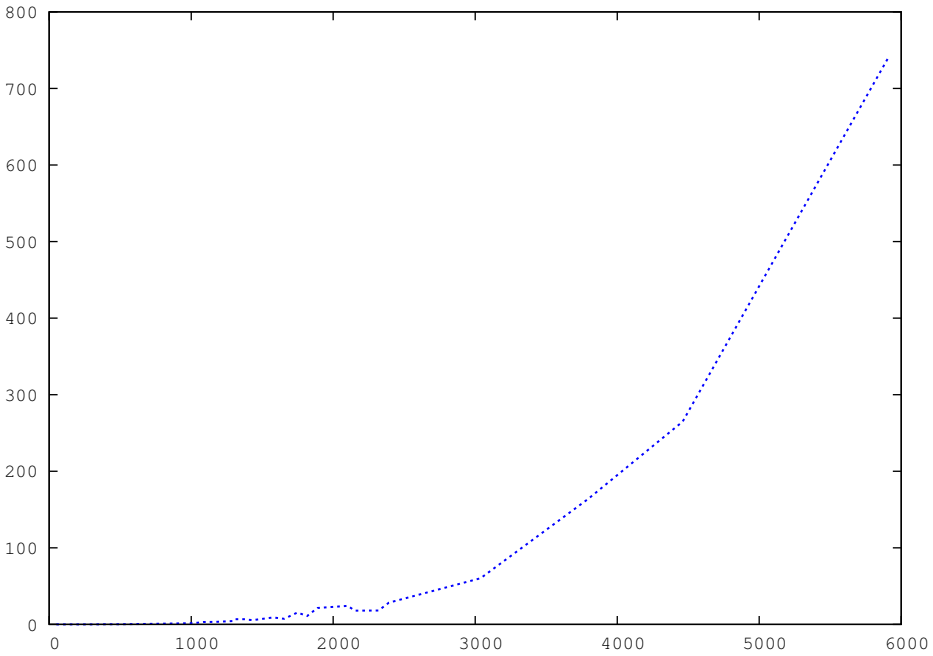
9. ábra. A legközelebbi beszúrás heurisztika: TSPLIB futásidők

	Min	**Max**	**Átlag**
Futásidő (másodperc)	0.000000	736.270000	19.875277
Összköltség/optimum	1.017019	1.235923	1.107729

