

Miskolci Egyetem
Gépészmérnöki és Informatikai Kar
Informatikai Intézet
Alkalmazott Informatikai Intézeti Tanszék



ERŐFORRÁS-KORLÁTOS PROJEKTÜTEMEZÉSI FELADATOK MODELLEZÉSE ÉS SZÁMÍTÓGÉPES MEGOLDÁSA

TDK DOLGOZAT

Készítette:

Venczel Tamás Bence

QDK6YO

Mérnök informatikus BSc hallgató

Konzulens:

Dr. Kulcsár Gyula

egyetemi docens

Alkalmazott Informatikai Intézeti Tanszék

2014, Miskolc

Tartalomjegyzék

I.	BEVEZETÉS	1
II.	ELMÉLETI HÁTTÉR	2
2.1	Termelésinformatikai háttér	2
2.1.1	Az iparvállalatok informatikai fejlődése	2
2.1.2	A termelésinformatika	3
2.1.3	A virtuális vállalat (Virtual Enterprise)	4
2.2	Projektütemezés	5
2.2.1	Projektütemezés elméleti háttére	5
2.2.2	Projektütemezési feladatok ábrázolása	6
2.2.3	Projektütemezési feladatok erőforráskorlát nélkül	10
2.2.4	Kritikus útvonal módszer (Critical Path Method)	12
2.2.5	Projektütemezési feladatok erőforráskorláttal	15
III.	FELHASZNÁLT SZOFTVER-TECHNOLÓGIÁK	17
3.1	A JAVA programozási nyelv	18
3.1.1	Áttekintés, történelmi háttér	18
3.1.2	Tulajdonságai	19
3.2	A fejlesztési és tesztelési környezet	20
3.2.1	PC	20
3.2.2	Eclipse integrált fejlesztői környezet	21
3.2.3	Maven	21
3.2.4	JAX-B	22
3.2.5	Git / Bitbucket	22
IV.	IMPLEMENTÁCIÓ	23
4.1	Áttekintés	23
4.1.1	Tesztvezérelt fejlesztés (TDD)	24
4.2	Alkalmazás logika	25
4.3	Ütemező algoritmusok	26
4.4	Osztálydiagram	27
4.5	Osztályok, interfészek	27

V.	MŰKÖDÉS ÉS KIÉRTÉKELÉS	32
	Áttekintés	32
5.1	Erőforrás-korlát nélküli projektütemezési feladatok	32
5.1.1	Funkcionális teszt.....	33
5.1.2	Futási eredmények.....	36
5.2	Erőforrás-korlátos projektütemezési feladatok	38
5.2.1	Funkcionális teszt.....	38
5.2.2	Futási eredmények.....	40
5.3	Összegzés	43
VI.	ÖSSZEFOGLALÁS	44
VII.	KÖSZÖNETNYILVÁNÍTÁS	45
	IRODALOMJEGYZÉK	46

I. BEVEZETÉS

A TDK dolgozat tárgya az erőforrás-korlátos projektütemezési feladatok modellezése és számítógépes megoldása. A témakör az alkalmazott informatikán belül a termelésinformatika, a diszkrét termelési folyamatok számítógépes tervezésére és irányítására terjed ki, ezért a feladat megoldása előtt elengedhetetlen a termelésinformatikában használatos alapfogalmak és modellek ismertetése.

A dolgozat során a témakörben való elmélyedés, az új kutatási eredmények megismerése és az abból feltárt következtetések ismertetése mellett a fő cél az, hogy felállítsak egy jól használható és adott esetben bővíthető modellt, mellyel a jelenleg kitűzött feladatokon túlmutató irányok felé is lehetőség adódik a program továbbfejlesztésére.

Az első részben ismertetem a téma megértéséhez elengedhetetlen elméleti tudnivalókat, a projektütemezés jellemzőit, sajátosságait, problémáit, a lehetséges megoldásokat, majd a megszerzett ismeretek alapján felállítok egy modellt, mellyel az általam vizsgált ütemezési problémák modellezhetők, majd – lehetőség szerint - megoldhatók.

Ezután a meglévő modellt kiterjesztem az erőforrás-korlátos projektütemezési feladatokra, mely alapján jelentősen bonyolultabb jellegű ütemezési feladatok is megoldhatók, felhasználva az alkalmazott informatikai módszerek által nyújtott lehetőségeket.

A továbbiakban az elkészült alkalmazást mutatom be, részletezve a téma és a modellezéséhez szükséges programozási feladatok közötti leképzés folyamatát, majd különböző teszt adatokkal ellátva tesztelem a működését, melyekből a kellő következtetések levonása után igyekszem igazolni a kutatómunka, a modell és az alkalmazás hatékonyságát.

II. ELMÉLETI HÁTTÉR

2.1 Termelésinformatikai háttér

2.1.1 Az iparvállalatok informatikai fejlődése

A számítógépek egyre nagyobb mértékű elterjedése során, a teljesítmény, ezáltal a számítási kapacitás és a kezelhető adatmennyiség óriási mértékben növekedett a korszerű informatikai megoldásoknak köszönhetően.

Az iparvállalatok kihasználva a rendkívül gyors fejlődést igyekeztek minél hamarabb integrálni a meglévő adatstruktúrákat valamilyen – informatikai eszközökkel kezelhető – más formába, lecserélve az akkoriban főként papír alapú rendszereket.

Ennek, valamint az informatikai hálózatok megjelenésének következtében a rendelkezésre álló információ mennyisége folyamatosan növekedett, melynek kezelése egy idő után komoly informatikai kihívást jelentett, hardver és szoftver tekintetében egyaránt.

Ekkor merült fel az igény olyan rendszerekre, melyek képesek ellátni az adatok rendszerezési, adminisztratív feladatait, valamint képesek a folyamatokat könnyen elérhető formában reprezentálni, és használható formában rendelkezésre bocsájtani.

Az egyre komolyabb feldolgozó egységek, tároló elemek, grafikus teljesítményű hardverek, valamint a modern szoftveres megoldások indokolták, hogy az új eredményeket az iparban is felhasználják a munka támogatására, a mérnöki, gazdasági és egyéb feladatok egyszerűsítése, gyorsítása érdekében.

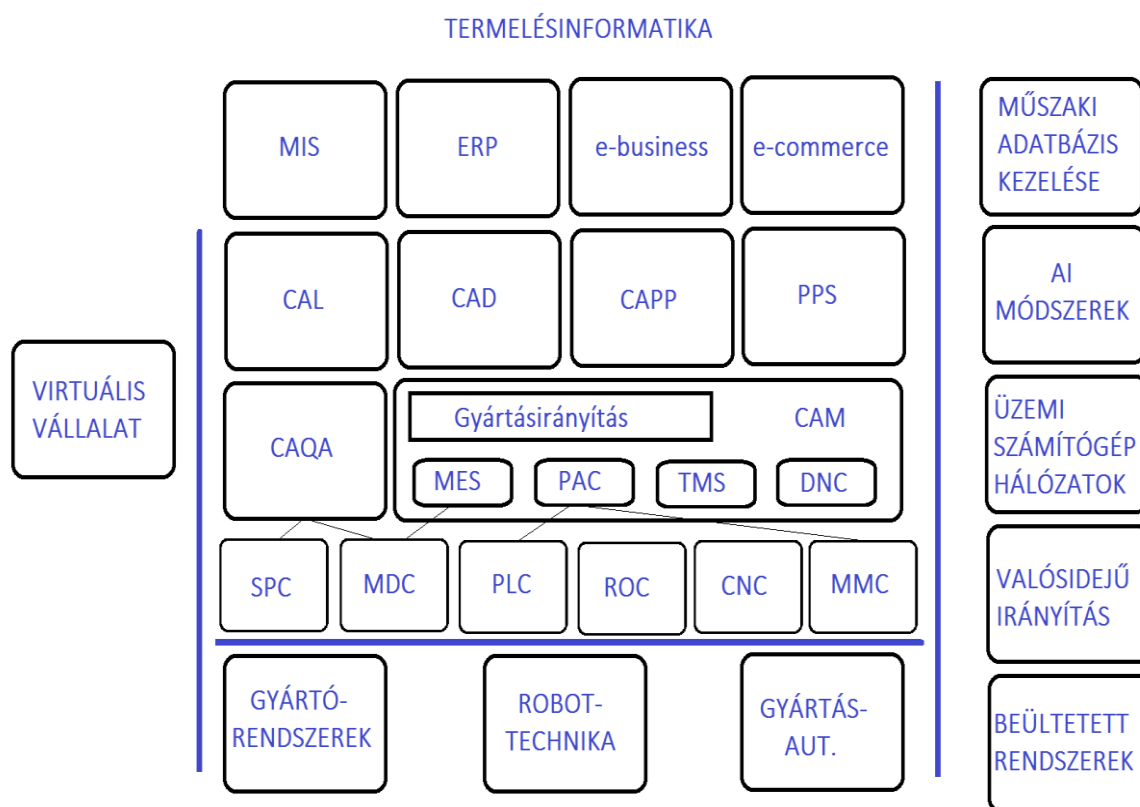
Ennek következtében különböző gyártástámogató, ütemező, adatbázis kezelő, támogató rendszerek fejlesztése indult el. Így kezdett kialakulni az informatika egyik új ága, a termelésinformatika.

2.1.2 A termelésinformatika

Az emberiség által használt fogyasztási javak tervszerű és sokszorozott előállítását a technológia és tudomány mindenkori szintjén nevezzük termelésnek, mely magában foglalja az adott termék előállításától kezdve, a nyújtott szolgáltatásokon keresztül a műszaki előkészítést és a minőségbiztosítást is.

A termelésinformatikát (*Production Information Engineering*) az informatika, azon belül az alkalmazott informatika egyik ágának tekintjük, mely felhasználja az információelmélet és számítástudomány, valamint az alkalmazott informatika módszereit, eszközeit, melyeket a megfelelő műszaki és gazdasági tudományok ismeretanyagával vegyítve modellezi, tervezi és irányítja a termelést. ^[1]

A termelésinformatika területén több alkalmazási szint különböztethető meg, a problémáktól függően, melyeknek elosztását az alábbi ábra mutatja be:



2.1. ábra: A termelésinformatika modellje ^[2]

2.1.3 A virtuális vállalat (Virtual Enterprise)

A dolgozatom témája alapvetően a termelésinformatika területén belül a virtuális vállalat témakör halmazába besorolható problémákat érinti, ezért főként ennek az egységnek a vázlatos ismertetését tartom célszerűnek.

A globalizáció során az internet kialakulása és hatalmas iramú fejlődése, valamint a kommunikációs sebesség növekedése miatt a gazdasági élet feltételeiben jelentős minőségi változás volt megfigyelhető. Ez a változás a vállalatok számára azt jelentette, hogy az információ és annak áramlása az addiginál még fontosabb szerepet töltött be a munkavégzés során, a döntési folyamatok jelentősen felgyorsultak, távoli szervezetek is könnyedén kommunikálhattak egymással valós időben, a különálló modulok egy egységet alkotva kommunikáltak a vállalaton belül és kívül is, így alkotva egy virtuális vállalatot.

A virtuális vállalat, mint fogalom, több nézőpontból is értelmezhető, a szakirodalmi értelmezések szerint az alábbi definíciók a legelfogadottabbak a komponensek leírására:

- Vállalatközi integráció paradigmája
- Vállalaton belüli integráció paradigmája
- Átfogó paradigma

Definíció szerint a *vállalatközi integráció* fogalma:

„ A VE független autonóm szervezetek (vállalatok, leányvállalatok, stb.) olyan, elektronikus információs rendszerre és szervezett integrációra alapozott időszakos együttműködési rendszere, amely lehetővé teszi, hogy a résztvevő szervezetek képesek legyenek hatékonyan működtetni több erőforrást, mint ami az egyes szervezeteknél rendelkezésre áll, jelentős expanzió nélkül. ” [3]

Definíció szerint a *vállalaton belüli integráció* fogalma:

„ A VE autonóm funkcionális szervezetek (telephelyek, osztályok, gyáregységek, stb.) olyan, elektronikus információs rendszerre és szervezett integrációra alapozott folyamatos együttműködési rendszere, amely lehetővé teszi, hogy a résztvevő szervezetek képesek legyenek hatékonyan működtetni osztott erőforrásokat, az egyes szervezeteknél fizikailag rendelkezésre álló erőforrások jelentős bővítése nélkül. ” [4]

Definíció szerint az *átfogó paradigma* megfogalmazása:

„A VE olyan, világhálóra alapozott, elektronikus információs rendszerre és szolgáltatásokra épülő cégműködés, amely lehetővé teszi, hogy a partner szervezetek és vevők képesek legyenek hatékonyan hozzáférni adat- és szolgáltatás jellegű erőforrásokhoz, üzleti folyamatokat kezdeményezni és biztonságosan lefolytatni az erőforrások fizikai felkeresése, elérése nélkül.” ^[5]

2.2 Projektütemezés

2.2.1 Projektütemezés elméleti háttere

A vállalat működése során úgynevezett projekteket különböztethetünk meg, melyek általában valamilyen összetett, nagyméretű feladatot reprezentálnak, s általában nagyszámú feladat végrehajtását foglalják magukban. A feladatokon végighaladva a projekt elvégezhető lesz és eredményeképpen megvalósul a projekt által kitűzött célra irányuló törekvés. A projektütemezés széles körben használt módszer a feladatalapú problémák megoldásához, alkalmazzák például többek között a termelés, kutatás és fejlesztés, tervezés, építészet, menedzsment, szoftverfejlesztés, karbantartás, telepítés, fenntartás területein.

