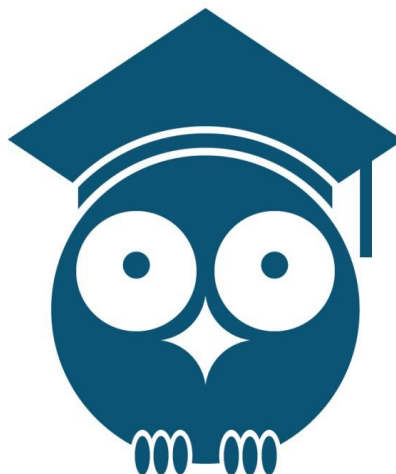




**SZÉCHENYI**  
ISTVÁN  
**EGYETEM**



**S-gráf alapú ütemező  
algorithmus  
párhuzamos  
hozzárendelést  
megengedő  
feladatokhoz**



Tudományos és Művészeti Diákkör 2019.

Molnár Gergő, Mérnökinformatikus Bsc. VI. évfolyam

Témavezető: Dr. Hegyháti Máté, tudomány főmunkatárs

Széchenyi István Egyetem,  
Gépészmérnöki, Informatikai és Villamosmérnöki Kar  
Informatika Tanszék

# Tartalomjegyzék

<b>1. Bevezetés</b>	<b>1</b>
<b>2. Irodalmi áttekintés</b>	<b>3</b>
2.1. Ipari ütemezési feladatok . . . . .	3
2.2. Megoldó módszerek az irodalomban . . . . .	7
2.2.1. MILP modellek . . . . .	7
2.2.2. Analízis alapú eszközök . . . . .	8
2.3. S-gráf módszertan . . . . .	8
2.3.1. A makespan minimalizálás algoritmusai . . . . .	12
2.3.2. Throughput maximalizálás . . . . .	15
<b>3. Probléma definíció</b>	<b>18</b>
<b>4. Az új módszer</b>	<b>22</b>
4.1. Vezérlő . . . . .	23
4.2. Maxprofit eljárás . . . . .	24
4.3. Profitbound eljárás . . . . .	27
<b>5. Implementálás</b>	<b>29</b>
5.1. S-gráf solver . . . . .	29
5.1.1. A megoldó működése . . . . .	30
5.2. Adatok beolvasása . . . . .	31
5.2.1. Bemeneti fájl . . . . .	31
5.2.2. Beolvasó függvény . . . . .	32

5.3. FlexBatchSchProblem osztály . . . . .	33
5.3.1. MakeDecisions függvény . . . . .	34
5.3.2. Branching függvény . . . . .	35
5.3.3. További metódusok . . . . .	35
5.4. SGraph osztály . . . . .	37
5.4.1. MakeMultipleBatches függvény . . . . .	37
5.4.2. UpdateProfitBound függvény . . . . .	39
5.4.3. UpdateProfitBoundFromTask függvény . . . . .	40
5.4.4. Egyéb új metódusok . . . . .	41
5.5. Argumentum hozzáadás . . . . .	42
5.6. Megoldás fájlba írása . . . . .	42
<b>6. Tesztelés</b>	<b>44</b>
6.1. A tesztfeladat megoldása . . . . .	46
6.2. A régi és az új megoldó összehasonlítása . . . . .	50
6.3. A megoldó módszerek sebessége . . . . .	52
<b>7. Összefoglalás</b>	<b>54</b>
<b>A. Tesztesetek</b>	<b>57</b>
<b>B. CD Melléklet tartalma</b>	<b>59</b>

# Ábrák jegyzéke

2.1. Különböző receptek szemléltetése blokkdiagramon . . . . .	5
2.2. A receptgráf szemléltetése . . . . .	9
2.3. Az ütemezési gráf szemléltetése . . . . .	10
2.4. E2-es berendezés által elvégzett részfeladatok . . . . .	11
2.5. Egy ütemezés Gantt diagramon való megjelenítése . . . . .	11
2.6. Throughput maximalizálás szemléltetés . . . . .	16
3.1. Kondili példafeladata . . . . .	19
5.1. Példafeladat . . . . .	38
6.1. Tesztfeladat . . . . .	46
6.2. A tesztfeladat bemeneti adatai . . . . .	47
6.3. A megoldást tartalmazó fájl első része . . . . .	48
6.4. A megoldást tartalmazó fájl második része . . . . .	48
6.5. A megoldást tartalmazó fájl harmadik része . . . . .	49
6.6. A solver által elkészített Gantt diagram . . . . .	49
6.7. Régi módszer esetén a B termékhez tartozó 4 gráf . . . . .	50
6.8. A régi megoldó bemeneti adatai . . . . .	51
6.9. A régi megoldó által nyújtott Gantt diagram . . . . .	52
A.1. Nyolcadik teszteset bemeneti adatai . . . . .	57

# Táblázatok jegyzéke

3.1. A 27 rögzített recept Kondili példájához . . . . .	20
3.2. Nem dominált esetek bevétel szerint növekvő sorrendben . . . . .	21
3.3. Összevont, nem dominált esetek bevétel szerint növekvő sorrendben . . . .	21
5.1. A taszkokhoz tartozó azonosítók . . . . .	39
6.1. Különböző módszerekkel megoldott feladatok összehasonlítása . . . . .	53
A.1. Második példafeladat tesztéseinek második fele . . . . .	58

# Algoritmusok jegyzéke

1.	A makespan minimalizálás pszeudó kódja . . . . .	14
2.	Az algoritmus pszeudó kódja . . . . .	15
3.	A vezérlő pszeudó kódja . . . . .	24
4.	Párhuzamos taszkvégrehajtást megvalósító algoritmus . . . . .	26
5.	A profitbound függvény pszeudó kódja . . . . .	28

# 1. fejezet

## Bevezetés

Az ütemezés feladatával az élet számos területén találkozunk, kezdve az egyszerű, hétköznapi problémáktól, mint például egy napon elvégzendő feladataink sorrendjének beosztása, professzionális sportcsapatok heti edzésprogramjának kialakításán át, egészen az ipari üzemek működéséig, ahol a rendelkezésre álló berendezésekhez kell rendelni az előállítani kívánt termékeket. Bár az életünk különböző területein fellépő ütemezési problémák különböznek egymástól, bizonyos mértékben hasonlóság is fellelhető közöttük. Eltérés lehet az ütemezési feladatok célfüggvénye, valamint az adott ütemezési probléma lehet online, offline, illetve sztochasztikus vagy determinisztikus. Minden fellépő probléma esetében az a cél, hogy az elvégzendő feladatokat a rendelkezésre álló erőforrások között megosszuk oly módon, hogy adott intervallumon belül a lehető legjobb megoldást kapjuk. Fontos az, hogy ezt úgy tegyük meg, hogy a folyamat során fellépő korlátokat betartjuk, azokat nem sértjük meg. Az ipari ütemezés két leggyakoribb célja a makespan minimalizálás, és a throughput maximalizálás. A szakirodalomban legtöbb esetben a *makespan* kifejezést az idő minimalizálására, a *throughput* kifejezést pedig a termelés során előállított mennyiség maximalizálására alkalmazzák.

Az ipari gyártási folyamatok ütemezésére különböző módszerek léteznek már. Ezek közé tartoznak a MILP (Mixed Integer Linear Programming - vegyes egészértékű lineáris programozás) megoldó módszerek, amelyek a lineáris programozáson alapulnak. Továbbá ide sorolható az időzített automaták, Petri hálók, valamint a S-gráf megoldó módszer, amely a munkám során a legfontosabb szerepet tölti be a felsorolt megoldó módszerek

közül. Dolgozatom második fejezetében ezek a módszerek kerülnek bemutatásra. A harmadik részben a problémát definiálása történik meg. A negyedikben az általam megvalósított módszer elmélete található. Az ötödik fejezetben a módszer megvalósítását és szoftverbe való beillesztését mutatom be. A hatodik fejezetben tesztelésről, és az eredmények összehasonlításáról lesz szó. A dolgozat végén található az összefoglalás a munkámról, a hivatkozások, valamint a függelék.



## 2. fejezet

# Irodalmi áttekintés

### 2.1. Ipari ütemezési feladatok

Az ipari folyamatok közé tartozó tevékenységeknek egy jelentős halmazát gyártásnak nevezzük, amely során az elkészítendő termék létrehozása, megvalósítása a feladat. Ehhez szükség van arra, hogy megfelelően vegyük igénybe a rendelkezésre álló erőforrásokat, amelyeket berendezéseknek, unitnak nevezünk a gyártási feladatok során. A folyamat során fellépő feladatokra a taszk elnevezés is használható. Az ütemezés az a folyamat, amely során különböző események sorrendjét határozzuk meg. Az ütemezési feladat során kell egy feladatot, munkát vagy tevékenységet egy tervbe beilleszteni, vagyis meghatározni annak végrehajtási időpontját. Minden ütemezési feladat rendelkezik végrehajtási idővel, ami megmutatja mekkora időtartam alatt valósítható meg. Ezenfelül lehet még a feladatoknak olyan időkorlátja, ami alatt kötelező elvégezni a feladatot, ezt időhorizontnak, time horizonnak hívjuk. A problémák kimenetelük szerint lehetnek megvalósíthatatlan (infeasible) és megvalósítható (feasible) feladatok. Egy feladat csak abban az esetben minősül megvalósíthatónak, ha minden korlátozásnak megfelel. Ha már akár csak egy korlátozással szemben nem bizonyul elfogadhatónak, akkor infeasible feladatról beszélhetünk.

Az ipari folyamatokat többféleképpen lehet csoportosítani. Az egyik felosztási módszer szerint folyamatos és szakaszos üzemű rendszerek csoportjára bontjuk őket. Az első típusban az anyag folyamatosan kerül a rendszerbe, a másodikban pedig ez a folyamat lépésekben valósul meg. A munkám az utóbbi típusba tartozó feladatokra koncentrál.

Másik lehetséges felosztás az, amikor online, offline, és semi-offline kategóriákba vannak a feladatok besorolva. Az offline esetben minden szükséges bemeneti adat rendelkezésre áll az optimalizálás idejében. Ezzel szemben az online esetben előbb kell a döntéseket meghozni, minthogy az adott paraméterekhez tartozó értékekre fény derülne. A semi-offline a kettő közé sorolható, azaz bizonyos információk, adatok már rendelkezésre állnak, mások viszont nem.

Az ütemezési feladatok modelljét receptnek nevezzük. Egy termék receptje tartalmazza az adott recept által előállítható termék elkészítéséhez szükséges információkat [1]. Egy receptet a következő elemek közösen alkotják:

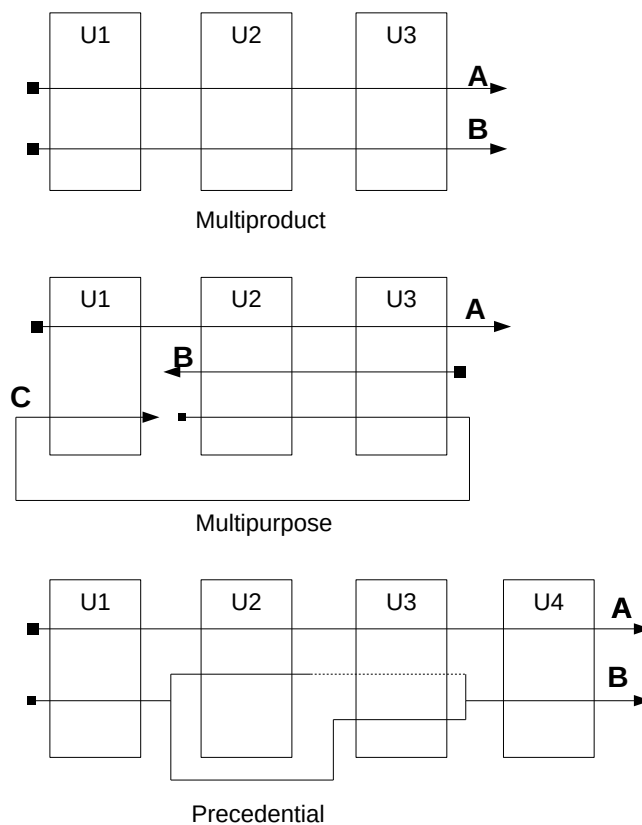
- termékek listája,
- taszkok listája, amelyek adott sorrendben történő elvégzése szükséges a termék előállításához,
- termeléshez kapcsolódó taszkok sorrendje,
- rendelkezésre álló berendezések,
- a lehetséges taszk-berendezés párokhoz tartozó feldolgozási idő.

A recepteket a feladatok precedenciája szerint az alábbi csoportokba lehet besorolni. A felsorolás a legegyszerűbbtől halad az általánosabb felé. Minden osztály a következőnek egy speciális esete.

- **Single Stage:** Egy lépésben állítható elő minden egyes termék.
- **Simple Multiproduct:** Minden terméket meghatározott számú fázison, szakaszon keresztül lehet elkészíteni. Előzővel szemben itt már nem csak egy lépésben lehetséges. Különösen fontos, hogy nem lehet elágazás benne, azaz csak szekvenciális lehet.
- **General Multiproduct:** Előzővel összehasonlítva az a különbség, hogy ennél lehetséges lépések kihagyása.

- **Multipurpose:** Megegyezik a Multiproduct esettel, azzal a különbséggel, hogy a berendezést nem azonos sorrendben kerülnek igénybe vételre.
- **Precedential:** Egy termék gyártása nem szükségszerűen lineáris, lehetnek elágazások, kör azonban nem megengedett. Minden taszk előfeltételét be kell fejezni mielőtt az adott lépés megkezdődik.
- **General Network:** A legáltalánosabb recept, ahol a taszkok a bemenetük és a kimenetük által adottak. Ilyen esetben kör is lehetséges.

Néhány előbb említett feladattípus szemléltetése látható a 2.1 ábrán. A receptek jobb oldalán lévő, T-vel jelölt kör reprezentálja a terméket, a többi pedig a taszkokat.



2.1. ábra. Különböző receptek szemléltetése blokkdiagramon

A vegyipari, gyártási ütemezési feladatoknál nagy szerepet játszik a tárolási irányelv, amely azt mutatja meg, hogy két egymást követő feladatok között az elkészített köztes termékeket, hogyan kell raktározni, tárolni, illetve ez mennyi ideig lehetséges. A tárolási irányelvek csoportosítására többféle lehetőség van. Egyik ezek közül, amikor az adott létesítmény infrastrukturális képességei korlátozzák az anyag mennyiségét és tárolásának módját.

- **UIS - Unlimited Intermediate Storage**
- **FIS - Finite Intermediate Storage**
- **NIS - No Intermediate Storage**

Az UIS eset a legmegengedőbb. Ebben az esetben van lehetőség a köztes anyagok bármely mértékű tárolására. FIS esetben van lehetőség a tárolásra, de csak korlátozott mennyiségben. A NIS esetében nincs külön tárolásra alkalmas egység, de az megoldható, hogy amíg a következő feldolgozó egységhez kerül, addig az előző tászk feldolgozó egységében várakozzon.

A tárolás második fajta csoportosításának alapját az idő adja, amely a köztes termék kémiai és fizikai tulajdonságait befolyásolhatja. Például a termék szavatossága, hőmérséklete megfelelő maradjon a következő részfeladat elkezdéséig.

- **UW - Unlimited Wait**
- **LW - Limited Wait**
- **ZW - Zero Wait**

ZW esetben nincs lehetőség a köztes anyag tárolására, azaz ha a berendezés befejezte a munkát, akkor azonnal folytatni kell a gyártást. Az LW esetben van egy idő, amíg a köztes termék várakozhat. Azonban, ha ez a rendelkezésre álló idő elfogy, akkor muszáj folytatni a gyártás folyamatát. Az UW eset a legmegengedőbb mind közül, ugyanis ha a köztes anyag tulajdonságai lehetőséget biztosítanak, akkor a tárolási idő nincs korlátozva, bármennyi ideig lehetőség van a tárolásra, raktározásra.

## 2.2. Megoldó módszerek az irodalomban

Az ütemezési feladatok megoldására számos megoldó módszer létezik. Ezek közül a legismertebbek, és legszélesebb körben elterjedt módszerek kerülnek bemutatásra a dolgozatom következő pontjaiban.