6. táblázat. A legtávolabbi beszúrás heurisztika eredményei

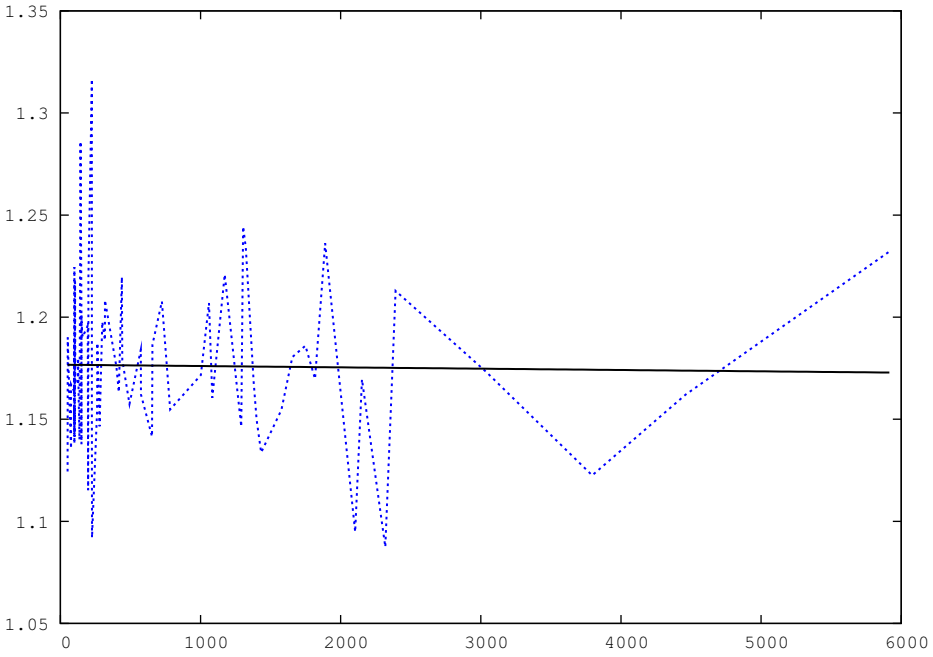


10. ábra. A legtávolabbi beszúrás heurisztika: TSPLIB eredmények

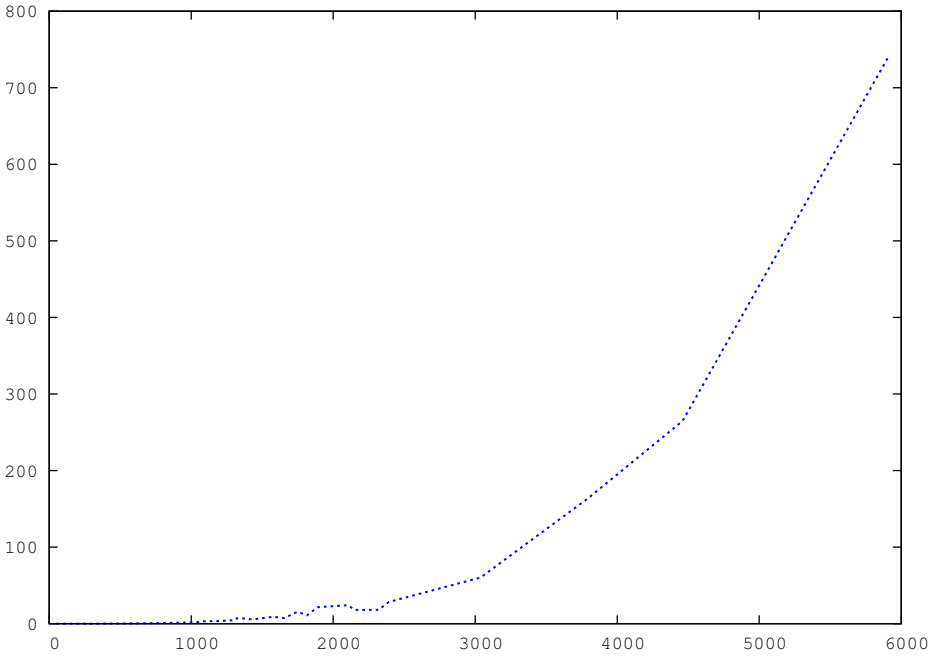


11. ábra. A legtávolabbi beszúrás heurisztika: TSPLIB futásidők

Legolcsóbb beszúrás A TSPLIB eredményeket az előzőekhez hasonló módon a 12. ábra, a futási időket pedig a 13. ábra mutatja. Ez az algoritmus 6000 város esetén 25 percig számolt. A 7. táblázat tartalmazza a minimális, maximális és átlagos eredményeket.



12. ábra. A legolcsóbb beszúrás heurisztika: TSPLIB eredmények



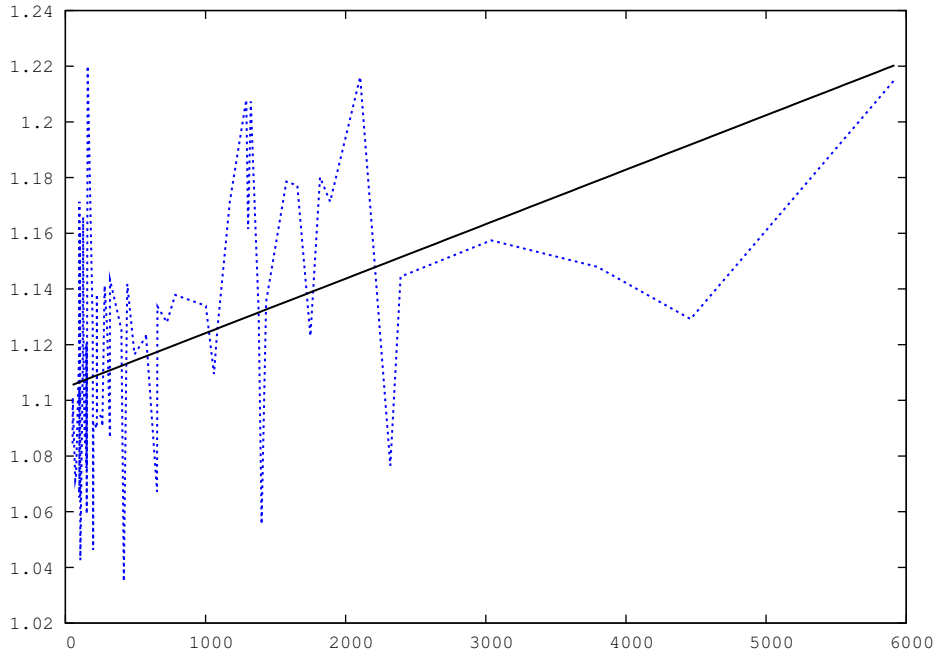
13. ábra. A legolcsóbb beszúrás heurisztika: TSPLIB futásidők

Véletlen beszúrás A TSPLIB eredményeket az előzőekhez hasonló módon a 14. ábra, a futási időket pedig a 15. ábra mutatja. Ez az algoritmus 6000 város esetén is nagyon

	Min	Max	Átlag
Futásidő (másodperc)	0.000000	1507.510000	42.924166
Összköltség/optimum	1.087451	1.316599	1.176176

7. táblázat. A legolcsóbb beszúrás heurisztika eredményei

alacsony, 1-2 másodperces futásidővel rendelkezik. A 8. táblázat tartalmazza a minimális, maximális és átlagos eredményeket.



