Új ütemező algoritmusok a magas szintű szintézisben[1]

Orbán András, Műszaki informatika szak, V. évfolyam
Mann Zoltán Ádám, Műszaki informatika szak, V. évfolyam

Konzulens: Dr. Arató Péter
Irányítástechnika és informatika tanszék

Budapest, 2000. október 18.

[1] Ez a dolgozat a Budapesti Műszaki és Gazdaságtudományi Egyetem Villamosmérnöki és Informatikai Karán a 2000. évi Tudományos Diákköri Konferencián 1. helyezést ért el; a 2001. évi Országos Tudományos Diákköri Konferencián szintén 1. helyezést ért el.

Tartalom

1. Bevezetés

A magas szintű szintézis alapfeladata egy magas szinten specifikált algoritmushoz az azt megvalósító optimális hardver illetve szoftver struktúra automatikus megtervezése. Ennek során az egyik kulcsfontosságú feladat az ütemezés, amely egy elemi műveleti gráfok segítségével megfogalmazott kombinatorikus optimalizálási probléma. Dolgozatunk e probléma megoldásával foglalkozik. Fő önálló eredményeink: két heurisztikus algoritmus a probléma megoldására és ezek konkrét megvalósítása, valamint két kapcsolódó tétel, melyek bizonyítását a Függelékben közöljük.

A dolgozat felépítése a következő: először röviden körvonalazzuk a magas szintű szintézis és azon belül az ütemezés feladatát, valamint általános elveket a kombinatorikus optimalizálás témaköréből, majd rátérünk algoritmusaink ismertetésére. Az implementációs részletek bemutatása után ipari benchmark feladatok segítségével összevetjük programjainkat egymással és más, eddig használt megoldásokkal.

2. A magas szintű szintézis

Dolgozatunknak nem célja a magas szintű szintézis szerteágazó témakörének alapos ismertetése. Csupán azon részletek bemutatására törekszünk, amelyek szükségesek a későbbiekben az ütemezés problémájának definiálásához, illetve érzékeltetik annak fontosságát. Részletesebb ismertetés tekintetében az irodalomjegyzékben megadott [Gajski1992], [Camposano1990/1991], [IEEE1993], [Jerraya1998] munkákra utalunk.

2.1. A feladat

A magas szintű szintézis (angolul *High-Level Synthesis*, gyakori rövidítéssel HLS) alapfeladata egy magas szinten specifikált algoritmushoz az azt megvalósító optimális hardver illetve szoftver struktúra automatikus megtervezése.

Ebben a definícióban több tisztázandó részlet van. Először is, a magas szintű specifikáció alatt pl. egy programozási nyelvre, pszeudokódra stb. gondolhatunk. Másodszor, nem világos, mit jelent az optimalitás. Nos, az optimalizálás különböző szempontok alapján történhet, például hardver esetén a megvalósításban szereplő processzorok száma, a chip mérete stb.

Hátra van még annak az értelmezése, hogy "hardver illetve szoftver" struktúrát tervezünk. Ez abból fakad, hogy az optimális hardver struktúra kialakítása hasonló problémákat vet fel, mint például egy optimalizáló fordító készítése, így hasonló eljárások használhatóak a hardver illetve a szoftver struktúra optimalizálására. Ezáltal lehetővé válik hardver és szoftver együttes tervezése (*Hardware-Software Codesign*), sőt, nem kell előre eldönteni, hogy a rendszer mely részeit kell szoftverben és melyeket hardverben megvalósítani, hanem ez a döntés is része lehet a számítógéppel támogatott optimalizálásnak.

2.2. A PIPE

Számos olyan eszköz van már a piacon, amely a hardver-leírásból automatikusan elkészíti magát a hardvert. Azonban egy adott magas szintű leíráshoz sok különböző minőségű hardver struktúra tartozhat, és ezek közül az optimális megtervezésére nem nagyon van támogatás. Azonban ahogy az iparban egyre rövidebb idő alatt egyre komplexebb rendszereket kell létrehozni, úgy válnak egyre költségesebbé a hibás ill. szuboptimális tervezési megoldások, így egyre nagyobb igény van a tervezés minél szélesebb körű számítógépes támogatására.

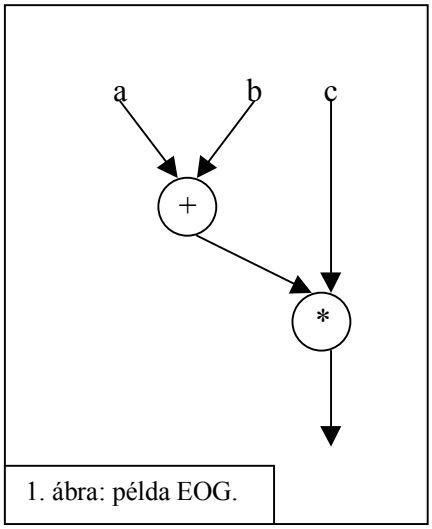
Ezért úttörő jelentőségű az Irányítástechnika és Informatika Tanszéken kifejlesztett PIPE rendszer [Arato1994/2000], mely éppen ezt a folyamatot automatizálja.

A PIPE nagyon fontos tulajdonsága, hogy támogatja a futószalag (*pipeline*) üzemmódú rendszerek tervezését is. Ezek nagyon jelentősek, mivel rosszul párhuzamosítható feladatokat is hatékonyan tudnak gyorsítani, amennyiben azokat többször kell elvégezni. Az ilyen rendszereket az jellemzi, hogy az egyes feladatok elvégzéséhez szükséges időnél kisebb időközönként kapják az újabb feladatot, így a rendszer egyes részei még az előző feladat végével vannak elfoglalva, amikor más részek már az új feladaton dolgoznak. Érezhető, hogy az ilyen rendszerek tervezése külön körültekintést igényel.

A pipeline üzemmód kapcsán két fogalom jelentését kell tisztázni. Az egyik az egyetlen feladat végrehajtási ideje, vagyis az az időtartam, ami a bemenetek megadása és a kimenet létrejötte között eltelik. Ennek jele L (az angol *latency*-ből). A pipeline rendszer másik fontos idő dimenziójú paramétere az újraindítási idő: ilyen időközönként adunk új bemenetet, vagyis egy új feladatot a rendszernek. Ennek a jele R (az angol *restart time*-ból). E jelölések segítségével azt mondhatjuk, hogy akkor beszélünk pipeline működésről, ha $R<L$. (Tulajdonképpen a nem pipeline eset is a pipeline határesetének tekinthető, ti. amikor $R=L$.) Ipari méretekben egyre inkább az válik lényegessé, hogy a rendszer adott idő alatt minél több feladatot tudjon elvégezni, nem pedig az, hogy az egyes feladatok elvégzése mennyi ideig tartott. Ez azt jelenti, hogy az R-et kell csökkenteni, adott esetben az L vagy a rendszer költségének növekedése árán is.

2.3. Az elemi műveleti gráf

Az optimalizálás során kulcsfontosságú szerepet játszik az elemi műveleti gráf (*Elementary Operation Graph*, EOG). Ez teszi lehetővé, hogy egyáltalán formálisan definiálni tudjuk a feladatot. Nézzünk erre egy példát: vegyük azt a nagyon egyszerű algoritmust, amely az a, b és c számokból elkészíti az $(a+b)*c$ számot. Az ebben szereplő elemi műveletek (az összeadás és a szorzás) lesznek az elemi műveleti gráf csúcsai, az élek pedig az adatok áramlásának felelnek meg. (1. ábra)



1. ábra: példa EOG.

Természetesen ez egy nagyon egyszerű példa, azonban sokkal bonyolultabb eljárásokat is lehet így modellezni. Például mi a programjaink tesztelésére a gyors Fourier transzformáció (FFT) valamint számos kriptográfiai eljárás (IDEA, RC6, MARS) elemi műveleti gráfját használtuk. Ezek némelyike több száz csúcsból áll.

Szinkron rendszerekben gondolkodunk, vagyis feltételezzük egy központi óra jelenlétét, amely minden egyes művelet számára az órajelet biztosítja. Ennek megfelelően az egyes műveletek végrehajtási idejét is órajel-ciklusokban adjuk meg. Pontosabban minden műveletnek van egy típusa, és az egyes típusokhoz lehet végrehajtási időt rendelni. Ezen kívül minden művelethez tartozik egy indítási idő, mely azt az órajel-ciklust adja meg, amikor az adott művelet (egy feltételezett vezérlőjel hatására) működésbe lép. (Az indítási idő előre nem ismert, éppen az ütemezés feladata lesz ennek meghatározása.) Az i. művelet végrehajtási idejét d_i (*duration*), indítási idejét s_i (*starting time*) jelöli. E jelölésekkel tehát az adott művelet s_i-től s_i+d_i-ig dolgozik.

Megjegyzés: mivel az elemi műveleti gráf csúcsai a modellezett algoritmusban szereplő műveleteknek felelnek meg, így a továbbiakban a “csúcs” és “művelet” szavakat gyakran egymás szinonimájaként fogjuk használni.

A rendszer helyes működését az alábbi axiómákkal lehet megfogalmazni: [Arato2000]

- A v csomópontnak megfelelő művelet csak akkor kezdheti meg működését, ha minden olyan u csúcshoz tartozó művelet befejeződött már, amire van $u \rightarrow v$ él.
- Az i. műveletnek a teljes működési ideje (d_i) alatt szüksége lehet a bemeneteire, ezért azok ezalatt nem változhatnak.
- Az i. művelet a teljes működési ideje (d_i) alatt változtathatja a kimeneteit.

– Az *i*. művelet kimenete a működési ideje végétől kezdve egészen a következő működésbe lépéséig változatlan.

A PIPE tevékenységének egyik része azzal kapcsolatos, hogy ezen axiómák megsértése nélkül úgy módosítsa az elemi műveleti gráfot, hogy az egy előre megadott *R* újraindítási idővel tudjon működni. Ennek során puffereket illeszt be a gráfba, illetve egyes műveleteket megtöbbszöröz. (A puffer szerepe: egy adott művelet kimenetét tárolja annak érdekében, hogy a művelet újrakezdhesse működését. Egy olyan elemi műveletnek tekinthető, melynek végrehajtási ideje 1.) Ezen kívül, amennyiben az eredeti EOG tartalmaz köröket, akkor azokat felbontja. Ezzel itt most részletesen nem foglalkozunk, a részletes leírás megtalálható [Arato2000]-ben. Ezután jön azonban a tulajdonképpeni optimalizálás: az ütemezés és az allokáció. Ezek tárgyalása előtt azonban még néhány fogalmat ismertetnünk kell.

Az elemi műveleti gráfban a bemenetekből a kimenetekbe menő utak között mindig van leghosszabb – esetleg több is. (Egy út hosszán a benne szereplő csúcsok végrehajtási idejének összegét értjük.) Ez az úgynevezett kritikus út: e mentén alakul ki a rendszer *L* működési ideje. Amennyiben több leghosszabb út van, akkor ezek együttese alkotja a kritikus részgráfot, ami ebben az esetben tehát már nem út. Ezért nem is a kritikus út, hanem a kritikus részgráf elnevezést használjuk.

Amennyiben az *L*-et nem akarjuk növelni, akkor a kritikus részgráfban lévő műveletek indítási ideje egyértelműen meg van határozva: ezeknek rögtön el kell kezdeniük dolgozni, amint minden bemenetük rendelkezésre áll. Ha ugyanis ezek közül valamelyik művelet késne, az az egész rendszer csúszásához vezetne. Nagyon hasonló ez a projekt menedzsmentben alkalmazott PERT (*Project Evaluation and Review Technique*) és CPM (*Critical Path Method*) módszerekhez (ld. 4.1.5. fejezet).

Azonban vannak olyan műveletek, amelyek indítási ideje nem egyértelmű, mivel nincsenek a kritikus részgráfban. Ezekről csak annyit tudunk meghatározni, hogy mi az a legkorábbi (ASAP, *As Soon As Possible*) illetve legkésőbbi (ALAP, *As Late As Possible*) időpont, amikor el lehet, illetve el kell indítani őket. Az ASAP annak az időpontnak felel meg, amikor már minden bemenet rendelkezésre áll, az ALAP pedig annak, amikor már a kimenet előállítását mindenképp meg kell kezdeni, nehogy az *L* növekedjen.

Az is elképzelhető, hogy a rendszer tervezője belemegy egy olyan kompromisszumba, hogy a rendszer költségének csökkentése érdekében megengedi *L* kismértékű növekedését. Ebben az

esetben kritikus út nem lesz; minden csúcs mobilitási tartománya (vagyis az [ASAP, ALAP] intervallum) annyival nő, amennyivel az *L*.

3. Az ütemezés problémája

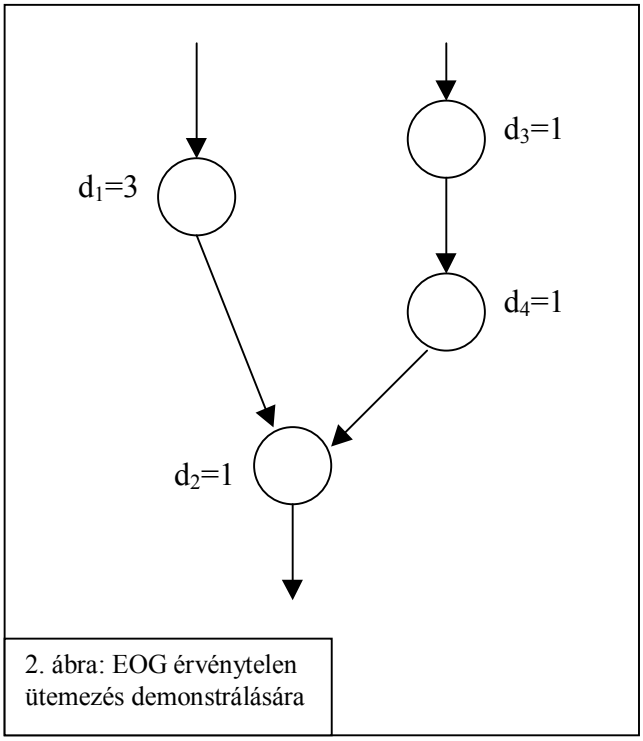
3.1. Ütemezés és allokáció

Az ütemezés (*scheduling*) feladata nehezen választható szét az allokációtól, ezért először ezeket együtt tárgyaljuk, majd később megmutatjuk, hogy hogyan és miért érdemes ezeket külön kezelni.

Ütemezésen az egyes elemi műveletek indítási idejének egy lehetséges lerögzítését értjük. [Hwang1991] ASAP és ALAP között általában nagyszámú különböző érvényes ütemezés képzelhető el, amelyek egyaránt az eredeti feladat megoldásai, azonban költségükben igen különbözőek lehetnek. Ezért van szükség optimalizálásra.

Költség alatt a fizikai megvalósítás költségét értjük. Erre egy jó mérce a felhasznált processzorok száma. Pontosabban, mivel az egyes processzor-típusok ára eltérő lehet, így az egyes processzor-típusokból felhasznált processzorok számának a megfelelő processzor-típus árával vett súlyozott összege adja a teljes rendszer költségét. (Ezek a súlyok nem csak tényleges árat reprezentálhatnak, hanem más költségtényezőket is, pl. hőtermelés, méret stb.) Kérdés viszont, hogy egy adott ütemezésből hogyan látszik, hogy a megvalósításban majd hány processzorra lesz szükség? Ez egy igen nehéz kérdés; ennek kiszámítására szolgál az allokáció. Ennek során meg kell próbálni minél több műveletet közös processzorban megvalósítani. Figyelembe kell azonban venni, hogy ha van olyan időpillanat, aminek során az adott ütemezésben két elemi művelet egyszerre dolgozik, akkor azok nem allokálhatók közös processzorba. Általában, ha két művelet nem kerülhet ugyanabba a processzorba, akkor *konkurens*, egyébként *kompatíbilis* csúcs-párról beszélünk.

