SZÉCHENYI ISTVÁN EGYETEM

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

Ipari környezetben gyakran fordulnak elő ütemezési problémák, ezen problémák megoldására számos módszer került már kidolgozásra, illetve publikálásra a szakirodalomban. Ilyen megoldómódszerek például a MILP (Mixed Integer Linear Programming) modellek, vagy az általam részletesen vizsgált S-gráf módszertan. A publikált módszerek többsége alapvetően determinisztikus paraméterekkel adott problémák megoldására lett elsősorban megalkotva. Az említett ütemezési problémák azonban sok esetben nem tisztán determinisztikusak, több paraméter is lehet sztochasztikus. Ilyen sztochasztikus paraméter lehet például a termék iránti kereslet, vagy éppen a termék aktuális ára, amin értékesíteni lehet.

Munkám során ilyen sztochasztikus környezetben adott szakaszos gyártórendszerek ütemezésével foglalkoztam, az S-gráf módszertan segítségével. Ezen belül is azon feladat osztály megoldását tűztem ki célul, melyben egy adott időn (időhorizonton) belül a várható profitot szeretnénk maximalizálni. Ezen probléma kör megoldása szolgáló tervezett elméleti algoritmusok, habár a szakirodalomban megtalálhatóak, ezek részleteinek kidolgozása, valamint a meglévő keretrendszerbe történő implementálása még váratott magára.

A probléma körhöz tartozó módszerek részleteinek kidolgozásával, valamint a keretrendszerbe történő implementálásáról, teszteléséről szól tehát jelen dolgozat. Az 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

Scheduling problems often occur in an industrial environment. Several different methods have been developed and published to solve these problems. These include different MILP (Mixed Integer Linear Programming) models for example, and the S-graph framework, which is in the focus of this thesis. Most of the published methods were primarily designed to solve problems with only deterministic parameters. However, these scheduling problems are in many cases not purely deterministic, because several parameters may be stochastic. Such a stochastic parameter can be, for example, the demand for the product or the current price of the product.

During my work, studied the scheduling of batch production systems in a stochastic environment, using the S-graph methodology, namely the case in which we would like to maximize the expected profit of in a given time horizon. Although theoretical algorithms for solving these kinds of problems can be found in the literature, the details of these methods were yet to be worked out and implemented in the existing framework.

This thesis is about the detailed elaboration, implementation and testing of these methods. 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. Ilyen ütemezési problémákra jó példa a tömegközlekedés, ahol a menetrendek megtervezése tipikusan ilyen ütemezési feladat, vagy ha egy moziban aktuálisan vetített filmek vetítő termek közti felosztását vesszük például, de gondolhatunk akár a számítógépek feladatainak ütemezésére is, ahol eldöntendő, hogy éppen melyik folyamat kapjon hozzáférést az erőforrásokhoz. 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 feladatok szétosztása az elérhető erőforrások között, valamint ezen feladatok idő intervallumokhoz 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, mellyel kapcsolatban számos probléma osztály fellelhető. Ilyen probléma lehet például a gyártáshoz szükséges idő minimalizálása, vagy éppen a profit maximalizálása. Ezen ütemezési feladatok 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 a termelési mennyiség (throughput)[1] maximalizálása. Sok esetben a termelés mennyiség és a profit szinonimaként használható, hiszen a termelt mennyiséggel arányosan nő a bevétel is, ez azonban nem mindig így van, különösen az általam a továbbiakban részletesen vizsgált sztochasztikus esetekben, éppen ezért ezen két fogalom megkülönböztetendő. 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. Ezenkívül adott a részfeladatok elvégzésének betartandó sorrendje is. 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.

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

Simple multiproduct recept: Minden termék több lineárisan egymást követő lépésből áll elő, valamint az egyes termékek ugyan olyan sorrendben használják a berendezéseket.

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

Multipurpose recept: Minden termék több lineárisan egymást követő lépésből áll elő, azonban az egyes termékek nem azonos sorrendben használják a berendezéseket.

Precedential recept: A lépések egymásra épülése nem lineáris, 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). Ez alapján három különböző esetet különböztetünk meg:

UW - Unlimited Wait: A legmegengedőbb, 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.

A másik szempont pedig a mennyiség, azaz, hogy adott köztes termékből adott üzemben mennyit lehet eltárolni. Ezen szempont alapján a következő kategóriák azonosíthatóak be:

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 a tároló véges kapacitású.

NIS - No Intermediate Storage: Ebben az esetben nem állnak rendelkezésre tároló egységek a köztes termék eltárolására, de a köztes termék továbbra is várakozhat az őt gyártó 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 ütemezés maga is osztályozható aszerint, hogy megvalósítható (feasible), vagy nem valósítható meg (infeasible). Egy ütemezés alatt általában magát az ütemtervet értjük, amely nem más mint az összes feladat hozzárendelése berendezésekhez, illetve időintervallumokhoz. Infeasible ütemezésről akkor beszélünk, ha az ütemterv nem elégíti ki a probléma definiálása során lefektetett megkötések akár egyetlen egy tagját is, ezzel szemben feasible az ütemezés, ha az ütemterv minden megkötést kielégít.

2.2. Megoldó módszerek

Az utóbbi évtizedekben számos különböző módszer került publikálásra, melyek vegyipari gyártási rendszerek ütemezési problémáinak 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.

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] Az ilyen ütemezési feladatokra felírt MILP modelleknek különböző típusai léteznek:

Időfelosztásos módszerek (Time discretization based) : Az időhorizont diszkrét pontokra, vagy résekre bontásán alapszanak. 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 felhasználásával. Történtek ilyen fejlesztések, amelyek ilyen szempontból gyorsabb, jobb modellekhez vezettek, azonban ezek bizonyos esetekben szuboptimális, vagy akár infeasible megoldást is eredményezhetnek.

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 az ilyen fajta modellek megoldása gyorsabb mint az előző típusé. 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, az első bináris változó a feladatok berendezésekhez való rendelését jelzi, a második pedig azt, hogy adott feladat egy másikat megelőz-e adott berendezésben.

Időzített automaták, Időzített Petri háló

A Petri hálók, és automaták gyakran használatosak diszkrét eseményű rendszerek modellezésére, ahhoz azonban, hogy ezen módszerek felhasználhatóak legyenek gyártásütemezési problémák megoldására is, időzítésekkel kellett kiegészíteni őket. Így születtek meg a Timed Place Petri Net (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 alapú módszerek, illetve az S-gráf módszertan algoritmusainak hatékonyságához képest is.

S-gráf keretrendszer

Az S-gráf keretrendszer volt az első publikált gráf elméleten alapuló módszer szakaszos gyártórendszerek ütemezési problémáinak megoldására. [8] Az S-gráf keretrendszer részletesebb bemutatása a 3. fejezetben olvasható.

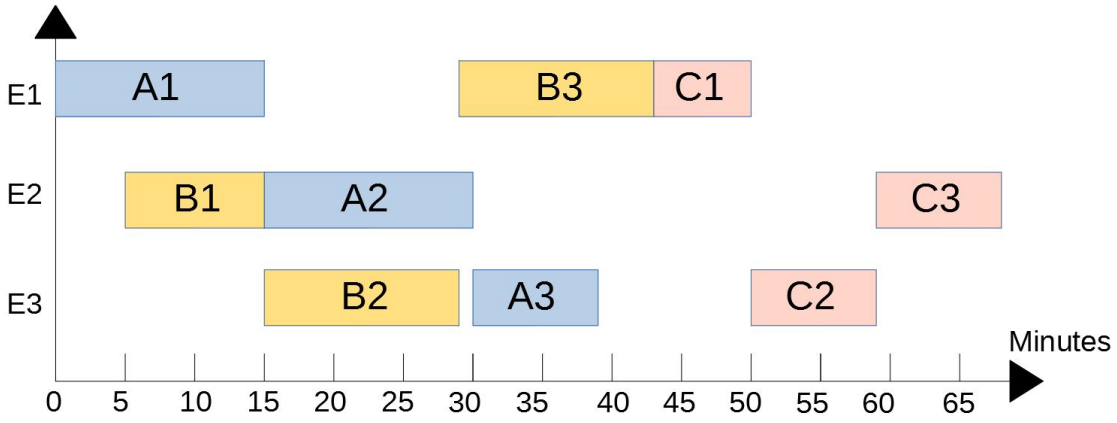
2.3. Az ütemtervek vizualizációja

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



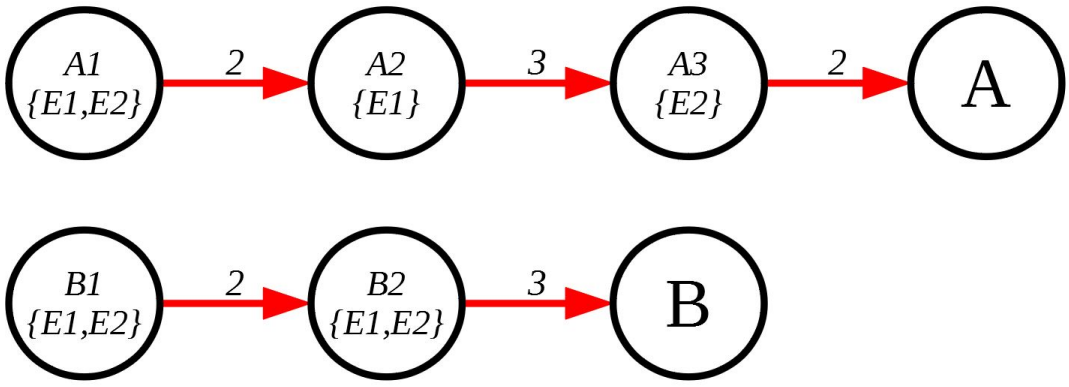
2.1. ábra. Gannt-diagram egy ütemterv szemléltetésére

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 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 csúcsok jelölik, amelyeket csomópontoknak nevezünk. Ezenkívül, ha két feladat között összeköttetés van, ezt a gráfon a két feladatot reprezentáló csomópontok közötti irányított él jelöli. Az ütemezési információ nélküli S-gráfot recept gráfnak (**recipe graph**) nevezzük, melyre egy példa a 3.1 ábrán látható.

