

TDK dolgozat

Klein Ferenc

Nemes Zsolt Ádám

Klein Ferenc, programtervező informatikus III. évfolyam

Nemes Zsolt Ádám, mérnök informatikus III. évfolyam

ÜTEMEZÉSI FELADATOK MEGOLDÁSÁNAK GYORSÍTÁSA HEURISZTIKUS ELJÁRÁSOKKAL

Témavezető: Hegyháti Máté, PhD hallgató

Pannon Egyetem, Műszaki Informatikai Kar, Rendszer- és Számítástudományi Tanszék

Tartalomjegyzék

1. Bevezetés	1
2. Szakirodalmi áttekintés	3
2.1. A feladatosztály nehézsége	5
2.2. Ismert algoritmusok	6
2.3. Gyakorlati előfordulás	10
3. Algoritmusok kiterjesztése FFC feladatosztályra	11
3.1. Kidolgozott heurisztikák	11
3.2. Implementáció	15
4. Eredmények	17
4.1. Heurisztikák jósága	18
4.2. Heurisztikák összehasonlítása	20
4.3. Megoldó teljesítménye heurisztikákkal	21
5. Összefoglalás	24

1. Bevezetés

Az ütemezés napjainkban egyre nagyobb szerepet kap a benne rejlő lehetőségek miatt. Feladata az erőforrások megfelelő allokációja a taszkokhoz, adott időn belül valamilyen - akár több - célfüggvény alapján. Az ütemezés felhasználása széleskörű lehet: támogathat vegyipari gyártási műveleteket [2], szállítási, logisztikai problémákat, vagy egyéb üzemekben előforduló helyzeteket (munkaerő vagy termelési szerkezet meghatározása). Fontos szerepe van az ütemezésnek a processzor (CPU) működésében is, azáltal, hogy az operációs rendszer a folyamatokat vezérli (futtatja, megszakítja, várakoztatja).

A szűkösök világában egyre jelentősebb az optimalizálás szerepe az ipari üzemek számára. Az információáramlás folyamatos gyorsulása miatt a rendszereknek, ezáltal a modelleknek is követniük kell ezt a folyamatos haladást. Bár az elsődleges célok (például profitmaximalizálás) továbbra is változatlanok, az üvegházhatás gyorsuló tendenciája miatt nem szabad figyelmen kívül hagyni az ökológiai lábnyomot. Ez a kifejezés azt jelenti, hogy adott technológiai fejlettség mellett egy emberi társadalomnak milyen mennyiségű földre és vízre van szüksége önmaga fenntartásához és a megtermelt hulladék elnyeléséhez. Manapság az üzemek többsége igyekszik csökkenteni ezt a mutatót amellet, hogy a régi célokat is figyelembe veszik. Az ökológiai lábnyomon kívül az ipari rendszerek ütemezésének optimalizálása még számos célból történhet: ez lehet például adott termék legyártása a legkevesebb idő alatt (makespan minimalizálás), költségcsökkentés, profitmaximalizálás, adott idő alatt a lehető legtöbb termék legyártása (throughput maximalizálás), termékek legyártása határidőre (due-date) [16].

Ezen feladatok vizsgálata az 50-es évek elején került előtérbe, és szerepe napjainkra egyre jelentősebbé válik. Ennek oka a komplexebb üzemek kiépítése, melyek hatékony működtetése komoly kihívást jelent. Egy 1977-ben tartott konferencián A. H. G. Rinnooy Kan 9000-re becsülte a kategorizálható különböző determinisztikus problémák számát. Ez a szám azóta minden bizonnyal sokszorosára növekedett, és a mai napig is folyamatosan nő.

Ma már számos módszer létezik megoldásukra, melyeket legtöbbször egy megoldó szoftver (solver) végez el a pontos matematikai modell alapján. A feladatok komplexitásának fokozódásával a megoldók futási ideje is jelentősen nő. A különböző heurisztikus módszerek [12] nem feltétlenül adnak optimális megoldást, viszont legtöbbször jó közelítő értéket szolgáltatnak, amely a megoldó számára mind a keresési teret, mind a futásidőt nagymértékben lecsökkentheti. Ezt a redukáló hatást az algoritmusok pontossága, jósági aránya határozza meg.

Munkánk során mi az átlagos nehézségű feladatoknál egy bonyolultabb feladatosztályra fókuszáltunk, és ezen feladatosztálynak speciális eseteire készített algoritmusait terjesztettük ki általános használatra, majd a megszerzett tapasztalatok alapján saját heurisztikákat alkottunk. Ezek segítségével csökkentettük a megoldó futási idejét azáltal, hogy a heurisztikus algoritmusok eredményeinek minimumát véve kezdeti felső korlátot határoztunk meg a gyártási időre. Munkánk során a tanszéki kutató-csoport saját S-graph megoldó szoftverét használtuk a fent említett eljárások tesztelésére.

2. Szakirodalmi áttekintés

Ebben a fejezetben néhány alapvető fogalmat ismertetünk, melyek elengedhetetlenek az ütemezési feladatok megértéséhez [5, 4]. A teljesség igénye nélkül csupán azok a kifejezések fognak szerepelni, amelyek a dolgozatban felmerülhetnek a munkánk kapcsán. Minden fontos kifejezést dőlt betűvel szedtünk, valamint esetlegesen zárójelbe utána írtuk a hivatalos angol megnevezését. Többféleképpen lehet csoportosítani magukat az ütemezési feladatokat [17], például *determinisztikus*, illetve *sztochasztikus*. Sztochasztikus esetben felmerülhetnek a gyártás során bizonytalan események, melyeknek nem ismerjük a bekövetkezési valószínűségét. Determinisztikus feladatoknál minden tényezőt előre ismerünk, nem lehetséges bizonytalan esemény bekövetkezése. További osztályozási lehetőség az *offline* és *online* ütemezés. Offline esetben minden adatot előre ismerünk, és úgy végezzük el az optimalizálást, online esetben viszont néhány adatot csak a gyártási folyamatok közben tudhatunk meg, ami jóval komplexebbé teheti a problémát.

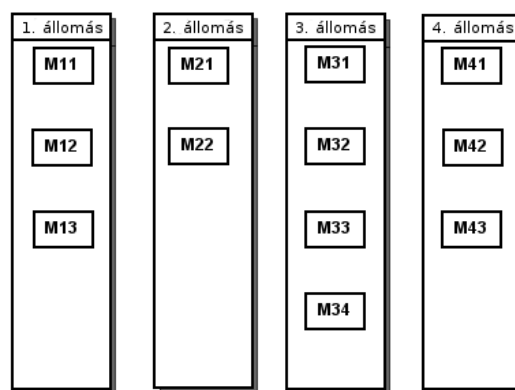
A feladatok számos paraméterrel rendelkezhetnek [13]. Ilyen például a *termék* (product, esetleg job), ami az elkészítendő feladatot jelenti. Egy termék gyártásához szükséges eszközt *berendezésnek* (equipment vagy machine) nevezzük. Legtöbb esetben, a feladatban szereplő összes berendezésnek csak egy kisebb halmaza tud legyártani egy-egy terméket, de identikus gépek esetén bármelyik képes erre. Egy adott feladatot bizonyos idő alatt tudunk legyártani egy berendezésen, ezt *végrehajtási időnek* (processing time) nevezzük. Bizonyos ütemezési feladatoknál nem csupán egyszerűen gépek vannak, hanem a gépek több különálló halmazt alkotnak, ezek az *állomások* (stage). Az állomások identikus berendezésekkel rendelkeznek. Fontos szempont komplex gyártási folyamatoknál, hogy az egyes gyártási fázisok között a terméket milyen módon, milyen mennyiségben, valamint mennyi ideig tárolhatjuk. Ezt nevezzük *tárolási stratégiának*. Ennek a csoportosítása többféle lehet, teljesség igénye nélkül ez lehet például tárhely szerint (FIS=Finite Intermediate Storage, UIS=Unlimited Intermediate Storage, NIS=Non Intermediate Storage, CIS=Common Intermediate Storage, MIS=Mixed Intermediate Storage), vagy az anyagok tárolási ideje szerint (ZW=Zero Wait, UW=Unlimited Wait, LW=Limited Wait) [18]. Az adott ütemezési feladatban foglalt termékek gyártásához szükséges időt *makespan*-nek (jelölés: C_{max}) nevezik, ennek a minimalizálása az egyik leggyakoribb célfüggvény a témakörben.

