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, [beadás dátuma] _____

hallgató

Kivonat

S-gráf alapú várható profit maximalizálás sztochasztikus környezetben

[1 oldalas, magyar nyelvű tartalmi kivonat]

Abstract

S-graph based expected profit maximization in stochastic environment

[1 oldalas, angol nyelvű kivonat]

Tartalomjegyzék

1. fejezet

Bevezetés

2. fejezet

Irodalmi áttekintés

2.1. Szakaszos gyártórendszerek ütemezése

2.2. Megoldó módszerek

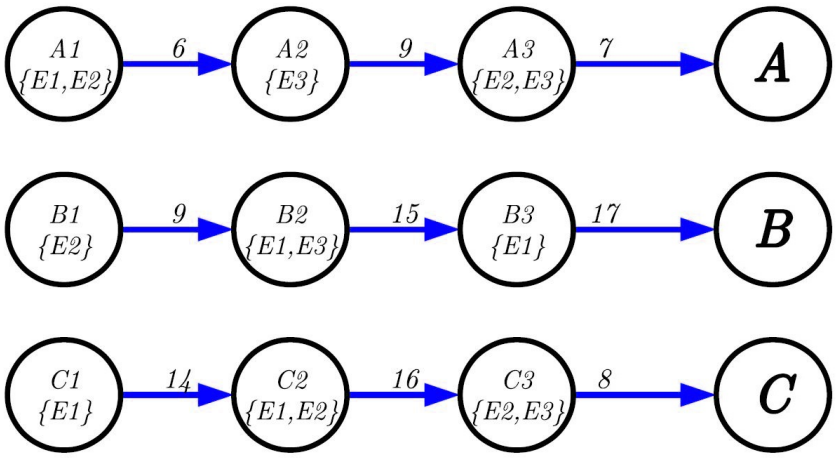
2.3. Sztochasztikus ütemezés

2.4. Az ütemtervek vizualizációja

3. fejezet

Az S-gráf keretrendszer

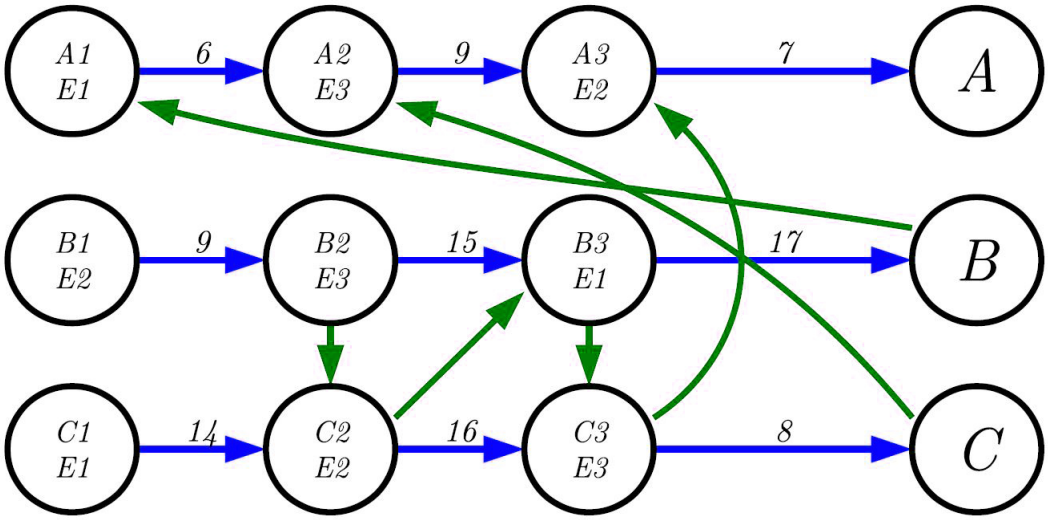
Az S-gráf keretrendszer volt az első publikált gráf elméleten alapuló módszer szakaszos gyártórendszerek ütemezési problémáinak megoldására. [5] A keretrendszer egy irányított gráf modellből, az S-gráfból és a hozzá tartozó algoritmusokból áll. [4] Az S-gráf egy speciális irányított gráf, amely nem csupán a probléma vizualizációjára képes, hanem egy matematikai modell is. A keretrendszerben a recepteket, valamint a félkész-, illetve a teljes ütemterveket is az S-gráf reprezentálja. Ezekben a gráfokban a termékeket, illetve a feladatokat a pontok jelölik, amelyeket csomópontoknak nevezünk. Ezenkívül, ha két feladat között összeköttetés van, ezt a gráfon a két feladatot reprezentáló csomópontok közötti nyíl jelöli. Az ütemezési információ nélküli S-gráfot recept gráfnak (**recipe graph**) nevezzük, melyre egy példa a 3.1 ábrán látható.



3.1. ábra. A recept gráf szemléltetése

Az ábrán látható jobb oldali három csomópont (A,B,C) jelöli a termékeket, a többi csomópont pedig a részfeladatokat, amelyeket el kell végezni a termékek előállítása érdekében. A recept gráf nyilai reprezentálják egyrészt két részfeladat közötti függőséget, abban ez esetben például, ha ez egyik részfeladat állítja elő a másik részfeladathoz szükséges bemeneti 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. A nyilakon található súlyok pedig a részfeladat megoldásához szükséges gyártási időt reprezentálják, abban az esetben, ha egy részfeladatot több berendezés is el tud végezni, a nyíl súlya a berendezésekhez tartozó gyártási idők közül a legkisebb lesz.

