# Matemática Discreta 2014

## Proyecto Dinic – Max-Flow Min-Cut

Integrantes: Rossi Nahuel

Ruiz Ezequiel

# Índice de contenido

# Introducción

Este informe esta orientado a explicar la implementación del algoritmo Dinic para resolver problemas de Max Flow Min Cut en networks. Se ofrece como complemento de un ReadMe.txt y de la documentación del programa. En el ReadMe.txt se busca dar una visión general y en la documentación se es mas preciso en la descripción del código, mientras que en este informe se profundiza mas en la implementación sin caer en demasiados detalles técnicos.

En este informe se explicara a grandes rasgos las estructuras mas importantes y el comportamiento del main dado que es ahí donde se implementa Dinic mientras que las otras librerías son solo herramientas. También se implemento un pequeño test para analizar el comportamiento del algoritmo en cuanto a rendimiento y robustez. Para una descripción mas detallada se podrá consultar la documentación provista.

# Resumen

Los *networks* son grafos dirigidos con un valor numérico asociado a sus aristas convirtiéndolos a grafos con peso (o grafos ponderados). Los networks son ampliamente usados en el área de la computación para representar diferentes tipos de redes, usualmente de computadoras o dispositivos móviles. Pero esta es una visión muy acotada. Un network puede representar una red neuronal o incluso un mallado de un modelo 3D que soporta determinadas fuerzas. Definir propiedades sobre este objeto es importante para el desarrollo de la industria y la ciencia. Entre la basta cantidad de características que podemos definir vamos a centrar el estudio en el *Max Flow* (Flujo Maximal) y el *Min Cut* (Corte Minimal).[1]

Se buscará escribir un programa en el lenguaje C que resuelva el problema de *Max-Flow Min-Cut* usando el algoritmo de Dinic con el objetivo de minimizar los tiempos y el uso de memoria para que sea escalable.

1 El flujo maximal queda definido como el máximo número de unidades que se pueden enviar del nodo *source* al nodo *sink* y el corte minimal como el conjunto minimal de nodos que incluye al nodo *source* y, a su vez, tiene una suma mínima de pesos con respecto a su complemento.

# Implementación

Llevar el modelo teórico a un lenguaje computación implicó una definición mas detallada de cada concepto teórico en función de una optimización en el tiempo de ejecución y en el uso de memoria de la aplicación. Se explicará dos implementaciones significativas en estos puntos.

*Problema del lado useless*

Debido a la estructura del network el tipo *lado* quedó prácticamente sin uso. Almacenar la información de cada lado en una estructura como esta:

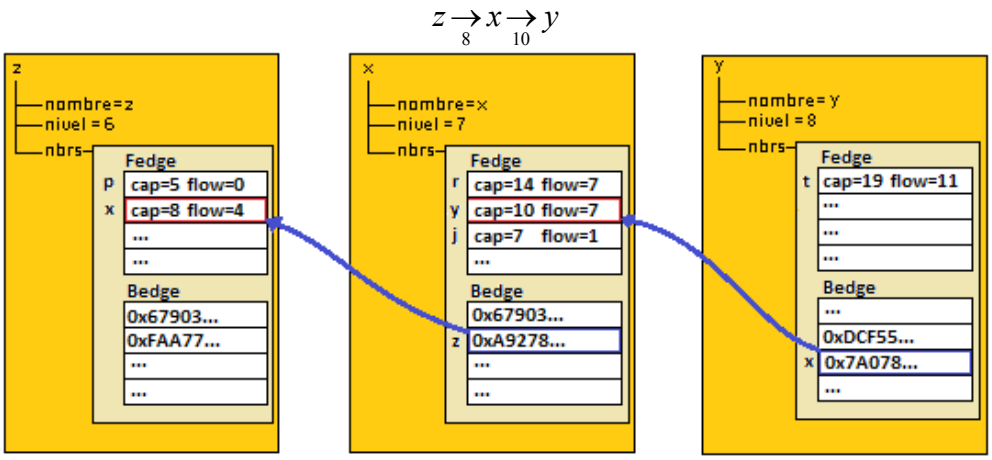
```
/** Estructura principal de un lado.*/
struct LadoSt{
    u64 x;  /**<El nombre del nodo 'x'.*/
    u64 y;  /**<El nombre del nodo 'y'.*/
    u64 c;  /**<La capacidad del lado 'xy'.*/
};
```

nos representaba un problema en el momento de hacer la búsqueda de manera eficiente. Pero, para cumplir con los requisitos de la implementación, se optó usarlo para cargar la información.

