SZAKDOLGOZAT

S-gráf alapú várható profit maximalizálás sztochasztikus környezetben

Dunár Olivér

Mérnök Informatikus BSc szak

2019

Nyilatkozat

Alulírott, Dunár Olivér (BOUE9E), Mérnök Informatikus BSc szakos hallgató kijelentem, hogy az „S-gráf alapú várható profit maximalizálás sztochasztikus környezetben” 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, 2019. ____________________________

hallgató

Kivonat

S-gráf alapú várható profit maximalizálás sztochasztikus környezetben

Szerző: Dunár Olivér, mérnökinformatikus BSc

Témavezető: Dr. Hegyháti Máté, tudományos főmunkatárs

Munka helyszíne: Széchenyi István Egyetem, Informatika tanszék

Szakaszos gyártórendszerek ütemezési problémáinak megoldására a szakirodalomban számos publikált módszer fellelhető, mint például a MILP modellek, vagy az általam részletesen vizsgált S-gráf módszertan.

Munkám során sztochasztikus környezetben adott szakaszos gyártórendszerek ütemezésével foglalkoztam, az S-gráf módszertan segítségével. Erre a témára azért esett a választásom, mert egy valós ipari problémáról beszélhetünk, hiszen sok esetben nem lehetséges a szakaszos gyártórendszerek ütemezésével kapcsolatos problémák paramétereinek leírása determinisztikus módon. Ezen probléma kör megoldása szolgáló tervezett elméleti algoritmusok, habár a szakirodalomban megtalálhatóak, ezek keretrendszerbe történő implementálása még váratott magára.

Először is részletesen megismerkedtem az S-gráf keretrendszerrel, a módszertanhoz kapcsolódó különféle determinisztikus algoritmusok működésével. Ezután áttanulmányoztam a sztochasztikus profit maximalizáláshoz az irodalomban található elméleti algoritmusokat, kidolgoztam ezen módszerek részleteit, majd a meglévő S-gráf keretrendszerbe implementáltam ezeket. Ezen implementáció során több alkalommal a már korábban, mások által az S-gráf keretrendszerbe implementált kód refaktorálására is szükség volt az egységes működés elérése érdekében. Éppen ezért az implementációt követően nem csupán a sztochasztikus-, de a determinisztikus profit maximalizáló is alapos tesztelésen esett át.

Munkám eredményeképpen az S-gráf keretrendszer immáron képes sztochasztikus környezetben adott szakaszos gyártórendszerek ütemezésére.

Kulcsszavak: S-gráf, profit maximalizálás, sztochasztikus, ütemezés

Abstract

S-graph based expected profit maximization in stochastic environment

There are many different published methods for solving the scheduling problems of batch production systems. These include different MILP models for example, and the S-graph framework, which is in the focus of this thesis.

During my work I researched different S-graph based algorithms, to provide a method for solving scheduling problems of batch production systems in stochastic environment. This is especially interesting because in real life industry examples, the problem often, cannot be defined only by using deterministic parameters, but with the inclusion of stochastic parameters. Furthermore, even though there are theoretical algorithms defined for solving these kinds of problems, these algorithms were yet to be implemented in the S-graph framework.

Firstly, I started my work by getting to know the S-graph framework and understanding the different deterministic algorithms it offers. After that I researched the theoretical algorithms defined for the stochastic profit maximization, I worked out the details of these methods, and finally implemented them in the S-graph framework. Since there were several occasions, when I had to refactor existing code, previously written by others, to achieve unified operation of the different modes, thorough testing of not just the new stochastic algorithms, but the already existing deterministic profit maximiser were required after the implementation of the code.

As a result of my work, the S-graph framework is now capable of handling the scheduling problems of batch production systems defined in stochastic environment.

Tartalomjegyzék

Ábrák jegyzéke

1. fejezet

Bevezetés

Az élet szinte minden területén találkozhatunk ütemezési problémákkal, például a logisztika területén, ha csak arra gondolunk, hogy egy futárszolgálatnak meg kell tervezni az optimális kiszállítási időpontokat, az útvonalat annak érdekében, hogy időben kézhez kaphassunk az általunk rendelt termékeket, de vehetjük példának akár a tömegközlekedést is, ahol a menetrendek megtervezése szintén ütemezési feladat. Számtalan példát fel lehetne még sorolni, azonban az belátható, hogy bármennyire is eltérhetnek ezen példák valamely aspektus szerint, ezek lényege minden esetben megegyezik: a cél az elvégzendő feladatokat szétosztása az elérhető erőforrások között, valamint ezen feladatok egy idő intervallumhoz rendelése oly módon, hogy a keletkező ütemterv valamilyen célkitűzés szempontjából a legkedvezőbb legyen, valamint a probléma definiálása során meghatározott korlátozásoknak megfeleljen. Az ütemezési feladatok egyik leggyakoribb megjelenési formája az iparban a gyártórendszerek ütemezése, mely során fellépő problémák megoldására számos megoldó módszer fellelhető a szakirodalomban. Ezen problémák csoportosításáról, néhány megoldó módszer ismertetéséről, valamint az ütemtervek vizualizációjáról szól a 2. fejezet. A 3. fejezetben részletesebben bemutatásra kerül az egyik megoldó módszer, az S-gráf keretrendszer, amely a dolgozatom további fókuszát képezi. A szakdolgozat témáját képező sztochasztikus profit maximalizálási probléma a 4. fejezetben kerül részletesen definiálásra. Az 5. fejezetben az S-gráf keretrendszerben definiált algoritmusok megoldására kifejlesztett S-gráf solver program kerül röviden bemutatásra. A 4. fejezetben definiált probléma megoldásához felhasznált módszerek részletes leírását, valamint az S-gráf solver program továbbfejlesztése közben végzett munka részletes dokumentációját a 6. fejezet tartalmazza. A solver programon végzett fejlesztési munkák végezte után szükséges tesztelés menetét írja le a 7. fejezet. A 8. fejezet képezi a dolgozat, valamint az elvégzett munka összefoglalását, értékelését, valamint a jövőbeli esetleges továbbfejlesztési lehetőségek ismertetését. A dolgozat végén található a melléklet, mely három függelékből áll: az A függelék tartalmazza a jelmagyarázatot, a solver program példa input fájljai a B függelékben találhatóak, a C függelék tartalmazza a CD melléklet tartalmának leírását.

2. fejezet

Irodalmi áttekintés

2.1. Gyártórendszerek ütemezése

Ahogyan az már a bevezetésben is említésre került az ütemezési problémák nagy hányadát a gyártásütemezési feladatok teszik ki, melyek elvégzése során a cél általában az átviteli kapacitás, vagy egyszerűen profit (throughput)[1] maximalizálása. Másik fontos célkitűzés lehet a ”makespan”, vagyis a feladatok elvégzéséhez szükséges idő minimalizálása. A gyártásütemezési problémák esetében a megoldandó feladatok alatt általában a késztermékek egy részének, összetevőjének a legyártása értendő, az erőforrások, amik között a feladatok kiosztásra kerülnek pedig nem mások mint az elérhető gépek, berendezések, amelyek az adott rész legyártására képesek, adott továbbá az is, hogy melyik berendezés melyik feladatot mennyi idő alatt képes elvégezni, valamint a részfeladatok elvégzésének betartandó sorrendje sorrendje. Ezen paraméterek együtt alkotják az un. receptet, mely a feladatok precedenciája alapján a következő kategóriákra bontható:

Single stage recept: Minden termék egy lépésben állítható elő.

Simple multiproduct recept: Minden termék több lineárisan egymást követő lépésből áll elő.

[1] A szakirodalomban általában az angol ”throughput” kifejezés a használatos, ha profit maximalizálásról, ”makespan”, ha idő minimalizálásról van szó, éppen ezért dolgozatomban a továbbiakban én is ezen kifejezések használatára törekszem.

General multiproduct recept: Az előző fajta speciális esete, ahol a lépések tetszőlegesen kihagyhatóak.

Multipurpose recept: A lépéseknek nincs meghatározott, lineáris sorrendjük, tetszőleges sorrendben hajthatóak végre.

Precedential recept: A lépések nem lineárisak, lehetnek elágazások is a lépések között (kör nem lehetséges), egy lépésnek akár több megelőző lépése is lehet, melyek teljesítése a következő lépés megkezdésének a feltétele.

General network recept: A legáltalánosabb recept kategória, melyben a feladatok a bemenetükkel, illetve a kimenetükkel adottak, indirekt módon meghatározva ezzel a precedenciákat.[2]

A receptek tulajdonságain kívül fontos tényező, különösen vegyi üzemek termelésének ütemezése során a különböző tárolási irányelvek (storage policy) betartása.[3] Tárolási irányelvek alatt azokat a megkötéseket értjük, amelyek a köztes termékek tárolására vonatkoznak a recept két egymást követő feladata között. A tárolási irányelvek alapvetően két szempont szerint csoportosíthatóak, az egyik az idő, ameddig a köztes termék tárolható anélkül például, hogy bizonyos kémia, fizikai tulajdonságai megváltoznának, amelyre egyébként a következő feladat során szükség lenne (pl. ne hűljön ki a termék). A másik szempont pedig a mennyiség, azaz, hogy adott köztes termékből adott üzemben mennyit lehet eltárolni. Ezek alapján a különböző kategóriák azonosíthatóak be:

UW - Unlimited Wait: A legegyszerűbb, és leggyakoribb eset, melyben a köztes termék bármennyi ideig eltárolható a következő feladat megkezdése előtt.

LW - Limited Wait: Ebben az esetben a köztes termék nem tárolható egy megadott időnél tovább a következő feladat előtt, hogy ne veszítse el valamilyen tulajdonságát.

ZW - Zero Wait: Ebben az esetben a köztes termék tárolásának megengedett ideje 0.

UIS - Unlimited Intermediate Storage: Ebben az esetben végtelen mennyiséget el tudunk tárolni az adott köztes termékből.

FIS - Finite Intermediate Storage: Az az eset, melyben a köztes termék tárolása megoldható, de véges számú a tároló kapacitás.

NIS - No Intermediate Storage: Ebben az esetben nem állnak rendelkezésre tároló egységek a köztes termék eltárolására, de a köztes termék továbbra is várakozhat az aktuális feladatot végrehajtó egységben.

A recepteken és a tárolási irányelveken kívül a gyártórendszerek ütemezésével kapcsolatos problémák is több szempont szerint is csoportosíthatóak[3]:

- Az ütemezés időpontjában elérhető paraméterek szerint:
  - Offline ütemezésről beszélünk akkor, ha minden szükséges adat rendelkezésre áll az ütemezés megkezdésekor.
  - Online ütemezésnek nevezzük azon eseteket, mikor az ütemezés megkezdésekor még nem minden paraméter áll rendelkezésre, ezért bizonyos paraméterek hiányában kell meghozni az ütemezési döntéseket.
- A bizonytalanságok alapján:
  - Determinisztikus problémának nevezzük azon eseteket, amelyekben minden paraméter értéke adott már a kidolgozott ütemterv megvalósítása előtt.
  - Sztochasztikus a probléma ezzel szemben, ha valamely paraméter értéke csak az ütemterv végrehajtása során (pl. termelés közben) válik világossá.

Az ütemezési problémák megoldásai is osztályozhatóak aszerint, hogy a megoldás megvalósítható (feasible), vagy nem valósítható meg (infeasible). A "megoldás" alatt általában magát az ütemtervet értjük, amely nem más mint az összes feladat hozzárendelése berendezésekhez, illetve időintervallumokhoz. Infeasible megoldásról akkor beszélünk, ha az ütemterv nem elégíti ki a probléma definiálása során lefektetett megkötések akár egyetlen egy tagját is, ezzel szemben feasible a megoldás, ha az ütemterv minden megkötést kielégít. A feasible megoldások további csoportját képezik a non-delay megoldások, amelyek esetén egyetlen egy berendezés esetén sem beszélhetünk üresjáratról, ha akár csak egy feladat is megvalósításra vár.

2.2. Megoldó módszerek

Az utóbbi évtizedekben számos különböző módszer került publikálásra, melyek az ütemezési problémák megoldására hivatottak. Az évek során ezek a módszerek jelentős javuláson mentek keresztül mind a megoldható problémák halmazát tekintve, mind a megoldáshoz szükséges idő gyorsaságát figyelembe véve. A következő alpontokban néhány megoldó módszer kerül megemlítésre..

2.2.1. MILP modellek

A publikációk túlnyomó része matematikai programozáson, név szerint a Mixed-Integer Linear Programming, azaz MILP modelleken alapszik. A modellek általában univerzális MILP algoritmusokkal kerülnek megoldásra. [10] A MILP modelleknek különböző típusai léteznek:

Időfelosztásos módszerek (Time discretization based) : Az időhorizont diszkrét pontokra, vagy résekre bontásán alapszanak, mely szuboptimális, vagy akár infeasible megoldáshoz is vezethet, azonban ezen módszerek segítségével az ütemezési problémák széles skálája valósítható meg. Időrendi szempontból ezen modellek jelentek meg először a szakirodalomban. [5] Minden időpontban bináris változók vannak hozzárendelve a feladatokhoz, melyek értéke jelenti, hogy adott feladat végrehajtása az adott időpontban kezdődik el vagy sem. Jól látható, hogy a változók száma arányos a kiválasztott időpontok számával, ezáltal az ütemezés teljesítéséhez szükséges számítási idő nagyban függ a diszkrét időpontok számától, éppen ezért mindig is kutatási szándék volt olyan módszer fejlesztése, amely képes az optimális megoldás meghatározására minimális diszkrét időpont alapján.

Precedencia alapú módszerek (Precedence based) : Az időfelosztásos módszerekkel szemben a precedencia alapú módszerek esetén nincs szükség az időhorizont diszkretizációjára, ezáltal nem használnak ismeretlen paramétereket a megoldás során. A megoldható problémák halmaza kisebb mint az előző típus esetében, azonban a megoldható problémákra alapvetően jobb megoldást szolgáltatnak a precedencia alapú módszerek. A legtöbb modell multiproduct és multipurpose receptekre lett bevezetve, azonban többségük egyszerűen kibővíthető, hogy általánosabb problémák megoldására is használható legyen. A módszerek alapja két különböző bináris változó halmaz: $Y_{i,j}$, melynek értéke 1, ha i feladatot j berendezéshez rendeljük, valamint $X_{i,j,i'}$, mely akkor vesz fel 1 értéket, ha i és i' feladatokat egyaránt j berendezéshez rendeljük, méghozzá úgy, hogy i elvégzése megelőzi i' elvégzését j berendezésen.

2.2.2. Időzített automaták, Időzített Petri háló

A Petri hálók, és automaták gyakran használatosak diszkrét eseményrendszerek modellezésére, ahhoz azonban, hogy ezen módszerek felhasználhatóak legyenek gyártásütemezési problémák megoldására is, időzítésekkel kellett kiegészíteni őket. Így születtek meg a Timed Place Petri Nets (TPPN), Timed Priced Automata (TPA) módszerek. Ezen módszerek Branch & Bound algoritmust használnak az állapottér bejárására, és a legkedvezőbb megoldás megtalálására. A hatékony modellezésnek köszönhetően a modellezési hibák könnyen kikerülhetőek, valamint a módszer könnyen kibővíthető a reaktív ütemezési feladatok kezelésére is, ennek ellenére azonban ezen módszerek hatékonysága elmarad a MILP modellek, illetve az S-gráf modell hatékonyságától is.

2.2.3. S-gráf keretrendszer

Az S-gráf keretrendszer, amely ezen szakdolgozat fő fókuszát képezi, volt az első publikált gráf elméleten alapuló módszer szakaszos gyártórendszerek ütemezésére. Az S-gráf keretrendszer részletesebb bemutatása a 3. fejezetben olvasható.

