Budapesti Műszaki- és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Elektronikai Technológia Tanszék

Akers 1956-os algoritmusának kiterjesztése három feladatos problémákra

TDK dolgozat

Készítette: Ponkházi András

Neptun kód: FG61Y0

Konzulens: Villányi Balázs

TARTALOMJEGYZÉK

Bevezetés 3

1. Termelésütemezés 4

1.1. Ütemezési probléma 4

1.1.1. Gyártási idő, mint célfüggvény: 5

1.1.2. Szalagrendszerű gyártás (Flow Shop) 5

1.1.3. Műhelyrendszerű gyártás (Job Shop) 7

2. Akers (1956) módszer bemutatása 9

2.1. Algoritmus lépései 9

2.2. Az Akers módszer továbbfejlesztésének lehetősége 11

3. Akers módszer kiterjesztése három feladatra 13

3.1. Az algoritmus bemutatása 13

3.2. Az algoritmus lépései 15

3.3. Az algoritmus pszeudokódja 17

3.4. Algoritmus példa futás 19

3.5. A kiterjesztés visszavezetése az Akers módszerre 26

4. A két módszer hasonlósága 28

4.1. A két módszer összehasonlítása teszteléssel 28

4.1.1. Akers módszer átlagos növekménye 28

4.1.2. A növekmények összegzése 31

4.1.3 Kiterjesztés átlagos növekménye 32

4.1.4. A növekmények összegzése 34

4.2. Növekmények összehasonlítása/összevetése 35

5. Összegzés 36

5.1. Továbbfejlesztési lehetőségek 36

Irodalomjegyzék 37

Ábrajegyzék 38

BEVEZETÉS

A dolgozatom témája Akers módszerének, illetve ennek az algoritmusnak az általam kitalált kiterjesztésének bemutatása. Ezen algoritmusok speciális műhelyrendszerű ütemezésre adnak optimális megoldást.

Akers módszerével a Termelésinformatika című órámon találkoztam, ahol nagyon megtetszett a termelésütemezés problémája és komplexitása. Akers módszerével az Önálló laboratórium című órámon foglalkoztam komolyabban, és ott tanulmányoztam, a továbbfejlesztési lehetőségeit. Az órám teljesítését követően fogtam bele a kiterjesztés kidolgozásába, és megvalósításába.

Akers módszere 2 feladatos ’m’ gépes problémákat old meg, míg a kiterjesztés, amit kitaláltam 3 feladatos ’m’ gépes problémákra is működik, így ez egy kiterjesztése Akers 1956-ban publikált módszerének. Első ránézésre Akers módszerét könnyen ki lehet terjeszteni 3 dimenzióba, azonban mint lentebb taglalom, ütköztem problémákba, amiket végül áthidaltam újabb szabályok bevezetésével, és ezek betartásával. Dolgozatomban az algoritmusok ismertetése előtt bemutatom a termelésütemezés feladatát, és főbb eljárásait, céljait, fontosságát, továbbá, bemutatom a szalagrendszerű és a műhelyrendszerű ütemezéseket, és a köztük lévő hasonlóságokat, és különbségeket. Illetőleg Akers módszerének kapcsolatát az imént említett ütemezésekkel. Akers módszerének bemutatása után teszek javaslatot a továbbfejlesztésére és itt taglalom milyen problémába ütköztem, és, hogy további szabályok bevezetése szükséges a megvalósításhoz, hogy egy valós, ütemező algoritmust kapjunk. A kiterjesztés bemutatása után, szemléltetem a lépéseit, illetőleg egy példa keretein belül bemutatom a pontos működését. Összehasonlítom Akers módszerével, mégpedig, hogy az alapgondolat az megegyezik mind a két módszernél, és, hogy a fő cél sem változott csak a feladatok és szabályok száma nőtt, illetőleg több példán és tesztelésen keresztül szemléltetem az algoritmus helyességét.

A Dolgozat végén javaslatot teszek, még egyéb továbbfejlesztési lehetőségre a kiterjesztést illetően, külön tekintettel a feladatok növelésének problémájára, illetőleg a feladatok priorizálásának lehetőségére is. A termelésütemezésben más és más is lehet a cél, nem csak a feladatok számának növelése vagy a végrehajtási idő csökkentése, hanem akár a feladatok priorizálása.

1. TERMELÉSÜTEMEZÉS

A termelésütemezés az erőforrások hatékony időbeli elosztását érinti, az áruk, termékek előállítása során. Ütemezési problémákban akkor ütközhetünk, amikor egy közös erőforrást – ez lehet közös gép/eszköz/ember – kell használni más-más termékek előállítására, vagyis nem tud minden termék egyszerre haladni, várakozniuk kell egymásra az erőforrás hiány miatt. Az ütemezésnek az a célja, hogy megtaláljuk a módját a közös erőforrások használatára, kiosztására és sorrendjére, bizonyos teljesítménykritériumok teljesítése érdekében.[1] A cél az általában a költségek minimalizálása, de sok fajta ütemezés van, és sok célfüggvény is, amit lehet elsődlegesnek tekinteni és az alapján sorba állítani a termékeket, feladatokat, gépeket, erőforrásokat stb.. A jobb termelés ütemezés az erőforrások termelékenységének és a hozzájuk kapcsolódó hatékonyságnak a növelése révén versenyelőnyt biztosít a versenytársakkal szemben.[2]

A termelés ütemezésével foglalkozó tudományos cikkek/kutatások már a XX. század közepén megjelentek. Ebben az időben fogalmazták még a problémát is, és készült az általam kiterjesztett algoritmus is. Mindenki arra törekedett akkoriban, hogy megtalálja az optimális megoldást adó algoritmust. Születtek is optimális megoldást adó algoritmusok, de mindegyik csak speciális esetekben volt alkalmazható, ilyen például a Jackson módszer, ami $m = 2$ paraméterű problémákra ad optimális megoldást, Akers aki a feladatok számát határozta meg kettőben és úgy adja meg az optimális ütemezést, illetőleg Johnson, aki az $m = 2$ paraméterű szalagrendszerű ütemezésre ad helyes ütemezést. Ez így visszatekintve nem meglepő, hogy csak speciális módszerekre sikerült olyan algoritmust mutatni, ami optimális megoldást ad, hiszen 1976-ban M. R. Garey bebizonyította, hogy azok a műhelyrendszerű ütemezések ahol $m > 2$ (tehát több mint 2 gépes az ütemezés), mind NP nehéz problémák, tehát nem is létezik optimális megoldást adó algoritmus feltéve ha $P \neq NP$.[3]

1.1. Ütemezési probléma

Az ütemezési probléma tipikusan egy sor elvégzendő feladatot tartalmaz, ahol minden egyes feladat az elvégzendő műveletek egy halmazából áll. A műveletekhez általában gépekre és anyagi erőforrásra van szükség, és azokat megfelelő technológiai sorrendben kell

[1] S. C., G. (1981). A review of production scheduling. Operations research volume 29 issue 4, 628-645.

[2] Rodammer, F., & White Jr., K. (1988). A Recent Survey of Production Scheduling. IEEE Transactions on system, man, and cybernetics vol18, no. 6, 841-851.

[3] M. R., G., D. S., J., & R., S. (1976). The complexity of flow shop and job-shop scheduling. Mathematics od Operations Research 1, 117-129.

megvalósítani. Az ütemezéseket sok különböző tényező befolyásolja, mint a munkaprioritás, az esedékességi követelmények, a kiadás dátuma, a költségkorlátozások, a termelési szintek, a raktározási kapacitások, a gépek rendelkezésre állása, illetve a műveletek priorizálása.

A gyártási ütemterv kidolgozása magába foglalja a műveletek olyan sorrendjének kiválasztását, mely egy munka elvégzését eredményezi. Továbbá feladata az egyes műveletek végrehajtásához szükséges erőforrások kijelölése, valamint az ütemezésben szereplő egyes műveletek végrehajtásának kezdő és befejező időpontjának megjelölése. Az útvonaltervek és az erőforrás kijelölések jellemzően a folyamattervezés eredményei.[4]

1.1.1. Gyártási idő, mint célfüggvény:

Habár gyártási idő minimalizálása nem tekinthető jó elméleti célfüggvénynek, a tudományos és ipari gyakorlatban széles körben alkalmazzák. Ennek a kritériumnak nagy történelmi jelentősége van, hiszen a fentebb említett Akers, Johnson és Jackson, is ezt a célfüggvényt alkalmazta az optimális algoritmus kutatása közben. Ez a célfüggvény matematikailag könnyen kezelhető, és egyszerűen megfogalmazható, ezért is lehetett oly népszerű az 1950-es években. következésképpen ez volt a tudományos kutatások fő célja, amely képes megragadni azt az alapvető számítási nehézséget, amely implicit módon létezik az optimális ütemterv meghatározására.[5]

Azért is tartottam fontosnak kiemelni ezt a célfüggvényt, mivel a lentebb bemutatott ütemezési problémáknál (lásd 1.1.2. Szalagrendszerű gyártás (Flow Shop), 1.1.3. Műhelyrendszerű gyártás (Job Shop)) is ezt használom, mint célfüggvényt, illetve Akers módszere is ezt a célfüggvényt használja, mint ahogy a három feladatos kiterjesztés is.

1.1.2. Szalagrendszerű gyártás (Flow Shop)

Akkor hívunk egy ütemezési problémát/feladatot Szalagrendszerű ütemezésnek, ha n darab feladat f0, …, fn, és m darab gép/munka állomás g0, …, gm esetén mindegyik feladat egy előre meghatározott sorrendbe mennek a munkaállomásokhoz, vagyis mindegyik feladatnak ugyan azt a sorrendet kell követnie. Természetesen itt is megvan adva minden feladathoz és géphez a megfelelő műveleti idő, és egy munkaállomáson egyszerre csak egy feladat lehet, míg egy feladaton egyszerre csak egy munkaállomás végezhet munkát. Itt a cél a

[4] Rodammer, White Jr.: A Recent Survey of Production Scheduling. i.m.

[5] A.S., J., & S. Meeran. (1999). Deterministic job-shop scheduling: Past, present and future. European Journal of Operational Research Vol. 113, Issue 2, 390-434.

makespan, vagyis a végrehajtási idő minimalizálása, ami az ütemezés kezdete, és az ütemezés vége között eltelt idő.[6]

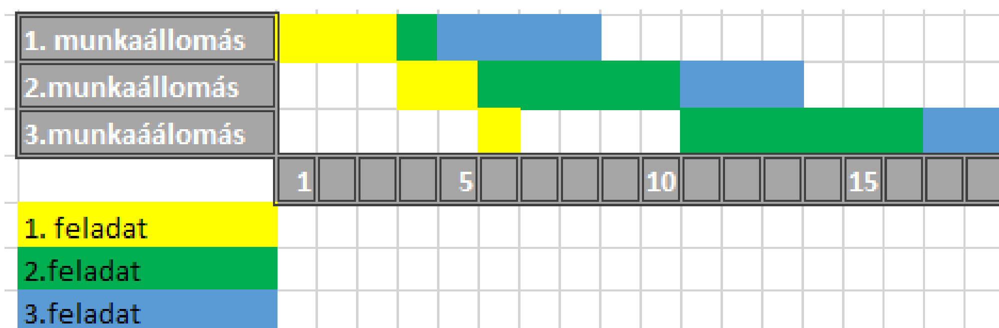
A szalagrendszerű ütemezés, igazából a műhelyrendszerű (lásd 1.1.3. Műhelyrendszerű gyártás (Job Shop)) ütemezésnek egy speciális változata.

Bemenete egy mátrix, ami tartalmazza a gépeket/munkaállomásokat és a munkákat/feladatokat, és tartalmaz, minden feladathoz és géphez egy műveleti időt. Minden feladat pontosan annyi géphez tartozik, ahány gép van összesen, tehát nem lehet a műveleti idő nulla sehol sem. Lehetséges bement:

Feladat\Munkaállomás	1. munkaállomás	2. munkaállomás	3. munkaállomás
1. feladat	3	2	1
2. feladat	1	5	6
3. feladat	4	3	2

1. ábra Flow Shop lehetséges bemenete (Saját szerkesztés)

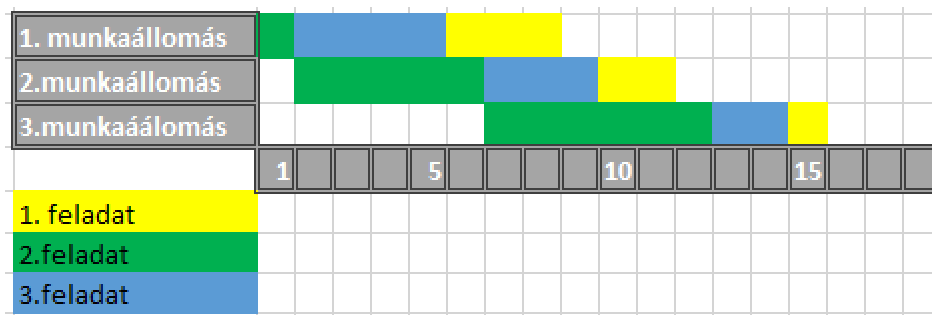
Ennek a bemenetnek egy nem optimális megoldása akár lehet a következő, ez az általános ábrázolási formája:



2. ábra Szalagrendszerű ütemezés nem optimális megoldás ábrázolása (Saját szerkesztés)

A képen a színek jelölik a feladatokat és szépen látszik, hogy melyik feladat mikor éppen melyik munkaállomáson van. Az is látszik, hogy nem optimális, hiszen, ha a zöld majd a kék és végül a sárga feladatok sorrendjébe mennénk, akkor lecsökkenne a teljes ütemezés átfutási ideje. Erre a feladatra az optimális ütemezés az alábbi:

[6] Peter, B., Yu N., S., & Frank Werner. (2007). Complexity of shop-scheduling problems with fixed number of jobs: a survey. Math. Meth. Oper. Res. 65, 461-481.



