



**SZÉCHENYI**  
ISTVÁN  
**EGYETEM**



# **SZAKDOLGOZAT**

## **Throughput maximalizálás szakaszos üzemű rendszerekben**

**Molnár Gergő**

**Mérnök Informatikus BSc szak**

**2019**

# Nyilatkozat

Alulírott, Molnár Gergő (RV3N4S), Mérnök Informatikus, BSc szakos hallgató kijelentem, hogy a Throughput maximalizálás szakaszos üzemű rendszerekben című szakdolgozat feladat kidolgozása a saját munkám, abban csak a megjelölt forrásokat, és a megjelölt mértékben használtam fel, az idézés szabályainak megfelelően, a hivatkozások pontos megjelölésével.

Eredményeim saját munkán, számításokon, kutatáson, valós méréseken alapulnak, és a legjobb tudásom szerint hitelesek.

Győr, [beadás dátuma]

---

hallgató

# Kivonat

## Throughput maximalizálás szakaszos üzemű rendszerekben

**Szerző:** Molnár Gergő, mérnökinformatikus BSc

**Témavezető:** Dr. Hegyháti Máté, tudományos főmunkatárs

**Munka helyszíne:** Széchenyi István Egyetem, Informatika tanszék

Gyártórendszerek optimális ütemezéséhez több módszer is rendelkezésre áll, mint például MILP modellek, vagy az általam részletesebben tanulmányozott S-gráf módszertan. Ezeknek közös jellemzője, hogy nagyméretű gyártási feladatok során nem képesek az optimális megoldást megtalálni elfogadható időn belül.

Munkám során azzal foglalkoztam, hogy a szakaszos üzemű rendszerek esetében hogyan lehet a különböző kapacitással rendelkező berendezéseket egyazon feladatokhoz hozzárendelni. Azért erre a témára esett a választásom, mert előfeldolgozó lépés nélküli algoritmus még nem volt kidolgozva és a keretrendszerbe implementálva.

A dolgozatomban szemléltetem a megismert megoldó módszereket. Kiemelt figyelmet kap az S-gráf keretrendszer, valamint az ehhez kapcsolódó algoritmusok. Ezeket követően bemutatom az új algoritmust, majd az implementálás részleteit. Az implementáció során a korábban az S-gráf keretrendszerhez elkészített kód refaktorálássára is sor került, hogy az új módszert is megfelelően lehessen végrehajtani.

Az algoritmus helyes működését tesztfeladatokkal szemléltetem, melyek során különböző, a végeredményeket befolyásoló adatok megváltoztatásának hatását bemutatom.

Munkám eredményeképpen az S-gráf keretrendszeren alapuló megoldó szoftver képes párhuzamos hozzárendeléseket megengedő feladatok megoldására.

**Kulcsszavak:** ütemezés, S-gráf, flexibilis batch, profit maximalizálás, optimalizálás

# **Abstract**

## **Throughput maximization in batch production systems**

There are many available methods for optimizing the schedule of manufacturing systems, such as MILP models or the S-graph methodology, which I have studied in more detail. Their common feature is that they are not able to find the optimal solution within an acceptable amount of time during large-scale manufacturing tasks.

During my work I studied how is it possible to assign the units with different capacity to the same tasks in batch production systems. I have chosen this topic because an algorithm without preprocessor have not been developed and implemented in a framework yet.

In my thesis I demonstrate the solving methods which I got to know. The S-graph framework and the related algorithms get special attention. After that, I present the new algorithm and the details of the implementation. During the implementation the refactoring of the code took place which was previously prepared for the S-graph framework. The goal of it was that the new method can be properly executed.

I illustrate the correct operation of the algorithm with test cases, during which I demonstrate the effect of changing the data affecting the end results.

As a result of my work, the solver software based on the S-graph framework can solve the problems with parallel assignments.

# 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 . . . . .	7
2.2.1. MILP modellek . . . . .	7
2.2.2. Analízis alapú eszközök . . . . .	8
2.2.3. S-gráf módszertan . . . . .	8
2.2.4. A makespan minimalizálás algoritmusai . . . . .	12
2.3. Throughput vagy profitmaximalizálás . . . . .	13
<b>3. Probléma definíció</b>	<b>16</b>
<b>4. Az új módszer</b>	<b>20</b>
<b>5. Implementálás</b>	<b>22</b>
5.1. S-gráf solver . . . . .	22
5.2. Adatok beolvasása . . . . .	23
5.2.1. Bemeneti fájl . . . . .	23
5.2.2. Beolvasó függvény . . . . .	24
5.3. FlexBatchSchProblem osztály . . . . .	25
5.3.1. MakeDecisions függvény . . . . .	26
5.3.2. Branching függvény . . . . .	26
5.3.3. További metódusok . . . . .	27

5.4. SGraph osztály . . . . .	28
5.4.1. MakeMultipleBatches függvény . . . . .	29
5.4.2. UpdateProfitBound függvény . . . . .	32
5.4.3. UpdateProfitBoundFromTask függvény . . . . .	34
5.4.4. Egyéb új metódusok . . . . .	35
5.5. Argumentum hozzáadás . . . . .	35
5.6. Megoldás fájlba írása . . . . .	36
<b>6. Tesztelés</b>	<b>37</b>
6.1. Taszkok közti kapacitások változtatásának hatása . . . . .	39
6.2. Idő horizont változtatás . . . . .	43
<b>7. Összefoglalás</b>	<b>45</b>

# 1. fejezet

## Bevezetés

Az ütemezés feladatával az élet számos területén találkozunk, kezdve az egyszerű, hétköznapi problémáktól, mint például egy napon elvégzendő feladataink sorrendjének beosztásával kezdve, professzionális sportcsapatok heti edzésprogramjának kialakításán át, egészen az ipari üzemek működéséig, ahol a rendelkezésre álló berendezésekhez kell rendelni a előállítani kívánt termékeket. Bár az életünk különböző területein fellépő problémák különböznek egymástól, bizonyos mértékben hasonlóság is fellelhető közöttük. Különbség lehet az ütemezési feladatok célfüggvénye, továbbá 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, miközben a folyamat során megjelenő, fellépő korlátokat betartjuk, azokat nem sértjük meg. Az ipari ütemezés két leggyakoribb célja a makespan minimalizálás, és a throughput maximalizálás. A makespan kifejezést az idő minimalizálására, a throughput kifejezést pedig a profit maximalizálásra alkalmazzák az irodalomban.

Az ipari gyártási folyamatok ütemezésére különböző módszerek léteznek már. Ezek közé tartoznak a MILP megoldó módszerek, amelyek a lineáris programozáson alapulnak. Továbbá ide sorolható az időzített automaták, Petri hálók, valamint a S-gráf megoldómódszer, amely a munkám során a legfontosabb szerepet tölti be a felsorolt megoldó módszerek közül. Dolgozatom második fejezetében ezek kerülnek bemutatásra. A harmadik részben a probléma kerül definiálásra. A negyedikben az általam megvalósított

módszer elmélete található. A hatodik fejezetben a módszer megvalósítását mutatom be. A hetedik fejezetben tesztelésről, és az eredmények más módszerekkel való összehasonlításáról lesz szó. A dolgozat végén található az összefoglalás a munkámról, a hivatkozások, valamint a melléklet rész.



## 2. fejezet

# Irodalmi áttekintés

### 2.1. Ipari ütemezési feladatok

Az ipari folyamatokat 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. 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 hívunk. A problémák kimenetelük szerint lehetnek megvalósíthatatlan (infeasible) és megvalósítható (feasible) feladatok. Ha egy korlátozásnak sem felel meg az adott probléma akkor infeasible, egyéb esetekben pedig már megvalósítható lesz.