2.3. Az ütemtervek vizualizációja

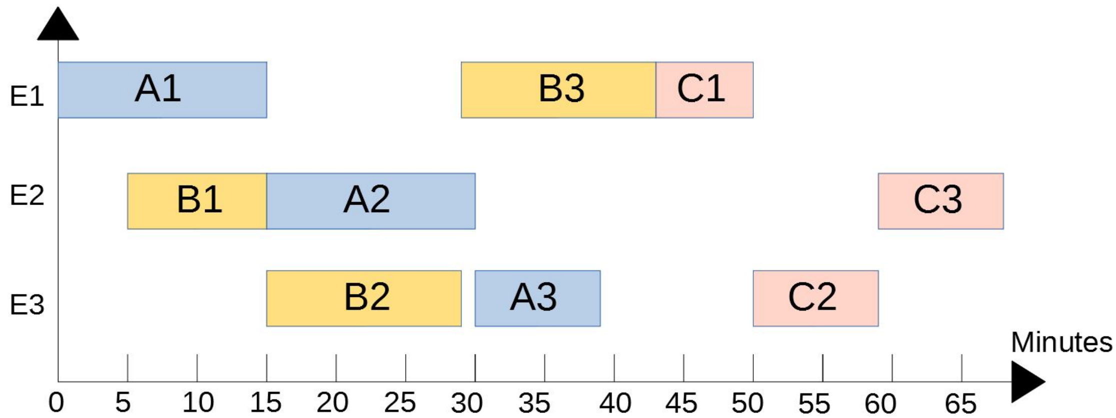
Az ütemezési problémák megoldása során keletkezett ütemterveket Gannt-diagrammal szokás ábrázolni, mely nevét feltalálójáról Henry Gantt amerikai mérnökről kapta. [1] Az ütemtervek lényegében a következő formájú négyesekből állnak: (i, j, t^s, t^f), ahol

i az elvégzendő feladat

j az i feladatot elvégző berendezés

t^s a feladat elvégzésének kezdeti időpontja

t^f a feladat elvégzésének vég időpontja



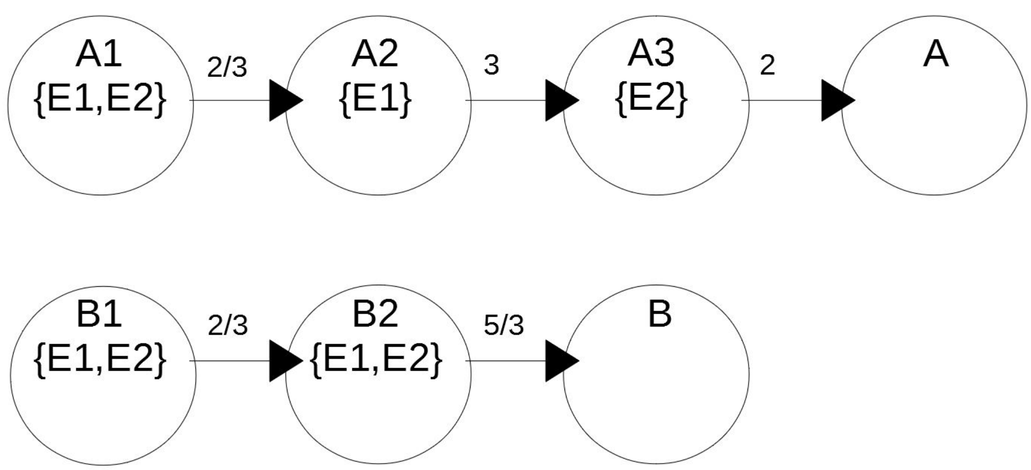
2.1. ábra. Gannt-diagram egy ütemterv szemléltetésére

Egy Gannt-diagram szemléltetésére szolgál a 2.1 ábra. Látható, hogy az y tengelyen a berendezések nevei (E1,E2,E3) találhatóak, míg az x tengely az idő szemléltetésére hivatott. Az ábra három különböző termék (A,B,C), valamint azok részfeladatainak ütemezését szemlélteti, melyeket különböző színekkel jelölünk. A Gannt-diagramos ábrázolás segítségével az ütemtervet képező négyesek minden paramétere egyszerűen leolvasható. Az elvégzendő feladat neve a színes téglalapokon találhatóak (pl. C1), a feladatot elvégző berendezést az adott téglalap y tengelyen vett helye határozza meg (pl. C1 esetében E1 berendezés). A feladatok elvégzésének a kezdeti, illetve vég időpontja pedig az téglalapok x tengelyen vett helye alapján könnyen beazonosíthatóak, meghatározva ezáltal a feladat teljesítéséhez szükséges időt is.

3. fejezet

Az S-gráf keretrendszer

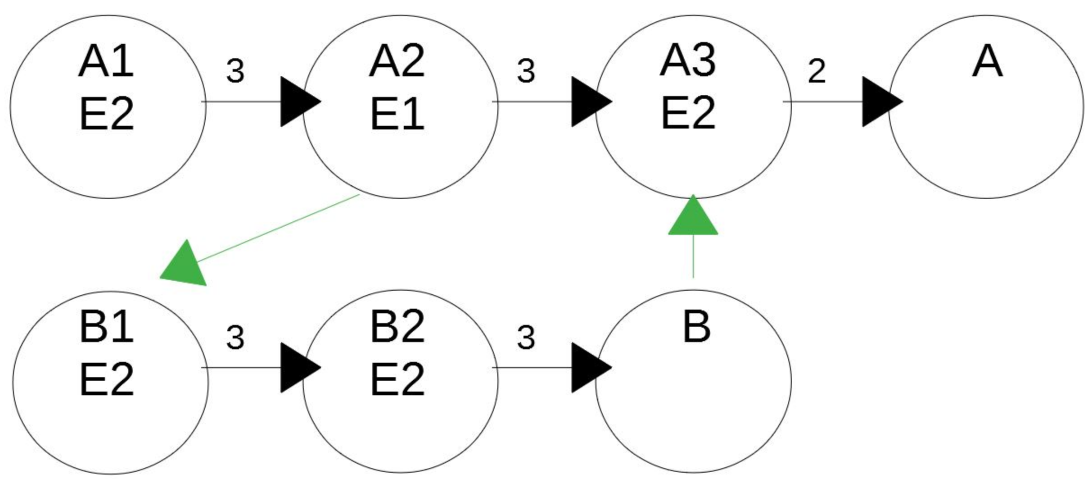
Az S-gráf keretrendszer volt az első publikált gráf elméleten alapuló módszer szakaszos gyártórendszerek ütemezési problémáinak megoldására. [8] A keretrendszer egy irányított gráf modellből, az S-gráfból és a hozzá tartozó algoritmusokból áll. [7] Az S-gráf egy speciális irányított gráf, amely nem csupán a probléma vizualizációjára képes, hanem egy matematikai modell is. A keretrendszerben a recepteket, valamint a félkész-, illetve a teljes ütemterveket is az S-gráf reprezentálja. Ezekben a gráfokban a termékeket, illetve a feladatokat a pontok jelölik, amelyeket csomópontoknak nevezünk. Ezenkívül, ha két feladat között összeköttetés van, ezt a gráfon a két feladatot reprezentáló csomópontok közötti nyíl jelöli. Az ütemezési információ nélküli S-gráfot recept gráfnak (**recipe graph**) nevezzük, melyre egy példa a 3.1 ábrán látható.



3.1. ábra. A recept gráf szemléltetése

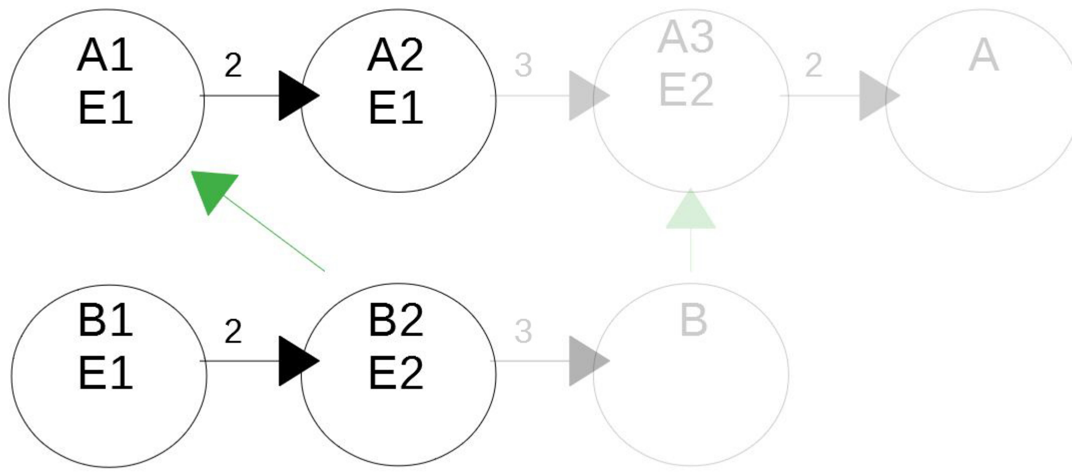
Az ábrán látható jobb oldali kettő csomópont (A,B) jelöli a termékeket, a többi csomópont pedig a részfeladatokat, amelyeket el kell végezni a termékek előállítása érdekében. A recept gráf nyilai reprezentálják egyrészt két részfeladat közötti függőséget, abban ez esetben például, ha ez egyik részfeladat állítja elő a másik részfeladathoz szükséges bemeneti résztermékét, másrészt a részfeladatok és a késztermékek közötti függőséget. A recept gráf minden részfeladathoz tartozó csomópontjához tartozik egy halmaz, amely azon berendezések nevét tartalmazza, amelyek képesek adott részfeladat megoldására. A nyilakon található súlyok pedig a részfeladat megoldásához szükséges gyártási időt reprezentálják, abban az esetben, ha egy részfeladatot több berendezés is el tud végezni, a nyíl súlya a berendezésekhez tartozó gyártási idők közül a legkisebb lesz.

Az S-gráf keretrendszerben található algoritmusok az előzőekben bemutatott recept gráfot egészítik ki ütemezési nyilakkal, amelyek az algoritmus által meghozott ütemezési döntéseket reprezentálják. Az ily módon előállított gráfot, függetlenül attól, hogy van-e még meghozatlan ütemezési döntés, vagy pedig a teljes ütemezés megtörtént, ütemezési gráfnak (**schedule graph**) nevezzük. A 3.1 pontban látható recept gráf alapján előállított ütemezési gráf a 3.2 ábrán látható.



3.2. ábra. Az ütemezési gráf szemléltetése

Az ábrán látható gráfon már minden ütemezési döntés lezajlott, a zöld nyilak jelölik az ütemező algoritmus által behúzott ütemezési nyilakat. A részfeladatokat reprezentáló csomópontokon immáron a lehetséges berendezések halmaza helyett egy konkrét berendezés jelölése található, amely az adott részfeladat elvégzésére hivatott az adott ütemterv szerint. Az ütemezési nyilak súlya alapértelmezetten 0, ha az adott problémában nem számolunk például részfeladatok közötti szállítási-, átállási-, illetve tisztítási időkkel. Az adott berendezéshez rendelt részfeladatok sorrendje könnyen leolvasható az ütemezési gráfról, erre egy példa a 3.3 ábrán látható.



3.3. ábra. Az E1 berendezéshez rendelt részfeladatok sorrendje

Az ábra alapján leolvasható, hogy az E1 berendezés először a B1 részfeladatot végzi el, majd a A1, azután pedig az A2 fog következni. Ahhoz azonban, hogy például a A3 részfeladatot E2 berendezés el tudja végezni, nem elegendő az, hogy B termék gyártása befejeződjön, elengedhetetlen az is, hogy minden részfeladat amitől A3 függ (nevezetesen A2) szintén véget érjen.

Az S-gráf keretrendszerben történő ütemezésre egy bővebb példa a makespan minimalizáló [1] algoritmuson keresztül szemléltetve megtalálható a CD melléklet **Algoritmusok** mappájában **DO_Sgraph_Makespan_Minimization.odg** néven.

[1] A makespan minimalizáló algoritmus segítségével adott receptgráffal reprezentált termékek gyártási ideje minimalizálható. Az algoritmus fontos szerepet játszik a 3.1. alfejezetben tárgyalt profit (throughput) maximalizáló algoritmusban is.

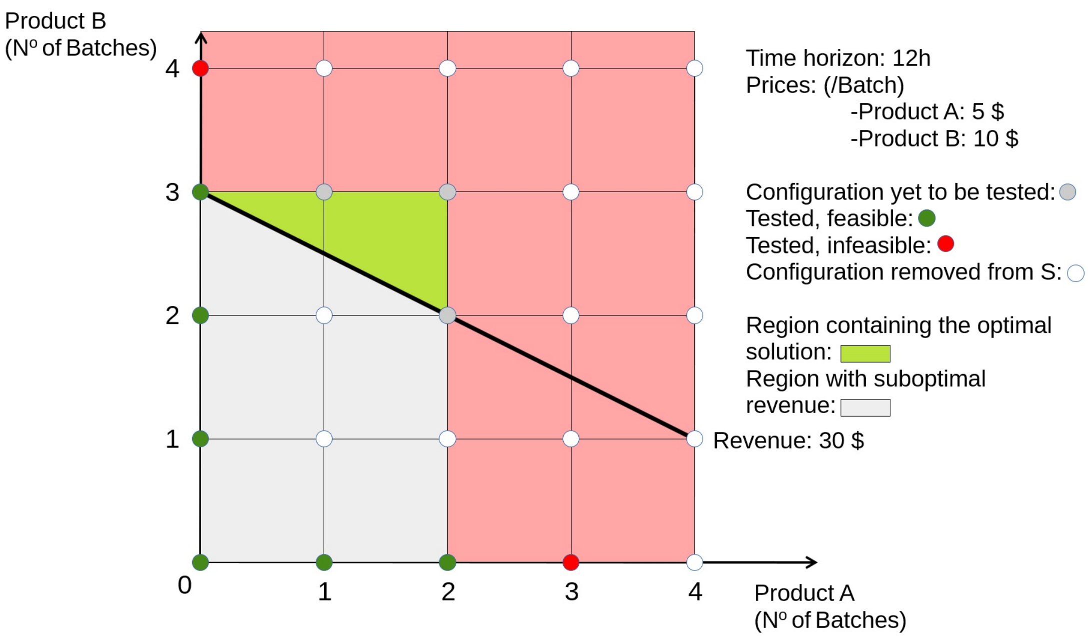
3.1. Profit maximalizálás az S-gráf keretrendszerrel

Az S-gráf keretrendszer eredetileg makespan minimalizációs célokra lett megalkotva, azonban később kibővítésre került egy throughput maximalizáló algoritmussal, amely segítségével immáron profit maximalizálásra is képes. Az throughput maximalizáló algoritmus alapötletét Majozi és Friedler [6], valamint Holczinger [4] fektették le, melynek lényege, hogy a termékek lehetséges batch darabszámai alapján különböző konfigurációk kerülnek létrehozásra[2], melyek között branch & bound algoritmus segítségével eredményül kapható a legnagyobb profitot eredményező konfiguráció, ha létezik megvalósítható (feasible) megoldás a problémára.

A throughput maximalizáló algoritmus működésének egy példán keresztül történő szemléltetésére készült folyamatábra a CD melléklet **Algoritmusok** mappájában található **DO_Sgraph_Throughput_Maximization.odp** néven. Kezdetben a konfigurációk halmaza tartalmazza az összes lehetséges konfigurációt az adott termékekre, majd minden iteráció során kiválasztásra kerül egy konfiguráció, amelynek teszteljük feasible-itását, azaz hogy a rendelkezésre álló időhorizont alatt megvalósítható-e adott konfiguráció legyártása. Ennek tesztelése a korábban már említett makespan minimalizáló algoritmus felhasználásával a legegyszerűbb. A makespan minimalizáló algoritmusnak átadásra kerül adott konfiguráció recept gráfja, majd az eredményül kapott idő érték összehasonlításra kerül a rendelkezésre álló időhorizonttal, ha a kapott érték nagyobb annál, az adott konfiguráció nem valósítható meg (infeasible). Ha az adott konfiguráció megvalósítható a rendelkezésre álló időhorizont alatt, kiszámításra kerül az adott konfiguráció által nyújtott profit, ha ez nagyobb az eddigi legjobb értéknél, frissíteni kell a legjobb értéket adott konfiguráció revenue értékével. Kezdetben a tengelyek menti konfigurációk kerülnek tesztelésre mind addig, még az összes tengelyen el nem jutunk az első megvalósíthatatlan (infeasible) konfigurációig. Ezzel előáll egy tér, amely tartalmazza az optimális megoldást, amelyet már csak meg kell találni, mivel azonban a konfigurációk tesztelése erőforrás igényes feladat, ezért a konfigurációk kiválasztásának sorrendje nem mindegy, többféle stratégia létezik az algoritmus gyorsítására. [3] Az egyik ilyen stratégia az úgynevezett revenue line behúzása, mely segítségével szerencsés esetben akár töredékére csökkenthető a

[2] A és B termékek esetén egy lehetséges konfiguráció például, ha A termékből 1 darab, B-ből 2 darab batch-et gyártunk.

tesztelendő konfigurációk száma. A revenue line segítségével lényegében eltávolításra kerülnek azok a konfigurációk, melyek profitja nem éri el az aktuális legjobb profit értéket, azaz a behúzott vonal alatt vannak. A revenue line szemléltetésére a 3.4 ábra hivatott, mely a már fent említett **DO_Sgraph_Throughput_Maximization.odp** fájl részlete.



