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. Bevezetés **1**

2. Irodalmi áttekintés **2**

3. Az S-gráf modell **3**

3.1. Profit maximalizálás S-gráffal 3

4. Problémadefiníció **4**

4.1. A problémák csoportosítása 4

4.2. A problémák matematikai modelljei 6

4.2.1. Preventív ütemezés fix batch mérettel 8

4.2.2. Preventív ütemezés változó batch mérettel 9

4.2.3. Two stage (két lépcsős ütemezés) 10

4.2.4. Következtetés 11

5. Az S-gráf keretrendszer **12**

6. A probléma megvalósítása **13**

6.1. A BrokenLine osztály 13

6.1.1. Koordináta pár hozzáadása 15

6.1.2. Kezdeti-, és vég meredekség kiszámítása 15

6.1.3. A függvény skalár értékkel való szorzása 16

6.1.4. A függvény x helyen vett értékének lekérdezése 16

6.1.5. Két függvény összeadása 17

6.1.6. A függvény horizontális nyújtása 18
6.1.7. A függvény maximális értékéhez tartozó koordináták lekérdezése . . . 18
6.2. Szükséges változtatások az általános throughput maximalizálón 18
6.3. Multiproduct receptek esete 18

7. Teszteredmények **19**

8. Összefoglalás **20**

1. fejezet

Bevezetés

2. fejezet

Irodalmi áttekintés

3. fejezet

Az S-gráf modell

3.1. Profit maximalizálás S-gráffal

4. fejezet

Problémadefiníció

A probléma lényege abban keresendő, hogy a korábban kidolgozott általános throughput maximalizáló algoritmus [**?**] valódi ipari környezetben nem minden esetben állja meg a helyét, ugyanis sok esetben a a probléma megoldásához használt paraméterek nem determinisztikusak. 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. Szakdolgozatom célja a 4.2 pontban bemutatott, Hegyháti által kidolgozott [**?**] matematikai modellek S-gráf keretrendszerbe történő implementálása, oly módon, hogy az általános throughput maximalizáló algoritmus sértetlen maradjon, a probléma típusától függően kompatibilis használat lehetséges legyen.

4.1. A problémák csoportosítása

A megoldandó problémák a sztochasztikus esetben is az általános throughput maximalizálásnál használthoz hasonló 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 általános paramétereken kívül azonban sztochasztikus esetben különböző bizonytalan paraméterek is adottak minden termékre, amelyek valószínűségeit

különböző scenariokba, forgatókönyvekbe csoportosítjuk. Ezáltal minden forgatókönyvre adott:

- A forgatókönyv valószínűsége
- A termék ára (1 batch ára)
- A termék iránti kereslet
- A túltermelés költsége
- Az alul termelés költsége

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 fix batch mérettel** Ebben az esetben minden termékhez adott egy batch méret, az egyetlen preventív döntés amit hoznunk kell, hogy hány darab batch-t 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.
- **Two stage (kép lépcsős ütemezés)** 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 reláció van, azaz egy recept sem eredményez több terméket, illetve egyetlen termék sem állítható elő több fajta recepttel. A 6.3 pontban azonban kitérek azokra az esetekre, amelyekben ez a feltételezés nem állja meg a helyét.

4.2. A problémák matematikai modelljei

A 4.1 pontban bevezetett sztochasztikus esetek kezeléséhez az általános throughput maximalizáló algoritmus jelentős része felhasználható változtatások nélkül (vagy csak minimális változtatások árán, lsd. 6.2 pont). Az egyetlen meghatározó különbség az un. ”revenue” függvényben figyelhető meg, amely célja, hogy az adott konfigurációra nézve kiszámítsa a várható profitot. A revenue függvény működésének leírásához szükséges néhány jelölés bevezetése:

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 $s \in S, p \in P$

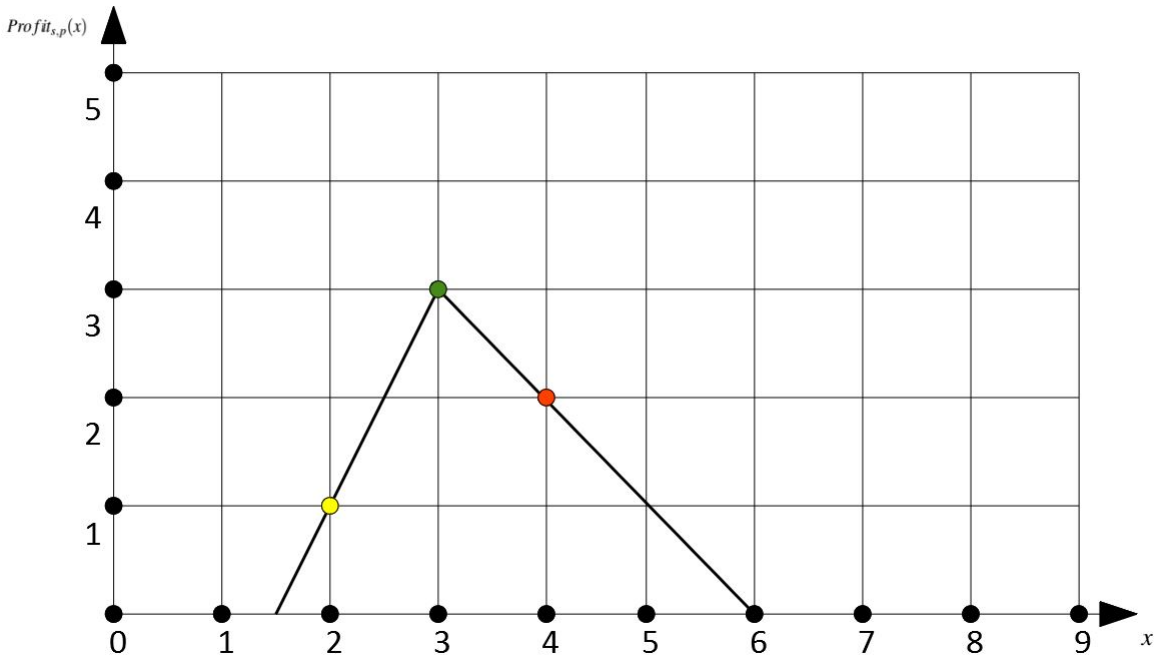
$oc_{s,p}, uc_{s,p}$ p termék túl-, és alul termelési költsége s forgatókönyvben $s \in S, p \in P$

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

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

A 4.1 ábra a profit függvény szemléltetését szolgálja, a következő paraméterekkel:

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



4.1. ábra. A profit függvény szemléltetése

Nyilvánvalóan a bevétel akkor lesz maximális, ha a kereslettel egyező darabszámot gyártunk az adott termékből (zöld pont az 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 az 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 az ábrán). 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 minimalizálásáról beszélhetünk, mintsem profit maximalizálásról,

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

4.2.1. Preventív ütemezés fix batch mérettel

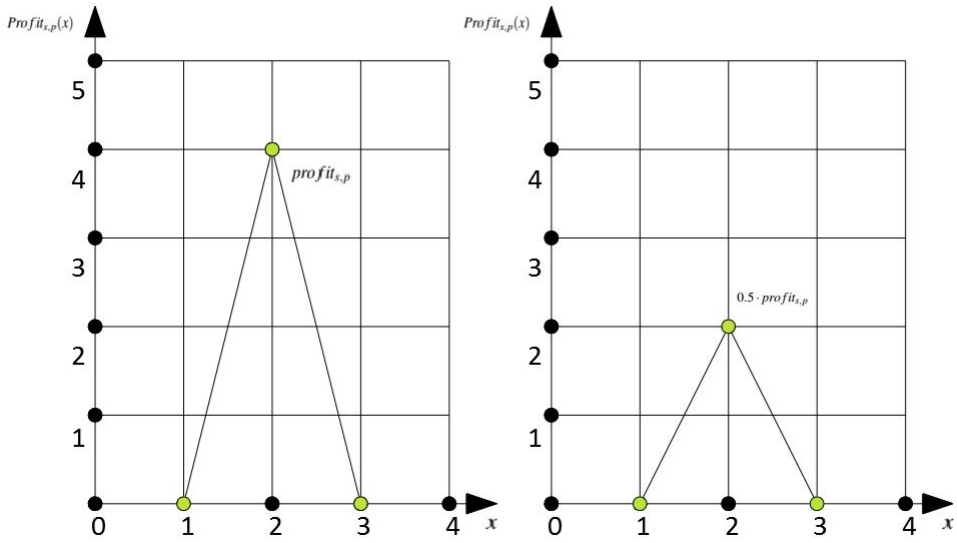
Ebben az esetben az egyetlen döntés, amit meg kell hozni, hogy az egyes termékekből hány darab batch-et gyártsunk, a várható profit könnyen kiszámítható:

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

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

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

Fontos megjegyezni, hogy az egyes termékek várható profitja egymástól független, hiszen nem osztoznak közös recepteken, éppen ezért, ha két különböző konfigurációban adott termékből gyártott batch darabszám megegyezik, akkor ha megnöveljük, vagy csökkentjük ezt a számot, mindkét konfiguráció várható profit értékében ugyan olyan változás fog történni, attól függetlenül, hogy más termékekből adott konfigurációban mennyit gyártunk. Ezenkívül tudjuk azt is, hogy az $ExpProfit$ függvény értéke sosem fog nőni, ha már egyszer elkezdett csökkenni, mert $ExpProfit$ egy folytonos, szakaszos, lineáris függvény. [**?**] $ExpProfit_p(x)$ 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 4.2 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.



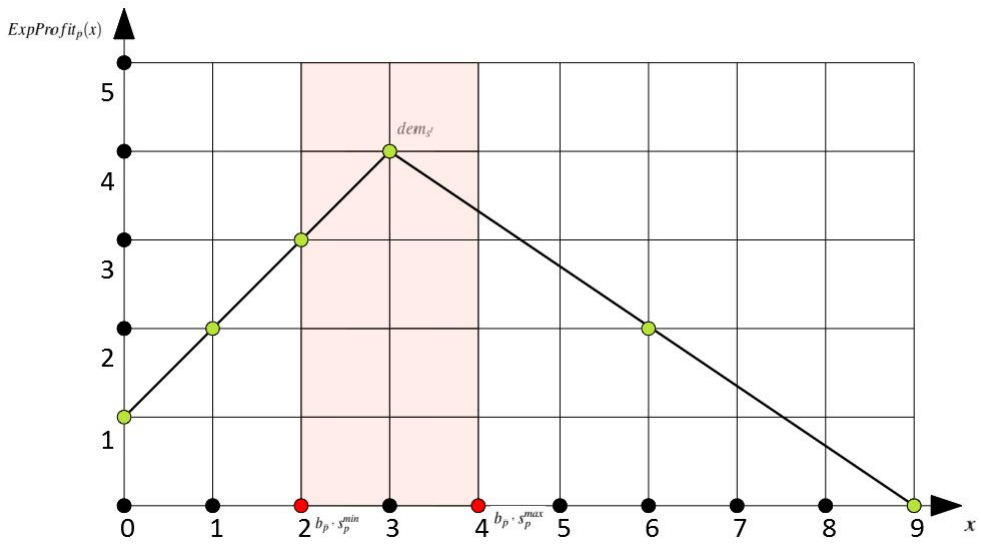
4.2. ábra. A profit függvény szorzásának szemléltetése

4.2.2. 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 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 4.3 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.



4.3. ábra. Az optimális x_p érték kiválasztásának szemléltetése

4.2.3. Two stage (két lépcsős ütemezés)

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 4.2.2 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:

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

4.2.4. Következtetés

A matematikai modellek ismerete arra enged következtetni, hogy a probléma az 5. fejezetben bemutatott S-gráf keretrendszerbe történő integrálásához elengedhetetlen lesz egy olyan osztály definiálása, amely képes folytonos, szakaszos, lineáris függvények modellezésére, tárolására, azokon történő műveletek végrehajtására. Ezen osztály részletes leírása a 6.1 pontban olvasható.

5. fejezet

Az S-gráf keretrendszer

6. fejezet

A probléma megvalósítása

6.1. A BrokenLine osztály

Ahogy az már korábban, a 4.2.4 pontban említésre került, a probléma implementációjához elengedhetetlen egy olyan osztály definiálása, amely kezelni képes folytonos, szakaszos, lineáris függvényeket. Erre hivatott az általam megalkotott *BrokenLine* osztály, amelynek forráskódja *brokenline.h*, illetve *brokenline.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 4.2.2 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 *BrokenLine* osztály adattagjai a következőek:

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

6.1. ábra. A *BrokenLine* osztály adattagjai

A 6.1 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 *BrokenLine* 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 6.3 pontban tárgyalt esetekhez)
- A függvény maximális értékéhez tartozó koordináták lekérdezése

6.1.1. 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.