A projektütemezés jelenti azt a folyamatot mely során a projekt (vagy akár egyszerre több projekt) feladatainak időbeli végrehajtását úgy tervezzük meg, hogy a kitűzött célok teljesülhessenek, figyelembe véve a lehetséges korlátozásokat, kockázatokat és egyéb belső és külső tényezőket, melyek a projekt végrehajtását befolyásolhatják (például korlátozottan rendelkezésre álló, nagyszámú erőforrások; futás közben megszakítható folyamatok; folyamatirányító diszpécser, mely a feladatokat fontosság szerint osztályozza és értékeli).

A témakör kutatása több mint ötven évre tekint vissza, számos tanulmány született a területen, a feladatok komplexitásából következően alapvetően a heurisztikus és meta-heurisztikus megoldási módszerek kerültek előtérbe. Az ilyen problémákat kivételes esetben lehet egzakt megoldási módszerekkel kezelni. A projektütemezési feladatok nagyon sokfélék lehetnek^[6]. Az egyik erőteljesen kutatott probléma típus az erőforrás korlátos projektütemezési feladat (*Resource Constrained Project Scheduling, RCPS*).

Ezekről a feladat típusokról számos jó összefoglaló található a szakirodalomban, ilyen például a Kolisch és Hartmann által végzett munka^[7] valamint a Briskin és Hartmann^[8] által készített tanulmány.

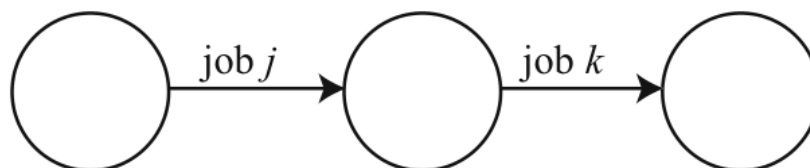
A projektet alkotó feladatok között megelőzési reláció van, ezért a feladatok egy hálózata alakul ki, ami a projekt végrehajtási vázát alkotja. Ezek a feladatok erőforrásokra is igényt tarthatnak, melyek különböző módon állhatnak rendelkezésre, ezért az ütemezés során ezekre a korlátokra tekintettel kell lennünk.

A projektütemezési feladatok komplexitásából adódik, hogy egy általános ütemezési modellnek tekinthető, melyből az egyéb ütemezési feladatok kiegyszerűsíthetők és megoldhatók. Így például a megfelelő korlátozások bevezetésével egy klasszikus *job-shop*, *flow-shop*, *open-shop* vagy egyéb ismert ütemezési probléma az erőforrás-korlátos projektütemezési feladat egy-egy speciális esetének tekinthető.

2.2.2 Projektütemezési feladatok ábrázolása

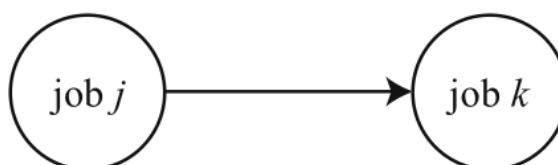
Az aktivitások közötti megelőzési relációk, és ez által az egész projekt reprezentálása több módon is történhet, ezek közül a szakirodalom főleg két ábrázolási módot használ:

- a) ***job-on-arc*** ábrázolási forma



2.2. ábra: *job-on-arc* ábrázolás^[9]

- b) ***job-on-node*** ábrázolási forma



2.3. ábra: *job-on-node* ábrázolás^[10]

A *job-on-arc* ábrázolásban az élek ábrázolják a *job*-okat (aktivitás, végrehajtható feladat), a csomópontok pedig a határokat jelölik ki. A fenti ábrában (2.2. *ábra*) látható, hogy a „j” jelű aktivitást a „k” jelű aktivitás követi. Az első csomópont a „j” feladat indítási idejét mutatja, a második csomópont a „j” feladat befejezési idejét és egyben a „k” feladat kezdési idejét jelzi, a harmadik csomópont pedig a „k” feladat befejezési idejét.

A *job-on-node* ábrázolási formában a csomópontok jelentik az aktivitásokat, a köztük lévő él pedig a sorrendiséget, ahogy az ábrán is látható (2.3. *ábra*).

Bár a *job-on-arc* ábrázolás széles körben használt, gyakran szükséges úgynevezett „dummy job”-ok bevezetése a formában, ugyanis a sorrendiség csak ezek segítségével lesz pontosan ábrázolható.

A két forma közti különbséget és a *dummy job*-ok használatának szükségességét az alábbi példa szemlélteti.

Példa 2.2.2 (egy új termék bevezetésének folyamata egy fiktív vállalatnál)

Az alábbi táblázat tartalmazza az elvégzendő feladatokat és azok adatait:

<i>Job (ID)</i>	<i>Leírás</i>	<i>Átfutási idő (p_j)</i>
1	Szerszámok tervezése	4 hét
2	Műszaki rajzok elkészítése	6 hét
3	Gyártóegység felkészítése az új termékhez	10 hét
4	Szerszámok beszerzése	12 hét
5	Alkatrészek beszerzése	10 hét
6	Alkatrészek szerelése	2 hét
7	Eszközök testre szabása	4 hét
8	Tesztelés	2 hét

2.1. táblázat: Példa projekt aktivitásai ^[11]

A táblázat első oszlopában az egyes feladatok azonosítói láthatóak, a második oszlopban az elvégzendő feladatok leírása, a harmadik oszlopban pedig az adott művelet végrehajtási ideje jelenik meg.

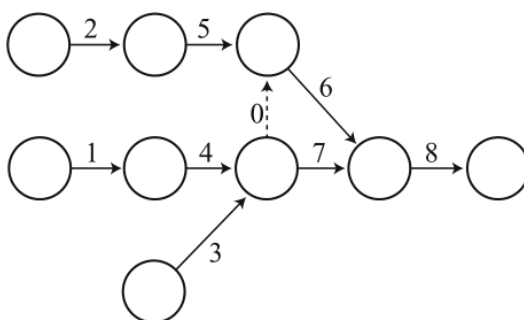
A feladatok és a folyamat ismeretében meghatározható hozzájuk a megelőzési relációk, melyeket az alábbi táblázat ismertet:

<i>Job (ID)</i>	<i>Megelőző aktivitások</i>	<i>Rákövetkező aktivitások</i>
1	-	4
2	-	5
3	-	6, 7
4	1	6, 7
5	2	6
6	3, 4, 5	8
7	3, 4	8
8	6, 7	-

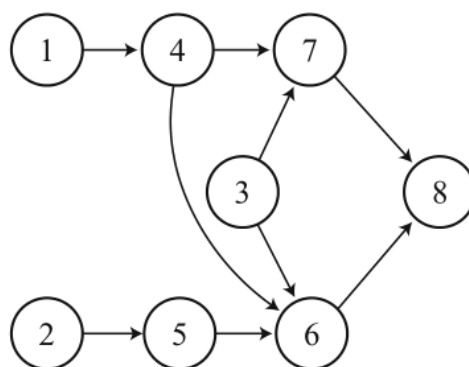
2.2. táblázat: Példa projekt, feladat sorrendiségi táblázat ^[12]

A táblázat első oszlopában az aktivitások azonosítói szerepelnek, a második oszlopban az adott feladatot közvetlenül megelőző aktivitások (*predecessors*), a harmadikban pedig az aktuális feladatot közvetlenül követő (*successors*) aktivitások találhatók.

Ezen adatok alapján felírható egy *precedencia gráf*, mellyel ábrázolható a feladatok végrehajtási lánc. A két ábrázolási mód szerinti gráf:



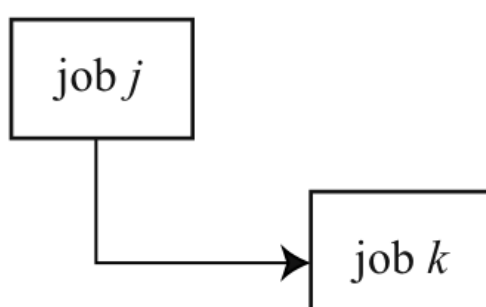
2.4. ábra: Példa projekt, job-on-arc ábrázolás ^[13]



2.5. ábra: Példa projekt, job-on-node ábrázolás ^[14]

A példa alapján jól látható a job-on-arc forma nagy hátránya, a dummy-jobok helyenkénti szükségessége. Ez főként akkor okoz problémát, ha nagy méretű projektekkel foglalkozunk, ugyanis a dummy-jobok akár a projektet alkotó aktivitások 10%-át is jelenthetik, amely megnehezíti a nyilvántartást, valamint a számításokat is nehezebben és lassabban feldolgozhatóvá teszi.

A job-on-node ábrázolás egy másik előnye (a dummy jobok nélkülözhetősége mellett), hogy ha a csomópontokat téglalapokként ábrázoljuk, akkor a téglalap vízszintes oldala jelzi, hogy egy aktivitás successora az aktivitás életciklusa alatt is elkezdődhet, mint ahogyan az alábbi ábra mutatja. Ebben az esetben a „k” jelű aktivitás már akkor kezdődhet, amikor a „j” aktivitás a futása felénél jár.



2.6. ábra: Job-on-node ábrázolás téglalapokkal ^[15]

A fenti ismeretek birtokában úgy gondolom, a job-on-node ábrázolási forma előnyösebb, mint a job-on-arc forma, ezért a továbbiakban ennek a használatát részesítem előnyben, azonban ehhez szükséges a gráfra vonatkozóan korlátozásokat bevezetni.

A job-on-node ábrázolás szerinti gráfban:

- egy csomópont egy aktivitást jelöl,
- egy irányított él kötelező sorrendiséget ír elő két csomópont között,
- a gráfban nem lehet irányított körút,
- a gráfban nem lehet redundáns él,
- a gráfban nem lehet izolált csomópont (a virtuális forrás-és nyelőpontok bevezetésével a probléma kezelhető).

2.2.3 Projektütemezési feladatok erőforráskorlát nélkül

A projektütemezési feladatok egyik alapvető csoportját az erőforráskorlát nélküli feladatok alkotják, ebben az esetben feltételezzük, hogy:

- adott n számú feladat megelőzési relációkkal,
- minden egyes feladat végrehajtási idejét ismerjük,
- párhuzamosan állnak rendelkezésre korlátlanul erőforrások.

Ilyen ütemezési feladatok során az ütemezés célja a projekt befejezési időpontjának (makespan) minimalizálása. Ebben az esetben a feladat megoldató úgy, hogy zero időpillanattól indítva vesszük az összes olyan aktivitást, aminek nincs öt megelőző aktivitása, majd mindig, amikor egy feladat befejeződik, elindítjuk az összes olyan aktivitást, aminek a megelőzői már befejeződtek.

Ahhoz, hogy az algoritmust formálisan is megfogalmazhassuk szükséges bevezetni az alábbi leírást egy adott j feladat esetére:

- p_j – a j feladat végrehajtási ideje,
- S'_j - a j feladat legkorábbi lehetséges kezdési időpontja,
- C'_j - a j feladat legkorábbi lehetséges befejezési időpontja.

Ebben az esetben:

$$C'_j = S'_j + p_j$$

és jelöljük a j feladatot megelőző aktivitásokat:

$\{ \text{minden } k \rightarrow j \}$ alakban, amely azokat az aktivitásokat jelöli, amelyeknek be kell fejeződniük mielőtt a j aktivitás elindul.

Ennek az algoritmusnak az elnevezése az előre haladó eljárás (**Forward Procedure**) és a bevezetett jelölésrendszerrel az alábbi formában írható le: ^[16]

1. lépés:

Legyen

$$t = t_s \quad (\text{pl. } t_s = 0, \text{ az indítás referencia időpontja}).$$

A megelőző feladattal nem rendelkező minden egyes j feladat esetében legyen:

$$S'_j = t \text{ és } C'_j = t + p_j$$

2. lépés:

A megelőző feladattal rendelkező minden egyes j feladat esetében legyen induktív módon:

$$S'_j = \max_{\{ \text{minden } k \rightarrow j \}} C'_k \text{ és } C'_j = S'_j + p_j$$

3. lépés:

A legkorábbi projekt-befejezési időpont:

$$C_{\max} = \max \{ C'_1, \dots, C'_n \}$$

Könnyen belátható, hogy az algoritmus optimális ütemtervet ad és a projekt befejezési időpontja viszonylag egyszerűen kiszámítható. A módszerrel a feladatok mindig a lehető legkorábbi időpontban kezdődnek és fejeződnek be, viszont ez nem mindig feltétlenül szükséges a befejezési idő minimalizálásának szempontjából, előfordulhat, hogy egy aktivitást nem kell a lehető legkorábbi időpontban elindítani, ezért célszerű megállapítani az aktivitások legkésőbbi lehetséges indítási és befejezési időpontját is.

Ez az algoritmus a visszafelé haladó eljárás (**Backward Procedure**), mely kiindulási bemenete a $C_{\max}$. A visszafelé haladó eljáráshoz az alábbi formális jelölések bevezetése szükséges:

- S''_j - a j feladat legkésőbbi lehetséges kezdési időpontja
- C''_j - a j feladat legkésőbbi lehetséges befejezési időpontja

Jelöljük a j feladatot követő aktivitásokat:

$\{ j \rightarrow \text{minden } k \}$ alakban, amely azokat az aktivitásokat jelöli, amelyek csak a j aktivitás befejeződése után indulhatnak el.