3.4. ábra. A revenue line szemléltetése

Jól látható, hogy jelen esetben a revenue line behúzása felére csökkentette az ellenőrizendő konfigurációk számát, gyorsítva ezzel az algoritmus lefutását. Az algoritmus akkor ér véget, ha a konfigurációk halmaza kiürül, ebben az esetben ha egyetlen egy megvalósítható megoldás sem található, a probléma megoldása lehetetlen az adott időhorizont alatt. Ha található feasible megoldás, az algoritmus az optimális konfigurációt adja eredményül, megkapva ezzel a maximális profit értékét, valamint annak előállításához szükséges ütemtervet.

4. fejezet

Problémadefiníció

A probléma lényege abban keresendő, hogy valódi ipari környezetben gyakran előfordulnak olyan esetek, amelyekben a megoldandó problémát nem lehet tisztán determinisztikus változókkal leírni, ezeket felhasználva megoldani, hanem szükség van új, a bizonytalanságokat kifejező, sztochasztikus változókra is. Változó piaci környezetben ilyen sztochasztikus paraméternek számítanak például a termék iránti kereslet, illetve a piaci ár, amin a terméket értékesíteni lehet. Belátható az is, hogy ezek a paraméterek sokban befolyásolják a maximalizálandó profitot. Vegyünk például egy olyan esetet, amelyben a keresletnél többet termeltünk, ez esetben a keletkező többletet nem tudjuk értékesíteni, ez akár további kiadásokkal is járhat a többlet termék esetleges tárolási költsége miatt. Jól látható, hogy az ilyen fajta problémák jóval másabbak, mint az eredeti determinisztikus problémák, éppen ezért egy új módszer kidolgozása szükséges ezek megoldásához. Szakdolgozatom célja a 6.1 pontban bemutatott matematikai módszerek segítségével az S-gráf keretrendszer felkészítése a különböző sztochasztikus paraméterek kezelésére, oly módon, hogy az eredeti determinisztikus throughput maximalizáló algoritmus sértetlen maradjon, a probléma típusától függően a determinisztikus, illetve a sztochasztikus throughput maximalizáló alkalmazása egyaránt lehetséges legyen.

4.1. A probléma csoportosítása

A megoldandó problémák a sztochasztikus esetben is a determinisztikus throughput maximalizálásnál használt paraméterekkel adottak, pl.: minden terméket a receptje azonosít be, ezen kívül adott a termékek előállítására használható berendezések halmaza, illetve a termelésre rendelkezésre álló időhorizont. Az determinisztikus paramétereken kívül azonban sztochasztikus esetben különböző bizonytalanságot kifejező paraméterek is adottak minden termékre, amelyek valószínűségeit különböző scenariokba, forgatókönyvekbe csoportosítjuk. Ezáltal minden forgatókönyvre adott:

- A forgatókönyv valószínűsége
- A termék ára (1 batch ára) az adott forgatókönyvben
- A termék iránti kereslet az adott forgatókönyvben
- A túltermelés, az alul termelés költsége az adott forgatókönyvben

A feladat az, hogy döntést hozzunk a termelt batch-ek darabszámát illetően, miközben egy olyan kivitelezhető ütemtervet biztosítunk, amelyet követve maximális várható profitot érhetünk el. A batch méretekkel kapcsolatos döntések alapján 3 eset különböztethető meg:

- **Preventív ütemezés kötött batch mérettel** Ebben az esetben minden termékhez adott egy batch méret, az egyetlen preventív döntés amit hoznunk kell, hogy hány darab batch-et gyártunk az adott termékből.
- **Preventív ütemezés változó batch mérettel** Ebben az esetben nem csak a batch darabszám ,de annak a mérete is kiválasztható, de csak preventív módon a bizonytalan események bekövetkezése előtt.
- **Két lépcsős ütemezés (two stage)** Ebben az esetben a batch darabszámot előre ki kell választanunk, azonban annak a méretéről a bizonytalan események bekövetkezése után is döntést hozhatunk.

Kezdetben feltételezzük, hogy a receptek és a termékek között 1-1 kapcsolat van, azaz egy recept sem eredményez több terméket, illetve egyetlen termék sem állítható elő több fajta recepttel. A 6.5 pontban azonban kitérek azokra az esetekre, amelyekben ez a feltételezés nem állja meg a helyét.

4.2. A probléma paraméterei

A 4.1 pontban bevezetett sztochasztikus esetek kezeléséhez az determinisztikus throughput maximalizáló algoritmus jelentős része felhasználható változtatások nélkül (vagy csak minimális változtatások árán, lsd. 6.4 pont). Az egyetlen meghatározó különbség az un. ”revenue” függvényben figyelhető meg, amely célja, hogy az adott konfigurációra nézve kiszámítsa a várható profitot. A probléma megoldása során használt paraméterek:

P a termékek halmaza

b_p a legyártott batch-ek darabszáma az adott konfigurációban

s_p a termék batch mérete (fix batch méret esetén) [Kg]

s_p^{min}, s_p^{max} adott termékhez tartozó lehetséges legkisebb, legnagyobb batch méret (válzotó batch méret esetén) [Kg]

S a forgatókönyvek halmaza

$prob_s$ s forgatókönyv valószínűsége $s \in S$

$dem_{s,p}$ p termék iránti kereslet az s forgatókönyvben $s \in S, p \in P$ [Kg]

$price_{s,p}$ p termék ára az s forgatókönyvben $s \in S, p \in P$ [Cost Unit/Kg]

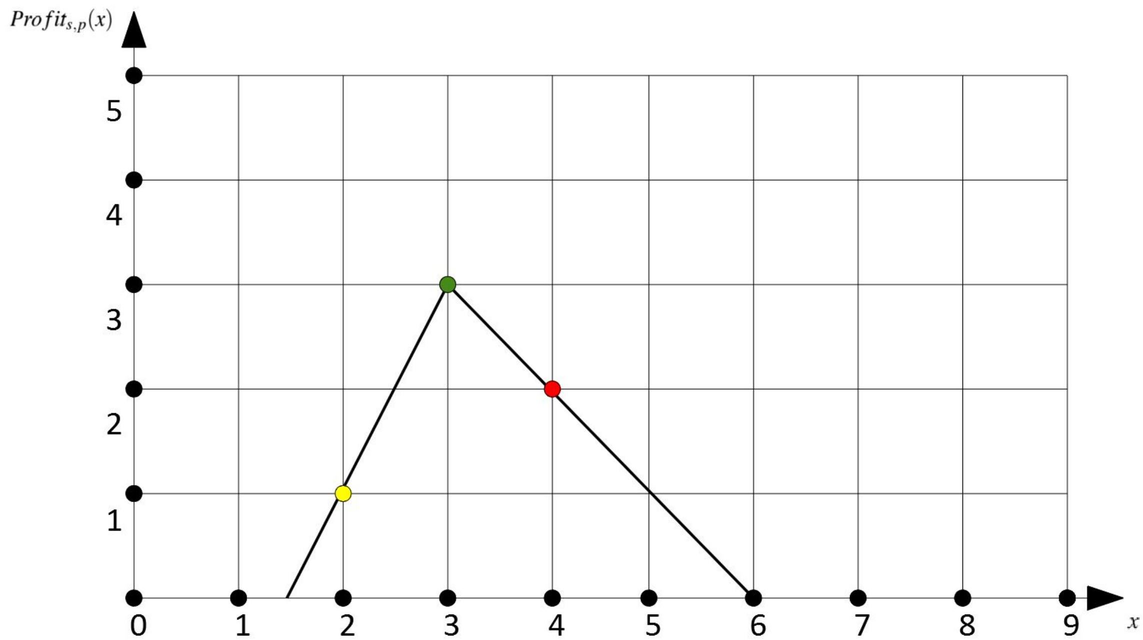
$oc_{s,p}, uc_{s,p}$ p termék túl-, és alul termelési költsége s forgatókönyvben $s \in S, p \in P$ [Cost Unit/Kg]

$ExpProfit_p(x) = \sum_{s \in S} prob_s \cdot profit_{s,p}(x)$ p termék x értékben vett várható profit értéke

Ezenkívül érdemes bevezetni még a $Profit_{s,p}(x)$ jelölést, amely megadja x mennyiségű p termék bevételét az adott s forgatókönyvben:

$$Profit_{s,p}(x) = \begin{cases} price_{s,p} \cdot x - (dem_{s,p} - x) \cdot uc_{s,p} & \text{ha } x < dem_{s,p} \\ price_{s,p} \cdot dem_{s,p} - (x - dem_{s,p}) \cdot oc_{s,p} & \text{egyébként} \end{cases}$$

Jól látszik, hogy adott termék bevétele akkor lesz maximális, ha a kereslettel egyező da-



4.1. ábra. A profit függvény szemléltetése a következő paraméterekkel:

$$s_p = 1, \quad dem_{s,p} = 3, \quad oc_{s,p} = 1, \quad uc_{s,p} = 1$$

rabszámot gyártunk az adott termékből (zöld pont a 4.1 ábrán), ha ennél kevesebbet gyártunk a termékből, akkor a kereslet kielégítéséből eredő profit is elmarad, illetve további többlet költség kerül levonásra a profit összegéből az esetleges alul termelési plusz költségek miatt (pl. sárga pont a 4.1 ábrán), abban az esetben pedig, ha a keresletet meghaladó mennyiséget gyártunk adott termékből, a kereslet kielégítődik ugyan, és bevételünk maximális lenne az adott piaci keresletet figyelembe véve, azonban a túltermelés következtében létrejött többlet tárolási költségét le kell vonjuk a profit értékéből (pl. piros pont a 4.1 ábrán) [1]. Arra kell törekedni tehát, hogy a lehetőségeket mérlegelve minden termékből annyit gyártsunk, hogy az az adott forgatókönyvben szereplő keresletet kielégítse, vagy azt a legkedvezőbb módon megközelítse valamelyik irányból, ügyelve az alul-, és túltermelési költségekre. Extrém esetekben előállhat olyan helyzet is, hogy a rendelkezésre álló determinisztikus paraméterek (pl.

[1] Előfordulhat olyan eset, amelyben az alul-, és túltermelési költségekkel nem kell számolni, ebben az esetben a kereslet értékének eléréséig a profit a termelt mennyiséggel arányosan lineárisan növekedni fog, majd onnantól kezdve konstans módon beáll a maximális értékre, ugyanis a felesleg értékesítése nem lehetséges, ha arra nincs kereslet, viszont annak tárolás nem okoz plusz költséget.

gépek száma), az aktuális időhorizont, illetve a sztochasztikus paraméterek aktuális értéke miatt a profit függvény x-ben felvett értéke negatív szám lesz, ez esetben inkább a veszteségek minimalizálásáról beszélhetünk, mintsem profit maximalizálásról, azonban könnyen belátható, hogy a definiált matematikai modellekben amelyeket használunk a profit kiszámítására, ez semmiféle változást nem eredményez, csupán arra kell figyelni, hogy az implementáció során felkészüljünk a negatív számok a programnyelvben történő kezelésére.

5. fejezet

Az S-gráf solver

Az S-gráf solver program egy C++ nyelven íródott, több szálas megoldó program, mely az S-gráf keretrendszerben foglalt algoritmusok segítségével különböző ütemezési problémákat képes megoldani. Jelenleg a NIS, UIS, UW, és LW tárolási irányelveket támogatja, valamint a következő célfüggvényekkel használható: makespan minimalizáció, throughput maximalizáció, cycle time minimalizáció. A solver parancssorból futtatható, különböző parancssori kapcsolók teszik lehetővé a különböző funkciókhoz tartozó paraméterek beállítását. Néhány, a throughput maximalizáláshoz fontos kapcsoló:

-i, –input [file] Az input fájl elérési útja, a fájl kiterjesztése .xml, vagy .ods lehet

-o, –output [file] Az output fájl elérési útja, a fájl kiterjesztése lehet .txt, vagy .png attól függően, hogy szöveges, vagy grafikus megjelenítést szeretnénk eredményül kapni

–obj [objective function] A célfüggvény típusa, throughput maximalizálás esetén ezen kapcsoló értéke: thmax

–timehor [time horizon] A throughput maximalizáláshoz rendelkezésre álló időhorizont mérete

A fentiek alapján a solver futtatására egy példa:

```
solver -i input-stoch.ods -o output.png --timehor 15 --obj thmax
```

5.1. ábra. Példa a solver futtatására

A solver az OpenMP API-t használja párhuzamosításra, a kód lefordításához Qt5 szükséges. Ezenfelül Linux alatt történő futtatáshoz letöltendő még a Google OR-Tools könyvtár, Windows operációs rendszer esetén pedig a Visual C++ Redistributable, valamint a boost könyvtár telepítése a követelmény. A fentiek telepítése, illetve a megfelelő könyvtárak elérési útjainak beállítása után a solver qmake segítségével fordítható, melynek következtében létrejönnek a futtatáshoz szükséges fájlok, köztük a solver.exe (Windows esetén), mellyel immáron futtathatjuk a megoldó programot az 5.1. ábrához hasonló módon.

5.1. A solver működése

A szolver futtatásához szükséges parancssori kapcsolók lehetséges értékeit, és egyéb paramétereit (pl. min/max érték) leíró opciókat tartalmazza az **Arguments.cpp** fájl. A **MainSolver** osztály **getOptions** metódusa a beolvasott parancssori kapcsolók értékei alapján létrehozza a **SolverOptions** osztály egy példányát amely objektumtól ezután lekérdezhetőek a különféle beállítások. Ezután a **MainSolver** objektum **Run** metódusa meghívja a **ReadInputFromFile** metódust, ami egy **RelationalProblemReader** objektum példányosítása, majd annak **ReadSGraph** metódusának meghívása után vissza kapja a recept gráfot tartalmazó **SGraph** objektumot. A **RelationalProblemReader** osztály feladata az input fájlban található paraméterek beolvasása, parse-olása, majd a recept gráf elkészítése ezek alapján. Az **SGraph** osztály egy példánya lényegében egy S-gráfot reprezentál, ez lehet recept gráf, illetve ütemezési gráf is. Ezen osztály példányosításával, paramétereinek beállításával zajlik tehát lényegében az ütemezés. Miután a **MainSolver** osztály **Run** metódusa visszakapta a recept gráfot, meghívásra kerül a **GetProblem** metódus, aminek segítségével a kapcsolók, illetve a recept gráf alapján megállapításra kerül a probléma pontos típusa, ezután a **GetSolver** metódus példányosítja a probléma típus megoldásához szűkséges solvert. Ezen solver **Solve** metódusának meghívásával elkezdődik a probléma megoldása, melynek eredményeként megkapjuk a megoldást tartalmazó **TreeNode** objektumot, melyen keresztül elérhető az optimális megoldást reprezentáló ütemezési gráfot tartalmazó **SGraph** objektum. Throughput maximalizálás esetén ez a solver nem más, mint a **ThroughputSolver** osztály egy objektuma, melynek működésének leírása az 5.2 alfejezetben található. Az eredmény visszakapása után a **MainSolver** egy **SolutionWriter** objektum **Write** metódusának meghívásával kiíratja az eredményt a megfelelő formátumban, legyen az szöveges fájl vagy .png formátumú Gannt diagram.

