

TDK-DOLGOZAT

Orosz Ákos



S-GRÁF ALAPÚ THROUGHPUT-MAXIMALIZÁLÁSI ALGORITMUSOK VIZSGÁLATA ÉS TOVÁBBFEJLESZTÉSE

Orosz Ákos
Mérnökinformatikus
BSc, 7. félév

Pannon Egyetem
Műszaki Informatikai Kar

Témavezető:
Dr. Holczinger Tibor
Egyetemi docens, PE MIK

Absztrakt

Az informatikát nagyon sok kutatási területen használják, ahol nagy mértékű számítási kapacitásra van szükség. Ezek közül egy sokak által kutatott témakör a gyártási folyamatok ütemezése, ahol különböző termékeket kell előállítani a rendelkezésre álló erőforrások (gyártó berendezések) segítségével. A rendszer működése során fontos, hogy melyik berendezést mikor mire használjuk, ennek az időzítésnek a meghatározását nevezzük ütemezésnek.

Szakaszos folyamatok optimalizálása során a berendezések működési idején kívül gyakran figyelembe kell még venni további időzítési paramétereket, mint például a berendezések esetleges tisztítását, az anyagok átviteli idejét a berendezések között, a berendezések átállási idejét két feladat között. Továbbá a feladatot ki lehet terjeszteni más nem időzítési jellegű feltételekkel, mint például a hőcserélési paraméterek, energiafogyasztás, szennyezés kibocsájtás.

A feladat, hogy ezeket a rendszereket valamilyen szempont szerint optimális módon ütemezzük, amely szempont leggyakrabban adott számú termék minél rövidebb idő alatti legyártását jelenti. Jelen munka esetében a célfüggvény, hogy adott idő alatt minél nagyobb összértékű termékeket gyártsunk.

Dolgozatom első felében röviden ismertetek az S-gráf módszertanon alapuló, korábban kifejlesztett algoritmusokat, melyeket implementáltam egy meglévő, robosztus keretrendszerbe. A dolgozat második részében bemutatok egy új eljárást, a globális keresőfa módszert, mely a korábbi algoritmusoktól eltérően generálja az optimális megoldást, a redundáns számítások minimalizálása érdekében. Az algoritmusok hatékonyságát több példán keresztül hasonlítom össze.

Tartalomjegyzék

1. Bevezetés	3
2. Szakirodalmi áttekintés	4
3. A feladat felvázolása	5
4. S-gráf leírás	7
4.1. Matematikai leírás	7
4.2. Recept-gráf	7
4.3. Ütemezési-gráf	8
5. Megvalósíthatósági teszten alapuló algoritmusok	11
5.1. Sorrendi bejárás	14
5.2. Mélységi és szélességi bejárás	15
5.3. Mohó bejárás	17
6. Egy új megközelítés: a globális keresőfa	18
6.1. Az algoritmus leírása	18
6.2. Lehetőségek az algoritmus gyorsítására	20
7. Összehasonlító tesztek	24
8. Összefoglalás	33

1. fejezet

Bevezetés

A kutatók már több, mint 30 éve foglalkoznak a gyártási folyamatok ütemezésének matematikai megfogalmazásával, és azzal, hogy lehet minél gyorsabban minél jobb megoldást találni ezekre a feladatokra. A leggyakoribb feladat, hogy adva van, mit szeretnénk gyártani, az egyes termékek pontos gyártási folyamata a szükséges időikkel együtt, és a kérdés, hogy mi az a legrövidebb idő, amely alatt minden terméket hiánytalanul le tudunk gyártani. Ez a gyártási idő (angolul a makespan) minimalizálása. A kezdetekben még MINLP (mixed-integer non-linear programming - vegyes-egész nemlineáris programozási) modelleket is kipróbáltak a megoldáshoz, de a feladatok nagy része megoldható MILP (mixed-integer linear programming - vegyes-egész lineáris programozási) modellekkel is.

Az ezredforduló környékén születettek meg az S-gráf alapjai Sanmartí és társai fejében. Az S-gráf megjelenése jelentős előrelépés volt a témában, ugyanis a kombinatorikus korlátozás és szétválasztás algoritmus és a feladatból felépített gráf tulajdonságainak kihasználása segítségével a feladatok megoldási ideje jelentős mértékben csökkent a korábbi, lineáris programozási modellekhez képest. Azóta mind az S-gráf módszertan, mind az MILP modellek folyamatosan fejlődnek. Továbbá olyan hibákra is felhívta a figyelmet az S-gráf, melyekre korábban senki sem gondolt.

Az S-gráf módszertan hátránya volt, hogy kevés típusú feladatot lehetett vele ábrázolni. A megoldható feladatok körének kibővítésére folyó kutatások eredményeképp napjainkban már az S-gráf is széles körben használható. Az új kiterjesztések közé tartozik a throughput maximalizálás, a dolgozat témája is.

Ez a dolgozat a feladat megoldására adott stratégiák leírásával és összehasonlításával foglalkozik több teszt példán keresztül, továbbá ismertet egy új algoritmust, amely az előzőekhez képest más módon közelíti meg a feladatot, és összeveti a korábbi heurisztikákkal. Az összehasonlított algoritmusokat egy olyan meglévő programba implementáltam, ami S-gráf segítségével hatékonyan tud megoldani makespan minimalizálási feladatokat. Ez a hatékonyság jelentkezik a későbbiekben tárgyalt megvalósíthatósági tesztek esetében is. Ezt a programot egészítettem ki úgy, hogy képes legyen megoldani throughput maximalizálási feladatokat is.

2. fejezet

Szakirodalmi áttekintés

Az ütemezési feladatot először Sparrow és társai írták le 1975-ben megjelent cikkükben. Akkor még MINLP módszerrel oldották meg a feladatot, mely tulajdonképpen a termelő rendszerek berendezéseinek működésének optimális ütemezése volt. Azóta ezzel a feladatkörrel több kutatócsoport is foglalkozik világszerte, melynek eredményeképpen újabb, gyorsabb, hatékonyabb módszerek keletkeznek. Továbbá egy ütemezési feladatot sokszínűen lehet kiegészíteni extra paraméterekkel, illetve más célfüggvényt kitűzve optimalizálás céljára. Ennek megfelelően az évek során több megközelítés is kialakult, mellyel ütemezési feladatok egy-egy típusát meg lehet oldani. Jellemző az MILP modellek használata, mivel a lineáris programozás segítségével széles feladatkörre lehet megoldásokat adni, továbbá egyre több és egyre hatékonyabb MILP-megoldó szoftver létezik.

A lineáris programozástól eltérő megközelítések jellemzően másképp reprezentálják a feladatot leíró receptet (3. fejezet), és az ezen reprezentációból adódó előnyöket használják ki a feladat megoldása során. Ez azt eredményezi, hogy bizonyos reprezentációk hatékonyan tudnak megoldani adott típusú feladatokat, azonban más feladattípusok megoldására esetleg kevésbé alkalmasak. A recept ábrázolásai közé tartozik a State Task Network (STN), melyet Kondili és társai [1] írtak le először. Ezt felhasználva született meg Pantelides munkája során a Resource Task Network (RTN) [2]. Majoji és Zhu [3] publikálták a kisebb feladatok hatékony matematikai megoldására szolgáló State Sequence Network-ot (SSN). Ezen reprezentációkról több összefoglaló és összehasonlító cikk is készült [4].

Ezen ábrázolások mind a gráfokra épülnek, és előszeretettel használják a speciális job-shop (rendszer-, vagy munka-ütemezési) feladatok megoldására meglehetősen hatásosnak bizonyult gráfalgoritmusokat. Az utóbbi években a kutatók heurisztikus korlátozások által adott vágósíkokkal próbálták a keresési tereket csökkenteni, továbbá speciális korlátozás és szétválasztás algoritmusokat fejlesztettek ki. 1998 és 2002 környékén Sanmartí és társai [5, 6] bemutatták az S-gráfot, mely szintén a gráfelméletet használja alapul, és elég hatékony korlátozás és szétválasztás algoritmust tesz lehetővé. Később Romero és társai [7], illetve Holczinger és társai [8] bővítették, illetve gyorsították az algoritmust. 2006-ban Majoji és Friedler [9] publikálták a S-gráf segítségével történő throughput maximalizálás alapjait, amit Holczinger és társai [10] is kiegészítettek.

3. fejezet

A feladat felvázolása

A throughput maximalizálási feladat egy fajtája az ütemezési feladatoknak, így az feladat leírásában szerepel a makespan minimalizálásnál is használt recept, mely definiálja, hogy hány fajta terméket képes a gyár termelni, ezek legyártásához milyen folyamatoknak kell lezajlaniuk, és ezeket a folyamatokat mely berendezésekben mennyi idő alatt lehet végrehajtani. A makespan minimalizálási receptekkel ellentétben azonban throughput maximalizálásnál nincs előre kikötve, hogy melyik termékből mennyit kell legyártanunk, mivel a cél pont ennek a megadása. Van azonban egy másik szükséges információ, mégpedig az egyes termékek értéke valamilyen szempont szerint (tömeg, ár), amit a megoldás során maximalizálni szeretnénk. A dolgozat folyamán bevételről lesz szó, de bármilyen maximalizálandó paramétert be lehet helyettesíteni.

A 3.1. táblázatban látható egy példa a feladat leírásához használt receptre. A táblázat első sora tartalmazza, hogy milyen termékeket lehet gyártani. A táblázatban található recept szerint 3 terméket tudunk gyártani: *A*-t, *B*-t, és *C*-t. Mindhárom termék gyártása három fázisból áll. A fázisok sorszámai a táblázat első oszlopában találhatók meg. Az *A* termék első fázisát (*A1*-es taszk) két típusú berendezés tudja végrehajtani: az *E1* típusú 10 óra alatt, valamint az *E2* típusú 5 óra alatt. Ezeket az állításokat hasonlóan fel lehet írni az *A* termék másik kettő fázisára, és a másik két termék fázisaira is. A receptben szereplő *E1*, *E2*, stb. nevek nem a konkrét berendezéseket, csak azok típusait jelölik. Előfordulhat, hogy valamelyik típusból több külön berendezés is rendelkezésünkre áll a gyártás folyamán. Ahogyan azt a 3.1. táblázat alsó része is jelzi, az *E3*, *E5*, *E6*, *E7* típusú berendezésekből 2-2 darab használható, így a feladat megoldásánál ezt is figyelembe kell venni. Természetesen egy taszkot csak egy berendezés gyárthat, kombinálni nem lehet. Ha az *A* termékből kettőt gyártunk, akkor mindkettő legyártása három fázisból áll, ami összesen 6 taszkot jelent. Így elképzelhető, hogy az egyiknek az első fázisát az *E1* berendezés gyártja, a másik első fázisát pedig az *E2*, mivel ez a kettő két külön taszk. Továbbá, mivel *E3*-as típusú berendezésből kettő van, ezért elképzelhető, hogy két *C* terméket kezdünk el gyártani egyszerre, mivel az első fázishoz két külön berendezés áll rendelkezésre. A feladathoz szükséges jósági adat (jelen esetben az eladási ár) minden termék mellé zárójelben van írva, pl. *A*(2) esetében az *A* termékből származó bevételünk 2.

Ezekon kívül már csak egy adat hiányzik, hogy elkezdhessük megoldani a feladatot, ez pedig az időkorlát

(pl. 100 óra). Ennél a típusú ütemezési feladatnál ugyanis a bevételt szetnénk maxmilizálni egy adott idő alatt. A gyártást az idő elkezdésekor elindítjuk, és amit az intervallum vége előtt legyártunk, azt tudjuk eladni. Az időkorlát nem tartozik szorosan a recepthez, továbbá mivel a dolgozatban közölt tesztek több különböző időkorlátra is lefuttatom, így a receptben ezt a paramétert nem is fogom feltüntetni. Az eredményeket összehasonlító táblázatokban azonban szerepelni fog, hogy az adott sor eredményei milyen időkorlát mellett jöttek ki.

		A(2)	B(3)	C(1)
1		E1(10)	E1(12)	E3(12)
		E2(5)	E2(10)	
2		E4(12)	E5(8)	E2(10)
3		E6(12)	E6(12)	E6(12)
		E8(12)	E7(12)	E7(12)
			E8(12)	

Berendezés	E1	E2	E3	E4	E5	E6	E7	E8
Darabszám	1	1	2	1	2	2	2	1

3.1. táblázat. Példa ütemezési feladat receptjére. Az első táblázat tartalmazza a termékeket és az eladási árakat ($A(2)$), illetve az egyes fázisok legyártásának lehetőségeit (berendezés és idő, $E2(5)$). A második táblázat a berendezéseket sorolja fel.

4. fejezet

S-gráf leírás

Az ütemezési feladatoknak sokféle ábrázolási, illetve megoldási módja van. Ezek közül az egyik az S-gráf módszer, melynek keletkezését és kibővítését a szakirodalmi áttekintésben már tárgyaltam. A gráf-elméletet gyakran hívják segítségül komplex kombinatorikai feladatok megoldásához, mert hatékony algoritmusokat tartalmaz, továbbá a gráfokat le lehet rajzolni, így jobban átlátható, hogy mi a feladat, és mi történik a megoldás során.

