ALGORÍSMIA

Airline Scheduling: Documentació

G15

Sergi Avila Sangüesa

Alejandro de Haro Ruiz

Pablo Ruiz Martinez

Index

1. Descripció del programa

El programa, *AirlineScheduling*, és un solver del problema de max-flow adaptat al cas concret de la problemàtica de distribuït pilots d'una aerolínia als diferents vols que hi ha planificats fer entre un seguit de ciutats al llarg d'un dia.

1.1. Compilació

Juntament amb el programa, a la carpeta arrel del projecte, es troba un fitxer *Makefile* de compilació automàtica. Tota l'etapa de compilació està automatitzada i apropiadament configurada, de forma que les úniques comandes que s'han d'executar són:

```
$ make
$ ./AirlineScheduling.exe
```

A la fase de compilació es fa servir el compilador *g++* configurat amb l'estàndard *c++11* per tal de ser compatible amb els equips de la FIB. La compilació s'executa amb un seguit de *flags* altament restrictius per tal d'assegurar que no hi ha cap element que pugui induir a l'error, i s'optimitzen els binaris de sortida tant com el compilador permet per tal de reduir el temps d'execució.

1.2. Execució

Per tal que el programa sigui més flexible que un simple programa estàtic de consola, les diferents funcionalitats implementades al programa han estat condicionades per tal que l'usuari que executa el programa pugui triar abans de cada execució quines funcionalitats de totes les implementades vol fer servir i quines no.

A continuació es detalla un seguit d'opcions d'execució a l'hora de fer funcionar el programa, que no casualment coincideixen amb les funcionalitats bàsiques implementades:

- Execució: l'execució requereix de 3 camps d'entrada amb un total de 4 arguments per al programa:

  ./AirlineScheduling [-12Lh] [-a algorithm] [-AM]

- Bloc 1: opcions de versió
  - **-1**: executa el problema en la versió 1 especificada per l'enunciat amb tots els algorismes detallat al següent camp.
  - **-2**: executa el problema en la versió 2 especificada per l'enunciat amb tots els algorismes detallat al següent camp.

- **-L, --ALL**: executa de forma seqüencial les versions 1 i 2 del programa amb tots els algorismes detallats al següent camp.
- **-h, --help**: mostra l'ajuda a l'execució del programa.

- Bloc 2: opcions d'algorisme

  - **-a, --algorithm**: camp obligatori que precedeix l'algorisme en sí
    - **DI**: executa el programa en les versions seleccionades fent servir l'algorisme de flux de bloqueig de Dinic.
    - **EK**: executa el programa en les versions seleccionades fent servir l'algorisme d'Edmonds-Karp.
    - **FF**: executa el programa en les versions seleccionades fent servir el mètode de Ford-Fulkerson implementat amb DFS.

- Bloc 3: opcions d'entrada

  - **-A, --auto**: entrada automàtica; executa un bucle sobre tots els arxius de dades de forma seqüencial.
  - **-M, --manual**: entrada manual; permet l'entrada manual de dades, o, amb una combinació encertada de pipes i redireccions de canals d'entrada/sortida, l'entrada de les dades d'un fitxer concret i la sortida de dades a un altre fitxer concret. Una execució d'aquest tipus seria de la forma:

input_file.ext | ./Airlinescheduling [-12Lh] [-a algorithm] [-AM] > output_file.ext

Hi ha un seguit de consideracions a tenir en compte i que són comunes a totes les configuracions de l'execució:

- L'entrada de dades manual es fa pel **canal d'entrada estàndard**.
- El programa **sempre** mostra com a sortida la sortida del programa descrita a l'enunciat, pel canal estàndard, de forma que es pugui redirigir a un fitxer en cas de voler fer-ho.
- Per a totes les execucions, és **obligatori introduir una opció de versió, una d'algorisme i una d'entrada**. En cas contrari, o en cas que alguna de les opcions introduïda no sigui reconeguda, es mostrarà la pantalla descriptiva del funcionament del programa i aquest acabarà la seva execució.
- Els fitxers de dades que es generen durant les execucions es guarden sota el directori ***results***, tal i com es descriu a l'apartat 1.3.

1.3. Estructura de directoris

El projecte està estructurat en els següents directoris, amb les següents utilitats:

- ***bin***: directori on es generen tot els arxius .o durant la compilació.
- ***data***: directori de dades on el programa ve a buscar els fitxers d'entrada quan es troba en mode d'entrada automàtica.

- ***docs***: directori on es troben els documents presentats i els fets servir al llarg del projecte.
- ***results***: directori on es guarden els diferents arxius del programa en funció del mode d'execució.
    - *results/benchmark*: subdirectori on es guarden els arxius de resultats de cost temporal d'execució de l'algorisme.
    - *results/output*: subdirectori on es guarden els arxius de sortida del programa (la mateixa sortida que es mostra pel canal estàndard, però copiada a un arxiu).
    - *results/simulations*: subdirectori on es guarden els arxius de resultats de pilots per instància. A partir d'aquests fitxers s'han generats els fitxers *Resultado1.txt* i *Resultado2.txt*.
- ***src***: directori on es situa el codi font del projecte.
    - *src/algorithms*: subdirectori on es guarden els arxius font de les classes corresponents a les implementacions dels diferents algorismes.
    - *src/benchmark*: subdirectori on es guarden els arxius font de les classes encarregades de definir el marc de les proves de temps i les execucions.
    - *src/structures*: subdirectori on s'emmagatzemen els arxius font de les classes encarregades de l'emmagatzematge de les dades del programa.
    - *src/utils*: subdirectori on s'emmagatzemen la resta d'arxius font que es fan servir al programa.