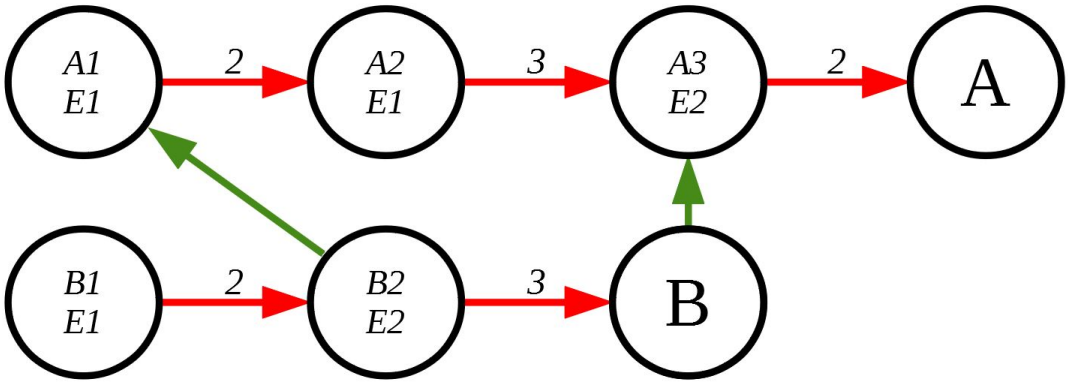


3.1. ábra. A recept gráf szemléltetése

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 irányított élei reprezentálják egyrészt két részfeladat közötti függőséget, abban ez esetben például, ha ez egyik részfeladat állítja elő a másik részfeladathoz szükséges bemeneti részterméket, 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. Az éleken található súlyok pedig a részfeladat megoldásához szükséges gyártási időt reprezentálják, abban az esetben, ha egy részfeladatot több berendezés is el tud végezni, az irányított él súlya a berendezésekhez tartozó gyártási idők közül a legkisebb lesz.

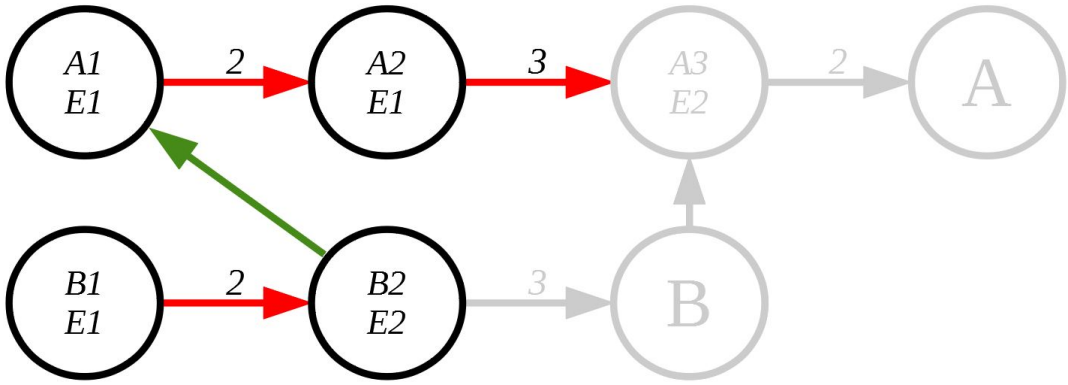
Az S-gráf keretrendszerben található algoritmusok az előzőekben bemutatott recept gráfot egészítik ki ütemezési élekkel, amelyek az algoritmus által meghozott ütemezési döntéseket reprezentálják. Az ily módon előállított gráfot, függetlenül attól, hogy van-e még meghozatlan ütemezési döntés, vagy pedig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



3.2. ábra. Az ütemezési gráf szemléltetése

Az ábrán látható gráfon már minden ütemezési döntés meghozásra került, a zöld élek modellezik az ütemező algoritmus által meghozott ütemezési döntéseket. 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 élek 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. Az ábrán megfigyelhető továbbá az is, hogy a zöld ütemezési nyíl nem közvetlenül a B1 csomópontból, hanem a B2-ből került behúzásra az A1 felé. Ezen jelölés azt szemlélteti, hogy A1 feladat elkezdéséhez nem elegendő az, hogy E1 berendezés elvégezze B1 feladatot, de a B1 feladat során termelt részterméket E2-nek először át kell vennie E1 berendezéstől, addig ugyanis az E1 berendezésben kerül tárolásra NIS tárolási irányelv szerint.

3.1. Makespan minimalizálás az S-gráf keretrendszerben

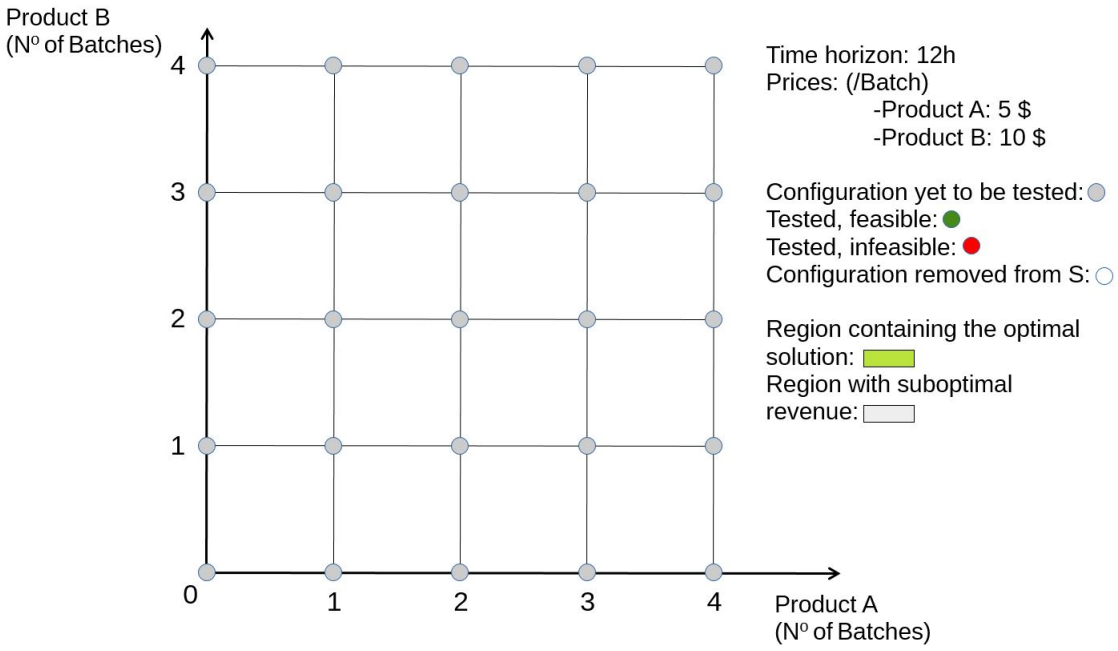
Az S-gráf keretrendszer eredetileg makespan minimalizációs célokra lett megalkotva. A makespan minimalizáló algoritmus egy Branch & Bound algoritmus, melynek segítségével adott receptgráffal reprezentált termékek gyártási ideje minimalizálható. Az algoritmus fontos szerepet játszik azonban a 3.2. alfejezetben tárgyalt throughput maximalizáló algoritmusban is, éppen ezért az algoritmus rövid áttekintése hasznos lehet a továbbiakban, ezt tartalmazza jelen alfejezet. Az S-gráf keretrendszer makespan minimalizáló algoritmusának lefutására egy részletes példa megtalálható a CD melléklet **Algoritmusok** mappájában