### 2.2.1. MILP modellek

Az egyik legszélesebb körben elterjedt modell a **Mixed Integer Linear Programming**, azaz a vegyes egészértékű lineáris programozás. Az ilyen modellekben vegyesen fordulnak elő folytonos és egész változók. Több altípus létezik:

**Időfelosztásos modellek - Time discretization based:** A módszer időpontokat és időréseket határoz meg. Ezek a modellek jelentek meg legkorábban kronológiailag [2]. Az időrésen és az időponton alapuló megközelítések sok hasonlóságot mutatnak, mivel egy időintervallumtól egy másikig terjedő időintervallumot tekinthetünk időrésnek [3]. Ellenkező irányból nézve pedig egy időrés kezdő időpontját tekintjük egy időpontnak.

Minden időpontban bináris változók vannak hozzárendelve a feladatokhoz aszerint, hogy az adott időpillanatban elkezdődik a feladat végrehajtása vagy sem. A bináris változók száma arányos lesz a kiválasztott időpontok számával, ezért a megoldáshoz szükséges idő nagy mértékben függ az időpontok számától. Mindig megvolt a szándék olyan módszer kifejlesztésére, amelyben a szükséges időpontok száma minél kisebb legyen amellet, hogy megtalálja az optimális megoldást. Létrejöttek jobb modellek, azonban készültek olyanok is, amelyek kevésbé voltak átláthatóak, a korlátozások még bonyolultabbá váltak, és modellezési hibák is előfordultak.

**Precedencia alapú modellek - Precedence based:** Ezeknél a módszereknél, szemben az időfelosztásos módszerekkel, nincs szükség az időhorizont diszkrétizációjára, azaz nem használnak ismeretlen paramétert a modellben. Általánosságban jobb számítási eredményeket nyújtanak az általuk kezelt problémákra, azonban ez a készlet sokkal kisebb, mint az időfelosztásos modellekhez tartozó kollekció. Alap-

vetően a multiproduct és multipurpose receptek esetében használható megfelelően, de kibővíthető, hogy a sokkal általánosabb precedential receptek is megoldhatók legyenek ezek segítségével.

Ez a módszer kétféle bináris változót használ. Az első  $Y_{i,j}$ , aminek az értéke abban az esetben lesz 1, ha  $i$  feladatot  $j$  berendezés végzi el. A második változó:  $X_{i,j,i'}$ . Értéke akkor lesz 1, ha ugyanaz a berendezés végzi el az  $i$  és  $i'$  taszktot, mégghozzá úgy, hogy előbb az  $i$ -t teljesíti.

### 2.2.2. Analízis alapú eszközök

Az automatákat és Petri hálókat széles körben alkalmazzák diszkrét eseményű rendszerek modellezésére [4]. Számos kísérletet tettek ezen eszközök modellezési teljesítményének kiterjesztésére annak érdekében, hogy batch folyamatok ütemezésére is alkalmassá tegyék az említett eszközöket. A meglévő modelleket időzítéssel egészítették ki, így jöttek létre a Timed Place Petri Nets (TPPN) and Timed Priced Automata (TPA) módszerek, amelyek Branch and Bound algoritmust használnak azért, hogy a legelőnyösebb megoldást megtalálják. Ezen módszereknek a hatékonysága elmarad a MILP és az S-gráf modell hatékonyságától is.

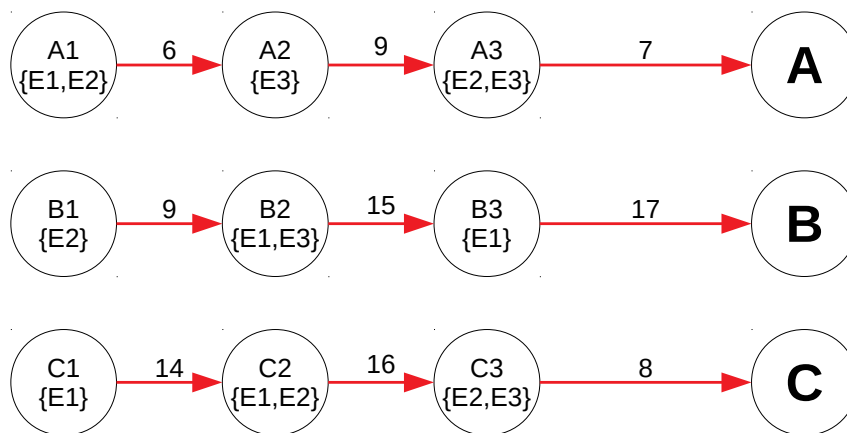
**Időzített automaták:** Ezekben a megközelítésekben a recepteket és a berendezéseket külön modellezzik, és a rendszer modellje ezeknek a párhuzamos összetételével jön létre. A bonyolultságot az jelenti, hogy az órák állapota végtelen lehet, és emiatt a rendszer állapotterülete is az lehet.

**Időzített Petri háló:** Az alap ilyen módszereknél, hogy az átvitel jele késleltetés alapján jön létre. Többben is foglalkoztak a témával, például Ghaeli [5], aki batch folyamatok ütemezésével tanulmányozta.

## 2.3. S-gráf módszertan

Az S-gráf keretrendszer volt az első olyan publikált módszer, amely gráf elméleten alapult, valamint szakaszos gyártórendszerek ütemezési problémáinak megoldására szolgált [6]. Ez

a keretrendszer egy irányított gráf modellből, az S-gráfból, és a hozzá tartozó algoritmusokból áll [7]. Az S-gráf egy speciális irányított gráf, amely ütemezési problémák számára lett létrehozva. Nemcsak a recept vizualizációja, hanem egyben matematikai modell is. A keretrendszerben az S-gráf reprezentálja a recepteket, a részleges és a teljes ütemterveket is. Ezekben a gráfokban a termékeket és a feladatokat csúcsok jelölik, amelyeket csomópontoknak (node) nevezünk. Az ütemezési döntés nélküli S-gráfot **Receptgráfnak** nevezzük. Erre példa a 2.2 ábrán látható [1].



2.2. ábra. A receptgráf szemléltetése

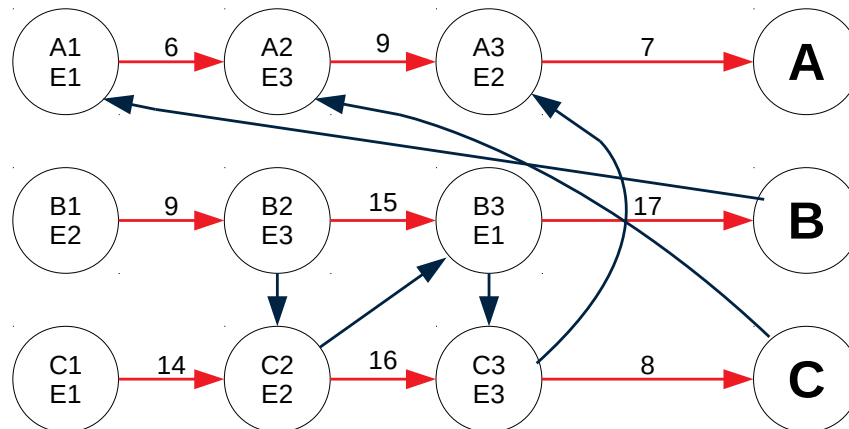
A jobb oldalon látható három, nagybetűvel jelölt csomópont felel meg a termékeknek, a maradék kilenc pedig a részfeladatokat jelenti. Ezt a kilenc részfeladatot el kell végezni a termékek előállításának érdekében. Az élek a csomópontok közti függőséget mutatják meg. Ezeket **Receptéleknek** nevezzük. Kétfajta függőséget tudunk megkülönböztetni:

- Két részfeladat között van él. Ebben az esetben az egyik készíti el a másinak a bemenetét.
- Egy termék és egy részfeladat között szerepel él. Ilyenkor a részfeladat készíti el a terméket.

Az éleken látható súlyok a részfeladat végrehajtásához szükséges időt mutatják meg. Ha

egy részfeladatot több berendezés is képes elvégezni, akkor az előbb említett súly mindig a legkisebb előállítási idő lesz.

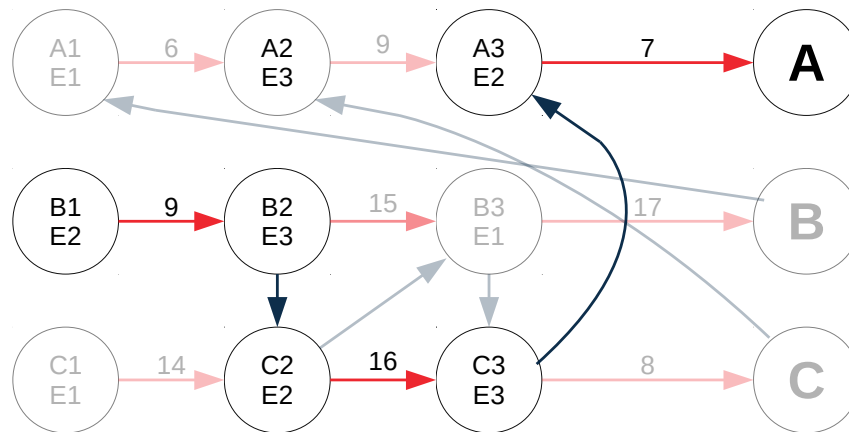
Minden S-gráf-hoz kapcsolódó algoritmus kiegészíti ezeket a gráfokat az úgynevezett **ütemezési élekkel**, amelyek az ütemezési döntést testesítik meg. Ezekkel az élekkel kiegészített gráfoknak a neve **Ütemezési gráf**. Példa a 2.3 ábrán nézhető meg. Az ábrán sötétkékkel jelölt élek az ütemezési élek. Az ütemezési élek súlya alapértelmezetten nulla, ha a probléma nem tartalmaz szállítási, vagy tisztítási időt. A részfeladatok csomópontjain már nem a lehetséges berendezések halmaza látható, hanem egy konkrét kiválasztott berendezés, az ütemezési döntésnek megfelelően.



2.3. ábra. Az ütemezési gráf szemléltetése

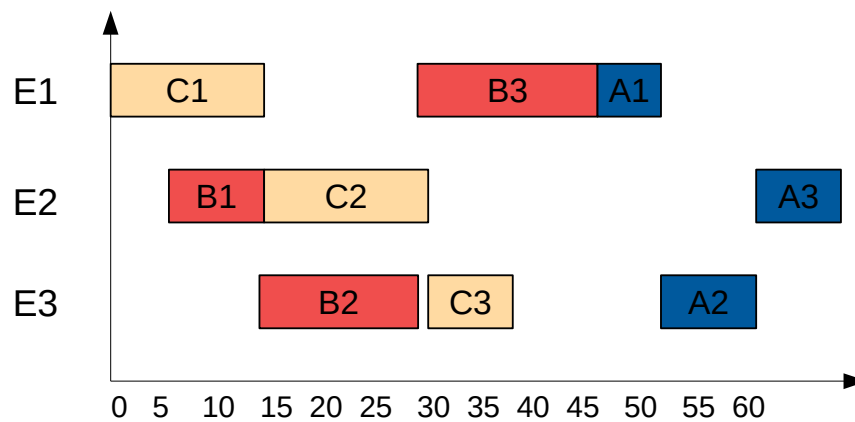
Ugyanahhoz a berendezéshez rendelt részfeladatok végrehajtási sorrendje könnyedén leolvasható a gráfról. A 2.4 ábrán látható példában az E2-es berendezés által elvégzett részfeladatok sorrendje  $B1 \rightarrow C2 \rightarrow A3$ . Az ütemezési él azt mutatja meg, hogy nem elég az, hogy az E2 berendezés végrehajtsa a B1 feladatot, hanem ezt a közbelső terméket a B2 feladathoz tartozó E3 berendezés átvegye. Csak ezeket követően tudja megkezdeni az adott részfeladat végrehajtását.





2.4. ábra. E2-es berendezés által elvégzett részfeladatok

Gyakran használt mód az ütemezési feladatok ábrázolására a Gantt diagram [8], [9]. Ezeken a diagramokon a függőleges tengelyen a berendezések, míg a vízszintes tengelyen pedig az idő szerepel. Az ábrán látható erőforrások szemléltetik az erőforrások elfoglaltságát. Egy Gantt diagram látható a 2.5 ábrán.



2.5. ábra. Egy ütemezés Gantt diagramon való megjelenítése

### 2.3.1. A makespan minimalizálás algoritmus

Az egyik legelső célja az  $S$ -gráf keretrendszer létrehozásának a makespan minimalizálás volt. Ennek alapja egy Branch & Bound algoritmus, amivel lehetséges a termékek előállítási idejét minimalizálni.

Az algoritmus első lépésben inicializálja a *makespan*<sup>cb</sup> értékét végtelennel, majd beállítja az  $S$  halmazt, amelyben az ütemezés során a nyitott részproblémák szerepelnek. Kezdetben csak a gyökér probléma szerepel benne, vagyis egy receptgráf bármilyen hozzárendelés nélkül. A **recipe** függvény visszaadja a probléma receptgráfjának modelljét, amit a  $G(N, A_1, A_2, w)$  jelöl, amiben:

- $N$  : a csomópontok halmaza (termékek és taszkok együttese)
- $A_1$  : a receptélek halmaza
- $A_2$  : az ütemezési élek halmaza (itt még üres halmaz)
- $w_{i,i'}$  : a receptélekhez tartozó súlyok, amelyek az  $i$  taszkok legkisebb feldolgozási idejét jelentik.

Az iteráció minden lépésében a **select\_remove** függvénnyel egy tetszőleges részprobléma kerül kiválasztásra, majd az  $S$ -ből eltávolításra. Ennek a függvénynek a viselkedése a különböző megvalósításokban más és más lehet, ami más keresési stratégiát eredményez. A kiválasztott részprobléma a következőképpen néz ki  $(G(N, A_1, A_2, w), I', J', \mathcal{A})$ , ahol:

- $G(N, A_1, A_2, w)$  : az ütemezési gráf
- $I'$  : a még nem ütemezett taszkok halmaza
- $J'$  : azon berendezések halmaza, amelyekhez az algoritmus még tud rendelni taszkokat
- $\mathcal{A}$  : taszk-berendezés hozzárendelések halmaza,  $(i, j)$  párok formájában

Az iteráció elején kiértékelődik, hogy a részprobléma képes-e egy optimális megoldást nyújtani vagy sem. Ez a **bound** függvénnyel történik. Ezzel szemben több követelmény

áll fenn. Alsó korlátot kell adnia azoknak a megoldásoknak, amelyek a részproblémából származnak. Biztosítania kell levél problémák pontos makespanját. Végtelen értékkel térjen vissza, ha a gráf tartalmaz kört, és jelezze, hogy megoldhatatlan. A leggyakrabban a leghosszabb út keresésével vizsgálja meg a részproblémát a **bound** függvény, de lehetséges LP alapú modellek használata is.

Ha a korlát nem kisebb, mint az eddig megtalált legjobb eredmény, akkor az iteráció véget ér, és amennyiben létezik egy következő részprobléma, akkor az kerül kiválasztásra. Ha viszont kisebb a korlát, abban az esetben ellenőrzi az algoritmus azt, hogy az összes taszk már ütemezett-e, vagyis a még nem ütemezett taszkok halmaza üres már ( $I' = \emptyset$ ). Ha így van frissül a legjobb megoldás gráfja, a legjobb makespan és a hozzárendelések halmaza. Ellenben, ha még szükséges további ütemezés, akkor a **select** függvény kiválaszt egy rendelkezésre álló berendezést a  $J'$  halmazból. A kiválasztott  $j$  berendezéshez az algoritmus hozzárendeli az összes lehetséges taszkot a még nem ütemezett taszkok és a kiválasztott berendezés által elvégezhető taszkok közös halmazából ( $i \in I_j \cap I'$ ). Ezek a kiválasztott taszkok kapnak egy másolatot az aktuális S-gráfról. Mivel a csomópontok és a receptélek halmaza nem változik, ezért ezek változatlanok maradnak, csak az ütemezési élek halmaza és súlyok változnak. Ezt a másolatot kibővíti az algoritmus az új hozzárendelés alapján az ütemezési élekkel ( $A_2^i := A_2^i \cup \{(i', i)\}$ ). Ezután minden  $i$ -ből induló receptél súlya frissül a  $t_{i,j}^{pr}$  értékével. Végezetül pedig az új részproblémát hozzáadja az  $S$  halmazhoz. Itt  $I'$  halmazból kikerül az előbb kiválasztott  $i$  taszk, és a hozzárendelések halmazába bekerül az új taszk-berendezés pár ( $\mathcal{A} \cup \{(i, j)\}$ ).