A jelölések definiálása után az algoritmus az alábbi módon írható le: ^[17]

1. lépés:

Legyen

$$t = C_{max}$$

A rákövetkező feladattal nem rendelkező minden egyes j feladat esetében legyen:

$$C''_j = C_{max} \quad \text{és} \quad S''_j = C_{max} - p_j$$

2. lépés:

A rákövetkező feladattal rendelkező minden egyes j feladat esetében legyen:

$$C''_j = \min_{\{k \rightarrow \text{minden } j\}} S''_k \quad \text{és} \quad S''_j = C''_j - p_j$$

3. lépés:

Ellenőrizzük, hogy:

$$t_s = \min \{ S''_1, \dots, S''_n \}$$

A visszafelé haladó eljárás tehát kiszámolja az aktivitások legkésőbbi lehetséges kezdési és befejezési időpontját, figyelembe véve C_{max} értékét, úgy hogy a minimális átfutási idő elérhető legyen.

Az előbbieken ismertetett két algoritmus felhasználásával meghatározhatók a *kritikus aktivitások* és a *kritikus út*.

2.2.4 Kritikus útvonal módszer (Critical Path Method)

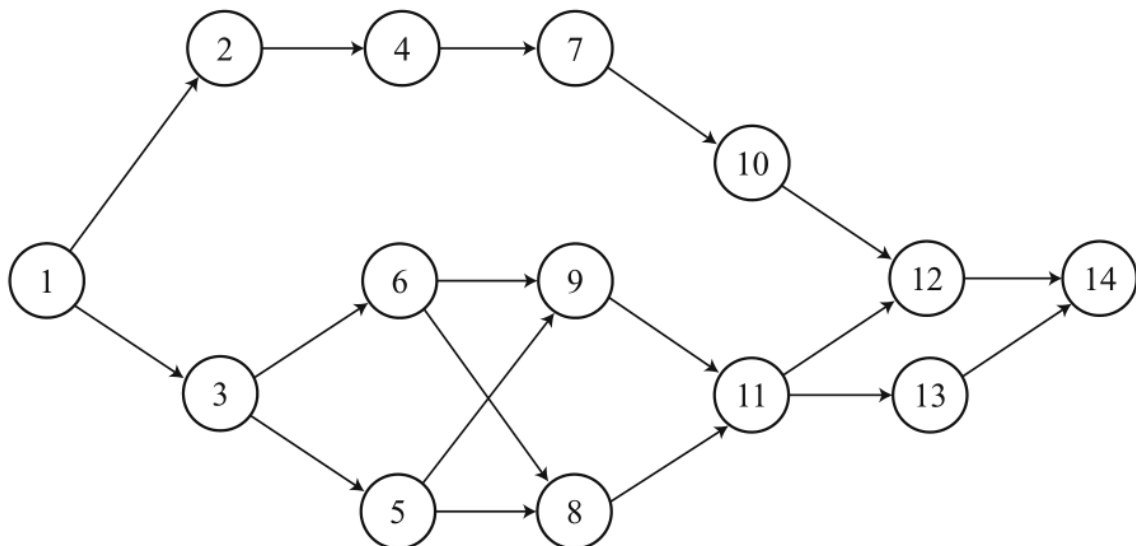
Az előre felé és visszafelé haladó eljárások tehát meghatározzák az aktivitások legkorábbi kezdési és befejezési, valamint legkésőbbi kezdési és befejezési időpontjait. Ha egy aktivitás legkorábbi kezdési időpontja kisebb, mint a legkésőbbi kezdési időpontja, akkor a feladatnak van tartaléka (*slack*). Azonban, ha egy aktivitás legkorábbi kezdési időpontja megegyezik a legkésőbbi kezdési időpontjával, akkor a feladat kritikus. A kritikus útvonal olyan kritikus feladatok láncolata, amely a kezdő csomóponttól a befejező csomópontig tart. Egy projekt több kritikus utat is tartalmazhat, a kritikus utak között átfedés is lehetséges. A következőkben egy példa található a fent ismertetett algoritmusok gyakorlati használatának demonstrálásaként.

Példa 2.2.4 (CPM módszer)

A példában adott 14 aktivitás, eltérő végrehajtási időikkel és sorrendiséggel. Az aktivitásokat és a végrehajtási idejüket a 9. ábra mutatja be, a köztük lévő megelőzési relációkat pedig a 10. ábra szemlélteti.

Jobs	1	2	3	4	5	6	7	8	9	10	11	12	13	14
p_j	5	6	9	12	7	12	10	6	10	9	7	8	7	5

2.3. táblázat: CPM módszer példa, aktivitások és végrehajtási idejük ^[18]



2.7. ábra: CPM módszer példa, precedencia gráf ^[19]

Az aktivitások idejének és sorrendiségének ismeretében az előre haladó eljárás megadja a legkorábbi kezdési időket:

Jobs	1	2	3	4	5	6	7	8	9	10	11	12	13	14
C'_j	5	11	14	23	21	26	33	32	36	42	43	51	50	56

2.4. táblázat: CPM módszer példa, legkorábbi kezdési idők ^[20]

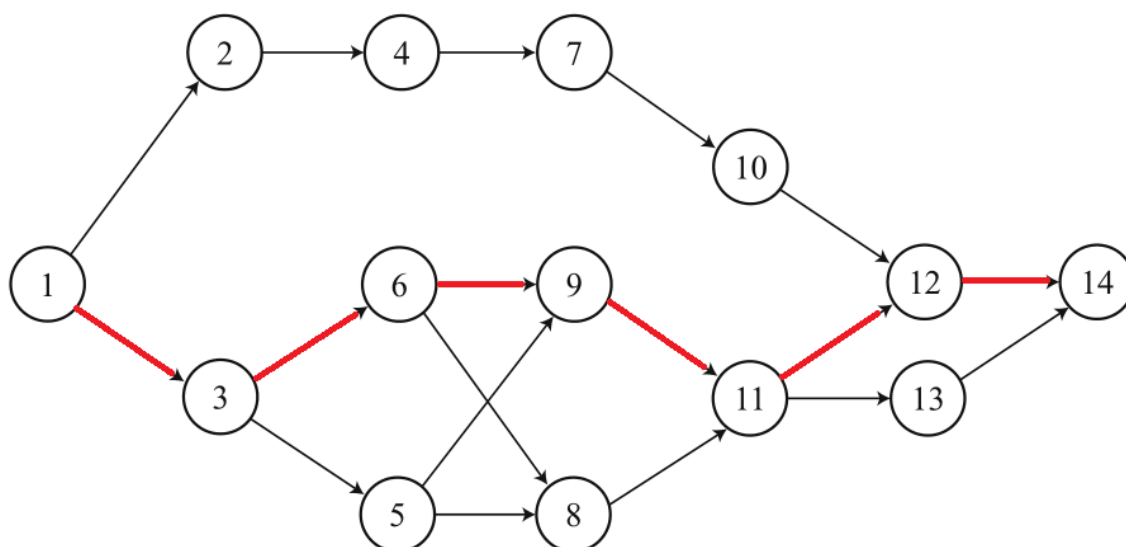
Az előre haladó eljárás alapján $C_{max} = 56$, ebből pedig a hátrafelé haladó eljárással kiszámíthatjuk az aktivitások legkésőbbi befejezési idejét:

Jobs	1	2	3	4	5	6	7	8	9	10	11	12	13	14
C_j''	5	12	14	24	30	26	34	36	36	43	43	51	51	56

2.5. táblázat: CPM módszer példa, legkésőbbi kezdési idők ^[21]

Az eljárások után megkapjuk azokat az aktivitásokat, amelyeknek a legkorábbi és legkésőbbi kezdési idejük megegyezik, ezek alkotják a kritikus utat:

$1 \rightarrow 3 \rightarrow 6 \rightarrow 9 \rightarrow 11 \rightarrow 12 \rightarrow 14$



2.8. ábra: CPM módszer példa, a kritikus út

2.2.5 Projektütemezési feladatok erőforráskorláttal

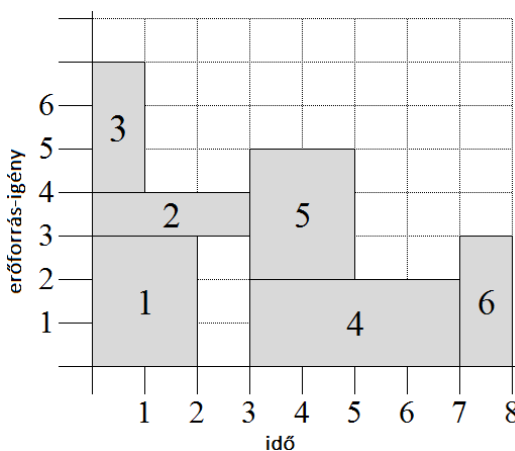
Az erőforrás korlátozott feladatoknál az aktivitások igényt tartanak valamilyen erőforrásra, amely a végrehajtásuk feltétele. Az erőforráskorlát nélküli projektütemezési feladatok esetében feltételeztük, hogy minden erőforrás korlátlanul, párhuzamosan áll rendelkezésre, ebben az esetben azonban az aktivitások igényt tarthatnak akár több erőforrásra is, mint az alábbi példában.

Példa 2.2.5 (erőforrás korlátozott feladat)

Job	p(j)	Predecessors	S'	C''	R(1,j)
1	2	-	0	3	3
2	3	-	0	3	1
3	1	-	0	6	2
4	4	1,2	3	7	2
5	2	2,3	3	8	3
6	1	4	7	8	3

2.6. táblázat: Erőforrás korlátozott feladat példa, adatok ^[22]

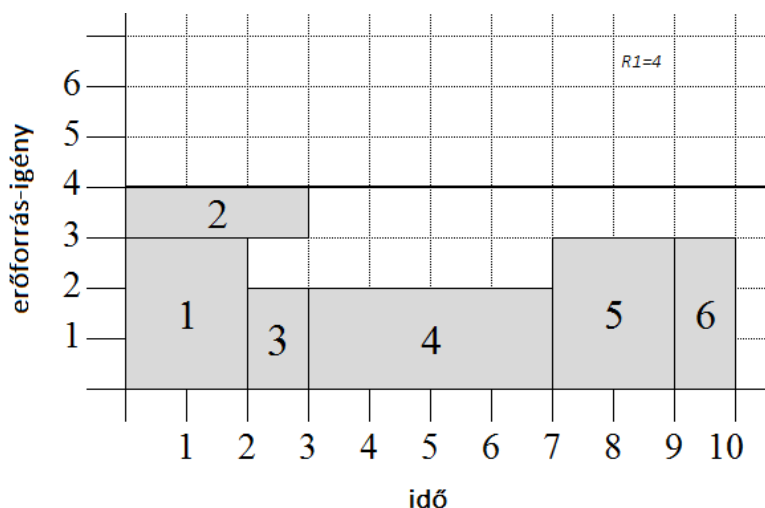
A táblázatban az első oszlop jelöli a feladatok azonosítóját, a második az átfutási idejüket, a harmadik a megelőző aktivitásait, a negyedik a legkorábbi indítási idejüket, az ötödik a legkésőbbi befejezési idejüket, a hatodik pedig az erőforrás igényüket. Az $R(1,j)$ jelölés jelenti az 1-es erőforrás és a j feladat közti függőséget. Az adatok alapján megjeleníthető a végrehajtási sorrend, figyelembe véve az erőforrás igényeket:



2.9. ábra: Erőforrás korlátozott feladat példa, ábrázolás korlát nélkül ^[23]

Jelenleg a feladatban nincs erőforráskorlát, tehát az aktivitások a szükséges erőforrást igényüknek megfelelően használhatják, akár egyszerre is, így a feladatot végrehajtva $C_{max} = 8$ lesz a projekt átfutási ideje.

Ez után tételezzük fel, hogy az $R1$ erőforrás csak 4 egységben áll rendelkezésre összesen minden időpillanatban, tehát $R1 = 4$ a projekt teljes ideje alatt, ekkor a feladatok végrehajthatósága az alábbiak szerint módosul:



2.10. ábra: Erőforrás korlátozott feladat példa, ábrázolás korláttal ^[24]

A példából jól látható, hogy az erőforrás-korlát alkalmazása után a projekt teljes átfutási ideje növekedett két egységgel, $C_{max} = 10$ lett.

Az erőforrás korlátos projektütemezési feladatok során a cél az, hogy az erőforráskorlátok (akár több erőforrás esetén is) figyelembevételével a feladatokat igényeinknek megfelelően ütemezzük. Az ütemezés során a célfüggvény irányulhat idő alapú optimalizálásra, erőforrás kihasználtságra és egyéb változókra.

A TDK munkám alapötlete abból a felismerésből született, hogy a projektütemezési feladatok egy bizonyos szint után olyan mértékben bonyolulttá válnak, hogy kézi megoldásuk és értékelésük rengeteg időt venne igénybe.

Ezért a szoftverfejlesztési munkám legfontosabb célkitűzése az volt, hogy egy saját fejlesztésű számítógépes alkalmazás segítségével a projektütemezési feladatok egyszerűen reprezentálhatók és hatékonyan megoldhatók legyenek, valamint egy olyan kiterjesztett modellt hozzak létre, amely támogatja a projektütemezési feladatok általánosítását.

III. FELHASZNÁLT SZOFTVER- TECHNOLÓGIÁK

Kétség kívül korunk egyik jelentős vívmánya a számítógép, azonban hosszú utat kellett megtenni ahhoz, hogy a ma ismert és használt technológiák - gyakorlatilag mindenki számára – hozzáférhetőek és felhasználhatóak legyenek.

Kezdetben nem voltak szuperszámítógépek, szerverparkok, mainframek és egyéb csúcsteljesítményű eszközök. Az idők során az egyre nagyobb léptékű fejlődéseknek köszönhetően napjainkban már olyan eszközök állnak a rendelkezésünkre, amelyek jelentős mértékben megkönnyíthetik a munkánkat.