Ütemezési feladatokat le lehet írni matematikai jelölésekkel [8]. Erre egy $\alpha|\beta|\gamma$ hármast használunk. Az α mezőben szerepel a feladatosztály jelölése. A β mezőben találhatóak a gyártással kapcsolatos különböző korlátozások és extra információk, melyekből több is előfordulhat egyszerre, ilyen esetben felsorolás-szerűen kell megadni az információkat. Ha az ütemezési feladat nem rendelkezik ilyen extra információkkal vagy korlátozásokkal, akkor a β mező üresen marad. A γ mezőben van a célfüggvény.

Példa: $Pm|prmp|C_{max}$. Ebben a feladatban az α mezőben látható Pm m darab párhuzamos gépet jelöl, melyek identikusak (Parallel Machine Model). A β mező egy elemmel rendelkezik ($prmp$), ez azt jelöli, hogy a folyamatban lévő termék gyártása megszakítható, és később folytatható. A γ mezőben található célfüggvény a makespan minimalizálását jelöli.

Munkánk kezdetén a multiproduct, illetve a multipurpose [3] feladatosztályokat tűztük ki célul. Végül kutatásunk fő tárgyaként egy, a multiproduct-hoz nagyon hasonló feladatosztályt választottunk. Ennek a neve *Flexible Flow Shop* [11]. Jelölés: FFc . Az FFc a Flow Shop (jelölés: Fm) és a Parallel Machine (jelölés: Pm) osztályok keveréke, illetve ezek általánosítása [1].

Flow Shop-oknál m darab berendezés van. Minden terméknek ugyanabban a sorrendben kell végighaladnia a berendezéseken, az elsőtől kezdve. Miután egy berendezésen adott termék gyártása befejeződött, akkor a termék „beáll” a következő géphez a sorba. Mivel sorról beszélünk, így *FIFO* (First In First Out) elv szerint kerülnek gyártásra a termékek az egyes berendezéseken. A Parallel Machine feladatoknál m darab identikus berendezés helyezkedik el párhuzamosan, az adott terméket bármelyik gépen legyárthatjuk.



2.1. ábra. Flexible Flow Shop példa.

Flexible Flow Shop esetben c darab állomás van, minden állomás valamennyi berendezéssel (állomásonként eltérő lehet a számuk) rendelkezik, melyek identikusak [1]. Minden terméknek ugyanabban a sorrendben végig kell haladnia az állomásokon, az elsőtől kezdve. Állomásonként a terméknek szükséges eljárást bármelyik gép elvégezheti, hiszen identikusak. Minden állomáson más a termékek végrehajtási ideje a berendezéseken. Számos szakirodalom más megnevezést használ erre a feladatosztályra, például „hybrid flow shop”, vagy „multi-processor flow shop” [15]. Abban az esetben, amikor egy állomás van, és minden termék egyetlen (bármelyiken az állomásról) berendezésen megy keresztül, akkor Single Stage esetről beszélünk, később ezen a néven fogunk erre hivatkozni. A 2.1 ábrán látható példa egy flexible flow shop feladatot ábrázol,

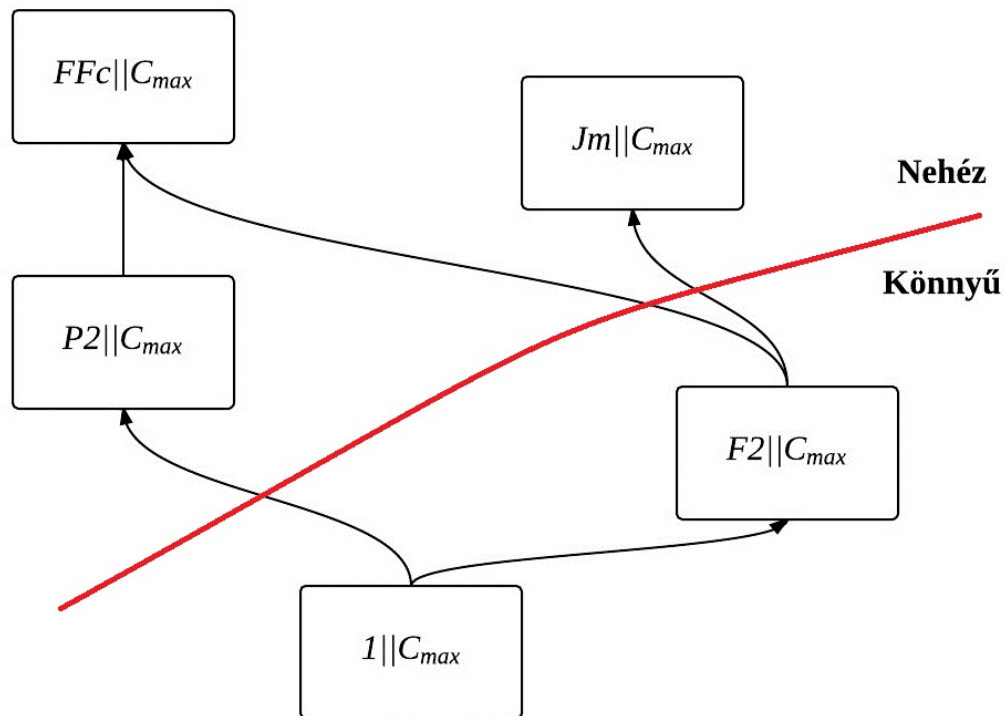
négy darab állomással. Az állomások függőleges oszlopokban helyezkednek el, minden állomás alatt pedig a megfelelő számú berendezés állomásonként (M_{xy}). Látható, hogy az első állomáson három, a másodikon kettő, a harmadikon négy, a negyedik állomáson pedig szintén három darab berendezés található.

2.1. A feladatosztály nehézsége

Vizsgálataink során legfőképp az $FFc||C_{max}$ ütemezési feladattal foglalkoztunk, melyben megengedtük a Single Stage eseteket is ($c = 1$). A Flexible Flow Shop feladatosztály elég komplex, Single Stage esetben is már NP -nehéz a feladat. Nem létezik általános algoritmus, amely garantálja az optimális megoldást. A legtöbb szakirodalomban fellelhető kutatás csupán ezen feladatosztály speciálisabb, egyszerűbb eseteit vizsgálta, és ezekre alkotott meg bizonyos algoritmust [9, 14, 10]. Ezek közül a lényegesebb eljárásokat a következő fejezet elején fogjuk részletesebben ismertetni. A probléma komplexitása miatt nem bonyolítottuk azt további megkötésekkel. Ezáltal az ütemezés típusa offline, valamint determinisztikus. A tárolási stratégia UIS, ami azt jelenti, hogy az egyes állomások között az anyagot korlátlan mennyiségben tárolhatjuk.

A 2.3 ábrán látható az általunk vizsgált feladat (FFc), valamint egyéb ütemezési feladatok komplexitása. $P2$ két párhuzamos gépet, Jm job-shopot, $F2$ két berendezéses flow shopot, 1 pedig az egyetlen géppel rendelkező ütemezési feladatokat jelölik. Egy téglalapból vezető nyíl speciális esete a nyíl másik oldalán lévő feladatosztálynak.