1.4. Estructura de classes

Per tal de **modularitzar** el projecte i abstraure les feines assignades a cadascun dels membres de les feines dels demés, s'ha definit una estructura de classes on se separa completament la implementació dels algorismes de tot el relacionat amb formats d'entrada i sortida, així com d'altres requeriments del projecte.

Aquesta estructura és la següent:

main

- *AirlineSchedule*: *main*, arxiu executable amb les funcions de traducció bàsiques de les opcions d'entrada del programa per tal d'executar una instància de *Benchmark* en el format correcte demanat.

benchmark

- *Benchmark*: classe encarregada d'encapsular el comportament dels diferents modes d'execució, així com les mesures de temps d'execució. Des d'aquesta classe es controla:
    - Quines i quantes instàncies es fan servir com a entrada.
    - Per a quantes versions s'executen les instàncies.
    - El flux d'execució detallat d'una simulació individual completa.

- Les mesures del temps d'execució de l'algorisme, tal i com s'explica a la descripció de la classe *Chrono*.

■ *Simulation*: classe que gestiona l'execució d'una simulació individual. Proporciona la interfície per tal de donar un bloc de suport i de dades comú a tots els algorismes, i abstrau la part pròpiament de càlcul del *Benchmark* i qualsevol altra classe.

A més a més, aquesta classe proporciona un seguit de mètodes que permeten gestionar cada pas del flux d'execució d'una simulació, des de l'inicialització de les variables fins a l'execució de l'algorisme i la finalització de la simulació:

- *load()*: carrega els arxius de dades a ser llegits per la funció *input()*
- *input()*: gestiona l'entrada de dades automàtica al programa en el format adequat per a que l'algorisme en faci ús.
- *manualInput()*: gestiona l'entrada manual de dades en el format adequat per a que l'algorisme en faci ús.
- *initialize()*: gestiona la inicialització de dades i estructures preparatòries per a la correcta execució de l'algorisme.
- *setAlgorithm()*: permet canviar d'algorisme en qualsevol moment.
- *run()*: executa l'algorisme. Aquesta funció conté únicament la crida a l'algorisme, per tal que durant les mesures de temps, col·locant únicament els comptadors abans i després d'aquesta crida, es perdi el mínim de temps possible en operacions altres que l'algorisme en sí.
- *processResults()*: post processat dels resultats, per tal d'interpretar-los i recuperar la solució.
- *end()*: tasques per a finalitzar l'execució de la simulació i deixar-la en l'estat inicial.

Amb aquest control de flux detallat, es poden modelar una gran quantitat de situacions, des de mesures precises de temps d'execució, fins a casos especials en que certes condicions fan que algunes parts del flux d'execució hagin de ser omeses (com és el cas de l'execució d'una única instància).

algorithm

■ *Algorithm*: classe pare de tots els algoritmes. Defineix la interfície a implementar per totes les seves subclasses: una única operació *algorithm()* amb uns paràmetres prefixats: font, sumider i graf.

■ *FordFulkerson*: classe pare de les seves dues variants: amb BFS (anomenat Edmonds-Karp) i amb DFS. Implementa el mètode base de Ford-Fulkerson en el que estan basades ambdues versions, i defineix com a interfície en mètode

travelGraph(), per tal que cada subclasse concreta implementi el mètode de recorregut de graph més adient.

- *Edmonds-Karp*: classe representant de l'algorisme d'Edmonds-Karp. Implementa com a *travelGraph()* un recorregut en cerca per amplada (BFS).
- *FordFulkersonDFS*: classe representant de l'algorisme de Ford-Fulkerson utilitzant DFS. Implementa com a *travelGraph()* un recorregut en cerca per profunditat (DFS).
- *DinicBlockingFlow*: classe filla directament d'*Algorithm* que implementa l'algorisme de Dinic per flux de bloqueig.

structures

- *Edge*: implementa els camps a emmagatzemar per una arista al problema:
  - Ciutat d'origen
  - Ciutat de destí
  - Capacitat
  - Límit inferior de capacitat
- *vèrtex*: implementa els camps a emmagatzemar per un vèrtex del problema:
  - Ciutat
  - Hora de sortida/arribada
  - Demanda
- *Graph*: implementa el graf del problema mitjançant llistes d'adjacència. Conté:
  - Contenidor de vèrtexs del graf.
  - Llista d'adjacències de la relació entre vèrtexs del graf.

utils

- *Chrono*: implementa una classe cronòmetre per tal de dur a terme mesures de temps de codi C++. Consta de tres mètodes:
  - *start()*: inicia el cronòmetre. Per tal de pertorbar la mesura el mínim possible, l'última acció realitzada pel mètode és la de prendre la mesura de temps, de forma que immediatament després s'ha de situar el codi a mesurar.
  - *stop()*: para el cronòmetre. Per tal de pertorbar la mesura el mínim possible, la primera acció realitzada pel mètode és la de prendre la mesura de temps.
  - *duration()*: consulta l'interval de temps entre la crida a *start()* i *stop()*.