Az ipari folyamatokat többféleképpen lehet csoportosítani. Egy fajta az, amelyben 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. Az online ezzel szemben úgy működik, hogy előbb kell döntéseket meghozni, minthogy adott paraméterekhez tartozó értékekre fény derülne. A semi-offline

a kettő közé sorolható. Bizonyos információk, adatok már rendelkezésre állnak, mások viszont nem. Egy harmadik besorolási lehetőség, hogy megkülönböztetünk sztochaikus és determinisztikus feladatokat. Sztochaikus esetekben a paraméterek már futás közben kapnak értéket. Ezzel szemben a determinisztikus feladatok értékei előre meg vannak határozva és beállítva.

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

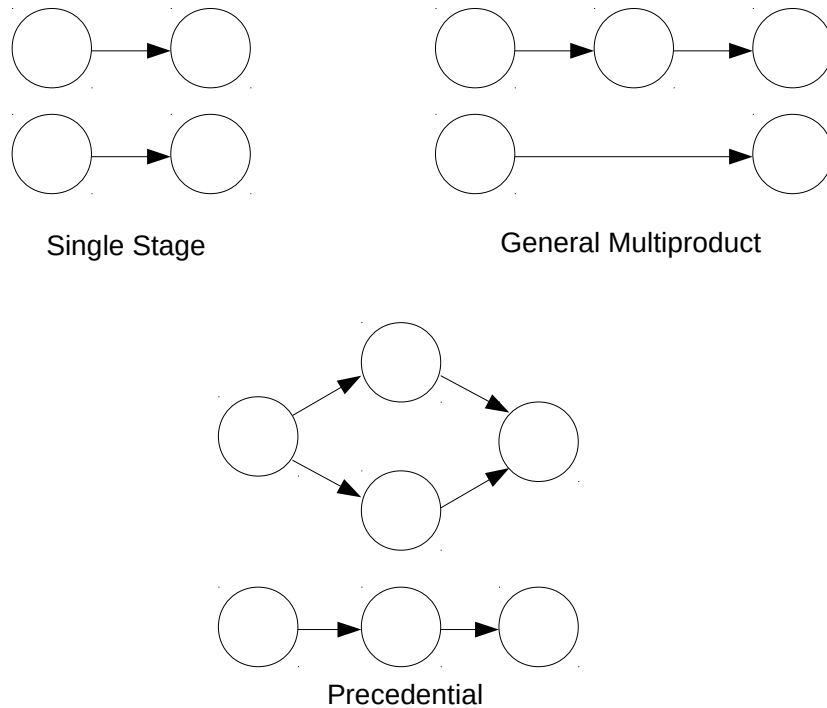
- Termékek listája
- Taszkok listája, amelyek adott sorrendben történő elvégzése szükséges a termékek előállításához
- Taszkok egymás közötti sorrendisége, amely megmutatja a taszkok sorrendjét
- Rendelkezésre álló berendezések
- A lehetséges taszk-berendezés párok feldolgozási ideje

A recepteket a feladatok precedenciájuk szerint következő csoportokba lehet besorolni. A felsorolás a legegyszerűbbtől halad az általánosabb felé. Ezen kívül 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.
- **General Multiproduct:** Előzővel összehasonlítva a különbség, hogy ennél lehetséges lépések kihagyása.
- **Multipurpose:** A termékek gyártásának lépéseit nem lehet egy balról jobbra tartó vonal mentén véghezvinni. A szakaszok száma és iránya tetszőleges, sőt egy szakasz többször is ismétlődhet ugyanabban a gyártásban.

- **Precedential:** A gyártásban résztvevő taszkok nincsenek hozzárendelve a szakaszokhoz. Egy termék gyártás 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. Ennél az esetén kör is lehetséges.

Néhány előbb említett feladattípus szemléltetése látható a 2.1 ábrán. A receptek jobb oldalán lévő kör jelenti a terméket, a többi pedig a taszkokat.



2.1. ábra. Különböző receptek szemléltetése

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

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

Az UIS eset a legegyszerűbb. Ebben az esetben van lehetőség a köztes anyagok bármely mértékű tárolására. FIS esetben van lehetőség a tárolása, 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 helyen várakozzon.

Második fajta csoportosítási lehetősége, amikor az idő ad korlátot.

- **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 legegyszerűbb mind közül. 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 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

A legtöbb ütemezési probléma modelljében vegyesen fordulnak elő folytonos és egész változók. Ilyen esetekben beszélünk vegyes egészcímű lineáris programozásról, azaz **Mixed Integer Linear Programming**. Több altípus létezik:

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

Minden időpontban bináris változók vannak hozzárendelve aszerint, hogy az adott időpillanatban kezdődik a feladat végrehajtása vagy nem. A bináris változók száma arányos lesz az időpontok számával. A szándék mindig megvolt olyan módszer kifejlesztésére, amelyben a szükséges időpontok száma minél kisebb legyen amellet, hogy megtalálja az optimális megoldást.

**Precedencia alapú modellek - Precedence based:** Ezeknél a módszereknél, szemben az időfelosztásos módszerekkel, nincs szükség az időhorizont diszkrétizációjára, 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. Alapvetően a multiproduct és multipurpose receptek esetében használható megfelelően, de kibővíthető, hogy a sokkal általánosabb precedential receptek is megoldhatók legyenek.

Ez a módszer kettő darab bináris változó 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 ugyan az a berendezés végzi el az  $i$  és  $i'$  taszkot, méghozzá úgy, hogy előbb az  $i$ -t teljesíti.

### 2.2.2. Analízis alapú eszközök

Az automatákat és Petri hálókat széles körben alkalmazzák diszkrét eseményrendszerek modellezésére [4]. Számos kísérletet tettek ezen eszközök modellezési teljesítményének kiterjesztésére annak érdekében, hogy batch folyamatok ütemezésére is alkalmassá tegyék. 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 algoritmus 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 bonyolultságos 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. Többen is foglalkoztak a témával, például Ghaeli [5], aki batch folyamatok ütemezésével foglalkozott.

### 2.2.3. S-gráf módszertan

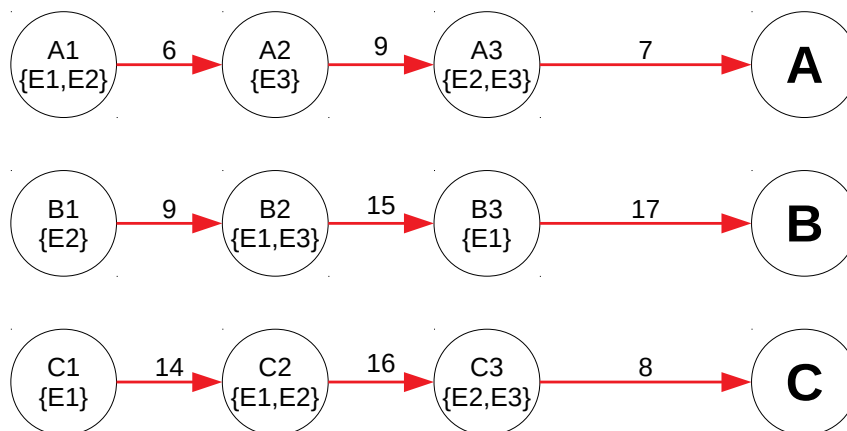
Az S-gráf keretrendszer volt az első olyan módszer, amely publikált gráf elméleten alapult, valamint szakaszos gyártórendszerek ütemezési problémák megoldására szolgált. [6] Ez a keretrendszer egy irányított gráf modellből, az S-gráfból, és a hozzá tartozó algoritmusokból áll. [7] Az S-gráf egy speciális irányított gráf, ami ütemezési problémák számára. 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. Ha két feladat között összeköttetés van, akkor ezt a gráfon lévő, feladatokat

reprezentáló csomópontok között lévő nyíl mutatja. Ütemezési döntés nélküli S-gráfot **Recept gráfnak** nevezzük. Erre példa a 2.2 ábrán [1] látható.

