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

[1 oldalas, magyar nyelvű tartalmi kivonat]

Abstract

Throughput maximization in batch systems!!!!!!!!!!!!!!!!!!

[1 oldalas, angol nyelvű kivonat]

Tartalomjegyzék

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 feladtok 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 horizontnak 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

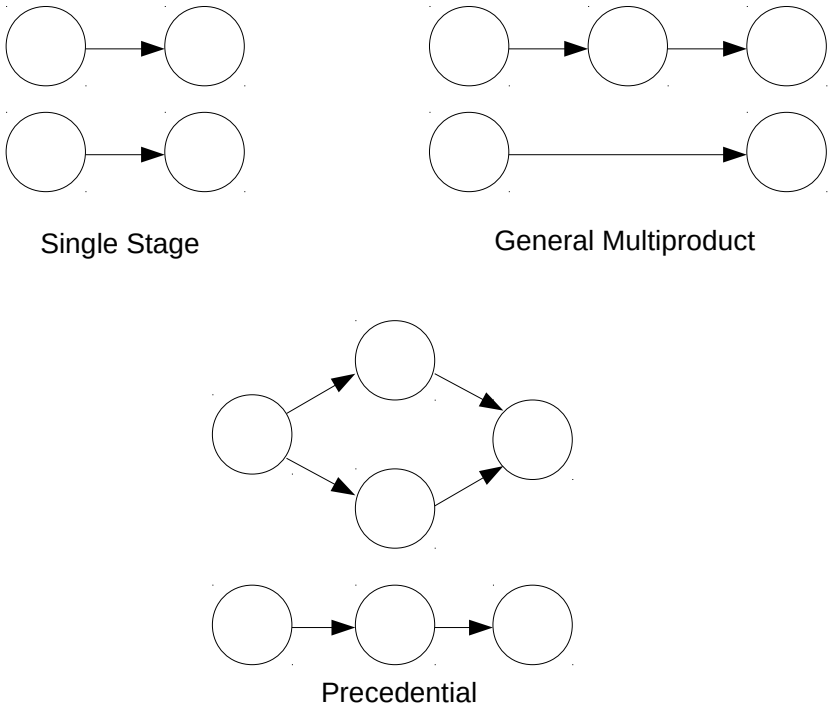
A recepteket a feladatok precedenciájuk szerint következő csoportokba lehet besorolni. A felsorolás a legegyszerűbbtől halad az általánosabb felé. Ezen kívül minden osztály a következőnek egy speciális esete.

- **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 egészértékű lineáris programozásról, azaz **M**ixed **I**nteger **L**inear **P**rogramming. 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 amellett, hogy megtalálja az optimális megoldást.

Precedencia alapú modellek - Precedence based: Ezeknél a módszereknél, szemben az időfelosztásos módszerekkel, nincs szükség az időhorizont diszkretizációjára, 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 modellezik, 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

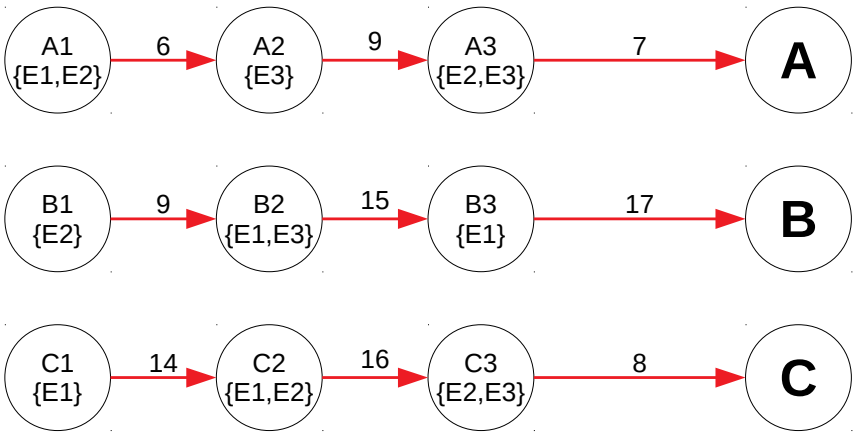
Az S-gráf keretrendszer volt az első olyan módszer, amely publikált gráf elméleten alapult, valamint szakaszos gyártórendszerek ütemezési problémák megoldására szolgált. [6] Ez a keretrendszer egy irányított gráf modellből, az S-gráfból, és a hozzá tartozó algoritmusokból áll. [7] Az S-gráf egy speciális 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.