3. ábra Szalagrendszerű gyártás optimális ütemezésének ábrázolása (Saját szerkesztés)

1.1.3. Műhelyrendszerű gyártás (Job Shop)

Azt az ütemezést hívjuk műhelyrendszerű ütemezésnek, ahol n darab feladat van, f0, …, fn és m darab munkaállomás, g0, …, gm, és mint a szalagrendszerű ütemezésnél is, minden feladat, több részmunkából áll össze. Minden feladathoz/munkához tartozik egy előre meghatározott sorrend, ami azt jelöli, hogy melyik gép után, melyik gépnek kell következnie.[7] Ez lehet akár eltérő is feladatonként, azonban ha mégsem az, akkor egy speciális műhelyrendszerű ütemezésről beszélünk, amit szalagrendszerű ütemezésnek hívnak (lásd 1.1.2. Szalagrendszerű gyártás (Flow Shop)). Ennél az ütemezésnél a cél, hogy csökkentsük a makespant vagyis a végrehajtási időt, ami az ütemezés kezdetétől, és az ütemezés vége között el telt idő.[8]

Műhelyrendszerű gyártásnál gyakori probléma, hogy holtpontra (deadlock) kerül két gép, vagyis egymás lépésére várnak, ezt hívják végtelen költség problémának (the problem of infinite cost).

1976-ban Garey bizonyította, hogy m > 2 (több mint két gép/munkaállomás) esetén ez NP nehéz probléma.[9]

Általában a bemenete két mátrix, egyik mátrixban az idők vannak felsorolva a feladatokhoz, másik mátrixban a munkaállomások sorrendje van felsorolva a feladatokhoz szintén. Példa:

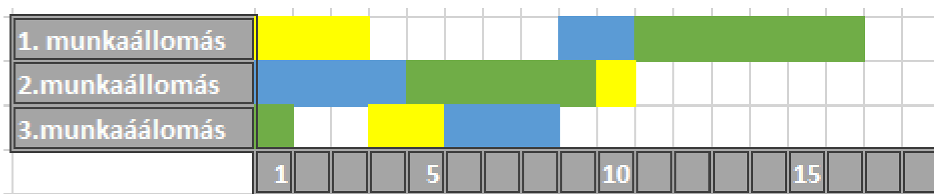
[7] Peter, B., & Bernd, J. (1993). A new Lower bound for the job-shop scheduling prodblem. European Journal of Operational Research 64, 156-167.
[8] Jien, X., Xinyu Li, Liang Gao, & Lin, G. (2021). A new neighboorhood structure for job shop scheduling problems. Wuhan: Huazhong University of Science and Technology.
[9] M. R., G., D. S., J., & R., S. (1976). The complexity of flow shop and job-shop scheduling. Mathematics od Operations Research 1, 117-129.

1. feladat	3	2	1
2. feladat	1	5	6
3. feladat	4	3	2
1. feladat	1. munkaállomás	3. munkaállomás	2. munkaállomás
2. feladat	3. munkaállomás	2. munkaállomás	1. munkaállomás
3. feladat	2. munkaállomás	3. munkaállomás	1. munkaállomás

4. ábra Műhelyrendszerű ütemezés bementi példa (Saját szerkesztés)

Kimenete nagyban függ a használt algoritmustól. Sok féle képen lehet ábrázolni egy ütemezést, a legelterjedtebb módja a gnatt diagramm (pl.: 5. ábra), de lentebb bemutatok másfajta ábrázolási módot is.

Egy lehetséges nem feltétlen optimális kimenete a fentebb (4. ábra) említett példának:



5. ábra Műhelyrendszerű ütemezés lehetséges kimeneti ábrázolása (Saját szerkesztés)

2. AKERS (1956) MÓDSZER BEMUTATÁSA

Akers ezt a módszert 1956-banmutatta be a. A módszert egy speciális műhelyrendszerű ütemezésre találta ki Akers (lásd 1.3.3 Műhelyrendszerű gyártás (Job Shop)). Azért speciális az eset, mert nem lehet bármilyen műhelyrendszerű ütemezésre lefuttatni, és ezáltal nem is ad mindegyikre optimális megoldást polinomiális időben (ez amúgy nem is lehetséges csak ha P = NP hiszen bizonyítottan a műhelyrendszerű gyártás NP nehéz). A specialitása abban rejlik, hogy csak két feladat lehet benne, tehát két feladatra és több gépre ad polinomiális időben optimális megoldást. A gépek lehetnek különböző sorrendben, míg a műveleti idők is lehetnek eltérőek. Az algoritmus célja tehát a legrövidebb idő alatt elvégezni mind a kettő feladatot, míg a feltétele, hogy mind a két feladat keresztül megy mindegyik gépen.[10]

2.1. Algoritmus lépései

1. Először is vegyünk fel egy 2 dimenziós koordináta rendszert és mérjük fel rá a feladatokat, mégpedig úgy, hogy az x tengelyre az 1. feladatnak a gépekhez tartozó műveleti idejeit majd az y tengelyre a 2. feladatnak a gépekhez tartozó idejeit vesszük.

2. Majd, ha felvettük a műveleti időket, akkor az egy géphez tartozó időket tiltsuk le, vagyis így egy téglalapban letiltjuk a közös gépekhez tartozó időket.

3. Ez után meg kell határozni a lehetséges útvonalakat. Az origóból kell kiindulni, és a cél az a koordináta, ami a P(1. feladat műveleti idejeinek az összege, 2. feladat műveleti idejeinek az összege). Lépni csak úgy lehet, hogy mindig 3 lépés lehetőségünk van, vagy 45°-ban lépünk, tehát mindegyik koordinátát egyel megnövelve lépünk, vagy jobbra megyünk egyet, vagy felfele megyünk egyet, ha valamelyiket megtettük, újra megnézzük a lehetőségeket és lépünk. Egy lépést akkor hajthatunk végre, ha nem érint olyan mezőt, ami le van tiltva, letiltott mezőnek ugye az számít, amit a második lépésben tiltottunk le és bele tartoznak a téglalapokba. Ha lehetséges, mindig 45°-ban kell lépni, ha ez nem lehetséges, akkor még három másik eset állhat fent, vagy csak jobbra tudunk, akkor lépünk, ha csak felfele, akkor szintén, ha pedig mind a két irányba tudunk, akkor mind a kettőt meglépjük, vagyis két vonalunk lesz, és már több lehetséges útvonalunk van, amik közül a végén fog kiderülni melyik az optimális.

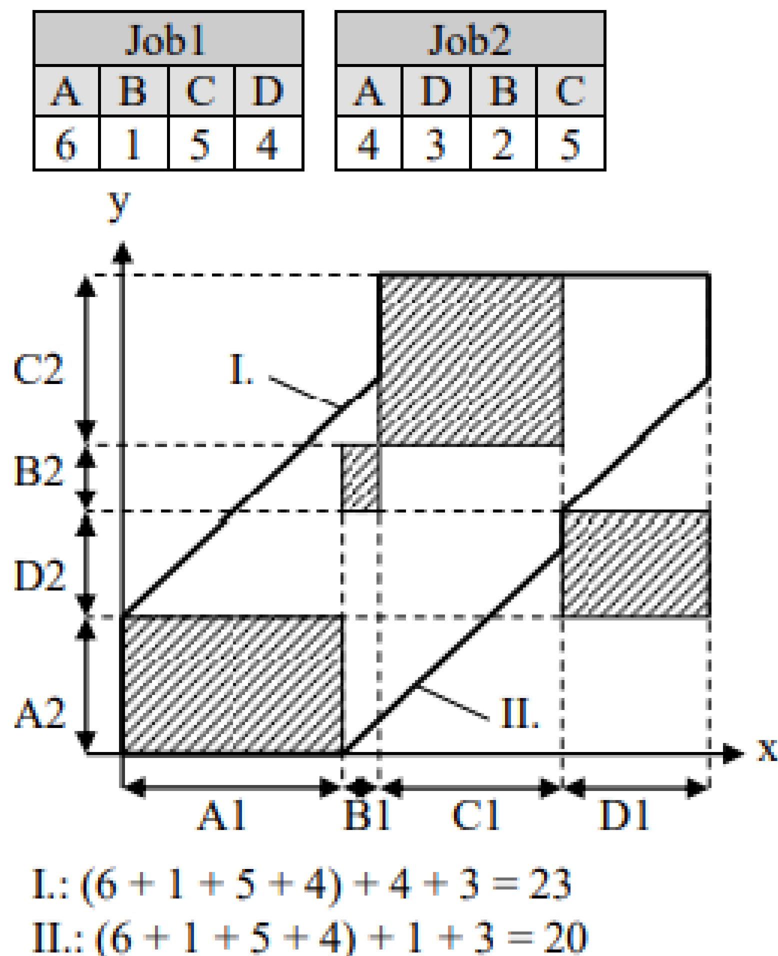
4. Egy útvonalat úgy jellemzünk, hogy megnézzük, hogy mennyi az x tengelyen nézett elmozdulása, (ez ugye mindig az 1. feladat műveleti idejeinek az összege), majd megnézzük,

[10] Sheldon B Akers Jr. (1956). A Graphical Approach to Production Scheduling Problems. Operations Research 4, 244-245.

hogy mennyi az y tengellyel párhuzamos elmozdulása, szóval csak akkor, amikor felfele lépett (ebben nincs benne az átló). Majd, ha az összes lehetséges útvonalra megnéztük, és jellemeztük őket, akkor válasszuk ki közülük a legkisebbet, és az az útvonal lesz az optimális.

5. Tehát a legkisebb számmal jelzett útvonal lesz az optimális. Ez azért van, mert az az algoritmus értelmezése, ha a 45°-ban megyünk, akkor ugye mind a két feladat egyszerre tud haladni, nincs ütközés, éppen nem egy gépet használnának. Ha pedig jobbra megyünk, akkor csak az első feladat halad, a második feladat várakozik ugyan arra a gépre. Ha pedig felfele megyünk, akkor ugyan ez fordítva, hogy az első feladat várakozik a másodikra, mert ugyan azt a gépet szeretnék használni, ami meg nem lehetséges, és ebből látszik is, hogy miért törekszünk mindig 45°-ban lépni.[11]

Ábra az algoritmusról:



6. ábra Akers módszer ábrázolása [12]

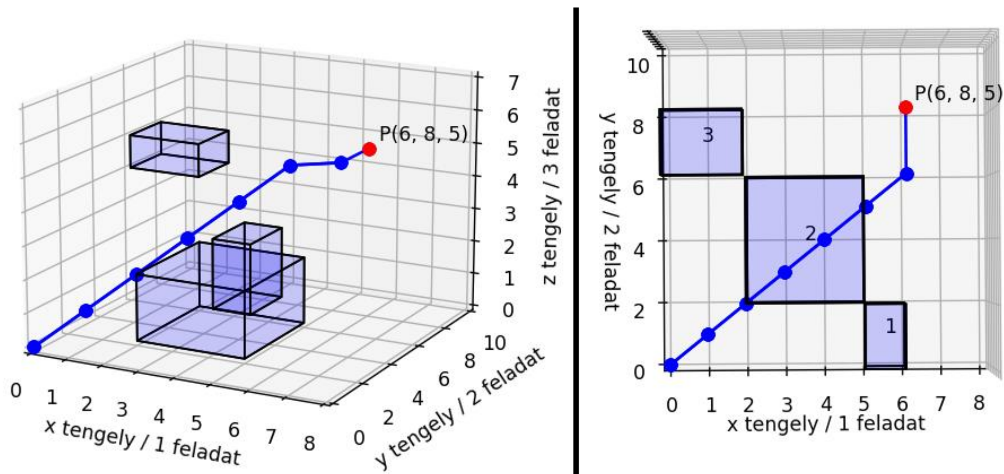
[11] Szikora, B. (dátum nélk.). Jegyzet. Termelésinformatika 5.48 változat. Budapesti Műszaki és Gazdaságtudományi egyetem.
[12] U.o.

Az ábrán is jól látszik, hogy az algoritmus lefuttatásával kettő potenciális megoldást kapunk. De ebben az esetben csak az egyik számít optimálisnak, hiszen ha a 4-es pont szerint eljárunk, látjuk, hogy az I.-el jelzett útvonal hossza 23, míg a II.-al jelzett útvonal hossza 20. A második számú többet tud menni átlósan, vagyis több olyan idő van, amikor párhuzamos munka végzés van, és kevesebb, amikor csak az egyik feladat halad. Tehát kijelenthetjük, hogy a kettő potenciális megoldásból az egyik a megoldás a II.-el jelzett és ezt az útvonalat kell követni, míg az I.-es útvonal nem megoldása az ütemezésnek. Természetesen van olyan eset, amikor több optimális útvonal, van, ilyenkor tetszőlegesen lehet választani közülük, hiszen mindegyik az optimális eredményt adja.

2.2. Az Akers módszer továbbfejlesztésének lehetősége

Akers módszere egy síkban elképzelt levezetett módszer. Azonban mi a helyzet, ha mi ezt megpróbáljuk átültetni 3 dimenzióba, és ugyan ezen a logikát elvégezve, végig menni rajta.