14. ábra. A véletlen beszúrás heurisztika: TSPLIB eredmények

	Min	Max	Átlag
Futásidő (másodperc)	0.000000	1.200000	0.052500
Összköltség/optimum	1.034989	1.220105	1.120734

8. táblázat. A véletlen beszúrás heurisztika eredményei

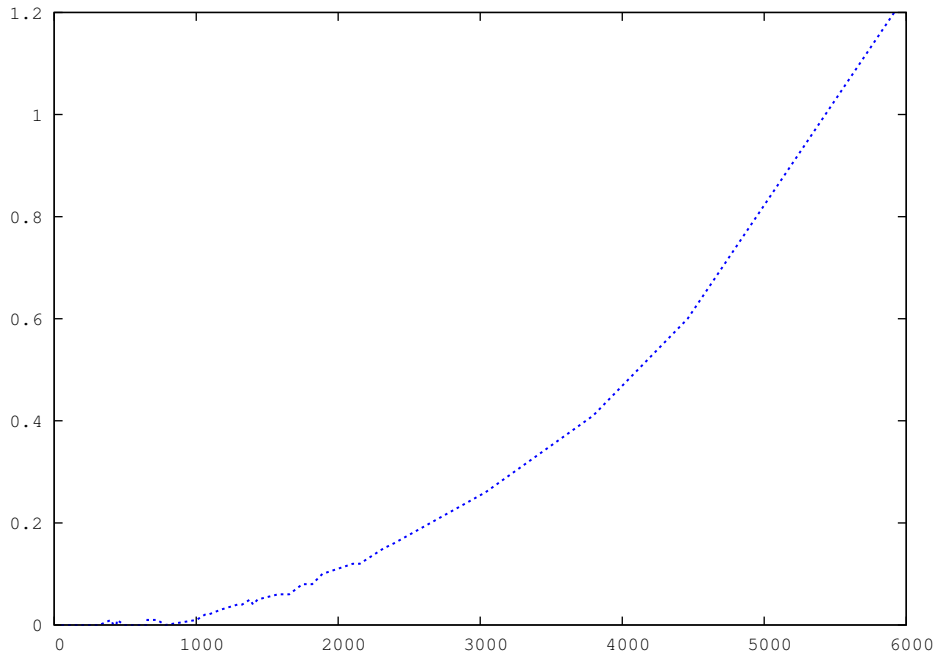
3.6. 2-opt heurisztika

3.6.1. Tárolt adatok

Az algoritmus működése során tárolnunk kell a már meglévő túrát, és az adatszerkezetnek támogatnia kell a gyors cseréket a túrában.

3.6.2. Megvalósítás

A túrát egy $2n$ hosszú vektorban tároljuk. A $2i$ és $2i + 1$-edik pont a vektorban az i. pont két szomszédját jelentik. Mivel az algoritmusban sokszor használjuk ezt az adatstruktúrát,



15. ábra. A véletlen beszúrás heurisztika: TSPLIB futásidők

így létrehozunk hozzá egy It iterátor osztályt, amely megkönnyíti a túra ilyen módon való bejárását. Az implementációs részletektől eltekintve (nem igényel különösebb magyarázatot) álljon itt az iterátor prototípusa:

```
class It {
  public:
    It(const std::vector<int> &path, int i=0);
    It(const std::vector<int> &path, int i, int l);

    int next_index() const;
    int prev_index() const;
    int next() const;
    int prev() const;
    It& operator++();
    operator int();

    private:
      const std::vector<int> &tmppath;
      int act;
      int last;
};

bool check(std::vector<int> &path, It i, It j) {
  if (c(i, i.next()) + c(j, j.next()) >
      c(i, j) + c(j.next(), i.next())) {

      path[ It(path, i.next(), i).prev_index() ] = j.next();
```

```
        path[ It(path, j.next(), j).prev_index() ] = i.next();

        path[i.next_index()] = j;
        path[j.next_index()] = i;

        return true;
    }
    return false;
}
```

A check(std::vector<int> &path, It i, It j) függvény ellenőrzi, hogy a csere javít-e a költségen vagy sem. Ha igen, akkor egyből át is írja a path változóban szereplő túrát, majd visszatér true-val, egyébként pedig false-szal.