1.5. Fitxers de sortida

El programa, en diversos dels seus modes, genera tres tipus d'arxius diferents: arxius de mesures de temps (*benchmark*), arxius de sortida de programa (*output*) i arxius de solució de simulació (*simulation*). Aquests arxius tenen els següents formats:

- ***benchmark***

  Benchmark[VERSIÓ]_[ALGORISME].txt

  - VERSIÓ: *1, 2*
  - ALGORISME: *EK, FF-DFS, D*I

  Internament, aquest arxiu està estructurat en dues columnes separades per una tabulació:
  - Columna 1: nom de la instància que s'està provant.
  - Columna 2: temps d'execució de l'algorisme (en segons).

- ***output***

  Output[VERSIÓ]_[ALGORISME].txt

  - VERSIÓ: *1, 2*
  - ALGORISME: *EK, FF-DFS, DI*

  Internament, aquest arxiu està estructurat en diversos blocs compresos per:
  - *k*: número mínim de vols necessaris per a efectuar totes les rutes.
  - *k* línies, cadascuna amb la relació de vols efectuats per a aquell pilot en concret.

- ***simulations***

  Resultado[VERSIÓ]_[ALGORISME].txt

  - VERSIÓ: *1, 2*
  - ALGORISME: *EK, FF-DFS, DI*

  Internament, aquest arxiu està estructurat en dues columnes separades per un espai:
  - Columna 1: nom de la instància que s'està provant.
  - Columna 2: nom mínim de pilots necessaris per tal de cobrir totes les rutes.

2. Algorismes

Pel que fa els algorismes que resolen la part de max-flow del problema, hem implementat tres. Dos d'ells utilitzen el mètode Ford-Fulkerson: un que troba el camí augmentat mitjançant BFS (l'anomenat algorisme Edmonds–Karp) i l'altre amb DFS. El tercer algorisme es l'algorisme de Dinic, molt semblant al Edmonds–Karp però que utilitza el concepte de flux de bloqueig (*blocking flow*) per trobar diferents camins augmentats alhora en lloc de un sol.

2.1. Mètode de Ford-Fulkerson

L. R. Ford Jr. i D. R. Fulkerson van publicar al 1956 un mètode per a resoldre el problema de max-flow, que troba el flux màxim que pot circular per una xarxa de flux. Aquest mètode funciona com un algorisme voraç on a cada iteració s'augmenta el màxim flux possible al camí augmentat que s'ha trobat. Ara bé, no es pot considerar un algorisme per si sol ja que no s'especifica com s'ha de trobar aquest camí augmentat i, per tant, pot variar a cada implementació. La implementació on el camí es troba mitjançant un recorregut de cerca per amplada (BFS) és l'algorisme d'Edmonds-Karp.

Primer es defineixen els conceptes de xarxa residual i camí augmentat. La xarxa residual d'una xarxa de flux $G(V, E)$ és un altre graf $G_f(V, E_f)$ que té els mateixos vèrtexs que l'original i té arestes $\{u, v\} \in V \times V$ amb capacitat c_f

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) si (u,v) \in E \\ f(v,u) si (v,u) \in E \\ 0 otherwise \end{cases}$$

Un camí augmentat *p* és un camí de *s* a *t* a la xarxa residual. La capacitat residual de *p* (*bottleneck capacity*) és $c_f(p) = min(c_f(\{u, v\}) : \{u, v\} \in p\}$. Per la definició de la xarxa residual, podem augmentar en $c_f(p)$ el flux de les arestes del camí augmentat a la xarxa original sense violar les restriccions de capacitat.

El mètode funciona de la següent manera:

1. S'inicialitzen tots els fluxos a 0.
2. Es genera la xarxa residual.
3. Mentre existeixi un camí augmentat a la xarxa residual:
    a. Troba el mínim de les capacitats residuals de les arestes del camí.
    b. Augmenta el flux de les arestes del camí en el mínim trobat al pas anterior.
    c. Genera la nova xarxa residual.
4. Retorna el flux màxim de la xarxa.

Els passos 3.b. i 3.c. es poden simplificar disminuint la capacitat residual a la xarxa residual i augmentant les arestes en sentit contrari.

En la implementació del programa, el mètode *FordFulkerson::algorithm()* implementa tots els passos excepte buscar el camí augmentat. D'això s'encarrega el mètode *travelGraph()*, que està implementat diferent a cada subclasse.

2.2. Algorisme d'Edmonds-Karp

L'any 1972, en Jack Edmonds i en Richard M. Karp van proposar que una manera eficient d'implementar el mètode de Ford-Fulkerson era trobant a cada iteració un dels camins amb menys arestes. Això es pot aconseguir amb una cerca per BFS del node *sink* (t). Si a la cerca no es troba el node t, no hi ha cap camí augmentat. La resta de l'algorisme és exactament igual al mètode descrit a l'apartat anterior.