Tehát, mint Akers módszerénél jelöljük a feladatokat a tengelyekkel, és így térben máris egy 3 feladatos problémát kapunk, ami lehet egy lehetséges kiterjesztése Akers módszerének. A következő lépés, hogy mint Akers, mi is letiltjuk az egy géphez tartozó időket. Ez térben természetesen nem téglalapokat, hanem téglatesteket fog jelenteni. Az 1-es géphez tartozó időket úgy tiltjuk le, hogy megnézzük az első feladatnál mikor lehetne leghamarabb elkezdeni az 1-es gépet, illetve mennyi ideig tart a munka. Így például, ha a 4. időegységben lehet elkezdeni, és 6 időegységig kell a feladatot végezni rajta, akkor az x tengelyen megjelöljük 4-10-ig a tengelyt, ezt megtesszük az y tengelyen is a kettes feladattal és a z tengelyen is a hármas feladattal, és így egy téglatest fog pont kirajzolódni, ami az 1-es géphez tartozó időket jelöli. Tehát megvan az Akers módszerben alkalmazott egy géphez tartozó idők letiltása is, így egyelőre úgy néz ki, teljesen jól működhet Akers módszere akár 3 dimenzióban is. A következő lépés, hogy a vonallal úgy jussunk el, a három feladat által meghatározott pontba, hogy nem keresztezzük a téglatesteket. Elvégeztem egy példán a felvetést, és rögtön tisztán látszik miért nem elég csak arra figyelni, hogy ne keresztezzük a téglatesteket, miért csak szükséges, és miért nem elégséges feltétel.



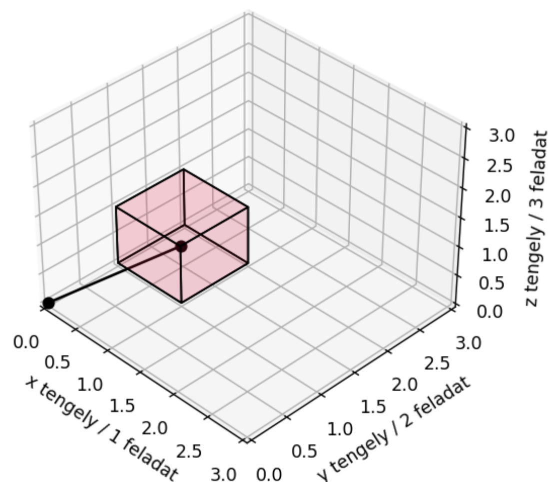
7. ábra Akers módszerének kiterjesztése hibás feltételezéssel (Saját szerkesztés)

A példán sajnos az látszik, hogy míg a vonal nem halad át egyik téglatesten sem, mégsem rajzolódik ki egy lehetséges ütemezés, hiszen, ha megnézzük, akkor a 2-es gépen egyszerre lenne az 1-es feladat és a 2-es feladat is. Ez abból látszik, hogy míg az 1-es feladat a 3-mas gépen kezd és két időegység múlva végez, majd menne a 2-es gépre, addig a 2-es feladat az 1-es gépen kezd, és ő is pont 2 időegység múlva menne a 2-es gépre, ami természetesen nem lehetséges, így máshogy kell próbálkozni. Tehát ilyen formában az Akers módszer nem alkalmazható, így téves volt az a feltételezésünk, hogy ugyan olyan szabályokkal működhet térben, de egy kis kiigazítás után mégis lehetséges az algoritmus kiterjesztése térben, vagyis három feladatra.

3. AKERS MÓDSZER KITERJESZTÉSE HÁROM FELADATRA

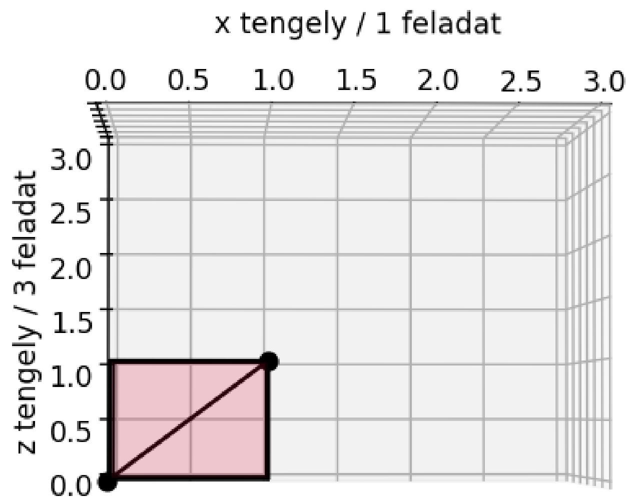
3.1. Az algoritmus bemutatása

Az alapgondolat, az az volt, hogy míg Akers a módszerében egy derékszögű sík koordináta rendszert használt, addig én gondoltam biztos átlehet ültetni az ötletét egy térbeli háromdimenziós koordináta rendszerbe. Az alapja az ugyan az, mégpedig, hogy letiltjuk az egy géphez tartozó időket. Szóval az x tengelyre felvesszük az első feladat gépekhez tartozó idejeit, az y tengelyre a második feladat idejeit, míg a z tengelyre a harmadik feladat idejeit. Ezután letiltjuk az egy géphez tartozó időket, így minden gép után egy téglatestet fogunk kapni. Az alapcél az az, hogy minimalizáljuk az átfutási időt, tehát itt is az a lényeg, hogy egy vonallal minél hamarabb, vagyis a legrövidebb úton eljussunk a végpontba, ami a V(x_max, y_max, z_max). Az x_max értékének meghatározása az az, hogy mennyi az első feladat gépekre meghatározott idejeinek szummája, és ugyan így határozzuk meg az y_maxot, illetve a z_maxot is. A vonal, az azt határozza meg, hogy éppen melyik feladat halad, vagyis melyik feladaton végez munkát egy gép. Ha pl a vonal elmozdulásának irányvektora $V_1(1, 1, 1)$, akkor mind a három feladat halad, ha az irányvektor $V_2(1, 1, 0)$ akkor csak az első illetve a második feladat halad, a harmadik az áll, mivel a z tengelyen nem történt elmozdulás, és így tovább. Figyelni kell, hogy a vonal ne haladjon át egyik téglatesten sem, hiszen akkor egy gépen egyszerre két feladatot is végeznénk egy időben, ami nem megengedett, tehát, így első ránézésre nincs is nagyon nagy változás Akers algoritmusához képest. Azonban itt nem elég arra figyelni, hogy a vonal ne haladjon át a téglatesteken, mert ez nem elégséges, csak szükséges feltétele annak, hogy minden gépen egyszerre csak egy feladat lehessen. Az alábbi példán látszik is, hogy miért nem elég:



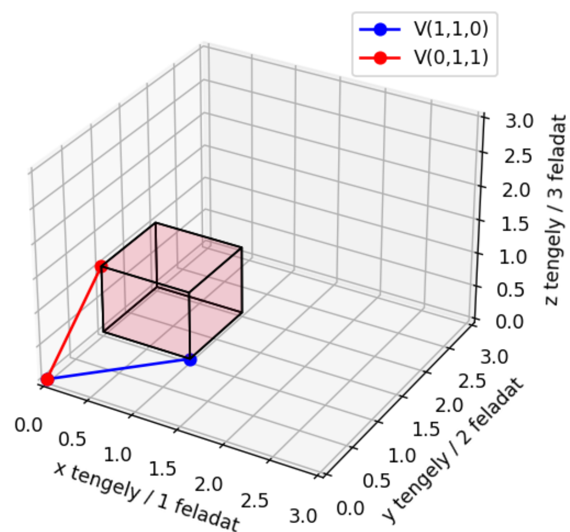
8. ábra Akers kiterjesztésének nehézsége I. (Forrás: Saját kód, Python matplotlib könyvtár használatával)

Itt az látszik, hogy a vonal nem sérti azt a szabályt, hogy áthaladna egy téglatesten, ennek ellenére mégsem jó, hiszen az 1-es gépen egyszerre két feladat is halad, az egyes, illetve a hármas. És ha megnézzük az x, illetve a z koordináta síkot, akkor látszik, hogy a vonal az sértené Akers módszerét. Ezt a problémát már bemutattam feljebb is (lásd 2.2. Az Akers módszer továbbfejlesztésének lehetősége).



9. ábra Akers kiterjesztésének nehézsége II. (Forrás: Saját kód, Python matplotlib könyvtár használatával)

Tehát mikor haladunk a vonallal, azt is le kell ellenőrizni, hogy nem e sértjük Akers módszerét, vagyis megkell nézni, minden feladatot minden feladatra, és az alapján, hogy sehol se sérüljön a módszer, lépni. Tehát a fenti példában a helyes haladása a vonalnak, vagyis az irányvektora az lehet V(1, 1, 0), vagy V(0, 1, 1), hiszen a lényeg, hogy az első, és a harmadik feladat egyelőre nem tud párhuzamosan haladni, hisz ugyan azt a gépet akarnák használni.



10. ábra Akers kiterjesztés problémájának feloldása (Forrás: Saját kód, Python matplotlib könyvtár használatával)

Itt is, mint a fenti példa is mutatja, akár több vonal is lehetséges, szóval több ütemezést is fogunk találni az algoritmussal, azonban természetesen az a legjobb útvonal, aminek a vonala a legrövidebb, így azt az ütemezést kell, hogy majd a végén válasszuk, ha az optimálisat akarjuk visszakapni.

3.2. Az algoritmus lépései

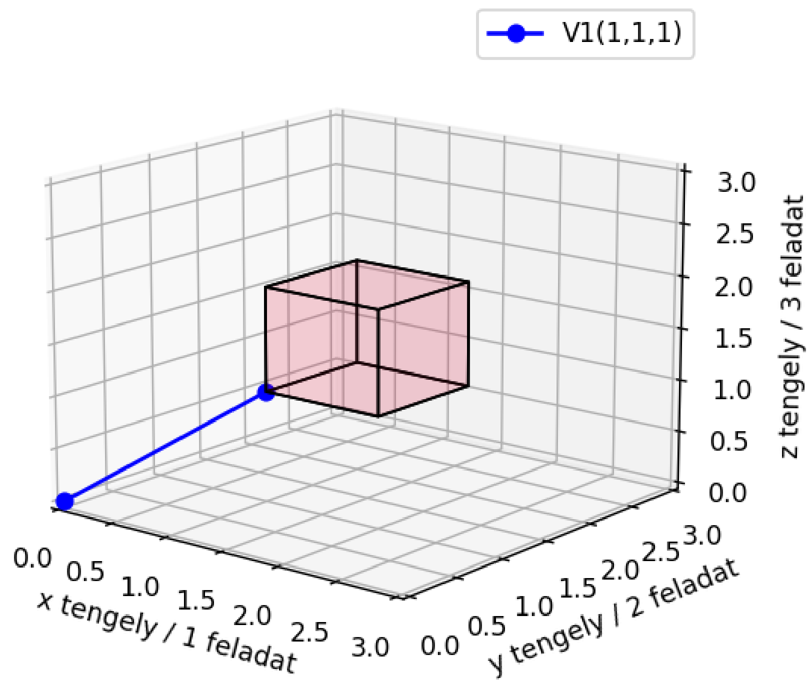
1. Az algoritmus első lépése az az, hogy mindegyik tengelyre felvesszük a feladatok gépekhez rendelt idejét. Az egy géphez tartozó idők meghatároznak egy téglatestet.

2. Ezeket a téglatesteket úgymond letiltjuk, tehát a vonalunk, ami majd jelöli, hogy éppen melyik munka halad az nem haladhat át ezeken a testeken.

3. Amint elkészültünk az első két lépéssel, elkezdhetjük a valódi ütemezést. A célunk az az, hogy a P(0, 0, 0)-ból eljussunk a P(1. feladat műveleti idejeinek az összege, 2. feladat műveleti idejeinek az összege, 3. feladat műveleti idejeinek az összege) pontba. Az utat, vagyis az ütemezést, egy vonallal tudjuk jelölni, a vonal haladása az x, az y, illetve a z tengelyre mutatja meg, hogy éppen halad-e a munka. Ha elértünk a végpontba, akkor minden feladatot végre hajtottunk az ütemezésünk elkészült.

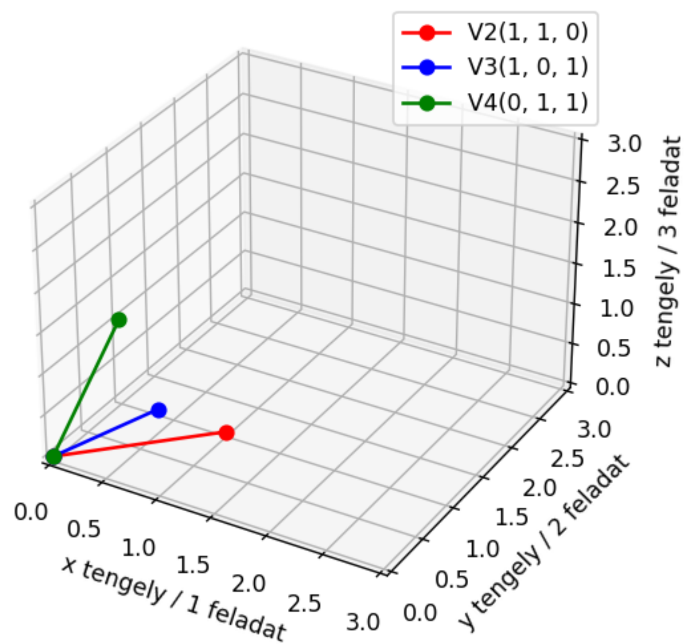
4. Most következik az algoritmus legfontosabb lépése, ami addig ismétlődik, amíg el nem érünk a 3. lépésben meghatározott pontba. Lépni, vagyis a vonallal haladni hét féle képen lehetséges. Ezen lehetőségek irányvektorai az alábbiak: $V_1(1, 1, 1)$, $V_2(1, 1, 0)$, $V_3(1, 0, 1)$, $V_4(0, 1, 1)$, $V_5(1, 0, 0)$, $V_6(0, 1, 0)$, $V_7(0, 0, 1)$. Az irányvektorok nem csak az irányt, de a lépés hosszát is meghatározzák. Az algoritmus mindig arra törekszik, hogy a lehető legtöbb feladat haladhasson egyszerre. A V_1 jelöli azt az irányt, amikor mind a három feladat halad, így mindig arra törekszünk, hogy a haladásunk iránya az ez legyen. Ha nem megoldható, hogy a V_1-et válasszuk, mert letiltott részt keresztez, akkor olyan irányba próbálunk haladni, ahol legalább két feladat halad, ezen irányok a V_2, V_3, V_4. Ha ezek sem használható irányok, mert mind a három feladat egy azon gépre várakozik, akkor a V_5, V_6, V_7 közül választunk. Természetesen, ha többféle képen tudunk lépni, akkor több vonalunk is lesz az algoritmus lefutása után, de ezek közül a legrövidebb, illetve a legrövidebbeknek jellemzettek lesznek az optimálisak. Itt nem elég csak arra figyelni egy lépésnél, hogy ne keresztezzünk, egy téglatestet, hanem, hogy az x és y vagy az x és z vagy az y és z által meghatározott síkra vett vetületnél se sértsük meg a letiltást (ezt feljebb képpekkel illusztráltam, 8. ábra, 9. ábra). Így valamivel bonyolultabb az irány választása, mint az Akers módszernél, mert itt egyszerre mind három vetületet kell vizsgálni, és mindegyikben úgy haladni, hogy ne haladjon át a vonal egyik letiltáson sem.