Abban az esetben, ha minden még nem ütemezett taszkot a kiválasztott  $j$  berendezésen kívül más berendezés is el tud végezni, akkor egy új részproblémát hoz létre az algoritmus, amelyben a  $j$  berendezés már nem végez több taszkot, azaz kikerül a még rendelkezésre álló berendezések halmazából ( $J' \setminus \{j\}$ ).

Ha az  $S$  halmaz üres lesz, akkor a  $G^{cb}$  gráf és a hozzárendelések a  $\mathcal{A}^{cb}$  halmazban leírják az optimális megoldást. Ha legalább egy megvalósítható, akkor az algoritmus visszatér ezzel az értékkel, ellenkező esetben nem ad vissza megoldást.

**Algoritmus 1** A makespan minimalizálás pszeudó kódja

---

```

1:  $makespan^{cb} := \infty$ 
2:  $\mathcal{S} := \{(\text{recipe}(), I, J, \emptyset)\}$ 
3: while  $\mathcal{S} \neq \emptyset$  do
4:    $(G(N, A_1, A_2, w), I', J', \mathcal{A}) := \text{select\_remove}(\mathcal{S})$ 
5:   if  $\text{bound}(G) < makespan^{cb}$  then
6:     if  $I' = \emptyset$  then
7:        $makespan^{cb} := \text{bound}(G)$ 
8:        $G^{cb} = G$ 
9:        $\mathcal{A}^{cb} := \mathcal{A}$ 
10:    else
11:       $j := \text{select}(J')$ 
12:      for all  $i \in I_j \cap I'$  do
13:         $G^i(N, A_1, A_2^i, w^i) := G(N, A_1, A_2, w)$ 
14:        for all  $i' \in \bigcup_{(i', j) \in \mathcal{A}} I_{i'}^+ \setminus \{i\}$  do
15:           $A_2^i := A_2^i \cup \{(i', i)\}$ 
16:        end for
17:        for all  $i' \in I_i^+$  do
18:           $w_{i, i'}^i := t_{i, j}^{pr}$ 
19:        end for
20:         $\mathcal{S} := \mathcal{S} \cup (G^i(N, A_1, A_2^i, w^i), I' \setminus \{i\}, J', \mathcal{A} \cup \{(i, j)\})$ 
21:      end for
22:      if  $I' \subseteq \bigcup_{j' \in J', j \neq j'} I_{j'}$  then
23:         $\mathcal{S} := \mathcal{S} \cup (G(N, A_1, A_2), I', J' \setminus \{j\}, \mathcal{A})$ 
24:      end if
25:    end if
26:  end if
27: end while
28: if  $makespan^{cb} \neq \infty$  then
29:   return  $(G^{cb}, \mathcal{A}^{cb})$ 
30: end if

```

---

### 2.3.2. Throughput maximalizálás

Eredetileg az S-gráf keretrendszer makespan minimalizációs problémák megoldására lett létrehozva, azonban a későbbiekben bővítésre került, így ezután throughput, profitmaximalizációs problémák megoldására is alkalmazhatóvá vált. Az alapötlet Majozi és Friedler [10], valamint Holczinger és társai [11] nevéhez fűződik. A termékek lehetséges batch darabszámai alapján az algoritmus konfigurációkat hoz létre. A konfiguráció tehát az, ami megmutatja, hogy egy termékből hány batch készül el. Ha  $n$  a termékek számát jelöli, akkor egy  $n$  dimenziós térben lehet elképzelni ezeket a konfigurációkat.

---

**Algoritmus 2** Az algoritmus pszeudó kódja

---

```

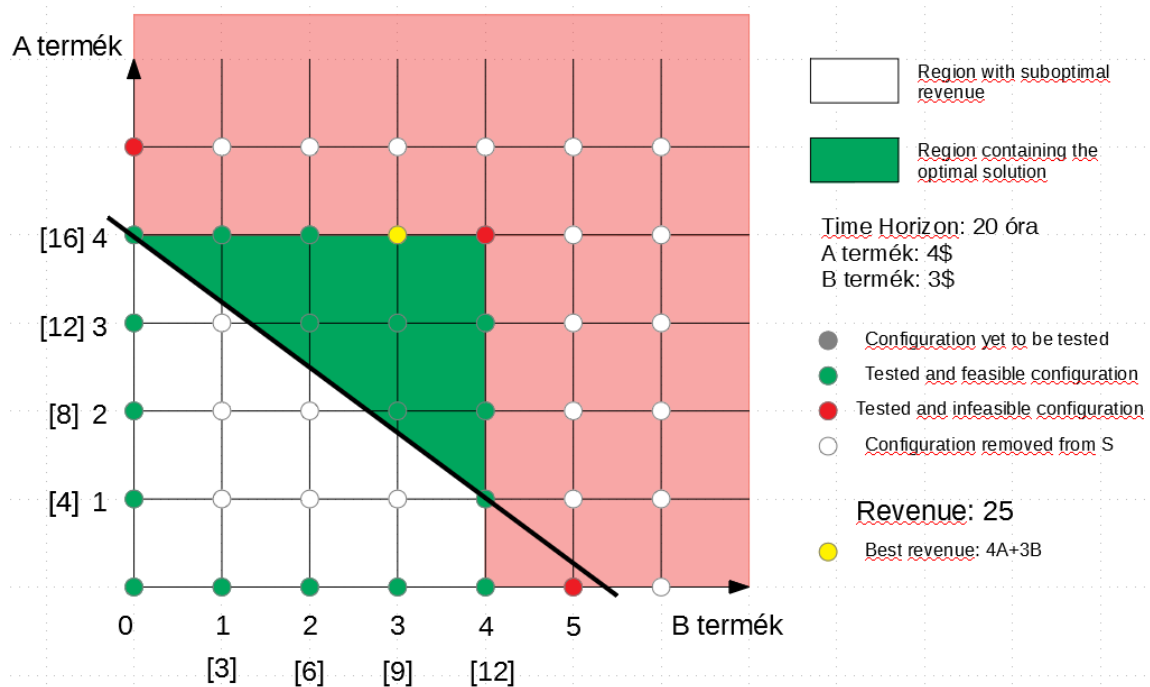
1:  $revenue^{cb} := 0$ 
2:  $\mathcal{S} := (\mathbb{Z}^*)^{|P|}$ 
3: while  $\mathcal{S} \neq \emptyset$  do
4:    $x := \text{select\_remove}(\mathcal{S})$ 
5:   if  $\text{feasible}(\text{recipe}(x), t^H)$  then
6:     if  $\text{revenue}(x) > revenue^{cb}$  then
7:        $revenue^{cb} := \text{revenue}(x)$ 
8:        $x^{cb} := x$ 
9:        $\text{update}(\mathcal{S}, revenue^{cb})$ 
10:    end if
11:  else
12:     $\mathcal{S} := \{x' \in \mathcal{S} \mid x' \not\geq x\}$ 
13:  end if
14: end while
15: if  $revenue^{cb} \neq 0$  then
16:   return  $(x^{cb}, revenue^{cb})$ 
17: end if
```

---

Az algoritmus először inicializálja az  $S$  halmazt a termékekre vonatkozóan minden lehetséges batch számmal. Fontos kiemelni, hogy ebben az esetben minden termékénél a batch méret rögzített, azaz egy termék batch-jének jövedelme ismert. Ezt követően minden iteráció során az előbb említett halmazból kiválasztásra kerül egy konfiguráció a **select\_remove** függvény segítségével. Ezután sor kerül a megvalósíthatóság tesztelésére, amely során eldől, hogy a megadott időhorizont alatt megvalósítható vagy sem. Ha meg-

valósítható és nagyobb jövedelmet biztosít, mint az eddig megtalált legnagyobb jövedelem, akkor a jelenlegi legjobb megoldás és az  $S$  halmaz frissül. Ha infeasible a kiválasztott konfiguráció, akkor ez, és minden ennél nagyobb konfiguráció eltávolításra kerül az  $S$  halmazból. Amint az  $S$  halmaz üressé válik, és volt feasible megoldás, akkor az algoritmus visszatér a legjobb konfigurációval, és az ehhez tartozó jövedelem mennyiségével.

A 2.6 ábrán látható egy Throughput módszerrel megvalósított feladat eredménye.



2.6. ábra. Throughput maximalizálás szemléltetés

Két termék esetén koordináta rendszeren szemléltethető az algoritmus működése. Egyik tengelyen az egyik termék, a másikon pedig a másik termék szerepel. Kezdetben az összes konfigurációt tartalmazta a konfigurációk halmaza. Az algoritmus először végighaladt az egyik tengely mentén, vagyis az egyik termék batch mérete 0 volt, a másik pedig növekedett. Ezt a haladási irányt addig követte amíg megtalálta az első nem megvalósítható, azaz infeasible konfigurációt. Ezt követően elvégzi a keresést a másik tengelyen is. Így kapott egy jelenleg maximális jövedelmet. Az utolsó még feasible batch számoknál nagyobb batch számokat már nem kell vizsgálni, hiszen azokat a megadott időhorizonton belül nem lehet megvalósítani. A képen látható feladatban a legnagyobb elérhető profit

16 volt. A feketével jelzett egyenes tartalmazza az összes 16 profittal rendelkező pontot. Ezt nevezzük revenue line-nak, wn a B termékhez tartozó 16-os érték nem egész batch számhoz tartozik, a legközelebbi 15 profittal rendelkezik, ami infeasible már. Az egyenes alatt azok a konfigurációk szerepelnek, amelyek jövedelme nem éri el a jelenlegi maximumot, így ezek nem a lehető legjobb megoldást adják meg. Az algoritmus addig nem állt le, amíg a halmaz, amely a konfigurációkat tartalmazza ki nem ürül. A példafeladatban a maximálisan elérhető jövedelem 25 egység. Ezt 4 darab A termék és 3 darab B termék legyártásával lehet elérni.

## 3. fejezet

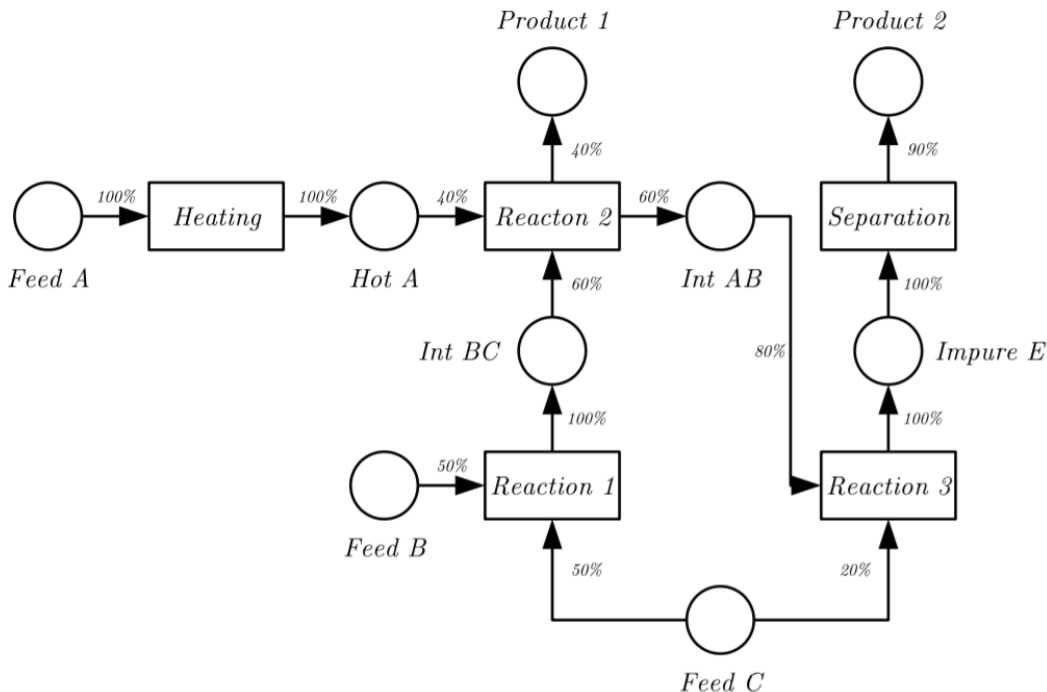
### Probléma definíció

Az előző fejezetben bemutatott throughput maximalizálás esetében kikötés volt, hogy minden termék batch mérete rögzített legyen, azaz a termékek batch-jének jövedelmét ismerjük. Abban az esetben minden taszkhoz egy berendezés hozzárendelése megengedett, azaz csak egy berendezés végezheti el. Azonban számos esettanulmány és irodalmi példa esetében a batch méretek nem rögzítettek. Ha több berendezés képes elvégezni ugyanazt a taszkot, akkor ezt megtehetik párhuzamosan. Ilyen eset főleg a throughput maximalizálás során léphet fel, de megjelenhet makespan minimalizálásnál is. Az időfelosztásos módszerek meg tudják oldani az ilyen problémákat, azonban az S-gráf ke-retrendszer esetén néhány módosítás szükséges. A throughput maximalizálás algoritmusai megköveteli, hogy a recept rögzített, valamint egy termék batch-jének jövedelme is előre ismert legyen. Az előbb említett problémák esetén viszont egyik sem garantált. A javasolt megközelítés szemléltetésére a Kondili és munkatársaitól származó példát veszem igénybe [2]. Ez látható 3.1 ábrán.

A folyamat 5 taszkból áll: fűtésből, 3 darab reakcióból, és az szétválasztásból. Ezekhez 4 berendezés áll rendelkezésre: a fűtőtest és szeparátor, mindkettő 100 kilogrammos kapacitással a fűtés és elválasztás folyamatához. A három reakciós folyamathoz van 2 darab reaktor ugyanakkora feldolgozási idővel. A kapacitásuk eltér, egyiké 80 kilogramm a másiké pedig 50 kilogramm, de ezeket a reaktorokat párhuzamosan is igénybe lehet venni. Feltételezzük, hogy az összes egység képes a kapacitásuknál kisebb terheléssel működni, azaz nincs meghatározva, hogy minimálisan mekkora mennyiség szükséges. A folyamat



során két termék készül azonos profittal. A következő korlátozások fennállnak: nem marad semmilyen köztes anyag a termelő folyamat végén, nem lehet csak az egyes számú terméket gyártani, illetve nincs tárolásra lehetőség a folyamat során.



3.1. ábra. Kondili példafeladata

Mindegyik reakció elvégezhető az egyik, a másik, vagy mindkettő reaktor által párhuzamosan. Ez  $3^3 = 27$  rögzített receptet eredményez, amelyek különböző batch mérettel rendelkezhetnek. Mindegyik esetben különálló S-gráf receptet kell létrehozni, hogy a korábban említett S-gráf algoritmust igénybe lehessen venni profit maximalizálásra, így a legfelső szinten lévő keresési terület 27 dimenziós térré válna. Ez óriási CPU igényhez vezetne az optimalizálás során, ezért az esetek számának csökkentése elengedhetetlen.