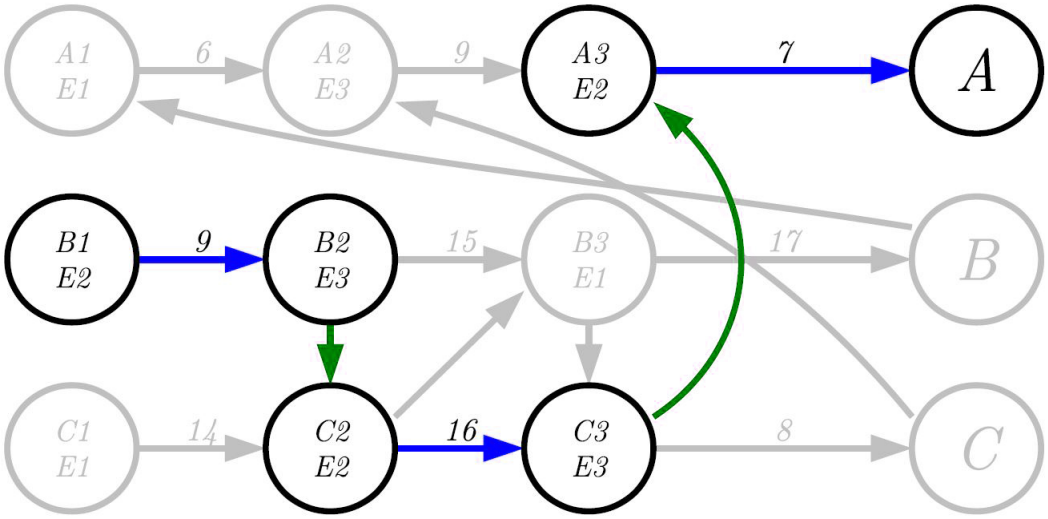
Az S-gráf keretrendszerben található algoritmusok az előzőekben bemutatott recept gráfot egészítik ki ütemezési nyilakkal, amelyek az algoritmus által meghozott ütemezési döntéseket reprezentálják. Az ily módon előállított gráfot, függetlenül attól, hogy van-e még meghozatlan ütemezési döntés, vagy pedig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



3.2. ábra. Az ütemezési gráf szemléltetése

Az ábrán látható gráfon már minden ütemezési döntés lezajlott, a zöld nyilak jelölik az ütemező algoritmus által behúzott ütemezési nyilakat. A részfeladatokat reprezentáló

csomópontokon immáron a lehetséges berendezések halmaza helyett egy konkrét berendezés jelölése található, amely az adott részfeladat elvégzésére hivatott az adott ütemterv szerint. Az ütemezési nyilak súlya alapértelmezetten 0, ha az adott problémában nem számolunk például részfeladatok közötti szállítási-, átállási-, illetve tisztítási időkkel. Az adott berendezéshez rendelt részfeladatok sorrendje könnyen leolvasható az ütemezési gráfról, erre egy példa a 3.3 ábrán látható.



3.3. ábra. Az E2 berendezéshez rendelt részfeladatok sorrendje

Az ábra alapján leolvasható, hogy az E2 berendezés először a B1 részfeladatot végzi el, majd a C2, azután pedig az A3 fog következni. Ahhoz azonban, hogy például a C2 részfeladatot E2 berendezés el tudja végezni, nem elegendő az, hogy B1 befejeződjön, elengedhetetlen az is, hogy minden részfeladat amitől C2 függ (nevezetesen C1) szintén véget érjen.

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

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

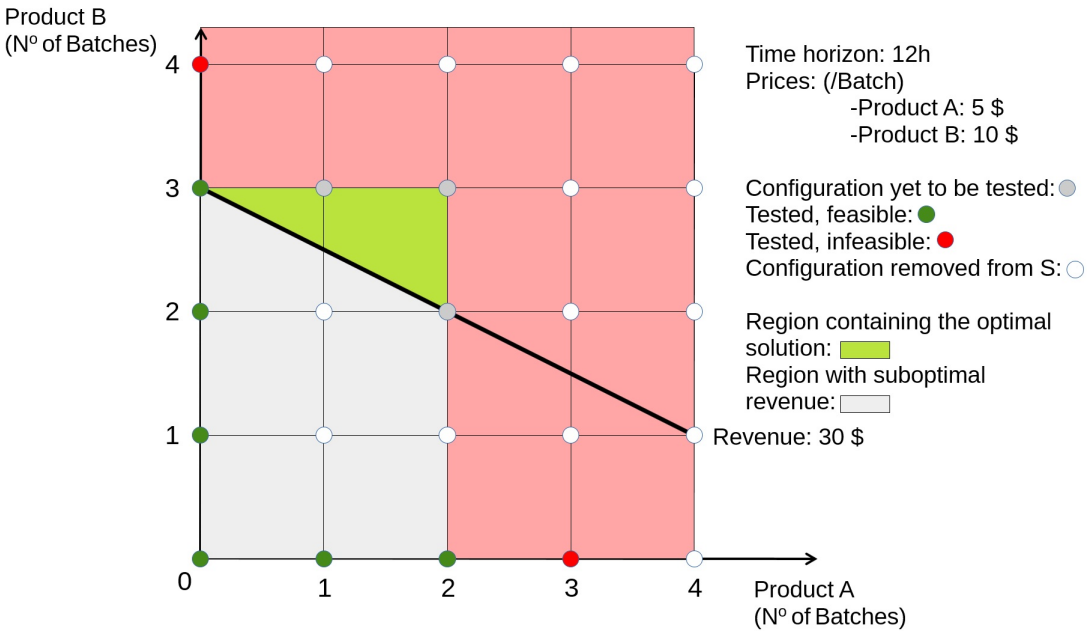
3.1. Profit maximalizálás az S-gráf keretrendszerrel