Még hátra van annak tisztázása, hogy mit is jelent a feladat definíciójában szereplő "érvényes ütemezés". Ehhez azt kell meggondolni, hogy az egyes műveletek indítási idejét nem lehet egymástól függetlenül ASAP és ALAP között tetszőlegesen megválasztani. Például az alábbi gráf esetén:



2. ábra: EOG érvénytelen ütemezés demonstrálására

L=4; az egyes csúcsok mobilitási tartományai:

$ASAP_1=0$, $ALAP_1=0$

$ASAP_2=3$, $ALAP_2=3$

$ASAP_3=0$, $ALAP_3=1$

$ASAP_4=1$, $ALAP_4=2$

Azonban ha például a 3. és 4. csúcsot egyaránt az 1. időpillanatban indítanánk, ez ellentmondana az axiómáinknak, hiszen a 4. csúcs működéséhez már szükség lenne a 3. csúcs kimenetére. Vagyis nem elég csupán arra figyelni, hogy minden csúcs indítási idejét az [ASAP, ALAP] intervallumból válasszuk, hanem az axiómákból eredő korlátokat explicit módon be kell tartanunk. Egy ütemezést tehát akkor nevezünk érvényesnek, ha nem sérti meg e korlátokat. Abban biztosak lehetünk, hogy ha minden csúcsot az ASAP idejére ütemezünk, akkor egy érvényes ütemezést kapunk, hiszen ez következik abból, ahogy az ASAP időket meghatározzuk. Hasonlóan, akkor is érvényes ütemezést kapunk, ha minden csúcsot az ALAP időre rögzítünk.

Összefoglalva tehát: az allokáció feladata, hogy egy adott ütemezés esetén az elemi műveletekhez fizikai processzorokat rendeljen, törekedve arra, hogy a felhasznált processzorok összköltsége minimális legyen. Az ütemezés feladata az elemi műveletek kezdési időpontjainak olyan lerögzítése, amire az allokáció minimális költséget tud majd elérni.

3.2. A feladat nehézsége

Az eddigiekből látható, hogy az ütemezés és az allokáció feladata szorosan kapcsolódik egymáshoz. Azért érdemes mégis külön kezelni őket, mert a két probléma együtt túl nehéz. Ez a dekompozíció teszi lehetővé, hogy egyáltalán közelítő megoldásokat tudjunk adni. Az allokáció tulajdonképpen egy ismert NP-teljes probléma, új köntösben: ekvivalens egy gráf (az ún. konkurencia-gráf) kromatikus számának meghatározásával (ld. 4.1.2. fejezet). Ebből persze következik az is, hogy az ütemezés és allokáció együttes feladata is NP-teljes. De NP-teljes problémák között is vannak jelentős különbségek – legalábbis a közelítő megoldások nehézségének tekintetében –, és éppen az adja a dekompozíció értelmét, hogy az allokációra léteznek hatékony közelítő algoritmusok.

Azonban sajnos az allokáció – bár az ütemezésnél lényegesen gyorsabb – még mindig túl lassú ahhoz, hogy az ütemezésben állandóan szubrutinként lehetne hívogatni az egyes ütemezések költségének megállapítására. Ezért szükség van egy olyan mennyiségre, amelynek kiszámítása gyorsabb, mint a processzorok számáé, de aminek optimalitása nagy valószínűséggel a processzorok számának optimalitását is maga után vonja.

Erre a kompatíbilis párok száma tűnt a legjobb választásnak, hiszen ha sok kompatíbilis pár van, akkor joggal remélhető, hogy a szükséges processzorok száma is alacsony lesz. (Mint később, az eredmények értékelésénél látni fogjuk, ez a hipotézis tényleg jól működik a gyakorlatban.) Ez másképp megfogalmazva azt jelenti, hogy arra törekszünk: a konkurencia-gráfban minél kevesebb él legyen. Valóban, egy ritkább gráfnak valószínűleg a kromatikus száma is kisebb. Tehát az ütemezési feladatot a következőképp fogalmazhatjuk meg: adott elemi műveleti gráf és ASAP, ALAP idők mellett keressük azt az érvényes ütemezést, amelynél a kompatíbilis művelet-párok száma maximális. Pontosabban, ha az egyes processzortípusokra költségtényezők vannak definiálva, akkor ennek megfelelően az egyes típusokon belüli kompatíbilis művelet-párok számának súlyozott összegét kell maximalizálni.

Egyik elméleti eredményünk abban áll, hogy bebizonyítottuk: ez is egy NP-teljes feladat. Ennek bizonyítása a Függelékben megtalálható. Az NP-teljesség miatt nyilván csak közelítő megoldást keresünk; rendszerint úgysem fontos, hogy az abszolút optimális megoldást megtaláljuk, csak hogy egy minél jobbat. Viszont, mivel nagy méretű bemeneti gráfokat is kell tudnunk kezelni, így a teljes keresési tér bejárása szóba sem jöhet.

Itt jegyezzük meg, hogy mivel a PIPE-ban a tulajdonképpeni optimalizálás az ütemezőben zajlik, így csakúgy végrehajtási idő, mint az eredmény minősége szempontjából az ütemezés

tekinthető az egész magas szintű szintézis kritikus pontjának. Valóban, nagy méretű bemeneti gráfok esetén a PIPE futási ideje lényegében megegyezik az ütemező futási idejével. Ez a tény indokolja, hogy miért érdemes új ütemező algoritmusokat keresni.

3.3. A kompatibilitás kezelése

Mivel feladatunk a kompatíbilis párok számának maximalizálása, így fontos tisztázni, hogy ezt a számot hogyan tudjuk megállapítani. Pontosabban egy olyan képletet fogunk most ismertetni, amely megadja, hogy két művelet kompatíbilis-e. Előbb azonban néhány megjegyzés: először is, kompatibilitást csak azonos típusú műveletekre kell vizsgálni. Ugyanis különböző típusú műveleteket sosem lehet ugyanahhoz a processzorhoz rendelni. Ez persze feltételezi, hogy minden processzor csak egyfajta műveletet tud elvégezni. Ha ez nem így van, a műveletekhez két fajta típust kell hozzárendelni. Az egyik típus azt adja meg, hogy az adott művelet mennyi ideig tart, míg a másik azt, hogy milyen fajta processzor tudja az adott műveletet elvégezni. Például egy művelet első értelemben vett típusa lehet az, hogy "16 bites szorzás", az utóbbi értelemben vett típusa pedig "aritmetikai művelet". Ilyenkor tehát a fenti kijelentés úgy módosítandó, hogy ha két műveletnek az utóbbi értelemben vett típusa különbözik, akkor semmiképp sem kerülhetnek ugyanabba a processzorba, így a köztük lévő kompatibilitást nem kell vizsgálni.

Egy másik megjegyzés: ha nem engednénk meg a pipeline üzemmódot, a kompatibilitás vizsgálata sokkal egyszerűbb lenne. A pipeline üzemmód azonban ezt bonyolultabbá teszi, hiszen így az elemi műveleti gráfban egymástól igen távol eső csúcsok is konkurenssé válhatnak. Itt említjük meg, hogy ha nem engednénk meg pipeline üzemmódot, az allokáció problémája sem lenne NP-teljes. Ugyanis ekkor a keletkező konkurencia-gráf úgynevezett intervallum-gráf lenne, amelynek színezésére már van hatékony algoritmus.

Térjünk tehát rá annak a képletnek az ismertetésére, amely pipeline üzemmód esetén is eldönti, hogy két csúcs kompatíbilis-e. Ehhez szükségünk lesz egy új fogalomra, a foglaltsági időre. Ez azt adja meg, hogy az adott művelet mennyi időre foglalja le az őt megvalósító processzort. Jele q_i. A foglaltsági idő mindig nagyobb a működési időnél, mert a foglaltságba beletartozik az az idő is, amikor az adott művelet már befejeződött ugyan, de még tartania kell a kimenetét annak érdekében, hogy azt más műveletek fel tudják használni. Ennek megfelelően $q_i=max(d_i+d_j)$, ahol a maximumot azon j csúcsokon kell venni, amelyekbe megy i-ből él. Ennek során azt is figyelembe kell venni, hogy ha egy j csúcs nem közvetlenül az i befejeződése utánra van ütemezve, akkor ez úgy értendő, hogy az i-t nem közvetlenül a j,

hanem egy puffer követi, melynek végrehajtási ideje 1. Tehát ilyenkor a maximum képzésekor d_i+1-et kell figyelembe venni.

Ennyi előkészület után már megadhatjuk a képletet, mely két csúcs kompatibilitását adja. Az *i*. és *j*. csúcs akkor és csak akkor konkurens (vagyis nem kompatíbilis), ha van olyan *K* egész szám, melyre $(s_i-s_j-q_j)/R<K<(s_i-s_j+q_i)/R$. [Arato2000] (A többszörözött csúcsok explicit kezelése esetén ez a képlet még bonyolultabb lenne, azonban a PIPE nem ezt a módszert követi.)

3.4. Korábbi megoldások és a PIPE

A szakirodalom alapvetően három különböző ütemező eljárásról ír, de általában egyikkel sincsen maradéktalanul megelégedve.

Az első megoldás egészértékű programozáson (ILP, *Integer Linear Programming*) alapul. Ennek lényege, hogy a feladatot egészegyüttható lineáris egyenlőtlenségekkel írják le, ilyen módon lineáris programozási feladatként definiálva azt. Ezután egy standard lineáris programozási eljárással (pl. szimplex módszer, [Schrijver1998]) keresik az optimumot. Ennek a módszernek fő hátrányai, hogy egyrészt nehezen becsülhető az algoritmus futási ideje, másrészt pedig nem teszi lehetővé a mérnöki tudás beépítését a keresésbe.

A másik megoldás az úgynevezett listás ütemezések családja. Ezek általában egyszerű és gyors eljárások, amelyek egyetlen egyszer haladnak végig vagy az idő tartományon, vagy a műveletek halmazán, és minden időpillanatra igyekeznek optimális számú művelet működését garantálni. A műveletek kiválasztásának sorrendjénél rendszerint valamilyen heurisztikát alkalmaznak, ami csökkenti azt a hátrányt, hogy egy rossz döntést később nem tudnak korrigálni. Ezeknek az eljárásoknak a legfőbb hibája, hogy általában nem adnak elég jó eredményt.

A harmadik megoldás az úgynevezett erő-vezérelt (*force-directed*) ütemező [Paulin1989]. Ez arra törekszik, hogy minden időpillanatban körülbelül ugyanannyi konkurens művelet legyen. Nevét onnan kapta, hogy az optimálistól való eltéréssel arányos mértékben igyekszik megváltoztatni a helyzetet, ami nagyon hasonlít a rugalmasságtanban alkalmazott Hooke-törvényre. Tudomásunk szerint ez a jelenleg ismert legjobb ütemező algoritmus, bár hátránya, hogy lépésszáma a bemenet méretének harmadik hatványával arányos, így nagy méretű bemeneti gráfok esetén nagyon lassú lehet.

A PIPE jelenleg egy force-directed ütemezőt tartalmaz. Mivel ez a bemenetét egy szöveges file-ban kapja, és a kimenetét is egy szöveges file-ba írja, lecserélhettük a saját valamelyik ütemező programunkra. Az algoritmusaink ismertetése után majd bemutatjuk a PIPE eredeti force-directed ütemezőjével való összehasonlítás eredményét.

A bemenő file formátuma a következő. Minden egyes csúcsnak egy sor felel meg, melynek felépítése:

csúcs_neve csúcs_típusa d_i $ASAP_i$ $ALAP_i$ *[elődök_listája] [utódok_listája]*

A kimeneti file formátuma nagyon hasonló, csak éppen ASAP és ALAP helyett az ütemezett indítási idő értékét tartalmazza.

4. Kombinatorikus optimalizálás

Mint arról már korábban szó esett, az ütemezés ill. az allokáció problémája egy kombinatorikus optimalizálási feladat, és sok hasonlóságot mutat más optimalizálási problémákkal, amelyeknek már hatalmas szakirodalmuk van. [Jordan1995] Ennek megfelelően kialakult már számos általános célú optimalizálási algoritmus, amely e problémák közül sokra jól alkalmazhatónak bizonyult. Ezért érdemes röviden áttekinteni, hogy milyen hasonló problémák kerültek elő, és milyen tipikus módszerek alakultak ki ezek megoldására. Természetesen az itt előkerülő néhány probléma és algoritmus csak igen kis részét fedi le a kombinatorikus optimalizálás szerteágazó területének.

4.1. Rokon problémák

4.1.1. Ládapakolás

Klasszikus kombinatorikus optimalizálási feladat. Vannak V térfogatú ládáink, és V_1, V_2, …, V_k térfogatú tárgyaink (minden i-re $V_i \leq V$). Nyilvánvalóan minden ládába legfeljebb V össztérfogatig helyezhetünk tárgyakat. A feladat az, hogy minél kevesebb ládába pakoljuk be az összes tárgyat.

Ez a feladat, ha elsőre nem is látszik, hasonló a mi feladatunkhoz. Tegyük fel ugyanis, hogy V és minden V_i egész, és képzeljünk el egy olyan elemi műveleti gráfot, amiben k csúcs van és nincsenek élek. Legyen továbbá a latency éppen V, az egyes csúcsokhoz tartozó végrehajtási idők pedig éppen a V_i számok. Feleltessük meg továbbá a processzoroknak, amikbe allokálunk, a ládákat, amikbe pakolunk. E megfeleltetésekkel tehát a ládapakolás átmegy az ismertetett gráfra vonatkozó együttes ütemezési és allokációs problémába. Ebből arra következtethetünk, hogy az ütemezési és allokációs probléma legalább olyan nehéz, mint a ládapakolás. Persze lehet, hogy sokkal nehezebb, hiszen egy igen egyszerű elemi műveleti gráfra vonatkozó ütemezés és allokáció bizonyult ekvivalensnek a ládapakolással.

4.1.2. Gráfszínezés

Szintén klasszikus optimalizálási feladat, lényege a következő. Adott egy irányítatlan gráf. Minden csúcsát meg akarjuk színezni egy-egy színnel úgy, hogy szomszédos csúcsok ne legyenek azonos színűek. Feladat: tegyük ezt meg a lehető legkevesebb szín felhasználásával.

(Azt a legkisebb számot, ahány szín már elég a színezéshez, a gráf kromatikus számának nevezik.)

Ez tényleg megfeleltethető az allokáció problémájának. Készítsük ugyanis el a megütemezett elemi műveleti gráfból azt az irányítatlan gráfot (a konkurencia-gráfot), aminek csúcsai megegyeznek az EOG csúcsaival, de két csúcs akkor van összekötve, ha az EOG-ban konkurensek. Ekkor az allokáció éppen e gráf kromatikus számának meghatározásával illetve az optimális színezés megtalálásával ekvivalens.

4.1.3. Boole-formulák kielégíthetősége

Ez szigorúan véve nem optimalizálási feladat, bár meg lehet akként is fogalmazni. Első sorban azért ismertetjük, mert a Függelékben az ütemezés problémájának NP-teljességét ennek segítségével bizonyítjuk.

A feladat a következő. Adott az x_1, x_2, …, x_k bináris változókból alkotott konjunktív normálforma. Állapítsuk meg, hogy van-e a változóknak olyan behelyettesítése, amelyre a formula értéke 1. (Megfogalmazás optimalizálási feladatként: keressük a változóknak olyan behelyettesítését, amelyre a konjunkcióban szereplő tagok közül a lehető legtöbb értéke 1.) Ez egy közismert NP-teljes probléma, gyakori elnevezése SAT (az angol *satisfiability*-ből). Van egy némiképp leegyszerűsített változata, a 3-SAT, mely azonban szintén NP-teljes: ilyenkor a konjunkció minden tagja egy-egy 3-tagú diszjunkció. [Jordan1995]

4.1.4. Órarendkészítés

Igen összetett optimalizálási feladat. Adva vannak osztályok, tanárok, tantárgyak és tantermek. Adva van továbbá, hogy az egyes osztályoknak mely tantárgyakat kell tanulniuk és milyen óraszámban, az egyes tanárok mely tárgyakat tanítják, és az egyes tantárgyakat mely tantermekben lehet oktatni. Ezen kívül vannak nyilvánvaló, szigorú korlátok, hogy pl. egy tanár nem lehet egyszerre két helyen, egy teremben egyszerre nem lehet két osztály stb. Ezen kívül lehetnek további, enyhébb korlátok illetve preferenciák, pl. ne legyenek nagy lyukak az egyes osztályok tanrendjében, egy heti két nap oktatott tárgy ne két egymást követő napon szerepeljen az órarendben stb. A feladat egy olyan órarend elkészítése, amely a szigorú korlátoknak egytől egyig eleget tesz, valamint az enyhébb korlátok közül is minél többnek. (Az enyhébb korlátok esetleg súlyozva vannak.)