2.3. Algorisme de Ford-Fulkerson amb DFS

Aquest algorisme no té cap diferència amb l'anterior a excepció que ara el camí augmentat es troba mitjançant DFS. D'aquesta manera no hi ha cap garantia que el camí trobat sigui el que menys arestes té. Ara, el camí que es trobarà primer serà el que contingui els nodes amb índex més petit a la matriu d'adjacències (perquè és així com es recorre la matriu en el cas concret de la implementació del projecte).

2.4. Algorisme de Flux de Bloqueig de Dinic

Sense ser conscient de les publicacions de Ford-Fulkerson i d'Edmonds-Karp, l'any 1970, en Yefim Dinitz va publicar un altre algorisme molt semblant als anteriors. La principal diferència és que amb l'algorisme de Dinic s'augmenta el flux de més d'un camí a cada iteració. Per dur això a terme s'introdueixen els conceptes de graf de nivells (*level graph*) i de flux de bloqueig (*blocking flow*).

El graf de nivells és el mateix que la xarxa residual on a cada vèrtex v se li assigna un nivell. Aquest nivell és la distància més curta de *s* a *v*. Això permet agafar tots els camins més curts, que seran aquells que tinguin els vèrtex amb nivells 0, 1, 2, 3… en ordre.

Un flux és un flux de bloqueig si no hi ha cap altre camí $s-t$ de vèrtexs amb nivells 0, 1, 2, 3… al graf de nivells.

L'algorisme funciona de la següent manera:

1. S'inicialitzen tots els fluxos a 0.
2. Es genera la xarxa residual.
3. Es genera el graf de nivells a partir de la xarxa residual.
4. Mentre existeixi un camí $s-t$ a la xarxa residual:
    a. Mentre existeixi un camí $s-t$ de vèrtex amb nivells 0, 1, 2, 3… al graf de nivells
        i. Troba el mínim de les capacitats residuals de les arestes del camí.

ii. Disminueix les capacitats residuals de les arestes del camí en el graf residual en el mínim trobat al pas anterior. Augmenta la capacitat residual de l'aresta en sentit contrari.

b. Genera el nou graf de nivells.

5. Retorna el flux màxim de la xarxa.

El graf de nivells es genera mitjançant una cerca de t amb BFS. A la nostra implementació això es fa al mètode *DinicBlockingFlow::buildLevelGraph()*. Aquest mètode també retorna cert si hi ha un camí $s-t$ a la xarxa residual i fals en cas contrari. La cerca dels camins amb vèrtex amb nivells 0, 1, 2, 3… al graf de nivells té lloc al mètode *DinicBlockingFlow::blockingFlow()*. La cerca està basada en DFS. Un cop troba un camí, retorna el flux que cal augmentar a aquestes arestes. Retorna 0 quan no hi ha cap camí $s-t$. La suma de tots els fluxos retornats fins que retorna 0 és el flux de bloqueig.

3. Anàlisi de cost temporal

Per analitzar el cost temporal de l'algorisme de resolució, s'analitzen les diverses fases per separat. Per tal de dur a terme aquest anàlisi, es defineixen un seguit de paràmetres d'entrada mitjançant els quals es prendrà referència dels costos temporals:

- f: número de vols que es duran a terme.
- k: número de pilots necessaris per a dur a terme un conjunt de vols. Aquest número de pilots està fitat per la pròpia definició del problema, de forma que $0 \leq k \leq f$.
- v: número de vèrtexs del graf. Aquest número ve determinat per la construcció del problema, ja que sempre hi haurà dos vèrtexs per cada vol que es realitzi, més quatre vèrtexs addicionals $s,\ t,\ s_d,\ t_d$, de forma que $v = 2 \cdot f + 4$
- e: número d'arestes del graf en tot moment. Aquest número ve inicialment determinat per la construcció del problema de forma que $e = 3 \cdot f + 3$: $\{s,\ o_i\}\ \forall i$, $\{o_i,\ d_i\}\ \forall i$, $\{d_i,\ t_d\}\ \forall i$, $\{s,\ s_d\}$, $\{s_d,\ t_d\}$ i $\{t_d,\ t\}$. Tot i això, després de l'aplicació de la transformació del problema amb les restriccions de les diferents versions i la reducció a un problema de max-flow sense demandes ni fites inferiors, el nombre d'arestes és desconegut però sempre acompleix que $e \in O(f^2)$.

Els passos que es duen a terme durant l'execució de l'algorisme són els següents:

1. En primer lloc, es realitza l'entrada de dades amb cost:

$$O(v + e) \equiv O(2 \cdot f + 4 + 3 \cdot f + 3) \equiv O(f)$$

2. Per a determinar el nombre mínim de pilots necessaris es realitzarà una cerca dicotòmica de $k \in [0,\ f]$, ja que sempre en el pitjor cas es pot fer una assignació de 1 pilot per vol. Per tant, les transformacions que modifiquen k i l'algorisme de max-flow s'executen $O(log(f))$ vegades.