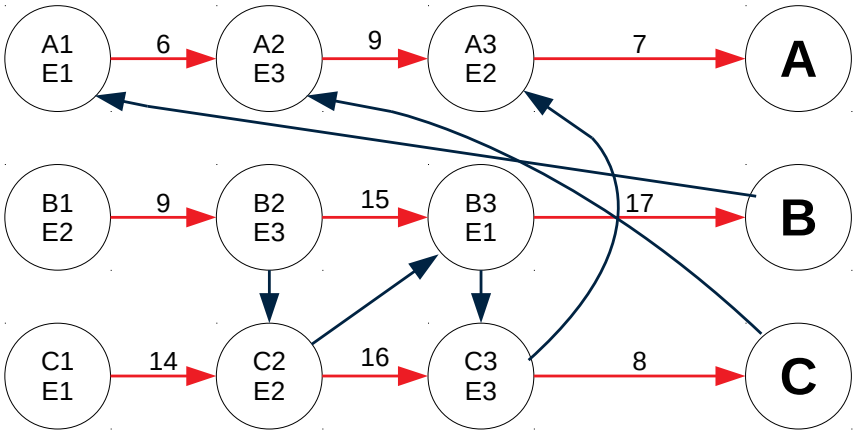
A nyilakon látható súlyok a részfeladat végrehajtásához szükséges időt mutatják meg. Ha egy részfeladatot több berendezés is képes elvégezni, akkor az előbb említett súly mindig a legkisebb előállítási idő lesz.



2.2. ábra. A recept gráf szemléltetése

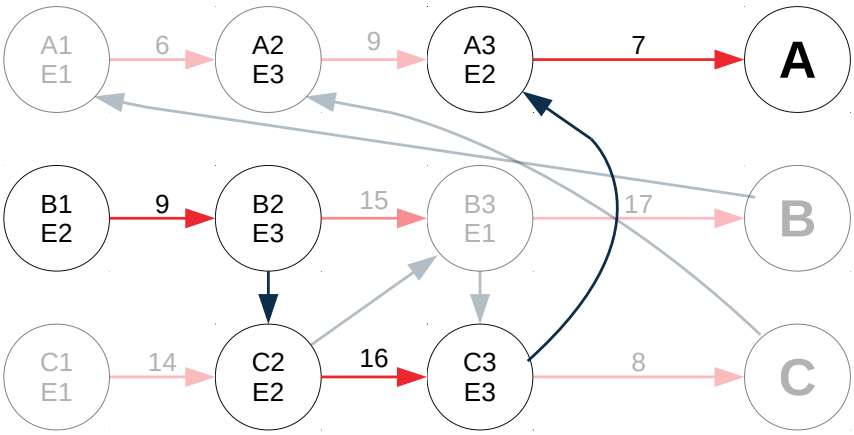
Minden S-gráf-hoz kapcsolódó algoritmus kiegészíti ezeket a gráfokat az úgynevezett **ütemezési élekkel**, amelyek az ütemezési döntést testesítik meg. Ezekkel az élekkel ki-

egészített gráfoknak a neve **Ütemezési gráf**. Példa a 2.3 ábrán nézhető meg. Az ábrán sötétkékkel 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.



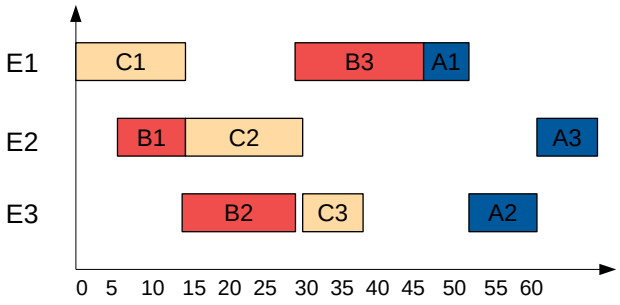
2.3. ábra. Az ütemezési gráf szemléltetése