A feladat jellegét tekintve hasonlít az ütemezés problémájára (az elemi műveleti gráf axiómái felelnek meg a szigorú korlátoknak, a kompatíbilis párok számának maximalizálása pedig annak, hogy az enyhébb korlátok közül minél többet kell teljesíteni). Külön érdekesség, hogy angolul az ütemezés és az órarendkészítés egyaránt *scheduling*.

4.1.5. Projekt menedzsment

Már korábban szó esett arról, hogy az elemi műveleti gráfok nagy hasonlóságot mutatnak a PERT és CPM módszerekben alkalmazott gráfokkal. (A szakirodalomban nincs teljes egyetértés abban, hogy a PERT és a CPM közül melyik melyik, illetve hogy egyáltalán két különböző módszerről van-e szó.) Azonban ezek a módszerek polinomidőben tökéletes eredményt adnak, így sejthető, hogy egy egyszerűbb problémát oldanak meg. Valóban, ezek csupán egyetlen projekt ütemezését adják, mégpedig erőforrás-korlátok figyelembe vétele nélkül.

Akkor kapunk az ütemezéshez és allokációhoz hasonló problémát, ha egyszerre több projekt fut (egymástól esetleg időben is eltolva; ez felel meg a pipeline szervezésnek), és a vállalatnak véges erőforrás készlet (emberek, tárgyi erőforrások, tőke) áll rendelkezésre. Ezek után a cél az egyes részfeladatok olyan ütemezése és olyan hozzárendelése az erőforrásokhoz, hogy a projektek minél rövidebb idő alatt, minél kevesebb erőforrással tudjanak lefutni.

4.2. Megoldási alternatívák

Mint már említettük, a kombinatorikus optimalizálás témakörének hatalmas irodalma van, rengeteg algoritmus született már a fenti problémákra és más, hasonló feladatokra. Egyes algoritmusok egész probléma-osztályokra alkalmazhatóak. Minket első sorban NP-teljes problémák közelítő megoldásai érdekelnek. (Mivel az NP-teljesség csak eldöntési problémákra értelmezett, így ez úgy értendő, hogy az adott optimalizálási problémából konstruálható eldöntési probléma NP-teljes.)

Az ilyen megoldások lényege, hogy nem járják be a teljes keresési teret, hanem bizonyos heurisztikák segítségével próbálják megjósolni, hogy mely állapotokat érdemes megvizsgálni. Ennek során vagy valamilyen, az adott problémára jellemző tudást lehet kihasználni, vagy pedig az ember vagy a természet valamilyen általános probléma-megoldó mechanizmusát szokták utánozni. Ebben az összefüggésben már bizonyos mértékű mesterséges intelligenciáról beszélhetünk.

A mi szempontunkból az ilyen algoritmusok egyik legjobb csoportosítása a determinisztikus és nem-determinisztikus algoritmusok megkülönböztetése. (Egyes szerzők más csoportosításokat részesítenek előnyben, illetve ezeket a kategóriákat némileg más értelemben használják.) Egy algoritmus attól determinisztikus, hogy nincsen benne a véletlennek szerepe, így többször lefuttatva ugyanarra a bemenetre garantáltan ugyanúgy fog lefutni. Ezzel szemben egy nem-determinisztikus algoritmus esetén a véletlen is szerephez jut, így előfordulhat, hogy többször lefuttatva az algoritmust ugyanarra a bemenetre, különböző eredményeket kapunk. A következőkben a két csoport előnyeit, hátrányait és általános megvalósítási elveit vizsgáljuk.

4.2.1. Determinisztikus eljárások

A determinisztikus algoritmusok legfőbb előnye éppen abból adódik, hogy determinisztikusak, így sok tekintetben megbízhatóbbak. Tehát biztosak lehetünk abban, hogy ha az algoritmus egyszer működött, akkor legközelebb is működni fog, ha pedig valamiért nem működik, akkor a hibát rendszerint generálni is lehet, így fel lehet deríteni a hiba helyét, és ki lehet javítani. A hatékonyságot is pontosabban meg lehet becsülni, és általában kisebb a szórás az eredményekben. További előny, hogy az algoritmus leírása magáért beszél: a működés ebből közvetlenül látszik. Mindebből világos, hogy ha van egy adott problémára egy jó megoldásunk, akkor azt – ha lehet – célszerű determinisztikusan megvalósítani. Gond rendszerint akkor van, ha nincs ilyen algoritmus, vagy egyes részletek még nem világosak, és további kísérletezésre van szükség. Ilyen esetekben általában nem-determinisztikus algoritmusokkal lehet előbbre jutni.

4.2.2. Nem-determinisztikus eljárások

A nem-determinisztikus algoritmusok fő előnye, hogy általában nem igénylik a probléma megoldási menetének pontos ismeretét, mégis nehéz problémákat hatékonyan meg tudnak oldani. További előny, hogy implementálásuk általában viszonylag egyszerű, és lényegében ugyanaz az eljárás számos különböző problémát megold. Emellett könnyen lehet velük kísérletezni, mivel új heurisztikák kipróbálásához nem kell az egész algoritmust átírni, csupán egyes paramétereket átállítani, más célfüggvényre optimalizálni stb.

A legegyszerűbb, széles körben alkalmazható nem-determinisztikus algoritmus a hegymászás (*hill climbing*). Ennek lényege a következő: kiválasztunk a keresési térben véletlenszerűen egy kezdőpontot. Minden lépésben kiszemeljük véletlenszerűen a keresési tér egy közeli

pontját, és megnézzük, hogy abban az állapotban a maximalizálandó célfüggvény értéke nagyobb-e, mint az aktuális pontban. Ha nagyobb, áttérünk az új pontra, különben maradunk. Ezt folytatjuk, amíg már nem tudunk tovább javítani.

Ez az eljárás egyrészt nagyon jó, mivel egyszerű, és garantáltan állandóan jobb és jobb eredményt ad. Viszont komoly hibája, hogy ha eljutott a célfüggvénynek egy lokális maximumába, akkor azt nem tudja elhagyni. Ennek kiküszöbölésére számos megoldás született, pl. szimulált lehűtés (*simulated annealing*) [Kirkpatrick], küszöb-elfogadás (*threshold accepting*), vagy a következő fejezetben ismertetésre kerülő genetikus algoritmusok.

A genetikus algoritmusok nagy előnye, hogy, bár nem determinisztikusak, de bizonyos feltételek mellett bizonyítható az alábbi konvergencia tétel:

Ha a lépésszám tart a végtelenhez, akkor a globális optimum megtalálásának valószínűsége tart 1-hez. [Kinnebrock1994]

5. Genetikus algoritmus

A genetikus algoritmusoknak is igen nagy irodalmuk van [Kinnebrock1994], [Davis1991], így ismét csak egy rövid áttekintést adunk, majd rátérünk annak ismertetésére, hogy a fentiekben leírt ütemezési probléma megoldásában hogyan lehet genetikus algoritmust alkalmazni. Arra is részletesen kitérünk, hogy az általunk készített genetikus algoritmus megvalósítása során milyen problémák merültek fel, illetve milyen egyedi megoldási módszereket alkalmaztunk.

5.1. Genetikus algoritmusok általában

5.1.1. Motiváció

A genetikus algoritmusok tulajdonképpen a 4.2.2. fejezetben vázolt hegymászás általánosításának tekinthetők. Tegyük fel ugyanis, hogy a keresési tér egy *n*-dimenziós euklideszi tér, vagyis a keresési tér minden pontja egy n hosszú vektorral írható le. Tegyük fel még azt is, hogy a hegymászás minden lépésében csak egyetlen koordináta módosítását engedjük meg. Ezen kívül a hegymászás alapját képező mozgásra úgy is gondolhatunk, hogy minden lépésben létrejön egy új "egyed", amelyik csupán egyetlen koordinátában tér el az előzőtől, és a két egyed közül az marad fenn, amelyik az adott célfüggvény szempontjából kedvezőbb.

Ha így gondolunk a hegymászásra, akkor látszik, hogy az némi hasonlóságot mutat az evolúció mozgatórugóját jelentő mutációval és szelekcióval: az új egyed az előzőből mutációval jön létre, majd a szelekció révén csak az életképesebb marad fenn. Így az egyre életképesebb egyedek révén tudjuk maximalizálni célfüggvényünket. Ha ezt a hasonlatot kiterjesztjük, eljuthatunk a genetikus algoritmusokhoz. A kiterjesztés lényege, hogy egyszerre nem egy vagy két egyedet használunk, hanem egy egész populációt. Minden egyedet egy vektor reprezentál, aminek elemei az adott egyed génjei, az egész vektor pedig a kromoszóma. Itt említjük meg, hogy a genetikus algoritmusok alkalmazhatóságának egyik kritikus feltétele, hogy lehessen találni egy ilyen "értelmes" vektor-reprezentációt, ahol tehát az egyes gének ténylegesen információt hordoznak.

A célfüggvényt, amire optimalizálunk, genetikus algoritmusok esetén általában fitness-függvénynek nevezik, mivel ez adja meg, hogy egy adott egyed mennyire alkalmas a túlélésre, vagyis mennyire fitt. Ez egy $F: M \rightarrow R$ függvény, ahol M a keresési teret jelöli.

Vagyis F az egyes kromoszómákhoz rendel valós számokat, és minél magasabb értéket rendel egy kromoszómához, az annál fittebb. Ez tulajdonképpen egy erős egyszerűsítés az evolúció tényleges menetéhez képest, hiszen valójában egy egyed génállománya (genotípusa) nem közvetlenül határozza meg az egyed túlélési esélyeit, hanem csak a tényleges tulajdonságain (fenotípusán) keresztül. A genotípus csak valószínűsíti, hogy milyen lesz a fenotípus. Modellünkben a fenotípus egyáltalán nem jelenik meg. Ez tulajdonképpen szerencsés, hiszen az evolúció egy elég lassú eljárás, így annak érdekében, hogy a gyakorlatban használni tudjuk, minden gyorsítási lehetőséggel élnünk kell. Az algoritmus szempontjából a fenotípus csak egy fölösleges indirekció lenne.

Abból adódóan, hogy egyszerre számos egyeddel dolgozunk, lehetőség van a mutáció és a szelekció mellett egy új ún. genetikus operáció, a rekombináció bevezetésére. Tulajdonképpen a rekombináció köti össze az egyedeket, ettől lesz a genetikus algoritmus több, mint sok független hegymászó. A rekombináció során két egyed tulajdonságainak (génjeinek) valamilyen összevegyítéséből állítunk elő újabb egyedeket. Szerencsés esetben a rekombináció során az eredeti egyedek előnyös tulajdonságai ötvöződnek az új egyedekben. Persze az is előfordulhat, hogy rosszabb tulajdonságú egyedek jönnek létre; ezek eliminálása a szelekció feladata.

5.1.2. Az algoritmus leírása

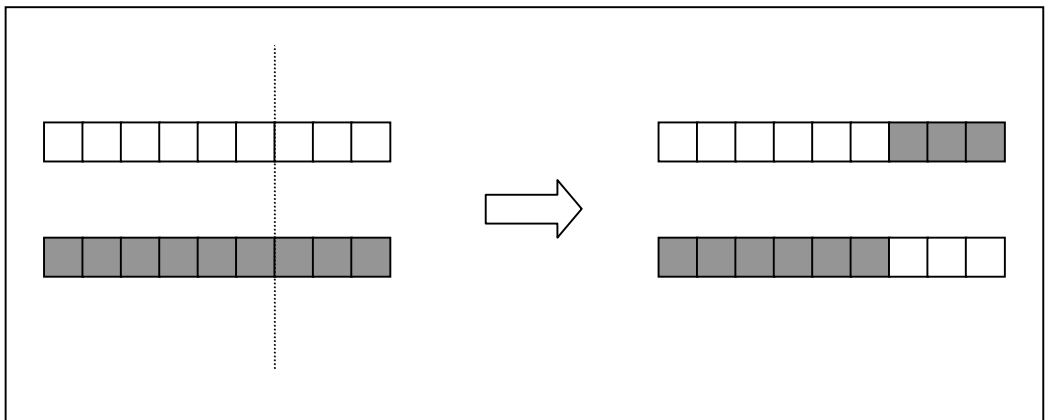
Ha a fentieket konkrétan meg akarjuk valósítani, a következőképp járhatunk el. Először is valamilyen módon létrehozunk egy kezdeti populációt. Ezután az eljárás minden lépésében az előző populációból a genetikus operációk révén előállítunk egy újat. Ennek során tehát egyszerre két populáció van jelen. Célszerűen az új populációnak egy megadott hányadát feltöltjük rekombináció segítségével (ezt a hányadot nevezzük rekombinációs aránynak), majd a maradékot szelekcióval. Végül az új populáció egyedeinek bizonyos hányadán mutációt hajtunk végre (mutációs arány). A sorrend azért fontos, mert így a rekombinációval és mutációval létrejött egyedek, még ha nem is túl életképesek, legalább egy lépés erejéig bekerülhetnek a populációba, így lehetőségük van a bennük rejlő esetleges pozitív tulajdonságok továbböröklésére.

Természetesen a fenti három genetikus operáción kívül még továbbiakat is lehet definiálni. Mi azonban megmaradtunk e háromnál, mivel úgy találtuk, hogy már ezek is elegendő rugalmasságot és sokszínűséget jelentenek a populáció építése során. Figyelembe kell venni azt is, hogy minden új operáció egy-két új paraméter felvételét is jelenti, és ezek nagyon

megnehezítik a genetikus algoritmus kiértékelését. Viszont végeztünk kísérleteket arra vonatkozóan, hogy mi történik, ha a három genetikus operáció közül egyet kihagyunk. Az eredmények alapján úgy tűnik, hogy csak a három operáció együttese alkalmas hatékony optimalizálásra. A rendszer teljesítménye nagyon érzékeny arra is, hogy a három operáció milyen arányban jut szerephez.

A mutáció azért fontos, mivel a sokféleséget garantálja, és így lehetővé teszi, hogy a lokális optimumokat elhagyjuk. Megvalósítása tipikusan úgy történik, hogy egy véletlenszerűen kiválasztott egyednek egy véletlenszerűen kiválasztott génjét véletlenszerűen módosítjuk. Ezen belül persze különböző variációkat lehet elképzelni aszerint, hogy egyszerre csak egy gént módosítunk vagy esetleg többet is, hogy csak kis módosítást engedélyezünk vagy nagyot is stb.

A rekombináció jelentősége abban áll, hogy a különböző egyedek jó tulajdonságait vegyíti, így ha egy egyedben létrejött egy kedvező génsorozat, akkor az el tud terjedni az egész populációban. A megvalósítása lényegében abból áll, hogy két egyed valamilyen keresztezésével egy vagy több (általában két) új egyedet hozunk létre. A keresztezés történhet a génsorozatok összefésülésével, átdarabolásával, átlagolásával stb. Egy gyakori módszer látható a 3. ábrán. A rekombináció során – szemben a mutációval – nem teljesen véletlenszerűen választott egyedekből érdemes kiindulni, hanem célszerű a jobb fitness-értékű egyedeket előnyben részesíteni. Ezekből ugyanis nagyobb valószínűséggel lehet jó tulajdonságokat továbbörökíteni.



3. ábra: a rekombináció megvalósítása.