4.1. Matematikai leírás

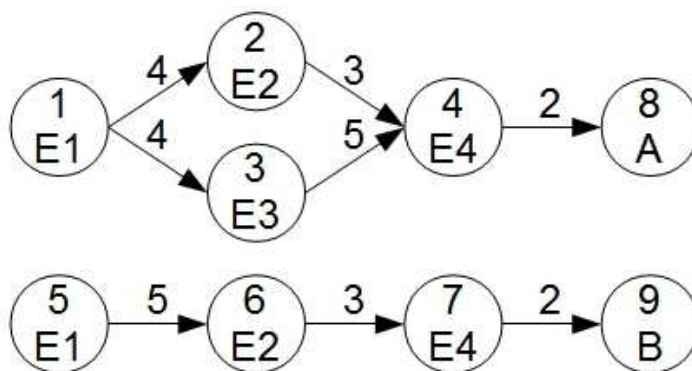
Az S-gráfban kétféle éllel dolgozunk, ezek az úgynevezett recept- és ütemezési-élek. Ezért az S-gráf matematikailag egy rendezett hármassal definiálható, $G(N, A_1, A_2)$, melyben az első elem jelöli a gráf csúcsait, a második a recept-, a harmadik az ütemezési-éleket. A csúcsoknak is két típusa van, taszk és termék típusú csúcsok, melyeket a leírásban nem különböztetünk meg, de később kitérek a köztük lévő különbségre. Az i és j csúcsok közötti él súlyát jelölje $c(i, j)$. Mivel a gráfban a csúcsok tevékenységeket, illetve gyártási fázisokat jelentenek, ha az i csúcs össze van kötve a j csúccsal egy $c(i, j)$ súlyú éllel, az azt jelenti, hogy a i csúcs által jelölt tevékenység elkezdése után legalább $c(i, j)$ időt kell várni, hogy elkezdhessük a j csúcs által jelölt tevékenységet. A legtöbb egyszerű feladat esetében ez az érték az i csúcs által jelölt tevékenység végrehajtási ideje, de bonyolultabb feladatoknál beleszámíthat az anyagok berendezések közti mozgatásának ideje, a tisztítási idő, és még sok más tényező.

4.2. Recept-gráf

A recept definiálja a feladatban szereplő termékek gyártásához szükséges taszkokat, azok végrehajtási sorrendjét, és minden taszkhoz az azt elvégezni képes berendezések halmazát a hozzájuk tartozó idővel együtt. Ezt ábrázolja az S-gráf a recept-gráfon, melyen a csúcsoknak két típusa van. Az egyik a taszk típusú csúcs, melyek a tevékenységeket jelölik. A másik a termékeket jelölő termék típusú csúcs. Utóbbihoz nem tartozik

berendezés, mert nem kell végrehajtani, csak az algoritmus követeli meg, hogy része legyen a gráfnak. Az egymást követő taszkok között a sorrendet a gráf élei jelölik. Ha egy termékből több egységnyt (több bachtet) gyártunk, akkor a termékhez tartozó teljes receptet többszörözzük a gráfban, beleértve magát a terméket jelölő csúcsot is. Az így kialakult végleges recept-gráfban jelölje N_t a taszkok, N_p a termékek halmazát. Nyilvánvaló, hogy $N_t \cap N_p = \emptyset$. A recept-gráfban szereplő élek adják a recept-élek halmazát, így a recept-gráf jelölése S-gráfként: $G(N, A_1, \emptyset)$.

Egy egyszerűbb példa látható recept-gráfra a 4.1 ábrán, ahol minden taszkot csak egy berendezés tud végrehajtani. A körökbe írt számok a csúcs sorszámát jelölik, míg alattuk a tevékenységet elvégezni képes berendezés azonosítója látható. Az éleken a súlyok a tevékenység elvégzéséhez szükséges időt jelölik. Ha egy taszkot több berendezés is végre tud hajtani, akkor adunk egy azonosítót az adott taszkhoz tartozó berendezések halmazának, és ezt írjuk a körbe, továbbá az élekre általában nem írunk semmit.



4.1. ábra. Példa recept-gráfra

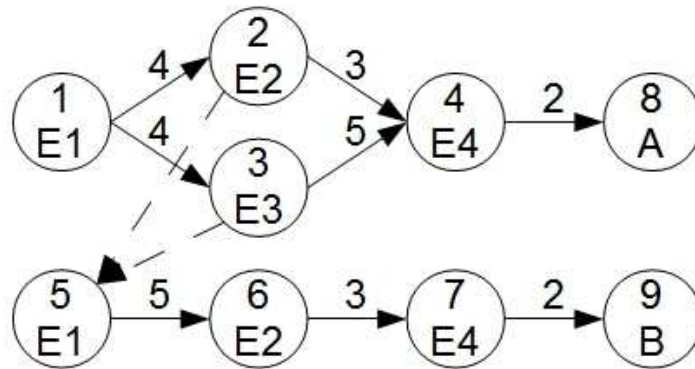
4.3. Ütemezési-gráf

Az ütemezési-gráf a recept-gráf kiegészítése ütemezési-élekkel, melyeket az A_2 halmaz jelöl. A $G'(N, A_1, A_2)$ gráfot a $G(N, A_1, \emptyset)$ recept-gráf egy ütemezési-gráfjának nevezzük. Egy feladat minden egyes megvalósítható megoldásához tartozik egy ütemezési-gráf, amely reprezentálja a megoldást, viszont ütemezési gráf csak a végleges megoldásokhoz tartozik. A cél, hogy megfelelő algoritmussal megtaláljuk az optimális ütemezéshez tartozó ütemezési-gráfot.

Az ütemezési feladatok körében sokféle tárolási stratégia létezik, ebben a dolgozatban csak a NIS (nonintermediate storage) tárolási stratégiával foglalkozom, aminek jellemzője, hogy nincs semmilyen köztes tároló egység, ahol az anyagokat lehet tárolni, ha éppen nem dolgozunk velük, vagyis egyik berendezésből közvetlenül kell át tölteni a következő berendezésbe, ami elvégzi rajta a következő feladatot. Ez egy igen népszerű típusa a feladatoknak, ugyanis nyilván egyik berendezés sem kezdheti el a következő feladatát, mielőtt az

aktuális anyagot ki nem töltjük belőle, illetve ha szükség van rá, akkor a tisztítási időt is figyelembe kell venni. Viszont, mivel nincs köztes tárolóegység, így nem tölthetjük ki az anyagot a berendezésből, amíg a következő berendezés nincs kész a befogadására. Tehát a feladat során alkalmazott S-gráfba úgy kell behúzni az ütemezési-éleket, hogy ez a tulajdonság teljesüljön.

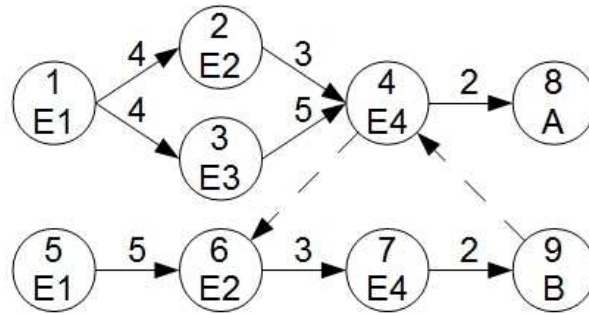
Legyen τ_j azon csúcsok halmaza, melyek a recept szerint közvetlenül követik a j csúcsot. Ha az E_i berendezés hajtja végre a j csúcsot, majd közvetlenül utána a k csúccsal folytatja működését, akkor τ_j minden eleméből húzunk k -ba egy 0 súlyú élet, illetve ha vannak extra követelmények, mint átkapcsolási idő, tisztítási idő, akkor az él súlya az ezeknek megfelelő idővel lesz egyenlő. Ez azt jelenti, hogy a k csúcs nem kezdődhet hamarabb, mint a τ_j bármelyik eleme, vagyis az E_i berendezés nem kezdheti el a következő tevékenységet, amíg az előző tevékenység által gyártott anyagot tovább nem adja a következő berendezésnek, ami azzal dolgozik majd. Ezt a típusú élet NIS ütemezési-élnek hívjuk. UIS (unlimited intermediate storage) esetén más módon kell az ütemezési-éleket behúzni, de erről a dolgozatban nem lesz szó. A NIS típusú ütemezési-élet behúzására mutat példát a 4.2 ábra, ahol jól látható, hogy az $E1$ berendezés nem kezdheti el az 5-ös számú taszkot, amíg az A termékhez tartozó receptben az $E2$ és $E3$ berendezésbe át nem töltötte a keletkezett anyagot.



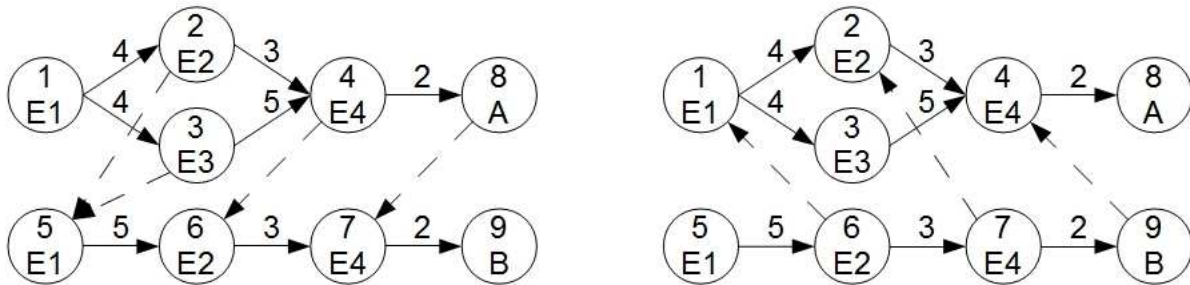
4.2. ábra. NIS ütemezési-élet

A megoldás során keletkező gráfokhoz tartozó rész-ütemezések megvalósíthatóságát is egyszerűen lehet ellenőrizni. Le kell futtatni egy körkeresést, ami irányított kört keres az gráfban. Mivel az élek definiálnak egy sorrendet a taszkok között, ha találunk irányított kört a gráfban, akkor ez azt jelenti, hogy az adott megoldás nem megvalósítható. Egy ilyen rossz rész-ütemezést szemléltet a 4.3 ábra, ahol a 4-es, 6-os, 7-es és 9-es csúcsok között található egy irányított kör.

Az optimális megoldás megtalálásához az S-gráf módszertan egy korlátozás és szétválasztás típusú algoritmust használ. A szétválasztási lépésre több stratégia létezik, de a leggyakoribb a berendezés alapú szétválasztási eljárás. Ennek lényege, hogy a szétválasztási lépés arról dönt, hogy az egyik kiválasztott berendezés melyik taszkot hajtja végre következőnek. A példa esetében az első döntés lehet az, hogy az $E1$ -es



4.3. ábra. Kör a gráfban



4.4. ábra. A példához tartozó összes megvalósítható ütemezés

berendezés az 1-es, vagy az 5-ös taszkot hajtsa végre elsőnek. Utána arról lehet dönteni, hogy melyiket hajtsa végre másodikkal.

A korlátozási lépés lényege, hogy alulról becsljük meg egy S-gráfról, hogy ha kiegészítjük ütemezési gráffá, akkor mennyi lesz ez esetben a gyártáshoz szükséges idő. Ezáltal, ha már a félkész ütemezés is tovább fut, mint a jelenlegi legjobb megoldás, akkor eldobható, mert az idő csak nőni fog. A korlátozásra az egyik módszer a leghosszabb út keresése, ami egyszerűen megnézi, hogy a jelenlegi rész-ütemezés esetén mennyi idő lenne a teljes gyártási folyamat, és ez pedig egyenlő a leghosszabb irányított út hosszával. Egy másik módszer LP (lináris programozási) feladatot ír fel a korlátozásra, ami a leghosszabb út keresését egészíti ki olyan plusz időkkal, melyek az ütemezés befejezése során biztosan fennállnak, és így pontosabb becslést ad, viszont több számolási időbe telik a becslés megadása.

A példában megadott recepthez csak két megvalósítható ütemezés létezik, ezek láthatóak a 4.4 ábrán. Innentől már csak az a kérdés, hogy melyik az optimális. Egy ütemezés végrehajtásához szükséges idő az ütemezési-gráfban található leghosszabb irányított út hosszával egyenlő. Az első esetben ez 14, a másodikban 16, így az ábrán látható első ütemezés az optimális a feladatra, feltéve, hogy mind a két termékből pontosan egyet szeretnénk gyártani.