Saját vizsgálataink során arra törekedtünk, hogy ne csupán speciális esetekkel foglalkozzunk, hanem egységesen magára a feladatosztályra dolgozzunk ki gyors, egyszerű heurisztikákat, melyek megfelelő közelítő értéket adnak. A heurisztikák kidolgozása során az alapötleteket szakirodalomból származó példákból merítettük. Ezek az eljárások felső korlátot adnak a problémának, hogyha a cél adott termék legyártása a lehető legkevesebb idő alatt. Ezen felső korláttal az S-graph megoldó szoftver gyorsabban kiszámíthatja az optimális megoldást. Tehát célunk ezekkel a heurisztikákkal nem az volt, hogy önállóan oldjanak meg feladatokat, hanem a megoldóba beépítve segítsék a program működését, ha ilyen feladattal kerül szembe.



2.2. ábra. A flexible flow shop, és egyéb hasonló feladatosztályok komplexitása.

2.2. Ismert algoritmusok

Bemutatjuk az irodalomban fellelhető legismertebb algoritmusokat, melyek nem feltétlenül a flexible flow shop feladatosztállyal foglalkoznak, de megfelelő alapul szolgáltak saját heurisztikáink kifejlesztésében. Az algoritmusok pseudokódjait az alfejezet végén ismertetjük.

Johnson algoritmusa:

A fent megnevezett algoritmus nem a flexible flow shop feladatosztályra készült, hanem a flow shop feladatok egy speciális formájára, a permutációs flow shopra. A flow shop ütemezést az előző fejezetben ismertettük dióhéjban. A permutációs flow shop abban különbözik ettől, hogy a munkáknek a végrehajtási sorrendje minden berendezésre megegyezik. Annak ellenére, hogy ez az algoritmus nem flexible flow shop feladatot optimalizál, azért került bele a dolgozatunkba, mert ez egy nagyon fontos alap algoritmus a hasonló feladatokhoz. Rengeteg kutatás foglalkozott ennek az eljárásnak a kiterjesztésével nehezebb feladatokra, valamint számunkra is megfelelő alapötletként szolgált az egyik heurisztika kifejlesztésében. Az algoritmus 2 gép esetén oldja meg a permutációs flow shop problémát [6].

SPT-elv:

Tekintsük az $FFc|p_{ij} = p_j| \sum C_j$ problémát. p_{ij} fejezi ki az i -edik feladat végrehajtási idejét a j -edik berendezésen. Ebben az esetben $p_{ij} = p_j$, tehát minden berendezésen ugyanakkora a végrehajtási idő. Erre a problémára optimális megoldást ad single stage esetben, valamint olyan multistage esetekre is, amelyekben minden állomáson egy-egy berendezés van [8].

1. Tétel. *Az SPT-elv optimális megoldást ad $FFc|p_{ij} = p_j| \sum c_j$ feladatra, ha minden állomáson legalább annyi berendezés van, mint az azt megelőző állomáson.*

Egyéb esetekben ez az elv nem mindig ad optimális megoldást. Másik nagy hátránya vizsgálataink szempontjából, hogy nem makespan a célfüggvény. Ennek ellenére felhasználtuk későbbi munkánk során, mivel bizonyos esetekben nagyon közeli, vagy optimális megoldást tud produkálni.

LPT algoritmus:

Az alábbi eljárást csupán olyan feladatokon vizsgálták [7], ahol 2 állomás van, és ezeken az állomásokon meghatározott számú berendezéssel. A berendezések száma az FF után lévő zárójelben látható. Például az $FF(1, m)$, azt jelenti, hogy az első állomáson 1 darab berendezés, a második állomáson m darab berendezés található. Az algoritmus komplexitása $O(n \log n)$.

2. Tétel. $FF(1, m)|p_{ij} = p_j|C_{max}$ feladatra $\frac{C_{max}(LPT)}{C_{max}(OPT)} \leq \frac{\sqrt{1+m^2}+1+m}{2m}$, ahol m a második állomáson lévő berendezések száma.

3. Tétel. $FF(1, 2)|p_{ij} = p_j|C_{max}$ feladatra $\frac{C_{max}(LPT)}{C_{max}(OPT)} \leq \frac{5}{4}$.

A $Pm||C_{max}$ feladatnál, ahol m az identikus gépek száma, az LPT algoritmus egy egyszerű eljárás-ként működik [8]. Az eljárás menete a következő:

A termékeket végrehajtási idejük alapján csökkenő sorrendbe állítjuk, és készítünk egy ilyen listát. Az első terméket az első berendezésre, a másodikat a másodikra ütemezzük, ez így megy tovább, amíg van szabad berendezés, tehát nem végzünk rajta semmilyen munkát. Amint felszabadul egy gép, a listában lévő következő terméket hozzárendeljük a berendezéshez, amíg a lista ki nem ürül.

Erre a feladatra $\frac{C_{max}(LPT)}{C_{max}(OPT)} \leq \frac{4}{3} - \frac{1}{3m}$, ahol m a berendezések száma.

Az LPT-algoritmus teljesítménye bonyolultabb Flexible Flow Shop típusú feladatoknál nagymértékben romolhat. Például ha kevés berendezés van állomásonként, termék pedig sokkal több van, akkor a makespan lényegesen eltolódhat ezáltal, mert sok terméknek várakoznia kell, amíg a berendezések elvégzik a feladatukat és felszabadulnak. Viszont számos feladatnál még ezen negatív tulajdonsága ellenére is remek közelítő megoldást adhat a problémának. Későbbiekben tehát ezt az

algoritmust is felhasználtuk egy egyszerű verzióban a heurisztikák megalkotásakor, melyről majd később írunk részletesebben a dolgozatban.

Pszeudokódok:

1 Johnson algoritmusa

1. lépés: Legyen $k := 1$ és $l := n$, és tekintsük a nem ütemezett munkák $J = j_1, \dots, j_n$ halmazát.

2. lépés: Legyen p_{ij} az i -edik munka végrehajtási idejét a j -edik gépen. ($j \in [1, 2]$). Keressük meg a $\{p_{i1}, p_{i2} | j_i \in J\}$ halmaz egy minimális elemét. Jelölje a hozzátartozó indexet i .
if i nem egyértelműen meghatározott **then** a lehetséges indexek közül válasszuk a legkisebbet.
end if

3. lépés:
if a 2. lépésben választott elem p_{i1} **then** helyezzük a j_i munkát listánk k -adik helyére, töröljük J halmazból, és növeljük k értékét 1-gyel.
end if
Majd lépünk az 5. lépésre.

4. lépés:
if a 2. lépésben választott elem p_{i2} **then** helyezzük a j_i munkát listánk l -edik helyére, töröljük J halmazból, és csökkentjük l értékét 1-gyel.
end if
Majd lépünk az 5. lépésre.

5. lépés:
if J üres **then** vége az eljárásnak. Az optimális ütemezést kapjuk meg, ha az eljárás során meghatározott lista sorrendje szerint hajtjuk végre a munkákat késlekedés nélkül.
else lépünk a 2. lépésre.
end if

2 SPT-elv

1. lépés: Készítsünk egy listát a munkákról, végrehajtási idő szerint növekvő sorrendben.

2. lépés: Indítsuk el az ütemezést a lista elejétől kezdve minden berendezéshez az állomáson.

3. lépés:

while A lista nem üres **do**

if Felszabadul egy berendezés **then** Rendeljük hozzá a a berendezéshez a listában soron következő munkát.

end if

end while

A fentieket hajtsuk végre minden állomásra.

3 LPT-algoritmus

1. lépés: Tegyük a munkákat p_j szerint csökkenő sorrendben az 1. állomáson.

2. lépés: Legyen $i = 1$ és $j = n$.

2-1. lépés: Rendeljük hozzá j . munkát az i . berendezéshez a 2. állomáson. Legyen $i = i + 1$ és $j = j - 1$.