A nem is olyan távoli múltban, ha egy programozó szeretett volna egy új eljárást kipróbálni, tesztelni egy új algoritmust, nem volt lehetősége arra, hogy tegyék fel, a notebookján megírja a programot, futtassa, és pillanatok alatt lássa az eredményeket, majd kiértékelje a látottakat. Gyakran a fejlesztők napokig vártak arra, hogy eredményt kapjanak és hozzá is férhessenek, majd hetekig tartó elemzések után volt lehetőségük újra tesztelni a programot.

Kezdetben a programozási nyelvek is jóval fejletlenebbek voltak, kezdve a lyukkártyás megoldásoktól, az alacsony szintű programozási nyelveken át a mai magas szintű programozási nyelvekig hosszú idő telt el. Manapság rengeteg jól használható és kiforrott eszköz áll a fejlesztők rendelkezésére a különféle problémák megoldásához.

Ebben a fejezetben ismertetem a fejlesztés során használt technológiákat, rövid áttekintést adok róluk és bemutatom azokat az előnyös tulajdonságaikat, amik miatt a feladatom elkészítéséhez ezeket használtam.

3.1 A JAVA programozási nyelv

3.1.1 Áttekintés, történelmi háttér

A Java általános célú, objektumorientált programozási nyelv, amelyet a Sun Microsystems fejlesztett a 90-es évek elejétől kezdve egészen addig, amíg a céget fel nem vásárolta az Oracle 2009-ben. 2011-ben a Java 1.7-es verzióját az Oracle gondozásában adták ki.

A Java alkalmazásokat jellemzően bájtkód formátumra alakítják, de közvetlenül natív (gépi) kód is készíthető Java forráskódból. A bájtkód futtatása a Java virtuális géppel történik, ami vagy interpretálja a bájtkódot vagy natív gépi kódot készít belőle, és azt futtatja az adott operációs rendszeren. Létezik közvetlenül Java bájtkódot futtató hardver is, az úgynevezett Java processzor.

A Java nyelv szintaxisát főleg a C és a C++ nyelvektől örökölte, viszont a Java sokkal egyszerűbb objektummodellel rendelkezik, mint a C++. A JavaScript szintaxisa és neve hasonló ugyan a Javához, de nincs közvetlen köze egymáshoz a két nyelvnek.

Bár a nyelv neve kezdetben Oak (tölgyfa) volt, (James Gosling, a nyelv atyja nevezte így az irodája előtt növe tölgyfáról), de később kiderült, hogy ilyen elnevezésű nyelv már létezik, ezért végül Java néven vált ismertté.

A Java szó a Sun Microsystems védjegye. Ennélfogva engedélye nélkül más nem használhatja más által kifejlesztett termékek megjelölésére még például Java-szerű stb. összetételekben sem, mert ez a védjegyjogosult jogaiba ütközik.

A Java nyelv története összeforrt a Sun Microsystems Java fordítójával és virtuális gépével és az ezekhez kapcsolódó fejlesztői programcsomaggal (Java SDK vagy újabban JDK – Java Development Kit), amely a Java nyelv és ahhoz kapcsolódó szabványok standard implementációjának tekinthető.

2006-ban megjelent egy változata a Java-nak JavaFX néven, ami egy szkriptnyelvet takar. Gyorsan, egyszerűen tudunk vele asztali alkalmazásokat készíteni. A JavaFX drasztikusan lerövidült termelési ciklust kínál a Java-fejlesztők és a webfejlesztők számára egyaránt, valamint megkönnyíti a grafikát, videót, audiót, animációt és gazdag szövegfunkciókat tartalmazó alkalmazások létrehozását. ^[25]

3.1.2 Tulajdonságai

Négy fontos szempontot tartottak szem előtt, amikor a Javát kifejlesztették:

- objektumorientáltság;
- függetlenség az operációs rendszertől, amelyen fut (többé-kevésbé);
- olyan kódokat és könyvtárakat tartalmazzon, amelyek elősegítik a hálózati programozást;
- távoli gépeken is képes legyen biztonságosan futni.

Objektumorientáltság

A nyelv első tulajdonsága, az objektumorientáltság („OO”), a programozási stílusra és a nyelv struktúrájára utal. Az OO fontos szempontja, hogy a szoftvert objektumok alapján csoportosítja, nem az elvégzett feladatok szerint. Ennek alapja, hogy az előbbi sokkal kevesebbet változik, mint az utóbbi, így az objektumok (az adatokat tartalmazó entitások) jobb alapot biztosítanak egy szoftverrendszer megtervezéséhez. A cél az volt, hogy nagy fejlesztési projekteket könnyebben lehessen kezelni, így csökken az elhibázott projektek száma.

Platformfüggetlenség

A második tulajdonság, a platformfüggetlenség azt jelenti, hogy a Javában íródott programok hasonlóan fognak futni különböző hardvereken. Ezt úgy lehet megvalósítani, hogy a Java fordítóprogram csak egy úgynevezett Java bájtkódra fordítja le a forráskódot, ami aztán futtatva lesz a virtuális gépben, amely lefordítja az illető hardver gépi kódjára. Továbbá, olyan szabványos könyvtárcsomagok léteznek, amelyek elérhetővé teszik az illető hardver sajátosságait (grafika, szálak és hálózat) egységes módon.

Hálózati programozási, távoli futás

Vannak olyan Java fordítóprogramok, amelyek natív gépi kódra fordítják le a forráskódot, ilyen például a GCJ, így valamelyest felgyorsítják a futtatást, de ugyanakkor a lefordított program elveszti hordozhatóságát. A Java rendszer volt az első, amely lehetővé tette a távoli gépeken való futtatást sandboxban (homokozóban).

Egy kisalkalmazás futtatható a felhasználó gépén (letöltve a Java kódot egy HTTP kiszolgálóról). A kód egy biztonságos környezetben fut, amely nem teszi lehetővé a "rossz szándékú" kód futtatását; a gyártók kiadhatnak olyan tanúsítványokat, amelyeket digitálisan aláírnak, tanúsítva, hogy a kisalkalmazás biztonságos, lehetővé téve ennek kilépését a biztonságos környezetből (ugyancsak a felhasználó felügyelete alatt). ^[26]

3.2 A fejlesztési és tesztelési környezet

3.2.1 PC

Jelenleg nincs lehetőségem egy napjainkban szuperszámítógépként értelmezett konfigurációval dolgozni, de szerencsére a mai fejlett hardveres és szoftveres technológiáknak köszönhetően már egy egyszerű asztali PC is rendelkezik olyan mértékű számítási teljesítménnyel, amely megfelel a projekt megvalósítására és elég gyors ahhoz, hogy a futási eredmények rövid időn belül kiértékelhetőek legyenek. Ezért az alkalmazás elkészítéséhez és a futási tesztekhez az alábbi konfigurációval ellátott PC-t használtam:

Típus	Techikai adatok
Processzor	Mobile DualCore Intel Core i3-3110M, 2400 MHz (24 x 100)
Memória	4 GB DDR3-1600 DDR3 SDRAM
HDD	WDC 320 GB, 5400 RPM, SATA-II
VGA	NVIDIA GeForce GT 720M - 1GB
Operációs rendszer	Microsoft Windows 8.1 Professional

3.1. táblázat: Fejlesztési – és tesztkörnyezet

A fent ismertetett PC kellő mértékben kiszolgálta a fejlesztés és tesztelés során felmerült igényeket, a fejlesztés során nem fordult elő hardveres vagy szoftveres meghibásodás, ezért a fejlesztés és tesztelés ideje alatt végig a fenti konfigurációt használtam. Más konfiguráción is teszteltem az alkalmazás működését, viszont a sebességbeli eltérés elhanyagolható volt, ezért az abból született eredmények alapján nem tekintettem fontosnak a több konfiguráción való részletes tesztelést.

3.2.2 Eclipse integrált fejlesztői környezet

A JAVA fejlesztést az Eclipse keretrendszerrel végeztem, mely a könnyen átlátható felülete mellett, gyors és agilis szoftverfejlesztést biztosít, alkalmas a debuggolásra, kiválóan skálázható különféle pluginokkal, ezért megfelelő eszköz az ilyen jellegű fejlesztésekhez.

Az Eclipse nyílt forráskódú, platformfüggetlen szoftver keretrendszer, amellyel úgynevezett vastag kliens (rich client) alkalmazásokat lehet készíteni. Ezt a keretrendszert eddig jellemzően integrált fejlesztőkörnyezetek (Integrated Development Environment, IDE) készítésére használták fel, mint például a Java IDE, a Java Development Toolkit (JDT) és fordító (ECJ) amelyet az Eclipse részeként terjesztenek (illetőleg az Eclipse fejlesztéséhez is használnak). Az Eclipse természetesen más kliensalkalmazások fejlesztéséhez is használható, mint például az Vuze nevű BitTorrent kliens.

Az Eclipse egyben egy felhasználói csoportot is takar, amely folyamatosan bővíti az Eclipse alkalmazási területeit. Ennek egyik példája az Eclipse modellező projekt, amely a modellvezérelt tervezés legtöbb területét lefedi.

Az Eclipse-et eredetileg az IBM fejlesztette ki a VisualAge eszközcsalád utódjaként. Az Eclipse projektet jelenleg az „Eclipse Foundation” (magyarul Eclipse Alapítvány) nevű szoftveripari nonprofit konzorcium kezeli. Több szoftvereszközzgyártó is az Eclipse-re kívánja alapozni a jövőben IDE termékeit. A szervezet alapító tagja a QNX Software Systems, amely 52%-os tulajdonrészt mondhat magáénak. ^[27]

3.2.3 Maven

A Maven definíció szerint "software project management and comprehension tool". Tehát nem egy egyszerű build eszköz, hanem annál sokkal fejlettebb, magasabb szintű megoldás. Alapja a project object model (POM), melyet XML-ben leírva tudunk megadni (pom.xml) és ez alapján a Maven képes további leírások nélkül elvégezni a build folyamatot, különböző riportok, dokumentációk, gyakorlatilag a teljes projekt website-jának legenerálását.

A Maven nem titkolt célja, hogy a Java fejlesztők körében egyfajta szabvány legyen, mely a konvenciók megtartását előnybe helyezi az egyéni konfigurációkkal szemben (convention over configuration).

Sajnos a Java nagyon sokáig nem adott szabványt arra, hogy hogyan kell egy projektet felépíteni, a Maven ezt az űrt próbálja betölteni. Azaz, ha tartjuk magunkat egy projekt struktúrához, a különböző állományainkat a megfelelő könyvtárakba helyezzük el, a Maven képes lesz elvégezni a build folyamatot, a források lefordítását, a különböző erőforrás állományok megfelelő helyre másolását, és az alkalmazásunk összezsomagolását.^[28]

3.2.4 JAX-B

A JAX-B (Java Architecture for XML Binding - Java Architektúra XML kötéshez) lehetővé teszi a Java fejlesztők számára, hogy feltérképezzék a Java osztályt XML reprezentáción keresztül. A JAXB két fő jellemzője: képes a Java objektumokat XML állományokba menteni és onnan visszatölteni. Más szavakkal, a JAX-B képes a memóriából adatokat XML formátumúra alakítani anélkül, hogy meg kellene valósítani a programban az XML mentés és beolvasás műveleteket.

A JAX-B különösen hasznos, ha a specifikáció komplex és gyakran változik. Ebben az esetben rendszeresen változik az XML séma definíciója azért, hogy a Java definíciókkal szinkronban tartsuk, ami sok időt vesz igénybe és növeli a hibák lehetőségét.

A JAX-B a Java SE platform része, és az egyik API a Java EE platformban. A JAX-B 1.0 kiadása a JSR 31 jelű Java Community Process fejlesztés eredménye. 2006 óta, a JAX-B 2.0 verziója a JSR 222 jelzésű fejlesztés keretein belül készül.^[29]

3.2.5 Git / Bitbucket

A projekt során célszerű volt verziókövető rendszert használni ezzel is elkerülve az esetleges adatvesztést és annak érdekében, hogy a projekt könnyebben nyomonkövethető legyen. Az egyszerű használat és az ingyenesség miatt a választásom a Git verziókezelőre esett, melyet Bitbucket repositorykon keresztül érek el. A Git lényege, hogy minden munkamásolat egy teljes értékű repository, teljes verziótörténettel, amely nem függ hálózati eléréstől, vagy egy központi szervertől. Az ehhez szükséges felületet és elérést pedig a Bitbucket szolgáltatja, ahol bizonyos korlátozások mellett nagyszámú projekt tárolható és követhető nyomon.

IV. IMPLEMENTÁCIÓ

4.1 Áttekintés

Az alkalmazás célja, hogy segítségével a projektütemezési feladatok modellezhetők és megoldhatók legyenek, ezért a fejlesztés során ezt tekintem elsődleges feladatommak.

Az elkészült program jelen állapotában képes:

- a feladathoz szükséges adatokat beolvasni XML fájlból,
- erőforrás korlát nélküli projektütemezési feladatokat reprezentálni,
- kezelni a több kezdő – és végpontú gráfokat,
- erőforrás korlát nélküli feladatok esetén meghatározni a CPM módszerhez szükséges adatokat,
- erőforrás korlát nélküli projektütemezési feladatokat megoldani a CPM módszer használatával,
- erőforráskorlátot kezelni egy adott erőforrás esetén,
- erőforrás igényeket rendelni az aktivitásokhoz,
- figyelembe venni erőforrás korlát esetén a precedencia gráf függőségeit,
- erőforrás korlátos projektütemezési feladatok reprezentálására,
- erőforrás korlátos projektütemezési feladatok megoldására,
- több rendező algoritmus alapján ütemtervet készíteni,
- összgezést adni a futási eredményekről, fájlba írni azokat.