5. fejezet

Megvalósíthatósági teszten alapuló algoritmusok

Az S-gráffal történő throughput maximalizálás alapötlete, hogy diszkrét megvalósíthatósági tesztek futtatunk. Egy ilyen teszt képes megmondani egy részfeladatról, hogy az megvalósítható-e a megadott időkorláton belül. A részfeladatok egy recept-gráfból állnak, amelyben beállítjuk, hogy az egyes termékekből mennyit gyártunk. Ez lehet 0 is. A megvalósíthatósági tesztek ezen részfeladatokra futtatjuk, és a tesztek eredményeit felhasználva jutunk el az optimális megoldáshoz.

Az első fázisban az algoritmus meghatározza, hogy ha csak egyféle terméket gyártana, akkor mi lenne a maximum mennyiség, ami belefér az időkorlátba. Ehhez is megvalósíthatósági tesztek használ, de hogy mennyit, konkrétan mely esetekre, az most nem lényeg, ezt a bejárési stratégia során lehet változtatni. Ha minden termékhez megkapta ezeket a maximum mennyiségeket, akkor meghatározza a maximális keresési teret, ahol a megvalósítható megoldások vannak. Semmiképp sem lehet olyan megvalósítható megoldás, ami bármelyik termékből többet gyárt, mint ezek a maximumok. Ezeket a megvalósítható eseteket el lehet képzelni egy n dimenziós koordináta-rendszer rácspontjainak, ahol az egyes koordináták azt jelentik, hogy az adott termékből mennyit gyártunk. Az n a termékek számát jelöli. Ez esetben a megoldható esetek abban az n -dimenziós téglatestben helyezkednek el, melynek bal alsó sarka az origó, jobb felső sarka pedig a legnagyobb potenciálisan megvalósítható eset, melyben minden termékből annyi készülne el, amennyi az előzetes számítások alapján legfeljebb lehet. Példaképpen tekintsünk egy olyan feladatot, ahol két terméket gyártunk, A -t és B -t. Az A termékért kapott bevétel 10 egység darabonként, míg a B termékért 19 egységnyi bevétel jár. A megvalósíthatósági tesztek alapján kiderült, hogy az A termékből legfeljebb 5, míg a B termékből legfeljebb 4 darab legyártása fér bele a megadott időkorlátba ($M_A = 5, M_B = 4$). Ez alapján a teljes keresési tér két dimenziós az 5.1(a) ábrán látható módon, ahol az egyes tengelyek a termékeknek felelnek meg a rácspontok pedig az egyes részeseteket jelölik. A koordinátarendszer (4,1) pontja azt a részesetet jelöli, ahol az A termékből 4-t, a B -ből pedig 1-et gyárt a rendszer.

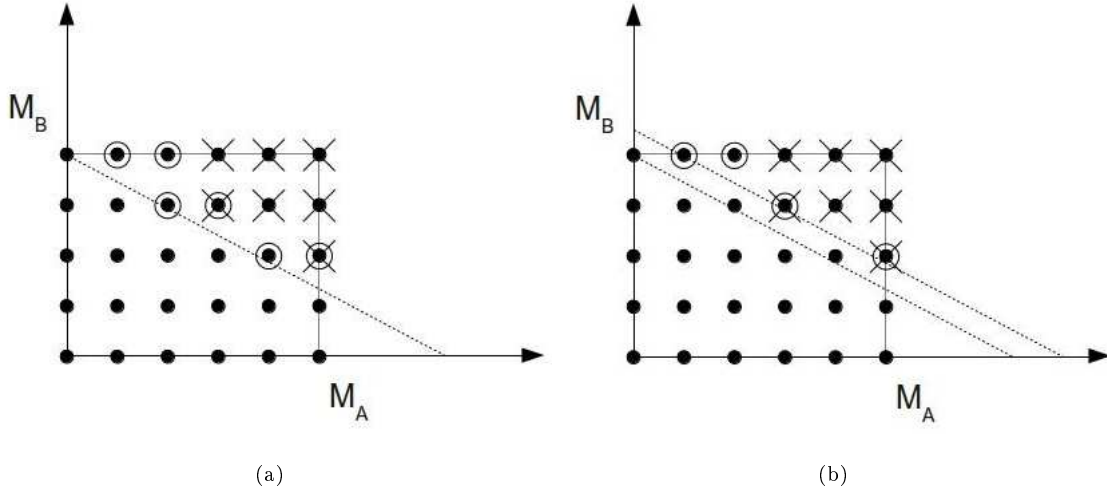
Az algoritmus ezután minden termékhez meghatározza, hogy ha abból a maximumot gyártja, akkor az mennyi bevételt hoz, és ezek közül a legnagyobbat eltárolja a pillanatnyi legtöbb bevételt tároló változóba. Ugyanis bármilyen megoldást talál is később, ennyi profitot már biztosan tud termelni a rendszer. Továbbá a nyílt feladatok halmazába elmenti az összes csúcst az n -dimenziójú téglatesten belül, ami legalább akkora bevételt hoz, mint az aktuálisan legjobb érték. Jelen példa esetében, az A termékből 5-et lehet legyártani, ez $5 \cdot 10 = 50$ egységnyi bevételt jelent, míg a B termékből 4 darab legyártása $4 \cdot 19 = 76$ egység. A kettő közül a 76 a nagyobb, így ez lesz a kiválasztott referencia érték.

Az algoritmus második fázisában amíg van részfeladat a nyílt feladatok halmazában, addig kiválaszt egyet. Ha ez megvalósítható az időkorláton belül, akkor ennek a bevételével felülírja az eddigi legnagyobb bevételt, és kizedi azokat az eseteket a nyílt feladatok halmazából, amik kevesebbet hoznak, mint az új legnagyobb bevétel. Az ábrákon az aktuálisan legjobb bevételt a szaggatott vonallal jelölt revenue (bevétel) vonal fogja jelölni, melynek kezdeti helyzete az 5.1(a) ábrán látható. Amennyiben egy részeset által hozott potenciális bevétel kevesebb, mint az aktuális legtöbb, akkor az a részfeladat a revenue vonal alatt fog elhelyezkedni, ha pedig pontosan ugyanannyi, akkor pont a vonalon. Ezért csak a vonal fölött elhelyezkedő részfeladatok közül kerülhet ki az aktuálisnál jobb megoldást hozó részeset. Ha az eset nem megvalósítható, akkor nem megvalósíthatónak tekint minden olyan részfeladatot is, ami az aktuálisnál nagyobb, és kiveszi őket a nyílt feladatok halmazából. Egy részfeladat akkor nagyobb a másiknál, ha minden termékből legalább annyit gyárt, és legalább az egyikből többet is. Lehetőség van arra is, hogy ezt a vonalat az algoritmus ne mozgassa, vagyis az aktuálisan legjobb értéket ne írja felül, ha talál jobbat, hanem állítsa be egy konstans értékre, hogy a teljes keresés során maradjon ott.

Az 5.1(a) ábrán látható az algoritmus működése. A tengelyek az egyes termékekből gyártott mennyiséget ábrázolják, a szaggatott vonal jelöli a revenue vonalat, a pontok pedig a részfeladatokat. A bekarikázott részfeladatokat vizsgálja az algoritmus, az áthúzott feladatok nem megvalósíthatóak. Az 5.1(b) ábrán az algoritmus működése látszik, a revenue vonal mozgatása esetén.

A megoldás során megvalósíthatósági teszt segítségével döntjük el, hogy az egyes részfeladatok megvalósíthatóak-e, vagy nem. Ez a teszt jelenleg egy vagy berendezés, vagy taszk alapú szétválasztást alkalmazó makespan minimalizáló algoritmus kicsit átírt változata. A módosítás annyiból áll, hogy ugyanúgy keres megoldásokat, mint a sima algoritmus, csak be van állítva az időlimit, hogy a fölött megoldásokat ne találjon, illetve ha talál az időkorláton belül egy megoldást, akkor leáll, és nem foglalkozik azzal, hogy megtalálja az optimálisat.

Miután minden termékből megkaptuk, hogy mi a maximálisan legyártható mennyiség, könnyen kiszámolhatjuk, hogy legfeljebb hány részfeladatot kell megvizsgálni. Például ha van 3 termékünk, A , B és C , és az A -ból legfeljebb hármat gyárthatunk, a B -ből négyet, a C -ből pedig kettőt, akkor összesen legfeljebb $(3+1) \cdot (4+1) \cdot (2+1) = 60$ részfeladat létezik, beleértve azokat is, amelyeket már megvizsgáltunk, mert csak egy terméket gyártanak. Ez alapján előre készítünk egy tömböt, ami ekkora méretű lesz, és ebben fogjuk tárolni a statisztikai adatokat a már megvizsgált részfeladatokról. Az olyan esetekről, amik csak egy terméket



5.1. ábra. (a) A keresési tér, a szaggatott vonal a revenue vonal, a bekarikázott esetekre futtat az algoritmus megvalósíthatósági tesztet, az áthúzott esetek megvalósíthatatlanok (b) A revenue vonal mozgatásával általában kevesebb részfeladatot kell vizsgálni

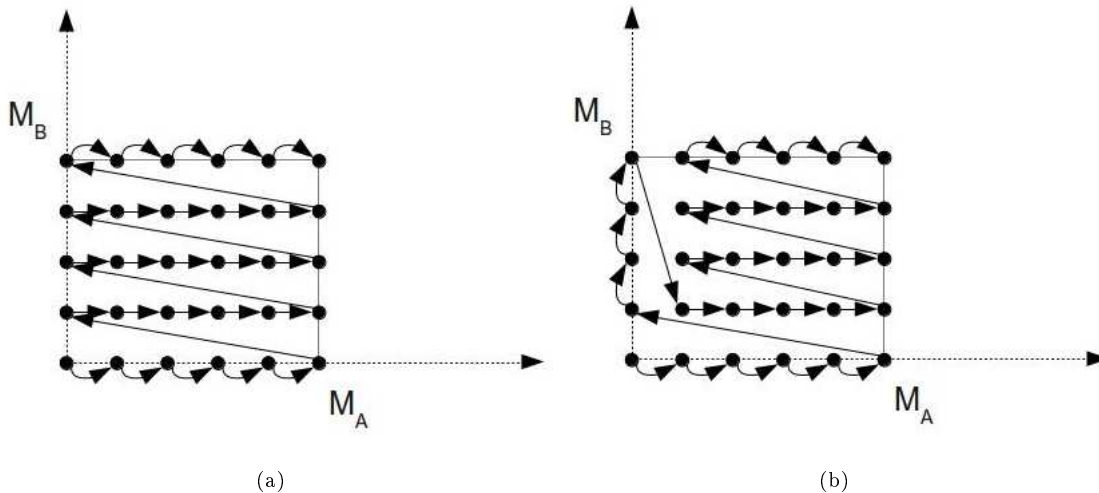
tartalmaznak, mivel már megvizsgáltuk őket, beírhatjuk a tömb megfelelő helyére, hogy megvalósíthatóak. A tömbben az egyes részfeladatok helyét egyszerű indextranszformációval számolhatjuk ki. Azért kell egydimenziós tömböt, és indextranszformációt alkalmazni a megoldás során, mert C-ben és C++-ban nincs lehetőség arra, hogy n dimenziós tömböt hozzunk létre, ha nem tudjuk már fordítási időben az n értékét. És jelen esetben nem tudjuk, mert a keresési tér annyi dimenziós, amennyi termékünk van a feladatban, és ez csak futás közben derül ki.

Az előbbi példában, ha az A , B , C termékekből rendre 1, 2, 3 batch-et gyártó esetet vizsgáljuk, akkor a tömbben ennek az $1 * ((4+1) * (2+1)) + 2 * (2+1) + 3 = 24$ -es index felel meg. A tömb számozása 0-val kezdődik, mert az algoritmusok C++ nyelven lettek implementálva. A transzformáció visszafele is elvégezhető, tehát az indexből is kinyerhetők a batch-számok. Ezen indexelés előnye, hogy egyszerű a szabálya, és a nagyobb részfeladat nagyobb indexszel szerepel a tömbben.

A következő alfejezetekben olyan korábbi algoritmusokról lesz szó röviden, melyek megvalósíthatósági teszteket alkalmazva adják meg egy throughput maximalizálási feladat megoldását. Az ismert algoritmusok között szerepel a széles körben alkalmazott sorrendi, mélységi és szélességi bejárás, továbbá egy újonnan kitalált heurisztika, a mohó bejárás. Az algoritmusok annyiban térnek el, hogy az előbb említett módon generált keresési teret más módszerrel járják be. A mélységi és a szélességi bejárás esetében lehetőség van a keresési tér szűkítésére a revenue vonal mozgatásával, de arra is, hogy ne szűkítsük a keresési teret. A tesztelések folyamán ez a két módszer mindkét lehetőség szerint tesztelésre került. Mindegyik algoritmus úgy kezdi futását, hogy meghatározza, melyik termék gyártásából mennyi fér bele az időbe, majd ez alapján meghatározza a teljes keresési teret a maximális méretű részfeladattal, a későbbiekben tehát csak a második fázisról lesz szó.