El camino que recorre un *lado* desde el momento de su creación hasta que se destruye es bastante corto. El *lado* se crea en el momento de leer standar input, se le asignan los campos con los datos leídos y luego se utiliza ese elemento para almacenar la información en nuestra estructura a través de un elemento del tipo *network*. Al finalizar esto, el elemento del tipo *lado* se destruye y los datos quedan correctamente almacenados.

*Implementación de los vecinos(Neighbourhood)*

Los vecinos de un nodo se implementaron en *NeighbourhoodSt*, esta estructura tiene dos campos, uno formado por una tabla de hash con todos los vecinos forward(*FedgeSt*) y el otro con todos los backwards(*BedgeSt*); y está incluida en *NetworkSt* que, como explicaremos más adelante, es al mismo tiempo un nodo del network y el network completo.



En la imagen se puede ver cómo es la relación de los nodos en un camino por el cual todavía no se envió flujo. Las flechas azules indican relaciones backward entre lados, son punteros a las estructuras que tienen la información. Las letras a la izquierda de cada entrada de la tabla *Fedge* y Bedge son las llaves de la tabla de hash.[2]

Esto se decidió implementar de esta manera para que ciertas búsquedas sean *O(1)*, como por ejemplo buscar la capacidad del la arista $\vec{xy}$ , para esto se realiza una búsqueda *O(1)* de *x* en el network, luego en la tabla *FedgeSt* se busca a *y* también *O(1)* y luego se tiene en el campo *cap* la capacidad. De esta manera se logra una búsqueda total de *O(1)*.

# Modularización

Se decidió modularizar lo mas posible el problema, existiendo una correlación directa entre algunos conceptos teóricos pero no entre otros.

La estructura de directorios es:

***RR/***
- ***apifiles/***
  - ***__bstrlib***
  - ***__uthash***
  - ***__lexer***
  - ***_queue***
  - ***_stack***
  - ***_u64***
  - ***API***
  - ***lado***
  - ***nbrhd***
  - ***parser_lado***
- ***dirmain/***
  - ***main***

*Api* provee las Herramientas necesarias para que se pueda implementar el algoritmo Dinic. Contiene la estructura del tipo *networkSt* que, dada la implementación, corresponde con un nodo del grafo en un network teórico y al mismo tiempo corresponde con todo el network teórico. Esto es así debido a que la tabla de hash permite ver un elemento ordenado a través de ella como todo el conjunto de elementos que ordena. Básicamente, es un nodo en particular al mismo tiempo que es un conjunto de todos los nodos del network. Esta característica se aplica a todos los elementos que son hasheados. *NetworkSt* tiene información de su vecindario almacenado en *nbrs*, esto significa todos los nodos que llegan y todos los nodos que son alcanzados por algún nodo.

*Api* también contiene a *DovahkiinSt*, que por un lado almacena meta datos del network como ser el grafo, la fuente y el resumidero entre otros pero también datos de corrida de Dinic, entre ellos el corte (que es minimal si el algoritmo terminó), el camino encontrado de *s* a *t* y un contador de caminos. Es una estructura bastante transparente así que no nos vamos a detener en su análisis.

2 Hay datos que se omitieron como ser el handler de la tabla de hash y el campo *y* se acomodo a la izquierda de la entrada que lo tiene como key para facilitar la lectura del gráfico.

*lado* es una herramienta que permite crear, manipular y destruir los lados del grafo que representa el network. Si tenemos un lado $\vec{xy}$ de capacidad *c* se crea un elemento de la estructura Lado que almacena los tres datos.

*nbrhd* Se utiliza en cada nodo del grafo y significa neighborhood, en español vecindario, agrupa todos los nodos con distancia 1 con respecto al nodo en el cual se esta almacenando. Esto incluye los lados forward y backward. Los elementos están ordenados en una tabla de hash y su llave es el nombre del nodo.