V_1 szemléltetése:



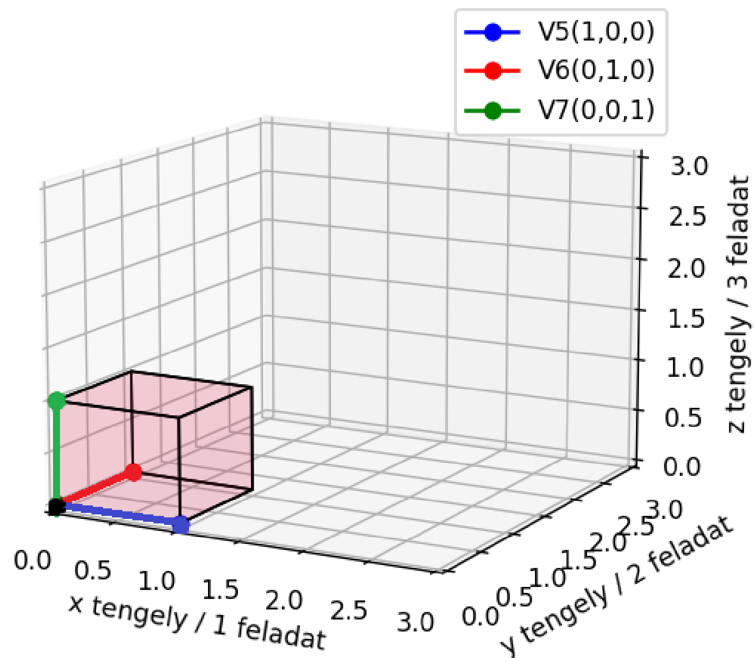
11. ábra $V_1(1, 1, 1)$ irány (Forrás: Saját kód, Python matplotlib könyvtár használatával)

V_2, V_3, V_4 szemléltetése:



12. ábra V_2, V_3, V_4, irányok (Forrás: Saját kód, Python matplotlib könyvtár használatával)

V_5, V_6, V_7 szemléltetés:



13. ábra V_5, V_6, V_7 irányok (Forrás: Saját kód, Python matplotlib könyvtár használatával)

5. Egy vonalat úgy jellemzünk, hogy megnézzük mennyi volt az x tengelyre vett elmozdulása (ezek az elmozdulások számítanak ide: $V_1(1, 1, 1)$, $V_2(1, 1, 0)$, $V_3(1, 0, 1)$, $V_5(1, 0, 0)$). Majd megnézzük, hogy mennyi volt az y tengelyre vett elmozdulása amikor x felé nem mozdult (ezek az elmozdulások számítanak ide: $V_4(0, 1, 1)$, $V_6(0, 1, 0)$). Végül megnézzük mennyi volt a z tengelyre vett elmozdulása, amikor sem x sem y–on nem mozdult (ez az elmozdulások számít ide: $V_7(0, 0, 1)$). Az összes vonalat jellemezzük és kiválasztjuk a legrövidebbet.

A legrövidebb vonal kiválasztásával le is zárul az algoritmus, elkészült az optimális ütemezés.

3.3. Az algoritmus pszeudokódja

Az algoritmusunk fő működése a vonal(ak) berajzolása a három dimenziós koordináta rendszerbe, ezért ennek a pszeudokódját fogom bemutatni, a téglatestek ábrázolását nem tartalmazza. Tehát feltételezzük, hogy van egy három dimenziós koordináta rendszerünk, aminek a tengelyei a feladatokat jelentik, és vannak téglatesteink, amik az egy géphez tartozó feladatok futási idejét jelölik. Az algoritmus dolga megtalálni az optimális ütemezést, vagyis a legrövidebb utat a koordináta rendszerünkben. Így ahogy már említettem arra törekszik az algoritmus, hogy ha lehet, mind a három feladat haladhasson, ha nem, akkor legalább kettő, ha az sem, akkor csak egy feladat halad.

```
1   vonalak = tömb
2   vonal = tömb
3   elágazások = tömb
4   Amíg <nem fedünk le minden elágazást>
5       vonal = [[0, 0, 0]]
6       Amíg <a vonal utolsó eleme != [1feladat futási idő; 2feladat futási idő; 3feladat futási idő]>
7           Ha <mindegyik sík párosításon lehet átlósan haladni>
8               vonal.hozzáfűz(vonal [-1] + (1, 1, 1))
9               Vissza Amíg
10          VégeHa
11          Ha <valahol nem lehet átlósan haladni>
12              Ha <több is teljesül a lenti háromból>
13                  elágazásokba feljegyezni
14              VégeHa
15              Ha <x-y síkon lehet átlósan haladni, és x irányba is lehet minden síkon,
16                  illetve y irányba is, elágazásokat is figyelni kell>
17                  vonal.hozzáfűz(vonal [-1] + (1, 1, 0))
18                  Vissza Amíg
19              VégeHa
20              Ha <x-z síkon lehet átlósan haladni, és x irányba is lehet minden síkon,
21                  illetve z irányba is, elágazásokat is figyelni kell>
22                  vonal.hozzáfűz(vonal [-1] + (1, 0, 1))
23                  Vissza Amíg
24              VégeHa
25              Ha <y-z síkon lehet átlósan haladni, és y irányba is lehet minden síkon,
26                  illetve z irányba is, elágazásokat is figyelni kell>
27                  vonal.hozzáfűz(vonal [-1] + (0, 1, 1))
28                  Vissza Amíg
29              VégeHa
30          VégeHa
31          Ha <sehol sem lehet átlósan haladni>
32              Ha <több is teljesül a lenti háromból>
33                  elágazásokbe feljegyezni
34              VégeHa
35              Ha <x irányba lehet haladni mindegyik síkon, elágazásokat is figyelni kell>
36                  vonal.hozzáfűz(vonal [-1] + (1, 0, 0))
37                  Vissza Amíg
38              VégeHa
39              Ha <y irányba lehet haladni mindegyik síkon, elágazásokat is figyelni kell>
40                  vonal.hozzáfűz(vonal [-1] + (0, 1, 0))
41                  Vissza Amíg
42              VégeHa
43              Ha <z irányba lehet haladni mindegyik síkon, elágazásokat is figyelni kell>
44                  vonal.hozzáfűz(vonal [-1] + (0, 0, 1))
45                  Vissza Amíg
46              VégeHa
47          VégeHa
48      VégeAmíg
49      vonalak.hozzáfűz(vonal)
50  VégeAmíg
51  Majd a vonalak tömbben megkeressük a legrövidebb vonalat és az lesz a megoldásunk.
```

14. ábra Kiterjesztés Pszeudokódja (Saját Szerkesztés)

A fenti pszeudokód megadja nekünk a vonalak tömbjét, ami még nem a megoldásunk, mert több vonalat is tartalmaz, köztük az optimálisat is, ami már a megoldásunk lesz. Ezt úgy határozzuk meg, hogy megnézzük, hogy egy-egy vonalunk milyen hosszú (itt és később is, nem a vonal valós hosszát értjük, hanem, hogy hány általunk meghatározott ponton megy keresztül), vagyis, hogy egy vonal hány tömbből épül fel. A vonalak egy tömbök tömbjének a tömbje, míg a vonal egy tömbök tömbje, és ezek a tömbök tartalmazzák azokat a pontokat a koordináta rendszerben, amiken keresztül megy a vonal. Ebből is látszik, hogy érdemes a V(1, 1, 1) irányba lépni, mert a vonal hosszát csak 1-el növeli, és mégis a legtöbbet halad, míg ha ez nem lehetséges, akkor próbálunk olyan irányba lépni ahol legalább két munka halad, mert akkor is többet halad a vonalunk, mintha csak az egyik tengely irányába mozdulna el. Tehát meghatározzuk a legkevesebb ponton keresztülmenő vonalat, és az lesz a megoldásunk.

3.4. Algoritmus példa futás

Szeretném bemutatni az algoritmusomat egy példán keresztül is, hogy hogyan is működik élesben.

Az algoritmus bemenete meghatározza, hogy egy-egy feladatnak mennyi időt kell eltöltenie egy-egy gépen, tehát tudjuk a gépek számát és a gépekhez rendelt időket feladatonként. Az alábbi táblázatban látható a bemenet:

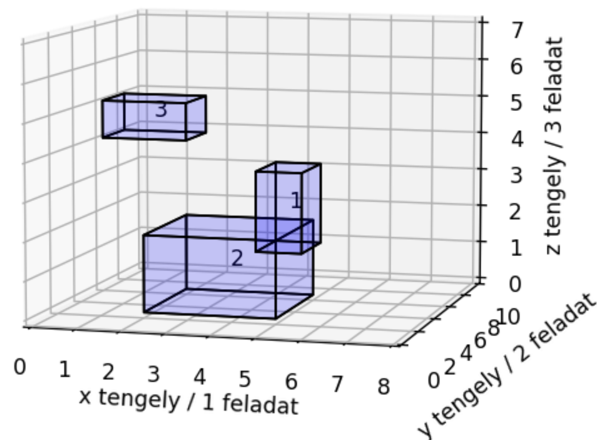
1. Job gép igénye sorrendben, alattuk a műveleti idő:				
3	2	1		
2	3	1		
2. Job gép igénye sorrendben, alattuk a műveleti idő:				
1	2	3		
2	4	2		
3. Job gép igénye sorrendben, alattuk a műveleti idő:				
2	1	3		
2	2	1		

15. ábra Példa bemenet (Forrás: Saját megszorításos inputgenerátorral generált bemenet)

Magyarázat: Ez azt jelenti például, hogy az első feladatnak az alábbi sorrendbe kell menni a géphez: 3-as gép, 2-as gép, illetve az 1-es gép. A gépek műveleti ideje az alábbi, ha az 1-es feladat érkezik: 3-as gépnek 2 egység idő, 2-es gépnek 3 egység idő, míg az 1-es gépnek 1 egység a műveleti ideje.

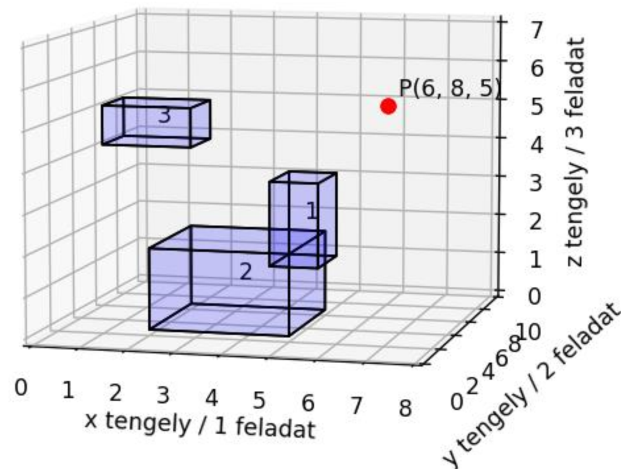
1. Az első teendő az algoritmusunk szerint az az, hogy vegyük fel a koordináta tengelyekre a feladatokhoz tartozó időket. Az x tengelyre az első feladat idejeit, az y tengelyre a második, míg a z tengelyre a harmadik feladat idejeit.

2. Letiltjuk az egy géphez tartozó időket, így megalakulnak a téglatesteink.



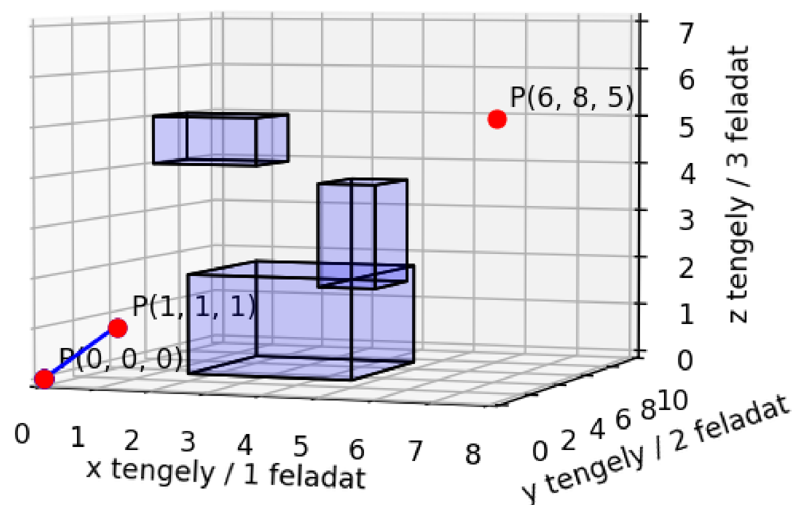
16. ábra Gépekhez tartozó idők letiltása (Forrás: Saját kód, Python matplotlib könyvtár használatával)

3. Meghatározzuk azt a pontot, ahová el kell érnünk, vagyis azt ahol már az összes munka be van fejezve. Ez a pont a P(2+3+1, 2+4+2, 2+2+1) vagyis a P(6, 8, 5).