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 (nyilak) a csomópontok közti függőséget mutatják meg. Ezeket **Recept éleknek** nevezzük. Kétfajta függőséget tudunk megkülönböztetni:

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

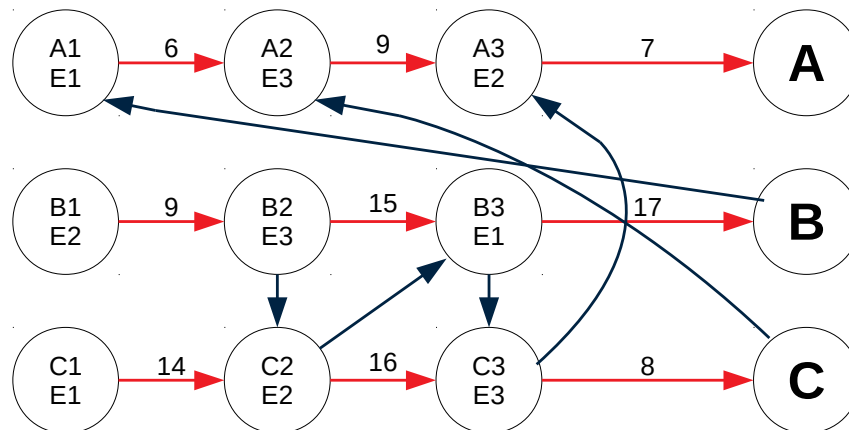
A nyilakon látható súlyok a részfeladat végrehajtásához szükséges időt mutatják meg. Ha egy részfeladatot több berendezés is képes elvégezni, akkor az előbb említett súly mindig a legkisebb előállítási idő lesz.



2.2. ábra. A recept gráf szemléltetése

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

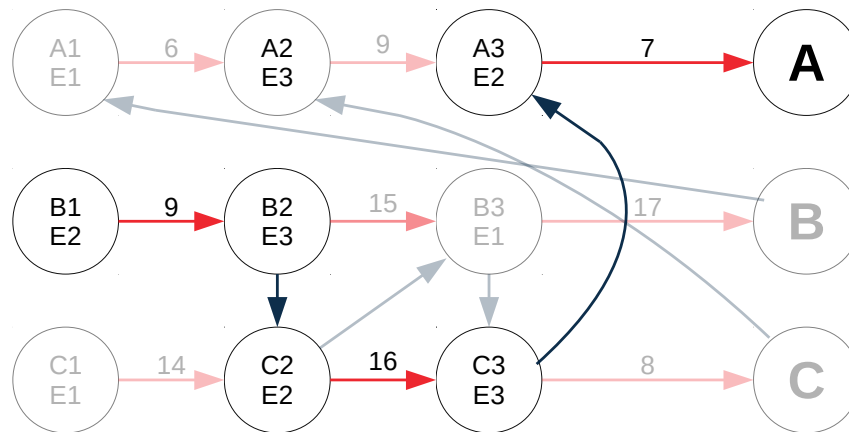
sötétkékekkel jelölt nyilak az ütemezési élek. Az ütemezési élek súlya alapértelmezetten nulla, ha a probléma nem tartalmaz szállítási, vagy tisztítási időt. A részfeladatok csomópontjain már nem a lehetséges berendezések halmaza látható, hanem egy konkrét kiválasztott berendezés, az ütemezési döntésnek megfelelően.



2.3. ábra. Az ütemezési gráf szemléltetése

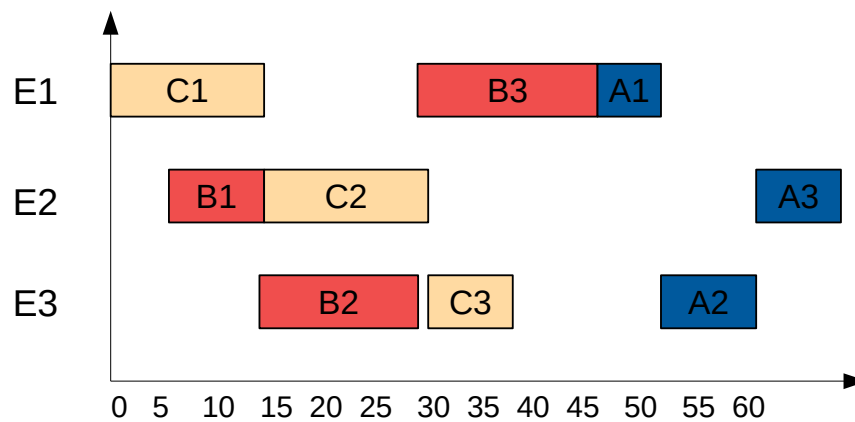
Ugyanahhoz a berendezéshez rendelt részfeladatok végrehajtási sorrendje könnyedén leolvasható a gráfról. A 2.4 ábrán látható példában az E2-es berendezés által elvégzett részfeladatok sorrendje B1 ->C2 ->A3. Ahhoz, hogy a berendezés el tudjon végezni egy adott részfeladatot nem elég, hogy az általa végzett előző részfeladatot befejezze, hanem szükség van még ezen kívül arra, hogy a soron következő részfeladathoz szükséges összes részfeladat elkészüljön. Csak ezeket követően tudja megkezdeni az adott részfeladat végrehajtását. Példa: C2-es részfeladat végrehajtásához szükséges, hogy az E2-es berendezés befejezze a B1-es részfeladatot, valamint az is, hogy a C1-es részfeladatot elkészítse az E1-es berendezés.





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 a 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

## 2.2.4. A makespan minimalizálás algoritmus

Az algoritmus 2.6 első lépésben inicializálja a  $makespan^{cb}$  értékét végtelennel, majd beállítja a  $S$  halmazt, amelyben az ütemezés során a nyitott részproblémák szerepelnek. Kezdetben csak a gyökér probléma szerepel benne, vagyis egy recept gráf bármilyen hozzárendelés nélkül. A **recipe** függvény visszaadja a  $G(N, A_1, A_2, w)$  által jelzett probléma recept gráfját, ahol

$N$ : taszkokat és termékeket reprezentáló csomópontok halmaza

$A_1$ : a recept élek halmaza

$A_2$ : ütemezési élek halmaza, ebben a pillanatban még üres

$w_{i,i'}$ : recept élek súlya, ami az  $i$  taszk minimális feldolgozási ideje .

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.

Az iteráció elején kiértékelődik, hogy a részprobléma képes-e optimális megoldást nyújtani, vagy sem. Ez a **bound** függvénnyel történik. A leggyakrabban a leghosszabb út keresésével vizsgálja meg a részproblémát **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 a következő részprobléma kerül kiválasztásra, amennyiben létezik.