*parser_lado* Es una librería que se encarga de parsear un lado desde standard input. Este lado tienen que ser de tres *u64* donde el primero es el origen del lado, el segundo el destino y el tercero la capacidad del lado. Una serie de estos elementos se diferencian en el standar input porque están separados por un ***\n*** y finalizan por un *EOF*, quedando:

*{'+int+' '+int+' '+int+'\n'}+EOF*

Un ejemplo sería:

*2 4 6\n2 4 54\n321321 321321 4888\nEOF*

Donde los lados son $\vec{2\,4}$ de capacidad *6*, $\vec{2\,4}$ de capacidad *54* y $\overrightarrow{321321\ \ 321321}$ de capacidad *4888*. Esta notación permite que cualquier elemento puede ser origen y/o destino de cualquier lado. En el ejemplo se ve que hay una arista doble entre el lado *2* y *4* como también se ve que hay un lado que va a si mismo. El algoritmo diseñado soporta este tipo de casos y varios más. Para más información ver el apartado Tets.

*main* Es un programa usado para probar el comportamiento de la API. Una vez compilado se ejecuta con el comando "*main -s source -t sink [OPCIONES] < NETWORK*"

Las opciones son:

- *-s SOURCE* Configura al nodo *SOURCE* como fuente.
- *-t SINK* Configura al nodo *SINK* como resumidero.
- *-vf –valorflujo* Imprime el valor del flujo.
- *-f --flujo* Imprime el flujo.
- *-c --corte* Imprime el corte.
- *-p --path* Imprime los caminos aumentantes.
- *-r --reloj* Imprime el tiempo en hh:mm:ss.ms de la ejecución

El main se comporta de una manera muy sencilla, luego de setear las flags y suponiendo que se cargaron los nodos y aristas de un grafo válido, entra en un loop que se ejecutará mientras se pueda llegar a *t* actualizando los niveles de cada nodo. Dentro de este loop se ejecuta otro loop mientras se encuentren nuevos caminos aumentantes con los niveles seteados anteriormente. Su función será aumentar el flujo tantas unidades como se puedan mandar por cada camino nuevo encontrado. Cuando se terminan los dos loops el flujo que se obtiene es maximal y el corte es minimal.

# Librerías auxiliares

Para no reinventar la rueda usamos librerías que, si bien algunas no son de nuestra autoría, no apuntan a resolver directamente el problema de *Max-Flow Min-Cut* sino que mas bien son herramientas destinadas a ser usadas por las librerías principales. Las librerías que no son nuestras fueron descargadas de internet, ya que son de distribución general, o proveídas por docentes de la FaMAF.

La nomenclatura que se utilizó en los nombres de las librerías es un guión bajo(_myLibrary) para las que fueron escritas por nosotros y doble guión bajo(__notMyLibrary) para las que no.

A continuación se hará una breve descripción de las librerías, para una más detallada se podrá consultar la documentación.

- __bstrlib: Librerías para el manejo de cadenas de caracteres. Simplifica la creación y manipulación de strings. http://bstring.sourceforge.net/
- __lexer: Librerías para el manejo del standar input. Facilita la lectura y la manipulación de caracteres. Suministrada por los profesores de la cátedra de Sistemas Operativos del año 2012.
- __uthash: Librerías para hacer búsquedas de O(1) utilizando tablas de *hash*. http://troydhanson.github.io/uthash/
- _queue: Librería que permite manejar una serie de elementos de cualquier tipo (tipo *void*) organizados como una cola.
- _stack: Librería que permite manejar una serie de elementos de cualquier tipo (tipo *void*) organizados como una pila.
- _u64: Implementa el tipo unsigned de 64 bits y algunas operaciones básicas entre ellos.

# Testing

Se han implementado varios test que evalúan el rendimiento y la robustez de diferentes características del código. Para compilar se uso gcc4.8.2/valgrind3.10 para los test hechos en Linux Mint y gcc4.8.1/valgrind3.8.1 para SUSE.