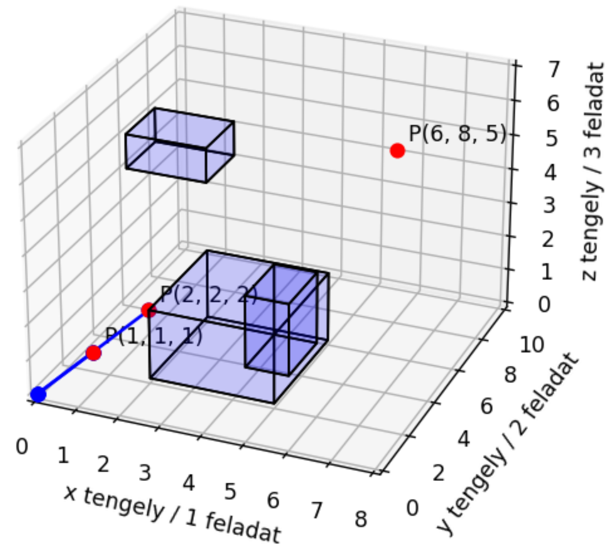
17. ábra Cél pont meghatározása (Forrás: Saját kód, Python matplotlib könyvtár használatával)

4. Most kell meghatározni a vonal útvonalát, és addig lépni egyesével, amíg el nem érünk a 3. lépésben meghatározott pontba. Megvizsgáljuk elsőnek az x-y tengelyek által meghatározott síkra vett vetületét a téglatesteknek, majd az x-z, és az y-z tengely párosokkal is elvégezzük ugyan ezt, és mindegyikben megnézzük, milyen irányokban tudnánk haladni. Az x-y síkon Akers módszer szerint lehet menni az x tengellyel párhuzamosan, az y tengellyel párhuzamosan, illetve 45°-os szöget bezáróan. Azt tapasztaljuk, hogy mindegyik sík vetületen tudunk az összes irányban haladni, így annak a vizsgálata jön, hogy melyik a legelőnyösebb. Ha az összes feladat tud haladni az a legjobb, így mindegyik síkban 45°-os szöget bezáróan lépünk, így a vonalunk (ami szakaszokból épül fel) első szakasza a $P_0(0, 0, 0)$ és a $P_1(1, 1, 1)$ közötti egyenes szakasz, mivel a $V_1(1, 1 , 1)$ irányvektorú lépést választottuk.



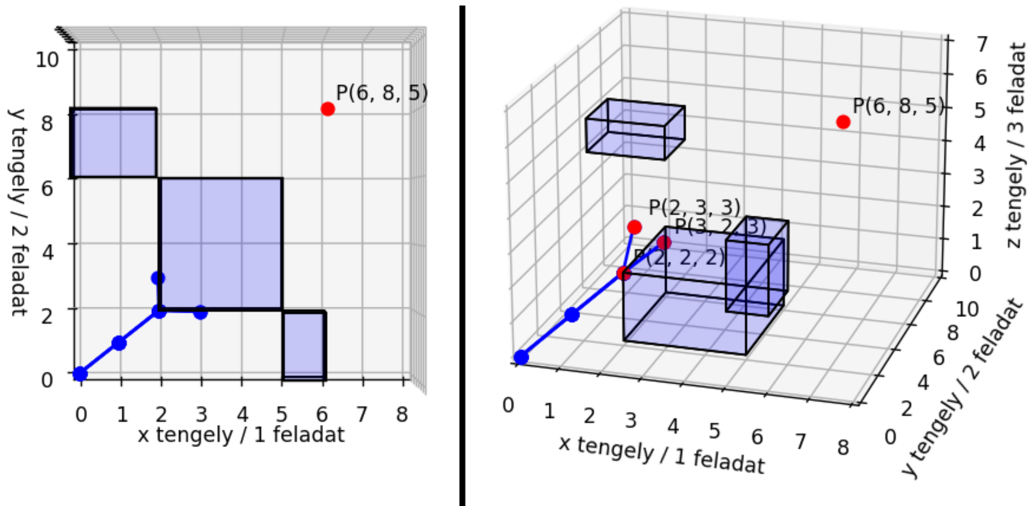
18. ábra 1. lépés (Forrás: Saját kód, Python matplotlib könyvtár használatával)

Majd miután berajzoltuk mindegyik síkba a vonalunk elejét ábrázoló szakaszt, utána vizsgáljuk tovább, mi lehet a következő lépés. Mivel továbbra is mind a három sík engedi a 45°-os tovább haladást, így a következő lépésünk is a $V_1(1, 1, 1)$ irányvektor irányába történik. Így a vonalunk már két szakaszból áll. A vonalat a továbbiakban pontok tömbjeként ábrázolom, mégpedig úgy, hogy ezek között a pontok között a vonal részeit képző egyenes szakaszok vannak. Tehát a vonalunk Vonal[(0, 0, 0),(1, 1, 1),(2, 2, 2)].



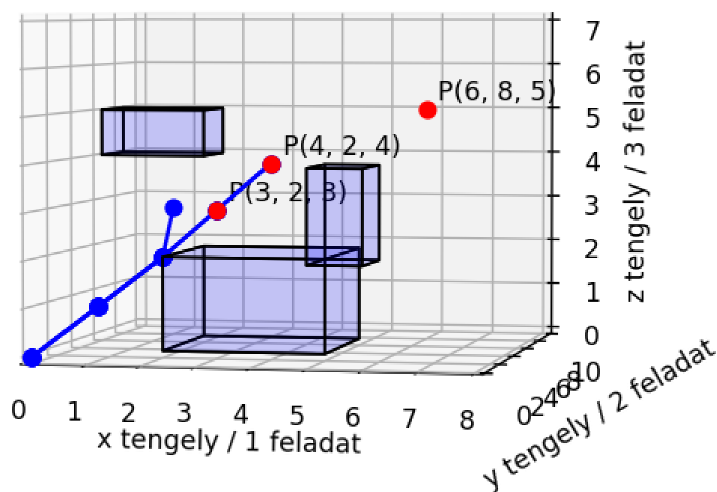
19. ábra 2. lépés (Forrás: Saját kód, Python matplotlib könyvtár használatával)

A következő lépésnél is megvizsgáljuk a sík vetületeket, és azt tapasztaljuk, hogy míg az x-z sík, és az y-z sík is engedi az átlós lépést, addig az x-y sík nem engedi. Itt megvizsgáljuk mindegyik síkot külön-külön, jobban. Azt ugye látjuk, hogy x-y síkon nem lehet átlósan lépni, de mind x, ás mind y irányba haladhatunk. Az x-z síkon tudunk átlósan haladni, és mivel a másik két síkon is tudunk x vagy esetleg z irányba haladni, így ez egy lehetséges lépés $V_3(1, 0, 1)$. Az y-z síkon is tudunk átlósan haladni, és szintén mivel mind a többi síkon lehetséges az y illetve a z irány a másik lehetséges lépés a $V_4(0, 1, 1)$. Azért nem lehetséges a $V_1(1, 1, 1)$ irány, mivel az x-y síkon látszik, hogy egyszerre nem lehet az x, illetve az y irányába sem lépni, mert akkor nem Akers algoritmusa szerint haladnánk síkonként. Látszik, hogy így egy elágazáshoz értünk, innentől kezdve két vonalunk van; Vonal[(0, 0, 0), (1, 1, 1), (2, 2, 2), (3, 2, 3)], és a Vonal[(0, 0, 0), (1, 1, 1), (2, 2, 2), (2, 3, 3)].



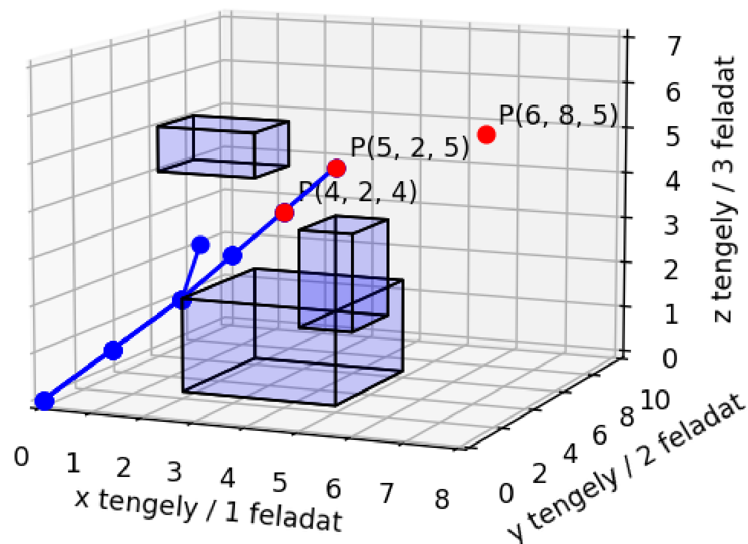
20. ábra 3. lépés, illetve elágazás (Forrás: Saját kód, Python matplotlib könyvtár használatával)

A következő lépésben az első vonalat fogjuk tovább vinni, de innentől ugyan úgy haladhatnánk a másik vonallal is akár. Tovább vizsgálva a síkokat azt látjuk, hogy megint nem tudunk mindenhol átlósan lépni, így amit az előző lépésben alkalmaztunk, megnézzük, hogy ahol igen ott a koordináták másik síkon is haladhatnak-e. Tehát x-z síkon mehetünk átlósan és x is mindegyik síkon haladhat, illetve a z is, így $V_3(1, 0, 1)$ egy lehetséges irány, és más nincs is, mivel átlósan még az y-z tengelyen tudnánk menni, de y nem haladhat az x-y tengelyen. Tehát a vonalunk Vonal[(0, 0, 0), (1, 1, 1), (2, 2, 2), (3, 2, 3), (4, 2, 4)].



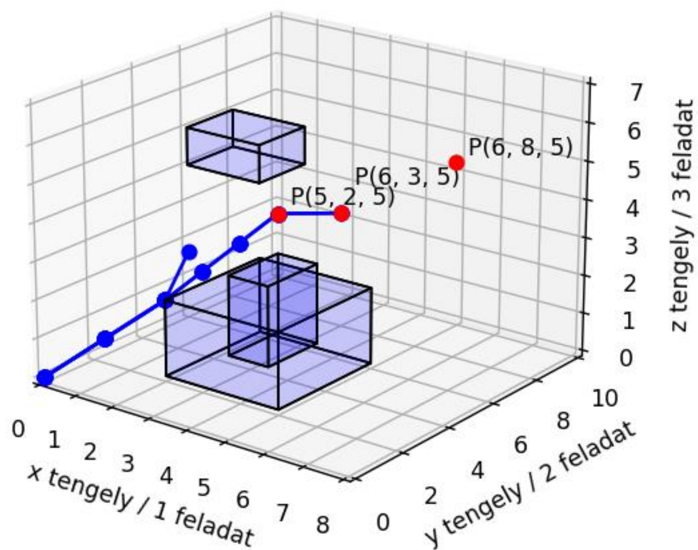
21. ábra 4. lépés (Forrás: Saját kód, Python matplotlib könyvtár használatával)

A következő lépésben pont ugyan az a helyzet, mint az előbb, így itt is a $V_3(1, 0, 1)$ irányba kell elmozdulni. A vonalunk; Vonal[(0, 0, 0), (1, 1, 1), (2, 2, 2), (3, 2, 3), (4, 2, 4), (5, 2, 5)].



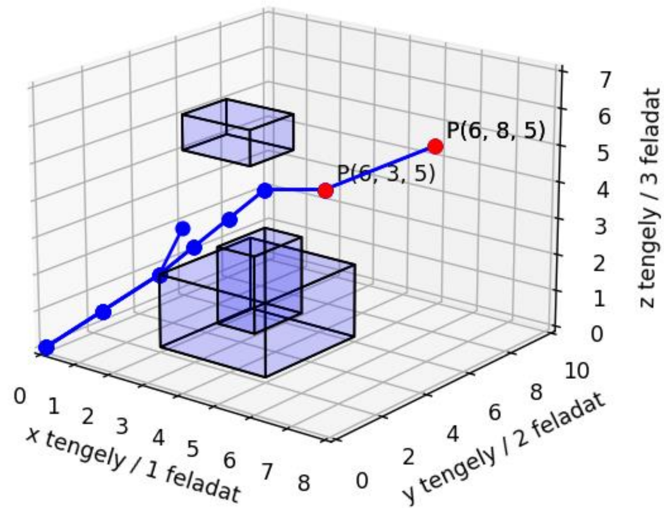
22. ábra 5. lépés (Forrás: Saját kód, Python matplotlib könyvtár használatával)

Ez után viszont azt tapasztaljuk, hogy z árnyba már nem kell mozdulnunk, így a $V_1(1, 1, 1)$, a V(0, 1, 1), a V(1, 0, 1), illetve a V(0, 0, 1) irányokra már nem lesz szükség, ezek már tuti nem jöhetnek. Azt látjuk viszont, hogy az x-y síkon tudunk átlósan menni, és mind az x illetve y irányba tudunk mozogni mindegyik síkon így haladunk tovább a $V_2(1, 1, 0)$ irányba. Vonal[(0, 0, 0), (1, 1, 1), (2, 2, 2), (3, 2, 3), (4, 2, 4), (5, 2, 5), (6, 3, 5)].