2-2. lépés:

if $i = m + 1$ **then** $i = 1$.

end if

2-3. lépés:

if $j = 0$ **then** menjünk a 3. lépéshez.

else menjünk a 2-1. lépéshez.

end if

3. lépés: Csúsztassunk balra annyi munkát a 2. állomáson, amennyi csak lehetséges, amíg egy munka végrehajtási ideje az 1. állomáson kisebb vagy egyenlő a munka kezdési idejével a 2. állomáson.

2.3. Gyakorlati előfordulás

Ütemezés során fontos, hogy a gyakorlatban is bevethessük mindazt, amit elméletben kidolgoztunk és teszteltünk. Fontos, hogy a valósághoz közel álló példákra végezzük vizsgálatainkat. Gyakran felmerül a kérdés, hogy a gyakorlatban vajon milyen elterjedt az előfordulása ezeknek a példának, tehát megéri-e foglalkozni vele, vagy sem. Ismertetünk egy konkrét példát bebizonyítva azt, hogy a flexible flow shop ütemezési feladatok igen gyakoriak az ipari üzemekben [8].

Képzeljünk el egy gyárat, ami papírzacskókat gyárt. Különböző lehet a felhasználása, például kutyatáphoz, cementhez, faszénhez, virágföldhöz. Minden kezdő folyamat alapanyagai papírtekercek. Hogy flexible flow shop ütemezésről lehessen beszélni, ezért megragadunk egyetlen példát, ez legyen a cementhez való zacskó gyártása, melyek kinézetre és méretre egyeznek. A gyártás három fázisból áll: a minta nyomtatása, a zacskó oldalainak beragasztása, valamint valamely vég, vagy mindkettő vég bevarrása. Minden állomáson található bizonyos mennyiségű berendezés. Ezek identikusak, és megegyezik a gyártási sebességük, mivel csak egyfajta zacskót készít az üzem. A zacskóknak végig kell haladni sorrendben minden fázison, mivel nem kerülhetnek ki például logó nélküli termékek. Ha az adott fázist befejezték a köztes terméken, azonnal átkerül a következő állomásra, egy szabad berendezésre. Ha végigment minden fázison, akkor a termék készen van.

A fenti példával bemutattuk, hogy az FFc osztály ismert az ipari alkalmazások területén, azonban adódhat még számos feltétel a valóságban, ami bonyolítja a feladaton. Például ha nem csak cementekhez való papírzacskót készít az üzem, akkor különböző mintájú és méretű termékekre van szükség. Ebből az következik, hogy például a festési fázisban a berendezéseknek eltérő lehet a sebessége, mivel különböző mintájú zacskókat készíthetnek egyszerre, valamint a ragasztásnál és varrásnál is eltérő lehet a sebesség a méretből adódó különbségek miatt. Akár az is előfordulhat, hogy egyes berendezések különleges zacskókat gyártanak, ezáltal csak bizonyos termékek készítéséhez alkalmazhatók. Ezekkel a megkötésekkel már kikerülünk a flexible flow shop hatásköréből. A valóságban számos FFc -hez hasonló eset előfordulhat, melyekre megfelelően kiterjeszthetők az általunk vizsgált algoritmusok, heurisztikák.



2.3. ábra. A példában bemutatott gyártósor egy részlete.

3. Algoritmusok kiterjesztése FFc feladatosztályra

3.1. *Kidolgozott heurisztikák*

A 2.2 fejezetben ismertetett algoritmusok megfelelő alapötletként szolgáltak a saját munka megkezdéshez. A témában fellelhető cikkek hiánya miatt sok időt emésztett fel a megfelelő, mi témánkhoz hasznosítható szakirodalom felkutatása.

Elsőként az LPT-algoritmussal [8, 7] ismerkedtünk meg, amely ezen feladatosztály körében az egyik legismertebb és leghatékonyabb algoritmus. Különböző speciális feladatoknál (2.2 fejezet) megfelelő jósági arányt produkál. E jó tulajdonsága miatt gondoltuk úgy, hogy az általunk vizsgált feladatosztályra kiterjesztjük, és megvizsgáljuk milyen mértékben tér el az optimális megoldástól. A munkánk során ez tűnt az egyik legígéretesebb heurisztikának.

Másik eljárás, melyet a szakirodalom [8] ihletett, az SPT-elv. Annak ellenére, hogy nem C_{max} ennek az elvnek az elsődleges célfüggvénye, hanem $\sum C_j$, mégis úgy gondoltuk, hogy érdemes lenne foglalkozni ezzel az algoritmussal. Az oka az, hogy számos esetben megelőzheti az LPT-algoritmust.

Vegyünk egy olyan példát, amikor az első állomáson a munkák többségének nagyon nagy a végrehajtási ideje, viszont akad egy-kettő olyan is, amelynek roppant kicsi, akár a többi állomáson is. Ebben az esetben az LPT sorban ütemezi a legnagyobb végrehajtási idővel kezdve, és a kis végrehajtási idővel rendelkező munkák "hátraszorulnak", így a C_{max} nagyon nagy eltolódhat, mivel a többi állomáson lévő berendezések rengeteg ideig tétlenek maradnak. Ehhez hasonló esetekben az SPT-elv nagymértékben megelőzheti az LPT-algoritmust.

A heurisztikákat, melyeket munkánk során alkottunk, főleg az előbb ismertett 2 eljárásnak a vizsgálata ihlette. Legtöbb esetben, ha az ütemezés során a nagy végrehajtási idővel rendelkező termékeket előre vesszük, úgy az össz-végrehajtási idő lecsökken. Ennek az oka, hogy a végére maradnak a kevesebb idő alatt elvégezhető munkák, melyeket már könnyebb elrendezni a megfelelő eredmény érdekében. Összesen hat saját heurisztikát teszteltünk, melyek közül egy az LPT kiterjesztése, másik három pedig ennek egy-egy módosított verziója. A további két esetben eltérő elvek szerint jártunk el.

Bár multistage feladatokat vizsgáltunk, az általunk készített algoritmus kizárólag az első állomás előtt határozza meg a termékek gyártásának sorrendjét, s arra az esetre, amikor két állomás közt várakozó termékek közül kell választani, nem tér ki. Ilyen esetekben az a termék kerül előre, amely az elején meghatározott sorrend szerint korábban van.

Az LPT algoritmusok mindig a hosszabb végrehajtási idejű termékeket veszik előre, azonban ez többféleképpen is lehetséges. az általunk elsőként letesztelt LPT* veszi egy terméknél az összes állomásra vonatkozó végrehajtási idők összegét, tehát az össz-végrehajtási időt, s ez alapján rendezi őket csökkenő sorrendbe. A későbbiekben azt vizsgáltuk meg, hogy javulnak-e a heurisztikák, ha nem az össz-végrehajtási idő szerint, hanem csak az első, vagy csak az utolsó állomás végrehajtási ideje szerint rendezünk. Az utolsó állomáson, ha a végére maradnak a rövidebb végrehajtási idejű munkák, akkor sűrűbben ütemezhetők, kevesebb a kihasználatlan idő. Ezzel azonban az probléma, hogy a többi állomásról semmit nem mond. Ugyanígy az első állomás végrehajtási idejének vizsgálatakor, amelyek az optimumtól általában távolabbi eredményeket szolgáltatnak.

Az LPT-től eltérő algoritmusaink egyike az SPT* volt, amely az LPT*-hoz hasonlóan össz-végrehajtási idő alapján állít fel sorrendet, azonban itt a rövidebb munkákat sorolja előre. Ilyenkor gyorsabban képes feltölteni a berendezéseket, s ellátni a későbbi állomásokat is. A másik eltérő heurisztikánk az LSPT, amely felváltva ütemez hosszú, majd rövid végrehajtási idejű munkákat. Ezt azért gondoltuk megfelelőnek, mert a felváltott ütemezés által a termékek jobban szétszóródnak. Az első állomásra triviális, hogy nem ütemez optimális megoldást (mert az LPT ad erre optimális megoldást), viszont a többi állomáson kevesebb ideig lesznek tétlenek a berendezések, jobban kitöltik majd az időt a munkák. Következtetésképp ez a heurisztika nagyobb számú állomások,

bonyolultabb feladatok esetében alkalmazható megfelelő eredménnyel.