5.2. A ThroughputSolver osztály működése

A **ThroughputSolver** lényegében a 3.1 alfejezetben leírtakat valósítja meg. Az osztály **Solve** metódusa először is megkeresi az optimális megoldást tartalmazó teret, ezt azzal éri el, hogy mind addig míg található feasible konfiguráció a tengelyek mentén, létrehoz egy újabb konfigurációt, amelyben az adott termékből gyártott batchek számát növeli. Ehhez a **FirstFeasible** metódust használja fel, mely a **SolveBest** metódus segítségével leteszteli az adott konfiguráció feasibilitását. Ha az adott konfiguráció megvalósítható, meghívásra kerül a **NewSolution** metódus, amely a konfiguráció alapján létrehoz egy új lehetséges megoldást reprezentáló objektumot, valamint elmenti a konfiguráció profit értékét, ha az nagyobb, mint az aktuális legjobb érték. A **SolveBest** metódus visszaadja a **FirstFeasible** metódus számára azt a mutatót, melyen keresztül az új megoldás elérhető, ha feasible volt a konfiguráció, ha nem volt az, akkor null érték kerül visszaadásra. Az első null érték visszakapása után a **Solve** metódus több konfigurációt már nem hoz létre, megtalálásra került ugyanis az optimális megoldást tartalmazó tér. Ezután következik ezen tér bejárása, az optimális megoldás megkeresése a lehetséges megoldások között. Erre azonban több stratégia is lehetséges, ezért a **Solve** metódus először is lekérdezi az ehhez kapcsolódó parancssori paraméterek értékét, majd ezek alapján meghívja a megfelelő bejárást végző metódust. Ez a metódus alapértelmezett esetben a **SearchThrSolution**, mely a 3.1 alfejezetben már említett revenue line gyorsítási stratégiát is használja a térben való kereséshez. A metódus lefutása után ideális esetben a **Solve** metódus számára elérhetővé válik az optimális megoldást reprezentáló objektum, mely visszaadható a **MainSolver** számára az eredmények kiíratása érdekében. Abban az esetben azonban, ha egyetlen lehetséges megoldás sem található a problémára, egy exception kerül eldobásra, melyet a **MainSolver** megfelelően le tud kezelni, jelezni tudja a felhasználó felé a tényt, miszerint nem található megoldás a problémára.

6. fejezet

A probléma megvalósítása

6.1. A felhasznált módszerek

A problémák megoldásához az S-gráf keretrendszerben már kidolgozásra kerültek elméleti algoritmusok [3]. Szakdolgozati munkám célja ezen elméleti algoritmusok tanulmányozása, részleteinek kidolgozása, valamint az S-gráf megoldó keretrendszerbe történő integrálása, implementálása, valamint tesztelése.

6.1.1. Preventív ütemezés kötött batch mérettel

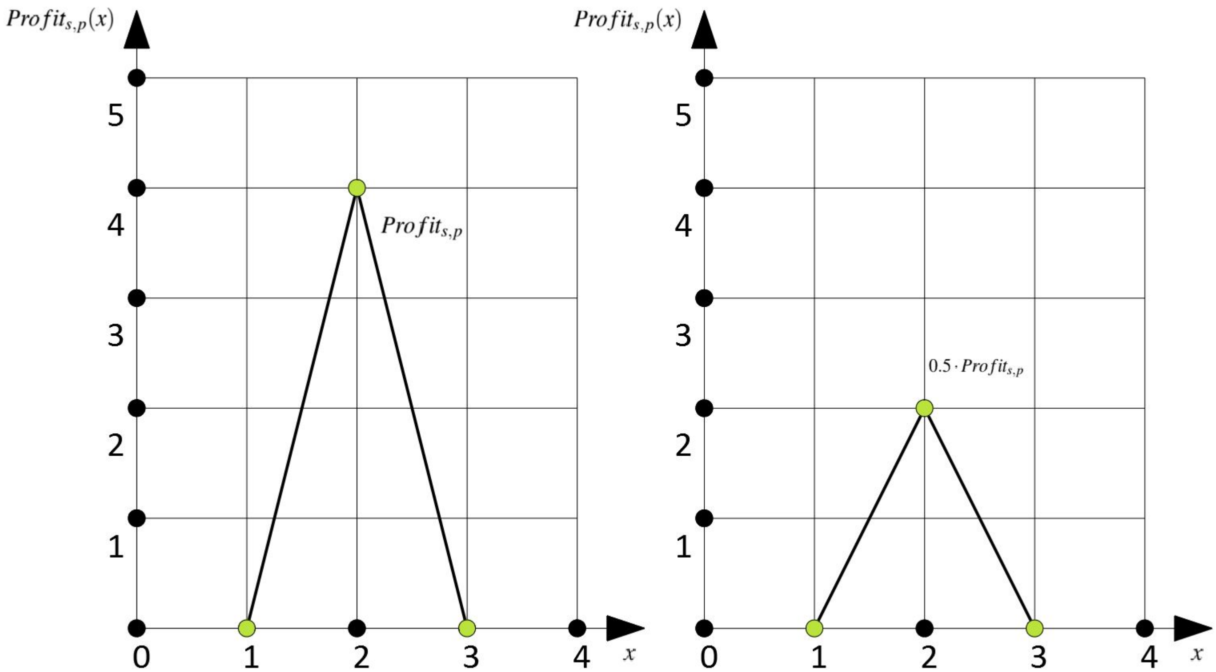
Ebben az esetben az egyetlen döntés, amit meg kell hozni, hogy az egyes termékekből hány darab batch-et gyártsunk, a várható profit a következőképpen számítható ki:

$$\sum_{p \in P} \left(\sum_{s \in S} prob_s \cdot profit_{s,p}(s_p \cdot b_p) \right)$$

Érdemes még bevezetni adott p termék x értékben vett várható profit értékére a következő jelölést:

$$ExpProfit_p(x) = \sum_{s \in S} prob_s \cdot profit_{s,p}(x)$$

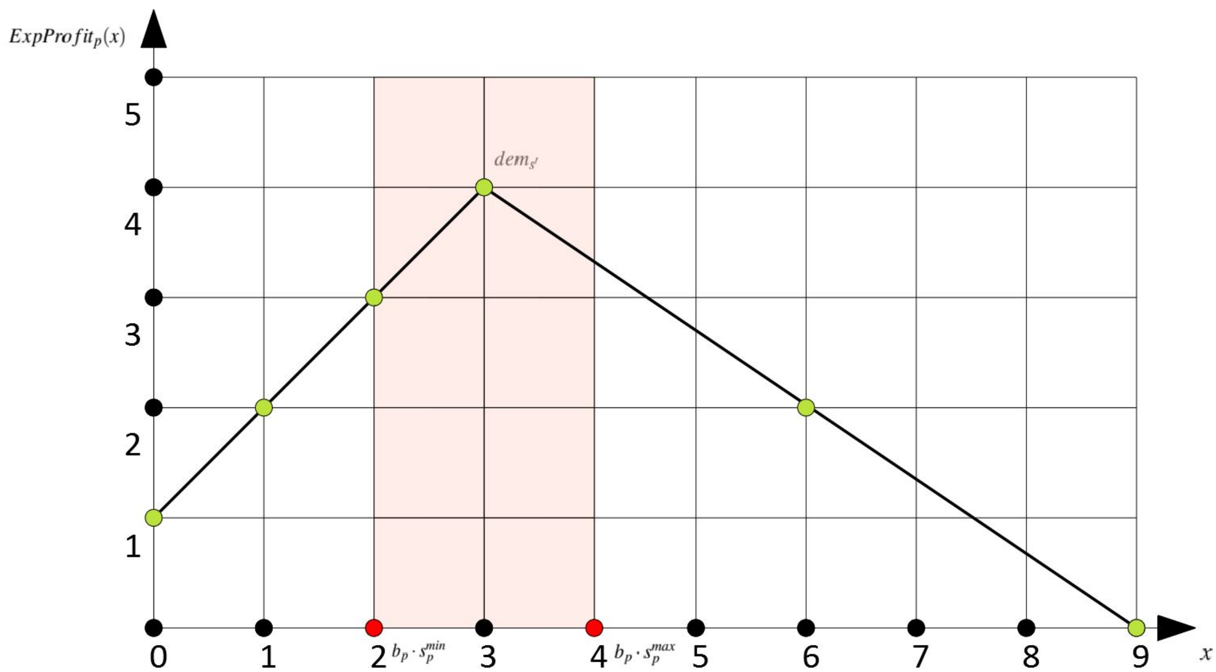
$ExpProfit_p(x)$ kiszámításához tehát nincs másra szükségünk, mint hogy az összes forgatókönyvre sorban felépítsünk az adott forgatókönyvre vonatkozó sztochasztikus paraméterekből a $profit_{s,p}$ függvényt, majd ezt a függvény beszorozzuk az aktuális $prob_s$ értékkel, amely lényegében a függvény "összenyomását" jelenti. Miután minden forgatókönyvre előállítottuk a 6.1 ábrához hasonlóan ezt az ”összenyomott” profit függvényt, ezen függvények összeadásával előáll az $ExpProfit_p$, ha ezt minden termékre megtesszük, az adott p termékek $ExpProfit_p(x)$ (ahol $x = s_p \cdot b_p$) értékének összegeként előáll a várható profit.



6.1. ábra. A profit függvény szorzásának szemléltetése

6.1.2. Preventív ütemezés változó batch mérettel

Az előző esettel ellentétben, változó batch méret esetén a batch darabszám nem határozza meg egyértelműen az adott termékből termelt mennyiséget. Ebben az esetben a batch méretről való döntés is a megoldó algoritmus feladata úgy, hogy p termék batch mérete s_p^{min} és s_p^{max} között legyen. Mivel ezt a döntést előre meg kell hozni, ezért minden forgatókönyvben azonos méretű lesz minden p termékhez tartozó batch. Ezután, már csak arról kell döntést hozni, hogy adott termékből mennyit gyártsunk, ez az x_p érték a következő intervallumból kerül kiválasztásra: $[s_p^{min} \cdot b_p, s_p^{max} \cdot b_p]$. Az $ExpProfit$ függvény maximális értékét az egyik keresleti értékben veszi



6.2. ábra. Az optimális x_p érték kiválasztásának szemléltetése

fel, legyen ez $dem_{s'}$. Az optimális x_p érték kiválasztása a következőképpen tehető meg:

$$x_p(b_p) = \begin{cases} b_p \cdot s_p^{max} & \text{ha } b_p \cdot s_p^{max} < dem_{s'} \\ dem_{s'} & \text{ha } b_p \cdot s_p^{min} \leq dem_{s'} \leq b_p \cdot s_p^{max} \\ b_p \cdot s_p^{min} & \text{ha } b_p \cdot s_p^{min} > dem_{s'} \end{cases}$$

A 6.2 ábra szemlélteti a fentieket. Látható, hogy ez esetben a $dem_{s'}$ érték beleesik a $[s_p^{min} \cdot b_p, s_p^{max} \cdot b_p]$ tartományba, ezért itt $x_p = 3$ lenne az optimális választás.

6.1.3. Két lépcsős ütemezés (two stage)

Ebben az esetben p termék gyártandó mennyiségét illető döntés egy bizonytalan esemény bekövetkezése után is meghozható, például, ha egy forgatókönyv már bekövetkezett. Éppen ezért a termék mennyisége az adott forgatókönyvtől függ, legyen ez: $x_{s,p}$. E mennyiség kiválasztása a következőképpen zajlik:

$$x_{s,p}(b_p) = \begin{cases} b_p \cdot s_p^{max} & \text{ha } b_p \cdot s_p^{max} < dem_s \\ dem_s & \text{ha } b_p \cdot s_p^{min} \leq dem_s \leq b_p \cdot s_p^{max} \\ b_p \cdot s_p^{min} & \text{ha } b_p \cdot s_p^{min} > dem_s \end{cases}$$

Látható, hogy a képlet hasonló a 6.1.2 pontban bemutatott képlethez, azonban míg ott az $ExpProfit$ függvényből kerül kiválasztásra az optimális x_p mennyiség (azaz, minden forgatókönyv esetén ez az érték ugyan annyi lesz), addig a két lépcsős ütemezés esetén minden egyes forgatókönyv $Profit$ függvényéből egyenként kerül kiválasztásra az optimális mennyiség. Ezzel megoldható az, hogy egy bizonyos forgatókönyv bekövetkezése után annak elvárásaihoz igazítsuk a termelt batch-ek méretét, jobb várható profitot elérve ezzel a legtöbb esetben. A várható profit két lépcsős ütemezés esetén a következőképpen számítható ki:

$$\sum_{p \in P} \left(\sum_{s \in S} (prob_s \cdot Profit(x_{s,p}(b_p)) \right)$$

6.1.4. Következtetés

Az előzőekben bemutatott módszerek ismerete arra enged következtetni, hogy a probléma megoldásához elengedhetetlen az S-gráf keretrendszerben egy olyan osztály definiálása, amely képes folytonos, szakaszos, lineáris függvények modellezésére, tárolására, azokon történő műveletek végrehajtására. Ezen osztály részletes leírása a 6.2 pontban olvasható.

6.2. A PiecewiseLinearFunction osztály

Ahogy az már korábban, a 6.1.4 pontban említésre került, a probléma implementációjához elengedhetetlen egy olyan osztály definiálása, amely kezelni képes folytonos, szakaszos, lineáris függvényeket. Erre hivatott az általam megalkotott **PiecewiseLinearFunction** osztály, amelynek forráskódja **piecewiselinearfunction.h**, illetve **piecewiselinearfunction.cpp** fájlokban található a solver **src\base** mappájában. Az általunk használt függvények tárolásához elegendő, ha kezdetben három pont koordinátái adottak, hiszen ezek elhelyezkedéséből a többi pont koordinátái később, ha valamilyen okból kifolyólag ez szükségessé válik, könnyen kiszámíthatóak, hiszen folytonos, lineáris függvényekről beszélünk. Ez a három pont a $profit_{s,p}$ függvények esetében nem más, mint:

- $Profit_{s,p}(x)$, ahol $x = dem_{s,p} - 1$
- $Profit_{s,p}(x)$, ahol $x = dem_{s,p}$
- $Profit_{s,p}(x)$, ahol $x = dem_{s,p} + 1$

E három pont koordinátái minden esetben kiszámíthatók, minden forgatókönyv-termék párosra már az input fájl beolvasását követően, hiszen minden sztochasztikus paraméter adott ehhez a fájlban. Az ehhez szükséges képlet a 6.1.2 pont szerint:

$$Profit_{s,p}(x) = \begin{cases} price_{s,p} \cdot x - (dem_{s,p} - x) \cdot uc_{s,p} & \text{ha } x < dem_{s,p} \\ price_{s,p} \cdot dem_{s,p} - (x - dem_{s,p}) \cdot oc_{s,p} & \text{egyébként} \end{cases}$$

Miután ennek a három pontnak a pontos koordinátái beazonosításra kerültek, már csak annyit kell tenni, hogy kiszámítunk két arány számot, amelyek a függvény kezdeti-, és vég meredekségét hivatottak letárolni. Ezen paraméterek ismeretében később a függvény bármely x pontjában felvett értéke számítható lesz. Ezek alapján a **PiecewiseLinearFunction** osztály adattagjai a következőek:

```
class PiecewiseLinearFunction{
private:
    std::vector <Coordinate> coordinates;
    double angle_begin;
    double angle_end;
```

6.3. ábra. A *PiecewiseLinearFunction* osztály adattagjai

A 6.3 ábrán látható **Coordinate** osztály a függvények pontjainak x és y koordinátáinak egyszerű tárolására, lekérdezésére, és összehasonlítására szolgál a megvalósított *setter*, *getter* és felültöltött egyenlőség operátorral.

Ahhoz, hogy a **PiecewiseLinearFunction** osztállyal a matematikai modellek minden szükséges művelete elvégezhető legyen, a következő funkcionalitást kell megvalósítani az osztálynak:

- Koordináta pár hozzáadása
- Koordináta párok rendezése x szerint növekvő sorrendbe
- Kezdeti-, és vég meredekség kiszámítása
- A függvény skalár értékkel való szorzása
- A függvény x helyen vett értékének lekérdezése
- Két függvény összeadása
- A függvény horizontális nyújtása (a 6.5 pontban tárgyalt esetekhez)
- A függvény maximális értékéhez tartozó koordináták lekérdezése

6.2.1. Koordináta pár hozzáadása

Egy új koordináta pár hozzáadását végző metódus az **addCoordinate(const Coordinate& c)**. Egy új koordináta pár hozzáadása esetén először is meg kell győződnünk, hogy adott koordináta pár szerepel-e már a koordináták tárolására szolgáló vectorban, hiszen ha már szerepel, nem adhatjuk hozzá újra. Ha nem szerepel még ilyen koordináta pár, hozzáadjuk, majd rendezzük a vectort x szerint növekvő sorrendbe. Ha ez megtörtént, meg kell vizsgálni, hogy a kezdeti-, és vég meredekségeket frissíteni kell-e. Abban az esetben, ha a most hozzáadott koordináta pár szerepel a rendezett vector első, vagy második helyén, a kezdeti meredekséget frissíteni kell. Hasonlóan, ha az új koordináta pár a vector utolsó, vagy utolsó előtti eleme, a vég meredekség frissítésre szorul. Mivel kezdetben akár mindkét feltétel igaz lehet, ezért ezek teljesülését két külön *bool* változóban kell tárolni. Miután megállapításra került, hogy melyik meredekségeket kell frissíteni, meghívásra kerül az ezeket kiszámító függvény.