5.1. Sorrendi bejárás

Miután az algoritmus elvégezte az első fázist, létrehozta a tömböt, és feltöltötte azokkal az eredményekkel, amelyeket korábban már kiszámolt, akkor elindul az elejéről, és egyszer végigmegy az egész tömbön, sorrendben. Ha talál olyan esetet, amiről még nem tudja, hogy megvalósítható-e, vagy nem, akkor először kiszámolja, hogy mekkora bevételt hozna az adott eset. Ha legfeljebb annyit, mint a jelenlegi legjobb megoldásunk (revenue vonal mozgatása esetén), akkor bejelöli a részfeladatot vizsgálnak, de nem számol vele tovább. Ha több bevételt hozna, mint az aktuális legjobb eset, akkor meghívja rá a megoldhatósági tesztet. A teszt egy logikai értékkel tér vissza, hogy az adott részfeladat megoldható-e az időkorláton belül, vagy nem. Ha megoldható, akkor találtunk egy kedvezőbb megoldást, mint az eddigi legjobb, tehát ennek a bevételére frissítjük a legnagyobb bevételt, továbbá a tömbben is eltároljuk, hogy ez az eset megvalósítható. Ha nem megoldható, akkor a tömbben ezt jelöljük, továbbá az összes olyan részfeladatot, ami a korábban definiált módon nagyobb az éppen vizsgálnál, szintén bejelölünk megvalósíthatatlannak, ugyanis, ha ez a részfeladat sem megvalósítható, akkor ha hozzáveszünk még batch-eket, azzal nem tudjuk csökkenteni a megvalósításhoz szükséges időt. Ez abból adódik, hogy csak olyan tulajdonságú feladatokon dolgozunk, ahol ez az állítás teljesül, és a valós életbeli feladatok is teljesíteni szokták ezt a tulajdonságot. Elképzelhető olyan eset, amikor nem így van, de ez még a nagyon komplex feladatok között is csak akkor fordul elő, ha szándékosan ilyen feladatot találunk ki.



5.2. ábra. (a)A sorrendi bejárás által a megvizsgált csúcsok sorrendje (b)A sorrendi bejárás, ha először meghatározzuk az egytermékes maximumokat

A 5.2(a) ábrán látható, hogy a sorrendi bejárást használva milyen sorrendben vizsgáljuk meg a részfeladatokat. Azonban ez a bejárési sorrend csak azután érvényes, hogy meghatároztuk a keresési teret, vagyis előtte minden termékről megmondtuk, mennyi a maximum, amit gyárthatunk belőle. Így azokat az eseteket, ahol csak egy terméket gyártanánk, már még egyszer nem kell megvizsgálni. Ezért a keresés sorrendje tulaj-

donképpen módosul a 5.2(b) ábrán láthatóra, ami a legalább két terméket gyártó eseteket már ugyanabban a sorrendben járja be, mint az eredeti eljárás.

A megoldás elején a termékeket sorba lehet rendezni valamilyen tulajdonság alapján, és kísérletezni, hogy milyen rendezési heurisztika ad gyorsabb futási időt. A dolgozat nem foglalkozik ezen sorbarende­zések lehetőségeivel és hatásaival.

5.2. Mélységi és szélességi bejárás

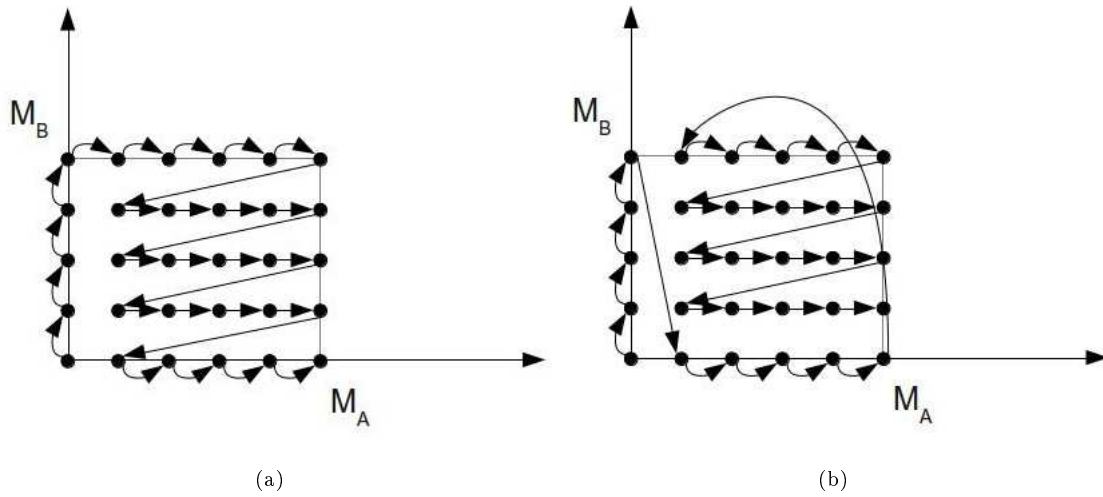
A mélységi bejárás algoritmus tárol egy listát is a szokásos tömb mellé, ami a nyílt részfeladatokat tartalmazza, a tervezett vizsgálati sorrendben. Az inicializálás után ennek a listának egy eleme lesz, az az eset, amikor semmit sem gyártunk. Továbbá sorba rakja a termékeket, hogy később e szerint a sorrend szerint generálja az egyes részfeladatok gyerekeit. A rendezés szabálya tetszőleges lehet, az implementált rendezés minden termékhez összeszorozza a maximálisan gyártható mennyiséget a batch-enkénti bevétellel, és ez alapján rendezi sorba a termékeket.

Az eljárás mindig a lista első elemét veszi ki, és vizsgálja meg. Amennyiben ezt a részfeladatot már inicializáláskor megvizsgáltuk, mert csak egy terméket gyártunk benne, úgy nem elemzi még egyszer, csak legyártja a még nem generált, és nem biztosan megvalósíthatatlan gyerekfeladatokat. A gyerekfeladatok generálása mindig úgy történik, hogy az előre megadott sorrendben veszi a termékeket a program, és a szülő problémát kiegészíti egy olyan batch-csel. Ha az így kapott esetről a tömbben az áll, hogy nem tudunk róla semmit, akkor hozzáfűzi a lista elejéhez. Ha már generált az adott eset, vagy már biztosan megvalósíthatatlan, akkor nem foglalkozunk tovább vele.

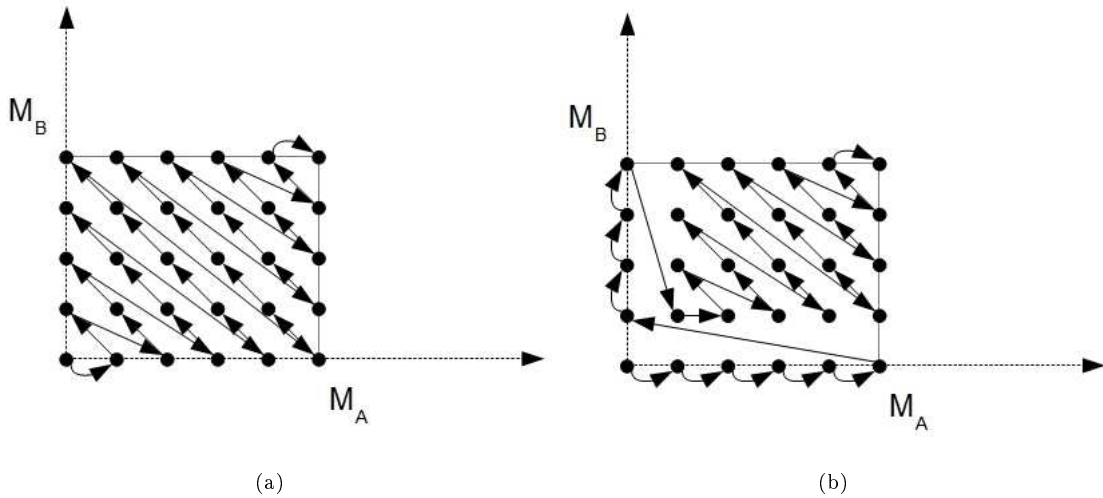
Amennyiben a vizsgált eset kevés bevételt hoz, úgy nem elemzi, csak generálja a gyerekeit, és halad tovább a következő esetre, ami az aktuálisnak egy gyereke lesz. Ha az aktuálisan tárolt határnál több bevételt hozna az adott eset, akkor megvizsgálja az algoritmus, és az eredményt elmenti a tömbbe. Ha megvalósítható az időkorláton belül, akkor generálja a gyerekeit, és hozzáteszi őket a lista elejéhez. Ha nem, akkor hasonlóan az előbbi eljáráshoz, a nála nagyobb feladatokat is bejelöli megvalósíthatatlannak.

A sorrendi bejáráshoz hasonlóan a mélységi bejárás esetében is az egytermékes eseteket járjuk be először. Ezért az eredeti, a 5.3(a) ábrán látható sorrend a 5.3(b) ábrán láthatóra módosul.

A mélységi mellett lehet szélességi bejárást is alkalmazni, ami nem az éppen vizsgált esetnek az egyik gyere­kével folytatja a keresést, hanem olyan sorrendben vizsgálja az eseteket, amilyen sorrendben generálta őket. Megvalósítás szempontjából ez nem jelent nagy különbséget, csak annyit, hogy ha egy eset megvalósítható, akkor a gyerekeit a még nem vizsgált részfeladatok listájának végére teszi, de továbbra is a lista első elemét veszi ki vizsgálatra. A 5.4(a) ábra szemlélteti ez esetben az elméleti bejárési sorrendet, ami az inicializáló vizsgálatok miatt a 5.4(b) ábrán láthatóra módosul.



5.3. ábra. (a)A mélységi bejárás által a megvizsgált csúcsok sorrendje (b)A mélységi bejárás, ha először meghatározzuk az egytermékes maximumokat



5.4. ábra. (a)A szélességi bejárás által a megvizsgált csúcsok sorrendje (b)A szélességi bejárás, ha először meghatározzuk az egytermékes maximumokat

5.3. Mohó bejárás

A bejárési stratégia lényege, hogy óvatosan haladva, de minél gyorsabban eljussan az optimális megoldáshoz. Kezdetben ugyanúgy megvizsgálja az egytermékes eseteket, és felépíti a tömböt, mint az előző algoritmusok. Ezek után sorba rendezi a termékeket, a batch-enkénti bevétel szerint. Amikor el kell döntenie, hogy melyik részfeladat legyen a következő, amit megvizsgál, akkor a keresés során két határt használ a bevételre vonatkozóan. Az alsó határ az éppen aktuális bevételnek felel meg, ez nyílt határ, vagyis az olyan eseteket, amiknek a bevétele megegyezik az aktuális legjobbal, már nem vizsgálja. A felső határt úgy kapjuk, hogy az aktuálisan legjobb bevételhez hozzáadjuk a legkisebb bevételű termék bevételét. Ez zárt határ, vagyis az erre illeszkedő esetek benne vannak a keresési térben. Ezután kiválasztja az így kapott keresési térből a legnagyobb bevételt hozó esetet, amit még nem vizsgáltunk, ez lesz a következő eset, amire elvégzi a megvalósíthatósági tesztet. Ha megvalósítható, akkor felülírja az eddig megtalált legjobb megoldást. Ha nem megvalósítható az időkorláton belül, akkor a nála nagyobb eseteket is kizárja, az előző algoritmusokhoz hasonlóan.

Az algoritmus így önmagában még nem ad biztosan optimális megoldást, ugyanis előfordulhat olyan speciális eset, amikor ez a fajta lépegetés nem működik. Ezért az algoritmushoz hozzá tartozik a következő kiegészítés. Először csak azokat az eseteket vizsgáljuk meg, ahol csak a két legnagyobb bevételt hozó terméket gyárthatjuk, és a többi nem. Mohó bejárással határozzuk meg, hogy ezzel a kikötéssel mi a legnagyobb bevétel, amit el tudunk érni. Az algoritmus egy termék esetére még optimális megoldást ad, azonban azokat az eseteket, amikor csak egy terméket gyárthatunk, megvizsgáltuk az algoritmus legelején, hogy felépítsük a keresési teret reprezentáló tömböt. Így ezt a részt átugorva az algoritmus még két termék esetében is biztosan megtalálja az optimális megoldást. Az így kapott legnagyobb bevételt állítsuk be alsó korlátnak. Majd lépünk egy szinttel feljebb, és foglalkozunk a harmadik legnagyobb bevételű termékkel is. A mohó bejárást alkalmazva most már biztosan meg tudjuk adni, hogy jelen esetben mi az optimális megoldás. Ezután léphetünk tovább, foglalkozhatunk a következő termékkel is, aminek a maradék közül legnagyobb a bevétele. Ezt addig folytatjuk, amíg az összes termék fel nem került a listára. Ezzel a kiegészítéssel az algoritmus már bizonyosan optimális megoldást ad, amire van matematikai bizonyítás is, de az a jelen dolgozatnak nem része.