4 LPT* algoritmus

```
function LPT*(Job* left, Job* right)
  int ProcessTime[left]  $\leftarrow$  0
  for  $i = 0$  to left->Stages do
    ProcessTime[left]  $\leftarrow$  ProcessTime[left] + left->ProcessTime[ $i$ ]
  end for
  int ProcessTime[right]  $\leftarrow$  0
  for  $i = 0$  to right->Stages do
    ProcessTime[right]  $\leftarrow$  ProcessTime[right] + right->ProcessTime[ $i$ ]
  end for
  return ProcessTime[left]  $\geq$  ProcessTime[right]
end function
```

5 SPT* algoritmus

```
function SPT*(Job* left, Job* right)
  int ProcessTime[left]  $\leftarrow$  0
  for  $i = 0$  to left->Stages do
    ProcessTime[left]  $\leftarrow$  ProcessTime[left] + left->ProcessTime[ $i$ ]
  end for
  int ProcessTime[right]  $\leftarrow$  0
  for  $i = 0$  to right->Stages do
    ProcessTime[right]  $\leftarrow$  ProcessTime[right] + right->ProcessTime[ $i$ ]
  end for
  return ProcessTime[left]  $\leq$  ProcessTime[right]
end function
```

6 LPT-A algoritmus

function LPT-A(Job* left, Job* right)

int ProcessTime[*left*] $\leftarrow$ left->ProcessTime[*left* - *Stages* - 1] - left->ProcessTime[0]

int ProcessTime[*right*] $\leftarrow$ right->ProcessTime[*right* - *Stages* - 1] - right->ProcessTime[0]

return ProcessTime[*left*] $\geq$ ProcessTime[*right*]

end function

7 LPT-FS algoritmus

function LPT-FS(Job* left, Job* right)

int ProcessTime[*left*] $\leftarrow$ left->ProcessTime[0]

int ProcessTime[*right*] $\leftarrow$ right->ProcessTime[0]

return ProcessTime[*left*] $\geq$ ProcessTime[*right*]

end function

8 LPT-LS algoritmus

function LPT-LS(Job* left, Job* right)

int ProcessTime[*left*] $\leftarrow$ left->ProcessTime[*left* - *Stages* - 1]

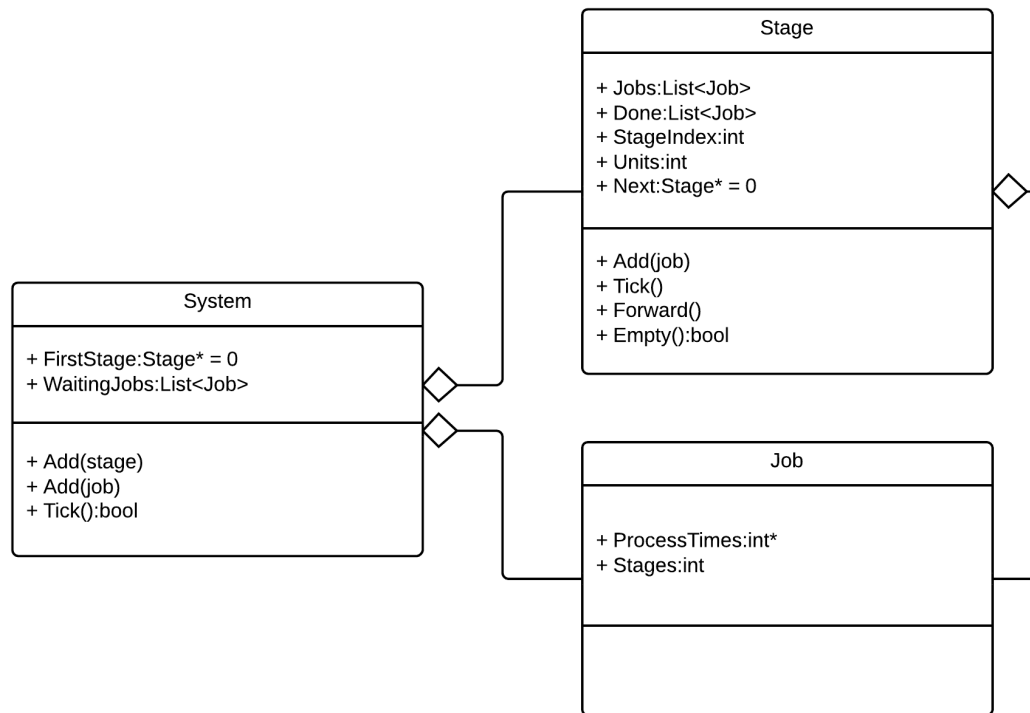
int ProcessTime[*right*] $\leftarrow$ right->ProcessTime[*right* - *Stages* - 1]

return ProcessTime[*left*] $\geq$ ProcessTime[*right*]

end function

3.2. Implementáció

Az implementáció során egy komplett rendszert szimuláltunk, amely állomásokból és azokon található identikus berendezésekből áll. A legfelső szinten maga a rendszer áll, amelynek ismernie kell a rendszerben lévő állomásokat és a végrehajtásra váró munkákat. A környezet - avagy a feladat - betöltésekor a létrehozott rendszerbe fel kell vinni a feladatban leírt állomásokat a megfelelő berendezéssel ellátva, továbbá a legyártandó termékeket, feltüntetve azok egyes állomásokra vonatkozó végrehajtási idejét.



3.1. ábra. Az implementált program UML-diagramja

A környezet felépítése után minden rendelkezésre áll a szimuláció elkezdéséhez. Ekkor, még a gyártás megkezdése előtt hajtódik végre az általunk tesztelt heurisztikus eljárás, amely a megfelelő algoritmus alapján a legyártandó termékeket sorba rendezi.

A szimuláció azt reprezentálja, hogy egy-egy óra elteltével hogyan változik a berendezések kihasználtsága, hol fejeződik be egy munka és kerül át a következő állomásra, vagy hol kezdődik el egy új. Egy óra elteltét a rendszer *Tick()* függvénye szimulálja. Ennek visszatérési értéke mindaddig *True*, ameddig van végrehajtásra váró, vagy végrehajtás alatt lévő munka. A rendszer ilyenkor végignézi az összes állomást, és azok berendezéseiben található munkák gyártását egy órával állítja előrébb. Ezután az összes állomáson megvizsgálja, hogy ezen egy óra elteltével van-e olyan

berendezés, amely végzett, és ezen berendezésekben található termékeket továbbítja a következő állomás felé. Ekkor a felszabadult helyet megpróbálja feltölteni a végrehajtásra váró munkák listájából. Ha a végrehajtásra váró és végrehajtás alatt lévő munkák száma minden állomáson nulla, akkor a gyártás véget ért. Hogy ehhez hány óra szükséges, azt a *tick()* függvény hívása mellé bevezetett számláló rögzíti.

4. Eredmények

A tesztelés során három fő szempontot tartottunk fontosnak: milyen a vizsgált heurisztikák jósága, hogyan teljesítenek egymáshoz képest, továbbá a heurisztikák felhasználásával milyen mértékben csökkenthető a megoldó futásideje. Tesztjeink során három különböző esetekre vizsgáltuk meg az algoritmusokat.