6.2.2. Kezdeti-, és vég meredekség kiszámítása

A kezdeti-, és vég meredekség kiszámítására és frissítésére hivatott metódus a **calculateAngle(bool begin,bool end)**, melynek két paramétere a 6.2.1 pontban említett két *bool* változó, amelyek megadják, hogy kell e frissíteni adott meredekségeket. A meredekségek kiszámítása roppant egyszerű a koordináták ismeretében:

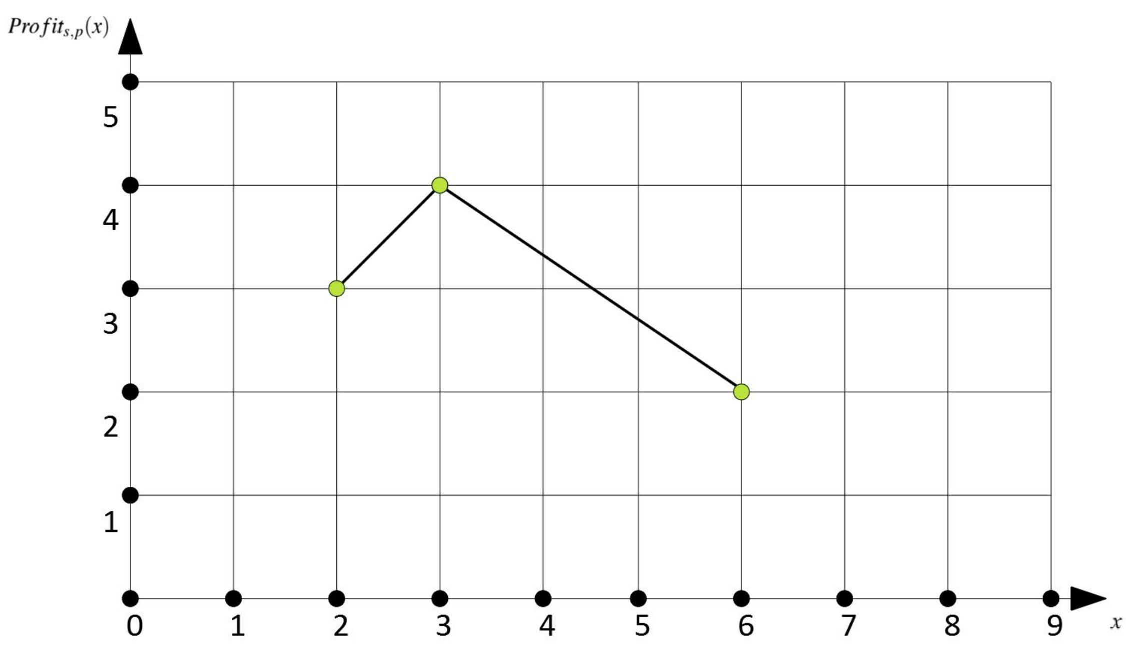
$$Angle_{begin} = (y_{second} - y_{first})/(x_{second} - x_{first})$$

$$Angle_{end} = (y_{penultimate} - y_{last})/(x_{last} - x_{penultimate})$$

A 6.4 ábrán látható három pont koordinátái alapján a meredekségek például a következőképpen alakulnak:

$$Angle_{begin} = (4-3)/(3-2) \quad \text{, azaz } Angle_{begin} = 1$$

$$Angle_{end} = (4-2)/(6-3) \quad \text{, azaz } Angle_{end} = 2/3$$



6.4. ábra. Példa a meredekségek kiszámítására

Ha egyik meredekség sem szorul frissítésre, a metódus semmiféle változtatást nem tesz.

6.2.3. A függvény skalár értékkel való szorzása

A függvény skalárral való szorzását a szorzás operátort felültöltő metódus végzi. Ahhoz, hogy megkapjuk a függvény skalárral való szorzatát, csupán be kell szorozni a vectorban tárolt összes pont y koordinátáját, valamint a kezdeti-, és vég meredekséget a paraméterként kapott s-el. A függvény szorzását szemléltető példa a 6.1 ábrán látható.

6.2.4. A függvény x helyen vett értékének lekérdezése

A függvény x helyen vett értékének lekérdezésére a ( ) operátort felültöltő metódus hivatott. A metódus először is megnézi, hogy a vectorban tárolt koordináta párok között található-e olyan, amelynek x koordinátája egyezik a paraméterként kapott x-el. Ha talál ilyet, egyszerűen visszaadja a megfelelő koordináta párost tartalmazó **Coordinate** objektumot. Ha nem található ilyen pont, akkor annak ki kell számítani a koordinátáit, és hozzá kell adni a vectorhoz, majd csak ezután lehet visszaadni a keresett **Coordinate** objektumot. Az x értékhez tartozó y érték kiszámítása a keresett x értéke és a vectorban tárolt pontok alapján háromféleképpen történhet:

- Ha a keresett x értéke kisebb mint a vectorban tárolt első pont x koordinátájának értéke, akkor: $y = y_{First} - (x_{First} - x) \cdot Angle_{begin}$,ahol x_{First} és y_{First} a vectorban tárolt első pont koordinátái.

- Ha a keresett x érték két a vectorban tárolt pont x koordinátájának értéke közé esik, akkor: $y = \left((x - x_{LastSmaller}) \cdot ((y_{FirstBigger} - y_{LastSmaller})/(x_{FirstBigger} - x_{LastSmaller})) \right) + y_{LastSmaller}$,ahol $x_{LastSmaller}$ és $y_{LastSmaller}$ a keresett x értéket megelőző pont koordinátái, míg $x_{FirstBigger}$ és $y_{FirstBigger}$ a keresett x értéket követő pont koordinátái.

- Ha a keresett x értéke nagyobb, mint a vectorban tárolt utolsó pont x koordinátájának értéke, akkor: $y = y_{Last} - (x - x_{Last}) \cdot Angle_{end}$,ahol x_{Last} és y_{Last} a vectorban tárolt utolsó pont koordinátái.

6.2.5. Két függvény összeadása

Két függvény összeadását az összeadás operátort felültöltő metódus végzi, melynek visszatérési értéke az összegként kapott függvényt tároló új **PiecewiseLinearFunction** objektum. Mivel a függvények tárolásához elegendő három pont tárolása, ezért gyakran előfordul olyan eset, hogy a két összeadni kívánt **PiecewiseLinearFunction** objektum nem tartalmazza a szükséges koordinátákat, ezért a hiányzó koordináta párokat először hozzá kell adni, ezt azonban megkönnyíti a 6.2.4 pontban bemutatott metódus, hiszen elég, ha lekérdezzük az aktuális x értékét mindkét függvény esetén, és ha az valamelyiknél nem található, automatikusan hozzá lesz adva annak pontjaihoz. Ennek következtében a két függvény összeadása már jóval egyszerűbb feladat, egyetlen *for* ciklussal megtehető.

```
PiecewiseLinearFunction PiecewiseLinearFunction::operator + (PiecewiseLinearFunction& b) {
    PiecewiseLinearFunction c = PiecewiseLinearFunction();
    std::vector<Coordinate> coordinates_b=b.getCoordinates();
    for(int i=0;i<coordinates_b.size();i++){
        this->operator()(coordinates_b.at(i).getX());
    }
    std::vector<Coordinate> coordinates_a=this->getCoordinates();
    for(int i=0;i<coordinates_a.size();i++){
        Coordinate current = Coordinate();
        current.setX(coordinates_a.at(i).getX());
        current.setY(coordinates_a.at(i).getY()+b(current.getX()).getY());
        c.addCoordinate(current);
    }
    return c;
}
```

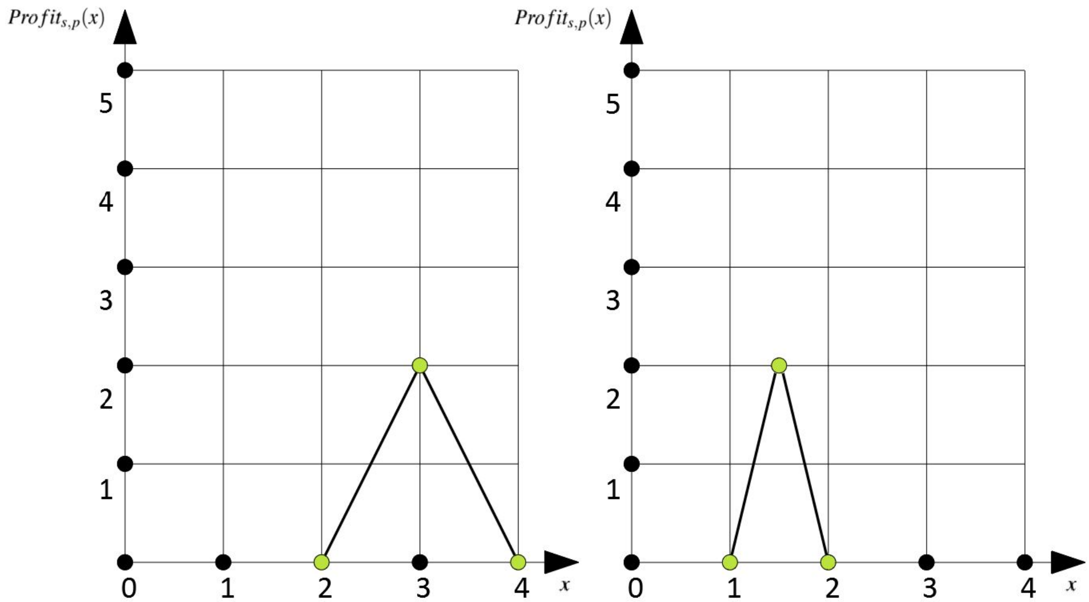
6.5. ábra. Az összeadást végző metódus

6.2.6. A függvény horizontális nyújtása

A függvény a 6.5 pontban használt, a 6.6 ábrán bemutatott horizontális nyújtását (illetve összenyomását, s értéktől függően) a **stretchHorizontally(double s)** metódus végzi. A metódus egyszerűen egy *for* ciklus segítségével a függvény összes pontjának x koordinátáját, valamint a kezdeti-, és vég meredekséget beszorozza a paraméterként kapott s értékkel, illetve a meredekségek esetén annak reciprokával.

6.2.7. A függvény maximális értékéhez tartozó koordináták lekérdezése

A függvény maximumának lekérdezését a **getMaximum()** metódus végzi, mely egy egyszerű maximum keresést valósít meg y koordinátára nézve. A metódusnak a 6.1.2 pontban bemu-



6.6. ábra. Példa a függvény horizontális nyújtására $s = 0.5$ értékkel

tatott változó batch méretű esetben van nagy jelentősége, ezt használjuk ugyanis $dem_{s'}$ meghatározására.

6.3. Az új paraméterek implementációja

Ahhoz, hogy az determinisztikus throughput maximalizáló használható legyen a 4.2 pontban bemutatott új sztochasztikus paraméterekkel, fel kell készíteni a megfelelő osztályokat ezen paraméterek kezelésére, be kell olvasni először is ezeket a paramétereket egy input fájlból, majd valamilyen formában le is kell őket tárolni, hogy később a 6.1 pontban bemutatott műveletek végrehajthatóak legyenek a várható profit kiszámítására.

6.3.1. Új kapcsoló definiálása

Mivel a sztochasztikus throughput maximalizáló működhet preventív, illetve két lépcsős módon, ezért szükségessé vált egy új parancssori kapcsoló bevezetése:

`--stages [single/twostage]` : specifies the number of stages in case of stochastic throughput maximization, variable batch size input needed for Two-stage to work (default: single)

6.7. ábra. Az új kapcsoló leírása a readme fájlban

Abban az esetben, ha nem adjuk meg a kapcsoló értékét, vagy azt single-re állítjuk, preventív módban fog futni az ütemező, ha twostage-t állítunk be, két lépcsős ütemezés fog lefutni, feltéve, hogy a bemeneti fájlban változó batch méretű adatokat adtunk meg.

6.3.2. Új input fájl definiálása

Az általam definiált új input fájl a **stochastic.ods** a solver **input** mappájában található. A fájl lényegében a **multipurpose.ods** kibővítése a sztochasztikus paraméterekkel, éppen ezért az utóbbi **equipment**, és **proctime** tábláit változtatás nélkül tartalmazza, hiszen ezek tartalmazzák a gépekre, illetve a recept gráfra vonatkozó determinisztikus paramétereket, amelyeket az új esetekben is fel kell használnia a megoldó algoritmusnak. Ezzel szemben a **product** tábla a sztochasztikus esetekben nem fogja megállni a helyét, hiszen a batch méretekre vonatkozó adatok hiányoznak belőle, ezeket hozzá kell adni a product táblához. Ezenkívül a forgatókönyvek adatait is tárolnunk kell, ezért bevezetésre kerültek a **scenario**, és a **scenario_data** táblázatok, melyek a 4.2 pontban leírt, forgatókönyvekre vonatkozó sztochasztikus adatokat tartalmazzák. Az általam definiált új formátumokra példa input fájlok megtekinthetőek a **B** függelékben.

6.3.3. A fájl beolvasása, beolvasott paraméterek tárolása

Az input fájl beolvasását a **RelationalProblemReader** osztály végzi, melynek feladata a fájlban található paraméterek alapján a recept gráf felépítése, annak visszaadása a **MainSolver** osztály számára. A **RelationalProblemReader** először is meggyőződik az input fájl típusáról, majd az alapján sorban beolvassa a megfelelő mezőket az input fájlból. Éppen azért szükséges egy metódus definiálása, mely képes a sztochasztikus típusú input fájlt megkülönböztetni a többi fajtától. Ez a metódus az **IsStochastic()**, illetve a 6.5 pontban használt extended input fájl esetén az **IsExtendedStochastic()**. Miután meggyőződtünk a fájl sztochasztikus mivoltáról, meghívásra kerül a **ReadStochastic()** metódus, amely beolvassa, majd eltárolja az **equipment**,

product, **scenario**, **scenario_data**, **proctime** táblák paramétereit a recept gráfot reprezentáló **SGraph** objektum **Recipe** objektumában. Jól látszik, hogy az új sztochasztikus paraméterek tárolásához, a **Recipe**, és az **SGraph** osztályokban szükséges létrehozni a megfelelő adattagokat, azok eléréséhez szükséges metódusokat. A termékek batch méretére vonatkozó új paraméterek kezelésére a **Product** osztály kiegészítésre került a batch_size, batch_size_min, batch_size_max adattagokkal, valamint ide kerülnek letárolásra a **scenario_data** tábla adatai is, az erre definiált **ScenarioDataEntry** objektumokból álló vectorba. A **ScenarioDataEntry** osztály tartalmazza a **scenario_data** táblázatban egy sorában található adatokat, például az adott forgatókönyv azonosítóját, a termék iránti keresletet az adott forgatókönyvben, valamint itt tároljuk le a sztochasztikus paraméterek alapján a 6.2 pontban bemutatott módon létrehozott **PiecewiseLinearFunction** objektumot, amely a $Profit_{s,p}$ függvényt reprezentálja. A fent említett műveleteket, a szükséges objektumok létrehozását a **RelationalProblemReader** osztály **ParseScenarioData** metódusa végzi el. Ezenkívül bevezetésre került még három *boolean* változó a **Recipe** osztályba, amelyek flag-ként szolgálnak, hogy a throughput maximalizáló könnyen, csupán a **Recipe** objektum segítségével meg tudja különböztetni a kötött-, változó batch méretű, és a két lépcsős eseteket.