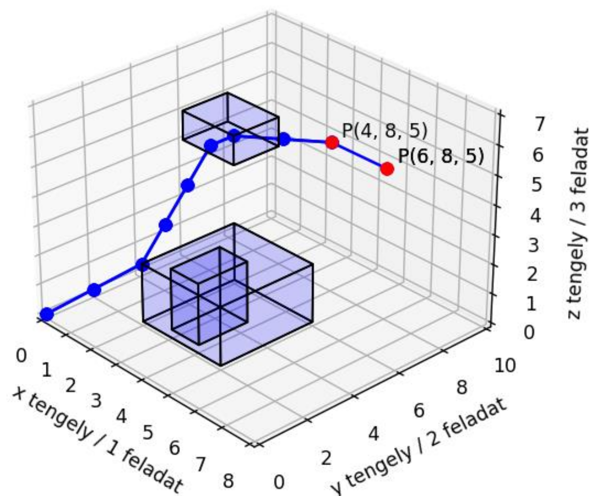
23. ábra 6. lépés (Forrás: Saját kód, Python matplotlib könyvtár használatával)

Most már csak, az y irányba tudunk lépni, mivel mind x, mind z elérte a maximumát, tehát annyiszor lépünk $V_6(0, 1, 0)$ irányba, amíg el nem érünk az általunk kijelölt P végpontot, ez a P(6, 8, 5). Tehát a vonalunk egybe; Vonal[(0, 0, 0), (1, 1, 1), (2, 2, 2), (3, 2, 3), (4, 2, 4), (5, 2, 5), (6, 3, 5), (6, 8, 5)].



24. ábra Első vonal ábrázolása (Forrás: Saját kód, Python matplotlib könyvtár használatával)

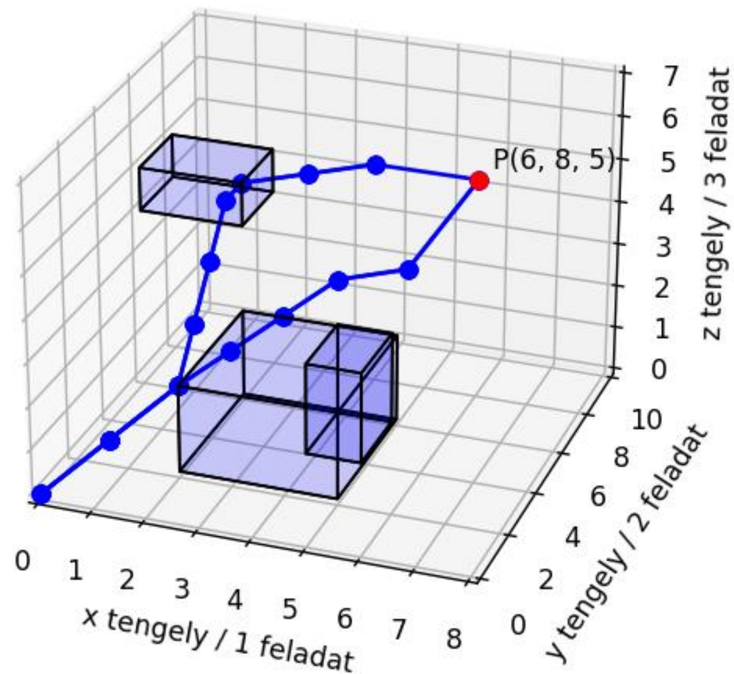
Ha az elágazástól végig csináljuk a másik vonalunkkal is, akkor ezt kapjuk; Vonal[(0, 0, 0), (1, 1, 1), (2, 2, 2), (2, 3, 3), (2, 4, 4), (2, 5, 5), (2, 6, 5), (3, 7, 5), (4, 8, 5), (6, 8, 5)].



25. ábra Második vonal ábrázolása (Forrás: Saját kód, Python matplotlib könyvtár használatával)

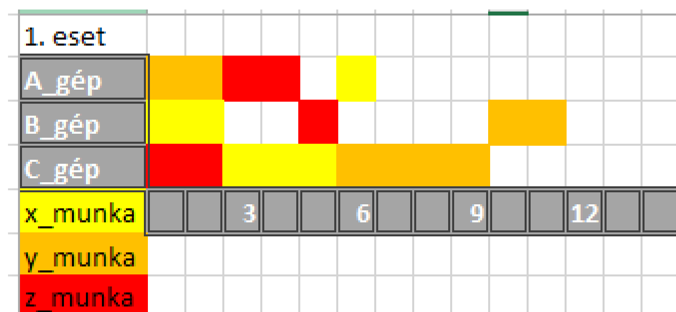
5. Ezután jön a Vonalak értékelése. Az első vonal az ment x irányba, 6-ot, ment y irányba úgy, hogy közben x be nem 5-öt, és ment z irányba úgy, hogy közben sem x, sem y irányba nem mozdult el 0-át, így az első vonalunk értéke 11. A második vonalunk x irányba 6-ot, y irányba úgy, hogy x be nem 4-et míg z irányba, hogy más irányba semmit 0-át, így ennek a vonalnak az értéke 10. Tehát az látszik, hogy a második vonal az optimális, így azt fogja az algoritmus megadni, mint jó ütemezés.

Teljes algoritmus ábrázolás:

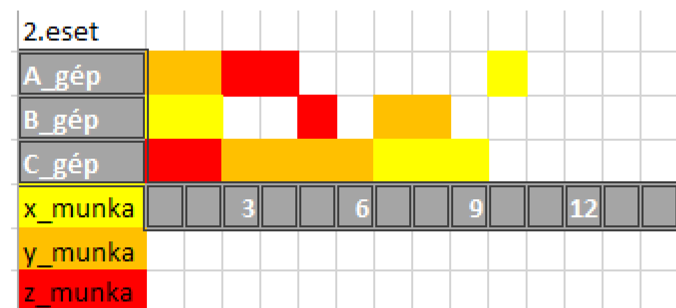


26. ábra Teljes ábrázolás (Forrás: Saját kód, Python matplotlib könyvtár használatával)

Ütemezések ábrázolása kicsit másképp, a hagyományosabb Gnatt-diagrammal:



27. ábra Első eset ábrázolása Gnatt-diagrammal(Saját szerkesztés)

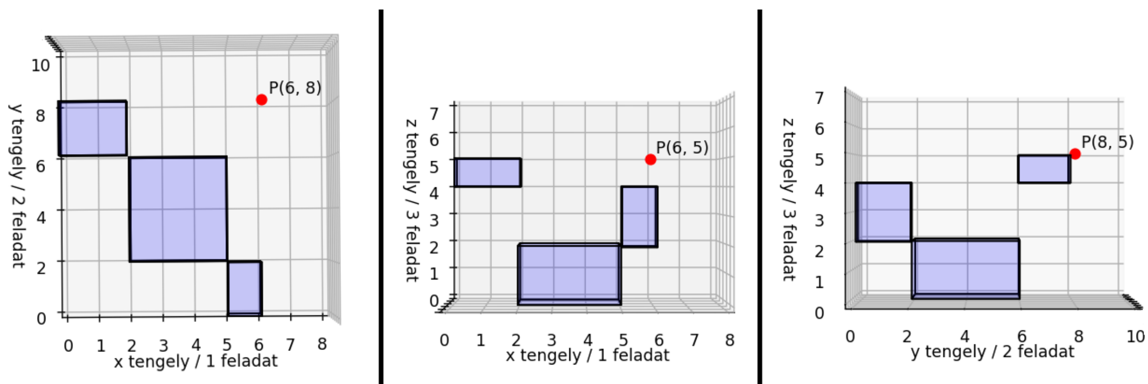


28. ábra Második eset ábrázolása Gnatt-diagrammal (Saját szerkesztés)

Itt is az látszik, hogy ha ábrázoljuk Gnatt-diagrammmon is az ütemezést, hogy a második ütemezés, amit megadott az algoritmusunk az optimális.

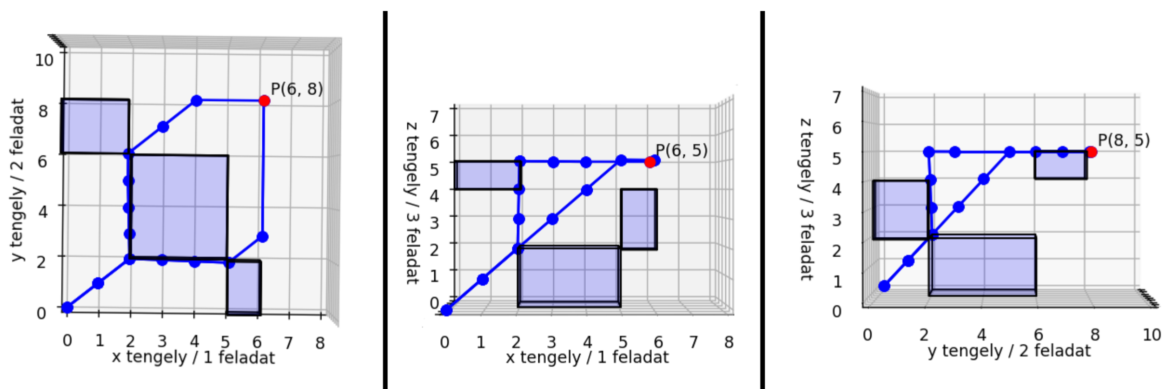
3.5. A kiterjesztés visszavezetése az Akers módszerre

A 3 dimenziós módszert könnyen visszavezethetjük Akers 1956-ban bemutatott módszerére. Ahogy az algoritmus használatához is többször alkalmaztuk Akers módszerét, így a visszavezetés sem bonyolult. Tulajdonképpen az algoritmusunk úgy épül fel, hogy háromszor kell megcsinálnunk Akers algoritmusát, természetesen kisebb módosításokkal, hogy együtt is megfelelőek legyenek, és ne sértsünk szabályt, vagyis egy helyes ütemezést kapjunk. A három tengely jelöli a munkákat, és a tengelyeket kell párosítani egymással így jön ki a 3 db 2 dimenziós koordináta rendszer. Lesz egy x-y, egy x-z, és egy y-z 2 dimenziós koordináta rendszerünk. Ha ezekre külön-külön alkalmazzuk az Akers módszert első lépéseit vagy a kiterjesztés első lépéseit és megnézzük a tengelyek által meghatározott 3 sík vetületét, akkor ugyan azt kapjuk.



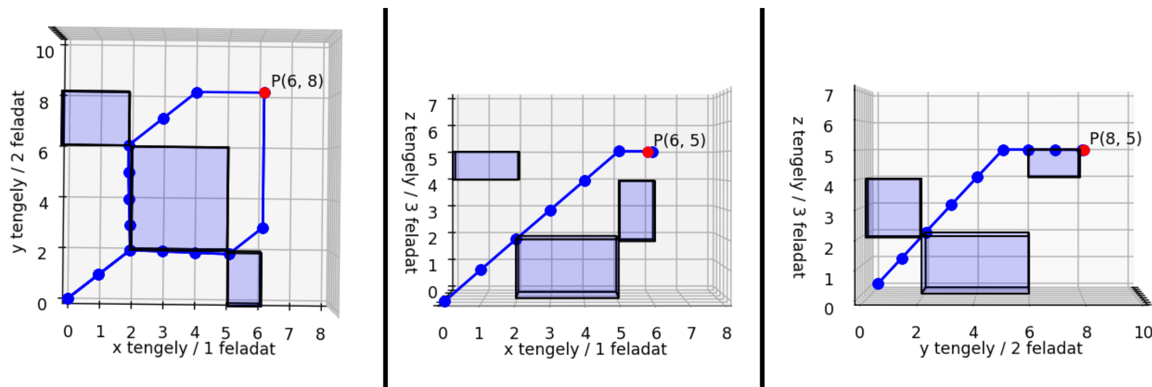
29. ábra Sík vetületek (Forrás: Saját kód, Python matplotlib könyvtár használatával)

Ha lefuttatjuk mind a két algoritmust akkor már nem pont ugyan azt fogjuk kapni, de azért lesznek hasonlóságok, és közös vonalak, de míg ha az Akerst külön-külön lefuttatjuk mind a három sík vetületre addig csak az egyik síkon kapunk két ütemezést, de ha a kiterjesztést futtatjuk akkor kettő lehetséges ütemezést fogunk kapni mind a három esetben, hiszen ezek összekötődnek és egy elágazást kell jelölni mindegyik síkon ha valahol elágazik.



30. ábra Kiterjesztés sík vetületei (Forrás: Saját kód, Python matplotlib könyvtár használatával)

A fenti képen a kiterjsztett algoritmus futását látjuk, úgy, hogy a sík vetületeket vizsgáljuk. Az első ábra pont úgy néz ki mintha az Akers módszert futtattuk volna le, míg a másik két ábrában van egy egy plusz ütemezés.



31. ábra Akers algoritmus futása (Forrás: Saját kód, Python matplotlib könyvtár használatával)

Ha az Akers módszert futtatjuk, akkor csak az első ábránál van elágazás, a másik kettőnél csak egy lehetséges ütemezést kapunk ami ugye az optimális is.

Látszik, hogy a kiterjesztett algoritmus tartalmazza az Akers módszer szerinti optimális megoldásokat, illetve a kettes és a hármas képen azért van több alternatíva, mert az első síkon ketté vált a vonalunk, és eszerint kellett lépni bennük is, tehát míg az elsőn az x vagy csak az y, halad, addig a második illetve a harmadik képen is csak az x, illetve csak az y haladhat. Tehát látszik, hogy a végső optimális ütemezés már nem optimális 2-2 feladatra külön-külön, de egybe mégis ez az optimális megoldás, hiszen ha megnézzük, hogy ahol éppen nem optimális az egyik síkon, úgy a másik kettőn igen, és azt az útvonalat fogja választani az algoritmusunk ahol a legtöbbször optimális legalább kettő feladat és a harmadik is a lehető legtöbbet halad eközben.

Tehát mindig arra törekszik, hogy ha lehetséges, akkor az összes feladat haladhasson egyszerre, illetve ha ez nem lehetséges akkor arra, hogy legalább kettő haladjon, és ha ebből több lehetséges párosítás is van, akkor megvizsgálja az összes lehetséges párosítást, és tovább halad az összes lehetséges megoldással, így összegezve, arra törekszik, hogy minnél több feladat haladhasson egy időben, és ha van több megoldás akkor megvizsgálja az összeset lehetésges esetet, így tuti megkapjuk az optimálisat is.