Az S-gráf keretrendszer eredetileg makespan minimalizációs célokra lett megalkotva, azonban később kibővítésre került egy throughput maximalizáló algoritmussal, amely segítségével immáron profit maximalizálásra is képes. Az throughput maximalizáló algoritmus alapötletét Majozi és Friedler [3], valamint Holczinger [2] fektették le, melynek lényege, hogy a termékek lehetséges batch darabszámai alapján különböző konfigurációk kerülnek létrehozásra[2], melyek között branch & bound algoritmus segítségével eredményül kapható a legnagyobb profitot eredményező konfiguráció, ha létezik megvalósítható (feasible) megoldás a problémára.

A throughput maximalizáló algoritmus működésének egy példán keresztül történő szemléltetésére készült folyamatábra a CD melléklet **Algoritmusok** mappájában található **DO_Sgraph_Throughput_Maximization.odp** néven. Kezdetben a konfigurációk halmaza tartalmazza az összes lehetséges konfigurációt az adott termékekre, majd minden iteráció során kiválasztásra kerül egy konfiguráció, amelynek teszteljük feasible-itását, azaz hogy a rendelkezésre álló időhorizont alatt megvalósítható-e adott konfiguráció legyártása. Ennek tesztelése a korábban már említett makespan minimalizáló algoritmus felhasználásával a legegyszerűbb. A makespan minimalizáló algoritmusnak átadásra kerül adott konfiguráció recept gráfja, majd az eredményül kapott idő érték összehasonlításra kerül a rendelkezésre álló időhorizonttal, ha a kapott érték nagyobb annál, az adott konfiguráció nem valósítható meg (infeasible). Ha az adott konfiguráció megvalósítható a rendelkezésre álló időhorizont alatt, kiszámításra kerül az adott konfiguráció által nyújtott profit, ha ez nagyobb az eddigi legjobb értéknél, frissíteni kell a legjobb értéket adott konfiguráció revenue értékével. Kezdetben a tengelyek menti konfigurációk kerülnek tesztelésre mind addig, még az összes tengelyen el nem jutunk az első megvalósíthatatlan (infeasible) konfigurációig. Ezzel előáll egy tér, amely tartalmazza az optimális megoldást, amelyet már csak meg kell találni, mivel azonban a konfigurációk tesztelése erőforrás igényes feladat, ezért a konfigurációk kiválasztásának sorrendje nem mindegy, többféle stratégia létezik az algoritmus gyorsítására. [1] Az egyik ilyen stratégia az úgynevezett revenue line behúzása, mely segítségével szerencsés esetben akár töredékére csökkenthető a

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

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



3.4. ábra. A revenue line szemléltetése

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

4. fejezet

Problémadefiníció

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

4.1. A probléma csoportosítása

A megoldandó problémák a sztochasztikus esetben is a determinisztikus throughput maximalizálásnál használt paraméterekkel adottak, pl.: minden terméket a receptje azonosít be, ezen

kívül adott a termékek előállítására használható berendezések halmaza, illetve a termelésre rendelkezésre álló időhorizont. Az determinisztikus paramétereken kívül azonban sztochasztikus esetben különböző bizonytalanságot kifejező paraméterek is adottak minden termékre, amelyek valószínűségeit különböző scenariokba, forgatókönyvekbe csoportosítjuk. Ezáltal minden forgatókönyvre adott:

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

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

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

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

4.2. A probléma paraméterei

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

P a termékek halmaza

b_p a legyártott batch-ek darabszáma az adott konfigurációban

s_p a termék batch mérete (kötött batch méret esetén)

s_p^{min}, s_p^{max} adott termékhez tartozó lehetséges legkisebb, legnagyobb batch méret (változó batch méret esetén)