6.4. Szükséges változtatások az S-gráf keretrendszerben

A sztochasztikus paraméterek letárolása után nincs más hátra, mint felkészíteni a throughput maximalizálást végző **ThroughputSolver** osztályt ezek kezelésére. Azonban ahhoz, hogy egységesen használható legyen az osztály determinisztikus, illetve sztochasztikus esetben is, némi refaktorálásra van szükség, ugyanis jelen állapotában a **ThroughputSolver** osztályban találhatóak olyan megoldások, melyek megkövetelik, hogy a profit egyszerűen a $b_p \cdot price_p$ képlettel megkapható legyen, azonban ez a sztochasztikus esetben korántsem ilyen egyszerű.

6.4.1. A meglévő kód refaktorálása

A **Throughputsolver** osztály jelenlegi állapotában, determinisztikus esetben kétféleképpen számítja ki a profitot, egyrészt az **SGraph** osztály **GetRevenue()** metódusát használva, másrészt az egyes termékek adatait az **SGraph** osztály **Recipe** objektumában letárolt **Product** objektumok adatait közvetlenül lekérdezve, majd azokat a fent említett $b_p \cdot price_p$ képlettel kiszámítva, és összegezve. Habár determinisztikus esetben ezek a módszerek megállják a helyüket, ezek jelen állapotban nem túl elegánsak, hiszen a **GetRevenue()** metódus lényegében ugyan azt a működést éri el, mint az utóbbi közvetlen elérés, és összegzés. A sztochasztikus esetek bevezetésével ráadásul az adatok közvetlenül a **Product** objektumoktól való elérése nem lesz működő módszer, ugyanis ezekben az esetekben a termék paraméterei, ahogy azt már korábban tárgyaltuk, forgatókönyv függőek. Éppen ezért a sztochasztikus esetekben használt **GetRevenue()** metódus a **Recipe** osztályban kell helyet kapjon, hogy a termék és a forgatókönyv adatok elérése egyaránt lehetséges legyen a metódus számára. Mivel egy jól karbantartható, hibamentes kódban a fent említett redundanciákat érdemes kiküszöbölni, ezért célszerű lenne a determinisztikus-, és a sztochasztikus esetben használt **GetRevenue()** metódusok összevonása, mégpedig a **Recipe** osztályban, ennek az összevont metódusnak a részletes leírása a 6.4.2 pontban olvasható. Az összevont **GetRevenue()** metódus megalkotásának köszönhetően a **Throughputsolver** osztály immáron egységes módon kérdezheti le a várható profitot, a probléma milyenségéről való döntés, valamint a profit kiszámítása pedig a **Recipe** osztály feladata.

6.4.2. A *GetRevenue* metódus

A **GetRevenue()** metódus arra szolgál, hogy adott konfiguráció várható összbevételét kiszámítsa, majd visszaadja azt. A metódus először döntést hoz a probléma típusát illetően a recept objektumban tárolt *boolean* flag segítségével. Determinisztikus esetben a profit kiszámítása meglehetősen egyszerűen, a 6.8 ábrán látható módon megtehető.

```
double toRet = 0.;
uint prodNum = GetProductCount();
for(uint i = 0; i < prodNum; i++)
    toRet += GetProduct(i).GetBatches() * GetProduct(i).GetRevenue();
return toRet;
```

6.8. ábra. A *GetRevenue() metódus lefutása determinisztikus esetben*

Sztochasztikus esetben is a fenti ábrához hasonló a ciklus, azonban szükség van egy metódusra, amely x számú p termék profitját képes kiszámítani, ez a függvény a **GetProductRe-**

venue(uint product_id, uint batches). A metódus megvalósítása közben kiderült továbbá az is, hogy ezenkívül további változtatásokra is szükség van a solver bizonyos karakterisztikái miatt. A probléma akkor jelentkezik, ha egy adott termékből (legyen ennek neve a példa kedvéért ”*A*”) többet termelünk egy batch-nél, ebben az esetben nem az általam várt módon történik a konfiguráció adatainak tárolása. Ideális számomra az lenne, ha például a konfigurációban két darab ”*A*” batch termelése esetén egy darab ”*A*” termék jelenne meg, és ennek a batch száma 2 lenne. Azonban nem ez történik. A konfigurációban egy ”*A*” és egy ”*A_2*” nevű termék van jelen egyaránt 1 batch számmal. Erre az ütemterv elkészítéséhez van szükség, hogy egyértelműen beazonosíthatóak legyenek az egyes termékek, illetve azok részfolyamatai. Ez a fajta működés nyilván determinisztikus esetben nem okoz gondot, hiszen ott ekvivalens az, ha két ugyan olyan termékből gyártunk egy-egy darabot, vagy egy termékből kettőt, hiszen a profit értékek nem függenek egymástól, azonban sztochasztikus esetben az alul-, és túl termelési költségek miatt a két eset nem ekvivalens. Éppen ezért ezt a működést valamilyen formában orvosolni kell. Erre a problémára nyújt megoldást a **Recipe** osztály **ReduceToBase()** metódusa. A metódus működéséhez elengedhetetlen, hogy az inputfájlban specifikált termékek neveit elmentsük a **Recipe** osztály egy vectorába, a fájl beolvasásakor. Ezen *vector* terméknevei reprezentálják az úgynevezett ”base product”-okat, azaz a kezdeti termékeket. Ennek a *vectornak* a birtokában a **ReduceToBase()** metódus képes az aktuális konfiguráció termékeinek nevét összevetni a kezdeti termékek nevekkel, visszavezetni a konfiguráció termékeit a kezdeti termékekre. A metódus egy *map*-el tér vissza, amely tartalmazza a kezdeti termékeknek megfeleltetett termékneveket és a hozzájuk tartozó mennyiségeket. Ha például a konfigurációban a fent említett ”*A*” és ”*A_2*” termékek szerepelnek egy-egy batch-el, a metódus által visszaadott map tartalma ”*A*” termék lesz kettő batch-el, ezzel kiküszöbölve az említett problémát.

A **GetRevenue()** metódusnak azonban egy másik, a sztochasztikus esetekkel kapcsolatos problémát is orvosolnia kell. Ez a probléma a úgynevezett ”axial revenue”, azaz a tengelyeken számított várható profit értékével kapcsolatos. Axial revenue-ról akkor beszélünk, ha az adott konfigurációban csupán egy fajta terméket gyártunk, a többi termékből (ha léteznek) ez esetben 0 darabot termelünk. Ez a sztochasztikus esetekben azért okoz problémát, mert a nem gyártott termékek esetleges alul termelési költségeit le kell vonni az összes profitból, így tehát hiába gyártunk csak egy terméket, a többi termék paraméteri is befolyásolják a várható profitot. Az alul termelésből eredendő költségek számítása alapvetően nem okozna problémát, hiszen csak ki kellene számolni adott termékek $x = 0$ helyen vett $ExpProfit_p(x)$ értékét, és összegezni a kapott értékeket. Mivel azonban ebben az esetben csak egy fajta terméket termelünk, ezért a konfigurációban csak a termelt termék adatai találhatóak meg, ezért a többi termék $ExpProfit_p(x)$ értékének számítása jelen helyzetben lehetetlen. Erre azonban megoldást nyújt, ha a korábban említett módon, az inputfájl beolvasását követően a **RelationalProblemReader** osztályban nem csak a kezdeti termékek neveit tároljuk el, hanem azok $x = 0$ helyen vett $ExpProfit_p(x)$ értékeit is (felhasználva a 6.4.3 pontban bemutatott **GetProductRevenue metódust**). Ezen értékek tudatában a nem termelt termékekből származó esetleges veszteség immáron levonható az várható profit értékéből, az axial revenue hibátlanul megkapható.

Ezen változtatások segítségével a **GetRevenue()** függvény ezentúl egységesen használható a **ThroughputSolver** osztály számára egy adott konfiguráció összprofitjának kiszámítására, függetlenül a probléma típusától.

6.4.3. A *GetProductRevenue* metódus

A **GetProductRevenue** metódus hivatott p termék x helyen vett várható profit értékének, azaz $ExpProfit_p(x)$ kiszámítására. Mivel a metódusnak kezelnie kell a különböző eseteket, ezért szerkezete a következőképpen alakul:

procedure GETPRODUCTREVENUE

  if not *stochastic* **then return** *non stochastic profit*

  if not *variable batch size* **then return** *fixed batch size profit*

  if *variable batch size* **and not** *two stage* **then return** *variable batch size profit*

  if *variable batch size* **and** *two stage* **then return** *two stage profit*

return 0

- Determinisztikus esetben a számítás meglehetősen egyszerűen, a $price_p \cdot b_p$ képlettel elvégezhető.
- Kötött batch méretű sztochasztikus esetben a várható profit számítása a 6.1.1 pontban bevezetett módon, az $ExpProfit(x)$ képlet segítségével lehetséges.

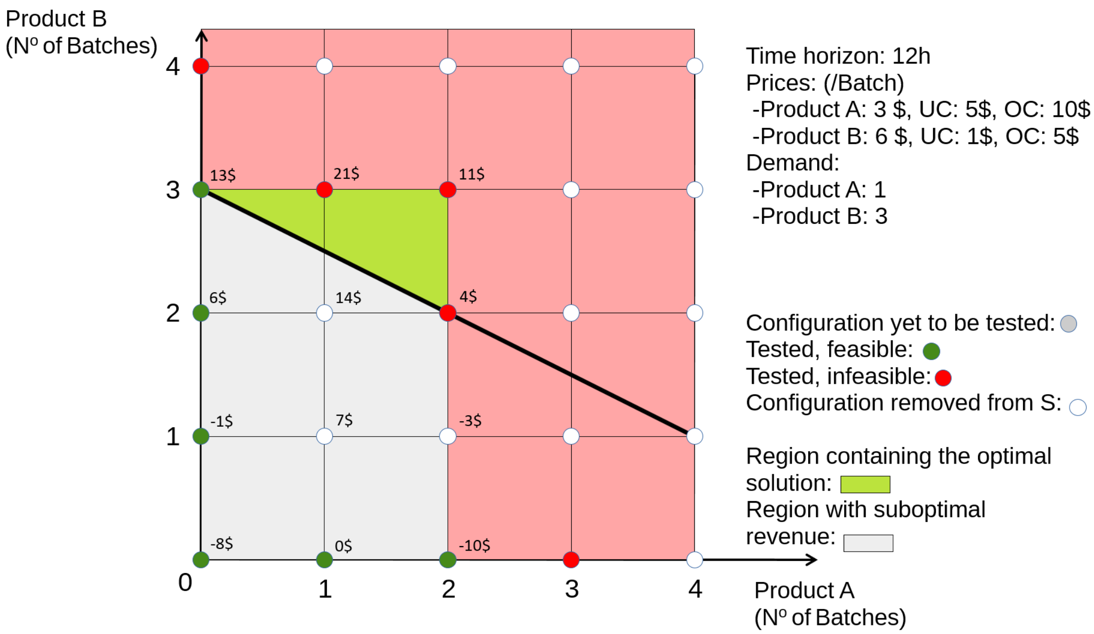
- Változó batch méretű sztochasztikus esetben a várható profit számítása a 6.1.2 pontban leírtak alapján $x_p(b_p)$ meghatározásával, majd $ExpProfit(x_p)$ kiszámításával tehető meg.
- Két lépcsős ütemezés esetén a várható profit számítása a 6.1.3 pontban ismertetett módon $x_{s,p}(b_p)$ meghatározása után, a következő képlettel lehetséges:

$$\sum_{s \in S} (prob_s \cdot Profit(x_{s,p}(b_p))$$

Ezen módszerek implementációját tartalmazza tehát a **GetProductRevenue** metódus, amely segítségével ezentúl adott termék profitja, függetlenül a probléma típusától egységesen megkapható.

6.4.4. A *Revenue Line* figyelmen kívül hagyása sztochasztikus esetben

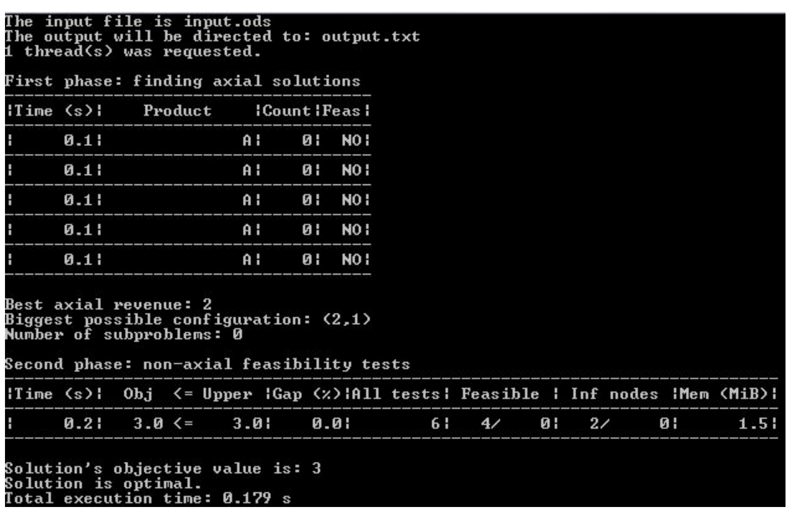
Ahogyan az már a 3.1. pontban említésre került, a determinisztikus throughput maximalizálóban használatos gyorsítási stratégia az un. Revenue Line, amely lényegében alsó bound-ként szolgál az egyes konfigurációk profitértékeinek összehasonlításához. A 3.4. ábrán jól látható, hogy ez a stratégia determinisztikus esetben jó eredménnyel használható, ideális esetben töredékére csökkentheti az ellenőrizendő konfigurációk számát. Sztochasztikus esetben azonban el kell tekintetnünk a Revenue Line használatától, belátható ugyanis, hogy az esetleges alul-, és túl termelési költségek miatt nem minden esetben állja meg a helyét az az állítás miszerint egy konfiguráció várható profitértéke magasabb lesz egy másikénál azért, mert adott konfigurációban több batch-et termelünk adott termékekből. Éppen ellenkezőleg. Akár olyan eset is fenn állhat, melyben a nagyobb konfiguráció várható profit értéke elmarad a kisebb konfiguráció profitjától. A fenti állítás szemléltetésére szolgál a 6.9. ábra. Jól látszik, hogy ha jelen paraméterek mellett a Revenue Line-t felhasználva futtatnánk a sztochasztikus throughput maximalizálót, szuboptimális eredményt kapnánk, hiszen a vonal behúzása miatt a valódi optimális eredmény kiszámításra sem kerülne. Szerencsére a Revenue Line figyelmen kívül hagyása sztochasztikus esetben egyetlen *if* feltétellel megtehető **ThroughputSolver** osztályban a megfelelő eredmény elérése érdekében.



6.9. ábra. A *Revenue Line* helytelenségének szemléltetése sztochasztikus esetben

6.4.5. A *ThroughputUI* osztály kiegészítése

A **ThroughputUI** osztály feladata a **ThroughputSolver** osztály által kiszámított eredmények megjelenítése a felhasználó számára. Determinisztikus esetben a Throughput maximalizáló egy példa lefutását a 6.10 ábra szemlélteti.