Ha megnézzük a 3.1 táblázatot észrevehető, hogy csupán néhány érték ismétlődik. Ennek oka az anyagok egyensúlyából származik. Például, hogy ha mind az R1-et mind az R2-t hozzárendeljük a hármas számú reakcióhoz ahelyett, hogy csak az R1 lenne hozzárendelve, akkor sem lesz nagyobb a kimenet, mert a korábbi reakciókból származó pótlás nem éri el a szükséges szintet. Ha a két különböző eset,  $c$  és  $c'$ , ugyanakkora maximális jövedelemmel rendelkezik, de a  $c$  eset csak kisebb részét használja a  $c'$  által használt egységeknek, ekkor

3.1. táblázat. A 27 rögzített recept Kondili példájához

Eset	Reakció 1	Reakció 2	Reakció 3	Max bevétel
1	R1	R1	R1	86,00
2	R1	R1	R2	71,67
3	R1	R1	R1&R2	86,00
4	R1	R2	R1	53,75
5	R1	R2	R2	53,75
6	R1	R2	R1&R2	53,75
7	R1	R1&R2	R1	114,76
8	R1	R1&R2	R2	71,67
9	R1	R1&R2	R1&R2	139,75
10	R2	R1	R1	86,00
11	R2	R1	R2	71,67
12	R2	R1	R1&R2	86,00
13	R2	R2	R1	53,75
14	R2	R2	R2	53,75
15	R2	R2	R1&R2	53,75
16	R2	R1&R2	R1	89,58
17	R2	R1&R2	R2	71,67
18	R2	R1&R2	R1&R2	89,58
19	R1&R2	R1	R1	86,00
20	R1&R2	R1	R2	71,67
21	R1&R2	R1	R1&R2	86,00
22	R1&R2	R2	R1	53,75
23	R1&R2	R2	R2	53,75
24	R1&R2	R2	R1&R2	53,75
25	R1&R2	R1&R2	R1	114,76
26	R1&R2	R1&R2	R2	71,67
27	R1&R2	R1&R2	R1&R2	139,75

az mondjuk, hogy a  $c$  *dominálja* a  $c'$ -t. A példából látható, hogy a 9-es eset dominálja a 27-es esetet. Továbbá a 24-es eset dominálva van a 4, 5, 6, 13, 14, 15, 22, 23 esetek által. Megállapíthatjuk, hogy ha egy eset legalább egy másik által dominálva van, akkor azt kizárhatjuk a vizsgálatból, mert az továbbra is garantálva van, hogy megtalálja az optimális megoldást. A 3.3 táblázat tartalmazza azokat az eseteket, amelyek nincsenek dominálva más esetek által.

3.2. táblázat. Nem dominált esetek bevétel szerint növekvő sorrendben

Eset	Reakció 1	Reakció 2	Reakció 3	Max bevétel
4	R1	R2	R1	53,75
5	R1	R2	R2	53,75
13	R2	R2	R1	53,75
14	R2	R2	R2	53,75
2	R1	R1	R2	71,67
11	R2	R1	R2	71,67
1	R1	R1	R1	86,00
10	R2	R1	R1	86,00
16	R2	R1&R2	R1	89,58
7	R1	R1&R2	R1	114,67
9	R1	R1&R2	R1&R2	139,75

Ezután a esetszám csökkenés után is még mindig 11 esetet kellene az S-gráf algoritmusnak megvizsgálni. Annak érdekében, hogy tovább csökkenjen ez a szám több esetet is össze lehet vonni. Például a 4-es és 5-ös eset teljes mértékben megegyezik, azzal a kivétellel, hogy a harmadik reakciós folyamatot más reaktor végzi. Ezt a két esetet össze lehet vonni úgy, hogy a harmadik reakciónál R1 vagy R2-es reaktor ( $R1 \vee R2$ ) üzemel. A 3.3 táblázatban látható a végleges összevonás eredménye.

3.3. táblázat. Összevont, nem dominált esetek bevétel szerint növekvő sorrendben

Eset	Reakció 1	Reakció 2	Reakció 3	Max bevétel
4,5,13,14	$R1 \vee R2$	R2	$R1 \vee R2$	53,75
2,11	$R1 \vee R2$	R1	R2	71,67
1,10	$R1 \vee R2$	R1	R1	86,00
16	R2	R1&R2	R1	89,58
7	R1	R1&R2	R1	114,67
9	R1	R1&R2	R1&R2	139,75

Látható, hogy az ilyen típusú feladatoknál nem lehet közvetlenül a megoldó programot igénybe venni a feladat megoldásához, szükség van előzetes lépésekre, hogy megfelelő formába kerüljön a feladat. Ez mindig plusz időbe kerül, és ez az idő annál nagyobb lehet, minél több taszk és berendezés szerepel az adott feladatban. A következő fejezetben bemutatott új módszer esetén ezek a lépések elhagyhatóak, ezáltal időt lehet megspórolni.

## 4. fejezet

# Az új módszer

Az új módszer létrehozását a meglévő megoldónál fellépő, a változó batch mérettel rendelkező feladatokból származó hátrányok ihlették. Abban az esetben az összes lehetséges, különböző hozzárendelést rögzíteni kell a receptben, így különböző termékek lesznek. Ennek következményeképp 1 dimenzió helyett többet kell bejárnia az algoritmusnak, ebből kifolyólag a futás lassabb lesz, mert több megvalósíthatósági tesztet kell elvégezni. Ezenkívül a nagyobb csúcsszám miatt nő az infeasible csúcsok száma is. Az új megoldó módszer ezen hátrányok kiküszöbölésére törekszik.

A két módszer nem teljes mértékben tér el, vannak olyan részek benn, amelyek megegyeznek. Először is mindkét esetben  $n$  dimenziós térben ( $n$  a termékek száma) keresi a legnagyobb profittal rendelkező konfigurációt. Ha talál egy megvalósíthatatlan konfigurációt akkor mindegyik módszer elveti ezt, és az ennél nagyobbakat. Mindegyik módszer esetén sor kerül a megvalósíthatóság tesztelésére, illetve ha talál egy megvalósíthatatlan konfigurációt akkor elveti ezt és az ennél nagyobbakat. Részütemezés idejét makespan minimalizálással meghatározzák, és ha ez nagyobb, mint a megadott korlát, akkor az adott részütemezést elveti.

A módszerek között eltéréseket is lehet találni. Fontos eltérés, hogy az új módszernél nem alkalmazható a revenue line arra, hogy csökkentsük a megvizsgálandó konfigurációk számát. Itt a elérhető jövedelmet nem skaláris szorzat adja meg, mint a régi megoldó esetében (termék száma szorozva a termék jövedelmével), hanem a mennyiség is közrejátszik, így előfordulhat, hogy kisebb konfigurációk esetén nagyobb profitra teszünk

szert. Ennek tekintetében lényeges az is, hogy ha talál egy részütemezés esetén megvalósítható megoldást, akkor nem tér azonnal vissza ezzel, hanem megkeresi a legjobb elérhető. Az új algoritmus nem csökkenti a beütemezendő feladatokat, mert engedélyezett, hogy több berendezés egyszerre végezze ugyanazt a feladatot. Ennek következtében a levél nem lehet az a részfeladat, ahol már nincs ütemezendő feladat. Ehelyett az számít levélnek, ha már minden berendezés ütemezése lezárt. Továbbá eltérés a felső korlátban is megfigyelhető. Az új módszernél a felső korlátot az adja, hogy a még elérhető berendezéseket hozzárendeli minden olyan feladathoz, amelyet az adott berendezés el tud végezni. Ezen a módon minden csúcsra meghatározza az elérhető legnagyobb kapacitást.

Az új módszernek 3 nagyobb elkülöníthető része van, amelyek bemutatása a következő pontokban található meg.

## 4.1. Vezérlő

A vezérlő feladata a konfigurációk kezelése. Első lépésben az algoritmus inicializálja a  $profit^{cb}$  változót mínusz végtelennel. Ennek oka az, hogy ha a feladatnak van megoldása, akkor ennél az értéknél biztosan nagyobb jövedelemre lehet szert tenni. Ezt követően az  $S$  halmazt inicializálja a termékek összes lehetséges konfigurációval. Ezek természetesen csak nem negatív egész számok lehetnek. Ezután az iteráció minden lépésében a **select\_remove** függvény segítségével kiválaszt egy konfigurációt. Először minden feladat esetében végigmegy azokon a konfigurációkon, ahol csak egy termék kerül legyártásra, így meghatározható egy régió, amely tartalmazza az összes megvalósítható megoldást. Az algoritmus meghívja a *MAXPROFIT* függvényt, amely által visszaadott értéket összehasonlítja a jelenlegi legjobb megoldással. Ha jobb, mint a meglévő profit, akkor ez az új érték lesz a legjobb megtalált megoldás, és a legjobb konfiguráció is frissítésre kerül. Azonban, ha a visszakapott érték mínusz végtelen, akkor az jelenti, hogy az adott konfiguráció esetén nincs megvalósítható megoldás, így az ennél nagyobb konfigurációk eltávolíthatóak az  $S$  halmazból, mert azok között se található feasible megoldás. Miután az iteráció véget ért, és volt megvalósítható megoldás, akkor az algoritmus visszatér a legjobb megoldást nyújtó konfigurációval, és az ehhez tartozó legjobb profittal.

**Algoritmus 3** A vezérlő pszeudó kódja

---

```

1: procedure CONTROLLER( $TH$ )
2:    $profit^{cb} := -\infty$ 
3:    $S := (\mathbb{Z}^*)^{|P|}$ 
4:   while  $S \neq \emptyset$  do
5:      $x := select\_remove(S)$ 
6:     if  $MAXPROFIT(TH, x) > profit^{cb}$  then
7:        $profit^{cb} := MAXPROFIT(TH, x)$ 
8:        $x^{cb} := x$ 
9:     else if  $MAXPROFIT(TH, x) == -\infty$  then
10:       $S := \{x' \in S \mid x' \not\geq x\}$ 
11:     end if
12:   end while
13:   if  $profit^{cb} \neq -\infty$  then
14:     return  $(x^{cb}, profit^{cb})$ 
15:   end if
16: end procedure

```

---

## 4.2. Maxprofit eljárás

Ennek a függvénynek több feladata van. Itt hívódik meg a megvalósíthatósági teszt, valamint a *ProfitBound* függvény is, illetve a részprobléma ütemezése is itt történik meg. A függvény alapját a Makespan minimalizáló algoritmus adja. A bemenetet az időhorizont ( $TH$ ), valamint a batch szám adja. Első lépésként a  $profit^{cb}$  értékét inicializálja mínusz végtelennel, mert ennél biztosan nagyobb lesz a profit megoldható probléma esetén. A *SOAA* halmaz kezdetben nem tartalmaz értéket, később futás során azok a berendezések kerülnek bele, amelyek már hozzá vannak rendelve egy adott taszkhoz. A **recipe** függvény ugyanúgy, ahogy a makespan minimalizáló algoritmus esetében van, visszaadja a probléma receptgráfjának modelljét, és hozzáadja az  $S$  halmazhoz.

Ezeket követően a **select\_remove** függvény kiválaszt egy tetszőleges részproblémát az iteráció minden lépésében. Ez addig tart, amíg az  $S$  halmaz ki nem ürül, azaz elfogynak a részproblémák. A kiválasztott részprobléma a  $PP$  változóba kerül a következő formában  $(G(N, A_1, A_2, w), I', J', \mathcal{A})$ . Ezután sor kerül a megvalósíthatóság tesztelésére. Az al-

goritmus megvizsgálja a kiválasztott részproblémát a *Feasible* függvény segítségével, ez megnézi, hogy a részprobléma a megadott időhorizonton belül megvalósítható-e. Megnézi, hogy a részproblémában lévő leghosszabb út nagyobb vagy kisebb a megadott korlátnál. Ha kisebb, azaz a feladat elvégezhető a megadott idő alatt, akkor feasible lesz a megoldás, máskülönben infeasible-nek minősül. Ha megvalósítható, akkor megvizsgálja az algoritmus, hogy a részprobléma által nyújtott legnagyobb elérhető jövedelem nagyobb-e, mint az eddig megtalált legjobb megoldás. Az részprobléma jövedelmét a *ProfitBound* függvény adja meg, amely kifejtése a következő pontban történik meg. Ha valamelyik feltételnek nem felel meg a részprobléma, akkor az iteráció ezen lépése véget ér, és egy másik részprobléma kerül kiválasztásra, amennyiben még van ilyen.

Ha mindkét feltételnek megfelel a részprobléma, akkor megnézi az algoritmus, hogy levél-e. Ez azt jelenti, hogy található-e még olyan berendezés, amely nincsen teljesen beütemezve, vagyis tud még feladatokat végezni. Abban ez esetben ha nincs ilyen berendezés ( $J' == \emptyset$ ), akkor a jelenlegi legjobb megoldás ( $profit^{cb}$ ) értéke felülíródik a vizsgált részprobléma által nyújtott megoldással. Ha viszont nem levél, akkor egy berendezés kerül kiválasztásra a még elérhető berendezések halmazából. A kiválasztott  $j$  berendezéshez az algoritmus hozzárendeli az összes lehetséges taszkot, amelyet el tud a berendezés végezni, és még nincs hozzárendelve. Ezután az éppen soron lévő részfeladat hozzáadódik ahhoz a halmazhoz, amelyben azok a taszkok szerepelnek, amelyet a  $j$  berendezés már végez. Ezek a kiválasztott taszkok kapnak egy másolatot az aktuális S-gráfról. Mivel a csomópontok és a receptélek halmaza nem változik, ezért ezek változatlanok maradnak, csak az ütemezési élek halmaza és súlyok változnak. Ezek után az algoritmus bővíti az ütemezési élek halmazát az új hozzárendelések alapján. Ezt követően az  $i$  taszkból induló receptél súlya megkapja  $t_{i,j}^{pr}$  értékét. Mindezeket követően az  $S$  halmazhoz hozzáadja az algoritmus az új hozzárendeléssel kiegészített részproblémát.

Azon taszkok esetében, amelyeket még egy berendezés sem végez, viszont a kiválasztott  $j$  berendezésen kívül más berendezés is meg tud oldani, olyan döntés is születhet, hogy az adott berendezés nem is fogja azt elvégezni. Ilyenkor az  $S$  halmaz kibővül egy olyan részproblémával, ahol a  $j$  berendezés már nem végez több feladatot.

Az ütemezés elvégeztével az algoritmus visszatér a megtalált legjobb megoldással,

vagyis az elérhető legnagyobb bevétel értékével.

---

**Algoritmus 4** Párhuzamos taszkvégrehajtást megvalósító algoritmus
 

---

```

1: procedure MAXPROFIT( $TH, \text{batch\_number}$ )
2:    $\text{profit}^{cb} := -\infty$ 
3:    $SOAA := \emptyset$ 
4:    $S := (\text{recipe}(), I, J, \emptyset)$ 
5:   while  $S \neq \emptyset$  do
6:      $PP := \text{select\_remove}(S)$ 
7:     if  $PP.\text{Feasible}(TH)$  then
8:       if  $PP.\text{ProfitBound} > \text{profit}^{cb}$  then
9:         if  $PP.\text{IsLeaf}()$  then
10:            $\text{profit}^{cb} := PP.\text{ProfitBound}$ 
11:         else
12:            $j := \text{select}(J')$ 
13:           for all  $i \in I_j \setminus SOAA$  do
14:              $SOAA_j := SOAA_j \cup i$ 
15:              $G^i(N, A_1^i, A_2^i, w^i) := G(N, A_1, A_2, w)$ 
16:             for all  $i' \in \bigcup_{(i',j) \in \mathcal{A}} I_{i'}^+ \setminus \{i\}$  do
17:                $A_2^i := A_2^i \cup \{(i', i)\}$ 
18:             end for
19:             for all  $i' \in I_i^+$  do
20:                $w_{i,i'}^i := t_{i,j}^{pr}$ 
21:             end for
22:              $S := S \cup (G^i(N, A_1, A_2^i, w^i), I, J', \mathcal{A} \cup \{(i, j)\})$ 
23:           end for
24:           if  $I_j \setminus \bigcup_{j \in J} SOAA_j \subseteq \bigcup_{j \neq j', j' \in J'} I_{j'}$  then
25:              $S := S \cup (G(N, A_1, A_2), I, J' \setminus \{j\}, \mathcal{A})$ 
26:           end if
27:         end if
28:       end if
29:     end while
30:   return  $\text{profit}^{cb}$ 
31: end procedure

```

---



### 4.3. Profitbound eljárás

A Profitbound függvény számolja ki egy részprobléma jövedelmét. A függvény először beállítja a *profitbound* változó értékét nullára. Ez a változó fogja megadni a termékek legyártásából elérhető jövedelmet. Ezután a függvény beállítja minden taszk kapacitását ( $CAP^i$ ). Ezt úgy lehet megkapni, hogy össze kell adni azoknak a berendezéseknek a kapacitását ( $C^j$ ), amely el tudja végezni az adott feladatot. Ezután a az algoritmus kiszámolja a ténylegesen hasznosított kapacitások mennyiségét. Ezt úgy teszi, hogy végigmegy a recepteleken, kezdve a nyersanyagoktól a kész termékek fel haladva, és az ott bemeneti adatként megadott százalékokat beleveszi a számításba. Egy receptélen kettőt tudunk megkülönböztetni. Az első ( $SP_{i,i'}$ ) az, amelyik az mutatja meg, hogy a befejezett taszk mekkora kapacitását hasznosítjuk. A második ( $DP_{i,i'}$ ) pedig, hogy a soron következő feladat kapacitás mekkora hányadát tudja felvenni. A következő képlet segítségével lehet meghatározni a kapacitást:

$$\text{kapacitás} = \text{előző taszk kapacitása} * \frac{\text{előző taszkból felvett kapacitás mennyiségnek százaléka}}{\text{a mennyiség százaléka, amit az éppen vizsgált taszk feltud venni}}$$

Fontos dolog, hogy az algoritmus mindig a legkisebb kapacitást veszi figyelembe egy taszknál, mert ez az a mennyiség, ami mindenképpen elérhető. A következő lépésben a *profitbound* kiszámolása történik. A profitboundot a végtermékek mennyisége és a legyártásból származó jövedelem szorzata adja meg. Végezetül ezzel az értékkel tér vissza a függvény.