Tesztesetek	Állomások	Berendezések állomásonként	Termékek	Gyártási idők
Első tesztet	1-5	1-4	2-6	2-7
Második tesztet	5-9	4-9	7-20	4-9
Harmadik tesztet	5-9	4-5	10-20	1-9

4.1. táblázat. Tesztesetek

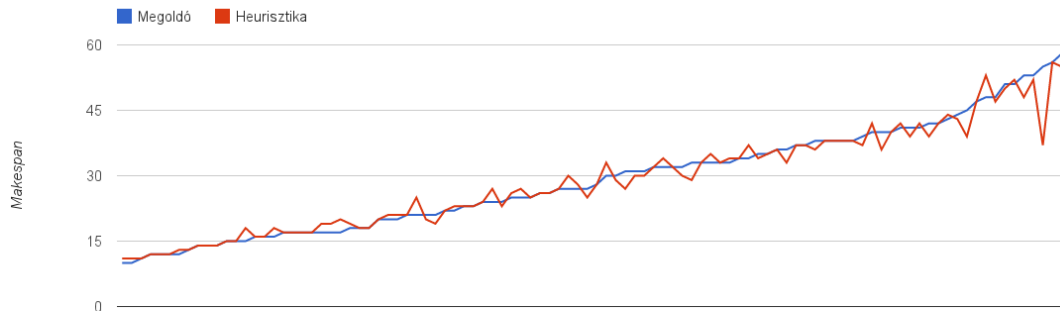
A tesztek generálásakor az egyes tesztesetek paramétereit használtuk a 4.1 táblázat alapján. Ezek az értékek intervallumokat jelölnek, melyből egy adott teszt generálásakor kiválasztottunk véletlenszerűen egyet. Minden alkalommal legalább 100 tesztet generáltunk és vizsgáltunk meg.

4.1. Heurisztikák jósága

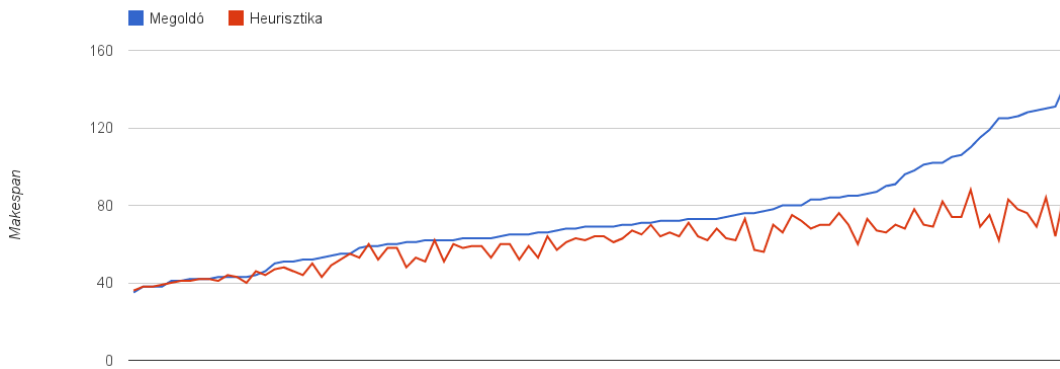
A heurisztikák jósága több szempontból is fontos kérdés. Amennyiben az algoritmus egy, az optimálishoz közelebbi eredményt tud szolgáltatni a megoldónak, az nagyobb mértékben képes csökkenteni a keresési teret csak úgy, mint a számoláshoz szükséges időt. A másik fontos kérdés, hogy tudjuk-e garantálni, hogy a heurisztikánk által megtalált megoldás az optimum ϵ sugarú környezetében van. Tesztelés során a megoldót 1000 másodperces időkorláttal futtattuk le, mert nagyobb problémák esetén az optimális megoldás megkeresése jóval hosszabb időt vett volna igénybe. Emiatt az alábbi grafikonokban a megoldóhoz tartozó értékek az 1000 másodperc alatt megtalált legjobb megoldást hasonlítjuk össze a hat heurisztika minimumával.

A 4.1 grafikonból arra következtettünk, hogy az algoritmusok nagyobb problémák esetén jobb eredményt találnak - továbbra is kisebb, mint 1 másodperces futásidővel -, mint az s-gráf megoldó szoftver. Hogy erről megbizonyosodjunk, lefuttattuk a teszteket nagyobb problémákra is.

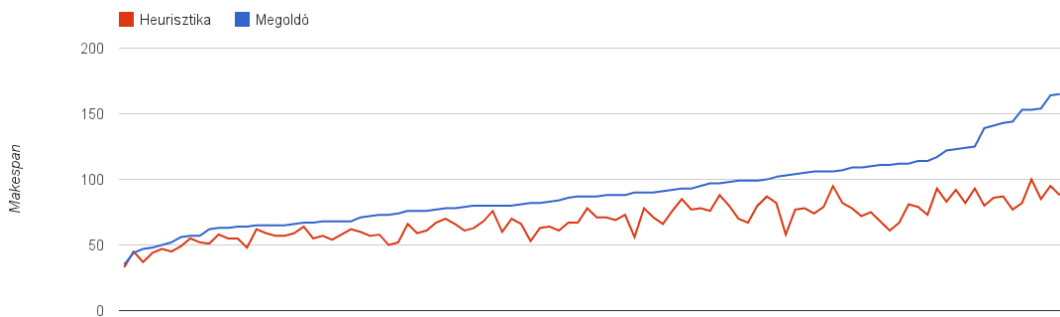
A 4.2 grafikonon szemmel látható, hogy az elvárásoknak megfelelően a heurisztikák az optimumhoz közelebbi megoldást találtak, mint a megoldó.



4.1. ábra. Heurisztikák minimuma a megoldóhoz képest: első tesztet



4.2. ábra. Heurisztikák minimuma a megoldóhoz képest: második tesztet



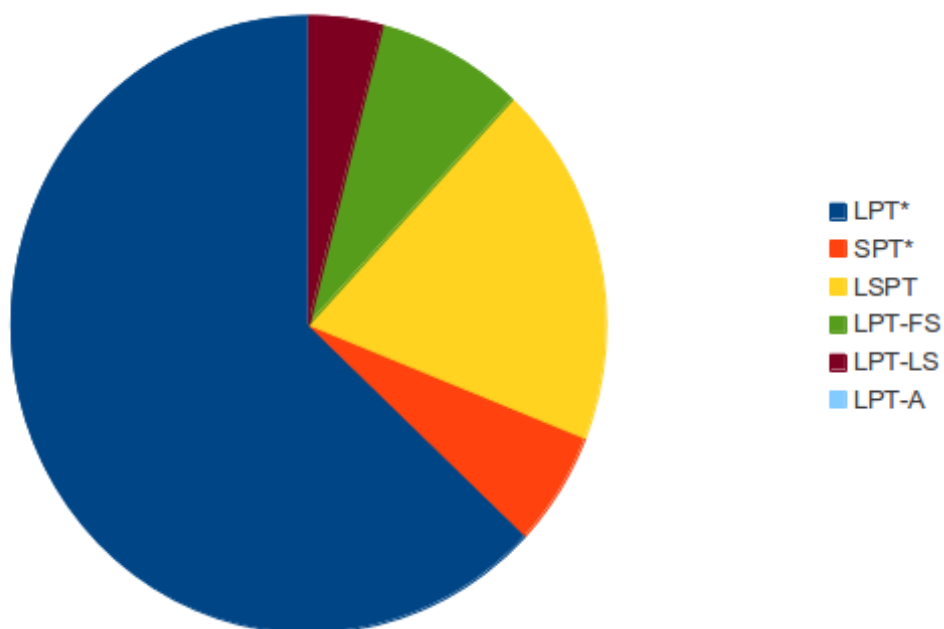
4.3. ábra. Heurisztikák minimuma a megoldóhoz képest: harmadik tesztet

4.2. Heurisztikák összehasonlítása

A tesztelés második fázisában arra voltunk kíváncsiak, hogy van-e olyan heurisztika a megfigyelték között, amely sosem ad jobb megoldást, mint a többi algoritmus minimuma. Ezeket a legkülönbözőbb tesztesetekre vizsgáltuk meg, hátha valamely heurisztika jobb eredményt nyújt speciálisabb kontextusban. A több, mint 20 000 teszt között voltak több - vagy épp kevesebb - állomást igénylő feladatok, valamint olyanok, amelyeknél a berendezések átlagos száma a termékek száma alatt, illetve fölött van.