6. fejezet

Egy új megközelítés: a globális keresőfa

6.1. Az algoritmus leírása

A megvalósíthatósági teszten alapuló algoritmusokra számos heurisztikát ki lehet találni a keresési tér bejárására vonatkozóan, egyetlen gyengesége azonban minden ilyen algoritmusnak lesz: a megvalósíthatatlan részesetek. Egy részfeladatról ugyanis úgy derül ki, hogy megvalósíthatatlan, hogy a részfeladathoz tartozó teljes keresési teret be kell járni, mert csak így lehetünk biztosak benne, hogy nem fér bele az időkorlátba. Ahogy nő egy részfeladatnak a mérete, a megvalósíthatatlan esetek detektálása hatványozottan egyre több számítási kapacitásba kerül. A megvalósítható esetekkel nincs nagy probléma, mert lehet olyan heurisztikát találni, ami az esetek nagy részében gyorsan talál megoldást, ha van. Az implementálás során a megoldhatósági teszthez használt berendezés alapú szétválasztó algoritmus is így működik.

Vegyünk egy olyan példát, ahol van három termékünk, A , B , és C . A futás során már meghatároztuk, hogy a $3A + 4B + 2C$ eset, vagyis amikor az A termékből 3-at, a B -ből 4-et, a C -ből pedig 2-öt gyártunk, megvalósítható. Azonban a teszteket tovább futtatva kiderül, hogy a $4A+4B+2C$, $3A+5B+2C$, $3A+4B+3C$ esetek közül egyik sem megvalósítható. Azt bizonyítani, hogy a három részfeladat közül egyik sem fér bele az időkorlátba, hatalmas mennyiségű számolást igényelt. Ha azonban meg tudnánk azt csinálni, hogy a $3A+4B+2C$ esetnek generáljuk a teljes keresési fáját, akkor is, ha megvalósítható, és ezeket az eredményeket felhasználjuk a belőle elágazó másik három eset vizsgálatához, akkor ezen részfeladatoknál már nem kellene előről felépíteni a teljes keresőfát, hanem egy jó része már előre kész lenne. Ez gyakorlatban azt jelenti, hogy a három előről felépített keresőfa helyett mind a hármat egy, már előre meglévő fát kiegészítve kapjuk meg, így a közös részt, amit a három keresés külön-külön elindításakor háromszor felépítünk, most elég egyszer.

A módszernek van azonban egy hátránya is. A berendezés alapú szétválasztási algoritmust nem lehet használni, mivel ha egy berendezéshez már hozzárendeltünk néhány taszkot, akkor a következő taszk esetében csak arra van lehetőségünk, hogy a végrehajtási lista végéhez fűzzük hozzá. Ez korlátozná a keresést abban az esetben, ha a részfeladat keresési fáját nem előről építjük fel, hanem egy másik részfeladatot folytatva.

Szerencsére ez a probléma áthidalható, ugyanis Adonyi Róbert a doktori disszertációjában [11] ismertette a taszk alapú szétválasztási algoritmust, aminek lényege, hogy egy berendezés már meglévő gyártási sorában a sorrenden nem változtathatunk, de újabb taszk végrehajtását bárhova ütemezhetjük, az elejére éppúgy, mint a végére, vagy akár a közepére is. Ezen módszer lehetővé teszi, hogy ha egy berendezéshez valamennyi (pl. 3) taszk már hozzá van rendelve, és az egymáshoz képesti sorrendjüket is meghatároztuk, akkor generáljuk az összes olyan gyártási sort a berendezéshez, ami minden hozzá tartozó taszkot végrehajt, de a korábban beütemezett 3 taszk sorrendje nem változik. A berendezés alapú szétválasztás esetén a gyártási sort csak úgy lehetne kiegészíteni, hogy a korábban ütemezett 3 taszk a gyártási sor első 3 helyét foglalja el.

Miért igaz az, hogy a taszk alapú szétválasztást használó globális keresőfa algoritmus minden megvalósítható megoldást generál, köztük az optimálisat is? Tegyük fel, hogy az előző példában már generáltuk és megvizsgáltuk a $3A + 4B + 2C$ részfeladat teljes keresési terét, és megtartottuk az összes megvalósítható megoldást. Majd ezt a fát kibővítvé szeretnénk generálni a $4A + 4B + 2C$ eset teljes keresési fáját. Ekkor a $3A + 4B + 2C$ eset minden egyes megvalósítható megoldásából kiindulva ütemezzük az újabb A termékhez tartozó taszkokat. Az állítás, hogy ezt a folyamatot végigcsinálva, a $4A + 4B + 2C$ feladat minden megvalósítható megoldását megkapjuk. Az egyértelmű, hogy nem kaphatunk olyan megvalósítható ütemezést, amely nem ehhez a részesethez tartozik, hiszen a megkapott ütemezésekben pontosan azok a taszkok és berendezések szerepelnek, amelyek a részfeladatban. Vegyük egy tetszőleges megoldását a $4A + 4B + 2C$ részfeladatnak. A részfeladat megoldását egyértelműen definiálják az egyes berendezésekhez tartozó gyártási sorok. Ennek igazolásához mélyebben bele kellene merülni a berendezés alapú és a taszk alapú szétválasztási algoritmusok működésébe, mely a dolgozatnak nem témája. Röviden összefoglalva S-gráf esetén a feladat megoldását az határozza meg, hogy mely élek vannak behúzva az ütemezési gráfban. Az élek alapján minden taszkhoz ki lehet számolni, hogy mekkora a leghosszabb út, ami az adott taszkhoz vezet, és ezen adatok alapján lehet megadni, hogy időben melyik taszkot mikor hajtsuk végre. A 4.3. alfejezetben definiáltuk az ütemezési éleket, melyek abból adódnak, hogy a berendezés végrehajtási sorában mi a taszkok sorrendje. Ezért ha adott minden berendezéshez a gyártási sor, azok alapján egyértelműen be lehet húzni az ütemezési éleket a gráfba, amik már egyértelműen definiálják a pontos megoldást. A taszk alapú részválasztási eljárás során más élek is kerülnek az ütemezési gráfba, de azok nem befolyásolják a megoldást [11].

Tehát a berendezésekhez tartozó gyártási sorok egyértelműen meghatároznak egy megoldást. Nézzünk tehát egy tetszőleges megoldását a $4A + 4B + 2C$ részfeladatnak. Ebben a megoldásban minden berendezéshez kész van a végleges gyártási sor. A megoldás megvalósítható. Vegyünk egy olyan ütemezést, melyben nem szerepelnek a negyedik A termékhez tartozó taszkok, de a többi taszkot ugyanaz a berendezés hajtja végre, ugyanabban a sorrendben, mint a vizsgált megoldást esetében. Ez egy megvalósítható megoldása lesz a $3A + 4B + 2C$ részfeladatnak, mivel a megfelelő taszkok és berendezések szerepelnek benne, minden taszk ütemezve van, és nyilván megvalósítható, mert taszkok elhagyásával a végrehajtási idő csak csökkenhet, mivel az ütemezési gráfból csak kiszedtük éleket és csúcsokat, újat nem raktunk hozzá, így kör sem alakulhatott ki, mivel eddig sem volt benne. Tehát a $4A + 4B + 2C$ részfeladat tetszőleges megoldásából kiszedve a 4. A

termékhez tartozó taszkokat, a $3A + 4B + 2C$ részfeladat egy megoldását kapjuk. Ebből a megoldásból kezdve az új A termékhez tartozó taszkokat több megoldást is kaphatunk a $4A + 4B + 2C$ részfeladatra, köztük azt is megkapjuk, amelyikről éppen szó volt. Látható tehát, hogy a $4A + 4B + 2C$ feladat minden megoldását megkaphatjuk a $3A + 4B + 2C$ valamelyik megoldásának tovább számolásával.

A taszk alapú szétválasztási eljárásan alapuló globális keresőfa módszer tehát bizonyítottan generál minden megvalósítható megoldást a feladatra, így megtalálja közülük az optimálisat is. Ez azonban eddig még csak egy elméleti leírás az új módszerről, a gyakorlati megvalósítás viszont egy külön témakör. Hasonlóan a megvalósíthatósági teszten alapuló algoritmusokhoz, a globális keresőfa bejárására is számos, egymástól eltérő lehetőség van. A legegyszerűbb megvalósítás az egyszerű mélységi bejárás. Ez esetben, amíg van ütemezetlen taszk a részfeladatnál, addig azt ütemezzük taszk alapú szétválasztási módszerrel. Ha az aktuális részfeladatra megoldást találtunk, akkor adjuk egy új terméket a feladathoz, aminek folyamán ismét lesznek ütemezetlen taszkok, amiket lehet ütemezni. Az állapotokat a keresési fába elmentjük, és minden esetben a jelen állapot egyik gyerekével folytatjuk a keresést, ezáltal megvalósítva a fa mélységi bejárását. Ha egy állaptonak minden gyereket generáltuk, és visszalépés következik az állapot szülőjére, akkor az aktuális állapotot törölhetjük. Hasonlóan egyszerű módszer lehet a szélességi bejárás, ezt azonban nehéz alkalmazni, mivel ez esetben a fában eltárolt állapotok száma jelentősen megnő, ami már kisebb méretű feladatoknál is kritikus lehet, hiszen a felépített globális keresőfa az összes részfeladat összes megoldását, és a hozzá tartozó állapotokat is tárolja, ami hatalmas mennyiségű állapotot jelent. Mélységi keresés esetén ez nem jelent gondot, hiszen ezekből egyszerre viszonylag kevés van generálva, és hamar töröljük is őket, de a szélességi keresés memóriaigénye nagy, mert a generált állapotokat csak sokára töröljük.

6.2. Lehetőségek az algoritmus gyorsítására

Ez természetesen csak az alpötlete a globális keresőfa algoritmusnak. Hasonlóan a megvalósíthatósági tesztelés ötletéhez, ehhez is lehet kitalálni különböző heurisztikákat, melyek alapján a keresés menete nem mélységi keresés lesz, hanem a fát valamilyen más módszerrel járja be. A megvalósíthatósági tesztelés módszerek esetében ezeket megvalósítani is egyszerű volt, mivel csak azt kellett eldönteni, hogy az egyes részfeladatok közül melyikre fusson le a teszt, és milyen sorrendben. Továbbá az algoritmusban az egyes tesztek lefuttatása tart sokáig, annak eldöntéséhez, hogy melyik legyen a következő vizsgálandó részeset, nem kell sok idő. A globális keresőfa esetében azonban bonyolultabb a helyzet. Itt ugyanis megengedett az, hogy egyszerre több részeset megvalósíthatóságát vizsgálja az algoritmus, és hol az egyikben számoljon egy keveset, hol a másikon. Továbbá, egy részesetet itt akkor lehet megvizsgálni tekinteni, ha minden megoldását generáltuk. Megvalósíthatósági tesztek esetén egy megoldás generálása elég volt minden részesetre, hiszen ez már bizonyítja, hogy megvalósítható. Globális keresőfa esetén viszont a részeset összes megoldását fel kell használni, ha abból kiindulva vizsgál meg az algoritmus egy másikat. Persze lehet kitalálni olyan heurisztikát is, amely előre eldönti, hogy melyik részesetet melyik másik folytatásaként kell vizsgálni, így amelyik részesetből nem

származtatunk, ott elég egy megoldást megkeresni.

Amennyiben az egyszerű mélységi keresésnél bonyolultabb elven fut az algoritmus, úgy más problémák is felléphetnek. A fában ugyanis minden nyitott állapotot el kell tárolni. Minél szerteágazóbban járja be a fát az algoritmus, annál több nyitott állapotot hagy maga mögött, amiket majd csak valamikor később vizsgál meg, viszont ezek mind memóriát foglalnak. Végeztem teszteket arra vonatkozóan, hogy a szélességi keresés mennyire alkalmazható globális keresőfa esetén, és az eredmény, hogy hatalmas mennyiségű memória szükséges hozzá. Ebből az következik, hogy bármilyen heurisztikát is alkalmaz az algoritmus a fa bejárására, arra mindig figyelni kell, hogy a nyitott állapotokból egyszerre semelyik időpillanatban se legyen túl sok, mert a számítógép memóriája véges, viszont a teljes keresőfa már közepes méretű problémákra is hatalmas méretű.

A memóriafelhasználáson kívül a számítási kapacitással is problémák lehetnek. Globális keresőfát alkalmazva ugyanis nem lehet egyszerű módon olyan felső réteget megvalósítani, mint megvalósíthatósági teszteket alkalmazva. Itt minden döntést arról, hogy melyik részeset melyik folytatása legyen, a keresés közben kell meghozni. Viszont a keresés egészét az egyes lépéseken belülről nem lehet tisztán átlátni, és a jó döntések meghozásához gyakran sokat kell számolni. Így elképzelehető, hogy egy heurisztika lecsökkenti a keresőfa által megvizsgált esetek számát, de ha ennek a heurisztikának a megvalósítása programozási szempontból nehéz, és a futása sok időt igényel, akkor ez kompenzálhatja a kevesebb eset megvizsgálásából származó előnyt, és az is elképzelehető, hogy egészében az algoritmus nem lesz gyorsabb.