Két ciklussal úgy megyünk végig a túrán, hogy legalább 2 távolság legyen a két iterátor között, mivel egymás mellett lévő éleket nem lehet cserélni (mivel van egy közös pontjuk). Minden lépésben lefuttatjuk a check() függvényt. Ha igazzal tér vissza, akkor újrakezdjük a keresést. Addig fut az algoritmus, amíg nem találtunk több javítható élpárt.

3.6.3. Általános leírástól való eltérések

A segítő névtér osztály neve: opt2_helper.

A default konstruktoron kívül még egy konstruktor van:

```
Opt2Tsp(const FullGraph &gr, const CostMap &cost,
        const std::vector<Node> &path);
```

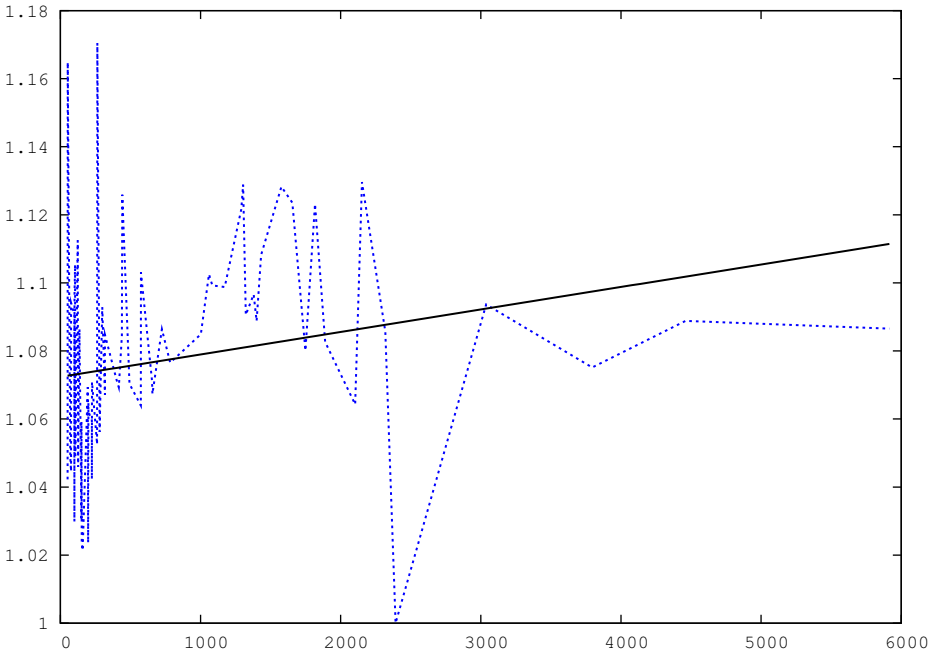
Ez a konstruktor egy harmadik paramétert is fogad, amely egy túrát tartalmaz. Ebben az esetben innen indítja az algoritmust, egyébként pedig a csúcsokat sorba rendezi az indexük alapján, és az így kapott túrából indul.

3.6.4. Hatékonyságelemzés

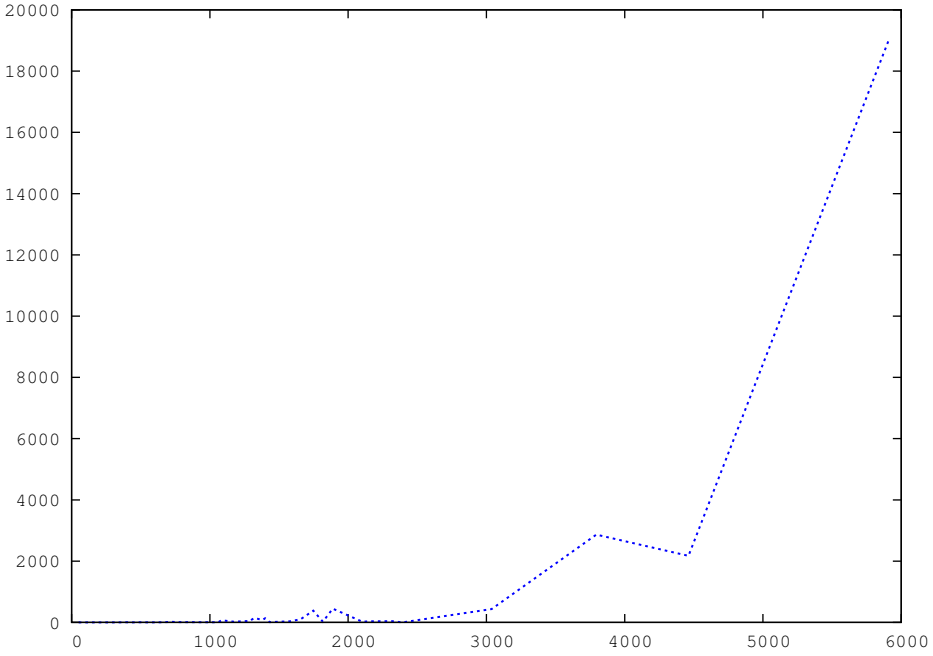
A TSPLIB eredményeket az előzőekhez hasonló módon a 16. ábra, a futási időket pedig a 17. ábra mutatja. Ez az algoritmus 6000 város esetén közel 5 órán keresztül futott, tehát ez nagyon költséges számolás. A 9. táblázat tartalmazza a minimális, maximális és átlagos eredményeket.