A program segítségével az általam vizsgálni kívánt feladatok megoldhatók és a futási eredmények alapján az egyes algoritmusok hatékonysága összehasonlítható.

A modell kialakítása során törekszem arra, hogy a jövőbeni fejlesztések minél egyszerűbben, hatékonyabban kivitelezhetőek legyenek, ezért a megfelelő modell és alkalmazás logika kialakítása kiemelten fontos szerepet játszik az implementációt megelőzően.

4.1.1 Tesztvezérelt fejlesztés (TDD)

A tesztvezérelt fejlesztés egy szoftverfejlesztési módszertan. Az agilis szemlélet térhódításának köszönhetően ez a technika manapság egyre népszerűbb.

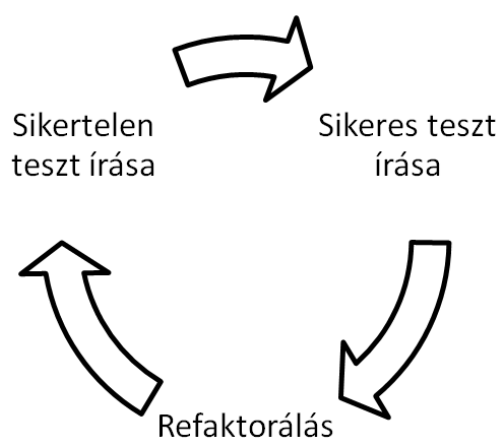
A tesztvezérelt fejlesztés (Test Driven Development – TDD) az agilis tanok egyike, amely azt mondja, hogy igenis a fejlesztőnek kell megírni saját programjának tesztjeit. A TDD még ennél is tovább merészkedik, amikor azt állítja, a tesztekkel kell először megírni, majd utána a programkódot. Ezzel felborul a világ rendje, hiszen a jól megszokott “utólag majd tesztelünk valamit” szemlélet nem működhet tovább.

Ha belegondolunk, így dolgozva a termék létrejötte után nem kell újra tesztekkel írni, mert nem elég, hogy megvan a teszt kód, a programunk már meg is felel annak.

A tesztvezérelt fejlesztés egy olyan – többek között – minőséget biztosító szoftverfejlesztési módszertan, amit, ha valaki kipróbál és ráérez az ízére, többé nem szívesen használja a “hagyományos” módszereket (kivétel, ha a munkaadója parancsolja).

A lényege, hogy mindig kiválasztunk egy kicsi, megírandó programrészt, elkészítjük a keretét és egy tesztet, lefuttatjuk a tesztet, hogy hibás eredményt adjon. Ezt követően addig fejlesztjük a kódot, amíg át nem megy a teszten. Ha kell, a tesztet is módosítjuk, és újra teszteljük rajta a programrészt (4.1 ábra).^[30]

A programozás során a legalapvetőbb funkciókat igyekszem a tesztvezérelt fejlesztés elveit követve implementálni, elkerülve az esetlegesen felmerülő hibákat, valamint tervezési pontatlanságokat.

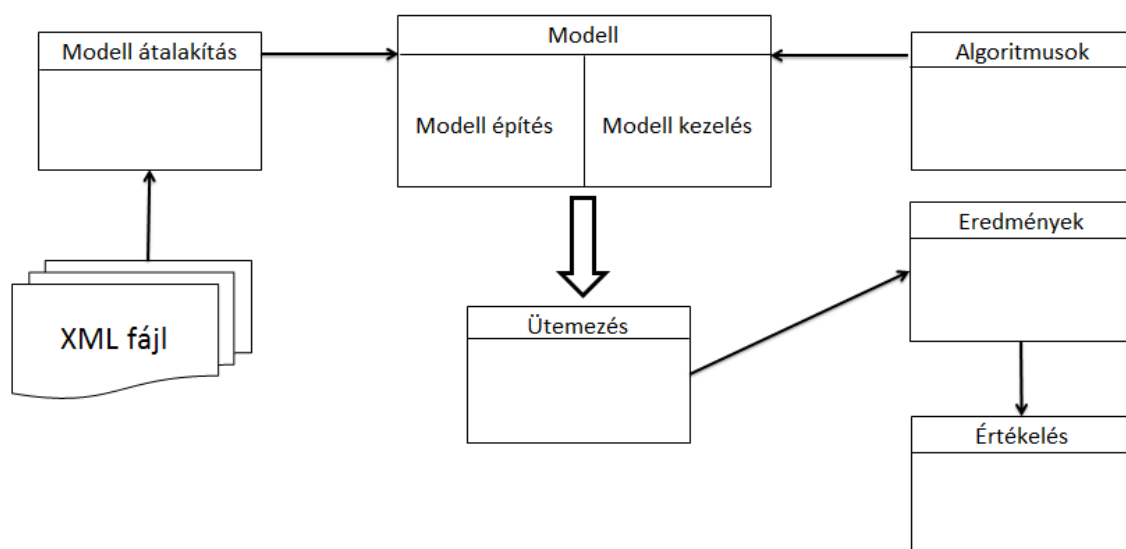


4.1. ábra: A tesztvezérelt fejlesztés logikája

4.2 Alkalmazás logika

A projekt kezdetekor egyértelmű volt, hogy a feladat komplexitása miatt célszerű egy olyan modellt megalkotni, amely könnyen átlátható és az egyes feladatok jól megkülönböztethetők. Ennek érdekében az implementáció előtti modellalkotást megelőzően olyan alkalmazási logikát próbálok meg felállítani, amely összeegyeztethető a feladattal és átfogja a program alapvető működési elveit.

Az alkalmazás logikáját az alábbiak szerint alakítottam ki:



4.2. ábra: Alkalmazás logika

Az alkalmazás bemeneteit egy XML fájl tartalmazza, azonban ahhoz, hogy ez a fájl - programozási szempontból - kezelhető legyen számunkra, át kell alakítani olyan formátumúvá, amellyel műveleteket lehet végezni.

Ezt végzi el a modell átalakításért felelős komponens, majd a kész modellt a bemeneti adatok alapján építi fel az alkalmazás.

A meglévő modellt a feladat típusától függően kell kezelni, ezért ezt a feladatot külön egységek látják el. Az algoritmusokat felhasználva megtörténik a projekt ütemezése, ezután létrejönnek a futási eredmények.

A végső fázisban történik az eredmények értékelése, valamint megjelenítése a felhasználó felé.

4.3 Ütemező algoritmusok

4.3.1 Determinisztikus eljárások

Determinisztikusnak nevezzük azokat a folyamatokat, amelyek időfejlődése determinisztikus, azaz ismerve a rendszer kezdő állapotát a kimenet egyértelműen meghatározható lesz. Az ilyen algoritmusok esetén az ütemtervet valamilyen rendezési elv adja. Az erőforrás-korlátos ütemezés esetén négy ilyen rendező algoritmus működését teszteltem, majd értékelem az eredményeit:

- az aktivitások átfutási ideje alapján csökkenő sorrendben ütemezés,
- az aktivitások átfutási ideje alapján növekvő sorrendben ütemezés,
- az aktivitások erőforrás igénye alapján csökkenő sorrendben ütemezés,
- az aktivitások erőforrás igénye alapján növekvő sorrendben ütemezés.

Ezeknek az algoritmusoknak nagy előnyük a futási sebességük, ugyanis nem igényelnek jelentős erőforrást, valamint számítási kapacitást, ezért gyorsan képesek ütemtervet generálni. Hátrányuk viszont, hogy erősen szituációfüggőek, tehát a feltételeken vagy bemenő adatokon alkalmazott legkisebb változtatás esetén is nagymértékű eltérés lesz tapasztalható az ütemtervben.

4.3.2 Sztochasztikus keresés

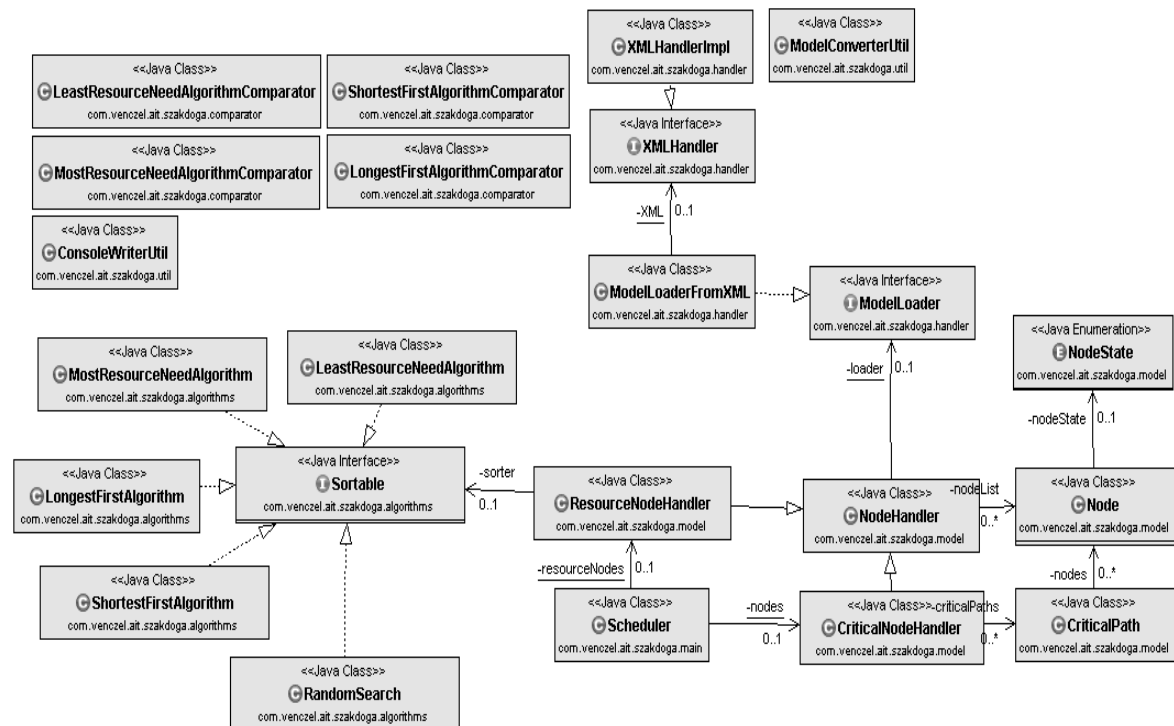
Sztochasztikus eljárások alatt az olyan algoritmusokat értjük, melyek (ál)véletlen számok segítségével generálnak új próbapontokat. Lényege, hogy egyenletes eloszlásban generáljuk a pontokat a tartományon, és a legkisebb értéket adót eltároljuk, mint a globális minimum pont egy közelítését. A projekt keretein belüli gráfok esetén az optimumhoz erősen közelítő értéket ad kellően nagy iterációs szám esetén.

Egy korai megfigyelés szerint annak valószínűsége, hogy az $N+1$. húzással jobb pontot találunk, ha már N húzás volt, $\frac{1}{N+1}$, és K újabb pontra ez a valószínűség $\frac{K}{N+K}$ -ra nő.

A sztochasztikus folyamatokra igaz, hogy ha $N \rightarrow \infty$, akkor a végén a globális optimumot megtaláljuk. Ez persze csak elméletileg nyugtathat meg minket, valójában célunk elérésének valószínűsége nagyban függ a sikerterület méretétől. ^[31]

4.4 Osztálydiagram

Az alkalmazás osztályhierarchiájának áttekintéséhez az UML modellező nyelvet választottam, ugyanis a segítségével jól reprezentálhatóak a projektet felépítő elemek és a közöttük lévő kapcsolatok. Az UML diagram után, az osztályok felépítését és funkcióit mutatom be, annak érdekében, hogy a program működése minél átláthatóbb és világosabb legyen.



4.3. ábra: UML diagram

4.5 Osztályok, interfészek

4.5.1 Modellépítés, adatbeolvasás

Node osztály

A Node osztály jelenti a program legalapvetőbb egységét, ugyanis ez az osztály egy csúcsot reprezentál a precedencia gráfban. Ezen az objektumon végzi a program az elemi tevékenységeket, ezért közvetlenül vagy közvetetten minden osztállyal kapcsolatban van. Inicializálható egy azonosító és egy végrehajtási idő adattaggal, vagy egy azonosító, végrehajtási idő és erőforrás igény adattaggal.

Ezen kívül tartalmazza a legkorábbi és legkésőbbi indítási időket, valamint a legkorábbi és legkésőbbi befejezési időket.

Az erőforráskorlát kezelése érdekében tartalmaz egy *allocatedResources* adattagot is, amely egy adott időpillanatban aktuálisan a *Node* objektum által lefoglalt erőforrásokat jelenti, valamint egy *nodeState* állapotjelző adattagot, amely a *Node* objektum futásidejű állapotát jelzi, melynek az erőforrás korlátos ütemezés során lesz fontos szerepe. Az adattagokon kívül adattag lekérő és beállító (getter, setter) metódusok szerepelnek, valamint objektum összehasonlító és adat kiírató metódusok.

NodeState enum

A *NodeState* enumerációs típus szolgál a *Node* objektumok állapotainak tárolására, melyben 4 alapvető objektum állapotot különböztetnek meg:

- megelőző aktivitás befejeződésére várakozó objektum,
- erőforrásra várakozó objektum,
- futó állapotú objektum,
- befejeződött állapotú objektum.