A szelekció sem nélkülözhető; feladata a jó tulajdonságokkal rendelkező egyedek megtartása és a rosszak eliminálása. Megvalósítása egyszerűen annyi, hogy a legjobb egyedeket

átemeljük az új populációba. Ezt is variálhatjuk úgy, hogy nem feltétlenül a legjobbakat emeljük át, de ezeket nagyobb valószínűséggel.

Ha az egyes műveletekhez az egyedeket a fitness-től függő valószínűséggel szeretnénk kiválasztani, ennek megvalósítására is többféle megoldás kínálkozik. Egyik lehetőség a roulette-módszer, melynek során a kiválasztás valószínűsége arányos a fitness-szel. Ennek konkrét implementációjáról írunk majd a következő fejezetben. Egy másik, egyszerűbb módszer az *(N,k)*-módszer: ennek során az *N* egyedből álló populáció legjobb *k* egyede közül választunk egyenlő valószínűséggel.

5.2. Alkalmazás az ütemezés problémájára

Az alábbiakban leírjuk, hogy a fenti, általános sémát hogyan tudtuk alkalmazni az ütemezés problémájára. Ehhez meg kell adnunk, mik az egyedek, mi a populáció, mik a genetikus operációk és mi a célfüggvény.

5.2.1. Egyed

Az ütemezés problémája tulajdonképpen nagyon szerencsés a genetikus algoritmusok alkalmazhatósága szempontjából. Mint arról már korábban szó volt, egy problémát akkor lehet nagy valószínűséggel jól kezelni genetikus algoritmus segítségével, ha a feladat lehetséges megoldásait jól lehet vektorokkal ábrázolni oly módon, hogy az egyes komponensek önálló, értelmes jelentéssel bírjanak. Ez a feltétele ugyanis annak, hogy a mutáció és főleg a rekombináció tényleg a megoldás tulajdonságaira hassanak.

Az ütemezés esetén szerencsére szinte kínálja magát a vektoros reprezentáció: a gének az egyes csúcsok indítási idejének felelnek meg. Nem mindegy a sorrend sem: a rekombináció hatékony működése szempontjából kívánatos, hogy a szomszédos gének tényleg összefüggő tulajdonságoknak feleljenek meg. Feltételezhetjük azonban, hogy az az ember, aki az elemi műveleti gráfot megadta, ezt nyilván valamilyen logikus sorrendben tette (például fölülről lefelé, azon belül pedig balról jobbra), így a csúcsok már a bemenetben a genetikus ábrázolásra alkalmas sorrendben vannak.

5.2.2. Populáció

Amilyen szerencsénk volt az egyedek ábrázolásával, olyan problémás a populáció megválasztása. Alapvetően arra a kérdésre kell ugyanis választ adnunk, hogy a populációban csak érvényes, vagy tetszőleges ütemezéseket engedjünk-e meg.

Mivel minket műszaki szempontból tulajdonképpen csak az érvényes ütemezések érdekelnek, így első közelítésben mindenképp azt kell válaszolnunk erre a kérdésre, hogy a populáció álljon kizárólag érvényes ütemezésekből. Azonban van két probléma ezzel a hozzáállással. Először is, könnyen lehet, hogy ezzel sokat veszítünk a hatékonyságból. Lehet ugyanis, hogy egy érvényes állapotból egy nem érvényes állapoton keresztül egy sokkal jobb érvényes állapotba tudunk eljutni, de csak érvényes állapotokon keresztül ugyanide nem, vagy csak nagyon hosszú, körülményes úton érhetünk el. Egy ilyen esetben lemondanánk a jó megoldásról, ha megszabnánk, hogy csak érvényes egyedek szerepelhetnek a populációban.

A másik szempont az, hogy nehéz garantálni, hogy érvényes egyedekből ne keletkezzenek a genetikus operációk során érvénytelen egyedek. Gond van egyrészt a rekombinációval, másrészt pedig a mutációval. A rekombináción lehet segíteni, ha nem a 3. ábrán látható módon definiáljuk, hanem átlagolással. Sikerült ugyanis bebizonyítanunk, hogy ilyenkor érvényes egyedekből érvényes egyed jön létre, sőt, ez nem csak átlagolásra, de tetszőleges arányú súlyozott középre igaz. Ennek bizonyítása megtalálható a Függelékben. Azonban sajnos e módszer alkalmazása erősen korlátozná a sokszínűséget. A mutáció esetében még rosszabb a helyzet: itt tulajdonképpen csak úgy tudnánk garantálni a létrejövő egyed érvényességét, ha az esetleges érvénytelen egyedektől azonnal megszabadulnánk. Ez azonban ellentmond annak az elvnek, hogy minden létrejövő új egyednek meg kell adni a lehetőséget, hogy legalább egy populációban szerepeljen, és továbbörökítse a tulajdonságait.

Ezen okokból kifolyólag tehát úgy döntöttünk, hogy nem kötjük ki, hogy a populációban csak érvényes egyedek lehetnek, hanem tetszőleges ütemezések. De persze ez az ütemező magánügye; ettől még valahogy garantálnunk kell, hogy végül érvényes ütemezést szolgáltatunk. E célból két dolgot teszünk: egyrészt gondoskodunk arról, hogy a kezdeti populációban legyenek érvényes egyedek is, másrészt pedig olyan célfüggvényt használunk, ami garantálja, hogy az érvényes egyedek ne vesszenek ki a populációból, sőt, lehetőleg újak keletkezzenek.

5.2.3. Genetikus operációk

Mutációt, rekombinációt és szelekciót használunk. A mutációt az új populáción végezzük, ehhez minden egyedet egyforma valószínűséggel választunk ki. A rekombinációhoz a 3. ábra módszerét használjuk, két régi egyedből két új egyedet állítunk elő. A rekombinálandó egyedek kiválasztásához roulette-módszert használunk. A szelekció során a régi populáció legjobb egyedeivel töltjük fel az új populáció még fel nem töltött helyeit. E célból először a QSort gyorsrendező algoritmussal fitness szerint csökkenő sorba rendezzük az egyedeket, és ezután az elöl lévő egyedeket rakjuk be az új populációba.

5.2.4. Célfüggvény

A célfüggvény két komponensből tevődik össze. Az egyik komponens az, amire tulajdonképpen optimalizálni akarunk: a kompatíbilis párok száma. A másik komponens abból adódik, hogy érvénytelen egyedeket is megengedünk, ám ezeket éppen a célfüggvény segítségével akarjuk büntetni, hiszen az a cél, hogy ilyen egyedek csak akkor legyenek, ha a másik komponens szempontjából valamilyen nagyon jó tulajdonságot hordoznak. Ilyenkor is arra kell ösztönözni ezeket az egyedeket, hogy egyre kevésbé legyenek érvénytelenek. Ennek megfelelően a célfüggvényben az szerepel, hogy az adott egyed hány ponton sérti meg az axiómákat. Nevezzük ezt a hibák számának. Ekkor tehát a célfüggvény a kompatíbilis párok számából valamint a hibák számából tevődik össze, előbbiben monoton nő, utóbbiban monoton csökken.

5.2.5. Kimenet

Az optimalizálás szempontjából célszerű, ha minél nagyobb populációval dolgozunk, ám végül mégis egyetlen megoldást kell adnunk. Azonban nem szeretnénk elveszíteni azt az előnyt, hogy az utolsó populációban esetleg több tucat jó és érvényes ütemezés van. Ezért az utolsó populációból kiválasztjuk az érvényes megoldások közül a legjobbakat, és mindre lefuttatjuk az allokációt. Megnézzük, melyik milyen költségű megoldást ad, és ennek alapján választjuk ki a legjobbat.

5.3. Implementációs részletek

Ebben a pontban még mélyebbre hatolunk genetikus megvalósításunk ismertetésében, és néhány érdekes implementációs részletet mutatunk be.

5.3.1. Kezdeti populáció

Annak érdekében, hogy biztosak lehessünk benne: végül érvényes ütemezést adunk, nem elég egy olyan célfüggvény, mely az alacsonyabb hibaszámú egyedeket előnyben részesíti, hanem a kezdeti populációban is el kell helyezni érvényes ütemezéseket. Esetleg érdemes többet is elhelyezni, mert így hamarabb túlsúlyba kerülhetnek az érvényes ütemezések, és ezáltal nem fenyeget az a veszély, hogy a számítási kapacitás magas hibaszámú egyedek kezelésére megy el. Kérdés viszont, hogy hogyan tudunk egyszerűen gyártani érvényes ütemezéseket? Két érvényes ütemezést szerencsére azonnal ismerünk: ez az ASAP és az ALAP. Ezekből pedig a már említett súlyozott közép módszerével tudunk új érvényes ütemezésekhez jutni.

Ez úgy van megvalósítva a programunkban, hogy az .ini file-ban meg lehet adni: a kezdeti populáció hányadrészét kell feltölteni érvényes egyedekkel. Tegyük fel, hogy Z darab érvényes egyedet kell létrehozni. Ekkor az i. egyed ($i=0...Z-1$) így néz ki: $ASAP+(ALAP-ASAP)*i/(Z-1)$.

Persze ezzel a módszerrel általában nem tudunk Z darab különböző ütemezést előállítani. Előny azonban, hogy egyrészt az egyedek létrehozása nagyon egyszerű, másrészt pedig változatosak lesznek abban az értelemben, hogy a két véglet (ASAP és ALAP) között egyenletesen vesznek fel különböző lehetőségeket. Így remélhető, hogy a későbbiekben a rekombinációk és mutációk révén ezekből igen különböző, de érvényes egyedek fognak létrejönni. A módszer helyességének bizonyítása megtalálható a Függelékben.

5.3.2. Tömbök

A program hatékonysági okokból C nyelven íródott, nem objektum-orientált, így az egyedek és populációk nem objektumokként, hanem tömbökként jelennek meg. Minden egyes egyedet egy tömb, az indítási idők tömbje reprezentál, a populáció pedig egyedek tömbje. Két ilyen nagy tömb van, és annak érdekében, hogy ne kelljen minden lépés végén az új populációt átmásolni a régi populációnak megfelelő tömbbe, felváltva egyszer az egyik tömb tartalmazza a régi populációt és a másik az újat, aztán fordítva.

Sajnos az egyes egyedekhez tartozó foglaltsági időknek, a kompatíbilis párok számának és a hibák számának az állandó kalkulálása rengeteg időt vesz el, így igyekeztünk ezt minden lehetséges módon csökkenteni. Ennek megfelelően e számadatokat is külön tömbökben tároljuk, és csak akkor számoljuk újra, ha muszáj. Például, ha egy egyed szelekció révén

változatlanul átkerül az új populációba, ezeket a számokat nem kell újra előállítani, hanem vele együtt másoljuk.

5.3.3. Roulette-módszer megvalósítása

A roulette-módszer során az egyes egyedek fitness-értékükkel arányos valószínűséggel kerülnek kiválasztásra. Ennek megvalósítása a következő. Tegyük fel, hogy a fitness mindig pozitív, valamint hogy az egyedek valamilyen módon sorba vannak rakva, és *0*-tól *n-1*-ig vannak számozva. Legyen S_i az *i*-nél kisebb indexű egyedek fitness-ének az összege ($i=0...n$). Ekkor persze $S_0=0$, S_n az összes fitness-értékek összege, továbbá *S* monoton nő. Válasszunk egy tetszőleges $0<m<S_n$ számot, és keressük meg, hogy melyik $[S_i, S_{i+1}]$ intervallumba esik. A választott egyed ezek után az *i*.

Látható, hogy mivel az $[S_i, S_{i+1}]$ intervallumok hossza éppen a megfelelő egyed fitness-ével egyezik meg, így az egyedek kiválasztási valószínűsége valóban a fitness-szel arányos. A módszer neve onnan ered, hogy ha ezen intervallumokat egy roulette-tányéron képzeljük el, akkor a véletlenszerű kezdősebességgel elindított roulette-golyó is minden cikkelyen annak méretével arányos valószínűséggel áll meg.

Végezetül annyit, hogy a módszer utolsó lépése, a megfelelő intervallum megkeresése jelentősen gyorsítható a nyilvánvaló lineáris kereséshez képest. Ugyanis, mivel az S_i értékek monoton növő sorozatot alkotnak, így alkalmazható a bináris keresés, melynek lépésszáma csupán *log n*.

5.3.4. Célfüggvény hangolása

A célfüggvény megválasztása kulcsfontosságú mind a hatékonyság, mind az eredményesség szempontjából. Mint arról már korábban szó esett, célfüggvényünk két komponensből, a kompatíbilis párok valamint a hibák számából áll. Számos különböző ötlet kipróbálása után végül az alábbi két célfüggvény vált be:

F1=max_hiba_szám-hiba_szám+kompatíbilis_szám/max_kompatíbilis_szám

*F2=C1-(L/R)*max_kompatíbilis_szám*hiba_szám/(1+C2*hiba_szám)+kompatíbilis_szám*

Mindkét esetben, ha a hibák száma egy előre megadott *max_hiba_szám* fölött van, a célfüggvény értékét nullára állítjuk. Egyébként mindkét esetben a célfüggvény értéke pozitív. (Ez kell is a roulette-módszer működésének helyességéhez.)

Az első függvény kitalálásakor az a kép vezetett minket, hogy az egyedeknek egy “lépcsőt” kell biztosítanunk, amin fel tudnak menni a minél jobb tulajdonságok felé. Ezen belül vannak nagy és kis lépcsők: a hibák számának minden egyes csökkenése egy nagy lépcsőt jelent, hiszen a célfüggvényt 1-gyel növeli. A kompatíbilis párok számának a növelése egy kisebb lépcsőt, mivel ez csak *1/max_kompatíbilis_szám* mértékű növekedést okoz. Ezáltal a hibák számának 1-gyel való csökkentése többet ér, mint a kompatíbilis párok számának akármilyen mértékű növelése. Ez biztosítja, hogy a kezdetben meglévő érvényes egyedeket semmiképp se veszítsük el, valamint további érvényes egyedek létrejöttét is ösztönzi.

A másik megoldás kevésbé szigorú. Itt nem állítjuk kategorikusan, hogy a hibák számának csökkentése többet ér, mint a kompatíbilis párok számának akármilyen mértékű növelése. Megengedjük a hibák számának növekedését is, de csak akkor, ha cserébe a kompatíbilis párok száma kellő mértékben növekszik. Kérdés, hogy mit jelent ez a “kellő mértékben”. Úgy találtuk, hogy ennek függenie kell az *R*, pontosabban az *R/L* hányados értékétől. Ha ugyanis az *R* *L*-hez képest kicsi, akkor viszonylag nehezebb a kompatíbilis párok számát növelni, így vigyázni kell, nehogy ez a hibaszám rovására menjen. Ezért ilyenkor a hibaszám növelését csak a kompatíbilis párok számának jelentős növelése mellett tudjuk elfogadni. Ha viszont *R* nem sokkal kisebb *L*-nél, akkor ez nem olyan szigorú. Emellett azt is figyelembe szeretnénk venni, hogy az, hogy egy hibát hány kompatíbilis párért cserébe engedünk meg, függjön attól is, hogy éppen hány hiba van. Tehát pl. egy egyed 0 hibásról 1 hibásra változásáért cserébe jóval több kompatíbilis párt követelünk meg, mint mondjuk 8-ról 9-re.

5.3.5. Paraméterek kezelése

Mint az már az eddigiekből is látható, programunknak igen sok paramétere van. Ez egyrészt előny, hiszen a működést rugalmasan lehet szabályozni, másrészt hátrány, mivel megnehezíti a program kiértékelését. Erről a 7. fejezetben részletesen fogunk írni.

A fontosabb paraméterek a következők: populáció mérete, iterációk száma, rekombinációs arány, mutációs arány, kezdeti populációban az érvényes egyedek aránya. Ezen kívül a feladatból adódó paraméter az input file neve valamint az *R* és *L*. Számos további paramétert is fel lehetne venni, pl. hogy a rekombinációban az egyedek elvágása milyen határok között engedélyezett, vagy hogy a mutációban maximum milyen mértékű módosítást lehet véghezvinni stb. De mint látni fogjuk, ezek nélkül is elég bonyolult a kiértékelés.