DO_Sgraph_Makespan_Minimization.odg néven. Az algoritmus először végtelenre állítja be a makespan értékét, valamint inicializálja a nyitott részproblémák halmazát, amelyben kezdetben csupán a recept gráf található. Minden iteráció során egy részprobléma kerül kiválasztásra ebből a halmazból, ez a kiválasztás többféle stratégia alapján történhet, ezért ennek mikéntje implementáció függő. Az iteráció kezdetén a részprobléma kiértékelésre kerül egy bound függvény által abból a szempontból, hogy optimális eredményt képes-e szolgáltatni. Ez a bound függvény leggyakrabban a gráfban található leghosszabb út alapján kerül kiszámításra. Ha az aktuális részprobléma bound értéke nem kisebb mint az aktuális legjobb eredmény, az iteráció véget ér, egy új részprobléma kerül kiválasztásra a halmazból. Ha a bound érték kisebb mint az aktuális legjobb érték, az algoritmus leellenőrzi, hogy a részprobléma esetén minden ütemezési döntés meghozásra került-e. Ha ez a feltétel igaz, akkor a makespan értéke frissítésre kerül az aktuális részprobléma bound-jával, valamint elmentésre kerül az ütemezési gráf is, mint az eddigi legjobb megoldás. Abban az esetben, ha a részproblémához tartozó ütemezési gráf nem teljes, az algoritmus az elérhető berendezésekhez rendeli a még elérhető feladatokat. Minden berendezéshez rendelt feladat esetén lemásolásra kerül az aktuális ütemezési gráf. Ezen másolatok kiegészítésre kerülnek a hozzárendelés alapján behúzható ütemezési élekkel, valamint ezen élek súlyaival. Ezután minden ilyen módon létrehozott új gráf hozzáadásra kerül a nyitott részproblémák halmazához. Az algoritmus akkor ér véget, ha ez a halmaz kiürül, ebben az esetben visszaadásra kerül a legjobb megoldást reprezentáló ütemezési gráf és a hozzá tartozó makespan érték, ha létezik a problémának legalább egy feasible megoldása.

3.2. Profit maximalizálás az S-gráf keretrendszerrel

A makespan minimalizáláson kívül az S-gráf keretrendszer később kibővítésre került egy throughput maximalizáló algoritmussal, amely segítségével immáron throughput maximalizálásra is képes. Az throughput maximalizáló algoritmus alapötletét Majozi és Friedler [6], valamint Holczinger és társai [4] fektették le. Az algoritmus lényege, hogy a termékek lehetséges batch darabszámai alapján különböző konfigurációk kerülnek létrehozásra, melyek között az algoritmus segítségével eredményül kapható a legnagyobb throughputot eredményező konfiguráció, ha létezik megvalósítható (feasible) megoldás a problémára. Egy konfiguráció

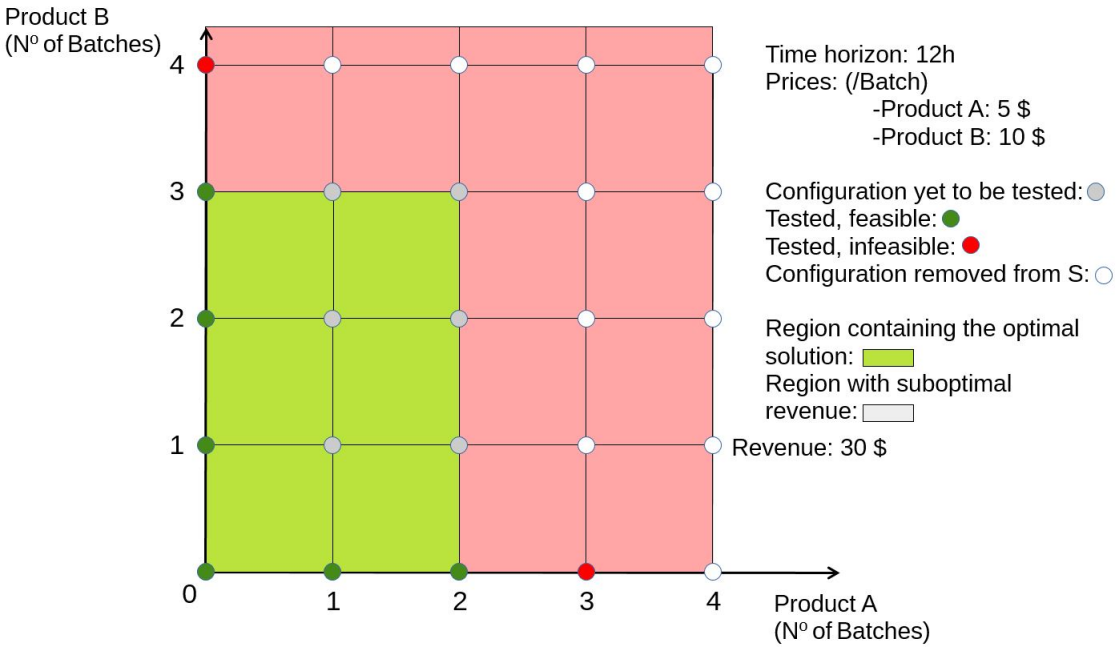
alatt tehát azt értjük, hogy adott termékből hány batch-et termelünk. Ezen konfigurációk elképzelhetőek egy n dimenziós térben, ahol n a különböző termékek számát jelöli. Ebben a térben a tengelyek menti megoldások azokat az eseteket jelölik, melyekben csak egy fajta terméket termelünk. A throughput maximalizáló algoritmus működésének egy példán keresztül történő szemléltetésére készült folyamatábra a CD melléklet **Algoritmusok** mappájában található **DO_Sgraph_Throughput_Maximization.odp** néven. Ezen fájl alapján a fentiekre példa két termék esetén a következő koordináta rendszer:



3.4. ábra. A konfigurációkat tartalmazó tér szemléltetése két termék esetén

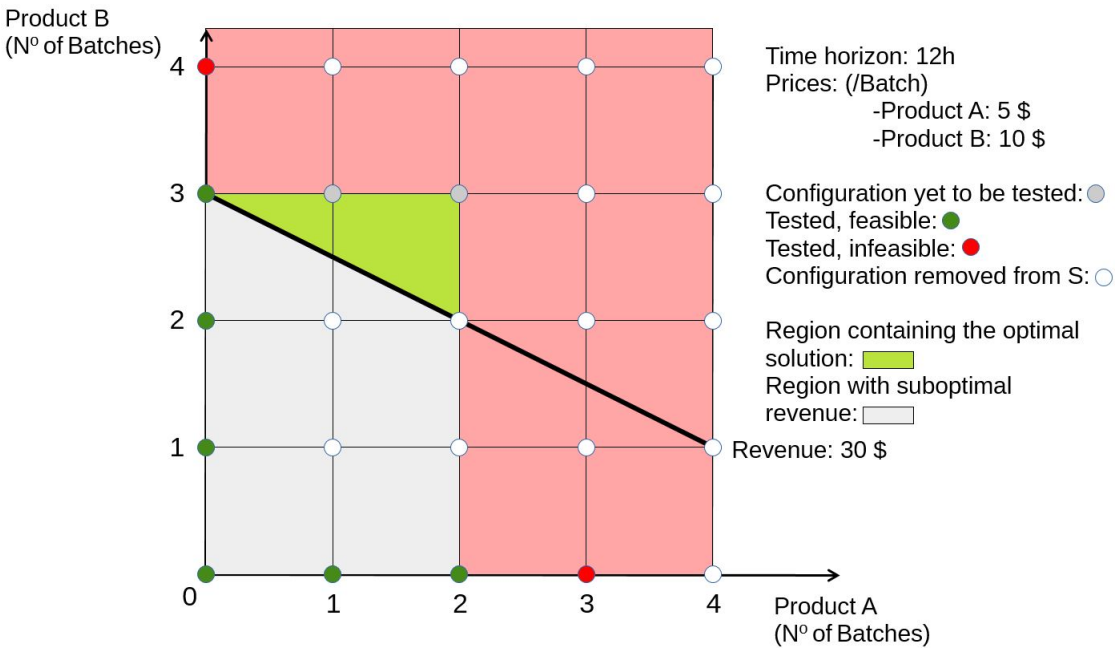
Kezdetben a konfigurációk halmaza tartalmazza az összes lehetséges konfigurációt az adott termékekre, majd minden iteráció során kiválasztásra kerül egy konfiguráció, amelynek teszteljük feasible-itását, azaz hogy a rendelkezésre álló időhorizont alatt megvalósítható-e adott konfiguráció legyártása. 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. Ennek szemléltetésére szolgál a 3.5. ábra.



3.5. ábra. Példa az optimális megoldást tartalmazó térre két termék esetén

Mivel a konfigurációk tesztelése erőforrás igényes feladat, ezért a konfigurációk kiválasztásának sorrendje ezen a téren belül 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 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 3.6 ábra hivatott.