S a forgatókönyvek halmaza

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

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

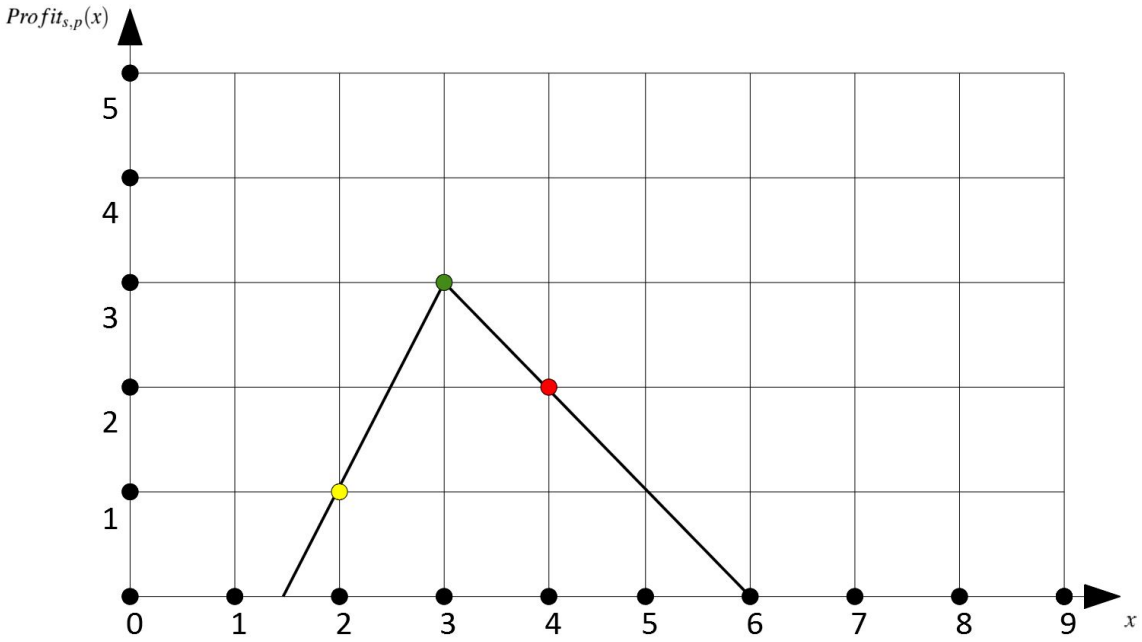
$price_{s,p}$ p termék ára az s forgatókönyvben $s \in S, p \in P$

$oc_{s,p}, uc_{s,p}$ p termék túl-, és alul termelési költsége s forgatókönyvben $s \in S, p \in P$

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

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

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



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

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

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

[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.

malizálásáról beszélhetünk, mintsem profit maximalizálásról, azonban könnyen belátható, hogy a definiált matematikai modellekben amelyeket használunk a profit kiszámítására, ez semmiféle változást nem eredményez, csupán arra kell figyelni, hogy az implementáció során felkészüljünk a negatív számok a programnyelvben történő kezelésére.

5. fejezet

Az S-gráf solver

5.1. A solver felépítése

5.2. A determinisztikus throughput maximalizáló

6. fejezet

A probléma megvalósítása

6.1. A felhasznált módszerek

A problémák megoldásához az S-gráf keretrendszerben már kidolgozásra kerültek elméleti algoritmusok [1]. Szakdolgozati munkám célja ezen elméleti algoritmusok tanulmányozása, részleteinek kidolgozása, valamint az S-gráf megoldó keretrendszerbe történő integrálása, implementálása, valamint tesztelése.

6.1.1. Preventív ütemezés kötött batch mérettel

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

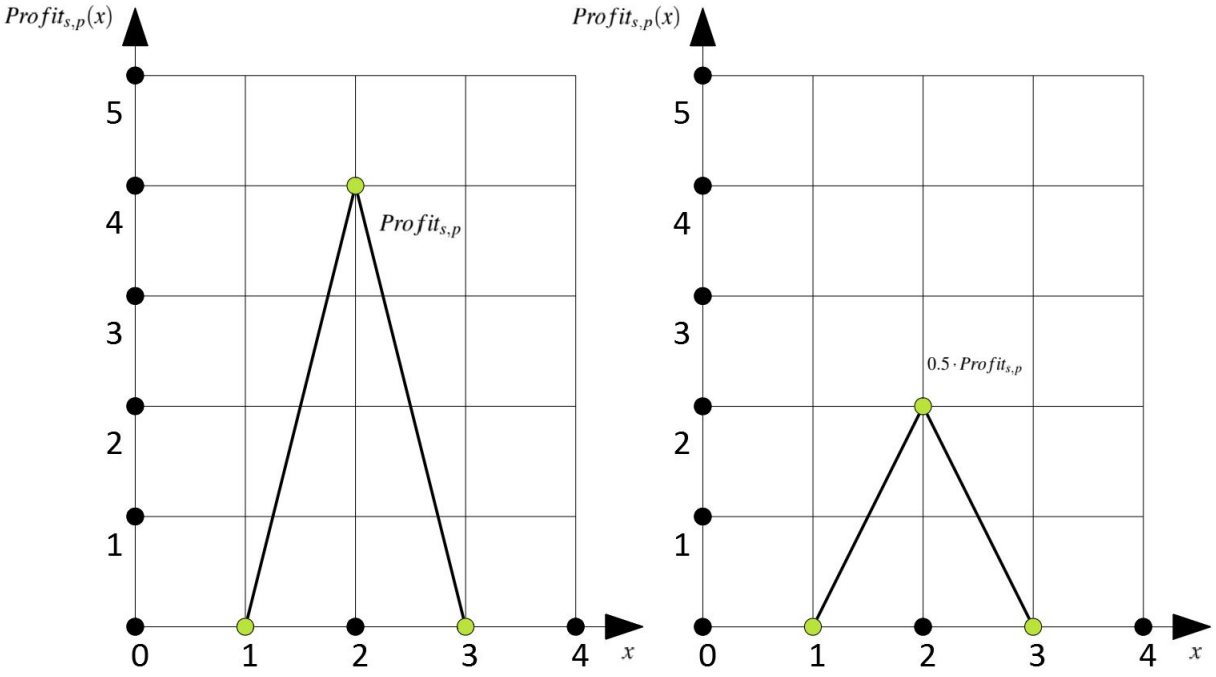
$$\sum_{p \in P} \left(\sum_{s \in S} prob_s \cdot profit_{s,p}(s_p \cdot b_p) \right)$$