4.2. táblázat. Heurisztikák

Heurisztika	Jobb megoldást talál
LPT*	63%
SPT*	6%
LSPT	19%
LPT-FS	8%
LPT-LS	4%
LPT-A	0%



4.4. ábra. Heurisztikák teljesítménye

A 4.2 táblázatban látható, hogy az LPT-A heurisztika éppen ilyen, így vizsgálata elhagyható. Itt megjegyzendő, hogy a későbbi tesztheink során nem tekintettünk el ezen algoritmus vizsgálatától az elhanyagolható futásidő miatt.

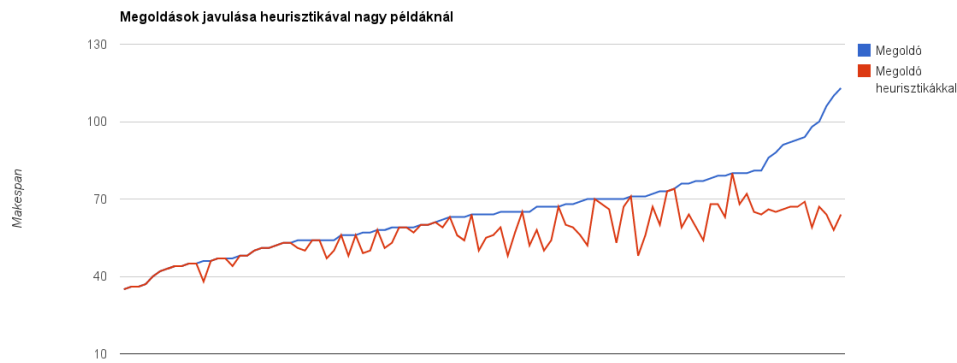
4.3. Megoldó teljesítménye heurisztikákkal

Munkánk célja az volt, hogy az algoritmusainkkal generált gyors kezdeti megoldást felhasználva leszűkítsük a keresési teret, hogy a megoldó szoftver rövidebb idő alatt legyen képes az optimum megtalálására. Úgy véltük, így akár nagyobb problémák is megoldhatóvá válnak, melyeket rengeteg teszt elvégzésével alá is támasztottunk. Amennyiben a probléma mérete kellően nagy, és a megoldó nem képes megvalósítható megoldást találni, úgy felmutathatunk egy elfogadható - bár nem optimális -, közelítő megoldást. Három esetet különböztettünk meg a tesztek kiértékelése után.

Elsőként említjük meg, hogy voltak olyan esetek, mikor a megoldó futási ideje csökkenés helyett nőtt. Ennek oka az, hogy az egyszerű, kisebb méretű példáknál az optimális megoldást a szoftver heurisztikák nélkül is hamar megtalálja, így ezen algoritmusok lefutása "felesleges", csak növelik a futási időt. Körülbelül az esetek 3%-ban történt futási idő növekedése.

Második eset, mikor bekövetkezett a várt gyorsulás a futási időben. 100 esetből nézve az átlagos javulás körülbelül 6,78% volt. Ezek a közepes méretű, vagy nagyobb példák már viszonylag nagyok voltak a megoldó számára ahhoz, hogy hasznosíthassa a heurisztikák segítségét, és megtalálja az optimumot. Nagyjából az esetek 23%-ban történt futási idő csökkenés.

Harmadik esetben nem történt semmilyen változás a futási időben. Az esetek 74%-ban túl nagy példákat generáltunk, amiket heurisztikák segítségével sem volt képes belátható időn belül (1000 másodperces időkorlát) megoldani a program. Külön elvégzett tesztekkel azt is megnéztük, hogy nagy példákban, ahol a megoldó heurisztikák nélkül és heurisztikákkal sem volt képes belátható időn belül optimális megoldást találni, ott mennyire vagyunk képesek jobb megoldást találni, mint a megoldó. A 4.5 ábrán jól látható, hogy az esetek többségében sikerült jobb eredménnyel szolgálni heurisztikákkal, mint nélkülük, pontosan 59%-ban.



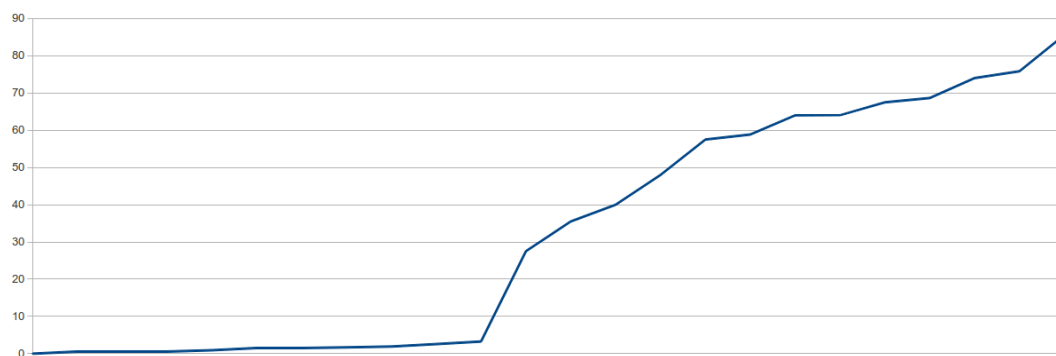
4.5. ábra. Megoldás javulása nagy példánál

A megoldás átlagos javulás majdnem 11 % volt a 100 esetből elvégzett tesztnél. A 4.6 ábrán látható, hogy hogyan oszlott el a javulás, és százalékban kifejezve mekkora volt a mértéke.



4.6. ábra. Megoldás javulása százalékban.

Érdemes itt megjegyezni, hogy annak ellenére, hogy az esetek többségében nem sikerült a várt gyorsítást elérni, mégis pozitív eredményeket értünk el azáltal, hogy nagyon rövid idő alatt tudtunk szolgáltatni eredményt, melyet a megoldó a hosszú időkorlát alatt sem bírt kiszámolni. A 4.7 ábrán látható a megoldó futási idejének csökkenése %-ban kifejezve. Ez a diagram csak azokra az esetekre vonatkozik, ahol gyorsulást értünk el. Látható, hogy az esetek felénél a gyorsulás elhanyagolhatóan kicsi, viszont voltak olyan példák, ahol akár 85%-os futási idő csökkenést is elértünk.



4.7. ábra. Megoldó sebességének növekedése a saját heurisztikákkal %-ban.

5. Összefoglalás

Vizsgálataink céljaul heurisztikus eljárások kidolgozását tűztük ki. Ezek az eljárások azért előnyösek, mert egy megoldó szoftver futási idejét nagymértékben csökkenthetik különböző okokból. Ilyen lehet például a keresési tér csökkentése valamilyen szempont alapján, ezáltal a megoldónak jóval kevesebb esetet kell megvizsgálni, ebből adódóan az optimális megoldás megtalálása gyorsabb. Célkitűzésünk kettéágazott: a megoldó futási idejének csökkentése az általunk alkotott eljárások segítségével a vizsgált feladatosztályra, másrészt nagy feladatokra, melyeket a megoldó szoftver nem képes belátható idő alatt megoldani, elfogadható megoldást találni a heurisztikák segítségével.

Munkánkat a szakirodalom áttekintésével kezdtük. A publikációk nagy része csupán egy-egy speciális esetet vizsgál meg, és az ezekre kidolgozott algoritmusokat ismerteti. A feladat komplexitásából következik, hogy a feladatosztályra olyan általános algoritmus nem létezik, amely optimális megoldást adna. Emiatt van szükség a heurisztikák és a megoldó együttműködésére a hatékony keresés, és gyors, optimális megoldás érdekében. Szakirodalmi áttekintést követve már megfelelő alapokkal rendelkezünk, hogy algoritmusokon gondolkodjunk. Az alapötleteket a cikkekből megismert algoritmusok nyújtották, melyekből kiindulva hat eljárást hoztunk létre, melyeket teszteltünk.