Ha viszont kisebb a korlát, akkor ellenőrzi az algoritmus azt, hogy az összes taszk már ütemezett-e, azaz a részprobléma teljesen ütemezett. Ha így van a legjobb megoldás frissítésre kerül. 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. Ezek kapnak egy másolatot az aktuális  $S$ -gráfról. Ezt kibővíti az új hozzárendelés alapján az ütemezési élekkel. Végezetül pedig az új részproblémát hozzáadja az  $S$  halmazhoz. Ha ez az  $S$  halmaz üres lesz, akkor a  $G^{cb}$  és a hozzárendelések leírják az optimális megoldást, és ha legalább egy megvalósítható akkor az algoritmus visszatér ezzel az értékkel. Ellenkező esetben nem ad vissza semmit.

```

 $makespan^{cb} := \infty$ 
 $S := \{\text{recipe}(), I, J, \emptyset\}$ 
while  $S \neq \emptyset$  do
   $(G(N, A_1, A_2, w), I', J', \mathcal{A}) := \text{select\_remove}(S)$ 
  if  $\text{bound}(G) < makespan^{cb}$  then
    if  $I' = \emptyset$  then
       $makespan^{cb} := \text{bound}(G)$ 
       $G^{cb} := G$ 
       $\mathcal{A}^{cb} := \mathcal{A}$ 
    else
       $j := \text{select}(J')$ 
      for all  $i \in I_j \cap I'$  do
         $G^i(N, A_1, A_2^i, w^i) := G(N, A_1, A_2, w)$ 
        for all  $i' \in \bigcup_{(i', j) \in \mathcal{A}} I_{i'}^+ \setminus \{i\}$  do
           $A_2^i := A_2^i \cup \{(i', i)\}$ 
        end for
        for all  $i' \in I_i^+$  do
           $w_{i, i'}^i := t_{i, j}^{pr}$ 
        end for
         $S := S \cup (G^i(N, A_1, A_2^i, w^i), I' \setminus \{i\}, J', \mathcal{A} \cup \{(i, j)\})$ 
      end for
      if  $I' \subseteq \bigcup_{j' \in J', j \neq j'} I_{j'}$  then
         $S := S \cup (G(N, A_1, A_2), I', J' \setminus \{j\}, \mathcal{A})$ 
      end if
    end if
  end if
end while
if  $makespan^{cb} \neq \infty$  then
  return  $(G^{cb}, \mathcal{A}^{cb})$ 
end if

```

2.6. ábra. A makespan minimalizáló algoritmus pszeudó kódja [1]

## 2.3. Throughput vagy profitmaximalizálás

Eredetileg az S-gráf keretrendszer makespan minimalizációs problémák megoldására lett létrehozva. Azonban a későbbiekben bővítésre került, így ezután throughput, profitmaximalizációs problémák megoldására is lehet alkalmazni. Az alapötlet Majozi and Friedler [10], valamint Holczinger [11] nevéhez fűződik.

```

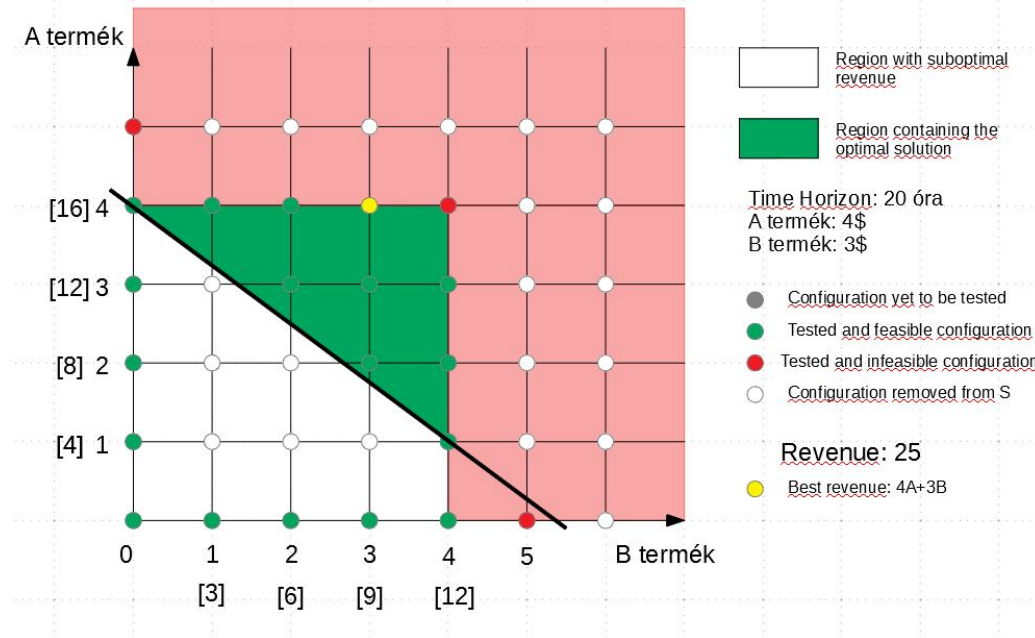
 $revenue^{cb} := 0$ 
 $S := (\mathbb{Z}^*)^{|P|}$ 
while  $S \neq \emptyset$  do
   $x := \text{select\_remove}(S)$ 
  if  $\text{feasible}(\text{recipe}(x), t^H)$  then
    if  $\text{revenue}(x) > revenue^{cb}$  then
       $revenue^{cb} := \text{revenue}(x)$ 
       $x^{cb} := x$ 
       $\text{update}(S, revenue^{cb})$ 
    end if
  else
     $S := \{x' \in S \mid x' \not\geq x\}$ 
  end if
end while
if  $revenue^{cb} \neq 0$  then
  return  $(x^{cb}, revenue^{cb})$ 
end if

```

2.7. ábra. A jövedelem maximalizáló algoritmus pszeudó kódja [1]

Az algoritmus először inicializálja  $S$  halmazt, minden lehetséges batch számmal, a termékekre vonatkozóan. Fontos kiemelni, hogy ebben az esetben minden termékénél a batch méret rögzített. Ezt követően minden iteráció során az előbb említett halmazból kiválasztásra kerül egy konfiguráció a **select\_remove** függvény segítségével. Ezután kerül feasibility tesztelésre, amely során eldől, hogy a megadott időhorizont alatt megvalósítható vagy sem. Ha megvalósítható és nagyobb jövedelmet biztosít, akkor a jelenlegi legjobb megoldás felülíródik. 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 2.8 ábrán látható egy Throughput módszerrel megvalósított feladat eredménye.