XMLHandler interfész

Az interfészben összesen két metódus található, az egyik az adatok írására szolgál, a másik az adatok olvasására, az interfész metódusait az *XMLHandlerImpl* osztály fogja implementálni.

XMLHandlerImpl osztály

Implementálja az *XMLHandler* osztályt, tehát a metódusait is. A beolvasásra szolgáló metódusban a *JAXB* használatával megvalósítja az írási műveletet, míg a beolvasó metódusban az olvasási műveletet.

ModelLoader interfész

Az interfész *Node* objektumok visszaadására szolgál.

ModelLoaderFromXML osztály

Ez az osztály felelős a hivatkozott XML fájl betöltésért. Létrehoz egy osztályszintű final *XMLHandlerImpl* objektumot, majd beolvassa a kívánt XML fájlt.

ModelConverterUtil osztály

Ez az osztály végzi el az XML és a modell közti leképzést. Beállítja az XML-ben lévő adattagokat a modellben az objektumok megfelelő értékeihez. Típusellenőrzést is végez, csak egész szám adható meg bemenetként.

4.5.2 Node objektumok kezelése***NodeHandler osztály***

Az osztály betölti egy listába az összes megkapott *Node* objektumot egy listaként. A listával alapvető műveleteket lehet végezni, valamint ez az osztály számolja ki automatikusan a *Node* objektumok megelőző aktivitásait, ugyanis csak a rákövetkezőket szükséges megadni az XML fájlban. Az osztályban ezek mellett a legkorábbi kezdési és befejezési idő számítása is megtörténik.

CriticalPath osztály

Az osztály egy *Node* objektumokból álló listát tartalmaz, lehetőség van a lista lekérésére és beállítására.

CriticalNodeHandler osztály

Az osztály a *NodeHandler* osztály leszármazottja, ebben történnek a *CPM* módszer megvalósításához szükséges folyamatok. Az osztályban inicializálódnak a legkésőbbi kezdési és befejezési időpontok, valamint a tartalékidők is kiszámításra kerülnek. Ezeken felül az osztály képes meghatározni a kritikus utat, valamint több kritikus út esetén a kritikus utakat.

ResourceNodeHandler osztály

Erőforrás korlátozott ütemezési feladatok megvalósításához szükséges osztály. A *NodeHandler* osztály leszármazottja. Ebben az osztályban történik meg az erőforrás korlát beállítása, valamint a *Node* objektumok ütemezése figyelembe véve az erőforrás korlátokat. A *Node*-ok ütemezésénél első lépésben megtörténik az állapotok beállítása, majd a gráfon végighaladva beállítja a kezdési időket és idő közben a szükséges állapotváltozásokat.

4.5.3 Ütemező algoritmust megvalósító osztályok

Sortable interfész

Az interfész egy *Node*-okat tartalmazó listát ad vissza, a rendező algoritmusok osztályai implementálják.

LongestFirstAlgorithmComparator osztály

Összehasonlít két *Node* objektumot a végrehajtási idejük alapján, majd visszaadja közülük a nagyobbik végrehajtási idejét.

LongestFirstAlgorithm osztály

Az osztály implementálja a *Sortable* interfészt és a *Node* objektumokból álló listát rendezi a *LongestFirstAlgorithmComparator* osztály alapján.

ShortestFirstAlgorithmComparator osztály

Összehasonlít két *Node* objektumot a végrehajtási idejük alapján, majd visszaadja közülük a kisebbik végrehajtási idejét.

ShortestFirstAlgorithm osztály

Az osztály implementálja a *Sortable* interfészt és a *Node* objektumokból álló listát rendezi a *ShortestFirstAlgorithmComparator* osztály alapján.

MostResourceNeedAlgorithmComparator osztály

Összehasonlít két *Node* objektumot az erőforrás igényük alapján, majd visszaadja közülük a nagyobbik erőforrás igényűt.

MostResourceNeedAlgorithm osztály

Az osztály implementálja a *Sortable* interfészt és a *Node* objektumokból álló listát rendezi a *MostResourceNeedAlgorithmComparator* osztály alapján.

LeastResourceNeedAlgorithmComparator osztály

Összehasonlít két *Node* objektumot az erőforrás igényük alapján, majd visszaadja közülük a kisebbik erőforrás igényűt.

LeastResourceNeedAlgorithm osztály

Az osztály implementálja a *Sortable* interfészt és a *Node* objektumokból álló listát rendezi a *LeastResourceNeedAlgorithmComparator* osztály alapján.

RandomSearchAlgorithm osztály

Az osztály implementálja a *Sortable* interfészt és a *Node* objektumokból álló listát rendezi véletlenszerű sorrend alapján.

4.5.4 Futtatás, eredmények értékelése***ConsoleWriterUtil osztály***

Az osztály látja el az adatmegjelenítési funkciókat, elvégzi az ütemezés során megkapott adatok kiíratását.

Scheduler osztály

Ez a projekt *Main* osztálya, itt történik meg az előzőekben ismertetett osztályok és interfészek példányosítása, a metódusok meghívása, majd az adatok megjelenítése.

V. MŰKÖDÉS ÉS KIÉRTÉKELÉS

Áttekintés

Az alkalmazás célja, hogy a segítségével a projektütemezési feladatok modellezhetők és megoldhatók legyenek, ezért szükséges a funkcionális tesztelés a konkrét futási eredmények mellett. Az erőforrás-korlát nélküli feladatok esetében a kritikus útvonal módszer (*CPM*) megvalósulását vizsgálom. A tesztek során igyekszem olyan teszteseteket generálni és bemutatni, amelyekkel a program működésének helyessége igazolható, a funkciók működése demonstrálható.

Elsőként az erőforrás-korlát nélküli feladatokra mutatok be példát, főként funkció tesztelés céljából, valamint a program futási idejének és a feladat egyes tényezőinek összefüggése között keresek kapcsolatot.

A következő részben az erőforrás-korlátozott feladatok esetében tesztelem az implementált funkciókat, majd a tesztesetek kiértékelése után megvizsgálom az implementált algoritmusok eredményeit és összehasonlítom a kapott adatokat.

5.1 Erőforrás-korlát nélküli projektütemezési feladatok

Ilyen jellegű feladatok végrehajtásánál a program minden esetben optimális ütemtervet ad, ugyanis a *CPM* módszer által meghatározott C_{max} értéke a maximális átfutási idő minimumát jelenti, tehát a kritikus út, vagy kritikus utak utolsó aktivitásának a befejezési idejét. Ezeknél a feladatoknál a tesztelési szempont elsősorban a funkcionalitás megfelelőségére irányult, ezért az alábbi példában egy viszonylag kevés elemszámú gráfban prezentálom a működést. A továbbiakban a futási eredményeket ismertetem és a futási időt vizsgálom a feladatok egyes változóinak függvényében.

5.1.1 Funkcionális teszt

Teszt 5.1.1 (Kritikus útvonal módszer)

Job (ID)	p(j)	Successors
0	0	1,4
1	4	2,3
2	7	6
3	7	5
4	1	5
5	2	100
6	2	100
100	0	0

5.1. táblázat: Teszt 5.1.1, input adatok

Az egyszerűbb nyilvántartás miatt a feladatok megoldása során bevezetnek két elemet, a „0” és „100” azonosítókkal ellátva, ez két virtuális csomópontot jelent, a tényleges kezdő és végpontokat reprezentálva, így ugyanis lehetséges a több kezdő – és végpontú gráfok tesztelése. A fenti adatok az XML fájl kezelőfelületének segítségével (természetesen az XML fájl közvetlen módosításával is) megadhatók az alábbiak szerint:

▲ [e] node	(successors?, numberOfResources)
[a] id	1
[a] time	4
[e] successors	2, 3
[e] numberOfResources	0
▲ [e] node	(successors?, numberOfResources)
[a] id	2
[a] time	7
[e] successors	6
[e] numberOfResources	0
▲ [e] node	(successors?, numberOfResources)
[a] id	3
[a] time	7
[e] successors	5
[e] numberOfResources	0

5.1. ábra: Teszt 5.1.1, példa az input adatok megadására XML kezelőfelülettel

Az input adatok megadása után a program futtatható, majd a futtatás közben elvégzi a számításokat:

- automatikusan meghatározza az aktivitások predecessorait, amelyek a fenti feladat alapján a következők (aktuális Node objektum, majd alatta a megelőző aktivitás, és a megelőző aktivitás adatai):

```
23 Predecessors list:
24
25 Node [id: 0, time: 0]
26 Node [id: 1, time: 4]
27   previous node: Node [id: 0, time: 0]
28 Node [id: 2, time: 7]
29   previous node: Node [id: 1, time: 4]
30 Node [id: 3, time: 7]
31   previous node: Node [id: 1, time: 4]
32 Node [id: 4, time: 1]
33   previous node: Node [id: 0, time: 0]
34 Node [id: 5, time: 2]
35   previous node: Node [id: 3, time: 7]
36   previous node: Node [id: 4, time: 1]
37 Node [id: 6, time: 2]
38   previous node: Node [id: 2, time: 7]
39 Node [id: 100, time: 0]
40   previous node: Node [id: 5, time: 2]
41   previous node: Node [id: 6, time: 2]
--
```

5.2. ábra: Teszt 5.1.1, predecessorok meghatározása

- meghatározza az aktivitások legkorábbi és legkésőbbi kezdési, valamint legkorábbi és legkésőbbi befejezési időpontjait az előrehaladó és hátrafelé haladó eljárás alapján (aktivitás ID, végrehajtási idő, legkorábbi kezdés, legkésőbbi kezdés, legkorábbi befejezés, legkésőbbi befejezés, erőforrás igény):

```
43 Calculated values: (Earliest Start, Latest Start, Earliest Finish, Latest Finish)
44
45 id: 0, time: 0, ES: 0, LS: 0, EF: 0, LF: 0, resource: 0
46 id: 1, time: 4, ES: 0, LS: 0, EF: 4, LF: 4, resource: 0
47 id: 2, time: 7, ES: 4, LS: 4, EF: 11, LF: 11, resource: 0
48 id: 3, time: 7, ES: 4, LS: 4, EF: 11, LF: 11, resource: 0
49 id: 4, time: 1, ES: 0, LS: 10, EF: 1, LF: 11, resource: 0
50 id: 5, time: 2, ES: 11, LS: 11, EF: 13, LF: 13, resource: 0
51 id: 6, time: 2, ES: 11, LS: 11, EF: 13, LF: 13, resource: 0
52 id: 100, time: 0, ES: 13, LS: 13, EF: 13, LF: 13, resource: 0
53 -----
--
```

5.3. ábra: Teszt 5.1.1, CPM módszerhez szükséges adatok meghatározása

- megjeleníti a tartalékidő nélküli aktivitásokat:

```

57 Nodes with 0 spare time:
58
59 Node [id: 0, time: 0]
60 Node [id: 1, time: 4]
61 Node [id: 2, time: 7]
62 Node [id: 3, time: 7]
63 Node [id: 5, time: 2]
64 Node [id: 6, time: 2]
65 Node [id: 100, time: 0]

```

5.4. ábra: Teszt 5.1.1, kritikus aktivitások

- végül kiszámolja a kritikus utakat, az átfutási időt, valamint a programfutási időt

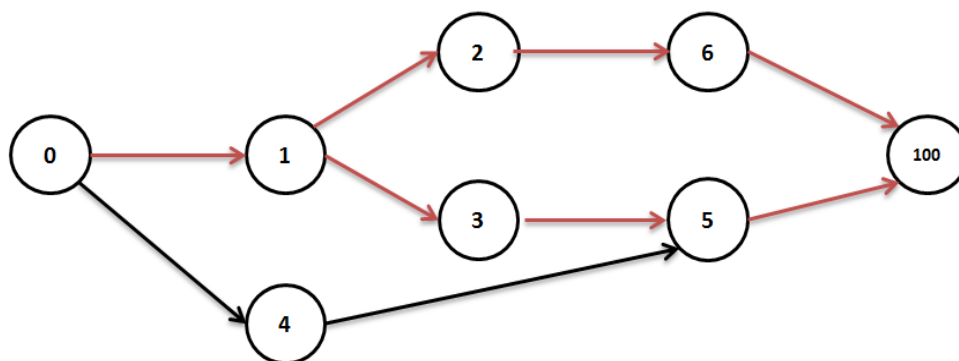
```

67 Critical Paths:
68
69 Node [id: 0, time: 0]
70 Node [id: 1, time: 4]
71 Node [id: 3, time: 7]
72 Node [id: 5, time: 2]
73 Node [id: 100, time: 0]
74
75 Node [id: 0, time: 0]
76 Node [id: 1, time: 4]
77 Node [id: 2, time: 7]
78 Node [id: 6, time: 2]
79 Node [id: 100, time: 0]
80
81 Cmax:13
82 Runtime: 0.0 s

```

5.5. ábra: Teszt 5.1.1, kritikus aktivitások

Az előbbieken ismertetett és megoldott feladatban, tehát az átfutási idő C_{max} eredménye 13 egység lesz, és a gráf két kritikus utat tartalmaz, a gráf reprezentációja az alábbi ábrán látható:



5.6. ábra: Teszt 5.1.1, gráf reprezentáció

A gráf egyszerűsége miatt könnyen ellenőrizhető a program működése, viszont az is látszik a példából, hogy képes kezelni azokat az eseteket, amikor több kezdőpontot és végpontot tartalmaz a gráf, valamint több kritikus út esetén is sikerrel határozza meg a kritikus út elemeit, majd a kritikus utakat.

5.1.2 Futási eredmények

A kritikus útvonal módszer végrehajtása során azt vizsgálom, hogy milyen összefüggés van a gráf és a futási idő között, figyelembe véve a gráf elemszámát valamint a teljes átfutási időt.