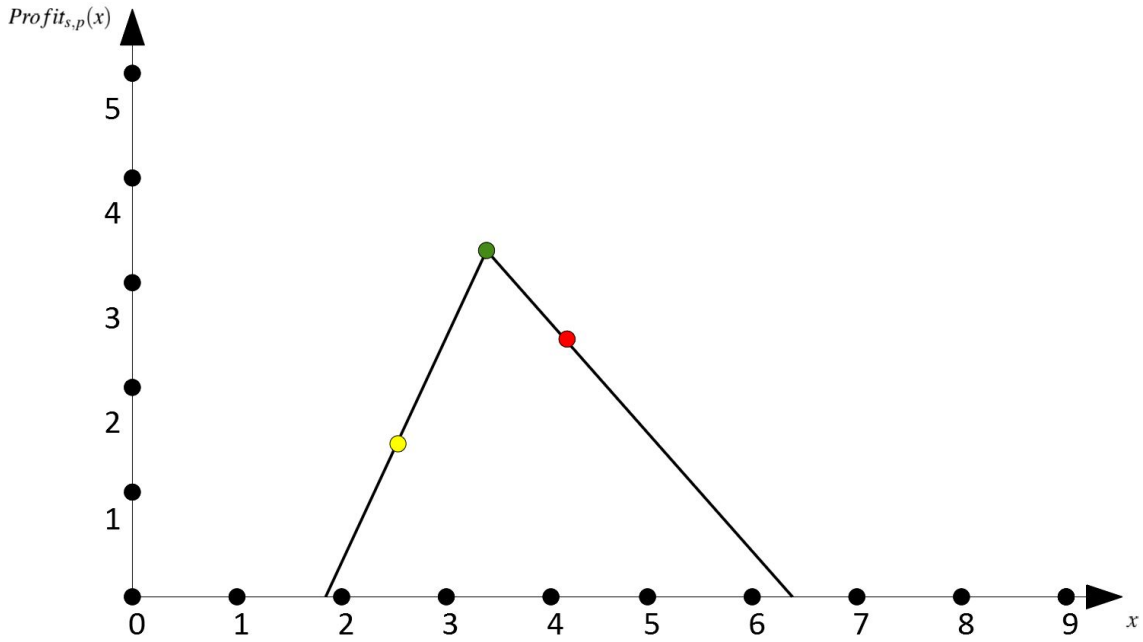
3.6. ábra. A revenue line szemléltetése

Jól látható, hogy jelen esetben a revenue line behúzása harmadára 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 konfiguráció sem valósítható meg az adott időkorláton belül, 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

Valódi ipari környezetben gyakran előfordulnak olyan esetek, amelyekben a megoldandó problémát nem lehet tisztán determinisztikus paraméterekkel leírni, ezeket felhasználva megoldani, hanem szükség van új, a bizonytalanságokat kifejező, sztochasztikus paraméterekre 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, ezeket a kiadásokat túltermelési költségeknek nevezzük. Beszélhetünk alul termelési költségről is, például abban az esetben, ha egy szerződésben foglalt elvárt darabszám legyártásának teljesítése nem következett be, ebben az esetben akár kötbért fizetésére is kötelezhető lehet adott cég. Ezek alapján egy lehetséges profit függvény szemléltetésére szolgál a 4.1. ábra. Az ábra alapján jól látszik, hogy adott termék bevétele akkor lesz maximális, ha a kereslettel egyező darabszámot 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



4.1. ábra. A profit függvény szemléltetése

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. Jól látható, hogy az ilyen fajta problémák merőben eltérőek, az eredeti determinisztikus problémákhoz képest, éppen ezért egy új módszer kidolgozása szükséges ezek megoldásához. Extrém esetekben előállhat olyan helyzet is, hogy a rendelkezésre álló determinisztikus paraméterek (pl. 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, az új módszerek kidolgozása során fel kell tehát ilyen esetekre is készülni.

[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.

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, és az alul termelés költsége az adott forgatókönyvben

A cél az, hogy döntést hozzunk a termelt batch-ek darabszámát, illetve egyes esetekben azok méretét illetően, miközben egy olyan kivitelezhető ütemtervet biztosítunk, amelyet követve maximális várható profitot érhetünk el.

4.1. A különböző feladat osztályok

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 batchet 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 formális bemenete

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 jelölései:

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 (változó 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]

5. fejezet

Az S-gráf solver

Mivel munkám része a kidolgozott új módszerek implementációja az S-gráf solver programba, ezért célszerű röviden bemutatni ezen program használatát, felépítését, illetve a további munkám szempontjából fontosabb részek működését. Ezek leírására szolgál jelen fejezet. 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:

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

A solver az OpenMP API-t használja párhuzamosításra, a kód lefordításához Qt5 szükséges, feltétel továbbá a boost könyvtár telepítése is. Ezenfelül Linux alatt történő futtatáshoz letöltendő még a Google OR-Tools könyvtár, Windows operációs rendszer esetén pedig a Visual C++ Redistributable telepítése a követelmény. A fentiek telepítése, illetve a megfelelő könyvtárak elérési útjainak beállítása után qmake segítségével elkészíthető a make file a projekt alapján. Ezen make fájl segítségével a make meghívja a fordítót, amely lefordítja a programot, melynek következtében létrejönnek a futtatáshoz szükséges fájlok, köztük a solver.exe (Windows esetén), mellyel immáron futtathatjuk a megoldó programot az előző példához 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 3.2 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 az aktuális tengelyen. Miután minden tengely mentén visszakapta az első null értéket, véget ér ez az első fázis, 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 3.2 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 megoldómódszer 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:

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

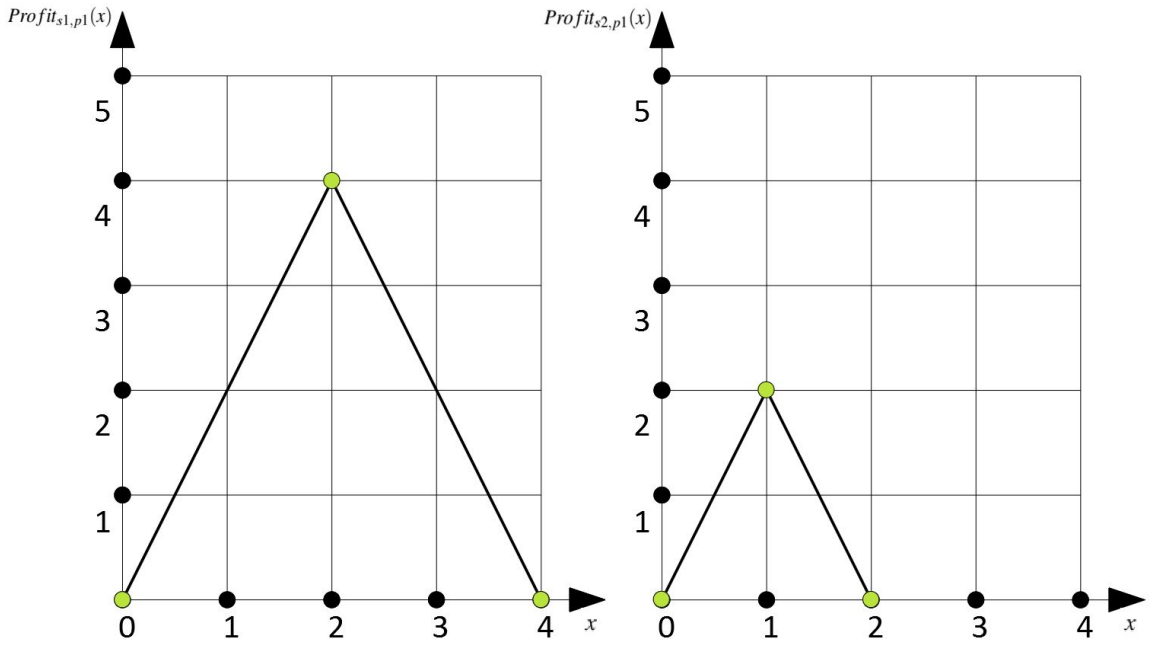
Érdemes még bevezetni a fenti képlet egyszerűsítésére adott p termék x értékben vett várható profit értékére a következő jelölést:

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

Ahol $profit_{s,p}(x)$, azaz adott termék x darabszámának profit értéke az s forgatókönyvben a következőképpen számítható ki:

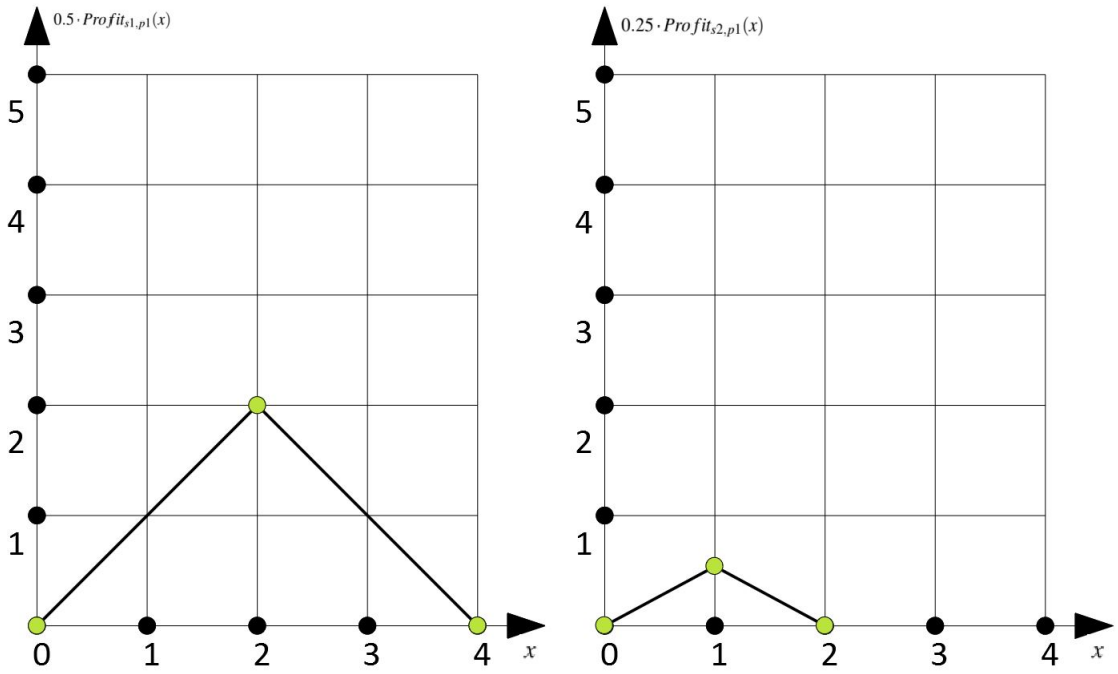
$$Profit_{s,p}(x) = \begin{cases} price_{s,p} \cdot x - (dem_{s,p} - x) \cdot uc_{s,p} & \text{ha } x < dem_{s,p} \\ price_{s,p} \cdot dem_{s,p} - (x - dem_{s,p}) \cdot oc_{s,p} & \text{egyébként} \end{cases}$$

$ExpProfit_p$ 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 az "összenyomott" profit függvényt adott forgatókönyv valószínűségét felhasználva, ezen előállított függvények összeadásával megkapható az $ExpProfit_p$ függvény. A fentiek szemléltetésére szolgálnak az alábbi ábrák $p1$ termék esetén, melynek két forgatókönyve $s1$ és $s2$:



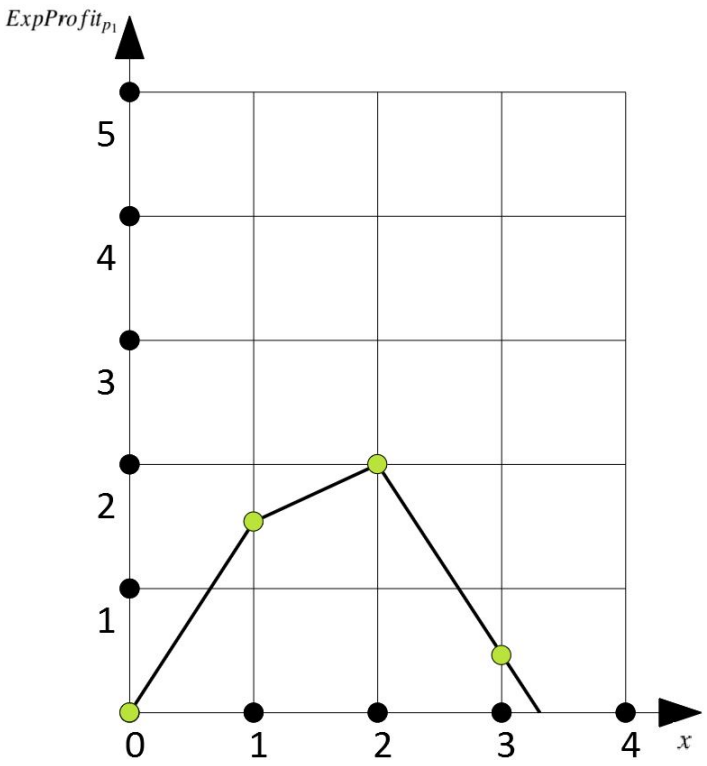
6.1. ábra. p_1 termék profit függvényei s_1 és s_2 forgatókönyvben

A 6.1. ábrán láthatóak $p1$ termék profit függvényei. A bal oldali függvény az $s1$, a jobb oldali az $s2$ forgatókönyvbeli profit függvénye $p1$ terméknek. Miután ezek a függvények megalkotásra kerültek adott termék összes forgatókönyvére, a következő lépésben ezek beszorzásra kerülnek adott forgatókönyvek valószínűségével. Ezt a lépést ábrázolja a 6.2. ábra. Ebben a



6.2. ábra. p_1 termék összenyomott profit függvényei s_1 és s_2 forgatókönyvben

példában $s1$ forgatókönyv valószínűsége 0.5, $s2$ valószínűsége pedig 0.25. Az ábrán látható függvények kiszámításával előáll minden forgatókönyvre a $prob_s \cdot profit_{s,p}$ függvény, amelyeket ezután össze kell adni, hogy előálljon $ExpProfit_{p_1}$ függvény. Ezt a fenti ábrák alapján előállított $ExpProfit_{p_1}$ függvényt ábrázolja a 6.3. ábra. Miután minden termékre előállításra kerül ez az $ExpProfit_p$ függvény, az $ExpProfit_p(x)$ (ahol $x = s_p \cdot b_p$) értékek összegeként előáll a várható profit. Azaz minden termék $ExpProfit$ függvényéből lekérdezésre kerül az x értékhez tartozó profit érték, ahol x az adott termékből termelt mennyiség, mely a termelt batchek száma (b_p) és azok méretének (s_p) szorzataként áll elő. Jelen esetben a batch méret (s_p) egy fixen adott paraméter, hiszen kötött batch méretű esetről beszélünk. Ezután ezen $ExpProfit_p(x)$ értékek összegzésre kerülnek, ez adja meg a várható profit értékét.

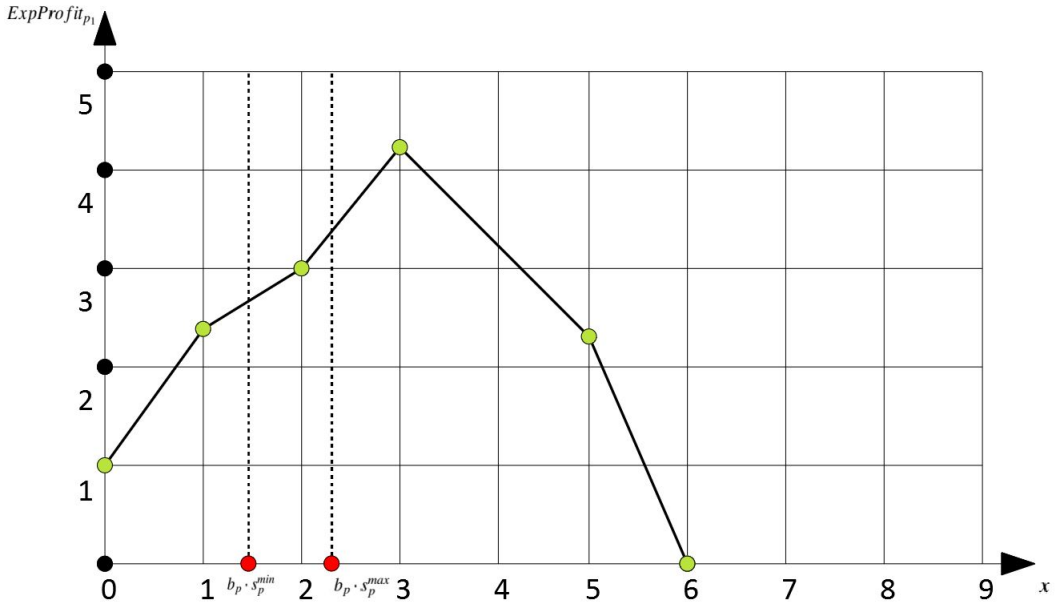


6.3. ábra. p_1 termék ExpProfit függvényének szemléltetése.