A 4. fejezetben ismertettük az elért eredményeket, több szempont alapján. A kitűzött célok közül a megoldó futásidejének javítása nem teljesen érte el a várt mértéket a túl nagy tesztesetek miatt. Azonban ahol sikerült gyorsulást elérni, ott szép arányban sikerült azt megvalósítani. Másik célunk az volt, hogy egy viszonylag pontos eredményt számoljunk ki algoritmusaink segítségével nagy példákra, amiket a megoldó képtelen kiszámítani belátható időn belül. Mivel a heurisztikák pillanatok alatt lefutnak, ezért mindig találnak megoldást, csak nyitott kérdés marad, hogy ez mennyire pontos.

Későbbi vizsgálataink során megmaradunk ennél a feladatosztálynál, de további fejlesztéseket eszközölünk algoritmusainkban. Tervezzük további, élesebb korlátot szolgáltató eljárások kutatását, alapul véve eddigi munkánkból származó tapasztalatainkat. Továbbá eddig úgy alkalmaztuk a heurisztikákat, hogy azok előre meghatározzák a munkák sorrendjét az egész ütemezésre nézve, ám ez tovább finomítható azáltal, hogy az ütemezést állomások szintjén hajtjuk végre, ezáltal pontosabb megoldást találhatnánk a megoldó számára. Az algoritmusok futási ideje továbbra sem nőne meg jelentősen, viszont elképzelhető, hogy a megoldóval együtt a végső idő nagymértékben csökkenne az élesebb korlát miatt.

Irodalomjegyzék

- [1] Fouad Riane & Abdelhakim Artiba & Salah E. Elmaghraby. A hybrid three-stage flowshop problem: Efficient heuristics to minimize makespan. *European Journal of Operational Research*, 109, 1998.
- [2] Carlos A. Méndez & Jaime Cerdá & Ignacio E. Grossmann & Iiro Harjunoński & Marco Fahl. State-of-the-art review of optimization methods for short-term scheduling of batch processes. *Computers and Chemical Engineering*, 30, 2006.
- [3] E. Sanmartí & L. Puigjaner & T. Holczinger & F. Friedler. Combinatorial framework for effective scheduling of multipurpose batch plants. *American Institute of Chemical Engineers*, 48, 2002.
- [4] Máté Hegyháti & Ferenc Friedler. Overview of industrial batch process scheduling. *Chemical Engineering Transactions*, 21, 2010.
- [5] Máté Hegyháti & Ferenc Friedler. Combinatorial algorithms of the s-graph framework for batch scheduling. *Industrial & Engineering Chemistry Research*, 50, 2011.
- [6] Johnson. Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly*, 1, 1954.
- [7] Byung-Cheon Choi & Kangbok Lee. Two-stage proportionate flexible flow shop to minimize the makespan. *Journal of Combinatorial Optimization*, 30, 2011.
- [8] Michael L. Pinedo. *Scheduling Theory, Algorithms, and Systems*. Springer, 2008.
- [9] Caixia Jing & Guochun Tang & Xingsan Qian. Heuristic algorithms for two machine re-entrant flow shop. *Theoretical Computer Science*, 400, 2008.
- [10] Adar A. Kalir & Subhash C. Sarin. Constructing near optimal schedules for the flow-shop lot streaming problem with subplot-attached setups. *Journal of Combinatorial Optimization*, 7, 2003.
- [11] C. Sriskandarajah & S. P. Sethi. Scheduling algorithms for flexible flowshops: Worst and average case performance. *European Journal of Operational Research*, 43, 1989.
- [12] Paul Martin & David B. Shmoys. A new approach to computing optimal schedules for the job-shop scheduling problem. In *In Proc. of the 5th International IPCO Conference*, pages 389–403. Springer, 1996.

- [13] Holczinger Tibor. *Módszer köztes tárolókat nem tartalmazó szakaszos működésű rendszerek ütemezésére*. PhD thesis, Veszprémi Egyetem Informatikai Tudományok Doktori Iskola, 2004.
- [14] Petra Schuurman & Gerhard J. Woeginger. A polynomial time approximation scheme for the two-stage multiprocessor ow shop problem. *Theoretical Computer Science*, 237, 2000.
- [15] Richard Linn & Wei Zhang. Hybrid flow shop scheduling: A survey. *Computers & Industrial Engineering*, 37, 1999.
- [16] Bertók Botond & Kovács Zoltán. *Gyártórendszerek modellezése*. 2011.
- [17] Danku Réka & Éles András & Flumbort Csaba & Kovács Balázs & Leitold Dániel & Nagy Norbert & Pigler Csaba & Orosz Ákos & Róth Gergő & Szili László & Tóth Benjámín. Optimalizációs módszerek. JÁSZN-2011.
- [18] Éles András & Kovács Balázs & Tóth Benjámín. Szakaszos üzemű termelő rendszerek ütemezése: feladatosztályok és megoldó módszerek. TDK-2011.


```

1  #include "System.h"
2  #include <fstream>
3  #include <sys/time.h>
4
5  class Job{
6  public:
7      int* ProcessTime;
8      int Stages;
9
10     Job(int stage , int* pt ){
11         ProcessTime = pt;
12         Stages = stage;
13     }
14 };
15
16 class Stage{
17 public:
18     List<Job> Jobs;
19     List<Job> Done;
20     int StageIndex;
21     int Units;
22     Stage* Next;
23
24     Stage(int idx , int units){
25         StageIndex = idx;
26         Units = units;
27         Next = 0;
28     }
29
30     void Tick(){
31         IEnumerator<Job>* i = Jobs.GetEnumerator();
32         while( i->Next() ){
33             if( i->Current->ProcessTime[ this->StageIndex ] > 0 )
34                 i->Current->ProcessTime[ this->StageIndex ]--;
35             if( i->Current->ProcessTime[ this->StageIndex ] <= 0 )
36                 Done.Add( i->Current );
37         }
38
39         i = Done.GetEnumerator();
40         while( i->Next() ){
41             if( Jobs.IndexOf( i->Current ) != -1 ){
42                 Jobs.Remove( i->Current );

```

```

43     }
44 }
45
46
47 }
48
49 bool Add( Job* j ){
50     if( Jobs.GetLength() < Units){
51         Jobs.Add(j);
52         return true;
53     }
54
55     return false;
56 }
57
58 void Forward(){
59     if( Next == 0 ) return;
60     Done.Sort(lpt);
61     while( Done.GetLength() > 0 && Next->Add( Done.GetFirst() ) )
62         Done.Remove( Done.GetFirst() );
63 }
64
65 bool Empty(){
66     return (Jobs.GetLength() == 0);
67 }
68 };
69
70 class System {
71 public:
72     Stage* First;
73     List<Job> Waiting;
74
75     System() {
76         First = 0;
77     }
78
79     void Add( Stage* s){
80         if( First == 0 ){
81             First = s;
82             return;
83         }
84

```

```

85     Stage*tmp;
86     for(tmp = First; tmp->Next != 0; tmp = tmp->Next);
87     tmp -> Next = s;
88 }
89
90 void Add( Job* j){
91     Waiting.Add( j );
92 }
93
94 bool Tick(){
95     Stage*tmp;
96     for(tmp = First; tmp != 0; tmp = tmp->Next)
97         tmp->Tick();
98
99     while(Waiting.GetLength() != 0 && First->Add( Waiting.GetFirst() ) )
100         Waiting.Remove( Waiting.GetFirst() );
101
102     bool check = true;
103     for(tmp = First; tmp != 0; tmp = tmp->Next){
104         tmp->Forward();
105         check = ( check && tmp->Empty() );
106     }
107
108     return !check;
109 }
110 };

```