2.8. ábra. Throughput maximalizálás szemléltetés

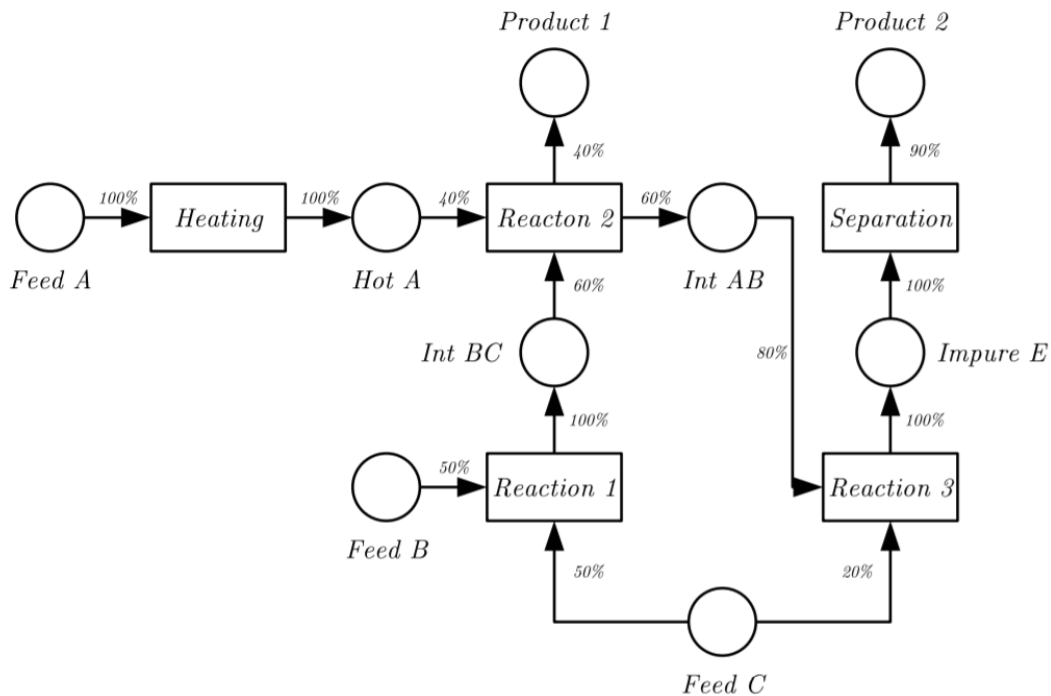
Az algoritmus először végighalad a tengelyek mentén, vagyis az egyik termék batch mérete 0 lesz, a másik pedig növekszik. Ezt addig folytatja amíg megtalálja az első nem megvalósítható, azaz infeasible konfigurációt. Ezt követően megteszi ezt a másik tengelyen is. Így kap egy jelenleg maximális jövedelmet. Az utolsó még feasible batch méreteknél nagyobb batch mértéket már nem kell vizsgálni, hiszen azokat a megadott időhorizonton belül nem lehet megvalósítani. A képen látható feladatban ez 16 volt. A két tengelyen lévő 16 értékhez tartozó batch méretet összeköti egy vonallal. Jelenlegi példában a B termékhez tartozó 16-os érték nem egész batch mérethez tartozik, a legközelebbi a 15-tel rendelkező, ami infeasible már. A behúzott vonal alatt azok a konfigurációk szerepelnek, amelyek jövedelme nem éri ez a jelenlegi maximumot, így ezek nem a lehető legjobb megoldást adják meg. Ezek tesztelésére már nem kell időt fordítani. Az algoritmus addig folytatja a futást, amíg a halmaz, amely a konfigurációkat tartalmazza ki nem ürül. Ha nem talált megvalósítható konfigurációt, akkor az adott problémát a megadott időhorizontot belül nem lehet megoldani. Ellenkező esetben az algoritmus megadja az elérhető legnagyobb profitot, és a konfigurációt, amellyel ez elérhető. A példafeladatban a maximálisan elérhető jövedelem 25. Ezt négy darab A termék és 3 darab B termék legyártásával lehet elérni.

## 3. fejezet

### Probléma definíció

Számos esettanulmány és irodalmi példa esetében a batch méretek nem rögzítettek, és ha több egység képes elvégezni egy részfeladatot, akkor párhuzamosan tehetik meg azt. Ez a fajta probléma meghatározás különösen gyakran az throughput vagy bevétel maximalizációs probléma esetén jelentkezik, de megjelenhet a makespan minimalizálásnál is. Az idő diszkréción alapuló megközelítések meg tudják oldani ezt a problémát, azonban az S-gráf keretrendszer esetén néhány módosítás szükséges. Az 2.7 ábrán látható algoritmus megköveteli, hogy a recept rögzített legyen, valamint egy termékhez tartozó batch jövedelme is előre ismert legyen. Az előbb említett problémák esetén viszont egyik sem garantált. A javasolt megközelítés szemléltetésére a Kondili és munkatársaitól [2] származó példát veszem igénybe. Ez látható 3.1 ábrán.

A folyamat 5 taszkból áll: fűtésből, 3 darab reakcióból, és az elválasztásból. 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 ugyanakkor feldolgozása idővel. A kapacitásuk eltér, egyiké 80 kilogramm a másiké pedig 50 kilogramm, de lehet őket párhuzamos igénybe venni. Feltételezzük, hogy az összes egység képes a kapacitásuknál kisebb terheléssel működni, azaz nincs teherbírásuknak alsó határa. 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 reakciós folyamat elvégezhető egyik vagy másik reaktor által, illetve ezekkel párhuzamosan. Ez  $3^3 = 27$  fix receptet eredményez, amelyek különböző batch méret intervallummal rendelkezhetnek. Mindegyik esetben különálló S-gráf receptet kell létrehozni, hogy a korábban említett S-gráf algoritmust igénybe lehessen venni profit maximalizálásra, így a keresési terület 27 dimenziós térré válna. Ez óriási CPU igényhez vezetne az optimalizálás során, ezért az esetek számának csökkentése elengedhetetlen.

3.1. táblázat. A 27 állandó 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

Ha megnézzük a 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 azt 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, ak-



kor azt kizárhatjuk a vizsgálatból, mert az továbbra is garantálva van, hogy megtalálja az optimális megoldást. A következő 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 következő 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

## 4. fejezet

# Az új módszer

Az új algoritmus alapját a?? ábrán látható

Az algoritmus változóinak elnevezése a következő:

- $profit^{cb}$ : a megtalált legjobb megoldás
- $SOAA$ : (Set Of Already Assigned): azon berendezések halmaza
- $S$ : a nyitott részproblémák halmaza
- $PP$ : az aktuális részprobléma
- $I$ : a taszkok halmaza
- $J$ : a berendezések halmaza
- $TH$ : az időhorizont
- $w$ : a recept él súlya

**Algoritmus 1** Párhuzamos taszkvégrehajtást megvalósító algoritmus

---

```

1: procedure MAXPROFIT( $TH, batchsize$ )
2:    $profit^{cb} = -\infty$ 
3:    $SOAA := \emptyset$ 
4:    $S := (recipe(), I, J, \emptyset)$ 
5:   while  $S \neq \emptyset$  do
6:      $PP := select\_remove(S)$ 
7:     if  $PP.Feasible(TH)$  then
8:       if  $PP.profit\_bound > profit^{cb}$  then
9:         if  $PP.IsLeaf()$  then
10:           $profit^{cb}.Update(PP.profit\_bound)$ 
11:        else
12:           $j := select(J')$ 
13:          for all  $i \in I_j \setminus SOAA$  do
14:             $SOAA^i := SOAA^i \cup j$ 
15:             $G^i(N, A_1^i, A_2^i, w^i) := G(N, A_1, A_2, w)$ 
16:            for all  $i' \in \bigcup_{(i',j) \in \alpha} I_{i'}^+ \setminus \{i\}$  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:             $S := S \cup (G^i(N, A_1, A_2^i, w^i), J', \alpha \cup \{(i, j)\})$ 
23:          end for
24:          if  $I \subseteq \bigcup_{SOAA^i = \emptyset, j = j'} I_j'$  then
25:             $S := S \cup (G(N, A_1, A_2), J' \setminus \{j\}, \alpha)$ 
26:          end if
27:        end if
28:      end if
29:    end if
30:  end while
31:  return  $profit^{cb}$ 
32: end procedure

```

---

## 5. fejezet

# Implementálás

### 5.1. S-gráf solver

Az S-gráf megoldó szoftver egy a Pannon Egyetem, Műszaki Informatikai Kar, Rendszer- és Számítástudományi Tanszák által fejlesztett szoftver. C++ nyelven íródott, amely képes nagy teljesítményre, ami fontos a tudományos területeken. A szoftverben a C++ nyelv sajátosságai fedezhetők fel. Többek között az objektum orientált paradigma, valamint különböző a nyelvbe nem beépített tárolók, algoritmusok.