6.1.2. Preventív ütemezés változó batch mérettel

Az ehhez az esethez tartozó lépések megegyeznek az előző kötött batch méretű eset lépéseivel, az $ExpProfit_p$ függvények előállításáig. Az előző esettel ellentétben azonban, 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. Az arról való döntés tehát, hogy adott termékből mennyit gyártsunk, azaz x_p érték meghatározása a következő intervallumból kerül kiválasztásra: $[s_p^{min} \cdot b_p, s_p^{max} \cdot b_p]$. Ezen érték kiválasztását a 6.4. ábra szemlélteti. Jelen példában b_p értéke 1, azaz egy darab batchet termelünk p termékből, ezen egy batch termelése mellett a minimális elérhető termelt mennyiség mennyiség, azaz $b_p \cdot s_p^{min}$ értéke 1.5 kg lesz. Egy batch termelésével elérhető maximális mennyiség, $b_p \cdot s_p^{max}$ pedig 2.3 kg lesz. A tartomány amiből tehát ki kell választani x értékét nem más, mint: $[1.5, 2.3]$, ezt a

tartományt jelölik az ábrán a szaggatott vonalak. Az x optimális értéke úgy kerül kiválasztásra, hogy megnézzük, hogy $ExpProfit_{p1}$ maximális értéke a fenti tartománytól balra, vagy jobbra esik-e, vagy esetleg a tartomány része. Ha a tartománytól jobbra esik, mint jelen példa szerint, akkor a tartományunk maximális x értéke, azaz jelen esetben 2.3 kg kerül kiválasztásra, hiszen ezzel az értékkel érhető el jelen tartományban a legnagyobb profit. Ha a tartománytól balra esne $ExpProfit_{p1}$ maximális értéke, akkor a $[s_p^{min} \cdot b_p, s_p^{max} \cdot b_p]$ tartomány legkisebb értéke, azaz jelen esetben 1.5 kg kerülne kiválasztásra. Abban az esetben pedig, ha $ExpProfit_{p1}$ maximális értéke beleesik a $[s_p^{min} \cdot b_p, s_p^{max} \cdot b_p]$ tartományba, akkor a maximális profitot eredményező x érték kerülne kiválasztásra, amely jelen esetben 3 kg lenne. Az x_p értékek kiválasztása után az összes várható profit a fix batch méretű esettel azonosan módon, $ExpProfit_p(x)$ értékek összegzésével állítható elő.



6.4. ábra. Az optimális x_p érték kiválasztásának szemléltetése

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. Azaz a 6.4. ábrán szemléltetett x érték kiválasztása nem az ábrán látható $ExpProfit_p$ függvényből történik, hanem minden, 6.1. ábrán látható $profit_{s,p}$ függvényből egyenként történik. Az így kiválasztott x értékekhez tartozó profit értékek ezután kerülnek beszorzásra a forgatókönyvek valószínűségével. Az így előállt $(prob_s \cdot Profit(x_{s,p}(b_p))$ értékek ezután kerülnek összegzésre a forgatókönyvekre, majd a termékekre nézve, ezzel áll elő az összes várható profit. 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 tehát 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. Egy folytonos, szakaszos, lineáris függvény tárolásához töréspontokat illetve a kezdeti- ,és vég meredekséget kell eltárolni. Az általam használt függvények eleve tartalmaznak egy törést a kereslet értékénél, hiszen az egyes $profit_{s,p}$ függvények csak a kereslet értékéig növekednek, onnantól pedig vagy beállnak a maximális értékre, ha nincsen felül termelési költség, vagy csökkenni fognak a felültermelési költségek miatt. Éppen ezért ezen függvények tárolásához elegendő a következő töréspont kiszámítása: $x = dem_{s,p}$, $y = Profit_{s,p}(x)$. Majd ezután a kezdeti-, és vég meredekségek a következő képlet segítségével, az alul-, és túl termelési költségek alapján kiszámíthatóak.

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

Ezek az értékek minden esetben kiszámíthatók, minden forgatókönyv-termék párosra már az input fájl beolvasását követően, hiszen minden sztochasztikus paraméter adott ehhez a fájlban. Később ezen adatok alapján a többi pont koordinátái, ha valamilyen okból kifolyólag ezek ismerete szükségessé válik, könnyen kiszámíthatóak, hiszen folytonos, lineáris függvényekről beszélünk. Ezek alapján a **PiecewiseLinearFunction** osztály adattagjai a 6.5. ábrán láthatóak. A 6.5 ábrán látható **Coordinate** osztály a függvények töréspontjainak 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:

- 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

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

6.5. ábra. A *PiecewiseLinearFunction* osztály adattagjai

6.2.1. 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 *vector*-ban tárolt összes pont y koordinátáját, valamint a kezdeti-, és vég meredekséget a paraméterként kapott s-el. A függvény szorzását szemléltető példa a 6.2 ábrán látható.

6.2.2. 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 *vector*-ban tárolt koordináta párok között található-e olyan, amelynek x koordinátája egyezik a paraméterként kapott x-el. Ha talál ilyet, egyszerűen visszaadja a megfelelő koordináta páros y értékét. Ha nem található ilyen pont, akkor annak ki kell számítani a koordinátáit, és hozzá kell adni a *vector*-hoz, majd csak ezután lehet visszaadni a keresett y értéket Az x értékhez tartozó y érték kiszámítása a keresett x értéke és a *vector*-ban tárolt pontok alapján háromféleképpen történhet:

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

- Ha a keresett x érték két a *vector*-ban 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 *vector*-ban tárolt utolsó pont x koordinátájának értéke, akkor: $y = y_{Last} - (x - x_{Last}) \cdot Angle_{end}$,ahol x_{Last} és y_{Last} a *vector*-ban tárolt utolsó pont koordinátái.

6.2.3. 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 6.2.2 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. A fentiek szemléltetésére szolgál a 6.6. ábra. A két összeadandó függvényt **a**-nak, illetve **b**-nek nevezzük, az összegként adott új függvény pedig a **c**. Az ábrán a 90. sorban lekérdezzük **b** függvény koordinátáit, melyeken a 91. sorban található *for* ciklussal végigiterálunk. A 92. sorban **a** függvénynek lekérdezzük x helyen vett értékét, ahol x **b** függvény soron következő koordinátájának x értéke. A lekérdezés következtében, ha **a** függvény eddig nem tartalmazott koordinátát x értékkel, ezen koordináta hozzáadásra kerül a 6.2.2. pontban leírtak alapján. Mivel ezt **b** függvény minden pontjára megtesszük, ezért a *for* ciklus végeztével **a** függvény immáron biztosan tartalmazza a **b** függvényben tárolt összes töréspont x értékéhez tartozó saját x értékű koordinátáit az azon a helyen vett y értékekkel. Ezután **a** függvény összes tárolt töréspontján végigiterálunk a 95. sorban található *for* ciklussal. Ebben a *for* ciklusban létrehozzuk az új **c** függvény koordinátáit. Ezen koordináták x értéke az **a** függvény x értékei lesznek rendre hiszen **a** immáron tartalmazza az összes közös x értéket **b**-vel. A koordináták y értéke pedig $a(x) + b(x)$ lesz. Mivel a koordináták **addCoordinate** metódussal való hozzáadása során a kezdeti és a vég meredekségek automatikusan frissítésre kerülnek, ha szükséges ezen értékek újraszámítása,

ezért a *for* ciklus végére előálló **c** függvényben ezeket már nem kell külön beállítani. A metódus végén pedig visszaadásra kerül az új **c** összegfüggvény.

```
88  PiecewiseLinearFunction PiecewiseLinearFunction::operator + (PiecewiseLinearFunction& b) {
89      PiecewiseLinearFunction c = PiecewiseLinearFunction();
90      std::vector<Coordinate> coordinates_b = b.getCoordinates();
91      for (int i = 0; i < coordinates_b.size(); i++) {
92          this->operator()(coordinates_b.at(i).getX());
93      }
94      std::vector<Coordinate> coordinates_a = this->getCoordinates();
95      for (int i = 0; i < coordinates_a.size(); i++) {
96          Coordinate current = Coordinate();
97          current.setX(coordinates_a.at(i).getX());
98          current.setY(coordinates_a.at(i).getY() + b(current.getX()));
99          c.addCoordinate(current);
100     }
101     return c;
102 }
```