## Rendimiento

El algoritmo de Dinic tiene un tiempo de ejecución de $O(mn^2)$ donde $m$ es la cantidad de aristas y $n$ la de nodos. Se buscó lograr que ninguna acción tomada por la implementación supere este orden, para de esta manera garantizar una cota máxima en el método y no en el código.

Una carga significativa que puede haber en la ejecución del programa esta relacionada con la organización de nodos en una tabla de *hash*. Si bien las tablas de *hash* ofrecen una búsqueda $O(1)$ en la mayoría de los casos, para alguno particulares el orden de búsqueda se ve afectado por la distribución de la llave con la que se almacenan los elementos en la tabla, de manera que una búsqueda en particular puede llegar a ser de $O(n)$ en casos muy raros. Si bien tenemos en cuenta esta posibilidad queda pendiente para futuras mejoras del software reconocer el tipo de distribución de datos de la key para utilizar una tabla de hash que garantice el $O(1)$ en toda búsqueda, aunque todavía se desconoce el costo. De cualquier manera puede ser mas económico tener una búsqueda $O(n)$ y el resto de $O(1)$. En general se puede ver al orden de búsqueda como $O(1)$.

## Robustez

Se entiende por robustez que el código sea capaz de reaccionar de mejor manera a situaciones no comunes. Esto significa que sea capaz de soportar desde grafos con características peculiares en cuanto a la morfología hasta grafos mal tipeados y, mientras sea posible, mantener la correctitud.

## Test Aplicados

Se eligió correr ciertos test para ver la correctitud y la velocidad del algoritmo. Dado que la velocidad de respuesta de la aplicación es una característica decisiva cuando se trata de escalar el programa, se le exigió al mismo que resuelva el problema planteado correctamente y en menor tiempo, ya que comprendemos que es tan malo que devuelva un valor incorrecto como que devuelva el correcto pero utilizando un tiempo extremadamente largo.

### Metodología

Los test de rendimiento se repitieron 30 veces en cada hardware sin la GUI de linux (para tener menos cargado el sistema) y se corrió una vez antes, la cual no se contabilizó, para que ya estén cargados en la memoria RAM los parámetros del programa(sobretodo el network). Luego se midieron los tiempos devueltos en cada prueba y se aproximaron los resultados para ver cómo variaban los tiempos.

Test de escalabilidad

Se correrá el programa en grafos de diferente complejidad para ver cómo varia el tiempo de ejecución. En el anexo podremos encontrar una descripción del hardware testeado. Se evaluará de qué manera influye en la velocidad y en la memoria ocupada la complejidad de los networks. Para esto se correrá el algoritmo en 3 tipos de networks diferentes.

1. networkSmall, network de 10 aristas y 8 vértices.
2. networkMedium, network de 1000 arista y 668 vértices.
3. networkLarge , network de 7498 aristas y 5000 vértices.
4. networkEpic, network de 27490 aristas y 16449 vértices.

Test de Robustez

Se Construyeron networks con características particulares para comprobar que el programa es capaz de resolver ciertas situaciones extrañas.

Estas situaciones comprenden:

1. Caso de un network vacío.

   Se espera que el programa no calcule nada.

2. Caso de un network sin camino aumentante posible.

   Se espera que devuelva flujo = 0 y el corte correcto.

3. Caso de un network con sólo dos aristas y desconectadas entre si.

   Se espera flujo = 0 y, al ser un grafo bipartito, se espera que retorne como corte la partición de nodos en la que se encuentra $s$.

4. Caso de network con un bucle de largo 1. $\vec{xx}$

   Se espera correctitud.

5. Caso de network con una arista inversa.

   Se espera correctitud.

6. Caso de un network con aristas paralelas.

   Se espera correctitud.

7. Caso de network con un bucle de largo 2. $\vec{xy}$ $\vec{yx}$

   Se espera correctitud.

8. Caso de network con un bucle de largo 3. $\vec{xy}$ $\vec{yz}$ $\vec{zx}$

   Se espera correctitud.

9. Caso de network donde el nodo fuente es el mismo que el resumidero. *s=t*

   Se espera flujo=0 y corte{s}.