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

2.3. Profitmaximalizálás

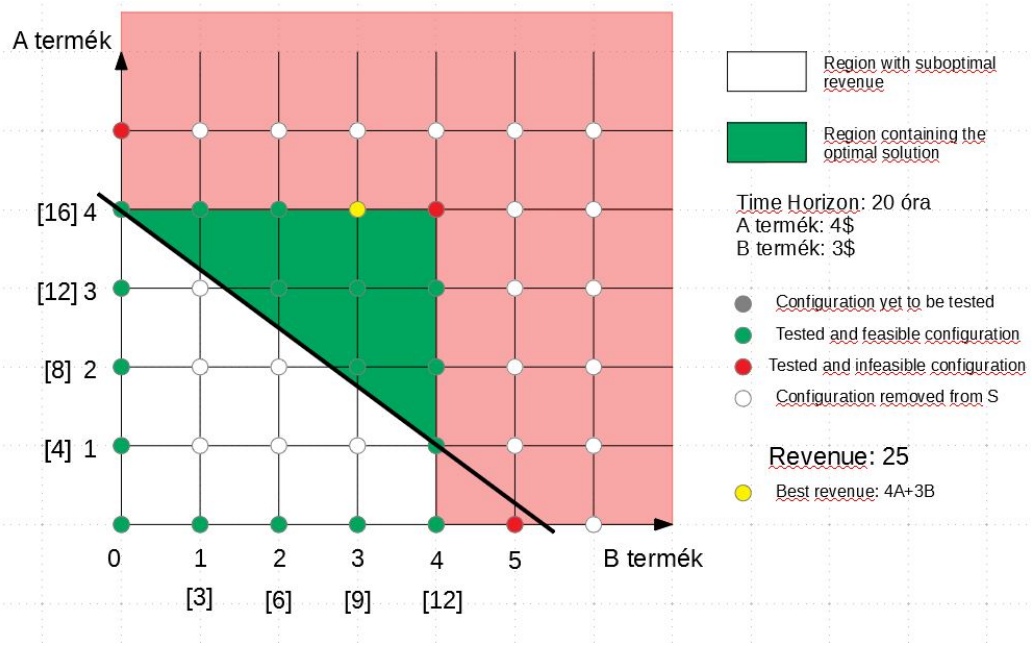
Eredetileg az S-gráf keretrendszer makespan minimalizációs problémák megoldására lett létrehozva. Azonban a későbbiekben bővítésre került, így ezután throughput, profitmaximalizációs problémák megoldására is lehet alkalmazni. Az alapötlet Majozi and Friedler [10], valamint Holczinger [11] nevéhez fűződik.

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

2.6. ábra. A jövedelem maximalizáló algoritmus pszeudó kódja

Az algoritmus először inicializálja S halmazt, minden lehetséges batch számmal, a termékekre vonatkozóan. Fontos kiemelni, hogy ebben az esetben minden terméknél a batch méret rögzített. Ezt követően minden iteráció során az előbb említett halmazból kiválasztásra kerül egy konfiguráció a **select_remove** függvény segítségével. Ezután kerül feasibiliy 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 2.7 ábrán látható egy Throughput módszerrel megvalósított feladat eredménye.



2.7. 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 mérteket 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 mérethet 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

```
Maxprofit(TH, batch_number){
  profit^cb = -∞
  alreadyAssignedTasks = 0
  S := {(recipe(), I, J, Ø)}
  while S != Ø do
    selectedSubProblem := select_remove(S)
    if(selectedSubProblem.feasible(TH))
      if(selectedSubProblem.profitbound_upper()>= profit^cb
        if(selectedSubProblem.isLeaf())
          profit^cb.update
        else
          j:= select(J')
            for all i∈Ij \  alreadyAssignedTasks do
              alreadyAssignedTasks =  alreadyAssignedTasks + j
              Gi(N,A1,A2^i,w^i) := G(N,A1,A2,w)
              for all i′ ∈ U(i′,j)∈A Ii′^+ \ {i} do
                A2^i := A2^i ∪ {(i′, i)}
              end for
              for all i′ ∈ Ii^+ do
                wi,i′ ^i := ti,j ^pr
              end for
              S := S ∪ (Gi(N,A1,A2^i,w^i), J′,A ∪ {(i, j)})
            end for
            if I ⊆ U j′∈J′, j!=j′ Ij′ then
              S := S ∪ (G(N,A1,A2), I, J′ \ {j},A)
            end if
        end if
      end if
    end if
  end while
  return profit^cb
}
```

4. fejezet

Az új módszer

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 fedezhetőek 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 Gannt-diagram adatait karakteres formában is eltárolja benne. Azonban lehetőség van arra is,

hogy a Gannt-diagramot képfájlban mentse el. A megoldó használata környezeti változók segítségével történik. Ezek segítségével adható meg a bementi fájl, az eredményeket tároló fájl, a kívánt megoldó módszer meghatározása, időhorizont megadása, továbbá számos különböző beállítási lehetőség. Erre egy példa:

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

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, taszkot 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.