E paraméterekre a később ismertetendő tesztelési eredmények alapján bevezetünk alapértelmezett értékeket, amik az esetek túlnyomó részében jónak bizonyultak. De ezeket felül lehet definiálni akár az .ini file-ból, akár parancssori argumentumokkal.

6. CCLS

6.1. Az algoritmus alapelve

A CCLS (Compatibility Controlled List Scheduling) a korábban már vázolt listás ütemezés egyik változata. A listás ütemezések alapgondolata az, hogy valamilyen korábbi mérnöki tapasztalatból származó heurisztika alapján sorrendbe állítjuk a csúcsokat, és ebben a sorrendben egyszer végighaladva, minden csúcsot az adott pillanatban legoptimálisabb pozícióba rögzítünk. Ennek a módszernek nagy előnye az egyszerűsége és gyorsasága, fő hátránya azonban, hogy a keresési tér nagyon kis hányadát járja csak be.

Az általunk készített algoritmus egy kompromisszumos megoldással próbálja kiküszöbölni a hátrányokat az előnyök megtartása mellett. Ahelyett, hogy minden csúcsot egyesével sorra vennénk, és ennek az egy csúcsnak a szempontjából optimális helyre rögzítenénk, a csúcsokból csoportokat alakítunk ki, és ezeken a csoportokon végighaladva az egész csoport legkedvezőbb rögzítését választjuk ki. Ezzel lényegesen több lehetőséget veszünk figyelembe, optimálisabb megoldást kapunk, mégis az egyszeri optimalizálásnak köszönhetően nem vesztünk sokat a sebességből.

Természetesen az algoritmus hatékonysága és sebessége nagyban függ a csoport méretének (*grp*) megválasztásától. Ha *grp=1*, akkor szélső esetként visszakapjuk az eredeti listás ütemezést, ha *grp* megegyezik a pontok számával, akkor a másik szélső esetet, az összes lehetőség bejárását kapjuk. (Erre persze nagyobb bemenetek esetén nincs mód véges idő alatt.) Látható, hogy *grp* megválasztásával finoman tudjuk hangolni a hatékonyság / futási idő arányt.

A leglényegesebb kérdés az, hogy egy konkrét pillanatban hogyan döntjük el, hogy az adott csoport tagjait hova időzítsük. Ehhez be kell vezetni egy mérőszámot, amely az optimalitást méri. Az algoritmus ezen változatában az optimalitást az azonos típusú kompatíbilis csúcspárok számának maximalitása jelenti – erre utal az algoritmus neve is.

Ahhoz, hogy egy adott rögzítési lépésben meg tudjuk állapítani a kompatíbilis csúcsokpárok számát, minden csúcsnak lerögzített állapotban kell lennie. Ehhez az algoritmus legelején egy kezdeti rögzítést kell választani. A jelen implementációban ez minden csúcs ALAP értéke, amiről garantáltan tudjuk, hogy egy érvényes ütemezés, és innen mozgatjuk a csúcsokat folyamatosan az ASAP felé. Egy adott lépésben az éppen aktuális csoport minden lehetséges

elhelyezkedését megvizsgáljuk, és ebből a legkedvezőbbet választjuk ki. Ezután ennek a csoportnak az elemeit a későbbi lépésekben már nem módosítjuk. Az algoritmus akkor terminál, ha végigmentünk az összes csúcson. Látható, hogy ez egy valamilyen értelemben vett lokális keresés, illetve lokális keresések kombinációja, és ennek megfelelően az algoritmus során szigorúan egyre jobb megoldásokat találunk. Ennek az az előnye, hogy az algoritmus futása bármikor megszakítható, ha nincs elég idő kivárni a végét, és ekkor is egy viszonylag jó megoldást kapunk.

6.2. CLP alkalmazásának célszerűsége

A legnagyobb nehézség az imént körvonalazott algoritmus megvalósításában az, hogy folyamatosan figyelnünk kell, hogy egy adott rögzítés hogyan módosítja a többi csúcs mobilitási tartományát. Az elemi műveleti gráf meghatároz a csúcsok között bizonyos precedenciákat, melyeket folyamatosan be kell tartani. Elképzelhető, hogy egy csúcs elmozdításával megsértünk egy ilyen feltételt, és ezért kénytelenek vagyunk valamelyik szomszédját is elmozdítani, ami újabb feltételek megsértését vonhatja maga után. Látható, hogy egy elmozdítás a kényszereken keresztül további mozgások egész láncolatát eredményezheti, mire a végén kiderül, hogy az eredeti mozdítás lehetséges vagy sem. Egy csúcs elmozdításával tehát számos másik csúcs mobilitási tartománya változhat meg, és az algoritmust végrehajtó programnak ezt folyamatosan figyelnie kell. Ezeknek a kényszereknek a kezelése hagyományos programozási nyelvekben, mint pl. a C, meglehetősen nehéz, ezért alkalmaztuk erre a problémára a logikai programozás eszköztárát, azon belül is a SICStus Prolog 3.8.4 CLP(FD) (Constraint Logic Programming Finite Domain) könyvtárát.

6.3. A SICStus CLP(FD) könyvtára

A SICStus CLP(FD) könyvtára véges tartományú egész változók kezelését támogatja. Minden változóra definiálnunk kell azt a halmazt, ahonnan a lehetséges értékeit felveheti, továbbá megadhatunk olyan kényszereket, kötöttségeket, amiknek az egyes változók között fenn kell állniuk. Ha egyszer ezeket a korlátokat (constraint) definiáltuk, akkor a továbbiakban nem kell ezek betartásával foglalkoznunk, ezt biztosítja a Prolog mechanizmusa.

Ezt úgy kell elképzelni, hogy van egy központi tár, ahol a SICStus tárolja minden egyes változó lehetséges értékeit, és amikor egy új korlátot definiálunk, akkor az hat a táron, azaz bizonyos értékeket kizár a lehetőségek közül. Ezt nevezik szűkítésnek. Komplikáltabb

összefüggések hatása nem tud ilyen közvetlenül megjelenni, esetleg nem tud azonnal szűkíteni a táron, ezért ezekből egy ún. démon lesz, amelyik folyamatosan figyeli az eseményeket, várva arra a pillanatra, amikor elég információ áll már rendelkezésre ahhoz, hogy kifejtse hatását. Nézzünk a fentiekre néhány példát.

```
domain([X,Y],2,5),X#=2*Y.
```

A fenti sorban az X és Y változók kezdeti tartományát a [2,5] zárt intervallumra állítottuk be, majd felvettük azt a korlátot, hogy X legyen Y kétszerese. Ekkor a korlát érvényre jut, leszűkíti a változók tartományát, és a SICStus válasza az alábbi:

```
X = 4,
Y = 2
```

Nézzünk egy példát arra is, amikor egy korlát nem tudja rögtön kifejteni a hatását, és démonként vár addig, amíg érvényre nem juthat.

```
domain([X,Y],2,5),X#\=Y.
```

Most csak annyit tudunk, hogy X és Y nem egyenlő. Jelen pillanatban ebből semmit nem lehet levonni, azaz a válasz:

```
X in 2..5,
Y in 2..5
```

Ha a program futása során valamikor pl. az X változóról további információk derülnek ki, akkor a fenti korlát is kifejti hatását; mondjuk ha megadjuk, hogy

```
X#=3
```

akkor a következtetőgép így reagál:

```
X = 3,
Y in{2}\/(4..5)
```

azaz az Y tartományából kizárja a 3 értéket megfelelően annak, hogy Y különbözik X-től. Ekkor az `X#\=Y` démon kifejtette hatását, és ezzel megszűnik, hiszen további szűkítésekre nincs lehetősége.

A CLP(FD) működésének egyik legfontosabb jellemzője, hogy az egyes változók tartománya folyamatosan csak szűkülhet, azaz nincsen mód arra, hogy a kezdeti tartományon kívüli vagy már egyszer kizárt értékek bekerüljenek a tárba. Ha mégis egy olyan korlátot szeretnénk felvenni, ami inkonzisztensé tenné a tárat, azaz nincsen olyan változó kombináció, amely minden korlátnak eleget tesz, akkor a program meghiúsul, jelezvén, hogy nincs megoldása a feladatnak. Nézzünk erre is egy példát:

```
domain([X,Y],2,5),X#=3*Y.
```

A válasz:

```
no
```

Jól definiált ütemezési feladatoknál nem fordulhat elő, hogy a programunk nem oldja meg a feladatot és végül meghiúsul, hiszen az ALAP ütemezés egy érvényes, minden kényszert kielégítő megoldás, és ebből minden lépésben érvényes ütemezéseken keresztül jutunk el a végső kimenethez. (De ha a feladat ellentmondást tartalmaz, például az elemi műveleti gráfban kör van, akkor azt a Prolog meghiúsulással jelzi.)

Ha felvettük az összes korlátot a változóinkra, akkor utasíthatjuk a SICStust, hogy adja meg a változók összes lehetséges értékeit, azaz sorolja fel a megoldásokat. Ez technikailag úgy történik, hogy a gép sorra veszi az összes változót minden lehetséges módon behelyettesíti az értékeiket, és megpróbál szűkíteni a táron. Amikor az utolsó változó lerögzítésekor is sikerül a szűkítés, akkor azt egy megoldásnak könyveli el. Erre szolgál a labeling (címkézés) parancs.

```
domain([X,Y],2,5),X#=Y+2, labeling([],[X,Y]).
X = 4,
Y = 2
```

és

```
X = 5,
Y = 3
```

Egy másik fontos beépített segítség a `minimize` ill. a `maximize` eljárás, melynek segítségével egy adott változó értékét minimalizálni, ill. maximalizálni tudjuk.

Korántsem törekedtünk a CLP(FD) könyvtárról átfogó ismereteket közölni (bővebben [Szeredi1998/2000]), csak egy kis betekintést adtunk annak érdekében, hogy világos legyen, hogy miért pont ezt a nyelvet választottuk. A következő fejezetben pontosan megfogalmazzuk, hogy hogyan használtuk fel céljainkra a SICStus korlátkezelési mechanizmusát.

6.4. CLP(FD) alkalmazása az ütemezésben

Az ütemezés feladata az egyes csúcsok indítási idejének meghatározása. Ezért döntöttünk úgy, hogy minden csúcshoz hozzárendelünk egy változót, aminek a jelentése az, hogy az adott

csúcsot mikor indítjuk el. Kezdetben annyit tudunk, hogy minden csúcs ASAP és ALAP között indulhat, azaz a változóink kezdeti tartományát ezekkel az értékekkel inicializáljuk.

Fel kell vennünk ezek után azt a feltételt, hogy az elemi műveleti gráfban éllel összekötött csúcsok egymás után következő műveleteknek felelnek meg. Ha v_i csúcsból mutat él v_j csúcsba, és a nekik megfelelő változók rendre V_i, V_j , továbbá v_i csúcs futási ideje d_i, akkor azt, hogy v_i-nek be kell fejeződnie v_j elkezdése előtt, az alábbi korláttal fogalmazhatjuk meg:

$$V_i + d_i \leq V_j$$

Ezzel látszólag a feladatot definiáltuk is, csupán azt kell még valahogy a Prolog tudomására hozni, hogy maximalizálni szeretnénk a kompatíbilis csúcspárok számát. Ehhez be kell vezetnünk egy változót, aminek az értéke minden pillanatban a kompatíbilis csúcspárok számával egyenlő. Azonban ennek kiszámítása viszonylag körülményes feladat, hiszen a 3.3. fejezetben közölt egyenlettel tudjuk leírni két csúcs kompatibilitását, és ezeket kell összegeznünk minden azonos típusú csúcspárra. Tovább bonyolítja a dolgot, hogy a képletben szerepel az egyes csúcsok foglaltsági ideje, ami szintén az adott elrendezéstől függő változó. A foglaltsági idő kiszámítása önmagában is meglehetősen bonyolult feladat, mert nem elég egyszer kiszámolni, hanem egy olyan korlátrendszert kell felvenni, aminek hatására a foglaltsági idő minden elrendezésben a korlátok hatására “magától” beállítódik.

Az alábbi kódrészlet azt mutatja, hogy hogyan határozzuk meg a V1, V2 változókkal jellemzett csúcsok kompatibilitását. Q1, Q2 a megfelelő foglaltsági időket jelöli, valamint a B Boole-változó aszerint vesz fel 0 ill. 1 értéket, hogy fennáll-e a kompatibilitás vagy sem. (Egy adott korlát fennállásának bool-változóba való leképezését *reifikációnak* nevezzük.) A kompatíbilis csúcspárok számát ezen B változók minden azonos típusú csúcspárra történő összegzésével kapjuk.

```
B#<=>(V1#<V2 #/\ (V1-V2+Q1)#=<0 #/\ (((V1-V2-Q2)/R+1#=(V1-V2+Q1)/R
      #/\ (V1-V2-Q2) mod R#=0) #\/ ((V1-V2-Q2)/R#=(V1-V2+Q1)/R)))
      #\/(V2#<V1  #/\ (V2-V1+Q2)#=<0 #/\  (((V2-V1-Q1)/R+1#=(V2-V1+Q2)/R
      #/\ (V2-V1-Q1) mod R#=0) #\/ ((V2-V1-Q1)/R#=(V2-V1+Q2)/R)))
```

Látható tehát, hogy nagyon sok korlát egymásra hatásából alakul ki végül az adott ütemezésre vonatkozó kompatíbilis csúcspárok száma.

A korlátok felvétele után az ismertetett keresési módszerrel maximalizáljuk a célváltozót.

6.5. Implementációs részletek

Ebben a fejezetben betekintést adunk néhány implementációs részletbe, mely vagy technikai nehézsége vagy éppen egyszerűsége miatt érdemel külön figyelmet.

6.5.1. DCG nyelvtan alkalmazása

Bár elméleti szempontból nem túl jelentős, mégis a program írásának egyik fontos része a bemenő és kimenő formátum kezelése. A bemenő file formátumát a 3.4. fejezetben megadtuk. Egy összetettebb struktúrájú bemenet esetén nehézséget jelent a bemenet szintaktikai helyességének ellenőrzése, ill. a bemenő formátumból az adatok kinyerése.

A SICStus DCG (Definite Clause Grammars) nyelvtanok használatával teszi kényelmessé a programozó számára ezt a feladatot. A deklaratív programozási elvnek megfelelően, a DCG szabályait betartva elég magas szinten leírnunk a bemenet formátumát, és a SICStus mintaillesztő mechanizmusa segítségével egyszerre történik a szintaktikai ellenőrzés és a beolvasás. Az alábbi (egyszerűsített) programrészlet jól illusztrálja a DCG nyelvtanok használatának egyszerűségét és erősségét. A bemenő file egy sorának leírására az alábbi magas szintű, jól olvasható kód szolgál:

```
line -->
     [Name],
     [Type],
     [Duration],
     [ASAP],
     [ALAP],
     nameList(Predecessors),
     nameList(Children).
```

Ez a kezelésmód feltételez egy előfeldolgozó részt, amely tokenekre bontja az inputot. A szintaktika ellenőrzése mellett az egyes változók is értéket kapnak, így például a Name változó az első token lesz.

6.5.2. Csúcsok sorrendje

Az optimalizálás során a változókból csoportokat alkotunk, és adott sorrendben végighaladva végérvényesen lerögzítjük őket. A végső időzítés minőségére nagy hatással lehet, hogy hogyan csoportosítjuk, és milyen sorrendben vesszük sorra a változókat.

Egy logikus gondolat lenne, ha azokat a csúcsokat optimalizálnánk egyszerre, amelyeknek nagy hatása van egymásra, és külön csoportba tennénk az egymástól távoli, független csúcsokat. Sajnos ennek eldöntése meglehetősen bonyolult feladat, mert a pipeline üzemmód következményeként az elemi műveleti gráfban egymástól távol eső csúcsok is lehetnek konkurensek, így közvetlenül hatnak egymásra.