A makespan minimalizáló algoritmusokhoz is létezik sokféle heurisztika, melyek jelentősen lecsökkentik a futási időt. Ezek azonban gyakran hosszú kutatás eredményeképpen jöttek létre. Hasonlóan a globális keresőfa módszerhez is lehet létrehozni különféle gyorsító mechanizmusokat, melyek csökkentik a futási időt. Ilyeneket kitalálni azonban nem egyszerű, és arra is oda kell figyelni, hogy az algoritmusok implementálásának minősége felett sem lehet elsiklani. Jelenleg még nincs olyan heurisztika ehhez a módszerhez, mely hatékonyan tudná gyorsítani az algoritmus, de megfelelő mennyiségű idő ráfordításával születhetnek olyan ötletek és megvalósítások, melyek a futási idő drasztikus csökkenéséhez vezethetnek.

Ha az implementálhatóság szempontját nem vesszük figyelembe, akkor lehet különféle ötleteket gyártani az algoritmus gyorsítására vonatkozóan. Vegyünk egy olyan esetet, ahol három terméket gyártunk (A , B , C). Az algoritmus már generálta az összes megoldását a $2A + 3B + 2C$, $3A + 2B + 2C$ és $3A + 3B + 1C$ eseteknek, és a globális keresőfában ezek a megoldások már szerepelnek. A következő részeset, amit az algoritmus vizsgálni szeretne a $3A + 3B + 2C$ eset. Ekkor 3 lehetőség van arra vonatkozóan, hogy a korábban kiszámolt megoldások közül mely eredményeket használja fel az algoritmus. Jelen megvalósításban ilyen kérdésben nem kell döntenie az algoritmusnak, mivel a mélységi keresés alkalmazása miatt minden megvizsgált részeset után rögtön az abból származó esetek vizsgálata következik, így nincs választási lehetőség. A példa esetében ez azt jelenti, hogy a $3A + 3B + 2C$ eset vizsgálatához mindenképpen a $2A + 3B + 2C$ eset megoldásait használja fel.

Azonban, ha programozástechnikai szempontból meg tudjuk oldani, célszerű lehet eltárolni minden olyan

megvizsgált részeset összes megoldását, amelyre igaz, hogy a nála közvetlenül nagyobb részesetek közül még nem generálta az algoritmus mindet. Egy részeset közvetlenül nagyobb a másiknál, ha egy termékből pontosan eggyel több darabot gyárt, a többiből pedig ugyanannyit. Ilyenkor különböző előszámításokat elvégezve az algoritmus dönthet arról, hogy az éppen vizsgálandó esethez melyik korábbiak a megoldásait használja fel. A döntés során figyelembe veheti azt, hogy melyik esetnek van a legkevesebb megvalósítható megoldása, hiszen kevesebb megoldásból kiindulva valószínűleg kevesebb új állapot keletkezik majd. Egy másik megközelítés lehet az is, hogy az újonnan hozzáadott termék legyártása hány új taszk elvégzését igényli, és azokat egyenként mennyi berendezés tudja végrehajtani. Ugyanis minél több taszk tartozik az új termékhez, és minél bonyolultabb a hozzá tartozó recept, annál több lehetőséget kell megvizsgálni, így annál szerteágazóbb lesz minden egyes megoldásból a felépített keresési fa. Az előbbi két ötletet lehet keverni is. Szorozzuk össze a megoldások számát azzal, hogy várhatóan hány különböző esetet eredményez az új termék hozzáadása, és amelyik részeset esetén ez a legkisebb, abból származtatva oldjuk meg a következőt.

Azt is figyelembe lehet venni a döntés során, hogy mennyire járunk közel a megadott időkorláthoz. Ugyanis, ha a feladathoz adott időkorlát 40, és az egyik részeset legjobb megoldása 37, akkor számítani lehet rá, hogy újabb termékek, és ezáltal újabb taszkok hozzáadásával viszonylag hamar túllépi az időkorlátot. Így célszerűbb ebből az esetből tovább számolni a következőt, és nem abból, aminek a legjobb megoldása csak 34, mert ahogy az korábban szerepelt, mindegy melyik részesetet folytatjuk, pontosan ugyanazok lesznek a megoldások. Viszont ha közelebb járunk az időkorláthoz, akkor a lehetőségek közül sok várhatóan hamar kiesik, így sokkal kevesebb számolás kell az megoldások meghatározásához.

Az algoritmus gyorsítható olyan heurisztikával is, amely a megvalósíthatósági tesztelés algoritmusokhoz hasonlóan nem vizsgálja meg az összes részesetet. Ugyanis ha a $3A + 2B + 2C$ esetet már megvizsgálta az algoritmus, ez elég ahhoz, hogy a $3A + 3B + 2C$ részesettel folytassa a munkát, a másik két előzményre ($2A + 3B + 2C$ és $3A + 3B + 1C$) ehhez nincs feltétlenül szükség. Így, ha vesszük a $2A + 3B + 2C$ esetet, ennél három közvetlenül nagyobb eset van: $3A + 3B + 2C$, $2A + 4B + 2C$ és $2A + 3B + 3C$. Ha a keresés folyamán az algoritmus úgy dönt, hogy ezen három eset közül egyiket sem származtatja a $2A + 3B + 2C$ részesetből, továbbá már talált olyan részesetet, ami megvalósítható és több bevételt hoz a tárgyalt esetnél, ez azt jelenti, hogy van a feladatra jobb megoldás, mint a $2A + 3B + 2C$ eset, és az eredményeit sem használná fel az algoritmus később, akkor eleve felesleges végigszámolni az esetet. Egy olyan heurisztika tehát, ami bonyolultabb bejárást alkalmaz a részeseteken időt spórolhat azzal, hogy nem vizsgál meg minden részesetet, igyekszik elhagyni a felesleges számolásokat. Hasonlóan, ha van egy részeset, amit nem vizsgál az algoritmus, akkor felesleges megvizsgálni minden olyan részesetet is, ami csak arra kellene, hogy annak folytatásaként vizsgáljon meg valami olyat, amit nem fog. Azonban vigyázni kell a részesetek kihagyásával. Könnyen előfordulhat, hogy a $2A + 3B + 2C$ részeset megvizsgálását az algoritmus feleslegesnek tekint, ezért kihagyja, viszont ha megvizsgálná, kiderülne, hogy az eset megvalósíthatatlan, nem fér bele az időkorlátba egyik megoldása sem. Ekkor a $3A + 3B + 2C$, $2A + 4B + 2C$ és $2A + 3B + 3C$ eseteket is megvalósíthatatlannak tekinthetné az algoritmus, amivel jelentős számításokat takaríthatna meg. Mivel azonban a $2A + 3B + 2C$

részesetet nem vizsgálja meg, így ez nem derül ki, és a másik három részesetről külön-külön kell megállapítani, hogy nem férnek bele a megadott időkorlátba, ami jóval több számolás. Előfordulhat tehát, hogy a túlzott spórolás eredménye többlétszámolás lesz, és az algoritmus nem gyorsul, hanem lassul. Ezért jó át kell gondolni, hogy mely eset megvizsgálását szabad elhanyagolni.

Az algoritmus működése során az sem törvény, hogy minden egyes részesethez csak egy korábbi eset előzményei használhatók fel. Ha ez megoldható, és keletkezik rá működő heurisztika, akkor könnyen elképzelhető, hogy a $2A + 3B + 2C$, $3A + 2B + 2C$ és $3A + 3B + 1C$ részesetekből is használunk fel eredményeket a $3A + 3B + 2C$ eset megoldásához. Ezáltal sokkal rugalmasabbá válhat az algoritmus, és részletesebb heurisztikák kidolgozására nyílnak lehetőségek.

Azt is meg kell jegyezni továbbá, hogy bár a részesetek egymás folytatásaként való elemzése jelentős mennyiségű redundáns számolás elkerülésében segít, elképzelhető további gyorsítás is ezen a téren. Ugyanis vegyünk egy példát, ahol szintén három termék van, A , B , C , de vizsgáljuk meg közelebbről az $1A + 1B$, $1A + 1C$ és $1B + 1C$ eseteket. Az $1A + 1B$ és az $1A + 1C$ esetek során keletkező keresési fának van közös része, az A termék miatt, hiszen ez mind a két esetben szerepel. Ebből kifolyólag redundáns számolást kerülünk el, ha mind a két esetet az $1A$ esetből számolja tovább az algoritmus. Azonban a B termék révén az $1A + 1B$ és az $1B + 1C$ esetek keresési fájának is van közös része, továbbá a C terméken keresztül az $1A + 1C$ és az $1B + 1C$ esetekre is igaz ez. Kérdés, hogy ez esetben melyik részesetet melyik másik folytatásaként határozzuk meg. Ha az $1A + 1B$ és $1A + 1C$ eseteket az $1A$ esetet folytatva számolja ki az algoritmus, akkor az $1A + 1B$ és $1B + 1C$ részesetekben szereplő közös részt továbbra is két alkalommal generálja az algoritmus, ami továbbra is redundáns számolás. Feltételezhető, hogy a redundáns számításokat nem lehet teljes mértékben eltüntetni. Azonban, ha jelen esetben az $1A + 1B$ eset számolásakor nem csak az $1A$, hanem az $1B$ eset eredményeit is felhasználnánk, ahogy az előző algoritmus is javasolja, akkor tovább lehetne csökkenteni a redundanciát a megoldáshoz szükséges számolásokban. Ezzel azonban egy olyan komplex és zavaros felépítésű globális keresőfa jöhetne létre a keresés folyamán, amit átlátni még program szinten is nehéz lehet, és egy ilyen keresőfa hibátlan kezelése nem csak bonyolult lehet, de sok számolást is igényelne.

Jól látható tehát, hogy a globális keresőfa módszer még csak kezdetleges állapotában jár, és jelentős mennyiségű lehetőség létezik a fejlesztésére. Azonban az összes lehetőség azon kívül, hogy a keresési teret, és az algoritmushoz tartozó szükséges számolások mennyiségét csökkenti, ugyanakkor több számolást is igényel ahhoz. A következő fejezetben is látható lesz, hogy egy algoritmusnál az elméleti sebesség mellett az implementálás minősége is sokat számít. A kérdés az, hogy melyek azok a heurisztikák, melyek úgy valósíthatóak meg implementálás szintjén, hogy az ezzel járó újabb számolás kevesebb legyen, mint az a mennyiség, amit a heurisztika használata spórol meg. Jövőbeni terveim között szerepel, hogy több heurisztikát is megvalósítok, amennyiben azok hatékonyan implementálhatók. Amennyiben viszont egy heurisztika a bonyolultságához képest csak kis mértékben segíti az algoritmus működését, úgy a használata felesleges.

7. fejezet

Összehasonlító tesztek

Magyarázat a táblázatokban használt rövidítésekhez:

példa: a példa száma

idő: időkorlát

megoldás: az legtöbb bevételt hozó részeset leírása, melyik termékből mennyit gyárt az rendszer az adott esetben. Zárójelben található az optimális bevétel.

ms.: makespan, az optimális megoldás legrövidebb megvalósítási ideje.

a: sorrendi bejárás

b: mélységi bejárás frissülő legjobb értékkel

c: mélységi bejárás konstans legjobb értékkel

d: szélességi bejárás frissülő legjobb értékkel

e: szélességi bejárás konstans legjobb értékkel

f: mohó bejárás

g: globális keresőfa.

Az algoritmusokat egy már meglévő programba implementáltam, amelyben a megoldhatósági teszt gyorsan lefut. A program hatékonysága jelentős szerepet játszott abban, hogy a tesztek viszonylag gyorsan lefutottak.

Az első példát a valós élet generálta, Majozsi és Friedler mutatták be 2006-os [9] throughput maximalizálásról szóló cikkükben. A példa egy dél-afrikai gyár által termelt öt termék kétlépéses gyártási folyamatát szemlélteti. Az eredeti példában 24 időegység (ami jelen esetben órát jelent) alatt kellett a legtöbb terméket legyártani. A tesztek során magasabb időkorlátokra is teszteltem az algoritmusokat.

A feladat receptje a 7.1 táblázatban látható. A feladatot ábrázoló receptgráf a 7.1 ábrán látható. A 7.2 táblázatból kiolvasható, hogy különböző időkorlátokra mennyi idő alatt futottak le az egyes algoritmusok. Az időket megjelenítő táblázatokban minden időkorláthoz minden algoritmusnál két idő található. Ezek közül a felső a berendezés alapú szétválasztást használó teszt futási ideje, míg az alsó időt taszk alapú szétválasztás esetén mértem. A globális keresőfa algoritmusnál csak az alul megjelenő, taszk alapú szétválasztásról van futási eredmény, mivel ezen algoritmus mindenképpen taszk alapú szétválasztást használ.