	Min	**Max**	**Átlag**
Futásidő (másodperc)	0.000000	19077.610000	365,390833
Összköltség/optimum	1.000000	1.170489	1.077811

9. táblázat. Az opt-2 heurisztika eredményei



16. ábra. Az opt-2 heurisztika: TSPLIB eredmények



17. ábra. Az opt-2 heurisztika: TSPLIB futásidők

3.7. Christofides heurisztikája

3.7.1. Tárolt adatok

Az algoritmus működése során tárolnunk kell a megtalált feszítőfát és teljes párosítást, valamint az ezek uniójából adódó nem feltétlenül egyszerű gráfot. Ehhez nem az eredeti FullGraph adatszerkezetet használjuk, hanem egy hatékony általános gráfreprezentációt, a SmartGraph-ot.

3.7.2. Megvalósítás

A feszítőfát a LEMON Kruskal algoritmusával számoljuk. Ezután egy InDegMap és egy FilterNodes segítségével kiszűrjük a páratlan fokú csúcsokat.

A LEMON MaxWeightedPerfectMatching algoritmusával számoljuk ki a minimális költségű teljes párosítást úgy, hogy egy NegMap segítségével negáljuk az élek költségét. A teljes párosításban szereplő éleket hozzáadjuk a SmartGraph-hoz, majd egy EulerIt iterátorral bejárjuk a gráfot. Az EulerIt egy Euler-sétát ad eredményül, amelyben átugorjuk a már túrában szereplő csúcsokat, így kapjuk végül a Hamilton-kört. A Hamilton-kör elemeit (amelyek SmartGraph::Node-ok) egy NodeCrossRef map segítségével képezzük le az eredeti FullGraph::Node-ra és így tároljuk el.

3.7.3. Általános leírástól való eltérések

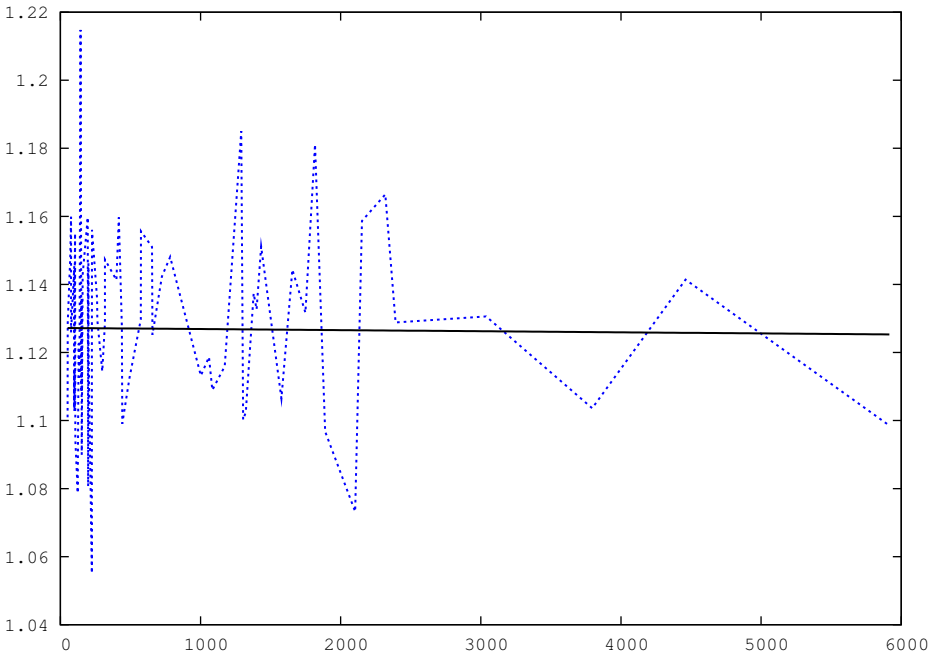
A segítő névtér osztály neve: christofides_helper. Elnevezéseken kívül mindenben megfelel az általános leírásban említetteknek.