A csúcsok sorrendjének meghatározására egy ettől eltérő, a mérnöki tapasztalaton alapuló heurisztikát használtunk, melynek lényege legtömörebben úgy fogalmazható, hogy "halogassuk a nagy döntéseket, ameddig csak lehet". Ez azt jelenti, hogy minden csúcshoz bevezetünk egy mérőszámot (λ), amely azt méri, hogy az adott csúcs lerögzítése mekkora szabadsági fok vesztéssel jár. λ szerint növekvő sorrendben rögzítjük le a csúcsokat, tehát a "legjelentéktelenebbtől" a "legjelentősebbekig" haladva. λ értéke két paramétertől függ, az adott csúcs mobilitásától és a művelet hosszától. Nyilvánvalóan egy nagy mobilitású csúcs lerögzítése nagy szabadsági fok vesztéssel jár; hasonlóan egy hosszú művelet várhatóan sok másikkal lesz konkurens, így elhelyezése egy nagy döntésnek számít. Ezért λ mindkét paraméterében monoton nő. Egy egyszerű, és általunk is használt ilyen függvény: $\lambda_i = m_i * d_i$, ahol m_i az i. csúcs mobilitása, d_i pedig a végrehajtási ideje.

6.5.3. Keresés megvalósítása Prolog-ban

Legnagyobb problémát a listás ütemezés elvét megvalósító keresés implementálása jelentette. A Prolog alapvető szemléletével ellentétes változók értékének módosítása. A Prolog számára egy változó kezdetben behelyettesítetlen, és bármi lehet még az értéke, vagy behelyettesített, és innentől nem változó többé. A CLP(FD) könyvtárban ez egy kicsit rugalmasabban jelenik meg, azaz a behelyettesítésnek nem kell azonnal megtörténnie, hanem egy hosszú folyamat lehet, amíg a kezdeti jelöltekből kiválasztjuk a "győztest", ami aztán a változó értéke lesz.

Sajnos itt sem oldható meg az, hogy egy teljesen behelyettesített változó új értéket kapjon, hiszen ez ellentmondana annak az alapelvnek, hogy csak folyton szűkíthetünk a táron, sohasem bővíthetünk. Pedig egy keresési algoritmus alapvető jellemzője, hogy megnézünk egy konkrét esetet, azaz mindent lerögzítünk egy adott értékre, majd kiértékelés után a változóknak új értéket adva áttérünk a következő esetre.

Az egyetlen mód, ahogy Prologban már megtörtént behelyettesítéseket semmissé tehetünk, az ún. visszalépés. Ez akkor következik be, ha valamilyen ellentmondás következtében meghiúsulás lép fel. Ekkor a Prolog visszalép a legutolsó választási pontig, és mindent

pontosan visszaállít arra az értékre, ami ennél a választási pontnál fennállt. A visszalépés és a választási pont fogalmának megvilágítására álljon itt egy kis példa:

```
member(ListaElem,[1,2,3]),ListaElem>2.
```

Amikor a Prolog elérkezik a member függvényhez, nem tudja, hogy a ListaElem változónak melyik értéket adja, mert eddig semmi sem szól egyik ellen sem. Ezért létrehoz egy választási pontot, és először ListaElem=1-et próbálja ki. A második utasításhoz érve meghiúsul, és visszatér az előző választási ponthoz, ahol ListaElem ismét behelyettesítetlenné válik, majd rögtön ListaElem=2-t választ. Hasonlóan jut el ahhoz, hogy ListaElem csak 3 lehet. Egy meghiúsulást szándékosan is előidézhetünk a `fail` paranccsal.

Ezt a visszalépéses mechanizmust trükkösen kihasználva tudtuk megoldani, hogy a keresés pontosan a mi algoritmusunk szerint működjön. A nehézség abban rejlik, hogy a visszalépés hatására minden visszaállítódik a legutolsó választási pont állapotára, és így "elfelejtjük", hogy eddig mi volt a legjobb talált megoldás. A trükk, amivel ezen segíthetünk az, hogy vissza nem fordítható eljárásokat használunk, ilyen pl. ún. *assert* függvény, amely képes dinamikus Prolog relációk létrehozására meg nem hiúsítható módon. (Ez lényegét tekintve hasonlít egy kiíráshoz, csak nem jelenik meg a képernyőn.) Ennek segítségével az előző lépésben talált eredményt "kiasszertáljuk", és így nem veszik el a visszalépés hatására.

7. Elemzés

Munkánk kezdetekor azt tűztük ki célul, hogy a PIPE-ban jelenleg működő force-directed ütemezőnél jobbat készítsünk. Mivel a PIPE jól szeparálható modulokból épül fel, melyek egymással csak szövegfájlokon keresztül kommunikálnak, így lehetőség van arra, hogy az általunk írt ütemezőt tegyük a PIPE-ba anélkül, hogy a többi részhez hozzá kellene nyúlni. Nincsen szinte semmilyen megkötés az ütemező algoritmus implementációjára nézve, csupán a be- és kimeneti fájlformátumot kell betartani. Ezért mindkét algoritmusunk alkalmas a korábbi ütemező helyettesítésére. A tesztelés feladata annak eldöntése, hogy az eddigi eredményeken miben (processzor felhasználás, sebesség) és milyen mértékben sikerült javítani.

7.1. A tesztelés nehézségei

7.1.1. Genetikus algoritmus

A genetikus algoritmus kezdeti paraméterekkel konfigurálható, melyek beállításai nagymértékben befolyásolják az algoritmus hatékonyságát és sebességét. A genetikus ütemező algoritmus az alábbi bemenő paramétereket kezeli:

- populáció mérete
- rekombinációs arány
- mutációs arány
- kezdeti populáció érvényes egyedeinek aránya
- lépésszám
- újraindítási idő
- lappangási idő (csak a genetikus algoritmus második verziójának van rá szüksége)

A PIPE-ba való beillesztés után csak az utóbbi két paraméter kézzel történő beállítására van lehetőség, mert ezek a PIPE bemenetének is részei. A többi paraméter esetében nincsen mód minden egyes futtatás során ezek átállítására, hiszen nem várhatjuk el a program felhasználójától, hogy pontosan ismerje a genetikus ütemező algoritmus belső paramétereit. A tesztelés során tehát a mi feladatunk a paraméterek legjobb kombinációjának meghatározása. Bizonyos paraméterek esetében könnyen látható, hogy az adott paraméter változtatása milyen hatással jár, másoknál erről csak homályos elképzeléseink vannak, és a tesztelés feladata a paraméter legkedvezőbb értékének megtalálása.

A kezdeti populáció méretének növelése pl. várhatóan hatékonyabb ütemezést tesz lehetővé, de lassítja az algoritmust. A lépésszám növelése egy határig jelentősen befolyásolja a hatékonyságot, míg egy bizonyos korlát felett nem tud javítani. A kezdeti populáció érvényes egyedeinek arányát megnövelve viszonylag sok érvényes egyedünk lesz azonnal, és nem kell az algoritmusnak erre "erőt pazarolnia", de ezzel kezdetben egy kevésbé változatos populációból indulunk ki, ami lehet hátrányos hatású. A mutációs és rekombinációs arány meghatározásakor leginkább a próbálgatás és a szakirodalomban található korábbi tapasztalatok segíthetnek.

A genetikus algoritmus lelke a célfüggvény, amit maximalizálni akarunk. Az algoritmusnak két változata született, amelyek csak a célfüggvényben térnek el egymástól. A tesztelés talán legfontosabb feladata eldönteni, hogy a két célfüggvény közül melyik a jobb.

Végezzünk egy közelítő becslést, hogy kb. hány tesztesetet kell futtatnunk. Látható, hogy hét bementi paramétert kell kezelnünk, ami paraméterenkénti három változat esetén kb. 2400 tesztesetet jelent. Egy további kettes szorzó a két célfüggvény változat, azaz minden egyes benchmarkhoz kb. 5000 tesztesetet kell lefuttatni. A tesztelés során három feladattal foglalkoztunk, ez tehát kb. 15000 futtatást jelent. Ilyen mennyiségű teszteset futtatása, valamint az eredmények feldolgozása rendkívül nehéz és összetett feladat, és természetesen a manuális tesztelést azonnal el kell vetnünk. A tesztelés menetéről lásd a 7.2 fejezetet.

7.1.2. CCLS

A CCLS tesztelése ennél egyszerűbb feladat, mivel ott csak az alábbi 3 bemenő paramétert kell kezelni:

- egyszerre kezelendő csúcsok száma (*grp*)
- újraindítási idő
- lappangási idő

Ebből nekünk csak a csoportméret ideális értékét kell tesztelés útján meghatározni, a másik két paraméter a későbbiekben is bemenetként rendelkezésre fog állni.

Az egyetlen probléma, ami a CCLS tesztelése során felmerült, sokkal gyakorlatibb jellegű: a nagyobb teszteseteknél (több száz pontú gráfnál) erőforráshiány miatt nem tudtuk lefuttatni a programot. Ekkor ugyanis olyan mennyiségű korlát felvételére van szükség, melyek mind démonként figyelnek, hogy betelítik a memóriát, és a folyamatos diszk elérések miatt a program jelentős mértékben lelassul, sőt a legnagyobb teszteknél le is áll. Ezért ezt az algoritmust a legnagyobb benchmarkra (RC6) nem teszteltük. Jelen pillanatban folyamatban

van az algoritmus implementációjának tökéletesítése, mely lehetővé teszi majd nagyobb tesztekre való futtatását.

7.2. Tesztelés menete

Algoritmusainkat három különböző feladaton teszteltük, melyek az alábbiak voltak:

- Gyors Fourier transzformáció (FFT), 25 csúcs
- IDEA kriptográfiai algoritmus, 116 csúcs
- RC6 kriptográfiai algoritmus, 328 csúcs

Az előző fejezetben érzékeltettük, hogy milyen nagyszámú teszteset lefuttatására van szükség minden egyes feladatnál az alapos teszteléshez. Erre a célra készítettünk egy Tcl script-et, amely futtatja a teszteket. A script az alábbi lépéseket hajtja végre:

- Futtatja a PIPE-ot az adott tesztesetre addig, amíg el nem ér az ütemezéshez. A PIPE ezzel előállítja az ütemezés bemenetét.
- Futtatja a PIPE-ban lévő force-directed ütemezőt
- Futtatja a PIPE-ban az allokációt a force-directed ütemező kimenetén
- Futtatja a SICStus-on a CCLS ütemezőt különböző *grp* értékekkel
- Futtatja a PIPE-ban az allokációt a CCLS ütemező összes kimenetén
- Futtatja a genetikus ütemező algoritmus mindkét változatát a paraméterek különböző konfigurációiban
- Futtatja a PIPE-ban az allokációt minden verzió minden konfigurációjának összes legjobb egyedén. Ez minden benchmark-nál több tízezer allokációt jelent!
- Kiírja egy fájlba minden ütemezés idejét és a felhasznált processzor számot olyan formában, hogy azt később Excel-ben könnyen feldolgozhassuk. A kimenet egy sora az alábbi formátumú:
  Teszt_név fd_idő fd_proc gen_idő gen_proc megoldások_száma CCLS_idő CCLS_proc csoport_méret
  (Itt az *fd* előtag a force-directed, a *gen* előtag a genetikus algoritmusra utal.)
  Egy példa erre:
  IDEA-100-268-50-0.2-0.3-300-0.1 56.29236 17 30.74035 16 37 312.076 16
  Itt tehát az IDEA tesztet futtattuk R=100, L=268 értékekkel, a populáció 50 egyedes, a mutációs arány 0.2, a rekombinációs arány 0.3, 300 lépést hajtunk végre, és a kezdeti

populációban 0.1 arányban helyezünk el érvényes egyedeket. Ekkor a forced directed algoritmus kb. 56 másodpercig futott, és 17 processzor kell az ütemezés megvalósítására. A genetikus algoritmus kb. 31 másodpercig futott, 16 processzorra van szüksége, és ilyen megoldásból 37 darabot készített. A CCLS 312 másodpercig futott, és szintén 16 processzort igénylő megoldást talált.

A tervezett tesztesetek elvégzése még a fenti script segítségével is nagyon bonyolult feladat. Említettük, hogy milyen nagyszámú tesztesetről van szó, és tovább nehezíti a problémát, hogy egyes konfigurációkban egy teszt lefuttatása hosszú ideig tarthat. A teljes teszt idejének becslésére megnéztük, hogy egy Pentium II-400-as gép egy éjszaka alatt a tesztek mekkora részét képes elvégezni. Az eredmény alapján úgy tűnt, hogy egy ilyen kapacitású számítógépen kb. egy-két hónapba telne az összes teszt elvégzése. Szerencsére a feladat jól párhuzamosítható, így egyszerre több számítógépet állítottunk munkába. A tesztelés 4-8 gépen futott szimultán, és így kb. egy hét alatt sikerült lefuttatni. Ehhez is zömmel Pentium II processzorral ellátott gépeket használtunk, Debian GNU/Linux operációs rendszer alatt.

7.3. Eredmények összehasonlítása

Az eredmények feldolgozását nagymértékben segítette, hogy a tesztek kimenete az Excel számára beolvasható formátumban készült, és így a diagrammok és táblák készítéséhez támogatást kaptunk.

7.3.1. Processzorok száma

A processzorok száma szerinti összefoglaló eredményeket az 1. táblázatban láthatjuk. Az itt megadott értékek az adott tesztre talált legjobb eredmények, tehát például a genetikus algoritmusoknál több száz konfiguráció közül az adott tesztre optimális eredményt közöltük. A – jel azt jelenti, hogy az adott algoritmus valamilyen oknál fogva nem tudott érvényes kimenetet produkálni a tesztesetre. (A force-directed ütemező esetében ez valamilyen programozói hiba folytán lépett fel, melynek javítása folyamatban van. A genetikus algoritmusnál előfordulhat, hogy nem születik érvényes egyed, ilyenkor a végső verzióban jobb híján az ASAP vagy ALAP ütemezést adhatjuk vissza, azonban a tesztelt verzióba ezt még szándékosan nem építettük bele. A CCLS esetében erőforrás problémák miatt nem sikerült minden tesztet futtatni, amire már korábban utaltunk.)

Látható, hogy az új algoritmusok minden tesztesetre elérték a korábbi force-directed ütemező eredményét, sőt majdnem minden tesztnél javítottak rajta. Ez a javítás sokszor nagyon

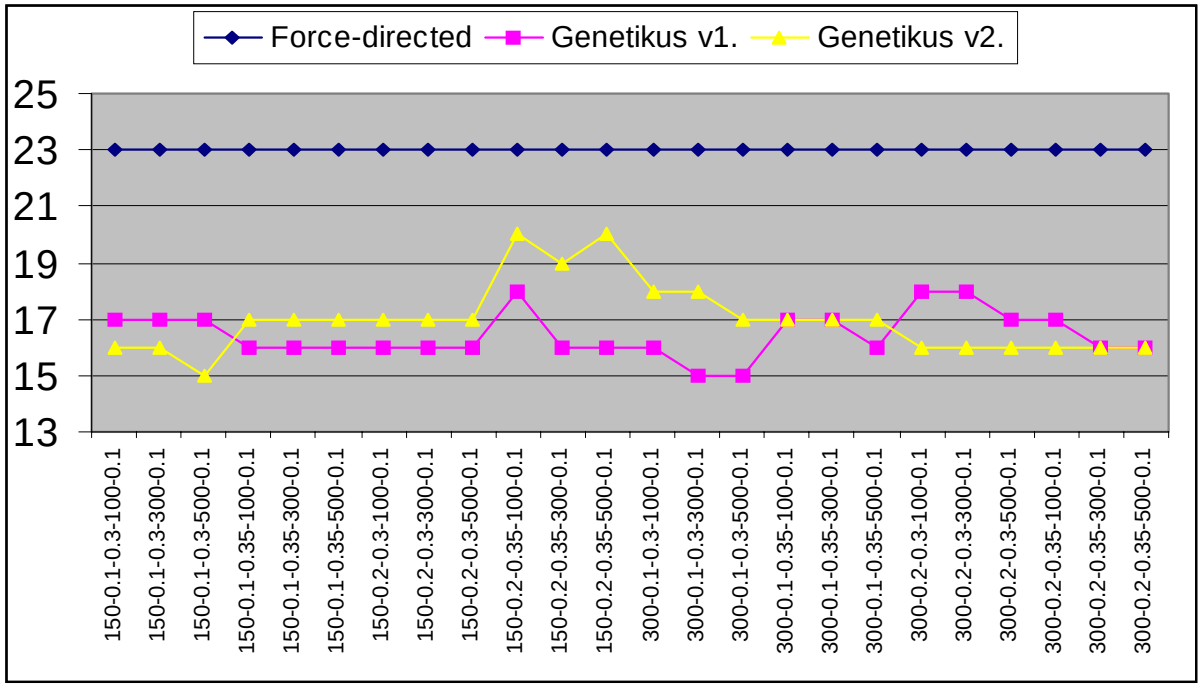
jelentős volt, pl. az FFT R=20 L=30 esetben kb. fele annyi processzorra volt szüksége a genetikus algoritmusnak, mint a force-directed ütemezőnek. Látható, hogy a genetikus algoritmus nagyobb tesztekkel is jól megbirkózik, hiszen pl. az RC6 R=10 L=211 tesztnél a korábbi 23 processzorszámot 15-re tudta javítani. Az is figyelemre méltó, hogy az RC6 R=10 L=201 eset kb. 200 processzorához képest a latency 10 egységgel való növelése a processzorszám tizedére csökkenését eredményezte.