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

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

$ExpProfit_p(x)$ kiszámításához tehát nincs másra szükségünk, mint hogy az összes forgatókönyvre sorban felépítsünk az adott forgatókönyvre vonatkozó sztochasztikus paraméterekből a $profit_{s,p}$ függvényt, majd ezt a függvény beszorozzuk az aktuális $prob_s$ értékkel, amely lényegében a függvény ”összenyomását” jelenti. Miután minden forgatókönyvre

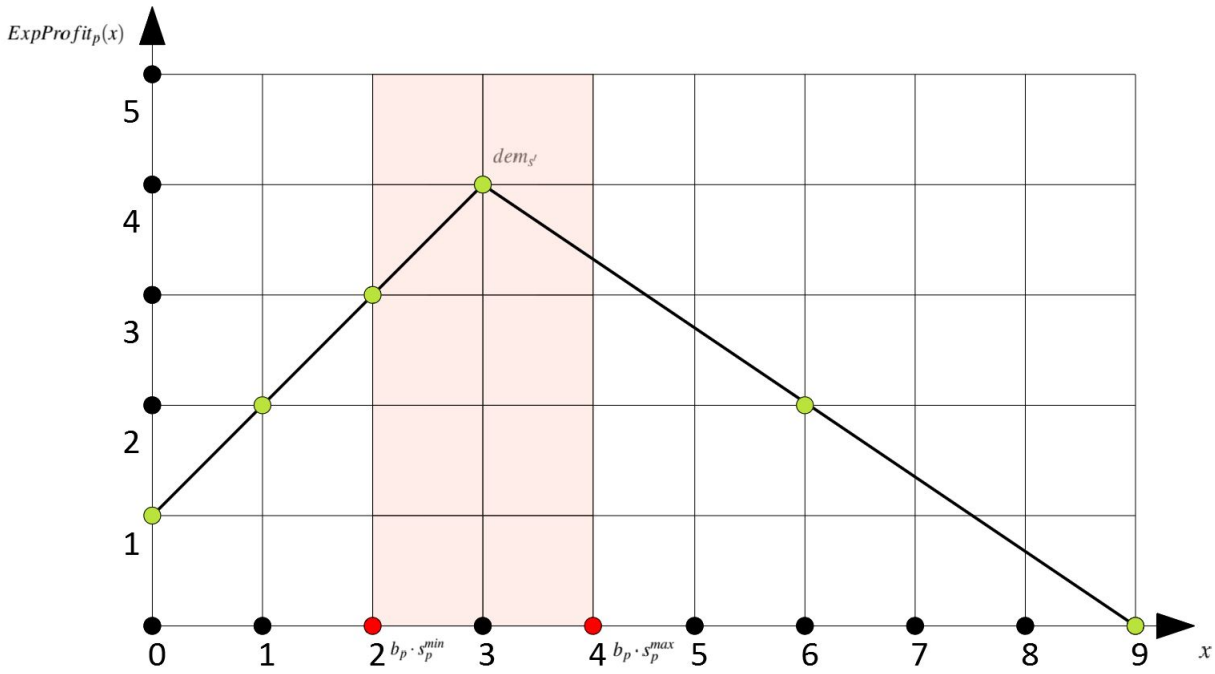
előállítottuk a 6.1 ábrához hasonlóan ezt az ”összenyomott” profit függvényt, ezen függvények összeadásával előáll az $ExpProfit_p$, ha ezt minden termékre megtesszük, az adott p termékek $ExpProfit_p(x)$ (ahol $x = s_p \cdot b_p$) értékének összegeként előáll a várható profit.



6.1. ábra. A profit függvény szorzásának szemléltetése

6.1.2. Preventív ütemezés változó batch mérettel

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



6.2. ábra. Az optimális x_p érték kiválasztásának szemléltetése

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

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

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

6.1.3. Két lépcsős ütemezés (two stage)

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

kiválasztása a következőképpen zajlik:

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

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

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

6.1.4. Következtetés

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

6.2. A PiecewiseLinearFunction osztály

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

hiszen folytonos, lineáris függvényekről beszélünk. Ez a három pont a $profit_{s,p}$ függvények esetében nem más, mint:

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

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

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

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

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

6.3. ábra. A *PiecewiseLinearFunction* osztály adattagjai

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

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

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

6.2.1. Koordináta pár hozzáadása

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

6.2.2. Kezdeti-, és vég meredekség kiszámítása

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

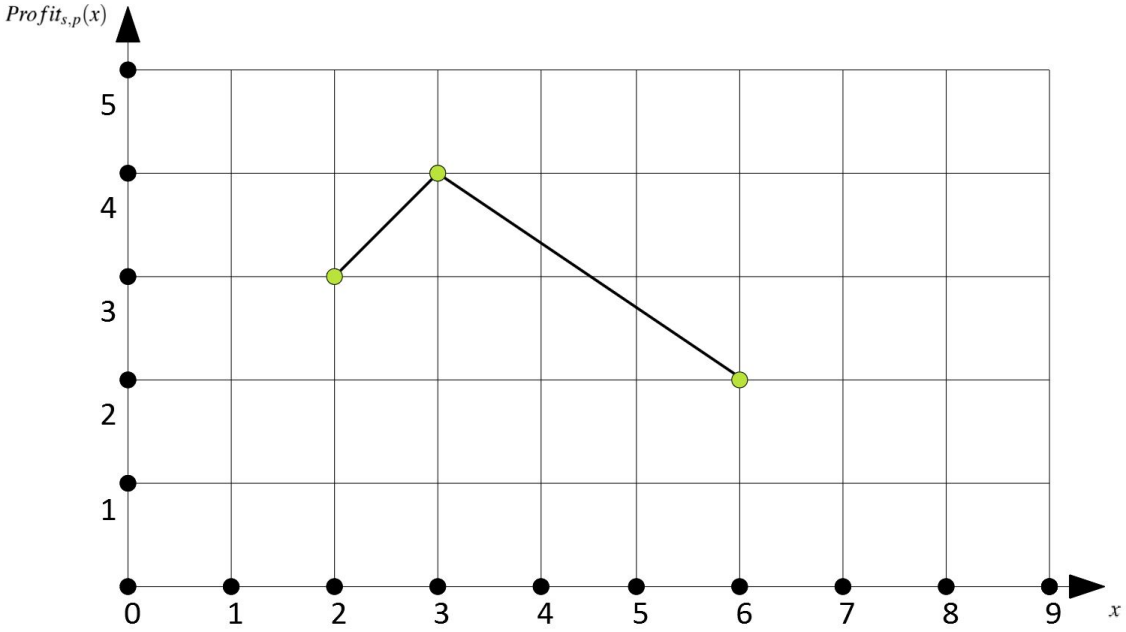
$$Angle_{begin} = (y_{second} - y_{first})/(x_{second} - x_{first})$$

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

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

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

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



6.4. ábra. Példa a meredekségek kiszámítására

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

6.2.3. A függvény skalár értékkel való szorzása

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

6.2.4. A függvény x helyen vett értékének lekérdezése

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

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

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

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

6.2.5. Két függvény összeadása

Két függvény összeadását az összeadás operátort felültöltő metódus végzi, melynek visszatérési értéke az összegként kapott függvényt tároló új **PiecewiseLinearFunction** objektum. Mivel a

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

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

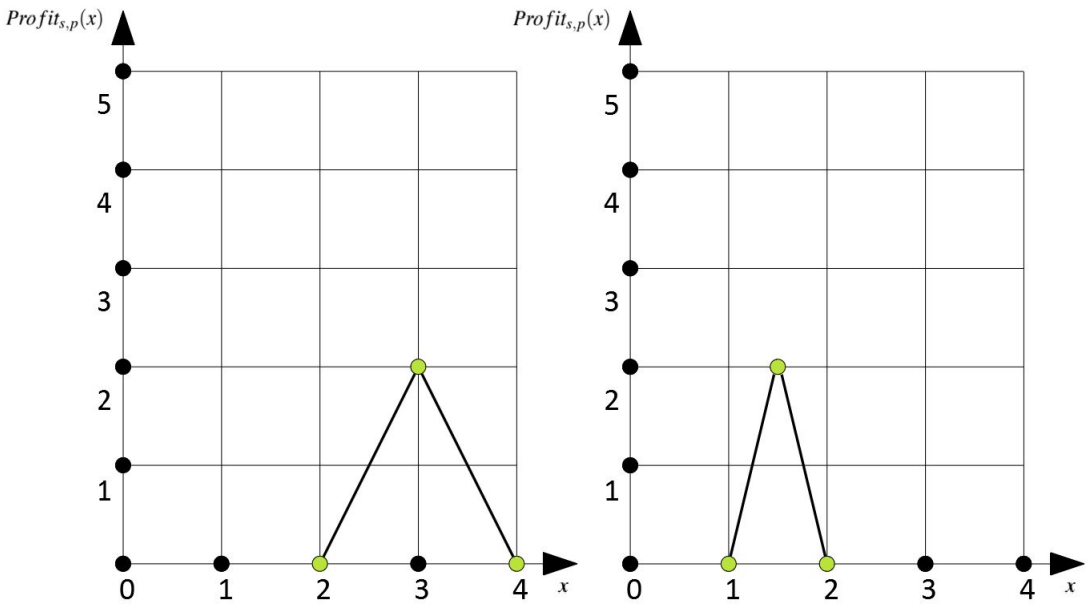
6.5. ábra. Az összeadást végző metódus

6.2.6. A függvény horizontális nyújtása

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

6.2.7. A függvény maximális értékéhez tartozó koordináták lekérdezése

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



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

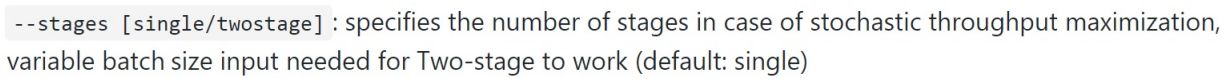
tatott változó batch méretű esetben van nagy jelentősége, ezt használjuk ugyanis $dem_{s'}$ meghatározására.