3.7.4. Hatékonyságelemzés

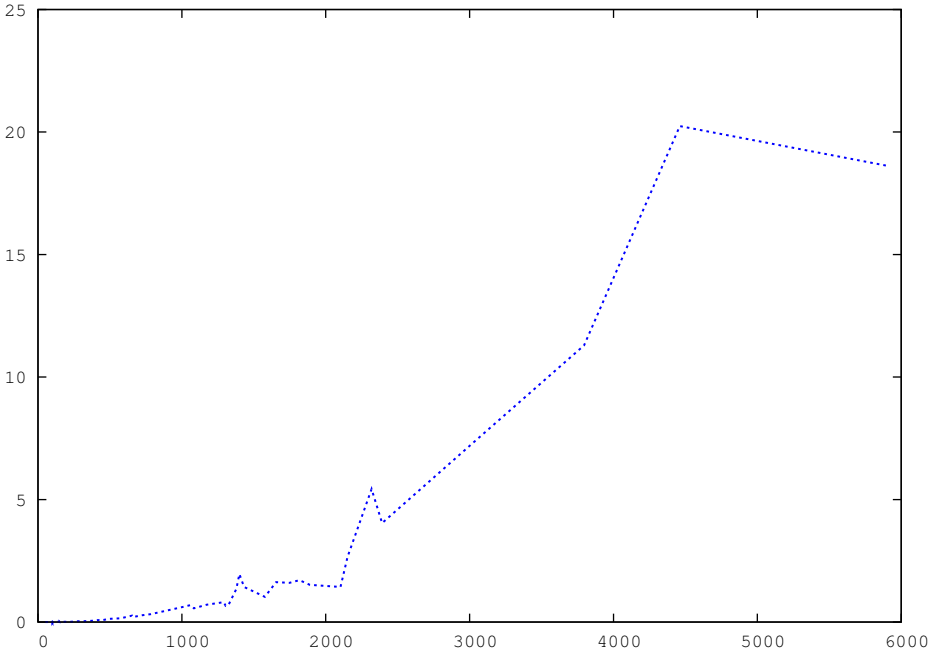
A TSPLIB eredményeket az előzőekhez hasonló módon a 18. ábra, a futási időket pedig a 19. ábra mutatja. Ez az algoritmus a legnagyobb példákra is mindössze 20 másodpercig futott, tehát egy elég gyors módszernek bizonyult. Megjegyezzük, hogy a hatékonysága elsősorban a LEMON-ban már megvalósított Kruskal és MaxWeightedPerfectMatching algoritmusok hatékonyságától függ. A 10. táblázat tartalmazza a minimális, maximális és átlagos eredményeket.

	Min	**Max**	**Átlag**
Futásidő (másodperc)	0.000000	20.240000	1.252083
Összköltség/optimum	1.055123	1.214787	1.126948

10. táblázat. A Christofides heurisztika eredményei



18. ábra. A Christofides heurisztika: TSPLIB eredmények



19. ábra. A Christofides heurisztika: TSPLIB futásidők

3.8. Összehasonlítás

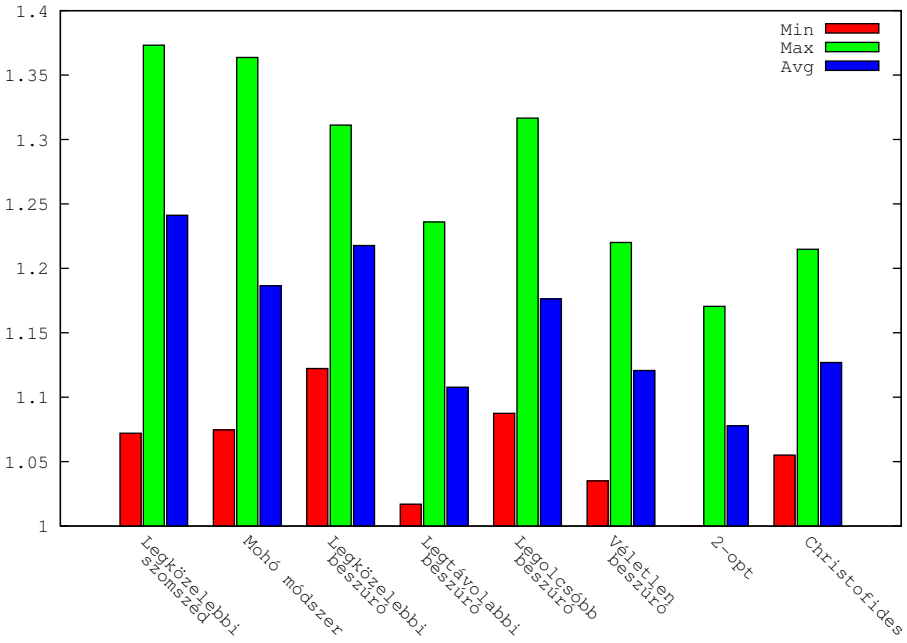
A 20. összehasonlító ábra és a 11. összefoglaló táblázat alapján egyértelműen látszik, hogy a 2-opt heurisztika adta a legjobb közelítéseket, de ezért futásidőben igen komoly árat kell fizetni. Összességében a legeredményesebbnek talán a legtávolabbi pontot beszúró heurisztikát vehetjük, mivel viszonylag alacsony futásidő mellett (kb. 20-szor gyorsabb mint a 2-opt) a legjobb átlagos közelítési faktortól csak 3%-kal maradt le.