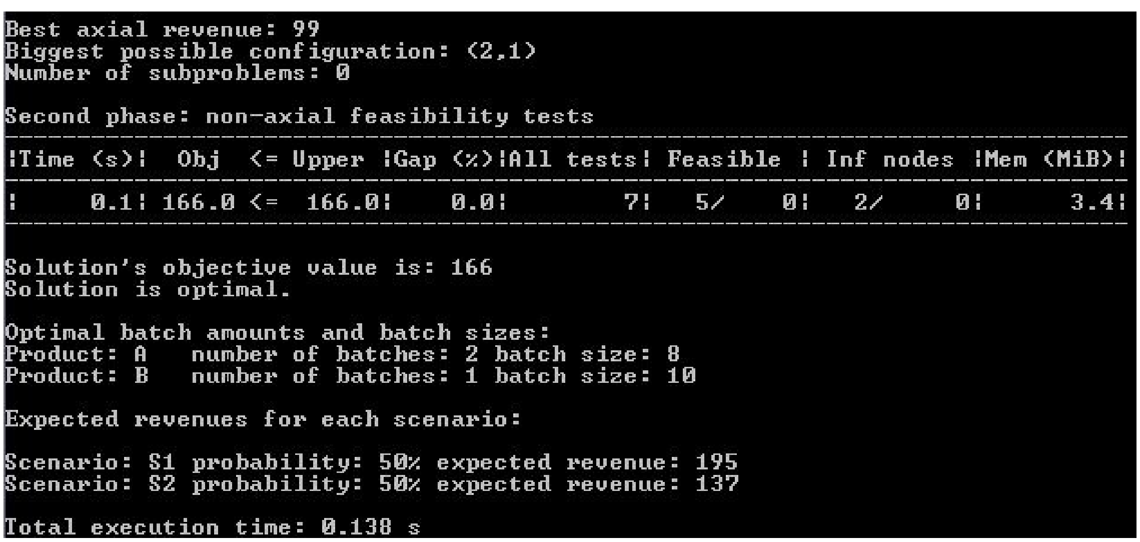


6.10. ábra. Példa a *ThroughputUI* szerkezetére determinisztikus esetben

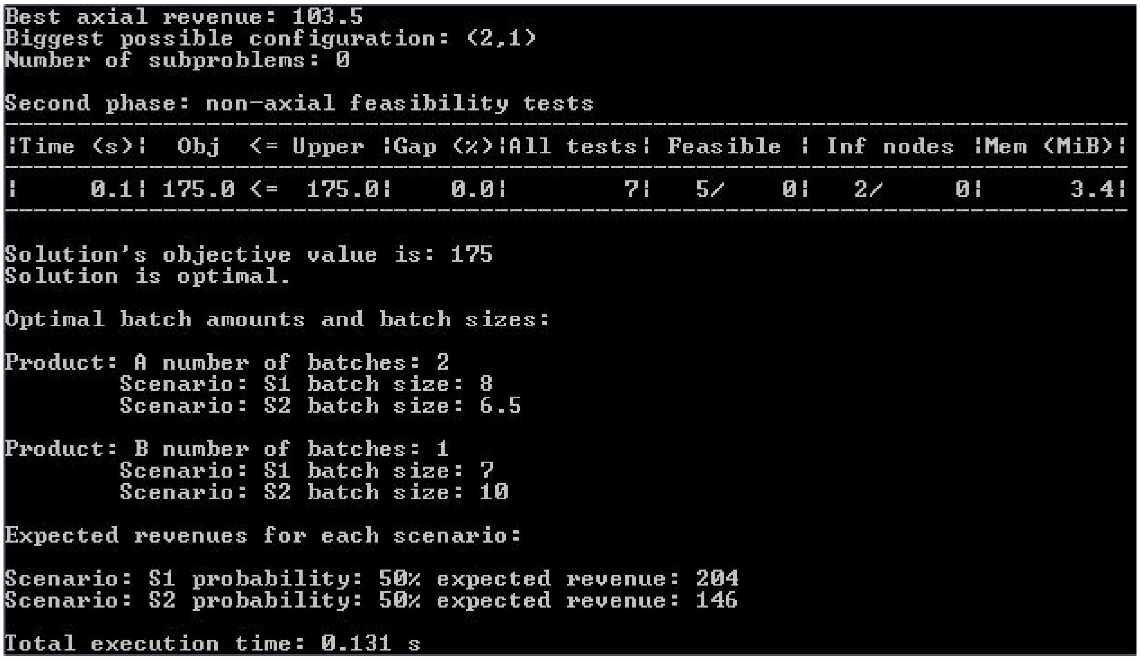
Jól látszik, hogy a **ThroughputUI** osztály is kiegészítésre szorul a sztochasztikus esetek kezeléséhez a következőkkel:

- Optimális batch darabszámok és méretek minden termékre (kötött batch méret és változó batch méret esetén)
- Optimális batch darabszámok minden termékre, batch méretek minden termék - forgatókönyv párosra (két lépcsős ütemezés esetén)
- Forgatókönyvek, és a hozzájuk tartozó valószínűségek
- A várható összprofit értéke forgatókönyvenként

Ezen adatok tárolását a **ThroughputSolver** osztály általam létrehozott metódusa a **SaveStochasticStatistics** végzi el felhasználva a **ThroughputSolver** osztály **Statistics** típusú objektumát, melynek feladata a futás közben a **ThroughputUI** osztály számára szükséges statisztikai adatok tárolása. A **SaveStochasticStatistics** metódus minden alkalommal lefut, mikor egy új konfiguráció kerül hozzáadásra a **NewSolution** metódus által. A sztochasztikus esetek lefutását a 6.11, és a 6.12 ábra szemlélteti.



6.11. ábra. Példa a *ThroughputUI* szerkezetére kötött, és változó batch méret esetén



```
Best axial revenue: 103.5
Biggest possible configuration: (2,1)
Number of subproblems: 0

Second phase: non-axial feasibility tests
-------------------------------------------------------------------------------
|Time (s)|  Obj  <= Upper |Gap (%)|All tests| Feasible  | Inf nodes  |Mem (MiB)|
-------------------------------------------------------------------------------
|     0.1| 175.0 <=  175.0|    0.0|        7|   5/     0|   2/      0|      3.4|
-------------------------------------------------------------------------------

Solution's objective value is: 175
Solution is optimal.

Optimal batch amounts and batch sizes:

Product: A number of batches: 2
        Scenario: S1 batch size: 8
        Scenario: S2 batch size: 6.5

Product: B number of batches: 1
        Scenario: S1 batch size: 7
        Scenario: S2 batch size: 10

Expected revenues for each scenario:

Scenario: S1 probability: 50% expected revenue: 204
Scenario: S2 probability: 50% expected revenue: 146

Total execution time: 0.131 s
```

6.12. ábra. Példa a *ThroughputUI* szerkezetére két lépcsős ütemezés esetén

Mivel az S-gráf keretrendszer throughput maximalizálója alapvetően többszálas működésre lett tervezve, ezért a **ThroughputSolver** osztály által használt **Statistics** osztály általam bevezetett új adattagjait, illetve azok *getter*, *setter* metódusait fel kell készíteni a párhuzamos használatra. A C++ programnyelven történő párhuzamos programozás megértéséhez, a szükséges változtatások bevezetéséhez nagy segítségnek bizonyult Anthony A. Williams a témában íródott könyve. [9] A primitív adattagok esetében az **AtomicVariable<T>** osztályt használtam fel, melynek definíciója a solver **base** mappájában található **parallel.h** fájlban olvasható. Az osztály megvalósítja T típusú változó párhuzamos elérését, azon végzett műveletek biztonságos lekezelését **Lock** objektumok használatával. A bonyolultabb adatszerkezetek (például map-ek) párhuzamos elérését a **Statistics** osztály *getter*, *setter* metódusaiba implementált **Lock** objektumokkal oldottam meg. A **Statistics** osztály általam bevezetett adattagjai a 6.13 ábrán láthatóak.

```
AtomicVariable<double> max_revenue;
AtomicVariable<bool> stochastic;

map <string,map <string,double>> product_scenario_batchSize_map;
map <string,int> product_amount_map;
map <string,double> scenario_revenue_map;
map <string,double> scenario_probability_map;

mutable Lock product_scenario_batchSize_map_lock;
mutable Lock product_amount_map_lock;
mutable Lock scenario_revenue_map_lock;
mutable Lock scenario_probability_map_lock;
```

6.13. ábra. A *Statistics* osztály új adattagjai

Mivel a *map* típusú adattagok *getter* metódusai konstans metódusok, ezzel szemben a solverben definiált **Lock** osztály **Set**, és **Unset** metódusai nem konstansok, ezért elengedhetetlen volt ez esetben a **Lock** típusú objektumok *mutable* kulcsszóval történő definiálása a **Statistics** osztályban. Az általam létrehozott, **Lock** objektummal védett *getter*, és *setter* metódusokra példa a 6.14 ábrán látható.

```
const map <string,double>& GetScenario_probability_map() const{
    scenario_probability_map_lock.Set();
    return this->scenario_probability_map;
    scenario_probability_map_lock.Unset();
}

void SetScenario_probability_map(const map <string,double>& map){
    scenario_probability_map_lock.Set();
    this->scenario_probability_map=map;
    scenario_probability_map_lock.Unset();
}
```

6.14. ábra. Példa a *Lock* objektummal védett *getter*, *setter* metódusokra

6.5. Multiproduct receptek esete

Ebben a részben tárgyalt esetekben az eddigi feltételezés, miszerint a receptek és a termékek között 1-1 kapcsolat áll fenn, nem teljesül. Egy recept akár több különböző terméket is előállíthat, illetve egy bizonyos terméket akár több különböző recept is előállíthat. Az ilyen probléma megoldásához szükséges új fajta input fájlra példa, a **stochastic_extended.ods**, amely a B függelék B.3 ábráján látható. Az ábra alapján jól látható, hogy az előzőekben használt **stochastic.ods** fájlhoz képest csupán egy új tábla került hozzáadásra, valamint a **scenario_data** táblában történt változás. Ez az egy új hozzáadott tábla a **sub_product** nevet kapta, ez hivatott kifejezni a **product** táblában található receptek által előállított termékeket. [1] A **sub_product** tábla tartalmazza az adott receptek által előállított termékek neveit, illetve egy arányszámot, mely megadja, hogy a recept egy batch-ének gyártása során adott termék milyen arányban áll elő. A **scenario_data** táblában pedig immáron a sub product-okra vonatkozó adatok találhatóak a product-ok adatai helyett. A multiproduct receptekkel kapcsolatos probléma alapvetően két esetre osztható:

- Az első, egyszerűbb esetről akkor beszélünk, ha egyetlen egy terméket sem eredményez két, vagy több recept.
- A második, bonyolultabb eset akkor áll fenn, ha egy terméket egyszerre több recept is előállít.

Az első eset könnyen megoldható, csupán sorra kell venni az adott recept által gyártott termékek $profit_{s,p}$ függvényeit minden egyes forgatókönyvre, majd a 6.2.6 pontban bemutatott metódus segítségével a 6.6 ábrán látható módon nyújtani azokat a fent említett arányszámmal, mely kifejezi a recept egy batch-e és az előállított termék mennyisége közti arányt. Miután ezeket a nyújtott függvényeket megkaptuk minden termékre az adott forgatókönyvben, amelyet az adott recept gyárt, ezek összegeként előáll a recept $profit$ függvénye az adott forgatókönyvben. Ezen

[1] Mivel a solverben, valamint az eddigi input fájlokban (pl. a multipurpose.ods-ben) is ”product” néven hivatkozunk a receptekre, ezért a konzekvenciát megtartandó a receptek által előállított termékek ebben az esetben a ”sub product” nevet kapták. A későbbiekben ezen elnevezések egységesítése, refaktorálása a solverben jóval érhetőbb, egységesebb kódot eredményezne, azonban ez túlmutat ezen szakdolgozat témáján.

$profit$ függvények letárolásával a továbbiakban ez az eset is tekinthető az eredeti sztochasztikus esetnek, az eddig említett módszerek felhasználhatóak a megoldáshoz. Az egyszerűbb átláthatóság kedvéért azonban célszerű lenne bevezetni pár új jelölést:

R a receptek halmaza

P_r a termékek halmaza, amelyet $r \in R$ előállít

$s_{r,p}$ az $r \in R$ recept egy batch-e által maximálisan előállítható $p \in P$ termék mennyisége

Az új jelölések alapján $r \in R$ recept $profit$ függvénye az adott forgatókönyvben a következőképpen fejezhető ki:

$$profit_{s,r} = \sum_{p \in P_r} profit_{s,p} \cdot s_{r,p}$$

Az előzőeket hivatott megvalósítani a **Recipe** osztály általam létrehozott **CalculateSubProductRevenueSum(uint product_id)** metódusa, amelyet a **GetProductRevenue** metódus hív meg abban az esetben, ha érzékeli, hogy multiproduct eset áll fenn. A metódus lefutása után a probléma a továbbiakban a sztochasztikus alapesetekkel ekvivalens, a várható profit értéke a szokásos módon megkapható.

A második esetben, vagyis amikor fennáll az, hogy egy terméket egyszerre több recept is előállíthat, a probléma megoldása kifinomultabb módszereket igényel, hiszen a receptek várható profitja ebben az esetben nem lesz független egymástól. Az optimális batch darabszámok, illetve méretek kiszámítására jelen esetben célravezető egy LP modell felírása. [3] Az inputfájl definiálásával, illetve a solveren végzett munkámmal megteremtettem ezen eset megoldásának feltételeit is, azonban ennek implementációja túlmutat jelen dolgozat munkáján.

7. fejezet

Teszteredmények

Ebben a fejezetben a tesztelése menete, a tesztelés által elért eredmények kerülnek bemutatásra. Mivel esetemben egy meglévő szoftver rendszer továbbfejlesztéséről beszélhetünk, ezért az új funkciók letesztelésén kívül fontos az eddigi funkcionalitás újratesztelése is. Éppen ezért a tesztesetek alapvetően három részre bonthatóak:

- A determinisztikus throughput maximalizáló tesztelése
- A sztochasztikus alapesetek (1-1) tesztelése
- A sztochasztikus multiproduct esetek tesztelése

A teszteléshez használt input fájlok tartalma, nevezetesen a termékekre, illetve a forgatókönyvekre vonatkozó paraméterek random szám generátorral készültek adott tartományokon belül. A tesztkonfigurácó leírása:

- Windows 7 operációs rendszer
- Qt 5.11.2, Microsoft Visual C++ Compiler 15.0, Boost Libraries 1.68.0
- Intel i5 3570K 3,8 Ghz processzor
- 8 GB RAM

7.1. A determinisztikus throughput maximalizáló tesztelése

A determinisztikus throughput maximalizáló teszteléséhez használt teszt fájlok a CD melléklet **Tesztelés/Tesztfájlok/Determinisztikus** mappájában találhatóak, míg a teszteredményeket rögzítő **DeterministicTestResults.ods** fájl a CD melléklet **Tesztelés/Teszteredmények** mappájában található. A determinisztikus teszt fájlokban található termékek profit értékei véletlenszerűen választott számok, a többi adat pedig a **multipurpose.ods** fájlból került átmásolásra. A 7.1 ábrán látható egy példa egy determinisztikus teszt fájlra.

product		
name	no_stages	revenue
A	2	24,50
B	3	12,3

equipment	
name	number
	1
E1	1
E2	1
E3	1
E4	1

proctime			
pr_name	eq_name	stage	time
product.name	equipment.name>		infty
A	E1	1	5
A	E3	1	3
A	E4	2	5
B	E2	1	6
B	E1	2	4
B	E4	3	2

7.1. ábra. Példa egy determinisztikus teszt fájlra

Ezen fájlt felhasználva a determinisztikus throughput maximalizáló 15 óra időhorizont érték mellett 61.3 profit egységet adott eredményül a következő gyártott mennyiségek mellett: $2A+1B$. Jól látható, hogy az eredmény helyes, hiszen $2 \cdot 24.5 + 12,3 = 61.3$. Ezenfelül a solver master git branchén található verziót futtatva (melyhez a dolgozat írásakor még nem került hozzáadásra a munkám) szintén megegyező eredményt kapunk, mind a profit értékeket, mind a kimeneti ütemterveket figyelembe véve, valamint a futási időkben sem figyelhető meg számottevő változás. Ezen adatokat figyelembe véve kijelenthető tehát, hogy a determinisztikus throughput maximalizáló továbbra is hibátlanul működik.

7.2. A sztochasztikus alapesetek tesztelése (1-1)

A sztochasztikus throughput maximalizáló alapeseteinek teszteléséhez használt teszt fájlok a CD melléklet **Tesztelés/Tesztfájlok/Sztochasztikus** mappájában találhatóak, míg a teszteredményeket rögzítő **StochasticTestResults.ods** fájl a CD melléklet **Tesztelés/Teszteredmények** mappájában található. A sztochasztikus teszt fájlokban található, a termékekre, illetve a forgatókönyvekre vonatkozó paraméterek (batch méretek, kereslet, termék ára, stb.) random szám generátorral lettek előállítva. A 7.2 ábra szemlélteti a **StochasticTestResults.ods** fájl tartalmát.