---

**Algoritmus 5** A profitbound függvény pszeudó kódja

---

```

1: procedure PROFITBOUND
2:    $profitbound := 0$ 
3:   for all  $j \in J$  do
4:     for all  $i \in I_j$  do
5:        $CAP^i += C^j$ 
6:     end for
7:   end for
8:   for all  $i \in I$  do
9:     for all  $i' \in I_i^+$  do
10:      if  $CAP^i > CAP^i * (SP_{i,i'}/100) * (100/DP_{i,i'})$  then
11:         $CAP^i := CAP^i * (SP_{i,i'}/100) * (100/DP_{i,i'})$ 
12:      end if
13:    end for
14:  end for
15:  for all  $i \in I$  do
16:    for all  $p \in P$  do
17:      if  $(i, p) \in A_1$  then
18:         $profitbound += CAP^i * REV^p$ 
19:      end if
20:    end for
21:  end for
22:  return  $profitbound$ 
23: end procedure

```

---

## 5. fejezet

# Implementálás

### 5.1. S-gráf solver

Az S-gráf solver program egy C++ nyelven megvalósított, nagy teljesítményre képes, több szálú program. A szoftver szakaszos üzemű termelőrendszerek rövidtávú ütemezésével foglalkozik. Tárolási stratégiákat tekintve jelenleg támogatja a NIS, UIS, UW és LW feladatokat, valamint lehetőség van AWS feladatok megoldására is. A célfüggvények közül a makespan minimalizáció, a throughput maximalizáció és a ciklusidő minimalizálás támogatott.

A szoftver felépítésében nagy szerepe van az objektum orientáltságnak az osztályok használatán keresztül. Ezek segítségével a megoldóban el vannak különítve a beolvasás, a végeredmény kiírás, valamint a különböző megoldó algoritmusokat végző részek, modulok. A program képes különböző formátumú fájlok beolvasására (Pl: xml, ods, csv), amelyek tartalmazzák a probléma megoldásához szükséges információkat. A bemeneti fájl alapján felépül a receptgráf, és ebből legenerálja a részproblémákat. A szoftver az eredmény képernyőn való megjelenítésén kívül fájlban is eltárolja azt. Továbbá a Gantt diagram adatait karakteres formában is elmenti az említett fájlba, illetve lehetőség van arra is, hogy ezt a diagramot képfájlban mentse el. A megoldó használata környezeti változók segítségével történik. Ezekkel adható meg a bementi fájl, az eredményeket tároló fájl, a kívánt megoldó módszer meghatározása, az időhorizont megadása, továbbá számos

különböző beállítási lehetőség. Erre egy példa:

$$-i \text{ input.ods} -o \text{ output.txt} -t 2 -m \text{ eqbased}$$

Az "-i" paranccsal adható meg a bemeneti, input fájl, az "-o" paranccsal pedig a kimeneti, output fájl lehet megadni. A "-t" utasítás a maximálisan használható szálak megadására szolgál. A példában az utolsó paranccsal, az "-m" kapcsolóval, pedig a megoldó módszert tudjuk kiválasztani.

A megoldóban egyik legfontosabb szerepet tölti be a Branch and Bound, azaz a Korlátozás és Szétválasztás algoritmus. A szétválasztás több módszerrel is végrehajtható, ezért a szoftver kialakítása révén létrehozhatóak, hozzáadhatóak új algoritmusok a megoldóhoz.

### 5.1.1. A megoldó működése

A szoftver parancssori kapcsolókkal futtatható. Az összes ilyen paraméter a **Arguments.cpp** fájlban van tárolva. A **MainSolver** osztályban meghívásra kerül a **getOptions** függvény, ami létrehoz egy példányt a **SolverOptions** osztályból, amely tartalmazza a megadott paraméter alapján létrejött beállításokat. Ezt követően megtörténik a bemeneti adatok beolvasása, és létrejön egy **SGraph** objektum, ami a recept gráfot tartalmazza. Ehhez meghívásra kerül a **ReadInputFromFile** metódus, majd azon belül a **RelationalProblemReader** objektum **ReadSGraph** metódusa. Miután ez sikeresen lezajlott a **getProblem** metódus segítségével meghatározza a program a probléma típusát. Ezt követően a **getSolver** függvény példányosítja a szükséges solvert, amellyel a problémát megoldja. Ez a solver throughput maximalizálás során a **ThroughputSolver** osztály egy példánya lesz. A **Solver** metódus meghívásával kezdődik a probléma megoldása. Egy **TreeNode** objektum tartalmazza az optimális megoldás ütemezési gráfját. Miután megvan a megoldás egy **SolutionWriter** objektum **Write** függvényének meghívásával kiírja a megadott fájlba a megtalált megoldást.

Az eddig leírtakban nem volt szükség nagyobb módosításra az új megoldó módszer létrehozásához. A **ThroughputSolver** osztály működésével kapcsolatban volt szükség

új kódrészek implementálásához és a meglévők módosítására. A már említett **Solve** metódus először megkeresi az optimális megoldást tartalmazó teret. Ehhez végigmegy a tengelyek mindaddig, amíg megvalósítható az adott konfiguráció. Ha megvalósítható akkor megnöveli az adott termék batch számát. Első nagyobb eltérés az új és a régi megoldó között, hogy még a régiben a **FirstFeasible** metódus hívódik meg, addig az újban a **SolveBest**. A **FirstFeasible** lényege abban van, hogy amint talál egy megvalósítható megoldást akkor nem keres tovább. Az újban viszont ez nem jó, mert nem minden esetben az először megtalált megoldás egy konfiguráción belül a legjobb. Ha az adott konfiguráció megvalósítható, akkor a **NewSolution** metódus meghívódik, és létrehoz egy új lehetséges megoldást tartalmazó objektumot. Ha nem lehet megvalósítani, akkor null érték kerül visszaadásra. Ha ez történik akkor az adott tengelyen befejeződik a keresés. Ha minden tengelyen ez végbement, akkor létrejön a keresési tér. A következő lépés ennek a térnek a bejárása a **SearchThrSolution** függvénnyel. Az új módszer létrehozása során szükség volt ennek a metódusnak a módosítására. Eredetileg ez használja a revenue line gyorsítási stratégiát. Ennek használatát feltételhez kellett kötni, mert az új módszer esetén ennek használata nem lehetséges. Továbbá a jövedelem módosítására is szükség volt, mert nem egyezik meg a vizsgált korlát a két módszer esetében. A keresés végeztél a **Solve** metódus visszaadja a legjobb megoldást a **MainSolver** számára. Abban az esetben, ha nincs megoldás akkor egy exceptiont dob a **Solve** függvény, amely kezelése során a felhasználót értesíti a program arról, hogy az adott feladatnak nincs megoldása.

## 5.2. Adatok beolvasása

### 5.2.1. Bemeneti fájl

A program működéséhez szükséges adatokat ods kiterjesztésű fájlban lehet megadni. A solver mappa input almappájában megtalálható az **extended\_precedential.ods** fájl, amely segítségül szolgál ahhoz, ha új fájlt kívánunk létrehozni a saját feladatunkhoz tartozó adatokkal. Ez a fájl egy, a szoftver működtetéséhez alkalmas, korábban létrehozott fájl kibővített változata. Az abban megtalálható táblázatokhoz további oszlopok kerültek

hozzáadásra, amelyben az új módszerhez szükséges adatok szerepelnek. Az új fájl a következő táblákat tartalmazza, jelezve az újonnan hozzáadott oszlopokat:

- Product tábla tartalmazza a termékekkel kapcsolatos adatokat. Itt a változtatás revenue oszlop hozzáadása, amely az adott termék elkészítésével szerzett jövedelem.
- Equipment táblában találhatóak a berendezésekhez kapcsolódó információk. Újonnan került hozzáadásra az úgynevezett b\_capacity oszlop, amely az adott berendezés kapacitását mutatja meg.
- Precendence tábla, amely a gráfban szereplő éleket az él kezdő csomópontjának és végpontjának feltüntetésével szemlélteti. Két új, hasonló oszlop lett a táblázathoz illesztve, ezek a következők:
  - Az s\_percent oszlop, amely a táblázat task1 oszlopában szereplő részfeladat kapacitásának hasznosuló százalékát mutatja.
  - A d\_percent oszlop, amely azt mutatja, hogy a task1 oszlopban szereplő részfeladat kapacitásának mekkora részét tudja felvenni a task2 oszlopban szereplő részfeladat.
- A taszkok adatait tartalmazó task tábla változatlanul került felhasználásra.
- A Proctime táblában találhatóak meg azok az adatok, hogy melyik taszkot melyik berendezés tudja elvégezni, valamint, hogy mennyi idő szükséges ehhez. Ez szintén módosítás nélkül lett átemelve.

### 5.2.2. Beolvasó függvény

Az új módszer megoldásához szükséges adatok beolvasása, hasonlóan a fájl létrejöttéhez, már egy meglévő függvény kibővítésével valósul meg. Az értékek beolvasásáért a **RelationalProblemReader** osztály a felelős. Ennek feladata, hogy felépítse a receptgráfot, illetve ezt eljuttassa a **MainSolver** osztályhoz. A beolvasást végző függvények forráskódjai az `src\lib` mappában megtalálható **relationalproblemreader.cpp** és **relationalproblemreader.h** fájlokban szerepelnek. Az ebben megtalálható függvények közül az adatok

programba történő átemeléséhez a **ReadPrecedential()** függvényre van szükség, amely ki lett bővítve azzal, hogy az új adatokat is képes legyen feldolgozni.

Az említett függvényben meghívásra kerül a **ParseEquipments(SGraph\* graph)** eljárás, ami a fejlesztés során ki lett egészítve azzal, hogy vizsgálja meg, hogy a fájlban lévő equipment tábla rendelkezik-e `b_capacity` nevű oszloppal. Ha igen, akkor ellenőrzi, hogy a beolvasott érték negatív vagy sem. Abban az esetben, ha negatív, akkor a szoftver dob egy kivételt és a működés leáll, mivel csak nem negatív értékekkel oldható meg a probléma. Ellenkező esetben pedig az **SGraph** objektum **Equipment** objektumában, amely ki lett bővítve egy `double` típusú változóval az új adatok megőrzésének érdekében, eltárolásra kerül a beolvasott adat.

Következő változtatás az, hogy a fájlban megtalálható precedence tábla két új oszlopában (`s_percent`, `d_percent`) található értékeket el tudja a szoftver tárolni. Ezek az értékek az **SGraph** objektum **Recipe** objektumában tárolódnak. A tárolást úgy lehet elképzelni, mint egy mátrixot, ami azt mutatja meg, hogy az adott taszkból, melyik taszkba mutat él. A mátrixok mérete  $N \times N$ -es, ahol az  $N$  a taszkok számát jelenti. A **sourcePercents** azt testesíti meg, hogy annak a taszknak, amelyből az él indul, a kapacitásának mekkora része hasznosul. A **demandPercents** pedig azt reprezentálja, hogy mekkora százalékot tud felvenni az a taszk, amelybe az él mutat.

### 5.3. FlexBatchSchProblem osztály

Ezen osztály feladata a branching, vagyis a szétválasztás elvégzése az ütemezés során. Megállapítja, hogy melyik taszkokhoz melyik berendezés vagy berendezések hozzárendelése szükséges. Másképpen fogalmazva azt kell meghatároznia, hogy melyik berendezésnek, melyik taszkokat kell elvégezni a lehető legnagyobb profit elérésének érdekében. Az osztály forráskódja megtalálható az `src\solver` mappában a **flexbatchschproblem.cpp** és a **flexbatchschproblem.cpp** fájlokban. Ez az osztály egy származtatott osztály. A szülőosztálya az **EqBasedSchProblem** osztály, aminek szintén van egy őosztálya az **SchProblem** osztály. Az új módszer lényege abban áll, hogy egy adott taszkhhoz több berendezést is hozzá lehet rendelni, ezért az **EqBasedSchProblem** osztályban megtalálható

Branching függvény az ott szereplő formában ehhez a megoldó módszerhez nem megfelelő. Az eddigi adattagok mellett, az átdolgozott kiválasztás módszer miatt, szükséges új adattagok bevezetése. Az első új adattag egy vectoron belüli vector segítségével megvalósított mátrix, amely azt reprezentálja, hogy melyik berendezéshez melyik taszk lett már hozzárendelve. A másik új adattag pedig egy **IndexSet** típusú változó, amelyben azok a berendezések szerepelnek, amelyek még nincsenek ütemezve, azaz még képes elvégezni taszkokat.

### 5.1. Kódrészlet. FlexBatchSchProblem osztály adattagjai

---

```
1 class FlexBatchSchProblem: public EqBasedSchProblem{
2     protected:
3         vector<vector<bool>> eqAssignedToTask;
4         IndexSet sounEqs;
5 }
```

---

#### 5.3.1. MakeDecisions függvény

Ennek a függvénynek a feladata az, hogy találjon egy berendezést, amelyhez a probléma során még lehet legalább egy taszkot rendelni. Ha már nincs olyan berendezés amely még nem ütemezett a részproblémában, akkor a függvény futása máris véget ér. Ha ez nem történik meg, akkor következik a berendezés keresése. Itt meg kell vizsgálni, hogy az éppen soron lévő berendezés szerepel-e azon berendezések halmazában, amelyeket még a részproblémák megoldásához igénybe lehet venni. A berendezések közti keresés addig tart, amíg nem talál egy olyat, amit legalább egy taszkhhoz hozzá lehet rendelni. Ezt követően a probléma **Decision** típusú adattagjában ez a berendezés, illetve azok a taszkok amelyeket el tud végezni, kerülnek eltárolásra. Továbbá a függvényben kerül sor arra, hogy az említett adattagba beállítódjanak azok a taszkok, amelyeket csak az éppen kiválasztott berendezés képes elvégezni, valamint azok, amelyeket más berendezéshez vagy berendezésekhez is hozzá lehet rendelni. A döntést tartalmazó adattagban tárolásra kerül ezeken felül még az is, hogy az adott részproblémának mennyi gyereke lehet. Abban az esetben, ha olyan berendezés kerül kiválasztásra, amelyet csak olyan taszkhhoz lehet rendelni, amelyeket más berendezés is képes végrehajtani, akkor meg kell növelni a gye-



rekek számát, mert lehetséges olyan döntést hozni, hogy az adott berendezést semelyik lehetséges taszkhhoz sem rendeljük hozzá.

### 5.3.2. Branching függvény

Ez a függvény valósítja meg a szétválasztást a probléma megoldása során azaz, minden éppen aktuális részproblémára meghívja a Branch and Bound módszert megvalósító függvényt. Amennyiben az előző pontban már említett, **Decision** típusú adattagja nem üres, akkor lehetséges további döntéseket, hozzárendeléseket végezni. Az említett adattagban szerepel, hogy jelenleg melyik berendezésről kell dönteni, illetve szerepelnek azok a taszkok, amelyeket el tud végezni. Ezek közül a sorban az elsőt kiválasztja és megpróbálja az ütemezést végre hajtani az őosztályban szereplő **Schedule** függvény meghívásával. Ha ez nem lehetséges, akkor nem felelt meg a feasible, megvalósíthatósági tesztnek. Ellenkező esetben az említett függvény hozzáadja a gráfhoz az ütemezési éleket, beállítja a hozzárendeléseket (az elvégzéshez szükséges időt), illetve újraszámolja a frissített ütemezési gráfhoz tartozó ProfitBoundot, a profit korlátot. Ezek után az osztály **eqAssignedToTask** adattagjában beállítja az imént a gráfban is beállított berendezés-taszk párost, hogy ezt később már ne lehessen újra egymáshoz rendelni. Ha mindezt követően a kiválasztott berendezést már csak egy taszkhhoz lehet hozzárendelni, akkor a berendezést kivesszük a nem ütemezett berendezések halmazából. Abban az esetben, ha a kiválasztott berendezést már nem kívánjuk hozzárendelni taszkhhoz, de van olyan taszk amit még el tudna végezni és ezt a taszkot más berendezés is el tudná végezni, akkor a berendezést kivesszük az ezt követő részproblémákból. Mindezen lépések után a **MakeDecisions** függvény segítségével ennek a részproblémának a gyerek problémájához hozunk döntést, valamint a korlátja is beállításra kerül.