10. Caso de un network con una entra inválida.

    Se espera que se cargue hasta la linea inmediata anterior válida y luego correr normalmente.

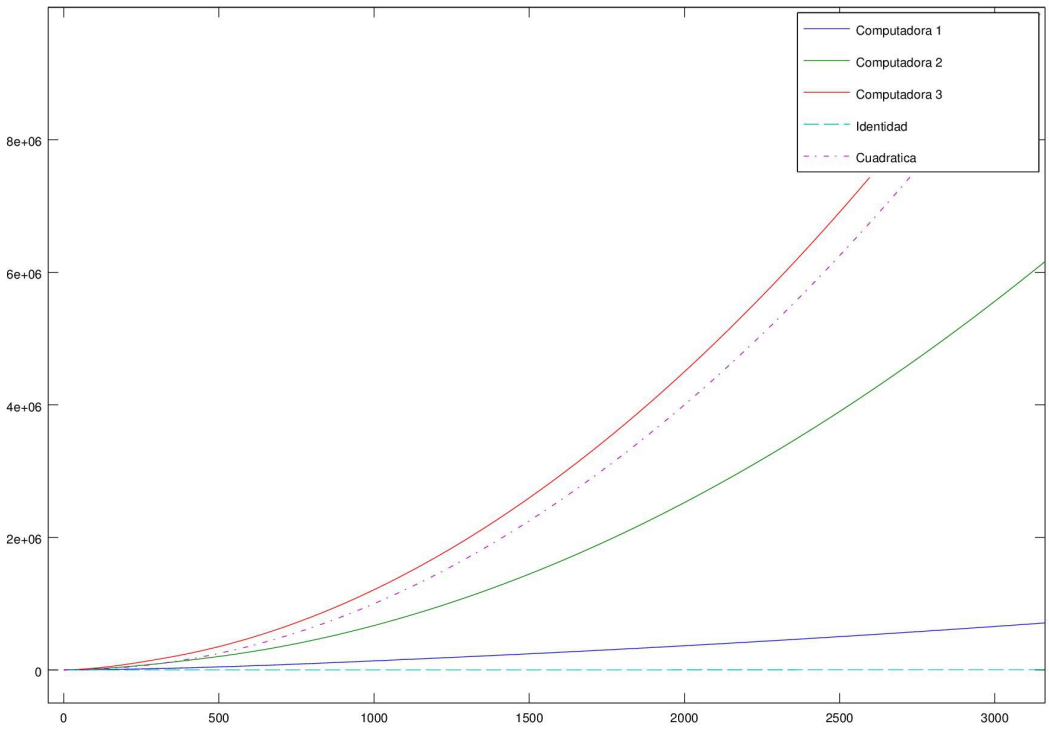
# Análisis de los resultados

## Rendimiento

En cuanto a los tiempos de respuesta del algoritmo en las 3 computadoras se obtuvieron los siguientes resultados promedio. Para cuantificar los resultados se le asigna a cada network una dificultad igual a la complejidad c del algoritmo. La desviación estándar es casi despreciable. No mas de 16ms en el hardware más lento con el network más complejo(peor de los casos), representa apenas un 0,12% del total.

| | Complejidad | Computadora 1 | Computadora 2 | Computadora 3 |
|---|---|---|---|---|
| networkSmall[3] | 1 | 1ms | 1ms | 1ms |
| networkMedium | 446 | 20ms | 8,3ms | 6,6ms |
| networkLarge | 187450 | 1.242,7ms | 473,7ms | 335,4ms |
| networkEpic | 7437958 | 13.164,7ms | 3.483ms | 2.597,7ms |

3 Los tiempos devueltos en este network son menores a 1ms pero lo redondeamos hacia arriba.




Se puede observar en los tres tipos de hardware que la cantidad de puntos de complejidad resueltos por cada unidad de tiempo aumentan[4], las computadoras 1 y 2 crecen entre la identidad y la cuadrática, mientras que la computadora 3 crece de manera mayor a la cuadrática.