A szoftver szakaszos üzemű termelőrendszerek rövidtávú ütemezésével foglalkozik. A megoldó jelenleg a tárolási stratégiákat tekintve a NIS, UIS, UW és LW feladatokat támogatja. Ezek mellett 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 megoldáshoz szükséges információkat. A bemeneti fájl alapján felépül a receptgráf, és ebből legenerálja a részproblémákat. A szoftver az eredmény képernyőn való megjelenítése mellett, fájlban is eltárolja azt, emellett a Gantt-diagram adatait karakteres formában is eltárolja benne. Azonban lehetőség van arra is,

hogyan a Gantt-diagramot képfájlban mentse el. A megoldó használata környezeti változók segítségével történik. Ezek segítségével adható meg a bemeneti fájl, az eredményeket tároló fájl, a kívánt megoldó módszer meghatározása, 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 fájlt lehet megadni. A "-t" utasítás a maximálisan használható szálak megadására szolgál. A példában az utolsó parancs, az "-m" pedig a megoldómódszer kiválasztására szolgál.

A megoldóban egyik legfontosabb szerepet tölti be a Branch and Bound, azaz a Korlátozás és Szétválasztás algoritmus. A döntések az ütemezés során azt reprezentálják, hogy melyik berendezés, melyik részfeladatot, taszkat végzi el, illetve ezek sorrendjét is megmutatja. A szétválasztás több módszerrel is végrehajtható, ezért a szoftver kialakítása révén létrehozhatóak, hozzáadhatóak új algoritmusok a megoldóhoz.

## 5.2. Adatok beolvasása

### 5.2.1. Bemeneti fájl

A program működéséhez szükséges adatokat ods kiterjesztésű fájlban lehet megadni. Példaként a solver mappa, input almappájában megtalálható az **extended\_precedential.ods** fájl. 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.

- Precedence tábla, amely a gráfban szereplő éleket szemlélteti az él kezdő csomópontjának és végpontjának feltüntetésével. Két új, hasonló oszlop lett a táblázathoz illesztve. Ezek a következők:
  - Az `s_percent` oszlop, amely a táblázat `task1` oszlopában szereplő részfeladathoz tartozó százalékot mutatja.
  - A `d_percent` oszlop, amely a táblázat `task2` oszlopában szereplő részfeladathoz tartozó százalékot mutatja.
- A task tábla a részfeladatok adatait tartalmazza. Ez 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 módosítása 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 **RealtionalProblemReader** osztály a felelős. Ennek feladata, hogy felépítse a recept gráfot, illetve ezt eljuttassa a **MainSolver** osztályhoz. A beolvasást végző függvények forráskódjai az `src\lib` mappában megtalálható **realtionalproblemreader.cpp** és **realtionalproblemreader.h** fájlokban szerepelnek. Az ebben megtalálható függvények közül az adatok programba történő átemeléséhez a **ReadPrecedential()** függvényre van szükség. Ez lett kibővítve, 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 bővíve 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 értékek el tudja a szoftver tárolni. Ezeket az értéket az **SGraph** objektum **Recipe** objektumában tárolódnak. A tárolást úgy lehet elképzelni, mint egy mátrixot, ami azt mutatja meg, hogy az adott taszkból, melyik taszkba van 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, mekkora kapacitása megy az adott élen. 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 berendezést, vagy berendezéseket kell hozzárendelni. 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 ősoztálya az **SchProblem** osztály. Az új módszer lényege abban áll, hogy egy adott taszkhoz 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 még 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 következik be akkor következik a berendezés keresése. Itt meg kell vizsgálni, hogy az éppen soron lévő berendezés még 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 taszkhoz 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, hogy az adott részproblémának mennyi gyereke lehet. Abban az esetben, ha olyan berendezés kerül kiválasztásra, amelyet csak olyan taszkokhoz lehet rendelni, amelyeket más berendezés is képes végrehajtani, akkor meg kell növelni a gyerekek számát, mert lehetséges olyan döntést hozni, hogy az adott berendezést semelyik lehetséges taszkhoz sem rendeljük hozzá.

#### 5.3.2. Branching függvény

Ez a függvény valósítja meg a szétválasztást a probléma megoldása során. 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 idő), 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 ezeket követően a kiválasztott berendezést már csak egy taszkhoz lehet hozzárendelni, akkor a berendezést kivesszük a nem ütemezett berendezések halmazából. Abban az esetben ha a kiválasztott berendezést már nem kívánjuk hozzárendelni taszkhoz, de van olyan taszk amit még el tudna végezni és ezt a taszkot más is el tudja végezni, akkor a berendezést kivesszük az ezt követő részproblémákból. Mindezen lépések után ennek a részproblémának a gyerek problémájához hozunk döntést a **MakeDecisions** függvény segítségével. Valamint annak 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 időhorizontot 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 szemben maximalizációra lett tervezve, a Solver keretrendszerben megtalálható további megoldómódszerekkel

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 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, hogy felkészítsük, 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 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

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-es 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 azt ott gyártott mennyiség ekkora részét képes átvenni. A **d\_percent** oszlopban lévő adatok pedig, hogy az a taszk, amelybe az él befut, ekkora százalékát képes felvenni a mennyiségnek.

Az eredetileg szereplő taszkok azonosítója megváltozik, mivel az új taszkokat nem csak hozzá adjuk őket 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 az ugyanazokat a taszkokat reprezentáló, különböző azonosítókat vizsgálja a feljebb lévő sorokban. A feltételek forráskódjai a követ-

kező részben tekinthető meg.

## 5.2. Kódrészlet. FlexBatchSchProblem osztály adattagjai

```

1  if(return_graph->IsProfitMaximization()){
2      vector<vector<double>>
          newSourcePercents(return_graph->GetRecipe()->GetTaskCount(),
          vector<double>(return_graph->GetRecipe()->GetTaskCount(),0));
3  vector<vector<double>>
          newDemandPercents(return_graph->GetRecipe()->GetTaskCount(),
          vector<double>(return_graph->GetRecipe()->GetTaskCount(),0));
4  bool wasAlreadyP = false;
5  bool over = false;
6  for(uint i = 0; i<return_graph->GetRecipe()->GetTaskCount(); i++){
7      for(uint j = 0; j<return_graph->GetRecipe()->GetTaskCount(); j++){
8          if(i==0){
9              if(j>0 && tasksOldIds[j]!=tasksOldIds[j-1]){
10                 wasAlreadyP = false;
11             }
12             if(j>0 && tasksOldIds[j]==tasksOldIds[j-1] &&
GetRecipe()->getSourcePercent(tasksOldIds[i],tasksOldIds[j])!=-1 &&
!wasAlreadyP){
13                 wasAlreadyP = true;
14             }
15             if(!wasAlreadyP){
16                 newSourcePercents[i][j] =
GetRecipe()->getSourcePercent(tasksOldIds[i],tasksOldIds[j]);
17                 newDemandPercents[j][i] =
GetRecipe()->getDemandPercent(tasksOldIds[j],tasksOldIds[i]);
18             }
19             else{
20                 newSourcePercents[i][j] = -1;
21                 newDemandPercents[j][i] = -1;
22             }
23         }
24         else{
25             over = false;
26             if(j>0 && tasksOldIds[j]!=tasksOldIds[j-1]){
27                 wasAlreadyP = false;
28             }

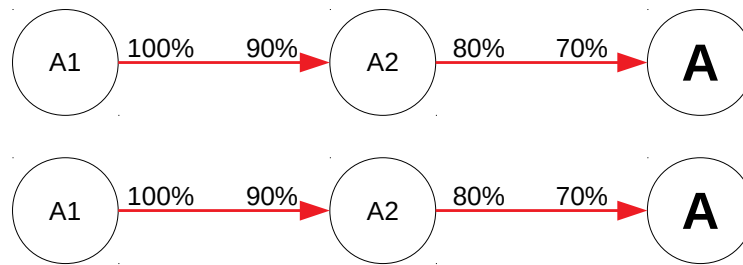
```

```

29         if(j>0 && tasksOldIds[j]==tasksOldIds[j-1] &&
GetRecipe()->getSourcePercent(tasksOldIds[i], tasksOldIds[j])!=-1 &&
!wasAlreadyP && newSourcePercents[i][j-1]!=-1){
30             wasAlreadyP = true;
31         }
32         for(uint k = 1; k<=i;k++){
33             if(tasksOldIds[i]==tasksOldIds[i-k] && newSourcePercents[i-k][j]!=-1
){
34                 over =true;
35             }
36         }
37         if(!wasAlreadyP && !over){
38             newSourcePercents[i][j] =
GetRecipe()->getSourcePercent(tasksOldIds[i], tasksOldIds[j]);
39             newDemandPercents[j][i] =
GetRecipe()->getDemandPercent(tasksOldIds[j], tasksOldIds[i]);
40         }
41         else {
42             newSourcePercents[i][j] = -1;
43             newDemandPercents[j][i] = -1;
44         }
45     }
46 }
47 }
48 return_graph->GetRecipe()->SetSourcePercents(newSourcePercents);
49 return_graph->GetRecipe()->SetDemandPercents(newDemandPercents);
50 }