A CCLS futtatásával többször voltak problémák, de látható, hogy amikor lefutott, szintén elég jó eredményt produkált. Volt egy olyan eset is (IDEA R=278 L= 278), amikor ez az algoritmus érte el a legjobb eredményt. A *grp* értékét 2-3-nak választva kaptuk a legjobb eredményeket.

	Force-directed	Genetikus v1.	Genetikus v2.	CCLS
FFT R=20 L=20	9	9	9	9
FFT R=20 L=30	11	6	-	7
IDEA R=10 L=316	-	74	74	-
IDEA R=100 L=268	17	15	15	16
IDEA R=100 L=278	16	15	16	16
IDEA R=200 L=268	13	10	11	11
IDEA R=268 L=268	6	6	6	6
IDEA R=278 L=278	8	7	7	6
IDEA R=50 L=268	25	25	25	26
IDEA R=50 L=278	29	23	23	25
RC6 R=10 L=201	210	207	207	-
RC6 R=10 L=211	23	15	15	-
RC6 R=100 L=201	25	23	23	-
RC6 R=201 L=201	13	11	11	-

1. táblázat: Az igényelt processzorok száma

A genetikus algoritmusnál érdekes megfigyelni azt is, hogy egy adott teszteseten belül a paraméterek változtatása hogyan befolyásolja az eredményt. Ezt illusztráljuk az RC6 R=10 L=211 teszt egy részletével a 2. táblázatban. Az egyes konfigurációkat *populáció_mérete-mutációs_arány-rekombinációs_arány-lépésszám-érvényességi_arány* formában adtuk meg. Látható, hogy a két verzió elég eltérő eredményeket produkál azonos paraméterezés esetén is, így a legnagyobb befolyásoló hatása a célfüggvény megválasztásának van. Az is látható, hogy az első verzió ért el ebben a tesztben jobb eredményeket, bár volt olyan konfiguráció is, amelyikben a második verzió adott optimálisabb ütemezést.



2. táblázat: Processzorszám alakulásakülönböző paraméterezés esetén

Ha a lépésszám paramétert figyeljük, akkor látható, hogy általában 100 lépés után is ugyanolyan eredményt értünk el, mint 300 vagy 500 lépéssel. Érdekes megfigyelni, hogy a populáció méretének 150-ről 300-ra való növelése sok esetben még romláshoz is vezetett, bár a legjobb eredményt egy 300-as populációnál értük el. Látható, hogy a mutációs- és rekombinációs arány változtatása meglehetősen rendszertelenül hat a végeredményre, elég nehéz valamilyen szabályosságot felismerni. Ez persze csak egy kiragadott részlet, de jól mutatja a paraméterek hangolásának problémáját.

Az alábbi táblázatban összefoglaljuk, hogy a genetikus algoritmus első verziója az 1. táblázat eredményeit melyik konfiguráció(k)ban érte el.

	pop_size	mut_rate	rec_rate	Step	valid_rate
FFT R=20 L=20	50	0.1	0.3	100	0.1
	50	0.2	0.3	100	0.1
	50	0.2	0.35	100	0.1
FFT R=20 L=30	300	0.2	0.3	500	0
IDEA R=10 L=316	50	0.1	0.35	400	0.1
	500	0.5	0.2	400	0.5
IDEA R=100 L=268	300	0.2	0.3	100	0
IDEA R=100 L=278	300	0.1	0.35	100	0.1
IDEA R=200 L=268	500	0.5	0.35	100	0.1
IDEA R=268 L=268	150	0.1	0.35	100	0
	150	0.2	0.35	300	0

	150	0.1	0.3	300	0.1
	300	0.1	0.35	100	0
	300	0.2	0.35	100	0
	300	0.1	0.3	100	0.1
	300	0.2	0.3	100	0
	300	0.2	0.3	100	0.1
	300	0.2	0.35	100	0.1
IDEA R=278 L=278	50	0.1	0.35	100	0.1
	150	0.1	0.3	100	0.1
	150	0.1	0.35	100	0.1
	150	0.1	0.35	100	0.1
	150	0.2	0.35	300	0.1
	300	0.1	0.3	100	0.1
	300	0.1	0.35	100	0.1
	300	0.2	0.3	100	0.1
	300	0.2	0.35	100	0.1
IDEA R=50 L=268	50	0.1	0.3	300	0.1
	150	0.1	0.3	100	0
	150	0.1	0.3	100	0.1
	150	0.2	0.3	100	1
	150	0.2	0.35	100	0.1
	300	0.2	0.3	100	0
	300	0.2	0.35	100	0
IDEA R=50 L=278	500	0.5	0.35	400	0.5
RC6 R=10 L=201	sok esetben	sok esetben	sok esetben	sok esetben	sok esetben
RC6 R=10 L=211	300	0.1	0.3	500	0.1
RC6 R=100 L=201	sok esetben	sok esetben	sok esetben	sok esetben	sok esetben
RC6 R=201 L=201	sok esetben	sok esetben	sok esetben	sok esetben	sok esetben

3. táblázat: A legjobb processzorszámhoz tartozó konfigurációk

A fenti táblázatban a lépésszám mezőbe mindig csak a legkisebb olyan lépésszám került be, ahol az adott eredményt elértük, mert feltehetően további lépések végrehajtásával az eredmény nem romlik el. Látható, hogy nagyon sok helyen már 100 lépés után megkapjuk a legjobb eredményt, és a további lépések már nem hoznak javulást. (Általában 500 lépésig teszteltünk, de egyes esetekben 1000 lépést is kipróbáltunk.)

A többi paraméternél sajnos nem ennyire egyértelmű a helyzet. A populáció méretének növelése, az elvárásokkal ellentétben, bizonyos esetekben egyáltalán nem javított, sőt akadt olyan teszt is, ahol rontott. Összességében az mondható el, hogy 300 egyednél nagyobb populációra már nincs szükség, de talán még ennél is lehet lejjebb menni. Kezdeti érvényes

egyedek behelyezése a populációba általában jótékony hatással járt, de ennek az aránynak kicsinek, mondjuk 0.1-nek kell lennie.

A mutációs és rekombinációs arány optimális meghatározása talán a legnehezebb feladat. A fenti táblázat alapján azt láthatjuk, hogy az optimális mutációs arány 0.1-0.2-ig változik, míg a rekombinációs arány 0.3 és 0.35 között mozog.

Felhívnánk még egyszer a figyelmet arra, hogy a fenti táblázatban csak a legjobb megoldásokhoz tartozó kombinációkat tüntettük fel, és sok tesztesetben előfordult, hogy nagyon sok majdnem ugyanolyan jó megoldás született, és néhány ennél egy processzorral jobb. Ilyen esetben a fenti táblázat csak a kiemelkedőket mutatja, holott a többi sem sokkal rosszabb, és lehet, hogy a paraméterek hangolásánál ezeket is figyelembe kell venni.

7.3.2. Futási idő

Mielőtt ismertetnénk a konkrét futási időket, elöljáróban megjegyezzük, hogy ebben a feladatban nem a futási idő általában a kritikus mennyiség, nem ez a fontosabb paraméter, hanem a processzorok száma. A gyakorlati alkalmazásokban még az is elfogadható eredmény, ha az ütemezés néhány óra alatt fut csak le, mivel a tervezés során nem kell az algoritmust sokszor egymás után lefuttatni, hanem tipikusan csak néhányszor. Látni fogjuk, hogy mindkét algoritmus futási ideje még a legrosszabb esetben is az előbbi korlátnál lényegesen kisebb.

Mivel az algoritmusok különböző programozási nyelven íródtak, továbbá eltérő kapacitású gépeken teszteltük őket, ezért az időadatok tulajdonképpen csak nagyságrendi viszonyokat tükröznek. Az azonban jól látszik, hogy a futási idő nem robban exponenciálisan a bemenettel.

Az 1. táblázat eredményeinek idejét a 4. táblázatban közöljük. Ezek tehát nem a leggyorsabb futások, hanem a legjobb processzorszámhoz tartozó idők. A táblázat ötödik oszlopában egyes esetekben feltűntettük, hogy mennyire felgyorsul a genetikus algoritmus , ha eggyel több processzor elérésével is megelégedünk.

	Force-directed	Genetikus v1.	Genetikus v2.	Eggyel több proc. esetén	CCLS
FFT R=20 L=20	0.99 s	1.27 s	0.51 s		3.99 s
FFT R=20 L=30	2.91 s	13,22 s	-	0.42 s	29.78 s
IDEA R=10 L=316	-	98.24 s	93.76 s		-
IDEA R=100 L=268	56.29 s	54,18 s	51.72 s	10.09 s	312.07 s
IDEA R=100 L=278	1779.3 s	49.77 s	26.12 s		687.25 s
IDEA R=200 L=268	158.08 s	432,55 s	32.31 s	29.24 s	564.15 s

IDEA R=268 L=268	118.68 s	28.74 s	57.88 s		95.22 s
IDEA R=278 L=278	1149.11 s	9.14 s	8.55 s		737.18 s
IDEA R=50 L=268	37.14 s	28.3 s	10.71 s		550.02 s
IDEA R=50 L=278	519.61 s	315.12 s	882,59 s	87.62 s	-
RC6 R=10 L=201	165.67 s	17.79 s	35.83 s		-
RC6 R=10 L=211	1069.23 s	1984,02 s	1101,48 s	116.04 s	-
RC6 R=100 L=201	399.73 s	50.28 s	54.23 s		-
RC6 R=201 L=201	1661.60 s	247.70 s	249.31 s		-

4. táblázat: A legjobb processzorszámhoz tartozó futási idők

A táblázatból az látszik, hogy ezen adatok alapján nem egyértelmű, hogy a genetikus és a force-directed algoritmus közül melyik a gyorsabb. Mindenesetre elég meglepő, hogy mindkét algoritmus javára igen nagy eltérések adódtak különböző teszteseteknél. Fontos figyelembe venni, hogy a genetikus algoritmus sokszor a legjobb eredményt egy hosszú ütemezés eredményeként adja, és ha már csak eggyel rosszabb processzorszámmal megelégszünk, akkor időben óriási nyereséget érhetünk el. Másképp fogalmazva, a genetikus algoritmust viszonylag korán leállítva is várhatóan csak egy processzorral kapunk rosszabb ütemezést, mintha még hosszú ideig futtatnánk. Ha a táblázatra úgy tekintünk, hogy ahol szerepel, ott az ötödik oszlop adatát vesszük figyelembe, akkor látszik, hogy a genetikus algoritmus általában jelentős mértékben gyorsabb a többinél.

A CCLS eléggé lassúnak bizonyult, de ez várhatóan az implementáció tökéletesítésével jelentősen javítható.

7.4. Értékelés

Összefoglalva megállapíthatjuk, hogy mindkét algoritmus hatékonynak bizonyult a processzor számának minimalizálásában, és általában jelentős javításokat sikerült elérni a korábbi ütemezőhöz képest. A futási idők tekintetében a genetikus algoritmus meglepően jó eredményeket produkált, de a CCLS is kivárható időn belül elvégzi az ütemezést.

A legjobb ütemezéseket a genetikus algoritmus első verziója adta, de a második verzió sem marad el számottevően mögötte. Futási idő tekintetében a genetikus algoritmus második verziója volt kis mértékben jobb a többinél.

A genetikus algoritmus egy további előnye, hogy egyszerre több, hasonlóan jó megoldást talál, melyek közül a tervező esetleg további szempontok alapján kiválaszthatja a legjobbat.

A CCLS rendelkezik azzal a tulajdonsággal, hogy a futását bármikor megszakítva is érvényes, jó ütemezést szolgáltat. Ennek akkor lehet jelentősége, ha nem akarjuk kivárni a teljes

optimalizálást, hanem sürgősen eredményre van szükségünk. A CCLS további előnye, hogy a definiált lokális keresésen belül bizonyíthatóan optimális eredményt ad.

8. Összefoglalás

Az ipari feladatok komplexitásának és sebességigényének növekedésével, a számítógéppel támogatott tervezés manapság egyre nagyobb jelentőségűvé vált. Korábban is lehetőség volt hardver-leírásból automatikusan előállítani az optimális hardvert, de eddig kevesen foglalkoztak azzal, hogy egy magas szintű specifikációból optimálisan hozzák létre a hardver struktúrát.

Ennek a tervezési fázisnak a legnehezebb és legkritikusabb része az ütemezés. Dolgozatunkban ennek a problémának a megoldására ismertettünk két új algoritmust és összevetettük korábbi megoldásokkal. Az egyik egy genetikus algoritmus, a másik logikai programozáson alapszik. Bár a két megközelítés igen eltérő, mégis mindkét eljárás hatékonynak bizonyult. A körültekintő tesztelés eredményeképpen kiderült, hogy munkánkkal jelentős mértékben sikerült megjavítani az előző ütemező algoritmusokat, és ezzel a teljes tervezés hatékonyságát növeltük meg.

9. Függelék

9.1. NP-teljesség bizonyítása

Azt szeretnénk belátni, hogy a 3.2. fejezetben definiált ütemezési probléma NP-teljes. Ehhez először eldöntési problémává kell átfogalmazni:

Adott egy elemi műveleti gráf, adott az egyes műveletek típusa, a típusokhoz tartozó végrehajtási idők és megvalósítási költségek, valamint az egyes csúcsok ASAP és ALAP ideje és az újraindítási idő (R). Adott ezen kívül egy K természetes szám. Kérdés: van-e olyan érvényes ütemezés, amelynél az egyes típusú kompatíbilis párok számának súlyozott összege legalább K?

Állítás: ez a feladat NP-teljes.

Bizonyítás. Az NP-teljességhez két dolgot kell belátni: egyrészt azt, hogy a probléma NP-beli, másrészt pedig azt, hogy minden NP-beli probléma visszavezethető rá.

A probléma NP-beliségéhez azt kell meggondolni, hogy igenlő válasz esetén – megfelelő extra információ segítségével – a válasz helyessége polinomidőben ellenőrizhető. Valóban, ha a válasz az, hogy van olyan ütemezés, amelyre a kérdéses mennyiség értéke legalább K, akkor extra információnak magát az ütemezést kérve, polinomidőben tudjuk ellenőrizni, hogy az ütemezés valóban érvényes-e (ehhez minden élre ellenőrizni kell az axiómák teljesülését), valóban ASAP és ALAP között van-e (ezt minden csúcsra kell ellenőrizni), valamint össze kell számolni a kompatíbilis párokat (ezt csúcspáronként kell számolni), és meg kell nézni, hogy a megfelelő súlyozott összeg valóban legalább K. A teljes ellenőrzés időigénye tehát $O(e)+O(n)+O(n^2)=O(n^2)$, ami valóban az input méretének polinomja. (A gráf csúcsainak száma n, éleinek száma e.)