6.1.2. 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 6.1.1 pontban említett két *bool* változó, amelyek megadják, hogy kell e frissíteni adott meredekségeket. A meredekségek kiszámítása roppant egyszerű a koordináták ismeretében:

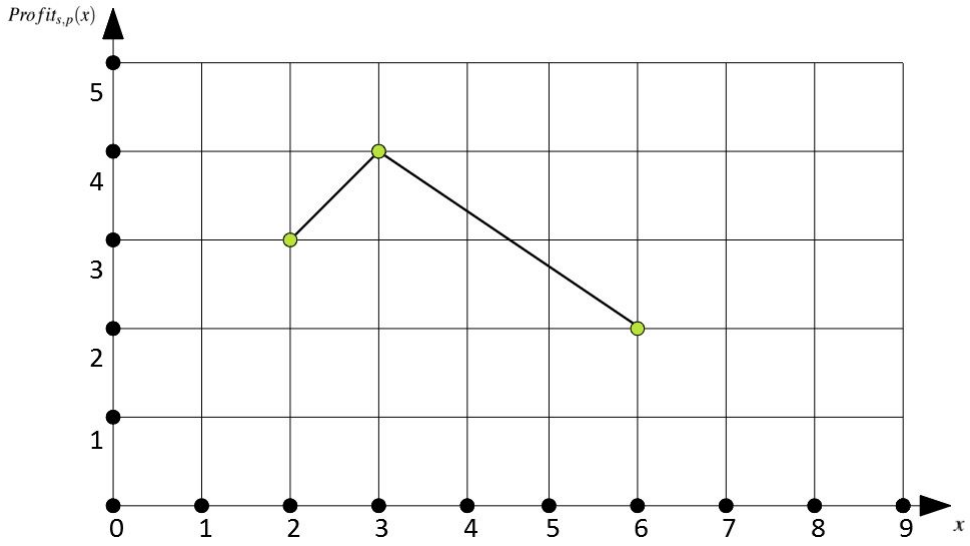
$$Angle_{begin} = (y_{second} - y_{first})/(x_{second} - x_{first})$$

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

A 6.2 ábrán látható három pont koordinátái alapján a meredekségek például a következőképpen alakulnak:

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

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



6.2. ábra. Példa a meredekségek kiszámítására

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

6.1.3. A függvény skalár értékkel való szorzása

A függvény skalárral való szorzását a *multiplyByScalar(double s)* 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 4.2 ábrán látható.

6.1.4. A függvény *x* helyen vett értékének lekérdezése

A függvény *x* helyen vett értékének lekérdezésére a *getValue(double x)* 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.

6.1.5. Két függvény összeadása

Két függvény összeadását az *addBrokenLine(BrokenLine b)* metódus végzi, melynek visszatérési értéke az összegként kapott függvényt tároló új *BrokenLine* objektum. Mivel a függvények tárolásához elegendő három pont tárolása, ezért gyakran előfordul olyan eset, hogy a két összeadni kívánt *BrokenLine* objektum nem tartalmazza a szükséges koordinátákat, ezért a hiányzó koordináta párokat először hozzá kell adni, ezt azonban megkönnyíti a 6.1.4 pontban bemutatott *getValue* 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.

```
BrokenLine BrokenLine::addBrokenLine(BrokenLine b){
        BrokenLine c = BrokenLine();
        std::vector<Coordinate> coordinates_b=b.getCoordinates();
        for(int i=0;i<coordinates_b.size();i++){
            this->getValue(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.getValue(current.getX()).getY());
            c.addCoordinate(current);
        }
        return c;
}
```

6.3. ábra. Az *AddBrokenLine* metódus

6.1.6. A függvény horizontális nyújtása

6.1.7. A függvény maximális értékéhez tartozó koordináták lekérdezése

6.2. Szükséges változtatások az általános throughput maximalizálón

6.3. Multiproduct receptek esete

7. fejezet

Teszteredmények

8. fejezet

Összefoglalás