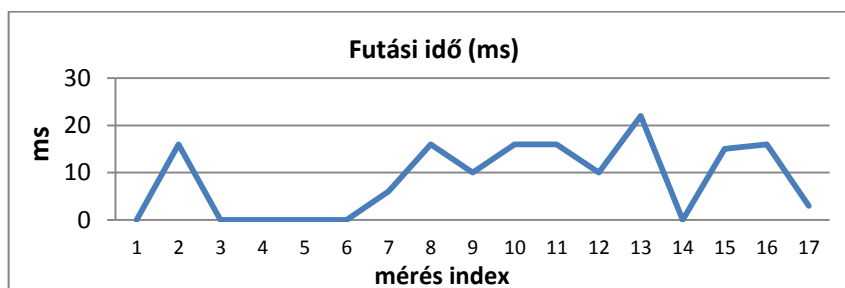
Teszt 5.1.2 (Kritikus útvonal módszer futási eredmények)

Az alábbiakban az elvégzett mérések eredményeinek táblázata látható:

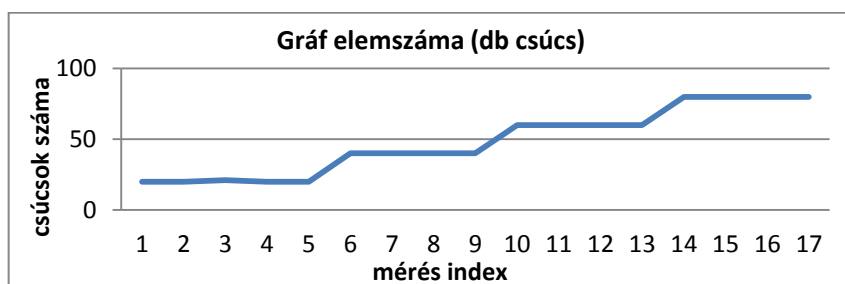
<i>mérés index</i>	<i>gráf elemszáma (db csúcs)</i>	<i>C_{max} (időegység)</i>	<i>futási idő (ms)</i>
1	20	97	0,016
2	20	122	16
3	21	97	0,016
4	20	83	0,016
5	20	97	0,016
6	40	137	0,016
7	40	409	6
8	40	911	16
9	40	845	10
10	60	611	16
11	60	885	16
12	60	817	10
13	60	865	22
14	80	181	0,016
15	80	905	15
16	80	753	16
17	80	989	3

5.2. táblázat: Teszt 5.1.2, mérési eredmények

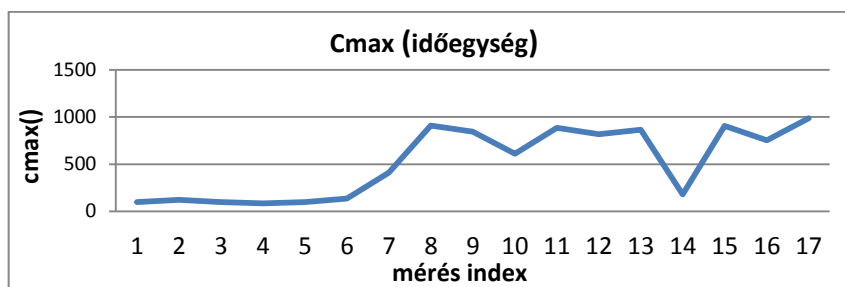
A mért adatok grafikus szemléltetéséhez célszerű grafikonok formájában ábrázolni az eredményeket, a könnyebb áttekinthetőség érdekében. Az alábbiakban látható négy grafikon a fenti táblázat eredményeit tartalmazza, a grafikonok x tengelyein találhatóak rendre a mérés indexek, az y tengelyen pedig az aktuálisan vizsgált érték.



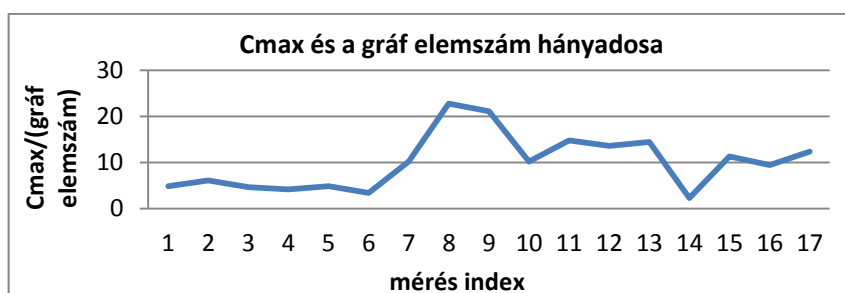
5.1. grafikon: Teszt 5.1.2, futási idők



5.2. grafikon: Teszt 5.1.2, csúcsok száma a gráfban



5.3. grafikon: Teszt 5.1.2, C_{max} átfutási idő



5.4. grafikon: Teszt 5.1.2, C_{max} és a gráf csúcspont számának hányadosa

Az adatok elemzése során néhány esetben az alacsony C_{max} idő ellenére magas futási idő volt megfigyelhető, ami azzal magyarázható, hogy ezeknek a gráfoknak a szerkezete jóval bonyolultabb volt, több megelőzési relációval, éllel és csomóponttal rendelkeznek.

Az így feltárt információk tudatában kijelenthető, hogy a gráfban a CPM módszer végrehajtási idejének, tehát a C_{max} idő meghatározásának futási ideje nem teljes mértékben a gráf nagyságától függ (elemszámától), hanem a gráf éleinek számától és elrendezésétől, a megelőzési relációk számától és elrendezésétől.

5.2 Erőforrás-korlátos projektütemezési feladatok

Az erőforrás-korlátos problémák esetében a program működési elve hasonló az erőforrás-korlát nélküli problémák esetében, viszont itt a megelőzési relációkon kívül figyelembe kell venni az erőforráskorlátot, az aktivitások erőforrás igényeit és a tartalékidőket, majd e tényezők figyelembevételével elkészíteni az ütemtervet.

Az első részben az alapvető funkciók tesztelését ismertetem, a következőkben, pedig a különböző algoritmusok futási eredményeit és azok összehasonlítását.

5.2.1 Funkcionális teszt

Az alkalmazás az erőforrás korlát nélküli feladatok során bemutatott alapon működik, az adatok bekérésénél viszont az XML fájlban már erőforrás igényt is rendelünk az aktivitásokhoz. A program az elindítása után bekéri az erőforrás-korlát szintet, valamint a random kereső algoritmushoz az iterációk számát. Ezután megtörténik a CPM módszer szerinti ütemezés, majd az öt rendező algoritmus szerint elkészülnek az ütemtervek. Az adatok kiírásánál rendre megjeleníti a képernyőn, az algoritmusok futási eredményeit, a generált ütemtervet, valamint az aktuális gráf nagyságát és a random keresés iteráció számát.

```
544 -----
545
546 SCHEDULE SUMMARY:
547
548 Number of nodes in the graph:
549 80
550
551 Scheduled with CPM method without resource constraint:
552 Cmax = 905
553 Runtime: 15.0 ms
554
555 Resource constraint is set to:
556 50
557
558 Scheduled with Shortest Firts Algorithm:
559 Cmax = 1233
560 Runtime: 250.0 ms
561
562 Scheduled with Longest First Algorithm:
563 Cmax = 1245
564 Runtime: 186.0 ms
565
566 Scheduled with Most Resource Need Algorithm:
567 Cmax = 1198
568 Runtime: 46.0 ms
569
570 Scheduled with Least Resource Need Algorithm:
571 Cmax = 1346
572 Runtime: 49.0 ms
573
574 Scheduled with Random Search Algorithm:
575 Cmax = 1153
576 Runtime: 1889.0 ms
577
578 Number of iterations: 50
579
```

5.7. ábra: Erőforrás-korlátos feladat futási eredménye

```
80 0-1: [Node [id: 4, time: 1]]
81 1-2: [Node [id: 1, time: 4]]
82 2-3: [Node [id: 1, time: 4]]
83 3-4: [Node [id: 1, time: 4]]
84 4-5: [Node [id: 1, time: 4]]
85 5-6: [Node [id: 2, time: 3], Node [id: 3, time: 7]]
86 6-7: [Node [id: 2, time: 3], Node [id: 3, time: 7]]
87 7-8: [Node [id: 2, time: 3], Node [id: 3, time: 7]]
88 8-9: [Node [id: 3, time: 7]]
89 9-10: [Node [id: 3, time: 7]]
90 10-11: [Node [id: 3, time: 7]]
91 11-12: [Node [id: 3, time: 7]]
92 12-13: [Node [id: 5, time: 2], Node [id: 6, time: 5]]
93 13-14: [Node [id: 5, time: 2], Node [id: 6, time: 5]]
94 14-15: [Node [id: 6, time: 5]]
95 15-16: [Node [id: 6, time: 5]]
96 16-17: [Node [id: 6, time: 5]]
~
```

5.8. ábra: Erőforrás-korlátos feladat elkészült ütemterve

Az első ábrán egy ütemezési példa összefoglaló adatai látszanak, másodikban pedig egy elkészült ütemterv, ahol az látható, hogy adott időpillanatban mely aktivitások futnak éppen. Ha egy sorban több aktivitás is található, akkor azok az adott időpillanatban egyszerre futnak.

5.2.2 Futási eredmények

A tesztelés során az algoritmusok futási idejét és a kapott C_{max} értéket mérem, sorra véve az implementált algoritmusokat. A négy determinisztikus módszer esetében az volt a cél, hogy az algoritmusok gyors ütemtervet készítsenek és megoldást adjanak a C_{max} értékére.

Gráf méret	Erőforrás-korlát	Ütemező algoritmusok										
		Legrövidebb idejű először		Leghosszabb idejű először		Legnagyobb erőforrás igényű először		Legkisebb erőforrás igényű először		Random Kereső		
		C_{max}	runtime (ms)	C_{max}	runtime (ms)	C_{max}	runtime (ms)	C_{max}	runtime (ms)	C_{max}	runtime (ms)	iteráció szám
20	40	110	32	124	15	104	~0	111	16	104	141	40
20	40	125	16	122	16	122	~0	122	15	122	110	40
21	40	126	16	118	~0	118	16	126	15	117	125	40
20	40	96	32	98	15	100	16	98	~0	98	141	40
20	40	361	47	414	30	379	~0	397	32	353	265	40
40	40	200	62	194	47	174	16	195	16	176	218	40
40	40	483	78	498	16	485	62	525	16	486	359	40
40	40	1172	110	1339	93	1112	32	1243	15	1072	649	40
40	40	1161	94	1128	78	1087	47	1073	31	1003	596	40
60	40	1343	203	1385	100	1249	63	1476	46	1215	2020	40
60	40	2887	266	2814	106	2606	144	2938	125	2648	4320	40
60	40	2802	281	2622	107	2503	165	2905	141	2482	4002	40
60	40	2812	235	2804	109	2510	125	2806	143	2492	3855	40
80	40	295	78	271	63	250	31	294	32	246	510	40
80	40	1491	234	1496	154	1430	51	1628	94	1412	1834	40
80	40	1626	250	1476	94	1282	62	1486	63	1252	1718	40
80	40	1608	219	1613	128	1481	66	1790	79	1468	2143	40

5.3. táblázat: Algoritmusok mérési eredményei

A táblázat első oszlopában fel van tüntetve az éppen vizsgált gráf elemszáma, a második oszlopban a beállított erőforrás-korlát szintje, majd a következő oszlopokban algoritmusonként a C_{max} értéke és az algoritmus futási ideje. A random kereső eljárásnál az iteráció száma is fel van tüntetve.

A zöld színnel jelölt oszlopok az aktuális mérés esetén mért legjobb (legkisebb) C_{max} értékeket jelentik, a kézzel jelöltek pedig a legjobb (legkisebb) algoritmus futási időket.

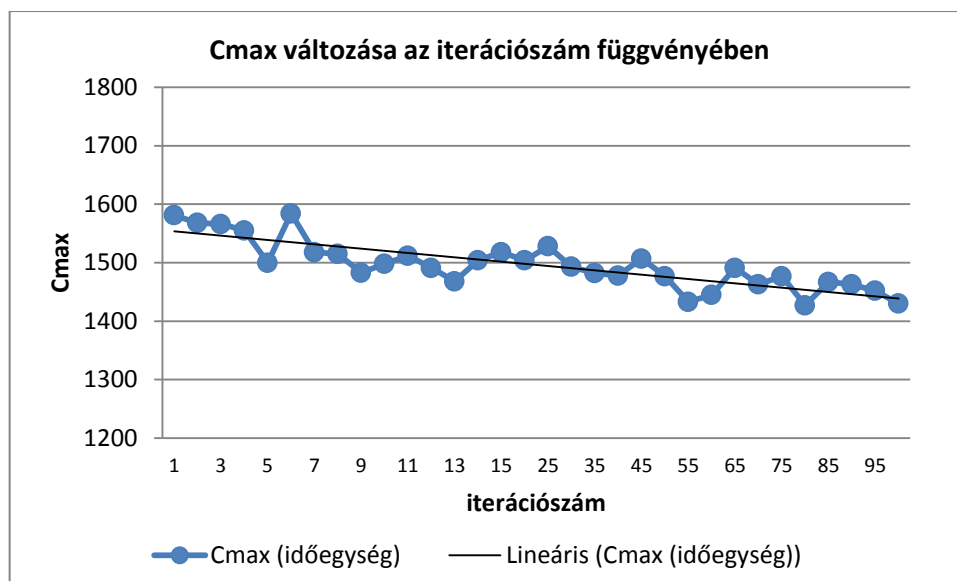
A mért adatok alapján jól látható, hogy a determinisztikus eljárások rendelkeznek a legjobb futási időkkel, a random kereső eljárás ilyen iteráció számnál meg sem tudja közelíteni a többi algoritmus futási sebességét, jelentősen rosszabb időkkel teljesít.