6.6. ábra. Az összeadást végző metódus

6.2.4. A függvény horizontális nyújtása

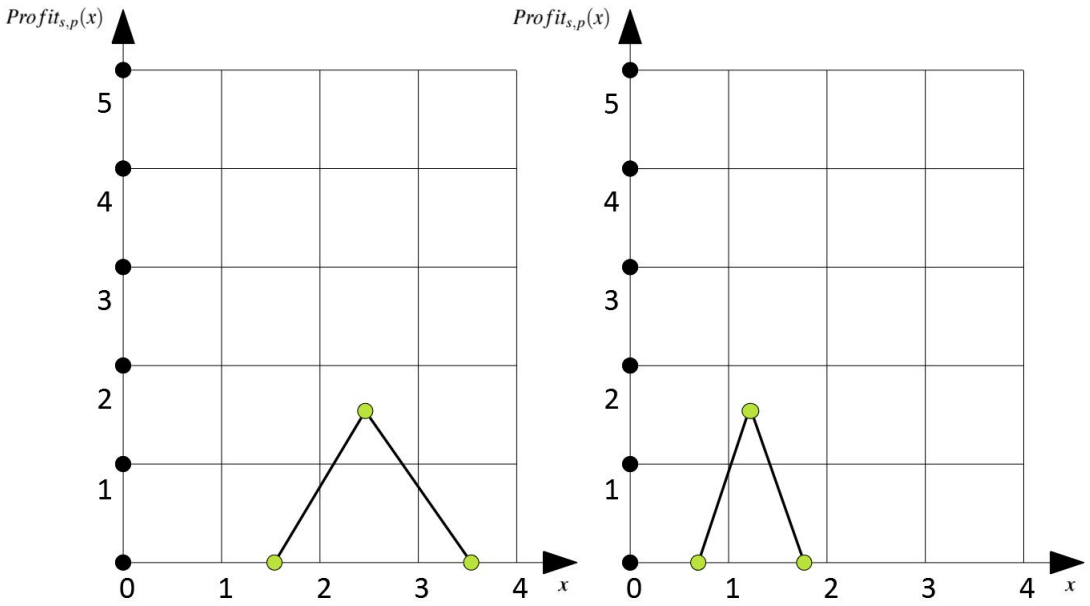
A függvény a 6.5 pontban használt, a 6.7 ábrán bemutatott horizontális nyújtását (illetve összenyomását, s értéktől függően) a **stretchHorizontally(double s)** metódus végzi. A metódus egyszerűen egy *for* ciklus segítségével a függvény összes pontjának x koordinátáját, valamint a kezdeti-, és vég meredekséget beszorozza a paraméterként kapott s értékkel, illetve a meredekségek esetén annak reciprokával.

6.2.5. 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 korábban bemutatott változó batch méretű, illetve két lépcsős esetekben van nagy jelentősége.

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



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

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)

6.8. 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. A **multipurpose.ods** fájl felépítése a 6.9. ábrán látható.

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	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
C	E3	1	4
C	E2	2	5

6.9. ábra. A *multipurpose.ods* fájl

A **stochastic.ods** fájl változtatás nélkül tartalmazza a **multipurpose.ods** fájl **equipment**, és **proctime** tábláit , 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. A **stochastic.ods** fájl felépítése a 6.10. ábrán látható.

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
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

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

6.10. ábra. A *stochastic.ods* fájl

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 *vector*-ba. 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 volt szükség, ugyanis korábbi állapotában a **ThroughputSolver** osztályban találhatóak voltak olyan megoldások, melyek megkövetelték, 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 re-

cept objektumban tárolt *boolean* flag segítségével. Determinisztikus esetben a profit kiszámítása meglehetősen egyszerűen, a 6.11 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;
```

6.11. 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 **GetProductRevenue(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 *vector*-ába, a fájl beolvasásakor. Ezen *vector* terméknevei reprezentálják az úgynevezett ”base product”-okat, azaz a kezdeti termékeket. Ennek a *vector*-nak 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 4 részre bontható:

Determinisztikus probléma product revenue számítása

Determinisztikus esetben a számítás meglehetősen egyszerűen, a $price_p \cdot b_p$ képlettel elvégezhető.

Kötött batch méretű probléma product revenue számítása

Kötött batch méretű sztochasztikus esetben a várható profit számítása a 6.1.1 pontban leírtak alapján a következő képlettel zajlik:

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

Először tehát egy *for* ciklus segítségével végigiterálunk az összes forgatókönyvön. Minden iterációban lekérdezzük az adott forgatókönyvhöz tartozó $profit_{s,p}$ függvényt reprezentáló **PiecewiseLinearFunction** objektumot, majd ezt megszorozzuk az aktuális forgatókönyv valószínűségével, hogy előálljon a 6.2. ábrához hasonlóan az összenyomott profit függvény. Ezután ezt hozzáadjuk az $ExpProfit_p$ függvényt reprezentáló **PiecewiseLinearFunction** objektumhoz, amely a *for* ciklus előtt került példányosításra. Miután a ciklus véget ér előáll $ExpProfit_p$ függvény végleges formája a 6.3. ábrához hasonlóan. Ezután kiszámításra kerül a gyártott mennyiség, azaz az x érték a batch méret és a gyártott batchek darabszámának szorzataként $(s_p \cdot b_p)$. Kiértékelésre és visszaadásra kerül végül $ExpProfit_p(x)$ értéke. A kötött batch méretű problémához tartozó forráskód részlete a 6.12. ábrán látható.

Változó batch méretű probléma product revenue számítása

Változó batch méret esetén a kötött batch méretű esettel azonos módon kapjuk vissza az $ExpProfit_p$ függvényt, tehát a 6.12. ábrán látható *for* ciklus lefutása ebben az esetben is azonos. Az $ExpProfit_p$ megkapása után azonban x érték, valamint $ExpProfit_p(x)$ érték kiszámítása a 6.1.2. alfejezetben leírtak alapján, a 6.4. ábrán szemléltetettek szerint történik. Meghatározásra kerül tehát az $[s_p^{min} \cdot b_p, s_p^{max} \cdot b_p]$ tartomány, mely segítségével immáron eldönthető az optimális x értéke, és visszaadható a hozzá tartozó $ExpProfit_p(x)$ érték. Ezen folyamat implementációja a 6.13. ábrán látható.

```
for(uint j=0;j<GetScenarios().size();j++){
    double probability=GetScenario(j).getProbability();
    currentScenario=GetScenario(j);
    entries=GetProduct(product_id).GetScenarioData();

    for(uint k=0;k<entries.size();k++){
        if(entries.at(k).GetScenarioName()==currentScenario.GetName()){
                        currentEntry=entries.at(k);
        }
    }

    currentRevenueFunction=currentEntry.GetRevenueFunction();
    currentRevenueFunction=currentRevenueFunction*probability;

    if(j==0){
        ExpProfit=currentRevenueFunction;
    }
    else{
        ExpProfit=ExpProfit+currentRevenueFunction;
    }
}
double batch_size=GetProduct(product_id).GetBatchSize();
return ExpProfit(batches*batch_size);
```

6.12. ábra. Kötött batch méretű probléma product revenue számítása

```
double min_batch=batches*GetProduct(product_id).GetBatchSizeMin();
double max_batch=batches*GetProduct(product_id).GetBatchSizeMax();
double demand_rev_x=ExpProfit.getMaximum().getX();

if(max_batch<demand_rev_x){
    return ExpProfit(max_batch);
}
else if(min_batch<=demand_rev_x && demand_rev_x<=max_batch){
    return ExpProfit(demand_rev_x);
}
else if(min_batch>demand_rev_x){
    return ExpProfit(min_batch);
}
```

6.13. ábra. Az optimális profit érték kiszámítása változó batch méret esetén

Két lépcsős probléma product revenue számítása

Két lépcsős ütemezés esetén a várható profit számítása a 6.1.3 pontban ismertetett módon történik, ezen eset forráskódja a 6.14. ábrán látható. Jól látható, hogy az előző két esettel ellentétben $ExpProfit_p$ függvény két lépcsős ütemezés esetén nem kerül felépítésre. A *for* ciklusban az egyes forgatókönyvek $profit_{s,p}$ függvényei segítségével kerül ugyanis kiválasztásra az optimális x érték. Ezután a $profit_{s,p}(x)$ érték kiértékelésre kerül, majd ezt az értéket megszorozzuk az aktuális forgatókönyv valószínűségével. Ezen beszorzott profit értékek összegeként áll elő, és kerül visszaadásra jelen termék várható profit értéke.

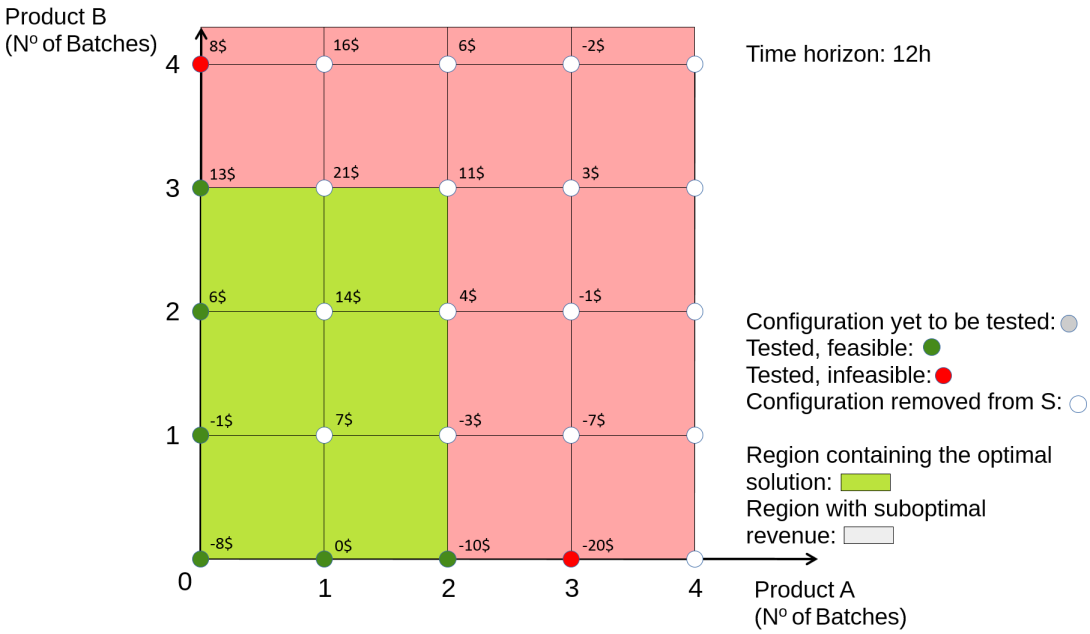
```
double min_batch=batches*GetProduct(product_id).GetBatchSizeMin();
double max_batch=batches*GetProduct(product_id).GetBatchSizeMax();
double current_max_rev;
double toRet = 0.;
for(uint j=0;j<GetScenarios().size();j++){
    double probability=GetScenario(j).getProbability();
    currentScenario=GetScenario(j);