```

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 következő táblázatban látható. Zárójelben 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, a 3-as azonosítóval rendelkező taszk között él keletkezzen, mert nem az 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 szeretnék. Jelen esetben kettő.

#### 5.4.2. UpdateProfitBound függvény

A függvény feladata, hogy kiszámolja az első S-gráfhoz tartozó korlátot, valamint minden egyes taszkhhoz 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 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 következő lépés a kezdő csomópontok, taszkok megkeresése. Ez 2 *for* ciklus segítségével történik, amelyekben vizsgáljuk, 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 elejére, 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 meghatározása, 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 következő képletet kell használni.

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

Abban az esetben, ha a kezdeti taszk még nem lett ellenőrizve, akkor nem lehet megvizsgálni az éppen kiválasztott taszkt, 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, valamint 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 az előző pontban bemutatott függvénytől eltérően nem a S-gráf bemeneti csomópontjait reprezentáló taszkokat kell először megkeresni, hanem a paraméterlistában átadottat, valamint annak közvetlen szomszédjait. Ehhez is a **deque-t** veszünk igénybe. Legelsőnek a az átadott taszk kerül bele, majd *for* ciklus segítségével megkeresi annak szomszédjait, és ezeket is a **deque** végéhez hozzáfűzi. Ezek után meg kell keresni a többi olyan taszkot is, amiknek a kapacitásukat újra át kell vizsgálni, és ha szükséges módosítani. Azt követően, hogy a két végű sor tartalmazza az összes átvizsgálandó taszkot megtörténik a tényleges kapacitás módosítás. 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 az új érték. Amennyiben a taszknak nincs bejövő éle, akkor az imént meghatározott kapacitása megmarad, nincs szükség további ellenőrzésekre. Ez 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 szükséges az, hogy két darab *for* ciklus bejárja a termékeket és a taszkat. 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 az új módszer került meghívásra parancssori paraméterek által a megoldó szoftver indításakor. Olyan esetekben kerül meghívásra, amiket csak abban az esetben kell végrehajtani, ha az új, taszkok párhuzamos végrehajtására alkalmas módszert 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

Új módszer elkészítése miatt szükség volt új argumentumok hozzáadására a meglévők mellé. Az argumentumokat 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 megoldómódszert lehet kiválasztani. A második új argumentum a ***profit\_max***, amely a **obj** kapcsolóhoz tartozik. Ez pedig a célfüggvényt reprezentálja.

## 5.6. Megoldás fájlba írása

Az új módszer 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 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 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 utána 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

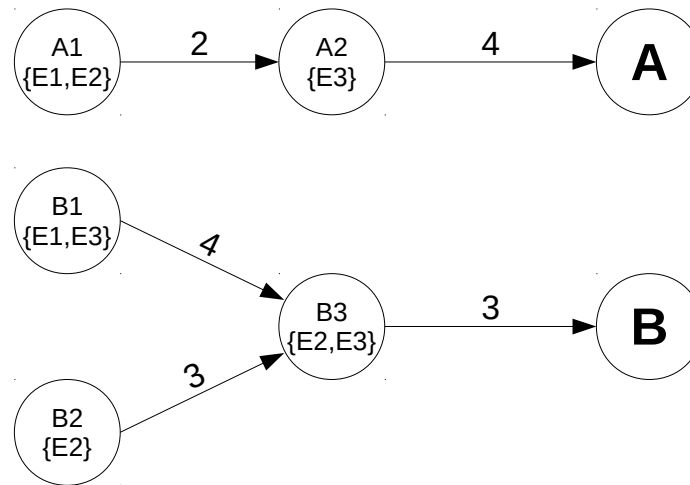
Ebben a fejezetben az elkészített megoldó algoritmus tesztelését mutatom be, hogy az megfelelően, hiba nélkül képes a feladatokat megoldani. A tesztelés a következő konfigurációval rendelkező számítógéppel történt:

- Processzor: Intel i5-7200, 2,50 Ghz
- 8 GB RAM
- Operációs rendszer: Windows 10
- Fejlesztési 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, végezze el 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 receptet, 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
- **-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 megoldószoftverhez hozzáadott profit maximalizálás kerül kiválasztásra.
- **-precycle off:** Az 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, illetve a részfeladatokat, valamint az ezeket megvalósítani képes berendezéseket. Az éleken megfigyelhetőek még a taszkok befejezéséhez szükséges idő.



6.1. ábra. Tesztfeladat

## 6.1. Taszkok közti kapacitások változtatásának hatása

A módszer sajátossága, hogy a receptélekkel az egyes taszkok kapacitásának meghatározott százaléka hasznosítható. Ezeket a százalékokat a bemeneti fájlban lehet megadni. A példafeladat a 6.2 ábrán megtekinthető bemeneti adatokkal rendelkezik. Az időhorizont a bemutatott mintafeladat során 20. Látható, hogy az egyes részfeladat által lehetséges kapacitásoknak nem a teljes mennyisége kerül tovább a következő köztes feladathoz. Ez, 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 A terméket és 1 darab B terméket gyártunk le. Mindkét termék legyártásából származó jövedelem mindössze egy egység. A *precendce* 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 70 százalékát veszi fel az A2-es részfeladat. A *proctime* táblán megtalálhatjuk, hogy

melyik taszkot, melyik berendezés tudja elvégezni, valamint ezt mennyi idő alatt.

product		
name	number	revenue
	1	
A	2	1
B	1	1

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

equipment		
name	number	b_capacity
	1	
E1	1	40
E2	1	50
E3	1	60

precedence			
task1	task2	s_percent	d_percent
<task.name>	<task.name>		
A1	A2	80	70
B1	B3	50	60
B2	B3	70	65

proctime		
task_name	eq_name	time
<task.name>	equipment.name	infity
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. Első feladat bemeneti adatai

A feladat megoldását tartalmazó TXT kiterjesztésű fájl 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 láthatók idő értékek, 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 a az ütemezési élek csoportjához. Mivel az egyik termékből, nevezetesen az A-ból, több mint egy darabot gyártunk ezért megkülönböztetjük az egyes receptekhez tartozó feladatokat. Láthatunk A1-et és A1\_2-t. Ugyanolyan típusú feladatról beszélünk, de különböző recepthez tartoznak, ezért kell megkülönböztetni egymástól ezeket.

```

A1 ---> A2  2
A1_2 ---> A2_2  2
A2 ---> A1_2  0
A2 ---> A2_2  0
A2 ---> A  4
A2_2 ---> A_2  4
B1 ---> A2  0
B1 ---> A2_2  0
B1 ---> B3  4
B2 ---> B3  3
B3 ---> A1  0
B3 ---> B  3
A ---> A2_2  0
B ---> A1  0
B ---> A2  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 173,846 lett. Ezt úgy kapjuk meg, hogy a termékekhez tartozó értékeket összeadjuk. Jelen példa esetében 3 érték kerül összeadásra, ezek a következők: A2, A2\_2, és a B3. Az A2-es és a A2\_2-es részfeladatok kapacitása 60, a B3-as taszk kapacitása pedig 53,846. Mivel a példában mindkét termék esetében a jövedelem egy, ezért az érték marad változatlanul a termékeknél. Ezeket összeadva kijön a 173,846, ami megegyezik a szoftver által kiszámolt megoldással. Ezek az értékek a 6.4 ábrán megtekinthetők.