### 5.3.3. További metódusok

Az **IsFeasible** függvénynek az a feladata, hogy elvégezze annak ellenőrzését, hogy az adott probléma megvalósítható vagy sem. Ehhez igénybe veszi az őosztályban megtalálható ugyanezzel a névvel rendelkező függvényt. Ebben megvizsgálásra kerül, hogy a gráfban

található-e kör. A kör olyan egymáshoz csatlakozó élek sorozata, amelyben az élek és pontok egynél többször nem szerepelhetnek, és a kiindulási pont megegyezik a végponttal. Az új osztályban szereplő függvény ezt kibővíti azzal, hogy megvizsgálja, hogy a megadott időhorizonton belül megoldható-e a feladat. Ha e kettő feltétel közül valamelyiknek nem felel az adott feladat, akkor az éppen vizsgált részprobléma nem lesz megvalósítható a megadott feltételek mellett.

A **Bound** eljárás a korlátot állítja be. Mivel az elkészített módszer maximalizációra lett tervezve, a Solver keretrendszerben megtalálható további megoldó módszerekkel szemben, amelyek pedig minimalizálnak, ezért szükséges a negatív szorzó, hogy a korábban elkészített függvényekben megfelelő eredményeket lehessen elérni.

Megtalálható még egy egyszerű **IsComplete** névvel fellelhető függvény, amelynek csupán annyi a szerepe, hogy egy bool értéket ad vissza, ami azt mutatja meg, hogy a berendezések halmaza üres vagy nem. Az üres állapot azt jelenti, hogy az összes berendezés már ütemezett, vagyis nem szándékozunk, vagy nem lehet hozzá taszkokat rendelni. Ellenkező esetben pedig, legalább egyhez még lehet taszkot hozzárendelni.

Az osztályhoz tartozik három másoló függvény is a **FastClone**, a **Clone** és a **MakeCopy**. Az elsőnek említett függvény egyszerűen létrehoz egy új objektumot, aminek paraméterlistájában átadjuk a másolni szándékozott problémát. A Clone függvény az előbb említetthez képest abban tér el, hogy paraméterként true-t ad meg, ami azt mutatja meg, hogy a receptet is másolja vagy sem. A harmadik viszont meghív egy **CopyInto** névvel ellátott függvényt, ami adattagonként végzi a másolást. Ennél meg lehet adni, hogy teljes másolást végezzen, illetve megtartsa az eredeti problémában lévő döntéseket. További eltérés a három függvény között, hogy a **FastClone** elérhetősége proceted, ami azt jelenti, hogy csak a származtatott osztályai érik el, nem pedig bármely függvény. A másik két függvény viszont public, így azokat máshonnan, osztályon, származtatott osztályokon kívülről is el lehet érni.

## 5.4. SGraph osztály

Az **SGraph** osztály egy olyan osztály, amely támogatja különböző műveletek elvégzését az S-gráfon. Többek között az ilyen feladatok közé tartozik az ütemezési élek hozzáadása, korlátok lekérdezése, leghosszabb út lekérdezése, valamint taszkok és berendezések közötti hozzárendelések megszüntetése. Az osztályban metódusok módosítására, valamint új függvények hozzáadására van szükség ahhoz, hogy képes legyen párhuzamos hozzárendelést megengedő feladatok elvégzésére. Két teljesen új függvény került hozzáadásra az **UpdateProfitBound** és az **UpdateProfitBoundFromTask**. Ezenkívül egy függvény nagyobb megváltoztatására is sor került, ez a függvény pedig a **MakeMultipleBatches**. Az előbb említett 3 metódus mindegyike az **sgraph.cpp** és **sgraph.h** fájlokban található meg. Ezek a fájlok az **src\solver** mappában fellelhetők. Az osztályt egy adattaggal kellett kibővíteni, amiben a taszkokhoz tartozó kapacitást lehet eltárolni, vagyis mekkora mennyiséget tud az adott taszk előállítani. Az adattag vector segítségével valósítja meg a tárolást, amelyben double típusú adatokat lehet elmenteni.

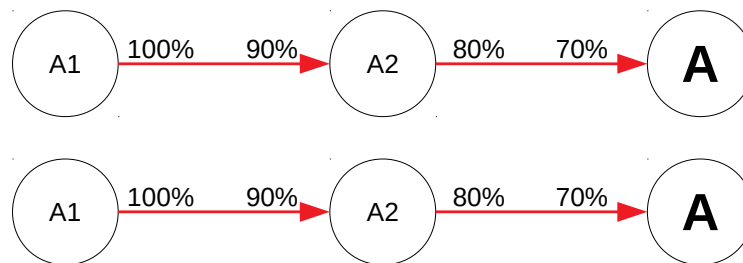
### 5.4.1. MakeMultipleBatches függvény

Ez a függvény abban az esetben játszik fontos szerepet, amikor legalább egy termékből egynél több darabot szeretnénk gyártani, vagyis a batch szám nagyobb lesz mint egy. Ilyenkor az adott termékhez tartozó minden taszk számát a program futása során annyira kell módosítani, amennyi terméket kell legyártani. Tekinthetünk úgy rá, hogy minden darab termékhez saját recept készül. A beolvasás eredetileg egy receptet készít el, vagyis minden taszkból csak egy szerepel. Ezen taszkok új száma alapján módosul már az 5.2.2. pontban említett **Recipe** osztályban lévő  $N \times N$ -es mátrixok mérete. A függvény a korábban meglévő módszerekhez szükséges adatok átdolgozásáról gondoskodik, csak az új adattagok módosításával kell foglalkozni, így a taszkok száma már megfelelő lesz, mikor a százalékokat tartalmazó mátrixot módosítani kell. Ezek a százalékok azt jelentik, hogy az él mekkora mennyiségekkel foglalkozik. A bemeneti fájlban megtalálható **s\_percent** oszlopban lévő adat azt mutatja meg, hogy az él, abból a taszkból, amelyből indul, onnan az ott gyártott mennyiség mekkora részét képes átvenni. A **d\_percent** oszlopban lévő

adatok pedig, hogy az a taszk, amelybe az él befut, mekkora százalékát képes felvenni a mennyiségnek.

Az eredetileg szereplő taszkok azonosítója megváltozik, mivel az új taszkokat nem csak hozzá adjuk azokhoz a következő azonosítóval. Az ugyanolyan taszkok egymást követő azonosítót kapnak, így a százalékokat tároló mátrixot is módosítani kell. Fontos dolog az, hogy csak az adott recepthez tartozó taszkok között lehet élt behúzni, nem lehet másik receptben szereplő taszkhoz hozzárendelni. Ezeket különböző feltételek bevonásával lehet megvalósítani. A megvalósítás úgy jött létre, hogy a mátrix első sorának ellenőrzése kis mértékben eltér a további sorok átvizsgálásától. Ez az eltérése az over változóban mutatkozik meg, mégpedig úgy, hogy ez a változó ugyanazokat a taszkokat reprezentáló, különböző azonosítókat vizsgálja a feljebb lévő sorokban. A feltételek forráskódjai a következő részben tekinthetők meg.

A könnyebb átláthatóság érdekében az 5.1 ábrán látható egy példa. Két darab A terméket akarunk legyártani. Az adatok fájlból való beolvasása során az A1-es taszk megkapja a 0. azonosítót, az A2 pedig az 1. sorszámot. Mivel kettőt gyártunk le, ezért meghívódik a **MakeMultipleBatches** függvény, és újra kiosztja az azonosítókat, miközben létrehozta kellő számban az eredetiről lemásolt taszkokat.



5.1. ábra. Példafeladat

Az új sorrend a 5.1 táblázatban látható. Zárójelben pedig látható, hogy az 5.1 ábrán melyik sorban lévő recepthez tartozik.

5.1. táblázat. A taszkokhoz tartozó azonosítók

Taszk	Azonosító
A1 (első)	0
A1 (második)	1
A2 (első)	2
A2 (második)	3

Azt nem lehet megengedni, hogy a 0. azonosítóval rendelkező taszk és a 3. azonosítóval rendelkező taszk között él keletkezzen, mert nem ez a kettő taszk tartozik egymáshoz. A példafeladatban látható, hogy egy él kezdő taszkjának, és annak a taszknak, amelybe érkezik, az azonosítójuk különbsége éppen annyi, amennyi terméket gyártani szeretnénk. Jelen esetben kettő.

#### 5.4.2. UpdateProfitBound függvény

A függvény feladata, hogy kiszámolja az első S-gráfhoz tartozó korlátot, valamint minden egyes taszkhoz tartozó kapacitást is meghatározza. Legelső lépésben beállítja a taszkoknak az úgynevezett alap kapacitását. Ezt az alapján lehet meghatározni, hogy egyes berendezések, melyek a részfeladatot képesek elvégezni rendelkeznek-e kapacitással. Ezeket a bemeneti fájlból olvassa be a szoftver. Egy taszk kapacitását az összes, őt elvégezni képes taszk kapacitásának összege adja meg. Miután ez megtörtént a következő lépés a kezdő csomópontok, taszkok megkeresése. Ez 2 *for* ciklus segítségével történik, amelyekben megvizsgáljuk, hogy az adott csomópont rendelkezik-e abba tartó, bementi éllel. Ha ilyen nincs akkor biztosak lehetünk benne, hogy az adott csomópont kezdő csomópont. Miután ezzel megvagyunk akkor megkeressük az előbb megtalált csomópontok szomszédjait. Ehhez egy **deque** (double-ended queue), azaz kétvégű sort veszünk igénybe. Ennek előnye abban rejlik, hogy mind az elejéhez, mind a végéhez lehetséges elemet fűzni, illetve onnan eltávolítani. Ebbe a változóba tároljuk a kezdő csomópontok szomszédjait. Ismét két *for* ciklus segítségével bejárjuk a taszkokat, amennyiben van köztük él, és még nem szerepel az adott taszk a **deque-ban**, akkor beletesszük.

Mindezeket követően elérkezik az a rész, ahol a kapacitások felülvizsgálata következik. Egy *while* ciklus segítségével minden **deque-ban** szereplő elemet vizsgálunk addig,

amíg az teljesen üressé nem válik. Először a **deque** első elemét kivesszük belőle, majd egy *for* ciklus segítségével ismét végighaladunk a taszkokon. Ha az éppen ciklusban lévő taszkból mutat él a **deque-ból** kivett taszkba, továbbá a taszknak, amiből az él indult, már korábban, a mostani függvény futása során felül lett vizsgálva a kapacitása, akkor lehet ellenőrizni a **deque-ból** kivett taszk kapacitását. Ha az aktuálisan kiszámolt kapacitás nagyobb mint az eddigi, akkor a korábbi helyett az újat jelöljük ki a taszk kapacitásának. Ennek kiszámításhoz a 4. fejezetben bemutatott képletet kell használni.

Abban az esetben, ha a kezdeti taszk még nem lett ellenőrizve, akkor nem lehet megvizsgálni az éppen kiválasztott taszkot, ezért visszatesszük a **deque** végére. Ha viszont lehetséges volt és végbe is ment az adott taszk felülvizsgálata, akkor megkeressük ennek a csomópontnak a szomszédjait és hozzáfűzzük a **deque** végéhez.

A függvény utolsó szakaszában történik meg a profit korlát meghatározása, kiszámolása. A korlátot azoknak a taszkoknak a kapacitása adja meg, amelyek a termékek előtti utolsó részfeladatok. Ezek megtalálása úgy történik, hogy *for* ciklus segítségével bejárjuk a termékeket, valamit a taszkokat is. Ha valamelyik taszkból indul él egy termékbe, akkor a taszk kapacitását megszorozzuk a termékből származó jövedelemmel.

### 5.4.3. UpdateProfitBoundFromTask függvény

Ez a függvény feladatában hasonlít az előző pontban bemutatott **UpdateProfitBound** metódushoz. A taszkokhoz tartozó kapacitást, valamint a korlátot kell meghatároznia. Különbséget abban lehet felfedezni, hogy ennek a függvénynek nem kell a teljes S-gráfot bejárnia, az összes kapacitást nem szükséges újraszámolnia, hanem csak a paraméterben megadott taszkokhoz, és az ezt követő taszkokhoz tartozó kapacitásokat kell újraszámolnia. Az ezt követő taszkokat úgy kell értelmezni, hogy a megadott taszk szomszédjait, valamint azoknak a szomszédjait (így tovább egészen addig, amíg létezik egy szomszéd taszk) kell átvizsgálni és szükség esetén megváltoztatni, módosítani a kapacitásukat.

Az előző pontban bemutatott függvénytől eltérően nem az S-gráf bemeneti node-jait, csomópontjait reprezentáló taszkokat kell először megkeresni, hanem a paraméterlistában átadottat, valamint annak közvetlen szomszédjait. Ehhez is a **deque-t** veszünk igénybe. Legelsőnek az átadott taszk kerül bele, majd *for* ciklus segítségével megkeresi annak szom-

szédjait, és ezeket is a **deque** végéhez hozzáfűzi. Ezek után meg kell keresni a többi olyan taszkot is, amelyeknek a kapacitását újra át kell vizsgálni, és ha szükséges módosítást végezni. Azt követően, hogy a két végű sor tartalmazza az összes átvizsgálandó taszkot megtörténik a tényleges kapacitásmódosítás. Itt *while* ciklus felhasználásával addig történik az ellenőrzés, amíg teljesen üressé válik a **deque**. Ebből kivételre kerül a legelső elem, és megvizsgáljuk, hogy van-e ebbe tartó él, vagyis az S-gráf kezdő csomópontja vagy sem. Későbbiekben lesz szerepe ennek. Az éppen vizsgált taszk kapacitását átállítjuk nullára, majd a még hozzárendelhető berendezések kapacitásának összegét megkapja, mint új értéket. Amennyiben a taszknak nincs bejövő éle, akkor az imént meghatározott kapacitása megmarad, nincs szükség további ellenőrzésekre. Ellenben, ha van bemenő él, akkor még további feltételekre meg kell vizsgálni. Ha az a taszk, amelyből az él érkezik még nem ellenőrzött, akkor nem lehetséges a mostani taszk kapacitásának pontos meghatározása sem, ezért visszakerül a **deque** végére. Azonban ha ellenőrzött a vizsgált taszkot megelőző részfeladat, akkor az előző pontban feltüntetett képlet szerint ki kell számolni a kapacitást. Ha ez nagyobb, mint a beállított, akkor ezt kapja meg a taszk új értéként. Ellenkező esetben pedig marad a már meglévő érték. Ezeket követően szükséges még egy *for* ciklus segítségével végigmenni a taszkokon, annak érdekében, hogy ha létezik megtalálja az összes szomszédját az imént vizsgált taszknak. Ha talált ennek megfelelő taszkot akkor a **deque** végéhez hozzáadjuk.

Utolsó lépés a függvényben a profit korlát kiszámítása. Ez teljes mértékben megegyezik az előző pontban szereplő függvény befejező lépésével. Az S-gráfon a termék előtt szereplő utolsó részfeladat kapacitása szükséges a korlát kiszámításához. Azért, hogy megtaláljuk ezt a taszkot szükség van arra, hogy két darab *for* ciklus bejárja a termékeket és a taszkokat. Ha megtalálta akkor annak kapacitása és a terméken szerzett jövedelem szorzata megadja a korlátot.

#### 5.4.4. Egyéb új metódusok

Az **IsProfitMaximization** függvény megadja, hogy a megoldó szoftver indításakor az új módszer került meghívásra parancssori paraméterek által. Olyan esetekben kerül meghívásra, amelyeket csak abban az esetben kell végrehajtani, ha az új – taszkok

párhuzamos végrehajtására alkalmas – módszer lett meghívva. Például a futás végén a fájlba írásnál kapacitásokat csak ennél a módszernél akarunk kiírni.