    entries=GetProduct(product_id).GetScenarioData();

    for(uint k=0;k<entries.size();k++){
        if(entries.at(k).GetScenarioName()==currentScenario.GetName()){
                        currentEntry=entries.at(k);
        }
    }

    currentRevenueFunction=currentEntry.GetRevenueFunction();
    double demand_rev_x=currentRevenueFunction.getMaximum().getX();

    if(max_batch<demand_rev_x){
        current_max_rev = currentRevenueFunction(max_batch);
    }
    else if(min_batch<=demand_rev_x && demand_rev_x<=max_batch){
        current_max_rev = currentRevenueFunction(demand_rev_x);
    }
    else if(min_batch>demand_rev_x){
        current_max_rev = currentRevenueFunction(min_batch);
    }

    current_max_rev = current_max_rev*probability;
    toRet+=current_max_rev;
}
return toRet;
```

6.14. ábra. Két lépcsős probléma product revenue számítása

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 3.2. 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 3.6. 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 sztochasztikus esetben ez a vonal nem húzható be, hiszen az azonos revenue értékű konfigurációk nem egy vonalon vannak a sztochasztikus paraméterek értékei miatt, hiszen a determinisztikus esettel szemben itt a revenue értékek nem monoton növekvőek az optimális megoldást tartalmazó téren belül. Ennek a szemléltetésére szolgál a 6.15. ábra.



6.15. 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 6.16 ábra szemlélteti.

```
The input file is input.ods
The output will be directed to: output.txt
1 thread(s) was requested.

First phase: finding axial solutions
-------------------------------------
|Time (s)|    Product    |Count|Feas|
-------------------------------------
|     0.1|             A|    0|  NO|
-------------------------------------
|     0.1|             A|    0|  NO|
-------------------------------------
|     0.1|             A|    0|  NO|
-------------------------------------
|     0.1|             A|    0|  NO|
-------------------------------------
|     0.1|             A|    0|  NO|
-------------------------------------

Best axial revenue: 2
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)|
-------------------------------------------------------------------------------
|     0.2|   3.0 <=    3.0|    0.0|        6|   4/    0|   2/     0|      1.5|
-------------------------------------------------------------------------------

Solution's objective value is: 3
Solution is optimal.
Total execution time: 0.179 s
```

6.16. 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 6.17, és a 6.18 ábra szemlélteti.

```
Best axial revenue: 99
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| 166.0 <=  166.0|    0.0|        7|   5/    0|   2/     0|      3.4|
-------------------------------------------------------------------------------

Solution's objective value is: 166
Solution is optimal.

Optimal batch amounts and batch sizes:
Product: A   number of batches: 2 batch size: 8
Product: B   number of batches: 1 batch size: 10

Expected revenues for each scenario:

Scenario: S1 probability: 50% expected revenue: 195
Scenario: S2 probability: 50% expected revenue: 137

Total execution time: 0.138 s
```

6.17. 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
```

6.18. 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 6.19 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;
```

6.19. 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 6.20 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();
}
```

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

6.5. Multiproduct receptek esete

Ebben az alfejezetben 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:

[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.

- 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 6.2.4 pontban bemutatott metódus segítségével a 6.7 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 $profit$ 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 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
oduct.name	uipment.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 például 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. A **DeterministicTestResults.ods** fájl adatait figyelembe véve kijelenthető tehát, hogy a determinisztikus throughput maximalizáló továbbra is hibátlanul működik.

7.2. A sztochasztikus alapesetek tesztelése

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 (változó 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	2
B	3	1	2	

scenario

name	probability
S1	0.5
S2	0.5

equipment

name	number
E1	1
E2	1
E3	1
E4	1

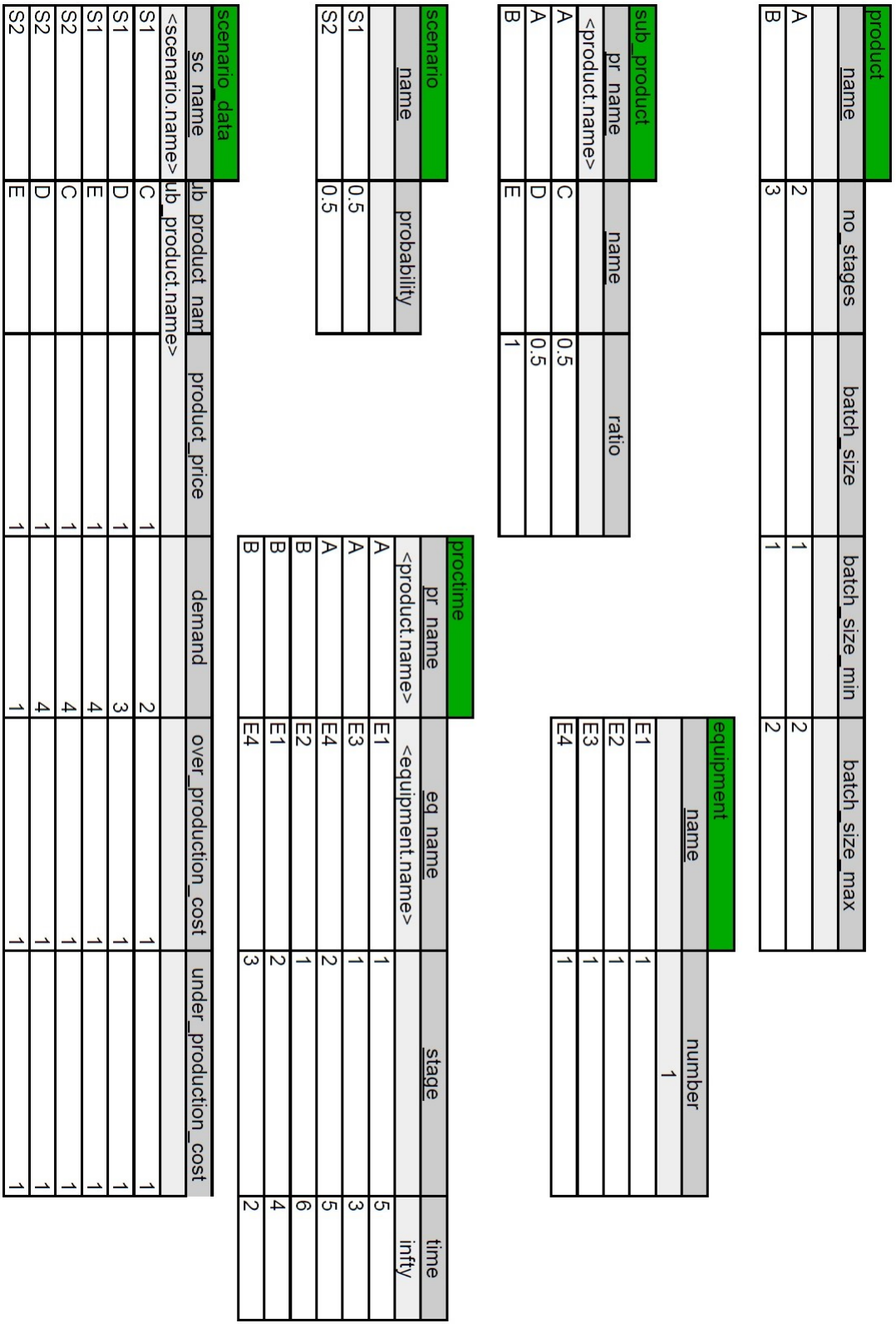
proctime

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

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

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

equipment	
name	number
	1
E1	1
E2	1
E3	1
E4	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	sub_product_name	product_price	demand	over_production_cost	under_production_cost
<scenario.name>	<sub_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	4	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
│       ├── Sztochasztikus
│       └── Sztochasztikus_Multiproduct
└── Solver kód
    ├── README.md
    ├── 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
```