A determinisztikus algoritmusok esetén az figyelhető meg, hogy kis feladatok esetében képesek a random keresővel egyező, vagy ahhoz közeli C_{max} eredményeket produkálni, viszont nagyobb feladatok esetében már csak elvétve sikerül jobb eredményt hozni. Néhány esetben azonban viszonylag közeli értéket képesek adni, ami jó teljesítménynek számít, főleg abban az esetben, ha az algoritmus futási idejének csökkentése fontosabb szempont, mint a minél jobb C_{max} érték.

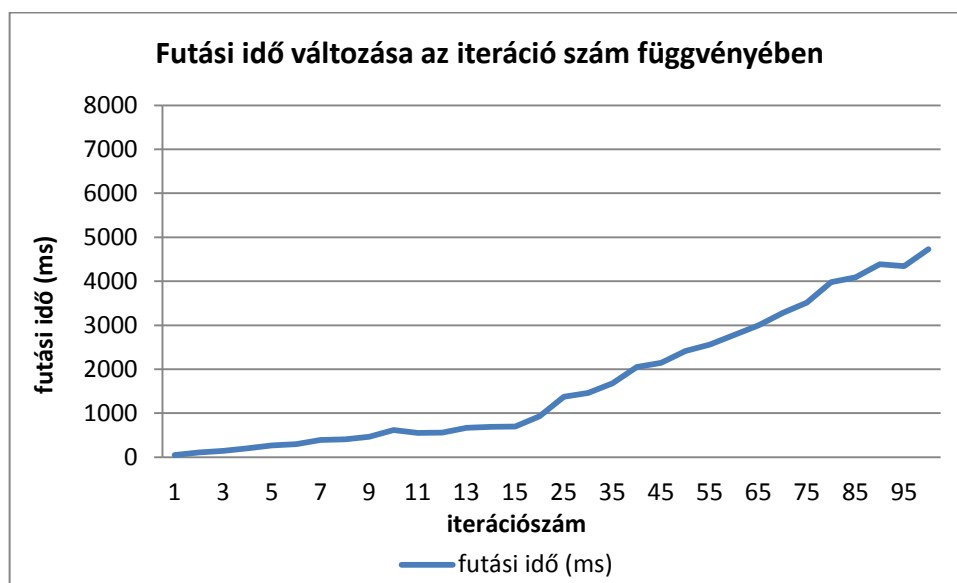
Az erőforrás-korlát igény szerinti rendezés, és az átfutási idő szerinti rendezés között is különbség figyelhető meg. Általánosságban véve az erőforrás-igény szerint rendező algoritmusok jobb teljesítményt nyújtottak, mely főként a futási időben nyilvánult meg. A C_{max} értékek viszont nem voltak jellemzően jobbak egyik algoritmus esetében sem hosszú távon, ami annak köszönhető, hogy a determinisztikus eljárások hatékonysága jelentős mértékben függ a feladattól.

Ezzel ellentétben a sztochasztikus eljárás feladattól függetlenül generált – a legtöbb esetben lényegesen – jobb C_{max} eredményeket, ezért érdemes arra vonatkozóan is méréseket végezni, hogy mit jelent az iteráció szám fokozatos növelése a C_{max} optimalizálásának és a sebesség növekedésének szempontjából a random keresés során.

A következő mérés során megfigyelhetjük, hogy hogyan közelíti az optimumot a mérések során a random kereső függvény, valamint hogy milyen mértékben növekszik a végrehajtási sebesség az iteráció szám növelés függvényében.

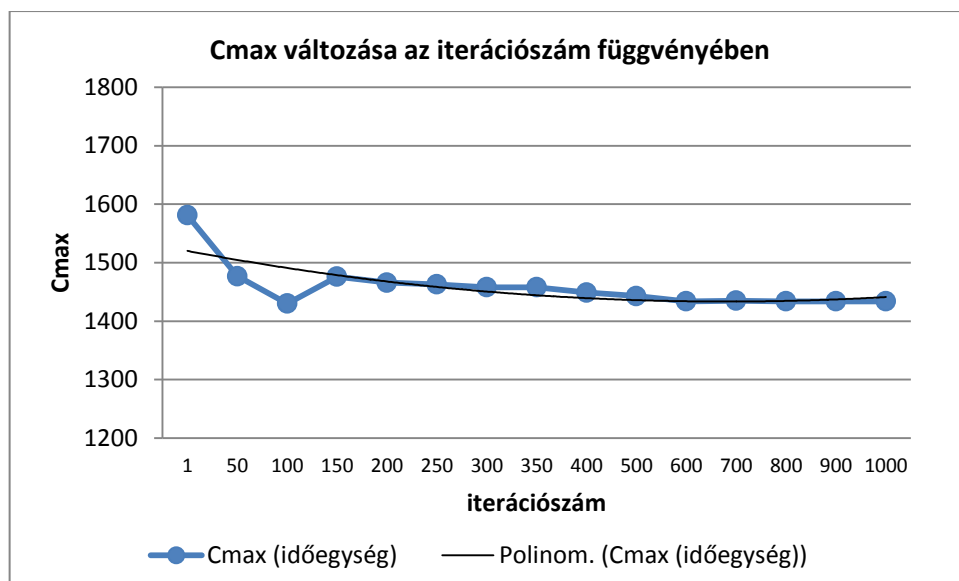


5.5. grafikon: Mérési eredmények, a C_{max} változása az idő függvényében random keresés esetén



5.6. grafikon: Mérési eredmények, az algoritmus futási idejének változása az idő függvényében random keresés esetén

A két grafikon alapján jól látható, hogy a random keresés esetén az iteráció számot növelve egyre jobb eredményeket kapunk a C_{max} értékére, amely egy bizonyos iteráció szám után stagnál (5.7 grafikon).



5.7. grafikon: Mérési eredmények, a C_{max} változása az idő függvényében random keresés esetén, nagyobb iterációs léptékkel

Az eredmények alapján levonható a következtetés, hogy a random kereső eljárás egyre inkább az optimumhoz közeli megoldást adja az iteráció számának növelése esetén, az optimális ütemtervhez közeledve pedig egyre kisebb mértékű az optimumtól való eltérés és egyre nagyobb az algoritmus futási ideje.

5.3 Összegzés

Az alkalmazás a tesztelés során az elvárásoknak megfelelően funkcionált, az elvárásoknak sikerült megfelelnie, segítségével a kívánt tesztek megvalósíthatóak voltak, az ütemező algoritmusok tesztelése lehetséges volt.

Az algoritmusok eredményeinek értékelése igazolta a program működésének hatékonyságát, a tesztelés során nem volt tapasztalható rendellenes működés, ami befolyásolta volna az eredményeket, ezért a tesztekben levont következtetések megfelelő eredményt adtak.

VI. ÖSSZEFOGLALÁS

A dolgozat célja az volt, hogy az erőforrás-korlát nélküli és erőforrás-korlátos projektütemezési feladatok az alkalmazás segítségével modellezhetőek és megoldhatóak legyenek. Az első részben, a szakirodalmi kutatás során ismertettem a probléma elméleti hátterét, valamint az eddigi kutatási eredményeket. Ezt követően felállítottam egy modellt, amely alkalmas a vizsgált feladatok reprezentálására és megoldására, majd a szükséges technológiák használatával implementáltam.

Az alkalmazás képes modellezni és megoldani az erőforrás-korlát nélküli projektütemezési feladatokat a *CPM* módszer alkalmazásával, figyelembe véve a precedencia gráfból adódó megelőzési relációkat és elkészíteni az optimális megoldási ütemtervet.

A program képes kezelni az erőforrás-korlátos projektütemezési feladatokat egy konstans erőforrás esetén, figyelembe véve egyszerre több olyan korlátozást is, mint a gráfban lévő megelőzési relációk, az egyes aktivitások tartalékideje és a hozzájuk rendelt erőforrás-igény, valamint a felállított erőforráskorlát. A megoldó algoritmusoknak köszönhetően több ütemterv generálása is lehetséges a projekt teljes átfutási idejének minimalizálására, a sztochasztikus kereső algoritmus segítségével pedig optimum közeli megoldás nyerhető.

A modell kialakításából adódóan lehetőség van a jövőben az alkalmazás továbbfejlesztésére. A főbb fejlesztési irányokat a több, egyszerre rendelkezésre álló erőforrás esetén felmerülő problémák, a megszakítható feladatok lehetősége és a heurisztikus értékelő megoldások határozzák meg. Céлом, hogy a későbbiekben az alkalmazás funkcióit bővítsem, tovább növelve a hatékonyságát és a hasznosságát.

VII. KÖSZÖNETNYILVÁNÍTÁS

A kutató munka a Miskolci Egyetem stratégiai kutatási területén működő Mechatronikai és Logisztikai Kiválósági Központ keretében valósult meg.

IRODALOMJEGYZÉK

- [1] Dr. Kulcsár Gyula [2012/13]: A termelésinformatika alapjai jegyzet ea.2., 18-21.old.
- [2] Dr. Kulcsár Gyula [2012/13]: A termelésinformatika alapjai jegyzet ea.2, 21.old .
- [3] Dr. Kulcsár Gyula [2013/14]: Virtuális vállalat jegyzet ea.3., 43.old.,
- [4] Dr. Kulcsár Gyula [2013/14]: Virtuális vállalat jegyzet ea.3., 44.old.,
- [5] Dr. Kulcsár Gyula [2013/14]: Virtuális vállalat jegyzet ea.3. 45.old.,
- [6] R.Kolisch, R.Padman [2001]: An integrated survey of deterministic project scheduling
- [7] Rainer Kolisch, Sönke Hartmann [1999]: Project Scheduling International Series in Operations Research & Management Science, Springer, Volume 14, pp 147-178
- [8] Dirk Briskorn, Sönke Hartmann [2009]: A survey of variants and extensions of the resource-constrained project scheduling problem
- [9] Michael L. Pinedo [2005]: Planning and Scheduling in Manufacturing and Services, *Springer*, 53.old.
- [10] Michael L. Pinedo [2005]: Planning and Scheduling in Manufacturing and Services, *Springer*, 53.old.
- [11] Michael L. Pinedo [2005]: Planning and Scheduling in Manufacturing and Services, *Springer*, 53.old.
- [12] Michael L. Pinedo [2005]: Planning and Scheduling in Manufacturing and Services, *Springer*, 53.old.
- [13] Michael L. Pinedo [2005]: Planning and Scheduling in Manufacturing and Services, *Springer*, 54.old.
- [14] Michael L. Pinedo [2005]: Planning and Scheduling in Manufacturing and Services, *Springer*, 54.old.
- [15] Michael L. Pinedo [2005]: Planning and Scheduling in Manufacturing and Services, *Springer*, 54.old.
- [16] Dr. Kulcsár Gyula [2013/14]: Virtuális vállalat jegyzet gy.05., 12.old.,
- [17] Dr. Kulcsár Gyula [2013/14]: Virtuális vállalat jegyzet, gy.05.,14.old.,
- [18] Michael L. Pinedo [2005]: Planning and Scheduling in Manufacturing and Services, *Springer*, 56.old.

- [19] Michael L. Pinedo [2005]: Planning and Scheduling in Manufacturing and Services, *Springer*, 57.old.
- [20] Michael L. Pinedo [2005]: Planning and Scheduling in Manufacturing and Services, *Springer*, 57.old.
- [21] Michael L. Pinedo [2005]: Planning and Scheduling in Manufacturing and Services, *Springer*, 57.old.
- [22] Dr. Kulcsár Gyula [2013/14]: Virtuális vállalat jegyzet gy.05., 22.old.
- [23] Dr. Kulcsár Gyula [2013/14]: Virtuális vállalat jegyzet gy.05., 22.old.,
- [24] Dr. Kulcsár Gyula [2013/14]: Virtuális vállalat jegyzet gy.05., 23.old
- [25] Wikipédia, a Java nyelv története
[http://hu.wikipedia.org/wiki/Java_\(programoz%C3%A1si_nyelv\)](http://hu.wikipedia.org/wiki/Java_(programoz%C3%A1si_nyelv))
 (ell.:2014.10.)
- [26] Wikipédia, a Java nyelv története
[http://hu.wikipedia.org/wiki/Java_\(programoz%C3%A1si_nyelv\)](http://hu.wikipedia.org/wiki/Java_(programoz%C3%A1si_nyelv))
 (ell.:2014.10.)
- [27] Wikipédia, az Eclipse fejlesztői környezet
[http://hu.wikipedia.org/wiki/Java_\(programoz%C3%A1si_nyelv\)](http://hu.wikipedia.org/wiki/Java_(programoz%C3%A1si_nyelv))
 (ell.:2014.10.)
- [28] Maven ismertető <http://www.jtechlog.hu/2010/04/maven-kezdolepesekek.html> (ell.:2014.10.21)
- [29] JAX-B ismertető <http://hu.wikipedia.org/wiki/JAX-B> (ell.:2014.10.21)
- [30] Tesztvezérelt fejlesztés ismertető
<http://codeguide.hu/2013/08/08/tesztvezereelt-fejlesztes-tdd/>
 (ell.:2014.10.21)
- [31] Global Optimization jegyzet 61.old., G.-Tóth Boglárka, BME
www.math.bme.hu/~bog/GlobOpt/globopt_hun.pdf (ell.:2014.10.21)