A másik egyszerűbb függvény a **GetTaskCapacity**. Paraméterként egy részfeladat azonosítóját várja, és ez alapján visszaadja az adott taszkhoz tartozó kapacitást.

## 5.5. Argumentum hozzáadás

Az új módszer elkészítése miatt szükség volt új argumentumok hozzáadására, amelyeket az **arguments.cpp** fájlban lehet elérni. Ez a fájl az **src\solver** mappában található. Két argumentum került hozzáadásra. Az első **flexbatch**, amely a **method** kapcsolóhoz tartozik, ezzel a megoldó módszert lehet kiválasztani. A második új argumentum a **pro-fit-max**, amely a **obj** kapcsolóhoz tartozik, ez pedig a célfüggvényt reprezentálja.

## 5.6. Megoldás fájlba írása

Az új módszerrel kapcsolatos kapacitások kezelése eddig nem volt része a solver megoldó szoftvernek. Ez alól a fájlba történő kiírásuk sem kivétel, emiatt szükség volt a már meglévő kiírást elvégző függvény módosítására. A szóban lévő függvény a **WriteText**, amely **SGraph** típusú változót vár paraméterként. A függvény **solutionwriter.cpp** fájlban található, míg a deklarációja a **solutionwriter.h** fájlban helyezkedik el. Ezeket a fájlokat a **src\lib** mappában találhatjuk meg.

Az eredetileg meglévő függvény a fájlba két elkülöníthető rész kiírását végezte. Az első az volt, amely megmutatta az éleket. Ebbe beletartozik mind a recept, mind az ütemezési élek csoportja. Ezenfelül még megjelenik itt egy időérték, amely azt mutatja, hogy az adott taszkot, amelyből az él kiindul mennyi idő alatt lehet befejezni, elvégezni. Továbbá a boundot, korlátot is itt írja ki a fájlba a függvény. A második fele pedig az elvégzett feladathoz tartozó Gantt diagramot jeleníti meg karakteres formában. Ez a rész megadja, hogy melyik taszkot, melyik berendezés végezi el és, hogy ez mikor történik.

Az említett két rész közé került az általam elkészített kapacitások kiírására szolgáló rész. Pontosabban, mivel az új módszer kapacitás és jövedelem szorzataként adja meg a



korlátot ezért, a rész a kapacitások után jelenik ezentúl meg. Először a taszkok kapacitása kerül kiírásra, ezeket követik a termékek, amelyekhez az utolsó taszk kapacitása és a termékből származó jövedelem adja meg az értéket. A korlát kiszámítása a termékekhez tartozó értékek összeadásával történik.

## 6. fejezet

# Tesztelés

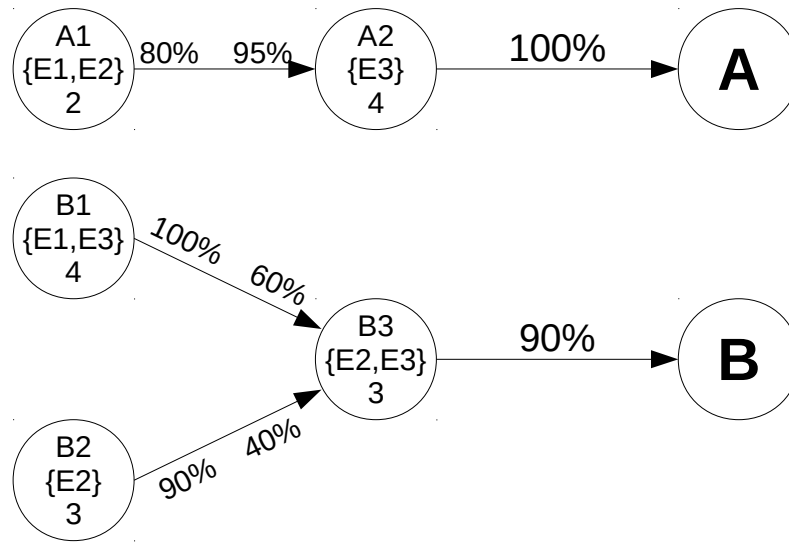
Ebben a fejezetben az elkészített megoldó algoritmus tesztelését mutatom be, hogy az megfelelően, hiba nélkül képes a feladatokat megoldani. A tesztelés a következő konfigurációval rendelkező számítógéppel történt:

- Processzor: Intel i5-7200, 2,50 Ghz
- 8 GB RAM
- Operációs rendszer: Windows 10
- Fejlesztésnél használt szoftverek:
  - Qt Creator 4.7.1
  - Qt 5.11.2
  - Boost Libraries 1.68.0
  - Microsoft Visual C++ Compiler 15.0

Az S-gráf megoldó szoftvert parancssori paraméterek segítségével lehet működtetni. A különböző megoldó módszereket, amelyek megtalálhatóak a szoftverbe implementálva, más és más kapcsolók segítségével lehet elérni, meghívni. Ezeknek listája megtalálható a **solver** mappában lévő **README.md** fájlban. Az általam megvalósított módszerhez a következő kapcsolókat mindenképpen használni kell, hogy a program hiba nélkül fusson és elvégezze az ütemezést:

- **-i extended\_precedential.ods:** A bemeneti fájl elérési útvonalát kell megadni ezzel.
- **-o output.txt:** A kimeneti fájl elérési útvonala. Két fajta kiterjesztésű fájlt lehet megadni: **TXT** és **PNG**. Előbbi esetében a fájlba kerülnek az élek, mind a receptek, mind az ütemezési élek, valamint, hogy mennyi ideig tart a részfeladat befejezése, amelyből ezek kiindulnak. Ezenfelül a Gantt diagram karakteres formában is megjelenik. PNG kiterjesztésű fájl megadása esetén pedig kirajzolásra kerül egy Gantt diagram. Azonban, ha mindkét kiterjesztésű fájlra szükség van, akkor erre a **-g** kapcsoló segítségével van lehetőség. Ehhez a kapcsolóhoz kell a PNG kiterjesztésű fájl nevét megadni.
- **-m flexbatch:** Ezzel a kapcsolóval a megoldó módszert lehet kiválasztani. A *flexbatch* határozza meg, hogy az általam megvalósított algoritmus kerüljön meghívásra.
- **-timehor:** Időhorizont megadása történik ezzel a kapcsolóval.
- **-obj profit\_max:** A célfüggvény kiválasztása, ebben az esetben az újonnan a megoldó szoftverhez hozzáadott profit maximalizálás kerül kiválasztásra.
- **-precycle off:** A precycle az ütemezés gyorsításra szolgál, mégpedig úgy, hogy előre lefut, és kört keres a gráfban. Az új módszer esetén hibásan működik, nem összeegyeztethető azzal, ezért szükséges a kikapcsolása.
- **-nopresolvers:** Hasonlóan az előzőhöz a presolver is az ütemezést gyorsítja, de nem egyeztethető össze az új megoldó módszerrel, emiatt kell mindenképpen inaktívvá tenni.

A 6.1 ábrán látható mintafeladat alapján kerül bemutatásra a megoldó módszer. Két termékhez tartozó receptet látunk, amely a taszkokat, az ezeket megvalósítani képes berendezéseket, és az ehhez szükséges időt tartalmazza. Az éleken megfigyelhető még, hogy a kapacitások hány százaléka kerül továbbadásra, illetve a következő taszk által felvételre.



6.1. ábra. Tesztfeladat

## 6.1. A tesztfeladat megoldása

A módszer sajátossága, hogy a receptélekkel az egyes taszkok kapacitásának meghatározott százaléka hasznosítható. Ezeket a százalékokat a bemeneti fájlban kell megadni. A példafeladat a 6.2 ábrán megtekinthető bemeneti adatokkal rendelkezik. Az időhorizont a bemutatott mintafeladat során 15. Látható, hogy az egyes részfeladat által lehetséges kapacitásoknak nem a teljes mennyisége kerül tovább a következő taszkhoz. Az, hogy kisebb mennyiség kerül felhasználásra nagy mértékben befolyásolja a korlátot. Emellett az ütemezés megoldását is befolyásolja, mert emiatt lehetséges, hogy bizonyos berendezések nem végezhetik el az adott taszkot.

Az egyszerűség kedvéért a példában csak 2 darab termék receptje szerepel. Az A termék legyártásából származó jövedelem egy egység, a B termék esetén pedig 2 egység. A *precedence* táblában látható, hogy a taszkok által elérhető mennyiségek nem 100 százalékban kerülnek további felhasználásra. Például az A1 és A2 taszkok esetében, az A1-es taszk mennyiségének 80 százaléka kerül továbbításra, illetve ennek a már kisebb mennyiségnek

a 95 százalékát veszi fel az A2-es részfeladat. A *proctime* táblán megtalálhatjuk, hogy melyik taszkt, melyik berendezés tudja elvégezni, valamint ezt mennyi idő alatt teszi.

product			
name	number	revenue	p_percent
	1		
A	1	1	100
B	1	2	90

task	
name	pr_name
	<product name>
A1	A
A2	A
B1	B
B2	B
B3	B

equipment		
name	number	b_capacity
	1	
E1	1	90
E2	1	70
E3	1	50

precedence			
task1	task2	s_percent	d_percent
<task name>	<task name>		
A1	A2	80	95
B1	B3	100	60
B2	B3	90	40

proctime		
task_name	eq_name	time
<task name>	<equipment name>	infy.
A1	E1	2
A1	E2	2
A2	E3	4
B1	E1	4
B1	E3	4
B2	E2	3
B3	E2	3
B3	E3	3

6.2. ábra. A tesztfeladat bemeneti adatai

A feladat megoldását tartalmazó TXT kiterjesztésű fájlt három darab elkülöníthető részre tudjuk felosztani. Az első része látható a 6.3 ábrán. Az ábra elején az éleket találhatjuk meg, mégpedig olyan formában, hogy melyik taszkból melyik taszkba mutat. Továbbá láthatóak időértékek, amelyek megadják, hogy az élt megelőző részfeladatot mennyi idő alatt lehet elvégezni. Ez a rész tartalmazza mind a receptéleket, mind az ütemezési éleket. Úgy tudjuk ezeket elkülöníteni, hogy amelyekhez 0 időérték van rendelve, azok tartoznak az ütemezési élek csoportjához. Mivel az egyik termékből, nevezetesen az B-ből, több mint egy darabot gyártunk ezért megkülönböztetjük az egyes receptekhez tartozó feladatokat. Láthatunk B1-et és B1\_2-t. Ugyanolyan típusú feladatról beszélünk, de különböző recepthez tartoznak, ezért kell megkülönböztetni egymástól ezeket.

```

A1 ---> A2  2
A2 ---> B1  0
A2 ---> A   4
B1 ---> B3  4
B1_2 ---> B3_2  4
B2 ---> B3  3
B2_2 ---> B2  0
B2_2 ---> B3_2  3
B3 ---> B   3
B3_2 ---> A1  0
B3_2 ---> B_2  3
A ---> B3  0
B_2 ---> A2  0
B_2 ---> B2  0

```

6.3. ábra. A megoldást tartalmazó fájl első része

A második részben a taszkok és a hozzájuk tartozó kapacitások, valamint a termékekhez tartozó kapacitások és a termékek előállításából származó jövedelem szorzata található. Ezeket követően szerepel a bound, a korlát, ami az adott feltételek és adatok mellett 482 lett. Ezt úgy kapjuk meg, hogy a termékekhez tartozó értékeket összeadjuk. Jelen példa esetében 3 érték kerül összeadásra, ezek a következők: A, B és B\_2. Az A termékből származó jövedelem 50, a B és a B\_2 termékek esetén 216. Ezeket összeadva kijön a 482. Ezek az értékek a 6.4 ábrán megtekinthetők.

```

A1: 90
A2: 50
B1: 90
B1_2: 90
B2: 70
B2_2: 70
B3: 120
B3_2: 120
A: 50
B: 216
B_2: 216
bound: 482

```

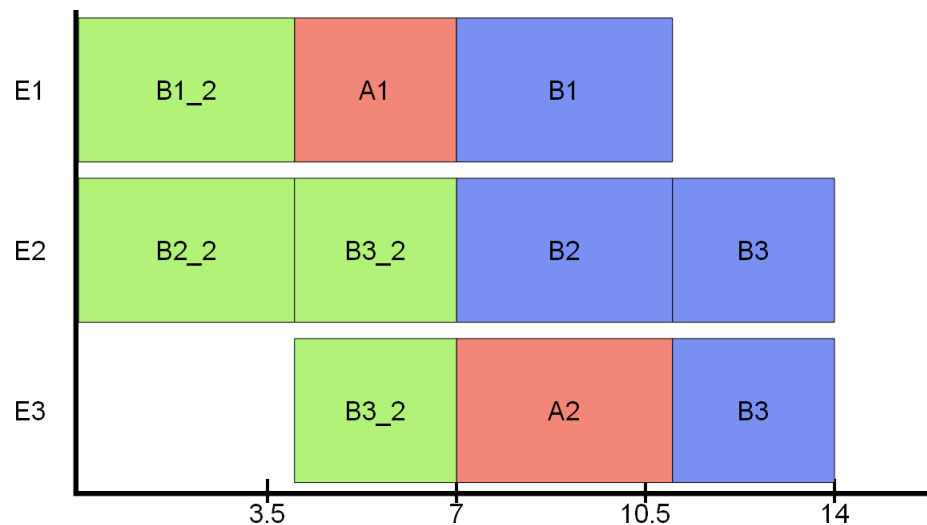
6.4. ábra. A megoldást tartalmazó fájl második része

Az utolsó szakaszban a Gantt diagram karakteres formában található meg. Látható, hogy vannak olyan taszkok, amelyeket több berendezés is el tud végezni. A 6.5 ábrán látható ez.

```
Gantt chart:
A - A1: 4-7, 3 E1
A - A2: 7-11, 4 E3
B - B1: 7-11, 4 E1
B_2 - B1_2: 0-4, 4 E1
B - B2: 7-11, 4 E2
B_2 - B2_2: 0-4, 4 E2
B - B3: 11-14, 3 E2 E3
B_2 - B3_2: 4-7, 3 E2 E3
The longest path: 14
```

6.5. ábra. A megoldást tartalmazó fájl harmadik része

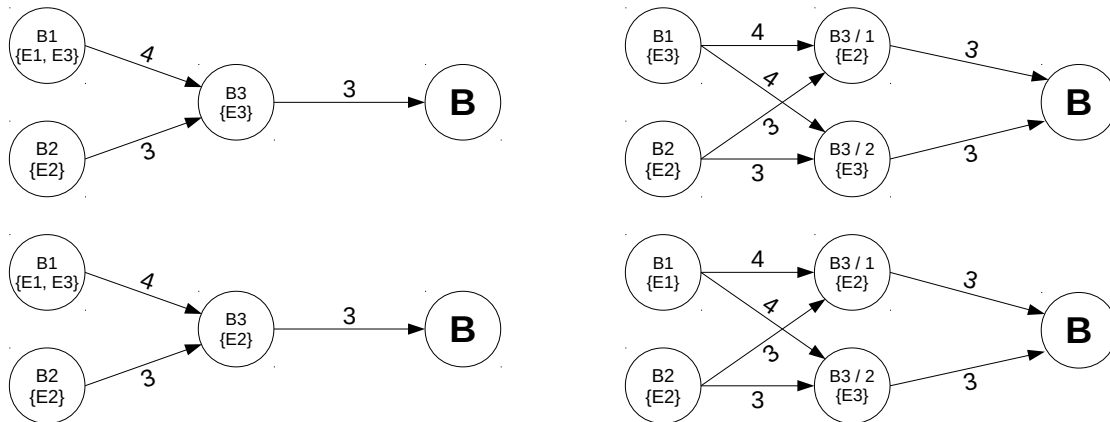
A 6.6 ábrán pedig a PNG kiterjesztésű fájl látható. Azonos színnel jelölt taszkok tartoznak egy termékhez. Ha az egyik termékből több példány készül, akkor a termékhez tartozó taszkok végéhez illesztett szám mutatja meg, hogy melyik termékhez tartoznak ezek. A diagramon könnyen észrevehető az új módszerben lévő újítás. A zölddel jelzett B3.2-es taszk, valamint a kék színű B3-as taszk két berendezéshez, nevezetesen az E2 és az E3 berendezéshez is hozzá lett rendelve.