#	File name	Number of recipes	Number of Scenarios	Runtime (sec)	Timehorizon	Fix	Variable	Two stage	Axial	Optimal
1	test001	2	2	0,129	15	Y	N	N	99	166
2	test002	2	2	0,129	15	N	Y	N	99	166
3	test002	2	2	0,131	15	N	N	Y	103,5	175
4	test003	2	2	0,13	15	Y	N	N	74,2	143,2
5	test004	2	2	0,129	15	N	Y	N	74,4	143,4
6	test004	2	2	0,134	15	N	N	Y	76,6	147,4
7	test005	2	2	0,131	15	Y	N	N	50,6	122
8	test006	2	2	0,14	15	N	Y	N	64,2	135,6
9	test006	2	2	0,132	15	N	N	Y	69,6	141

7.2. ábra. Részlet a *StochasticTestResults.ods* fájlból

Az ábra alapján látható, hogy egy teszteset teszteléséhez két különböző fájlra van szükség: egyre a kötött batch méretű esethez, valamint egy másikra a változó batch méretű, illetve a kétlépcsős esethez (hiszen utóbbi bekapcsolásához csupán egy parancssori kapcsolóra van szükség). Az egy tesztesethez használt fájlokban (Pl. test001 és teszt002) található random generált paraméterek megegyeznek, ezek csupán a batch méretekben térnek el, oly módon, hogy a kötött batch méretű esetben generált számok (Pl. test001 fájl esetén $s_A = 8, s_B = 10$) a változó batch méretű fájlban a batch méretek alsó, vagy felső korlátait reprezentálják (azaz s_p^{min}, vagy s_p^{max} értékének fognak megfelelni), a másik korlátot szintén véletlenszerűen választjuk ki ezekben az esetekben. (Pl. test002 fájl esetén $s_A^{max} = 8, s_B^{max} = 10$ értékek kerültek beállításra, $s_A^{min} = 5, s_B^{min} = 7$ értékek lettek generálva.) Ezen módszert használva felfedhetőek a különböző módok esetleges hibái, könnyen belátható ugyanis, hogy helyes működés esetén a változó batch méretű optimális eredménynek legalább meg kell egyeznie, vagy jobbnak kell lennie a kötött batch méretű várható profit értéknél, hiszen ha a kötött batch méretek a változó batch méretek alsó, vagy felső határát reprezentálják, valamint a solver szabadon választhatja ki $x_p(b_p)$ értéket a $[s_p^{min} \cdot b_p, s_p^{max} \cdot b_p]$ tartományból, az eredmény legrosszabb esetben is a tartomány azon legszélső értéke lesz, amely egyenlő a kötött batch méret esetén számolt $s_p(b_p)$ értékkel. Hasonló reláció elmondható a változó batch méretű, és a két lépcsős esetek profit értékeire is, hiszen míg a előbbi esetén azonos batch méret kerül beállításra minden forgatókönyv esetén, addig az utóbbi esetben forgatókönyvenként választható meg az ideális batch méret az adott tartományból, tovább növelve esetlegesen a várható profit értékét a változó batch méretű esethez képest, de nem rontva azt a legrosszabb esetben sem. A tesztesetek között a különböző recept és forgatókönyv darabszámokat tartalmazó eseteken kívül megtalálható néhány speciális eset is:

- Abban az esetben például, ha a kötött batch méret a változó batch méretű, illetve a két lépcsős esetben a batch méret alsó és a felső korlátja is egyben (test033 és test034), mindhárom várható profit értéknek meg kell egyeznie.
- Abban az esetben, ha csak egy darab forgatókönyv létezik (test035 és test036), a változó batch méretű eredmény és a két lépcsős eredmény megegyező kell legyen.

#	File name	Number of recipes	Number of Scenarios	Runtime (sec)	Timehorizon	Fix	Variable	Two stage	Axial	Optimal
49	test033	2	2	0,178	15	Y	N	N	-1,4	29,2
50	test034	2	2	0,128	15	N	Y	N	-1,4	29,2
51	test034	2	2	0,128	15	N	N	Y	-1,4	29,2
52	test035	2	1	0,13	15	Y	N	N	43	176
53	test036	2	1	0,127	15	N	Y	N	70	213
54	test036	2	1	0,128	15	N	N	Y	70	213

7.3. ábra. A speciális teszt esetek eredményei.

A teszteredmények, ahogyan az 7.2 ábrán, valamint bővebben a **StochasticTestResults.ods** fájlban is látható alátámasztják a fent leírtakat, tehát a sztochasztikus throughput maximalizáló ilyen szempontból jól működik az alapeseteken, továbbá manuálisan kiszámítva az input fájlokban található paraméterek alapján a várható profit értékét egy találomra választott teszt esetre a 6.1.1 pontban ismertetettek alapján, alátámasztást nyer azon várható profit értékek helyessége is.

7.3. A sztochasztikus multiproduct receptek tesztelése

A sztochasztikus throughput maximalizáló multiproduct receptekkel való teszteléséhez használt teszt fájlok a CD melléklet **Tesztelés/Tesztfájlok/Sztochasztikus_Multiproduct** mappájában találhatóak, míg a teszteredményeket rögzítő **StochasticMultiproductTestResults.ods** fájl a CD melléklet **Tesztelés/Teszteredmények** mappájában található. Mivel a multiproduct esetek lényegében a sztochasztikus alapesetek speciális változatai, ezért a tesztelés hasonlóképpen zajlik, mint az előző pontban leírtak. A receptek, termékek, valamint a forgatókönyvek megfelelő adatai véletlenszerűen kerültek kiválasztásra, a tesztesetek lényegében hasonlóak az előző pontban ismertetettekhez, különös figyelmet szentelve az előzőekben már ismertetett speciális esetekre. A teszteredményeket tartalmazó **StochasticMultiproductTestResults.ods** fájl felépítése hasonló a 7.2. ábrán látottakhoz, azzal a bővítéssel, hogy egy plusz oszlop került felvételre a termékek számának dokumentálására.

File name	Number of recipes	Number of products	Number of Scenarios	Runtime (sec)	Timehorizon	Fix	Variable	Two stage	Axial	Optimal
test001	2	3	2	0,207	15	Y	N	N	5,32	12,92
test002	2	3	2	0,174	15	N	Y	N	5,32	12,92
test002	2	3	2	0,161	15	N	N	Y	5,36	14,56
test003	2	3	1	0,159	15	Y	N	N	0,7	2,7
test004	2	3	1	0,151	15	N	Y	N	1	6,71429
test004	2	3	1	0,16	15	N	N	Y	1	6,71429
test005	2	3	2	0,163	15	Y	N	N	10,92	16,92
test006	2	3	2	0,13	15	N	Y	N	10,92	16,92
test006	2	3	2	0,13	15	N	N	Y	10,92	16,92
test007	2	6	2	0,186	15	Y	N	N	25,6	28,8
test008	2	6	2	0,203	15	N	Y	N	30,2	44,9
test008	2	6	2	0,136	15	N	N	Y	31	46,4

7.4. ábra. Részlet a *StochasticMultiproductTestResults.ods* fájlból

A teszteredmények alátámasztják, hogy a sztochasztikus throughput maximalizáló megfelelően működik az alapesetekhez tartozó inputfájlokkal, illetve multiproduct receptekhez tartozóakkal egyaránt.

8. fejezet

Összefoglalás

Szakdolgozatomban szakaszos gyártórendszerek ütemezésével foglalkoztam sztochasztikus környezetben. A feladat az S-gráf keretrendszer, azon belül annak profit maximalizálójának felkészítése volt a sztochasztikus környezet kezelésére, új algoritmusok a rendszerbe történő implementálásával. Ehhez először is áttanulmányoztam az ütemezéssel kapcsolatos szakirodalmat, valamint az S-gráf keretrendszerhez tartozó irodalmat, megértettem a különféle ütemezési algoritmusok, főként a determinisztikus profit maximalizáló működését. Ezután a sztochasztikus profit maximalizáláshoz szükséges elméleti algoritmusok megvizsgálása, illetve azok részleteinek kidolgozása, a keretrendszerbe történő implementálása következett. Az implementáció végeztével alapos tesztelésen esett át a determinisztikus, illetve a sztochasztikus profit maximalizáló is. A teszteredmények tudatában elmondható, hogy mind a determinisztikus, mind az új sztochasztikus profit maximalizáló jól működik, az elvárt eredményeket adják vissza.

Munkám eredményeképpen, tehát immáron lehetséges sztochasztikus környezetben adott szakaszos gyártórendszerek ütemezési problémáinak megoldása az S-gráf szolver segítségével. Számos feladat van azonban az S-gráf keretrendszerben ami még megvalósításra vár, mint például az általam a 6.5. alfejezetben említett LP modell implementálása, mellyel a multiproduct receptek sztochasztikus profit maximalizálása esetén egy terméket akár több recept is előállíthatna, sokat javítva ezzel a szolver által megoldható problémák körét. Jövőbeli tervek közé tartozik továbbá az is, hogy a különböző konfigurációk feasibilitásának cachelésével, a profit maximalizáló újbóli futtatása során, elkerülhető legyen a feasibilitások újra tesztelése, ha az azt befolyásoló paraméterek nem, csupán a profitot befolyásoló paraméterek változtak.

Például, ha a profit maximalizáló előző futtatásának másnapjára az elérhető gépek száma, a termék előállításához szükséges idők, az időhorizont, stb. nem változtak, csupán a kereslet, a termék ára, vagy egyéb sztochasztikus paraméterek változtak. Ez a módszer ezekben az esetekben nagyban meggyorsítaná a profit maximalizáló lefutását, ugyanis throughput maximalizálás során a feasibilitások tesztelése a legidőigényesebb feladat, cache-eléssel ez sok esetben kiküszöbölhető lenne.

Irodalomjegyzék

[1] H. L. Gantt. *Work Wages and Profits (Management in History No 41)*. Hive Pub Co, 1973.

[2] Mate Hegyhati and Ferenc Friedler. Overview of industrial batch process scheduling. *Chemical Engineering Transactions*, 21:895–900, 01 2010.

[3] Máté Hegyháti. *Batch Process Scheduling: Extensions of the S-graph Framework*. PhD thesis, Doctoral School of Information Science and Technology – University of Pannonia, 2015.

[4] Tibor Holczinger, Thokozani Majozi, Mate Hegyhati, and Ferenc Friedler. An automated algorithm for throughput maximization under fixed time horizon in multipurpose batch plants: S-graph approach. In *17th European Symposium on Computer Aided Process Engineering*, volume 24 of *Computer Aided Chemical Engineering*, pages 649 – 654. Elsevier, 2007.

[5] E. Kondili, C.C. Pantelides, and R.W.H. Sargent. A general algorithm for short-term scheduling of batch operations—i. milp formulation. *Computers & Chemical Engineering*, 17(2):211 – 227, 1993. An International Journal of Computer Applications in Chemical Enginnering.

[6] Thokozani Majozi and Ferenc Friedler. Maximization of throughput in a multipurpose batch plant under a fixed time horizon: S-graph approach. *Industrial & Engineering Chemistry Research*, 45(20):6713–6720, 2006.

[7] 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, 22:S847 – S850, 1998. European Symposium on Computer Aided Process Engineering-8.

[8] E. Sanmarti, T. Holczinger, L. Puigjaner, and F. Friedler. Combinatorial framework for effective scheduling of multipurpose batch plants. *AIChE Journal*, 48(11):2557–2570, 2002.

[9] Anthony A. Williams. *C++ Concurrency in Action: Practical Multithreading*. Manning Publications Co., 20 Baldwin Road, Shelter Island, NY 11964, 1st edition, 2012.

[10] Laurence A. Wolsey. *Integer Programming*. Wiley-Interscience, 1998.

A. függelék

Jelmagyarázat

P a termékek halmaza

b_p a legyártott batch-ek darabszáma az adott konfigurációban

s_p a termék batch mérete (fix batch méret esetén) [Kg]

s_p^{min}, s_p^{max} adott termékhez tartozó lehetséges legkisebb, legnagyobb batch méret (válzotó batch méret esetén) [Kg]

S a forgatókönyvek halmaza

$prob_s$ s forgatókönyv valószínűsége $s \in S$

$dem_{s,p}$ p termék iránti kereslet az s forgatókönyvben $s \in S, p \in P$ [Kg]

$price_{s,p}$ p termék ára az s forgatókönyvben $s \in S, p \in P$ [Cost Unit/Kg]

$oc_{s,p}, uc_{s,p}$ p termék túl-, és alul termelési költsége s forgatókönyvben $s \in S, p \in P$ [Cost Unit/Kg]

$ExpProfit_p(x) = \sum_{s \in S} prob_s \cdot profit_{s,p}(x)$ p termék x értékben vett várható profit értéke

$$Profit_{s,p}(x) = \begin{cases} price_{s,p} \cdot x - (dem_{s,p} - x) \cdot uc_{s,p} & \text{ha } x < dem_{s,p} \\ price_{s,p} \cdot dem_{s,p} - (x - dem_{s,p}) \cdot oc_{s,p} & \text{egyébként} \end{cases}$$

B. függelék

Input fájlok

product		
name	no_stages	number
		1
A	2	1
B	3	1
C	2	1

equipment	
name	number
	1
E1	1
E2	1
E3	1
E4	1

proctime			
pr_name	eq_name	stage	time
product.name	equipment.name>		infty
A	E1	1	5
A	E3	1	3
A	E4	2	5
B	E2	1	6
B	E1	2	4
B	E4	3	2
C	E3	1	4
C	E2	2	5

B.1. ábra. A *multipurpose.ods* fájl

product				
name	no_stages	batch_size	batch_size_min	batch_size_max
A	2		1	2
B	3		1	2

scenario	
name	probability
S1	0.5
S2	0.5

equipment	
name	number
	1
E1	1
E2	1
E3	1
E4	1

proctime			
pr_name	eq_name	stage	time
<product.name>	quipment.name>		infty
A	E1	1	5
A	E3	1	3
A	E4	2	5
B	E2	1	6
B	E1	2	4
B	E4	3	2

scenario_data					
sc_name	pr_name	product_price	demand	over_production_cost	under_production_cost
<scenario.name>	<product.name>				
S1	A	1	1	1	4
S1	B	1	1	1	1
S2	A	1	2	1	4
S2	B	1	1	1	1

B.2. ábra. A *stochastic.ods* fájl

product

name	no_stages	batch_size	batch_size_min	batch_size_max
A	2		1	2
B	3		1	2

equipment

name	number
	1
E1	1
E2	1
E3	1
E4	1

sub_product

pr_name	name	ratio
<product.name>		
A	C	0.5
A	D	0.5
B	E	1

scenario

name	probability
S1	0.5
S2	0.5

proctime

pr_name	eq_name	stage	time
<product.name>	<equipment.name>		infty
A	E1	1	5
A	E3	1	3
A	E4	2	5
B	E2	1	6
B	E1	2	4
B	E4	3	2

scenario_data

sc_name	ub_product_nam	product_price	demand	over_production_cost	under_production_cost
<scenario.name>	ub_product.name>				
S1	C	1	2	1	1
S1	D	1	3	1	1
S1	E	1	4	1	1
S2	C	1	4	1	1
S2	D	1	4	1	1
S2	E	1	1	1	1

B.3. ábra. A *stochastic_extended.ods* fájl

C. függelék

CD melléklet tartalma

```
CD melléklet/
├── Algoritmusok
│   ├── DO_Sgraph_Makespan_Minimization.odg
│   └── DO_Sgraph_Throughput_Maximization.odp
├── Tesztelés
│   ├── Teszteredmények
│   │   ├── DeterministicTestResults.ods
│   │   ├── StochasticTestResults.ods
│   │   └── StochasticMultiproductTestResults.ods
│   └── Tesztfájlok
│       ├── Determinisztikus
│       │   ├── test001.ods
│       │   ├── ...
│       │   └── test005.ods
│       ├── Sztochasztikus
│       │   ├── test001.ods
│       │   ├── ...
│       │   └── test040.ods
│       └── Sztochasztikus_Multiproduct
│           ├── test001.ods
│           ├── ...
│           └── test012.ods
└── Solver kód
    ├── input
    │   ├── multipurpose.ods
    │   ├── stochastic.ods
    │   ├── stochastic_extended.ods
    │   └── ...
    └── src
        └── arguments.cpp
```

```
│
│   │
│   ├── base
│   │   ├── piecewiselinearfunction.h
│   │   ├── piecewiselinearfunction.cpp
│   │   └── ...
│   ├── lib
│   │   ├── relationalproblemreader.h
│   │   ├── relationalproblemreader.cpp
│   │   └── ...
│   ├── solver
│   │   ├── mainsolver.cpp
│   │   ├── recipe.h
│   │   ├── recipe.cpp
│   │   ├── sgraph.h
│   │   ├── sgraph.cpp
│   │   ├── statistics.h
│   │   ├── throughputsolver.h
│   │   ├── throughputsolver.cpp
│   │   ├── throughputui.cpp
│   │   └── ...
│   └── ...
├── README.md
└── ...
```