- Precendence tábla, amely a gráfban szereplő éleket szemlélteti az él kezdő csomópontjának és végpontjának feltüntetésével. Két új, hasonló oszlop lett a táblázathoz illesztve. Ezek a következőek:
  - 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 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 értékek el tudja a szoftver tárolni. Ezeket az érték 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 ősosztálya 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 mellet, az átdolgozott kiválasztás módszer miatt, szükséges új adattagok bevezetése. Az első új adattag egy vectoron belüli vector segítéségvel 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.

FlexBatchSchProblem adattagjai

```
class FlexBatchSchProblem: public EqBasedSchProblem{
protected:
    vector<vector<bool>> eqAssignedToTask;
    IndexSet sounEqs;
}
```

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 ősosztályban 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 ősosztályban 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 él, 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 fellelhetőek. 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.

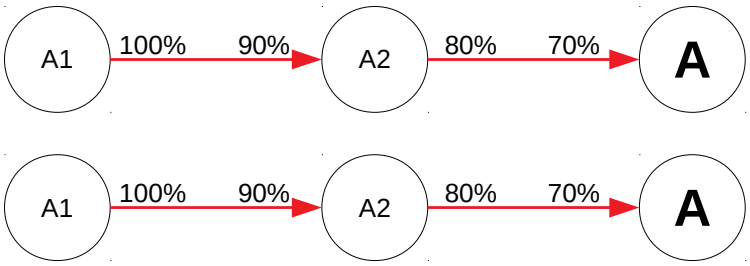
Az eredetileg szereplő taszkok azonosítója megváltozik, mivel az új taszkokat nem csak hozzá adjuk őket a következő azonosítóval. Az ugyanolyan taszkok egymást követő azonosítót kapnak, így a százalékokat tároló mátrixot is módosítani kell. Fontos dolog az, hogy csak az adott recepthez tartozó taszkok 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 az 5.1

ábrán láthatóak.

```
over = false;
if(j>0 && tasksOldIds[j]!=tasksOldIds[j-1]){
    wasAlreadyP = false;
}
if(j>0 && tasksOldIds[j]==tasksOldIds[j-1] && GetRecipe()->getSourcePercent(tasksOldIds[i],tasksOldIds[j])!=-1
        && !wasAlreadyP && newSourcePercents[i][j-1]!=-1){
    wasAlreadyP = true;
}
for(uint k = 1; k<=i;k++){
    if(tasksOldIds[i]==tasksOldIds[i-k] && newSourcePercents[i-k][j]!=-1 ){
        over =true;
    }
}
if(!wasAlreadyP && !over){
    newSourcePercents[i][j] = GetRecipe()->getSourcePercent(tasksOldIds[i],tasksOldIds[j]);
    newDemandPercents[j][i] = GetRecipe()->getDemandPercent(tasksOldIds[j],tasksOldIds[i]);
}
else {
    newSourcePercents[i][j] = -1;
    newDemandPercents[j][i] = -1;
}
```

5.1. ábra. Éleken lévő százalékok beállításának feltételei

A könnyebb átláthatóság érdekében az 5.2 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.



5.2. ábra. Példafeladat

Az új sorrend a következő táblázatban látható. Zárójelben látható, hogy az 5.2 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 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 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.

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

Abban az esetben, ha a kezdeti taszk még nem lett ellenőrizve, akkor nem lehet megvizsgálni az éppen kiválasztott 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 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 értékként. 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 taszkokat. Ha megtalálta akkor annak kapacitása és a terméken szerzett jövedelem szorzata megadja a korlátot.

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.

6. fejezet

Tesztelés

7. fejezet

Összefoglalás

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 Enginnering.

[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.