6.3. Az új paraméterek implementációja

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

6.3.1. Új kapcsoló definiálása

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



6.7. ábra. Az új kapcsoló leírása a readme fájlban

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

6.3.2. Új input fájl definiálása

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

6.3.3. A fájl beolvasása, beolvasott paraméterek tárolása

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

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

6.4. Szükséges változtatások az S-gráf keretrendszerben

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

6.4.1. A meglévő kód refaktorálása

A **Throughputsolver** osztály jelenlegi állapotában, determinisztikus esetben kétféleképpen számítja ki a profitot, egyrészt az **SGraph** osztály **GetRevenue()** metódusát használva, másrészt az egyes termékek adatait az **SGraph** osztály **Recipe** objektumában letárolt **Product** objektu-

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

6.4.2. A *GetRevenue* metódus

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

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

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

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

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

A **GetRevenue()** metódusnak azonban egy másik, a sztochasztikus esetekkel kapcsolatos problémát is orvosolnia kell. Ez a probléma a úgynevezett "axial revenue", azaz a tengelyeken számított várható profit értékével kapcsolatos. Axial revenue-ról akkor beszélünk, ha az adott konfigurációban csupán egy fajta terméket gyártunk, a többi termékből (ha léteznek) ez esetben 0 darabot termelünk. Ez a sztochasztikus esetekben azért okoz problémát, mert a nem gyártott termékek esetleges alul termelési költségeit le kell vonni az összes profitból, így tehát hiába gyártunk csak egy terméket, a többi termék paraméteri is befolyásolják a várható profitot. Az

alul termelésből eredendő költségek számítása alapvetően nem okozna problémát, hiszen csak ki kellene számolni adott termékek $x = 0$ helyen vett $ExpProfit_p(x)$ értékét, és összegezni a kapott értékeket. Mivel azonban ebben az esetben csak egy fajta terméket termelünk, ezért a konfigurációban csak a termelt termék adatai találhatóak meg, ezért a többi termék $ExpProfit_p(x)$ értékének számítása jelen helyzetben lehetetlen. Erre azonban megoldást nyújt, ha a korábban említett módon, az inputfájl beolvasását követően a **RelationalProblemReader** osztályban nem csak a kezdeti termékek neveit tároljuk el, hanem azok $x = 0$ helyen vett $ExpProfit_p(x)$ értékeit is (felhasználva a 6.4.3 pontban bemutatott **GetProductRevenue metódust**). Ezen értékek tudatában a nem termelt termékekből származó esetleges veszteség immáron levonható az várható profit értékéből, az axial revenue hibátlanul megkapható.

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

6.4.3. A *GetProductRevenue* metódus

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

procedure GETPRODUCTREVENUE

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

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

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

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

return 0

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

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

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

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

6.4.4. A *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 6.9 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
```

6.9. 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 6.10, és a 6.11 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
```

6.10. 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
```

6.11. 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. [6] 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 6.12 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;
```