La función que relaciona la complejidad con el tiempo está dada por r=mc/t.[5] Esto se ve reflejado en la tabla siguiente.

| | Complejidad | Computadora 1 | Computadora 2 | Computadora 3 |
|---|---|---|---|---|
| networkSmall | 1 | 1,00 | 1,00 | 1,00 |
| networkMedium | 446 | 22,3 | 53,73 | 67,58 |
| networkLarge | 187450 | 150,84 | 395,71 | 558,88 |
| networkEpic | 7437958 | 564,99 | 2135,50 | 2863,28 |

A medida que aumenta la complejidad del grafo el programa responde de mejor manera. De cualquier manera habría que repetir las pruebas para networks más complejos para corroborar si existe un máximo.[6]

4 Notar la escala de los eje, la función identidad se asemeja a una constante. Las funciones se graficaron haciendo una interpolación cúbica con los datos obtenidos de los test.

5 Un mc es un millón de puntos de complejidad. En este caso también esta redondeado hacia arriba.

6 Puede no tener cota máxima dado que un set de *n* instrucciones puede resolver mas de *n* puntos de complejidad. Osea la complejidad de un grafo puede aumentar más rápido que la cantidad de instrucciones para correr Dinic en el grafo.

En cuanto al uso de memoria, solo depende de la complejidad del network y del sistema operativo en el que se corrió el test. En ninguno de los networks superó 34mb de memoria

| | SUSE | mem/mc | Linux Mint | mem/mc |
|---|---|---|---|---|
| networkSmall | 2224128 | 2224128 | 6541312 | 6541312 |
| networkMedium | 2899968 | 6502,17 | 7622656 | 17091,16 |
| networkLarge | 8052736 | 42,96 | 15835136 | 84,48 |
| networkEpic | 22003712 | 2,96 | 34709504 | 4,65 |

[insertar grafico???????????]

El programa usa menos memoria por cada punto de complejidad mientras más complejo es el network. Ideal para que el programa escale. Teniendo en cuenta que en la actualidad una computadora hogareña tiene 8GB de RAM, el uso de memoria de este software es despreciable.

## Robustez

En este caso no hay mucho que describir. El programa pasó los 10 test sin ningún problema ni repercusión en su desempeño.

## Conclusión

Debido a la cantidad de memoria usada por el programa y al tiempo de respuesta mientras se aumenta la complejidad de los networks se puede determinar que, en estos puntos, escala de manera óptima. En cuanto a la robustez con respecto a la morfología del network, el programa se diseñó para que pase los 10 tipos de problemas además del caso correcto y lo hace sin complicaciones. Si bien, en base a estos tests, los resultados son satisfactorios se concluye que no son determinantes debido a que habría que hacer test mas intensivos. Se sugiere en revisiones futuras:

- Testear el programa en diferentes SO pero manteniendo el mismo hardware. Evalúa la repercusión del sistema operativo en el programa.
- La unidad que se utilizó para analizar la complejidad del network no contempla la dificultad que significa para el algoritmo la morfología del network.
- Rehacer los test de rendimiento. De las cuatro características que influyen en el rendimiento del programa(morfología del network, cantidad de nodos, aristas y caminos) evaluar cómo varía el comportamiento modificando una de ellas a la vez para networks con diferente morfología pero de igual complejidad.

# Anexo

## Características de las maquinas del Testing

### Computadora 1:

| | |
|---|---|
| Modelo de CPU: | AMD Sempron(tm) Processor 3400+ |
| Frecuencia: | 1800mhz |
| Cache: | L1: 64KB / L2: 256KB |
| Núcleos: | 1 |
| Modelo de la Memoria: | Kingston DDR2 1gb<br>novatech DDR2 512mb |
| Memoria Total: | 1544044KB (1.5GB) |
| Placa Madre: | MSI K9MM-V |
| modelo del HDD: | Western Digital Black Caviar 80GB sata I |
| Sistema operativo | Linux Mint 17 Qiana 3.13.0-24 32bits |

### Computadora 2:

| | |
|---|---|
| Modelo de CPU: | Intel Core 2 Duo E6750 |
| Frecuencia: | 2,66GHz |
| Cache: | L1: 2x32kb / L2:1x4mb |
| Núcleos: | 2 |
| Modelo de la Memoria: | Kingston DDR2 Dual Channel 667Mhz |
| Memoria Total: | 2GB |
| Placa Madre: | ASUSTeK P5W DH Deluxe<br>Chipset: i975X<br>Southbridge: W83627DHG |
| modelo del HDD: | Western Digital Blue Caviar 169GB sata III |
| Sistema operativo | SUSE Linux 3.11.10-17 32bits |