6.6. ábra. A solver által elkészített Gantt diagram

## 6.2. A régi és az új megoldó összehasonlítása

Ebben az alfejezetben az eddig meglévő throughput maximalizáló megoldót hasonlítom össze az általam létrehozottal. Ehhez 6.1 ábrán látható feladatot veszem igénybe. Mivel ebben a feladatban a termékek változó batch mérettel rendelkeznek, ezért a régi megoldó módszerhez szükség van a 3. fejezetben bemutatott diszkretizálás végrehajtására. Ezt elvégezve eredményül azt kapjuk, hogy az A termék gyártása esetén mindenképpen 50 jövedelemre tudunk szert tenni. B termékhez viszont 4 különböző recept jött létre, amelyek mind eltérő jövedelmet biztosítanak. Ezek alapján felrajzolható a 4 gráf, amelyek a 6.7 ábrán láthatóak.



6.7. ábra. Régi módszer esetén a B termékhez tartozó 4 gráf

Az ábra bal oldalán szereplő gráfok megegyeznek a példafeladaton szereplő gráffal. Eltérés csak a taszkokat elvégezni tudó berendezésekben van. Azokat a taszkokat, amelyeket 2 darab berendezés is el tud végezni, úgy kell figyelembe venni, hogy vagy az egyik vagy a másik berendezés végezi majd el az ütemezés után. A jobb oldalon látható, hogy a B3-as taszkból kettő darab lesz. Ezek az estek az jelentik, hogy mindkét berendezés elvégzi ezt a részfeladatot. A régi módszer esetén úgy tekintünk a termékekre, mintha 5 különböző termék lenne. Ezt az 1 darab A termék receptje és a 4 darab különböző B termék receptje adja meg. Ezek alapján elkészíthető a régi megoldóhoz szükséges bemeneti



fájl, amely a következő ábrán látható.

product			task		precedence		proctime		
name	number	revenue	name	pr_name	task1	task2	task_name	eq_name	time
	1			<product.name>	<task.name>	<task.name>	<task.name>	<equipment.name>	infly
A	1	50	A1	A	A1	A2	A1	E1	2
B	1	90	A2	A	B1	B3	A1	E2	2
BB	1	126	B1	B	B2	B3	A2	E3	4
BBB	1	150	B2	B	BB1	BB3	B1	E1	4
BBBB	1	216	B3	B	BB2	BB3	B1	E3	4
			BB1	BB	BBB1	BBB31	B2	E2	3
			BB2	BB	BBB1	BBB32	B3	E3	3
			BB3	BB	BBB2	BBB31	BB1	E1	4
			BBB1	BBB	BBB2	BBB32	BB1	E3	4
			BBB2	BBB	BBBB1	BBBB31	BB2	E2	3
			BBB31	BBB	BBBB1	BBBB32	BB3	E2	3
			BBB32	BBB	BBBB2	BBBB31	BBB1	E3	4
			BBBB1	BBBB	BBBB2	BBBB32	BBB2	E2	3
			BBBB2	BBBB	BBBB31	BBBB32	BBB31	E2	3
			BBBB31	BBBB	BBBB2	BBBB32	BBB32	E3	3
			BBBB32	BBBB			BBBB1	E1	4
							BBBB2	E2	3
							BBBB31	E2	3
							BBBB32	E3	3

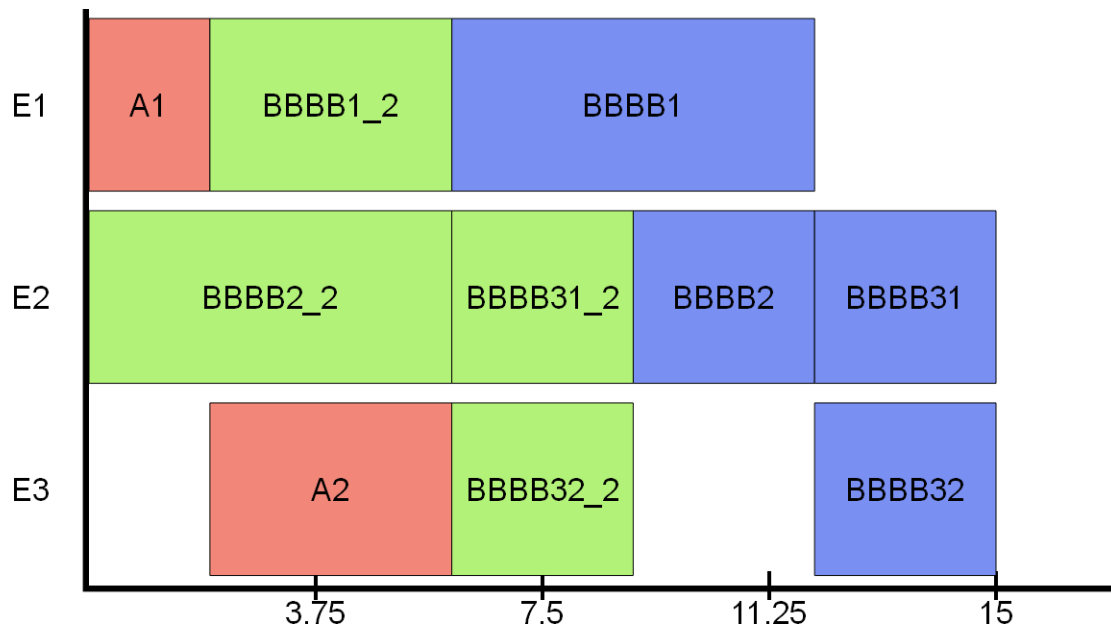
  

equipment	
name	number
	1
E1	1
E2	1
E3	1

6.8. ábra. A régi megoldó bemeneti adatai

Látható, hogy ilyen típusú feladat esetében jóval több adat van a fájlban. Ennek az oka a több recept, ami több taszkot és élt jelent. Itt az 5 darab termék miatt egy 5 dimenziós teret lehet elképzelni. Az új megoldó módszernél ez csak 2 dimenziós tér lesz. Láthatjuk, hogy kevesebb dimenziót kell megvizsgálni, viszont egy dimenzió belül több számítást végez el az algoritmus a különböző hozzárendelési lehetőségek miatt.

Az algoritmus lefutásának végeztével azt várjuk, hogy a régi és az új megoldó ugyanazt az eredményt szolgáltatassa. Pontosítva a jövedelem mennyisége és a legyártott termékeknek meg kell egyezniük, a taszkok ütemezésétől azonban ezt nem várjuk el. A 6.9 ábrán látható a régi megoldómódszer által elkészített Gantt diagram. Összehasonlítva a 6.6 ábrán látható diagrammal meg lehet állapítani azt, hogy mindkét esetben 2 B termék és 1 darab A terméket lehet legyártani a megadott időhorizonton belül. A másik fontos dolog, hogy a jövedelem is megegyezzen. Az új megoldónál ez 482 volt. A régi által nyújtott eredményben az A termék jövedelme 50, a két B termékből pedig olyanokat gyárt, amelyeknek 216 a jövedelme. Ezeket összeadva,  $2 * 216 + 50$ , itt is kijön a 482. Az ütemezés során a taszkok sorrendje a két megoldó módszer esetén eltér, azonban ez nem hiba. Lehetnek olyan feladatok, ahol ez is teljes mértékben megegyezik.



6.9. ábra. A régi megoldó által nyújtott Gantt diagram

### 6.3. A megoldó módszerek sebessége

A következő táblázatban a megoldó módszerek sebességének összehasonlítása látható. A régi módszerbe már több, különböző gyorsítási megoldás be van építve. Ezek közé tartozik a korábban már említett precycle és presolver. Az új módszer esetén ezek a gyorsítások jelen állapotban nem működnek. Egy másik kiemelt fontosságú gyorsítás a régi megoldó módszer esetében a revenue line. Ez alsó korlátként szolgál az egyes konfigurációkból származó jövedelmek összehasonlításához. Ennek segítségével, feladattól függően, nagy számú konfiguráció eltávolítható a keresési térből, így jelentősen csökkenthető a számítások mennyisége az ütemezés során. Az új módszer használata során azonban ezt nem vehetjük igénybe, mert nem egy egyenesen szerepelnek azok a konfigurációk, amelyek azonos jövedelemmel rendelkeznek.

6.1. táblázat. Különböző módszerekkel megoldott feladatok összehasonlítása

	Megoldó módszer	Futás ideje	Time Horizon	Legyártott termék	Profit
Feladat 1	Új	1,057 mp	15	1A+2B	482
	Régi	0,149 mp	15	1A+2B	482
	Régi (gy. n.)	0,185 mp	15	1A+2B	482
	Új	94,964 mp	20	1A+3B	698
	Régi	8,17 mp	20	1A+3B	698
	Régi (gy. n.)	13,553 mp	20	1A+3B	698
	Új	>8000 mp	25	4B	864
	Régi	705,439 mp	25	4B	864
	Régi (gy. n.)	1175,11 mp	25	4B	864
Feladat 2	Új	0,371 mp	15	2C+1D	188
	Régi	0,07 mp	15	2C+1D	188
	Régi (gy. n.)	0,113 mp	15	2C+1D	188
	Új	15,486 mp	20	3C+1D	252
	Régi	1,306 mp	20	3C+1D	252
	Régi (gy. n.)	3,201 mp	20	3C+1D	252
	Új	741,29 mp	25	3C+3D	360
	Régi	58,28 mp	25	3C+3D	360
	Régi (gy. n.)	222,679 mp	25	3C+3D	360
Feladat 3	Új	14,78 mp	10	2H+2I	86
	Régi	0,374 mp	10	2H+2I	86
	Régi (gy. n.)	0,381 mp	10	2H+2I	86
	Új	44,693 mp	15	4I	94
	Régi	0,985 mp	15	4I	94
	Régi (gy. n.)	0,806 mp	15	4I	94

Ugyanaz a feladat 3 módszerrel került megoldásra. A régivel, amelyben szerepelnek a gyorsítások, ismét a régi megoldóval (réggy. n.), de itt a precycle és a presolver gyorsítások nélkül, illetve az elkészített új megoldó módszerrel. Látható, hogy a régi módszerek gyorsabban találják meg a feladat legjobb megoldását, mint az új a jelenlegi állapotában. Jövőbeni tervek közé tartozik a meglévő gyorsítások átalakítása, hogy kompatibilisek legyenek az új módszerrel is. Ezek megvalósítása után nagy csökkenés várható az új megoldó módszer futási idejében.

## 7. fejezet

# Összefoglalás

A dolgozatomban a throughput maximalizálás szakaszos üzemű rendszerekben témakörrel foglalkoztam. Először az ütemezéssel kapcsolatos irodalmat tanulmányoztam, amelynek során megismerkedtem az ipari környezetben használt ütemezési megoldó módszerekkel. Ezek közül az S-gráf keretrendszer lett az, amely a dolgozatom alapját képezi. Ezt követően megismerkedtem a a makespan minimalizálás és a throughput maximalizálás algoritmusával. Ezek megismerése adta meg az alapot, hogy elkészülhessen a flexibilis batch mérettel rendelkező feladatok megoldására alkalmas módszertan. Ez egy már meglévő S-gráf keretrendszert megvalósító szoftverbe történő implementálással valósult meg. Az implementáció megvalósítását követően tesztelésnek vetettem alá a bővített rendszert. A tesztelés eredményei bizonyították, hogy megfelelően működik a megoldó módszer, mivel jó eredményeket szolgáltat. Ennek az új módszer segítségével idő spórolható meg az előfeldolgozó lépések során, mert már nem szükséges a diszkretizációs folyamat elvégzése. Azonban a futási sebessége nem hozott jobb eredményeket, mint a régi megoldó módszer. A jövőbeni tervek közé tartozik ennek az időnek a csökkentése. Ehhez olyan keresési módszereket kell kifejleszteni, amelyekkel csökkenthető a keresési térben lévő konfigurációk száma, illetve egy adott konfiguráción belüli számítások csökkentésére alkalmasak.

# Irodalomjegyzék

- [1] M. Hegyháti, „Batch process scheduling: Extensions of the s-graph framework,” 2015.
- [2] E. Kondili, C. Pantelides, and R. Sargent, „A general algorithm for short-term scheduling of batch operations—i. milp formulation,” *Computers & Chemical Engineering*, vol. 17, no. 2, pp. 211 – 227, 1993. An International Journal of Computer Applications in Chemical Engineering.
- [3] J. L. N. Susarla and I. A. Karimi, „A novel approach to scheduling multipurpose batch plants using unit-slots,” pp. 1859–1879, 2010.
- [4] C. Cassandras and S. Lafortune, „Introduction to discrete event systems,” 2008.
- [5] P. L. M. Ghaeli, P. A. Bahri and T. Gu, „Petri-net based formulation and algorithm for short-term scheduling of batch plants,” pp. 249–259., 2005.
- [6] E. Sanmarti, F. Friedler, and L. Puigjaner, „Combinatorial technique for short term scheduling of multipurpose batch plants based on schedule-graph representation,” *Computers & Chemical Engineering*, vol. 22, pp. S847 – S850, 1998. European Symposium on Computer Aided Process Engineering-8.
- [7] E. Sanmarti, T. Holczinger, L. Puigjaner, and F. Friedler, „Combinatorial framework for effective scheduling of multipurpose batch plants,” *AIChE Journal*, vol. 48, no. 11, pp. 2557–2570, 2002.
- [8] H. L. Gantt, *Work Wages and Profits*. New York, The Engineering magazine co., 1913.

- [9] H. L. Gantt, *Organizing for work*. New York : Harcourt, Brace and Howe, 1919.
- [10] T. Majozi and F. Friedler, „Maximization of throughput in a multipurpose batch plant under a fixed time horizon: S-graph approach,” *Industrial & Engineering Chemistry Research*, vol. 45, no. 20, pp. 6713–6720, 2006.
- [11] T. Holczinger, T. Majozi, M. Hegyhati, and F. Friedler, „An automated algorithm for throughput maximization under fixed time horizon in multipurpose batch plants: S-graph approach,” vol. 24, pp. 649 – 654, 2007.

# A. függelék

## Tesztesetek

product		
name	number	revenue
	1	
A	2	10
B	1	5

task	
name	pr_name
	<product name>
A1	A
A2	A
A3	A
B1	B
B2	B

equipment		
name	number	b capacity
	1	
E1	1	40
E2	1	60

precedence			
task1	task2	s_percent	d_percent
<task name>	<task name>		
A1	A2	100	50
A2	A3	50	100
B1	B2	50	50

proctime		
task_name	eq_name	time
<task name>	<equipment name>	infity
A1	E1	2
A1	E2	2
A2	E1	3
A3	E2	2
B1	E1	3
B1	E2	3
B2	E1	5
B2	E2	5

A.1. ábra. Nyolcadik teszteset bemeneti adatai

A.1. táblázat. Második példafeladat tesztéseinek második fele

Fájl név	Időhorizont	A termék	B termék	Kapacitások	Hozzárendelések	Bevétel	Megoldás ideje	Gantt
teszt06	25	2 db	1 db	A1: 100 A1_2: 100 A2: 40 A2_2: 40 A3: 40 A3_2:40 B1: 100 B2: 100	E1: A1, A1_2, A2, A2_2 B1, B2 E2: A1, A1_2, A3, A3_2, B1, B2	1300	0,162 sec	er06
teszt07	20	2 db	1 db	A1: 60 A1_2: 100 A2: 40 A2_2: 40 A3: 20 A3_2:20 B1: 60 B2: 60	E1: A1_2, A2, A2_2 E2: A1, A1_2, A3, A3_2, B1, B2	700	0,844 sec	er07
teszt08	25	2 db	1 db	A1: 100 A1_2: 100 A2: 40 A2_2: 40 A3: 20 A3_2:20 B1: 100 B2: 100	E1: A1, A1_2, A2, A2_2 B1, B2 E2: A1, A1_2, A3, A3_2, B1, B2	900	0,220 sec	er08



## B. függelék

### CD Melléklet tartalma

**solver** mappa: Az S-gráf megoldó szoftver forrásfájljait tartalmazza.

**Felkészüles** mappa: A keretrendszer megismerése során kézzel elkészített feladatok (makespan minimalizálás, throughput maximalizálás).

**Teszteles** mappa: A dolgozatban említett tesztfeladatok bemeneti adatait és megoldásait tartalmazó fájlok.

**Dolgozat** mappa: A dolgozat PDF változata.