3. Després, s'ha d'afegir les arestes addicionals indicades per les restriccions del problema:

   - **Versió 1:** Per cada parell de vols $\{u_i,\ v_i\}$, $\{u_j,\ v_j\}$ es comprova si la transició $\{v_i,\ u_j\}$ és possible i si ho es, s'afegeix una aresta. **Cost:** $O(f^2)$

   - **Versió 2:** Si en una parella de vols $\{u_i,\ v_i\}$, $\{u_j,\ v_j\}$ es pot fer una transició $\{v_i,\ u_j\}$, es fa un pas més per intentar trobar un vol $\{u_k,\ v_k\}$ de forma que la transició $\{v_j,\ u_k\}$ sigui possible, i per tant $\{u_i,\ v_k\}$ també ho sigui. **Cost:** $O(f^3)$ (mai serà exactament cubic, ja que si $\{v_i,\ u_j\}$ és possible, aleshores $\{u_j,\ v_i\}$ és impossible, doncs $\{u_i,\ v_i\}$ surt abans que $\{u_j,\ v_j\}$).

4. Es creen les arestes $\{s_d,\ o_i\}\ \forall i$. **Cost:** $O(f)$

5. Es creen les arestes $\{o_i,\ t_d\}\ \forall i$. **Cost:** $O(f)$

6. A continuació, es procedeix a realitzar la reducció a un problema de max-flow sense demandes ni límits inferiors:

   i. S'eliminen els límits inferiors de les arestes dels vols. **Cost:** $O(f)$

   ii. S'eliminen les demandes creant arestes $\{s,\ s_d\}$, $\{s,\ d_i\}\ \forall i$, $\{o_i,\ t\}\ \forall i$ i $\{t_d,\ t\}$. **Cost:** $O(2 \cdot f + 2) \equiv O(f)$

7. Execució de l'algorisme de max-flow

   - **Ford-Fulkerson (DFS):** El cost de l'algorisme es $O(C \cdot e)$ sent C el flux màxim del graf, i e el nombre d'arestes. En el cas concret del problema, **cost:** $O((k+f) \cdot e) \equiv O(f \cdot e)$.
   - **Edmonds-Karp:** El cost es $O(v \cdot e^2)$**,** en el cas concret del problema, **cost:** $O((2 \cdot f + 4) \cdot e^2) \equiv O(f \cdot e^2)$.
   - **Dinic's Blocking Flow:** El cost d'aquest algorisme es $O(v^2 \cdot e)$, de forma que en el cas concret del problema, **cost:** $O((2 \cdot f + 4)^2 \cdot e) \equiv O(f^2 \cdot e)$.

8. Post-processat de les dades: per tal de recuperar la solució, s'ha de recórrer el graf residual passant per totes aquelles arestes que estan saturades. D'aquesta forma, el cost total del post-processat és igual al nombre de vols f.

Cost total

- **Versió 1 - Ford-Fulkerson :** $O(f + f^2 + log(f) \cdot (k+f) \cdot e + f) \equiv O(f^2 + f \cdot log(f) \cdot e)$
- **Versió 1 - Edmonds-Karp:** $O(f + f^2 + log(f) \cdot f \cdot e^2 + f) \equiv O(f^2 + f \cdot log(f) \cdot e^2)$
- **Versió 1 - Dinic:** $O(f + f^2 + log(f) \cdot f^2 \cdot e + f) \equiv O(f^2 + f^2 \cdot log(f) \cdot e)$

- **Versió 2 - Ford-Fulkerson :** $O(f + f^3 + log(f) \cdot (k+f) \cdot e + f) \equiv O(f^3 + f \cdot log(f) \cdot e)$
- **Versió 2 - Edmonds-Karp:** $O(f + f^3 + log(f) \cdot f \cdot e^2 + f) \equiv O(f^3 + f \cdot log(f) \cdot e^2)$
- **Versió 2 - Dinic:** $O(f + f^3 + log(f) \cdot f^2 \cdot e + f) \equiv O(f^3 + f^2 \cdot log(f) \cdot e)$

4. Diferències entre la versió 1 i la versió 2

La principal diferència entre la versió 1 i la versió 2 del problema es troba en les restriccions que permeten la creació d'arestes entre vèrtexs: mentre que la versió 1 és més restrictiva (només permet generar un conjunt d'arestes mínim E_1), la versió 2 permet, a més de totes les arestes de la versió 1, afegir-ne de noves (E_2) entre vèrtexs que abans no estaven directament connectats, de forma que $E_1 \subseteq E_2$.

Com que en ambdós casos el conjunt de vèrtex és el mateix ($V_1 = V_2$), es pot dir que la xarxa de fluxos originada per la versió 1 és un subgraf d'expansió (*spanning subgraph*) de la versió 2, cosa que garanteix que qualsevol solució generada per la versió 2 serà com a molt igual a la generada per la versió 1, ja que les arestes addicionals afegeixen la possibilitat de prendre camins que abans no existien, portant així a una solució més òptima. D'aquesta forma, el problema de la versió 1 és un subproblema de la versió 2.