4. A KÉT MÓDSZER HASONLÓSÁGA

Azért is mondhatjuk, hogy e két módszer nagyon hasonlít, mert ugyan az az alapgondolata a kiterjesztésnek is, mint az alap Akers által kitalált módszernek.

A fő gondolat, mégpedig az, hogy minél hamarabb, vagyis a lehető legrövidebb úton eljussunk egy koordináta rendszerben egy bizonyos előre meghatározott ponthoz, illetve mind a két módszernél vannak a koordináta rendszerben olyan geometriai elemek (Akers esetében téglalapok, míg a kiterjesztésben téglatestek), amiket nem keresztezhet a vonalunk (habár a kiterjesztésben nem elég csak erre figyelni).

Tehát anélkül is, hogy tudnánk, mit jelentenek a téglalapok, téglatestek, tengelyek, vagy vonalak, igen hasonlónak hathat a két algoritmus működése és végeredményének ábrázolása. Igazából a módszerek nem is egy ütemezési problémát oldanak meg elsősorban, hanem egy legrövidebb út keresést. Akers-é egy síkban, míg a kiterjesztés térben keresi két pont között a legrövidebb lehetséges utat (bizonyos szabályok betartása mellett), így belátható, hogy igen nagy a hasonlóság a két algoritmus között, de mégsem lehet őket ugyan olyannak tekinteni.

4.1. A két módszer összehasonlítása teszteléssel

A két módszert összevetettem, mégpedig az alapján, hogy mennyi az átlagos növekményük ahhoz a megoldáshoz képest, mint amikor mindegyik feladat folyamatosan haladhatna az összes gépen sorban, megszakítás nélkül. Tehát ez alapján látszik, hogy az alap, amihez képest a megoldásainkat fogjuk hasonlítani az a leghosszabb feladat végrehajtási ideje. A növekmény egy arányszám, amit úgy számolunk, hogy az algoritmus által létrehozott ütemezésből kivonjuk a leghosszabb feladat végrehajtási idejét, majd el is osztjuk vele, így megkapjuk, hogy hány százalékkal lesz hosszabb a feladatok ütemezése, mintha teljesen függetlenül csinálnánk őket egymástól és mindegyik haladhatna megszakítás nélkül.

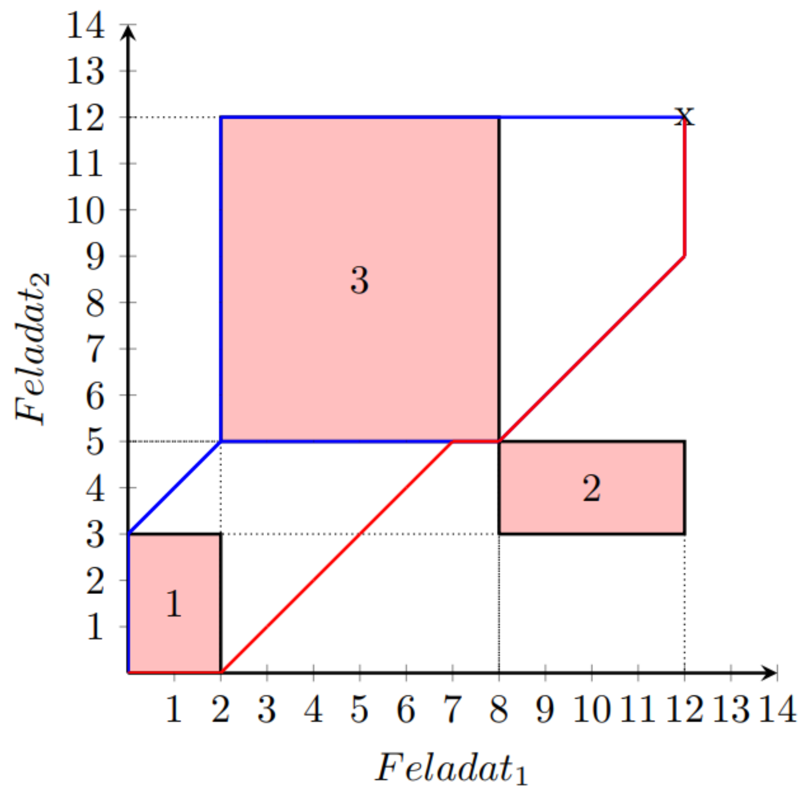
4.1.1. Akers módszer átlagos növekménye

Ezt az átlagot 3 példa után fogjuk vonni. Mégpedig 3 gép esetén. Első példa bemenete:

1 feladat gép igénye sorrendben, alattuk a műveleti idő:				
1	3	2		
2	6	4		
2 feladat gép igénye sorrendben, alattuk a műveleti idő:				
1	2	3		
3	2	7		

32. ábra Első példa bemenet (Forrás: Saját megszorításos inputgenerátorral generált bemenet)

Ábrázolása:



Az optimalis vegrehajtast a piros vonal jeloli.
Vegrehalytasi ido = 12 + 3 = 15

33. ábra Első példa kimenet (Forrás: Saját Python kód Latex megjelenítéssel)

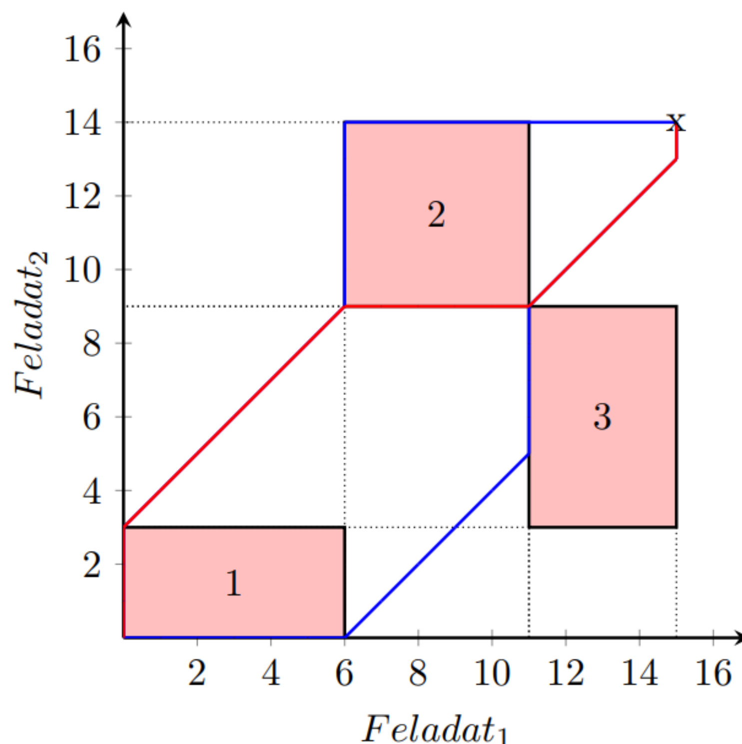
A példa alapján a leghosszabb feladat végrehajtási ideje 12, míg az Akers módszer alapján elkészített ütemezés 15 egység hosszú, így itt a növekmény $N_1 = 3/12$.

Második példa bemenet:

1 feladat gép igénye sorrendben, alattuk a műveleti idő:				
1	2	3		
6	5	4		
2 feladat gép igénye sorrendben, alattuk a műveleti idő:				
1	3	2		
3	6	5		

34. ábra Második példa bemenet (Forrás: Saját megszorításos inputgenerátorral generált bemenet)

Ábrázolása:



Az optimalis vegrehajtast a piros vonal jeloli.
Vegrehalytasi ido = 15 + 4 = 19

35. ábra Második példa kimenet (Forrás: Saját Python kód Latex megjelenítéssel)

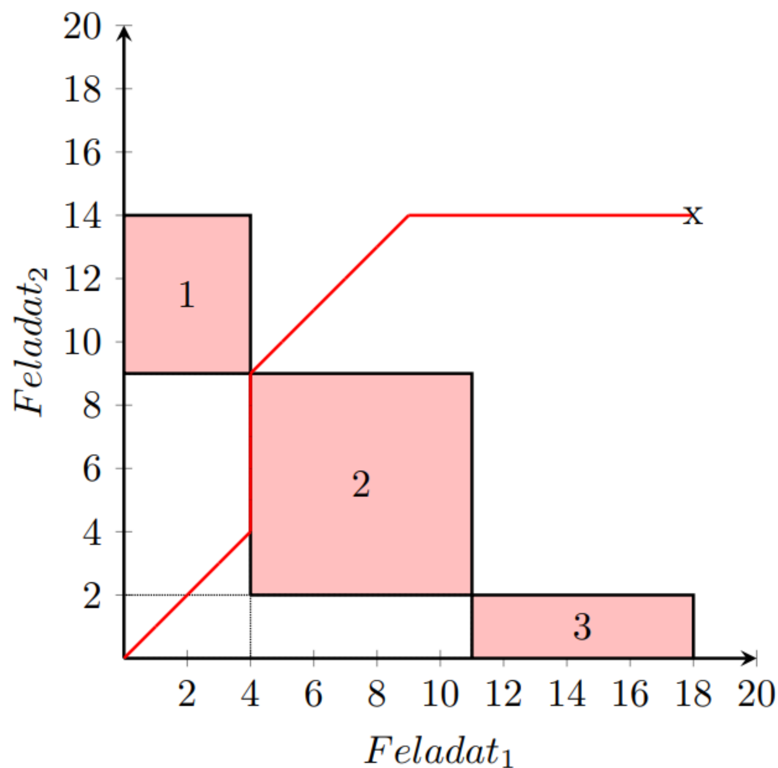
A feladatok végrehajtási ideje a következő; első feladaté 15, míg a másodiké 14, így 15 lesz a legtöbb időbe kerülő feladat végrehajtási ideje ha az megszakítás nélkül tud haladni. A módszer által meghatározott optimális ütemezés 19 hosszúságú/időegységű ütemezést határozott meg, így ennek a növekménye $N_2 = 4/15$.

Harmadik példa bemenet:

1 feladat gép igénye sorrendben, alattuk a műveleti idő:				
1	2	3		
4	7	7		
2 feladat gép igénye sorrendben, alattuk a műveleti idő:				
3	2	1		
2	7	5		

36. ábra Harmadik példa bemenet (Forrás: Saját megszorításos inputgenerátorral generált bemenet)

Ábrázolása:



37. ábra Harmadik példa kimenet (Forrás: Saját Python kód Latex megjelenítéssel)

Itt sem lesz a növekmény nulla. Az első feladatnak 18 a végrehajtási ideje, míg a másodiknak 14, így a leghosszabb feladat végrehajtási ideje 18, míg az ütemezésünk 23 hosszú. A növekményünk pedig $N_3 = 5/18$.

4.1.2. A növekmények összegzése

A növekményeket összegezzük és elosztjuk hárommal őket, így megkapjuk az átlagos növekményt. Az inputokat egy inputgenerátorral hoztam létra, ami random sorsolja a feladatokhoz a gépek sorrendjét, illetve a gépigényeket (1-7-ig).

Az átlagos növekmény:

$$N_1 + N_2 + N_3 = 3/12 + 4/15 + 5/18 = 143/180$$

$$(143/180)/3 = 143/540 \text{ ami kerekítve } 0{,}26.$$

Tehát az Akers módszer átlagos növekménye 3 gép esetén és 3 példa alapján 0,26.

4.1.3 Kiterjesztés átlagos növekménye

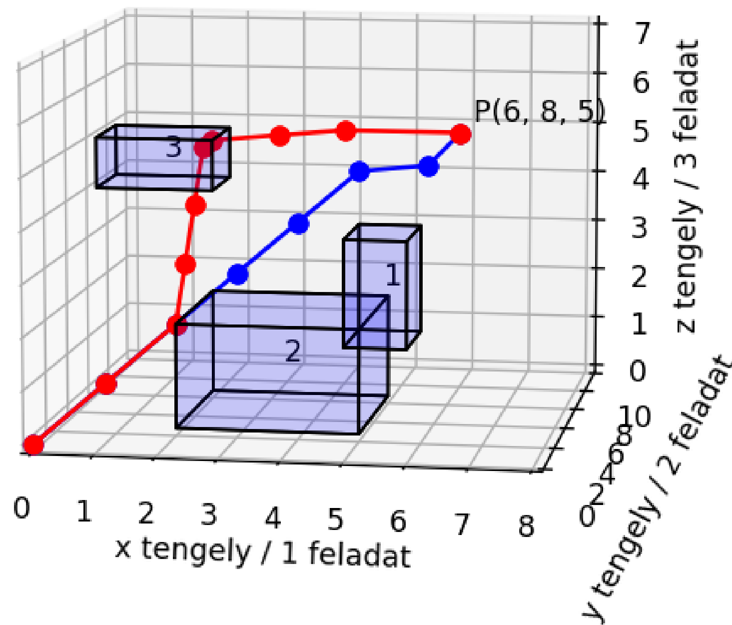
Itt is 3 gépes problémákon nézzük meg az átlagos növekményt, illetve szint úgy három példából vonjuk az átlagunkat.