### Computadora 3:

| | |
|---|---|
| Modelo de CPU: | AMD FX 6300 |
| Frecuencia: | 3.5 Ghz |
| Cache: | L1: 6x16kb /L2: 3x2mb / L3: 8mb |
| Núcleos: | 6 |
| Modelo de la Memoria: | Gskill Sniper DDR3 Dual channel 1600 |
| Memoria Total: | 2x4GB |
| Placa Madre: | ASUSTeK M5A97 R2.0<br>Chipset: RD9x0<br>Southbridge: SB910/950 |
| modelo del HDD: | Western Digital Black Caviar 1TB sata III |
| Sistema Operativo | Linux Mint 17 Qiana 3.13.0-24 64bits |

# Repositorio

El SVN del proyecto se encuentra hosteado en los servidores provistos por Google. Se puede ingresar desde https://code.google.com/p/dinic-network/

Para poder descargarlo, ejecutar el comando

*svn checkout http://dinic-network.googlecode.com/svn/trunk/*

# Glosario

## Términos y sus notaciones

Aquí figuran las notaciones de todos los términos que estaremos utilizando. Daremos por hecho que el lector tiene una base de conocimiento teórico del tema para no entrar en detalles de definiciones. El formato es: *Término(notaciones): descripción*.

- Dovahkiin(*dova*): Estructura principal que contiene toda la información necesaria para correr Dinic.
- Network(*net*): Grafo dirigido $(V, E): E \subseteq V \times V$ con pesos en los lados.
- Nodo(*node*): Usaremos letras del alfabeto latino, usualmente *x, y, z, s, t*.
  - Fuente(*source, src, s*): Nodo productor de flujo.
  - Resumidero(*sink, snk, t*): Nodo consumidor de flujo.
- Lado(*edge*): $\{x, y\} \to \overrightarrow{xy}$ si se sobreentiende por el contexto también $xy$.
  - Lado Forward(*Fedge*): $\overrightarrow{xy}: f(\overrightarrow{xy}) < C(\overrightarrow{xy})$ , el flujo $f$ es menor a la capacidad.
  - Lado Backward(*Bedge*): $\overrightarrow{yx}: f(\overrightarrow{yx}) > 0$ , i.e. que haya mandado flujo.
- Capacidad(*cap*): $C: E \to \mathbb{R}^{\geq 0}$ donde $C(x, y)$ sería el peso del lado $xy$.
- Flujo(*flow*): $f: E \to \mathbb{R}^{\geq 0}$ admisible, conservador y $s$ produce.
  - Valor del flujo(*vflow, vf*): $V(f) = \sum_{y \in \Gamma^+(x)} f(\overrightarrow{xy}) - \sum_{y \in \Gamma^-} f(\overrightarrow{yx})$
  - Flujo Maximal(*maxFlow*): $V(f) = f(S, \bar{S}) - f(\bar{S}, S)$ y $V(f) = Cap(S)$ donde S es corte (y por consecuencia es minimal).
- Vecindario(*nbrhd*): $\Gamma(x) = \Gamma^+(x) \cup \Gamma^-(x)$ , i.e. todos los vecinos forward y backward.
  - Vecinos Forward(*fNbrs*): $\Gamma^+(x) = \{y \in V : \overrightarrow{xy} \in E\}$
  - Vecinos Backward(*bNbrs*): $\Gamma^-(x) = \{y \in V : \overrightarrow{yx} \in E\}$
- Corte: $S \subseteq V : s \in S \wedge t \notin S$ con capacidad $cap(S) = C(S, \bar{S})$ donde $\bar{S} = V - S$ . Cuando

hablemos de corte, implícitamente nos estaremos refiriendo a Corte Minimal porque es con lo que nos interesa trabajar.

- Corte Minimal(*cut*): $V(f)=Cap(S)$ donde $f$ es flujo maximal.