```

A1: 90
A1_2: 90
A2: 60
A2_2: 60
B1: 100
B2: 50
B3: 53.8462
A: 60
A_2: 60
B: 53.8462
bound: 173.846

```

6.4. ábra. A megoldást tartalmazó fájl második része

Az utolsó szakaszban a Gantt diagram karakteres formában található meg. Látható,

hogyan a megadott bemeneti adatok alapján minden lehetséges taszk és berendezés kombináció elő áll. A6.5 ábrán látható ez.

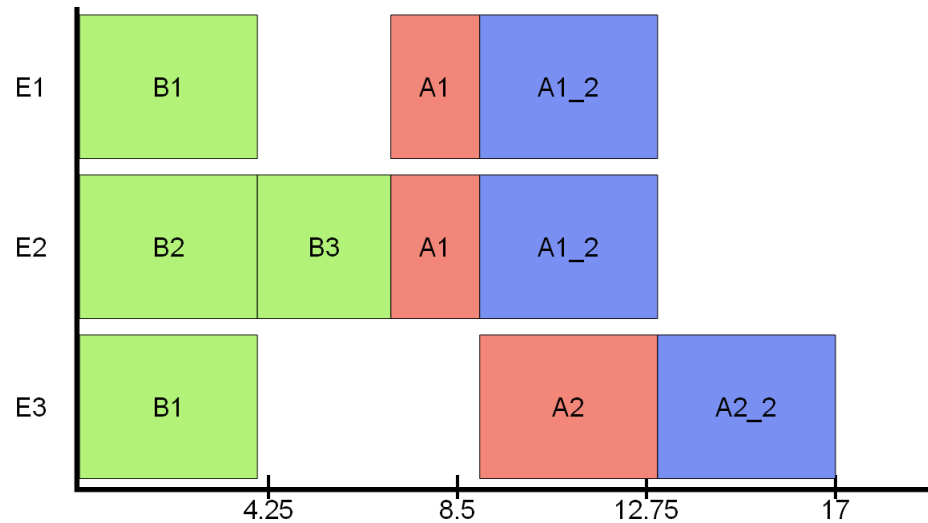
```
Gantt chart:
A - A1: 7-9, 2  E1  E2
A_2 - A1_2: 9-13, 4 E1  E2
A - A2: 9-13, 4  E3
A_2 - A2_2: 13-17, 4      E3
B - B1: 0-4, 4  E1  E3
B - B2: 0-4, 4  E2
B - B3: 4-7, 3  E2  E3
The longest path: 17
```

6.5. ábra. A megoldást tartalmazó fájl második része

A következő esetben újra lefuttatjuk a példafeladatot, azzal a kis változtatással a bemeneti adatokon, hogy az élekhöz tartozó százalékok mindegyikét felemeljük 100 százalékra, vagyis a teljes mennyiség minden taszk esetében tovább kerül a következő részfeladathoz. A többi adat, köztük a kapcsolóval megadott idő horizont is változatlan marad.

Ahogy várható volt az újbóli futtatás során eltérések vannak az előzőhöz képest. Egyik legfontosabb eltérése, hogy a bound megváltozott. Ebben az esetben már csak 170 lett. A jelenlegi futás során a bound kerekén 170 lesz. További eltérése, hogy most már nem minden lehetséges taszk - berendezés pár lett összerendelve, mert abban az esetben a megadott idő horizonton belül nem lenne megoldható a feladat. A változás a B3-as taszk esetében történt meg. Az E3-as berendezés már nem lett hozzárendelve, azaz ezt a feladatot nem végzi el. A megoldó szoftver által elkészített png kiterjesztésű fájl látható a 6.6 ábrán. Könnyen leolvasható, hogy a berendezések párhuzamosan képesek ugyanazt a taszkot elvégezni. Erre példa a B1-es köztes feladat rögtön az ütemezés elején.





6.6. ábra. A solver által elkészített Gantt diagram

## 6.2. Idő horizont változtatás

A parancssori paraméterben megadott idő horizont is fontos szerepet tölt be az ütemezés során. Az eddig bemutatott példáknál nem változott meg. Ebben a részben viszont ennek a változtatására bekövetkező változások bemutatása a cél. Az idő horizont azt az időt adja meg, amelyen belül meg kell oldani a feladatot.

Minden receptnek van egy olyan minimális ideje, amely alatt nem teljesíthető. Ha ennél kisebbet adunk meg, akkor a feladat megoldhatatlan, elvégezhetetlen lesz. A 6.1 ábrán látható feladat esetében, amelyet a 6.2 táblákban szereplő adatokkal futtatunk, ez a mindenképpen szükséges idő 12 egység lesz. Szóval ha 11 vagy annál kisebb idő horizontot adunk meg, akkor nem kapunk megoldást a feladatra. A Minden egyes gyártási munkafolyamat során találunk egy felső határt, amelynél hiába biztosunk több időt a munkafolyamat befejezésére, nem lehet nagyobb bevételre szert tenni. A mintafeladatnál ez 17 időegységnél következik be. Ebben az esetben minden berendezés elvégzi azokat a taszkokat, amelyeket el is tud végezni. A példából láthatjuk, hogy minél több idő van a feladat megoldására, annál nagyobb bevételre tehetünk szert. Ennek oka az, hogy több berendezést tudunk beállítani adott feladat elvégzésére, mert már belefér abba az időtartamba, amelyet korlátként megadtunk. Láthatjuk a 15 és 16 időegység esetében,

hogy hiába biztosítunk nagyobb idő horizontot nem tudunk nagyobb bevételt elérni. Ilyen eset akkor fordulhat elő, mikor egy részfeladat elvégzéséhez szükséges idő nagyobb, mint amekkora szabadon fennmaradó idővel rendelkezünk még a határ alatt.

6.1. táblázat. Adott időhorizont alatt elérhető bevételek

Idő horizont	$\leq 11$	12	13	14	15,16	$\geq 17$
Max bevétel	nem megoldható	139,048	153,333	155,714	159,56	173,846

## 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 batchmérettel 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, az jó eredményeket szolgáltat.

A feladat elvégzését követően a megoldó szoftver már képes flexibilis batchmérettel rendelkező feladatok megoldására, azonban további lehetőségek vannak, hogy a módszer gyorsabban elvégezhesse a feladatát, megtalálja az optimális megoldást a problémára. Ha az eddigi meglévő algoritmus módosításra kerülne úgy, hogy ha az adott rész nem ad jobb megoldást a meglévőnél, de mégis megvizsgáljuk, hogy megoldhatóak az ott lévő batchek akkor változhat és jobb teljesítmény lenne elérhető a módszerrel.

# 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.