Első példa bemenete:

1. feladat gép igénye sorrendben, alattuk a műveleti idő:				
3	2	1		
2	3	1		
2. feladat gép igénye sorrendben, alattuk a műveleti idő:				
1	2	3		
2	4	2		
3. feladat gép igénye sorrendben, alattuk a műveleti idő:				
2	1	3		
2	2	1		

38. ábra Első példa bemenet (Forrás: Saját megszorításos inputgenerátorral generált bemenet)

Ábrázolása:



39. ábra Első példa kimenet (Forrás: Saját kód, Python matplotlib könyvtár használatával)

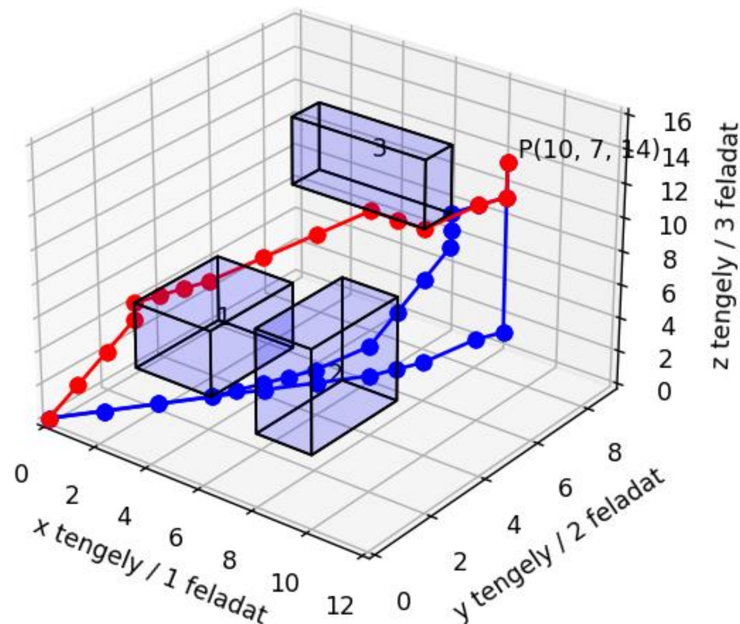
A példa a fentebb bemutatott eset (lásd 3.4 Algoritmus Példa), ezen nézzük meg elsőnek az átlagos növekményt. Az algoritmus pirossal jelöli az optimális útvonalat, vagyis az optimális ütemezést. A leghosszabb feladat futási ideje 8, míg az algoritmusunk végrehajtási ideje az 10, vagyis a növekménye $N_1 = 2/8$.

Második példa bemenete:

1. feladat gép igénye sorrendben, alattuk a műveleti idő:				
1	3	2		
3	5	2		
2. feladat gép igénye sorrendben, alattuk a műveleti idő:				
2	1	3		
3	3	1		
3. feladat gép igénye sorrendben, alattuk a műveleti idő:				
1	2	3		
4	6	4		

40. ábra Második példa bemenet (Forrás: Saját megszorításos inputgenerátorral generált bemenet)

Ábrázolása:



41. ábra Második példa kimenet (Forrás: Saját kód, Python matplotlib könyvtár használatával)

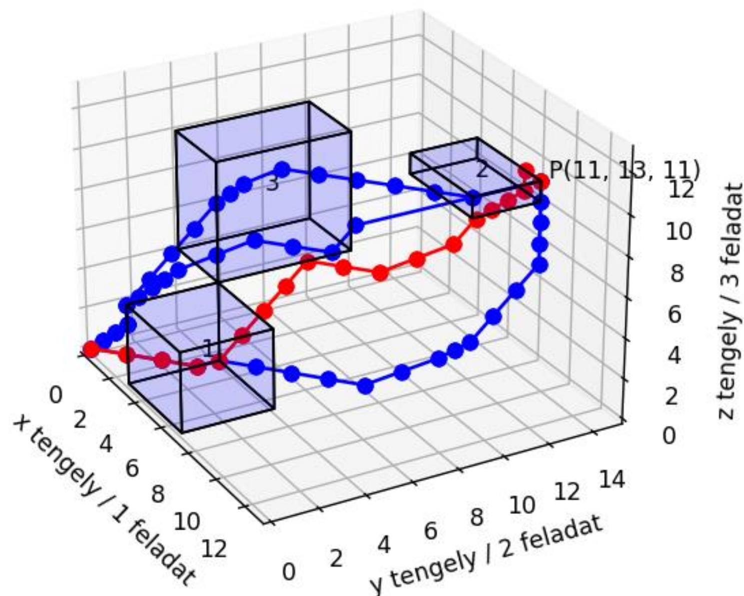
Az Algoritmus továbbra is pirossal jelöli az optimális ütemezést. A leghosszabb feladat futási ideje az 14 időegység, míg az algoritmusunk 16 hosszú ütemezést adott, ami azt jelenti, hogy a növekményünk az $N_2 = 2/14$.

Harmadik példa bemenete:

1. feladat gép igénye sorrendben, alattuk a műveleti idő:				
3	1	2		
3	4	4		
2. feladat gép igénye sorrendben, alattuk a műveleti idő:				
1	3	2		
4	6	3		
3. feladat gép igénye sorrendben, alattuk a műveleti idő:				
1	3	2		
4	6	1		

42. ábra Harmadik példa bemenet (Forrás: Saját megszorításos inputgenerátorral generált bemenet)

Ábrázolása:



43. ábra Harmadik példa kimenet (Forrás: Saját kód, Python matplotlib könyvtár használatával)

A második feladatunk a leghosszabb, aminek 13 lenne a futási ideje, ha nem lenne más feladat is. Az ütemezés, amit az algoritmusunk ad, az viszont 18 hosszú, így a növekményünk az N_3 = 5/13.

4.1.4. A növekmények összegzése

A növekményünket itt is összegezzük, mint Akers-nél, majd átlagoljuk, hogy megkapjuk a 3 gépes problémákra az átlagos növekményt.

$$N_1 + N_2 + N_3 = 2/8 + 2/14 + 5/13 = 283/364$$

$$(283/364)/3 = 283/1092 \text{ ami kerekítve } 0{,}26.$$

Ami pontosan annyi, mint az Akers módszerénél 2 gép esetén, így mondhatjuk, hogy a kiterjesztés ezek alapján a példák alapján nem okoz többlet növekményt. Itt is csak, mint Akers módszerénél saját random generátort használtam a bemenetek előállításához. Természetesen ez nem feltétlen mindig igaz, hogy pont ugyan annyi a növekmény, de nekünk a példáink alapján mégis ez látszik.

4.2. Növekmények összehasonlítása/összevetése

Az alábbi táblázatban összefoglaltam a fentebbi pontokban megkapott eredményeket.

	Akers módszer			Kiterjesztés 3 feladatra		
	Problémák	Növekmények	Értékük	Problémák	Növekmények	Értékük
	1. példa	N1	1/4	1. példa	N1	1/4
	2. példa	N2	4/15	2. példa	N2	1/7
	3. példa	N3	5/18	3. példa	N3	5/13
Összeg:	-	-	143/180	-	-	383/364
Átlag:	-	-	143/540	-	-	283/1092
Kerekítve:	-	-	0.26	-	-	0.26

44. ábra Növekmények összegzése (Saját szerkesztés)

A táblázat alapján látszik, hogy Akers módszeréhez képest nem jár nagyobb növekménnyel a 3 feladatra való kiterjesztés sem, hiszen random generált problémák megoldásánál, pontosan ugyan annyi lett a növekmény, ami természetesen, nem lesz mindig ugyan annyi, de látszik, hogy nem fog nagyságrendekkel eltérően más megoldást adni. Így ezzel is szemléltethető, hogy a kiterjesztésünk optimális, hiszen egy másik optimális algoritmussal összevetve nem jár nagyobb növekménnyel.

5. ÖSSZEGZÉS

A Dolgozatom célja, az Akers által bemutatott módszer, kiterjesztése volt 3 feladatra. Akers algoritmusának bemutatása után tettem javaslatot a kiterjesztésnek a lehetőségére, majd beláttuk, hogy további szabályok alkalmazása szükségesek. A kiterjesztett algoritmus bemutatása után, végig vettem az összes lépését sorról sorra, ami szerint haladni kell, majd bemutattam a pszeudo kódját is, ami alapján már nem volt nehéz a megvalósítás. Ezek után a lépései, és a pszeudokódja segítségével bemutattam egy példán keresztül a működést, aminek eredménye képen két lehetséges ütemezés közül az algoritmus megadta az optimális ütemezést. Ez nem volt elég, így összehasonlítottam az Akers módszernek az átlagos növekményével, és azt tapasztaltuk, hogy közel megegyezik, így beláthatjuk, hogy az algoritmusunk is optimális megoldást ad. A növekmény tesztelése közben teszteltem az algoritmust komplexebb példákon keresztül ahol akár 4 ütemezést is adott, amiből kiválasztotta a legrövidebbet, ami az optimális megoldásnak bizonyult. Tehát az algoritmus az komplexebb példákon is megtalálja a megoldást, így a dolgozatom célját elértem, és sikerült Akers módszerének kiterjesztése 3 feladatos problémákra.

5.1. Továbbfejlesztési lehetőségek

Továbbfejlesztési lehetőségen is gondolkodtam. Háromnál több dimenzióba már nem lehetne kiterjeszteni, vagyis igen nehéz lenne azt ábrázolni, illetve elképzelni. Ha az egy ábrán való ábrázolástól eltekintünk, akkor elképzelhető, egy olyan tovább fejlesztés, amikor több feladat is van. Mindegyik feladatot ábrázoljuk mindegyik feladattal az Akers módszer szerint, ami n*(n-1)/2 darab ábrázolás, így 4 feladat esetén már 6 illetve 5 feladat esetén 10 Akers módszert kell megoldani, kisebb kiegészítésekkel, vagyis figyelembe kell venni mindegyik síkon a többi sík adta lehetőséget egy feladat léptetésénél. Így elképzelhető további fejlesztés, és kiterjesztés is.

Illetve nem csak a feladatok ütemezésében lehet még fejlesztésekben gondolkodni, hanem akár lehetne egyes feladatokat kiemelni, vagyis azt mondani, hogy egyes feladatok prioritást élveznek, így gyorsabban készen kell, hogy legyenek. Ezt úgy lehet megvalósítani, hogy az algoritmusunk mikor több lehetséges útvonal közül adja meg az optimálisat, akkor az elején beállított prioritások alapján más ütemezést választ. Vagy akár a futásánál is mindig azt a lépést részesíti előnybe, amelyikkel mindig halad a prioritásba helyezett feladat. Persze ennek a priorizálásnak ára van, mégpedig, hogy nem feltétlen kapunk így optimális megoldást a teljes rendszerre nézve, de mégis egy megfelelő ütemezés lesz a prioritást tartva szem előtt.

IRODALOMJEGYZÉK

A.S., J., & S. Meeran. (1999). Deterministic job-shop scheduling: Past, present and future. *European Journal of Operational Research Vol. 113, Issue 2*, 390-434.

Jien, X., Xinyu Li, Liang Gao, & Lin, G. (2021). *A new neighboorhood structure for job shop scheduling problems.* Wuhan: Huazhong University of Science and Technology.

M. R., G., D. S., J., & R., S. (1976). The complexity of flow shop and job-shop scheduling. *Mathematics od Operations Research 1*, 117-129.

Peter, B., & Bernd, J. (1993). A new Lower bound for the job-shop scheduling prodblem. *European Journal of Operational Research 64*, 156-167.

Peter, B., Yu N., S., & Frank Werner. (2007). Complexity of shop-scheduling problems with fixed number of jobs: a survey. *Math. Meth. Oper. Res. 65*, 461-481.

Rodammer, F., & White Jr., K. (1988). A Recent Survey of Production Scheduling. *IEEE Transactions on system, man, and cybernetics vol18, no. 6*, 841-851.

S. C., G. (1981). A review of production scheduling. *Operations research volume 29 issue 4*, 628-645.

Sheldon B Akers Jr. (1956). A Graphical Approach to Production Scheduling Problems. *Operations Research 4*, 244-245.

Szikora, B. (dátum nélk.). Jegyzet. *Termelésinformatika 5.48 változat*. Budapesti Műszaki és Gazdaságtudományi egyetem.

ÁBRAJEGYZÉK

1. ábra Flow Shop lehetséges bemenete 6
2. ábra Szalagrendszerű ütemezés nem optimális megoldás ábrázolása 6
3. ábra Szalagrendszerű gyártás optimális ütemezésének ábrázolása 7
4. ábra Műhelyrendszerű ütemezés bementi példa 8
5. ábra Műhelyrendszerű ütemezés lehetséges kimeneti ábrázolása 8
6. ábra Akers módszer ábrázolása 10
7. ábra Akers módszerének kiterjesztése hibás feltételezéssel 12
8. ábra Akers kiterjesztésének nehézsége I. 13
9. ábra Akers kiterjesztésének nehézsége II. 14
10. ábra Akers kiterjesztés problémájának feloldása 14
11. ábra V_1(1, 1, 1) irány 16
12. ábra V_2, V_3, V_4, irányok 16
13. ábra V_5, V_6, V_7 irányok 17
14. ábra Kiterjesztés Pszeudokódja 18
15. ábra Példa bemenet 19
16. ábra Gépekhez tartozó idők letiltása 19
17. ábra Cél pont meghatározása 20
18. ábra 1. lépés 20
19. ábra 2. lépés 21
20. ábra 3. lépés, illetve elágazás 22
21. ábra 4. lépés 22
22. ábra 5. lépés 23
23. ábra 6. lépés 23
24. ábra Első vonal ábrázolása 24
25. ábra Második vonal ábrázolása 24
26. ábra Teljes ábrázolás 25
27. ábra Első eset ábrázolása Gnatt-diagrammal 25
28. ábra Második eset ábrázolása Gnatt-diagrammal 25
29. ábra Sík vetületek 26
30. ábra Kiterjesztés sík vetületei 26
31. ábra Akers algoritmus futása 27
32. ábra Első példa bemenet 28

33. ábra Első példa kimenet 29
34. ábra Második példa bemenet 29
35. ábra Második példa kimenet 30
36. ábra Harmadik példa bemenet 30
37. ábra Harmadik példa kimenet 31
38. ábra Első példa bemenet 32
39. ábra Első példa kimenet 32
40. ábra Második példa bemenet 33
41. ábra Második példa kimenet 33
42. ábra Harmadik példa bemenet 34
43. ábra Harmadik példa kimenet 34
44. ábra Növekmények összegzése 35