A legközelebbi szomszéd, a mohó módszer, a véletlen pont beszúrása és a Christofides heurisztikája hasonlóan rövid idő alatt adtak eredményt, így ha sebességkritikus alkalmazási területen kell használni (és a legtávolabbi pont beszúrás heurisztika túl lassúnak bizonyul), akkor ezek közül érdemes választani. A véletlen pont beszúró heurisztika előnye lehet, hogy implementálni sokkal egyszerűbb, mint a Christofides algoritmust. A 2-opt akkor jó választás, ha az idő nem elsődleges tényező, de pár százaléknyi javulás is nagyon hasznos lehet.

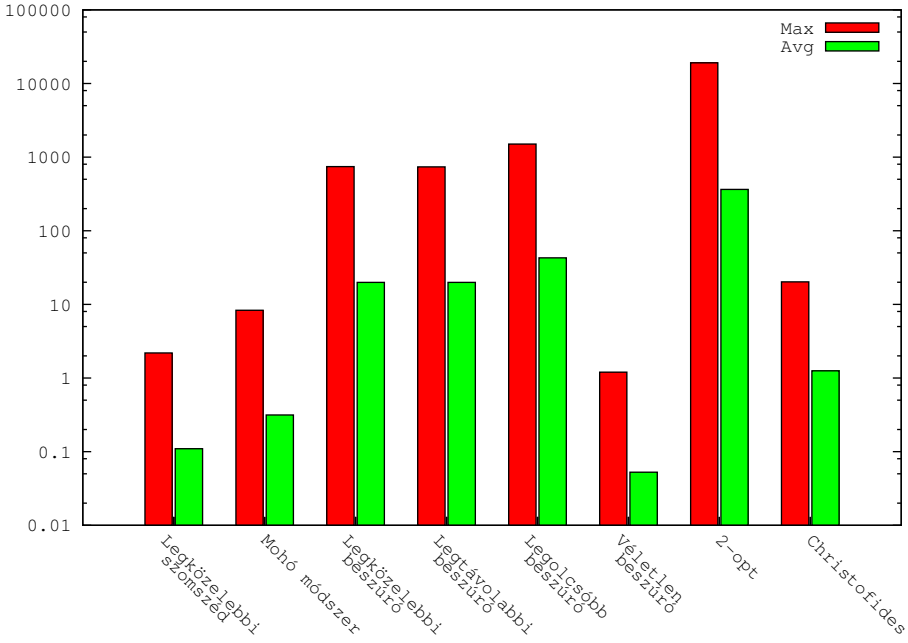
Algoritmus neve	Átlagos közelítési faktor	Átlagos futásidő
Legközelebbi szomszéd	1.241208	0.109722
Mohó módszer	1.186397	0.315416
Legközelebbi pont beszúrása	1.217736	19.932777
Legtávolabbi pont beszúrása	1.107729	19.875277
Legolcsóbb pont beszúrása	1.176176	42.924166
Véletlen pont beszúrása	1.120734	0.052500
2-opt heurisztika	1.077811	365,390833
Christofides-algoritmus	1.126948	1.252083

11. táblázat. Összefoglaló táblázat

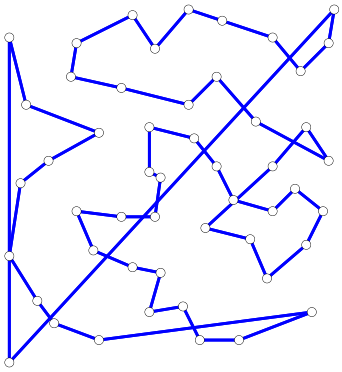
A 22. ábrán láthatóak a TSPLIB eil51 nevű problémájához tartozó 8 algoritmus által adott eredmények. A legközelebbi szomszéd és a mohó módszer hibái nagyon jól látszanak: a végső kényszerből behúzott él átszeli az egész területet. A 2-opt és a legtávolabbi beszúrás heurisztika szemmel láthatóan jobb eredményt adott mint a többi.