5. Resultats

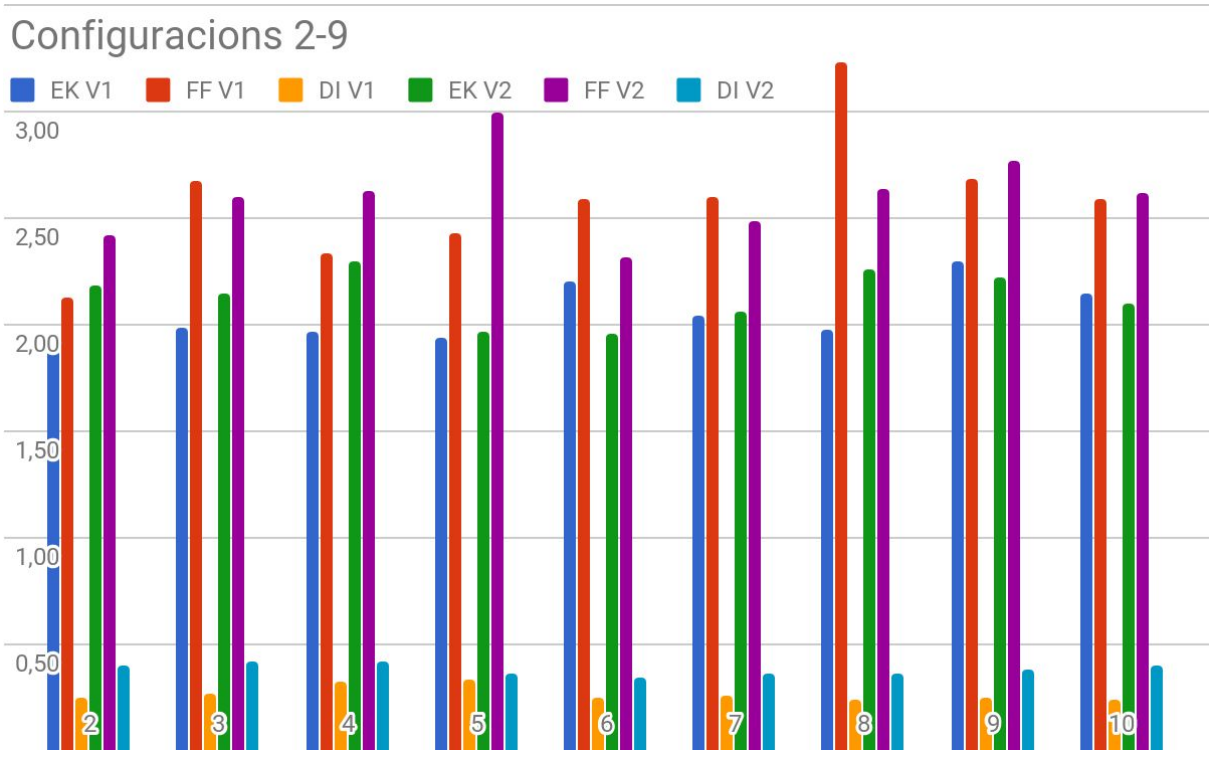
5.1. Taula

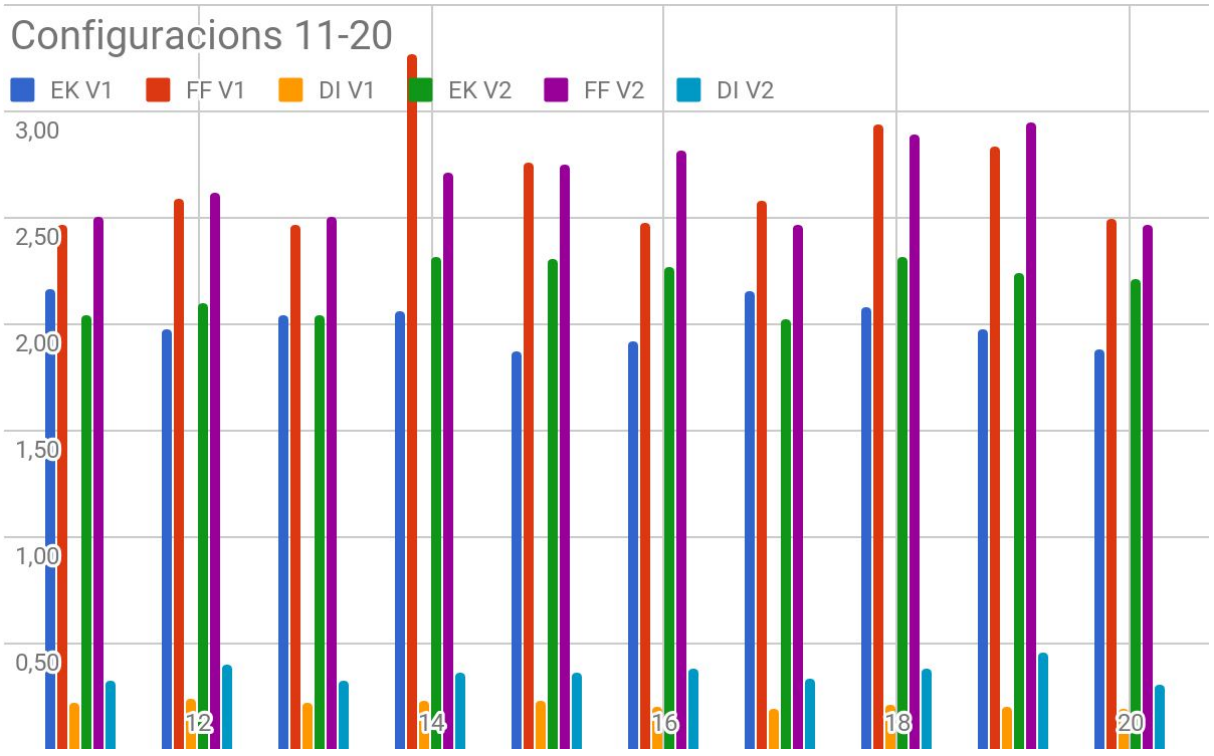
La següent taula mostra els resultats de l'execució del problema amb els diferents algorismes i versions. Cada cel·la és la mitjana del temps d'execució en segons de les 10 instàncies per a cadascuna de les 30 configuracions del problema:

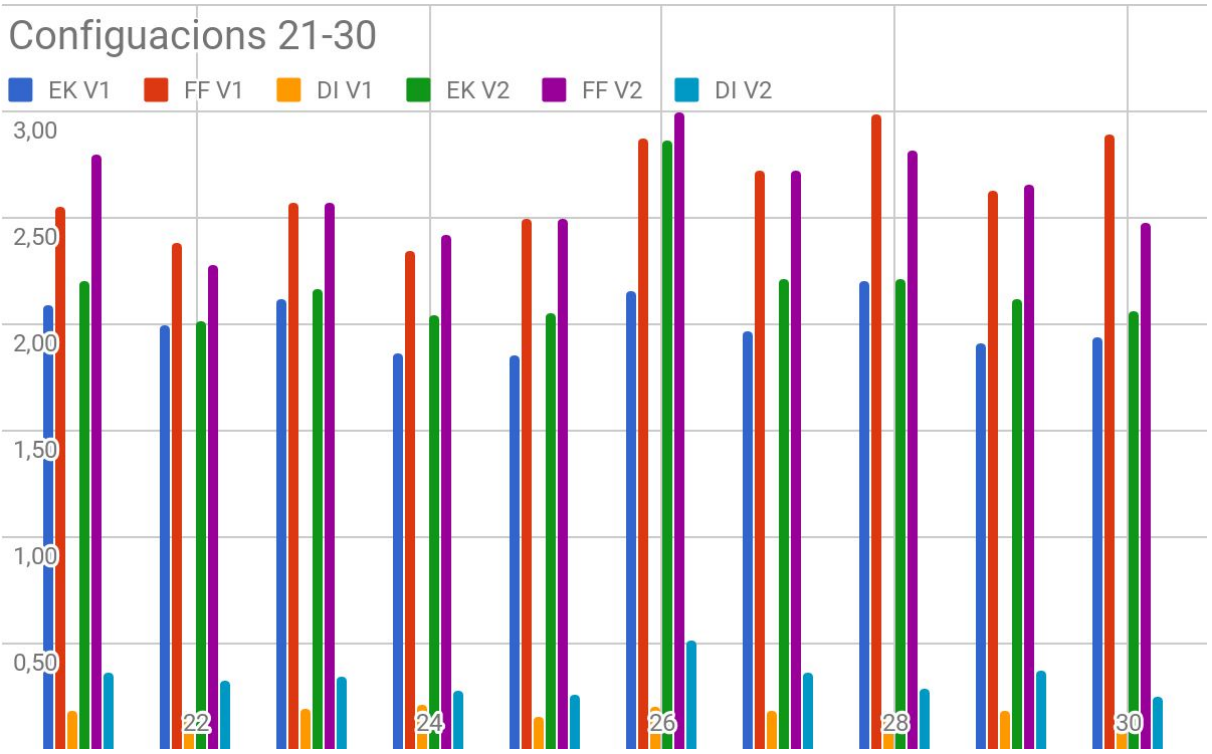
#	EK V1 (s)	FF V1 (s)	DI V1 (s)	EK V2 (s)	FF V2 (s)	DI V2 (s)
2	1,87919	2,11851	0,247	2,18118	2,41968	0,39722
3	1,97646	2,66926	0,262	2,13744	2,59245	0,41118
4	1,95784	2,33197	0,317	2,29451	2,62676	0,41127
5	1,93129	2,42308	0,333	1,96049	2,98844	0,36305
6	2,19572	2,58739	0,245	1,95130	2,30734	0,33627
7	2,03783	2,59620	0,251	2,05572	2,47787	0,35680
8	1,97134	3,22458	0,234	2,25647	2,63509	0,36049
9	2,29589	2,67777	0,241	2,21792	2,76707	0,37673
10	2,13999	2,58914	0,239	2,09065	2,61273	0,39996
11	2,16417	2,46154	0,216	2,03882	2,50398	0,31854
12	1,97193	2,58914	0,239	2,09065	2,61273	0,39996
13	2,03389	2,46154	0,216	2,03882	2,50398	0,31854
14	2,05663	3,26224	0,230	2,31484	2,71020	0,36007
15	1,86959	2,75728	0,224	2,30306	2,74577	0,35669
16	1,91647	2,47493	0,197	2,26526	2,81572	0,37772
17	2,15035	2,57438	0,191	2,02074	2,46062	0,33391
18	2,07758	2,93106	0,209	2,31314	2,88597	0,37334
19	1,97567	2,82690	0,198	2,23582	2,94174	0,45202
20	1,88004	2,49227	0,187	2,20991	2,46352	0,30033
21	2,08322	2,54498	0,181	2,20115	2,79574	0,35586
22	1,99090	2,37451	0,177	2,00593	2,27274	0,32366
23	2,11480	2,56236	0,190	2,15963	2,56923	0,33804
24	1,85955	2,34031	0,207	2,03316	2,41864	0,26959
25	1,85138	2,49305	0,154	2,04510	2,49089	0,25183
26	2,15020	2,86535	0,197	2,85864	2,98672	0,50557
27	1,96442	2,71725	0,179	2,20858	2,71898	0,35503
28	2,19617	2,97832	0,175	2,20846	2,81183	0,28109
29	1,90599	2,62005	0,178	2,11733	2,65461	0,36651
30	1,93853	2,88733	0,166	2,05229	2,47159	0,24209