	A(2)	B(3)	C(1)	D(3.5)	E(1.5)			
1	V1(10)	V1(12)	V3(12)	V2(8)	V1(10)			
	V2(5)	V2(10)		V3(13)	V2(6)			
	V4(5)	V3(7)			V4(9)			
2	P1(12)	P1(12)	P1(12)	P1(12)	P1(12)			
	P2(12)	P2(12)	P2(12)	P2(12)	P2(12)			
	P3(12)	P3(12)	P3(12)	P3(12)	P3(12)			
Berendezés		V1	V2	V3	V4	P1	P2	P3
Darabszám		1	1	1	1	1	1	1

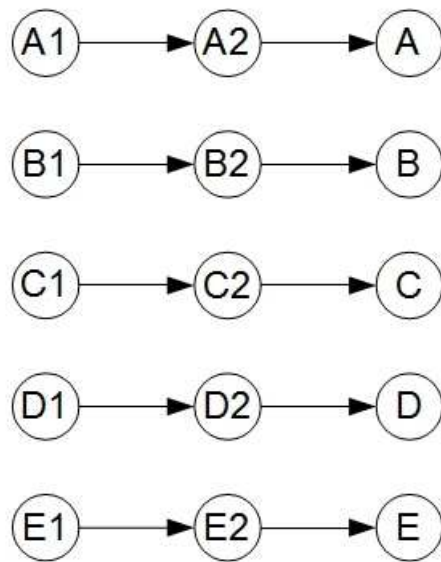
7.1. táblázat. Az első tesztpélda receptje

Az egyes megvalósíthatósági teszten alapuló algoritmusok közötti különbségekre részletesen nem térnek ki. A legszembetűnőbb jelenség, hogy berendezés alapú szétválasztás esetén a konstans legjobb értékkel dolgozó algoritmusok (c és e) a leggyorsabbak, míg taszk alapú szétválasztás esetén a mélységi keresés (b és c) a leglassabb. Általánosan elmondható, hogy erre a példára a taszk alapú szétválasztás gyengébben teljesít, mint a berendezés alapú szétválasztás, a globális keresőfa algoritmus pedig egyértelműen lassabb a többi algoritmusnál.

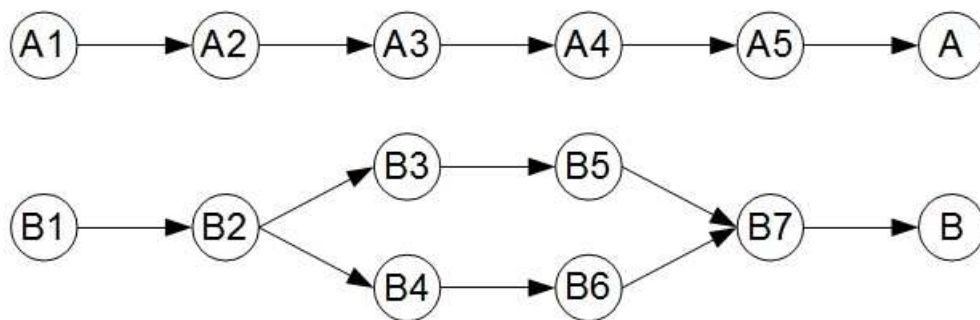
A második feladat receptje a 7.3 táblázatban látható. A feladatot ábrázoló receptgráf a 7.2 ábrán látható, a futási időket pedig a 7.4 táblázat ábrázolja. Ezen példa esetében az vehető észre, hogy a taszk alapú szétválasztás a legtöbb esetben gyorsabb a berendezés alapú szétválasztásnál, sőt, néha sokkal gyorsabb. A globális keresőfa algoritmus hullámzó teljesítményt mutat.

A második feladat receptje a 7.5 táblázatban látható. A feladatot ábrázoló receptgráf a 7.3 ábrán látható, a futási időket pedig a 7.6 táblázat ábrázolja. A tesztek eredményeit vizsgálva az a jelenség figyelhető meg, hogy a berendezés alapú szétválasztással futó algoritmusok jóval rövidebb idő alatt megoldották a problémát, mint a taszk alapú szétválasztást használók. Érdekes módon azonban a globális keresőfa meg ennél is kevesebb időt vett igénybe.

Mivel azonban az algoritmusok egy már meglévő rendszerbe lettek implementálva, mely rendszer kialakításakor olyan programozástechnikai gyorsításokkal lett felszerelve, amelyeket rendkívüli módon segítettek a berendezés alapú szétválasztást használó makespan minimalizáló algoritmust, ez hatással van a tesztek eredményeire is. A globális keresőfa módszer, és ehhez tartozóan a taszk alapú szétválasztás ebbe a rendszerbe történő implementálásakor viszonylag kevés hangsúlyt fektettem a kód optimalizálására. Ennek legfőbb oka, hogy egy algoritmust minőségi szinten implementálni nehéz, és rendkívül sok időt vesz igénybe. Továbbá a meglévő rendszer nagy, és egyszerre több irányból is kap kiegészítéseket, hogy más fajta feladatok megoldására is alkalmas legyen. Egy algoritmus megfelelően optimalizált módon történő implementálása azonban azt



7.1. ábra. Az első teszt példa recept-gráfja



7.2. ábra. A második teszt példa recept-gráfja

példa	idő	megoldás	ms.	a	b	c	d	e	f	g
#1	20	1,1,0,1,0 (8.5)	20	0.002 0.006	0.01 0.002	0.014 0.002	0.01 0.006	0.009 0.003	0.008 0.004	0.046
	21	1,1,0,1,0 (8.5)	20	0.009 0.01	0.013 0.01	0.009 0.005	0.01 0.006	0.012 0.007	0.01 0.008	0.044
	22	1,1,0,1,0 (8.5)	20	0.02 0.065	0.026 0.088	0.026 0.066	0.028 0.086	0.025 0.073	0.018 0.064	0.311
	23	1,1,0,1,0 (8.5)	20	0.017 0.073	0.027 0.111	0.025 0.078	0.026 0.087	0.022 0.093	0.023 0.073	0.346
	24	0,2,0,1,0 (9.5)	24	0.034 0.148	0.078 0.292	0.072 0.288	0.049 0.185	0.048 0.199	0.04 0.142	0.588
	25	0,1,0,2,0 (10)	25	0.044 0.205	0.104 0.432	0.1 0.439	0.046 0.224	0.045 0.227	0.052 0.213	0.758
	26	0,1,0,2,0 (10)	25	0.048 0.299	0.102 0.645	0.108 0.642	0.056 0.32	0.044 0.337	0.044 0.312	0.89
	27	0,1,0,2,0 (10)	25	0.047 0.402	0.106 0.825	0.106 0.828	0.054 0.468	0.044 0.435	0.05 0.42	1.035
	28	0,0,0,3,0 (10.5)	28	0.057 0.494	0.191 1.174	0.171 1.172	0.06 0.511	0.063 0.51	0.06 0.506	1.118
	29	2,1,0,2,0 (14)	29	2.102 8.435	3.47 13.595	4.617 12.231	2.541 10.913	4.448 10.137	2.246 9.28	27.335
	30	2,1,0,2,0 (14)	29	5.616 12.08	8.198 19.144	10.826 16.762	6.338 14.165	9.606 13.755	6.105 12.922	35.905
	31	2,3,0,1,0 (16.5)	31	19.299 31.528	25.353 72.105	17.608 57.264	21.987 37.02	12.66 44.033	21.413 36.353	93.586
	32	1,2,0,3,0 (18.5)	32	42.956 62.259	50.783 154.666	29.512 111.411	58.694 78.75	25.596 94.867	54.936 75.362	144.068
	33	1,2,0,3,0 (18.5)	32	64.454 99.035	71.65 250.801	30.713 161.073	85.252 129.898	26.325 127.816	80.29 121.655	300.739

7.2. táblázat. Az első teszt példa teszteredményei. Az egyes sorok az adott időkorlátokkal történő futtatás futási idejeit tartalmazzák. Az idő oszlopban látható az időkorlát, az a-g oszlopok pedig az egyes algoritmusok futási idejeit tartalmazzák.

	A(3.7)	B(4.5)
1	R1(2)	R1(2)
	R2(2)	R2(2)
2	R3(3)	R3(3)
	R4(3)	R4(3)
3	R3(1)	R3(1)
	R4(1)	
4	E2(1)	R4(1)
5	E1(3)	E2(1)
6		E2(1)
7		E1(3)

Berendezés	R1	R2	R3	R4	E1	E2
Darabszám	1	1	1	1	2	3

7.3. táblázat. A második teszt példa receptje

igénylené, hogy a rendszer már meglévő részeiben is történjenek változások, ami két okból is nehéz feladat. Az egyik, hogy mivel a programot egyszerre többen fejlesztjük, nem célszerű olyan részhez hozzányúlni, ami mások munkájához is kell. A másik ok az, hogy hibamentesen implementálni a változtatásokat szintén nehéz tekintve a meglévő forráskód méretét. Egy ilyen beavatkozás tehát hosszabb tervezést és kísérletezést igényel, mielőtt véglegesítésre kerül, ehhez pedig időre van szükség. A jövőben mindenképpen szeretnék azzal foglalkozni, hogyan lehetne az algoritmust implementálási szinten gyorsítani.

Viszont ha csupán az algoritmusok hatékonyságát szeretnénk vizsgálni, akkor nem árt egy olyan környezetben tesztelni, ahol a programozástechnikai megoldásokból adódóan egyik algoritmusnak sincsen előnye.

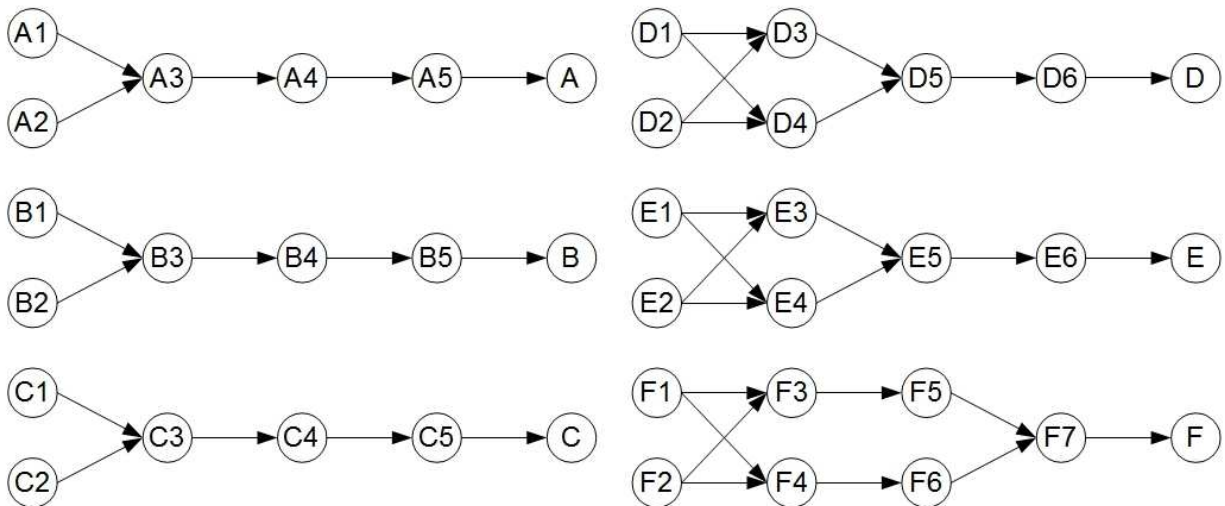
példa	idő	megoldás	ms.	a	b	c	d	e	f	g
#2	12	1,1 (8.2)	11	0.034	0.028	0.029	0.025	0.026	0.027	
				0.092	0.087	0.089	0.085	0.092	0.01	0.622
	13	1,1 (8.2)	11	1.147	1.148	1.141	1.147	1.154	0.587	
				0.375	0.378	0.377	0.375	0.374	0.021	2.542
	14	4.0 (14.8)	14	88.422	89.703	89.404	88.612	89.079	89.171	
				16.617	16.306	16.345	19.108	19.018	16.274	75.373
	15	3,1 (15.6)	15	132.528	133.365	133.079	131.578	133.482	132.202	
				87.392	82.636	82.701	176.632	176.262	82.641	373.582

7.4. táblázat. A második teszt példa teszteredményei

	A(53.75)	B(71.66)	C(86.0)	D(89.58)	E(114.66)	F(139.75)
1	H1(1)	H1(1)	H1(1)	H1(1)	H1(1)	H1(1)
2	R1(2) R2(2)	R1(2) R2(2)	R1(2) R2(2)	R1(2)	R2(2)	R2(2)
3	R1(2)	R2(2)	R2(2)	R2(2)	R2(2)	R2(2)
4	R1(1) R2(1)	R1(1)	R1(1)	R1(2)	R1(2)	R1(2)
5	S1(2)	S1(2)	S1(2)	R2(1)	R2(1)	R2(1)
6				S1(2)	S1(1)	R1(1)
7						S1(2)