6.12. 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 6.13 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();
}
```

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

6.5. Multiproduct receptek esete

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

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

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

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

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

R a receptek halmaza

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

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

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

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

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. [1] Az inputfájl definiálásával, illetve a solveren végzett munkámmal megteremtettem ezen eset megoldásának feltételeit is, azonban ennek implementációja túlmutat jelen dolgozat munkáján.

7. fejezet

Teszteredmények

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

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

7.1. A determinisztikus throughput maximalizáló tesztelése

7.2. A sztochasztikus alapesetek tesztelése (1-1)

7.3. A sztochasztikus multiproduct receptek tesztelése

8. fejezet

Összefoglalás

Irodalomjegyzék

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

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)

s_p^{min}, s_p^{max} adott termékhez tartozó lehetséges legkisebb, legnagyobb batch méret (változó batch méret esetén)

S a forgatókönyvek halmaza

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

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

$price_{s,p}$ p termék ára az s forgatókönyvben $s \in S, p \in P$

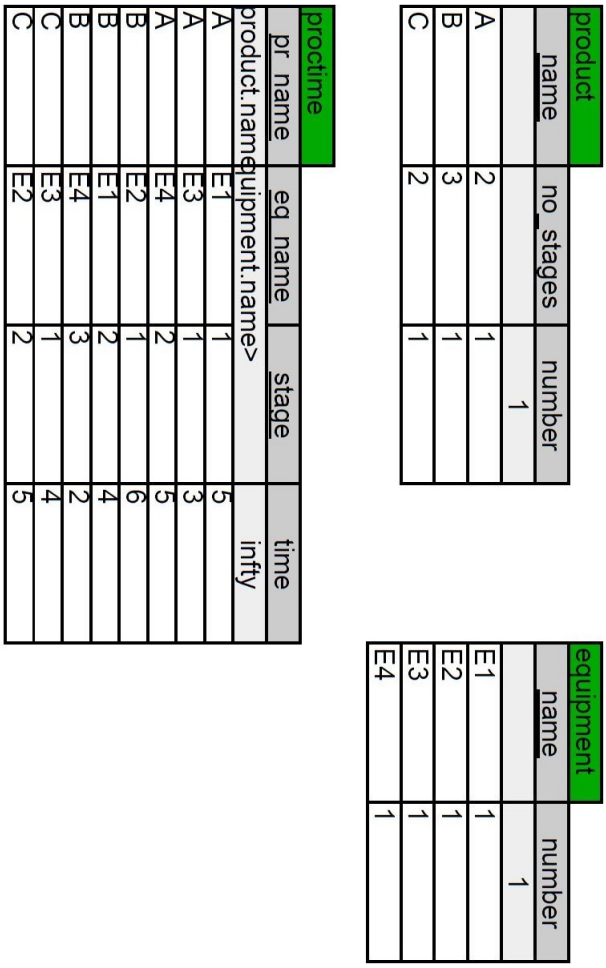
$oc_{s,p}, uc_{s,p}$ p termék túl-, és alul termelési költsége s forgatókönyvben $s \in S, p \in P$

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

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

equipment

name	number
	1
E1	1
E2	1
E3	1
E4	1

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

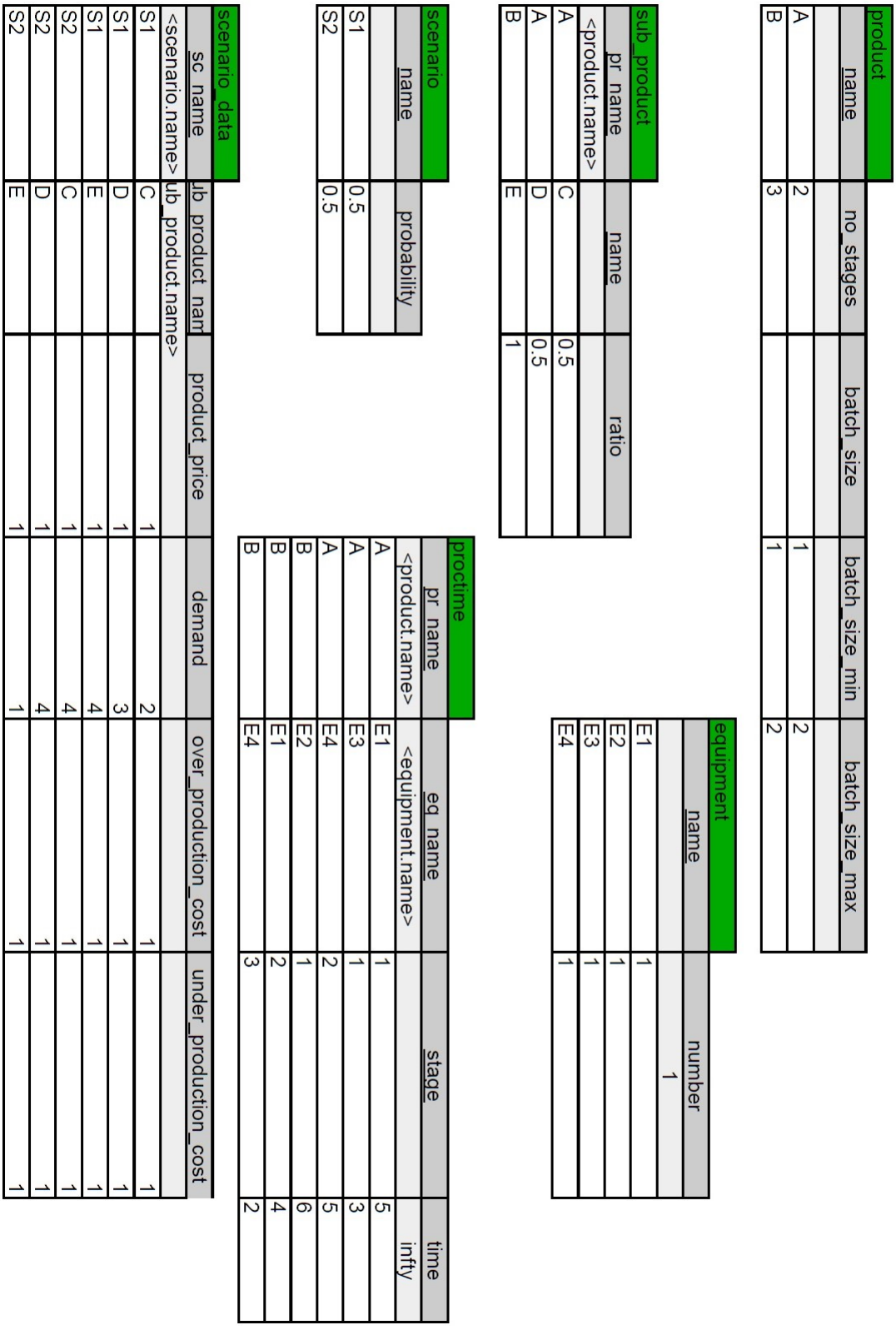
proctime

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	2
B	3	1	2	2

sub_product		
pr_name	name	ratio
<product.name>	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