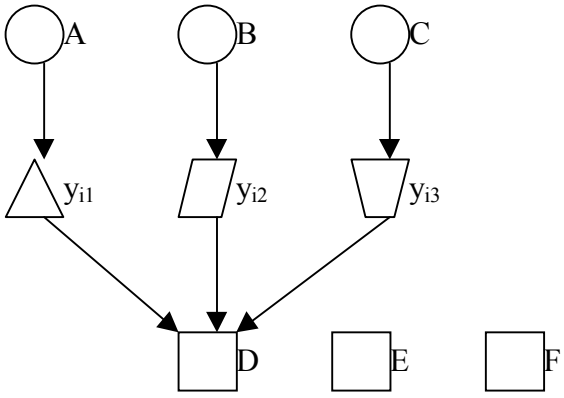
A nehezebb annak bizonyítása, hogy a feladatra minden NP-beli feladat polinomidőben visszavezethető (Karp redukció). Ehhez elég azt belátni, hogy egy – már bizonyítottan – NP-teljes feladat polinomidőben visszavezethető az ütemezés problémájára. Azt fogjuk megmutatni, hogy a 4.1.3. fejezetben definiált 3-SAT probléma redukálható az ütemezés problémájára. Ehhez feltesszük, hogy van egy olyan szubrutinunk, mely egy lépésben megoldja az ütemezés problémáját. Van továbbá egy Boole-kielégíthetőségi problémánk, amit ennek segítségével szeretnénk megoldani. A kielégíthetőségi probléma egy

$$F=(y_{11}+y_{12}+y_{13})(y_{21}+y_{22}+y_{23})\ldots(y_{k1}+y_{k2}+y_{k3})=1$$

egyenlet megoldhatóságát jelenti. Az egyenlet tulajdonképpeni változói az $x_1,\ldots,x_m$ bináris változók; az y_{ij} szimbólumok nem külön változók, hanem mindig valamilyen x_l változó ponált vagy negált értékét jelölik. Az összeadás diszjunkciót, a szorzás pedig konjunkciót jelöl.

Ehhez a feladathoz akarunk konstruálni egy elemi műveleti gráfot, melyre az ütemezési szubrutin az eredeti feladat megoldását fogja szolgáltatni. A konstrukció a következő. Minden egyes x_i változóhoz felveszünk két csúcsot: az egyik fogja a változó ponált, a másik pedig a negált értékét reprezentálni. E csúcsok mobilitási tartománya legyen az *[1,2]* intervallum. Ha egy csúcs az 1. időpillanatra ütemeződik, ez annak felel meg, hogy a megfelelő változó értékének 1-nek, ha pedig a 2. időpillanatra, akkor 0-nak kell lenni. Annak érdekében, hogy egy változó ponált és negált értékének megfelelő csúcsok ne ütemeződjenek ugyanakkorra, ezek típusa legyen egyező, így ha majd elég nagy *K* értékkel hívjuk meg a szubrutint, erről az ütemezés gondoskodik.

Vegyük ezek után a konjunkció egyik tényezőjét: $y_{i1}+y_{i2}+y_{i3}$. Ehhez a tényezőhöz már tartozik három csúcs; most felveszünk további hatot:



4. ábra: a konjunkció egy tényezőjének megfelelő EOG-részlet.

Az ábrán a különböző szimbólumok különböző, az azonosak pedig azonos típust reprezentálnak. Az A, B, C csúcsok mobilitási tartománya a *[0,1]* intervallum, a D csúcsé a *[2,3]* intervallum, az E és F csúcsoké pedig a *[3,3]* intervallum (vagyis ezek már le vannak rögzítve).

Nézzük, mi értelme van ennek. Azt akarjuk elérni, hogy a tényező értéke 1 legyen, vagyis, hogy az y_{i1}, y_{i2}, y_{i3} változók közül legalább az egyik értéke 1 legyen. Könnyen ellenőrizhető, hogy ha ebben az elrendezésben a kompatíbilis párok számát akarjuk maximalizálni, akkor ez

tényleg ahhoz vezet, hogy e három csúcs közül legalább az egyik az 1. időpillanatra ütemeződik, ugyanis ilyenkor két kompatíbilis pár keletkezik, különben pedig egy sem. Tehát, ha előírjuk, hogy az egész gráfban a kompatíbilis párok száma legalább *2k* legyen, akkor egy megfelelő ütemezés megoldja a kielégíthetőség problémáját.

Hátra van még annak végiggondolása, hogy az egyes típusok milyen súllyal legyenek képviselve a kompatíbilis párok számában. A fenti ábrán szereplő körnek és négyzetnek megfelelő típusok súlya legyen 1. Viszont azt kell garantálni, hogy az ugyanazon változó ponált és negált értékének megfelelő csúcs semmiképp se legyen konkurens, így az ezekhez a típusokhoz tartozó súly legyen olyan nagy, hogy egy ilyen kompatíbilis pár hiányát a másfajta kompatíbilis párok ne tudják pótolni. Esetünkben lehet ez például *2k+1*. Ha tehát az ütemező szubrutint $K=2k+m*(2k+1)$ értékkel hívjuk meg, akkor az, hogy van-e megfelelő ütemezés, ekvivalens azzal, hogy a Boole-formula kielégíthető-e. Mivel a pipeline üzemmódra nem volt szükség, így lehet $R=L=4$. Mindebből tehát következik, hogy az ütemezés feladata legalább olyan nehéz, mint a Boole-kielégíthetőség problémája, ezért NP-teljes. Ezt kellett bizonyítani.

9.2. Érvényes egyedek előállítása

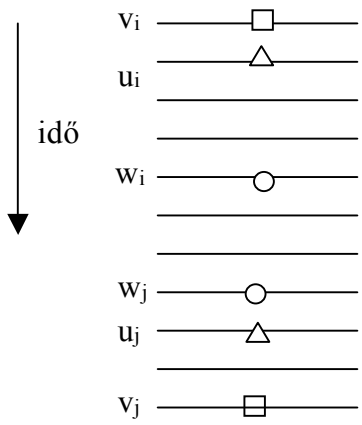
Ebben a fejezetben azt az általunk megfogalmazott állítást bizonyítjuk be, hogy két érvényes ütemezés adott arányú "keverése" is érvényes ütemezést ad. Ugyanez precízen:
Állítás: Legyen az egyik érvényes ütemezésben az *n* csúcs indítási ideje rendre $v_1, v_2,\ldots,v_n$, a másikban $w_1, w_2,\ldots,w_n$. Ekkor tetszőleges $0 \le \lambda \le 1$ számra az $int(\lambda w_1 + (1-\lambda)v_1), int(\lambda w_2 + (1-\lambda)v_2),\ldots,int(\lambda w_n + (1-\lambda)v_n)$ is érvényes ütemezés. (*int* az alsó egészrészt jelöli.)

Bizonyítás: Ahhoz, hogy egy ütemezés érvényes legyen, azt kell ellenőrizni, hogy sehol sem sérülnek az elemi műveleti gráfban adott precedenciák, azaz éllel összekötött csúcsoknál valóban fennáll-e a jó sorrend. Nyilván ehhez elég egy tetszőleges éllel összekötött csúcspárról megmutatni, hogy azok az új ütemezésben is eleget tesznek a precedenciafeltételeknek. Mutasson egy él az elemi műveleti gráfban az *i.* csúcsból a *j.* csúcsba. Mivel a két kiindulási ütemezés érvényes, ezért fennállnak a $v_i + d_i \le v_j$ és $w_i + d_i \le w_j$ összefüggések. Innen $d_i \le min(v_j - v_i , w_j - w_i) =: \delta$. Az $u_i = int(\lambda w_i + (1-\lambda)v_i)$ jelölést bevezetve azt kell belátnunk, hogy $u_i + d_i \le u_j$ is fennáll. Ehhez elég, hogy $u_j - u_i \ge \delta$. Szemléletesen ez azt jelenti, hogy az új ütemezésnek olyannak kell lennie, hogy az *i.* és *j.* csúcs távolsága nem lehet kisebb, mint a két kiindulási ütemezésben lévő távolságok

minimuma, hiszen különben elképzelhető lenne, hogy megsértjük a precedenciát. Az alábbiakban ezt fogjuk bebizonyítani.

Az általánosság megsértése nélkül feltehető, hogy $v_i \le w_i$. A v_i, v_j, w_i, w_j számok egymáshoz való viszonyától függően megkülönböztetjük az alábbi eseteket: (minden esetet egy ábra illusztrál, amelyek következőképpen értelmezendőek: azonos jelek azonos ütemezések két egymás utáni csúcsát jelentik.)

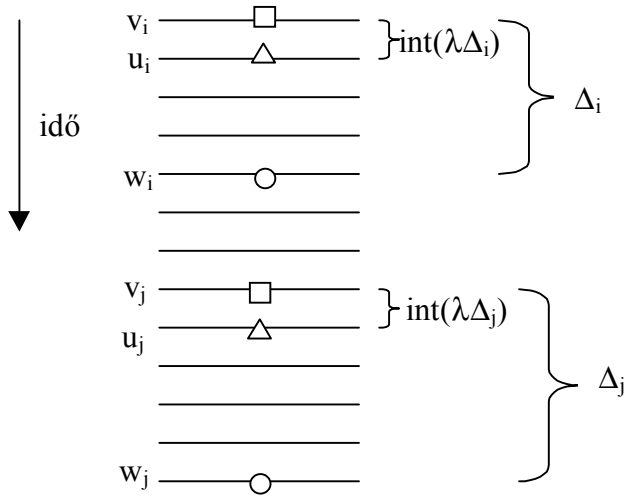
1. $v_j \ge w_j$ (5. Ábra). Ekkor az állítás triviális, hiszen ilyenkor $\delta = w_j - w_i$ és csupán annyit felhasználva, hogy $v_i \le u_i \le w_i$ és $w_j \le u_j \le v_j$ már adódik, hogy $u_j - u_i \ge w_j - w_i = \delta$, amit be akartunk látni.



5. Ábra

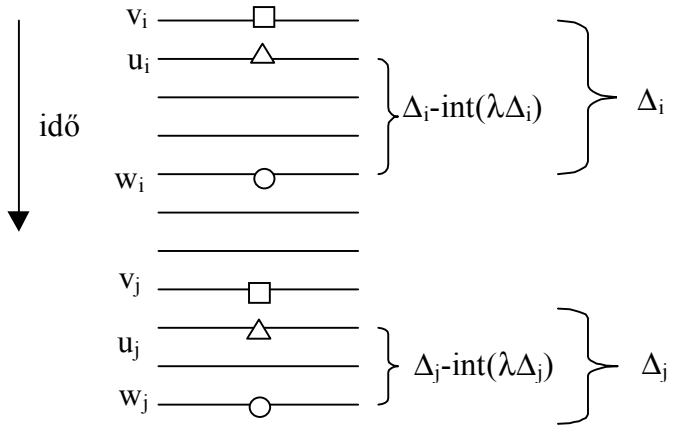
2. $v_j \le w_j$. A továbbiakban vezessük be a $\Delta_i := w_i - v_i$ és $\Delta_j := w_j - v_j$ jelöléseket.

   2.1. $w_i - v_i \le w_j - v_j$ (6. Ábra). Ekkor a feltétel tehát az, hogy $\Delta_i \le \Delta_j$. Ebből következik, hogy $\lambda\Delta_i \le \lambda\Delta_j$ és innen $int\ (\lambda\Delta_i) \le int\ (\lambda\Delta_j)$. A feltételekből adódik, hogy ilyenkor $\delta = v_j - v_i$ és mivel $u_i = v_i + int\ (\lambda\Delta_i)$, valamint $u_j = v_j + int\ (\lambda\Delta_j)$, így $u_j - u_i = v_j + int\ (\lambda\Delta_j) - v_i - int\ (\lambda\Delta_i) \ge \delta$. Tehát valóban érvényes ütemezést kaptunk.



6. Ábra

2.2. $w_i - v_i > w_j - v_j$ (7. Ábra), azaz $\Delta_i > \Delta_j$. Innen $\lambda\Delta_i > \lambda\Delta_j$, amiből $int(\lambda\Delta_i) \geq int(\lambda\Delta_j)$. A feltételekből adódik, hogy ekkor $\delta = w_j - w_i$ és mivel $u_i = w_i - (\Delta_i - int(\lambda\Delta_i))$ valamint $u_j = w_j - (\Delta_j - int(\lambda\Delta_j))$, így $u_j - u_i = w_j - w_i + (\Delta_i - int(\lambda\Delta_i)) - (\Delta_j - int(\lambda\Delta_j))$. Másrészről a $\Delta_i > \Delta_j$ feltételből kapjuk, hogy $(1-\lambda)\Delta_i > (1-\lambda)\Delta_j$, azaz $\Delta_i - \lambda\Delta_i > \Delta_j - \lambda\Delta_j$. A bal oldalt nem csökkentve szintén fennáll az egyenlőtlenség $\Delta_i - int(\lambda\Delta_i) > \Delta_j - \lambda\Delta_j$. Innen $\Delta_i - int(\lambda\Delta_i) \geq \Delta_j - int(\lambda\Delta_j)$ adódik, azaz $(\Delta_i - int(\lambda\Delta_i)) - (\Delta_j - int(\lambda\Delta_j)) \geq 0$, így $u_j - u_i \geq \delta$ ebben az esetben is fennáll.



7. Ábra

Mivel minden esetben fennállt, hogy $u_j - u_i \geq \delta$, azaz az új ütemezésben nem kerültek egymáshoz közelebb a pontok, mint a kiindulási ütemezésekben előforduló minimális távolság, így az állítást bebizonyítottuk.

10. Irodalomjegyzék

[Arato1994] P. Arató, I. Béres, A. Rucinski, R. Davis, R. Torbert: A high-level datapath synthesis method for pipelined structures. Microelectronics Journal, 25:237-247, 1994.

[Arato2000] P. Arató, T. Visegrády, I. Jankovits, Sz. Szigeti: High-Level Synthesis of Pipelined Datapaths. Panem, 2000.

[Camposano1990] R. Camposano: From behaviour to structure: High-level synthesis. IEEE Design & Test of Computers, 10:8-19, 1990.

[Camposano1991] R. Camposano, W. Wolf: High-Level VLSI synthesis. Kluwer Academic Publisher, 1991.

[Davis1991] L. Davis: Handbook of genetic algorithms, New York, Van Nostran Reinhold, 1991

[Gajski1992] D. Gajski: High-level synthesis. Kluwer Academic Publisher, 1992.

[Hwang1991] C.-T. Hwang, J.-H. Lee, Hsu Y.-C.: A formal approach to the scheduling problem in high-level synthesis. IEEE Transactions on Computer Aided Design, April 1991.

[IEEE1993] IEEE. Special issue on high-level synthesis. IEEE Transactions on Very Large Scale Integration Systems, 1(3), September 1993.

[Jerraya1998] A. A. Jerraya, J. Mermet: System-level synthesis. Kluwer Academic Publisher, 1998.

[Jordan1995] Jordán T., Recski A.: Kombinatorikus optimalizálás, BME, 1995

[Kinnebrock1994] W. Kinnebrock: Optimierung mit genetischen und selektiven Algorithmen, Oldenburg, 1994

[Kirkpatrick] S. Kirckpatrick, C. D. Gelatt, JR., M.P. Vecchi : Optimization by simulated annealing, Science

[Micheli1994] G. De Micheli: Synthesis and optimization of digital circuits. McGraw-Hill, 1994.

[Paulin1989] P. G. Paulin, J. P. Knight: Force-directed scheduling for the behavioural synthesis of ASICs. IEEE Transactions on Computer Aided Design, 1989.

[Schrijver1998] A. Schrijver: Theory of linear and integer programming, Wiley, 1998

[Szeredi1998] Szeredi P., Benkő T. : Bevezetés a logikai programozásba. BME, 1998.

[Szeredi2000] Szeredi P.: Nagyhatékonyságú logikai programozás, előadásjegyzet, 2000