Berendezés	R1	R2	H	S
Darabszám	1	1	1	1

7.5. táblázat. A harmadik teszt példa receptje



7.3. ábra. A harmadik teszt példa recept-gráfja

példa	idő	megoldás	ms.	a	b	c	d	e	f	g
#3	12	2,0,0,0,0,0 (279.50)	12	0.045 0.344	0.065 2.106	0.069 2.124	0.045 0.708	0.049 0.708	0.041 0.344	0.195
	13	2,0,0,0,0,0 (279.50)	12	0.365 0.84	0.716 3.908	0.72 3.907	0.439 2.046	0.436 2.056	0.362 0.837	0.336
	14	1,1,0,0,0,1 (308.16)	14	1.328 1.406	2.032 5.815	2.007 5.792	1.244 6.13	1.496 3.22	0.943 2.279	0.736
	15	1,0,0,2,0,1 (365.50)	15	6.352 8.131	12.28 33.164	9.195 15.26	8.386 28.883	4.598 5.036	4.705 20.812	1.6
	16	1,1,0,1,0,1 (394.16)	16	29.069 30.081	49.424 109.667	44.788 103.909	24.553 335.963	28.467 67.251	13.741 95.509	3.829
	17	3,0,0,0,0,0 (419.25)	17	169.263 371.053	408.905 2229.85	403.283 2230.81	176.342 992.524	176.528 993.032	170.494 371.053	10.419

7.6. táblázat. A harmadik tesztpélda teszteredményei

Ezért az összes fent tárgyalt algoritmust implementáltam egy különálló programban, teljesen az elejétől kezdve minden részletet, mellőzve mindenféle gyorsító mechanizmust. Mivel a globális keresőfa a taszk alapú szétválasztásra épül, és az újonnan implementált program teljes egészében ugyanazt az elvet használja a kereséshez, az összehasonlító tesztek eredménye mérvadóbb lehet, mint az előző esetben. Az előző tesztekhez használt programban ugyanis a taszk alapú megvalósíthatósági teszt esetében is működött olyan heurisztika, ami hatott a megvizsgált állapotok sorrendjére, míg a globális keresőfa algoritmus kizárólag mélységi keresés alapján dolgozott. Az újonnan elkészített programban viszont a megvalósíthatósági tesztek és a globális keresőfa is mélységi alapon működnek, tehát ilyen szempontból sem lehet eltérés.

A programba implementálásra kerül a berendezés alapú szétválasztási eljárás is, de mivel erősen függ a feladat felépítésétől, hogy melyik szétválasztási eljárás hatékonyabb, ezért a berendezés alapú szétválasztással futtatott tesztek csak érdekességgéppen szerepelnek a táblázatokban. Néhány helyen az így kapott futtatási idők helyett - jel szerepel, ami azt jelenti, hogy az adott esetre nem is történt futtatás, de feltehetően a futási idő jóval több lett volna, mint fél óra.

A összehasonlítás érdekes része tehát a taszk alapú szétválasztást használó megvalósíthatósági teszt algoritmusok és a globális keresőfa között van. A futási idők a 7.7, 7.8 és 7.9 táblázatokban láthatók. Ezeket a táblázatok figyelembe véve azt lehet mondani, hogy a globális keresőfa módszer egyenlő feltételek mellett valóban hatékonyabbnak bizonyult, mint a megvalósíthatósági tesztet futtató algoritmusok. Ez alapján kijelenthető, hogy a globális keresőfa módszer tényleg hatékony, viszont az implementálás minősége is nagyon sokat számít. Ahogyan azt lehet látni a táblázatok alapján, a berendezés alapú szétválasztást alkalmazó algoritmusok az új programban sokkal lassabbak voltak, mint a másokban, hiszen nem volt akkora

példa	idő	megoldás	ms.	a	b	c	d	e	f	g
#1	20	1,1,0,1,0 (8.5)	20	0.011 0.012	0.017 0.015	0.022 0.027	0.013 0.013	0.021 0.013	0.016 0.01	0.023
	21	1,1,0,1,0 (8.5)	20	0.03 0.017	0.032 0.022	0.035 0.023	0.027 0.014	0.034 0.02	0.028 0.015	0.023
	22	1,1,0,1,0 (8.5)	20	0.062 0.317	0.114 0.471	0.066 0.344	0.121 0.404	0.072 0.351	0.064 0.322	0.202
	23	1,1,0,1,0 (8.5)	20	0.063 0.334	0.11 0.517	0.07 0.393	0.12 0.427	0.072 0.392	0.063 0.346	0.237
	24	0,2,0,1,0 (9.5)	24	0.177 0.724	0.503 1.434	0.501 1.431	0.288 0.909	0.25 0.908	0.209 0.726	0.425
	25	0,1,0,2,0 (10)	25	0.249 0.911	0.681 2.205	0.68 2.085	0.277 0.936	0.202 0.958	0.285 0.953	0.581
	26	0,1,0,2,0 (10)	25	0.231 1.145	0.72 2.646	0.682 2.538	0.242 1.194	0.19 1.307	0.273 1.214	0.782
	27	0,1,0,2,0 (10)	25	0.249 1.515	0.882 3.609	0.693 3.308	0.232 1.573	0.208 1.554	0.279 1.551	0.916
	28	0,0,0,3,0 (10.5)	28	0.422 1.952	1.711 5.75	1.471 5.087	0.376 1.896	0.386 1.908	0.387 1.901	1.054
	29	2,1,0,2,0 (14)	29	37.778 104.459	50.115 137.08	78.977 120.087	39.001 119.805	73.772 94.9	37.924 105.953	22.172
	30	2,1,0,2,0 (14)	29	83.868 128.273	103.145 175.889	179.475 150.834	86.675 142.81	159.134 119.39	85.351 128.205	30.897

7.7. táblázat. Az első teszt példa teszteredményei a másik programban implementálva az algoritmusokat

támogatottságuk, de a taszk alapú algoritmusok is az esetek nagy részében jelentősen lassabbak voltak. Ezért a későbbiekre nézve két módon is lehet gyorsítani az globális keresőfa módszeren. Egyrészt lehet heurisztikákat kitalálni. Másrészt az implementálásban is lehet változtatni a hatékonyság növelése érdekében.

A tesztek 2,3 GHz-es Intel core i5 2410M Sandy Bridge processzoron lettek futtatva, Linux Mint 12 operációs rendszer alatt.

példa	idő	megoldás	ms.	a	b	c	d	e	f	g
#2	12	1,1 (8.2)	11	0.386 0.716	0.255 0.741	0.249 0.725	0.265 0.941	0.235 0.683	0.282 0.349	0.658
	13	1,1 (8.2)	11	30.354 3.233	36.258 5.493	37.353 3.272	24.823 3.29	36.69 3.138	15.942 2.863	2.64

7.8. táblázat. A második tesztpélda teszteredményei a másik programban implementálva az algoritmusokat

példa	idő	megoldás	ms.	a	b	c	d	e	f	g
#3	12	2,0,0,0,0,0 (279.50)	12	11.336 0.312	11.406 0.602	11.562 0.576	11.16 0.304	11.202 0.301	10.471 0.299	0.204
	13	2,0,0,0,0,0 (279.50)	12	349.473 0.753	536.224 1.248	529.605 1.235	363.371 0.861	364.6 0.855	357.54 0.754	0.355
	14	1,1,0,0,0,1 (308.16)	14	1784.31 1.759	2592.25 2.475	2550.59 2.451	1716.34 1.788	1741.05 1.865	1601.87 1.573	0.768
	15	1,0,0,2,0,1 (365.50)	15	- 4.287	- 6.114	- 5.816	- 3.771	- 4.483	- 3.033	1.645
	16	1,1,0,1,0,1 (394.16)	16	- 13.16	- 17.999	- 17.987	- 12.621	- 14.717	- 9.757	3.675
	17	3,0,0,0,0,0 (419.25)	17	- 29.065	- 46.585	- 47.186	- 31.643	- 31.891	- 28.412	8.727
	18	1,0,0,3,0,1 (451.50)	18	- 60.709	- 95.255	- 94.717	- 66.775	- 68.223	- 55.748	17.525

7.9. táblázat. A harmadik tesztpélda teszteredményei a másik programban implementálva az algoritmusokat

8. fejezet

Összefoglalás

A dolgozatban röviden ismertettem a throughput maximalizálási feladatot, illetve az S-gráf módszertant. Bemutattam az S-gráf segítségével történő throughput maximalizálás alapelvét, mely megvalósíthatósági tesztek szisztematikus futtatásán alapul. Leírtam már korábban létező heurisztikákat a megvalósíthatósági teszttel történő megoldás során keletkező keresési tér bejárására. Részletesebben ismertettem továbbá egy új módszert, mellyel throughput maximalizálási feladatokat lehet megoldani S-gráf segítségével. A bemutatott algoritmusokat és az új algoritmust is implementáltam egy létező, robosztus keretrendszerbe.

Az implementált algoritmusok hatékonyságát három példán keresztül teszteltem, és hasonlítottam össze. A megvalósíthatósági teszten alapuló algoritmusoknál külön teszteket futtattam taszk és berendezés alapú szétválasztás esetére is. A tesztek alapján azt a következtetést lehet levonni, hogy egyik algoritmus sem egyértelműen jobb, mint a többi, hiszen példánként változó volt, éppen melyik algoritmus futott le kiemelkedően kevés vagy sok idő alatt. Olyan kijelentést azonban lehet tenni, hogy a berendezés alapú szétválasztás több esetben bizonyult erősebbnek a taszk alapúnál, mint gyengébbnek. Továbbá a globális keresőfa módszer a már meglévő rendszerben implementálva, támogatás híján lassabbnak bizonyult a többi algoritmusnál. Új programként megírva azonban az algoritmusokat, ahol mindegyik egyenlő esélyekkel indul már a globális keresőfa bizonyult a leghatékonyabbnak.

Végző következtetésként tehát a tesztek alapján azt lehet levonni, hogy a globális keresőfa módszer hatékony ötlet, ami tényleg redukálja a kereséshez szükséges számítás mennyiségét a redundáns lépések csökkentésével. Azonban igazán hatékony csak akkor lehet az algoritmus, ha megfelelő mennyiségű idő ráfordításával, színvonalasan is implementálásra kerül. Illetve több lehetőség is van heurisztikákkal való kiegészítésre.

Irodalomjegyzék

- [1] E. Kondili, C.C. Pantelides, and R.W.H. Sargent. A general algorithm for short-term scheduling of batch operations–i. milp formulation. *Computers & Chemical Engineering*, 17(2):211–227, February 1993.
- [2] C. C. Pantelides. Unified frameworks for optimal process planning and scheduling. In Rip pin D. W. T., Hale J. C., and Davis J., editors, *Proceedings of the second international conference on foundations of computer-aided process operations*, pages 253–274, 1993.
- [3] Thokozani Majozi and X. X. Zhu. A novel continuous-time milp formulation for multipurpose batch plants. 1. short-term scheduling. *Industrial & Engineering Chemistry Research*, 40(25):5935–5949, 2001.
- [4] Christodoulos A. Floudas and Xiaoxia Lin. Continuous-time versus discrete-time approaches for scheduling of chemical processes: a review. *Computers & Chemical Engineering*, 28(11):2109–2129, October 2004.
- [5] E. Sanmarti, F. Friedler, and L. Puigjaner. Combinatorial technique for short term scheduling of multipurpose batch plants based on schedule-graph representation. *Computer Aided Chemical Engineering*, 22:S847–850, 1998.
- [6] E. Sanmarti, Tibor Holczinger, Louis Puigjaner, and Ferenc Friedler. Combinatorial framework for effective scheduling of multipurpose batch plants. *AIChE Journal*, 48(11):2557–2570, 2002.
- [7] J. Romero, L. Puigjaner, T. Holczinger, and F. Friedler. Scheduling intermediate storage multipurpose batch plants using the s-graph. *AIChE Journal*, 50(2):403–417, 2004.
- [8] T. Holczinger, J. Romero, L. Puigjaner, and F. Friedler. Scheduling of multipurpose batch processes with multiple batches of the products. *Hung. J. Ind. Chem.*, 30:305–312, 2002.
- [9] Thokozani Majozi and Ferenc Friedler. Maximization of throughput in a multipurpose batch plant under fixed time horizon: S-graph approach. *Industrial & Engineering Chemistry Research*, 45:6713–6720, 2006.
- [10] Tibor Holczinger, Thokozani Majozi, Mate Hegyháti, and Ferenc Friedler. An automated algorithm for throughput maximization under fixed time horizon in multipurpose batch plants: S-graph approach. In Valentin Pleşu and Paul Şerban Agachi, editors, *17th European Symposium on Computer Aided Process Engineering*, volume 24 of *Computer Aided Chemical Engineering*, pages 649 – 654. Elsevier, 2007.

- [11] Robert Adonyi. *Batch process scheduling with the extensions of the S-graph framework*. PhD thesis, Doctoral School of Information Science and Technology - University of Veszprem, 2008.