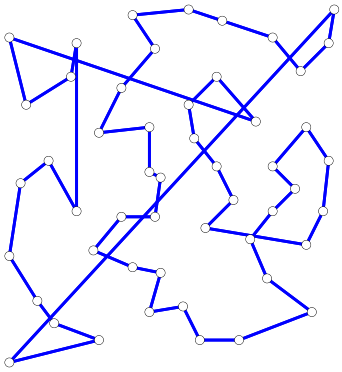
20. ábra. Összehasonlítás TSPLIB mérőszám alapján



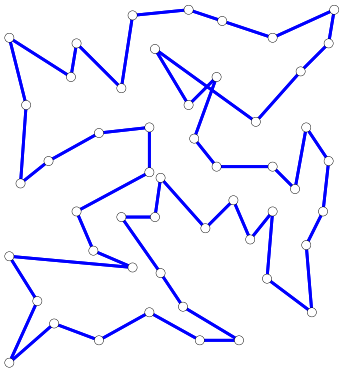
21. ábra. Összehasonlítás futásidő alapján



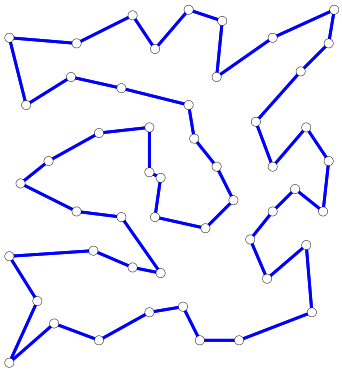
Legközelebbi szomszéd



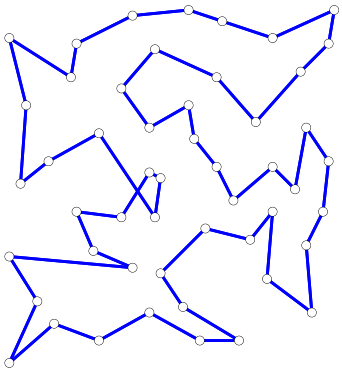
Mohó módszer



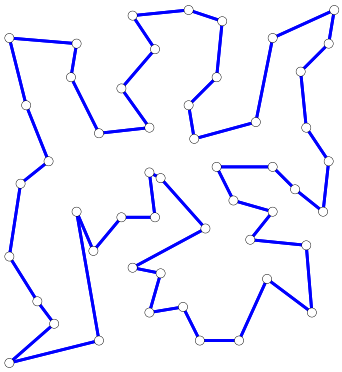
Legközelebbi beszúró



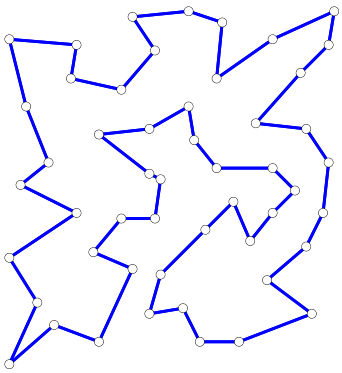
Legtávolabbi beszúró



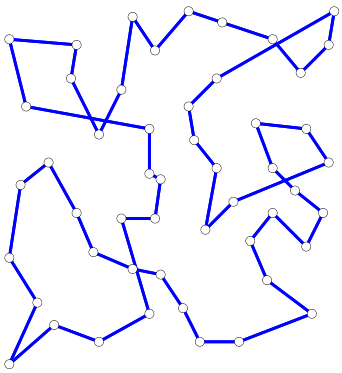
Legolcsóbb beszúró



Véletlen pont beszúrása



2-opt



Christofides

22. ábra. TSPLIB eil51 közelítései

Hivatkozások

[1] http://en.wikipedia.org/wiki/Travelling_salesman_problem, 2010-05-10

[2] David L Applegate, Robert E Bixby, William J Cook: The Traveling Salesman Problem: A Computational Study, Princeton University Press, 2006, [593], ISBN: 069 1129 93 2

[3] Jordán Tibor, Recski András, Szeszlér Dávid: Rendszeroptimalizálás, Typotex Kiadó, 2004, [189], ISBN: 963 9548 39 1

[4] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/, 2010-05-13

[5] http://lemon.cs.elte.hu/ 2010-06-09

[6] http://lemon.cs.elte.hu/pub/tutorial/, 2010-06-09

[7] http://lemon.cs.elte.hu/pub/doc/latest/index.html, 2010-06-09

[8] http://lemon.cs.elte.hu/trac/lemon/wiki/Downloads, 2010-06-09

[9] http://download.opensuse.org/repositories/science/, 2010-06-09

[10] http://lemon.cs.elte.hu/pub/doc/1.2/a00515.html, 2010-06-10

[11] http://lemon.cs.elte.hu/pub/doc/1.2/a00245.html, 2010-06-10