5.2. Gràfics

Els següents gràfics mostren la mateixa informació que a la taula d'una forma més visual. Estan separats en 3 grups (per raons de visualització) i el temps està en segons.







5.3. Conclusions

La diferencia més clara en els resultats és entre l'algorisme de Dinic i els altres 2, ja que totes les execucions triguen menys de 0.5 segons, mentre que la resta sempre triga més de 1s. Això és degut a que en aquest algorisme el nombre de vèrtexs de la xarxa de flux té menys pes en temps d'execució que el nombre d'arestes, ja que, per les característiques del problema, el nombre d'arestes sempre és estricament major que el nombre de vèrtexs. Per tant, la diferencia entre Dinic i Edmonds-Karp concorda amb el cost calculat a l'apartat anterior.

En el cas de Ford-Fulkerson, en canvi, els resultats reals no concorden amb el cost teòric calculat, ja que en teoria hauria de ser mes ràpid que Dinic i Edmonds-Karp. Per eliminar la qualitat i eficiència del codi com a factor, el compararem amb Edmonds-Karp, ja que l'única diferencia al codi és si per a trobar els camins s'utilitza un BFS o un DFS. Un dels factors que pot haver influït ha estat que hem sigut pessimistes per calcular el cost, ja que hem considerat un nombre d'arestes més gran que el real, cosa que perjudica més al cost de Edmonds-Karp. A la descripció de l'algorisme de Ford-Fulkerson no es va definir de quina forma s'havien de trobar el camins, en canvi, a Edmonds-Karp sí que s'especifica que s'ha de fer servir un BFS. Nosaltres hem utilitzar un DFS per trobar el camí qualsevol, i això pot haver afectat al temps d'execució.

Respecte a les dues versions, la diferencia de temps quadràtic a la 1, i cúbic a la 2 per generar la xarxa de flux es nota en la majoria de configuracions. Tot i així, hi ha casos en que l'execució és mes ràpida a la versió 2. Això és degut a que com els camins de s a t són iguals o més curts que a la versió 1, pot passar que el temps que es perd per generar la xarxa sigui menor que el que es guanya a l'execució de l'algorisme de max-flow.

6. Bibliografia

1. Kleinberg, J. and Tardos, E. (2009). *Algorithm Design*. Boston: Pearson.
2. Cormen, T. H. (2009). *Introduction to algorithms*. Cambridge: MIT Press.
3. Dasgupta, S., Papadimitriou, C. and Vazirani, U. (2008). *Algorithms*. Bostron: Mc Graw Hill Higher Education.
4. Sedgewick, R. and Wyk, C. J. Van (2002). *Algorithms in C++*. Boston: Addison-Wesley.
5. *Maximum Flow Problem*, Wikipedia, the Free Encyclopedia. 10 Jan. 2018. https://en.wikipedia.org/wiki/Maximum_flow_problem
6. *Ford-Fulkerson algorithm*, Wikipedia, the Free Encyclopedia. 17 Dec. 2017. https://en.wikipedia.org/wiki/Ford%E2%80%93Fulkerson_algorithm
7. *Edmonds-Karp algorithm*, Wikipedia, the Free Encyclopedia. 9 Jan. 2018. https://en.wikipedia.org/wiki/Edmonds%E2%80%93Karp_algorithm
8. *Dinic's algorithm*, Wikipedia, the Free Encyclopedia. 12 Jan. 2018. https://en.wikipedia.org/wiki/Dinic%27s_algorithm
9. *Ford-Fulkerson algorithm for maximum flow problem*, Geeks for Geeks. https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/